Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

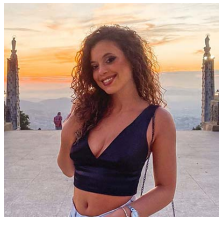
Unidade Curricular de Sistemas Distribuídos

Ano Letivo de 2021/2022

Sistema de gestão de reservas de voos

Grupo 12


62608 - Marco Sousa


93198 - Mariana Marques


93271 - José Malheiro


94269 - Miguel Fernandes

SD

15 de Janeiro de 2022

Resumo

A crescente procura por serviços cliente - servidor, obrigada a que um sistema seja capaz de responder a múltiplos pedidos de forma concorrente. Como consequência, surge a necessidade de moldar o sistema para que este tenha um comportamento determinístico, independentemente da circunstância de utilização. Assim, na sequência da Unidade Curricular de **Sistemas Distribuídos**, surge a oportunidade de desenvolver uma aplicação com múltiplos clientes e um servidor, ambos *multi-threaded*.

O projeto proposto corresponde a um sistema de reservas de voos, denominado *Flight Manager*. Este, conforme referido, será capaz de ter um servidor que responde a múltiplos clientes, de forma concorrente. A linguagem de programação utilização foi ***Java***, com recurso a *Socket* TCP para efetuar a comunicação entre cliente-servidor.

Área de Aplicação: Sistemas Distribuídos

Palavras-Chave: Sistemas Distribuídos, *Threads*, *Multithreaded*, *Middleware*, Servidor, Cliente, Serialização, *Data Transfer Object*, *Sockets*, TCP, *Tagged Connection*

1 Introdução

O presente relatório foi escrito no âmbito da Unidade Curricular de Sistemas Distribuídos (SD), sendo o tendo como principal objetivo apresentar uma possível solução para um Sistema de Gestão de Reservas, que iremos designar *Flight Manager*, capaz de responder às funcionalidades propostas pela equipa docente.

Assim, será apresentado um sistema *cliente-servidor* que, do lado do servidor, será capaz de atender múltiplos clientes e, do lado do cliente, será capaz de comunicar com o servidor e executar métodos assíncronos e síncronos.

Para dar resposta ao proposto, utilizou-se um componente comum a ambos os sistemas, um **Middleware**, que servirá de ponte de ligação entre os *hosts*. O objeto de comunicação, será um *Data Transfer Object* e, a partir deste, será derivado múltiplos objetos relacionados com pedidos e respostas.

Com o desenvolvimento deste projeto, pretende-se apromirar vários conceitos de Sistemas Distribuídos, tais como:

- Programação com *Sockets*
- Programação com múltiplas *Threads* (*MultiThread*)
- Programação Concorrente (estado partilhado)
- Exclusão Mútua
- Serialização de Objetos
- Arquitetura Cliente-Servidor
- Desenvolvimento de *Middleware*

2 Análise de Requisitos

O enunciado proposto, remete à conceção e implementação de uma plataforma para gestão de reservas de voos, apresentando um conjunto de requisitos funcionais que o sistema deve responder.

A partir do enunciado, identificou-se dois tipos de utilizadores: cliente e adminstrador. Cada um terá um conjunto de funcionalidades associadas, tendo sido captadas num Diagrama de Use Case (ver 1).

Assim, a nível estrutural, definiu-se três componentes principais:

Cliente Interface de interação com o utilizador para utilizar as funcionalidades definidas

Middleware Ponte de comunicação entre o cliente e o servidor

Servidor Responsável pelo tratamento dos pedidos do cliente

Definida a estrutura e as funcionalidades, passou-se para a modelação comportamental do sistema.

2.1 Modelação Comportamental

Por forma a permitir uma implementação sustentada, optou-se por efetuar a modelação comportamental do sistema. Para isso, começou-se por definir a forma como o cliente e o servidor iriam comunicar. Esta revelou-se como uma das etapas fundamentais para o funcionamento da aplicação, tendo como perspetiva a sua escalabilidade e manutenção.

2.1.1 Comunicação

A troca de mensagens entre o cliente e o servidor será mediada por um **Middleware** (ver 3). Pode-se encontrar um esboço da estratégia analisando o diagrama 2.

2.1.2 Middleware

No *middleware* (ver 3), haverá especialização do lado do cliente, por forma a receber respostas e enviar pedidos e do lado do servidor, para receber pedidos e enviar respostas. Esta especialização, serve apenas para facilitar a implementação, pois, na realidade, o objeto de comunicação será sempre o mesmo: uma `Frame` que terá no seu corpo informação sobre o pedido, nomeadamente uma *tag* que a identifica e um *Data Transfer Object* (DTO). Este último, será um dos principais atores na troca de informação, pois a partir da sua especialização, será possível criar cada um dos pedidos e cada uma das respostas, conforme o conteúdo necessário. Com o objetivo de generalização do código, colocou-se que a classe *DTO* será abstrata e terá como método, também este abstrato, a serialização (*serialize*).

2.1.3 Servidor

Dotado de um *ServerSocket*, este será capaz de estabelecer ligação com cada cliente, sendo-lhe atribuída uma *Thread* e *Socket* própria, partilhando um estado comum com arquitetura **Facade**: `IFlightManager` (ver 4). Neste, poderá identificar-se dois subsistemas: *Booking Manager* e *User Manager*.

Sendo um servidor *multi-threaded*, é necessário que haja controlo de concorrência. Assim, pode-se verificar junto dos diagramas que serão referenciados e na secção de implementação as estratégias utilizadas.

Utilizou-se os princípios *SOLID* [1] ao longo da implementação de todo o código, assim como o seu planeamento.

2.1.4 Booking Manager

Lógica relacionada com os processos de reserva de vôos, tais como os vôos diários disponíveis no sistema, uma abstração do dia (com os respetivos vôos associados) e as reservas propriamente ditas. Ver 5

2.1.5 User Manager

Lógica relacionada com a gestão dos utilizados. Tal como referido, existe dois tipos: *Administrador* e *Cliente*. Para permitir a escalabilidade do sistema, ambos especializam uma classe abstrata designada **Utilizador**. Ver 6

2.1.6 Flight Manager

Conforme referido, foi utilizada uma estratégia *Facade* para o desenvolvimento do estado partilhado no **Servidor**. Assim, a classe agregadora designa-se *FlightMinegerFacade* que implementa a API definida pela interface *IFlightManager*. Como é habitual neste tipo de estratégias, esta classe permite conjugar métodos relacionados com o *Booking Manager* e o *User Manager*.

2.1.7 Cliente

Em termos estruturais, o cliente apresenta uma lógica bastante mais simples. No entanto, por forma a implementar algumas funcionalidades adicionais, nomeadamente a possibilidade de efetuar uma reserva sem que se tenha de esperar pela sua

conclusão, houve necessidade de implementar um *ClientManager* (2.1.8).

2.1.8 ClientManager

Classe que permite ao cliente que este seja *multi-threaded* mas apenas ao nível dos pedidos, com o objetivo de serem não bloqueantes. Quer isto dizer que, apesar de haver uma *thread* principal a tratar do interface e da lógica do cliente, este pode utilizar métodos assincronos, que são executados por outras *threads* que lhe ficam associadas.

3 Implementação

Devido ao curto número de páginas definido pela equipa docente, será apenas abordado alguns pontos considerados fulcrais na implementação do sistema proposto.

4 Middleware

Tanto o cliente como o servidor utilizam a mesma classe base para gerir a ligação, *TaggedConnection*. Cada mensagem foi encapsulada numa *Frame* que é identificada por uma *tag*. Tem, ainda, o nome da classe que está a transportar e a própria classe. Desta forma, permite-te que se faça a *deserialization* de forma bastante simples e elegante. Para isso, foi introduzido um `Map<String, Class<? extends DTO>>` que mapeia o nome da classe para o seu tipo concreto com entradas do tipo `entry(UnitDTO.class.getSimpleName(), UnitDTO.class)`. Depois desta etapa, obtém-se o método *deserialize* da classe que, conforme definido, é implementado por todas as `Class<? extends DTO>`. Assim, a leitura do *Socket* é efetuada em 3 etapas:

1. Ler o nome da classe
2. Mapear o nome para o tipo concreto
3. Invocar p método *deserialize* adequado

```
Class<? extends DTO> p = getMapping(cName);
Method m = p.getMethod("deserialize",
↪  DataInputStream.class);
DTO dto = (DTO) m.invoke(null, in);
return new Frame(tag, dto);
```

Por forma a melhorar a eficiência do cliente, foi desenvolvido um *Demultiplexer* que corresponde a ter uma *Thread* em execução apenas para efetuar a leitura da *Socket*, efetuar o *deserialize* adequado e adicionar à *Entry* correspondente, identificada pela **tag**. Posteriormente, a *Thread* que estava à espera é notificada para que possa ler a partir da *Queue* de **DTO** associado.

```
final Condition cond = mLock.newCondition();
final ArrayDeque<DTO> queue = new ArrayDeque<>();
int waiters = 0;
```

Ainda a este nível, foi implementado uma classe que especializa a *Thread*: `class QueryThread`. Pretende-se que esta permita uma chamada assíncrona do método. Para isso, o construtor recebe um **DTO** (*Query*), a *tag* e um apontador para o *Demultiplexer*: `QueryThread(DTO queryDTO, int tag, Demultiplexer c)`. Ao utilizar esta classe, permite que um pedido ao servidor seja feito por uma *Thread* especifica para o efeito. Caso pretenda esperar pelo resultado, utilizará o método `public DTO result()`. Se pretender obter o resultado (caso já esteja disponível), utilizará o método `public DTO getResponse()`. Com esta estratégia, foi possível implementar a funcionalidade adicional proposta que corresponde ao paralelismo na execução de pedidos, de uma forma generalizada e escalável.

5 Cliente

Posto a implementação definida no ponto anterior, procedeu-se à implementação do *interface* gráfico de interação com o utilizador. Atendendo que este não era o foco do projeto, implementou-se uma versão estritamente essencial. Assim, pode-se encontrar um sistema de menus que foi representado numa máquina de estados (ver 7).

Para lidar com os pedidos, introduziu-se dois métodos principais: (i) `public QueryThread asyncHandler(DTO dto)` e (ii) `public DTO queryHandler(DTO dto)`.

(i) Coloca uma nova *thread* em execução, ficando esta a aguardar pela resposta. Adiciona-a, ainda, ao **ClientManager** (2.1.8) para ficar acessível ao cliente.

(ii) Utiliza o método (i) e fica a aguardar pelo resultado através do método *result()* que efetua uma espera passiva pela conclusão da *thread*. Corresponde à transformação de um método assincrono em um síncrono.

Para além destes, existe ainda um método específico para lidar com o login, pois tem a particularidade de definir no cliente o **token** que recebeu do servidor e terá de utilizar nas comunicações futuras.

A estratégia utilizada para pedir algo ao servidor corresponde, portanto, à sequência de 3 etapas:

1. Construção do *DTO* associado ao pedido, pedindo ao cliente os parâmetros necessários através do *Stdin*;
2. Utilização do método síncrono ou assíncrono (queryHandler ou asyncHandler).
3. Leitura e apresentação da resposta do servidor, em conformidade com o recebido

Como exemplo, pode-se verificar o menu **reserva** em que é efetuado um pedido de forma assincrona (apenas a componente de pedir a query):

```
BookFlightQueryDTO q = new BookFlightQueryDTO(params,
↪  min, max);
this.c.asyncHandler(q);
System.out.println("O seu pedido foi efetuado.");
System.out.println("Pode acompanhar no menu de
↪  pendentes!");
```

Neste excerto de código, pode-se verificar que não é esperado a resposta do pedido efetuado. O sistema remete o cliente para um menu próprio onde pode consultar pedidos (de reservas) que estejam concluídos.

6 Servidor

Por ser um servidor *multi-threaded*, a sua implementação foi mais complexa, por forma a evitar *dead-locks*, alteração de estado a meio de transações, entre outros. Assim, houve a necessidade de implementar vários *Locks* em níveis diferentes da estrutura e para controlo de diferentes estados.

Portanto, o servidor segue a seguinte sequência de acontecimentos:

1. Efetua a leitura do estado (FlightMinegerFacade) a partir de um ficheiro de objetos;
2. Inicia um *Logger*
3. Injeta esse estado numa classe que estará à escuta de uma *ServerSocket* na porta 4444;
4. Inicia a *thread* paralela
5. Coloca a *thread* principal à escuta do *System.in*

Começando pelo último ponto, é permitido que o utilizador termine o servidor a qualquer altura. Esta terminação é efetuada de forma eloquente, i.e. começa por fechar a socket de entrada no servidor (não aceitando novas ligações), mas permite que os clientes ligados terminem as suas transações. Apenas quando todos os clientes fecharem a conexão, o servidor encerrará e gravará o seu estado atual num ficheiro de objetos.

Sempre que um cliente se estabelece ligação com um servidor, é criado uma nova *thread* designada `class ServerWorker` que terá acesso ao modelo (estado partilhado), à sua *Socket* e a uma `class ThreadHandler` (partilhada).

Começando pela *ThreadHandler*, permite a gestão dos clientes que estão ligados ao servidor. O método *run()* está em espera passiva e sempre que há uma nova ligação, percorre a sua lista de *threads* e remove as que já não estiverem ativas. Esta classe é utilizada, também, quando se pretende encerrar o servidor, pois será esta que ficará à espera que todos os clientes terminem a sua transação.

O *ServerWorker* permite que cada *thread* que estabeleceu ligação lide com os pedidos do cliente. Tal como anteriormente, utilizou-se uma estratégia semelhante para lidar com a conversão do **DTO** abstrato para o especifico enviado pelo cliente. Utiliza-se um `Map<String, Function<QueryDTO, DTO>>` que permite, a partir do nome da classe, chamar o método que efetuará o tratamento do pedido. As entradas do mapa são do tipo:

```
entry(LoginQueryDTO.class.getSimpleName(), (x) ->
↪  this.loginHandler(x))
```

Desta forma, sempre que o servidor recebe um pedido, para além de escrever um *Log* que permite ter um histórico de pedidos recebidos, é capaz de o converter num resultado e responder ao cliente aplicando o método obtido no mapeamento:

```
public void requestHandler(Frame f)  {
        QueryDTO dto = (QueryDTO) f.getDto();
        String method =
        ↪  dto.getClass().getSimpleName();
        DTO r = getMapping(method).apply(dto);
        this.c.send(new Frame(f.tag, r));
}
```

Cada um dos métodos utiliza a API definida como *IFlightManager* que é implementada pela `class FlightManagerFacade`. Consequentemente ao servidor aceitar pedidos de múltiplos clientes em simultâneo, i.e. haverá concorrência no acesso aos recursos disponibilizados Portanto, houve a necessidade de definir variáveis com acesso crítico, assim como regiões críticas. Para acesso a estas regiões, utilizou-se a classe *ReentrantLock* garantindo o acesso *single-threaded*. Ao implementar a lógica relacionada com a gestão de reservas e utilizadores, identificou-se dois padrões: lógica relacionada apenas com um dos sub-sistemas (**Utilizadores** ou **Reservas**) e lógica relacionada com métodos que têm dependencias em ambos.

Em ambas as situações, procurou-se utilizar os **locks** de nível superior durante o menor tempo possível. Assim, começa-se por obter o **lock** para variáveis gerais e, posteriormente, obtém-se o **lock** da classe específica, procedendo à libertação do **lock** anteriormente adquirido. Um exemplo da estratégia utilizada:

```
this.l.lock();
BookingDay bd;
try {
        bd = this.getBookingDay(LocalDate.now());
        bd.l.lock();
} finally {
        this.l.unlock();
}
try {
        bd.closeDay();
        return true;
} finally {
        bd.l.unlock();
}
```

No caso de soluções que dependam de ambos os sub-sistemas, houve necessidade de ser o *Facade* a obter o **lock** da estrutura de um sistema e, posteriormente, executar o método pretendido no outro sistema. No final, liberta o **lock** adquirido. Um exemplo de métodos que dependem de ambos os sistemas:

```
if (users.hasUser(user)) {
        User u = users.getUser(user);
        u.lock.lock();
        String bookId;
        try {
                bookId = booking.bookFlight(user,
                ↪  percurso, de, ate);
                users.addReserva(user, bookId);
        } finally {
                u.lock.unlock();
        }
        return bookId;
}
throw new UserNaoExistente(user);
```

7 Funcionalidade Adicional

Introduziu-se a estratégia de travessia de grafos, permitindo a construção de caminhos entre dois destinos. Neste caso, o caminho foi limitado (conforme indicado no enunciado) a um máximo de 2 paragens, correspondendo a 4 cidades (A→B →C →D). A estrutura de suporte ao grafo é `Map<String, List<String>>` que permite, para uma dada origem, saber a lista de destinos possíveis.

Sempre que é adicionado um novo destino (vôo), este é adicionado ao grafo. Para garantir o acesso concorrente ao grafo, sem afetar a eficiência do servidor, optou-se por lhe alocar um **lock** específico.

```
g.lock();
try {
        if (grafo.containsKey(v.getOrigem())) {
                List<String> arestas =
                ↪  grafo.get(v.getOrigem());
                arestas.add(v.getDestino());
                grafo.put(v.getOrigem(), arestas);
        } else {
                List<String> arestas = new
                ↪  ArrayList<>();
                arestas.add(v.getDestino());
                grafo.put(v.getOrigem(), arestas);
        }
} finally {
        g.unlock();
}
```

Por último, conforme já abordado no Cliente (ver 5), implementou-se a possibilidade deste realizar pedidos que fiquem em execução em segundo plano, podendo, desta forma, continuar a trabalhar na aplicação sem esperar pela sua conclusão.

8 Considerações Finais

8.1 Conclusões

Com o objetivo de facilitar a implementação do sistema, começou por se efetuar a modelação funcional, estrutural e comportamental com recurso à ferramenta *Visual Paradigm*. Ao utilizar esta estratégia, permitiu ter uma visão mais abstrata do problema, levando a uma solução mais estruturada, escalável e sustentável.

Uma das principais dificuldades durante a implementação da solução, deveu-se à decisão de quando e como utilizar os *locks*. Para tal, conforme referido, começou por se identificar regiões críticas e fez-se um processo de implementação iterativa, i.e. começou por se introduzir *locks* ao nível da raíz e procedendo à sua especialização.

Não obstante, a dificuldade indicada, procurou-se que a implementação fosse o mais simples e generalizável possível. Para isso, procedeu-se à introdução de um Middleware cujo objeto de comunicação fosse facilmente identificável. Este requisito levou à introdução de uma *Frame* que encapsula um **DTO**, podendo ser especializado como uma *query* ou uma resposta.

Ao nível de controlo de concorrência, pode-se acrescentar que se procurou obedecer às regras lecionadas ao longo da UC de Sistemas Distribuídos, nomeadamente através da aquisição de *lock* de forma ordenada, assim como a respetiva libertação. Esta estratégia foi implementada quando se precisava de adquirir controlo sobre várias estruturas em simultâneo.

8.2 Trabalho Futuro

Com a perspetiva de melhorar a interação com o cliente, seria uma mais valia fazer uma restruturação do *User Interface*. Especificamente, através de uma *interface* mais atrativa e utilizável. Mediante a nova UI, seria possível que os pedidos que fossem efetuados de forma assincrona, apresentassem o seu resultado com uma *pop-up* ou algum tipo de aviso que não interfira diretamente com o que o cliente se encontra a fazer. Por exemplo, no sistema desenvolvido, caso os pedidos assincronos fossem apresentados mal terminassem, podiam intercalar com um *output* que estive a ser exibido.

Poderia, ainda, introduzir-se persistência através de utilização de **Bases de Dados**. Tal funcionalidade não foi implementada, pelo facto dos elementos do grupo não terem conhecimentos sobre esse tipo de sistemas.

Para além destes, será necessário desenvolver mais testes para validar o funcionamento da aplicação.

O aperfeiçoamento da aplicação, de modo a chegar a um estado competitivo para o mercado atual, passaria pela implementação de várias e diferentes funcionalidades que lhe permitisse ser distinguido dos restantes concorrentes.

Referências

[1] *SOLID - Wikipedia*. URL: `https://en.wikipedia.org/wiki/SOLID`.

9 Anexos

9.1 Anexo 1 - Diagrama de Use Case

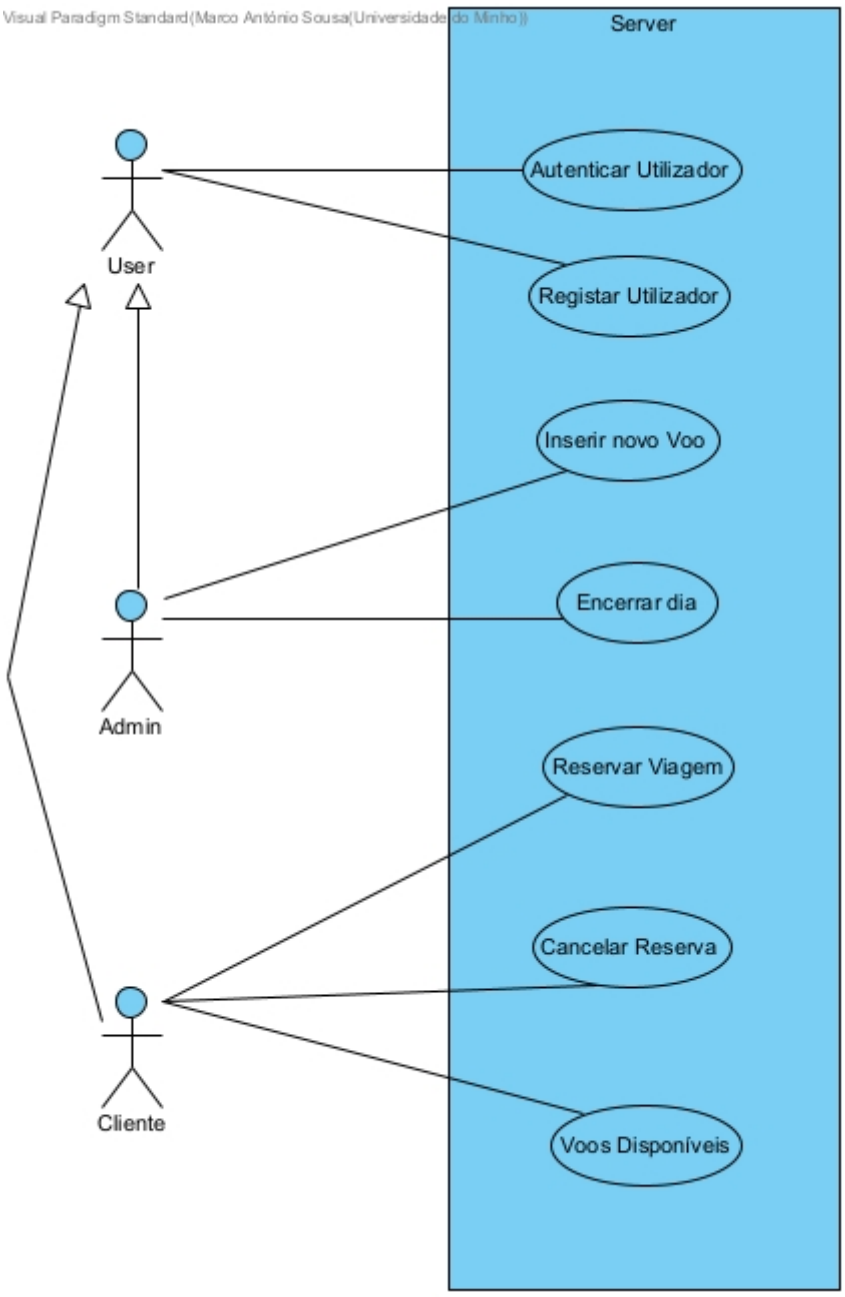


Figura 1: Diagrama de Use Case

9.2 Anexo 2 - Diagrama de Comunicação

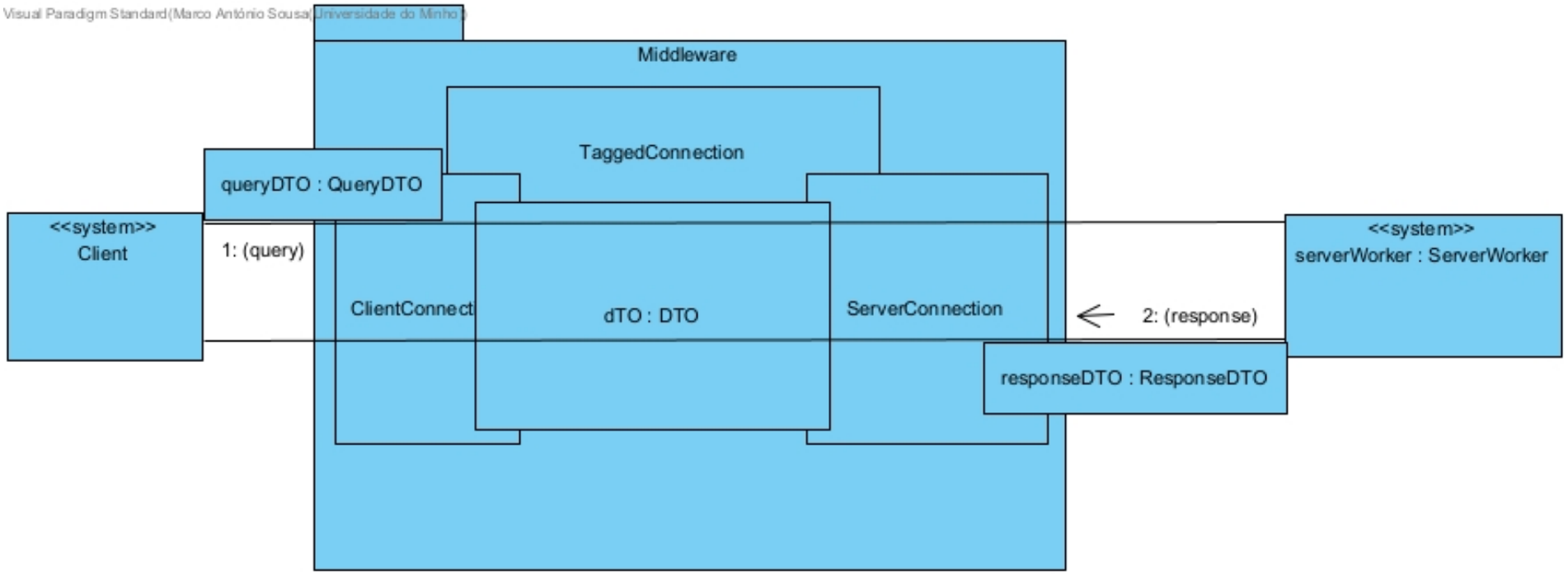


Figura 2: Diagrama de Comunicação - versão β

9.3 Anexo 3 - Diagrama de Classes do Middleware

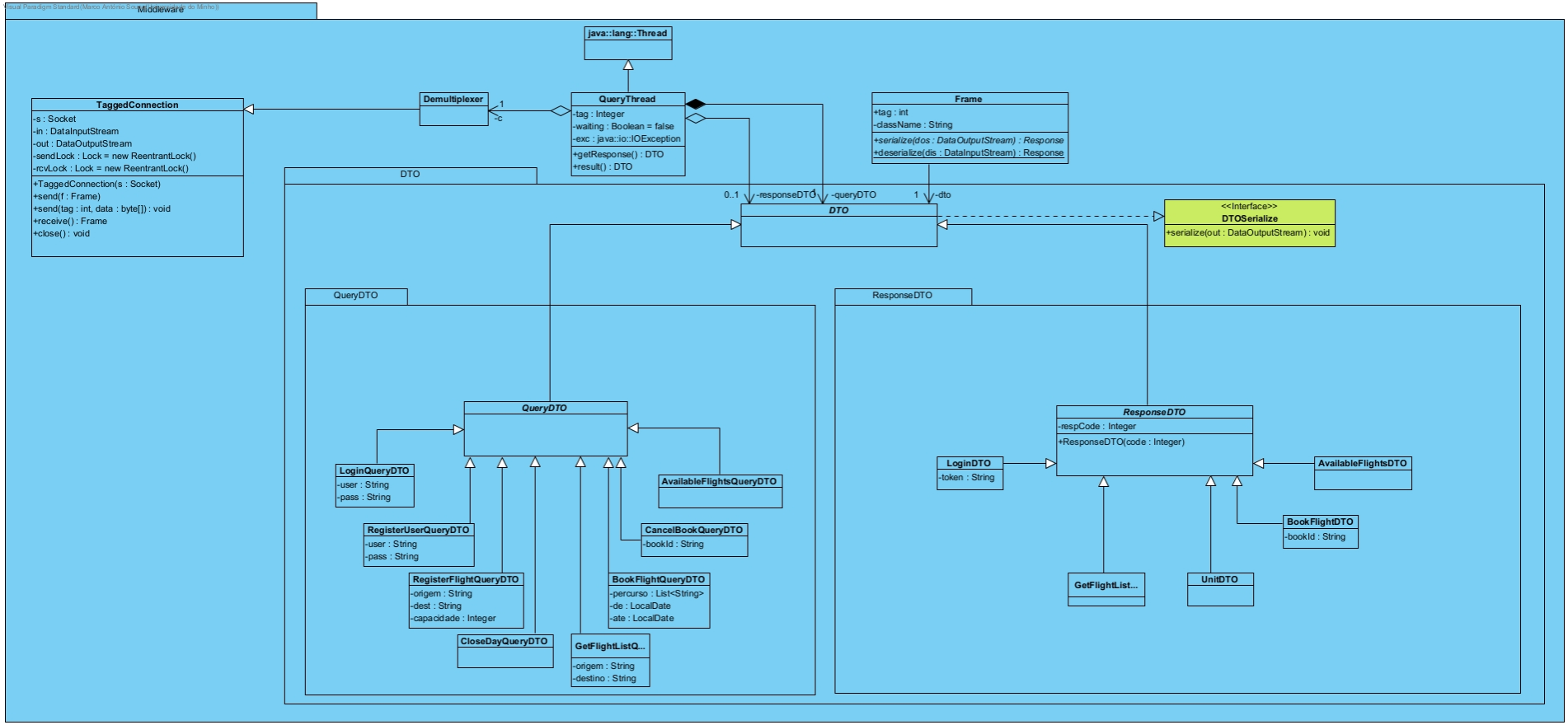


Figura 3: Diagrama de Classes - Middleware

9.4 Anexo 4 - Diagrama de Classes do FlightManager (Servidor)

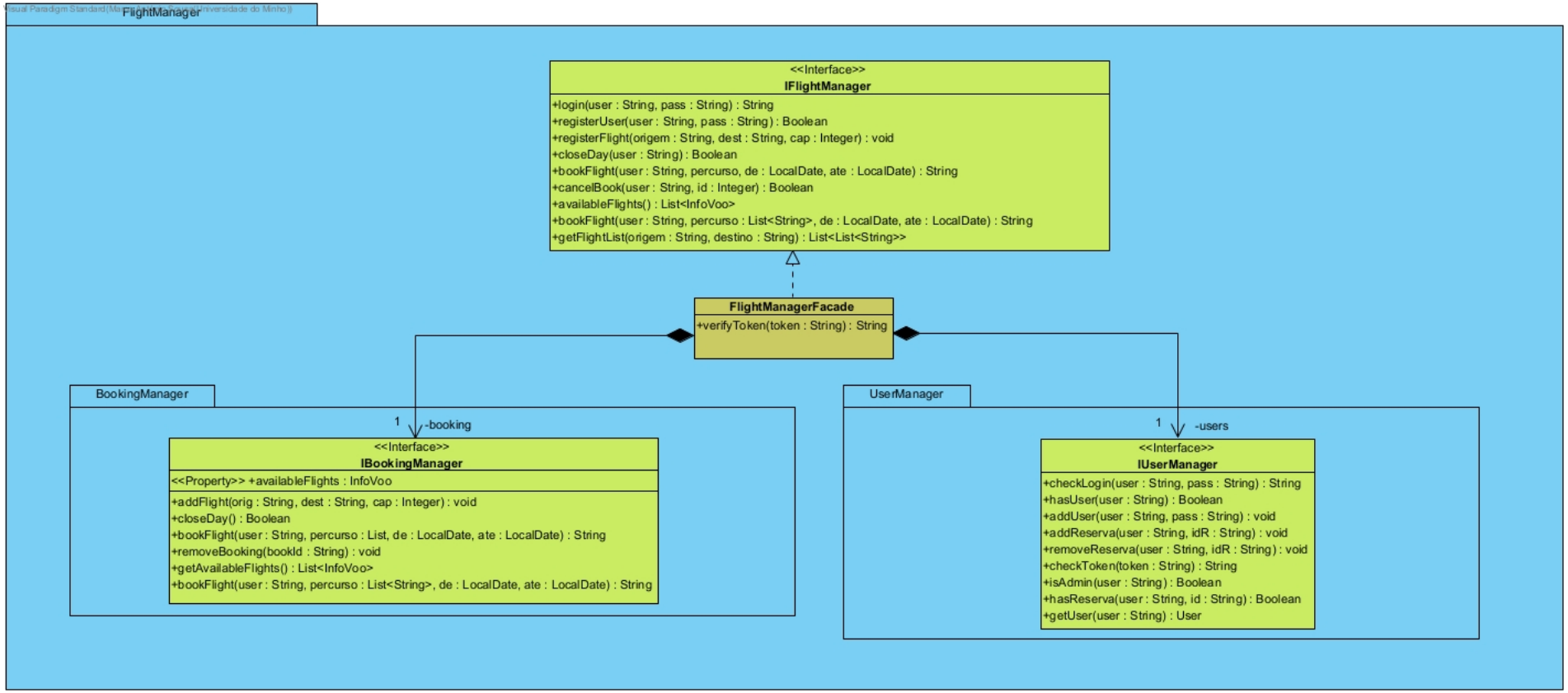


Figura 4: Diagrama de Classes - FlightManager

9.5 Anexo 5 - Diagrama de Classes do Booking Manager (Servidor)

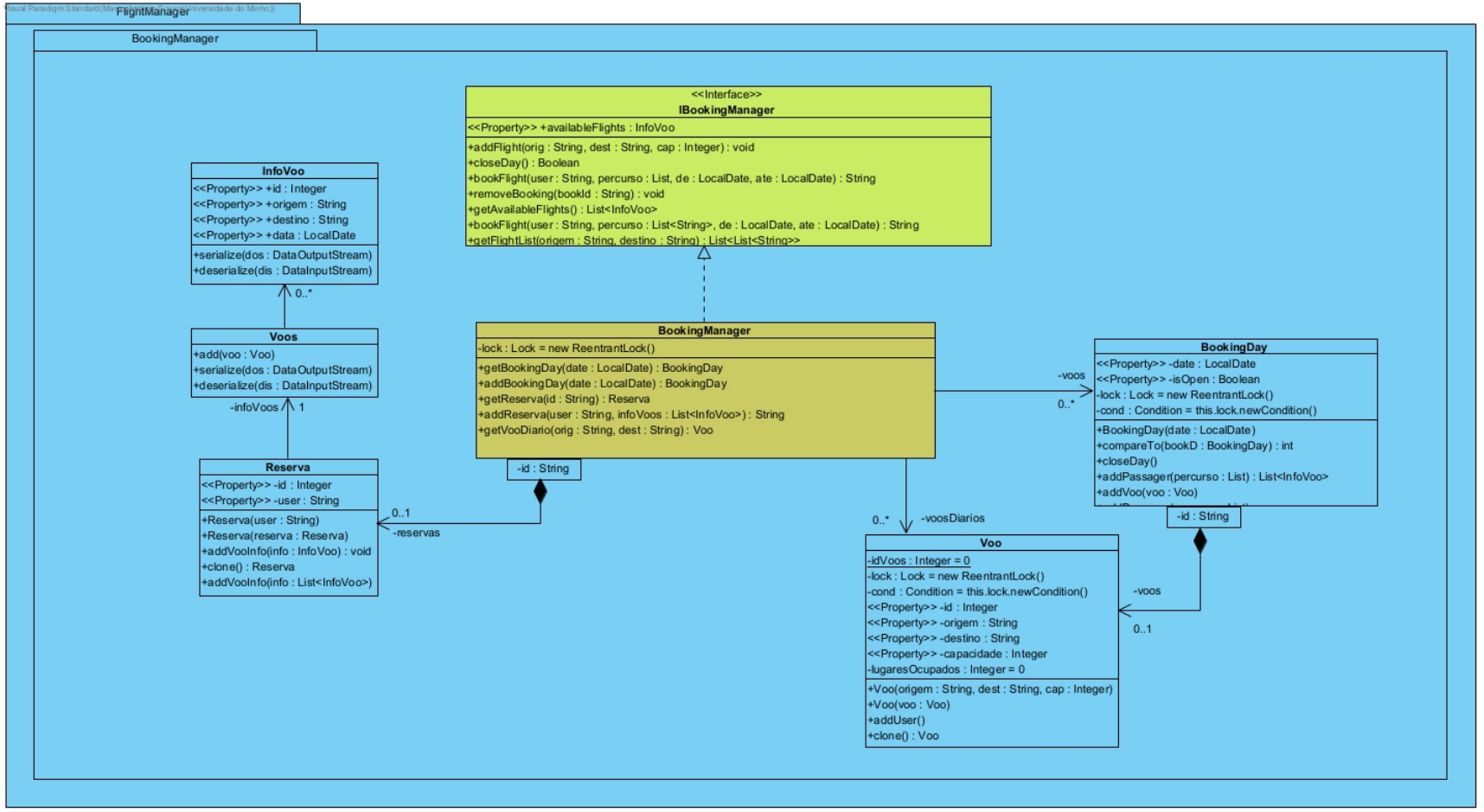


Figura 5: Diagrama de Classes - BookingManager

9.6 Anexo 6 - Diagrama de Classes do User Manager (Servidor)

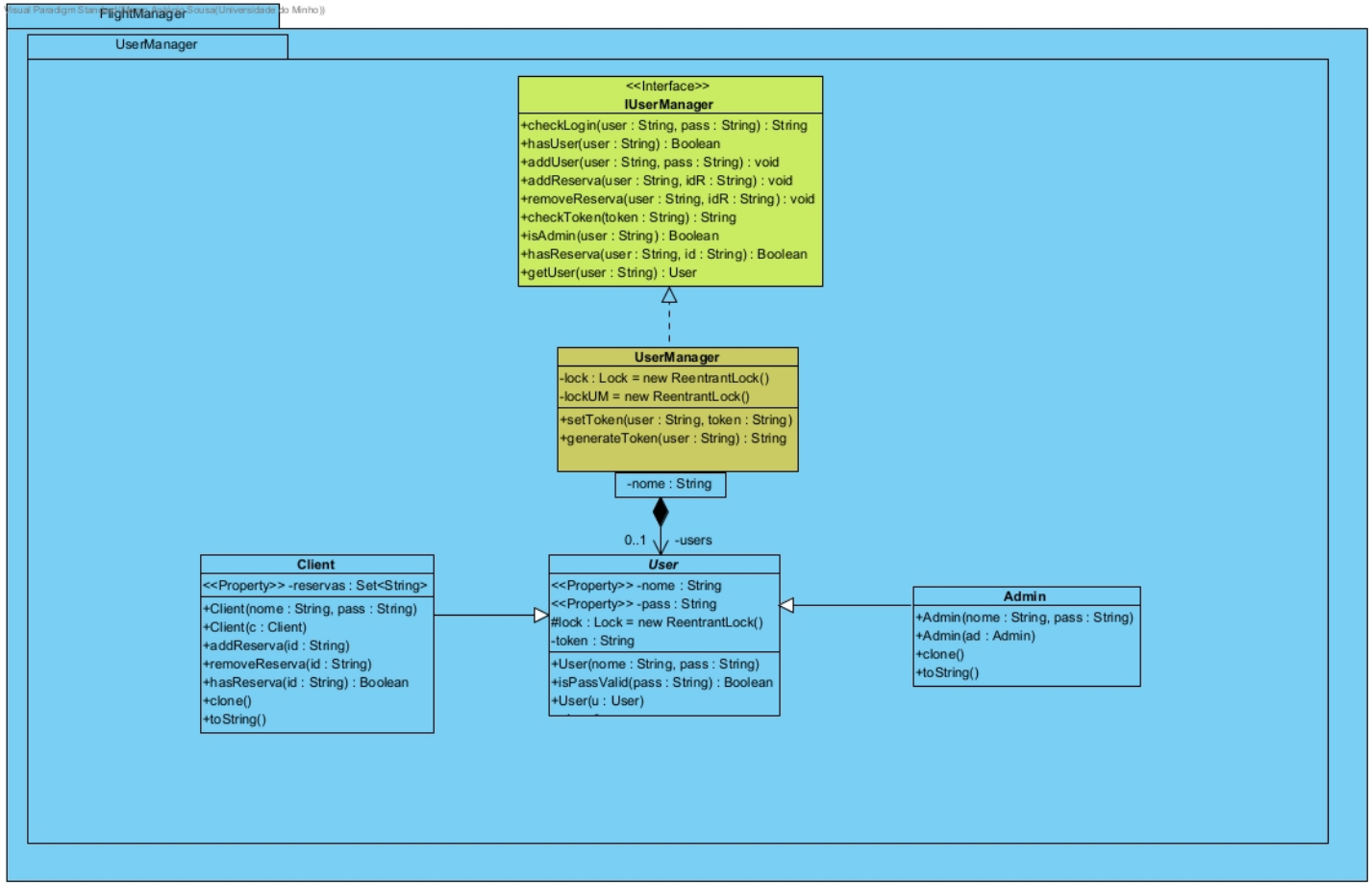


Figura 6: Diagrama de Classes - BookingManager

9.7 Anexo 7 - Diagrama Pseudo-Estado Menu de UI

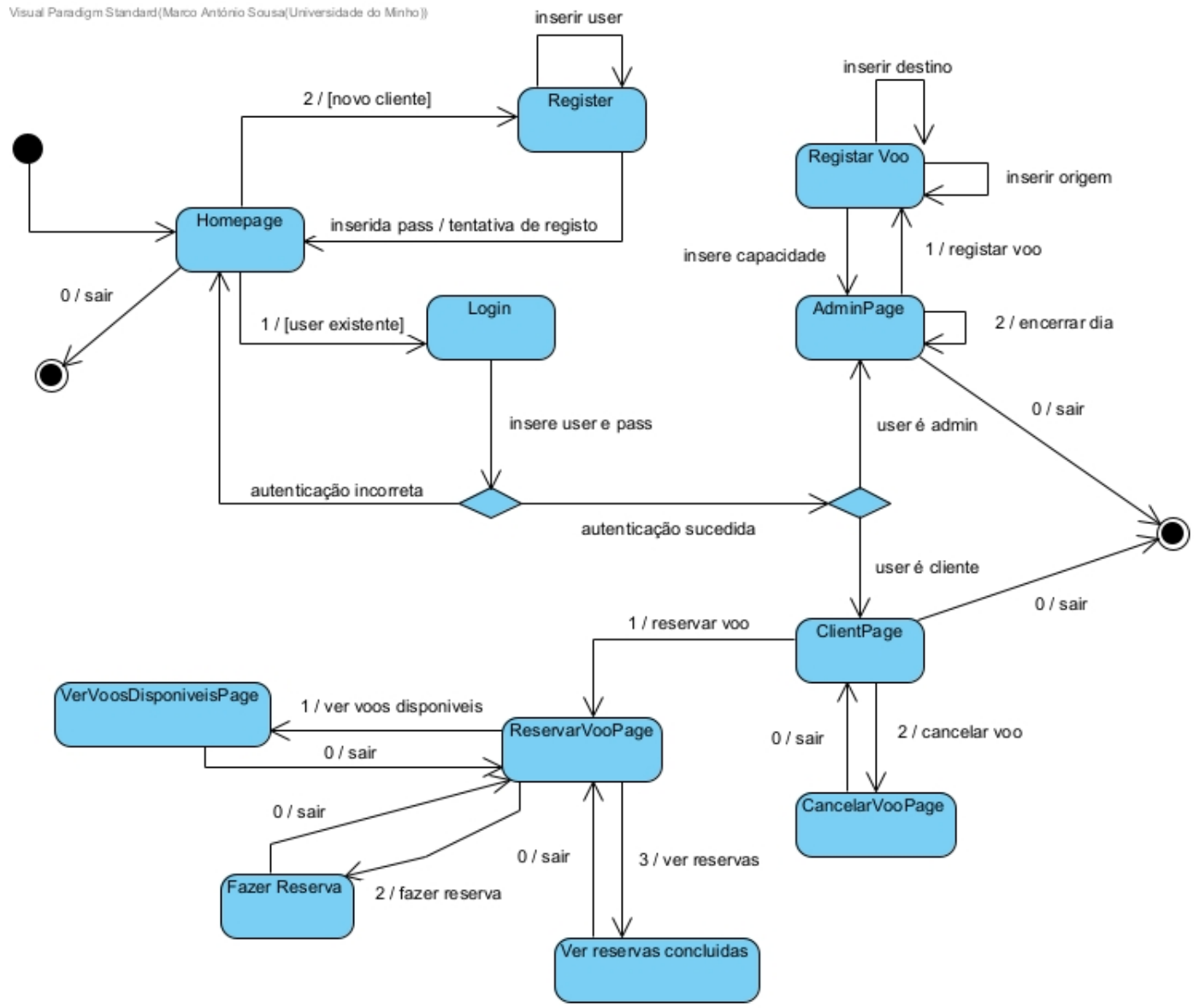


Figura 7: Diagrama de Estado - Sistema de Menu