SISTEMAS DISTRIBUÍDOS

João Leitão, Sérgio Duarte, Pedro Camponês

(baseado nos slides de Nuno Preguiça)

Aula 3: Capítulo 3

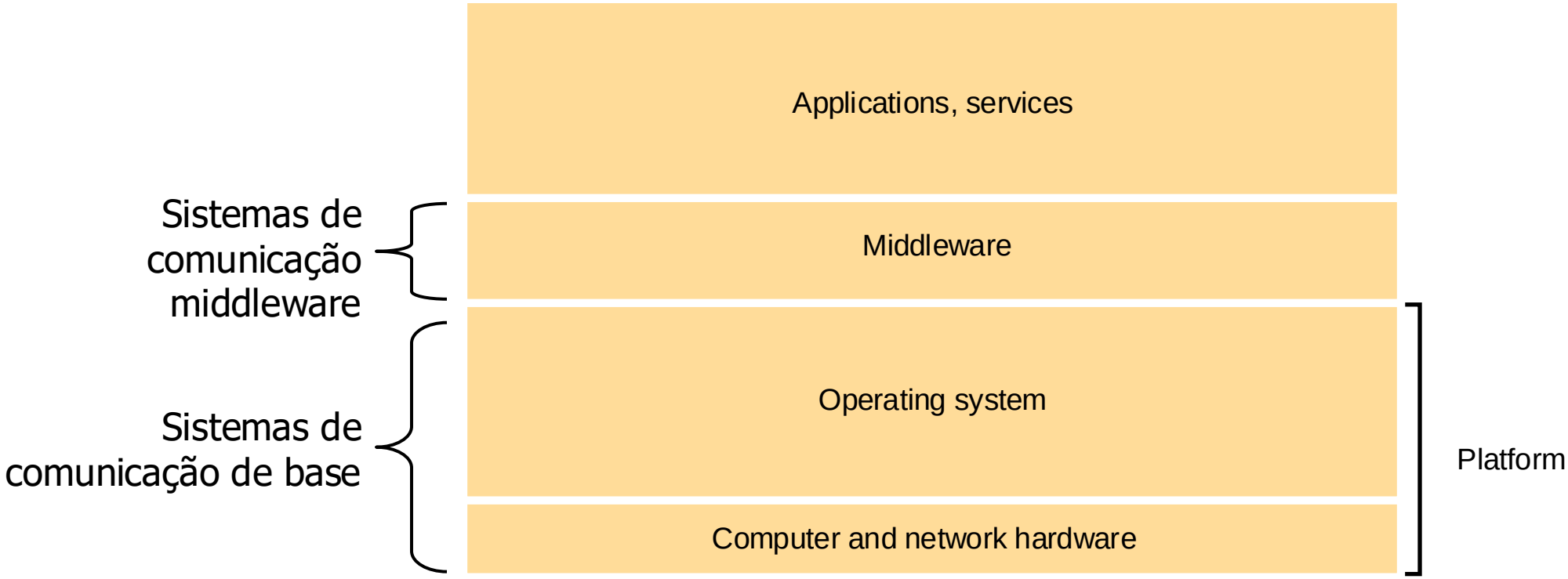
Comunicação direta

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 4th Edition, 2005

COMUNICAÇÃO NUM SISTEMA DISTRIBUÍDOS



SISTEMAS DE COMUNICAÇÃO DE BASE

Os sistemas de operação podem suportar a comunicação de dados entre os diferentes computadores envolvidos num sistema distribuído.

Protocolos mais populares:

- TPC/IP
- HTTP

TCP/IP: UDP

Comunicação por mensagens.

Mensagens podem-se perder, duplicar e chegar fora de ordem.

```
DatagramSocket socket = new DatagramSocket( 9000 ) ;

byte[] buffer = new byte[1500] ;
DatagramPacket packet = new DatagramPacket( buffer, buffer.length ) ;
socket.receive( packet ) ;
```

```
byte[] msg = ...
DatagramSocket socket = new DatagramSocket() ;

DatagramPacket packet = new DatagramPacket( msg, msg.length) ;
packet.setAddress( InetAddress.getByName( "servername" ) ) ;
packet.setPort( 9000 ) ;
socket.send( packet ) ;
```

TCP/IP: IP MULTICAST

Comunicação por mensagens com múltiplos recetores.

Cliente envia mensagem para endereço do grupo. Qualquer processo se pode juntar ao grupo para receber mensagens.

Mensagens podem-se perder, duplicar e chegar fora de ordem.

```
MulticastSocket socket = new MulticastSocket( 9000 ) ;
socket.joinGroup( InetAddress.getByName( "225.10.10.10" ));

byte[] buffer = new byte[1500] ;
DatagramPacket packet = new DatagramPacket( buffer, buffer.length ) ;
socket.receive( packet ) ;
```

```
byte[] msg = ...
MulticastSocket socket = new MulticastSocket() ;

DatagramPacket packet = new DatagramPacket( msg, msg.length) ;
packet.setAddress( InetAddress.getByName( "225.10.10.10" ) ) ;
packet.setPort( 9000 ) ;
socket.send( packet ) ;
```

TCP/IP: TCP

Dados transmitidos como fluxo contínuo.

Dados chegam de forma fiável a menos que o stream seja quebrado.

```
ServerSocket ss = new ServerSocket( 9000) ;
while( true ) {
        Socket cs = ss.accept() ;
    ....
}
```

```
byte[] msg = ...
Socket cs = new Socket("servername", 9000) ;
OutputStream os = cs.getOutputStream() ;
InputStream is = cs.getInputStream() ;

os.write( msg)
int b = is.read();
```

HTTP

Comunicação pedido/resposta sobre TCP, invocando URL.

Dados chegam de forma fiável a menos que o stream seja quebrado.

```
HttpURLConnection con = (HttpURLConnection)
                new URL("http://asc.di.fct.unl.pt/~jleitao").openConnection();
con.setRequestMethod(”GET");
con.setDoOutput(true);
con.setDoInput(true);
OutputStream os = con.getOutputStream();
....
os.flush();
InputStream is = con.getInputStream();
....
```

HTTP ASSÍNCRONO

Comunicação pedido/resposta, com resposta a ser recebida de forma assíncrona.

- Qual o interesse?

Solução adotada nos browser: JavaScript nativo ou bibliotecas JavaScript (e.g. Jquery)

```
[javascript]
var url = …
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
                result = xmlhttp.response;
                // process result
        }
};
xmlhttp.open("GET", url, true);
xmlhttp.responseType = "json";
xmlhttp.send();
```

readyState
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

WEB SOCKETS

Comunicação full-duplex sobre TCP entre clientes e servidores Web.

- Permite notificações dos servidores, streaming.

Suporte generalizado nos browsers.

```
[javascript]
var ws = new WebSocket("ws://asc.di.fct.unl.pt/websocket");

ws.onopen = function() {
        ws.send("Connecting... ");
};
ws.onmessage = function (evt) {
        var received_msg = evt.data;
        ...
};
ws.onclose = function() {
        ...
};
```

HTTP/3 e QUIC

A combinação HTTP/TLS/TCP tem alguns problemas:

- Criação de conexões lenta – handshake TCP + TLS;
- TCP *slow start;*
- Caso haja um erro na propagação dum pacote, esse erro tem impacto na conexão – se uma conexão estiver a ser usada para propagar vários streams (como e.g. quando se vão buscar múltiplas imagens duma página web), todos são afetados.

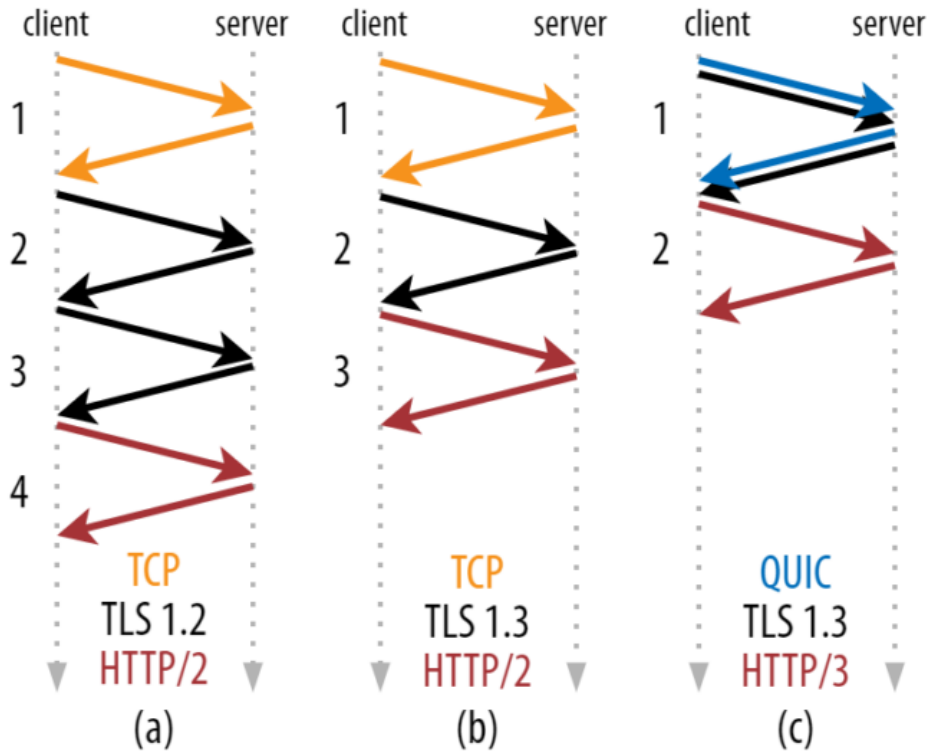


Imagem de: https://blog.apnic.net/2023/09/25/why-http-3-is-eating-the-world/

QUIC

QUIC é um novo protocolo de transporte, contruído usando UDP, com suporte para múltiplos fluxos (streams) dentro da mesma conexão; integração com TLS; rápido início de conexão.

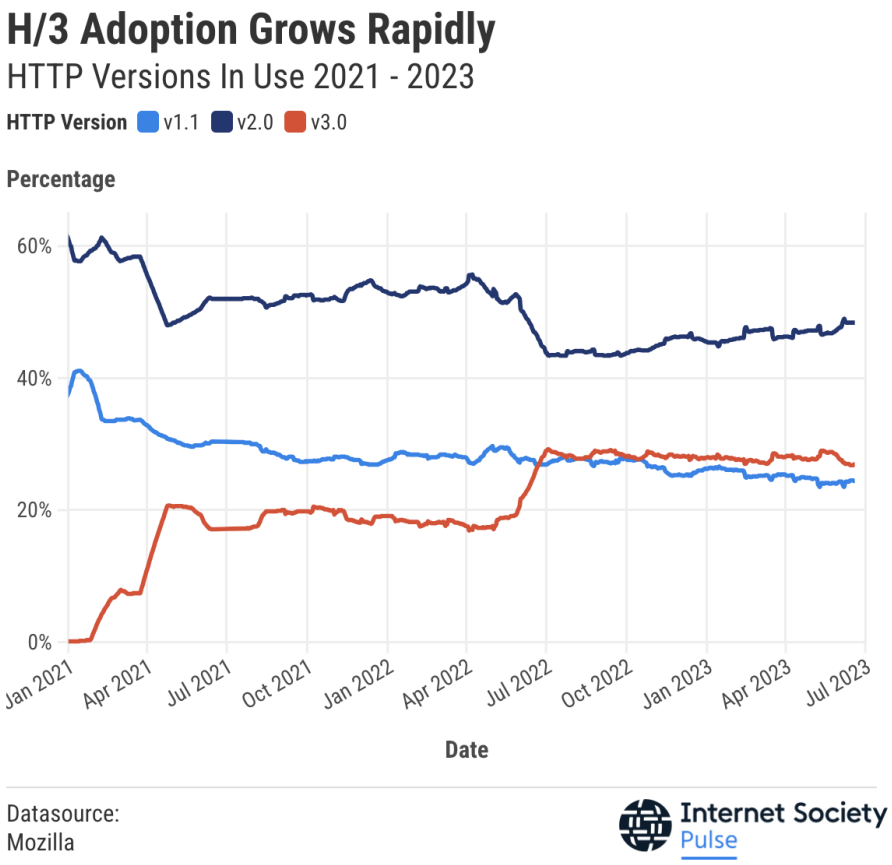


Imagem de: https://blog.apnic.net/2023/09/25/why-http-3-is-eating-the-world/

HTTP/3

HTTP/3 é a nova versão do HTTP, que integra diretamente com o QUIC.

Suportado pelos browsers mais usados.

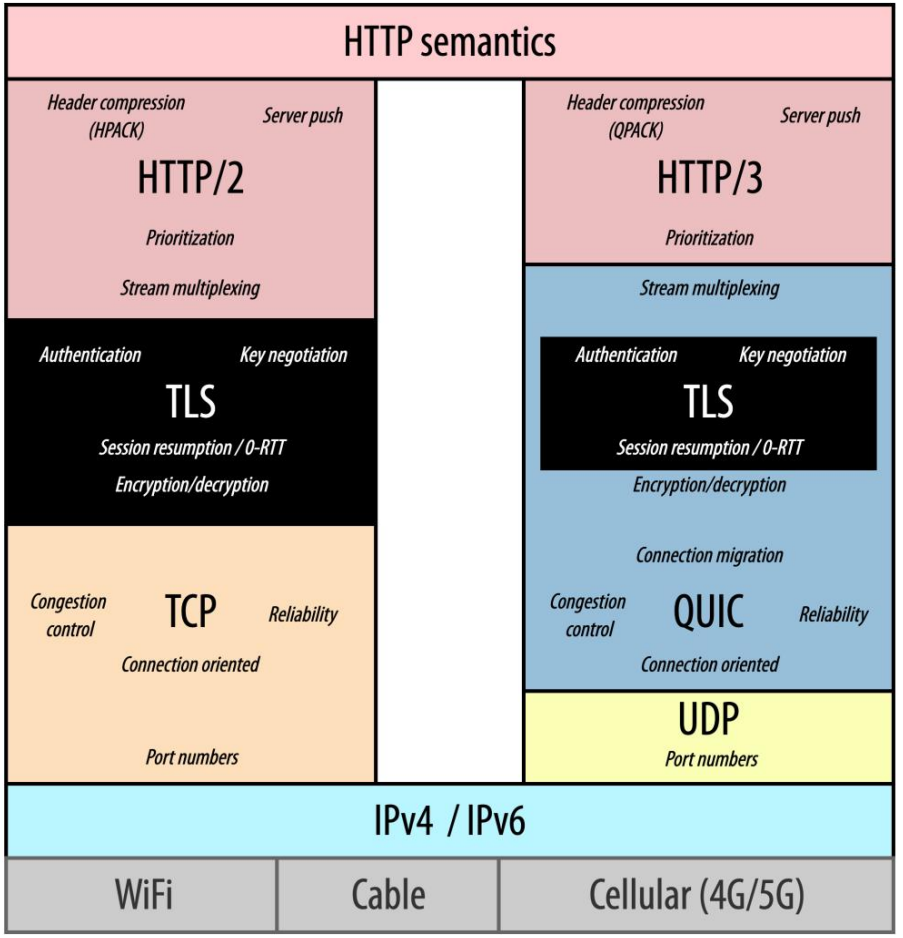


Imagem de: https://blog.apnic.net/2023/09/25/why-http-3-is-eating-the-world/

COMUNICAÇÃO NO NÍVEL MIDDLEWARE

Implementa sistema de comunicação recorrendo às primitivas de comunicação base

Fornece propriedades adicionais, atrasando a entrega das mensagens

- **Definição**: **Entrega** de uma mensagem num sistema de comunicação representa a ação do sistema disponibilizar a mensagem para ser lida (i.e., observada) pelas aplicações

- Porque é que pode ser útil atrasar a entrega de uma mensagem?
- Atrasar a entrega de uma mensagem pode, por exemplo, permitir que a **ordem de entrega** das mensagens seja diferente da **ordem de chegada**.

FACETAS DA COMUNICAÇÃO

Forma da interação
- Streams
- Mensagens
  - Ordenação das mensagens

Número de destinatários
- Ponto-a-ponto
- Multi-ponto (estudado mais tarde)
- Um-de-muitos (anycast)

Direção de interação
- Uni-direcional
- Bi-direcional

Tipo de sincronização
- Comunicação síncrona
- Comunicação assíncrona

Persistência
- Comunicação persistente
- Comunicação volátil

Fiabilidade (modelo de falhas)

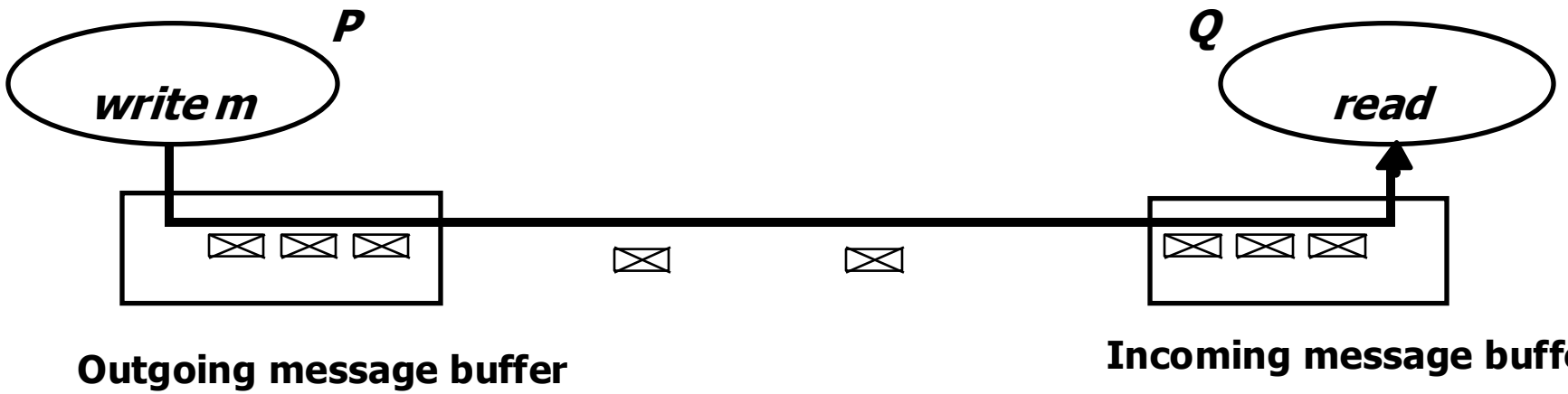
FORMA DE INTERAÇÃO: STREAMS

Emissor e recetor estabelecem um fluxo contínuo de dados

- Ordem dos dados enviados é mantida;
- Fronteira das escritas dos dados não é preservada.

Exemplos de situações em que é apropriado?

NOTA: poderíamos implementar sobre UDP... QUIC faz isso.



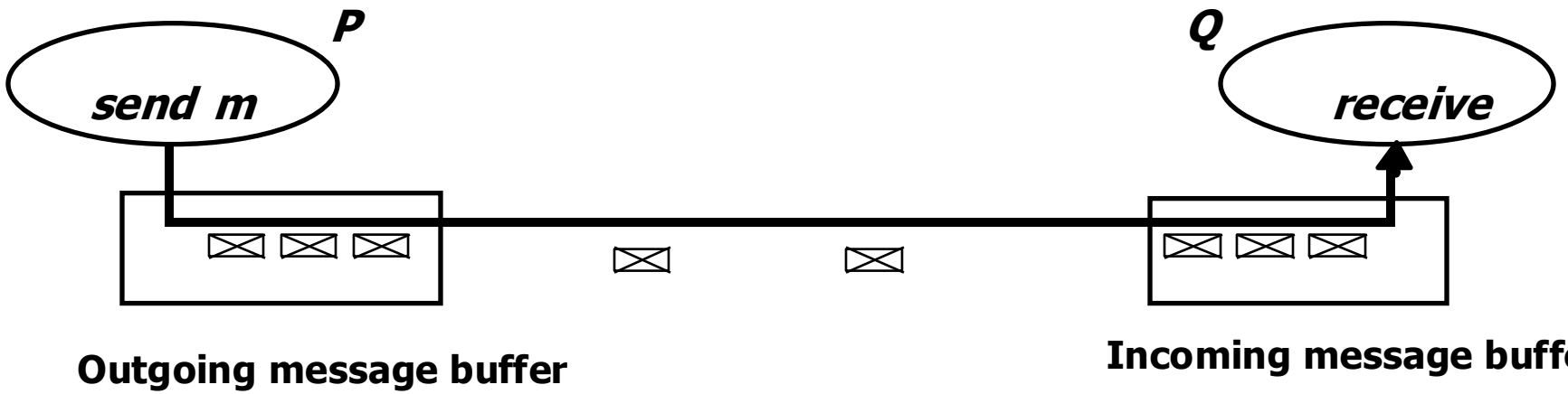
FORMA DE INTERAÇÃO: MENSAGENS

Emissor e recetor comunicam trocando mensagens

Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?



FORMA DE INTERAÇÃO: MENSAGENS

Emissor e recetor comunicam trocando mensagens

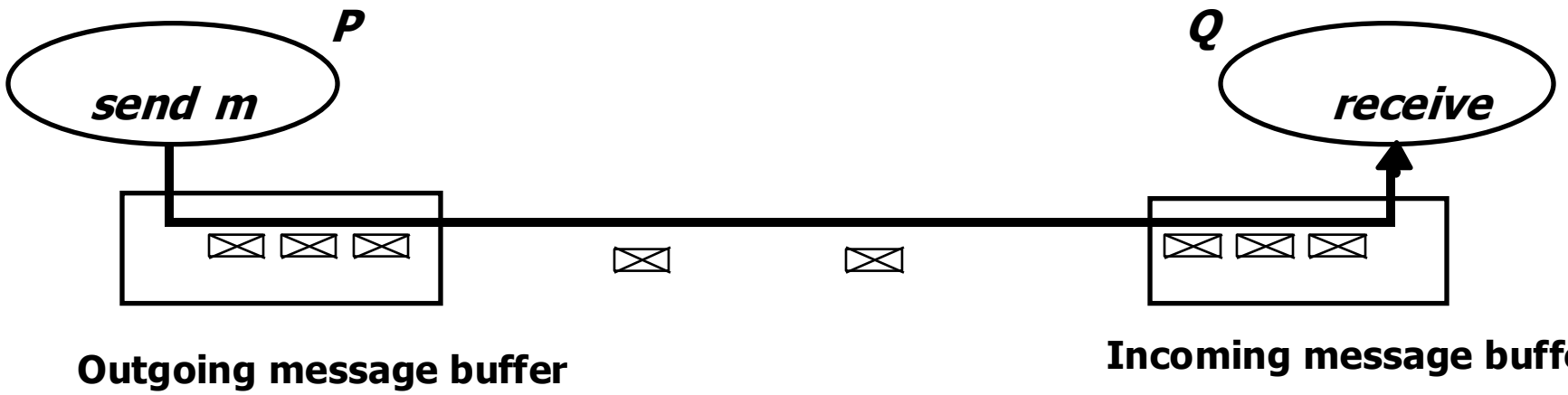
Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?

<dimensão, dados> ou <dados,delimitador> ou ...

vantagens? desvantagens de cada opção?



Forma de Interação: mensagens

Emissor e recetor comunicam trocando mensagens

Cada mensagem tem um limite (e dimensão) bem-definida.

Exemplos de situações em que é apropriado?

Como implementar sobre TCP?

<dimensão, dados> ou <dados,delimitador> ou ...

No primeiro caso é necessário conhecer a priori a dimensão da mensagem. Para mensagens grandes, geradas dinamicamente poderá obrigar a ter a mensagem em memória antes de a poder enviar. Para mensagens de pequena dimensão é uma boa solução.

No segundo caso, é preciso garantir que o delimitador não ocorre dentro da mensagem, sob pena de o emissor e o receptores ficarem dessincronizados.

Uma terceira abordagem, adequada para mensagens de grande dimensão e geradas dinamicamente, pode-ase partir a mensagem numa sequência de blocos de dimensão fixa (pequena), mantendo apenas o útimo bloco em memória.

FORMA DE INTERAÇÃO: ORDENAÇÃO DAS MENSAGENS

Sem garantias de ordem

- Sistema não garante que as mensagens são entregues pela ordem que foram enviadas

Entrega pela mesma ordem da emissão – FIFO (first in first out)

- Sistema garante que as mensagens dum emissor são entregues pela mesma ordem que foram enviadas. Como implementar em TCP/UDP?

- Haverá outras garantias de ordem?

NÚMERO DE DESTINATÁRIOS

Comunicação ponto-a-ponto

- Comunicação entre um emissor e um recetor

Comunicação multi-ponto

- Comunicação entre um emissor e um conjunto de recetores
- Broadcast: envio de 1 emissor para todos os recetores
- Multicast: envio de 1 emissor para todos os recetors de um grupo
- Anycast: envio de 1 emissor para um recetor de um grupo

DIRECÇÃO DE INTERAÇÃO

Comunicação uni-direccional:

- Comunicação apenas num sentido: emissor->recetor

Comunicação bi-direccional:

- Comunicação nos dois sentidos

SINCRONIZAÇÃO

Comunicação assíncrona:

- o emissor só fica bloqueado até o seu pedido de envio ser tomado em consideração
- o recetor fica bloqueado até ser possível receber dados
  - Em geral, o sistema de comunicação do receptor armazena (algumas) mensagens caso não exista nenhum recetor bloqueado no momento da sua recepção. Assim, funciona como um *buffer* entre o emissor e o recetor
  - É possível variante em que o recetor não fica bloqueado e devolve erro ou a receção é efectuada em background

Comunicação síncrona:

- o emissor fica bloqueado até:
  - o recetor "receber" os dados – comunicação síncrona unidireccional
  - receber a resposta do receptor – comunicação pedido / resposta ou cliente / servidor
- o receptor fica bloqueado até ser possível consumir dados

PERSISTÊNCIA

Comunicação volátil: mensagens apenas são encaminhadas se o recetor existir e estiver a executar, caso contrário são destruídas.

- Exemplo: ???

Comunicação persistente: mensagens são guardadas pelo sistema de comunicação até serem consumidas pelos destinatários, que podem não estar a executar. Mensagens são guardadas num receptáculo independente do recetor – mailbox, canal, porta persistente, etc.

- Exemplo: ???

FIABILIDADE

Comunicação fiável: o sistema garante a entrega das mensagens em caso de falha temporária. Como implementar?

Comunicação não-fiável: em caso de falha, as mensagens podem-se perder

PARA SABER MAIS

George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair,
Distributed Systems – Concepts and Design,
Addison-Wesley, 5th Edition, 2011

- Capítulo 4.1-4.3.

SISTEMAS DISTRIBUÍDOS

Aula 4: Capítulo 4

Invocação remota

NOTA PRÉVIA

A estrutura da apresentação é semelhante e utiliza algumas das figuras do livro de base do curso

G. Coulouris, J. Dollimore and T. Kindberg,
Distributed Systems - Concepts and Design,
Addison-Wesley, 5th Edition, 2009

Para saber mais:

- RMI/RPCs - capítulo 5.
- Representação de dados e protocolos - capítulo 4.3.
- Web services – capítulo 9

NA ÚLTIMA AULA

Mecanismos de comunicação base

- TCP, UDP, IP multicast
- HTTP assíncrono, Web sockets

Propriedade dos sistemas de comunicação

- **Forma da interação**: streams, mensagens
- **Número de destinatários**: ponto-a-ponto, multi-ponto, um-de-muitos
- **Direção de interação**: uni-directional, bi-direcional
- **Tipo de sincronização**: comunicação síncrona, comunicação assíncrona
- **Persistência**: comunicação persistente, comunicação volátil
- **Fiabilidade**: fiável, não fiável

MOTIVAÇÃO

Estruturar uma aplicação distribuída com base nas mensagens trocadas pelos seus componentes é a abordagem mais óbvia, mas:

- exige atenção a muitos detalhes de baixo nível; a estrutura dos programas espelha os padrões de comunicação, em vez da lógica da aplicação no seu todo.
- em particular, os servidores ficam estruturados em função das mensagens que sabem tratar.

PROBLEMAS ?

De aplicação para aplicação, verifica-se que muitas linhas de código são *repetitivas*, não contêm nenhum significado aplicacional específico e referem-se apenas à gestão e processamento das comunicações.

Em particular, boa parte do código está dedicado a:

- criação de *communication end points* e sua associação aos processos criação, preenchimento e interpretação das mensagens;
- seleção do código a executar consoante o tipo da mensagem recebida gestão de temporizadores/tratamento das falhas

OBJETIVO

- Não será possível automatizar aquilo que é repetitivo?
- Não será possível que o programador apenas especifique o código aplicacional?

INVOCAÇÃO REMOTA

Nas linguagem imperativas definem-se funções / procedimentos / métodos para executar uma dada operação. Num ambiente distribuído, uma extensão natural consiste em permitir que a **execução ocorra noutra máquina**.

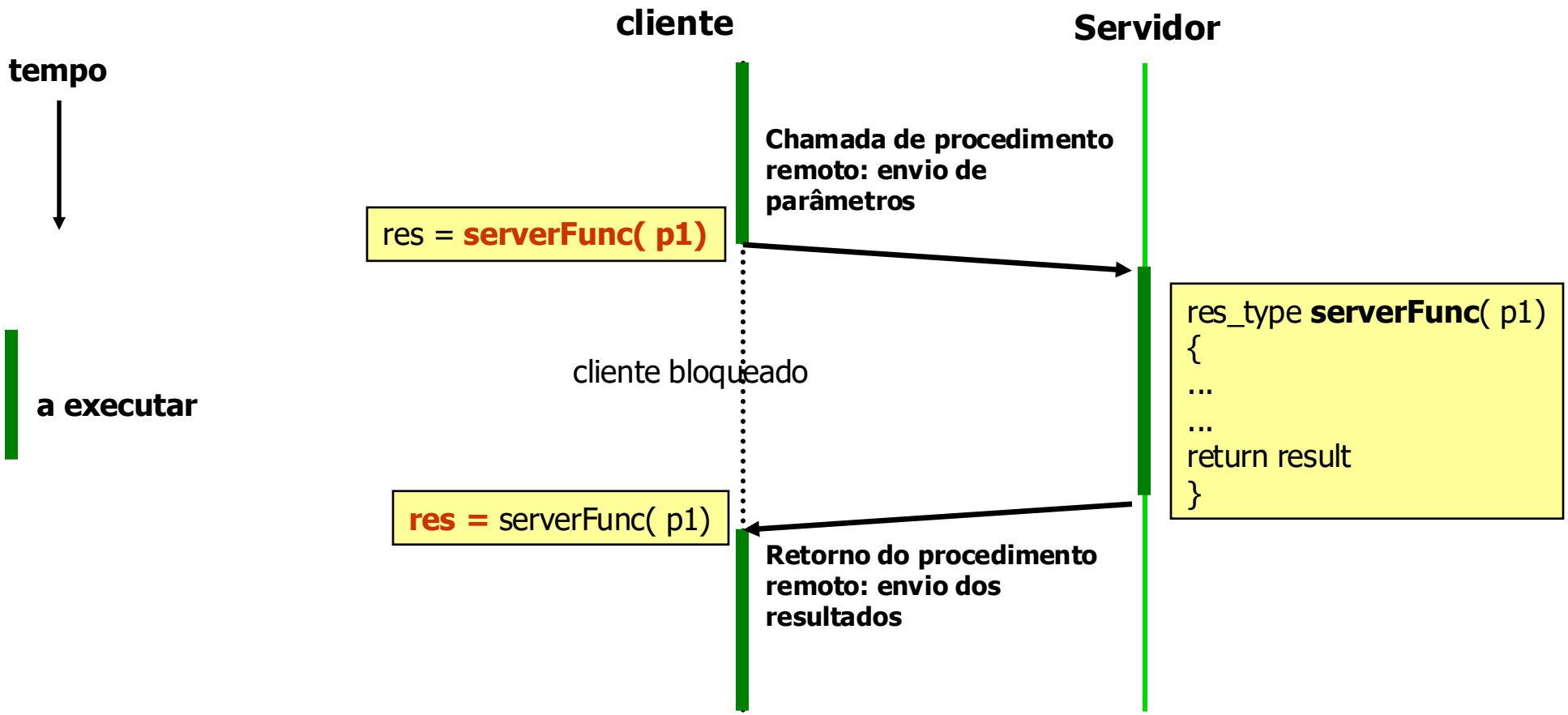
Invocação Remota de Procedimentos (RPCs), quando são executados funções/procedimentos remotamente.

- gRPC (https://grpc.io/), ONC/RPC, DCE

Invocação Remota de Métodos (RMI), quando são executados métodos de objetos remotos.

- JAVA RMI, .NET Remoting, Corba.
- Web Services (REST e SOAP)

INVOCAÇÃO DE PROCEDIMENTOS REMOTOS (RPCS)



Modelo

Servidor exporta interface com operações que sabe executar

Cliente invoca operações que são executadas remotamente e (normalmente) aguarda pelo resultado

INVOCAÇÃO REMOTA - PROPRIEDADES

Extensão natural do paradigma imperativo/procedimental a um ambiente distribuído

- Modelo síncrono de comunicação suporta chamadas bloqueantes no cliente

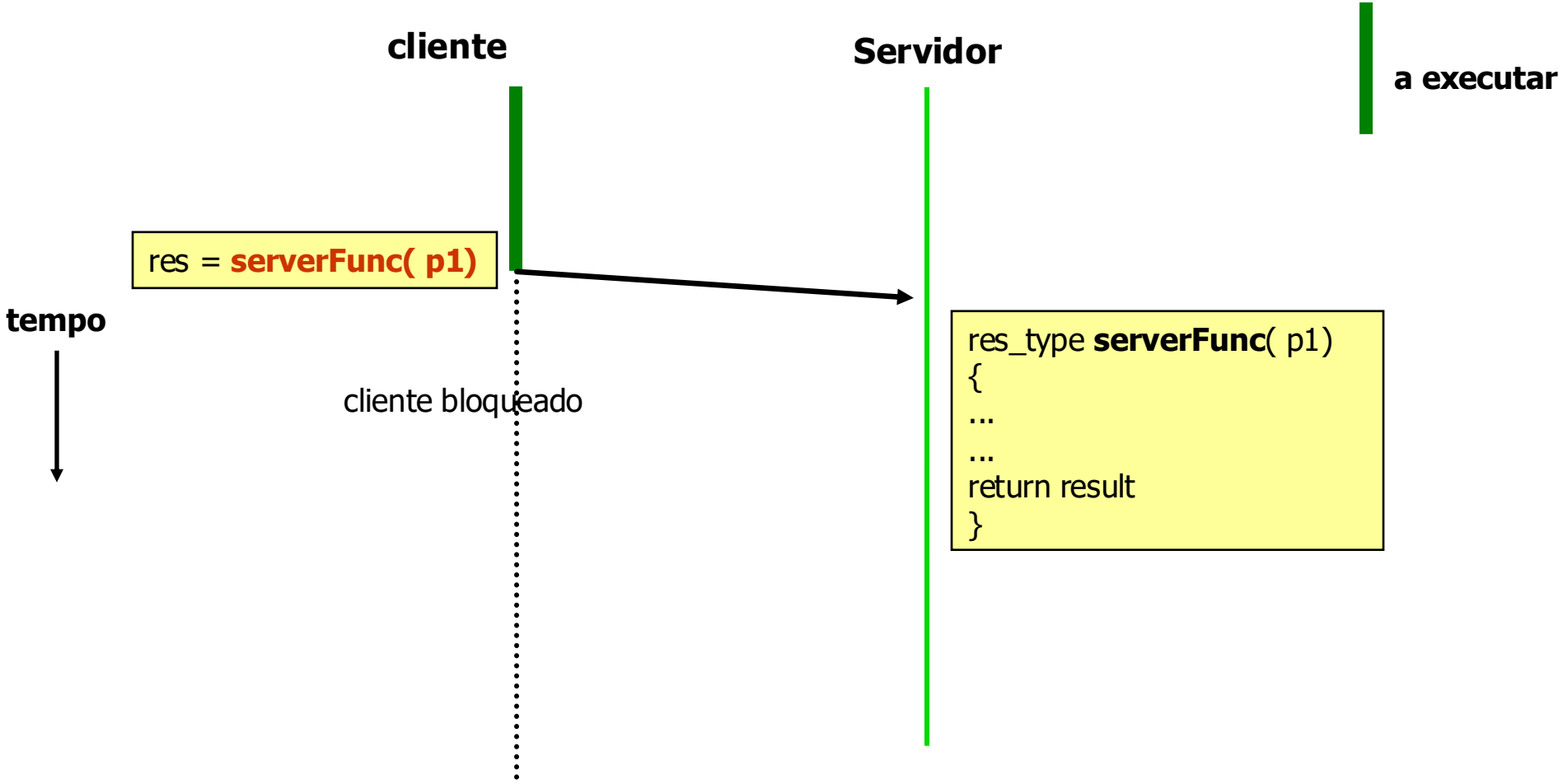
Esconde detalhes de comunicação (e tarefas repetitivas)

- Construção, envio, receção e tratamento das mensagens
- Tratamento básico de erros (devem ser tratados ao nível da aplicação)
- Heterogeneidade da representação dos dados

Simplifica disponibilização de serviços

- Interface bem definida, facilmente documentável e independente dos protocolos de transporte
- Sistema de registo e procura de serviços

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

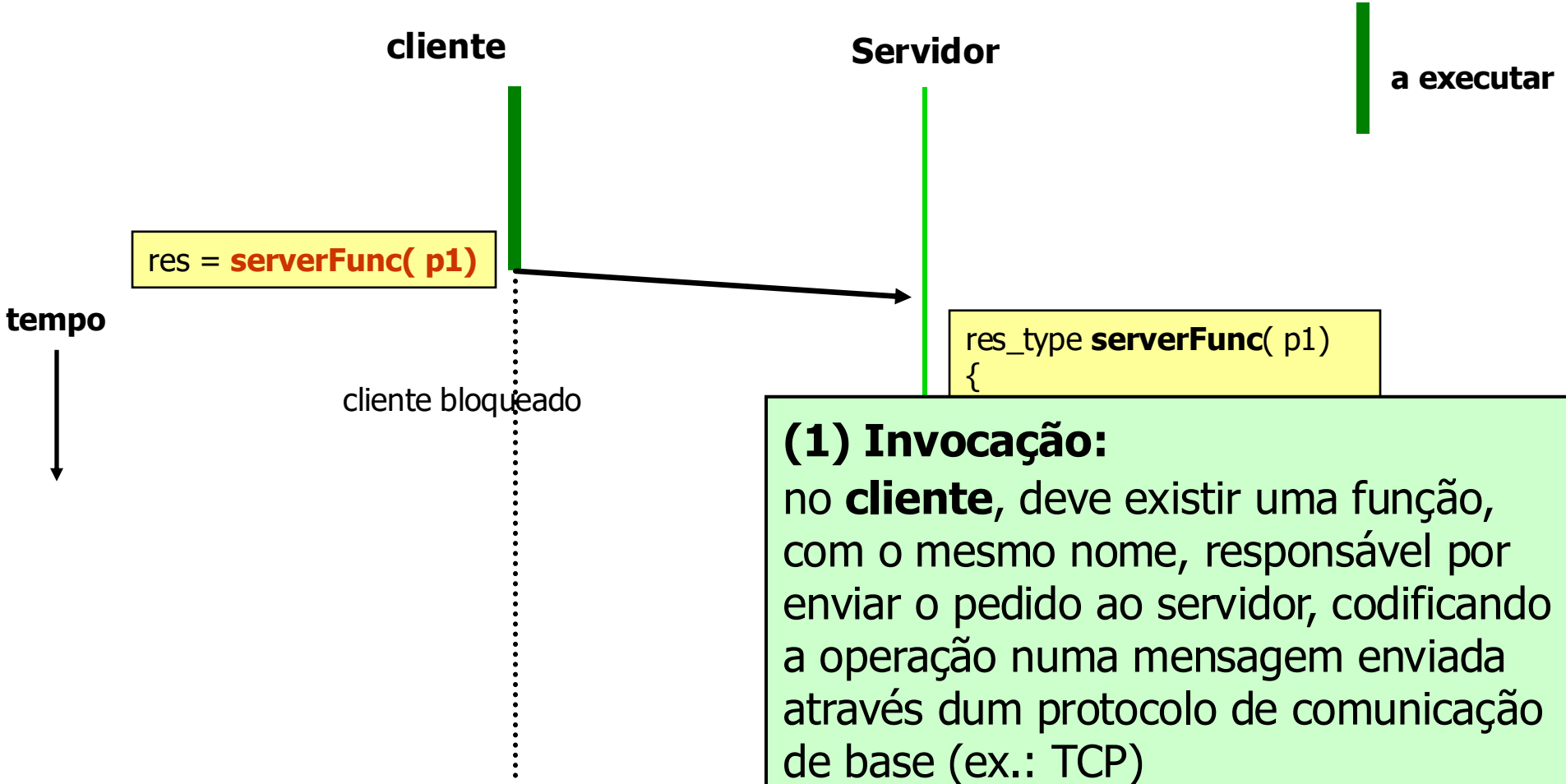
Cliente

```
res = serverFunc( p1)
```

Servidor

```
res_type serverFunc( T1 p1) {
        ...
        return result
}
```

RPCs: COMO IMPLEMENTAR



(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LING

(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

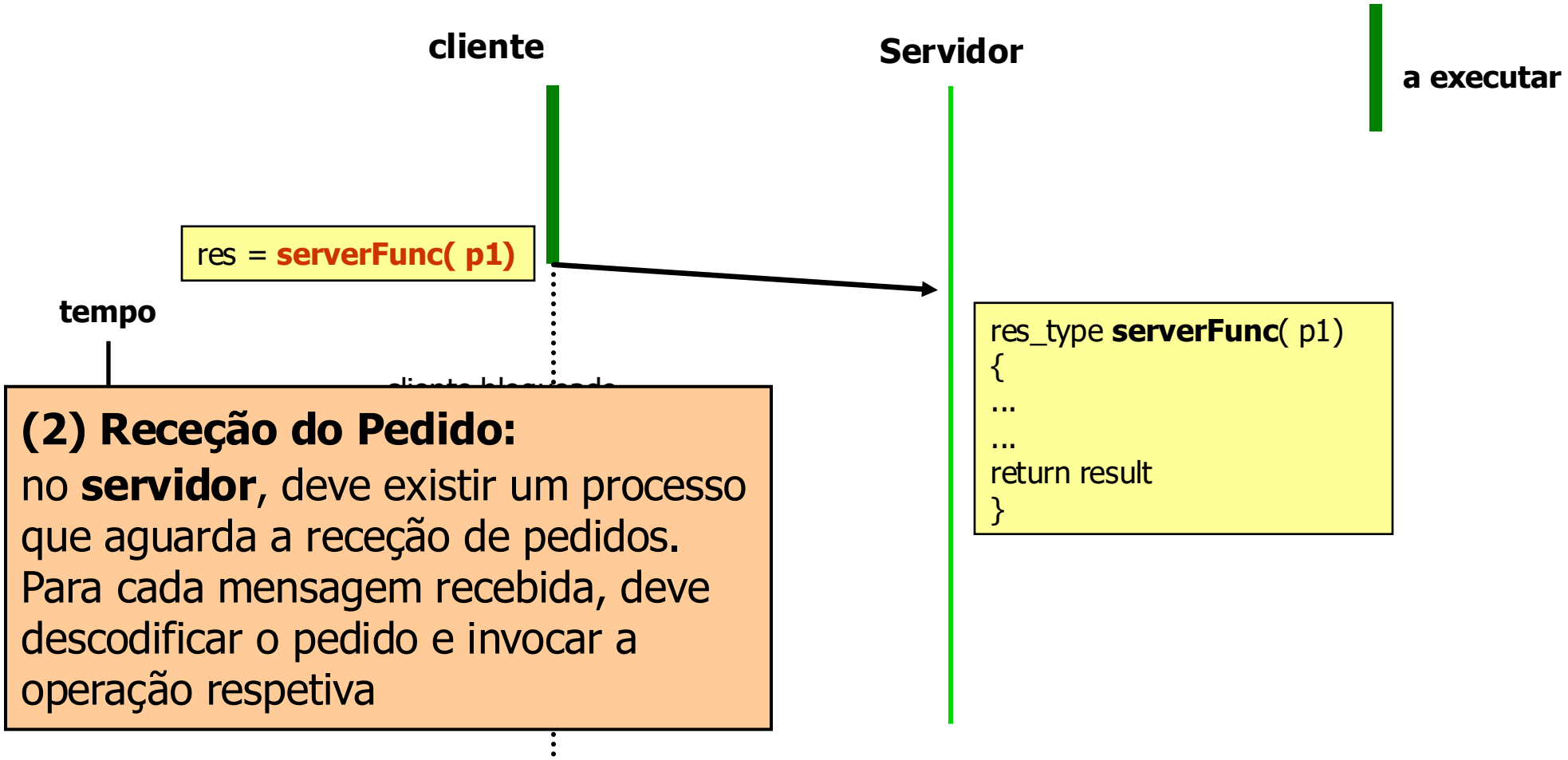
Cliente

```
res = serverFunc( p1)
```

```
}
```

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg[“serverFunc”,[p1]])
```

RPCs: COMO IMPLEMENTAR



(2) Receção do Pedido:
no **servidor**, deve existir um processo que aguarda a receção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respetiva

COMO ESCONDER OS DETALHES?

...IGUAGEM)

(2) Receção do Pedido:
no **servidor**, deve existir um processo que aguarda a receção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respetiva

Servidor

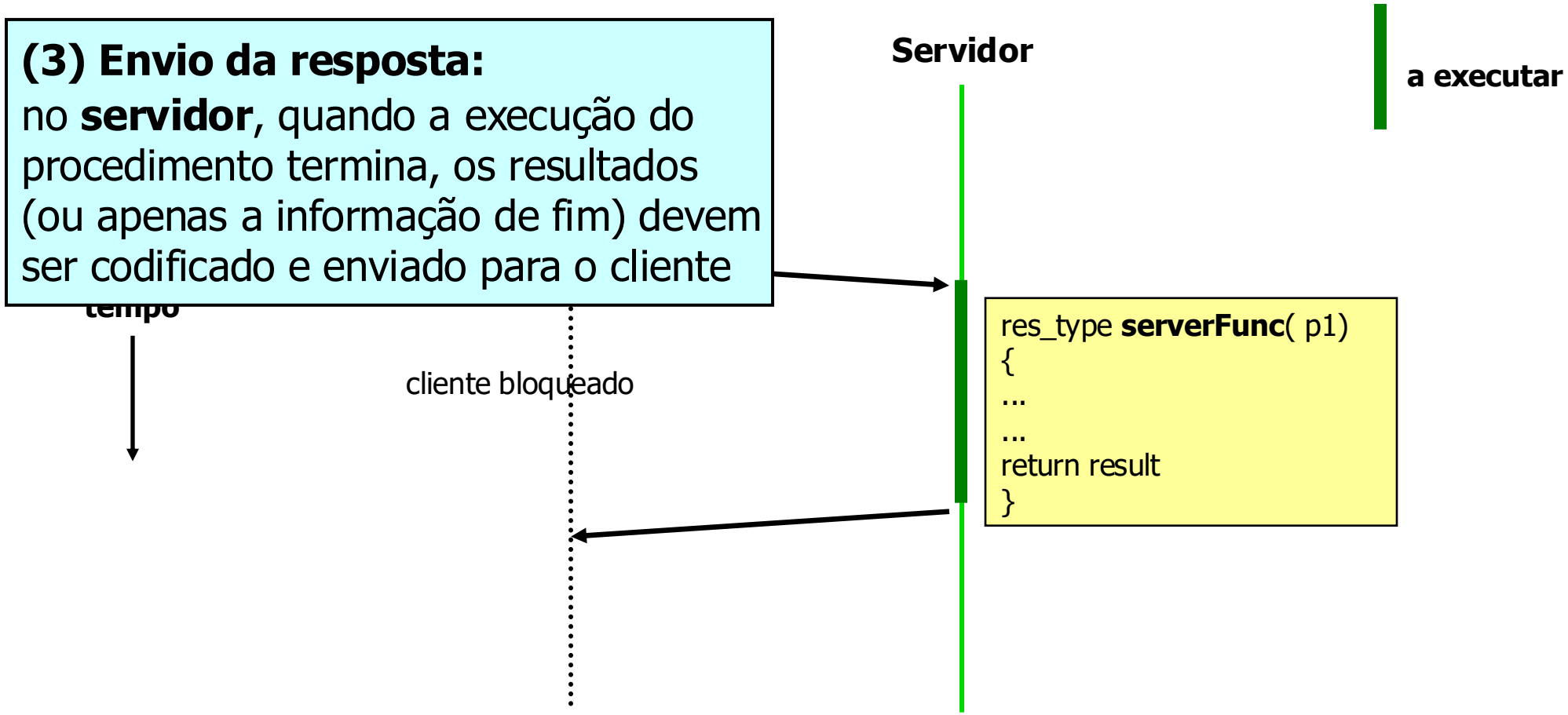
```
res_type serverFunc( T1 p1) {
        ...
        return result
}
```

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg["serverFunc",[p1]])
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg[op, params])
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
```

RPCs: COMO IMPLEMENTAR

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente



COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

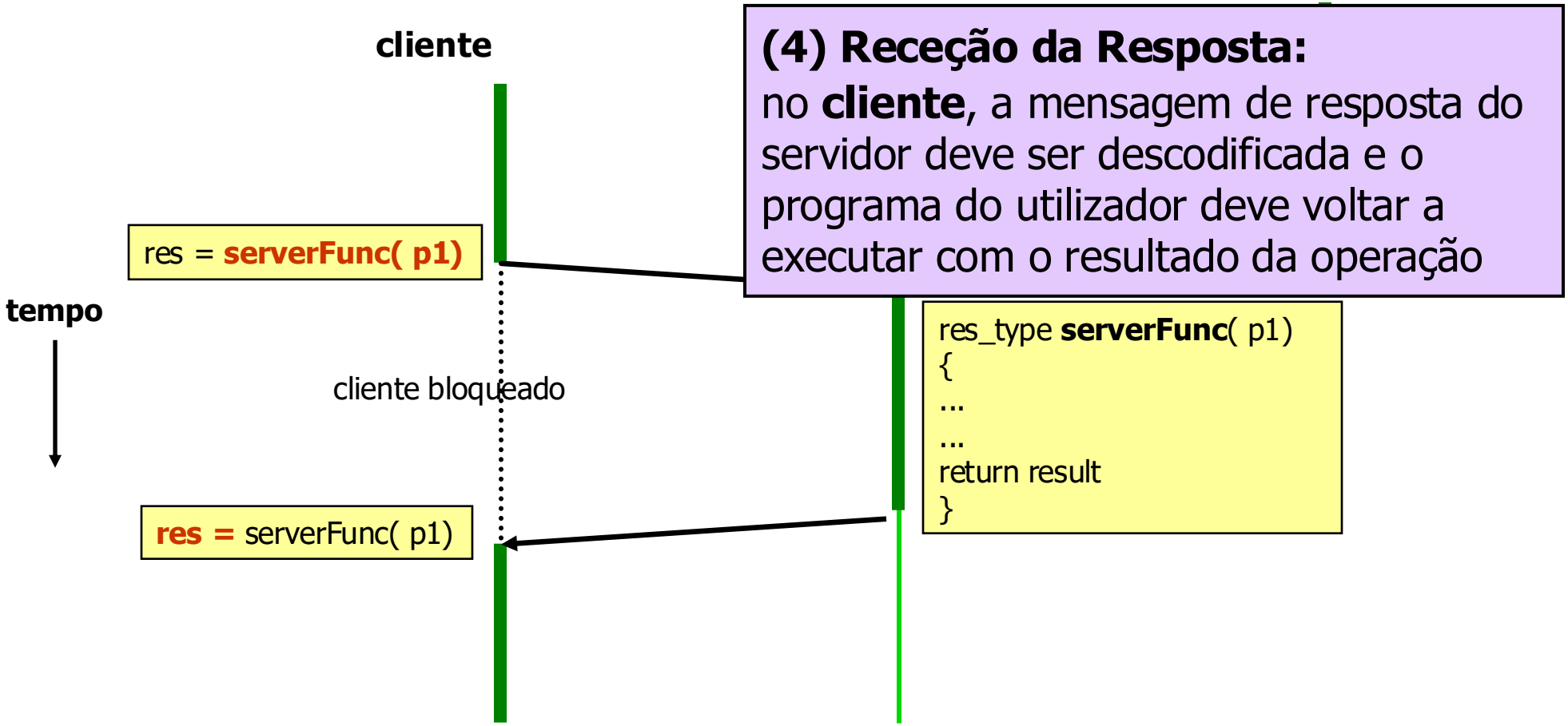
Servidor

```
res_type serverFunc( T1 p1) {
        ...
        return result
}
```

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg[ "serverFunc",[p1]])
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg[op, params])
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
    c.send( msg[res])
    c.close
```

RPCs: COMO IMPLEMENTAR



COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg( "serverFunc",[p1]))
    s.receive( msg( result))
    s.close
    return result
```

(4) Receção da Resposta:
no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

```
}
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
    if( op = "serverFunc")
        res = serverFunc( params[0]);
    else if( op = ...)
        ...
    c.send( msg(res))
    c.close
```

COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LI

Cliente

```
res = serverFunc( p1)
```

```
res_type serverFunc( T1 p1)
    s = new Socket( host, port)
    s.send( msg( "serverFunc",[p1]))
    s.receive( msg( result))
    s.close
    return result
```

Na prática, sucessivas invocações podem partilhar o mesmo socket…

Stub do cliente ou *proxy* do servidor

(4) Receção do Pedido:
no **cliente**, a mensagem de resposta do servidor deve ser descodificada e o programa do utilizador deve voltar a executar com o resultado da operação

(1) Invocação:
no **cliente**, deve existir uma função, com o mesmo nome, responsável por enviar o pedido ao servidor, codificando a operação numa mensagem enviada através dum protocolo de comunicação de base (ex.: TCP)

COMO ESCONDER OS DETALHES?
(SEM SUPORTE ESPECÍFICO DO RUNTIME DA LINGUAGEM)

Stub ou skeleton do servidor

(3) Envio da resposta:
no **servidor**, quando a execução do procedimento termina, os resultados (ou apenas a informação de fim) devem ser codificado e enviado para o cliente

(2) Recepção do pedido:
no **servidor**, deve existir um processo que aguarda a recepção de pedidos. Para cada mensagem recebida, deve descodificar o pedido e invocar a operação respectiva

Servidor

```
res_type serverFunc( T1 p1) {
            ...
            return result
}
```

```
s = new ServerSocket
forever
    Socket c = s.accept();
    c.receive( msg( op, params))
     if( op = "serverFunc")
         res = serverFunc( params[0]);
     else if( op = ...)
         ...
    c.send( msg(res))
    c.close
```

RPCs – Automatização (Proxy/Stub Compilers)

Nos sistemas de RPC/RMI, o código de comunicação é transparente para a aplicação.

É costume designar-se de **stub do cliente** às funções do cliente que efetuam a comunicação com o servidor para executar o método no servidor;

Do lado do servidor, o **stub ou *skeleton* do servidor** corresponde ao código de comunicação para esperar as invocações e executá-las, devolvendo o resultado;

RPCs – Automatização (Proxy/Stub Compilers)

Em alguns sistemas e ambientes usam-se ferramentas (compiladores) para gerar os stubs; e.g., wsimport para WebServices SOAP; gRPC.

Noutros sistemas a geração é automática: no servidor, quando este é instanciado; no cliente quando este se liga ao servidor da primeira vez:

- e.g., Java RMI, Servidor JAX-RS(Jersey);

Há ainda casos onde a invocação remota faz parte da própria especificação do ambiente/linguagem e é parte integrante do runtime:

- e.g., .NET Remoting.