# RELATÓRIO PROJETO 3
# MC833 - LAB. DE REDES
# 1s2019

**NOME:** João Paulo Soubihe

**RA:** 151106

- ## Introdução

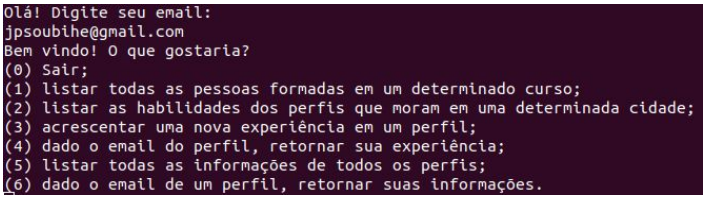
  Nessa última etapa do projeto de MC833 buscamos aprender um pouco mais sobre um método de comunicação em redes de computadores. O Remote Method Invocation, mais conhecido como RMI, propõe uma abordagem distinta das tratadas nos projetos anteriores, apesar de se basear em uma comunicação pelo protocolo TCP. Sua implementação, em mais alto nível, fornece um nível de abstração maior, escondendo os detalhes de cada passagem de informação e facilitando a vida do programador. Também aproveitamos a situação para fazer uma última comparação entre os protocolos TCP/UDP e o RMI, as suas diferentes abstrações, o jeito com que se organizam e o tempo gasto no transporte de informações.

- ## Sistema

  Nosso sistema terá 6 funcionalidades, similares às desenvolvidas no *Projeto 1*, mas sem imagens agora, com o intuito apenas de avaliar o funcionamento da comunicação pelo método RMI.

  (1) listar todas as pessoas formadas em um determinado curso;
  (2) listar as habilidades dos perfis que moram em uma determinada cidade;
  (3) acrescentar uma nova experiência em um perfil;
  (4) dado o email do perfil, retornar sua experiência;
  (5) listar todas as informações de todos os perfis;
  (6) dado o email de um perfil, retornar suas informações.

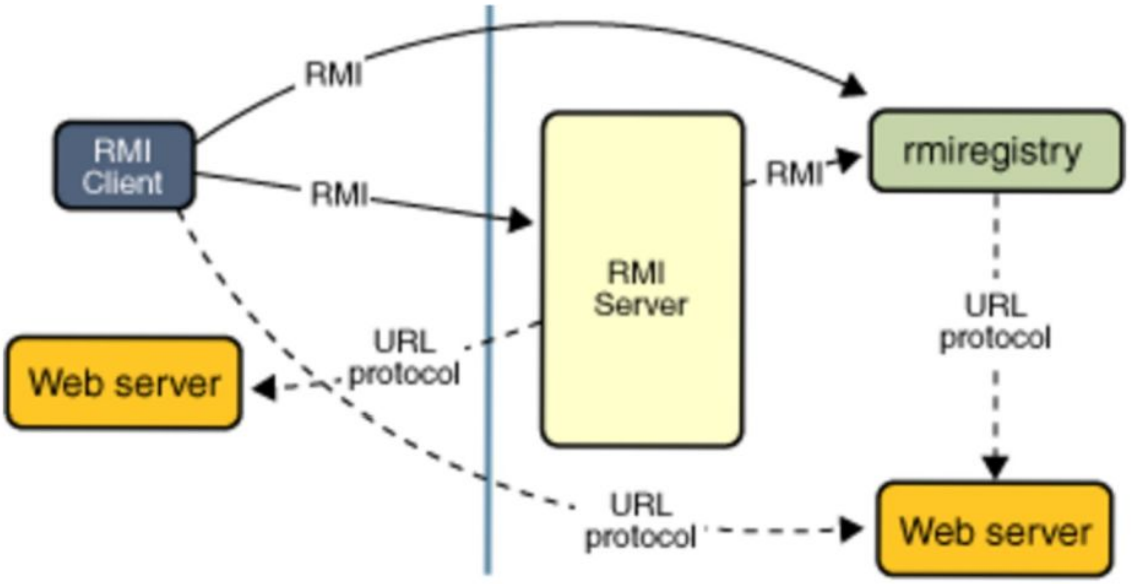
  Após realizada a conexão com o servidor, o lado cliente pedirá ao usuário um email de login, checada pela função *checkLogin()*

  Caso o login seja validado, um menu aparece na tela do cliente para a visualização das funcionalidades disponíveis.
  O usuário então pode escolher o que fazer no sistema e digita o número respectivo no terminal. Cada número representa uma funcionalidade e um método diferente a ser invocado.

  É importante salientarmos um aspecto importante do RMI.  O programa servidor é instanciado em uma máquina e registrado como atributo em um objeto(servidor) remoto de registros, que servirá como um diretório de consulta aos clientes, ligando dados como porta e endereço IP a um nome de conhecimento público. Essa camada, atribuída a um stub na porta cliente, referenciando uma interface pública, remota. Também trata do stream de dados, baseado no protocolo TCP, que converte

os dados para um jeito “streamavel”. A descrição pode ficar mais clara com o auxílio da imagem abaixo:




O RMI apresenta uma concorrência diferente dos modelos estudados nos projetos anteriores. Aqui, ela se dá por threads, um processo menos custoso que a geração de um novo processo inteiro para permitir a execução simultânea, ou concorrente, por dois clientes distintos.

- **<u>Armazenamento</u>**

  O programa Server se localizará na pasta RMI_Server, juntamente da interface Compute, que permitirá a instância e a implementação de objetos remotos e a classe Profile. Já o programa Client se localizará na pasta RMI_Client, juntamente a interface Compute e a classe Profile, que serão necessárias também nesse lado da comunicação. Os perfis se encontrarão no diretório pai de RMI_Server, na forma de arquivos .txt.

  ➔ Arquivos .txt

  Estarão reunidos na mesma pasta do programa servidor e terão nomes correspondentes ao seu email organizados como no exemplo abaixo

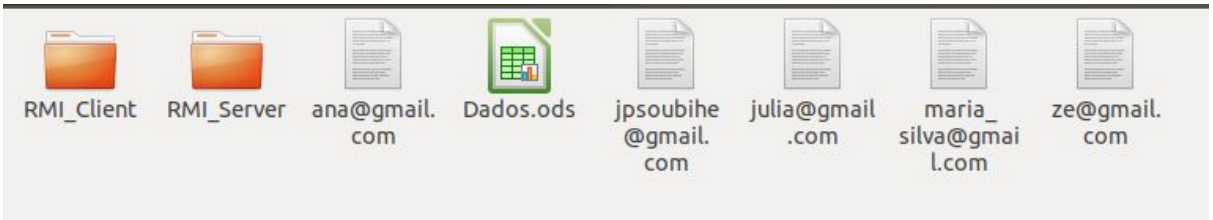



Figura 1 - Diretório de arquivos do projeto

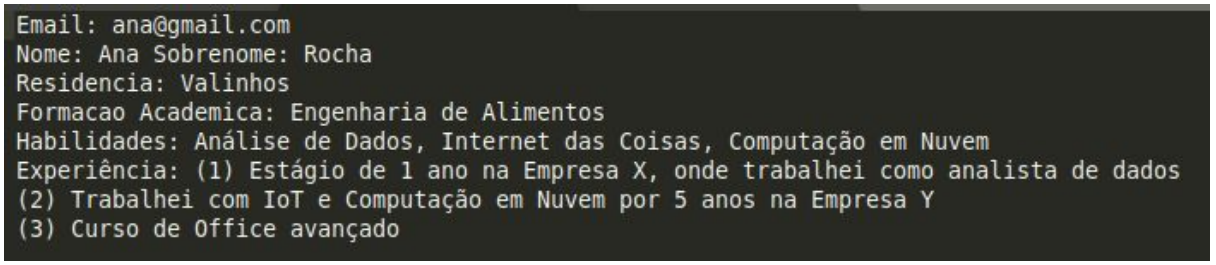



Figura 2 - Modelo de perfil do cliente

- **<u>Desenvolvimento</u>**

  De primeira vista, ao ter conhecimento do projeto, já identificamos um maior nível de abstração por parte da implementação. Não temos controle de como determinada informação está sendo passada, do tamanho de cada pacote de dados, ou do modo que a conexão é estabelecida.
  Por outro lado, a implementação se torna mais fácil, sem muitas restrições. O RMI dá mais liberdade ao programador de executar métodos e instanciar outros objetos, sem se preocupar com a passagem de dados.
  A preocupação com a localização (porta e endereço IP) do server também ficou em segundo plano, o rmiregistry nos cobre nessa questão.

  Para começar o desenvolvimento de nosso servidor RMI, determinamos os métodos presentes na interface *Compute*. Ela será peça chave em nosso projeto, permitindo as máquinas clientes de invocar tais métodos seguindo a implementação no servidor.
  Uma outra classe criada apenas para auxílio em alguns métodos foi a classe *Profile*, ela guarda como atributos os dados presentes nos arquivos .txt de cada perfil cadastrado, facilitando a leitura por parte dos clientes em algumas situações, dos seus respectivos dados.

  O Server conterá um array de String's com o nome respectivo de cada perfil, identificados pelo email. Como citado acima, a função checkLogin() é a responsável por validar o login por parte do cliente, através do login com o email correto.
  Para cada funcionalidade escolhida pelo cliente, um método específico é invocado pelo próprio cliente, com acesso às assinaturas através da interface. Novamente, o *stub* converte os parâmetros a serem passados ao servidor, que efetivamente executará a função, e os "decodifica" para a resposta ao cliente. Com esse efeito, surge a necessidade da serialização dos parâmetros da classe, para que seja possível a conversão dessas mensagens em um stream de dados. Tanto a classe Server quanto a classe Client.

  A primeira ação da execução main da classe Client é atribuir a variável *registry* o registro do Server, registrado anteriormente, através do método lookup (args[0]) onde args[0] seria o ip do servidor remoto que armazena tais registros, que contém a

informação necessária para que eventuais clientes consigam se conectar com as máquinas servidores.
Após a conexão, validamos o login do cliente, como citado acima. No caso de uma resposta positiva do servidor, uma mensagem de boas vindas, juntamente com o menu de funcionalidades é impressa na tela.
A partir desse momento, foi implementado um loop que se quebra apenas com a escolha da funcionalidade 0, por parte do cliente. Assim, tentamos abranger a utilização máxima de um cliente sem a necessidade de reconexão, custosa em termos do nosso sistema.

- **Análises**

Para efetuar as análises de tempo de operação, utilizamos o método System.currentTimeMillis() para medir o tempo TS de consulta no servidor e o tempo TT de consulta no cliente (Como mostrado na referência [2]). A partir destes dois tempos, determinamos o tempo de comunicação como sendo TT - TS.
Medimos 20 execuções de cada operação, portanto nossas medidas não apresentaram uma grande variação de tempo, como será mostrado a seguir.
Usamos dois computadores em uma mesma rede local (Rede do IC 3.5) durante a manhã, ou seja, um horário de baixo uso da rede.
Importante ressaltar que a média , o desvio padrão, o erro padrão e a margem de erro, foram calculadas pelas fórmulas descritas em (*),(**),(***),(****). Seguimos então os passos indicados por [3].

(*) média = $\sum Xi/n$ , onde Xi é a amostra e n o número de amostras no total
(**) desvio padrão = $[\sum(Xi - media)^2/n]^{1/2}$
(***) erro padrão = desvio padrão / $n^{1/2}$
(****) margem de erro = erro padrão * W(Z/2), onde Z representa o nível de confiança exigido e W() mostra que o valor foi retirado de [4]

Sendo assim, sabemos que os resultados poderão oscilar de Tempo de comunicação ± margem de erro, determinada pelo nível de confiança de 95%.

Os resultados obtidos estão abaixo:

| Task 1 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 26 | 31 | 5 |
| 11 | 13 | 2 |
| 8 | 11 | 3 |
| 8 | 11 | 3 |
| 5 | 9 | 4 |
| 7 | 10 | 3 |
| 5 | 8 | 3 |
| 6 | 9 | 3 |
| 6 | 7 | 1 |
| 4 | 7 | 3 |
| 5 | 6 | 1 |
| 5 | 8 | 3 |
| 5 | 8 | 3 |
| 4 | 7 | 3 |
| 4 | 7 | 3 |
| 4 | 8 | 4 |
| 4 | 7 | 3 |
| 4 | 7 | 3 |
| 4 | 6 | 2 |
| 5 | 9 | 4 |
| Media | 2,95 | |
| Desvio padrão | 0,944513241388333 | |
| Erro padrão | 0,211199581339298 | |
| Margem de erro | 0,413951179425024 | |

Figura 3 - Tempos medidos na execução da 1ª tarefa

| Task 2 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 11 | 15 | 4 |
| 4 | 7 | 3 |
| 3 | 6 | 3 |
| 4 | 7 | 3 |
| 4 | 6 | 2 |
| 3 | 7 | 4 |
| 4 | 7 | 3 |
| 4 | 7 | 3 |
| 4 | 7 | 3 |
| 4 | 7 | 3 |
| 4 | 6 | 2 |
| 4 | 8 | 4 |
| 3 | 6 | 3 |
| 3 | 6 | 3 |
| 2 | 5 | 3 |
| 3 | 6 | 3 |
| 3 | 5 | 2 |
| 4 | 6 | 2 |
| 3 | 6 | 3 |
| 4 | 6 | 2 |
| Media | 2,9 | |
| Desvio padrão | 0,640723275517187 | |
| Erro padrão | 0,143270079882276 | |
| Margem de erro | 0,28080935656926 | |

Figura 4 - Tempos medidos na execução da 2ª tarefa

| Task 3 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 6 | 9 | 3 |
| 3 | 6 | 3 |
| 4 | 6 | 2 |
| 3 | 6 | 3 |
| 4 | 6 | 2 |
| 4 | 6 | 2 |
| 4 | 6 | 2 |
| 4 | 6 | 2 |
| 4 | 6 | 2 |
| 4 | 6 | 2 |
| 16 | 18 | 2 |
| 5 | 7 | 2 |
| 4 | 5 | 1 |
| 4 | 7 | 3 |
| 4 | 5 | 1 |
| 6 | 8 | 2 |
| 3 | 7 | 4 |
| 4 | 5 | 1 |
| 4 | 5 | 1 |
| 4 | 6 | 2 |
| Media | 2,1 | |
| Desvio padrão | 0,788068925652412 | |
| Erro padrão | 0,176217568871402 | |
| Margem de erro | 0,345386434987948 | |

Figura 5 - Tempos medidos na execução da 3ª tarefa

| Task 4 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 3 | 7 | 4 |
| 2 | 4 | 2 |
| 1 | 4 | 3 |
| 1 | 5 | 4 |
| 1 | 4 | 3 |
| 2 | 5 | 3 |
| 1 | 5 | 4 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| 1 | 4 | 3 |
| 1 | 5 | 4 |
| 2 | 4 | 2 |
| 1 | 3 | 2 |
| 1 | 4 | 3 |
| 1 | 4 | 3 |
| 1 | 6 | 5 |
| 1 | 5 | 4 |
| 2 | 5 | 3 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| Media | 3 | |
| Desvio padrão | 0,917662935482247 | |
| Erro padrão | 0,205195670417031 | |
| Margem de erro | 0,40218351401738 | |

Figura 6 - Tempos medidos na execução da 4ª tarefa

| Task 5 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 24 | 38 | 14 |
| 10 | 15 | 5 |
| 9 | 16 | 7 |
| 5 | 10 | 5 |
| 7 | 14 | 7 |
| 5 | 7 | 2 |
| 6 | 12 | 6 |
| 4 | 8 | 4 |
| 5 | 11 | 6 |
| 4 | 9 | 5 |
| 4 | 9 | 5 |
| 4 | 9 | 5 |
| 4 | 7 | 3 |
| 4 | 9 | 5 |
| 4 | 9 | 5 |
| 5 | 10 | 5 |
| 5 | 10 | 5 |
| 5 | 10 | 5 |
| 4 | 8 | 4 |
| 2 | 6 | 4 |
| Media | 5,35 | |
| Desvio padrão | 2,34576886733271 | |
| Erro padrão | 0,524529864685862 | |
| Margem de erro | 1,02807853478429 | |

Figura 7 - Tempos medidos na execução da 5ª tarefa

| Task 6 | | |
|---|---|---|
| Tempo de consulta (Server) | Tempo de consulta (client) | Tempo de comunicação(ms) |
| 2 | 5 | 3 |
| 1 | 4 | 3 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| 1 | 4 | 3 |
| 1 | 4 | 3 |
| 1 | 4 | 3 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| 2 | 4 | 2 |
| 1 | 3 | 2 |
| 1 | 3 | 2 |
| 1 | 4 | 3 |
| 1 | 4 | 3 |
| 1 | 4 | 3 |
| 1 | 3 | 2 |
| 2 | 3 | 1 |
| 1 | 5 | 4 |
| 1 | 4 | 3 |
| Media | 2,5 | |
| Desvio padrão | 0,688247201611685 | |
| Erro padrão | 0,153896752812773 | |
| Margem de erro | 0,301637635513035 | |

Figura 8 - Tempos medidos na execução da 6ª tarefa

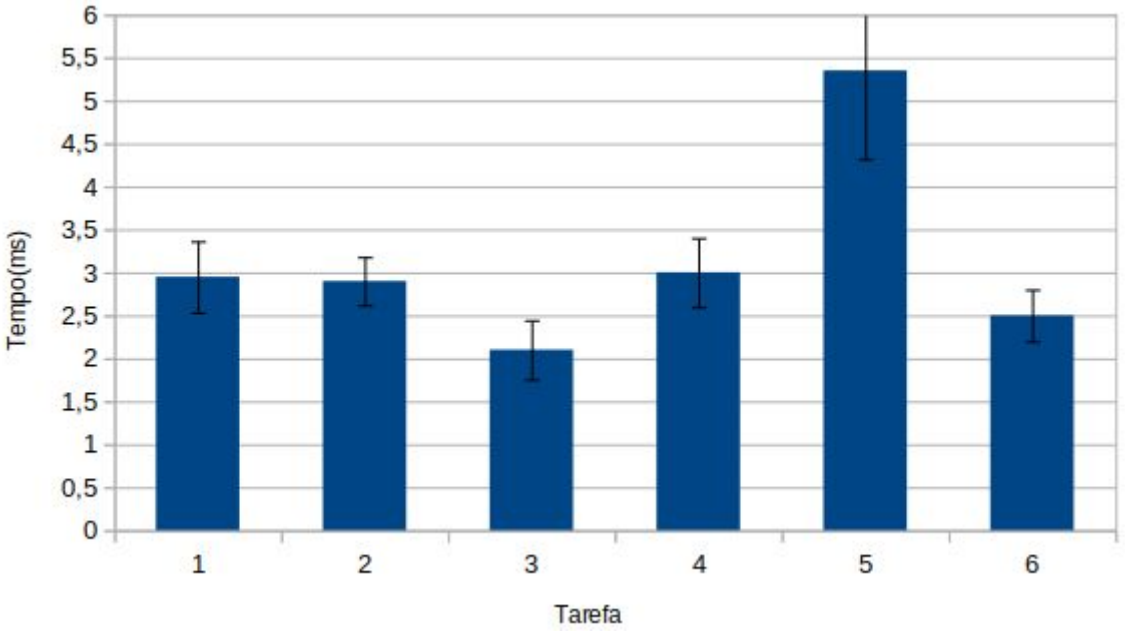



Gráfico 1 - Média dos tempos por tarefa

Analisando o gráfico acima vemos claramente que o comportamento da transmissão de dados é altamente dependente da quantidade de dados a serem enviados. A operação 5, por exemplo, é a que demora mais tempo para ser transmitida, muito disso pois ela é responsável por passar todos os arquivos armazenados no servidor. Se compararmos com a funcionalidade 3, que só escreve a nova experiência num arquivo no servidor, vemos que a última retorna uma resposta ao cliente de maneira muito mais rápida.

Importante salientar que em algumas de nossas tabelas os primeiros tempos medidos apresentaram grande discrepância com o comportamento do restante de nosso teste. Pensamos que isso se deve ao fato de que o RMI funciona em cima de um protocolo TCP, ou seja, para um primeiro contato, é preciso criar todo um contexto, um canal, entre cliente e servidor para a troca de informações. Esse fato pode vir a trazer uma imprecisão para nossa medida.

- ## Conclusão

Com todo esse trabalho realizado, podemos tirar algumas conclusões sobre o RMI em comparação com a implementação em C do protocolo TCP.
O fato mais evidente é que no RMI não é mais necessário informar o endereço IP e a porta em que a mensagem será enviada, precisamos apenas da referência a um servidor remoto de registros, que passará esse tipo de dados aos clientes. Com isso, funcionalidades o acesso de novos clientes ou distribuição de fluxo em servidores (supondo muitos acessos simultâneos), por exemplo, podem ajudar (e muito) o desenvolvedor. Além disso, ainda tratando do ponto de vista do desenvolvedor, não há mais uma preocupação com a passagem de informações, o sistema já trata isso. Por outro lado, se a questão é desempenho, talvez seu programa seja um pouco prejudicado, pois não há a garantia de como é feita a transmissão. A interface

remota possibilita o compartilhamento de métodos existentes entre várias aplicações, como um sistema universal, padronizado.
Outra diferença é a respeito da concorrência. No caso do RMI ela é por threads, muito menos complexo que um fork, a instanciação de um processo inteiro novo, especialmente para o tratamento daquela conexão, como é feito na implementação em C.
Em suma, podemos dizer que a implementação em C ocorre em um nível muito mais baixo de abstração, com mais detalhes e restrições, já o RMI foca muito mais na aplicação, em mais alto nível, do que com o core da comunicação entre as máquinas. Podemos mencionar a sua proximidade com o sistema de Web Services, mais utilizados no desenvolvimento de aplicações ao usuário comum.

- **Referências**


[1] https://bit.ly/2KDYl34

[2] www.ic.unicamp.br/~edmundo/MC833/mc833/computar_tempo_trab.pdf

[3] https://pt.wikihow.com/Calcular-o-Intervalo-de-Confian%C3%A7a

[4] https://www.statisticshowto.datasciencecentral.com/tables/z-table/

[5] www.ic.unicamp.br/~edmundo/MC833/turma1s2019/proj2_18331s19.pdf