Universidade do Minho

Mestrado Integrado em Engenharia Biomédica

Trabalho Parte 1

Aplicações Distribuídas

Grupo 1

A74727 Ana Ramos

A74753 Ana Sousa

A75088 Ana Machado

Braga

Novembro 2017

Resumo

Este projeto teve como principal objetivo a utilização da linguagem de programação *Python* para a criação de estruturas de dados, da arquitetura cliente-servidor, recorrendo ao XMLRPC, e do controlo de concorrência. Assim sendo, desenvolveu-se um sistema de gestão de receitas e farmácias onde é tido em conta os medicamentos, os médicos e os utentes.

Índice

1. Introdução

O presente trabalho tem como intuito a criação de um sistema de gestão de farmácias e receitas eletrónicas. Assim sendo, e com o objetivo de aplicar os conhecimentos da linguagem *Python* para a implementação deste sistema, tem-se em vista a utilização de estruturas de dados, neste caso listas e dicionários, a arquitetura cliente/servidor, com o XMLRPC, e o controlo da concorrência. Foi também necessário garantir os seguintes requisitos:

- Levantamento dos requisitos das aplicações envolvidas;
- Definição das entidades envolvidas;
- Definição das interfaces dos diferentes serviços que idealizar;
- Implementação da lógica de negócio necessárias ao correto funcionamento da(s) aplicações;
- Implementação da(s) aplicações cliente e servidor;
- Sempre que ao aviar uma receita não exista stock, a receita deve ficar bloqueada até à existência de stock;
- Definição de estatísticas:
    - Utentes
        - Média de receitas por utente;
        - Utentes com mais receitas;
        - Utentes mais gastadores;
    - Médicos
        - Média de receitas por médico;
        - Médicos com mais receitas;
    - Medicamentos
        - Stocks;
        - Stocks em alarme;
        - Medicamentos mais vendidos.

1.1. Estruturas de Dados

1.1.1. Dicionários

Os dicionários permitem associar os dados, podendo ser definidos como um conjunto não ordenado de pares chave-valor, onde as chaves são únicas numa dada instância do dicionário. O principal objetivo da sua utilização é a possibilidade de armazenar dados para posterior utilização. É de salientar que num dicionário a sua chave pode conter uma lista.[1][2]

1.1.2. Listas

As listas são estruturas abstratas de dados utilizadas para agrupar um conjunto de valores repetidos, ou não repetidos.

Cada ocorrência de uma lista deve estar separada por vírgulas e pode ser designada de item, entrada ou elemento. Devem ser escritas entre parêntesis retos e podem ser de diferentes tipos (*string*, valor e lista dentro de lista). [1][3]

1.2. *Threads*

As *threads* permitem dividir um processo em duas ou mais tarefas que podem ser executadas simultaneamente e concorrentemente.

Em *python*, a classe *thread* representa uma atividade que está a ocorrer numa *thread* de controlo.[4]

2. XML-RPC

No presente trabalho recorreu-se ao RPC (*Remote procedure calls)* para construir a uma aplicação distribuída, baseada no modelo cliente-servidor.

Um protocolo específico do RPC é o *SimpleXML-RPC.* Este envia uma requisição HTTP ao servidor que implementa o protocolo, por outro lado, o cliente apenas chama

um método do sistema remoto. Este protocolo caracteriza-se por ser *single thread*, isto é, apenas opera com uma única *thread.*

Para o sistema de base de dados em estudo é necessário utilizar várias *threads* em simultâneo, sendo necessário recorrer ao *ThreadingTCPServer*. Assim sendo, quando o servidor recebe mais do que um pedido, este irá criar uma *thread* para cada, respondendo a todos, sem que seja bloqueada a *thread* principal.[4]

3. Desenvolvimento da Base de Dados

O presente sistema de base de dados está estruturado da forma representada na **figura 1**. Primeiramente, existem os Modelos que são constituídos pelas respetivas entidades. Estes modelos entram em contacto com o Negócio e com a Base de Dados. Por um lado, a base de dados contem todos os dicionários com os dados de cada entidade. Por outro lado, o negócio representa o sistema de gestão (classe *ManagerFarmacia*). Por último, a classe *FarmaciaServices* representa a API do sistema e contem todos os métodos que estão disponíveis ao Cliente.

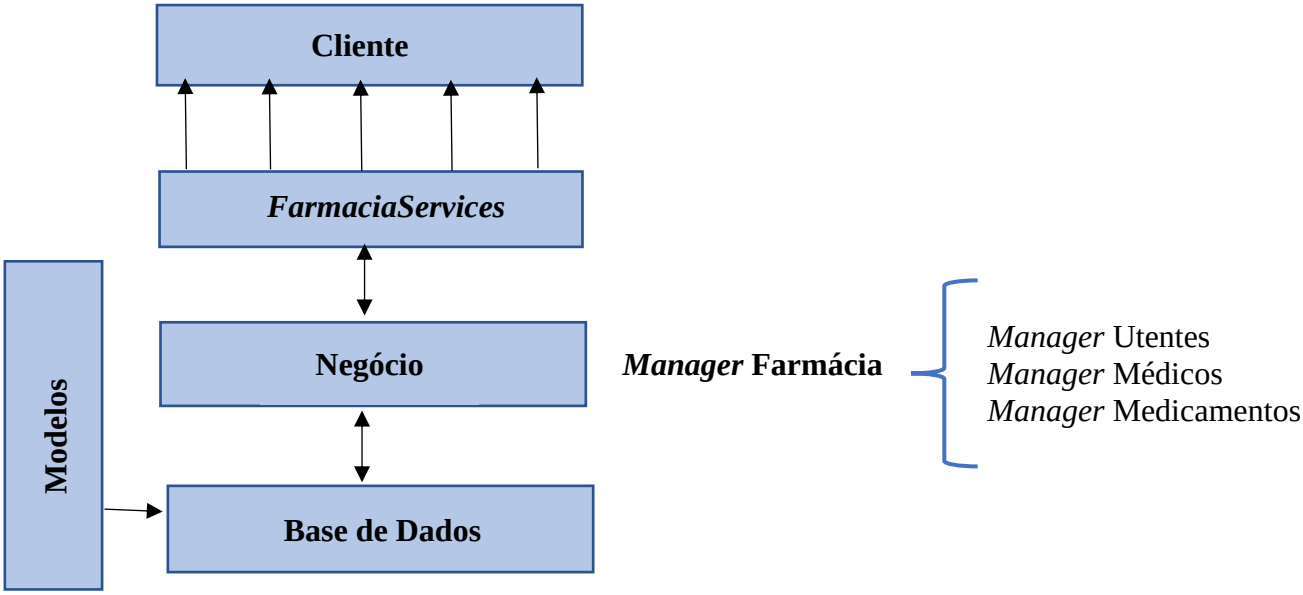

Figura 1 – Representação esquemática do sistema de base de dados.

3.1. Entidades em Estudo

Neste trabalho, para uma melhor compreensão do caso em estudo, recorreu-se à criação e respetiva definição de cinco entidades essenciais para um sistema de gestão de stocks em uma ou mais farmácias. Para tal, recorreu-se ao programa *TerraEr* para mostrar as relações entres as diferentes entidades. É de salientar que existem três formas de

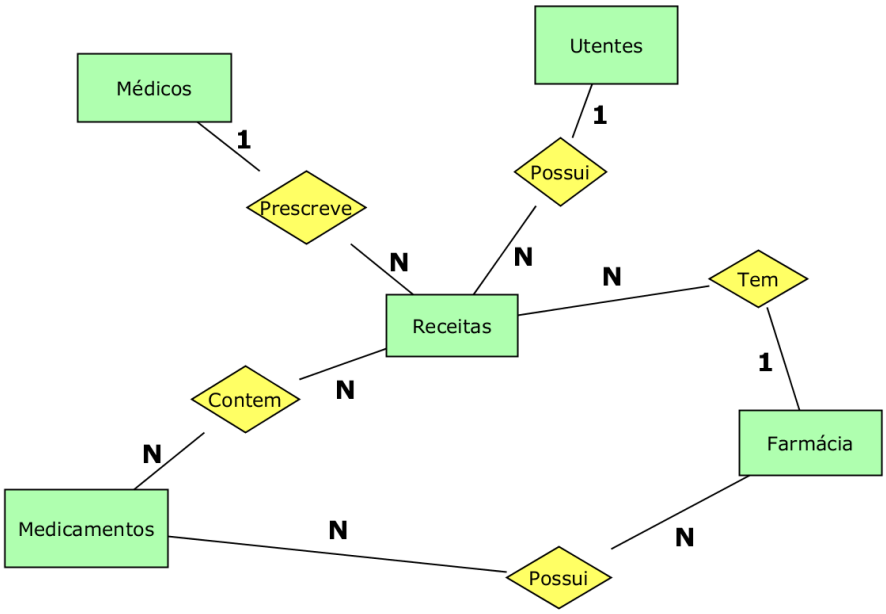

Figura 2- Relação entre as Entidades.

relacionar duas entidades: N-N, 1-N ou N-1.

Pela observação da **figura 2**, pode-se constatar que uma Farmácia se relaciona com os medicamentos, uma vez que uma farmácia pode ter em stock vários medicamentos, assim como um medicamento pode estar presente em uma ou mais farmácias (relação N-N). No entanto, a entidade Farmácia também se relaciona com as Receitas, já que contém uma lista de receitas, mas em contrapartida uma determinada receita não pode estar em mais que uma farmácia (relação 1-N). Posteriormente, a entidade Receita relaciona-se com os medicamentos, utentes e médicos. Na primeira está presente uma relação de N-N, já que um medicamento pode estar presente numa ou mais farmácias, mas também uma receita pode conter mais que um medicamento. Relativamente à segunda e terceira, ambas possuem uma relação de 1-N. Isto deve-se ao facto de que uma receita apenas contem um utente e um médico, mas tanto o médico como o utente podem ter N receitas.

3.1.1. Entidade Pessoa

A classe *Pessoa* diz respeito aos dados pessoas, tais como: nome, morada, número de identificação fiscal (NIF), número do cartão de cidadão e contacto telefónico.

3.1.2. Entidade Médico

A entidade médico é constituída pela classe *Medico,* que herda da classe *Pessoa* para alguns atributos, tais como: nome, morada, número de identificação fiscal (NIF), número do cartão de cidadão e contacto telefónico. Para além destes atributos, os médicos têm ainda um atributo relacionado com a sua profissão, a especialidade.

3.1.3. Entidade Utente

A entidade Utente é constituída pela classe *Utente* e está associada aos utentes que fazem parte, ou serão inseridos, na base de dados do sistema de gestão apresentado.

A classe *Utente* contém os atributos associados ao utente enquanto pessoa, daí ser herdada a classe *Pessoa*. Além disso, contém o atributo referente ao identificador do utente, único para cada utente.

3.1.4. Entidade Medicamento

A entidade medicamento é constituída pela classe *Medicamento.* Refere-se ao conjunto de medicamentos existentes na base de dados, e tem como atributos o seu nome e a dosagem.

3.1.5. Entidade Farmácia

A entidade farmácia contem informações acerca das farmácias existentes na base de dados, tais como a identificação da farmácia, o seu nome e a sua morada.

3.1.6. Entidade Receita

A entidade Receita, constituída pela classe *Receita*, tem como atributos a identificação da farmácia, da receita (identificador único) e do utente, assim como o nome do médico e a lista de medicamentos. A entidade está relacionada com as receitas prescritas aos utentes pelos médicos, numa determinada farmácia.

3.1.7. Entidade Stock

A entidade Stock é dada pela classe *Stock* e é constituída pelos seguintes atributos: identificação da farmácia, nome e quantidade do medicamento. Esta entidade representa um determinado medicamento numa farmácia, sendo definido por uma condição que está associada a um *lock* (espera, bloqueio e libertação).

4. Gestão da base de dados

4.1. Exceções

De modo a garantir a coerência da base de dados, foram implementadas quatro exceções. Para tal, implementou -se quatro tipo de exceções, sendo elas *IDJaExisteException, NomeJaExisteException, NIFJaExisteException, CCJaExisteException.*

No método que permite adicionar médicos à base de dados(*Add)* as exceções impossibilitam adicionar médicos com determinados atributos iguais aos já existentes na base de dados, esses atributos são: o nome (*NomeJaExisteException*), o número de identificação fiscal (*NIFJaExisteException*) e o número de cartão de cidadão *(CCJaExisteException*). Desta forma, é garantido que todos os médicos têm nome, número de identificação fiscal e número de cartão de cidadão únicos.

O método usado para adicionar utentes à base de dados(*AddUtente*) garante que cada utente os atributos identificação de utente, número de identificação fiscal e número de cartão de cidadão são únicos. Para isso recorre a exceções, sendo elas: *IDJaExisteException, NIFJaExisteException e CCJaExisteException, respectivamente.*

A exceção *IDJaExisteException* também foi usada para adicionar farmácias e à base de dados, método *CreateFarmacia*, respectivamente. Deste modo, impede que existam na base de dados farmácias com a mesma identificação.

4.2. Sistema de Gestão de Médicos

No presente trabalho, criou-se uma classe denominada *ManagerMedicos* que possibilita gerir a entidade médicos.

Assim sendo, é possível adicionar médicos (método *Add)* à base de dados, definindo os seus atributos, tais como: nome, morada, número de identificação fiscal, cartão de cidadão, contacto e especialidade. A inserção de médicos garante, através de exceções, que cada médico da base de dados tem nome, número de identificação fiscal e número de cartão de cidadão únicos.

Além disso, é possível eliminar médicos (método *Delete*) do sistema, obter as informações de um determinado médico através do seu nome, ou de todos os médicos existentes na base de dados (método *Get* e *GetAll*, respetivamente). Também permite alterar os atributos de um médico já registado (método *Update*), determinar a média de receitas que um médico já prescreveu (método *MediaRceitasMedico)*, assim como determinar o médico que prescreveu mais receitas.

4.3. Sistema de Gestão de Utentes

A gestão dos utentes da base de dados realiza-se na classe *ManagerUtentes*. Esta classe permite, entre outras coisas, inserir utentes na base de dados (método *AddUtente*), sendo necessário definir os seus atributos, tais como: identificação, nome, morada, número de identificação fiscal, cartão de cidadão e contacto. Havendo a garantia, através de exceções, que cada utente tem uma identificação, um número de identificação fiscal e um número de cartão de cidadão únicos.

Além disso, esta classe permite eliminar utentes (método *Delete*) da base de dados, alterar os atributos de um utente já registado (método *UpdateUtente)*, determinar a média de receitas que um determinado utente recebeu(*MediaReceitasporUtente*), bem como os que receberam mais receitas(*UtenteMaisReceitas*), a quem foi prescrito maior número de

medicamentos(*UtentesGastadores*), e possibilitou obter as informações de um determinado utente através da sua identificação, ou obter as informações de todos os utentes existentes na base de dados (método *GetUtente* e *GetUtentes*, respetivamente).

4.4. Sistema de Gestão de Medicamentos

A gestão dos medicamentos faz-se na classe *ManagerMedicamentos*, esta classe possibilita, por exemplo, inserir medicamentos na base de dados (método *Add*), sendo necessário definir os seus atributos (nome e dosagem). Para além disso, permite obter as informações acerca de um determinado medicamento e de todos os medicamentos registados (métodos *Get* e *GetAll*, respetivamente). Também permite alterar as informações e apagar um medicamento especifico, através dos métodos *Update* e *Delete*, respetivamente.

4.5. Sistema de Gestão de Farmácias

A gestão das farmácias, ocorre na classe *ManagerFarmacia*, possibilita a gestão das farmácias, gerindo as suas receitas e *stocks*, e invoca os métodos mais relevantes nas restantes classes de gestão (de médicos, medicamentos e de utentes), reunindo-os nesta classe. Ela permite inserir farmácias na base de dados(*CreateFarmacia*), sendo necessário definir os seus atributos (identificação, nome e morada), e obter informações acerca de todas as farmácias da base de dados.

4.5.1. Sistemas de gestão de receitas e stocks

O *ManagerFarmacia* possibilita também a inserção de receitas numa determinada farmácia, através do método *InsertReceita*, os seus argumentos são: identificação da farmácia, do utente, o nome do médico e a lista de medicamentos.

Ao inserir a receita da base de dados da farmácia, verifica-se a existência do médico e do utente na base de dados, caso não existam, adiciona-se o nome do médico e a identificação do utente. Para uma posterior atualização dos dados, é necessário recorrer aos métodos *UpdateMedico* e *UpdateUtente*, respetivamente.

É também necessário que esta tenha a quantidade de medicamentos pedida na receita. Para controlar este processo, recorre-se ao método *Pendente.*

O método *Pendente* verifica se existe quantidade de medicamentos suficiente. Inicialmente, o medicamento encontra-se bloqueado, no caso de não existir, o medicamento fica em espera(*condition.wait*) até que, através do método *UpdateStock,* seja inserida quantidade suficiente de medicamento para libertar o pedido. Após a inserção da quantidade de medicamentos, este método notifica o método Pendente, através do *condition.notifyall,* fazendo com que o último liberte o medicamento (*condition.release*). Por outro lado, caso a quantidade de medicamento seja suficiente, ocorre imediatamente a libertação do medicamento(*condition.release*).

Admite-se que a receita só e adicionada à farmácia caso exista a quantidade necessária de todos os medicamentos pedidos e esses medicamentos têm de existir na base de dados. A existência dos medicamentos na base de dados é assegurada pelo método *ValidarMedicamentos.*

A quantidade dos medicamentos em stock numa farmácia deve ser superior a quatro unidades, caso contrário são adicionados à secção de alarmes da base de dados.

5. Cliente-Servidor

No presente trabalho, criou-se uma classe *FarmaciaServices,* que será registada no servidor, onde se reuniram todos os métodos a que o cliente terá acesso. Para esta classe realizou-se uma ligação *localhost* à qual o cliente terá de estabelecer ligação para aceder aos métodos pretendidos.

Para criação do servidor construiu-se a classe *MyXMLRPCServer* que permite implementar, através do *ThreadingTCPServer,* um sistema *multi-thread.* Ou seja, permite realizar vários processos simultaneamente. Este sistema cria a necessidade de resolver problemas de concorrência (**figura 3**).

Relativamente ao cliente, apenas precisará de aceder a este *localhost* para ter acesso a todos os métodos (**figura 4**).

```
class MyXMLRPCServer (ThreadingTCPServer,SimpleXMLRPCServer):
    pass

# Create server
server = MyXMLRPCServer(("localhost", 12345),SimpleXMLRPCRequestHandler, logRequests=True,
  allow_none=True, encoding=None)
server.register_introspection_functions()
server.register_multicall_functions()

server.register_instance(FarmaciaServices())
```

Figura 3 - Criação do servidor e registo da instancia *FarmaciaServices*.

```
server = xmlrpc.client.ServerProxy('http://localhost:12345/', encoding=None)
```

Figura 4 - Estabelecimento da ligação do cliente ao servidor.

6. Problemas de Concorrência

A arquitetura cliente-servidor com *multi-threads* origina problemas ao nível da concorrência, ou seja, quando vários clientes estão a aceder simultaneamente. Para tal, foi necessário implementar, nos métodos que manuseavam os dados da base de dados, um *threading.Rlock.*

O *threading.Rlock* permite que uma *thread* em utilização bloqueie uma outra *thread* que queira iniciar o mesmo processo. Esta segunda *thread* só é libertada (*release())* quando a primeira terminar.

7. API (*Aplication Programming Interface*)

A classe *FarmaciaServices* representa a API (*Aplication Programming interface*) deste sistema, isto é, é a classe que contém um conjunto de métodos de comunicação entre o cliente e o servidor. Nesta classe, tem-se os métodos relacionados com as farmácias **(figura 5**), com as receitas (**figura 6),** com os utentes (**figura 7**), com os médicos (**figura 8**), com os medicamentos (**figura 9**).

```
    def __init__(self):
        self.manager = ManagerFarmacia()

    def listaFarmacias(self):
        try:
            content = []
            content = self.manager.GetFarmacias()
            if len(content) == 0:
                info = {}
                info["Farmacia Id"] = ""
                info["Nome"] = ""
                info["Morada"] = ""
            return content
        except Exception as ex:
            raise Exception(ex)

    def adicionarFarmacia(self,idfarm,nome,morada):
        try:
            self.manager.CreateFarmacia(int(idfarm),nome,morada)
        except Exception as ex:
            raise Exception(ex)
```

Figura 5 -Secção da API relacionada com as farmácias.

```
def RegistarReceita(self, idFarmacia, idReceita, idUtente, nomeMedico, listMedicamentos):
    result = self.manager.InsertReceita(int(idFarmacia),int(idReceita),int(idUtente), nomeMedico, listMedicamentos)
    print(result)

def UpdateStock(self, idFarmacia, nomeMedic, quantidade):
    self.manager.UpdateStock(int(idFarmacia), nomeMedic, quantidade)

def listarReceitas(self,idFarm):
    return self.manager.ListarReceitas(int(idFarm))
```

Figura 6- Secção da API relacionada com as receitas.

```
    def adicionarUtente(self,idUtente,nomeUtente, morada, nif, cc, tel):
        self.manager.AddUtente(int(idUtente),nomeUtente, morada, nif, cc, tel)

    def updateUtente(self,idUtente,nomeUtente, morada, nif, cc, tel):
        self.manager.UpdateUtente(int(idUtente),nomeUtente, morada, nif, cc, tel)

    def apagarUtente(self,idUtente):
        self.manager.DeleteUtente(int(idUtente))

    def selectUtente(self,idUtente):
        return self.manager.GetUtente(int(idUtente))

    def listUtentes(self):
        try:
            content = self.manager.GetUtentes()
            return content
        except Exception as ex:
            raise Exception(ex)

    def UtentesMaisReceitas(self):
        return self.manager.UtentesMaisReceitas()

    def MediaReceitaPorUtente(self,idUtente):
        return self.manager.MediaReceitaPorUtente(int(idUtente))

    def UtentesGastadores(self):
        return self.manager.UtentesGastadores()
```

Figura 7-Secção da API relacionada com os utentes.

```
def AddMedico(self,nomeMedico,morada, nif, cc, tel,especialidade):
    self.manager.AddMedico(nomeMedico,morada, nif, cc, tel,especialidade)

def updateMedico(self,nomeMedico,morada, nif, cc, tel,especialidade):
    self.manager.UpdateMedico(nomeMedico,morada, nif, cc, tel,especialidade)

def apagarMedico(self,nomeMedico):
    self.manager.DeleteMedico(nomeMedico)

def SelectMedico(self,nomeMedico):
    return self.manager.GetMedico(nomeMedico)

def listMedico(self):
    return self.manager.GetMedicos()

def MedicosMaisReceitas(self):
    return self.manager.MedicosMaisReceitas()

def MediaReceitaPorMedico(self,nomeMedico):
    return self.manager.MediaReceitaPorMedico(nomeMedico)
```

Figura 8- Secção da API relacionada com os médicos.

```
    def AddMedicamento(self,nome, dosagem):
        self.manager.AddMedicamento(nome, dosagem)

    def listMedicamentos(self):
        val=self.manager.GetMedicamentos()
        return val

    def listMedicamentosStock(self,idfarm):
        val=self.manager.ListMedicamentoStock(idfarm)
        return val

    def MedicamentosAlarme(self,idfarm):
        return self.manager.MedicamentosAlarme(idfarm)

    def VerificarAlarme(self,ifdfarm):
        return self.manager.AlarmeMedicamentos(ifdfarm)

    def MedicamentosMaisVendidos(self,idFarmacia):
        return self.manager.MedicamentVendidos(idFarmacia)
```

Figura 9- Secção da API relacionada com os medicamentos.

8. Conclusão

Ao longo do projeto desenvolvido consolidou-se conhecimento sobre a linguagem de programação *Python*. Deste modo, através da implementação de uma estrutura de dados cliente-servidor utilizando XML-RPC e controlo de concorrência, foi implementado um sistema de gestão de farmácias e receitas eletrónicas aplicado a várias farmácias.

Apesar das dificuldades encontradas, pode afirmar-se que todos os requisitos do projeto foram concretizados com sucesso.

Em suma, o resultado conseguido foi bastante satisfatório uma vez que corresponde a uma representação bastante próxima da realidade.

9. Referências Bibliográficas

[1] BORGES, Luiz Eduardo. *Python para Desenvolvedores. 2ed. Rio de Janeiro, 2010*

[2]Site de documentação da linguagem de programação Python: http://www3.ifrn.edu.br/~jurandy/fdp/doc/aprenda-python/capitulo_10.html (Visitado a 5 de Novembro 2017)

[3]Site de documentação da linguagem de programação Python: http://www3.ifrn.edu.br/~jurandy/fdp/doc/aprenda-python/capitulo_08.html (Visitado a 5 de Novembro 2017)

[4] The Python Standard Library: https://docs.python.org/3/library/xmlrpc.client.html (Visitado a 8 de Novembro 2017)