arduino-meteorolog Documentation

Release 0.1

Nelso G. Jost

September 03, 2015

Contents

Contents:

CHAPTER 1

Visão geral

Este documento procura oferecer uma visão geral sobre o funcionamento do código deste projeto, útil para aqueles que desejam modificá-lo ou simplesmente endendê-lo.

1.1 Funcionamento

Este projeto compreende as duas seguintes ferramentas:

- **Firmware**

  Executado no processador do Arduino, é responsável por ler os sensores conectados de acordo com solicitações enviadas à porta serial. Utiliza bibliotecas de terceiros para leitura de sensores complexos.

- **Logger**

  Executado em uma máquina Linux (PC, Raspberry, etc), é responsável por coletar dados da placa através de uma leitura serial, fazer armazenamento local e também remoto através do envio de dados para o nosso servidor em dados.cta.if.ufrgs.br/emm ou algum outro especificado pelo usuário.

Ambos encontram-se no mesmo repositório pois o logger está preparado para enviar comandos pela serial cujo formato o firmware está preparado para receber. Por exemplo, considere a seguinte string enviada pelo logger à porta serial onde está a placa Arduino:

```
readSensors,LDR,DHT22_TEMP
```

Ao receber esses caracteres, o firmware determinará que trata-se de um comando para leitura de sensores e que os sensores a serem lidos são, nessa ordem: o `LDR` e o `DHT22_TEMP` (luminosidade e temperatura, respectivamente). O firmware retorna pela serial uma resposta com números separados por vírgula, algo como:

```
84,24.5
```

indicando 84 % de luminosidade e 24,5 ºC de temperatura. O logger estará então preparado para receber dois valores, guardá-los em um arquivo de log local (juntamente com a hora do sistema) e também fazer uma tentativa de envio ao servidor. Caso o envio falhe, a leitura será adicionada ao arquivo `outgoing.json` para futuras tentativas de comunicação com o servidor.

Opcionalmente, poderá ser utilizada a hora de um relógio `RTC_DS1307` da placa. Caso este não esteja presente ou não retorne valores consistentes, a hora do sistema é utilizada por padrão.

1.1.1 Exemplos

Segue abaixo o exemplo de um log de execução para uma estação 100% funcional, possuindo os 4 sensores oficialmente suportados (`DHT22_TEMP`, `DHT22_AH`, `BMP085_PRESSURE` e `LDR`) juntamente com o relógio `RTC_DS1307`:

```
2015-09-03 16:12:24 : INFO : ========================================
2015-09-03 16:12:24 : INFO : Serial<id=0x7f1146fc5dd8, open=True>(port='/dev/ttyACM0', baudrate=9600,
2015-09-03 16:12:26 : INFO : sent: 'read,DHT22_TEMP,DHT22_AH,BMP085_PRESSURE,LDR,RTC_DS1307' (55 byte
2015-09-03 16:12:29 : INFO : read: b'22.700001,66.199997,101224,40.762466,2015-9-3 16:12:26\r\n' (56
2015-09-03 16:12:29 : INFO : JSON: {'datetime': {'format': '%Y-%m-%d-%H-%M-%S', 'source': 'RTC_DS130
2015-09-03 16:12:29 : INFO : Updated datalog file at '/home/nelso/lief/arduino-meteorolog/data/datalo
2015-09-03 16:12:29 : INFO : Starting new HTTP connection (1): localhost
2015-09-03 16:12:29 : INFO : Server response: {'success': '1 new points were saved on the board.'}
2015-09-03 16:12:29 : INFO : Going to sleep now for 0.2 minutes
```

A exemplo de como os erros são reportados, segue abaixo o log de execução para uma placa Arduino sem nenhum sensor, com um servidor fora do ar, mas com a mesma configuração `settings.ini` do exemplo anterior:

```
2015-09-03 16:17:10 : INFO : ========================================
2015-09-03 16:17:10 : INFO : Serial<id=0x7f2c89ffb438, open=True>(port='/dev/ttyACM0', baudrate=9600,
2015-09-03 16:17:12 : INFO : sent: 'read,DHT22_TEMP,DHT22_AH,BMP085_PRESSURE,LDR,RTC_DS1307' (55 byte
2015-09-03 16:17:15 : INFO : read: b'<NaN>,0.000000,<bmp085_not_found>,50.537636,2165-165-165 165:165
2015-09-03 16:17:15 : WARNING : SensorReadingError: [DHT22_TEMP]: '<NaN>'
2015-09-03 16:17:15 : WARNING : SensorReadingError: [BMP085_PRESSURE]: '<bmp085_not_found>'
2015-09-03 16:17:15 : WARNING : DateTimeError: [RTC_DS1307]: Expected format '%Y-%m-%d %H:%M:%S' but
2015-09-03 16:17:15 : INFO : JSON: {'sensors': {'DHT22_AH': '0.000000', 'BMP085_PRESSURE': 'NaN', 'DH
2015-09-03 16:17:15 : INFO : Updated datalog file at '/home/nelso/lief/arduino-meteorolog/data/datalo
2015-09-03 16:17:15 : INFO : Starting new HTTP connection (1): localhost
2015-09-03 16:17:15 : ERROR : Request: None. Unable to reach the server at 'http://localhost:5000/api
2015-09-03 16:17:15 : INFO : Updated local file '/home/nelso/lief/arduino-meteorolog/data/outgoing.js
2015-09-03 16:17:15 : INFO : Going to sleep now for 0.2 minutes
```

Apenas sensores que utilizam o protocolo I2C podem ter sua presença detectada de antemão, como é o caso do `BMP085` e do `RTC_DS1307`, retornando um erro como `<bmp085_not_found>`. Repare que embora o **DHT22** não esteja presente na placa, o valor retornado pela leitura de umidade do ar foi `0.000000`, claramente sem significado físico. O mesmo acontece com o **LDR**.

O log dispara **WARNINGS** para as falhas de leitura detectadas. No caso do relógio, o erro indica data inválida e portanto, a hora do sistema será utilizada. Por fim, o log também disparou um **ERROR** na tentativa de conexão com o servidor. A consequência é a criação do arquivo `data/outgoing.json` contendo dados a serem enviados em tentativas posteriores.

1.2 Estrutura de arquivos

Segue uma breve descrição dos arquivos/diretórios presentes na pasta raiz do projeto:

```
arduino-meteorolog/
-- data/                # contém dados gerados pelo logger
-- docs/                # contém essa documentação
-- logger/              # software que faz coleta de dados e envio para o servidor
-- meteorolog/          # projeto ".ino" do firmware (compilável pela Arduino Toolchain)
-- scripts/             # scripts utilizados pelo Makefile
-- settings.ini         # configurações do logger
-- Makefile             # proporciona diversos comandos para facilitar a manutenção
```

1.3 Makefile

Esse arquivo contém diversos comandos simples a serem passados para a ferramenta `make` [1] de modo a facilitar o uso e manutenção dos softwares desse projeto. Basta estar na pasta onde se encontra o `Makefile` e executar:

```
$ make <target>
```

para realizar alguma tarefa. Os *targets* possíveis são listados com `make` ou `make help`:

```
setup       Execute once to prepare the required Python virtual environment
firmware    Compile and upload the firmware to the Arduino board via serial
serial      Starts a serial session with Python for board communication
sync-rtc    Synchronizes the board RTC_DS1307 with this system's time

run         Execute the logger on the foreground. Hit Ctrl+C to stop it.
deploy      Install logger on the Supervisor daemon tool (exec background)
undeploy    Undo the 'deploy' command

tail-log    Follow updates on the last execution log
tail-data   Follow updates on the last data log
plot-data col=x   Uses Gnuplot to plot last data log col number x
```

Na prática o usuário deverá fazer, ao obter uma cópia do repositório:

1. `make setup` para instalar as dependências do Logger em um ambiente virtual de Python
2. `make firmware` para compilar e gravar o firmware na placa Arduino. Alternativamente, isso pode ser feito pela IDE do Arduino.
3. `make serial` para testar a leitura dos sensores com `>>> send('read,...')` e também sincronizar o relógio da placa com o do sistema com `>>> sync_rtc()`, caso possível.
4. `make run` para testar a execução do logger com a configuração atual de `settings.ini`.
5. `make deploy` para instalar o logger no Supervisor (gerenciador de processos em background).
6. `make tail-log` para acompanhar o log da execução em background e certificar-se de que tudo ocorre como esperado.

1.3.1 Variáveis

Na parte superior encontram-se definidas variáveis a serem utilizadas pela macro `${VARIABLE_NAME}`.

- `PYBIN`

  Nome do executável de Python 3 a ser utilizado pelo comando `make setup`. Padrão: `python3`. Alguns sistemas utilizam outros nomes, como `python-3.x` (onde x é um número). Nesse caso, o usuário deverá passar o nome correto como em:

  ```
  $ make setup PYBIN=python-3.x
  ```

- `VENVDIR`

  Nome do diretório onde será instalado o ambiente virtual de Python pelo comando `make setup`. Padrão: pasta `.venv` ao lado do `Makefile`.

- `VENVPY`

[1] Requer que o programa `make` esteja instalado no sistema Linux. Felizmente ele vem por padrão nas principais distribuições.

Caminho do interpretador Python dentro do ambiente virtual. Mesmo que a versão instalada de Python seja 3.x, a ferramenta `virtualenv` disponibiliza o link simbólico `python` para acessar o interpretador, seja qual versão for.

1.3.2 Sintaxe

Cada *target* do `Makefile` contém uma série de comandos para o shell cuja funcionalidade é auto-explicativa. Vale apenas notar o detalhe de que um *target* pode ser executado por outro e, em caso de falha, nenhum outro comando ou *target* será executado.

A exemplo, considere a *target* `run`:

```
run: check-venv
    ${VENVPY} logger/run.py --verbose
```

Antes de executar seus comandos (no caso, apenas uma linha conforme identação), será executada a *target* `check-venv`, que verifica a existência do ambiente virtual de Python e imprime uma menssagem de ajuda caso negativo.

Note: A sintaxe do `Makefile` impõe o uso de tabulação para comandos de um *target*. Editores configurados para expandir tabs em espaços (o que é recomendado para programação Python, por exemplo) deverão ser configurados para tratar arquivos `Makefile` de maneira separada, i.e., sem expandir tabs em espaços. Isto acontece por padrão no editor `Vim`.

CHAPTER 2

Firmware

Escrito na linguagem C++ suportada pela Arduino Toolchain, pode ser compilado utilizando a IDE do Arduino ou pelo terminal através de `$ make firmware`.

2.1 Estrutura de arquivos

```
meteorolog/
-- libs/                          # bibliotecas de terceiros
|   -- Adafruit_BMP085.cpp
|   -- Adafruit_BMP085.h          # sensor BMP085 (licença BSD)
|   -- DHT.cpp
|   -- DHT.h                      # sensor DHT11 e 22 (licença MIT)
|   -- RTClib.cpp
|   -- RTClib.h                   # relógio RTC DS1307 (domínio público)
-- meteorolog.ino                 # setup() e loop() da Arduino Toolchain
-- mysensors.cpp
-- mysensors.h                    # leitura dos sensores disponíveis
-- boardcommands.cpp
-- boardcommands.h                # execução de comandos para a placa
-- utils.cpp
-- utils.h                        # utilidades suplentes da Arduino Toolchain
```

Note: Os arquivos `.h` (cabeçalhos) contém os protótipos juntamente com a documentação do código implementado nos `.cpp`.

O ponto de entrada é o arquivo `meteorolog.ino`, pois ele define as duas seguintes funções padrões de um Sketch Arduino:

- `setup()`

  Executada uma vez quando a placa é ligada, inicializa a comunicação serial e chama `mysensors_setup()`, que fará inicialização dos sensores.

- `loop()`

  Executada enquanto a placa estiver ligada, verifica constantemente se há caracteres disponíveis na porta serial. Caso afirmativo, lê a string ali presente e encaminha ela para `execute_board_command()`. Essa função, por sua vez, interpreta o comando presente na string recebida e retorna uma string como resposta, que é então devolvida para a porta serial e o loop recomeça.

2.2 boardcommands.h

Os comandos esperados pelo firmware constituem strings no seguinte formato CSV:

```
nomeDoComando,arg1,arg2,...,argN
```

Essa string contendo o comando e seus argumentos é enviada para `execute_board_command()`, que interpretará a parte inicial `nomeDoComando` para delegar uma ação apropriada. O retorno de `execute_board_command()` é uma string contendo a resposta do comando ou, em caso de erros (comando inexistente, argumentos insuficientes, etc):

```
<invalid_commmand:nomeDoComando,...>
```

Note: Esses comandos devem ser enviados através de um monitor serial, como por exemplo o presente na IDE do Arduino. Alternativamente, esse projeto disponibiliza o *target* `$ make serial` para inicializar uma seção Python com uma comunicação aberta conforme configurado em `settings.ini`. Nesse caso, os comandos da placa devem ser enviados como segue:

```
>>> send('nomeDoComando,arg1,arg2,...,argN')
```

onde `send()` é uma função definida no script `init_serial.py` que recebe uma string a ser enviada à porta serial e retorna uma string contendo a resposta lida pela porta.

2.2.1 readSensors

A leitura dos sensores é feita pelo seguinte comando da placa:

```
readSensors,nome1,nome2,...,nomeN
```

onde os argumentos `nome1,nome2,...,nomeN` são transmitidos para `read_sensors()`, que fará as solicitações de leitura. Essa função itera sobre cada nome/apelido, passando o mesmo para `call_read_sensor()` de modo que a função correta de leitura seja invocada.

Por exemplo, sejam os dois seguintes sensores passados como argumento:

```
LDR,p
```

O primeiro deve levar à execução da função `read_LDR()` e o segundo, à execução de `read_BMP085_PRESSURE()` (pois `"p"` é um apelido para `BMP085_PRESSURE`). Ambas funções não recebem nenhum argumento e retornam uma string contendo, presumivelmente, o número medido ou um indicador de erro conforme programado em my_sensors.h.

A operação de `call_read_sensor()` depende então de mapear-se uma string como `"LDR"` para um ponteiro da função `read_LDR()`. Isso é alcançado em boardcommands.cpp através dos três seguintes vetores globais:

- `_sensor_names[]`: Contém o nome de todos os sensores disponíveis.
- `_sensor_nicknames[]`: Contém todos os respectivos apelidos.
- `_fp_read_sensor[]`: Contém os ponteiros de função das `read_X()`, onde `X` é o nome de um sensor – por exemplo, `&read_LDR` é o ponteiro de `read_LDR()`.

Percorrendo-se os dois primeiros, `call_read_sensor()` busca por um nome/apelido válido. Caso encontre, o índice é utilizado para acessar `_fp_read_sensor[]`, obter o ponteiro da função e finalmente executá-la.

Os vetores são incializados com as respectivas constantes declaradas em my_sensors.h.

2.2.2 setRTC

A configuração do relógio (se presente) na placa é feita com o comando:

```
setRTC,ano,mes,dia,hora,minuto,segundo
```

onde os argumentos `ano,mes,dia,hora,minuto,segundo` são repassados para `set_time_from_csv()` (my_sensors.h), cujo funcionamento depende do RTC em questão (ver seção sobre o `RTC_DS1307` e suas funções).

Exemplo:

```
setRTC,2015,8,17,14,43,10
```

caso bem sucedido deverá retornar a string:

```
done: 2015-08-17 14:43:10
```

2.3 mysensors.h

Esse módulo contém:

- Funções `read_X()` onde `X` é o nome de um sensor disponível;
- Função `mysensors_setup()` para inicialização programada de todos os sensores ao ligar a placa;
- Constantes a serem usadas por boardcommands.cpp nos vetores de lookup das funções `read_X()` :
  - `__SENSOR_COUNT`: total de sensores;
  - `__SENSOR_NAMES`: vetor de strings de nomes de todos os sensores;
  - `__SENSOR_NICKNAMES`: vetor de strings de apelidos de todos os sensores;
  - `__FP_READ_SENSOR`: vetor de ponteiros de função das `read_X()`.

Note: Entende-se aqui **sensor** por um elemento de software capaz de proporcionar um valor medido. Ou seja, ainda que um único componente eletrônico possa oferecer diversas medições (como temperatura e umidade do ar pelo DHT22), em termos do software cada medição é devida a um sensor, conforme cadastrado em http://dados.cta.if.ufrgs.br/emm.

De um modo geral as funções `read_X()` são bem simples, pois apenas invocam funções externas para obteção da medição numérica que é então convertida para string – o tipo de retorno esperado.

Por exemplo, sensores que atuam diretamente em pinos analógicos podem fazer uso de `analogRead()` (biblioteca do Arduino) e alguma matemática para calibração diretamente nas `read_X()` (a exemplo de `read_LDR()`). Já sensores que possuem controladoras Wire ou I2C em gearl acabam fazendo uso de bibliotecas separadas para melhor organização do código.

Note: Bibliotecas de terceiros são mantidas no subdiretório `libs/`.

2.3.1 Inserindo novos sensores

O software do repositório contém o código básico para os sensores suportados oficialmente, mas nada impede que novos sensores sejam adicionados. Para isso, siga os seguintes passos:

1. (opcional) Disponibilize uma biblioteca dentro de libs/

Caso o código de leitura seja complexo demais, considere criar uma nova biblioteca:

```
libs/novo_sensor.h
libs/novo_sensor.cpp
```

Dica: utilize orientação a objetos para melhor organização.

2. Registre o protótipo da nova função read_X() em mysensors.h

```
#include "libs/novo_sensor.h"
String read_NOVO_NOME();
```

onde `NOVO_NOME` será o nome do novo sensor.

3. Registre novo nome e apelido

- Incremente `__SENSOR_COUNT`;
- Inclua `NOVO_NOME` no vetor `__SENSOR_NAMES`;
- Inclua um apelido curto qualquer no vetor `__SENSOR_NICKNAMES`, na mesma posição utilizada por `NOVO_NOME` anteriormente;
- Inclua o ponteiro de função `&read_NOVO_NOME` no vetor `__FP_READ_SENSOR`, na mesma posição utilizada por `NOVO_NOME` anteriormente.

4. Implemente o código em mysensors.cpp

Exemplo :

```
// === NOVO_NOME SETUP ======================================

#define NOVO_NOME_PIN        8   // digital

String read_NOVO_NOME()
{
    return FloatToString(...);
}

// ==========================================================
```

CHAPTER 3

Logger

Software responsável por solicitar leitura dos sensores pela placa, guardar os dados localmente e também enviá-los ao servidor. Foi pensado para execução initerrupta em background através de um **daemon** registrado no gerenciador supervisor.

Seguem abaixo algumas das filosofias do software:

- Configuração amigável ao usuário leigo;
- Sistema completo de logging (dados e execução);
- Sincronismo de dados locais e remotos;

3.1 Estrutura de arquivos

Este software está escrito na linguagem Python 3 e apresenta a seguinte estrutura de arquivos:

```
logger/
-- app/                  # Python package contendo a aplicação
|   -- __init__.py       # torna essa pasta um package e faz inicializações
|   -- config.py         # gerencia configurações do programa
|   -- main.py           # implementa a classe Meterologger
-- logs/                 # guarda logs de execução
-- deploy.py             # script para registrar o daemon do logger
-- init_serial.py        # script para obter uma conexão serial
-- requirements.pip      # bibliotecas Python de terceiros
-- run.py                # ponto de entrada para excução do logger
```

Note: Normalmente em projetos Python, o arquivo de configuração fica presente no nível superior da pasta package ao lado de `run.py`. No caso deste projeto, optamos por mantê-lo na raíz do repositório, na posição de destaque ao lado do `Makefile`.

Projetos Python multi-arquivos fazem uso do conceito de **package**: pasta que contém um arquivo `__init__.py` para tornar-se acessível exteriormente como um módulo. Assim, o arquivo `run.py` que está fora do package pode fazer:

```
from app.main import Meteorologger
```

Módulos internos do package podem acessar uns aos outros por importação relativa, como acontece em `app/main.py`:

```
from .config import Config
```

onde o operador `.` refere-se ao nível atual (`main.py` e *]]]ncia]ncia]n'config.py'* estão na mesma pasta), `..` indica nível superior e assim por diante.

Em resumo, o código do aplicativo logger está todo na pasta `app/`, onde `Meteorologger` é a classe principal, e sua execução se dá pelo arquivo `run.py` com o seguinte ponto de entrada:

```
Meteorologger().run()
```

Note: A instanciação `Meteorologger()` é responsável principalmente pelo carregamento do arquivo de configurações e sua validação. Já o método `run()` contém o loop infinito que consiste na execução do logger.

3.2 Dependências

Além da linguagem Python 3 o logger depende das seguintes bibliotecas de terceiros:

- `pyserial`

  Possibilita comunicação entre Python e portas seriais. Aqui é utilizada para enviar comandos à placa Arduino e ler as respostas obtidas.

- `requests`

  Alternativa à biblioteca padrão `urllib`, usada para comunicação HTTP. Aqui é utilizada para enviar requests para a API do servidor em http://dados.cta.if.ufrgs.br/emm.

Note: As bibliotecas e suas versões estão listadas no arquivo requirements.pip para instalação automatizada através do gerenciador de pacotes **pip3** (vem por padrão com Python 3.4+).

Afim de evitar comprometer a instalação global do Python do usuário, optamos aqui pelo uso da ferramenta virtualenv. Todo o processo é automatizado pelo comando `$ make setup`, cujo resultado é a criação de uma pasta `.venv` contendo uma instalação isolada de Python 3 e as bibliotecas mencionadas acima. A execução correta desse comando depende dos seguintes programas no sistema:

- **python3** : interpretador da linguagem Python 3.x (recomenda-se versão 3.4);
  - Instalação no Debian: `$ sudo apt-get install python3`
- **pip3** : gerenciador de pacotes do Python 3;
  - Instalação no Debian: `$ sudo apt-get install python3-pip`
- **virtualenv** : criação de ambientes virtuais de Python;
  - Instale via **pip3**: `$ sudo pip3 install virtualenv`

Adicionamente, para que o logger possa ser executado em background (ver seção *deploy.py*) esse projeto também requer a seguinte ferramenta:

- **supervisor** : gerenciador de daemons (processos background);
  - Instalação no Debian: `supervisor`

Note: Algumas distribuições podem possuir o executável de Python 3.x registrado em nomes diferentes de `python3` (assumido por `$ make setup`). Nesse caso, forneça o nome correto fazendo, por exemplo:

```
$ make setup PYBIN=python-3.x
```

onde x é um número. O mesmo vale para o **pip3**:

```
$ sudo pip-3.x install virtualenv
```

3.3 run.py

Este arquivo consiste no ponto de entrada da aplicação, permitindo a execução do logger por um interpretador Python: `$ python3 run.py [options]`. Entretanto, conforme descrito na seção anterior sobre dependências, deve ser utilizado o interpretador do ambiente virtual através de:

```
$ make run
```

Note: Deve ser executado após a criação do ambiente virtual com `$ make setup`.

3.3.1 Parâmetros

- `--background`

  Desabilita impressão de menssagens de log na saída padrão.

3.4 deploy.py

Conforme mencionado na introdução, o logger foi pensado como um programa para ser executado em background. Por exemplo, as menssagens do log de execução são escritas em um arquivo dentro de `logger/logs` através da biblioteca padrão logging. O script `deploy.py` é responsável por registrar um novo processo *daemon* no Supervisor para colocar o logger em execução no background, persistindo mesmo após a máquina ser reiniciada.

A operação é feita pelo seguinte comando, que requer permissões de root:

```
$ make deploy
```

O registro de um *daemon* no supervisor consiste na criação de um arquivo de configuração em `/etc/supervisor/conf.d/` e a subsquente execução de `supervisorctl update`. É exatamente isso que faz a função `deploy_supervisor()`. O arquivo de configuração utiliza o seguinte modelo presente na string `TEMPLATE_SUPERVISOR_CONF`:

```
[program:{PROCESS_NAME}]
command={BASE_DIR}/.venv/bin/python {BASE_DIR}/logger/run.py
directory={BASE_DIR}
user=root
autostart=true
autorestart=true
redirect_stderr=true
stdout_logfile={BASE_DIR}/logger/logs/stdout.log
logfile={BASE_DIR}/logger/logs/supervisor-{PROCESS_NAME}.log
```

Os valores substituídos nesse template estão declarados nas constantes globais, também utilizadas em outros lugares:

- `PROCESS_NAME`: apelido para o *daemon* dentro do supervisor. Valor: `meteorologger`.
- `BASE_DIR`: diretório raiz do projeto, que contém o `Makefile`. Obtido pelo cálculo relativo da posição do arquivo `deploy.py`.

Sobre as configurações do **Supervisor**, vale destacar:

- `redirect_stderr`: menssagens de erro serão escritas na saída padrão.
- `stdout_logfile`: além das menssagens da saída padrão, o traceback aparecerá nesse arquivo caso o programa falhe.

Por fim, o mesmo script `deploy.py` é utilizado também para *undeployment*, isto é, remoção do *daemon* no Supervisor. Isso é feito passando-se o argumento `-u` para o script, operação disponibilizada pelo comando:

```
$ make undeploy
```

3.5 app/config.py

O arquivo de configuração utilizado pelo logger, settings.ini, encontra-se na pasta raiz do projeto ao lado do `Makefile` por ser uma posição de destaque. Foi concebido para ser configurado por um usuário leigo em computação.

Existem várias opções de sintaxe para arquivos de configuração no universo Python: *XML*, *JSON*, *YAML*, *INI*, etc. Apesar de que sintaticamente o *YAML* seja mais interessante para projetos Python por levar em conta a identação, esse mesmo motivo dificultaria a configuração por usuários leigos. A flexibilidade do formato *INI*, tratado pela biblioteca padrão `configparser`, determinou sua escolha para esse projeto.

O módulo app/config.py é responsável pela leitura e validação do arquivo de configuração através da classe `Config`, que deverá se comportar como um dicionário para obteção das seções e chaves:

```
config = Config()
config['reading']['sleep_time']     # acessa a chave 'sleep_time' da seção 'reading'
```

Como todos valores lidos e armazenados pelo objeto `configparser.ConfigParser` são strings, optamos aqui por utilizar e manipular uma cópia em dicionário das configurações através do atributo `_sections` deste objeto. Assim, quando uma seção de configuração é acessada dentro de `Config()` (instancia) com o operador `[]`, o método mágico `__getitem__()` retorna um dicionário dentro de `_sections` podendo conter qualquer tipo de dados como chaves e valores.

Note: As chaves do dicionário `_sections` são todas em *lowercase*, independente do original em `settings.ini`! Esse fato é levado em conta na implementação da classe `main.Meteorologger`.

No que diz respeito à validação dos dados, a classe `Config` implementa as três seguintes **exceptions** (classes que herdam de `Exception`):

- `ConfigMissingSectionError`

  Exemplo de menssagem:

```
[reading]
 ^
Missing section!
```

- `ConfigMissingKeyError`

  Exemplo de menssagem:

```
[reading]
; time between logger cycles, in minutes
SLEEP_TIME =
             ^
Missing key!
```

- `ConfigValueError`

  Exemplo de menssagem:

```
[reading]
; time between logger cycles, in minutes
SLEEP_TIME = 5-
             ^
TypeError: Number expected!
```

Baseando-se na máxima pythônica de que "nenhum erro deve passar despercebido", `ConfigMissingSectionError` e `ConfigMissingKeyError` poderão acontecer no método `assert_config_keys()` responsável por assegurar a existência de seções e chaves em `settings.ini` tomando `DEFAULT_INI` como referência. Já `ConfigValueError` poderá acontecer ao longo dos métodos `validate_section_()`, descritos na próxima seção.

3.5.1 Validações

- `validate_section_server()`

  Utiliza valores da seção `[server]` para compor a URL utilizada na postagem de dados. O valor `URL` consiste na base do endereço do servidor, opção disponibilizada para o caso de o usuário desejar utilizar outro servidor que não o nosso – por exemplo, um servidor local como `http://localhost`. O valor `BOARD_ID` é utilizado pela URL e também pela API do site, juntamente com `USER_HASH`, ao realizar autenticação do usuário da placa.

- `validate_section_reading()`

  Utiliza os valores da seção `[reading]` para determinar quais sensores terão a leitura solicitada pelo logger e também se deverá ser lido o relógio da placa (visto como um sensor de nome `RTC_DS1307`). A ordem dos sensores na chave `SENSORS` determinará as colunas do arquivo *datalog.csv* (armazenamento local de dados).

  Introduz a nova chave `reading/command` contendo a linha de comando a ser enviada para a porta serial. Essa linha vai conter todos os sensores da chave `SENSORS`, e também o `RTC_DS1307` caso a chave `RTC_DS1304` seja `true`.

- `validate_section_datalog()`

  Valida o caracetere utilizado como separador CSV do arquivo *datalog.csv*, configurado na chave `CSV_SEP` da seção `[datalog]`. Além de eliminar opções inválidas, decodifi)ncia)ncica o caractere para uso ASCII correto posteriormente.

- `validate_section_arduino()`

  Valida a chave `SERIAL_PORT` da seção `[arduino]`. O usuário pode especificar uma ou mais portas separadas por vírgula para que o logger tente conexão caso uma delas falhe. Adicionalmente, essa chave pode ser deixada em branco, caso em que será gerada a seguinte lista de portas:

  `['/dev/ttyACM0', '/dev/ttyUSB0', ..., '/dev/tty/ACM4', '/dev/ttyUSB4']`

  para que o logger tente buscar sozinho a porta onde está a placa.

3.6 app/main.py

Este módulo contém toda a funcionalidade do logger em si implementada na classe `Meteorologger`. Uma leitura do método `Meteorologger.run()` (ponto de entrada) dá uma idéia clara de cada etapa necessária ao fluxo de

execução.

3.6.1 Meteorologger.__init__()

A instanciação dessa classe inicializa o atributo `background` (flag utilizada pelo método `setup_logging`) e também atributo `config` com uma instancia da classe `Config`. Conforme discutido na seção anterior sobre `app/config.py`, é nesse momento que ocorre a validação do arquivo de configuração.

3.6.2 Meteorologger.setup_session_files()

Os seguintes arquivos serão criados a cada nova execução do logger (seja em *foreground* ou em *background*):

```
logger/logs/exec-%Y-%m-%d-%H-%M-%S.log
data/datalog-%Y-%m-%d-%H-%M-%S.csv
```

onde `%Y-%m-%d-%H-%M-%S` consiste no `datetime` do início da execução. Estabelecer o nome desses arquivos é o objetivo primário de `setup_session_files()`. O primeiro arquivo é o log de execução e o segundo é o log de dados no formato **CSV** (*comma-separated values*), cuja primeira linha contendo o nome das colunas é escrita já na execução deste método para garantir existência e permissões de arquivo.

3.6.3 Meteorologger.setup_logging()

Considerando que o logger foi pensado para execução em *background*, o uso de `print()` para menssagens de log não consiste na melhor abordagem – por exemplo, deseja-se que a mesma menssagem apareça tanto em arquivo como na saída padrão. A excelente biblioteca padrão `logging` traz diversas soluções para esses e outros problemas relativos à criação de logs.

O log em arquivo é criado conforme especificações de `logging.basicConfig()`, seja a execução feita em *background* ou *foreground*. Neste último caso, desejamos imprimir também na tela as mesmas menssagens de log. Isto é alcançado adicionando-se o objeto `logging.StreamHandler()` ao logger principal `root`.

3.6.4 Meteorologger.get_serial()

Esse método varre a lista de portas seriais `self.config['arduino']['serial_port']` em busca de uma conexão válida. Quando uma tentativa falha, registra-se um `logging.error()` prossegue-se com o próximo item da lista, retornando ao início quando o último item também falha.

Note: Vale lembrar que uma lista de portas é gerada automaticamente quando a chave `arduino/SERIAL_PORT` de `settings.ini` é deixada em branco. Nesse caso, deve-se assegurar de que a única placa Arduino presente na máquina é aquela na qual deseja-se conectar.

3.6.5 Meteorologger.serial_read()

A leitura dos dados consiste no envio de uma string para a porta serial e a consequente leitura da string de resposta. Logo, a primeira coisa a ser feita é obter uma conexão serial pelo método `get_serial()`. Em seguida, entra-se em um loop que encerra apenas quando a resposta obtida é uma string ASCII válida.

A comunicação serial ocorre através do objeto `serial.Serial()` (biblioteca pyserial) retornado pelo método `get_serial()`. Tendo a conexão estabelecida, envia-se a string contendo o comando de leitura configurado em `self.config['reading']['command']` – detalhe para o fato que strings em Python 3 são *unicode* por padrão e portanto devem ser convertidas para `bytes()`.

Uma boa prática consiste em dormir por um intervalo de tempo (`BOARD_RESPONSE_DELAY`, 3 segundos, por exemplo) para aguardar enquanto a placa é reiniciada pelo fato da conexão serial ter sido estabelecida via `pyserial`.

A leitura da string de resposta retorna bytes que devem ser convertidos para string. No entanto, pode acontecer de bytes retornados não serem caracteres ASCII válidos (por exemplo, contém códigos de controle de envios interrompidos anteriormente). O método `_decode_bytes()` assegura essa validação.

3.6.6 Meteorologger.create_json()

Esse método recebe uma string de valores CSV, por exemplo:

```
<NaN>,80.0,101201,45.5,2015-09-01 18:30:12
```

correspondendo aos sensores cuja leitura foi solicitada conforme `self.config['reading']['sensors']`, por exemplo:

```
DHT22TEMP,DHT22AH,BMP085_PRESSURE,LDR,RTC_DS1307
```

e então retorna um dicionário *JSON* válido, por exemplo:

```
{
    "datetime":
    {
        "format": "%Y-%m-%d-%H-%M-%S",
        "source": "RTC_DS1307",
        "value": "2015-09-01-18-30-12"
    },
    "sensors":
    {
        "DHT22_TEMP": "NaN",
        "DHT22_AH": 80.0,
        "BMP085_PRESSURE": 101201,
        "LDR": 45.5
    }
}
```

O formato de serialização *JSON* é bastante usado na web, inclusive pela API do site http://dados.cta.if.ufrgs.br. O dicionário acima contém tudo que o servidor precisa para armazenar os valores corretamente no banco de dados.

No exemplo acima, a leitura de `DHT22_TEMP` retornou a string `<NaN>`. É uma convenção deste projeto que todos os erros retornados pelo firmware apareçam entre <> para facilitar a identificação. O sensor `BMP085_PRESSURE`, por exemplo, poderia ter retornado `<BMP085_not_found>`. Independente do erro acusado pelo firmware, `"NaN"` será gravado como leitura tanto no datalog local como no servidor pois é um valor tratável pelas bibliotecas de plotagem.

3.6.7 Meteorologger.write_datalog()

Não há segredo neste método: simplesmente adiciona uma nova linha CSV no arquivo de log de dados local com base no *JSON* recebido. Naturalmente o arquivo de dados não deve incluir notas de erro, de modo que apenas o valor `NaN` apacerá nas colunas onde algum erro de leitura tenha ocorrido.

Caso deseje se informar sobre o erro o usuário pode fazer uma simples busca textual pelo *timestamp* no log de execução.

3.6.8 Meteorologger.send_to_server()

Utiliza a excelente biblioteca `requests` para enviar os dados ao servidor, processo elaborado em diversas etapas para garantir o tratamento de possíveis erros:

1. Adiciona-se o *JSON* resultante da leitura atual no arquivo `data/outgoing.json` (será criado caso não exista). Cada linha desse arquivo conterá um *JSON* válido para o servidor.
2. Abre-se o arquivo `data/outgoing.json` para leitura e converte-se as linhas para uma lista de dicionários *JSON* válidos ao servidor. Essa lista é armazenada no atributo `"data"` do *JSON* principal.
3. Adiciona-se o atributo `"user_hash"` contendo a chave de autenticação do usuário da placa ao *JSON* principal.
4. É feita uma tentativa de envio do *JSON* principal. Caso bem suscedida, apaga-se o arquivo `data/outgoing.json`. Caso falhe, seja por servidor fora do ar ou seja por uma resposta negativa do mesmo (resposta da API ser algo como `{"error": ...}`), nada se faz ao arquivo `data/outgoing.json`.

Naturalmente, enquanto a comunicação do servidor falhar, novas linhas são adicionadas ao arquivo `data/outgoing.json` e se acumularão com o tempo até que um envio único seja bem sucedido. Repare que tudo isso acontece de maneira independente ao log local.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search