Estudo de caso: modelo cliente-servidor

Aplicação de passagem de mensagens

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

07 de agosto de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

Crédito de parte das imagens, a menos se especificado: Wikipedia

Objetivos

- Praticar o modelo de passagem de mensagens com concorrência e paralelismo
- Familiarizar-se com arquitetura cliente-servidor

Clientes e servidores

Servidor

Processo que repetidamente serve requisições de processos-cliente

- Modelo cliente-servidor envolve comunicação bidirecional
- O servidor possui uma porta que recebe requisições
- Cada cliente possui uma porta para receber respostas
- Clientes enviam requisições para o servidor pela porta de requisição
- Servidor envia resposta a um cliente pela porta de resposta
- Alternativas de implementação:
  1. Servidor possui índice de clientes e respectivas portas de resposta
  2. Abstrair “requisição” como um objeto que possui mensagem e a porta de resposta

Modelo cliente-servidor intra-processo

- Podemos propor um modelo cliente-servidor com as seguintes caraterísticas:
  - O servidor é uma thread
  - Os clientes são threads
  - Portas de requisições e de respostas são canais (síncronos ou assíncronos)
  - Mensagens são objetos que encapsulam dados e porta de resposta
- Podemos mais tarde adaptar esse modelo para comunicação inter-processo (i.e. via protocolo TCP)

Objetos como closures

Gerenciamento de estado local

- A abstração de objetos será útil para representar servidores, clientes e requisições
- "Objetos" são tradicionalmente associados com encapsulamento e troca de mensagens
- Podemos emular um sistema de objetos por meio de closures[1]

> *OOP to me means only* ***messaging***, ***local retention*** *and* ***protection and hiding of state-process***, *and extreme* ***late-binding*** *of all things. It can be done in Smalltalk and in LISP. (...)*
>
> *– Alan Kay, criador do termo "programação orientada a objetos"*
>
> *Fonte:* `http://bit.ly/1vFSqdk`

[1] Racket inclui os tipos `struct`, que têm o mesmo propósito de definir objetos simples, porém de um modo menos verboso e mais conveniente do que closures. Porém, closures são mais gerais e funcionam em qualquer linguagem com funções de primeira classe.

Objetos como closures

Encapsulamento de objetos

- Em POO, interação entre objetos é via troca de mensagens
- Quando um objeto recebe uma mensagem, "despacha" para o método adequado (com base numa tabela de métodos)
- Um objeto pode ser representado por uma closure:

```
1 (define (make-person name surname)
2   (lambda (method)
3     (cond [(eq? method 'get-name) (lambda () name)]
4           [(eq? method 'get-surname) (lambda () surname)])))
```

- Intuição: as variáveis livres (name e surname) são capturadas no ambiente e são "encapsuladas" pela lambda interna

```
> (define p (make-person "joao" "silva"))
> ((p 'get-name))
"joao"
> ((p 'get-surname))
"silva"
```

- Todo objeto define seu próprio dispatcher (que é representado pela `cond`)

Objetos como closures

Imutabilidade

- Estamos interessados apenas na emulação de *objetos imutáveis*
- Caso haja mudança de estado nas informações encapsuladas, implica-se a geração de um novo objeto

```
1  (define (make-person name surname)
2    (lambda (method)
3      (cond [(eq? method 'get-name) (lambda () name)]
4            [(eq? method 'get-surname) (lambda () surname)]
5            [(eq? method 'set-name)
6             (lambda (new-name)
7               (make-person new-name surname))]
8            [(eq? method 'set-surname)
9             (lambda (new-surname)
10              (make-person name new-surname))]
11           [else (error "Unknown method")])))
```

```
> (define p (make-person "joao" "silva"))
> ((p 'get-surname))
"silva"
> (define p1 ((p 'set-surname) "souza"))
> ((p1 'get-surname))
"souza"
```

Servidor de quadrados

Arquitetura e objetos de requisição

- Para modelar a arquitetura de cliente-servidor, vamos começar com um servidor bem simples
- A única operação realizada é efetuar o quadrado de um número:
  1. Clientes mandam objetos de requisição, contendo o dado (um número) e a porta de resposta
  2. Servidor recebe a requisição, e gera um quadrado
  3. Servidor envia o resultado na porta de resposta de cada cliente
- O construtor de objetos de requisição:

```
1 (define (make-request-obj message response-port)
2   (lambda (m)
3     (cond ((eq? m 'get-message) (lambda () message))
4           ((eq? m 'get-response-port) (lambda () response-port))
5           (else (error "Unknown method")))))
```

Servidor de quadrados

Objetos de clientes

- Vamos supor que cada cliente envia uma sequência de requisições
- Podemos modelar os clientes como objetos
- No caso, terão apenas uma operação, que gera uma sequência de números[2]:

```
1  (define (make-client request-port)
2    (define response-port (make-channel))
3    (define (send-sequence start end)
4      (for ([i (in-range start end)])
5        (async-channel-put request-port
6                           (make-request-obj i response-port))
7        (println (channel-get response-port))
8        (random-sleep)))
9    (lambda (m)
10     (cond ((eq? m 'send-sequence) send-sequence)
11           (else (error "Unknown method")))))
```

- Cada cliente encapsula sua própria porta de resposta
- O método `send-sequence` gera uma sequência de requisições, encaminha na porta de requisição e dorme por um tempo aleatório
- Após acordar, aguarda na porta de resposta

[2] Podemos usar essa mesma infra-estrutura para criar clientes mais complexos, com várias operações

Servidor de quadrados

Objetos de servidores

- O servidor também pode ser modelado como um objeto:

```
1  (define (make-square-server)
2    (define request-port (make-async-channel))
3    (define (serve)
4      (let loop ([request (async-channel-get request-port)])
5        (let ([msg ((request 'get-message))]
6              [response-port ((request 'get-response-port))])
7          (channel-put response-port (* msg msg))
8          (loop (async-channel-get request-port)))))
9    (define (get-request-port)
10     request-port)
11   (lambda (m)
12     (cond ((eq? m 'serve) serve)
13           ((eq? m 'get-request-port) get-request-port)
14           (else error "Unknown method"))))
```

- O servidor encapsula sua porta de requisição, que pode ser obtida via `get-request-port`
- O método `serve` faz o servidor entrar num loop infinito, aguardando requisições na porta de requisição
- Quando recebe uma requisição, extrai o número a processar e a porta de resposta
- Em seguida, envia o resultado (quadrado) pela porta de resposta e aguarda nova requisição

Servidor de quadrados

Simulação da comunicação

- A `main` cria servidor e clientes
- Cada cliente gera 10 números e envia ao servidor

```
1  (define (main args)
2    (define server (make-square-server))
3    (define request-port ((server 'get-request-port)))
4
5    (define srv (thread (lambda () ((server 'serve)))))
6    (define c1 (thread
7                 (lambda ()
8                   (define client (make-client request-port))
9                   ((client 'send-sequence) 0 10))))
10   (define c2 (thread
11                (lambda ()
12                  (define client (make-client request-port))
13                  ((client 'send-sequence) 10 20))))
14   (define c3 (thread
15                (lambda ()
16                  (define client (make-client request-port))
17                  ((client 'send-sequence) 20 30))))
18
19   (map thread-wait (list srv c1 c2 c3)))
20
21 (main (current-command-line-arguments))
```

Servidor de quadrados

Teste de execução

```
  $ racket squarer-server.rkt
400
100
0
121
441
1
4
144
484
9
16
169
529
25
196
225
36
576
256
625
289
49
324
64
361
676
81
729
784
841
```

Servidor de operações aritméticas

Servidor

- Vamos generalizar o servidor para realizar mais de uma operação
- A nova versão realiza as quatro operações aritméticas:

```
1  (define (make-arithm-server)
2    (define request-port (make-async-channel))
3    (define (serve)
4      (let loop ([request (async-channel-get request-port)])
5        ; request demarshaling
6        (define response-port ((request 'get-resp)))
7        (define op ((request 'get-op)))
8        (define arg1 ((request 'get-arg1)))
9        (define arg2 ((request 'get-arg2)))
10       ; dispatcher
11       (cond [(eq? op '+) ; suppose addition is slow
12              (channel-put response-port (+ arg1 arg2))
13              (println "Heavy long processing started...")
14              (long-sleep)
15              (println "Heavy long processing finished...")]
16             [(eq? op '-) (channel-put response-port (- arg1 arg2))]
17             [(eq? op '*) (channel-put response-port (* arg1 arg2))]
18             [(eq? op '/) (channel-put response-port (/ arg1 arg2))])
19       (loop (async-channel-get request-port))))
20   (define (get-request) request-port)
21   (lambda (m)
22     (cond ((eq? m 'serve) serve)
23           ((eq? m 'get-request) get-request)
24           (else error "Unknown method"))))
```

- Note que agora o dispatcher do servidor lida com as quatro operações
- O objeto de requisição também ficou mais complexo, com três itens de dados

Servidor de operações aritméticas

Objeto de requisição e cliente

- O objeto de requisição é uma estrutura simples:

```
1  (define (make-request-obj op arg1 arg2 resp)
2    (lambda (m)
3      (cond ((eq? m 'get-op) (lambda () op))
4            ((eq? m 'get-arg1) (lambda () arg1))
5            ((eq? m 'get-arg2) (lambda () arg2))
6            ((eq? m 'get-resp) (lambda () resp))
7            (else (error "Unknown method")))))
```

- O objeto cliente é especializado para uma única operação (`op`) e envia `qty` mensagens:

```
1  ; a client request only one a specific operation
2  (define (make-client request-port op name)
3    (define response-port (make-channel))
4    (define (send-sequence qty)
5      (for ([i (in-range qty)])
6        (async-channel-put
7          request-port
8          (make-request-obj op
9                            i
10                           (add1 i)
11                           response-port))
12       (printf "~a sent ~a ~a ~a\n" name op i (add1 i))
13       (printf "~a received ~a\n" name (channel-get response-port))))
14   (lambda (m)
15     (cond [(eq? m 'send-sequence) send-sequence]
16           [(eq? m 'get-name) name]
17           (else (error "Unknown method")))))
```

Programa principal

```
(define (main args)
  (cond [(< (vector-length args) 1)
         (displayln "Usage: arithmetic-server <qty>"
                    (current-error-port))
         (exit 1)])
  (define qty (string->number (vector-ref args 0)))
  (define server (make-arithm-server))
  (define request ((server 'get-request)))

  (define srv (thread (lambda () ((server 'serve)))))
  (define add-client
    (thread
      (lambda ()
        (let ([client (make-client request '+ "add-c")])
          ((client 'send-sequence) qty)))))
  (define sub-client
    (thread
      (lambda ()
        (let ([client (make-client request '- "sub-c")])
          ((client 'send-sequence) qty)))))
  (define div-client
    (thread
      (lambda ()
        (let ([client (make-client request '/ "div-c")])
          ((client 'send-sequence) qty)))))
  (define mult-client
    (thread
      (lambda ()
        (let ([client (make-client request '* "mult-c")])
          ((client 'send-sequence) qty)))))
  (map thread-wait (list srv
                         add-client
                         sub-client
                         div-client
                         mult-client)))
```

Servidor de operações aritméticas

Teste de execução

```
$ racket arithmetic-server.rkt 3
sub-c sent - 0 1
div-c sent / 0 1
mult-c sent * 0 1
mult-c received 0
mult-c sent * 1 2
add-c sent + 0 1
div-c received 0
div-c sent / 1 2
mult-c received 2
mult-c sent * 2 3
"Heavy long processing started..."
add-c received 1
add-c sent + 1 2
sub-c received -1
sub-c sent - 1 2
"Heavy long processing finished..."
div-c received 1/2
div-c sent / 2 3
"Heavy long processing started..."
add-c received 3
mult-c received 6
add-c sent + 2 3
"Heavy long processing finished..."
sub-c received -1
sub-c sent - 2 3
"Heavy long processing started..."
add-c received 5
div-c received 2/3
"Heavy long processing finished..."
sub-c received -1
```

Servidor de operações aritméticas

Requisições concorrentes

- Por ora, o servidor só consegue processar uma requisição por vez
  - Isso é perceptível pelas “pausas” quando executamos as operações lentas
- As outras ficam esperando enfileiradas na porta (canal de requisição)
- Podemos adotar uma arquitetura mais flexível:
  - Servidor atua apenas como despachador
  - Ao receber uma requisição o trabalho é despachado para uma thread trabalhadora
  - A thread trabalhadora, ao terminar, envia a resposta ao cliente
- Nessa arquitetura, a thread do servidor fica liberada na maior parte do tempo
- Aproveita-se melhor o tempo do computador (i.e. no caso de operações bloqueantes)

Servidor de operações aritméticas

Requisições concorrentes

- Para facilitar, podemos fatorar as operações do servidor em amarrações locais:

```
1  (define (add arg1 arg2 response-port)
2    (channel-put response-port (+ arg1 arg2))
3    (println "Heavy long processing...")
4    (random-sleep))
5  (define (sub arg1 arg2 response-port)
6    (channel-put response-port (- arg1 arg2)))
7  (define (mult arg1 arg2 response-port)
8    (channel-put response-port (* arg1 arg2)))
9  (define (div arg1 arg2 response-port)
10   (channel-put response-port (/ arg1 arg2)))
```

- O procedimento dispatch cria uma thread trabalhadora com uma thunk:

```
1  (define (dispatch proc)
2    (thread proc))
```

Servidor de operações aritméticas

Requisições concorrentes

- O servidor despacha as operações criando thunks:

```
1     ; dispatcher
2     (cond [(eq? op '+) ; suppose addition is slow
3            (dispatch
4              (lambda () (add arg1 arg2 response-port)))]
5           [(eq? op '-)
6            (dispatch
7              (lambda () (sub arg1 arg2 response-port)))]
8           [(eq? op '*)
9            (dispatch
10             (lambda () (mult arg1 arg2 response-port)))]
11          [(eq? op '/)
12           (dispatch
13             (lambda () (div arg1 arg2 response-port)))])
14    (loop (async-channel-get request-port))))
```

- Essas medidas deixam o servidor mais “responsivo”

Servidor de operações aritméticas

Requisições concorrentes: teste de execução

```
$ racket arithmetic-server-multithreaded.rkt 3
sub-c sent - 0 1
div-c sent / 0 1
add-c sent + 0 1
mult-c sent * 0 1
mult-c received 0
mult-c sent * 1 2
sub-c received -1
sub-c sent - 1 2
"Heavy long processing started..."
div-c received 0
div-c sent / 1 2
add-c received 1
mult-c received 2
mult-c sent * 2 3
sub-c received -1
sub-c sent - 2 3
div-c received 1/2
div-c sent / 2 3
add-c sent + 1 2
sub-c received -1
"Heavy long processing started..."
mult-c received 6
add-c received 3
add-c sent + 2 3
div-c received 2/3
"Heavy long processing started..."
add-c received 5
"Heavy long processing finished..."
"Heavy long processing finished..."
"Heavy long processing finished..."
```

Servidor de operações aritméticas

Requisições paralelas

- Supondo que as operações fossem caras, poderíamos delegar o trabalho para places
- Nesse caso, as operações são executadas paralelamente
- Para isso, precisamos redefinir as operações do servidor para usar places:

```
1  (define (add arg1 arg2 response-port)
2    (dispatch-place "add-plc-worker.rkt"
3                    arg1
4                    arg2
5                    response-port))
6  (define (sub arg1 arg2 response-port)
7    (dispatch-place "sub-plc-worker.rkt"
8                    arg1
9                    arg2
10                   response-port))
11 (define (mult arg1 arg2 response-port)
12   (dispatch-place "mult-plc-worker.rkt"
13                   arg1
14                   arg2
15                   response-port))
16 (define (div arg1 arg2 response-port)
17   (dispatch-place "div-plc-worker.rkt"
18                   arg1
19                   arg2
20                   response-port))
```

Servidor de operações aritméticas

Requisições paralelas

- O procedimento dispatch-place cria uma place trabalhadora:

```
1 (define (dispatch-place worker arg1 arg2 response-port)
2     (let ([plc (dynamic-place worker 'main)])
3       (place-channel-put plc (list arg1 arg2))
4       (place-wait plc)
5       (let ([res (place-channel-get plc)])
6         (channel-put response-port res))))
```

- O servidor irá despachar as places com uma thread, de modo que a obtenção da resposta pelo canal do place seja assíncrona:

```
1 (define (dispatch thunk)
2     (thread thunk))
```

- O dispatcher do servidor permanece como na última versão que vimos

Servidor de operações aritméticas

Requisições paralelas

- Os place workers são simples, com a mesma estrutura[3] (e.g. o que faz subtração):

```
1  #lang racket
2
3  (provide main)
4
5  (define (main plc-chan)
6    (let ([message (place-channel-get plc-chan)])
7
8      (place-channel-put
9        plc-chan
10       (- (first message)
11            (second message)))))
```

- Vamos simular o cenário em que a worker que efetua adição é mais lenta (faz um processamento mais pesado):

```
1  #lang racket
2  (provide main)
3  (define (long-sleep)
4    (sleep 5))
5  (define (main plc-chan)
6    (let ([message (place-channel-get plc-chan)])
7      (println "Long processing started...")
8      (long-sleep)
9      (println "Long processing finished...")
10     (place-channel-put
11       plc-chan
12       (+ (first message)
13            (second message)))))
```

Servidor de operações aritméticas

Requisições paralelas: teste de execução

```
$ racket arithmetic-server-par-place.rkt 3
sub-c sent - 0 1
div-c sent / 0 1
mult-c sent * 0 1
add-c sent + 0 1
mult-c received 0
mult-c sent * 1 2
div-c received 0
div-c sent / 1 2
"Long processing started..."
sub-c received -1
sub-c sent - 1 2
div-c received 1/2
div-c sent / 2 3
mult-c received 2
mult-c sent * 2 3
sub-c received -1
sub-c sent - 2 3
sub-c received -1
div-c received 2/3
mult-c received 6
"Long processing finished..."
add-c received 1
add-c sent + 1 2
"Long processing started..."
"Long processing finished..."
add-c received 3
add-c sent + 2 3
"Long processing started..."
"Long processing finished..."
add-c received 5
```

Tema extra: cliente-servidor em rede

Cliente-servidor TCP

- Podemos generalizar o modelo cliente-servidor para comunicação inter-processo
- Substituímos canais por portas de rede, e a comunicação se dá por protocolos
- Em Racket, podemos facilmente utilizar o protocolo TCP para definir um servidor de eco:

```
(define (make-server)
  (define (serve port-number)
    (let ([listener
           (tcp-listen port-number
                       10
                       #t)])
      (printf "Listening on
               localhost:~a...\n"
               port-number)
      (let loop ()
        (accept-client listener)
        (loop))))
  (define (accept-client listener)
    (let-values ([(in out)
                  (tcp-accept listener)])
      (handle-connection in out)
      (close-input-port in)
      (close-output-port out)))
  (define (handle-connection in out)
    (let ([message (read in)])
      (println message)
      (write message out)
      (flush-output out)))
  (lambda (m)
    (cond [(eq? m 'serve)
           (lambda (port-number)
             (serve port-number))])))
```

- Servidor espera conexões na porta (`tcp-listen`)
- Quando uma conexão é feita (`tcp-accept`) obtém-se uma porta de entrada e outra se saída (`in` e `out`)
- em `handle-connection`, o servidor recebe a mensagem e ecoa a mensagem para o cliente

Cliente-servidor TCP

- O cliente também é simples, recebe argumentos via linha de comando, envia mensagem, recebe o eco e o imprime

```
1  #lang racket
2
3  (define (main args)
4    (cond [(< (vector-length args) 2)
5           (displayln "Usage: tcp-client <port> <message>"
6                      (current-error-port))
7           (exit 1)])
8    (define port-number (string->number (vector-ref args 0)))
9    (define message (vector-ref args 1))
10   (let-values ([(in out) (tcp-connect "localhost" port-number)])
11     (write message out)
12     (flush-output out)
13     (println (read in))
14     (close-input-port in)
15     (close-output-port out)))
16
17 (main (current-command-line-arguments))
```

- Testes:

```
$ racket tcp-server.rkt 8081
Listening on localhost:8081...
"hello server"
"eco eco eco"
```

```
$ racket tcp-client.rkt 8081 "hello server"
"hello server"
$ racket tcp-client.rkt 8081 "eco eco eco"
"eco eco eco"
```

Servidor de operações aritméticas TCP

- Podemos adaptar o nosso servidor de operações aritméticas paralelo para operar em rede
- Basta substituirmos o canal de entrada (`request-port`) por uma porta TCP (via `tcp-listen`):

```
1  (define (make-arithm-server)
2    (define (serve port-number)
3      (let ([listener (tcp-listen port-number 10 #t)])
4        (printf "Listening on localhost:~a...\n"
5                port-number)
6        (let loop ()
7          (accept-client listener)
8          (loop))))
9    (define (accept-client listener)
10     (let-values ([(in out) (tcp-accept listener)])
11       (handle-connection in out)))
```

- O método `handle-connection` abre a mensagem e despacha a operação:

```
1  (define (handle-connection in out)
2    (let* ([message (read in)]
3           [op (first message)]
4           [arg1 (string->number (second message))]
5           [arg2 (string->number (third message))])
6      (cond [(string=? op "+")
7             (dispatch
8               (lambda () (add arg1 arg2 in out)))]
9            [(string=? op "-")
10            (dispatch
11              (lambda () (sub arg1 arg2 in out)))]
12           (...)
13           [else
14            (write "Unknown operation" out)])))
```

Servidor de operações aritméticas TCP

- As operações mantém-se as mesmas (i.e. iniciam as mesmas place workers), porém agora recebem as portas de entrada e saída TCP:

```
1  (define (add arg1 arg2 in out)
2    (dispatch-place "add-plc-worker.rkt"
3                    arg1
4                    arg2
5                    in out))
6  (define (sub arg1 arg2 in out)
7    (dispatch-place "sub-plc-worker.rkt"
8                    arg1
9                    arg2
10                   in out))
11 (define (mult arg1 arg2 in out)
12   (dispatch-place "mult-plc-worker.rkt"
13                   arg1
14                   arg2
15                   in out))
16 (define (div arg1 arg2 in out)
17   (dispatch-place "div-plc-worker.rkt"
18                   arg1
19                   arg2
20                   in in out))
```

Servidor de operações aritméticas TCP

- dispatch-place inicia a respectiva place-worker e envia o resultado para a porta de saída TCP:

```
(define (dispatch-place worker arg1 arg2 in out)
    (let ([plc (dynamic-place worker 'main)])
      (place-channel-put plc (list arg1 arg2))
      (place-wait plc)
      (let ([res (place-channel-get plc)])
        (write res out)
        (flush-output out)
        (close-input-port in)
        (close-output-port out))))
```

- O cliente simplesmente recebe os dados como argumentos, envia requisição e recebe resposta do servidor:

```
(define (main args)
  (cond [(< (vector-length args) 4)
         (displayln "Usage: arithm-client-tcp-repl <port> <oper> <arg1> <arg2>"
                    (current-error-port))
         (exit 1)])
  (define port-number (string->number (vector-ref args 0)))
  (define oper (vector-ref args 1))
  (define arg1 (vector-ref args 2))
  (define arg2 (vector-ref args 3))

  (let-values ([(in out) (tcp-connect "localhost" port-number)])
    (write (list oper arg1 arg2) out)
    (flush-output out)
    (println (read in))
    (close-input-port in)
    (close-output-port out)))

(main (current-command-line-arguments))
```

Servidor de operações aritméticas TCP

Testes

- Servidor:

```
$ racket arithmetic-server-parallel-tcp.rkt 8081
Listening on localhost:8081...
"Long processing started..."
"Long processing finished..."
"Long processing started..."
"Long processing finished..."
```

- Cliente:

```
1  $ racket arithmetic-client-tcp.rkt 8081 "+" 1 2
2  3
3  $ racket arithmetic-client-tcp.rkt 8081 "+" 1 2 &
4  $ racket arithmetic-client-tcp.rkt 8081 "-" 3 4
5  -1
6  $ 3
7  [1]+  Done racket arithmetic-client-tcp.rkt 8081 "+" 1 2
8  $ racket arithmetic-client-tcp.rkt 8081 "*" 5 6
9  30
10 $ racket arithmetic-client-tcp.rkt 8081 "/" 6 2
11 3
12 $ racket arithmetic-client-tcp.rkt 8081 "/" 5 7
13 5/7
```

  - Note que na linha 3 exploramos a possibilidade de dois clientes em paralelo (colocamos um cliente em background e iniciamos outro em seguida)
  - O servidor atendeu ambos em paralelo (o segundo envolve processamento mais rápido), enviando a resposta posteriormente (linha 6) e o cliente em background pôde completar (linha 7)

Exercício

Exercício

No final dos slides de teoria da semana 9, foram apresentados dois exercícios envolvendo o processamento de imagens em Racket (conversão para tons de cinza e separação de canais de cores).
Você irá implementar uma solução para esses problemas no modelo cliente-servidor utilizando passagem de mensagens. O servidor possui duas operações:

- Uma operação que recebe uma imagem colorida e retorna a mesma imagem, porém em grayscale
- Uma operação que recebe uma imagem colorida e retorna a mesma imagem com um dos canais de cores isolado

Cada operação deve ser efetuada por um place específico.
O processamento envolve então os seguintes passos:

1. Cada cliente (definir ao menos 3) lê uma imagem do sistema de arquivos e envia a imagem ao servidor, junto com a operação a efetuar
2. O servidor efetua a operação, envia a nova imagem como resposta ao cliente
3. O cliente recebe a imagem e salva no sistema de arquivos com um novo nome

Bônus (valendo pontos adicionais): implementar a comunicação cliente-servidor via TCP.

Estudo de caso: modelo cliente-servidor

Aplicação de passagem de mensagens

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

07 de agosto de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

Crédito de parte das imagens, a menos se especificado: Wikipedia