

# Resumo - Prova I

Computação Distribuída

\* **Sistema Distribuído** → sistema em que pelo menos um dos componentes fica em uma rede de computadores e se comunicam por mensagens.

↳ compartilhamento de recursos.

### > Definição:

- heterogeneidade:

- redes (protocolos, velocidades ≠)
- hardware (arquiteturas e formatos ≠)
- SOs
- linguagens (modulos de dados e gerenciamento de memória ≠)

- agilidade

- escalabilidade

- gerenciamento de falhas

- transparência: (esconder do usuário a complexidade e distribuição do sistema)

- de acesso (acesso a objetos remotos e locais é falso = )
- de localização (acesso a um recurso sem saber onde ele está).
- de concorrência (vários usuários podem aceder um mesmo recurso sem interferência mutua)
- de replicação (uso de múltiplos cópias de um recurso é invisível)
- de falhas (não mencionados quando possível)
- de migração (recursos se movem sem afetar a operação)
- de desempenho (o sistema se reconfigura p/ manter performance)
- de escala (o sistema pode expandir sem mudar estrutura)

### > Características:

• **concorrência** (rodar os processos simultaneamente em máquinas diferentes).

• **anterioria de um relógio global** (coordenação entre processos feita através da troca de mensagens).

• **extensíveis** (extensão de hardware e software devem ser integráveis ao sistema)

nenhuma máquina conhece

• ataque global

- tolerância a falhas (redundâncias de hardware, recuperação de software, etc).

→ depende tb da distância entre servidores e clientes (atrasos)

### > Propriedades Curtas:

- transparência (a + abstrangeante)

→ atrasos:

\* ambiente dinâmico e não-confiável

- congestionamento
- retenção

- replicação entre servidores
- novas rotas pq → servidor importante

- flexibilidade → núcleo monolítico = gerenciamento de arquivos, diretórios e processos

modelos de estrutura

- micro núcleo
- mecanismo p/ comunicações entre processos
- gerenciamento básico da memória, processos e operações I/O a baixo nível.

→ modularidade (serviços disponíveis a todos os clientes, independentemente de localização, reparos nem causar perda total, usuários podem adicionar mais).

### - confiabilidade

- disponibilidade (quando o sistema pode ser usado)
- exatidão (replicação x consistência)
- segurança (como o servidor pode verificar a origem de uma mensagem)
- tolerância a falhas (replicação x desempenho)

### - desempenho

- escalabilidade → grito de trabalho envolvido no processamento de requisições de acesso a um recurso compartilhado deve independe do tamanho da rede

→ técnicas:

- replicação
- caching
- servidores múltiplos

→ evitores

- componentes centralizadas
- tabelas centralizadas
- algoritmos centralizados

## > Elementos:

- sistemas de nomes:
  - independentes da localização
  - deve escalar bem
  - deve ser acessível por programa.
- responsável pela efetividade ou fracasso de um SD.
- comunicação:
  - desempenho / confiabilidade
  - dilema: otimização comunic. × alto nível de modulação de programação
- estrutura de software:
  - interfaces de software bem definidas
  - serviço = gerenciador de objetos de um tipo
  - interface = conjunto de operações
- alocação de carga de trabalho: otimização do uso da capacidade de processamento, comunicação e recursos de rede
- manutenção de consistência:
  - de atualizações (instantânea)
  - de falha
  - de replicação (cópias de recursos devem ser sincronizadas)
  - de interface de usuário
  - de cache (deve ser propagado)
  - de relógio (minimização e timestamp)
- primordial
- segurança:
  - políticas de segurança inadequadas
  - senhas não criptografadas
  - rotativamente de comunicação → viola privacidade
- ex: cookies!

## > SDs Abertos

- oferecem serviços seguindo regras padronizadas, descritas em protocolos.
- os regras não formalizadas com IDL → Linguagem de Definição de Interface).
- permite que um processo arbitrário que precise de uma interface reconheça como um outro que fornece essa interface.

- o aninh, 2 partes independentes contêm implementações +, SDs +, que funcionam da mesma forma.

## > Middleware

- fornece a interoperabilidade, uma camada de software que se situa entre os SOs e os aplicativos
- ele abstrai a heterogeneidade do hardware, redes e SOs, oferecendo 1 plataforma única transparente p/ construir aplicações distribuídas
  - o ex: CORBA, Java RMI, Web Services, APIs Restful

Também dita como trabalhar em conjunto, com base na mera confiança mútua nos serviços de cada um, especificados por um padrão comum (middleware)

(processual)  
(padrões)



Redes de Comunicação  
Computação Distribuída  
Computação Paralela

Característica	Redes	S. D.	S. P.
Parece um único processador virtual?	Não	Sim	Sim
Todos tem que rodar um mesmo SO?	Não	Não	Sim
Quantas cópias do SO existem	Várias	Várias	1
Como é feita a comunicação?	Arquivos compartilhados	Troca de Mensagens	Memória compartilhada
É necessária padronização de protocolos?	Sim	Sim	Não
Existe apenas uma fila de tarefas?	Não	Não	Sim
Compartilhamento de arquivos tem semântica bem definida?	Normalmente não	Sim	Sim

# \* Comunicação entre Processos (de SDs)

- ↳ se baseia em esquemas de comunicação que definem como os dados trocam.
  - broadcast (tudo p/ todos os nós)
  - comunicação de circuitos (canais dedicados p/ comunicação)
  - comunicação de pacotes (estrutura de armazenamento e encaminhamento de pacotes com base em info de origem/destino)

\* é generalizado no boot do sistema e fica esperando requisição

## > Serviços de Rede:

- modelo cliente/servidor (a + predominante)
  - servidores = programa passivo (fica esperando requisição) que provê um serviço ou recurso *ele que inicia a comunicação*
  - cliente = programa que usa a reunião disponibilizada pelo servidor
- podem estar na = máquina ou em qualquer lugar da rede
  - ↳ Servidor e Cliente não são papéis, não máquinas
  - ↳ interação regida por um protocolo\*
  - Daemon = servidor que fica permanentemente ativa e possui um endereço bem conhecido.
- \*protocolo → conjunto de regras que define como o servidor e cliente vão interagir (define precisamente os comandos aceitos pelo servidor e o formato das mensagens).

## > Protocolos : ↳ Open Systems Interconnection

- OSI → modelo de referência teórica com 7 camadas:

#	Camada	Descrição	Protocolos
7	Aplicação	Atende aos requisitos de comunicação de aplicativos específicos, definindo uma interface para um serviço	HTTP, SMTP, SNMP, FTP, Telnet, SSH, NFS, DNS
6	Apresentação	Transmitem dados em uma representação de rede independente das usadas em cada nó. Criptografia, se exigida é feita nesta camada	Segurança TLS, SMB, AFP
5	Sessão	Realiza operações relacionadas com a confiabilidade das conexões, detecção de falhas e recuperação automática	SIP, SSH, RPC, NetBIOS, ASP
4	Transporte	Nível mais baixo de manipulação das mensagens que são endereçadas para portas de comunicação	TCP, UDP, SPX
3	Rede	Transfere pacotes com base no endereçamento dos nós, o que pode envolver o roteamento entre redes	IP, ICMP, IGMP, X.25, ARP, RARP, BGP, OSPF, RIP, IPX
2	Enlace de dados	Transmite pacotes entre nós fisicamente conectados	Ethernet, Token Ring, PPP, HDLC, Frame Relay, ISDN, ATM, Wi-Fi
1	Física	Transmite sequências de dados binários envolvendo hardware e seus circuitos	Elétrico, radio, laser

- TCP / IP ou UDP / IP → são os protocolos mais praticados
  - IP → protocolo da camada de rede. Endereça 2 máquinas
  - TCP → protocolo de transporte. Cada porta em 1 máquina é um serviço (processo). Orientado a conexões.
  - UDP → parecido com TCP, mas não-orientado a conexões e não garante entrega das mensagens.
- endereçamento → [Endereço IP, Numb. Porta]
  - id. máquina na rede ↗ ↗ id. processo

### > Troca de Mensagens:

- monolíticos: funções, variáveis globais, etc
  - distribuídos: trocas de mensagens
- (em um conjunto)*
- conceito primitivo e de baixo nível.*
- conversão de formato*      *primitivas send e receive*  
*padrão da rede*      *\* marshalling* → empacotamento (preparação de dados para transmissão)  
*\* Unmarshalling* → desempacotamento (recuperação desse conjunto)

Implicit typing ou tipagem implícita: apenas os dados são transmitidos. Requer:

- Linguagem para especificação – ONC RPC XDR e DCE RPC NDR – e compilador para esta linguagem.

Explicit typing ou tipagem explícita: O tipo de cada campo é transmitido.

- Exemplo: JSON, XML e Google Protocol Buffers.

### - modalidades:

- sincrona (bloqueante) = Send permanece bloqueada até que outro processo execute receive e vice-versa
  - ↳ simples e fácil de programar
  - ↳ ↓ desempenho da CPU (bloqueada) → solução: múltiplos threads
- assíncrona (não-bloqueantes) = Send copia a mensagem pr um buffer do Kernel e retorna imediatamente. O receive fornece um buffer pr ser preenchido em background, e o processo é notificado via polling ou interrupções.
  - ↳ ↑ desempenho
  - ↳ ↑ complexo de implementar

Síncrona	Assíncrona
2 cópias da mensagem	4 cópias da mensagem
Remetente espera envio da mensagem	Remetente envia mensagem pro buffer do kernel
Destinatário espera recebimento de mensagem	Destinatário é notificado de chegada de mensagem no buffer do kernel via interrupção ou polling
Simples de programar	Complexo: tratamento de várias linhas de execução
Sub-utilização dos recursos de máquina	Melhor desempenho

## > Abstrações de Nível + Alto da Comunicação:

- ↳ troca de mensagens é pouco natural
- ↳ as abstrações "escondem" a troca de mensagens.
- RPC (Chamada Remota de Procedimento)
  - permite que um cliente invoque um procedimento em um servidor remoto como se fosse uma chamada local
  - stubs = "esqueletos" gerados automaticamente por um compilador a partir de uma interface definida em uma IDL (como XDR)

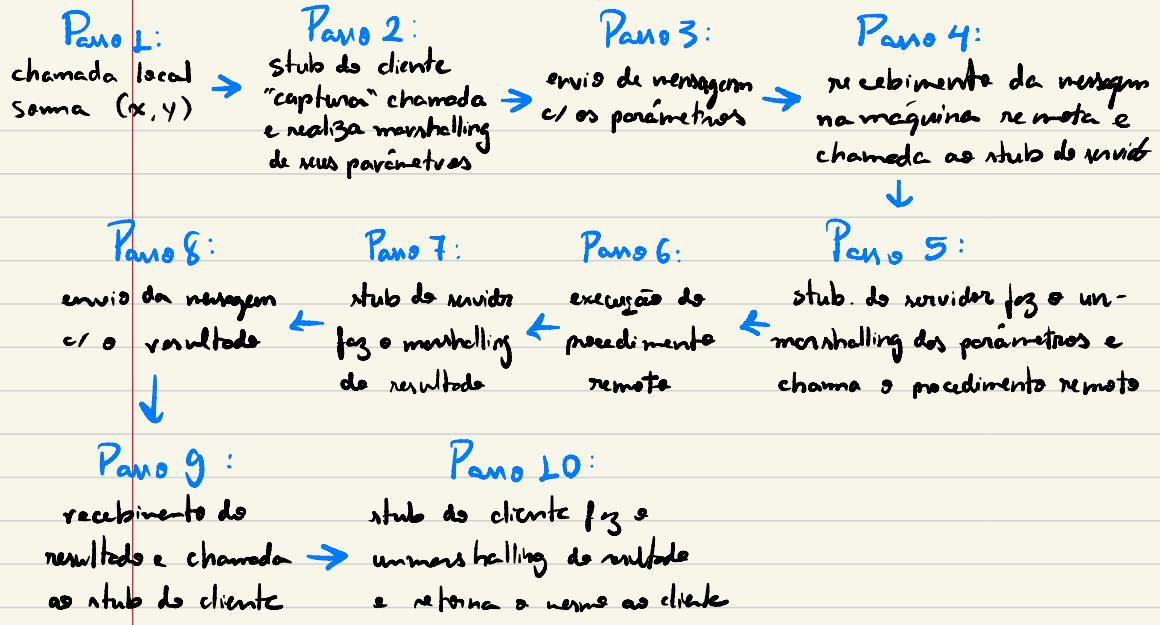
+ Segurança:

o cliente  executa c/ o   
mínimo de  
privilegio

gerado pelo  
componente  
rpcgen  
no Sun RPC

- stub do cliente = se comporta como o procedimento local. Faz o marshalling dos argumentos e unmarshalling dos resultados
  - módulo de comunicação = lida os problemas de rede, como retransmissões e timeouts.
  - dispatcher = recebe a requisição, identifica o procedimento corrente c/ base no ID e encaminha pra stub do servidor correspondente.
  - stub do servidor = faz o unmarshalling dos argumentos, chama o procedimento real de serviço, faz o marshalling do resultado e envia a resposta de volta.
- \* portmap → registra e consulta portos de serviços RPC





**Vantagens**

- princípio da separação de privilégios  $\Rightarrow$  + segurança (código remoto é executado c/ privilégios  $\neq$  do cliente)

\* operações idempotentes:  
- podem ser executadas qualquer nº de vezes, pois não tem efeito colateral.

**Dificuldades**

- parâmetros de parâmetros (é complexo). Soluções:
  - restaurar/copiar valor/marshalled
  - não ter parâmetros por valor.
- ligação/binding  $\Rightarrow$  como o cliente descobre o servidor?
  - estaticamente - endereços
  - dinamicamente - serviço
- semântica de chamador. Com erros, quantas vezes é executada?
  - exactly-once
  - at-least-once
  - at-most-once
- segurança

*→ Solução da SUN para Java*

## \* java RMI (Invocação Remota de Métodos)

permite que programas executados em uma JVM façam referências a objetos de outras JVMs.

	Objeto Local	Objeto Remoto
Definição	classe java (class)	Interface com métodos que serão acessados remotamente, herda da interface Remote
Implementação	local interna à classe	classe remota que implementa a interface acima.
Construção	operador new	Servidor constrói com new, cliente referencia objeto pela rede. Pacote Java 2 Remote Object Activation permite utilização de new remoto.

	Objeto Local	Objeto Remoto
Acesso	variável é uma referência local ao objeto	Objeto remoto acessado por uma referência Proxy ("stub").
Finalização	Método finalize()	interface unreferenced
Exceções	classes que herdam de RuntimeException ou Exception	classes que herdam de RemoteException

## > Implementação de um RMI

- 1- definição de Interface para serviços remotos
- 2- implementação dos serviços remotos
- 3- geração dos arquivos stub e skeleton (proxy)
- 4- servidor hospeda o serviço remoto
- 5- servidor de nomes permite clientes encontrar um servidor remoto
- 6- disponibilização das classes por um servidor HTTP ou FTP
- 7- cliente usa os serviços remotos.

\* Registry → objeto remoto que especia nomes em referências p/ objetos. (nome: String → referência p/ o objeto)  
o rebind → associa um nome a uma referência já existente.  
Usado pelo servidor  
o lookup → retorna uma referência p/ um objeto remoto. Usado pelo cliente.

gerados pelo rmic → { \* stub → parte que executa no cliente → instalado em um diretório especial e buscado automaticamente pelos clientes  
\* skeleton → parte que executa no servidor  
\* rmic → compilador usado p/ gerar stub e skeleton a partir da interface remota.

Dispor os serviços de nomes em uma máquina servidora conhecida → Execuções do Registry → Execução do Servidor (da aplicação RMI)

Exemplo de Aplicação Distribuída com Java RMI

Codificar um cliente que pode chamar os métodos oferecidos pelo banco.

1. Solicitar a adição de um novo cliente que possui um identificador único próprio (id).
2. Solicitar a consulta do saldo de um cliente a partir de seu id.
3. Depositar um determinado valor na conta do usuário de id x.
4. Sacar um determinado valor da conta do usuário de id x.

## > Parâmetro de Parâmetros

- objetos podem ser passados como parâmetros, usando parâmetro por valor como métodos remotos.
  - utilizar a interface `java.io.Serializable`
  - objeto passado como argumento → é serializado (marshaling) de lado do cliente e deserializado de lado do servidor, efetuando uma paragem por valor.

## > Portas dinâmicas e Firewalls

→ **desafios em ambientes restritivos**

- RMI usa portas com alocação dinâmica, mas Firewalls normalmente bloqueiam portas TCP não-padrão
- Solução:
  - configurar um port range fixo p/ exportação de objetos
  - usar uma `RMI SocketFactory` personalizada que encapsula o tráfego RMI dentro de túneis HTTP/HTTPS, contornando o Firewall

## > Conclusão:

- RMI é rápido e flexível p/ criar conexões seguras entre objetos específicos

### vantagens

- portável entre plataformas que tem implen. em Java
- mobilidade de objetos

### vs desvantagens

- solução proprietária da linguagem em Java.

## \* CORBA (Common Object Request Broker Architecture)

Objetivo é permitir que objetos escritos em linguagens diferentes se comuniquem.

- fornece um ambiente de rede padronizado e aberto.

## > ORB (Object Request Broker):

- ↳ "barreira lógica de software" que controla toda a comunicação entre os componentes da arquitetura
- transparência de localização, implementação e acesso
- ajuda um cliente a invocar um método em um objeto

→ seguindo o estilos RMI

## \* O CORBA é a especificação de um ORB

### Java RMI vs CORBA

- O programador define interfaces remotas e depois usa um compilador para produzir os proxies (stubs) e esqueletos.
  - No CORBA, os proxies são gerados na linguagem do cliente e os esqueletos, na linguagem do servidor.
- Os clientes não são necessariamente objetos, podendo ser qualquer programa que envie mensagens de requisição para objetos remotos e receba respostas.
- Um objeto CORBA pode ser implementado por uma linguagem não orientada a objeto.
  - Instâncias de classes não podem ser passadas como argumentos: usa-se estruturas de dados de vários tipos e complexidade arbitrária.

## > IDL (Interface Definition Language)

↳ linguagem declarativa baseada em C++ com palavras-chave específicas p/ distribuição (interface, in, out, inout, oneway, return).

- é compilada p/ gerar os stubs (cliente) e esqueletos (servidor)  
↳ encapsula a complexidade da comunicação remota.
- é independente de linguagens de programação (java, C++, C, Python, Ada, Smalltalk, COBOL)

## > Modelo de Objetos

- Distinção entre objetos CORBA e Servants

entidades abstratas que  
recebem requisições

objects que implementam  
objetos CORBA

implementações concretas que  
executam a lógica dos objetos.

+ flexibilidade na alocação de recursos ↗

- Um mesmo servant pode implementar múltiplos objetos CORBA.
- invocação → segue o modelo at-most-once (coneway permite até talvez p/ os que não precisam de confirmação).
- comunicações entre componentes → protocolo IIOP (Internet Inter-ORB Protocol), baseado em TCP/IP
- referências a objetos → feitos com IORs (Introspectable Object References) que tem 3 componentes:
  - repository ID (identifica a interface do objeto)
  - endpoint info (conexões, portos, etc)
  - object Key (identificativo do objeto no servidor)
- manipulação de IOR →
  - conversão para string (orb.object\_to\_string())
  - recuperação de objeto (orb.string\_to\_object())
  - necessidade de narrow() p/ conversão de tipo

## > Ciclo de Desenvolvimento

- 1- definir as interfaces em IDL
- 2- compilar IDL p/ gerar stubs e skeletons
- 3- implementar servants no servidor
- 4- implementar código do servidor (registrar na ORB) }
- 5- implementar cliente c/ invocações remotas o POA
- 6- compilar e executar
  - políticas de ciclo de vida
  - portabilidadeatua como "cola" entre o ORB e servants

## Estrutura Completa

- Cliente: aplicação que invoca métodos remotos
- Stub: proxy local que representa o objeto remoto
- ORB Core: módulo de comunicação responsável pelo transporte
- Skeleton: recebe requisições e as repassa ao servant
- POA (Portable Object Adapter): gerencia ciclo de vida dos objetos, permite portabilidade entre ORBs de diferentes fabricantes e separa criação de objetos CORBA da criação de servants
- Servant: implementação real do objeto

## > Econet interna JAVA

- Java IDL fornece a implementação padrão
- compilador idlj gera automaticamente 6 arquivos (interface remota, operações, classes PDA, stubs, helpers e holds)
- serviço de nomes CORBA → permite localização dinâmica de objetos.
- Repetidores de interface e implementação → armazenam metadados

## > Fluxo de Mensagens

Requisição do cliente → ORB → IIOP → <sup>ORB</sup>  
provider → PDA → Servant

## > Conclusões

→ CORBA tem presença significativa em sistemas legados críticos (robustez e maturidade), mas vem sendo substituído pelos + leves (Rest APIs, Web Services e gRPC)

### vantagens

- separação entre interface e implementação
- definições de contratos independentes da plataforma
- mecanismo de transparência de acesso.

vs

### desvantagens

- pesado e
- alta complexidade

acesso concorrente  
pode levar a  
inconsistências )

## \* Sincronização de Processos:

- ↳ processos são considerados cooperantes quando afetam ou não afetados por outros processos, podendo compartilhar um espaço de end. lógico ou arquivos.
  - ↳ (threads)

### > Inconsistências

- ↳ acesso concorrente → possíveis inconsistências.
  - ex: caso produtor / consumidor (2 processos achaem um buffer compartilhado, que rodados paralelamente geram "race conditions").
    - ↳ ação crítica

### > Técnicas p/ solução de ação crítica:

- ↳ devem satisfazer esses 3 requisitos:
  - exclusão mútua → se um processo executa na ação crítica, nenhum outro mais pode.
  - progressão → se nenhum processo está na crítica e vários querem entrar, só os que querem decidem quem entra.
  - espera limitada → deve existir um limite superior p/ o n. de vezes que outros processos não permitidos a entrar na ação crítica depois de um querer entrar (previtar starvation - adiamento indefinido).
    - ↳ (fazer requisição)

- ↳ Em sistemas centralizados → técnicas como semáforos, monitores e variáveis de condição na mem. compartilhada

- ↳ Em sistemas distribuídos → sincronização por troca de mensagens, implementadas por algoritmos distribuídos.

## > Sincronizações Física de Relógios

- 1948 - invenção do relógio atômico
- Tempo Atômico Internacional (TAI): escala de tempo precisa e independente das variações da rotação ( $\varphi$ )
- Tempo Universal Coordenado (UTC): TAI + correções ocasionais (leap seconds) → usado por satélites, GPS, etc
- Relógios de Hardware de Computadores → baseados em cristais de quartzo
  - são menos precisos que o atômico
  - sujeitos ao clock drift (gradualmente adianta ou atrasa)
  - limite tolerável de desvio:  $|C_i - C_j| < \epsilon$  ∀ máquinas  $i \neq j$

## > Algoritmos de Sincronização Física

### - Algoritmo de Cristal

- 1- cliente solicita o tempo a um servidor UTC confiável
  - 2- cliente calcula o tempo de propagação (TP) de viagem p/ ajustar o relógio.  
momento de recebimento da resposta → momento de envio da solicitação
- $$TP = \frac{(Tr - Te)}{2}$$
- ↳ tempo de propagação
- $$\text{Novo Tempo} = \underbrace{\text{current UTC}}_{\text{current UTC?}} + \text{TP}$$

\* se o cliente estiver adiantado, ele não pode retroceder (viola a ordem de tempo), então a solução é só deixar ele + devagar ate igualar.

um dos +  
antigos e  
estáveis

modelos

cliente-servidor

passivo

### - Network Time Protocol (NTP) ↳ implementa essa ideia

• estrutura hierárquica em camadas (strata)

1- Servidores Stratos 0 → alta precisão (relógios atômicos, etc)

2- Stratus 1 → se comunicam diretamente c/ os Os.

3- Usuários comunicam c/ o 1, o 3 c/ o 2, assim vai.

4- o NTP usa UDP e algoritmos que filtram atraso de rede.

### - Algoritmo de Berkeley

1- o servidor de tempo (daemon) é ativo

2- consulta periodicamente o tempo de todos os outros máquinas,

3- calcula a média dos tempos recebidos e

4- informa cada máquina o deslocamento necessário p/ ajuste.

também ajusta  
seu próprio relógio  
com cálculo

## - Reference Broadcast Synchronization (RBS)

- projetado p/ redes wireless e sensores (p/economia de energia)
  - assume que ninguém tem o tempo absoluto. Foca na sincronização absoluta entre os nós.
- 1- um nó transmite uma mensagem de referência (beacon) p/ seus vizinhos
  - 2- cada nó receptor registra o horário que recebeu a mensagem.
  - 3- os nós trocam esses horários entre si e o deslocamento de tempo entre 2 nós  $p$  e  $q$  é calculado como média das diferenças entre seus tempos de receção p/ vários msgs beacon
  - 4- assim, cada nó mantém uma tabela de offsets em relação aos outros.

## > Sincronização Lógica e Ordenações de Eventos

↳ conhecer a ordem causal dos eventos é + importante

### - Relações Happens-Before ( $\rightarrow$ ) $\Rightarrow$ ordem parcial que captura a ordem dos eventos.

- se  $A \in B$  ocorrem no mesmo processo e  $A$  é executada antes de  $B$ , então  $A \rightarrow B$ .
- se  $A$  envia de uma mensagem e  $B$  é a receção,  $A \rightarrow B$ .
- se  $A \rightarrow B$  e  $B \rightarrow C$ ,  $A \rightarrow C$  (transitiva)
- eventos não relacionados (nem  $A \rightarrow B$  ou  $B \rightarrow A$ ) são ditos concorrentes ( $A || B$ ), não causalmente independentes (mas há como definir qual ocorreu 1º).

### - Relógios Lógicos de Lamport $\Rightarrow$ cada processo $P_i$ mantém um contador interno $C_i$ (o relógio lógico) que é incrementado antes da ocorrência de cada evento local.

- 1- antes de executar um evento,  $P_i$  executa  $C_i := C_i + 1$
- 2- quando  $P_i$  envia mensagem  $m$ , ele anexa a essa mensagem o timestamp  $t = C_i$
- 3- ao receber uma mensagem  $(m, t)$ , o processo  $P_j$  calcula  $C_j := \max(C_j, t) + 1$  e entrega a msg. p/ a aplicação.

- Imagine que se  $A \rightarrow B$ , então  $C(A) < C(B)$ , mas se  $C(A) < C(B)$  não podemos dizer que  $A \rightarrow B$ , pode haver outros eventos concorrentes!
- fornece ordem total consistente c/ a causalidade.
- p/ ter a causalidade total, implementar os relógios legítimos + um ID único (IP, por ex) dos processos.

- Relógios Vitoriais  $\Rightarrow$  capturam completamente a relação de causalidade

- relógio vitorial  $V_i$ : p/ um sistema c/ N processos = vetor de N inteiros, onde:

- $V_i[i] = n$  de eventos que ocorreram em  $P_i$
- $V_i[j]$  ( $\forall j \neq i$ ) = conhecimento que  $P_i$  tem sobre o número de eventos ocorridos em  $P_j$ .

- regras de atualização:

- 1- inicialmente  $\rightarrow V_i[j] = 0 \quad \forall i, j$
- 2- antes de um evento em  $P_i \rightarrow V_i[i] = V_i[i] + 1$
- 3-  $P_i$  inclui  $t = V_i$  em cada mensagem que envia
- 4- ao receber uma msg c/ timestamp  $t$ ,  $P_i$  faz  $V_i[j] := \max(V_i[j], t[j]) \quad \forall j$ , e então aplica a regra de incremento p/ o evento de receção.

- relações causais!  $\rightarrow$  e é causante

- se  $V(e) < V(e')$ , então  $e \rightarrow e'$  anterior a  $e'$
- se  $V(e) \parallel V(e')$ , então não vetores concorrentes e incomparáveis

- desvantagem = OVERHEAD alto (nº de processos, tam vetor?).

## Exclusão Mútua Distribuída

↳ feita por troca de mensagens nos SDs.

- 2 requisitos: regularidade e vivacidade

no máx 1 processo executando na região crítica

- algoritmo centralizado  $\Rightarrow$  processos solicitam acesso a um processo gerenciador e coordenador p/ entrar na região crítica, rodar e finalizar finalizações.

- só precisar de 3 mensagens (solicitação, concessão e liberação)
- ponto de falha: se o coordenador falhar, o sistema todo é comprometido.

### - algoritmos distribuídos - Ricart e Agrawala

- consentimento mútuo entre todos os processos.
- $P_i$  que quer entrar na SC → envia solicitação c/ timestamp p/ todos.
- os outros só concedem se não tiverem interesse em entrar na SC ou se sua própria solicitação tiver um timestamp + recente
- garante justiça e corretaçāo, mas overhead de  $2(n-1)$  msgs por ação.

### - melhoria → algoritmo de Carvalho e Reucanier

- autorizações implícitas. Uma autorização concedida a  $P_i$  permanece válida até que uma nova dê o mesmo  $P_i$  seja recebida.
- assim, um processo que acabou de entrar na SC pode voltar a ele sem ter que pedir direto (de que outro processo tenha pedido). →  $O = 2(n-1)$  mensagens

### - algoritmo de Token Ring

- 1- organiza os  $P_i$ s em um anel lógico.
- 2- um token circula continuamente pelo anel, que concede ao processo o direito de posse indefinida.
- 3- se um  $P_i$  não quer acionar, repassa o token ao vizinho.
- peca no tratamento de falhas → perde o token se falha de  $P_i$ .

## \* Algoritmos de Eleição

(gerenciamento de recursos  
 coordenação de atividades  
 tomada de decisão centralizada)

- é necessário eleger um processo p/ atuar como coordenador
- processos possuem ID único e viram eleger o de maior ID (em cascata)

simples, mais + overhead

- Bully Algorithm → quando um  $P_i$  detecta que o coordenador atual não está respondendo, ele envia "ELIGÍCIAO" p/ todos os  $P_i$ s c/ ID > que o dele. Se ninguém responder, ele se auto-eleciona "COORDENADOR". O processo c/ ID + alta que estiver ativo é o vencedor e broadcasta "COORDENADOR".

+ eficiente

→ menores em redes maiores

- Algoritmo de Anei → assume que há uma topologia de anel lógico, onde cada processo conhece seu sucessor.

1- quando um Pi detecta que o coordenador atual não está respondendo, ele envia "ELIGÍCTO" p/ o vizinho com ID < os demais respondam com seus IDs.

2- quando completa a volta, ele analisa o maior ID de todos e nomeia COORDENADOR.

• depende da integridade do anel lógico.

adaptado  
p/ topologias  
específicas.

- Yo-Yo Algorithm → p/ ambientes específicos como redes de sensores sem fio (WSNs), redes ad hoc, etc.

↳ leva em consideração:

- proximidade do sink (ponto de coleta de dados)
- nível de energia residual das baterias
- capacidade computacional / poder de processamento.

↳ o algoritmo busca por mínimo ou máximo onde nós são classificados como sources (mínimos locais), sinks (máx. locais) ou internos.

2- sources propagam seus IDs

3- sinks respondem, permitindo que apenas o nó com menor ID (ou outro critério) seja eleito.

## ★ Deadlocks em SDs

o custo de detec-

cção e prevenção

é maior que o

de se recuperar

de um deadlock

- Algoritmo de Averittuz → ignora o problema

- Detecção → um coordenador coleta periodicamente grafos de alocação de recursos de todos os máquinas e os consolida em um grafo global p/ buscar ciclos.

◦ falsos deadlocks podem atrapalhar

◦ Chandy - Mirra - Itcas → quando um Pi fica bloqueado, inicia uma onda (mensagem especial) enviada pelo grafo de

espera. Se ela retornar ao processo que a originou, um deadlock é detectado.

- Prevenção → tornar deadlocks impossíveis

2 técnicas

- { 1) exigir que um processo solicite todos os recursos que precisa de uma só vez, liberando depois.  
2) estabelecer uma ordenação global total de todos os tipos de recursos e exigir que os PIs solicitem os recursos nessa ordem.

- Evitação → antes de o sistema alocar um recurso, verifica se aquela alocação potencialmente poderia levar o sistema a um deadlock. Se sim, é evitada.

- é complexo (difícil que achar um estado global certo em tempo real).

## \* Web Services e SOA → (Arquitetura Orientada a Serviços)

> SOA: "representar funcionalidades do software como serviços".

→ modelo arquitetural p/ construções de softwares que encapsulam funcionalidades de negócio em unidades modulares, autônomas e reutilizáveis → serviços.

→ pacotes acoplados,  
- interoperáveis e orientados.

Os principais conceitos da SOA incluem:

- **Serviços**: Unidades atômicas que fornecem funcionalidades de negócio.
- **Interfaces auto-descritivas**: A interface, separada da implementação, descreve as operações do serviço de forma independente de plataforma.
- **Troca de mensagens**: A comunicação entre serviços ocorre através da troca de mensagens padronizadas, garantindo interoperabilidade.
- **Baixo acoplamento**: A dependência entre serviços é minimizada através do uso de interfaces e protocolos padronizados.
- **Registro de serviços**: Um diretório (como UDDI) atua como uma "lista telefônica" onde os serviços são publicados e podem ser descobertos.
- **Composição de serviços**: Serviços individuais podem ser orquestrados para criar processos de negócio mais complexos.
- **Qualidade de Serviço (QoS)**: Atributos como confiabilidade, segurança e desempenho são preocupações fundamentais.

dependência  
(minima)

## > SOC - Service Oriented Computing

- ↳ paradigma de computação distribuída que usa serviços como blocos fundamentais p/ aplicações.
- stateful (orientado a objetos que têm estado) → stateless (orientado a serviços, em que cada requisição tem todos os info.)

## > Web Services → tecnologia de chamadas remota.

- ↳ é a realização tecnológica + comum de um serviço em SOA.
- alto grau de abstração em relação a linguagens de programação e plataformas.
- sua função é comunicação entre sistemas (B2B) → não fornece interface gráfica

### ★ Pilares fundamentais do "Big Web Services":

1. **SOAP (Simple Object Access Protocol)**: Protocolo baseado em XML para troca de informações estruturadas em um ambiente distribuído. Define um formato de mensagem encapsulado em um Envelope, que pode conter um Header opcional (para informações de controle como autenticação e roteamento) e um Body obrigatório (com o conteúdo da mensagem ou payload). O SOAP é independente do protocolo de transporte, mas é mais comumente usado sobre HTTP por sua capacidade de atravessar firewalls facilmente.
2. **WSDL (Web Services Description Language)**: Linguagem XML para descrever a interface de um serviço web de forma precisa. Um documento WSDL é composto por uma parte abstrata e uma parte concreta:
  - **Parte Abstrata:** Define o que o serviço faz.
    - <types>: Define os tipos de dados usando XML Schema.
    - <message>: Define os parâmetros de entrada e saída das operações.
    - <portType> (ou <interface> em versões mais recentes): Agrupa operações relacionadas.
  - **Parte Concreta:** Define como e onde acessar o serviço.
    - <binding>: Especifica os protocolos de comunicação (ex: SOAP) e formatos de dados para um determinado portType.
    - <service>: Agrupa endpoints (portas) onde o serviço está disponível.
    - <port>: Especifica um endereço (URL) para um binding. = rede
3. **UDDI (Universal Description, Discovery, and Integration)**: Serviço de diretório usado como registro para publicar e descobrir serviços web. Funciona como uma lista telefônica onde os provedores publicam seus serviços e os clientes os buscam. Na prática, seu uso tornou-se menos comum, pois frequentemente o WSDL é obtido diretamente pelo consumidor do serviço por meio de outros canais.

↳ (Framework independente de plataformas usado na comunicação entre provedores de serviços e consumidores)

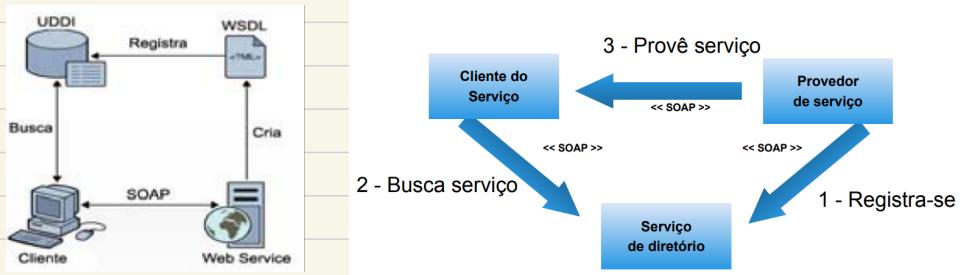
comunicação  
de busca

transporte

↳ de descreve como criar registro p/ armazenar as info. sobre Serviços Web (como se fosse uma lista telefônica).

### - arquitetura de operação:

- 1- provedor publica o serviço no Registro UDDI
- 2- cliente consulta o registro p/ descobrir o serviço, obtém sua descrição WSDL e a partir dela gera um código proxy/stub p/ se comunicar diretamente c/ o Provedor usando msg SOAP.



### - padrões de troca de mensagens definidos no WSDL:

- In-Only: Mensagem de entrada sem resposta.
- Robust In-Only: Mensagem de entrada com confirmação de entrega.
- Out-Only / Robust Out-Only: Mensagem de saída (iniciada pelo servidor).  
Estes padrões permitem modelar interações mais complexas e assíncronas.

### - ferramentas de desenvolvimento que automatizam o processo:

inverso ↗ gerar WSDL por interface Java usando anotações (@WebService)  
↗ gerar código cliente (stubs ou proxies) p/ WSDL com wsimport ou wsdl .exe

### - estilo arquitetural REST (Representational State Transfer)

- Recursos com URLs: Todo recurso (dado ou serviço) é identificado por uma URI.
- Interface uniforme: Usa os métodos HTTP padrão (GET, POST, PUT, DELETE) para operar sobre os recursos.
- Stateless: Cada requisição deve conter todo o contexto necessário, sem estado de sessão no servidor.
- Representações: Os recursos podem ser representados em diferentes formatos, como JSON (mais comum), XML ou HTML.

Web Services baseados em SOAP/WSDL/UDDI = implementação  
mediante de uma SOA  $\Rightarrow$  interoperabilidade  
- contratos rígidos  
- operações complexas

## SOAP vs. REST

### • SOAP:

- Baseado em XML
- Alta complexidade (WSDL, UDDI)
- Usado em bancos e governo
- Suporta uma variedade de protocolos (transporte, autenticação, criptografia, etc)

### • REST:

- JSON ou XML
- Simples, direto sobre HTTP (métodos GET, PUT, POST, DELETE)
- Usado em web, mobile, APIs públicas

Muitos sites suportam ambos os padrões

oferece uma alternativa + flexível + leve.

\* hosts, impressoras,  
máquinas, portas, etc

## ★ Serviços de Nomes em SDs:

→ atua como BD distribuído que gerencia o mapeamento entre  
nomes legíveis e os atributos dos recursos \* que representam.  
↳ nomes textuais (strings) ← binding → identificadores de sistema (bits)

## > Abordagens p/ Resolução de Nomes

- nomeações rígidas  $\Rightarrow$  broadcastting
  - mensagem c/ ID do destinatário é enviada. Só o destinatário com esse ID responde
  - ARP (IP  $\rightarrow$  MAC)
  - falta de escrivibilidade (insuficiente)
- nomeações rígidas  $\Rightarrow$  localização native (home agent)
  - um agente fixo e conhecido = agente nativo.
  - toda comunicação é inicialmente direcionada a ele,

responsável por receber o loc. do recurso móvel e encontrar os requisitos p/ seu endereço atual.

• é + escalável.

- nomeação estruturada → DNS
  - Domain Name System

→ em outras palavras  
muito pq já estudamos  
isso em Redes, não ta  
afim.

## > O DNS:

- estrutura e espaço de nomes:

O DNS organiza seu espaço de nomes em uma **estrutura hierárquica em forma de árvore**, permitindo a descentralização da administração.

- **Domínio Raiz (Root-level domain)**: Representado por um ponto final (.). É gerenciado por 13 grupos de servidores raiz (de a.root-servers.net a m.root-servers.net).
- **Domínios de Primeiro Nível (Top-level domains - TLDs)**: São os domínios imediatamente abaixo da raiz. Incluem:
  - **gTLDs (generic TLDs)**: .com, .org, .net, .edu, .gov, etc.
  - **ccTLDs (country-code TLDs)**: .br, .uk, .fr, etc., representando países ou regiões.
- **Domínios de Segundo Nível (Second-level domains)**: São domínios registrados por organizações ou indivíduos sob um TLD (p.ex., pucminas.br, google.com). Esses domínios podem, por sua vez, ser divididos em **subdomínios** (p.ex., sd.pucminas.br).

- funcionamento e tipos de consulta:

O princípio de funcionamento do DNS baseia-se no conhecimento parcial. Cada servidor de nomes é **autoritativo** para uma ou mais zonas (partes contíguas do espaço de nomes) e conhece os endereços dos servidores responsáveis pelas zonas filhas (delegadas) e pelos servidores raiz.

- **Consulta Recursiva**: O cliente (ou resolver) envia a consulta a um servidor e espera que este retorne a resposta final (o endereço IP) ou uma mensagem de erro. O servidor assume a tarefa de consultar outros servidores em nome do cliente até encontrar a resposta.
- **Consulta Iterativa**: O servidor, se não tiver a resposta, retorna ao cliente a "melhor resposta possível", que geralmente é o endereço de um servidor mais próximo da zona autoritativa do nome desejado. Cabe então ao cliente consultar esse novo servidor. Os servidores raiz e TLD normalmente só realizam consultas iterativas.

Para melhorar a eficiência e reduzir o tráfego, os servidores DNS utilizam **cache** extensivamente. As respostas obtidas de cache são marcadas como "**não autoritativas**" (*non-authoritative*). Cada entrada no cache possui um **TTL (Time-to-Live)**, determinado pela zona autoritativa, que define por quanto tempo a entrada pode ser reaproveitada antes de precisar ser atualizada.

## - tipos de servidores e registros:

- **Servidor Primário (Master):** Mantém os arquivos de zona originais e definitivos para um domínio. As alterações são feitas neste servidor.
- **Servidor Secundário (Slave):** Obtém uma cópia dos arquivos de zona de um servidor primário (ou de outro secundário) através de uma **transferência de zona** (*zone transfer*). Fornece redundância, balanceamento de carga e reduz a latência para clientes em locais remotos.
- **Servidor de Cache (Resolver ou Recursive Resolver):** Não é autoritativo para nenhum domínio. Sua função é realizar consultas recursivas em nome dos clientes e armazenar as respostas em cache. ISPs e empresas geralmente operam esses servidores para seus usuários.

O banco de dados DNS é composto por vários tipos de **registros de recursos (RR)**. Os principais são:

- **SOA (Start of Authority):** Marca o início de uma zona e contém parâmetros administrativos, como o e-mail do administrador e tempos de atualização.
- **NS (Name Server):** Identifica um servidor de nomes autoritativo para a zona.
- **A (Address):** Mapeia um nome de host para um endereço IPv4.
- **AAAA:** Mapeia um nome de host para um endereço IPv6.
- **PTR (Pointer):** Usado para *resolução reversa*, mapeando um endereço IP para um nome de host.
- **MX (Mail Exchanger):** Especifica os servidores de e-mail (mail exchangers) para um domínio, definindo a prioridade de cada um.
- **CNAME (Canonical Name):** Define um apelido (alias) para um nome de host canônico.

## **\* funcionalidades avançadas do DNS:**

Conforme detalhado no livro de Coulouris, o DNS vai além do simples mapeamento nome-IP:

- **Resolução Reversa:** Mapeamento de endereços IP para nomes, crucial para ferramentas de diagnóstico como traceroute e para sistemas de autenticação e logging.
- **Balanceamento de Carga:** Atribuindo múltiplos registros A para o mesmo nome, o DNS pode distribuir requisições entre vários servidores.
- **Redirecionamento de Serviços:** Usando registros CNAME e MX, é possível redirecionar tráfego para diferentes serviços ou provedores de forma transparente para o usuário final.

## > WINS (Windows Internet Name Service)

↳ serviço de nomes projetado p/ resolver nomes NetBIOS em rede IP.

NetBIOS = interface de programação de APIs + antiga, da época de servos.

### WINS vs DNS

nomespace	{ tem nomes planos (flat), sem estrutura de domínios e subdomínios <i>(limite escalabilidade)</i>	→ estrutura do namespace é hierárquica
atualização	{ atualizações do BD automática e dinâmica ( <i>Name registration na iniciização, renewal e depois release</i> )	→ atualizações majoritariamente estáticas e manuais <i>(DDNS mudou isso)</i>
arquitetura	{ concebido como + centralizado no dicionário	→ inherentemente distribuído e descentralizado por design.

Nível 1: Pilar Fundamental	Nível 2: Conceito ou Desafio	Nível 3: Abordagem ou Camada de Abstração	Nível 4: Ferramentas, Protocolos e Algoritmos Específicos
Fundamentos e Desafios	Gerenciar a Complexidade e Heterogeneidade	Middleware (Camada de software que provê uma visão única e transparente)	<ul style="list-style-type: none"> <li>- Objetos Distribuídos: Java RMI, CORBA</li> <li>- Serviços: Web Services (SOAP, REST)</li> </ul> <p style="color: red; margin-left: 20px;">gerencia a complexidade da comunicação entre ambientes heterogêneos (SO → Middleware → APP)</p>
	Transparência	<ul style="list-style-type: none"> <li>- De Acesso: Fazer acesso remoto parecer local.</li> <li>- De Localização: Encontrar recursos sem saber onde estão.</li> </ul>	<ul style="list-style-type: none"> <li>- Proxies (Stubs/Skeletons)</li> <li>- Serviços de Nomes</li> </ul>
	Escalabilidade	<ul style="list-style-type: none"> <li>- Replicação: Copiar dados/serviços.</li> <li>- Caching: Armazenar dados perto de quem usa.</li> <li>- Distribuição: Evitar componentes e algoritmos centralizados.</li> </ul>	<ul style="list-style-type: none"> <li>- Servidores DNS secundários</li> <li>- Servidores Múltiplos</li> </ul>
	Tolerância a Falhas	<ul style="list-style-type: none"> <li>- Detecção: Saber que um componente falhou.</li> <li>- Recuperação: Lidar com a falha.</li> </ul>	<ul style="list-style-type: none"> <li>- Algoritmos de Eleição para substituir um coordenador.</li> <li>- Redundância de hardware e software.</li> </ul>
Comunicação entre Processos	Paradigma de Comunicação	Troca de Mensagens (Base para toda comunicação)	<ul style="list-style-type: none"> <li>- Primitivas: send , receive</li> <li>- Semântica: Síncrona (bloqueante) vs. Assíncrona (não-bloqueante)</li> </ul> <p style="color: red; margin-left: 20px;">proxy (representante local do objeto remoto)</p>
		Chamada de Procedimento Remoto (RPC)	<ul style="list-style-type: none"> <li>- Conceitos: Stubs (cliente) e Skeletons (servidor), Marshalling/Unmarshalling.</li> <li>- Implementação: Sun RPC, XDR (linguagem de especificação).</li> </ul>
		Invocação de Método Remoto (RMI)	<ul style="list-style-type: none"> <li>- Foco em Java: java.rmi , rmiregistry .</li> </ul> <p style="color: red; margin-left: 20px;">geram os stubs e skeletons</p>
		Objetos Distribuídos Interoperáveis	<ul style="list-style-type: none"> <li>- Padrão: CORBA (Common Object Request Broker Architecture)</li> <li>- Componentes: ORB, IDL (Interface Definition Language), IIOP (protocolo).</li> </ul>
Capacidade de sistemas de fornecedores se comunicarem		Arquitetura Orientada a Serviços (SOA)	<ul style="list-style-type: none"> <li>- Web Services (baseado em XML): SOAP (protocolo de mensagem), WSDL (descrição do serviço), UDDI (diretório de serviços).</li> <li>- Web Services (baseado em HTTP): REST (estilo arquitetural), JSON/XML, Verbos HTTP (GET, POST, PUT, DELETE).</li> </ul>
	Nomeação e Localização de Recursos	Serviços de Nomes	<ul style="list-style-type: none"> <li>- Hierárquico e Distribuído: DNS (Domain Name System)</li> <li>- Flat e Centralizado: WINS (Windows Internet Name Service) para NetBIOS.</li> <li>- Outros: Broadcasting, ARP.</li> </ul>
	Ordem de Eventos	Sincronização de Relógios Físicos (Manter a "hora certa")	<ul style="list-style-type: none"> <li>- Baseado em Servidor: Algoritmo de Cristian (NTP).</li> <li>- Descentralizado: Algoritmo de Berkeley.</li> </ul>
		Ordenação Lógica de Eventos (Manter a relação "aconteceu antes")	<ul style="list-style-type: none"> <li>- Ordem Parcial: Relógios Lógicos (Timestamps de Lamport).</li> <li>- Ordem Causal: Relógios Vetoriais.</li> </ul>
Concorrência e Acesso a Recursos Críticos		Exclusão Mútua Distribuída	<ul style="list-style-type: none"> <li>- Centralizado: Um coordenador gerencia o acesso.</li> <li>- Distribuído (baseado em permissão): Algoritmo de Ricart &amp; Agrawala.</li> <li>- Distribuído (baseado em token): Algoritmo Token Ring.</li> </ul>
	Concorrência e Acesso a Recursos Críticos		

	Gerenciamento de Deadlocks	Detecção de Deadlock	- Centralizada: Um coordenador analisa o grafo global de espera. - Distribuída: Algoritmo de Chandy-Misra-Haas.
		Prevenção de Deadlock	- Ordenação Global de Recursos - Algoritmo Wait-Die
	Liderança e Coordenação	Algoritmos de Eleição (Escolher um processo para ser o coordenador)	- Baseado em ID: Algoritmo do Valentão (Bully). - Baseado em Topologia: Algoritmo do Anel. - Para Redes sem Fio: Algoritmo Yo-Yo.
Segurança	Autenticação e Privacidade	Mecanismos de Autenticação	- Estilo UNIX (UID/GID) em RPC. - Criptografia com chave compartilhada (DES) em RPC. - Cookies e Monitoramento (menionado como desafio).

