Universidade do Minho
Escola de Engenharia

Paradigmas de Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática

Alarme Covid

[Trabalho Prático]

A84961 Alexandre Ferreira
A84462 Alexandre Miranda
A85227 João Azevedo
A85315 Miguel Cardoso
A85729 Paulo Araújo

Braga,
Janeiro 2021

Resumo

Este trabalho consiste na elaboração de uma plataforma para suporte de rastreio de contactos e deteção de concentração de pessoas. A plaforma consiste em vários componentes de *software*, desde uma aplicação **cliente** para interagir com o sistema, efetuando atualizações e interrogações de informação, uma componente *frontend* que permite não só efetuar a autenticação dos clientes como também criar um "túnel" para o fluxo de informação privada, como notificações de contágios e atualização da localização de um utilizador, e todo um *backend* composto por duas componentes principais: os servidores distritais que fazem a gestão dos mapas e contatos em distritos, e uma *API RESTful* para consulta de informação estatística.

Detalhe da solução

Nesta secção iremos detalhar, para as diferentes componentes da aplicação, as soluções adotadas, o que inclui a indicação ferramentas utilizadas e alguns diagramas de fluxo de informação/mensagens, de modo a clarificar toda a explicação apresentada.

Frontend

O *frontend* deste sistema distribuído consiste num servidor, escrito em *erlang*, que pretende prestar o tradicional serviço de autenticação e registo de utilizadores. Para além disto e após o processo anterior estar completo, os clientes podem agora enviar pedidos e receber as respetivas respostas, incluindo notificações espontâneas, segundo um encaminhamento "privado" de/para servidores distritais.

Assim, estas foram as funcionalidades e considerações tomadas no desenvolvimento desta componente:

- **Registo e Autenticação** de utilizadores: Quando algum cliente pretende interagir com o servidor distrital e, periodicamente, fornecer a sua localização geográfica, assim como ser notificado de um possível contágio, este deve se registar na aplicação, conectando-se ao servidor *frontend*, através de uma mensagem que segue a seguinte estrutura:

  ```
  Estrutura da mensagem : "create <username> <password> <distrito>"
  Exemplo               : "create Jose segredo Porto"
  ```

  O servidor *frontend*, de todas as vezes que é iniciado, faz a gestão de uma estrutura, em memória, que mapeia o nome de utilizador (único) para um conjunto de informações:

  ```
  key   = {username}
  value = {password, isLogged?, Distrito, TCPSocket?, isInfected?}
  ```

  Deste modo, após o registo, podemos exigir que o utilizador faça o *login*, podendo atualizar, através da autenticação, o valor das etiquetas **isLogged** e **TCPSocket** com, respetivamente, **true** e o **socket** (*gen_tcp:accept(ListenSocket)*) criado para interagir com o cliente[1].

  A autenticação é feita enviando uma mensagem com uma estrutura semelhante, excluindo a informação do distrito, visto já ter sido guardada aquando o registo:

[1] De notar que a razão de guardar informação deste *socket* será explicada mais à frente.

```
Estrutura da mensagem : "login <username> <password>"
Exemplo               : "login Jose segredo"
```

Toda esta gestão explicada anteriormente está implementada num módulo, em **erlang**, chamado *login_manager.erl* que implementa estas e outras funções auxiliares, chamadas de *rpcs* (*remote procedure calls*).

- **Envio de pedidos e receção de respostas** (e notificações privadas): Após uma autenticação ser feita com sucesso, o novo processo criado para gerir o utilizador tem agora um conjunto de tarefas a realizar:

  1. Fazer o pedido, a um servidor central, de um "endereço[2]" onde o servidor distrital está a correr e, em caso de sucesso, conectar-se a este;

     ```
     Tipo de comunicação : REQ - REP
     Pedido              : "cliente_<username>_<distrito>"
     Resposta            : "centralserver_<ok|error>_<district-router>"
     ```

  2. Receber pedidos do cliente e replicá-los para o respetivo servidor distrital, segundo uma comunicação *request-router* podendo, de seguida, ler as respostas e enviá-las de volta para o cliente respetivo.
  3. Receber notificações privadas através de um servidor *zeromq* do tipo *pull* e posterior filtragem da notificação e encaminhamento para o cliente ao qual a mesma pertence:

     ```
     1. Ler Notificação no Socket PULL : <user>_<corpo-notificação>
     2. Pedir ao login manager : login_manager:get_socket(<user>)
     3. Enviar notificacao : socket:send(<user>_<corpo-notificação>)
     ```

A imagem que vamos apresentar de seguida consiste numa secção de um diagrama principal, anexado (7) a este relatório, que contém todo o sistema representado, no entanto, com vista a facilitar a compreensão das diferentes componentes do sistema, vamos "reutilizando" o mesmo para complementar as explicações.

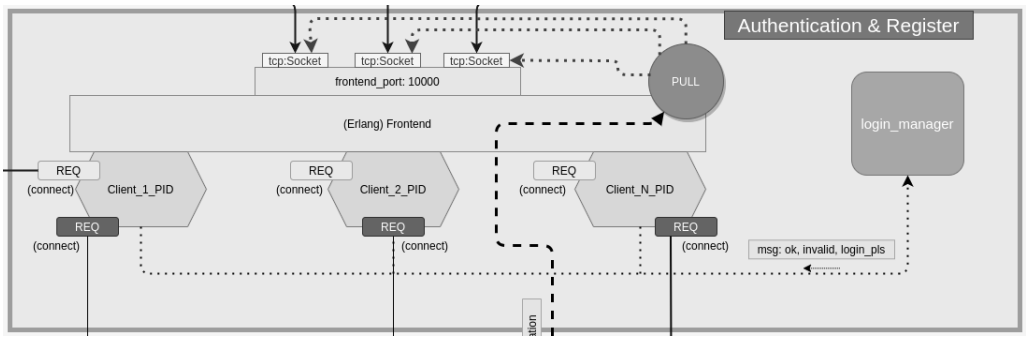


Figura 1: Servidor de registo e autenticação (*frontend*).

[2]Todos os serviços desta aplicação correm num ambiente *localhost*, por isso a identificação dos diferentes componentes é dada pela identificação da **porta** onde corre a aplicação.

Servidor Distrital

Os servidores distritais, como dito no enunciado, mantém a informação enviada pelo *frontend*, nomeadamente, sobre a localização dos utilizadores do distrito. A persistência da informação na memória deste servidor é feita mantendo um registo de contactos, ou seja, quando o utilizador vai para uma certa posição, de uma matriz NxN, que representa o distrito, são adicionados aos contactos todos os utilizadores que se encontram naquela posição. De forma geral, este serviço, replicado 18 vezes para cada distrito, tem as seguintes funcionalidades:

- **Manter, em memória, estruturas para gerir um mapa NxN:** Existem duas estruturas principais consideradas no desenvolvimento desta aplicação: por um lado, temos uma matriz NxN que contém, em cada posição, uma lista (um conjunto, sem repetidos), de utilizadores que se encontram naquela posição; por outro lado, temos um mapeamento entre um utilizador e o conjunto de contatos registados para todas as posições que este passou:

  ```
  Estrutura 1: UtilizadoresNaPosicao[N][N] matrizGlobal;
  //...onde UtilizadoresNaPosicao é um HashSet<String>

  Estrutura 2: Map<User, HashSet<String>> ContactoUtilizadores;
  //...onde User é um objeto que contém o <username> e a <posicao>
  ```

  Por fim, é mantida uma estrutura que permite, por distrito, gerir um *top* 5 de localizações (pontos X, Y) e o respetivo recorde de pessoas naquela posição:

  ```
  Estrutura 3: TreeSet<Record> top5Positions;
  //...onde Record é um par <Posicao, ValorRecorde>
  ```

- **Processamento dos pedidos dos clientes:** Existe um conjunto de pedidos possíveis a efetuar ao *backend*, pelo que este servidor tem de processá-los, alterar as estruturas anteriores e atualizar o servidor **Diretorio**, que irá ser explicado mais à frente. Aqui iremos delinear todos os pedidos possíveis:

  1. Registo de utilizadores no *backend* e atualização de posições:
     (a) Quando um utilizador se autentica no *frontend* é automaticamente enviada uma mensagem de registo para o servidor distrital, onde se insere o utilizador na matriz global e registam-se todos os contactos (utilizadores naquela posição).

     `Pedido: <username>_registo_<posx>_<posy>`

     (b) Também devem ser registadas as alterações de posição e, novamente, registo de contactos, através da seguinte mensagem:

     `Pedido: <username>_track_<posx>_<posy>`

     (c) As notificações públicas ocorrem quando queremos transferir um utilizador para uma certa posição e podem ser as seguintes:
     - A quantidade de utilizadores na posição antiga é zero, então enviamos uma notificação pela rede de *brokers*, com o *type "no-users"*;
     - A quantidade de utilizadores, na posição antiga, é superior a um **limite** previamente definido, por distrito, então enviamos uma notificação que reporta excesso de concentração de pessoas, com o *type "too-many"*;
     - A quantidade de utilizadores, na posição destino, é inferior a um **limite** previamente definido, por distrito, então enviamos uma notificação que reporta uma diminuição de concentração de pessoas, com o *type "less-users"*;

O ***type*** referido anteriormente aplica-se à seguinte estrutura de notificação:
`Notificação: <distrito>_<type>_<posx>_<posy>`

2. Processamento de contactos e utilizadores infetados:
   - Reportar infetados é das funções principais deste sistema e dá-se através de um pedido com a seguinte estrutura:
     `Pedido: <username>_infected`
     Quando o servidor distrital recebe um pedido de registo de um infetado devem ser feitas duas coisas: envio de uma notificação privada para todos os contactos, através de um **push** para um servidor de **pull** (criado no *frontend*), e envio de uma notificação pública de que houve um infetado no distrito para todos os interessados (subscritores), através de uma rede pública de *brokers*. As notificações referidas anteriormente, seguem a seguinte estrutura:
     `Notificação de contacto (privada): <district>_infected`
     `Notificação de infetado (pública): <username>_got-contact`

3. Informação acerca das localizações do mapa:
   - Pode também ser pedido, por um dado utilizador num distrito, quantas pessoas se encontram numa certa posição:
     `Pedido: <username>_n-users-in-pos_<x>_<y>`

O diagrama que representa o fluxo de mensagens no servidor distrital e os serviços/conexões existentes é o seguinte:

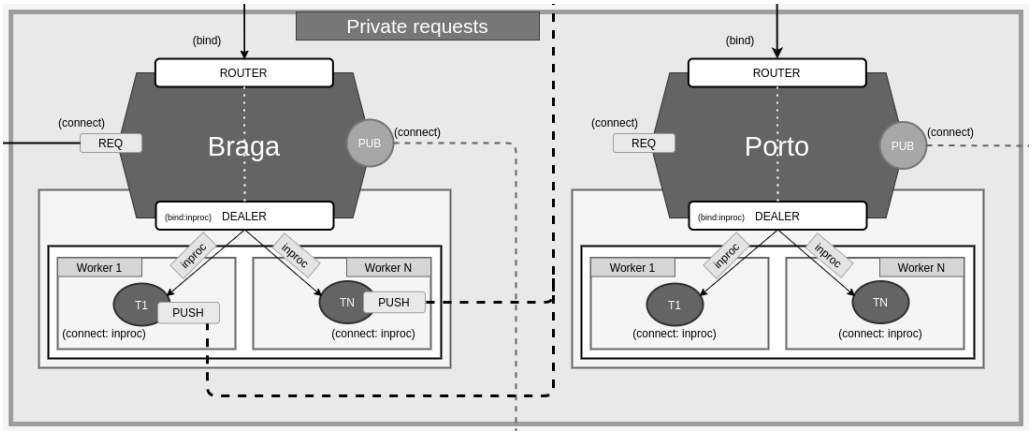


Figura 2: Servidor Distrital (*backend*).

Deste modo, podemos verificar que a solução adotada pretende priorizar uma experiência de utilização cliente-servidor segundo uma política *request-reply* para um servidor distrital *router* que divide o processamento dos pedidos, através de *round-robin*, para diferentes *workers* numa prespetiva de comunicação *inproc*, sendo o *Dealer* importante na distribuição de carga.

Redes de notificações públicas

A nossa solução tem em atenção que certas mensagens devem ser partilhadas segundo diferentes tipos de abordagens *zeromq* para priorizar, por um lado, a privacidade das informações partilhadas e, por outro lado, a rápida obtenção de informação pública. Neste sentido, desenvolvemos as seguintes componentes:

- Rede pública de *brokers*: Um dos requisitos desta aplicação é permitir a um cliente poder declarar interesse (subscrever) um ou mais distritos, ou seja, receber notificações públicas acerca dos mesmos. Com vista a implementar isso mesmo, decidimos introduzir, na nossa solução, o conceito de *broker* como servidor intermédio entre mensagens que ligam N servidores a N clientes.

  Na verdade, de modo a aumentar o *load balancing* de pedidos de publicação de notificações pelos servidores distritais, implementamos uma **primeira camada (*broker layer 1*)** com 2 (ou mais) servidores, que irão ligar subconjuntos de distritos, isto é, dividimos a receção das notificações pelos diferentes *brokers* da camada 1. Assim, desenvolvemos **uma segunda camada *(broker layer 2)*** que tem a função de espalhar as mensagens recebidas da camada 1 para os clientes, ou seja, cada *broker* da camada 2 liga-se a todos os *brokers* da camada 1 e, portanto, obtém todas as notificações partilhadas.

  Deste modo, cada cliente quando pretende subscrever um dado distrito, este subscreve, na verdade, um *broker* da camada 2[3].

Diretorio REST

Este servidor funciona como um recurso para consultas estatísticas, via *API REST*, acerca dos constantes registos feitos pelos servidores distritais e permite a vários clientes consultar esta informação sem interferir com o funcionamento do resto dos serviços da aplicação.

Em termos de ferramentas utilizadas, o servidor foi escrito em *Java*, utilizando a *framework dropwizard* e, após arquiteturar os recursos necessários e métodos a serem desenvolvidos, chegamos à seguinte estrutura:

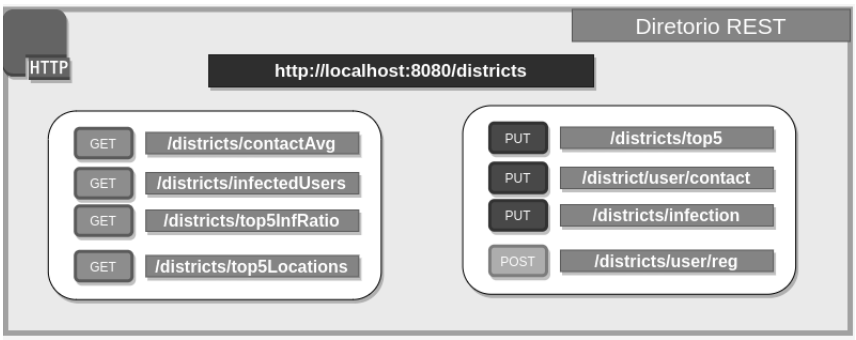


Figura 3: Recursos do diretório.

- Inserção e atualização de informação: É necessário, após cada atualização de posição dos utilizadores e pedidos de registo de infeções/contactos, que o diretório seja atualizado com essa informação, primariamente, mantida pelos servidores distritais. Logo, estes últimos são clientes *http* para esta API de modo a inserir/atualizar estes recursos. Neste sentido, desenvolvemos os seguintes métodos:

  1. **Exemplo de POST** */districts/user/reg*: Insere um novo utilizador, num dado distrito, para que mais tarde seja possível consultar o número de utilizadores por distrito. O *body* do pedido deve conter o seguinte objeto *JSON*:

     ```
     {"district": ..., "username": ...}
     ```

[3]Os *brokers* da camada 2 são atribuídos aos clientes através de uma política *round-robin*, feita pelo servidor central.

2. **Exemplo de PUT** */districts/top5*: Atualiza o *top5* distrital (em termos do número de utilizadores), através do seguinte *body*:

   ```
   {"district": ..., "record": ..., "positionX": ..., "positionY": ...}
   ```

- Consultas genéricas:

  1. **Exemplo 1 de GET** */districts/contactAvg*: Devolve o número médio de utilizadores que se cruzaram com utilizadores declarados doentes.
  2. **Exemplo 2 de GET** */districts/top5InfRatio*: Devolve a porção (em percentagem) de utilizadores infetados em relação ao número total de utilizadores por distrito.

Servidor Central, Configuração e aplicação Cliente

O Servidor Central foi uma adição proposta pelo nosso grupo para gerir todas as portas e a sua distribuição pelos clientes, servidores distritais e *brokers*, visto que, ao estarmos a falar num ambiente *localhost* torna-se difícil gerir a utilização dos endereços. Assim, por exemplo, quando um servidor distrital se conecta este faz um *request* ao central para obter as portas que necessárias ao seu funcionamento, o mesmo decorre para os clientes ou os próprios *brokers*.

Juntamente com este servidor, temos um ficheiro de configuração global **src/config.toml** que contém todos os *defaults* associados a este sistema.

Por fim, a aplicação cliente foi a componente mais trabalhosa, pois exigiu que existisse muito código relacionado com controlo de fluxo e de *input* do utilizador, além da criação da própria interface do mesmo. Temos os seguintes exemplos da GUI desenvolvida:

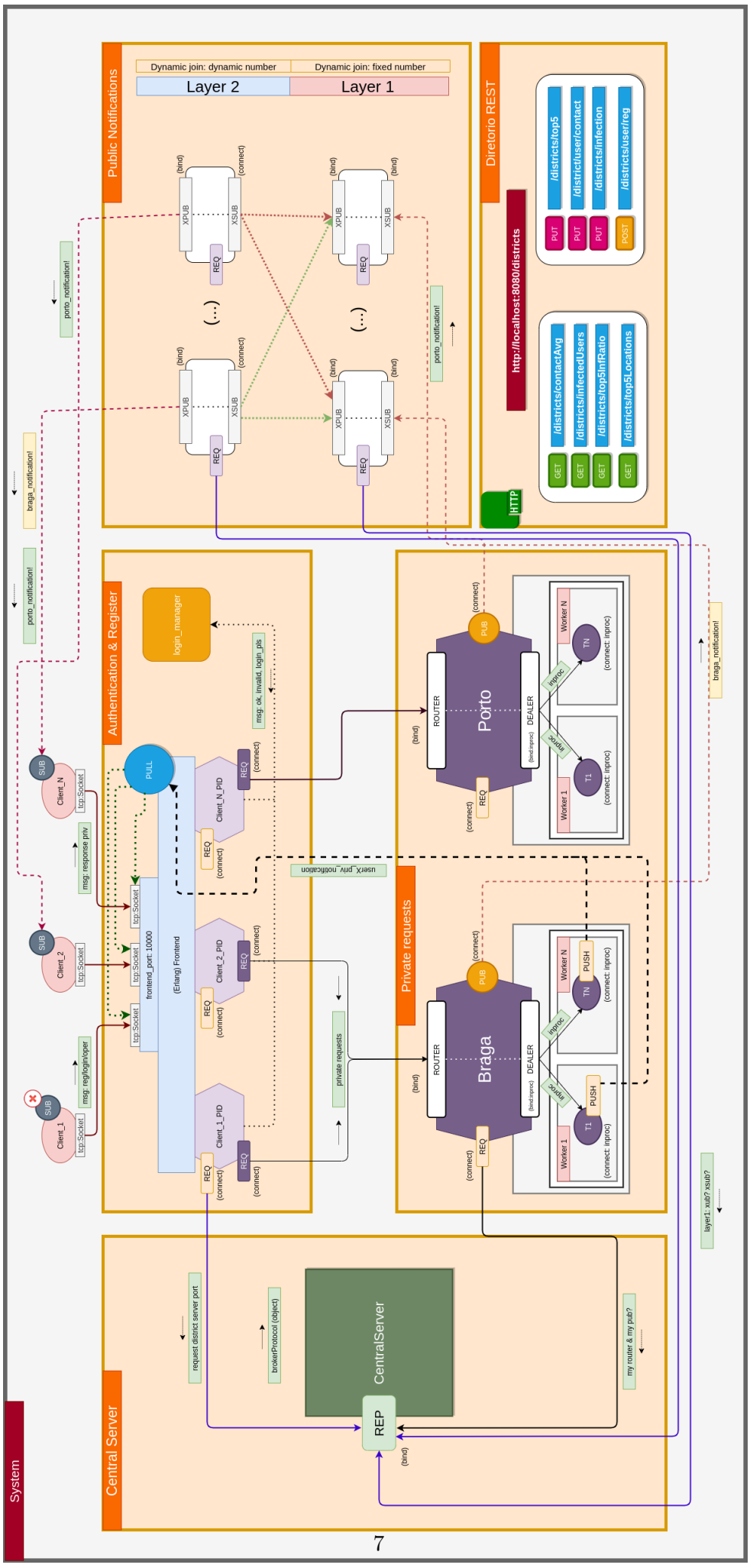
Figura 4: Menu inicial e para um utilizador infetado (com subscrição Porto e Braga).

Conclusão

Em suma, fazemos um balanço positivo do trabalho desenvolvido, isto é, pensamos ter atingido todos os objetivos estabelecidos para este TP, ou seja, todas as componentes foram desenvolvidas.

Quanto às principais dificuldades, estas passaram pela elaboração da rede privada de notificações, isto é, pensamos ter encontrado uma solução interessante mas não excluímos a hipótese de utilizar, no futuro, políticas *zeromq router-router* tanto no servidor distrital como no servidor *frontend* para que os *request-reply* possam ser feitos em ambos os sentidos.

Finalmente, pensamos que faltou um pouco de automatização de execução das componentes e avaliação de desempenho/*performance* ao nosso sistema para que possamos, por exemplo, concluir melhor acerca da rede de *brokers* pública no que toca ao número ótimo de servidores a utilizar.


System
Central Server
CentralServer
REP
(bind)
request district server port
brokerProtocol (object)
my router & my pub?
Authentication & Register
login_manager
msg: ok, invalid, login_pls
Client_N
Client_2
Client_1
SUB
tcp:Socket
msg: response priv
msg: reg/login/oper
PULL
frontend_port: 10000
(Erlang) Frontend
Client_N_PID
Client_2_PID
Client_1_PID
REQ
(connect)
user's_priv_notification
private requests
Private requests
Braga
Porto
ROUTER
DEALER
(bind)
(connect)
PUB
REQ
Worker N
Worker 1
TN
T1
inproc
(connect: inproc)
PUSH
Public Notifications
Dynamic join: dynamic number
Dynamic join: fixed number
Layer 2
Layer 1
XPUB
XSUB
(bind)
(connect)
(...)
porto_notification!
braga_notification!
layer1: xsub? xsub?
Diretorio REST
http://localhost:8080/districts
HTTP
/districts/top5
/districts/user/contact
/districts/infection
/districts/user/reg
PUT
POST
/districts/contactAvg
/districts/infectedUsers
/districts/top5InfRatio
/districts/top5Locations
GET

```
Execução simples (exemplo) do sistema:

    $ cd src/backend:

        1. Execução do servidor central:
            $ make run_central
        2. Execução da rede pública de layers:
            $ make run_test_layers
            Program arguments: 2 2
            //para executar 2 brokers da camada 1 e 2 brokers da camada 2

    $ cd src/rest/Diretorio

        3. Execução da API REST (Diretorio):
            $ make run

    $ cd src/frontend:

        4. Execução do frontend:
            $ make run
            Eshell V11.0.3  (abort with ^G)
            1> frontend:start_frontend().

    $ cd src/backend:

        5. Inserção de servidores distritais (individual):
            $ make run_district
            Program arguments: Porto

    $ cd src/client:

        6. Executar a aplicação cliente:
            $ make run

Outros notas:

    1. Execução/Inserção dinâmica de brokers:
        //Podem ser inicializados manualmente
        $ make run_layer1
        //ou make run_layer2 (dinâmicos, i.e, tendo todos os layer 1 definidos)

    2. Scripts para automatização:

        $ cd src/scripts/
        //Correr servidor central e api rest
        $ bash init-startup.sh
        //Correr os 18 distritos
        $ bash init-districts.sh

    3. Testes para a API REST:
        $ cd src/scripts/rest
        //contém testes para valores aleatórios e específicos
```