# *Arquitectura de SD*

*Escola Superior de Tecnologia*
*Eng. Informática e das*
*Tecnologias da Informação*

Sistemas Cliente/Servidor

**2005/2006**

**Pedro António**
**Mário Calha**

## Aula 4 – Comunicação em Sistemas Distribuídos

# Arquitectura de SD

- **Comunicação entre Processos**
  - A comunicação entre processos (*interprocess communication – IPC*) consiste num conjunto de mecanismos de programação que permitem a um programador criar e gerir processos que se executam concorrentemente num sistema operativo.
  - Possibilita que um programa trate de vários pedidos utilizadores ao mesmo tempo.
  - Um simples pedido de utilizador pode resultar na execução de vários processos no sistema operativo, onde normalmente estes processos necessitam de comunicar entre si.
  - Os mecanismos de comunicação entre processos (IPC) tornam esta situação possível.
  - Cada mecanismo IPC tem as suas vantagens e limitações, podendo mesmo surgir que um programa utilize todos os mecanismos disponíveis.

# Arquitectura de SD

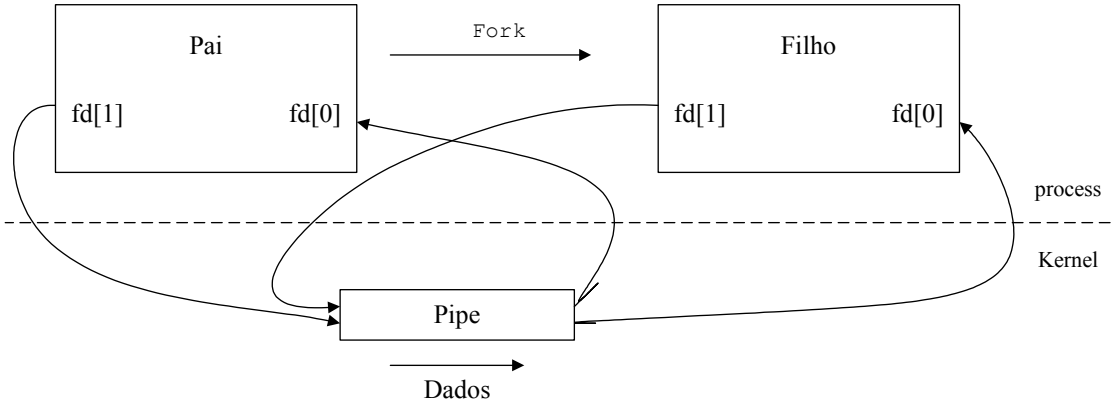
- **Comunicação entre Processos (Cont.)**
  - Mecanismos de comunicação entre processos (IPC):
    - *Pipes*.
    - *Named pipes* ou FIFOs.
    - *Message queueing*.
    - Semáforos.
    - Memória partilhada.
    - *Sockets*.

# Arquitectura de SD

- **Passagem de Mensagens – *Pipes***
  - Um *pipe* é uma técnica para passar informação de um processo para outro. Ao contrário dos outros mecanismos de IPC, um *pipe* fornece apenas comunicação num sentido.
  - Basicamente, um *pipe* passa um parâmetro como saída de um processo para outro processo que o aceita como entrada. O sistema guarda temporariamente a informação até que esta seja lida.
  - Na linha de comandos do UNIX, o *pipe* é representado por uma barra vertical (|) entre dois comandos. A chamada sistema *pipe* é idêntica à chamada dentro de um programa.
  - Uma das limitações dos *pipes* para comunicação entre processos é que os processos que utilizam os *pipes* devem ter um processo pai em comum. Os processos devem partilhar uma inicialização de um processo, resultado da chamada sistema *fork* executada pelo processo pai.
  - Um *pipe* é fixo no tamanho e é, normalmente, de pelo menos 4KB.

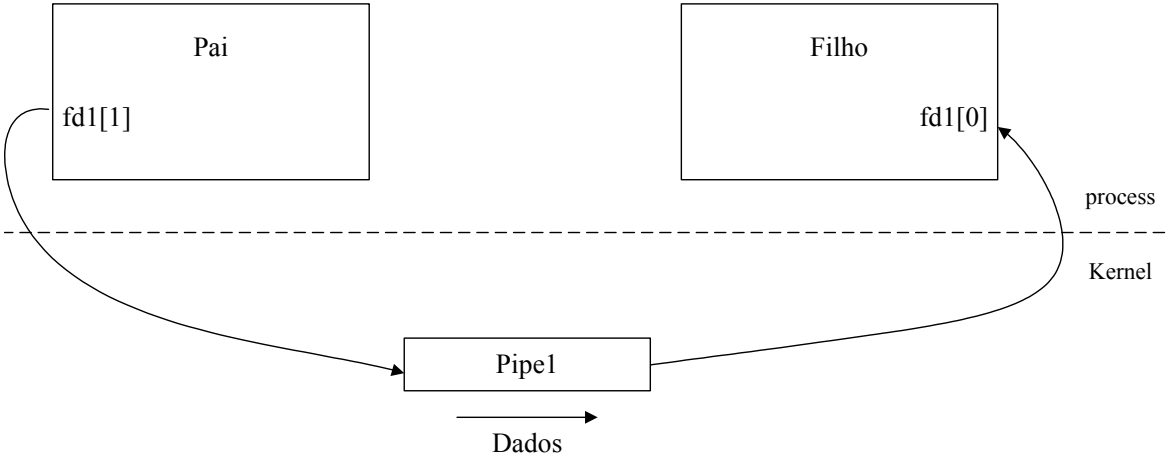
# Arquitectura de SD

- **Passagem de Mensagens – *Pipes* (Cont.)**
  - O *pipe* disponibiliza um fluxo de dados unidireccional. Estes são habitualmente utilizados para comunicação entre 2 processos (pai e filho).
    - Um processo (que irá ser o pai) cria o *pipe*.
    - O processo cria um processo filho (*fork*) que é uma cópia dele próprio.



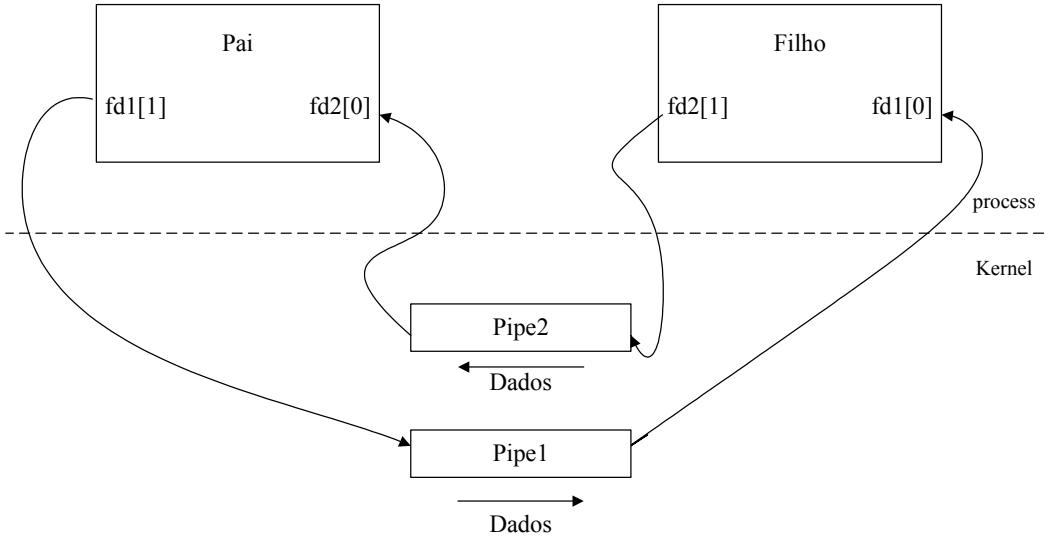
# Arquitectura de SD

- **Passagem de Mensagens – *Pipes* (Cont.)**
  - De seguida, o processo pai fecha o descritor de leitura do *pipe* e o filho fecha o descritor de escrita do *pipe*. Assim, resulta num fluxo de dados unidireccional entre dois processos.



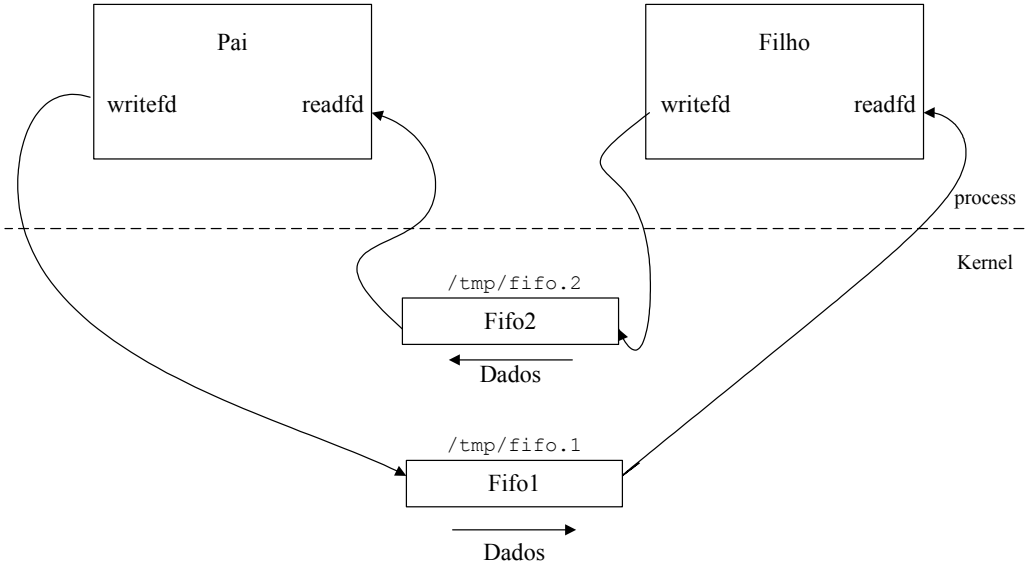
# Arquitectura de SD

- **Passagem de Mensagens – *Pipes* (Cont.)**
  - Se pretendermos ter um fluxo de dados bidireccional, temos de criar dois *pipes* e um em cada direcção:
    - Criar *pipe* 1 (fd1[0] e fd1[1]) e *pipe* 2 (fd2[0] e fd2[1]).
    - *Fork* – criar o processo filho.
    - O pai fecha o descritor de leitura do *pipe* 1 (fd1[0]) e escrita do *pipe* 2 (fd2[1]).
    - O filho fecha os descritor de escrita do *pipe* 1 (fd1[1]) e leitura do *pipe* 2 (fd2[0]).



# Arquitectura de SD

- **Passagem de Mensagens – *FIFOs***
  - Os *pipes* não têm nenhum nome associado a eles, e logo só podem ser usados em processos relacionados. Os FIFOs são semelhantes aos *pipes* com a vantagem que têm um nome associado e logo podem ser usados por processos não relacionados. Os FIFOs também proporcionam um fluxo de dados unidireccional. Estes são também chamados de *named pipes*.



# Arquitectura de SD

- **Passagem de Mensagens – *FIFOs* (Cont.)**
  - As diferenças no código de um programa que usa *pipes* para um programa que usa FIFOs são as seguintes:
    - Para criar e abrir um *pipe* é necessário chamar a função `pipe()`. Para criar um FIFO utiliza-se a função `mkfifo()`, para abrir os descritores do FIFO usa-se a função `open()`, ou outra semelhante, `fopen()`.
    - Um *pipe* desaparece automaticamente no último fecho dos seus descritores, o FIFO apenas é destruido do disco quando se chama a função `unlink()`.
  - Os FIFOs podem ser utilizados entre processos independentes.
  - Ao contrário dos *pipes*, um FIFO pode ser utilizado por processos que não têm de partilhar a origem de um processo comum. As mensagens enviadas para um FIFO podem ser lidas por qualquer processo autorizado que conhece o nome do FIFO.
  - Um *named pipe* é chamado FIFO, porque os primeiros dados escritos no *pipe* são os primeiros dados a serem lidos.

# Arquitectura de SD

- **Passagem de Mensagens – *Message Queues***
  - Message Queues → Fila de Mensagens
  - *Message queueing* é um método pelo qual os processos podem trocar ou passar dados utilizando uma interface de um sistema de gestão de fila de mensagens.
  - As mensagens podem ter tamanhos diferentes e podem ser atribuídos diferentes tipos e utilizações.
  - A fila de mensagens pode ser criada por um processo e utilizada por vários processos que lêem e/ou escrevem mensagens na fila.
  - Por exemplo:
    - Um processo servidor pode ler e escrever mensagens de e para uma fila de mensagens criada por processos clientes.
  - O tipo das mensagens pode ser utilizado para associar uma mensagem com um processo cliente em particular, mesmo que estejam todas as mensagens numa única fila.

# Arquitectura de SD

- **Passagem de Mensagens – *Message Queues* (Cont.)**
  - A fila de mensagens é gerida pelo sistema operativo (ou *kernel*).
  - Os processos criam as filas de mensagens e enviam e recebem mensagens utilizando uma API.
    - Em sistemas UNIX, na linguagem de programação C a função `msgget` é utilizada com vários parâmetros que especificam a acção requerida, o identificador da fila de mensagens, o tipo da mensagem, etc.
  - O tamanho máximo de uma mensagem numa fila de mensagens é limitado pelo sistema operativo e é, tipicamente, de 8KB.

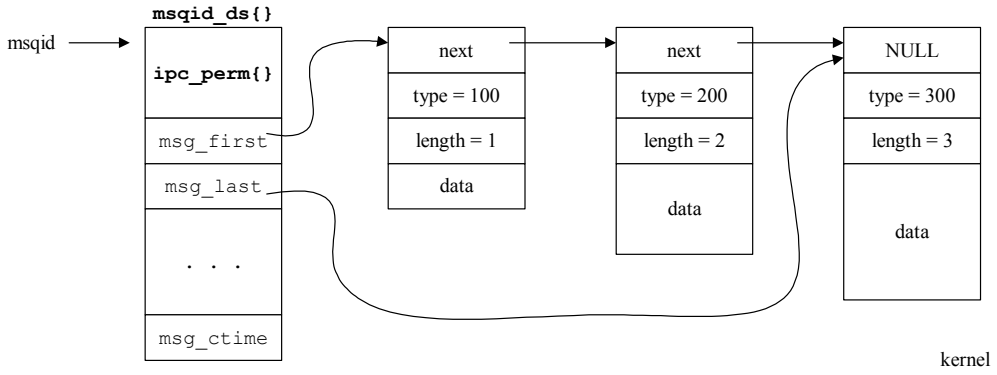
```
#include <sys/msg.h>

int msgget(key_t key, int oflag);
int msgsnd(int msqid, const void *ptr, size_t lenght, int flag);
ssize_t msgrcv(int msqid, void *ptr, size_t lenght, long type, int flag);
int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

# Arquitectura de SD

- **Passagem de Mensagens – *Message Queues* (Cont.)**

```
struct msqid_ds{
struct ipc_perm     msg_perm;          /*read-write permissions, pag 3.7*/
struct msg          *msg_first;        /*ptr to first message on queue*/
struct msg          *msg_last;         /*ptr to last message on queue*/
msglen_t            msg_cbytes;        /*current # bytes on queue*/
msgqnum_t           msg_qnum;          /*current # of messages on queue*/
msglen_t            msg_qbytes;        /*max # of bytes allowed on queue*/
pid_t               msg_lspid;         /*pid of last msgsnd()*/
pid_t               msg_lrpid;         /*pid of last msgrcv()*/
time_t              msg_stime;         /*time of last msgsnd()*/
time_t              msg_rtime;         /*time of last msgrcv()*/
time_t              msg_ctime;         /*time of last msgctl()*/
};
```



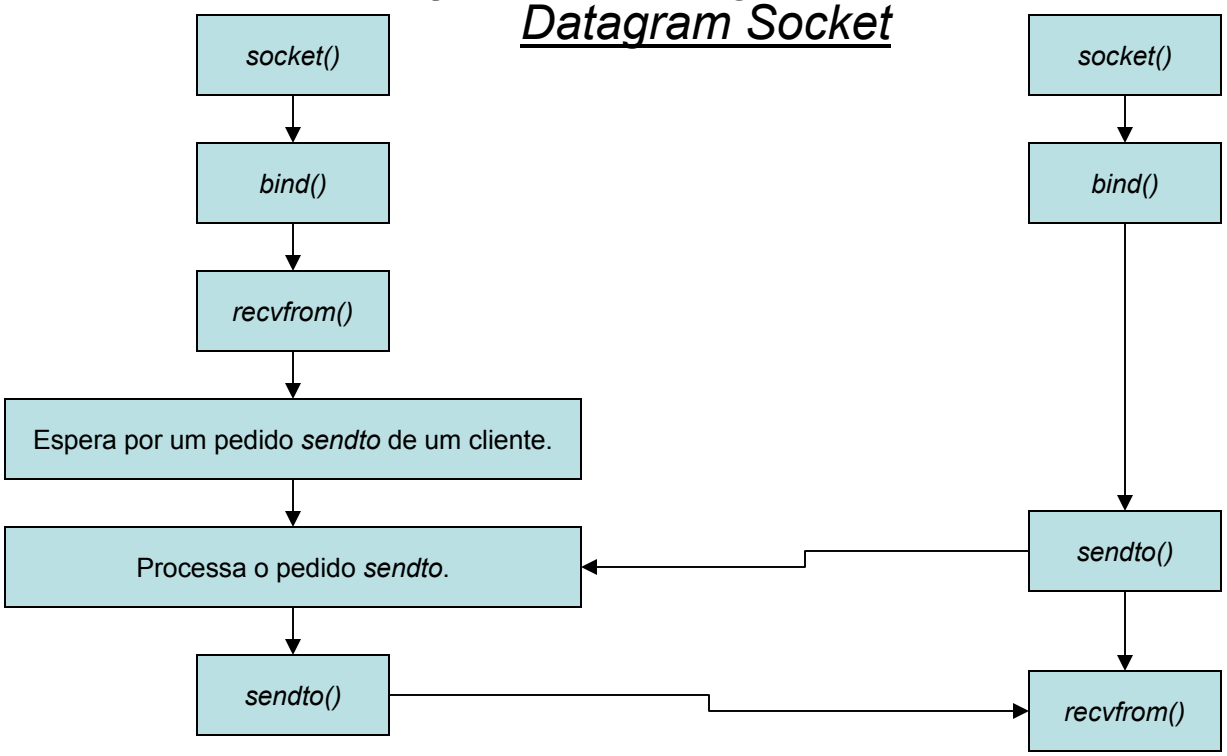
# Arquitectura de SD

- **Passagem de Mensagens – *Sockets***
  - Os *sockets* podem ser considerados como uma evolução dos *pipes*, dado que se inserem na estrutura habitual dos descritores de ficheiros do UNIX.
  - Os *sockets* são bidireccionais e foram concebidos para responder às necessidades de comunicação entre processos.
  - Os *sockets* são criados num determinado domínio que especifica os protocolos utilizados e as convenções de identificação.
  - Os *sockets* consiste num mecanismo de comunicação entre um programa cliente e um programa servidor numa rede.
  - Os sockets também podem ser utilizados para implementar comunicação entre processos num mesmo computador.

# Arquitectura de SD

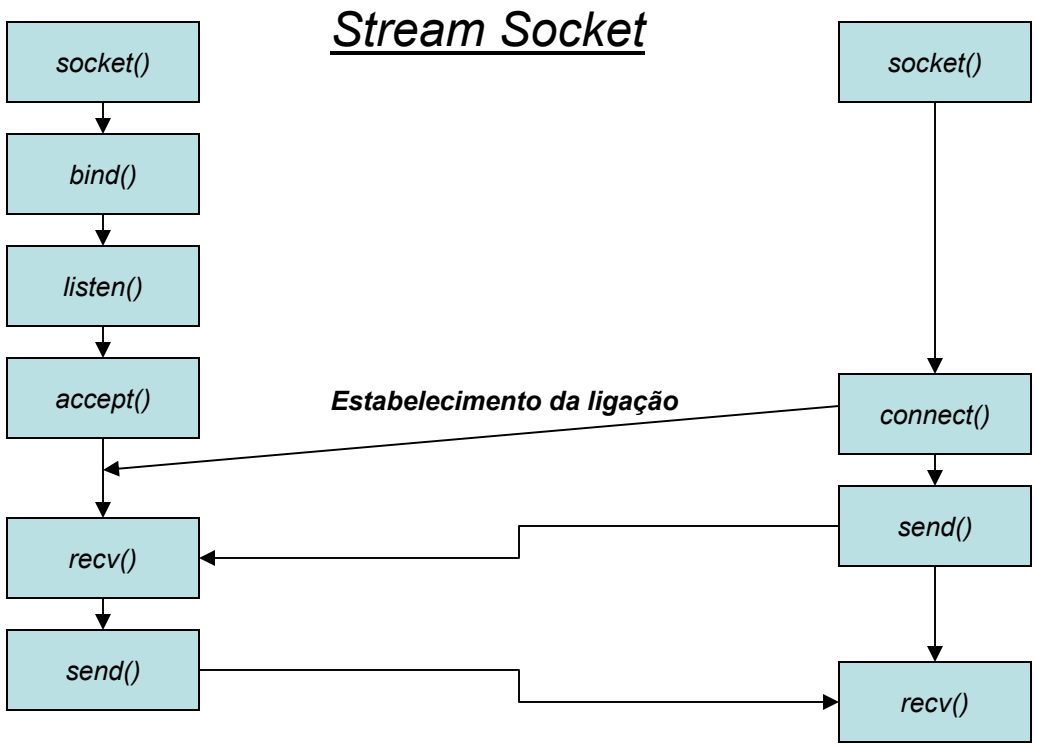
- **Passagem de Mensagens – *Sockets* (Cont.)**
  - Sequência de instruções de um servidor e um cliente *connetionless* (comunicação bidireccional, não sendo garantida a sequencialidade, fiabilidade e eliminação de mensagens duplicadas):

*Datagram Socket*



# Arquitectura de SD

- **Passagem de Mensagens – *Sockets* (Cont.)**
  - Sequência de instruções de um servidor e um cliente *connetion-oriented* (comunicação bidireccional, fiável e sequencial):



# Arquitectura de SD

- **Sincronização – Semáforos**
  - Os semáforos são técnicas para coordenação e sincronização de actividades, nas quais existem vários processos que competem pelos mesmos recursos.
  - Um semáforo é um valor que está guardado no sistema operativo (ou *kernel*) e que pode ser consultado e modificado por cada processo.
  - Dependendo do valor do semáforo, o processo pode utilizar o recurso ou fica a saber que o recurso já está a ser utilizado e, consequentemente, deve aguardar algum tempo até voltar a tentar.
  - Os semáforos podem ser binários ou podem ter valores adicionais.
  - Normalmente, um processo que utiliza semáforos verifica o seu valor e, de seguida, modifica o valor do semáforo para reflectir a aquisição do recurso.
  - Os semáforos são utilizados com os seguintes propósitos:
    - Partilhar um espaço de memória comum.
    - Aceder a ficheiros partilhados.

# Arquitectura de SD

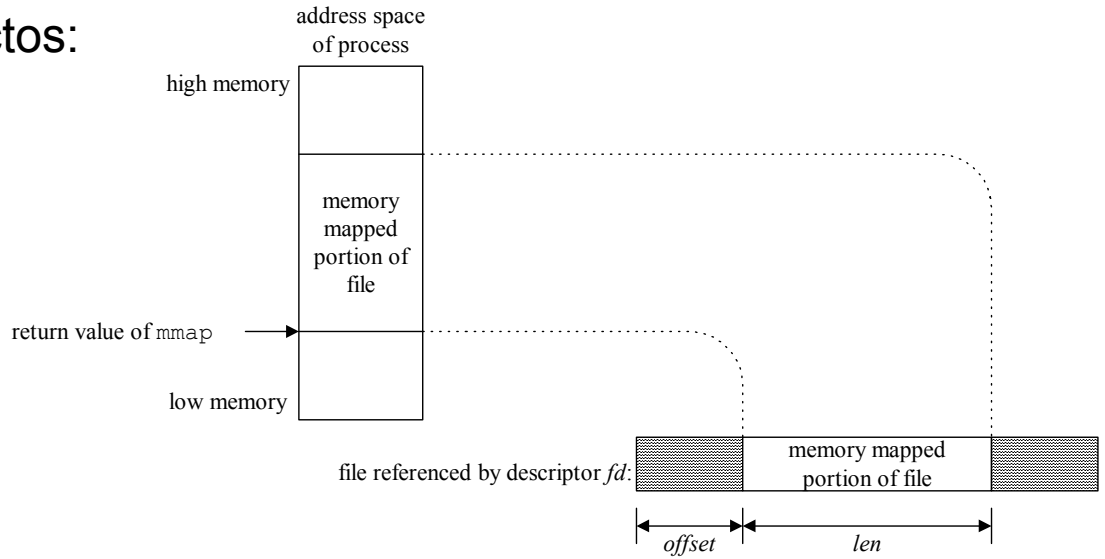
- **Sincronização – Semáforos (Cont.)**
  - Para além dos semáforos existem outros mecanismos de sincronização:
    - ***Mutexes* –** Mutex significa *mutual exclusion* e representa a forma mais básica de sincronização. Este protege dados, acedidos na secção crítica, que são partilhados entre vários processos.
    - **Variáveis de condição –** Permitem que os processos aguardem pela ocorrência de condições. Este mecanismo baseia-se na utilização do anterior.
    - **Trincos de Leitura e Escrita –** Este mecanismo distingue a obtenção do fecho para leitura da obtenção do fecho para escrita. Também é conhecido como *shared-exclusive*.
    - **Trincos de Gravação –** Este mecanismo é uma extensão do trinco de leitura/escrita e pode ser utilizado entre processos não relacionados para partilha da leitura e escrita de um ficheiro.

# Arquitectura de SD

- **Memória Partilhada**
  - A memória partilhada é um mecanismo pelo qual um processo pode trocar dados mais rapidamente que realizando leituras e escritas utilizando os serviços do sistema operativo.
  - Por exemplo:
    - Um processo cliente pode ter de passar dados para um processo servidor que modifica os dados e retorna-os ao cliente.
    - Normalmente, seria necessário que o cliente escrevesse num ficheiro e o servidor iria ler os dados desse ficheiro.
    - Utilizando uma zona de memória partilhada, os dados podem ser acedidos imediatamente pelos dois processos sem utilizar serviços do sistema operativo.
    - Para colocar dados na zona de memória partilhada, o cliente adquire acesso à memória partilhada verificando o valor de um semáforo, escreve os dados e liberta a memória partilhada (libertando o semáforo), assinalando o servidor que está à espera do acesso à memória partilhada.
    - O servidor processa os dados e actualiza os dados. Liberta a zona de memória partilhada para o cliente aceder aos dados modificados.

# Arquitectura de SD

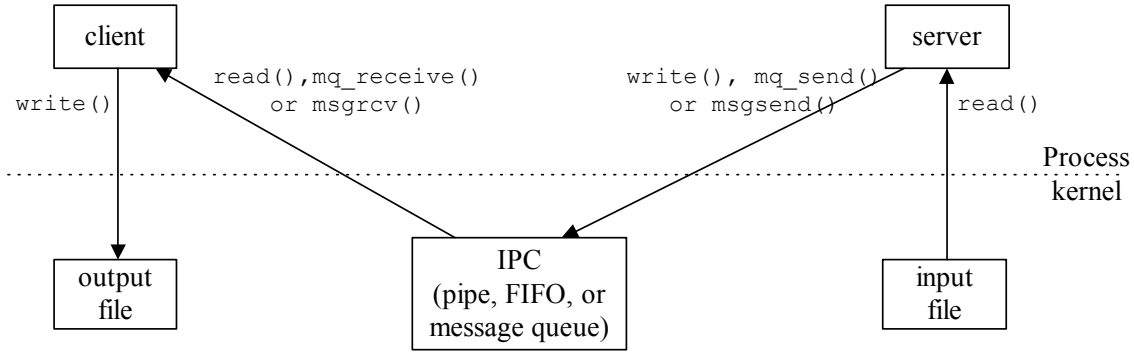
- **Memória Partilhada (Cont.)**
  - Este é o mecanismo de comunicação entre processos mais rápido. Assim que a memória é mapeada, para o espaço de endereçamento dos processos que partilham a região de memória, deixa de haver intervenção do *kernel*.
  - A utilização de memória partilhada envolve, normalmente, a utilização de um dispositivo de sincronização entre os vários processos.
  - Mapeamento de objectos:



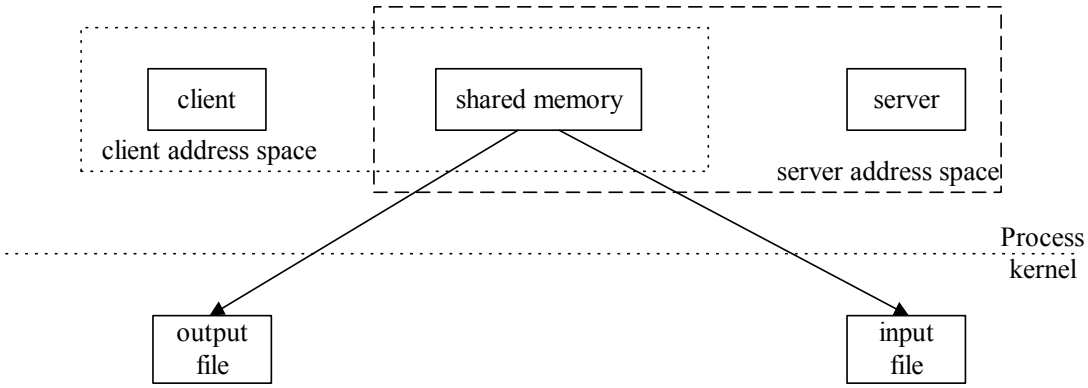
# Arquitectura de SD

- **Memória Partilhada (Cont.)**

**Comunicação utilizando**
- **Pipes,**
- **FIFOs ou**
- **Message queues**




**Comunicação utilizando memória partilhada**



# Arquitectura de SD

- **Memória Partilhada (Cont.)**

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t len);
int msync(void *addr, size_t len, int flags);
```

# Arquitectura de SD

- **Memória Partilhada vs. Passagem de Mensagens**
  - Os processos necessitam de comunicar entre si, de modo a partilhar informação.
  - Em alguns sistemas antigos, os processos tinham a possibilidade de partilhar entre si todo o espaço de memória. Obviamente, os resultados eram, por vezes, caóticos uma vez que era necessário implementar sincronização para coordenar o acesso a informação partilhada.
  - Uma alternativa consiste em não possibilitar a partilha da memória, mas assim não era possível a cooperação de processos na resolução de um problema. Apenas deve ser permitida a partilha de memória em determinadas situações (por exemplo, variáveis).
  - O sistema operativo deve possuir mecanismos através dos quais os processos consigam copiar informação de um processo para o outro.

# Arquitectura de SD

- **Memória Partilhada vs. Passagem de Mensagens (Cont.)**
  - Um processo consiste num ambiente de execução com uma ou mais *threads*. Um ambiente de execução é composto por:
    - Espaço de endereçamento.
    - Recursos de sincronização e coordenação de *threads* tal como semáforos e interfaces de comunicação (portos).
  - O espaço de endereçamento é o componente mais complexo de criar e gerir num ambiente de execução.
  - Uma região de memória é uma área de memória virtual que é acedida por uma *thread*. As regiões de memória não se sobrepõem.
  - Propriedades de uma região de memória:
    - Extensão (endereçamento virtual e tamanho).
    - Permissões de leitura, escrita e execução das *threads*.
  - A existência de um número indefinido de regiões é motivado por vários factores. Um dos factores consiste na partilha de dados entre processos, ou entre processos e *kernel*, num único computador. Uma região de memória partilhada é uma região sustentada pela mesma memória física.

# Arquitectura de SD

- **Memória Partilhada vs. Passagem de Mensagens (Cont.)**
  - Os processos acedem aos mesmos conteúdos de memória nas regiões que são partilhadas:
    - **Bibliotecas:** o código das bibliotecas pode ser demasiadamente grande e ocupar um espaço de memória considerável se o código das bibliotecas for carregado em cada processo que a utiliza. Em vez disto, uma cópia do código da biblioteca pode ser partilhado, mapeando uma região no espaço de endereçamento dos processos que necessitam da biblioteca.
    - **Kernel:** o código e os dados do *kernel* são mapeados em todos os espaços de endereçamento no mesmo local. Quando um processo efectua uma chamada sistema ou gera uma excepção, não há necessidade de criar um novo espaço de endereçamento.
    - **Partilha de dados e comunicação:** Dois processos, ou um processo e o *kernel*, podem necessitar de partilhar dados, de modo a cooperarem na execução de uma tarefa. Pode ser consideravelmente mais eficiente se os dados a partilhar forem mapeados como regiões, em ambos os espaços de endereçamento, em vez de serem passados em mensagens.

# Arquitectura de SD

- **Memória Partilhada vs. Passagem de Mensagens (Cont.)**
  - Como é natural, a distribuição introduz um conjunto de problemas ou agudiza outros. Por exemplo, na comunicação por mensagens, a fiabilidade e o determinismo da comunicação por mensagens é totalmente diferente da habitual nos sistemas centralizados, onde se efectua com base numa memória partilhada.
  - Em sistemas distribuídos, a memória partilhada pode ser utilizada:
    - Na transferência de ficheiros de grande dimensão entre processos locais.
    - Para comunicações rápidas entre um processo e o *kernel*, ou entre dois processos. Os dados são escritos e lidos de uma região partilhada. Assim, os dados são passados eficientemente, sem copiá-los para o espaço de endereçamento do *kernel*. No entanto, é sempre necessário implementar esquemas de sincronização.
  - Uma região de memória partilhada deve ser criada apenas se for utilizada vezes suficientes para compensar o custo de criação.

# Arquitectura de SD

- **Memória Partilhada vs. Passagem de Mensagens (Cont.)**
  - Comunicação exclusivamente por mensagens
    - Apesar de numerosos sistemas multiprogramados apresentarem mecanismos de comunicação baseado em mensagens, no interior do *kernel* toda a comunicação é efectuada através de um espaço de endereçamento partilhado. Num ambiente distribuído, a utilização de meios de comunicação que não partilham a mesma memória física comum introduz características diferentes no modelo de comunicação.
    - A comunicação à distância fica sujeita a um conjunto de factores que podem afectar a fiabilidade.
    - Na maioria das redes, as relações de ordem entre mensagens enviadas e recebidas não são determinísticas, devido a perdas, duplicações de mensagens e atrasos introduzidos por outras entidades.
    - Para garantir a fiabilidade e sequenciamento das mensagens, é necessário um nível suplementar de protocolos que assegurem estas propriedades.

# Arquitectura de SD

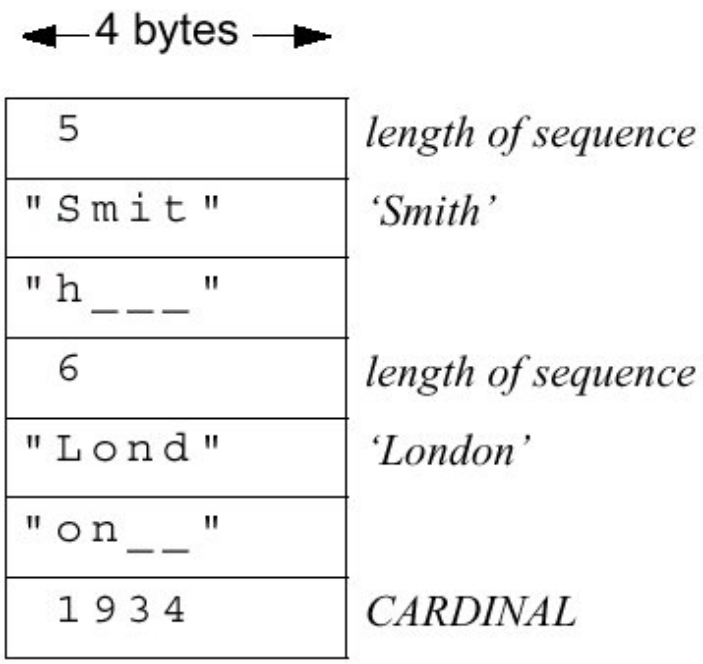
- **Comunicação em Sistemas Distribuídos**
  - **Mapeamento de estruturas de dados para mensagens**
    - Os dados, em programas, são representados como estruturas de dados enquanto que a informação em mensagens é sequencial.
    - Independentemente da forma de comunicação utilizada, as estruturas de dados têm de ser aplanadas (serializadas) antes da transmissão e reconstruídas após a recepção. Dado que diferentes computadores armazenam os vários tipos de dados de diferentes maneiras então para que quaisquer dois computadores possam trocar dados:
      - Os valores são convertidos para uma forma externa combinada antes da transmissão e convertidos para a forma local após a recepção;
      - Para comunicações entre computadores do mesmo tipo, a conversão para uma forma externa pode ser omitida;
      - Uma alternativa à utilização de uma representação externa de dados é a transmissão de dados na sua forma nativa juntamente com um identificador da arquitectura.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Representação externa de dados**
    - *Sun XDR* (*External Data Representation*) e *Courier* [*Xerox*] são exemplos de standards que definem a representação dos tipos de dados, simples e estruturados, mais utilizados. O *Sun XDR* é utilizado para troca de mensagens entre clientes e servidores no *Sun NFS*. O *Courier* é utilizado no *ANSA testbench*.
    - O exemplo a seguir mostra uma mensagem em *Sun XDR* em que a mensagem é constituída por uma sequência de objectos com 4 bytes, em que os números ocupam um objecto e os caracteres se encontram em ASCII.
    - A utilização de um tamanho fixo para cada objecto numa mensagem reduz a carga computacional à custa da largura de banda. Com um sistema deste tipo a largura de banda utilizada é aumentada, visto que são utilizados caracteres de *padding* (ver exemplo).

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Representação externa de dados – Exemplo**



# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - ***Marshalling***
    - *Marshalling* é o processo de montagem de um conjunto de dados numa forma adequada para a sua transmissão numa mensagem. *Unmarshalling* é o processo inverso.
    - Assim o *marshalling* consiste em:
      - Aplanar as estruturas de dados para uma sequência de dados.
      - Tradução desses dados para uma representação externa de dados.
    - As operações de *marshalling* podem ser geradas automaticamente a partir da especificação dos tipos de dados a serem transmitidos na mensagem. Para isso os ficheiros fonte que incluírem especificações de tipo para mensagens serão pré-processadas de modo a serem introduzidas as operações apropriadas de marshalling, para mensagens a serem transmitidas, e *unmarshalling*, para mensagens a serem recebidas.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Operações de Envio e Recepção**
    - A passagem de mensagens pode ser suportada pelas operações *Send* e *Receive*.
    - Para um processo comunicar com outro, envia uma mensagem (sequência de dados) para um destino e outro processo no destino recebe a mensagem. Esta actividade pode envolver sincronização entre os dois processos.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Comunicação Síncrona e Assíncrona**
    - Associado a cada destino de mensagens encontra-se uma fila.
      - Os processos que enviam adicionam mensagens a filas.
      - Os processos que recebem retiram mensagens das filas
    - A comunicação entre processos pode ser síncrona ou assíncrona.
      - Na comunicação síncrona os processos sincronizam-se a cada mensagem. Neste caso, tanto a operação de envio como a de recepção são bloqueantes. Sempre que um processo envia uma mensagem, este bloqueia até que a respectiva recepção seja feita. Sempre que um processo pretende receber alguma mensagem este bloqueia até a receber.
      - Na comunicação assíncrona a operação de envio é não bloqueante, dado que o processo que envia pode prosseguir assim que a mensagem tenha sido copiada para um *buffer* local e a transmissão da mensagem prossegue concorrentemente com a execução do processo. A operação de recepção pode, ou não, ser bloqueante.
        - Se for não bloqueante, o processo que recebe pode prosseguir após a operação de recepção que disponibiliza um *buffer* para ser enchido em background, mas deve receber, separadamente, uma notificação de que o buffer está cheio, através de *polling* ou de interrupção.
        - No caso da operação ser bloqueante poderá estar associado um limite temporal, findo o qual a recepção é abortada.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Destino de Mensagens**
    - Um dos argumentos da operação de envio especifica um identificador que define o destinatário da mensagem. Este identificador deve ser conhecido por qualquer processo com potencial de envio.
    - Identificadores independentes da localização
      - Nos protocolos utilizados na Internet, os endereços de destino são especificados como um porto utilizado por um processo e pelo endereço de Internet do computador em que se encontra em execução. Isto tem o efeito de obrigar a que um determinado serviço seja sempre executado no mesmo computador para que o seu endereço se mantenha sempre válido.
      - No caso dos sistemas distribuídos, um dos seus objectivos principais é a existência de transparência nas localizações. Isto obriga à utilização de identificadores independentes da localização. Estes serão mapeados para um endereço de baixo nível, que permitirá o envio da mensagem, através de software de encaminhamento. Assim num sistema distribuído os serviços podem mudar de computador sem ser necessário informar os clientes dessa mudança.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Destino de Mensagens (Cont.)**
    - Tipos de destinos de mensagens
      - A tabela abaixo mostra a variedade de destinos de mensagens utilizados nos sistemas operativos distribuídos actuais.

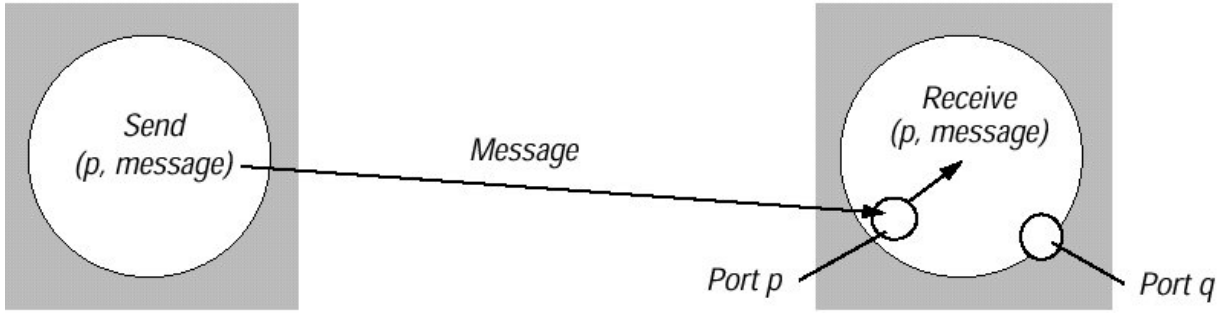
| *Message destinations* | *Operating systems and distributed programming environments* | *Location-independent?* |
|---|---|---|
| Processes | V | yes |
| Ports | Mach, Chorus and Amoeba | yes |
| Sockets | BSD 4.x UNIX (see Section 4.5) | no |
| Groups of processes | V, Amoeba | yes |
| Groups of ports | Chorus | yes |
| Objects | Clouds, Emerald | yes |

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Destino de Mensagens (Cont.)**
    - A maioria dos sistemas operativos optou por utilizar processos ou portos como destinos das mensagens.
    - O porto é um dos muitos pontos alternativos de entrada num processo receptor. Os processos podem utilizar múltiplos portos para recepção de mensagens.
    - Um porto é um destino que tem apenas um receptor, mas pode ter vários emissores. Qualquer processo que conheça o identificador de um porto pode enviar uma mensagem para ele.
    - Alguns sistemas de comunicação entre processos disponibilizam a possibilidade de enviar mensagens para grupos de destinos, processos ou portos. Assim o endereço de destino é um identificador de grupo que é mapeado para um conjunto de destinos da mensagem.
    - Os sistemas operativos *Mach*, *Chorus* e *Amoeba* disponibilizam portos criados dinamicamente com identificadores independentes da localização.

# Arquitectura de SD

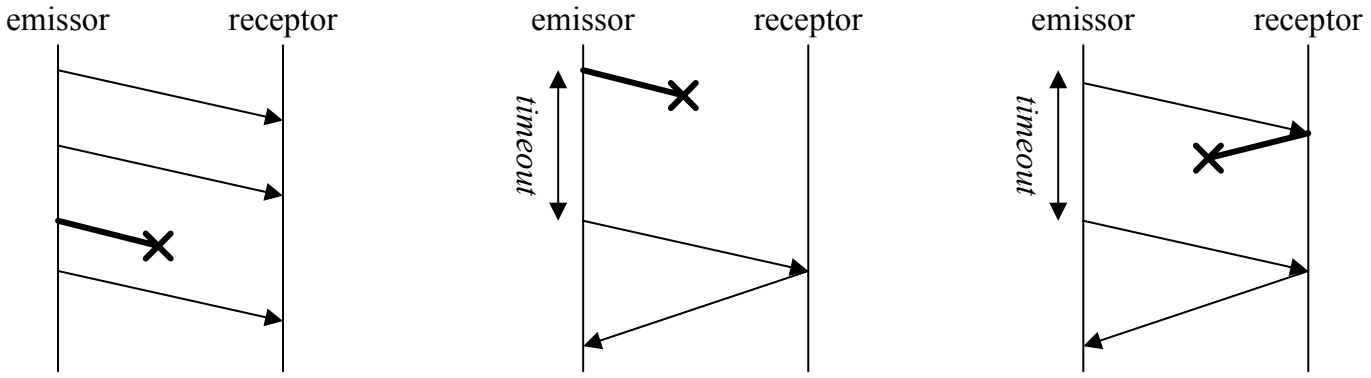
- **Comunicação em Sistemas Distribuídos**
  - **Destino de Mensagens (Cont.)**




  - **Fiabilidade**
    - A designação mensagem não fiável é utilizada quando uma mensagem é transmitida do emissor para o receptor, sem confirmação de recepção ou repetição da transmissão. O protocolo UDP é um exemplo de um protocolo não fiável pois não existe confirmação da recepção de uma mensagem, tendo a vantagem de poupar largura de banda.
    - Um sistema fiável pode ser construído em cima de um não fiável através da utilização de confirmações. As confirmações positivas (confirmação de recepção) são utilizadas na comunicação cliente-servidor. As confirmações negativas (confirmação de não recepção) podem ser úteis em *multicast* de grupo.

# Arquitectura de SD

- **Comunicação em Sistemas Distribuídos**
  - **Fiabilidade (Cont.)**
    - Um identificador de mensagem consiste em:
      - *requestId*, que é tirado de uma sequência de inteiros do processo emissor;
      - Identificador do processo emissor, que poderá indicar o porto pelo qual pode receber respostas.
    - A primeira parte torna o identificador único para o emissor e a segunda parte torna-o único para o sistema distribuído.
    - Primitivas fiáveis vs. não fiáveis:



# Arquitectura de SD

- **Comparações entre 3 Sistemas Operativos**

| *Questão* | *Sistema Operativo de Rede* | *Sistema Operativo Distribuído* | *Sistema Operativo Multiprocess.* |
|---|---|---|---|
| O sistema parece-se como um uni processador virtual? | **Não** | **Sim** | **Sim** |
| Todos têm de ter o mesmo sistema operativo? | **Não** | **Sim** | **Sim** |
| Quantas cópias do sistema operativo existem? | **N** | **N** | **1** |
| Como é realizada a comunicação? | **Ficheiros Partilhados** | **Mensagens** | **Memória Partilhada** |
| Existe concordância quanto aos protocolos de rede? | **Sim** | **Sim** | **Não** |
| Existe alguma fila única a executar-se? | **Não** | **Não** | **Sim** |
| A partilha de ficheiros tem uma semântica bem definida? | **Não** | **Sim** | **Sim** |

# Arquitectura de SD

- **Sumário**
  - Comunicação entre processos (IPC)
    - Passagem de Mensagens
    - Sincronização
    - Memória Partilhada
    - Memória Partilhada vs. Passagem de Mensagens
  - Comunicação em Sistemas Distribuídos
    - Mapeamento de estruturas de dados para mensagens
    - Representação externa de dados
    - Marshalling
    - Operações de Envio e Recepção
    - Comunicação Síncrona e Assíncrona
    - Destino de Mensagens
    - Fiabilidade