Revendo o Modelo: Memória Partilhada

- A Aplicação é uma coleção de threads (podem ser criadas dinâmicas)
- Cada thread tem um conjunto de vars Privadas e Partilhadas
    - As threads comunicam-se implicitamente através de operações nas variáveis partilhadas

- PTHREADS Um padrão POXIS Portátil, mas pesado e lento
- OPENMP Suporta programação científica
- TBB (Thread Building Block) Biblioteca de modelos c++ para programação paralela da intel
- CILK e CILK++ Linguagens para programação paralela baseada em c e c++ com estrutura de threads menos pesada que PTHREADS
- Em java Objetos baseados nos threads POSIX

Criação de Threads POSIX

Assinatura:

```
int pthread_create(pthread_t *,                 // identificador do thread
                   const pthread_attr_t *,      // atributos do thread
                   void * (*)(void *),          // função a executar
                   void *);                     // parâmetro para a função
```

```
errcode = pthread_create(&threads[i], NULL, SumValues, (void *)i);
```

- Se o segundo parâmetro for NULL usa Atributos pré-definido (exemplo: Memoria mínima, prioridade etc)
- Errcode != de 0 se a criação da thread falhar

Join()

Em PTHREADS

```
pthread_join(threads[i], (void **)&value);
total += value;
```

Em java

```
thrd[threadcount].join();
```

- Join espera pelo final da execução da thread especificado
  - Pode receber o valor de retornos (Em PTHreads) e é passado pela instrução _Exit(void *value)
  - Pode incluir um timeout em milissegundos (java)

Locks(Mutexes)

Para criar um lock ou mutex:

```
pthread_mutex_t umMutex = PTHREAD_MUTEX_INITIALIZER;
```

Para usar:

```
int pthread_mutex_lock(umMutex);
…
int pthread_mutex_unlock(umMutex);
```

Deadlock (é uma situação em que duas ou mais threads ficam bloqueadas indefinidamente, aguardando uns pelos outros para liberar recursos)

```
thread 1:   lock(a)      thread 2:   lock(b)
            lock(b)                  lock(a)
```

Programação Paralela em OpenMP

É usada como alternativa a programação com threads e é uma especificação aberta para processamento paralelo

- O padrão OpenMP permite
  - Separar o programa em regiões sequencias e regiões paralelas
  - Confiar na gestão automática de memoria
  - Usar mecanismos de sincronização já disponíveis
- O padrão OpenMP NÃO permite
  - Paralismo automático
  - Garantir o melhor desempenho
  - Evitar por completo inconsistências no acesso partilhado de dados

OpenMP: Ciclos

O OpenMP paraleliza os ciclos de maneira simples, requer que não haja dependência de dados entre iterações.

Dependência: pares ler/escrever ou escrever/escrever

O pré-processador calcula os ciclos para cada thread diretamente do código fonte sequencial:

```
#pragma omp parallel for
for (i=0; i<20; i++)
{
  printf("Olá");
}
```

Diferenças na Partilha de Dados

- A programação paralela usa dados partilhados e dados privados
  - Privados: Visíveis apenas numa threads
  - Partilhados: Visíveis por todas threads

- Em PTHREADS
  - Vars Globais são partilhadas
  - Vars Locais as funções são privadas
- No OpenMP
  - Vars Shared São partilhadas
  - Vars Private São Privadas

Sincronização Em OpenMP

❑ Definição de seções críticas
■ Com ou sem nomes
■ Sem *locks/mutexes* explícitos
	`#pragma omp critical`
	`{`
	`  // Código crítico aqui`
	`}`

❑ Diretivas **`barrier`** (barreira)	`#pragma omp barrier`

❑ Funções de *lock* explícitas
■ Usadas como último recurso
	`omp_set_lock( lock 1 );`
	`  // Código aqui`
	`omp_unset_lock( lock 1 );`

❑ Regiões de *thread* único dentro de regiões paralelas
■ Diretivas **`master`**, **`single`**
	`#pragma omp single`
	`{`
	`  // executado somente uma vez`
	`}`

Resumo OpenMP

É Uma técnica baseada na compilação para criar código concorrente(paralelo) a partir de código principalmente sequencial

Pode de forma simples permitir a paralelização de código cíclico

O OpenMP através de benchmarks demonstrou um desempenho próximo de código manualmente configurado para multithreading (Escalável e Portável)

Não é a solução para todo o tipo de aplicações

Estilos de Arquiteturas

Um estilo é formulado em termos:

- Componentes com interfaces bem definidas
- A forma das ligações dos componentes
- Os dados trocados entre componentes
- como estes componentes e conectores são configurados conjuntamente num sistema.

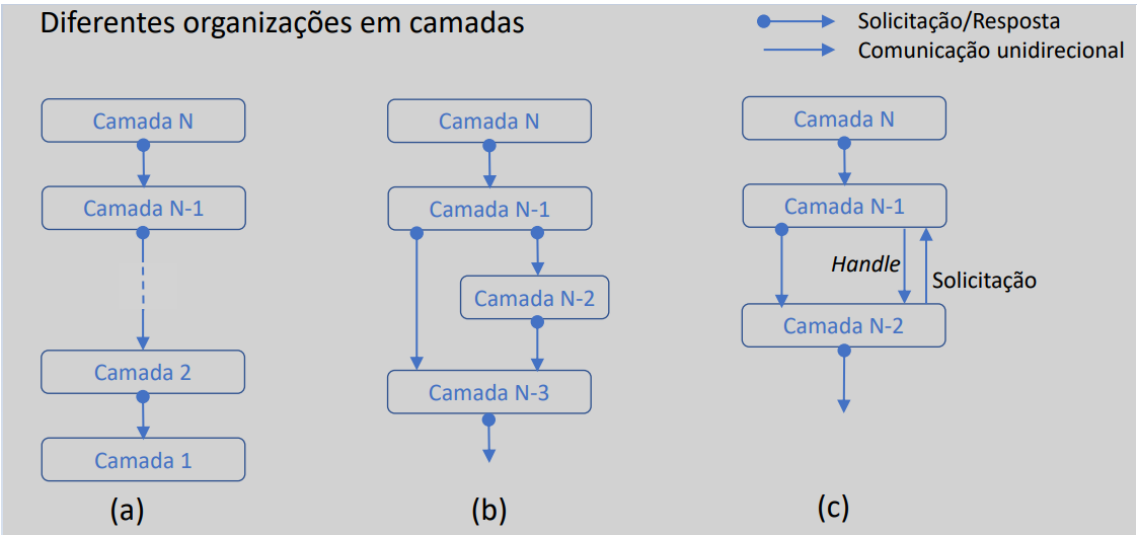
Conector

Um mecanismo que media a comunicação, a coordenação ou a cooperação entre os componentes. Exemplo: instalações para chamada de procedimento (remota), mensagens, ou streaming.

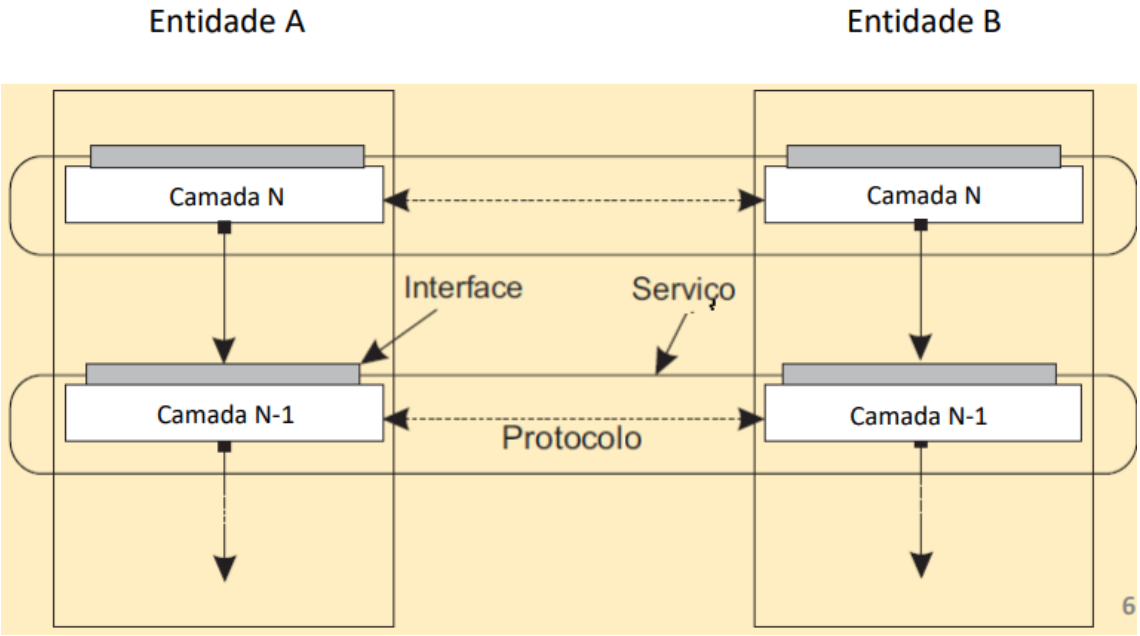
Existem os seguintes tipos de estilos:

- Camadas
  - Camadas Aplicacionais
- baseadas em objetos e serviços
- baseadas em recursos
- editor-assinante (publisher-subscriber)
- Wrapper
- Organizações centralizadas
  - Arquiteturas de sistema centralizadas
  - Arquiteturas de sistema multi-níveis (multitiered)
- Organizações descentralizadas: sistemas peer-to-peer
- Híbridas

Camadas



Exemplo protocolo de comunicação



Comunicação entre 2 intervenientes

Servidor

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept() # retorna um
          novo socket e o endereço cliente
4 while True: # para sempre
5 data = conn.recv(1024) # recebe dados do
                          cliente
6 if not data: break # pára se o cliente
                       parar
7 conn.send(str(data)+"*") # envia os dados
                         recebidos e um "*"
8 conn.close() # fecha a conexão
```

Cliente

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # conecta-se ao
      servidor (bloqueia enquanto aguarda)
4 s.send('Olá mundo') # envia dados
5 data = s.recv(1024) # recebe resposta
6 print data # mostra o resultado
7 s.close() # fecha a conexão
```

Camadas Aplicacionais

Tradicionalmente dividido em 3 tipos de camadas

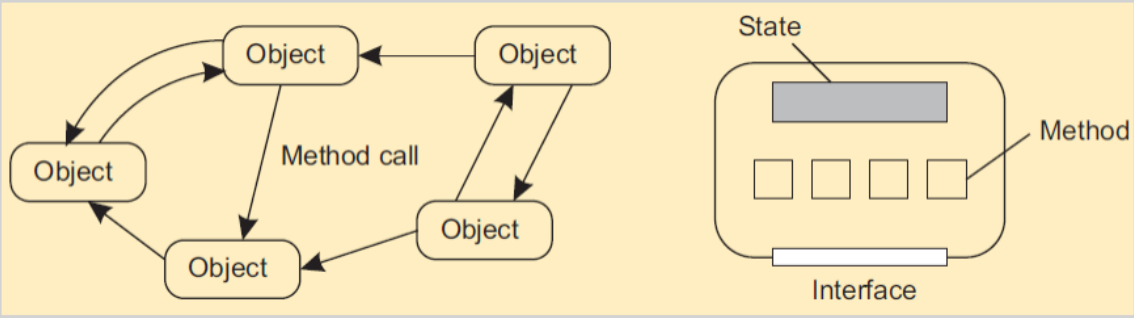
- Camada de interface
  - Contém unidade para interagir com os utilizadores ou aplicações externas.
- Camada de processamento
  - Contém as funções da aplicação.
- Camada de dados
  - Contém os dados que um cliente quer manipular.

Baseadas em objetos e serviços

Os componentes são objetos ligados entre si através de chamadas de procedimentos, estes componentes podem estar em diferentes máquinas.

Encapsulação

Diz-se que os objetos encapsulam dados e oferecem métodos sobre estes dados sem revelar a implementação interna.



Arquiteturas RESTful

Vista de um sistema distribuído como uma coleção de recursos, geridos individualmente por componentes. Os recursos podem ser adicionados, removidos, recuperados e modificados por aplicações (remotas).

Operações básicas:

- PUT Cria um recurso
- GET Retorna o estado atual de um recurso
- DELETE Elimina um recurso
- POST Modifica um recurso transferindo um novo estado

Exemplo Serviço Armazenamento Amazon

Os objetos (por exemplo, ficheiros) são colocados em contentores (por exemplo, diretorias).

SOAP vs. *RESTful*

Questão:

- Muitos preferem o *RESTful* porque a interface para um serviço é muito simples. O lado menos positivo é que muito precisa ser feito no espaço dos parâmetros.

- Interface SOAP da Amazon:

Operações sobre contentores	**Operações sobre objetos**
`ListAllMyBuckets`	`PutObjectInline`
`CreateBucket`	`PutObject`
`DeleteBucket`	`CopyObject`
`ListBucket`	`GetObject`
`GetBucketAccessControlPolicy`	`GetObjectExtended`
`SetBucketAccessControlPolicy`	`DeleteObject`
`GetBucketLoggingStatus`	`GetObjectAccessControlPolicy`
`SetBucketLoggingStatus`	`SetObjectAccessControlPolicy`

Sobre interfaces

Simplificações: Assume uma interface bucket que oferece operações pre - feitas (tipo create) para realizar diversas opções

SOAP:

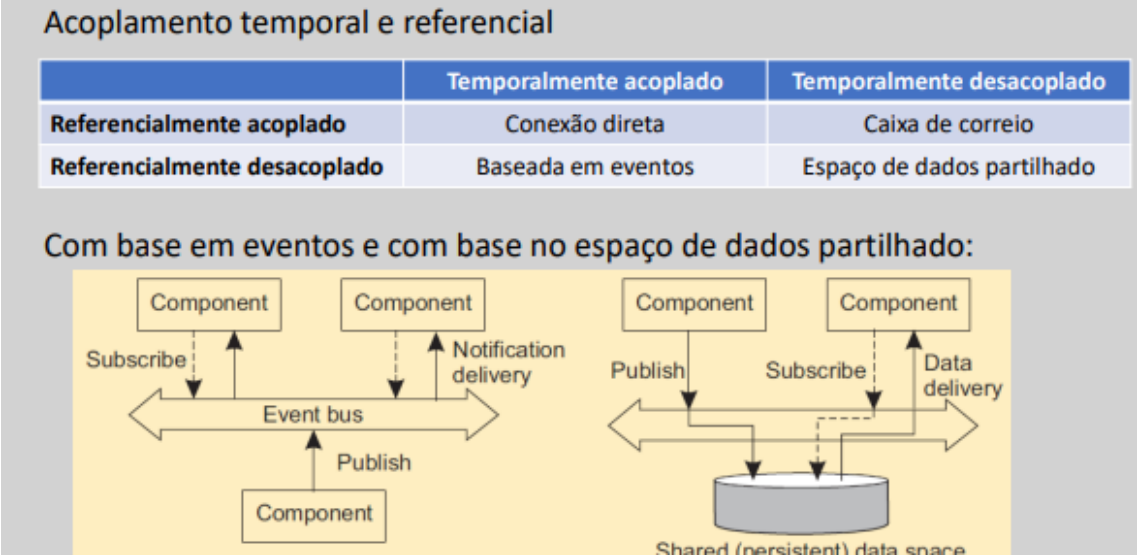
```
import bucket
bucket.create("mybucket")
```

- RESTful:

```
PUT http://mybucket.s3.amazonsws.com/
```

Arquiteturas editor-assinante (publisher-subscriber)

Nessa arquitetura, existem dois tipos principais de entidades: os editores (publishers) e os assinantes (subscribers). Os editores são responsáveis por produzir e enviar mensagens contendo informações, enquanto os assinantes são responsáveis por receber e processar essas mensagens.

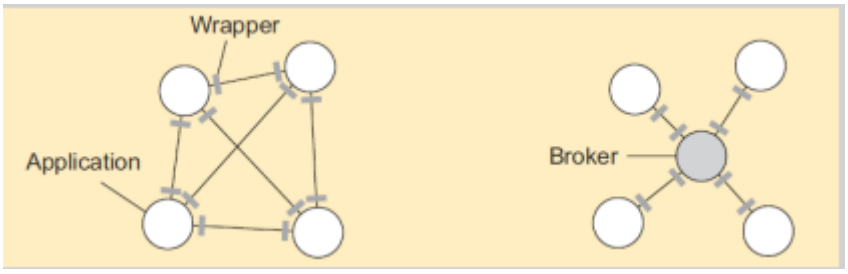


Wrapper (invólucro, envelope)

Um wrapper ou adaptador oferece uma interface aceitável para uma aplicação do cliente. As suas funções transformam-se nas disponíveis no componente

Pode ser em:

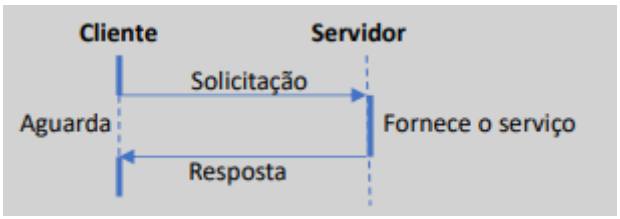
- 1-para-1
  - Para N aplicações requer N x (N – 1) = O(N2) adaptadores
- Através de um broker
  - Para N aplicações requer 2N = O(N) adaptadores



Arquiteturas de sistema centralizadas

Modelo básico cliente-servidor

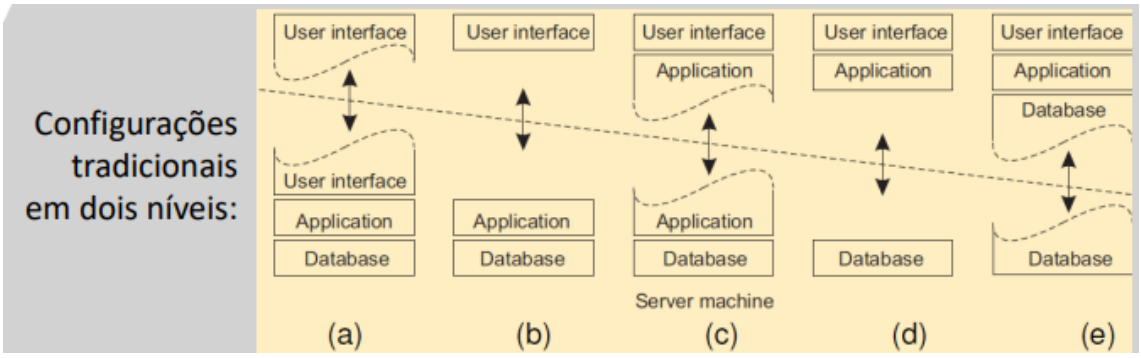
- Há processos a oferecer serviços (servidores)
- Há processos que usam serviços (clientes)
- Clientes e servidores podem estar na mesma máquina ou não
- Clientes seguem o modelo solicitação/resposta para usar serviços



Arquiteturas de sistema multi-níveis (multi-tiered)

Algumas organizações tradicionais:

- Nível único: configuração de terminal/computador central
- Dois níveis: configuração cliente/servidor único (servidor pode tmb ser cliente)
- Três níveis: cada camada em uma máquina separada



Organizações alternativas

Distribuição vertical:

- Vem da divisão de aplicações distribuídas em três níveis lógicos, e da execução dos componentes de cada nível em um servidor diferente (máquina).

Distribuição horizontal:

- Um cliente ou servidor pode estar fisicamente dividido em partes logicamente equivalentes, mas cada parte está a operar na sua própria parte do conjunto de dados completo.

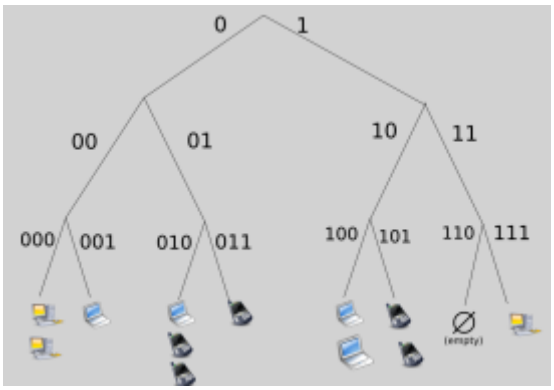
Arquiteturas peer-to-peer (P2P):

- Os processos são todos iguais: as funções que precisam de ser executadas são representadas por todos os processos; cada processo funcionará como cliente e servidor ao mesmo tempo.

P2P Estruturado

Em redes estruturadas peer-to-peer a estrutura lógica é organizada numa topologia específica, e o protocolo garante que qualquer nó pode pesquisar eficientemente a rede por um ficheiro/recurso, mesmo que o recurso seja extremamente raro.

Diagrama lógico para uma rede P2P estruturada, utilizando uma tabela de hashing distribuída (DHT) para identificar e localizar nós/recursos:



P2P não estruturado

Cada nó mantém uma lista ad hoc de vizinhos. A estrutura lógica resultante assemelha-se a um grafo aleatório: uma aresta existe somente com uma certa probabilidade P[<u,v>]

Pesquisa na rede P2P não estruturada:

- Inundação *(flooding):* o nó emissor ***u*** passa o pedido pelo recurso ***d*** a todos os vizinhos. O pedido é ignorado quando o nó recetor ***v*** já o tinha visto antes. Caso contrário, ***v*** procura localmente ***d*** (recursivamente). Pode ser limitado por um *Time-To-Live*: um número máximo de saltos *(hops)*.
- Caminhada aleatória *(random walk):* o nó emissor ***u*** passa o pedido pelo recurso ***d*** a um vizinho escolhido aleatoriamente, ***v***. Se ***v*** não tem ***d***, ele reencaminha o pedido a um dos seus vizinhos escolhido aleatoriamente, e assim por diante.
- Comparação: a caminhada aleatória é mais eficiente na comunicação, mas pode demorar mais tempo até encontrar o recurso.

Modelos híbridos para redes P2P

- São combinação de modelos peer-to-peer e cliente-servidor.
- É comum ter um servidor centrar que ajuda os pares a encontrarem-se.
- Existe uma variedade de modelos híbridos, com diversas relações custo/benefício entre a funcionalidade centralizada fornecida por uma rede estruturada de servidor/cliente e a igualdade de nó proporcionada pelas redes puramente P2P não estruturadas.
- Atualmente, os modelos híbridos têm um melhor desempenho do que redes puras e não estruturadas ou redes estruturadas puras

Modelo Skype: *A* deseja contatar *B*

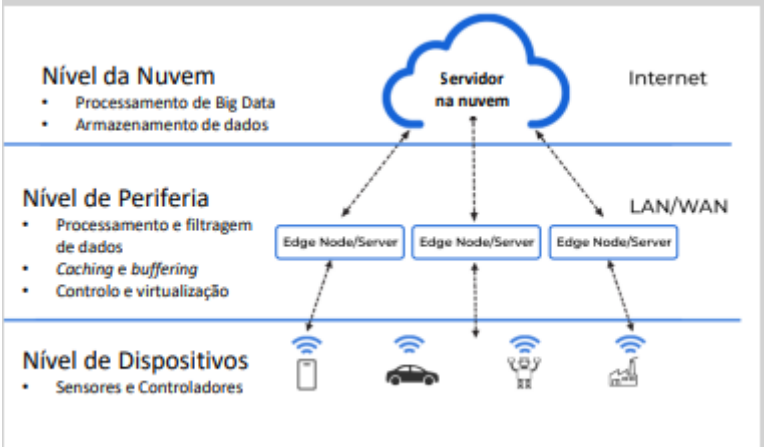
- **1. Tanto *A* quanto *B* estão na Internet pública:**
- Uma conexão TCP é estabelecida entre *A* e *B* para pacotes de controlo.
- A chamada real dá-se utilizando pacotes UDP entre portas negociadas

- **2. *A* opera atrás de um *firewall*, e *B* está na Internet pública:**
- *A* estabelece uma conexão TCP para pacotes de controlo com um servidor *S*
- *S* estabelece uma conexão TCP com *B* para repassar pacotes de controlo
- A chamada real dá-se utilizando pacotes UDP e diretamente entre *A* e *B*

- **3. Tanto *A* como *B* operam atrás de *firewalls***
- *A* conecta-se com um servidor *S* utilizando TCP
- *S* estabelece uma conexão TCP com *B*
- Para a chamada real, outro servidor *R* é contatado como estafeta (*relay*): *A* estabelece uma conexão com *R*, e *B* também o faz.
- Todo o tráfego de voz é direcionado sobre as duas conexões TCP, e através do servidor *R*.

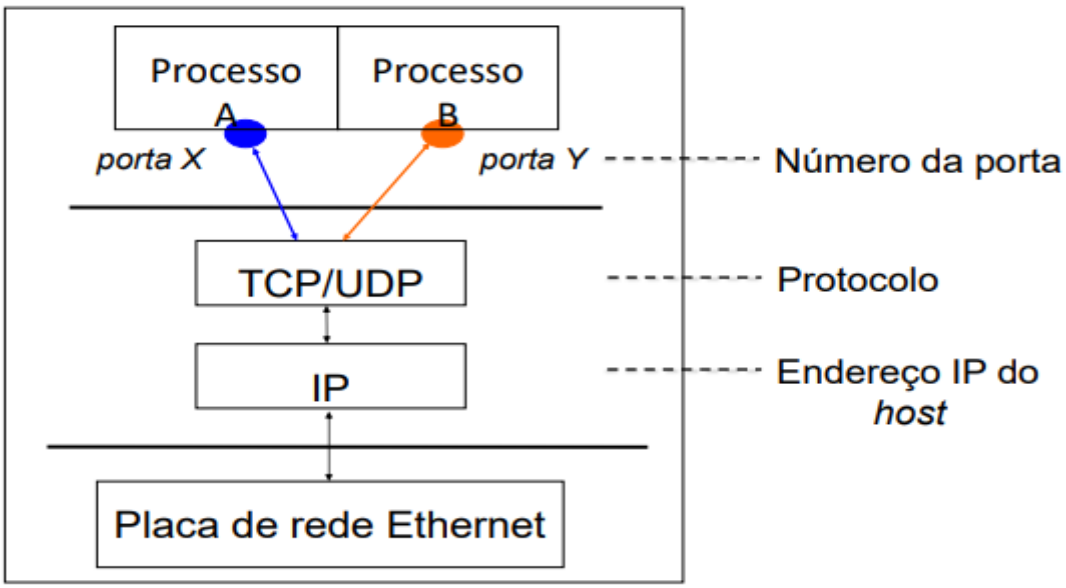
Edge computing

- A edge computing, ou computação de borda ou na periferia, é aquela na qual o processamento acontece no local físico (ou próximo) do cliente ou da fonte de dados.
- Com o processamento mais próximo, os utilizadores se beneficiam de serviços mais rápidos e fiáveis, e as empresas desfrutam da flexibilidade de usar e distribuir um conjunto de recursos por um grande número de locais.



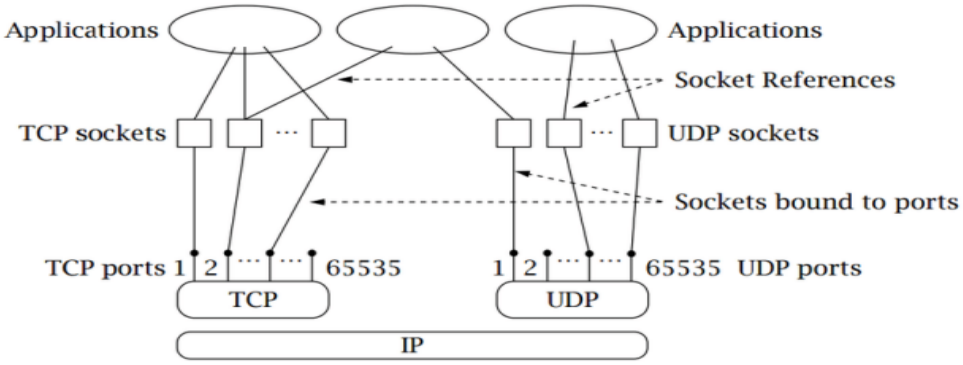
Sockets

- O Socket é a interface que o SO fornece para acesso ao seu subsistema de rede
- Em essência, a porta da casa (identifica a aplicação)
- Socket é uma API
    - Dá suporte a criação de aplicações de rede
- Tem um endereço de ip com 32 ou 128 bits
- O host pode ter muitos processos
- Um PortNumber identifica o processo e tem 16 bits



Tipos de comunicação entre processos

- Socket de datagrama(UDP)
  - Coleção de mensagens
  - Sem garantias, melhor esforço
  - Sem estabelecimento de sessão
- Socket de Strean(TCP)
  - Fluxo de bytes
  - Garantias de entrega e integridade
  - Estabelece uma sessão subjacente ao tráfego de dados



Clientes e Servidores

- Cliente
  - Inicia a comunicação
  - Tem de saber o ip e porta do sv
  - Solicita um serviço
- Servidor
  - Aguarda conexões
  - Obtém o ip e a porta do cliente na conexão
  - Fornece serviço

Aplicações populares têm portas bem conhecidas

e.g., porta 80 para servidor Web e porta 25 para e-mail

Ver http://www.iana.org/assignments/port-numbers

Portas bem conhecidas vx. Portas efêmeras

Os processos servidores têm uma porta bem conhecida

Entre 0 and 1023

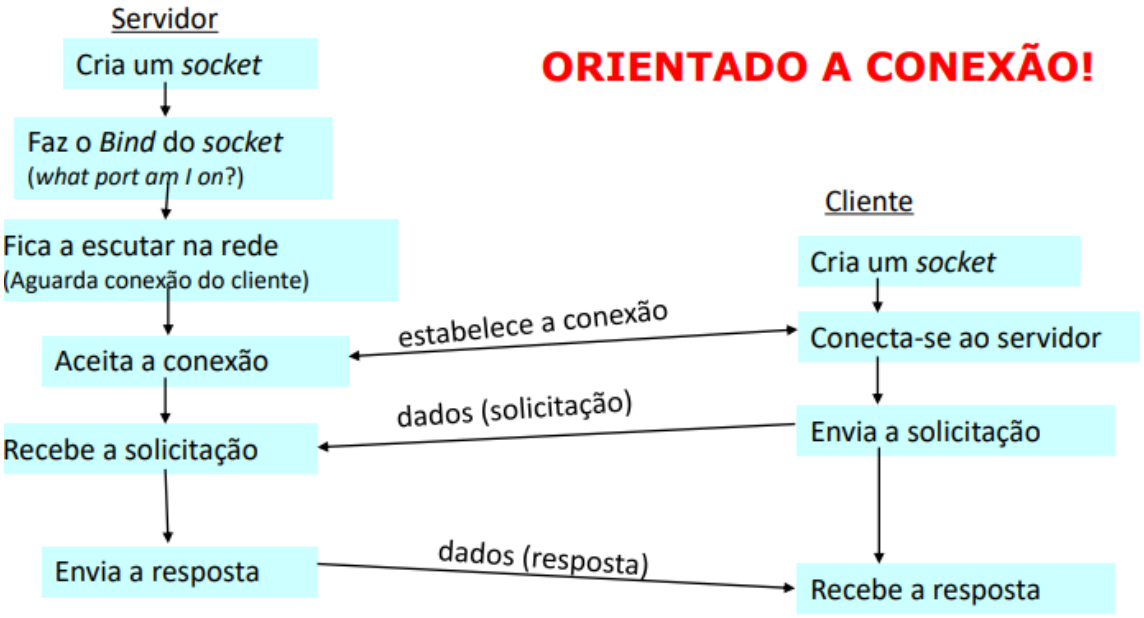
O cliente escolhe uma porta efêmera (temporária) não utilizada

Entre 1024 e 65535

Identifica univocamente o tráfego entre processos

Dois endereços IP e dois números de porta

Comunicação cliente-Servidor Com Sockets TCP

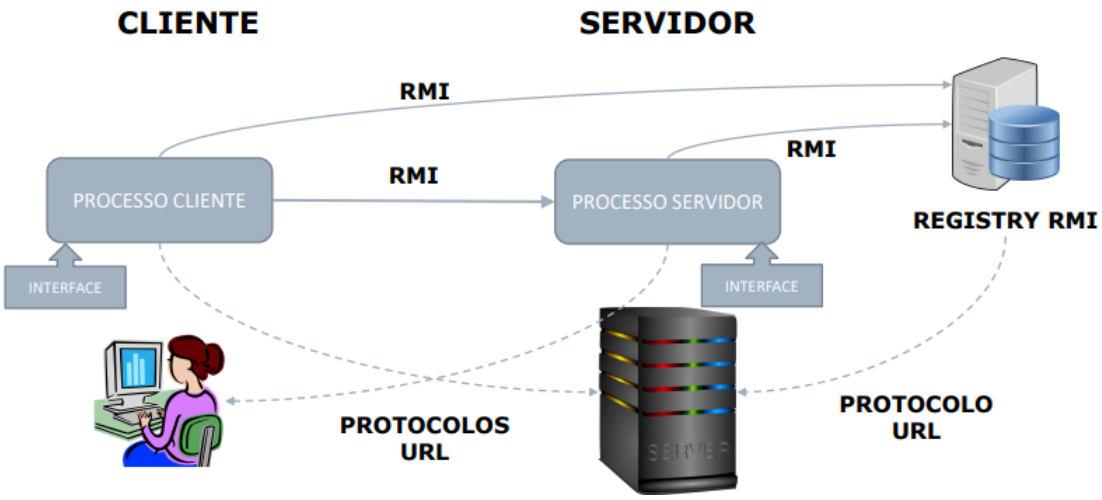


Comunicação Cliente-Servidor com Sockets UDP

Como Lidar com Múltiplos Clientes

- Criar um loop para aceitar conexões
- Após estabelecer uma conexão, lança outro processo (por exemplo com o fork()) que vai usar outra porta
- Indica ao cliente nova porta
- Voltar a ficar à espera de conexões

JAVA RMI



O Java RMI – Remote Method Invocation

O próximo passo evolutivo: grupos de máquinas **atuando como uma só**

- Uma coleção de computadores ligados em rede, não somente para compartilhar dados, mas para cooperar entre sí, distribuindo a carga computacional entre diversas máquinas físicas

A distribuição deve ser **transparente** para o usuário e para os programas, que devem apenas utilizar a **interface de serviço** de sistema

- Um usuário/programador não deveria necessitar saber que uma máquina remota está sendo utilizada
- Um sistema distribuído deveria parecer um sistema convencional para o usuário/programador

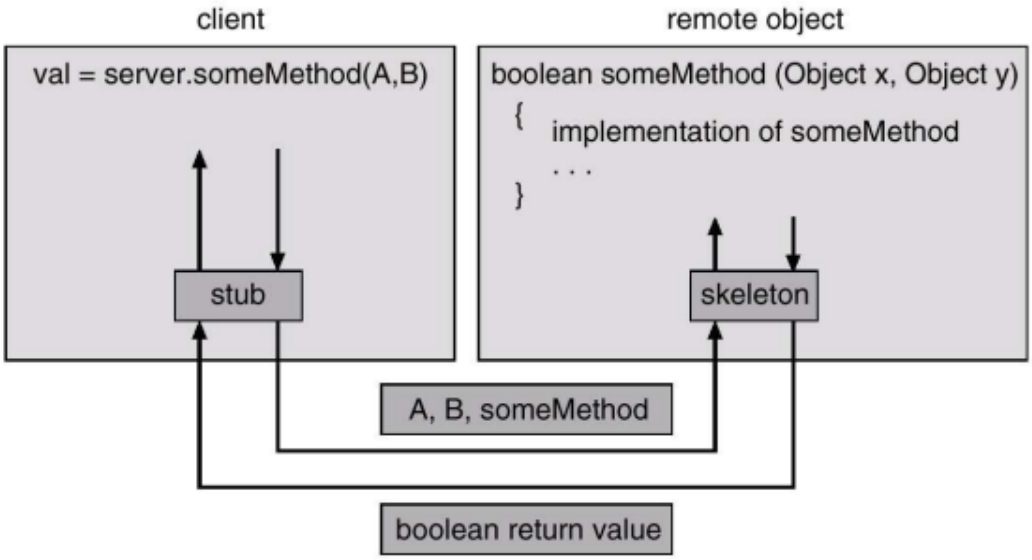
Funcionamento do RMI

RMI – Stub (Cliente)

- Transforma os parâmetros em formato independente de máquina (marshalling)
- Envia requisições para o objeto remoto através da rede passando o nome do método e os argumentos transformados

RMI- Esqueleto (Servidor)

- Recebe a requisição com o nome do método, decodifica (unmarshals) os parâmetros e os utiliza para chama o método no objeto remoto
- Transforma (marshals) os dados resultantes e devolve-os para o cliente



RPC VS RMI

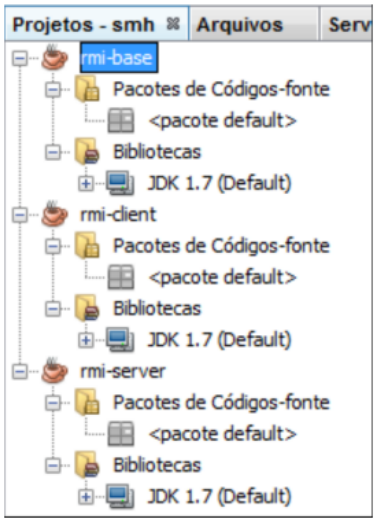
Rpc(Remote Procedure Call):

- Chamada procedural de um processo de uma máquina servidora a partir de um processo em uma máquina cliente
- Permitem que os procedimentos tradicionais sejam executados em múltiplas máquinas a partir de chamadas pela rede de comunicação

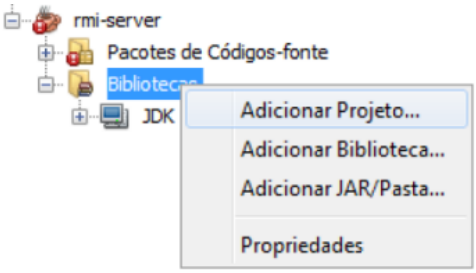
RMI(Remote Method Invocation):

- RPC para o ambiente orientado a objetos
- Chamada de métodos em objetos remotos

Implementação RMI



- **No projeto rmi-base devemos criar a interface de cada objeto remoto**
  - Estas interfaces devem estar disponíveis para a implementação da aplicação servidora e da aplicação cliente
  - A Interface deve estender **java.rmi.Remote**
  - A interface deve declarar todos os métodos visíveis remotamente
  - Todos os métodos devem declarar **java.rmi.RemoteException**



- **Configurar dependencias do projeto `rmi-server`**
  - O projeto **`rmi-server`** deve incluir o projeto **`rmi-base`** como biblioteca de dependência
  - Para tanto, clicar com o botão da direita do mouse no item ***bibliotecas*** do projeto e selecionar ***adicionar projeto...*** e selecionar projeto **`rmi-base`**

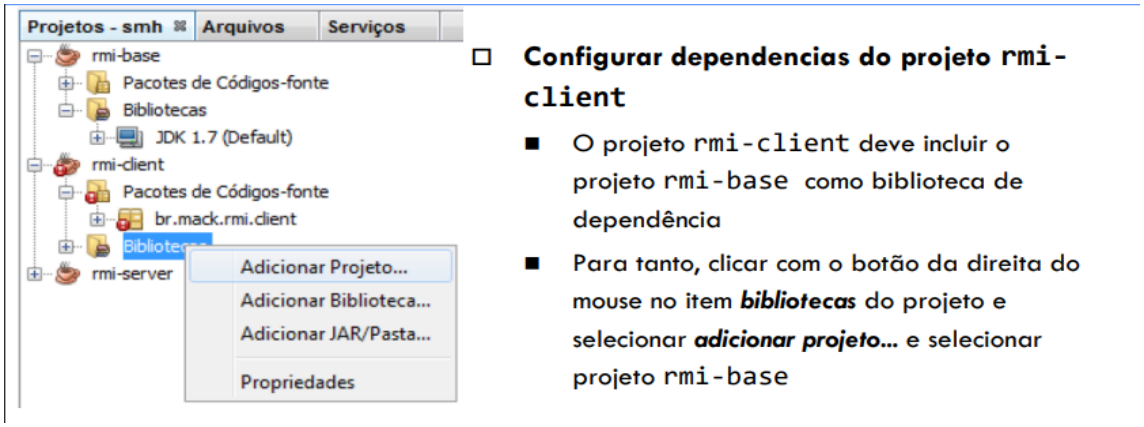
- **Implementar a aplicação servidora**
  - Definir um *SecurityManager* (obrigatório p/ download de código remoto)

```
import rmi.IExemplo;
import rmi.impl.ExemploImpl;
import java.rmi.RMISecurityManager;
import java.rmi.registry.*;
public class RmiServer {
  public static void main(String[ ] args) {
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new  RMISecurityManager());
    } // continua...
```

☐ Implementar a aplicação servidora

- Registrar os objetos remotos no *RMIRegistry* usando `Naming.bind()`

```
try {
  IExemplo exemplo = new ExemploImpl();
  Registry registry = LocateRegistry.createRegistry(1099);
  registry.rebind("exemplormi", exemplo);
  } catch (Exception e) {
    if (e instanceof RuntimeException)
    throw (RuntimeException) e;
    System.out.println("" + e);
    }
  }
}
```



☐ Configurar dependencias do projeto `rmi-client`

- O projeto `rmi-client` deve incluir o projeto `rmi-base` como biblioteca de dependência
- Para tanto, clicar com o botão da direita do mouse no item ***bibliotecas*** do projeto e selecionar ***adicionar projeto...*** e selecionar projeto `rmi-base`

☐ Implementar a aplicação cliente (`rmi-client`)

- Definir um *SecurityManager* e conectar-se ao *Registry* do servidor
- Recuperar os objetos remotos usando

`(Tipo do Objeto)registry.lookup()`

```
try {
  Registry registry =
    LocateRegistry.getRegistry("Server1");
  IExemplo exemplo = (IExemplo)
              registry.lookup("exemplormi");
  System.out.println(exemplo.metodo1());
  exemplo.metodo2("Adeus!");
  System.out.println(exemplo.metodo1());
} catch (Exception e) {
  if (e instanceof RuntimeException)
    throw (RuntimeException) e;
    System.out.println("" + e);
  }
}
```