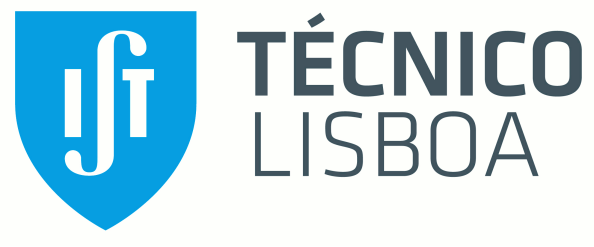


Sistemas de Controlo Distribuído em Tempo-Real

Real-Time Cooperative Decentralized Control of a Smart Office Illumination System

Grupo N.º 14

Autores:

Diogo Rodrigues 84030
Leandro Almeida 84112
Pedro Moreira 85228

Professores:

Alexandre Bernadino
João Pedro Gomes

13 de Junho de 2019

Resumo

O trabalho que vai ser reportado no seguinte relatório teve em vista o desenvolvimento de um sistema de controlo distribuído e em tempo real para a iluminação de um escritótio composto por várias secretárias. Este cenário foi simulado representando cada secretária por um arduino (responsável pelo seu controlo) ligado a um circuito elétrico capaz de ler o ambiente luminoso (com um LDR) e impôr luz sobre o mesmo (com um LED). O objetivo principal do projeto foi minimizar o consumo energético das luminárias (os LEDs), sem que para isso fosse posto em causa o conforto do(s) utilizadore(s). O controlo é distribuído pois é necessário delinear o comportamento de cada uma das secretárias, tendo em conta o efeito que têm umas nas outras, e é também em tempo real pois todo o sistema está sujeito a contantes alterações, pelo que se torna necessário agir constantemente sobre o mesmo em função das condições atuais. Para que pudesse ser feita uma consulta/recolha de dados, também em tempo real, para efeitos de manutenção/estatísticos, foi desenvolvido um servidor capaz de recolher e interpretar todos os dados relevantes das secretárias e fornecê-los a um ou vários clientes, consoante os pedidos. O servidor consistiu num programa desenvolvido em C++, a correr numa máquina UNIX capaz de fazer a recolha de dados(*Raspberry Pi*).

Palavras-chave: *consensus*, *feedforward*, *anti-windup*, *deadzone*, *feedback*, i^2c, *jitter*, *flickering*, *ack*, C++, *asynchronous programming*, Boost.ASio *library*, TCP *socket programming*, *Multi-Threading*, *mutual exclusion variables*, *condition variables*, PGPIO *library*.

1 Introdução

Com o passar dos anos, os sistemas de iluminação utilizados têm vindo a tornar-se cada vez mais eficientes a nível do consumo energético, no entanto, a um ritmo mais baixo do que se poderia esperar tendo em conta as tecnologias que existem à disposição. Um dos maiores avanços da última década, relativamente a esta área, foram os LEDs (*Light Emitting Diodes*), que apesar do baixo custo, quando em comparação com as lâmpadas fluorescentes, trabalham a valores de tensão de menor escala e revelam uma menor dissipação de calor. Assim sendo, o uso de LEDs foi óbvio. Outro tópico bastante levantado neste campo, assenta nos sistemas de iluminação autónomos que, muito raramente, agem consoante a ocupação (ou não) do espaço iluminado, desperdiçando assim energia a gerar luz a mais desnecessáriamente. A maior motivação para este projeto foi então o "esbanjamento"de energia que é observado na larga maioria dos sistemas de iluminação. Conseguiu-se construir um sistema que age, autonomamente, tendo em conta todas as variantes faladas anteriormente.

Uma das maiores barreiras atravessadas neste projeto foi assumir a capacidade de desenvolver um produto escalável, que não só funcionasse corretamente no espaço de trabalho que foi pedido (e testado), mas que também tivesse a capacidade de ser implementado num sistema de maior dimensão, sendo capaz de trabalhar da mesma forma e com a mesma eficiência.

Um trabalho que trata de resolver o mesmo problema que o abordado é o presente em [1]. A solução apresentada em [1] apresenta uma topologia centralizada, em que um nó tem conhecimento de toda a rede. O nosso projeto apresenta uma topologia descentralizado(que como se verá à frente traz vantagens para o problema em questão), onde se assume a mesma responsabilidade e capacidade a todos os nós, e em que o controlo distribuído é realizado através do algoritmo **consensus**.

2 Configuração do Sistema

Para simular a sala de trabalho em que cada secretária tem a sua própria iluminária, usa-se uma caixa de sapatos, com o interior revestido de branco, de modo a que haja maior reflexão da luz.

Cada secretária contém um microcontrolador (*Arduino* **UNO**), que representa um nó, bem como um sistema de iluminação (com um LED) e um LDR (*Light Dependent Resistor*), que é um sensor de luz que tem como objetivo ler a iluminação atual. É utilizado também um *Raspbery Pi* **3B** que serve como servidor. As ligações entre todos os componentes são feitas através de uma *breadboard*, como se pode verificar através da figura 1, realizada para duas secretárias.

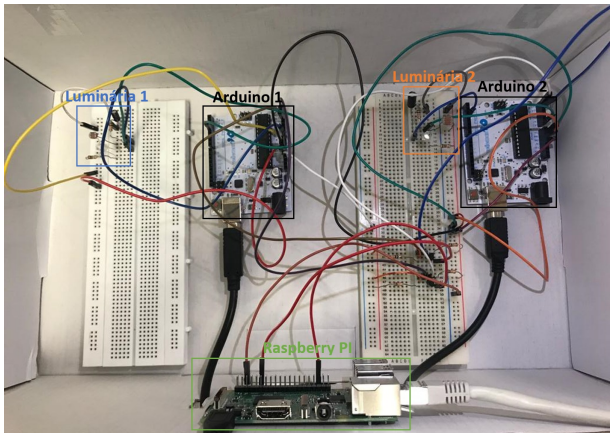
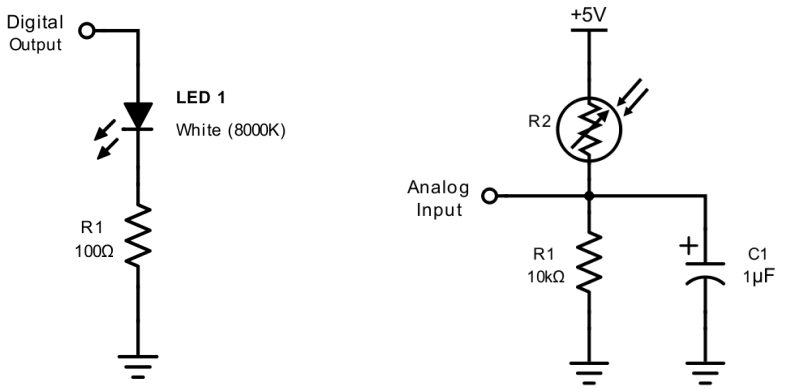


Figura 1: Disposição do modelo físico (caixa de sapatos) do escritório

3 Identificação do Sistema

3.1 Identificação Individual do Sistema

Numa primeira fase, analisou-se individualmente cada luminária. A figura 2 ilustra os circuitos presentes em cada.



(a) Circuito do LED (b) Circuito de leitura da iluminação

Figura 2: Esquema do atuador e do sensor

A intensidade luminosa do LED é definida pelo PWM (*Pulse-Width Modulation*), que consiste em alternar entre acender e desligar o LED muito rapidamente de acordo com o *duty-cycle* imposto. Este efeito, ao olho humano, é aproximado ao de uma fonte de luz contínua com intensidade variável. Contudo, estas transições não são instantâneas, o que leva a que haja ruído de alta frequência. Para minimizar este efeito, introduz-se um condensador, que funciona como filtro passa-baixo, reduzindo assim o ruído captado pelo sensor.

3.1.1 Calibração do LDR

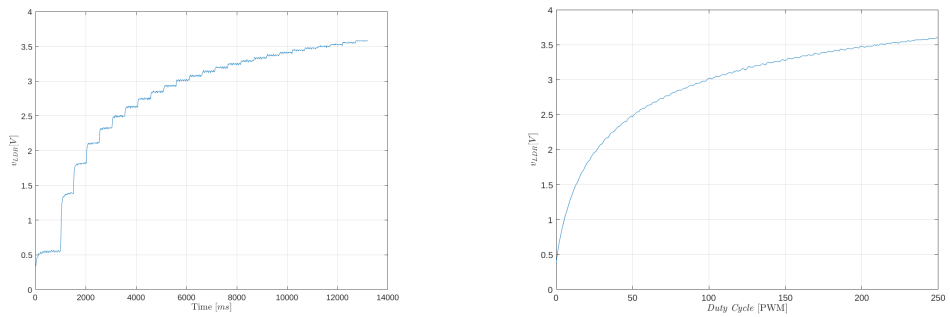
Primeiramente, é necessário converter a tensão medida no sensor para as unidades **SI** de iluminação, conhecida como LUX. A resistência do LDR altera-se quando exposta a luz, sendo que diminui quanto maior for a intensidade de luz incidente. Isto pode ser observado através expressão do divisor de tensão, em estado estacionário, aplicado ao circuito da figura 2(b):

$$V = V_{CC}\frac{R_1}{R_1 + R_2(x)} \Leftrightarrow R_2(x) = R_1\left(\frac{V_{CC}}{V} - 1\right), \tag{1}$$

onde x é a ilumínção no LDR (em LUX), V_{CC} é a tensão de alimentação (5V) e V é a tensão medida no pin analógico. Olhando para a carateristica do LDR (prensente em [2]), retira-se que a iluminação depende da resistência medida através de $\log_{10}\left(R_2(L)\right) = m\log_{10}(L) + b$. Como primeira referência as constantes m e b calcularam-se fazendo uso de um luxímetro, onde através da recolha de vários pontos se calculou uma regressão linear. Sabe-se que em estado estacionário incrementos no LED correspondem a incrementos idênticos nos LUX (ou seja, o sistema tem um ganho DC) e há portanto uma relação linear entre PWM e LUX. Devido a fatores externos, como por exemplo a diferença do ângulo do luxímetro relativamente ao LDR os resultados obtidos por este método não corresponderam ao esperado. Após feitos alguns ajustes experimentais (principalmente ao m), a partir da primeira referência, de forma a tornar a caraterística PWM-LUX linear, obtiveram-se os valores $m = -0.7$ e $b = 1.9252$ (para um dos *arduinos*). Para o outro alterou-se o valor de $b = 2.2472$ de forma a que ambos, para um determinado valor de PWM, tivessem valores de LUX semelhantes.

3.1.2 Linearização do Sistema

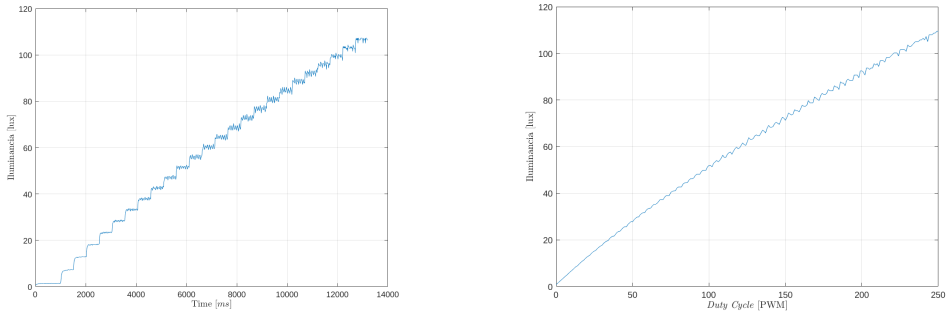
O LDR é um elemento não-linear, isto é, o ganho (que é o quociente entre a variação da iluminação "produzida" pelo LED e a variação da tensão medida) varia consoante o ponto de funcionamento. Isto pode ser demonstrado fazendo pequenas alterações no PWM de 0 a 255, com o PWM a variar de 10 em 10 (figura 3(a)). A resposta em estado estacionário com as várias referências de PWM a variar de 1 em 1, está representado na figura 3(b).



(a) Variações do PWM (b) Estado estacionário

Figura 3: Caraterística da tensão

Verifica-se desta forma que a resposta da tensão com a referência do PWM é não-linear, contudo fazendo a conversão da tensão para LUX, tal como descrito na secção 3.1.1, obtém-se uma caraterística linear do sistema como representado na figura 4.



(a) Variações do PWM (b) Estado estacionário

Figura 4: Caraterística da intensidade luminosa

3.1.3 Sistema de Primeira Ordem Equivalente

O objetico é modelar o sistema como um sistema de primeira ordem, descrito pela função transferência $G(s) = K_0(x)/\left(1 + s\tau(x)\right)$, onde x é a iluminação pretendida. Para tal, efetuaram-se medições do valor da iluminação para diferentes valores impostos de PWM. Na figura 5 encontra-se algumas das medições efetuadas.

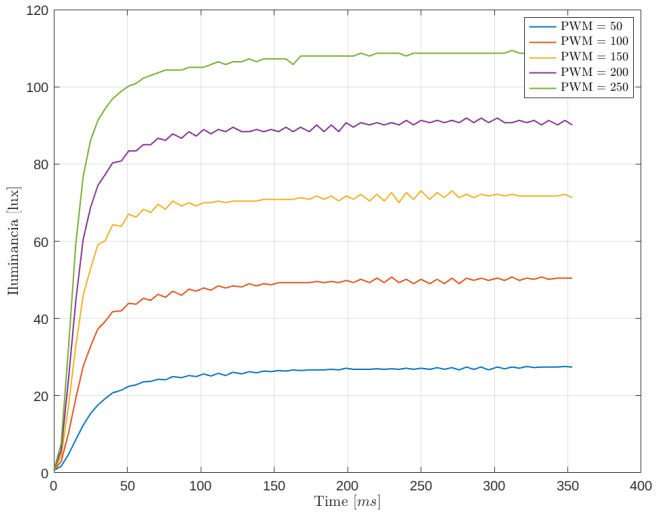


Figura 5: Resposta da intensidade luminosa para diferentes entradas de referência de PWM

As respostas obtidas são do tipo $g(t) = K_0\left(1 - e^{-\frac{t}{\tau}}\right)u(t)$. Em que a estimação do valor do ganho estático K_0 é feita através do quociente entre o sinal de saída estacionário e o valor de entrada. A contasnte de tempo, τ, é o instante de tempo em que a resposta toma um valor de aproximadamente $\left(1 - e^{-1}\right) \simeq 63\%$ do valor final. Na tabela 1 apresentam-se os valores obtidos para as constantes referidas.

PWM	K_0	$\tau\,(ms)$
25	0.49	33.75
50	0.47	27.91
75	0.46	24.55
100	0.46	23.37
125	0.45	21.91
150	0.44	19.79
175	0.44	19.14
200	0.43	18.01
225	0.42	17.22
250	0.41	16.50

Tabela 1: Valores obtidos para o ganho estático $K_0(x)$ e a constante de tempo $\tau(x)$

O circuito da figura 2(b) é um sistema dinâmico, em que é possível estabelecer a relação

$$\dot{v}\tau(x) = -v + V_{CC}\frac{R_1}{R_1 + R_2(x)}, \tag{2}$$

com

$$\tau(x) = R_{eq}(x)\,C_1 \quad \text{e} \quad R_{eq}(x) = \frac{R_1 R_2(x)}{R_1 + R_2(x)}. \tag{3}$$

A expressão (3) permite explicar o porquê de na tabela 1 se verifcar uma diminuição do valor da constante de tempo quanto maior for a intensidade luminosa imposta, uma vez que a resistência R_2

diminui com o aumento da iluminação. Assim sendo, a resposta do sistema será mais rápida quanto maior for o valor de PWM imposto.

Para um dado x (iluminação pretendida), a tensão v lida no pin analógico do circuito de leitura comporta-se como um sistema de primeira ordem como já vimos, pelo que a solução da equação (2) é dada por

$$v\left(t\right) = v_f - \left(v_f - v_i\right) e^{-\frac{t-t_i}{\tau\left(x_f\right)}}, \tag{4}$$

em que t_i é o instante de tempo da alteração; v_i o valor inicial de v (mesmo antes da alteração); v_f o valor final de v (estado estacionário) e x_f o valor final de x (estado estacionário). Refere-se que $\tau\left(x_f\right)$ é calculado através de um polinómio de segundo grau aproximado a partir dos pontos da tabela 1.

A expressão (4) será utilizada daqui para a frente como sendo a simulação da tensão lida no circuito de leitura da iluminação sempre que uma referência de iluminação seja alterada.

4 Calibração

Com a calibração relaciona-se os valores medidos de intensidade de luz, com o valor de PWM que se aplica ao LED, obtendo-se assim um ganho.

A rede é constitída por sistemas individuais. Cada sistema apresenta um ganho independente dos outros. Por sua vez, existem ganhos entre cada sistema individual, pois a iluminação de um pode influenciar o valor de iluminação lido por outro.

4.1 Uma secretária

Para o caso em que existe apenas uma secretária, assumindo que se tem uma resposta linear de iluminação com PWM, tem-se $L = K \cdot d + o$, em que o representa a perturbação externa, d o valor de PWM e K é o declive da reta da figura 4(b). O primeiro parâmetro é obtido através da leitura da iluminação quando se desliga o LED. O segundo parâmetro é calculado através de dois pontos, associados a dois diferentes valores de PWM.

4.2 *N* secretárias

Devido ao efeito de acoplamento existente, causado pelas outras secretárias, no caso da existência de N secretárias, tem de haver uma matriz de ganhos de dimensão N x N. Os elementos da diagonal da matriz representam a dependência do sistema de iluminação consigo próprio, ao passo que as outras entradas representam a influência causada pelos outros nós da rede. Assumindo uma resposta de iluminação linear com o valor de PWM e tendo também em conta a perturbação externa sentida por cada nó, tem-se o sistema:

$$\begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{bmatrix} = \begin{bmatrix} K_{11} & K_{12} & \dots & K_{1N} \\ K_{21} & K_{22} & \dots & K_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ K_{N1} & K_{N2} & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{bmatrix} + \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_N \end{bmatrix} \tag{5}$$

Realça-se que como uma pequena mudança na disposição do circuito dentro da caixa, ou uma pequena alteração na perturbação externa, leva a uma mudança dos ganhos, pelo que sempre que se inicia o sistema efetua-se o sistema de calibração.

Cada arduino calcula o seu o com todos os LEDs desligados. A entrada K_{ij} é calculada pela secretária i quando está apenas o LED da secretária j ligado.

Em detalhe, descreve-se em pseudocódigo o processo de `Calibração`, onde apenas o nó com o índice mais baixo realiza o método `início`, que permite que todos leiam a sua perturbação externa. Depois, ao passar para o método `Recalibração`, é o único que liga a iluminária com o valor máximo e dá indicação para os restantes calcularem a sua influência (para o caso de ser o nó i é calculada a coluna i da matriz K). Recebida a confirmação de que todos calcularam a sua influência, dá indicação

ao próximo nó para fazer o mesmo a partir do método `Recalibração`, garantindo-se desta forma o síncronismo.

Algorithm 1: Processo de `Calibração` do sistema geral

Início
- Apaga o LED;
- Dá indicação para os outros nós desligarem o LED;
- Aguarda confirmação que todos desligaram;
- Indicação para os outros nós lerem a perturbação externa, e ele próprio lê;
- Aguarda respostas de confirmação;

Recalibração
- Liga o LED no máximo;
- Cálculo do seu próprio K e indicação para todos calcularem o K_i;
- Aguarda confirmações;
- Desliga o seu LED;
- Verifica o próximo nó (de acordo com o índice);

if existe
- Dá indicação para o outro nó mudar o seu estado para `Recalibração`;

else
- Muda o estado para **consensus** pois é o que tem o maior índice;

5 Controlo

5.1 Controlador

No funcionamento normal do sistema, cada nó tem uma intensidade luminosa pretendida, que irá ser a referência do sistema a seguir. Isto é garantido através do controlador. Este controlador é constituido por dois componentes fundamentais: controlador **Feedforward** e controlador **Feedback**, em que este último é um controlador Proporcional Integral (PI). A figura 6 apresenta um diagrama de blocos detalhado do controlador implementado.

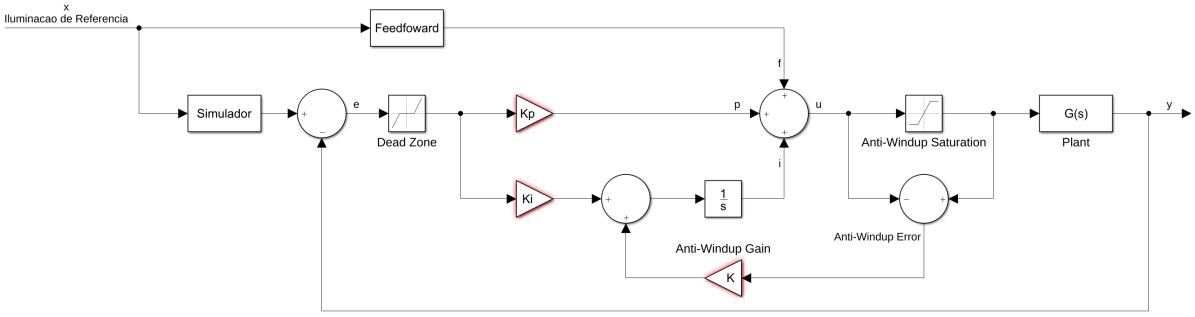


Figura 6: Diagrama de Blocos do Controlador

5.1.1 Controlador Feedforward

O objetivo do controlador **Feedforward** é guiar o LED em malha aberta para alcançar um primeiro valor de iluminação de referência. Obviamente que este valor não será exatamente o pretendido devido às perturbações externas e erros de modelação, mas é muito importante pois permite acelerar a resposta do sistema. Para corrigir este valor existe o controlador PI(descrito à frente). O valor de saída do controlador **Feedforward**, f (é o valor de PWM final que vai dar origem ao valor de iluminação de referência) é dado por $f = \frac{x-o}{G_0}$, em que x é a iluminação pretendida, o é iluminação exterior medida inicalmente e G_0 é o ganho do sistema. No caso do controlo distribuído(secção seguinte), o termo x é a iluminação de referência resultante do algoritmo aplicado e G_0 é o K_{ii} (para o nó i) resultado da calibração efetuada. Conclui-se assim que o **Feedforward** permite obter uma resposa rápida do sistema quando ocorre uma mudança da referência, no entanto não reage a perturbações.

5.1.2 Controlador Feedback - termo PI

De notar desde já que não foi implementado o termo derivativo. Este termo fornece uma correção antecipada do erro, acelerando a resposta do sistema e diminuindo o *overshoot*, no entanto é muito sensível ao ruído. Não é necessário a sua utilização em sistemas de resposta rapida, como neste caso em que a resposta é quase imediata. Como referido anteriormente, o controlador PI tem como objetivo corrigir o valor final do **Feedforward** de forma a levar o sistema até a referência. Este controlador consiste num termo proporcional e outro integral, dado por

$$u(t) = \underbrace{K_p e\left(t\right)}_{p(t)} + \underbrace{K_i \int_0^t e\left(t'\right) dt'}_{i(t)}, \tag{6}$$

em que $e\left(t\right)$ é o erro (diferença entre a saída do simulador e a tensão medida). Realça-se que como o sistema é um sistema discreto, usa-se uma transformação bilinear para a implementação do controlador PI, no cálculo do termo proporcional e integral.

O termo proporcional produz um sinal de saída que é proporcional à amplitude do erro $e\left(t\right)$, aumentando assim a velocidade de resposta. Como contrapartida, pode levar a algum *overshoot* e aumentar a frequência de oscilação, ou mesmo que o sistema não estabelize (dependente do valor de K_p).

O termo integral é proporcional à magnitude e à duração do erro, ou seja, ao erro acumulado. É essencial para a eliminação do erro em regime estacionário gerado pelo termo proporcional e acelera a resposta do sistema, permitindo-o chegar ao valor de referência mais rapidamente.

Conclui-se assim que o uso do **Feedback** permite reagir a perturbações e correção de possíveis desvios.

5.1.3 Outras Melhorias

Mesmo depois de implementado o controlador com a combinação entre o controlador **Feedforward** e o controlador PI, houve alguns problemas que não foram solucionados. De seguida, descrevem-se estes problemas, assim como a sua resolução.

- ***Anti-Windup:*** O *windup* é um fenómeno que ocorre no controlador PI quando há uma mudança abruta na referência ou quando as perturbações externas levam a que o termo integral acumule um erro absoluto muito elevado. Isto irá resultar em atrasos na resposta ou *overshooting*. Um exemplo prático deste projeto acontece quando é estabelecido um valor de referência inferior ao valor que se está a ler na saída, devido a uma perturbação externa. Assim, o erro que sai do simulador é negativo, pelo que o integrador vai acumular erros negativos durante muito tempo, uma vez que o sistema não consegue "retirar" mais luminosidade. Ao se retirar a perturbação externa, o sistema vai demorar bastante a convergir (pois tem de descarregar o que está acumulado).

  A solução passa por colocar um bloco de **Anti-Windup** que irá limitar a saída do atuador para os valores limite de PWM (0 e 255). Assim, soma-se ao integrador a diferença entre saída do controlador e a do atuador, que é a saturada, multiplicando a mesma por um ganho de **Anti-Windup**. Desta forma descarrega-se o integrador. Quando a saída satura o termo integral é recalculado de forma a ficar dentro dos limites de saturação. Este exemplo prático é demonstrado na secção 9.

- ***Deadzone:*** Uma vez que se tem um conversor analógico-digital de 10 bits com um alcance de 5V, existem erros de quantização(precisão de leitura de aproximadamente 5 mV). Estes erros levam a que haja **flicker**. Para miminizar o **flicker**, implementou-se a **deadzone**, que consiste em ignorar os erros que se encontram dentro de uma banda de valores pequenos. Estas pequenas alterações que o controlador tentava fazer aos valores contidos nesta banda apenas traziam oscilações indesejadas.

- ***Filtro Passa-baixo:*** Mais uma vez de forma a minimizar as oscilações em torno da referência devido a erros de quantização no conversor AC/DC, foi implementado um filtro digital. Este filtro consiste em adquir várias amostras por cada período de amostragem e calcular a média dos valores. O número de leituras escolhida foi de 20 de forma a garantir que eram todas feitas num intervalo significativamente inferior ao do período de amostragem sem influenciar o funcionamento

normal do controlador. Desta forma o sinal de saída oscila com menores amplitudes em relação ao sinal de referência.

5.1.4 Determinação dos parâmetros

Os parâmetros do controlador PI, numa primeira abordagem, foram calculados através do Método da Resposta em Frequência de *Ziegler Nichols*, que é baseado na procura experimental do ponto crítico de Nyquist. Este método consiste em, utilizando apenas o termo proporcional do controlador, aumentar o ganho até atingir um ganho (o ganho crítico) em que o sistema comece a oscilar. Daí retira-se o período crítico de oscilações. A partir desses dois parâmetros, é possível através da tabela presente no slide 15 de [3] determinar os parâmetros K_p e K_i.

Contudo, após uma primeira referência para estes valores, fez-se um ajuste experimental, uma vez que com o médoto descrito os resultados apresentados não estavam a ser precisos.

No que toca, ao ganho de **Anti-Windup**, este ganho também foi ajustado de modo experimental de forma a que o integrados "descarregasse"o mais rápido possível.

5.2 Controlo distribuído - Algoritmo de optimização

Pretende-se calcular a solução óptima para a saída de cada nó de modo a que todos eles satisfaçam os seus objetivos (ou seja o valor de iluminação desejada para o grau de conforto), garantindo que as restrições impostas (associadas a limites físicos) estão a ser cumpridas. Utilizou-se uma topologia descentralizada, em que cada secretária apenas trata de cumprir os seus objetivos mediante as restrições impostas, e comunica para todas as outras o seu resultado. Não existe uma secretária que tem conhecimento de todos os parâmetros da rede e que executa todo o processamento do algoritmo, pelo que esta solução é mais rápida em cada iteração. Além disso, tambem é uma melhor opção devido à limitação de memória existente no *arduino*. No entanto para uma secretária atingir o seu objetivo tem de ter em conta os efeitos causados por todas as outras secretárias, pelo que tem de se optimizar uma função de custo que contabiliza estes efeitos, e, através de comunicação entre todas as secretárias, chega-se coletivamente a uma solução global, que deve ser óptima. Para a resolução deste problema, o sistema deve minimizar o consumo de energia, que depende linearmente do *duty-cycle* imposto em cada secretária, pelo que a função de custo a minimizar é apenas um termo que penaliza o consumo de energia. Assim, pode-se escrever o problema de optimização como:

$$\begin{aligned} \underset{d \in R^n}{\text{minimize}} \quad & c^T d \\ \text{subject to} \quad & Kd + o \geq L \\ & 0 \leq d_i \leq 100, \;\; i = 1, ..., N \end{aligned} \tag{7}$$

Em que $d \in \mathrm{R}^n$ é o vetor de *duty-cycle* aplicado a cada secretária, $c \in \mathrm{R}^n$ é o vetor de custo energético, $L \in \mathrm{R}^n$ é o limite inferior de luminosidade para cada secretária e $o \in \mathrm{R}^n$ a sua perturbação externa.

No que toca às restrições, a primeira indica que a iluminação total presente na secretária i (que contém a influência das outras secretárias, bem como a sua perturbação externa) tem de ser igual ou superior ao seu limite de luminosidade inferior, L_i. A segunda deve-se ao facto de o *duty-cycle* só puder variar entre 0 e 100 % . Através das restrições chega-se a uma *feasible set*, que é a região onde a solução do problema tem de estar. A titulo de exemplo, à semelhança do slide 14 de [4], com número de nós igual a 2 e k=$\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$ chega-se à *feasible set* representada em 7.

Na figura 7 a reta verde é definida por $d_2 = \frac{L_1 - o_1 - K_{11} d_1}{k_{12}}$ e a azul é definida por $d_2 = \frac{L_2 - o_2 - K_{21} d_1}{K_{22}}$

Para a resolução do problema aplica-se o algoritmo **consensus** que, como especificado em [5], é um método iterativo que distribui o cálculo pelos nós da rede. O algoritmo **consensus** usa o método **ADDM**(*alternated direction method of multipliers*). Cada nó tem uma cópia local $d_i \in R^n$ da variável global $d \in R^n$. Chama-se variável global, pois em cada iteração garante-se que todos os nós têm conhecimento do seu valor, que é igual para todos. Esta variável representa a média de todos os d_i. Conhecendo a variável global cada um consegue calcular a sua variável de controlo/optimizar o seu d_i mediante as restrições. Assim cada nó precisa de ter conhecimento dos parâmetros locais $c_i, \;\; L_i, \;\; o_i \in R$ e só precisa de armazena um vetor de ganhos, e não uma matriz, pelo que $k \in R^n$. Conclui-se portanto que as restrições locais do nó i são dadas por:

$$C_i : \{0 \leq d_{ii} \leq 100 \;\; e \;\; L_i - o_i \leq K_i^T d_i\} \tag{8}$$

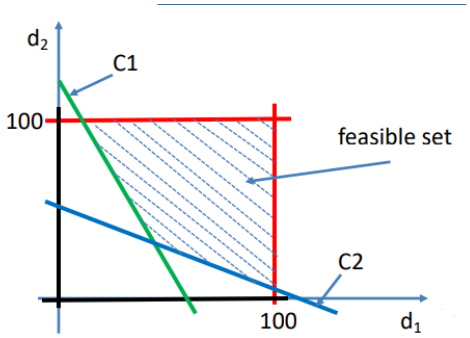


Figura 7: *Feasible set* para o caso N=2 - Retirado do slide 14 de [4]

O método **ADMM** usa um método Lagrangiano aumentado para o cálculo das soluções locais, d_i e dos multiplicadores de Lagrange, y_i. Em cada iteração t de cada nó i são feitas as operações representadas em 9, onde p representa o parâmetro de penalização do método de Lagrangiano aumentado.

$$\begin{cases} d_i(t+1) = \underset{d_i \in C_i}{argmin}\{c_i^T d_i + y_i^T(t)(d_i - \bar{d}_i(t)) + \frac{\rho}{2}\left\|d_i - \bar{d}_i(t)\right\|_2^2\} \\ \bar{d}_i(t+1) = \frac{1}{N}\sum_{j=1}^{N} d_j(t+1) \\ y_i(t+1) = y_i(t) + \rho(d_i(t+1) - \bar{d}_i(t+1)) \end{cases} \tag{9}$$

Resolvendo o problema, como detalhado em [5], obtém-se:

$$d_i(t+1) = \underset{d_i \in C_i}{argmin}\{\frac{1}{2}\rho d_i^T d_i - d_i^T z_i(t)\}$$

onde a função de custo é

$$f^t(d_i) = \frac{1}{2}\rho d_i^T d_i - d_i^T z_i(t) \quad \texttt{com} \quad z_i(t) = \rho\bar{d}_i(t) - c_i - y_i(t)$$

Uma vez que se tem um problema de optimização com uma função quadrática e convexa, a solução ou está no interior da região *feasible* ou nas suas fronteiras. No programa realizado são analisados todos os casos possíveis e, no fim, é considerado aquele que apresenta a melhor solução.

No final de cada iteração cada *arduino* envia o seu d_i para todos os outros em *broadcast* e calcula a média de todos os d_i (d). Espera pela confirmação de que todos receberam a sua solução e avança para a próxima iteração. Isto repete-se enquanto não se verifica a condição de paragem. Esta condição é dada por: a norma da diferença do vetor de soluções com o vetor de médias é nula, ou o número de iterações igual a 50. O valor da norma igual a zero deve-se ao facto de se enviar sempre apenas a parte inteira da solução. Com o acréscimo do cálculo da norma à condição de paragem verificou-se uma melhoria significativa no tempo de execução do algoritmo.

A solução do **consensus** permite obter o novo valor a utilizar no **feedforward**, assim como o novo valor de referência a utilizar no sistema de controlo, que é $L_i = \sum_{j=1}^{N} K_{ij} d_{ij} + o_i$.

Refere-se que, antes da execução do algoritmo, calcula-se os limites máximo e mínimo da solução (para o máximo do nó i é $L_{max} = \sum_{j=1}^{N} K_{ij} 100 + o_i$). Se a luminosidade pedida não estiver neste intervalo, o problema de minimização não tem solução. Assim sendo, se for pedido um valor superior, satura-se para o valor máximo. O mesmo acontece para o caso do mínimo.

6 Comunicações I^2C

O protocolo síncrono utilizado para a comunicação entre arduinos, bem como na comunicação com o *Raspberry Pi*, foi o I^2C. Com este protocolo usam-se dois fios: *Clock line* (SCL) e *Data line* (SDA), e é possível a utilização de múltiplos *masters* e *slaves*. Utilizou-se o modo de funcionamento *Master Transmitter - Slave Receiver* (MT/SR), onde o *master* envia dados para o *slave*. Embora não utilizado, também existe o modo de funcionamento em que o *master* pede dados ao *slave* (MR/ST). A implementação do protocolo foi realizada com a biblioteca *WSWire*.

6.1 Protocolo utilizado entre *arduinos*

Cada *arduino* precisa de ter um endereço de 7 bits para se usar durante a comunicação. Assim, o endereço associado a cada *arduino* foi escrito na `EEPROM`. É utilizado o endereço *broadcast* (0x00) quando se pretende enviar um pacote para todos os nós da rede (excepto o *master*) - na recalibração por exemplo - e é utilizado o endereço do único destinatário quando se pretende enviar para um único nó (por exemplo quando se envia um **ack**). Realça-se que sempre que é enviado um pacote, se a *source* não receber um número de **ack**s igual ao número de destinatários do pacote, volta-se a reiniciar o sistema, de modo a perceber se algum nó saiu da rede ou não (pois a não resposta pode dever-se a uma saída de um nó da rede). Este mecanismo de retransmissão permite a prevenção de perda de mensagens por congestionamento do *bus*.

No protocolo de comunicação estabelecido entre *arduinos* são enviados três tipos de pacotes: pacote **ação simples**, pacote **consensus** e pacote **ack**.

O primeiro é usado para ações simples (por exemplo desligar o LED). Neste pacote apenas é enviado o endereço da *source* - 1 byte - e uma *label*, de também um byte, que especifica a ação a fazer.

No segundo tipo de pacote, que é usado durante o algoritmo *consensus* é enviado o endereço da *source*, a ação e o vetor com o resultado do algoritmo em cada iteração.

O último tipo de pacote é usado para respostas de ações que foram realizadas a mando de algum nó(que é o destinatário deste pacote). Envia-se apenas uma label para o destinatário - 1 byte.

De modo a especificar melhor cada um dos três tipos de pacotes, enumera-se todos os tipos de mensagens utilizadas:

- ***hello***: *label* "h"- Enviada por um nó para o endereço *broadcast* quando entra na rede/é ligado;
- **calibração**, *label* "x" - Enviada do *arduino* com o maior endereço da rede para o *arduino* com o endereço mais baixo, para este começar a calibração;
- **desligar o LED**, *label* "v" - Enviada por um *arduino* em *broadcast* para todos desligarem o LED;
- **Ler perturbação externa**, *label* "l" - Enviada em *broadcast* pelo *arduino* com o índice mais baixo da rede (quando sabe que todos os nós têm o seu LED desligado), para que todos leiam a sua perturbação externa;
- **Calcular o ganho K**, *label* "m"- Enviada pelo *arduino* que se está a calibrar no momento, em *broadcast*, para todos calcularem a sua influência;
- **recalibração**, *label* "s" - Enviada de um *arduino* (o que acaba de calcular o seu *array* com os ganhos), para o *arduino* com o índice acima do seu, para este se recalibrar;
- **consensus**, *label* "c" - Enviada em *broadcast* por um nó quando acaba de obter uma solução de uma iteração do algoritmo de optimização distribuído. Neste pacote é enviado o valor da sua solução;
- **Refazer o consensus**, *label* "d" - Enviada em *broadcast* por um nó quando este sofre uma mudança do estado de ocupação do sistema. Os *arduinos*, ao receberem esta mensagem, começam o algoritmo de optimização distribuída;
- **ack**, *label* "k" - Enviada por um *arduino* quando acaba de executar uma ação ordenada por parte de outro *arduino* (que é agora o destinatário desta mensagem).
- **ack** de um **hello** , *label* "a" - Enviada por um nó em resposta a um **hello** para dizer que está na rede. Este **ack** difere do anterior por se enviar o seu endereço (necessário para saber que índices existem na rede), ao passo que no caso de cima não é necessário.

6.2 Protocolo entre *arduinos* e *Raspberry Pi*

Antes de mais, faz sentido referir que, como especificado em [6], pelo facto de o *Raspberry Pi* funcionar a 3.3V e o *arduino* a 5V, a maneira mais segura para os conectar no *bus* do I^2C é através de um *I2C bi-directional level shifter* como ilustrado na figura 8.

No protocolo definido entre os *arduinos* e o servidor o valor do primeiro *byte* enviado define o tipo de mensagem. Assim, apresentam-se de seguida os possíveis valores do primeiro *byte* enviado e o respetivo pacote:

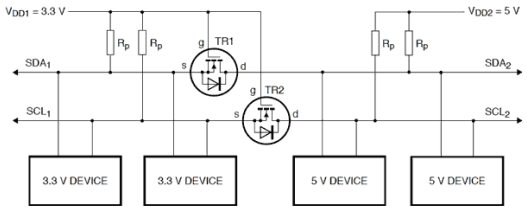


Figura 8: Circuito utilizado para a comunicação I^2C entre *arduinos* e *Raspberry Pi* - Retirada de [6]

- **stream**, primeiro *byte* **0-126**: Se o primeiro *byte* estiver nesta gama de valores, o servidor sabe que é uma mensagem periódica, e o primeiro *byte* representa também o endereço (menos uma unidade) do *arduino* que enviou a mensagem. O segundo *byte* enviado representa a parte inteira do que se está a ler (em LUX), seguido do terceiro byte que é a parte fracionária. Assim, se a saída do controlador do *arduino* número 2 for 20.96 é enviado primeiro um 1(índice -1), depois o valor 20 e depois o 96. O último byte enviado representa a saída do controlador em PWM.
- **calibração**, primeiro byte **127-253**: Se o primeiro *byte* estiver neste intervalo, a mensagem é referente a uma calibração, onde o primeiro byte representa também o endereço do *arduino* que enviou a mensagem (+ 126). Os bytes enviados seguidamente representam a parte inteira e fracionária do *Lower Bound* para o estado ocupado e desocupado, bem como a iluminação externa. Mais uma vez, separam-se a parte inteira e fracionária e enviam-se separadamente em 2 *bytes*. Desta forma evita-se enviar 4 bytes por cada float, enviando-se apenas 2.
- **Mudança para estado desocupado**, primeiro byte **254**: O segundo *byte* indica o endereço do *arduino*, e os dois últimos o valor de iluminação desejado (parte inteira e fracionária).
- **Mudança para estado ocupado**, primeiro byte **255**: Igual ao caso anterior.

Para facilidade de leitura e compreensão deste protocolo, resumiram-se, pela ordem descrita em cima, as possibilidades de comnuicação na Tabela 2.

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
(0-126) arduino	lux_{int}	lux_{frac}	PWM	—	—	—
(127-253) arduino	$offBound_{int}$	$offBound_{frac}$	$onBound_{int}$	$onBound_{frac}$	exterior_{int}	exterior_{frac}
(254-255) estado	arduino	referência_{int}	referência_{frac}	—	—	—

Tabela 2: Possibilidades de comunicação entre *arduinos* e *Raspberry Pi*

Faz também sentido referir que todas as escolhas feitas para este protocolo se focaram na minimização do número de *bytes* transferidos para o *buffer* de comunicação, tendo em conta que a comunicação, de longe, mais efetuada será a do período de amostragem, na qual se conseguem enviar apenas 4 *bytes*.

6.3 Latência

A frequência da linha SCL utilizada foi 100KHz, que é a frequência *default* do protocolo. Como se pode observar na figura 9, retirada de [7], cada pacote é inicializado com um bit da condição de *START* e terminado com um bit devido à condição de *STOP*. Além disto, cada pacote contém também 7 bits com o endereço de destino, um bit para dizer se se pretende escrever ou ler do *slave*, seguido de um **ack**, e um bit com um **ack** por cada byte enviado.

- **consensus**: Como estabelecido no protocolo de comunicação entre *arduinos*, no final de cada iteração do algoritmo, é enviado um *byte* com a ação, um *byte* com a *source* e, posteriormente, um *byte* por cada posição do vetor de solução do consensus. Ou seja, se existirem N nós na rede, a soma de *bytes* de *data* enviados são 2+N bytes. Assim o número de bits total do pacote de envio será $(2+N)\cdot 8$+(2+N)+1+7+1+1+1 = 29+9N bits. No entanto tem também de se considerar as

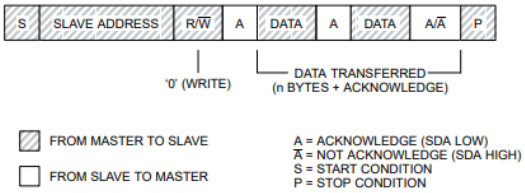


Figura 9: Pacote de I^2- Retirado da figura 12 da página 10 de [7]

mensagens de **ack** que o nó que envia a solução espera como resposta. O pacote de **ack** tem uma *data* de 1 *byte*. Assim cada nó, ao enviar um **ack**, envia um pacote de 8+1+1+7+1+1+1 = 20 bits. Como vão ser N-1 nós a responder com este pacote, juntando os dois pacotes, tem-se num total de 29+9N+20·(N-1)= 9+29N bits. Para o caso testado, N=2, tem-se número total de 67 bits por cada iteração, o que resulta num tempo de 0.67ms por iteração. Se forem realizadas 12 iterações (o número médio verificado de iterações até à convergência) resulta num tempo de 8.04 ms.

- **stream**: No que toca às mensagens periódicas enviadas para o servidor, são enviados em cada período de amostragem 4 *bytes* de *data*, por cada *arduino*, e não são recebidos **acks**, pelo que o número total enviado por um *arduino* são 4 · 8+4+1+7+1+1+1= 47 bits. Para N *arduinos* resulta num total de 47N bits enviados. Para o caso testado, N=2, tem-se um total de 94 bits, pelo que o tempo é 0.94 ms.

7 Arquitetura dos Arduinos

Cada nó pode estar num dos 3 estados principais: `Calibração`, `Controlador` ou `Consensus`.

Dentro de estes estados, existem subestados(ligar o LED ou desligar, por exemplo), em que cada um destes subestados (determinado pela *flag* `deskStatus`) funciona de forma independente. No `Loop` do programa principal é verifica o subestado atual e consoante o mesmo executa as respetivas ações.

Em cada nó, receber mensagens é uma operação assíncrona. A receção de uma mensagem é detetada através do *handler* `receiveEvent`, que de acordo com a mensagem recebida atualiza o estado(e, por sua vez, o subestado) do respetivo nó.

No programa principal, um rotina de interrupção é ativada a cada 10 ms, onde as operações do `Controlador` são efetuadas. Nesta rotina, é feita a leitura do LDR, os cálculos do controlador são executados e é atualizado o valor do PWM do LED. As restantes tarefas são efetuadas já fora da interrupção, nomeadamente: o *display* no *Serial Monitor* das variáveis do sistema, o estado de ocupação, a respetiva iluminação pretendida e a leitura da iluminação atual, o envio da informação para o *Raspberry Pi* e a atualização das variáveis do controlador.

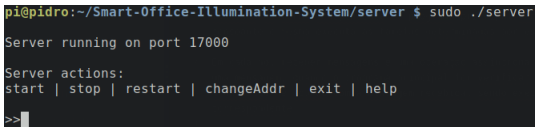
Todas as operações de comunicação I^2C estão organizadas na classe `I2COMMUN`, em `i2commun.cpp` e `i2commun.h`. Todas as funções associadas ao sistema de controlo organizam-se numa classe(`CONTROLLER`), em `controller.h` e `controller.cpp`. As funções do **consensus** encontram-se em `node.cpp` e `node.h`. Neste algoritmo é necessário muitas operações algébricas sobre vetores, pelo que se define à parte uma classe (`Vector`) em `Vector.h` que contém os métodos que permitem fazer toda a manipulação sobre vetores.

8 Servidor

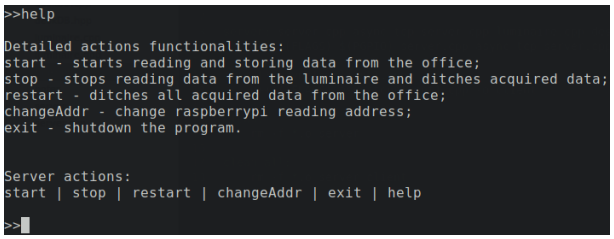
Para este projeto foi desenvolvido um servidor em C++ com o objetivo de recolher dados estatísticos de todo o sistema de luminárias (cada luminária individualmente), através do protocolo de comunicação I^2C. O servidor aceita, paralelamente à recolha de dados, ligações TCP de clientes remotos, reportando para estes, quando pedidos, os dados adquiridos. Este servidor correu numa *raspberrypi*, em tempo real, ligado ao sistema de luminárias.

Para facilitar a utilização, foi escrito um ficheiro `makefile` para compilar o código do servidor, e também com a possibilidade de compilar o cliente "exemplo"da página da cadeira (há também a opção de apagar os executáveis).

Para correr/lançar o servidor basta executar, com permissões de *superuser*, o executável `servidor`, gerado pelo `makefile`, seguido do porto por onde se pretende receber as ligações dos clientes TCP. Não se especificando nenhum porto, o servidor TCP fica à escuta no porto 17000, definido como o porto *default*. Pode ver-se um exemplo desta execução na Figura 10 (a).



(a) Execução do programa do servidor



(b) Ações dispníveis na consola que correu o servidor

Figura 10: Consola do servidor

Também na Figura 10 (a), pode ver-se que na consola que corre o servidor, há algumas ações disponíveis que interagem com a aquisição de dados do sistema de luminárias. É escusado explicar a utilidade destas ações, pois, escrevendo help, obtemos essa informação, como se pode observar na Figura 10 (b).
NOTA: Assim que o servidor é iniciado, este começa a aquisição e armazenamento de dados do sistema de luminárias, pelo que a ação "start"já vem ativada de origem.

8.1 Arquitetura do Servidor

Todo o desenvolvimento do servidor foi efetuado no sentido de uma arquitetura em camadas. Podemos dividir o programa em classes (neste caso as camadas), absolutamente independentes e bem definidas. A comunicação entre camadas oferece um elevado nível de abstração, pois cada uma delas não conhece o funcionamento interno das outras, apenas a API que oferecem. Poder-se-ia assim substituir qualquer uma das camadas, por outra diferente, desde que a nova conhecesse a API das camadas com as quais teria de interagir. Desta forma o projeto pode fácilmente ser esacalado para sistemas de maior dimensão, ou, podem também reaproveitar-se partes independentes do programa que possam "encaixar"noutros tipos de propósitos. Uma definição objetiva das diferentes camadas está presente na secção "Classes Concebidas".

8.1.1 Classes Concebidas

Foram desenvolvidas as seguintes classes em C++:

- **`deskDB`** (deskDB.hpp) – classe associada a uma só secretária, responsável por guardar todos os dados relativos à mesma, garantindo fazê-lo de forma segura e eficiente mesmo quando acedida por vários *processos/threads* em simultâneo.
- **`luminaire`** (luminaire.hpp) – classe associada ao sistema físico de luminárias como um todo, responsável por: ler os dados vindos diretamente deste; instanciar as bases de dados (`deskDB`) necessárias, isto é, o número de luminárias do sistema físico; e manuseá-las (aceder aos seus métodos para inserções, remoções ou consultas).
- **`session`** (async_tcp_server.hpp) – classe associada a um cliente TCP, responsável por: através do *endpoint* respetivo, interpretar os seus pedidos; obter uma resposta (acedendo para isso a um objeto do tipo `luminaire`, instânciado já previamente pelo programa principal); e responder ao cliente.

- **`server`** (async_tcp_server.hpp) – esta classe representa o servidor TCP, responsável pela escuta num porto definido por novos clientes para este protocolo. Sempre que aceita uma nova ligação, cria um *socket*, que será o *endpoint* para todas as comunicações com o cliente sujeito, e abre uma sessão (`session`) para fazer o tratamento dos seus pedidos. Logo que a sessão é aberta, retorna a escuta no porto por novas ligações.

8.1.2 Gestão de Threads

Este programa dispõe também de uma arquitetura *Multithreading*. No decorrer operacional do programa, estão continuamente ativas 3 threads, criadas com os seguintes nomes no código "main"(`server.cpp`):

- `luminaireThread` – Uma vez previamente instanciado um objeto do tipo `luminaire`, esta *thread* evoca o método deste que adquire dados do sistema físico de luminárias e os guarda na base de dados que lhes diz respeito.
- `consoleThread` – *Thread* que espera por ordens vindas diretamente do terminal que correu o programa (como já foi anteriormente expresso na Figura 10)
- **"main thread"** – *Thread* que aceita continuamente novos clientes TCP e trata, assíncronamente, dos seus pedidos(este assincronismo será explicado mais à frente neste capítulo).

8.2 Funcionamento do servidor

Uma vez que já ficaram explícitas não só as classes utilizadas e os seus propósitos, como também as *threads* evocadas, é mais fácil agora de perceber a orientação do funcionamento do programa e o fluxo de dados que nele decorre. Na Figura 11 está ilustrado todo este processo.

Contornados a tracejado, estão representados os endpoints externos ao núcleo do programa(clientes TCP e sistema físico). A cor verde é referente à `luminaireThread`, a veremelha à **main thread**, e a azul, simplesmente a `consoleThread`, onde apesar de não haver fluxo de dados, há a possibilidade de interagir com a `luminaireThread` ou de fechar, seguramente, todas as threads (término do programa). As setas sem enchimento representam o fluxo de dados, sendo que as representadas a tracejado pretendem significar o direcionamento/processamento dos mesmos pela entidade onde se encontram na figura. Olhando em mais detalhe para os dois fluxos de dados:

- **verde** – os dados do sistema físico estão contínuamente a ser lidos por uma instâcnia da classe `luminaire`, dentro desta são processados e enviados para a base de dados correta.
- **vermelho** – todos os pedidos efeutados pelos clientes, implicam uma interpretação do pedido pela sessão respetiva, que por sua vez pede à instânciada `luminaire` o valor pretendido e de que luminária, a `luminaire` pede o valor à base de dados respetiva, reevnia-o para a sessão e esta finalmente, responde com este valor ao cliente.

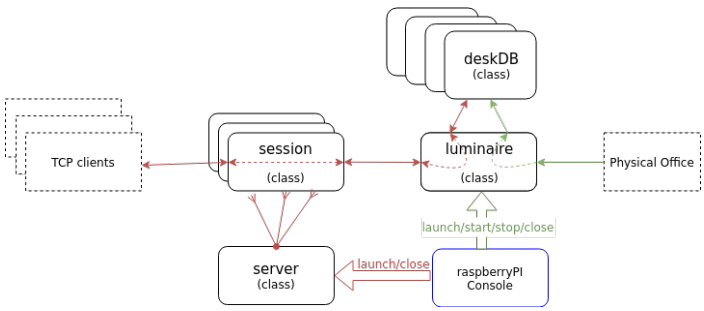


Figura 11: Fluxo de dados nas *threads* utilizadas

8.3 Tecnologias utilizadas e detalhes do programa

Para finalizar a análise do servidor, vão ser expostos neste capítulo as principais tecnologias utilizadas, bem como algumas particularidades implementadas no programa:

- **Programação assíncrona** – O tratamento de clientes TCP foi feito com base na programação assíncrona disponibilizada pela bilbioteca Boost.Asio do C++. Associando todas as operções assíncronas a um serviço I/O, consegue-se, apenas com uma thread, efetutar o tratamento de pedidos de vários clientes em "simultâneo", apenas recorrendo a um *core* do processador. É relevante referir, que, exceptuando os pedidos de *live streams*, até 50 clientes com pedidos recorrentes, não se notou, a olho nu, nenhuma quebra nos tempos de resposta. A partir dos 100 clientes já conseguia notar-se alguma diferença, ainda que pouca e a partir dos 400 clientes a demora já era claramente notória, embora que o funcionamento decorresse corretamente. Por volta dos 900 clientes, o *socket* vinculado (*binded*) pelo servidor deixa de aceitar novas ligações, no entanto, continua a servir os clientes já ligados de forma correta.

- **PGPIO** – através da biblioteca "pgpio.h"do C++, conseguiu-se, pelo protocolo com que os arduinos já comunicavam entre si (i2C), interpretar as mensagens vindas dos arduinos desde que o *buffer* de leitura não excedesse o seu limite (16 bytes). Para tentar impedir ao máximo este acontecimento, aplicou-se o protocolo explicitado no capítulo "Protocolo entre arduinos e Raspberry Pi".

- **Regiões críticas** – Para impedir múltiplos acesso à mesma região de memória (na base de dados) por *threads* diferentes em simultâneo, recorreu-se a variáveis de exclusão mútua (*mutexes*). Posto isto, uma vez que atrasos na escrita podem implicar perdas de dados, ao invés da leitura que apenas atrasa a recessão dos mesmos, implementou-se um sistema de 3 *mutexes* na base de dados (`deskDB`), que não só apenas permite um acesso de cada vez à base de dados, como garante que no pior dos casos, a escrita apenas espera por 1 leitor. Para facilitar a leitura das regiões críticas estas estão identadas com um *tab*. Nesta implementação minimizou-se ao máximo o tempo de computação dos troços efetuados dentro das regiões críticas.

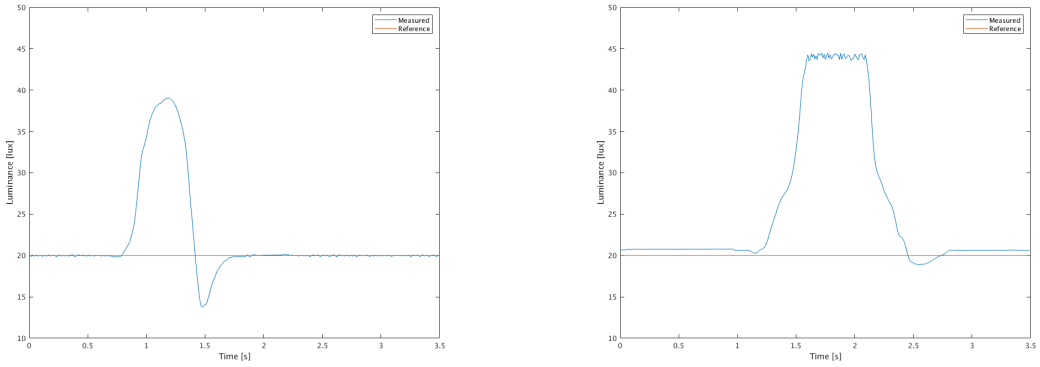
- **Stream de dados** – Para efetuar a stream de dados para o(s) cliente(s) foram implementadas variáveis de condição. Optou-se por este método, não só para que cada rotina à espera de uma nova *sample* não tivesse que fazer uma espera ativa pela mesma (consumindo abusivamente o *core*), como também desta forma se conseguem notificar todos os clientes interessados com um *unix signal*. Com as variáveis de condição perde-se também a necessidade de lançar uma nova *thread* para gerar a *stream* de dados. Esta espera tem também um *timeout* associado, para que se deixarem de chegar *samples*, se continue a enviar os dados para o cliente, podendo este ver que se mantêm inalterados. Finalmentem optou-se por não utilizar o típico ciclo `while` de proteção para prevenir eventuais "spurious wake ups"na espera bloqueante da variável de condição, pois enviar um valor a mais para o cliente durante uma *stream* de dados não provoca qualquer tipo de problema, conseguindo-se ganhar assim uma maior simplicidade de código.

- **Término seguro do programa** – De forma a poder fechar o programa de forma segura, liberta-se uma ação de cada vez da *queue* de serviços, com a função `poll_one()`, e termina-se o serviço `PGPIO` iniciado no arranque do programa, que quando não fechado devidamente, pode causar problemas. Fez-se também um *handler* para o *unix signal* gerado pelo `Ctrl+C`, para que este feche o programa da mesma forma. Toda a memória dinâmica, foi alocada por `shared_points`, pelo que não foram precisas preocupações a libertar memória no decorrer nem no término do programa.

9 Experiências/Discussão

9.1 Caraterização do feedforward e feedback

De modo a caraterizar o **feedback**, introduziu-se uma perturbação no sistema (abriu-se a caixa), observando a resposta para os casos em que este está presente no sistema e não está. A figura 12 ilustra estra este caso.

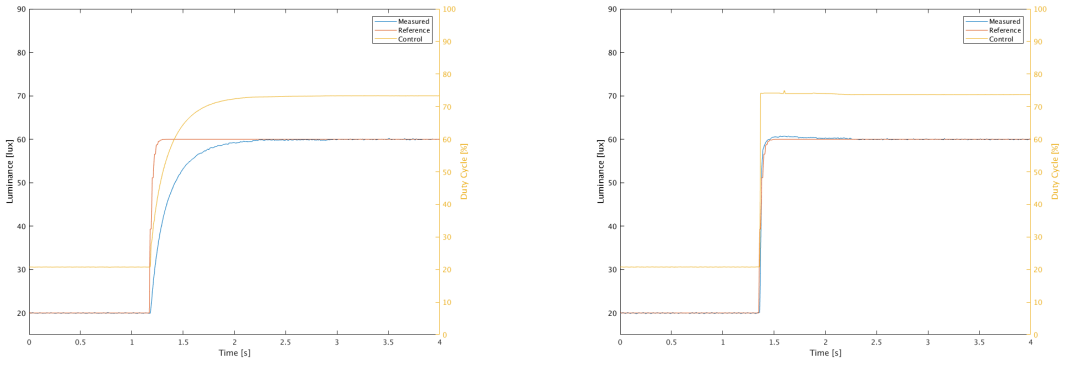


(a) Com **feedback** e **feedforward** (b) Sem **feedback**

Figura 12: Experiência em que se coloca uma perturbação no sistema

Sem a presença do **feedback** o sistema nem sequer converge para o valor de referência, estando sempre acima. Ao haver uma perturbação, sem a presença de **feedback**, o sistema não consegue reagir.

A figura 13 representa as medições efetuadas quando se mudou a referência para os casos em que existe e não existe **feedforward**.



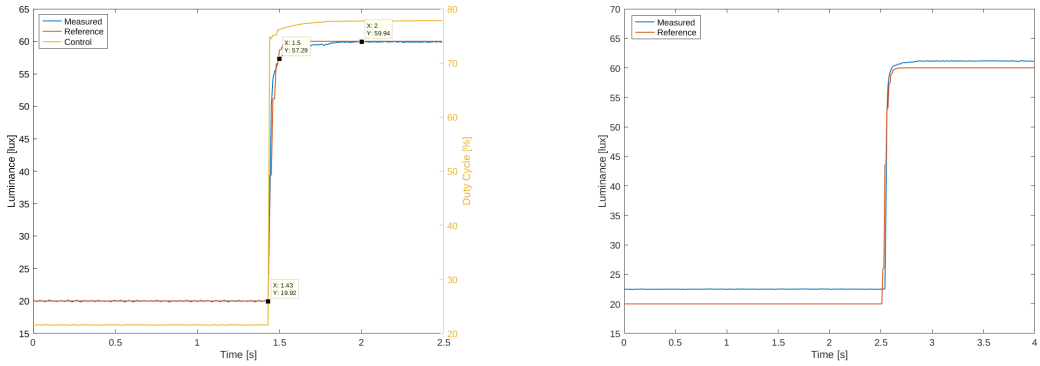
(a) Sem **feedforward** (b) Com **feedforward**

Figura 13: Experiência com e sem **feedforward**, e ambas com **feedback**

Como esperado, para o caso em que não existe **feedforward** o sistema demora mais tempo a convergir para a referência. Apesar de quando não existir **feedforward** se atingir a referência, o tempo de subida é superior.

Para caraterizar o erro no **feedforward**, calculou-se o valor médio quadrado entre o sinal de referência e o medido, para o caso da figura 14(b), em que se fez a experiência só com o **feedforward**. O valor obtido foi de 4.7851 LUX. Este erro é depois compensado com o **feedback**. Através da figura 14(a) observa-se que o fator de amortecimento assim como a sobreelevação são aproximadamente nulos. O tempo de subida obtido foi de 70 ms.

Uma forma de caraterizar o **jitter** observado, foi verificando o valor do período de amostragem, que não se mantém sempre em 10 ms. Apesar de se ter uma interrupção *timer*, o período de amostragem varia aproximadamente entre 9.5 ms e 10.2 ms. Em cada periodo de amostragem, as ações do controlador demoram aproximadamente 0.4 ms.

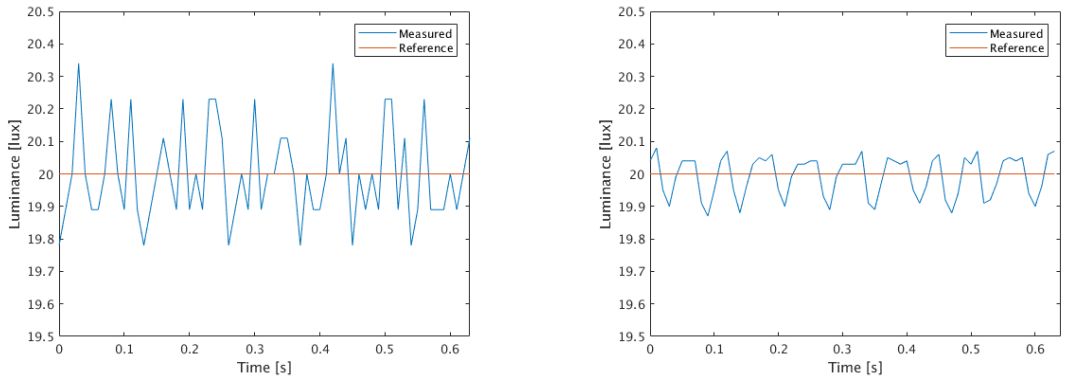


(a) Resposta com **feedback** (b) Apenas **feedforward**

Figura 14: Caraterização dos parâmetros do **feedback** e **feedforward**

9.2 Melhorias no controlo feedback

Com a implementação do filtro Passa-baixo, conseguiu-se uma diminuição na na amplitude de oscilações dos valores de saída em torno do valor de referência, como se pode observar na figura 15.

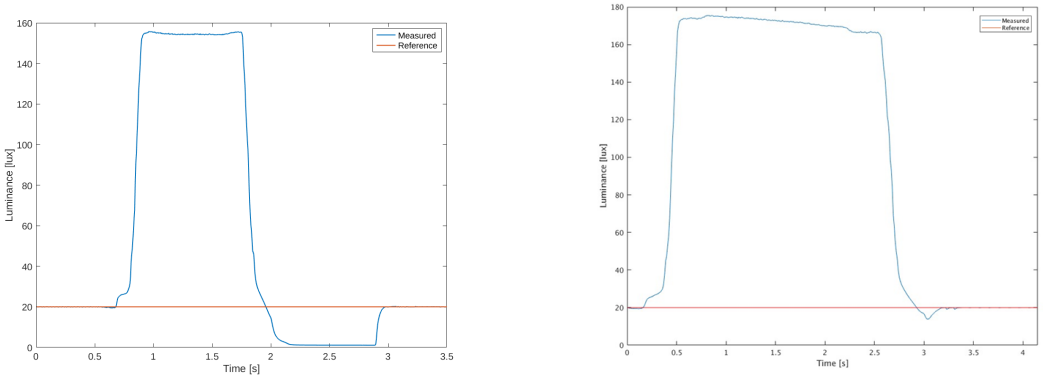


(a) Sem filtro (b) Com filtro

Figura 15: Experiências realizadas com e sem filtro passa-baixo

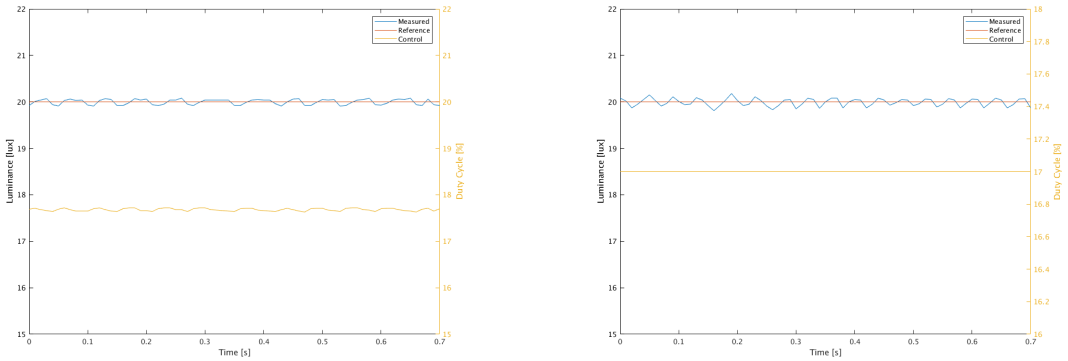
A figura 16 ilustra as experiências para o caso em que se tem o **Anti-windup** ativado e desativado. Nestas experiências, através de uma lanterna, introduziu-se maior luminosidade do que a que o sistema tem de referência. Desta forma, sem o **Anti-windup**, o LED desliga mas o termo integral vai acumular erros negativos. Assim que se deixa de incidir a lanterna sobre o sistema, este demora bastante tempo a convergir para a referência pois está a descarregar o erro acumulado. Na figura 16(b) nota-se uma grande melhoria. O sistema converge muito mais rápido e a saída não desce tanto a baixo da referência.

Com a introdução da **deadzone** consegue-se obter um sinal de controlo com menos oscilações, ou seja, menos **flickering**, como se observa na figura 17.



(a) Sem **Anti-windup** (b) Com **Anti-windup**

Figura 16: Experiência com e sem **Anti-windup**



(a) Sem **Deadzone** (b) Com **Deadzone**

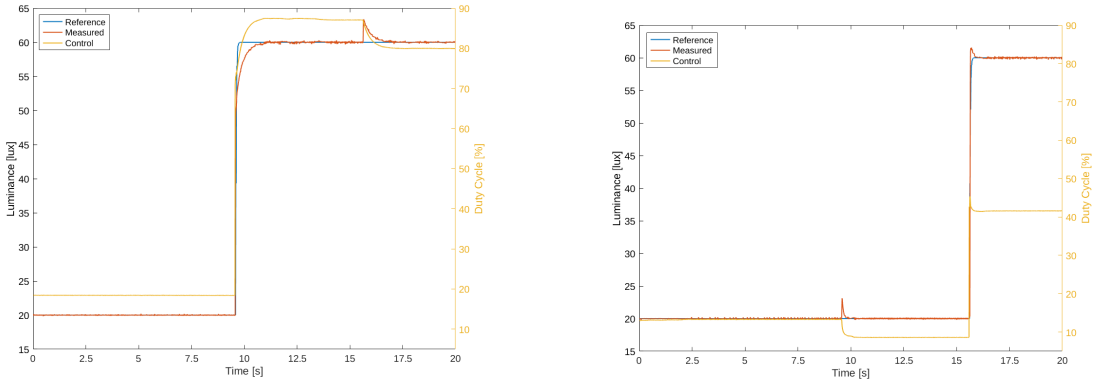
Figura 17: Experiência com e sem **Deadzone**

9.3 Controlo distribuído

Realizou-se a experiência de mudar a referência do sistema para cada um dos nós em instantes de tempo diferentes, observando-se o comportamento de ambos para os casos em que existe e não existe o controlo distribuído. Para ambos os casos o nó 1 altera a sua referência em aproximadamente $t = 10$ s e o nó 2 altera a sua referência em aproximadamente $t = 15$ s. Observando a figura 18 (experiência sem controlo distribuído) é notório a dificuldade em acompanhar a referência sempre que alguma mudança é feita. Neste caso, quando um dos nós altera a sua referência para o estado ocupado, o outro nó não tem conhecimento da existência dos outros nós e a influência dos mesmos, pelo que irá tratar esta alteração como uma perturbação externa. Esta é a razão para a sobreelevação apresentada em aproximadamente $t = 10$s no nó 2 quando o primeiro muda a referência, e também no nó 1 quando o 2 muda a referência.

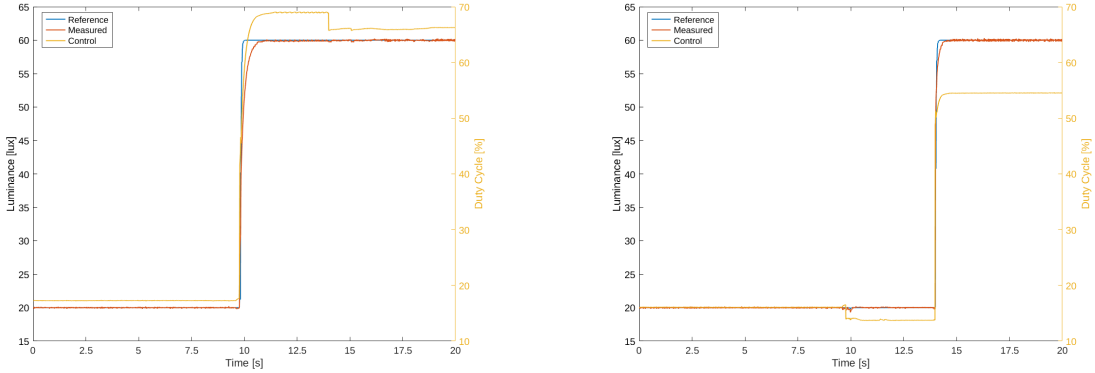
Constatando a figura 19 é visível os benefícios que o controlo distribuído trazem ao sistema. Quando existe uma mudança de referência por parte de um nó, praticamente não é notada pelo sistema, já que ao aplicar-se o **consensus**, ambos os nós ajustam o valor do sinal de controlo para satisfazer ambas as referências. Em $t = 15$ s o nó 2 alterou a sua referência, no entanto o sinal medido por parte do nó 1 não se alterou, alterando-se apenas o seu sinal de controlo.

Por último, através do servidor, analisaram-se as métricas para os casos em que há e não há controlo distribuído. Estas experiências estão ilustradas na figura 20.



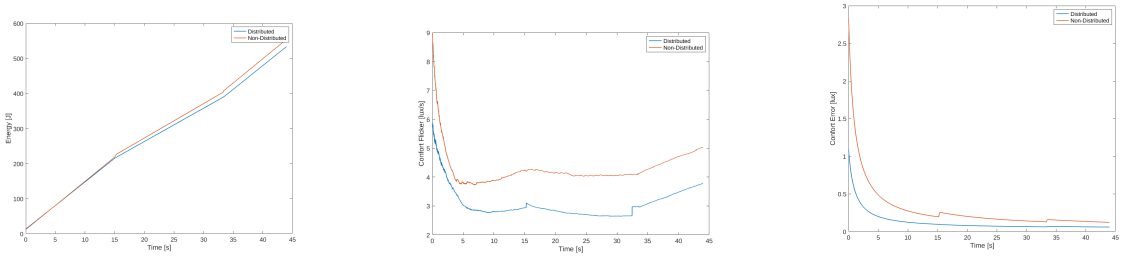
(a) Nó 1 (b) Nó 2

Figura 18: Experiência sem controlo distribuído



(a) Nó 1 (b) Nó 2

Figura 19: Experiência com controlo distribuído



(a) Energia Acumulada (b) Oscilação de conforto (c) Erro de conforto

Figura 20: Gráficos realizados através da recolha de valores do *Raspberry Pi* com e sem controlo distribuído

O declive da energia é um pouco mais acentuado para o caso em que não existe controlo distribuído, pelo que gasta mais energia por instante de tempo. Conclui-se que com o controlo distribuído obtém-se uma melhor performance: menor gasto de energia por instante de tempo, menos oscilações em torno da referência e menor erro de conforto.

10 Conclusão

No controlo do sistema, para que este reagisse melhor a perturbações externas e mudanças de ocupação da secretária, foi fulcral a adição de alguns componentes além **feedback**(por exemplo **Anti-windup**, **feedforward**, filtro passa-baixo). Com a presença destes componentes, notou-se bastantes melhorias úteis para o uso prático do utilizador.

De um ponto de vista superficial, pôde-se observar que face ao desperdício de energia comum, exposto no início deste trabalho, se conseguiu mostrar que é possível corrigir este infortúnio a baixos custos.

Fazendo uma análise mais pormenorizada, após olhar para os testes realizados, foi fácil de entender os objetivos práticos de cada uma técnicas aplicadas no controlodar e observar o impacto monumental que a remoção/adição destas causa nos resultados. Foi também possível, através de uma elaborada afinação de software, suceder no desenho de um sistema escalável, sempre funcional e descentralizado, sem que para isso se adicionasse uma grande *overhead* nas comunicações efetuadas.

Finalmente, após uma recolha coerente de dados estatísticos, considerou-se que os resultados obtidos foram bastantes promissores, prevendo-se por isso que aplicando este protótipo a uma situação real se providenciaria, de facto, uma experiência de conforto para os utilizadores, obtendo para este fim um baixo consumo de energia.

O vídeo do projeto pode ser visto aqui .

10.1 Contribuições Relatório

Apesar de cada aluno ter áreas onde teve uma maior contribuição que outras, todos trabalharam e estão familiarizados com todos os tópicos do projeto.

- **Diogo Rodrigues**: Identificação do Sistema(3), Controlador(5.1),Arquitetura dos Arduinos(7), Experiências(9) e Vídeo(9)
- **Leandro Almeida**: Configuração do sistema(2), Calibração(4), Controlo Distribuído(5.2), Comunicações I^2C(6), Arquitetura dos Arduinos(7), Experiências(9) e Vídeo (9)
- **Pedro Moreira**: Resumo, Introdução(1), Protocolo entre arduinos e Raspberry Pi (6.2), Servidor(8), Experiências(9), Conclusão (10) e vídeo.

Referências

[1] Distributed smart lighting systems: sensing and control
Caicedo Fernandez, D.R,
https://pure.tue.nl/ws/portalfiles/portal/3799294/774336.pdf

[2] https://fenix.tecnico.ulisboa.pt/downloadFile/845043405474933/GL5528.pdf

[3] https://fenix.tecnico.ulisboa.pt/downloadFile/563568428770266/MODULE9-PID.pdf

[4] https://fenix.tecnico.ulisboa.pt/downloadFile/1689468335617052/MODULE17-DISTR-OPTIM.pdf

[5] Alexandre Bernardino, Solution of Distributed Optimization Problems: The consensus algorithm, 2018

[6] https://playground.arduino.cc/Main/I2CBi-directionalLevelShifter

[7] https://www.i2c-bus.org/fileadmin/ftp/i2c_bus_specification_1995.pdf