Isochronous Control of Sensor Networks: Final Report

Aishwarya Parasuram, Jack Kolb, Nikhil Goyal
UC Berkeley

I. Introduction

Our project aims to create a network of sMAP sensor nodes and actuators. The actuators are controlled isochronously using a time synchronization protocol. Our work incorporated several key concepts from the course, including modeling with FSMs, time-triggered systems, time synchronization, and modular design.

A. Overview of sMAP

sMAP is a specification for a protocol which exposes and publishes time-series data from a variety of sensors. The main components of sMAP include sources (a restful web service that accepts requests and publishes sensor data), archivers (persistent storage of sensor data in the form of a time-series), and applications (perform visualization and analysis of sensor data to create actuation signals).

B. Definition of Isochrony

Our project centers on achieving isochronous control over a collection of sensors and actuators. In an isochronous system, all components react simultaneously, and a fixed period of time elapses between reactions. Thus, isochronous control can be viewed as a stricter form of synchronous control. We focused on isochrony because it serves as a foundation for more sophisticated time-based control schemes; most of the work that would go in to implementing such schemes is also required to achieve isochronous control. Moreover, isochronous control has several important benefits, such as keeping control loops stable and ensuring that new sensor data is acted upon in a timely manner.

C. Goals

1) Modularizing sMAP architecture by isolating sMAP sources and moving them to embedded drivers
2) Finding minimal embedded device for sMAP Source
3) Isochronous Actuation
4) Introducing the concept of a global notion of time among all nodes by using a time synchronization protocol

II. System Design and Modelling

A. Overall Design

Our system consists of temperature sensors interacting with sMAP sources. The sMAP source runs on an Arduino Mega and accepts requests from a web service wanting to subscribe to the sensor's data. The source sends data as HTTP/JSON to the zone-controller, which is a centralized controller that keeps track of the sensors and actuators and acts as the local NTP server. The zone controller computes intelligent actuation signals based on the combination of data from all the sensors. These actuation signals are sent out as soon as they are computed, however the actuation itself takes place at the actuators isochronously. The signals are buffered at the actuator until it is time to take the actuation. A global notion of time is maintained throughout the network by the zone-controller. Time signals are sent by the zone-controller as a JSON object in an HTTP `PUT` Request (to the actuators) or HTTP Response (to the sources) and these are updated locally at the sources and the actuators. Figure 1 depicts the exchange of messages between the various components in the network. The Global Data Plane (GDP) acts as a replacement for the sMAP archiver in this architecture. We built an HTTP-GDP interface that accepts HTTP/JSON from the zone-controller, converts them into a GDP friendly format and sends it to a distributed log. Chimera, an open-source routing library, is responsible for location independent routing. The complete design of our system is given in Figure 2.

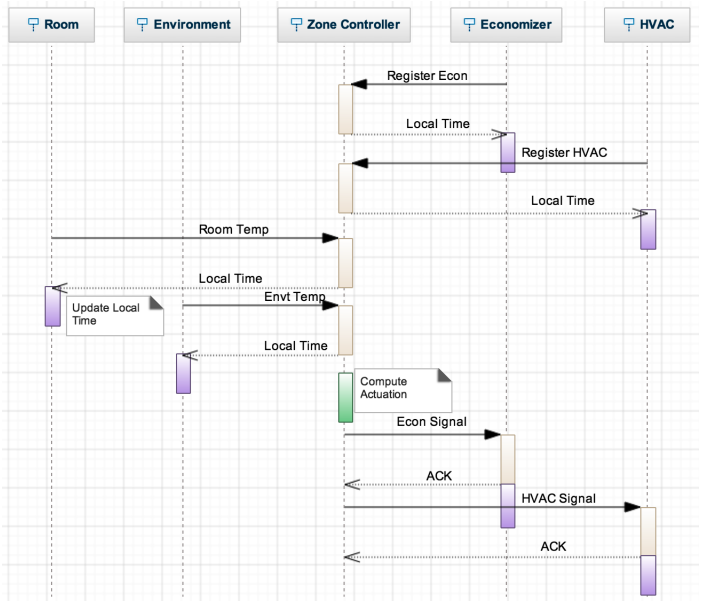

Fig. 1. Typical Sequence of Network Messages

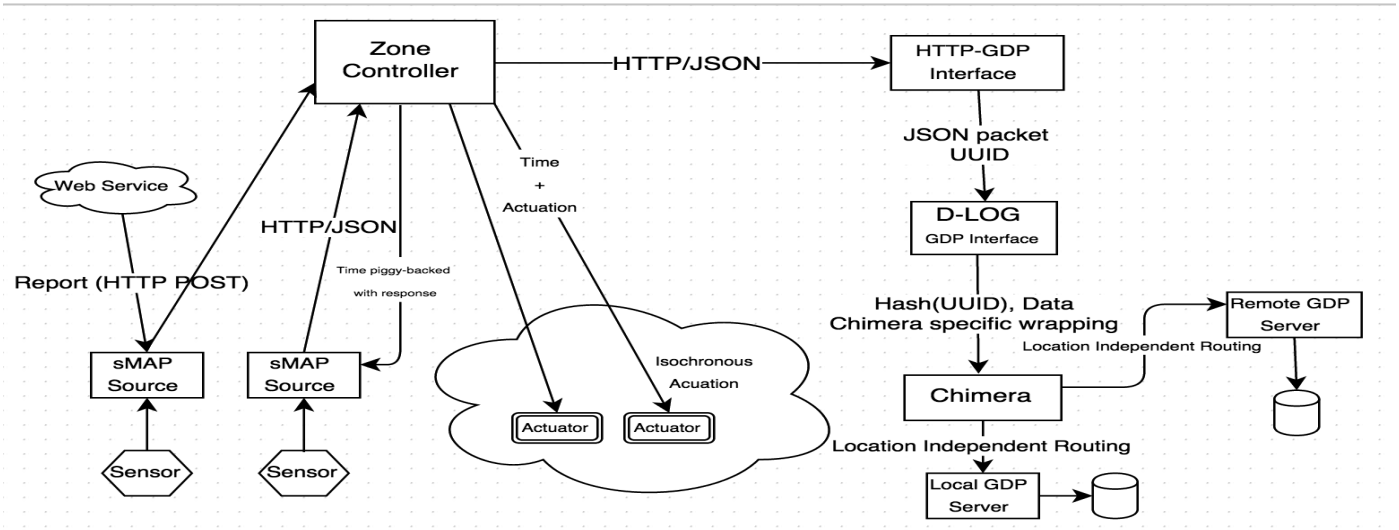

Fig. 2. System Architecture

B. Zone Controller

The zone controller ties the network's sensors and actuators together. It accepts data from sensors in the form of HTTP requests with a JSON payload. The zone controller then analyzes this data to compute intelligent actuation signals. Finally, the zone controller sends actuation signals to the network's actuators in the form of HTTP requests, also with a JSON payload. It also functions as the server for time synchronization. The controller sends its local time to the sensors and actuators piggy-backed on HTTP `PUT` requests and HTTP responses, so that the time can be updated on the embedded drivers.

III. System Implementation

A. Choosing an Embedded Driver

The original implementation of sMAP is through Python and a set of Python dependencies on FitPC. Isolating sources enables ease of installation and flexibility. Deploying sources on embedded drivers achieves energy efficiency and reduces cost. Raspberry Pi (RPi) seemed to provide a good starting point, as it runs Linux and supports Python. sMAP was installed on RPi along with its dependencies, and we created an sMAP driver to run a motion detector. The sMAP source functioned as an independent unit, and relayed time-series data to sMAP archivers. However, CPU usage analysis indicated that RPi was a wasteful use of resources for a dedicated sMAP source, pushing us towards a lower-level implementation of sMAP. Since most embedded devices speak C, the ideal approach is to create a C implementation of the sMAP source, and deploy this on an embedded platform such as Arduino Uno, which can natively interact with a sensor. Since a sMAP source needs to act as a HTPP client and source at the same time, Arduinos lack of support for multiple threads was unhelpful. We attempted to use timer interrupts to implement a multi-threaded model but due to the limitations of embedded platforms and complexity of the code, this implementation was unstable. Round-robin scheduling of client and server activities served as a work-around.

As the complexity of sMAP source functions increased, the Uno was unable to process the code in real time due to its limitations in memory and computational power. When the same code was ported to the Arduino Mega, which has higher capacity in terms of processing power and program memory, the system showed ideal behavior. Hence we are currently using the Mega as an sMAP source.

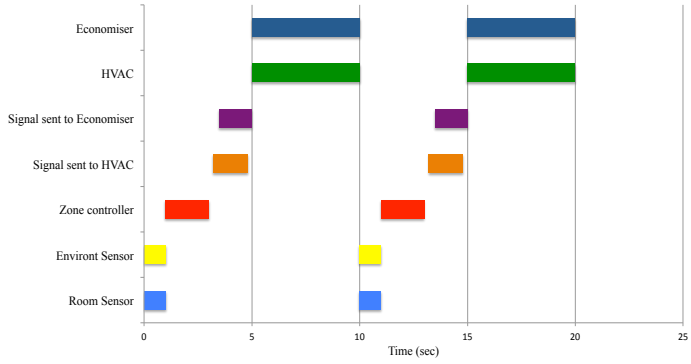

Fig. 3. Timeline of Events

B. Use Case Description

We designed our network of sensors and actuators based on a motivating use case: controlling the temperature of a hypothetical room while also minimizing energy consumption. The system features two sensors to measure room and outdoor temperatures. The system also features two actuators. The first is an HVAC system that can supply artificially heated or cooled air to the room. The second actuator is an economizer, which allows air to enter the room from outside. This allows energy to be conserved by avoiding use of the HVAC system when unnecessary while maintaining a constant airflow.

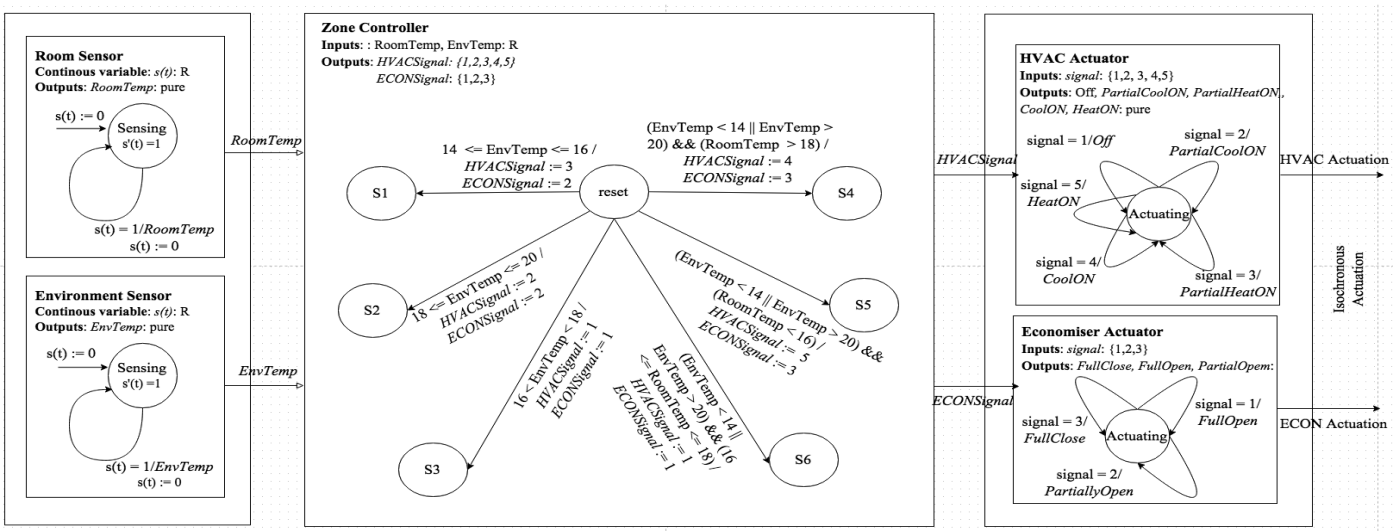

Fig. 4. Finite State Model

The zone controller analyzes sensor data in order to maintain a comfortable temperature in the room while minimizing energy consumption. It accomplishes this by controlling the HVAC and economizer actuators together. For example, if the outside air is within a comfortable range, then the economizer is fully opened to allow external air to enter the room, and the HVAC is deactivated. Figure 3 shows the sequence of events that occur during each cycle of system operation.

In order to simulate this use case, we constructed a network of Arduino boards connected via Ethernet. Two of the Arduinos represent the sensors. Another two Arduino boards are hooked up to multiple LEDs to visually indicate which actuation signal was received. For example, one of the boards represents that HVAC actuator and illuminates different LEDs depending on whether it received signals to heat, cool, or turn off. Finally, we integrated the sensors and actuators together using the zone controller described earlier, which was deployed on a laptop PC. Our system can be modeled as a composition of finite state machines, as seen in Figure 4.

C. Components

The important hardware components used in our system are shown in Figure 5. We used TMP36GT9 as a temperature sensor. This is interfaced with the Arduino Mega that functions as the sMAP source. The Mega collects sensor data and uses an ethernet shield to wrap data in an HTTP PUT request as a JSON packet. This is sent over a local network to the zone controller, which is implemented as a Python program on a laptop. The actuator logic is again implemented on Arduino Mega. The actuators are boards soldered with LEDs to represent the various states of the economizer and the HVAC system. All sensors and actuators are affixed with real time clocks (DS1307) as the Arduino does not have a notion of wall time, although it does have an internal timer that can be programmed. The nodes also include LCD displays (LCM1602 IIC V1) that are used to display the time and sensor readings for debugging purposes.

D. Testing

1) Component Test: Since our project involves a number of embedded components purchased from third party vendors, we performed component tests on each of them individually before building the system as a whole, to simplify the debugging process. This helped us find a faulty Real Time Clock with a burnt IC, which would have otherwise given us erroneous time readings.

2) Unit Test: Each unit of our system such as the sMAP source, zone controller, actuator and the HTTP-GDP Interface was individually tested with synthetic inputs and outputs before integration. This helped us modify signal formats before building it into the larger architecture.

3) System Test: Since our project consists of a number of heterogeneous modules and additionally combines two larger projects, the sMAP and GDP, we performed intensive integration tests to ensure smooth functioning of the system as a whole. These tests involved receiving acknowledgements for messages sent over the network, load testing for the server, testing time synchronization across nodes through a native display and load testing.

4) Integration Test with Synthetic benchmarks: Since we were particularly concerned with the integration of sMAP and GDP, we created artificial sensors that generated synthetic data, embedded this data as JSON within a HTTP packet and sent it to the interface. The test involved querying sensor data from remote servers. Successful results from queries indicated that the data passed through all stages of processing.

IV. Challenges and Limitations

A. Lack of Support for Multithreading in Arduino

One of the limitations of our design is that both the actuators may miss some of the signals sent to them by the zone controller. This is because each actuator performs two tasks:

1) Listening on the network port for actuating signals from the zone controller
2) Obtaining the signal from the port and processing it.

In an ideal case, these two tasks should be performed using two separate software threads, where one of the threads would be dedicated to listening the network port and storing the incoming signals in a temporary buffer. However, Arduino Mega does not provide any support for multithreading. Thus we were forced to perform these two tasks sequentially and listened for signals through periodic polling. While performing the signal processing tasks, if the controller overwrites the previous unprocessed actuating signal with another, the actuators will miss the first signal sent by the controller.

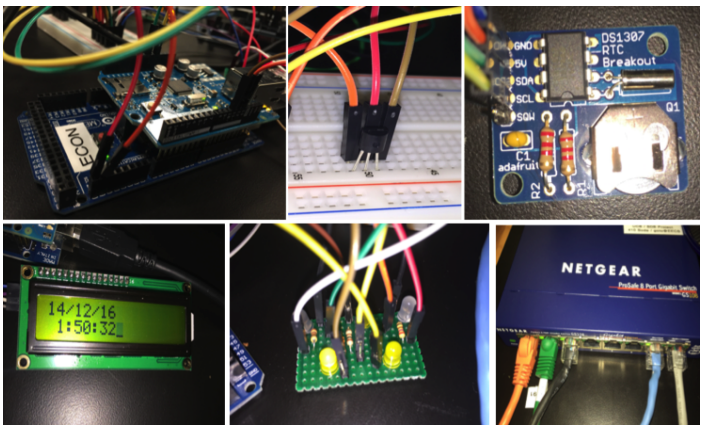

Fig. 5. Hardware Components (Clockwise from top left) - sMAP Source and Actuator on Arduino Mega, TMP36GT9 Temperature Sensor, Real-time Clock DS1307, YwRobot Arduino LCM1602 IIC VI, Actuators Implemented as LEDs and Netgear Ethernet Switch

B. Problems with Ethernet shields

For the two sensors, we were using two different types of Ethernet shields, which used separate built-in libraries. As a result the kind of code implemented to use these libraries had to be significantly different.

C. Network Configuration

Our system required us to build our own local network without connecting to the Internet. Hence, we had to manually allocate an IP address and a MAC address to each component in the network to establish communication between the sensors, actuators and the zone controller. We also had to devise our own time synchronization protocol, as we couldn't establish a connection with global NTP servers.

D. Time Precision

The real-time clocks we used were limited to a precision of one second. Thus, the clocks on each Arduino board could only be synchronized with a granularity of one second, which resulted in cases where the two actuators were slightly out of sync. As a future step, we could use the RTCs and the internal Arduino clocks in combination to achieve millisecond precision.

V. Conclusion

Our project produced a successful prototype network of sensors and actuators featuring intelligent and isochronous actuation based on sensor data. The network components interacted via an HTTP/JSON interface with a central zone controller, and a time synchronization protocol was used to establish and maintain a global notion of time. There are several potential next steps we could take, including better integration with the GDP and developing more intricate time-based control schemes.

References

[1] "Arduino," http://arduino.cc/en/Reference/Ethernet.

[2] "Bottle Library," http://bottlepy.org/docs/dev/index.html.

[3] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, "smap: a simple measurement and actuation profile for physical information," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2010, pp. 197–210.

[4] "The Global Data Plane," https://swarmlab.eecs.berkeley.edu/projects/.

[5] "NTP & PTP," http://www.en4tel.com/pdfs/NTPandPTP-A-Brief-Comparison.pdf.

[6] "Chimera," http://current.cs.ucsb.edu/projects/chimera/.

[7] "Seeed Studio Ethernet Library," http://www.seeedstudio.com/wiki/File:W5200_Ethernet_Shield_Library.zip.