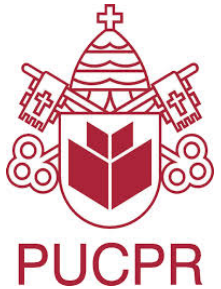



# Programação com Sockets

André Murbach Maidl

Escola Politécnica

Pontifícia Universidade Católica do Paraná

# Objetivos da aula

- Discutir a comunicação entre sistemas distribuídos por meio de troca de mensagens.
- Apresentar alguns modelos de comunicação.
- Apresentar a programação com *sockets* como um mecanismo para a implementação de troca de mensagens.
- Apresentar exemplos de programação com *sockets* em Go.

# Processos distribuídos

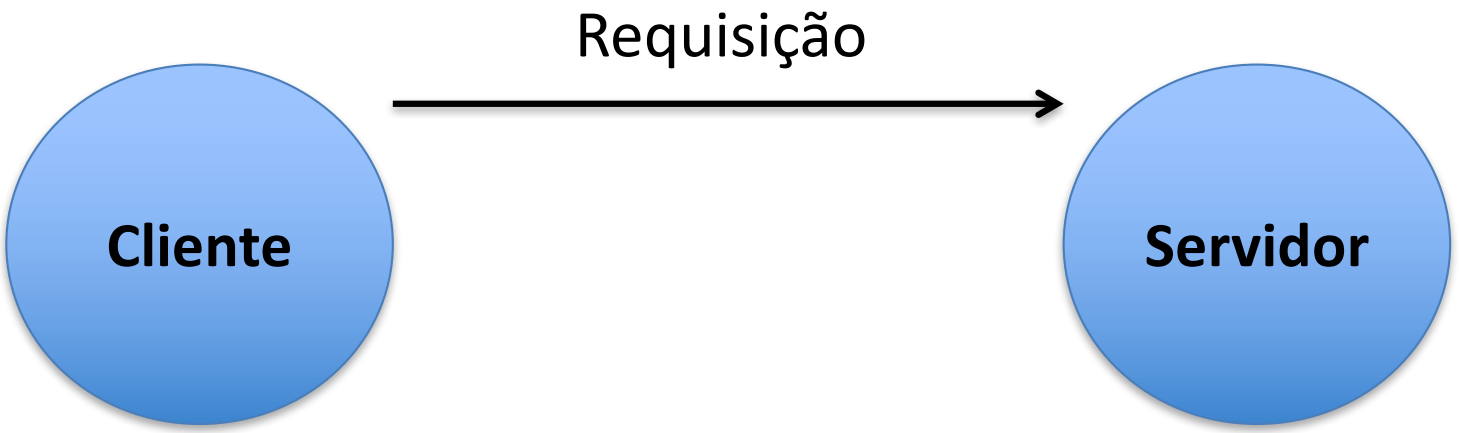
- São processos que rodam em diferentes máquinas e estão separados por uma rede.
- Eles podem ser *multithreaded* ou *singlethreaded*.
- Neste curso geralmente vamos tratar uma *thread* como um processo independente.

# Comunicação em SD

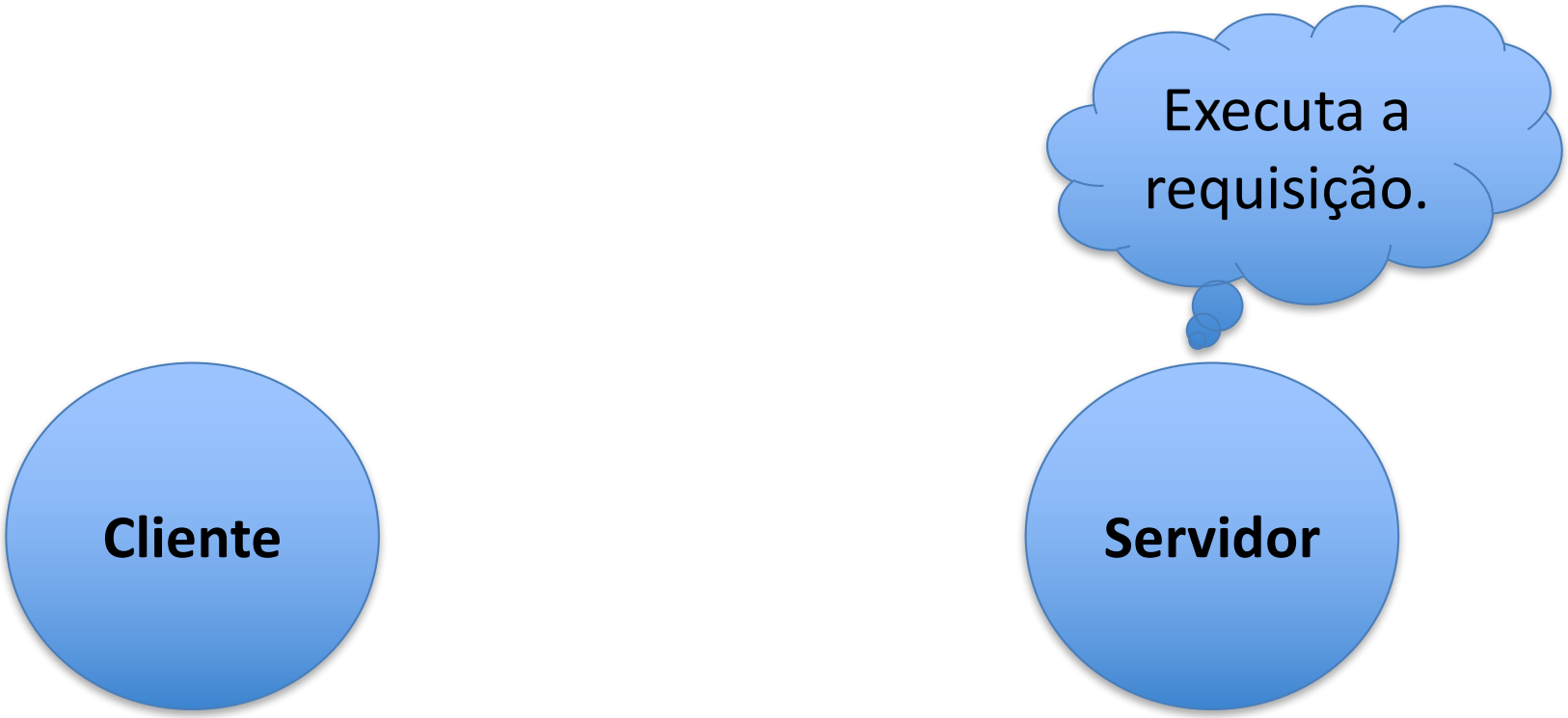
- Devido a ausência de memória compartilhada, toda a comunicação em SD é baseada na troca de mensagens.
- Existem diversos modelos para se implementar a comunicação entre processos em um sistema distribuído.

# O modelo de comunicação comum

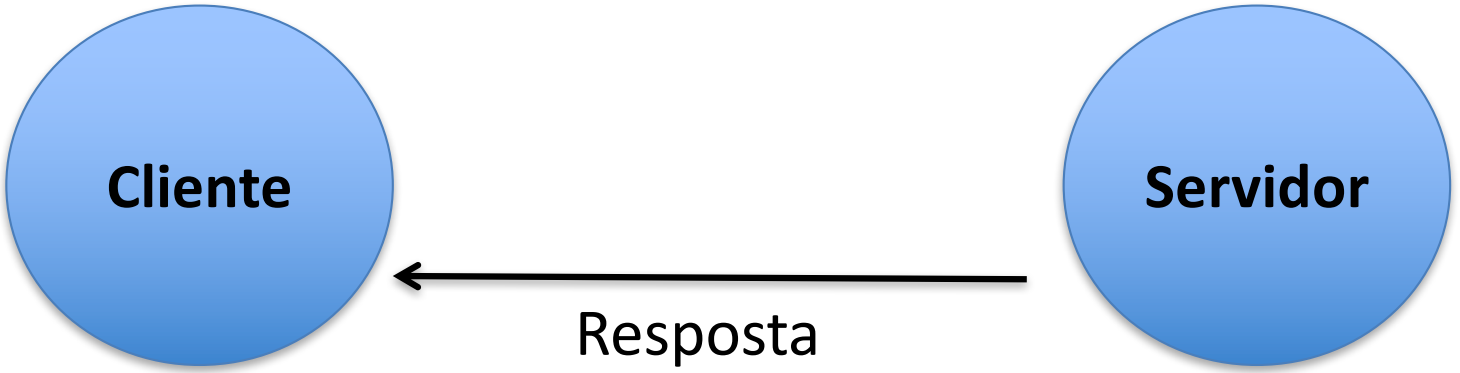
# O modelo de comunicação comum



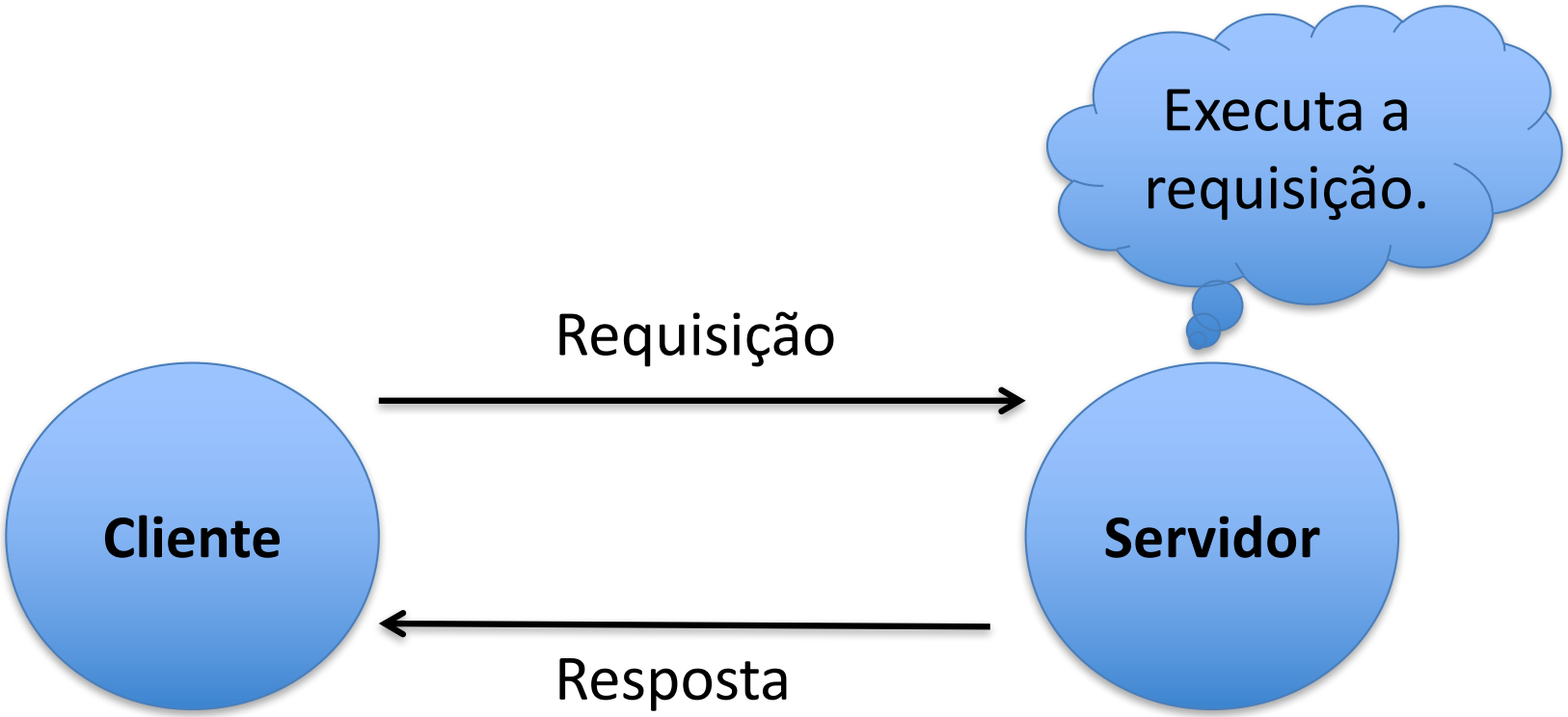
# O modelo de comunicação comum



# O modelo de comunicação comum



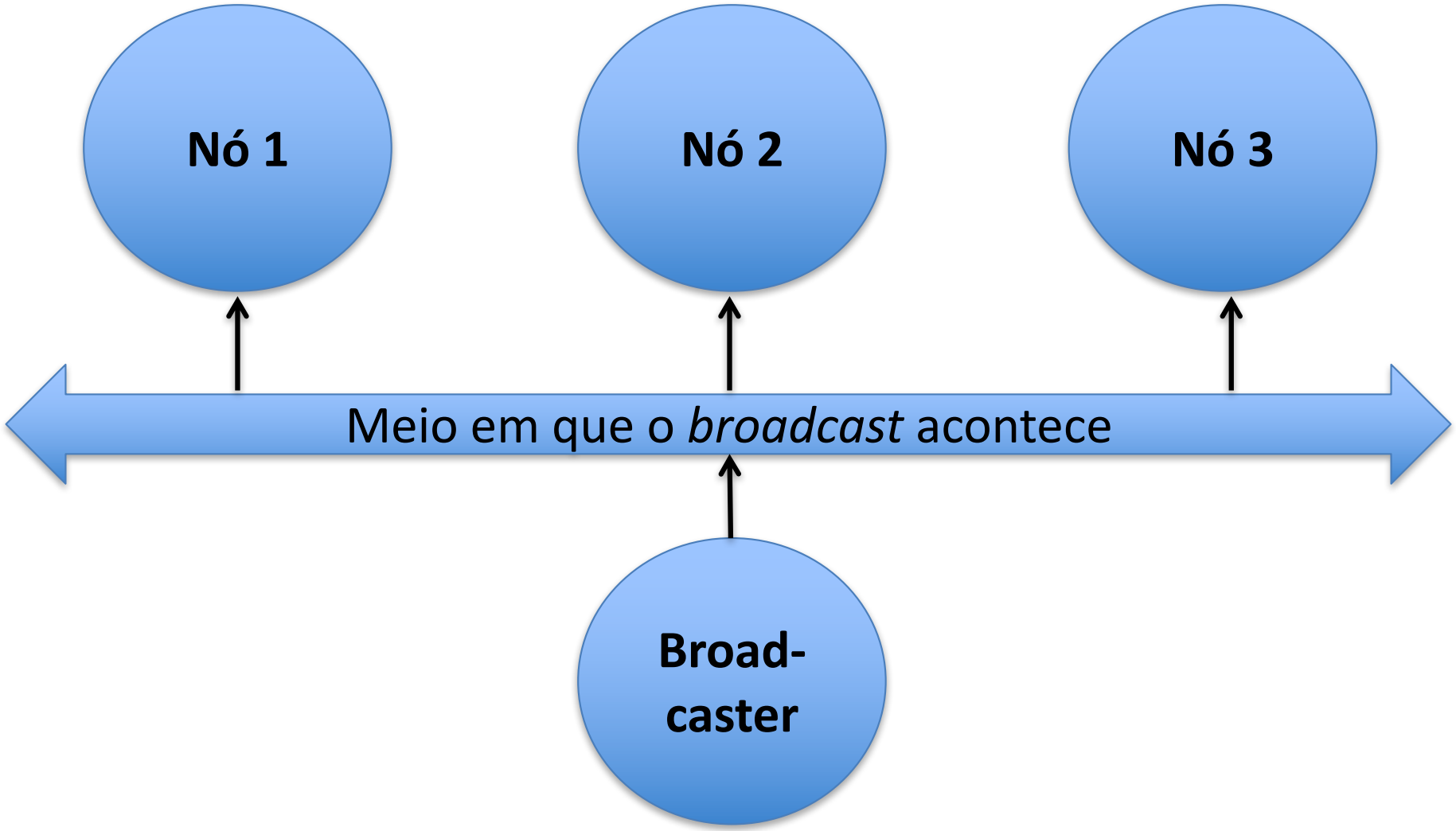
# Cliente-Servidor



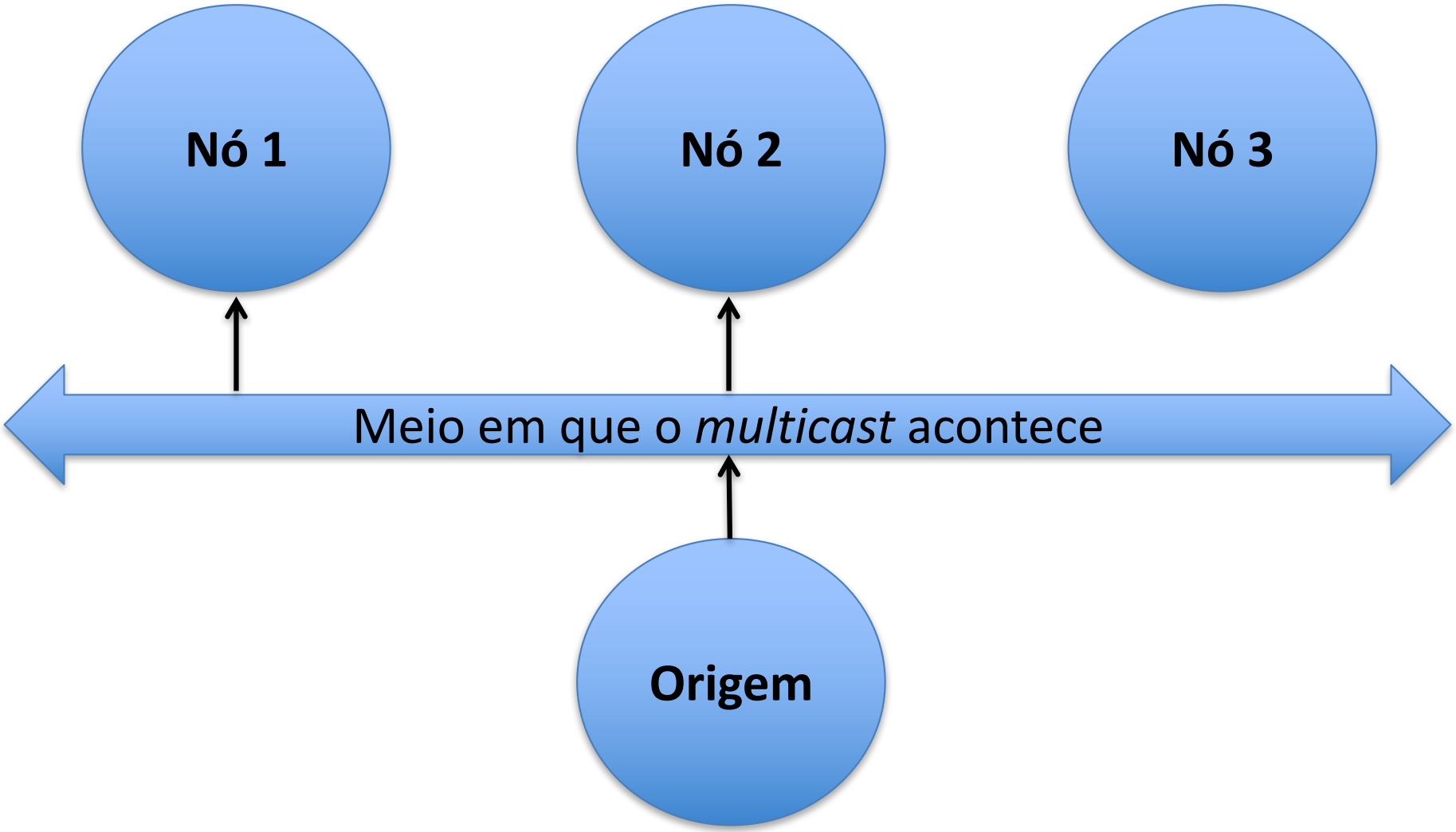
# Outros modelos de comunicação

- Além do modelo cliente-servidor, temos outros modelos de comunicação em SD:
  - Broadcast;
  - Multicast;
  - Publish/Subscribe;
  - Etc.

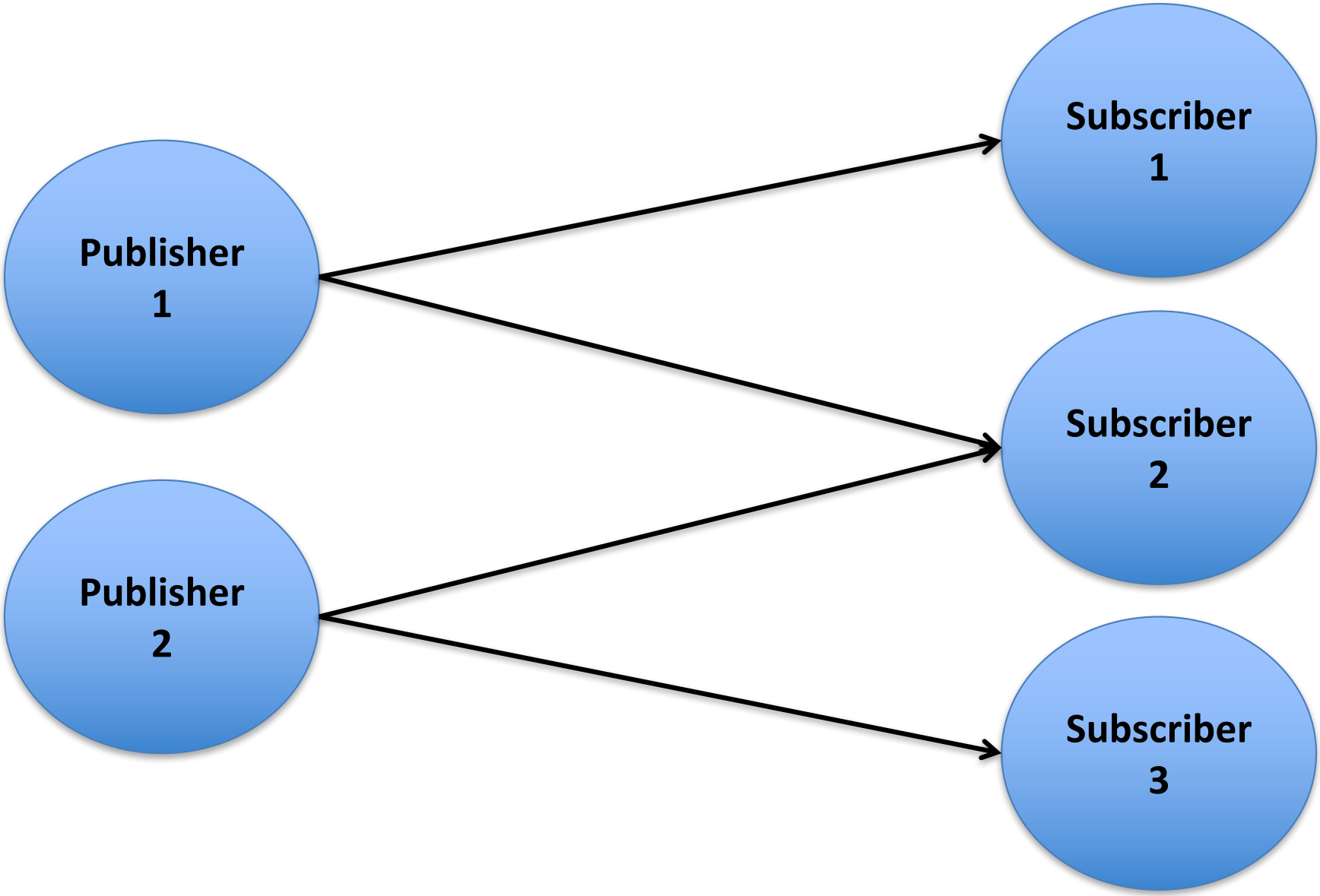
# *Broadcast*



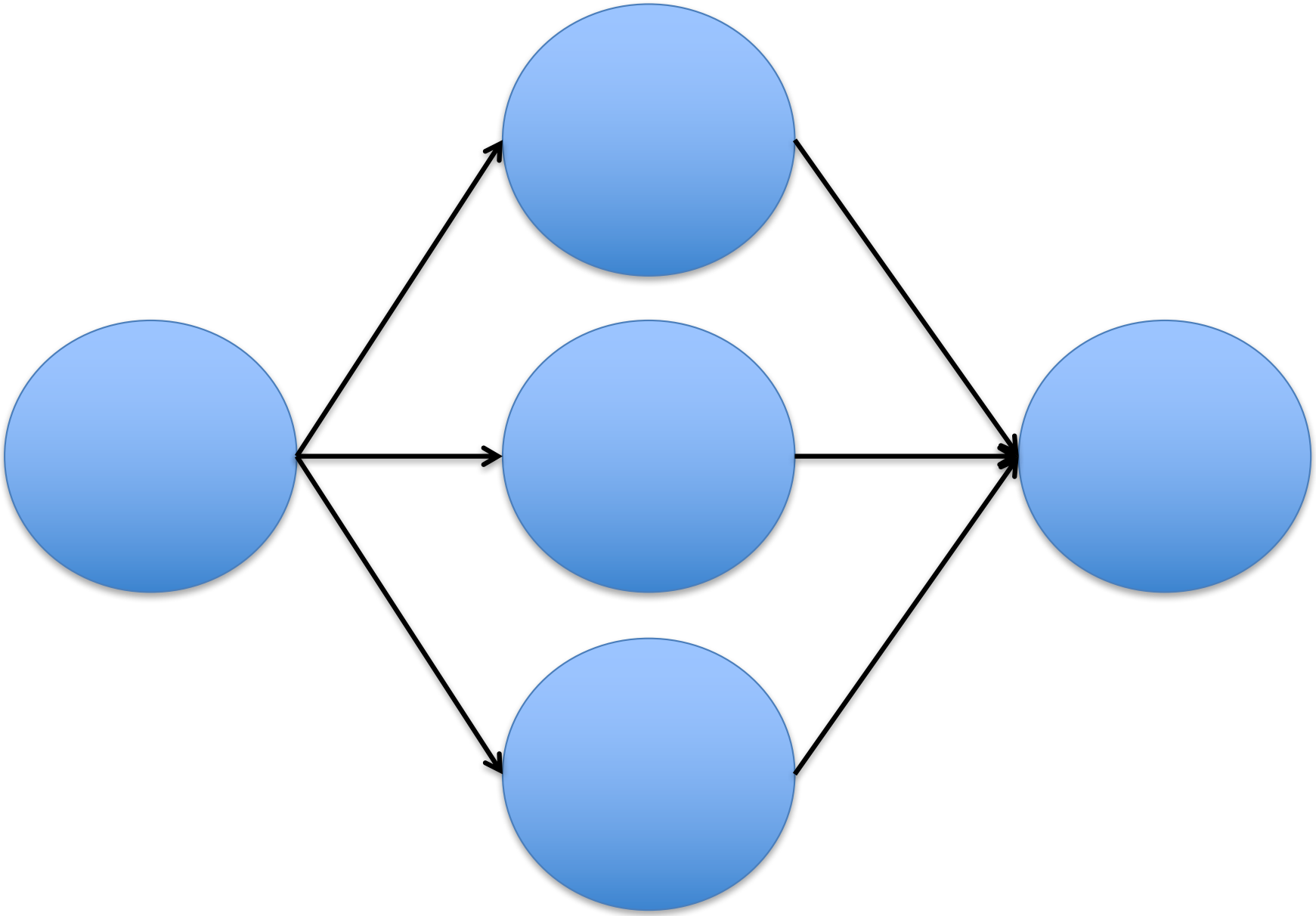
# *Multicast*



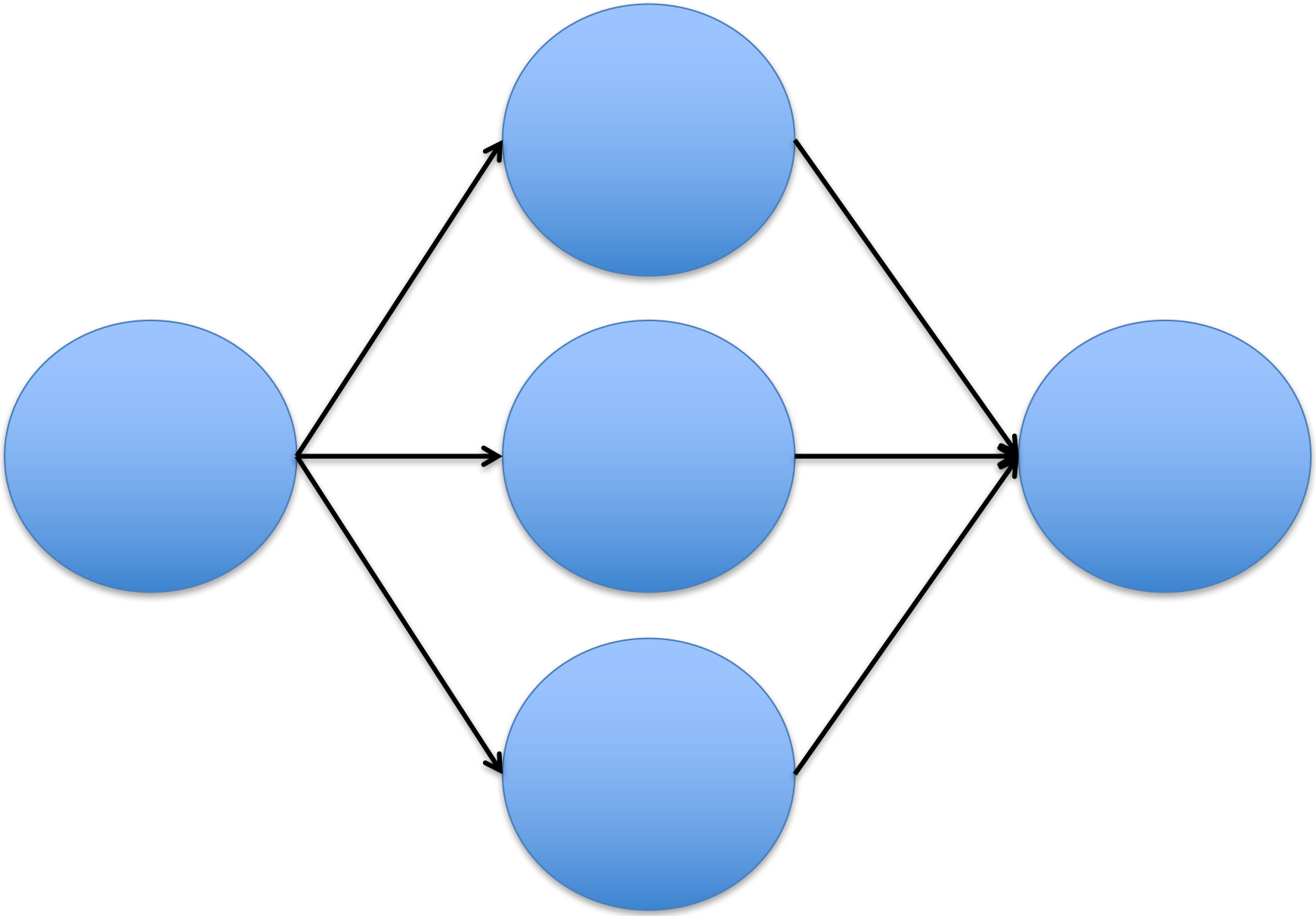
# Publish/Subscriber



# Qual modelo é este?



# É uma simplificação do Map/Reduce



# Mecanismos de comunicação

- Existem diversos mecanismos e protocolos para se implementar a comunicação em SD:
  - Sockets;
  - RPC;
  - DSM (*Distributed Shared Memory*);
  - Map/Reduce;
  - MPI;
  - Etc, etc, etc.

# Comunicação com sockets

- Escrevemos o protocolo do nosso serviço em cima de um protocolo de transmissão (por exemplo, TCP ou UDP).
- O que é TCP?
- O que é UDP?
- Quando é melhor usar TCP?
- Quando é melhor usar UDP?

# TCP e UDP

- TCP (*Transmission Control Protocol)*
  - É um protocolo construído sobre o protocolo de rede IP, o qual possibilita a transmissão em duas vias sobre uma conexão (ou sessão, stream) entre dois *sockets* de maneira sequenciada e confiável.
- UDP (*User Datagram Protocol*)
  - Também é um protocolo construído sobre o protocolo IP, mas que possibilita o melhor esforço de transmissão de *datagramas* simples.

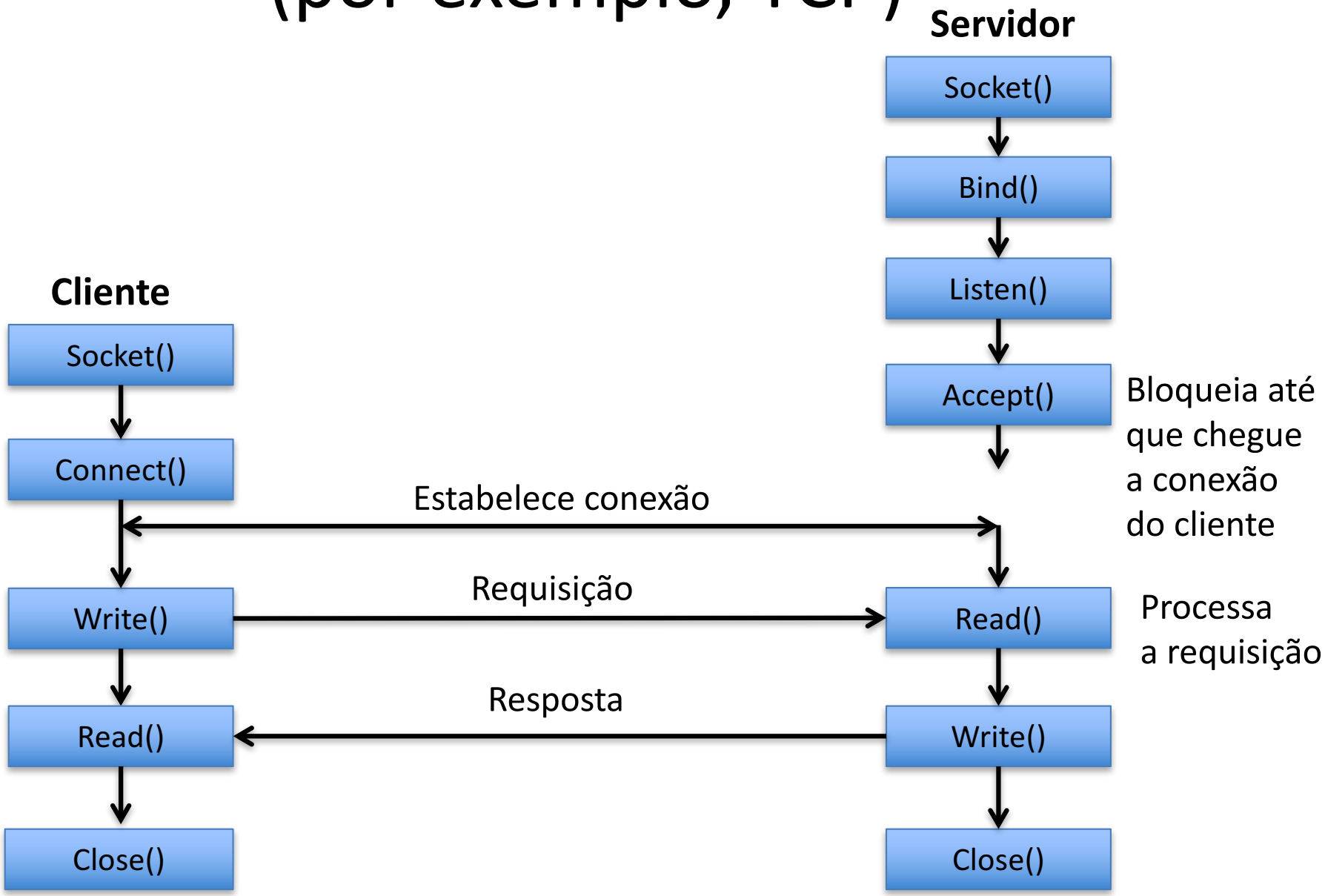
# Quando usar um e outro?

- TCP
  - Usamos TCP quando precisamos de confiabilidade, ou quando a latência de comunicação é pequena (e.g., transmissão de arquivos).
- UDP
  - Usamos UDP quando podemos perder mensagens, reordena-las e duplica-las, mas ainda queremos baixa latência (e.g., jogos online).

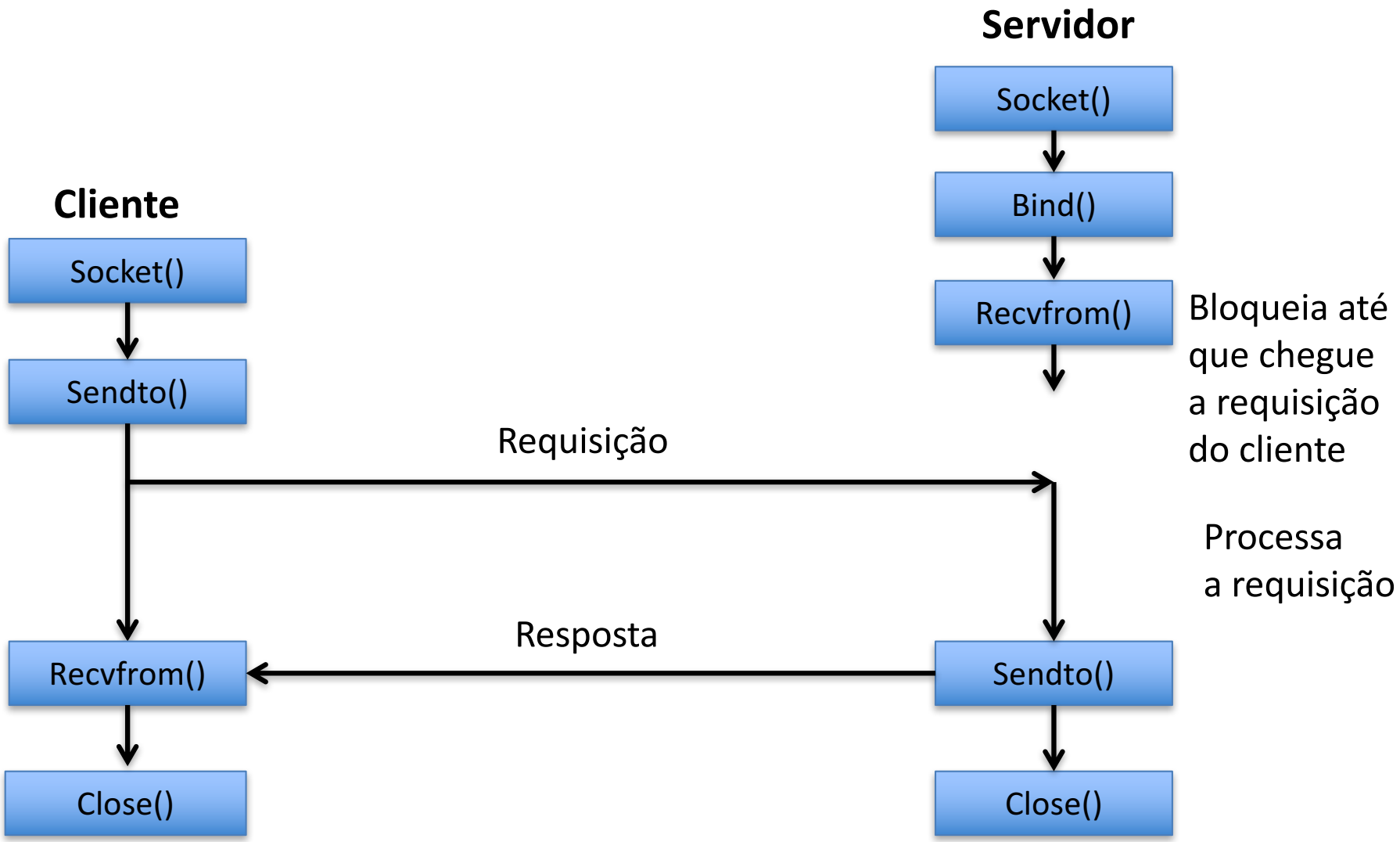
# O que é um socket?

- Um *socket* é uma maneira de ler e escrever mensagens.
- Temos dois tipos básicos de *sockets*:
  - Com conexão – existe o estabelecimento de uma conexão antes da comunicação ser iniciada. Geralmente oferece garantias como confiabilidade e ordem FIFO.
  - Sem conexão – não há conexão preestabelecida e não oferece garantias.

# Cliente-Servidor com conexão (por exemplo, TCP)



# Cliente-Servidor sem conexão (por exemplo, UDP)



# Serviços

- Usamos TCP e UDP para implementar serviços.
- Serviços rodam em um servidor.
- Seu design típico é esperar por requisições e responder essas requisições.
- Como servir mais de um serviço em uma mesma máquina?

# Portas (de 1 até 65535)

- Um servidor pode servir vários serviços diferentes.
- A maneira de distinguir cada serviço diferente é usar uma porta padrão para cada serviço.
- Exemplos:
  - HTTP roda na porta 80;
  - Telnet roda na porta 23;
  - SMTP roda na porta 25;
  - Etc, etc, etc.
- Geralmente não podemos usar uma porta abaixo de 1024 sem acesso de administrador.

# Exemplo: *Daytime server*

- Um dos serviços mais simples que podemos construir é um servidor de data e hora.
- O protocolo desse serviço é definido pela RFC 867.
- A porta padrão é a 13.
- Pode ser implementado tanto em TCP quanto em UDP.

# RFC 867

**Daytime Protocol**

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet that choose to implement a Daytime Protocol are expected to adopt and implement this standard.

A useful debugging and measurement tool is a daytime service. A daytime service simply sends a the current date and time as a character string without regard to the input.

TCP Based Daytime Service

One daytime service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 13. Once a connection is established the current date and time is sent out the connection as a ascii character string (and any data received is thrown away). The service closes the connection after sending the quote.

UDP Based Daytime Service

Another daytime service service is defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 13. When a datagram is received, an answering datagram is sent containing the current date and time as a ASCII character string (the data in the received datagram is ignored).

Daytime Syntax

There is no specific syntax for the daytime. It is recommended that it be limited to the ASCII printing characters, space, carriage return, and line feed. The daytime should be just one line.

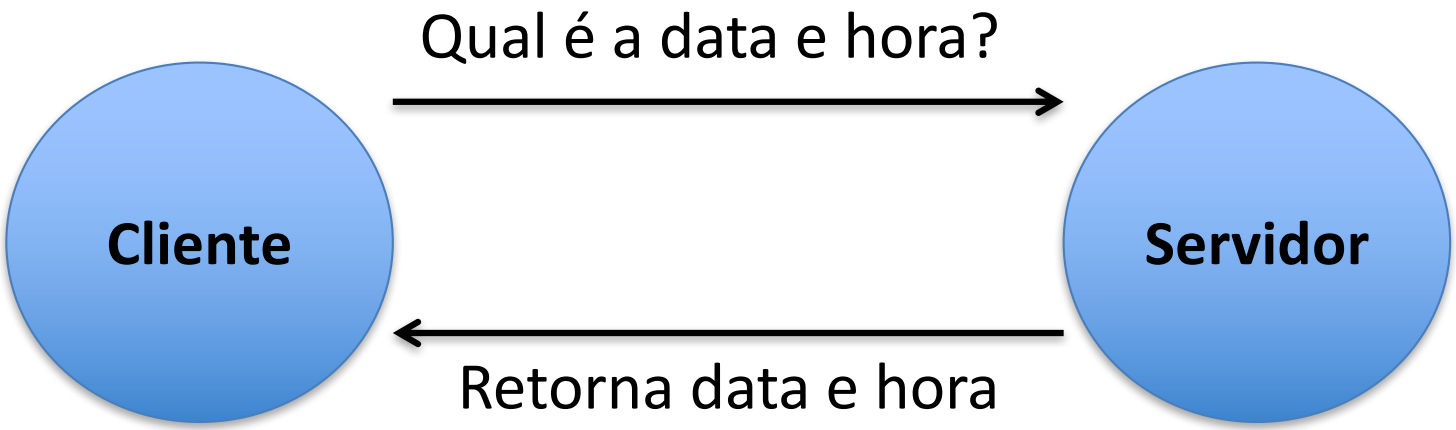
One popular syntax is:

Weekday, Month Day, Year Time-Zone

Example:

Tuesday, February 22, 1982 17:37:43-PST

# O protocolo na prática



# Servidor TCP em Go

```
package main

import (
  "io"
  "os"
  "fmt"
  "net"
  "time"
)

func main () {
  listener, err := net.Listen("tcp", "localhost:13000")
  if err != nil {
    fmt.Println("Erro no Listen: ", err)
    os.Exit(1)
  }
  for {
    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Erro no Accept: ", err)
      continue
    }
    handleConn(conn)
  }
}
```

# Código da função `handleConn`

```
func handleConn (conn net.Conn) {
  defer conn.Close()
  daytime := time.Now().String() + "\n"
  io.WriteString(conn, daytime)
}
```

# Como testar o servidor TCP de data e hora sem escrever um cliente?

# Como testar o servidor TCP de data e hora sem escrever um cliente?

- Podemos usar o telnet:
  - `$ telnet localhost 13000`

# Exercício

- Rode o servidor de data e hora e use o telnet para testa-lo.
- Depois insira um *sleep* de 20 segundos dentro da função `handleConn`.
  - `time.Sleep(time.Second * 20)`
- Rode o servidor novamente e tente efetuar mais de uma conexão ao mesmo tempo.

# Qual é o problema do nosso servidor?

# Qual é o problema do nosso servidor?

- Ele atende apenas um cliente de cada vez!

# Servidor *multithread*

- O nosso servidor atende apenas um usuário de cada vez.
- Como torna-lo *multithread*?
- Isto é, o que devemos mudar no nosso servidor para que ele passe a atender múltiplos usuários?

# Servidor *multithread*

- O nosso servidor atende apenas um usuário de cada vez.
- Como torna-lo *multithread*?
- Isto é, o que devemos mudar no nosso servidor para que ele passe a atender múltiplos usuários?
  - Basta rodar `handleConn` com *goroutines*!

# Servidor TCP *multithread* em Go

```
package main

import (
  "io"
  "os"
  "fmt"
  "net"
  "time"
)

func main () {
  listener, err := net.Listen("tcp", "localhost:13000")
  if err != nil {
    fmt.Println("Erro no Listen: ", err)
    os.Exit(1)
  }
  for {
    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Erro no Accept: ", err)
      continue
    }
    go handleConn(conn)
  }
}
```

# Cliente TCP em Go

```
package main

import "io"
import "os"
import "fmt"
import "net"

func main () {
  conn, err := net.Dial("tcp", "localhost:13000")
  if err != nil {
    fmt.Println("Erro no Dial: ", err)
    os.Exit(1)
  }
  defer conn.Close()
  mustCopy(os.Stdout, conn)
}

func mustCopy (dst io.Writer, src io.Reader) {
  _, err := io.Copy(dst, src)
  if err != nil {
    fmt.Println("Erro no mustCopy: ", err)
    os.Exit(1)
  }
}
```

# Exercício

- Rode o servidor na *espec* e execute o cliente na máquina do laboratório.
- Vamos precisar criar um túnel SSH para conseguir redirecionar a requisição do cliente para o servidor.

# Servidor UDP em Go

```
package main

import (
  "os"
  "fmt"
  "net"
  "time"
)

func main () {
  udpAddr, err := net.ResolveUDPAddr("udp", "localhost:13000")
  if err != nil {
    fmt.Println("Erro no ResolveUDPAddr: ", err)
    os.Exit(1)
  }
  conn, err := net.ListenUDP("udp", udpAddr)
  if err != nil {
    fmt.Println("Erro no ListenUDP: ", err)
    os.Exit(1)
  }
  for {
    handleConn(conn)
  }
}
```

# Código da função handleConn

```
func handleConn (conn *net.UDPConn) {
  var buf [1024]byte
  _, addr, err := conn.ReadFromUDP(buf[0:])
  if err != nil {
    fmt.Println("Erro no ReadFromUDP: ", err)
    return
  }
  daytime := time.Now().String()
  conn.WriteToUDP([]byte(daytime), addr)
}
```

# Cliente UDP em Go

```
package main

import (
  "os"
  "fmt"
  "net"
)

func main () {
  udpAddr, err := net.ResolveUDPAddr("udp", "localhost:13000")
  checkError("Erro no ResolveUDPAddr:", err)
  conn, err := net.DialUDP("udp", nil, udpAddr)
  checkError("Erro no Dial:", err)
  _, err = conn.Write([]byte("data e hora, por favor"))
  checkError("Erro no Write:", err)
  var reply [1024]byte
  n, err := conn.Read(reply[0:])
  checkError("Erro no Read:", err)
  fmt.Println(string(reply[0:n]))
}
```

# Código da função `checkError`

```
func checkError (str string, err error) {
  if err != nil {
    fmt.Println(str, err)
    os.Exit(1)
  }
}
```

# Exercício

- Execute o servidor e o cliente UDP.

# Outro exemplo: servidor de *echo*

- Um servidor de *echo* simplesmente retorna como resposta a própria mensagem que recebeu na requisição.
- O protocolo desse serviço é definido pela RFC 862.
- A porta padrão é a 7.
- Pode ser implementado tanto em TCP quanto em UDP.

# Variação do servidor de *echo*

- O nosso servidor de data e hora usa apenas uma *goroutine* por conexão.
- Vamos construir um servidor de *echo* que usa múltiplas *goroutines* por conexão.
- Vamos simular reverberações de um *echo* real: “OLA!”, “Ola!”, “ola!”, ...

# Código da função `echo`

```
func echo (conn net.Conn,
           shout string,
           delay time.Duration) {
  fmt.Fprintln(conn, "\t", strings.ToUpper(shout))
  time.Sleep(delay)
  fmt.Fprintln(conn, "\t", shout)
  time.Sleep(delay)
  fmt.Fprintln(conn, "\t", strings.ToLower(shout))
}
```

# Código da função `handleConn`

```
func handleConn (conn net.Conn) {
  defer conn.Close()
  input := bufio.NewScanner(conn)
  for input.Scan() {
    go echo(conn, input.Text(), time.Second * 1)
  }
}
```

# Exercício

- Adapte o servidor TCP de data e hora para ser um servidor de *echo* conforme a nossa especificação.
- Adapte o cliente TCP de data e hora para ser um cliente do serviço de *echo* que acabamos de criar.
  - Dica: seu cliente deve ler e escrever ao mesmo tempo.
- Teste a variação do serviço de *echo*.

# Mais um exemplo: servidor de *chat*

```
package main

import (
   "os"
   "fmt"
   "net"
   "bufio"
)

type client chan <- string

var (
   entering = make(chan client)
   leaving = make(chan client)
   messages = make(chan string)
)
```

# Código da função `main`

```
func main () {
  listener, err := net.Listen("tcp",
                                   "localhost:8080")
  if err != nil {
    fmt.Println("Erro no Listen: ", err)
    os.Exit(1)
  }
  go broadcaster()
  for {
    conn, err := listener.Accept()
    if err != nil {
      fmt.Println("Erro no Accept: ", err)
      continue
    }
    go handleConn(conn)
  }
}
```

# Código da função `broadcaster`

```
func broadcaster () {
  clients := make(map[client]bool)
  for {
    select {
      case msg := <- messages:
        for cli := range clients {
          cli <- msg
        }
      case cli := <- entering:
        clients[cli] = true
      case cli := <- leaving:
        delete(clients, cli)
        close(cli)
    }
  }
}
```

# Código da função handleConn

```
func handleConn (conn net.Conn) {
  ch := make(chan string)
  go clientWriter(conn, ch)

  who := conn.RemoteAddr().String()
  ch <- "ID: " + who
  messages <- who + " entrou"
  entering <- ch

  input := bufio.NewScanner(conn)
  for input.Scan() {
    messages <- who + ": " + input.Text()
  }

  leaving <- ch
  messages <- who + " saiu"
  conn.Close()
}
```

# Código da função `clientWriter`

```
func clientWriter (conn net.Conn,
                   ch <- chan string) {
  for msg := range ch {
    fmt.Fprintln(conn, msg)
  }
}
```

# Exercício

- Altere o código do servidor para *logar* na tela toda vez que um usuário entra ou sai do *chat*.
- Rode o servidor em uma máquina do laboratório e use o `telnet` com os seus amigos em várias máquinas diferentes para testar o seu servidor de *chat*.