`ConnectedHearts`: A Distributed System for Viewing the Human Heartbeat

Serena Booth[a,1], Michelle Cone[b,1]

[a] *sbooth@college.harvard.edu*
[b] *mcone@college.harvard.edu*

Abstract

We present a physical distributed system for viewing the human heartbeat. We modified a medicine cabinet by embedding 13 light bulbs around the frame, as well as by replacing the cabinet's glass with one-way mirror. When a person stands in front of the mirror, a webcam hidden behind the mirror measures their pulse. The 13 bulbs, each a virtual machine, run Bully leader election. First the leader starts pulsating with the captured pulse, and then instructs neighboring virtual machines to pulsate as well. Finally, we ensure the synchronization of the bulbs through a distributed gossip algorithm.

1. Introduction

`ConnectedHearts` straddles the boundary between art and computer science. The piece is inspired by an exhibit shown around the world in modern art museums, but our re-implementation emphasizes distributed computing, as `ConnectedHearts` is a physical representation of a distributed system. We modify a retro medicine cabinet to hold 13 light bulbs around its frame.

[1] We, Serena Booth and Michelle Cone, affirm our awareness of the standards of the Harvard College Honor Code.

While in an ideal world each light bulb would be powered by its own microprocessor, we simulate this interaction instead by each light bulb representing a virtual machine as a process in effort to reduce expenditure. These light bulbs attempt to self-synchronize while displaying the human heartbeat.

In this paper, we discuss the components and construction of `Connected Hearts`, the architecture and algorithms powering the software, as well as analysis of our system.

1.1. Artistic Inspiration

This piece is inspired by Rafael Lorzeno-Hemmer's "Pulse Room,"[**?**] an artistic piece in which an entire room is outfitted with light bulbs. A physical heartbeat monitor is present in the room. A user approaches this monitor and grabs onto it. As soon as the conductance of the skin is felt, all light bulbs in the room turn off. Within a few seconds, the heartbeat monitor detects a pulse. With this, a single bulb directly above the user begins to pulsate with their heartbeat. This message is then spread to neighboring bulbs, as well as bouncing off of the walls; with this message-passing, the room becomes full of chaos. After a further few minutes, the bulbs synchronize, leaving the entire room pulsating with the heartbeat. We note that this system uses a single address-space to achieve this effect.

2. Components & Construction

We construct the piece from the following materials:

- 2 × Ubiquiti mPower Strips: 8-outlet power strips running Linux
- Network switch and ethernet cables

- 13 × filament bulbs and bulb holders
- Vintage Medicine Cabinet, one-way glass
- Raspberry Pi
- Raspberry PiCamera

The end product looks to be a medicine cabinet with 13 light bulbs around its mirror; the majority of the components—the cables, the Raspberry Pi, etc.—are hidden from view, inside the cabinet.

3. System Design

Our code is written in Python and makes extensive use of the multiprocessing library. In short, the program runs a `main.py`, the parent process. This process creates one process for running pulse detection; two more processes for turning all the relays off at the start; and 13 more processes, each corresponding to a bulb. Of these 13 "bulb" processes, each spawns its own process to handle inter-bulb synchronization. Of these inter-bulb synchronization processes, each spawns an additional process which is tasked with maintaining the state of a continually-blinking bulb.

3.1. Inter-bulb Communication

We of course do not assume the existence of a global clock. However, we do allow processes to communicate via shared memory. In particular, many processes share queues. For example, in `main.py`, we start up 13 "bulb" processes, as described above. These processes each have two process-safe

queues: one of which, designated the `election_q` stores information pertaining to leader election, and the other of which, `state_q` stores information regarding whether the bulb should be on or off, as well as any adjustments to this.

4. Architecture & Algorithms

4.1. Pulse Detection

When a face is visible to a camera hidden behind the one-way glass of the medicine cabinet, we detect the pulse of the viewer. We use a Haar Classifier and OpenCV to detect the presence of a face, and we then apply code developed by NASA's Glenn Research Center[**?**] to detect the pulse of the visible face. This pulse detection works by monitoring the *optical intensity* of the green channel of pixels centered around the visible area of the forehead in frame.

4.2. Bully: Leader Election

Our leader election is modeled after the Bully algorithm, as originally described by Garcia-Molina in 1982 [**?**]. We made modifications to the Bully algorithm: in our first leader election, in the way we deal with a previous leader's recovery, and how we deal with multiple "new election" messages.

In our first leader election, we know that all the processes will be participating and that they know all the uuids because they have them on their own `election_q`. We simplify the Bully algorithm by having each process find the max uuid in its queue and choose the process with the corresponding uuid as the leader. All the processes agree because the main thread guarantees that the same uuids are put on every `election_q`.

After the leader is chosen, the other processes must make sure that it remains responsive. We achieve this by having each "follower" ping the leader every several seconds and wait for a response. Each process is given a random `ping_time` that determines how frequently they ping or respond to pings. This ensures that the leader is not overloaded with messages from the followers.

If a follower does not receive a response from the leader after `max_timeout`, a time greater than the largest possible `ping_time`, it starts a new election that follows the procedure of the Bully algorithm. It sends a "new election" message to all processes with uuids higher than its uuid, which causes these processes to stop pinging the leader and participate in the election, and then it waits for a response from these processes for a time of `max_timeout`. If any processes with higher uuids respond, it waits to receive a message from a new leader. If it doesn't receive a message from a new leader within the timeout, it starts leader election again.

If a process never receives responses from a higher uuid process, or it has a higher uuid than all the responses, it assumes it's the leader and broadcasts this message to all other processes in the system. If there is not agreement on the new leader, this procedure repeats. See Figure 1 for a diagram of how this works in our system. After the broadcast from the new leader, all the followers begin pinging it and it responds to their pings. If a follower does not receive a response, then leader election starts again.

Our Implementation Differences

In our leader election, while a process waits for a response from all higher uuid processes, it also checks for "new election" messages. If it receives a

"new election" message, it sends its uuid to the appropriate processes and continues waiting to hear back from the higher uuid processes.

If a process every receives a "new leader" message while it is pinging the leader, it starts a new leader election.

To allow a process that was previously a leader to recover, the current leader pings any higher uuid processes with a "new leader" message every `ping_time`.

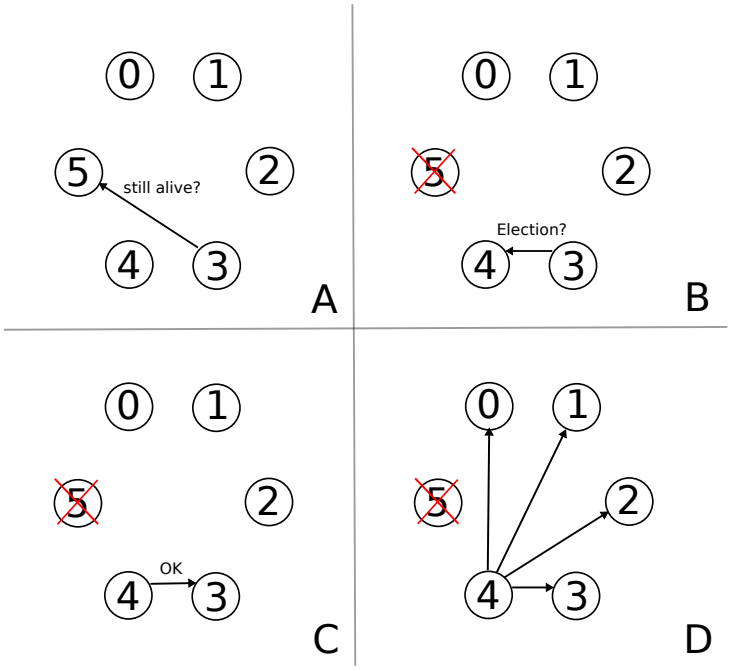

Figure 1: In the above graphic, we demonstrate the bully algorithm for determining a leader. In sub-image A, node 3 attempts to confirm that the known leader, node 5, is still alive. In sum-image B, node 5 does not respond to node 3 for a set period of time—a time set by node 3. Hence node 3 initiates an election by contacting all nodes with a higher id than itself. In sub-image C, node 4 responds to node 3's request to initiate an election by confirming it remains alive. Finally, in sub-image D, node 4 broadcasts a message to all nodes to inform them of its leader status.

4.3. Realtime Synchronization via Gossip

In our system, each "bulb" process has its own SSH connection to the Linux Ubiquiti box by which it is powered. This process repeatedly sends message to the box of the form `echo X > /proc/power/relayY`, where `X` $\in \{0, 1\}$, and `Y` represents the id of the relay controlling the power supply to the box. Commands to turn the box on are followed by some amount of wait, determined by the user's pulse, followed by a command to turn the box off and wait the same. However, as our system is "distributed," and we have no concept of a global time, the bulb relay switches are poorly aligned lacking any inter-process synchronization. We thus implement a simple gossip algorithm, as inspired by Babaoglu et al, [**?**].

As the physical layout of our system is fixed, we can select the more *trustworthy* of our neighbors, an algorithm which we describe below. With this neighbor, we attempt to iteratively align our synchronization over time. This algorithm is further described in Algorithm 1.

4.3.1. Physical Layout

The physical layout of our system is predetermined, as shown in Figure 2. Each bulb is assigned a fixed, position-based id as well as the formerly discussed, generated 64-bit uuid. Because of this fixed layout, we are able to personalize gossip for synchronization in order to ascribe neighboring processes with scores of *trustworthiness*. As the leader bulb initiates the heartbeat pulsation, the leader is the single most trustworthy bulb. This has implications for all other bulbs, however: as bulbs increase in distance from the leader, a known measure based on the system's fixed layout, their trustworthiness decreases. Thus when a bulb is receiving contradictory gossip

Algorithm 1 Synchronization via Gossip

```
1: procedure GOSSIP
2:     while True do
3:         Receive messages from {trustworthy neighbor, self}
4:         if diff(self_time, neighbor) < diff(self_time, neighbor + bpm) then
5:             Increase wait (while bulb off)
6:         else
7:             Decrease wait (while bulb on)
8:         end if
9:     end while
10: end procedure
```

from its two neighboring bulbs, it prioritizes the message which was sent by the bulb closer to the leader.

5. Synchronization Testing & Analysis

In order to analyze the success of our synchronization via gossip, we implement a simple computer vision system. The steps of analysis are as follows:

1. We capture footage at a high-frame rate.
   - 120 fps; however, we note that this frame rate is slower than the processing rate of the human eye.
2. With GUI, present user with first frame of the captured footage.
   - User selects 13 pixels corresponding to bulb filaments.
3. Brightness timeseries computed.

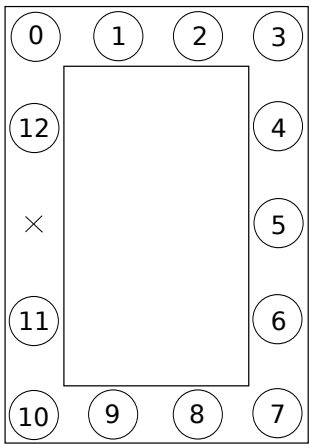

Figure 2: The system layout. While each bulb is assigned a 64-bit uuid during the program, each bulb also has an id, shown above, ranging from $0 - 12$, which allows the system to consistently converge toward the leader bulb's rate of pulsation during the synchronization routine.

- The brightness of these pixels are averaged for every frame in the video, and ultimately we produce a timeseries, which we graph in comparison to idealized data. A demonstration of this result is shown in Figure 3.

6. Conclusion

The final product of `ConnectedHearts` exceeded our expectations: while we were originally unsure that gossip could be achieved given the unreliability of some of our components, we found that, to a rate of 120 frames-per-second, the visual appearance of the mirror is indeed synchronized. Over the course of this project, we learned a substantial amount about distributed systems, some of which we relay here:

1. Debugging a (fake!) distributed system is hard, and the separation

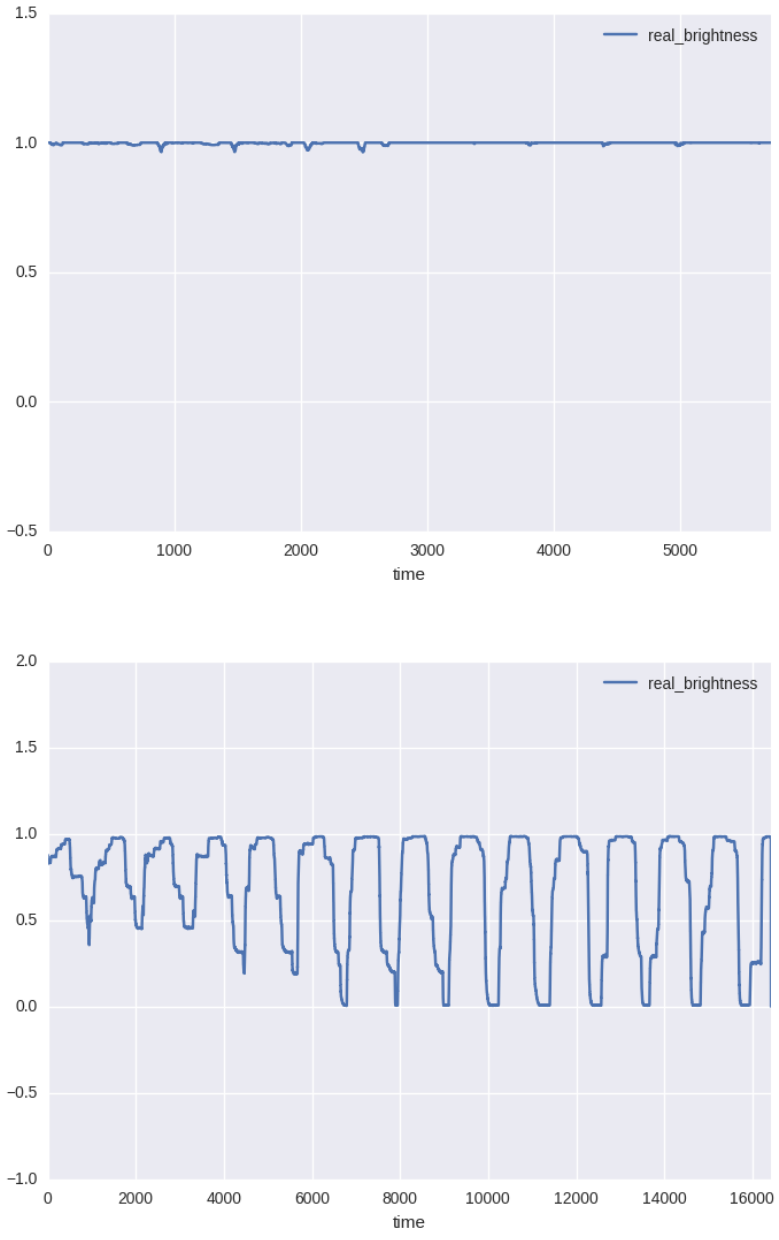

Figure 3: With heartbeat 50bpm, we demonstrate the synchronization of our system as a function of time. Above: the system in the absence of any gossip. The average brightness in the scene is consistently extremely high. Below: we run synchronization via gossip. The system starts out somewhat desynchronized, so the average brightness values of the system range from 0.4 to 1. Once the system is synchronized, the values range neatly from 0 to 1, the full possible range.

between unreliable hardware and buggy code can be indecipherable.

One intriguing aspect of our system is our dependence on unreliable Linux boxes. Due to the presence of these unreliable machines, debugging our implementation of gossip was extremely challenging.

2. Referring to the same memory address between different processes without use of locks is erroneous; using process-safe objects is extremely important when sharing memory between processes.

   We encountered a bug that took several hours to unravel where all the processes shared a queue that we were sure had the same memory address. Despite this, when something was successfully added to the queue by one process, only that process was able to see the change in the queue. The queue was a non process safe `Queue.Queue`. After switching to a `multiprocessing.queues.Queue`, the problem disappeared and the processes were able to correctly share the queue.

6.1. Sap

Serena is graduating in two weeks, and this project was her last piece of coursework at Harvard; an incredibly high note to end her time here!

Michelle is graduating in a year, but is really sad that this class is now over and she won't get to listen to Professor Waldo's hilarious lectures anymore.