Applications and Computation for the Internet of Things

3ST Lab Work: Internet of Sensors and Actuators

Group: 12		
Student 1	98380	Dominika Florczykowska
Student 2	97144	Pedro Mendes

1 Describe the changes in the program developed in lab 2 (Sensing the Real World) to port it to the new configuration with two controllers

One important change made to maintain coherence between the code of the two controllers is the addition of a `config.hpp` file where the values used pins are set, this file can be used as a configuration file or as a specification sheet for anyone trying to assemble this setup.

```
enum Led : u8 {
    Yellow = 5,
    Red = 9,
    Green = 11,
};
enum Sensors: u8 {
    STemperature = A2,
    SLight = A1,
    SPotentiometer = A0,
};
struct Config {
    Led led;
    Sensors analog;
};
constexpr Config Potentiometer = {Green, SPotentiometer};
constexpr Config Light = {Red, SLight};
constexpr Config Temperature = {Yellow, STemperature};
```

This way, when writing the code of the either of the modules inconsistent led to sensor choices can't be made.

1.1 Describe any changes in the design pattern (such as changing tasks in the round-robin loop)

In the sensor loop, we are now constantly checking for the sensor analogue reading and immediately sending the value via serial connection.

In the actuator loop, there are 2 tasks to run, the first one being later broken up into 3 more.

The first task is trying to read from the serial port, if this fails (because there was nothing to read or because the values read were invalid), then the sub tasks are not run.

In the case where the read succeeds than an update function is run for each of the 3 sensors.

The second task is to blink the green led, which means checking if the time turn it off or on has passed and change it's state to the opposite.

It's important to note that in this design when one value changes, all actuators are updated, this is only doable because there are so few actuators and updating them is very simple, as well as some other optimizations made in other parts of the project[1.2] [5].

We found this to simpler and allows for a higher data rate, as the amount of data needed in each message is 8 bytes and requires nearly zero parsing. This message is composed of 3 two byte values and a two byte error detection code.

If we had gone for a model where each message only pertains to one sensor, we would need to exchange more data to transmit the same amount of information. The message would need to be composed of 1 byte for de sensor identifier, 2 bytes for the value, and 1 or 2 bytes for the error detection code, all this times 3 sensors for a total of 12–15 bytes (depending on the length of error code). With the additional downside of needing to make 3 reads over the serial port which can be even slower.

These design choices are only possible because:

- The number of sensors is small;
- Their update functions are very short to compute;
- The fastest update rate of the blinking green led is slow enough.

Had any of these reasons been different we might have opted for more traditional approach like the alternative explained above.

1.2 Describe other implementation issues changed (implementation of tasks)

Two tasks suffered changes when transitioning to this model, the light detector task did, however, not change at all.

Temperature Task After reviewing the formula to convert an analogue read to degrees we came up with 3 new options and benchmarked them to see which was best

The benchmark was done as follows:

> Every 10000 iterations write 'r\n' to the serial port. Run the program for 5 minutes and gather how many 'r\n' are written to the serial port.

Version name	formula	Iterations/s
float 1 (original)	$(((voltage/1024.0) \times 5.0) - 0.5) \times 100.0$	12400
float 2	$((500.0 \times voltage)/1024.0) - 50.0$	13066
short (16bit)	$((50 \times voltage)/102) - 50$	14600
long (32bit)	$((500 \times (int32\_t)voltage) \gg 10) - 50$	15533

> **Note:** For the short (16bit) version, the formula had to be adapted to avoid overflows, hence the 50 instead of 500 and the 102 instead of the 1024.

Since the long (32bit) version was the fastest and the most accurate (using integers that is) we decided to implement this one.

Potentiometer Task This task was changed so that the updating part was separated from the checking when to swap on and off part.

The updating part can be seen in the `potentiometer` function and the checking part can be seen in the `blink_potentiometer` function, in the actuator code (Subsection 3).

2 Controler #1 program — I2C master:

```
#include "config.hpp"

// WARNING: This value must be a power of 2
#ifndef N_SAMPLES
#    define N_SAMPLES 16
#endif

class Sensor {
  private:
    size_t read_index;
    u16 total;
    u16 readings[N_SAMPLES];
    Sensors const pin;
    Led const led;

  public:
    char const sensor_name;

    constexpr explicit Sensor(Config const conf, char const sensor_name)
        : read_index{0},
          total{0},
          readings{},
          pin{conf.analog},
          led{conf.led},
          sensor_name{sensor_name} {}

    /**
     * Reads a value from the analog pin. Every 16 reads (or another value
     *  defined by N_SAMPLES) the passed handler is called with the averaged out
     *  value.
     *
     * @param value_handler A handler for the value
     */
    auto read() -> u16 {
        // subtract the last reading:
        total -= readings[read_index];
        // read from the sensor:
        readings[read_index] = analogRead(pin);
        // add the reading to the total:
        total += readings[read_index];
        // advance to the next position in the array:
        read_index = (read_index + 1) & (N_SAMPLES - 1);

        // calculate the average:
        return total / N_SAMPLES;
    }

    /**
     * Sets up the error indicator led of this sensor
     */
    void setup() {
        pinMode(led, OUTPUT);
    }
};


static Sensor SENSORS[] = {
    Sensor(Temperature, 'T'),
    Sensor(Light, 'L'),
```

```
    Sensor(Potentiometer, 'P'),
};

/** Check for errors comming from the actuator side */
void read_errors() {
    if (Serial.available()) {
        byte err[1];
        while (Serial.readBytes(err, sizeof err) < 1)
            ;
        digitalWrite(*err, HIGH);
    }
}

void setup() {
    Serial.begin(9600);
    pinMode(LED_BUILTIN, OUTPUT);
    for (auto& s: SENSORS) {
        s.setup();
    }
}

void loop() {
    for (auto& s : SENSORS) {
        auto value = s.read();
        char buf[10];
        buf[0] = s.sensor_name;
        sprintf(buf + 1, "%hd\n", value);
        Serial.print(buf);
    }
    read_errors();
}
```

3 Controler #2 program — I2C slave:

```
#include "config.hpp"

class Actuator {
  public:
    using Feedback = void (*)(Actuator const&, u16);

    constexpr explicit Actuator(Config const conf, Feedback const f)
        : feedback{f}, led{conf.led}, check_led{conf.analog} {}

    //** Setup this actuator's led
    void setup() const { pinMode(led, OUTPUT); }

    //** Run the update function passing it the new value
    void update(u16 value) const { (feedback)(*this, value); }

    /** Digital write to a led pin, reading it's output to test
     * if the led failed or not
     */
    void checked_digital_write(bool on) const {
        digitalWrite(led, on);
        if (on) test_and_report(100);
    }

    /** Analog write to a led pin, reading it's output to test
     * if the led failed or not.
     */
```

```
    void checked_analog_write(u8 value) const {
        analogWrite(led, value);
        if (value > 0) test_and_report(1);
    }

  private:
    const Feedback feedback;
    Led const led;
    u8 const check_led;

    // Check if this actuator failed and report this to the interface
    void test_and_report(u16 threshold) const {
        u16 voltage = analogRead(check_led);
        if (voltage < threshold) {
            u8 l = led;
            Serial.write(&l, sizeof led);
        }
    }
};


enum { TemperatureThreshold = 28 };

/* Update state functions */

/** Update temperature */
void temperature(Actuator const& self, u16 voltage) {
    int32_t degreesC = ((500 * (int32_t) voltage) >> 10) - 50;
    auto w = degreesC > TemperatureThreshold ? HIGH : LOW;
    self.checked_digital_write(w);
}

/** Update light */
void light(Actuator const& self, u16 voltage) {
    self.checked_analog_write(map(voltage, 1, 1021, 255, 0));
}

/** Update potentiometer */
static u32 half_blink_interval = 100;

void potentiometer(Actuator const& self, u16 voltage) {
    (void) self;
    half_blink_interval = map(voltage, 0, 1023, 100, 1000);
}


/** Blink green led task */
void blink_potentiometer(Actuator const& self) {
    static u32 last_blink_time = 0UL;
    static bool led_on = false;

    u32 const now = millis();
    if (last_blink_time + half_blink_interval < now) {
        self.checked_digital_write(led_on = !led_on);
        last_blink_time = now;
    }
}

#define NUM_SENSORS 3

constexpr Actuator SENSORS[NUM_SENSORS] = {
```

```
    Actuator(Temperature, temperature),
    Actuator(Potentiometer, potentiometer),
    Actuator(Light, light)};

//** Try to read from the serial port into `msg`
auto read_serial(u16* msg) -> bool {
    byte buf[(NUM_SENSORS + 1) * 2];
    if (!Serial.available()) return false;

    // Read to end
    size_t read_so_far = 0;
    while (read_so_far != sizeof buf)
        read_so_far +=
            Serial.readBytes(buf + read_so_far, sizeof(buf) - read_so_far);

    // Check error code
    if (buf[sizeof(buf) - 1] != 4 || buf[sizeof(buf) - 2] != 0xff) return false;

    // Parse the message (little endian encoding of 3 u16)
    msg[0] = buf[0] | ((u16) buf[1]) << 8;
    msg[1] = buf[2] | ((u16) buf[3]) << 8;
    msg[2] = buf[4] | ((u16) buf[5]) << 8;
    return true;
}

void setup() {
    for (auto& s : SENSORS) s.setup();
    pinMode(LED_BUILTIN, OUTPUT);
    Serial.begin(9600);
}

void loop() {
    static u16 msg[NUM_SENSORS + 1];
    if (read_serial(msg)) {
        for (size_t i = 0; i < NUM_SENSORS; ++i) SENSORS[i].update(msg[i]);
    }
    blink_potentiometer(SENSORS[1]);
}
```

4 PC #1 and #2 configuration and programs:

For communication between the arduinos we designed two programs in the programming language Go. These act as mere communicators between the 2 controllers.

4.1 The entry point of the command line program that interacts with the controllers

The first of these programs is connects to the serial port and to a relay server. Through command line arguments this program can be used both for the sensor and the actuator side. This program can also be used to "mock" a controller by reading from and writing to a terminal, this was very useful during testing.

This part of the code configures the application and the connects to the relay server, telling it which kind of interface is connecting: sensor or actuator.

```
package main

import (
  "communication/util"
  "encoding/binary"
  "flag"
```

```
  "fmt"
  "github.com/jacobsa/go-serial/serial"
  "io"
  "net"
  "os"
  "os/signal"
  "syscall"
)

type Options struct {
  Host string
  Port string
  Mode string
  Mock bool
}

func parseArgs() Options {
  var opt Options
  flag.StringVar(&opt.Host, "h", "iot-lab3.herokuapp.com", "the host to connect to")
  flag.StringVar(&opt.Port, "p", "80", "the port to connect to")
  flag.StringVar(&opt.Mode, "m", "sensor", "Either 'sensor' or 'actuator'")
  flag.BoolVar(&opt.Mock, "i", false, "read from terminal instead of arduino")
  flag.Parse()
  return opt
}

func main() {
  options := parseArgs()
  fmt.Println("Connecting")
  conn, err := net.Dial("tcp", fmt.Sprintf("%s:%s", options.Host, options.Port))
  if err != nil {
    fmt.Print("Error connecting to socket:", err.Error())
    os.Exit(1)
  }

  fmt.Println("Writing mode:", options.Mode)
  util.DoOrDie(util.WriteToEnd(conn, []byte(options.Mode)), "Error writting to socket: %v", err)

  var buffer [1]byte
  util.DoOrDie(util.ReadToEnd(conn, buffer[:]), "Error reading mode response: %v", err)
  if buffer[0] != 1 {
    fmt.Println("Invalid handshake")
    os.Exit(1)
  }
  fmt.Println("Connected to relay")

  var port io.ReadWriteCloser
  if options.Mock {
    port = Terminal{}
  } else {
    port = openSerial()
  }
  go handle_signals(conn)
  switch options.Mode {
  case "sensor":
    handleSensor(conn, port)
  case "actuator":
    handleActuator(conn, port)
  default:
    fmt.Println("Invalid type:", options.Mode)
    os.Exit(1)
```

```
  }
}

func openSerial() io.ReadWriteCloser {
  options := serial.OpenOptions{
    PortName:        "/dev/ttyACM0",
    BaudRate:        9600,
    DataBits:        8,
    StopBits:        1,
    MinimumReadSize: 1,
  }

  fmt.Println("Connecting to serial port")
  port, err := serial.Open(options)
  fmt.Println("Connected")
  if err != nil {
    fmt.Println("Failed to open serial port:", err.Error())
    os.Exit(1)
  }
  return port
}

func handle_signals(conn io.Closer) {
  sig_chan := make(chan os.Signal, 1)
  signal.Notify(sig_chan, syscall.SIGHUP, syscall.SIGINT, syscall.SIGQUIT)
  <-sig_chan
  fmt.Println("Signal received, shutting down")
  conn.Close()
  os.Exit(0)
}

// Mock an arduino, by reading stdin and writting to stdout instead
type Terminal struct{}

func (Terminal) Write(p []byte) (n int, err error) {
  var s string
  if len(p) == 1 {
    s = fmt.Sprintf("Led failed %d\n", p[0])
  } else if len(p) == int(util.PacketSize) {
    s = fmt.Sprintf(">> %d:%d:%d\n",
      binary.LittleEndian.Uint16(p[0:2]),
      binary.LittleEndian.Uint16(p[2:4]),
      binary.LittleEndian.Uint16(p[4:6]),
    )
  } else {
    s = "Unknown write type: ["
    for _, b := range p {
      s += fmt.Sprintf("%d,", b)
    }
    s += "]"
  }
  return os.Stdout.Write([]byte(s))
}

func (Terminal) Read(p []byte) (n int, err error) {
  n, err = os.Stdin.Read(p)
  if len(p) == 1 {
    switch p[0] {
    case 'G':
      p[0] = util.GREEN_LED
    case 'Y':
```

```
      p[0] = util.YELLOW_LED
    case 'R':
      p[0] = util.RED_LED
    }
  }
  return n, err
}

func (Terminal) Close() error {
  return nil
}
```

4.2 The code to interact with a sensor

The sensor part of this program does the parsing of the message sent from the sensing controller to the packet format described in Subsection 1.1 and sending said packet to the relay server.

```
package main

import (
  "bufio"
  "communication/util"
  "encoding/binary"
  "fmt"
  "io"
  "net"
  "strconv"
)

func handleSensor(conn net.Conn, port io.ReadWriteCloser) {
  defer port.Close()
  bufReader := bufio.NewReaderSize(port, 10)
  buffer := util.MakeEmptyPacket()
  err := util.ReadToEnd(conn, buffer[:])
  if err != nil {
    fmt.Println("Error getting current state", err.Error())
    buffer = util.MakePacket()
  }
  fmt.Println("Read the initial state", buffer)
  go report_errors(conn, port)
  for {
    line, err := bufReader.ReadString('\n')
    if err != nil {
      if err != io.EOF {
        fmt.Println("Error reading from port", err.Error())
      }
      break
    }
    switch line[0] {
    case 'T':
      parseToSlice(line[1:(len(line)-1)], buffer[0:2])
    case 'P':
      parseToSlice(line[1:(len(line)-1)], buffer[2:4])
    case 'L':
      parseToSlice(line[1:(len(line)-1)], buffer[4:6])
    }
    if err := util.WriteToEnd(conn, buffer[:]); err != nil {
      fmt.Println("Error reading from server", err.Error())
      break
    }
  }
```

```
}

func parseToSlice(s string, slice []byte) {
  v, err := strconv.Atoi(s)
  if err != nil {
    fmt.Printf("Error converting value from string: '%s'\n", s)
  } else {
    binary.LittleEndian.PutUint16(slice, uint16(v))
  }
}

func report_errors(conn net.Conn, port io.ReadWriteCloser) {
  for {
    var buf [1]byte
    n, err := conn.Read(buf[:])
    switch err {
    case io.EOF:
      fmt.Println("Terminating error detection")
      return
    case nil:
    default:
      fmt.Printf("Error reading '%v' terminating\n", err)
      return
    }
    switch buf[0] {
    case util.YELLOW_LED:
      fmt.Printf("Yellow led failed\n")
    case util.RED_LED:
      fmt.Printf("Red led failed\n")
    case util.GREEN_LED:
      fmt.Printf("Green led failed\n")
    default:
      if n > 0 {
        fmt.Printf("Invalid byte received %v\n", buf[0])
      }
      continue
    }
    util.WriteToEnd(port, buf[:])
  }
}
```

4.3 The code to interact with the actuator side

The actuator part reads from the relay server and writes to the serial port, since the message format is the same, no parsing is done, only the error code is checked as to ignore faulty messages. One final check is done before sending the message, to avoid redundant messages it checks if this message is a repeat of the last message and if so it's ignored. This avoids overloading the controller with useless information.

```
package main

import (
  "communication/util"
  "fmt"
  "io"
  "net"
  "time"
)

func handleActuator(conn net.Conn, port io.ReadWriteCloser) {
  defer port.Close()
```

```
  buffer := util.MakeEmptyPacket()
  old := util.MakeEmptyPacket()
  first_write := true
  go detect_errors(conn, port)
  for {
    if err := util.ReadToEnd(conn, buffer[:]); err != nil {
      fmt.Println("Error reading from server", err.Error())
      break
    }
    if !util.IsPacketValid(buffer) {
      continue
    }
    if old != buffer {
      fmt.Printf("Writing %v\n", buffer)
      err := util.WriteToEnd(port, buffer[:])
      if err != nil {
        fmt.Printf("Failed to write %v Reason: %v\n", buffer, err)
        return
      }
      if first_write {
        time.Sleep(2 * time.Second)
        err = util.WriteToEnd(port, buffer[:])
        if err != nil {
          fmt.Printf(
            "Failed to make backup write %v Reason: %v\n",
            buffer,
            err,
          )
          return
        }
        first_write = false
      }
      copy(old[:], buffer[:])
    }
  }
}

func detect_errors(conn net.Conn, port io.ReadWriteCloser) {
  for {
    var buf [1]byte
    n, err := port.Read(buf[:])
    switch err {
    case io.EOF:
      fmt.Println("Terminating error detection")
      return
    case nil:
    default:
      fmt.Printf("Error reading '%v'\n", err)
      continue
    }
    if n > 0 && contains([]byte{util.YELLOW_LED, util.RED_LED, util.GREEN_LED}, buf[0]) {
      util.WriteToEnd(conn, buf[:])
    }
  }
}

func contains(s []byte, e byte) bool {
  for _, a := range s {
    if a == e {
      return true
    }
```

```
  }
  return false
}
```

4.4 Relay server, establish communication between the two communication programs

The relay server serves as a mediator between the two interfaces, it stores the most recent state and sends it to the actuator interface that is connected.

It also guarantees that the errors are sent from one interface to the other.

The relay only allows one actuator and one sensor to be connected at a time.

```
package main

import (
  "communication/util"
  "encoding/binary"
  "fmt"
  "net"
  "os"
  "sync"
  "time"
)

const (
  DEFAULT_PORT = "80"
)

var VALUES1 = util.MakePacket()
var VALUES2 = util.MakePacket()
var READ_VALUES = &VALUES1
var WRITE_VALUES = &VALUES2
var READ_MUTEX sync.Mutex
var HAS_SENSOR bool
var HAS_ACTUATOR bool

func main() {
  go printState()
  var port string
  if len(os.Args) > 1 {
    port = os.Args[1]
  } else if p := os.Getenv("PORT"); p != "" {
    port = p
  } else {
    port = DEFAULT_PORT
  }
  l, err := net.Listen("tcp", "0.0.0.0:"+port)
  if err != nil {
    fmt.Println("Error listening: ", err.Error())
    os.Exit(1)
  }
  fmt.Printf("Listening on 0.0.0.0:%s\n", port)
  defer l.Close()
  error_channel := make(chan byte, 512)
  for {
    conn, err := l.Accept()
    if err != nil {
      fmt.Println("Error accepting: ", err.Error())
      continue
    }
    fmt.Println("Client connected")
```

```
    var buf [1024]byte
    reqLen, err := conn.Read(buf[:])
    if err != nil {
      fmt.Println("Error reading")
      continue
    }
    kind := string(buf[:reqLen])
    switch {
    case kind == "sensor" && !HAS_SENSOR:
      HAS_SENSOR = true
      fmt.Println("Sensor connected")
      go handleSensor(conn, error_channel)
    case kind == "actuator" && !HAS_ACTUATOR:
      HAS_ACTUATOR = true
      fmt.Println("Actuator connected")
      go handleActuator(conn, error_channel)
    default:
      fmt.Println("Invalid type or already has one connected", kind)
      fmt.Println("Has sensor: ", HAS_SENSOR)
      fmt.Println("Has actuator: ", HAS_ACTUATOR)
      if err := util.WriteToEnd(conn, []byte{0}); err != nil {
        fmt.Println("Couldn't send rejection byte")
      }
      conn.Close()
    }
  }
}

func handleSensor(conn net.Conn, error_channel chan byte) {
  defer conn.Close()
  defer func() { HAS_SENSOR = false }()
  if err := util.WriteToEnd(conn, []byte{1}); err != nil {
    fmt.Println("Couldn't send confirmation byte")
    return
  }
  bytes := READ_VALUES
  if err := util.WriteToEnd(conn, bytes[:]); err != nil {
    fmt.Println("Failed to write initial state, continuing..")
  }
  quit := false
  go receive_errors(conn, error_channel, &quit)
  for {
    if err := util.ReadToEnd(conn, WRITE_VALUES[:]); err != nil {
      fmt.Println("Sensor disconnecting: ", err.Error())
      quit = true
      HAS_SENSOR = false
      return
    }
    swap_pointers()
  }
}

func receive_errors(conn net.Conn, error_channel chan byte, quit *bool) {
  for {
    select {
    case err := <-error_channel:
      fmt.Println("Receiving error", err)
      util.WriteToEnd(conn, []byte{err})
    default:
      if *quit {
        fmt.Println("receive_errors stopping")
```

```
        return
      }
    }
  }
}

func handleActuator(conn net.Conn, error_channel chan byte) {
  defer conn.Close()
  defer func() { HAS_ACTUATOR = false }()
  if err := util.WriteToEnd(conn, []byte{1}); err != nil {
    fmt.Println("Couldn't send confirmation byte")
    return
  }
  go send_errors(conn, error_channel)
  for {
    READ_MUTEX.Lock()
    var bytes = *READ_VALUES
    READ_MUTEX.Unlock()
    if err := util.WriteToEnd(conn, bytes[:]); err != nil {
      fmt.Println("Actuator disconnecting: ", err.Error())
      HAS_ACTUATOR = false
      return
    }
  }
}

func send_errors(conn net.Conn, error_channel chan byte) {
  var buf [1]byte
  for {
    if n, err := conn.Read(buf[:]); err != nil {
      return
    } else if n == len(buf) {
      fmt.Println("Sending error", buf[0])
      select {
      case error_channel <- buf[0]:
      default:
      }
    }
  }
}

func swap_pointers() {
  tmp := READ_VALUES
  READ_MUTEX.Lock()
  READ_VALUES = WRITE_VALUES
  READ_MUTEX.Unlock()
  WRITE_VALUES = tmp
}

func printState() {
  for {
    time.Sleep(5 * time.Second)
    bytes := *READ_VALUES
    fmt.Printf(
      "bytes: %v | T%d | P%d | L%d | C%d\n",
      bytes,
      binary.LittleEndian.Uint16(bytes[0:2]),
      binary.LittleEndian.Uint16(bytes[2:4]),
      binary.LittleEndian.Uint16(bytes[4:6]),
      binary.LittleEndian.Uint16(bytes[6:8]),
    )
```

```
  }
}
```

5 Evaluate the performance of the communication link

The program that interacts with the arduino on the actuator side is smart to not write repeated messages consecutive, this way the arduino doesn't have to do useless work and has more "free time". As such the data rates might seem a bit low. But this is only because most of the data is redundant and not counted. However, to give a sense of how much difference this makes, the code was temporarily altered to not make this check.

Type	Average (bytes/s)	Maximum (bytes/s)	Minimum (bytes/s)
without repeats	682	752	461
with repeats	805286	876945	521628

6 Design interfaces to detect failures of the LEDs and signal these occurrences at the sensors side. (Consider that when a LED fails it behaves as an open-circuit.)

6.1 Draw the circuits

On the sensor side we have leds connected to the board which we use to signal an error on the other side.

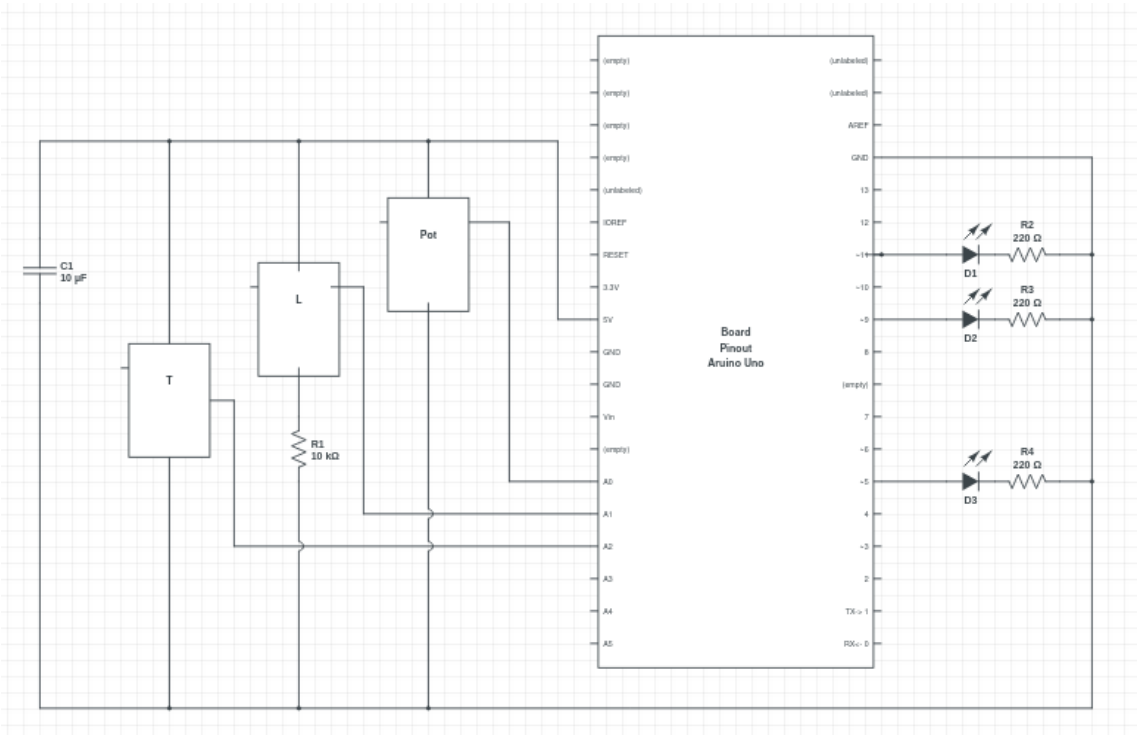

Figura 1: Sensor

On the actuator side, we have the "output" of the LEDs being fed to analogue pins so that their value can be inspected

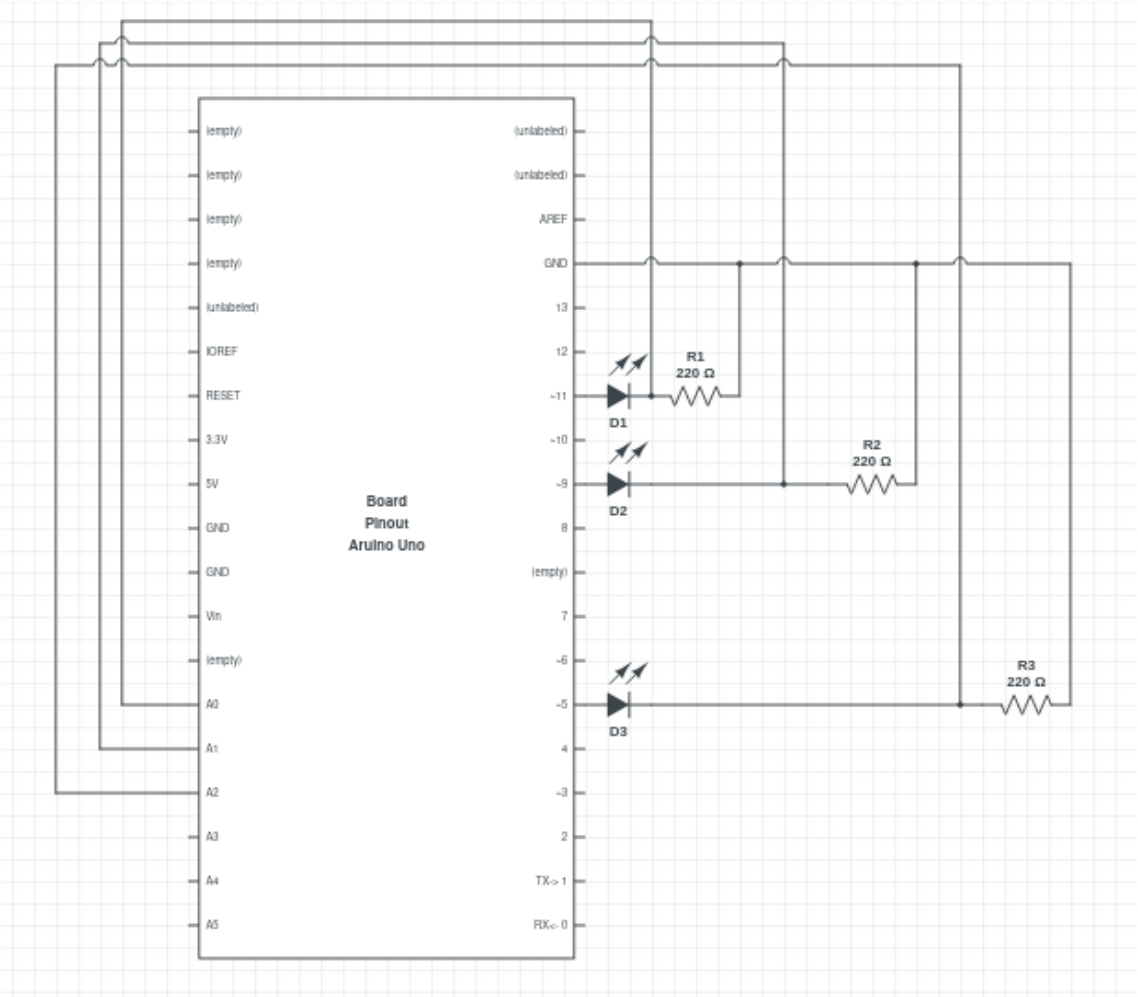

Figura 2: Actuator

Latency was not tested as it was very dependent on the speed of the internet connection, if that is not taken in to account latency is negligible.

6.2 Program the detection

For yellow and green leds (the ones that are activated by digital writes) we replaced the calls to `digitalWrite` with a `checked_digital_write` function. For the red led we use a `checked_analog_write` function, these are very similar except for the for the fact that one does a digital write and the other an analogue write. The used threshold passed to the test and report function is also different, this value unfortunately couldn't not be thoroughly tested due to lack of time, it does however work for most cases.

```
void update(u16 value) const { (feedback)(*this, value); }

/** Digital write to a led pin, reading it's output to test
 * if the led failed or not
*/
void checked_digital_write(bool on) const {
    digitalWrite(led, on);
    if (on) test_and_report(100);
}
```

Which delegate the test and reporting this function:

```
    if (value > 0) test_and_report(1);
}
```

```
private:
  const Feedback feedback;
  Led const led;
  u8 const check_led;
```

6.3 Describe any changes required in the communication interfaces

The actuator side of the communication interface now has to also send things to the sensor side, since TCP sockets are two way channels this wasn't very hard to implement.