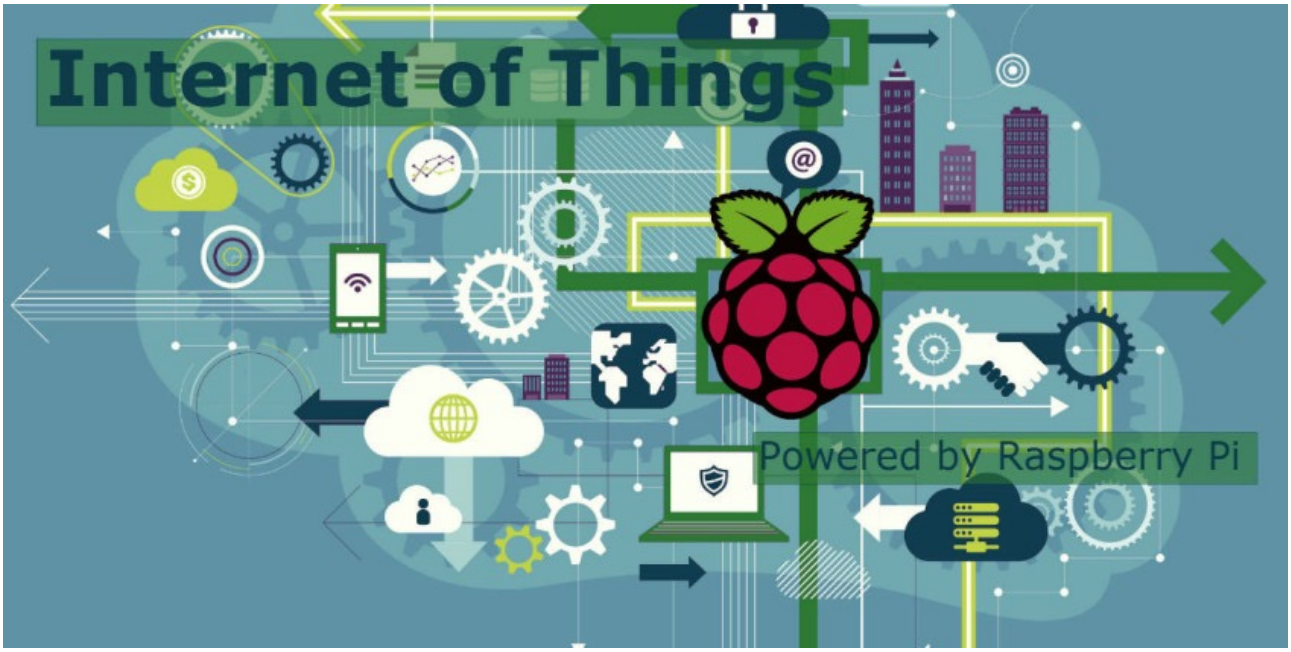
PUC
RIO

INF1822 Projeto Orientado

Renan Almeida 1211055



Construção de aplicações distribuídas com sensores e RaspberryPIs

Introdução

O projeto desenvolvido envolve o estudo de tecnologias que possibilitariam a criação de aplicações para casas com o tema IOT, através da utilização de RaspberryPIs conectados a sensores de movimento, temperatura e luminosidade.

Para o desenvolvimento da aplicação, planejamos estudar as possíveis formas de comunicação entre os RaspberryPIs. Mais especificamente, qual seria a melhor escolha de arquitetura para um sistema distribuído em um ambiente doméstico com um grupo limitado de aparelhos.

As tecnologias exploradas e estudadas foram: RPC (CORBA), IDL e soquetes. Buscamos selecionar dois modelos com níveis de abstração distintos, a fim de entender as diferenças entre os problemas e benefícios que ambas trazem. Além disso, a linguagem utilizada para programar as aplicações foi python, pois essa possui bibliotecas que facilitam a interação com a interface física do RaspberryPI.

Especificação

Para melhor entendimento do que seriam os objetivos do sistema a ser especificado, segue abaixo uma lista de exemplos de tarefas que poderiam ser realizadas pelo mesmo.

- Controlar, monitorar e regular automaticamente a temperatura ambiente.

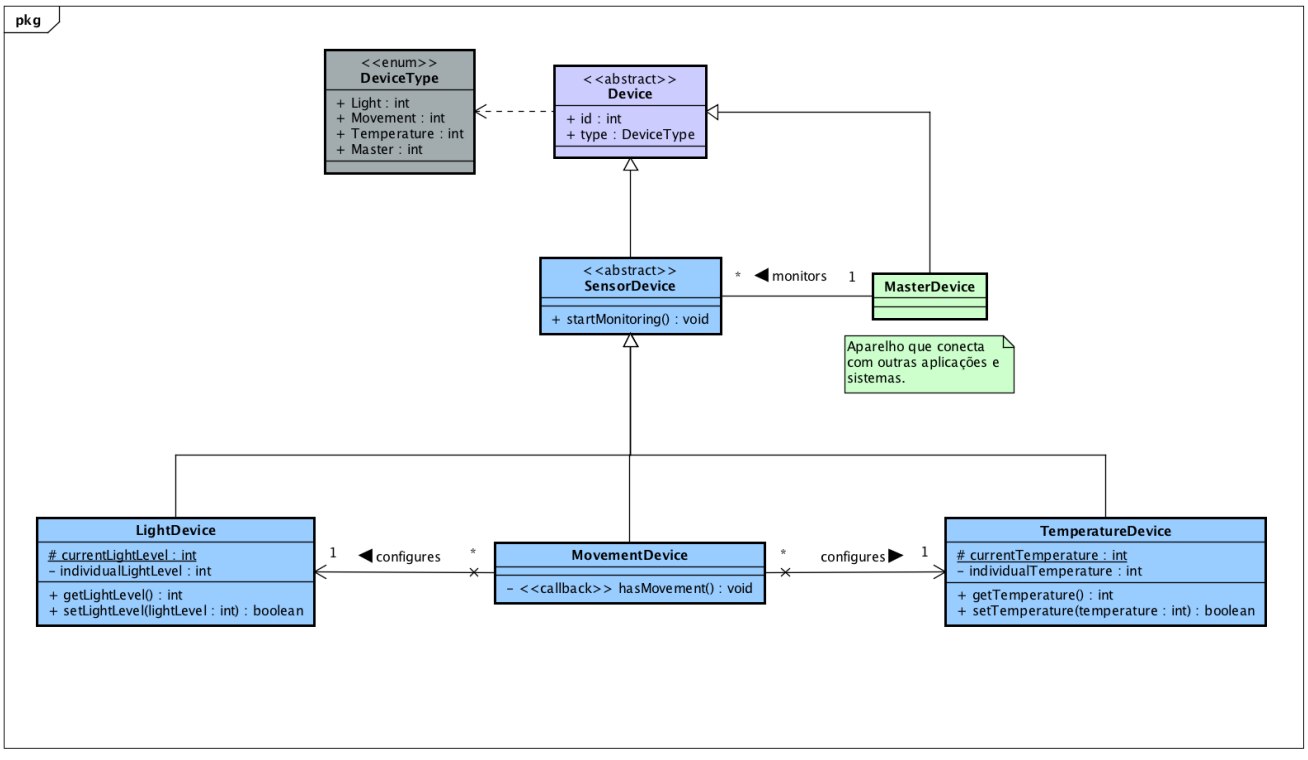
- Controlar, monitorar e regular automaticamente o nível de luminosidade de um cômodo.

- Estabelecer diferentes tipos de ações programáveis a serem disparadas quando alguém entra no cômodo (ex.: acende as luzes e liga a televisão).

- Controlar de uma aplicação no celular ou tablet as funções do ar condicionado e televisão.

Dito isso, o sistema se organiza em grupos de dispositivos com sensores (luz, movimento e temperatura) e sem sensores (mestres), de forma que os últimos gerenciam os primeiros. Os dispositivos com sensores de luz e temperatura se organizam em subgrupos gerenciados por um líder. Além disso, os dispositivos mestre cuidam da interação com APIs externas ao sistema. Essa funções encontram-se melhor detalhadas nas seções abaixo.

Diagrama de classes



Grupos

Os **LightDevices** e **TemperatureDevices** encontram-se organizados em grupos comandados por um aparelho especial, a partir daqui definido como **GroupLeaderDevice.** Um GroupLeaderDevice articula a interação dos aparelhos dentro de um mesmo grupo e serve de proxy na comunicação com aparelhos de outras classes. Além disso, ele também possui os sensores necessários para desempenhar as funções de sua superclasse (seja LightDevice ou TemperatureDevice).

Todos os aparelhos de um mesmo grupo precisam se conectar ao GroupLeaderDevice durante sua inicialização. Para isso, cada aparelho lê as informações necessárias para localizar o GroupLeaderDevice de um arquivo de configuração. Se o GroupLeaderDevice não se encontra ativo ou alcançável durante qualquer período de funcionamento de um aparelho, o mesmo deve, se possível, armazenar informações relevantes internamente e tentar repetidamente conectar-se ao GroupLeaderDevice em intervalos de tempo pré-configurados.

Para cada aparelho que se conecta ao GroupLeaderDevice, o mesmo atualiza a lista de aparelhos que deve comandar. Periodicamente, o GroupLeaderDevice verifica se todos os aparelhos em sua lista encontram-se ativos e atualiza a mesma caso necessário.

LightDevices

Um grupo de aparelhos com sensores de luz deve monitorar o nível de luminosidade de um cômodo e alterar o mesmo conforme uma pré-configuração ou os comandos dos usuários. Para tal, cada aparelho mantém uma variável de controle interna denominada **currentLightLevel.** Toda vez que os sensores do aparelho realizam a leitura do nível de luminosidade, o currentLightLevel do aparelho é atualizado.

Periodicamente, o GroupLeaderDevice pergunta qual é o currentLightLevel de cada um dos aparelhos. Conforme as medições adquiridas, o GroupLeaderDevice altera, ou não, o nível de luminosidade do cômodo. O nível de luminosidade global, calculado pelo GroupLeaderDevice a cada ciclo de medições, é denominado **globalLightLevel.**

É possível que o usuário escolha por controlar individualmente o nível de luminosidade de cada área do cômodo. Nesse caso, o GroupLeaderDevice altera diretamente os respectivos níveis de luminosidade.

TemperatureDevices

Os TemperatureDevices funcionam exatamente com a mesma lógica que os LightDevices, monitorando a temperatura ao invés dos níveis de luminosidade. As respectivas variáveis são denominadas **currentTemperature** e **globalTemperature.**

MovementDevices

Um aparelho com sensor de movimento ou um grupo de aparelhos com sensores de movimento monitoram uma mesma área em busca de indícios de movimentação. Assim que qualquer um dos aparelhos capta algum tipo de movimento, ele notifica seu GroupLeaderDevice. O GroupLeaderDevice, então, executa uma ação pré-determinada e, caso necessário, se conecta com aparelhos de outras classes.

Possível caso de uso para a aplicação com o sensor de movimento: toda vez que alguém entrar no cômodo, o cômodo deve estar iluminado. Para tal, quando um sensor de movimento for ativado, o GroupLeaderDevice deve se comunicar com o GroupLeaderDevice do grupo de sensores de luminosidade e avisar que o mesmo deve modificar seu globalLightLevel. Assim, o GroupLeaderDevice dos sensores de luminosidade regulará os níveis de luminosidade caso necessário.

MasterDevice

O MasterDevice possui acesso às configurações do ar condicionado e da televisão. Ele pode, através de uma interface, ligar e desligar a televisão e o ar condicionado. Além disso, também pode alterar a temperatura do ar condicionado.

A aplicação funciona como um grupo, onde o MasterDevice é o GroupLeaderDevice e os líderes de grupo dos outros grupos são os aparelhos dependentes. O MasterDevice possui acesso ao globalLightLevel e à globalTemperature, e pode alterar os níveis esperados de temperatura e luminosidade.

Através de uma API externa aos grupos de aparelhos, o usuário é capaz de interagir com a aplicação. A API acessa as funções disponíveis ao MasterDevice e assim serve como uma camada extra de abstração para, por exemplo, a configuração e visualização dos níveis de luminosidade. Dessa forma, sistemas externos à infraestrutura da aplicação, como um aplicativo de celular, conseguem acessar as funções do MasterDevice sem se preocupar com as peculiaridades da forma de comunicação usada pelos aparelhos (CORBA ou soquetes).

Desenvolvimento

Etapa 1

Antes de começar a desenvolver propriamente a aplicação, decidimos fazer uma experimentação simples com o RaspberryPI e sua interface de sensores. Para tal, foi necessário entender quais eram as características da interface física do RaspberryPI, o GPIO (general purpose input/output). Assim, descobri que o GPIO possui 17 pinos de input/output ao total (26 ao todo incluindo *ground* e energia), de forma que é possível interagir com o mesmo usando uma biblioteca de python homônima.

Abaixo, temos um exemplo simples de o que seria um programa que liga um LED conectado ao pino de número quatro, dorme por cinco segundos e desliga o mesmo pino antes de terminar de executar.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

led = 4

GPIO.setup(led, GPIO.OUT)

GPIO.output(led, 1)

time.sleep(5)

GPIO.output(led, 0)

GPIO.cleanup()
```

Percebe-se que, quanto à forma de programação, o programa em python não entra em loop automaticamente como no Arduino[1] ou fica em espera como em outros sistemas. Os programas que se comunicam com a interface física do RaspberryPI rodam como scripts de python normais e, caso seja de interesse deixá-los executando perpetuamente, será necessário implementar um loop eterno manualmente (while True: {...}) ou usar alguma função wait(). Acredito que essa característica seja beneficial para o projeto pois garante mais controle sobre o software.

---

[1] Para mais informações sobre Arduinos, consultar https://www.arduino.cc

Etapa 2

Para as primeiras experimentações com Python[2] e CORBA[3] (utilizando o omniORB), decidimos por implementar, incrementalmente, os seguintes programas:

1) Um programa que cria uma *thread* "A", onde esta deve entrar em loop infinito, imprimindo um valor de luminosidade (fictício) na tela e dando *sleep* de um segundo a cada iteração.
2) O mesmo programa do item 1) de forma que, após criar a *thread* "A", o mesmo fica esperando pelo input do usuário. O valor obtido deve ser usado para alterar o tempo de *sleep* da *thread* "A".
3) Dois programas, ambos usando omniORB: um programa que espera pelo input do usuário e outro que imprime um valor de luminosidade lido. Os programas devem utilizar o ORB para se comunicarem e realizarem a mesma tarefa do item 2).

Após a implementação dos ítens descritos, vale a pena ressaltar os seguintes pontos:

- Paralelismo e concorrência em Python: devido à forma como é implementado o principal interpretador de Python (CPython[4]), não é possível se aproveitar facilmente dos benefícios de performance da programação paralela. Como o interpretador não é *thread safe*, ele utiliza um *lock* global conhecido como Global Interpreter Lock (GIL[5]). Por causa dessa característica, um programa utilizando a biblioteca *threads* não ganha em desempenho com o aumento do número de núcleos de um processador. Contudo, através da biblioteca *multiprocessing*[6], é possível burlar essa característica criando-se novos processos cada um com sua própria instância do interpretador. Vale notar que a utilização dessa biblioteca traz consigo as características da criação e manutenção de processos, como áreas separadas de memória para diferentes processos, a consequente necessidade de se comunicar através de mensagens e a queda de performance gerada pela troca de contexto entre elevados números de processos. Por fim, como a aplicação não necessita de alto desempenho, optamos por não utilizar a biblioteca *multiprocessing* e continuar usando a biblioteca *threads*.

---

[2] A versão de Python utilizada a 2.7.12. Para mais informações, consultar https://www.python.org

[3] O ORB de CORBA (http://www.corba.org/) utilizado foi o omniORBpy versão 4.2.1. Para mais informações sobre omniORB, consultar http://omniorb.sourceforge.net

[4] https://en.wikipedia.org/wiki/CPython

[5] https://wiki.python.org/moin/GlobalInterpreterLock

[6] https://docs.python.org/2/library/multiprocessing.html

- CORBA e omniORB: o processo de instalação do ORB não é simples. Não foi possível fazer o *build* do projeto e utilizá-lo dessa forma. Foi necessário a utilização de um *package manager* (*homebrew*) para instalar o ORB, tornando difícil a instalação da biblioteca no RaspberryPI, que, por ser um sistema limitado, não dispõe de *package managers* similares disponíveis. Esse fator, e a falta de documentação e informação sobre a tecnologia na internet, levou-me a considerar se CORBA poderia ser facilmente utilizado em outros projetos e certamente tornou-se um grande empecilho para o desenvolvimento.

Etapa 3

Passadas as experimentações iniciais, julgamos necessário começar a implementar, em IDL, as interfaces definidas pela especificação. Para isso, foi criado o módulo INF1822, que contém as interfaces dos aparelhos e a interface para um serviço de catálogo customizado (optamos por não utilizar o *naming service* nativo do omniORB, pois este não fornecia a complexidade requerida para a indexação de objetos que desejávamos).

Considerações:

- O *naming service*, a partir daqui denominado catálogo, foi programado de forma a servir os propósitos da aplicação. Dessa forma, como as regras do sistema são definidas em termos de grupos de aparelhos de um tipo controlados por um aparelho mestre, pareceu sensato implementar a função de registro em função de um *DeviceType*. Assim, torna-se desnecessária a forma tradicional de obter um objeto remoto através de seu "nome", pois o sistema não requer esse tipo de identificação entre aparelhos. Por consequência, alterei a interface para obtenção de referências, como explicado abaixo:
  - **IORList getByType(in DeviceType type)**
    Através dessa função, é possível obter as referências de um conjunto de objetos de um mesmo tipo. Esse método deve ser utilizado por um aparelho mestre que, devido a condições adversas (ex.: falta de luz), parou de funcionar momentaneamente e, ao religar, precisa obter as referências para os sensores que estava monitorando previamente.
  - **string getMasterForType(in DeviceType type)**
    Através dessa função, é possível obter o IOR do aparelho mestre para um dado tipo de sensores. Os aparelhos com sensores utilizam esse método para saber quem os está monitorando. Assim, não há necessidade de tornar fixa a referência para o mestre, de forma que o mesmo pode ser alocado e realocado dinamicamente em momentos distintos da aplicação.
  - Dados esses métodos, vale a pena repensar a necessidade da função de assinatura **boolean startMonitoringDevice(in string**

deviceIor), que deve ser chamada pelo sensor em sua inicialização, usando a referência do mestre. Acredito que essa função deva continuar a existir, pois dado que a aplicação é dinâmica, mais sensores podem ser adicionados a um sistema a qualquer momento, de forma que seria impossível para o mestre saber quando atualizar sua lista de referências a menos que avisado. Assim, o sistema possui uma redundância proposital na forma com a qual a lista de referências do mestre é atualizada, a fim de garantir sua dinamicidade.
  - Com a implementação sugerida do catálogo, todo aparelho, mestre ou não, deve saber o IOR do catálogo para pode operar. Contudo, como o IOR muda toda vez que que o programa é reiniciado, seria necessário mudar constantemente a configuração dos aparelhos. Para evitar esse problema, podemos usar uma URL *corbaloc*, que representa outra forma de obter a referência para o objeto do catálogo. Uma URL *corbaloc* opera usando uma combinação de **host**, **porta** e **nome** para mapear a referência para o objeto. Como a URL *corbaloc* é fixa, podemos usá-la para acessar o catálogo de forma estática, dispensando a necessidade impraticável de reconfigurar cada aparelho toda vez que o catálogo fosse reiniciado.
- Sobre a experiência de implementar o catálogo, vale a pena ressaltar que inicialmente eu considerava o *naming service* nativo do omniORB uma ferramenta mais complexa do que ela realmente o é. Assim, entender que o esse *naming service* é como outro programa qualquer implementado usando CORBA foi fundamental para o desenvolvimento do projeto.

Etapa 4

Uma vez projetada a interface em IDL, comecei a programar de fato a aplicação.

Observação: Com relação às threads de python, foi necessário averiguar se elas são threads de sistema ou se operam de forma diferente, a fim de entender melhor os momentos em que ocorre a preempção e qual seria a necessidade da utilização de locks para evitar problemas de concorrência. Assim, foi constatado que python mapeia suas threads para threads POSIX/pthreads ou Windows threads que, portanto, são gerenciadas pelo sistema operacional (a preempção ocorre de forma não determinística), necessitando a utilização de *locks* no código para impedir possíveis condições de corrida.

Durante a implementação, percebi que, devido à forma como o catálogo está configurado, seria impossível, por exemplo, definir dois mestres para uma mesmo tipo de aparelho. Contudo, numa aplicação envolvendo IOT isso seria um problema, pois é lógico pensar que o sistema pode, por exemplo, controlar os sensores e

controladores de luminosidade de dois cômodos distintos. Assim, é preciso que existam, além de divisões por **tipos** de aparelho, divisões por **grupos** de aparelhos. Para essas divisões, daremos o nome de *clusters*. Dessa forma, é necessário que o catálogo separe seus registros em três camadas, sendo elas *cluster*, *tipo* e *nome*. Por consequência, a interface do catálogo precisou ser ajustada para incluir o identificador de cada *cluster* quando apropriado.

O código da aplicação encontra-se dividido em quatro arquivos principais: catalogue.py, master.py, client.py, aux.py. O arquivo **aux.py** serve como repositório de funções auxiliares utilizadas pelos outros scripts. Os scripts **master.py** e **client.py** são, respectivamente, implementações parciais para *GroupLeaderLightDevices* e *LightDevices*, conforme visto na especificação. Além disso, ambos recebem o id do cluster ao qual pertencem e um id pessoal de identificação quando inicializados.

Para facilitar os testes e observar a interação entre diversas instâncias de clientes, foi criado o script **runtest.sh**, que possibilita simular, em uma única máquina, a inicialização de um número configurável de *clusters* e aparelhos:

```
#!/bin/bash

echo "Starting test"

if [ "$1" = "" ]
then
     echo "First parameter must be number of clusters..."
elif [ "$2" = "" ]
then
     echo "Second parameter must be number of clients for each
cluster..."
else
     python catalogue.py &
     sleep 1

     # Clusters
     for i in `seq 1 $1`
     do
          python master.py $i 0 &
          # Clients
          for j in `seq 1 $2`
          do
               sleep 1
               python client.py $i $(($j+($i*100))) &
          done
     done
fi
```

Assim, executando “./runtest 3 9” através da linha de comando é possível criar três *clusters* diferentes com dez aparelhos em cada, de forma que um deles é um GroupLeaderDevice. Os clientes possuem uma lista de valores de luminosidade prefixada para facilitar a simulação e os líderes informam, a cada cinco segundos, o valor médio de luminosidade calculado a partir de todos os aparelhos registrados.

Apesar de não contar com todos os recursos inicialmente especificados, a implementação apresentada serviu para exemplificar e estudar os principais conceitos visados dentro do projeto, como a comunicação entre múltiplos aparelhos, a arquitetura de sistemas distribuídos e os problemas de infraestrutura e concorrência decorrentes da mesma.

Código: https://github.com/renan061/inf1822/corba

Etapa 5

A fim de estudar formas de comunicação com níveis de abstração diferentes, julgamos interessante analisar como seria o implementação proposta usando **soquetes**. Dessa forma, decidi programar parte do catálogo com soquetes, visando entender quais seriam as principais diferenças entre esse modelo e CORBA.

Vale a pena ressaltar que a linguagem escolhida foi novamente python, por conta das facilidade envolvendo sua utilização junto com o RaspberryPI. Além disso, python providencia um biblioteca homônima para o uso de soquetes, que simplifica parte do processo de criação e gerenciamento dos mesmos.

A primeiras diferenças que observei referem-se à própria estrutura do soquete. Em CORBA, era necessário que o catálogo armazenasse o IOR de cada aparelho, enquanto que, com soquetes, o mesmo deve armazenar o **endereço do aparelho** e a **porta** a ser utilizada pelo programa. Além disso, cada instância da aplicação deve possuir, pelo menos, um soquete tipo servidor e outro tipo cliente para realizar a comunicação com outros aparelhos. Para casos em que é necessário se comunicar com *N* aparelhos, o programa deve possuir *N* soquetes tipo cliente. A partir dessas características, é possível ter uma noção de como CORBA abstrai grande parte da complexidade de soquetes para o uso do programador e, ao mesmo tempo, gerencia um número dinâmico de conexões simultâneas.

No que se refere ao tratamento de mensagens, utilizar soquetes significa ter que implementar todo o protocolo de comunicação manualmente. A tecnologia ocupa-se apenas de transmitir bits de informação entre cliente e servidor, de forma que o processo de traduzir e dar sentido a esse conteúdo é externo aos soquetes. Da

mesma forma, é necessário garantir que ambos os lados da comunicação estejam sintonizados quanto aos tipos de mensagens a serem aceitos, papel esse que era desempenhado pelo *middleware* de CORBA em conjunto com a interface em IDL, através da checagem de tipos de dados e assinaturas de métodos.

Assim, defini o seguinte protocolo de mensagens para o acionamento de métodos do catálogo:

catalogue#<method>#<args>
- <method>: nome do método a ser chamado dentro do catálogo. Opções: *register*, *registerMaster*, *deregister*, *deregisterMaster*, *getByType* e *getMasterForType*.
- <args>: lista de parâmetros a serem passados para o método, divididos por vírgula. Cada método aceita uma quantidade de parâmetros diferente.

Por exemplo, para o caso em que um aparelho deseja se registrar junto ao catálogo, a mensagem ficaria na forma:

```
“catalogue#register#
     host,port(long),id(long),type(string),clusterId(long)”
```

Onde “host” e “type” devem ser do tipo *string* e “port”, “id” e “clusterId” do tipo *long*.

Por fim, foi necessário criar um método “parse” para cuidar do tratamento da mensagem dentro do catálogo. Esse método recebe uma string, checa o formato da mesma para garantir que ela está de acordo com o protocolo e instância um objeto de mensagem que será consumido pelo catálogo. Percebi, nesse caso, que o tratamento de erros usando soquetes também é excepcionalmente convoluto. No caso em que a mensagem encontra-se formatada errada, por exemplo, seria necessário repassar o erro para a parte do programa que está cuidando do soquete do cliente e, através dele, reportar a falha.

Código: https://github.com/renan061/inf1822/sockets

Conclusão

Para analisar ambos os modelos de comunicação estudados, devo ponderar suas vantagens e problemas junto ao objetivo da aplicação.

Assim, começo pelo processo de configuração do ambiente de desenvolvimento e produção. Não foi possível instalar CORBA facilmente, como já dito previamente. Mais do que isso, a instalação no RaspberryPI agravou-se por conta das restrições do aparelho, que roda com um sistema operacional limitado e, portanto, não se propõe a estar em conformidade com as necessidades previstas para o processo de instalação de CORBA. A documentação sobre o framework é escassa no que se refere à sua configuração, de forma que não foi possível testar o sistema no RaspberryPI (não acredito que seja impossível instalar CORBA no aparelho, contudo tal processo exige mais tempo e pesquisa do que eu podia fornecer à essa parte do projeto). Em contrapartida, a utilização de soquetes não foi empecilho no RaspberryPI, que vem com python e a biblioteca *sockets* pré-instalados.

Quanto ao desenvolvimento, é inegável que CORBA é um *framework* muito poderoso e a própria estrutura da aplicação se adequa muito bem à noção de objetos distribuídos. É intuitivo pensar em cada aparelho como um objeto idêntico a vários outros em uma rede e, portanto, torna-se fácil planejar a projetar a aplicação. O mesmo cenário não se repete com soquetes, uma ferramenta muito mais simples que requer muito mais do programador. Para a utilização de soquetes, o programa precisa possuir uma complexidade além da introduzida pelas regras de negócio da aplicação, sendo necessário desenvolver um módulo inteiro a parte que cuide do gerenciamento das conexões e da comunicação com outros aparelhos. Por consequência, a probabilidade do programador introduzir uma falha ou produzir código subóptimo é altíssima, enquanto que CORBA, uma tecnologia com nível de abstração de mais alto nível, não enfrenta os mesmos problemas.

Além disso, como os testes foram rodados em redes internas suficientemente estáveis e controladas, não foi possível enxergar todo o espectro de problemas e erros com soquetes provenientes desse tipo de falha. O tratamento de todas essas situações certamente teria deixado o código do exemplo com soquetes muito mais complexo. No que se refere a CORBA, a tecnologia cuida de encapsular esse tipo de erro e fornece uma interface de exceções muito mais enxuta e simplificada.

Por fim, acho válido notar que a especificação proposta não é especialmente complexa, mas também não é excessivamente simples. Por conta do peso atrelado à utilização de um framework externo (o problema da instalação, por exemplo), usar CORBA acaba por complicar de forma desnecessária o processo de desenvolvimento.

Para CORBA, restrinjo-me a pensar que somente aplicações que *realmente* necessitam de seus benefícios devam usá-la. Em paralelo, as dificuldades que acompanham a utilização de soquetes são igualmente relevantes para que consideremos se esta é a melhor tecnologia a ser utilizada. Para extensões ou implementações completas do projeto, entendemos, então, que é necessário buscar uma tecnologia com nível de abstração intermediário. Um caminho possível seria explorar AMQP[7] (*advanced message queuing protocol*) e outras ferramentas que fornecem protocolos intermediários para a comunicação e troca de mensagens.

[7] https://www.amqp.org/