Algoritmos Distribuídos

Markus Endler
Sala RDC 503
endler@inf.puc-rio.br
www.inf.puc-rio.br/~endler/courses/DA

Roteiro

1. Introdução
2. Causalidade e noções de tempo
3. Exclusão Mútua Distribuída
4. Eleição de Coordenador
5. Detecção de Predicados Globais e Snapshots
6. Problemas de Acordo Distribuído (Distributed Agreement)
7. Detectores de Falha
8. Serviços de Grupo (Comunicação confiável e Pertinência)

Avaliação e Datas

Avaliação:

- Dois **trabalhos práticos** T1, T2 (com relatórios) – simulação de algoritmos
- **Monografia** sobre um problema de coordenação distribuída e comparação de 2-3 algoritmos
- **Apresentação** (40 mins) sobre a monografia: nas últimas duas aulas (20 e 27 junho)
- Prova PF (*apenas se houver necessidade),* Data : 5/julho

• Cáclulo do Grau final:

M = (T1 + T2 + APRES+MONO) /4 se > 7.0 passou. Senão, precisa fazer a PF

M = (PF + M)/2

Pré-requisitos:

- experiência com Java
- **Desejável:** ter noções básicas de Redes de Computadores e Sist. Operacionais

Dinânica de aula:
Em sala de aula: aulas expositivas + discussão sobre o material estudado
Em casa: leitura de textos (capitulos de livro ou artigos), ou um video, implementação e

Obs: Na web-page da disciplina há os slides, enunciados dos trabalhos, notas, avisos, etc.

Monografia

Algoritmos distribuídos utilizados em certa área de aplicação (exemplo, jogos, segurança da informação, processamento em núvem, etc.)

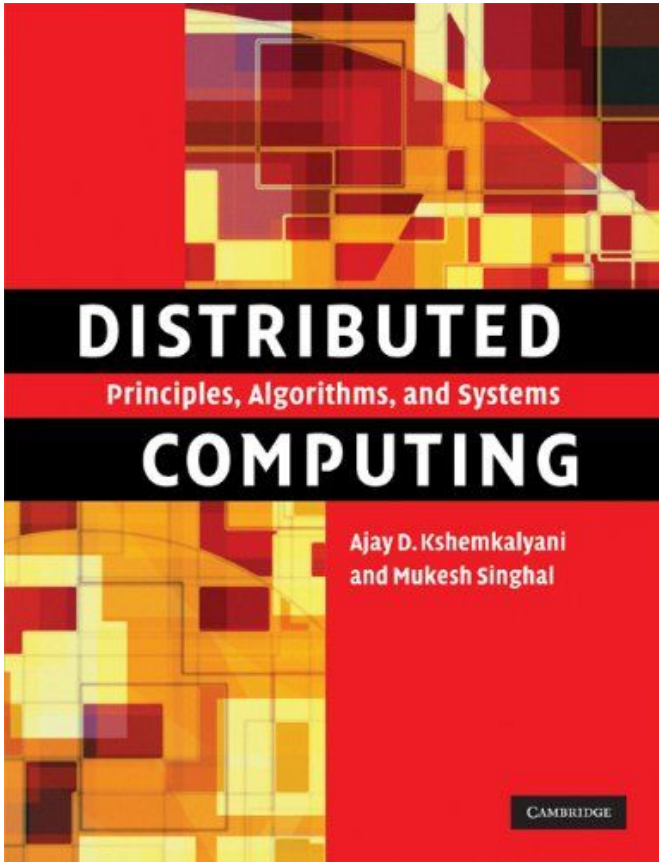
- Apresentar a área de aplicação e
- Descrever um ou mais problemas intrinsticamente distribuídos, que demandam algoritmos distribuidos
- Estudar trabalhos na literatura que descrevam os algoritmos usados
- Fazer uma discussão comparativa sobre os algoritmos e seus modelos de sistema associados
- Elaborar conclusões sobre o assunto

- Um documento de umas 8-15 páginas

Livro texto

Ajay D. Kshemkalyani, Mukesh Singhal, *Distributed ComputingPrinciples, Algorithms, and Systems*, Cambridge University Pres, 2008 – (PDF junto com as transparências)

(Capítulos 1-4, 5.6, 6,9, 11,14, 15)



Bibliografia Complementar

R.Chow & T.Johnson, *Distributed Operating Systems and Algorithms*, Addison Wesley, 1997 (Capítulos 9-12)

V. Garg, *Elements of Distributed Computing*, John Wiley & Sons, 2002. (Capítulos 2-3, 6-7, 9-12,16-17, 21, 23, 26-28)

G.Coulouris & J.Dollimore, T.Kindberg, *Distributed Systems: Concepts and Design,* Addison Wesley, 1997 (Capítulos 2,10,11,14)

Juho Hirvonen & Jukka Suomela, Distributed Algorithms 2020, Algoritmos baseados em grafos (https://jukkasuomela.fi/da2020/) -livro, slides, e videos

Alguns artigos sobre problemas e algoritmos específicos.

Apresentação + Monografia

Data de entrega: 21/06 durante a aula
Apresentações: dias 14+28/06

Roteiro sugerido:

1. Descrição da área de aplicação (Objetivos, tarefas comuns, relevância, etc.)
2. Descrever os 2-3 principais problemas de *coordenação distribuída* encontrados na área
3. Descrever os principais algoritmos distribuídos que são (ou poderiam) ser utilizados para resolve-los.
4. Citar e resumir quais ambientes/middlewares/serviços específicos que são utilizados para resolver os problemas citados em 2.
5. Conclusão sobre o "estado da arte"
6. Referências Bibliográficas

Simulando Algoritmos Distribuídos: T1 e T2

Ambientes de programação e simulação a serem usados

- **Sinalgo**, ETH Zürich, (no T1)
- **GrADyS-SIM**, LAC, DI, PUC-Rio (no T2)

Do que consistirão os trabalhos?

- Adaptação de algoritmos vistos em aula para um sistema de nos/agentescom obilidade total ou parcial.

Objetivo:

a) Entender o algoritmo distribuido e ver como adapta-lo a noca realiade mobile

b) Analisar o comportamento do algoritmo implementado em relação a quantiadde de nós, de taxa de mobilidade, e outros parâmetros

c) Comparação da facilidade de programação, depuração e performance dos algoritmos usando cada uma das ferramentas

Ferramentas de Simulação: Sinalgo

- é um simulador para testar e validar algoritmos distribuidos com e sem mobilidade.
- Ao contrário da maioria dos outros simuladores, que simulam as diferentes camadas da pilha de protocolo, Sinalgo foca na verificação dos algoritmos de rede e aplicação, abstrai das camadas subjacentes:
- Ele explicita a visão de troca de mensagem, que capta bem a visão que cada nó da rede tem, e como ele deve reagir a cada evento.
- É open source (licença BSD) em Java
- Permite sumulação sincrona (lock-step) e assíncrona (baseado em escalonamento de eventos)
- Permite customizar o grafo de nós da rede.
- Baixar o projeto de: **https://sinalgo.github.io/**

***Depoimento de Guilherme Oliveira (maio 2022)**:*
Para rodar o Sinalgo no ambiente Ubuntu, fiz alguns experimentos com a versão (antiga) 0.75.3 e a mais recente, 0.88, e com diferentes formas de compilar/executar os arquivos Java: usando Gradle, um wrapper do Gradle, ou com o Eclipse.
No final, consegui compilar e executar o Sinalgo usando a versão 0.75.3 no Eclipse com a versão 1.5 do Java SE. Só não consegui compilar com uma verão mais recente de Java por conta de um bin´ario jdom.jar que estava j´a compilado para Java 1.5.

Ferramentas de Simulação: Sinalgo

Tutorial:

- http://disco.ethz.ch/projects/sinalgo/tutorial/Documentation.html

Cada nó executa um código para:

- enviar uma mensagem a um vizinho específico, ou a todos os seus vizinhos (alcançaveis)
- reagir às mensagens que chegaram,
- Ativar times para escalonar ações no futuro

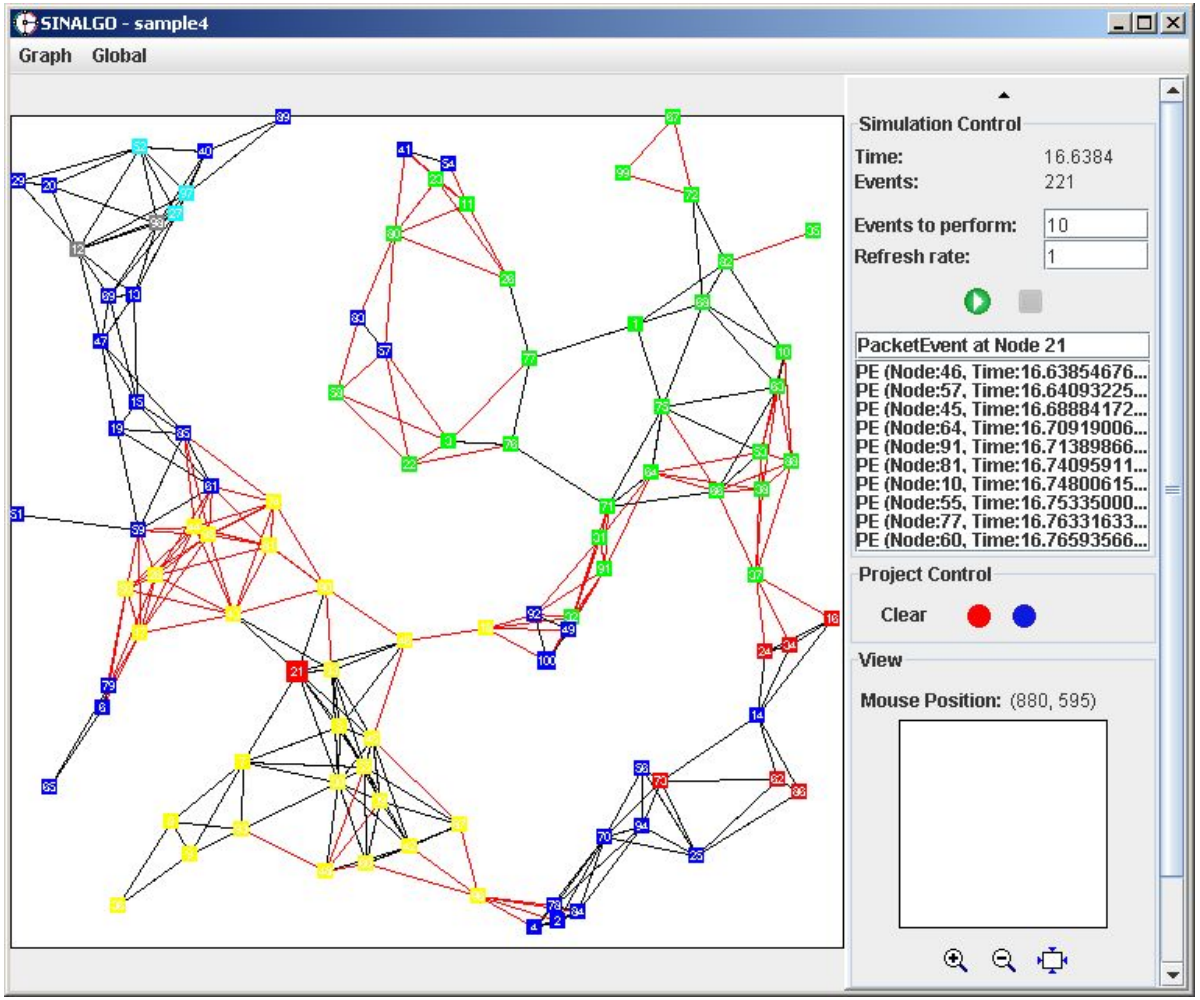
- Não há simulação da passage do tempo real: apenas uma variável discrete (contador)

Sinalgo é extensível e customizável em relação a vários modelos:

- mobility & distribution models: initial location of each node and how it changes its position over time (only in synchronous mode)
- connectivity model: when two nodes are able to communicate
- reliability model: for each message, whether it should arrive or not
- transmission model: how long it takes for a message to arrive

- Não há um modelo de falha separado mas pode ser implementado por voce. Nó com falha crash: o nó simplesmente não reage a eventos

Sinalgo



- Pode ser executado em batch ou no modeo passo a passo
- Execução com animação (as cores indicam os estados do nó e do link)
- Um log de execução pode ser examinado

Ferramentas de Simulação: GrADyS-SIM Nextgen

Simulador baseado em OMNet++ e INET

- Com vários módulos de simulação com responsabilidades específicas, incluindo uma extensão do seu componente de veículos terrestres para incluir altura (no *Modulo de Mobilidade)*
- *Incluindo também o componente de controle*

Referências:

- Thiago Lamenza, Marcelo Paulon, Breno Perricone, Bruno Olivieri, Markus Endler, **GrADyS-SIM -- A OMNET++/INET simulation framework for Internet of Flying things,** ArXiv, https://arxiv.org/abs/2202.08134

Video:

- https://youtu.be/xLcKtGQpmCE
- https://www.youtube.com/watch?v=Im6d5TEes4Y

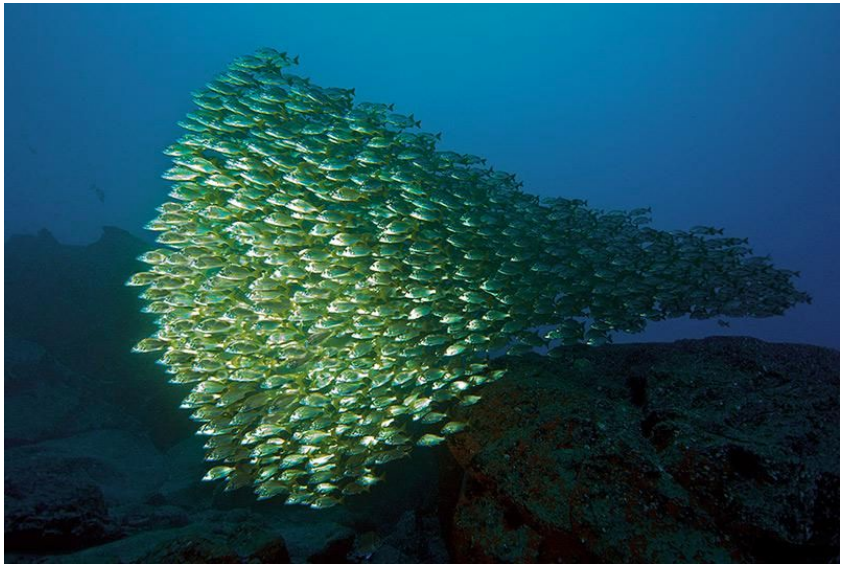
Git-Hub:

- https://project-gradys.github.io/gradys-sim-nextgen/

O Que são Sistemas Distribuídos?

- Uma coleção de **entidades independentes** (nós) que **cooperam entre sí** para resolver um problema que não pode ser resolvido individualmente por cada nó.

Existem muitos exemplos na natureza (coordenação de movimento)



O Que são sistemas Distribuidos?

Outros Exemplos :

- requerem comunicação para delegação, coordenação e manutenção/atualização da força tarefa

O Que são Sistemas Distribuídos?

Nem todos visam acões/movimentos coordenados ou sincronizados….

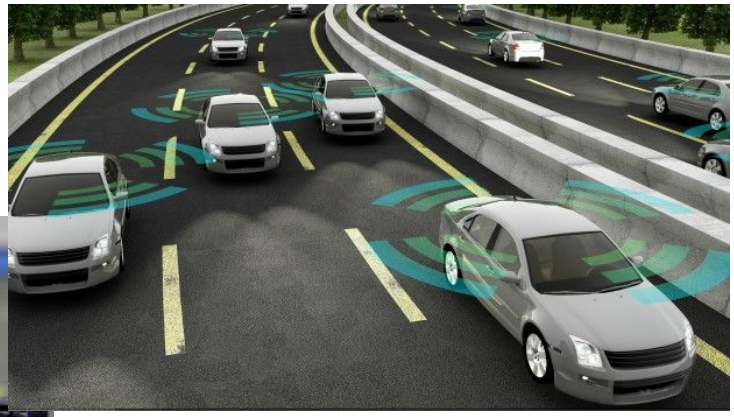
Outros exemplos:

- Ter uma visão coerente dos nós que estão ativos;
- Manter nós replicados com seus estados sincronizados;
- Convergir para um valor idêntico em todos os nós
- Definir um único nó coordenador
- Identificar quando um processamento terminou vs quando se está em um deadlock ou livelock
- Evitar a existência de instabilidades indefinidas
- …

Onde os nós podem ser: dispositivos IoT, smartphones, estações em uma WAN, veiculos, robôs/drones, servidores ou data-centers inteiros.

Tarefa: tente formular propriedades que precisam ser garantidas por um algoritmo distribuído que resolva um problema em uma dessas áreas.

O Que são Sistemas Distribuídos?



Sistema Distribuído :: sempre que houver comunicação entre nós autonômicos (“agentes inteligentes”) para atualização/coordenação de seus estados.

Principais Características gerais:

- distribuição geográfica dos nós;
- sem relógios sincronizados (sem referência a tempo global único);
- sem meio de comunicação compartilhado (por exemplo: memória ou Banco de dados)

O Que são Sistemas Distribuídos?

Algumas Definições:

- “You know you are using one (Distributed System) when the crash of a computer you have never heard of prevents you from doing work.” [L. Lamport].

- “A collection of independent computers that appears to the users of the system as a single coherent computer” [A. Tanenbaum]

- “A collection of computers that **do not share common memory** and **a common physical clock**, that **communicate by message passing** over a communication network, and where **each computer has its own memory** and runs its own operating system.
- Typically the computers are semi-autonomous and are loosely coupled while they cooperate **to address a problem collectively**” [Singhal & Shivaratri, 94].

Algoritmos Distribuidos?

Um algoritmo distribuído implementa um Serviço Distribuído que tem:

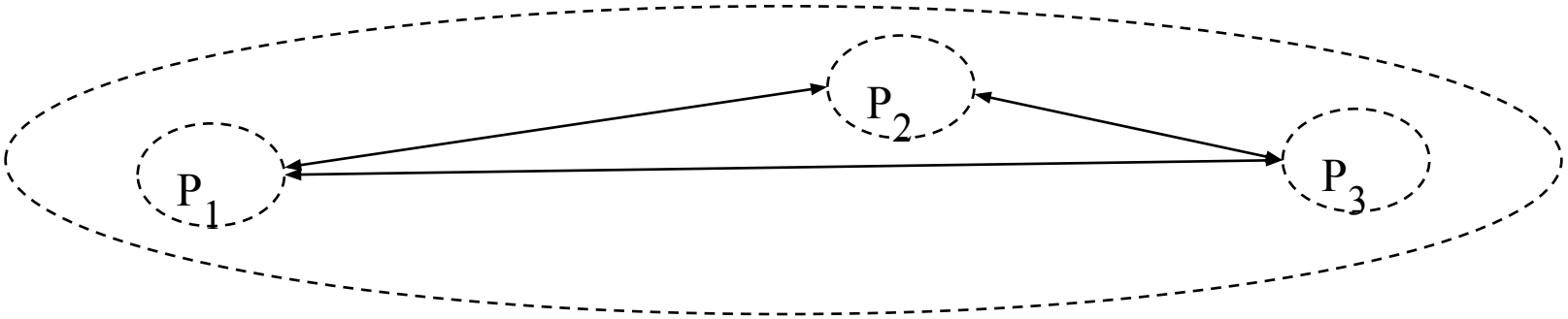
- componentes/agentes que executam em diferentes computadores em rede,
- se comunicam e coordenam ações passando somente mensagens entre sí.
- Não compartilham nenhum recurso (nem tempo) e possuem uma visão parcial do Sistema

Os componentes interagem uns com os outros para **atingir um objetivo comum**.

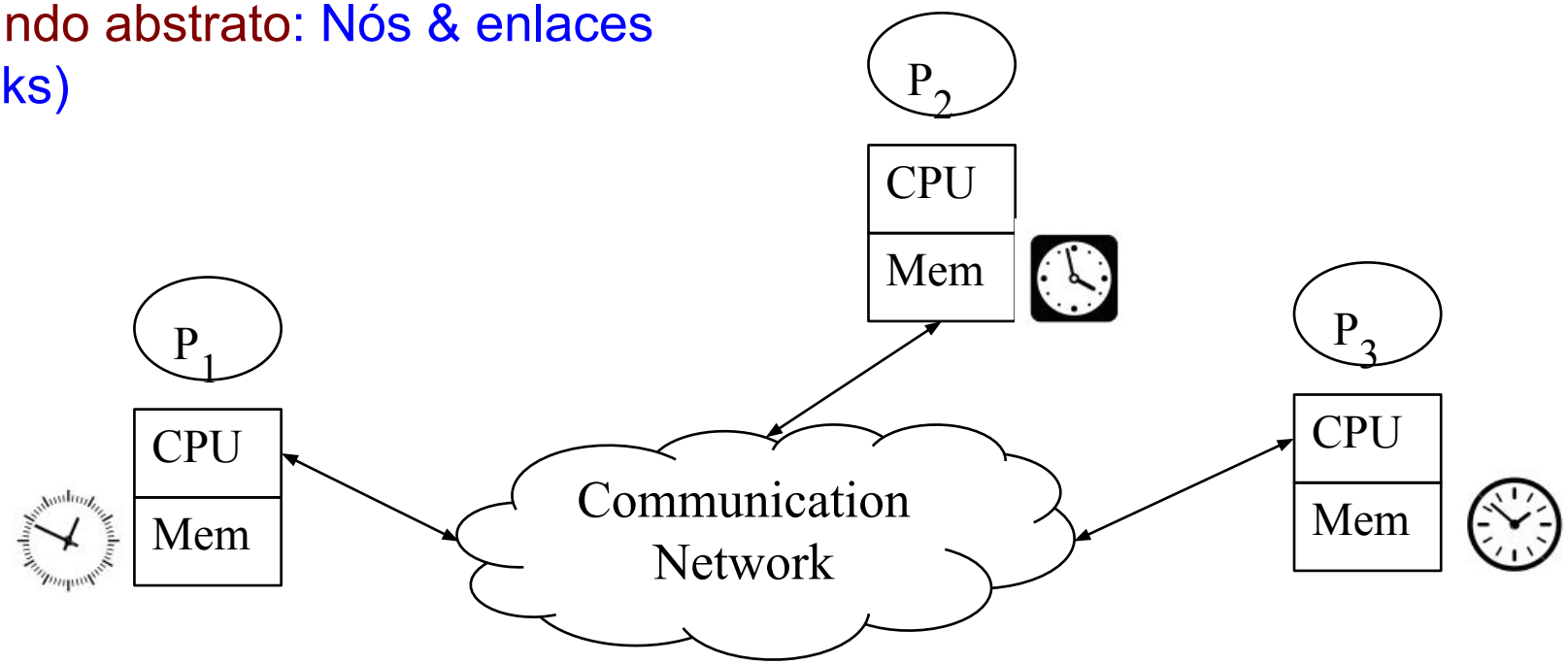
Algumas características de infra-estrutura que são comumente assumidas:

- Concomitância de componentes;
- Não conhecimento da identidade dos componentes interlocutores
- Ausência de um relógio global; e
- Falhas independentes de componentes
- Atraso independente e falha de links de rede.

Visão Geral



Mundo abstrato: Nós & enlaces (links)



Mundo real: Rede física

Verificação Formal e Implementacão de Algoritmos

Problema específico

Teoremas (expressam propriedades desejáveis)

Algoritmo:
- Transições do estado (distribuido), causados por tipos de eventos
- Regras de Transmormação

Lógica, Sist. deducão

Demonstração

Axiomas (hipóteses usdas) referente a:
- Topologia de interconexão
- Conhecimento sobre os PIDs
- Sincronismo vs asincronismo
- Tipos de falhas
- Persistência de estado

Garantias:
Propriedade 1, Propriedade 2
Invariante, etc.

Algoritmo:
- Comportamento de cada nó
- Estados e eventos
- Tipos de Mensagem

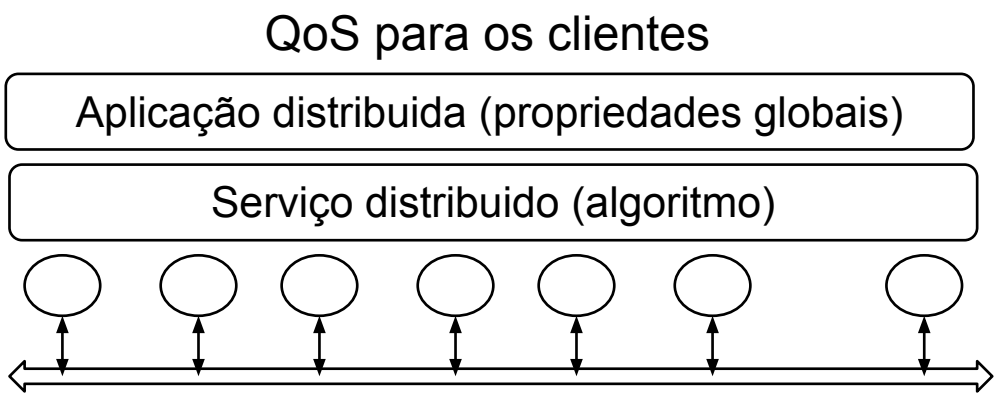
Pseudo-código

Execução/Simulação

Modelo do Sistema:
- Topologia de interconexão
- Conhecimento sobre os PIDs
- Sincronismo vs asincronismo
- Tipos de falhas
- Persistência de estado

Serviço Distribuido usa Alg. Distribuidos

para dar uma visão uniforme so ssistema



Propriedades garantidas

Modelo do Sistema

Exemplos:

- Aplicação distribuida:
  - Processamento tolerante a falha com replicacão de dados (processamento em nuvem)
  - não precisa saber nada sobre quantidade de nós
- QoS para os clientes:
  - Processamento eficiente (Mflops/seg) e confiável (0.001 prob. de perda)
- Serviço Distribuido:
  - Ex: consenso, visão consistend da configuração do sistema
  - não precisa saber nada como encamnhar mensagens para um determinado nó

Características de Sistemas Distribuídos e suas implicações

- Ausência de um relógio global
  - impossibilidade de sincronização perfeita das ações distribuídas
- Ausência de memória compartilhada
  - Cada nó possui memória local, e portanto coordenação só é possível através de envio de mensagem
  - Não é possível conhecer o estado global do sistema
  - Mensagens em trânsito precisam ser consideradas
- Incerteza sobre tempos de comunicação e de processamento
  - Não se conhece o tempo de transmissão de mensagens
  - Os nós podem ter capacidades de processamento e comunicação bem distintos
- Possibilidade de falhas independentes – de nós e da rede
  - Mensagens podem não chegar
  - Pprocessadores podem falhar
  - É impossível diferenciar um processador muito lento de um processador falho.

Por que estudar Algorítmos Distribuídos?

Algorítmos Distribuídos são os programas que executam nos nós de uma rede (com atrasos e confiabilidade não conhecidos)

São compostos por processos que conjuntamente - e de forma coordenada - realizam uma tarefa (um serviço) e que precisam manter uma consistência.

Alguns Serviços Distribuídos uteis:

- sincronização de ações (em exclusão mútua, ou processamento uniforme)
- definir ordem global de eventos independentes (p.ex. Servidores replicados)
- comunicação confiável para todo o grupo de processos
- atingir o consenso entre vários processos
- garantir a integridade e a autenticidade de transações
- captura de estado global (p.ex. detecção de deadlock)

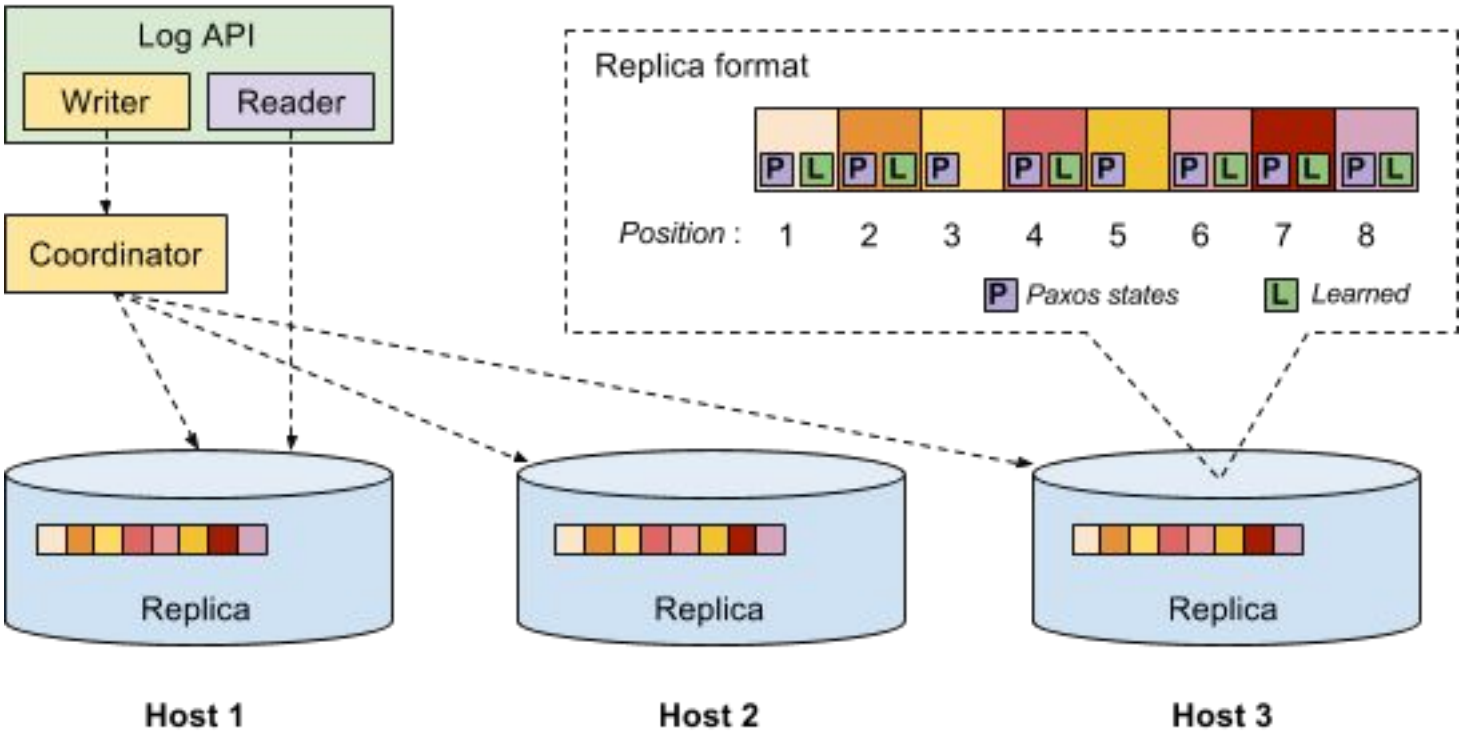
Por que estudar Algorítmos Distribuídos?

Alguns exemplos:

- sistemas operacionais distribuídos e em rede: escalonamento de processos, acesso a memória distribuida compartilhada
- Tolerância à falha usando processos replicados (em um cluster ou em nuvem)
- comunicação de grupo confiável/ atômica: (com entrega garantida para todos os membros e na mesma ordem)
- Massive multiplayer games: visão coerente do estado do jogo em tempo real.
- Ambientes colaborativos (de co-edição) distribuídos
- Consenso em Blockchain

Logs Perfeitamente Replicados

Exemplo: manter vários logs distribuidos perfeitamente sincronizados sem permitir que escrita de log x+1 seja bloqueada por falta de uma confirmação para entrada log x



Premissas e Propriedades Desejáveis

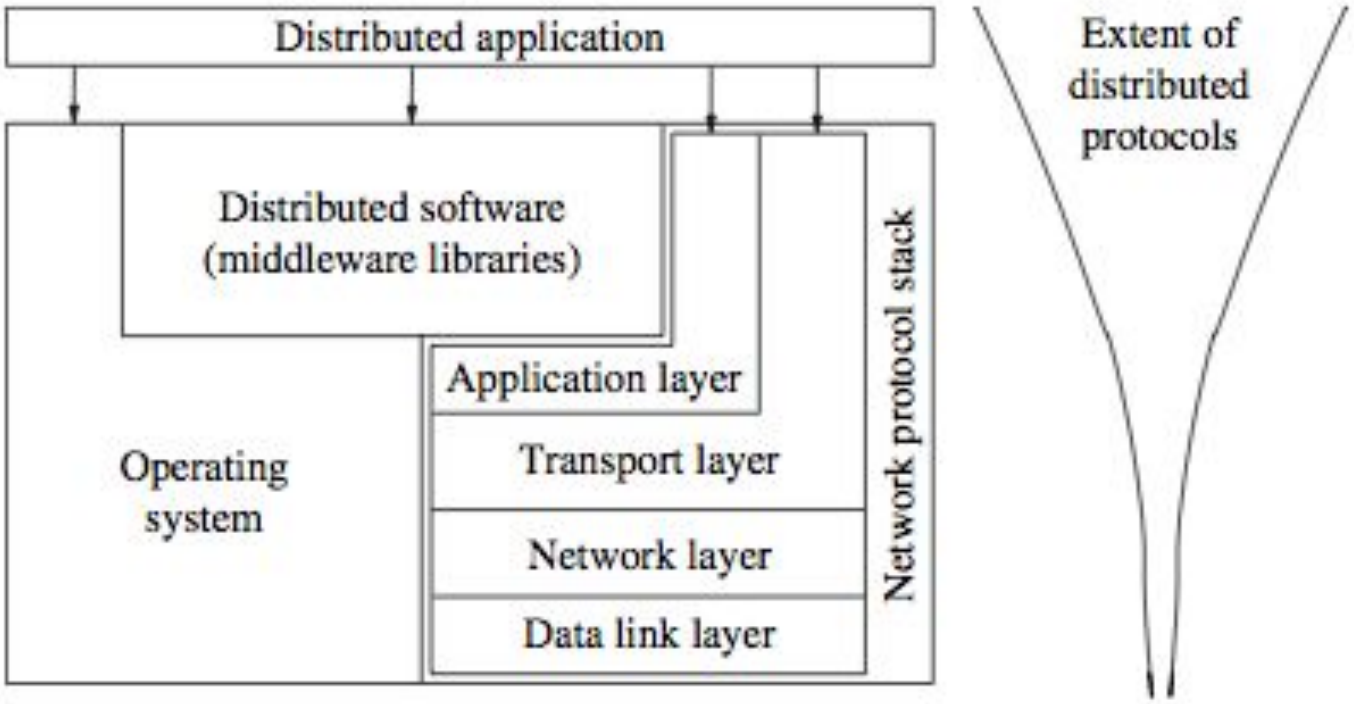
O que esperamos de um algoritmo?

- Estar correto
  - fazer aquilo a que se propõe
  - não violar qualquer condição sobre o estado dos processos
  - que termine
- Ter baixa complexidade: O(n), O(n *log*n), O(n^2), etc.
- Ser efciente na maioria dos casos
- Levar em conta o maior numero possivel de incertezas.

Para algoritmos distribuídos, tudo isso depende das premissas sobre o ambiente de execução (modelo de sistema), por exemplo:

- Processamento síncrono x assíncrono
- Garantias sobre a entrega de mensagens
- Possibilidade de falhas (nós ou rede) e constância/distribuição das falhas

Onde se usa Algoritmos Distribuídos?



- Na camada de aplicação (aplicação distribuída)
- Na camada de middleware
- Serviços para computação em núvem
- Em Sistemas Operacionais Distribuídos

O Modelo básico de um Sistema Distribuído

- O sistema é composto de um conjunto de processos assíncronos

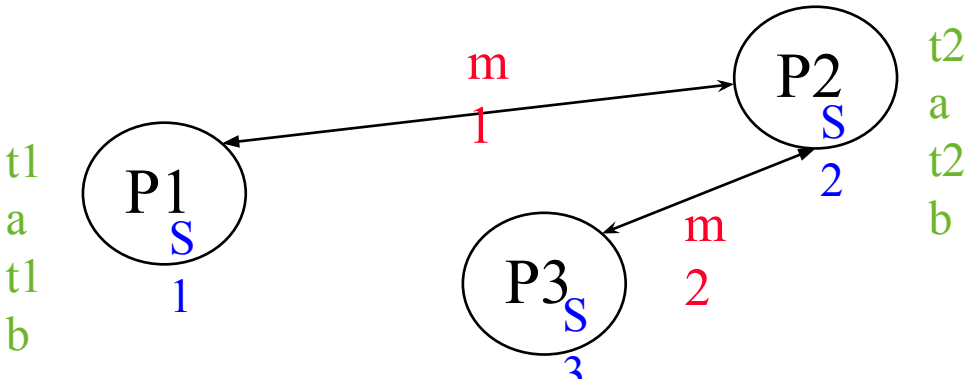
$$p_1, p_2, \ldots, p_i, \ldots, p_n$$

- que se comunicam por mensagem, por intermédio de uma rede de comunicação. (cada processo executa em processador próprio).

- Processos não compartilham memória, mas somente interagem apenas através do envio e recebimento de mensagens. (mensagens unicast)
- Cada processo está um estado local:

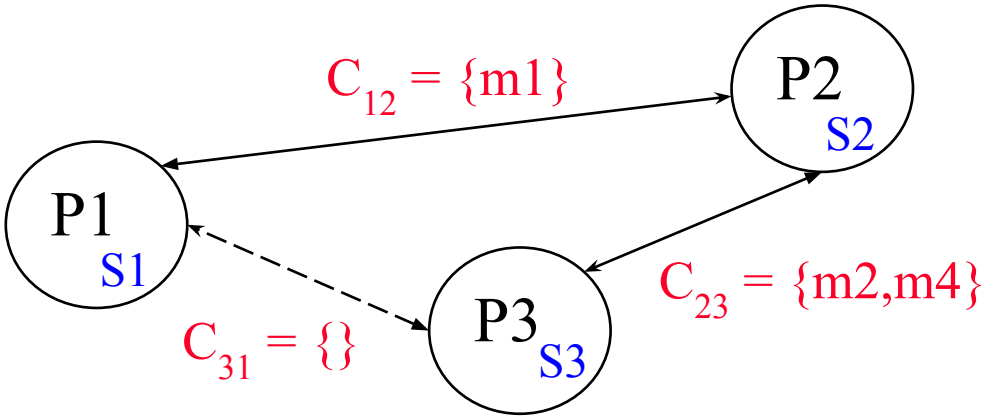
$$S1, S2, \ldots Si, \ldots Sn$$

- A cada evento local em pi, ou recebimento de mensagem por pi, muda o estado local $Si^x \Rightarrow Si^{x+1}$
- Cada processo executa eventos locais (mudanças de estado, eventos de timers), e eventos de envio de mensagens m



O Modelo básico de um Sistema Distribuído

- Seja C_{ij} o canal de comunicação de processo pi para processo p_j, e m_{ij} uma messagem enviada de p_i para p_j .
- O delay da comunicação pelo canal C_{ij} é finito, mas desconhecido.
- Os processos não têm acesso a um relógio global (relógio de parede - global clock), e relógios locais não são sincronizados
- O estado de um canal de comunicação Cij é caracterizado pelo conjunto de mensagens de Pi para Pj em trânsito no canal.

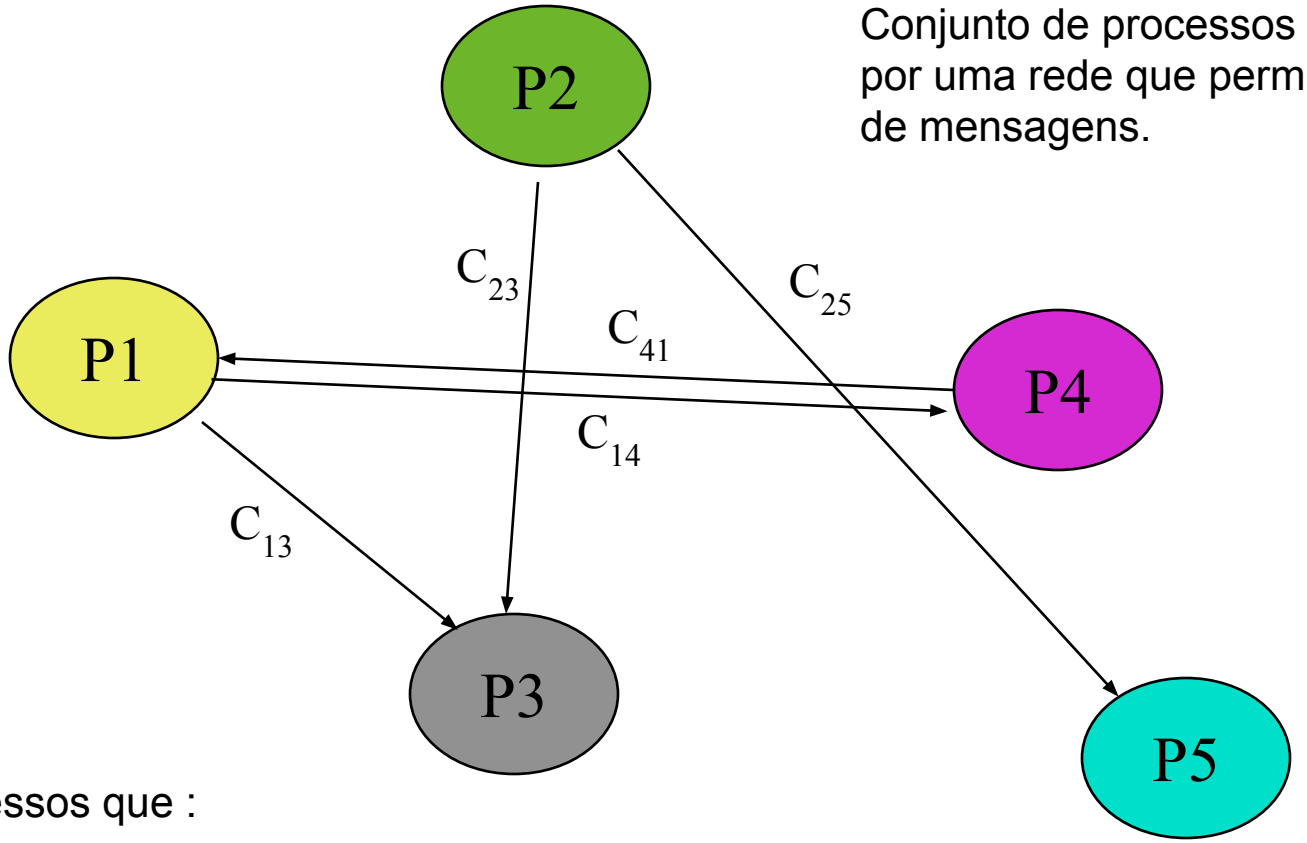


O Modelo básico de um Sistema Distribuído

- A execução assíncrona dos processos e a transferência de mensagens
  - o processo remetente (sender) não espera até que a mensagem seja entregue no destinatário.

- O estado global de uma computação distribuída consiste da união dos estados (locais) dos processos e dos estados dos canais de comunicação:

O Modelo Básico de um Sistema Distribuído



Conjunto de processos conectados por uma rede que permite a troca de mensagens.

Pi são processos que :

- encapsulam um estado (=conjunto de variáveis com valor corrente)
- reagem a eventos externos (mensagens ou temporizadores)
- somente interagem através do envio de mensagens (não há compartilhamento de memória)

Concorrência(s) e Modelos de Execução

- Nossa mente tem uma dificuldade intrínseca de compreender execuções paralelas e simultaneidade (concorrência)
- Existe uma diferença entre a concorrência física (simultaneidade no tempo) e a concorrência lógica (independência causal) de eventos
- Devido a assincronia intrínseca de sistemas distribuídos, a concorrência física é impossível de ser comprovada, e não acrescenta nada à compreensão de algoritmos distribuídos. => em outra execução poderia não ter ocorrido.
- A **concorrência lógica** é mais importante, pois mostra as possíveis permutações de eventos independentes.
- Dois modelos utilizados são:
  - Interleaving model –> todas as possíveis permutações de eventos logicamente concorrentes (como se não houvesse concorrência física)
  - Modelo de ordem parcial –> mostrar eventos distribuidos e explicitar as suas dependências causais
-

Mecanismos de Sincronizacão

- A única forma de sincronizar os estados locais de processos é através de mensagens.
- Tal sincronização é essencial para que os processos (de um sistema distribuido) possam adquirir uma visão (parcial) do estado global do sistema, e se coordenarem para uma ação conjunta.
- Os mecanismos de sincronização também servem para a gestão de concorrência no acesso a recursos.
- Alguns exemplos de problemas que requerem sincronização:
  - Sincronização de relógios locais
  - Eleição de líder
  - Exclusão Mútua
  - Detecção de término
  - Coleta de lixo distribuida

Tempo e estado global

- Alguma noção de causalidade é necessária.
- Isso pode ser através da sincronização de **relógios físicos** ou através da noção de **tempo lógico**.
- Tempo lógico é uma **abstração de ocorrência de eventos e sua relações de causa e feito.** Essa causalidade certamente será consistente com o tempo real (de parede).
- Tempo lógico permite:
  - (i) expressar a causalidade entre eventos no programa distribuído, e
  - (ii) acompanhar o progresso relativo em cada processo.
  - (iii) decidir sobre o que é um estado global consistente do sistema distribuído
- Devido à natureza distribuída, não é trivial um processo capturar o **estado global** do sistema (todos os processos),
- Para tal, precisa-se capturar os estados locais de cada processo de forma coordenada
- E lembrar que o estado global também engloba mensagens em trânsito

Comunicação de Grupo e entrega ordenada de mensagens

- Um grupo é um conjunto de processos que compartilham um contexto comum e colaboram em uma tarefa comum em um domínio de aplicação.
- Processos podem entrar/sair do grupo dinamicamente, e podem falhar
- Um grupo precisa de comunicação e um gerenciamento de grupo eficiente
- Quando vários processos enviam mensagens simultaneamente, diferentes destinatários podem receber as mensagens em ordens diferentes, o que pode violar a semântica do programa distribuído.
- Por isso, precisa-se definir o que exatamente significa entrega ordenada e depois implementada-la eficientemente.

Monitoramento de Predicados globais

- Predicados são definidos em termos de variáveis locais de diferentes processos e são usadas para especificar condições sobre o estado do sistema global,
- são necessárias para aplicações como depuração, sensoriamento do ambiente, e controle de processos industriais.
- Precisa-se portanto de algoritmos capazes de monitorar continuamente tais predicados.
- Um paradigma usado para tal monitoramento é streaming de eventos, em que os fluxos de eventos relevantes relatados de dife-rentes processos são examinados coletivamente para detectar predicados.
- Tipicamente, a especificação de tais predicados utiliza relações temporais físicos ou lógicos.

Manutenção de consitência entre processos replicados

- Vários serviços em um SD são implementados por processos replicados => objetivo: maior eficiência, escalabilidade, ou tolerância à falhas.
- O gerenciamento de réplicas, na presença de acessos concorrentes e adição e remoção de réplicas, introduz o problema de garantir a consistência entre o estado das réplicas.
- Além disso, a decisão sobre o número de réplicas e o seu local de execução são determinantes para garantir a eficácia do serviço replicado.
- Em alguns casos, também depende da localização dos recursos (p.ex. Bases de dados)

Sistemas Tolerantes a Falhas

- Sistemas reais podem falhar de várias formas, e isso deveria ser levado em conta nos algoritmos.
- O problema específico em AD é que as falhas afetam somente alguns processos, e não são imediatamente detectadas
- Um sistema seguro e tolerante a falhas tem vários requisitos e aspectos, e estes podem ser tratados usando várias estratégias:

Tipos de Sistemas Distribuídos

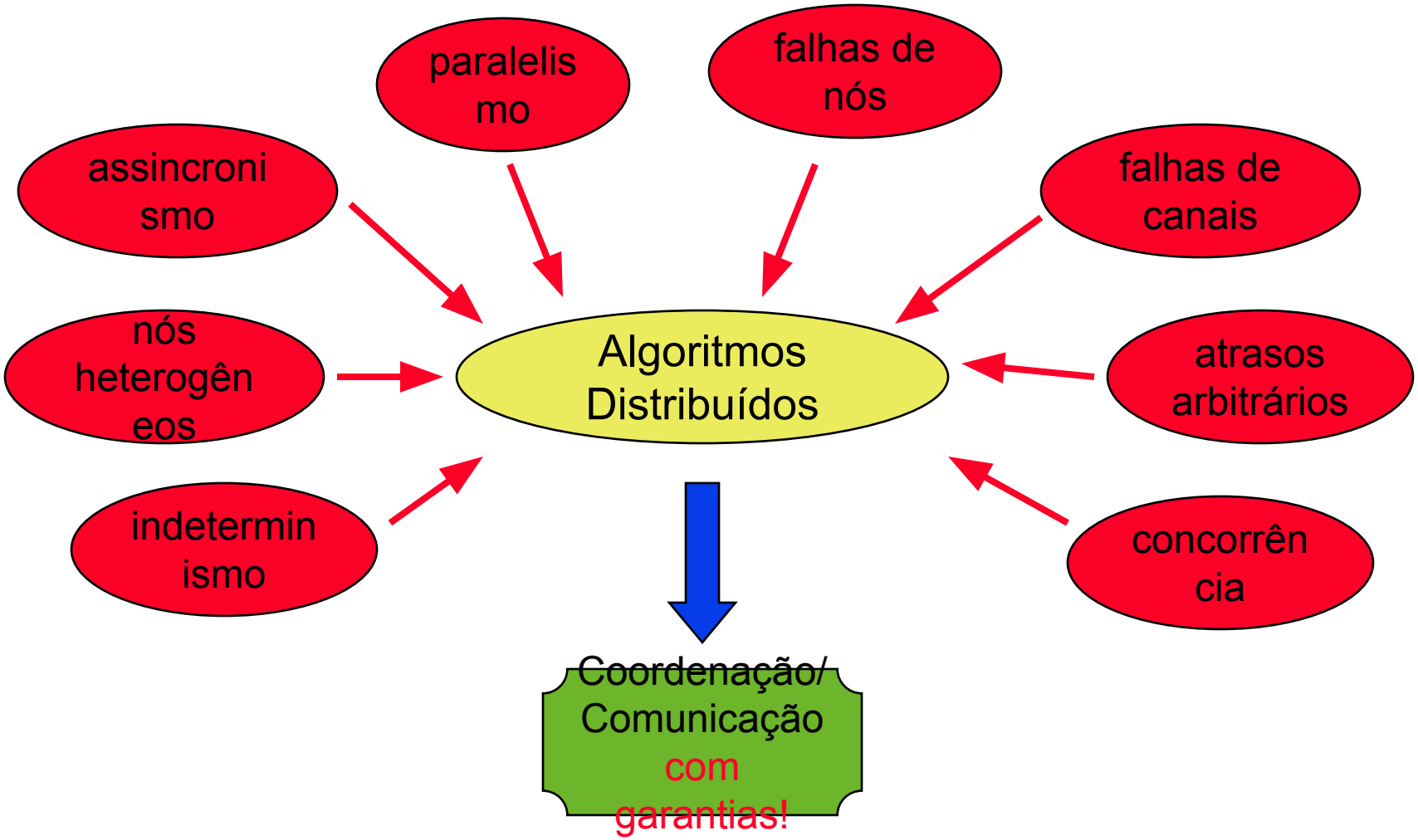
- Client-servidor:
  - recurso é centralizado
- Multi-tier
  - Serviços replicados, acesso coordenado a recursos distribuidos
- Clusters: homogêneo
  - (HW e plataforma)
- Cloud Computing
  - (heterogeneidade e falhas)
- Peer-to-Peer
  - (não há necessidade de coordenação no acesso)
- Redes de Sensores
  - Aplicações são simples coletas de dados

Problemas Fundamentais de AD

- **Eleição**: Escolher um único processo, e fazer com que todos os demais saibam qual é;
- **Exclusão mútua**: garantir acesso exclusivo a um recurso compartilhado (p.ex. base de dados);
- **Terminação**: detectar, com certeza, que o processamento distribuído terminou;
- **Consenso**: determinar um único valor a ser escolhido por todos os processos;
- **Snapshot**: obter uma "fotografia" correta de um estado distribuído do sistema a partir de coleta de estados nos processos;
- **Multicast**: garantir a entrega confiável de mensagens a **todos** os processos (e que estes saibam que todos receberam)
- **Membership:** Todos os processos ativos terem a mesma informação de quem está ativo.

- Obs: roteamento em redes geralmente não é descrito tratado em algoritmos distribuidos porque as tabelas de roteamento não precisar estar totalmente consistentes

Elementos Complicadores



Problemas Intrínsecos em Sistemas Distribuídos

Paralelismo e Indeterminismo:

- Processos executam em paralelo (simultaneamente)
- Processos executam em nós independentes
- Nós podem ter velocidades de processamento diferentes
- Escalonamento de execuções em cada nó está fora de controle (pelo S.O.)

Impossibilidade de Sincronização perfeita entre processos.

As dificuldades são:

- ausência de sincronismo dos relógios dos nós da rede
- a duração da transmissão de cada mensagem é imprevisível
- falhas de processadores ou na comunicação podem impossibilitar a sincronização

Exemplo: É imposível garantir que em Δt segundos todos processos troquem mensagens

Problemas Intrínsecos em Sistemas Distribuídos

Estado Global Distribuído: como a execução é paralela e sincronização só ocorre através de mensagens é:

- impossível saber qual é o estado global de um programa distribuído em um instante exato de tempo
- é dificil verificar a invariância da maioria dos predicados sob o estado global.

Exemplo: seja X variável de P1, Y variável de P2:

P1 executa: X = X - 10; send(P2,m);

P2 executa: receive(P1,m); Y= Y+10;

Assumindo-se que mensagem m não é perdida, pode-se dizer que X+Y é invariante (vale em qualquer estado global)?

A depuração de algorítmos distribuídos é complexa!

- em cada execução apenas uma dentre muitas possíveis sequências de ações são executadas
  - é impossível re-criar um estado global específico
  - para se ter certeza da corretude do algorítmo teria-se que testar todas as possíveis sequências

Problemas Intrínsecos

Falhas: precisa-se garantir o funcionamento correto do algoritmo mesmo com falhas independentes de nós e dos canais de comunicação.

- Para tolerância a falhas, deve-se ter:
  - redundânca de dados, de processamento e/ou comunicação
- Redundância à falhas → algoritmos menos eficientes

Compartilhamento: os elementos de um programa distribuído também compartilham recursos de diferentes tipos.

Exemplos de compartilhamento:

- meio de comunicação (largura de banda ou acesso exclusivo)
- dispositivos e serviços globais (servidor de nomes, de arquivos, etc.)
- arquivos (necessidade de lock/unlock) ou dados (banco de dados, registros)
- controle (eleição de coordenador), esquema distribuído de prioridades

 Tudo isso se reflete no indeterminismo da execução do algoritmo distribuído.

Caracteristicas fundamentais esperadas

Caracteristicas fundamentais esperadas de algoritmos distribuídos:

- evitar deadlocks = nenhum processo consegue prosseguir
- evitar starvation = um processo nunca ganha acesso
- garantir fairness = todos os processos têm chances iguais
- evitar livelock = subconjunto dos processos interagem, mas processamento global não progride
- garantir consitência = garantir que os processos convirjam para uma vião consistente/identica

Propriedades podem ser classificadas em duas categorias:

Segurança/Safety: Uma propriedade segura é aquela que vale para qualquer execução e em **todo estado (global) alcançável da execução**.
Exemplos: Ausência de deadlock, de starvation, ...

Progresso/Liveness: Uma propriedade de progresso é aquela que vale para qualquer execução e **se manifesta em algum estado alcançável da execução**.
Exemplos: um consenso é alcançado, um processo consegue entrar na sessão crítica, etc.

Modelos de Execução e Algoritmos

Um algoritmo realiza uma tarefa, baseado em certas premissas (o modelo de sistema)

Um modelo contém os elementos essenciais para:

uma análise do comportamento de um algoritmo distribuído em um determinado tipo de sistema.

Principais questões que definem um modelo:

- Quais são as entidades ativas e passivas do sistema (rede+nós) ?
- Como estas entidades interagem?
- Quais eventos externos ao algoritmo (ações do sistema/ambiente) podem influenciar o comportamento do algoritmo?

Através de um modelo:

- explicita-se todas as premissas relevantes sobre o sistema (características de comunicação, grau de sincronização da execução, e de falhas)
- dadas as premissas, pode-se provar se existe um algoritmo capaz de realizar a tarefa (algumas vezes por prova matemática)
- cria-se abstrações, que caracterizam os sistemas alvo, e permitem focar nas propriedades relevantes do algoritmo

Modelando Sistemas Distribuidos

Quando se descreve uma computação distribuida precisa-se determiner:

- Computação: Conjunto de Processos, comportamento determinístico vs probabilistico
  - papeis iguais ou existe um papel singular?

- Interação: através de mensagens, que acarretam:
  - Fluxo de informação para compartilhar o que?
  - Coordenação: sincronização e ordem de atividades
  - Ordem das mensagens (FIFO) é preservada?

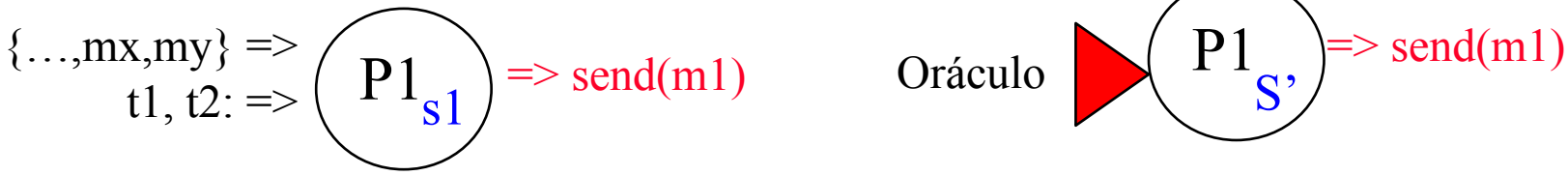
- Falhas: quais tipos de falha podem ocorrer?
  - Falhas benignas vs malignas (Bisantinas)
  - No processamento vs na comunicação
  - Falhas ocorrem aleatóriamente
  - Temporárias vs definitivas

- Tempo: pode-se fazer alguma premissa sobre limites de tempo
  - na comunicação? e
  - nas velocidades de processamento?

Modelando Sistemas Distribuidos

- Processos Determinísticos: o processamento local e as mensagens enviadas pelo processo dependem exclusivamente do estado corrente e das mensagens recebidas previamente

- Processos Probabilísticos: processos acessam oráculos para escolhar aleatoriamente a computação local e a nova mensagem a ser enviada.



Modelos de Execução e Algoritmos

Quais são os principais elementos do modelo e como interagem?

Geralmente:

- processos
- canais de comunicação (entidade ativa ou passiva)

Quais características adicionais do sistema devem ser levadas em conta pelo algoritmo?

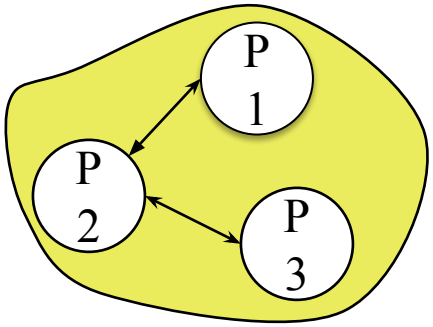
- Canal: Sincronização entre envio e recebimento, confiabilidade, e ordem de entrega de mensagens;
- Tipos de Falhas assumidas;
- Riscos à segurança.

O Modelo de Execução e Corretude de algoritmos

A corretude de um algorítmo distribuído (suas propriedades desejáveis), só pode ser verificada em relação ao modelo de sistema assumido.

Cada modelo expressa um conjunto de premissas sobre o tipo específico de sistema/rede (p.ex.: propriedades de atraso, confiabilidade, falha, temporizacão).

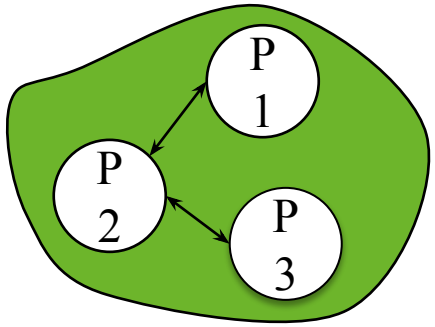
A corretude de um AD depende da prova de que o mesmo satisfaz as propriedades, em qualquer execução (i.e. na presença de qualquer conjunto de eventos possíveis no modelo).



Modelo 1

Possíveis propriedades desejáveis:

1) mensagens são **sempre** entregues na ordem em que foram enviadas
2) o algoritmo **eventualmente** termina/converge
3) **Sempre** existe um único coordenador
4) Todos as réplicas possuem **sempre** o mesmo estado



Modelo 2

Premissas do **Modelo 1:**

- latência máxima de todas transmissões são iguais e são conhecidas
- mensagens nunca são duplicadas ou modificadas

Premissas do **Modelo 2**:

- latência de transmissão arbitrária
- toda mensagem enviada em algum momento chega
- mensagens nunca são duplicadas ou modificadas

Premissas de Modelos de Execução

Premissas do Modelo surgem das respostas a perguntas como:

- há um conjunto estático ou dinâmico de processos?
- a topologia/ grafo de canais de comunicação é fixo ou variável?
- os tempos máximos de transmissão de mensagens são conhecidos?
- o processamento nos processos é síncrona (em rodadas, ou barreiras de sincronização?)
- os relógios dos processos diferem de no máximo Δt ?
- qual é modelo de falhas de nós (persistência de estado, falha detectável por outros processos)?
- qual é modelo de falhas de comunicação?
- qual é o modelo de compartilhamento de recursos (no nó e no canal de comunicação)?

A importância do Modelo de Execução

Algorítmos que funcionam para modelos menos restritivos são mais gerais, mas tendem a ser menos eficientes,

 pois introduzem redundâncias (de mensagens, de dados, de tempo) para compensar ausência de algumas premissas no modelo (p.ex. transmissão confiável de mensagens)

Algorítmos para um modelo mais restritivo tendem a ser mais simples, aproveitam melhor algumas carcaterísticas intrínsecas do sistema

 mas não funcionam se as premissas do modelo não forem satisfeitas

O grau de complexidade de um algorítmo distribuído depende :

- do grau de **incerteza** determinado pelo modelo (p.ex. possibilidade de falha, topologia dinâmica, etc.) e
- da **independência do comportamento** (execução/falhas) dos processos
- da **ausência de sincronismo**

Complexidade de um Algoritmo Distribuído

Geralmente é expressa em função do número de processos N e do número de possiveis falhas f.

- **Tempo**:
  - Numero de rodadas necessárias (para modelos síncronos)
- **Espaço**:
  - Tamanho de dados de controle adicionais acrescidos às mensagens
  - Tamanho do buffer a ser mantido em cada nó
- **Comunicação**:
  - Número de mensagens por rodada
  - Número de mensagens por evento inesperado

Modelos de Execução Fundamentais: Principais Aspectos

Principais aspectos dos Modelos de Execução:

Modelo de Interação & Sincronismo (Timing Models):

- toda comunicação e sincronização é através de mensagens
- topologia, grau de confiabilidade das mensagens e sincronismo
- definição do possivel atraso na comunicação & grau de sincronismo (capacidade de sincronizar ações)

Modelo de Falhas:

- os tipos de falha esperados (omissão ou aleatória)
- número máximo de falhas simultâneas
- permite definir a estratégia para mascarar as falhas (*k resiliência*)

Modelo de Segurança:

- definição das possíveis formas de ameaças/ataques

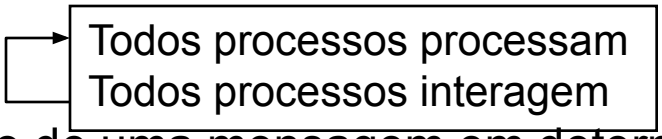
permite uma escolha dos mecanismos & protocolos para impedir (ou dificultar) os ataques (algoitmos criptográficos, gerenciamento de chaves, certificados,…)

Modelos de Interação e Sincronização

O Modelo Síncrono:

- cada “passo de execução” em um processo demora entre [min, max] unidades de tempo
- a transmissão de qualquer mensagem demora um limite máximo de tempo
- o relógio local a cada processo tem uma defasagem limitada em relação ao relógio real

A execução ocorre em rodadas de “processamento & comunicação” perfetamente sincronizadas:



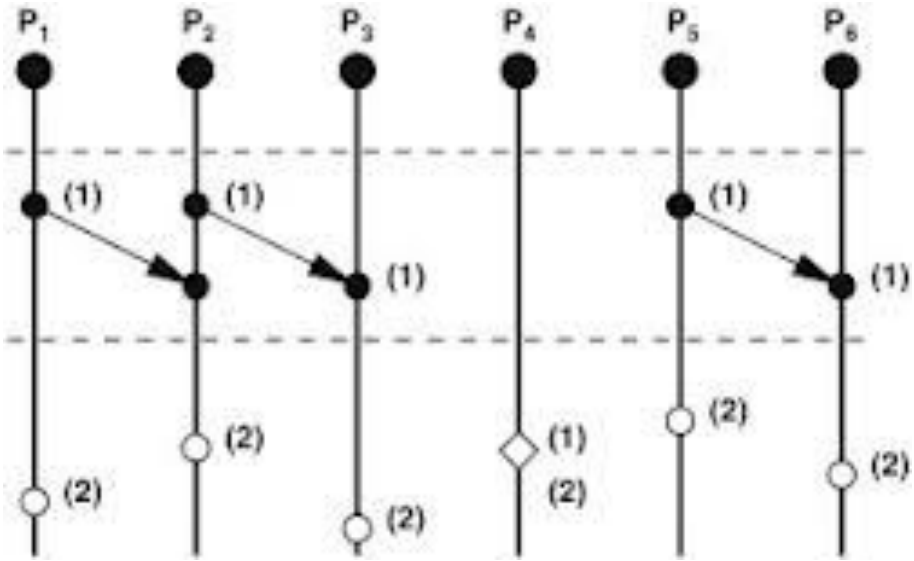
- O não-recebimento de uma mensagem em determinado período também traz informação!!

Obs: Em sistemas reais, é difícil determinar estes limites de tempo. Mas este modelo é adequado quando a rede é dedicada e as tarefas (processos) têm tempos de processamento fixos e bem conhecidos (p.ex. controle de processos, processador paralelo)

Timing Model Sincrono

Em cada rodada:

- Processamento em cada processo
- envio de mensagem para demais processos
- Início da proxima rodada apenas quanto todas as mensagens forem entregues



Exemplo de Algoritmo para Modelo Síncrono

Algoritmo para Consenso:

Cada nó escolhe um valor $v \in [0, max]$ e difunde este valor para todos os demais nós na rede. Quando um nó tiver recebido o valor de todos os demais nós, escolhe o valor mais votado (ou no caso de empate, um valor default) como o valor de consenso d.

```
Pseudo-código para nó i:
Init => {
      escolhe valor v;
      broadcast (v:i) para todos os vizinhos
      localset ← {(v:i)}
}
Nova rodada => {
      recebe conjunto c de mensagens dos demais;
      diff  ← c - localset; // diferença entre conjuntos
      se diff ≠ ∅ {
        localset ← localset ∪ diff;
        broadcast diff para vizinhos;
      } senão se (já recebeu de todos nós) {
        d ← acha_mais_votado(localset);
        terminou ← true;
      }
}
Terminou => imprime o valor de consenso d;
```

Modelo de Sistema:

- **Toplogia qualquer, com N nós**
- **Comunicação é confiável**

Propriedade garantida (?)

Após N-1 rodadas, um mesmo valor de consenso deverá ter sido impresso por todos os nós ativos (não falhos).

O Algoritmo está correto?

Consenso para sitema síncrono

- “assumindo-se comunicação confiável” □ não há perda de mensagens
- Todos os nós repassam aos seus vizinhos os novos valores recebidos □ isso garante a difusão dos valores para nós ainda não alcançados
- Com N-1 rodadas, garante-se ter atingido todos os nós, mesmo na topologia em cadeira (a menos favorável)
- A rede tem N nós, mas não diz-se nada sobre a conectividade mútua entre os nós. □ não exclui-se a possibilidade de partição de rede

Timing Models (de Sincronização)

Modelo assíncrono total:

- não há um limite máximo de tempo para a execução em um processo
- não há limite máximo de tempo de transmissão de uma mensagem
- não há limite máximo para a defasagem entre os relógios dos nós

Nada se sabe!!!

O Modelo assíncrono não permite qualquer suposição sobre tempos de qualquer execução.

Muitos sistemas/ redes (p.ex. Internet) são assíncronos, devido ao compartilhamento de roteadores e o uso de buffers, e a heterogeneidade de nós

Motivos do mundo real para presença do assincronismo:

- Comunicação: delays em roteadores, contenção no meio de transmissão
- Processos: escalonamento dos processos ddesconhecido, acesso concorrente a recursos locais, acesso a disco, etc.

OBS:

- O modelo síncrono é um caso particular do modelo assíncrono.
- Existem alguns problemas que simplesmente não têm solução em um modelo puramente assíncrono. Por isto é comum adotar-se modelos semi-síncronos com algumas suposições sobre tempos máximos.

Modelo de Falhas

Em um SD tanto processos como os canais de comunicação podem falhar.

O *Modelo de Falhas* define a maneira pela qual os elementos podem falhar.

Usa-se as seguintes macro-categorias:

- *Omissão:* quando um processo/canal deixa de executar uma ação (p.ex. uma transmissão)
- *Arbitrárias (ou Bizantinas):* quando qualquer comportamento é possível
- *Temporização: (somente sistemas síncronos)* quando o relógio local pode diferir da hora real, ou o tempo de execução em um processo (ou de comunicação) ultrapassam limites pré-estabelecidos

Falhas: uma classificação mais precisa

Falhas permanentes: principais categorias são:

- fail-stop: processo pára e permanece neste estado, e isto é detectável pelos outros processos
- crash: processo pára e permanece neste estado; outros processos podem ser incapazes de detectar isto
- omissão (de um canal): uma mensagem enviada por um processo não é recebida pelo outro processo
- omisão de envio (de processo): um processo completa o send(m), mas m não é inserida no buffer de envio do processo
- omissão de recebimento (de processo): mensagem é colocada no buffer de recebimento, processo nunca recebe a mensagem
- arbitrária/bizantina: o processo ou canal apresentam comportamento qualquer (p.ex. envio de menagens quaisquer, a qualquer momento, omissão ou parada)

Tem-se a seguinte hierarquia entre os tipos de falha:

fail-stop $\subset$ crash $\subset$ omissão $\subset$ bizantina

Tios de Falha

Falhas permanentes vs temporárias

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure - *Receive omission* - *Send omission*	A server fails to respond to incoming requests - A server fails to receive incoming messages (e.g. no listener) - A server fails to send messages (e.g. buffer overflow)
Timing failure	A server's response lies outside the specified time interval (e.g. time too short and no buffer is available or time too long)
Response failure - *Value failure* - *State transition failure*	The server's response is incorrect - The value of the response is wrong (e.g. server recognizes request but deliver wrong answer because of a bug) - The server deviates from the correct flow of control (e.g. server does not recognize request and is not prepared for it – no exception handling for example)
Arbitrary failure (Byzantine failures)	A server may produce arbitrary responses at arbitrary times

Different types of failures.

Fonte: Prof. Kalpakis

Modelo de Segurança

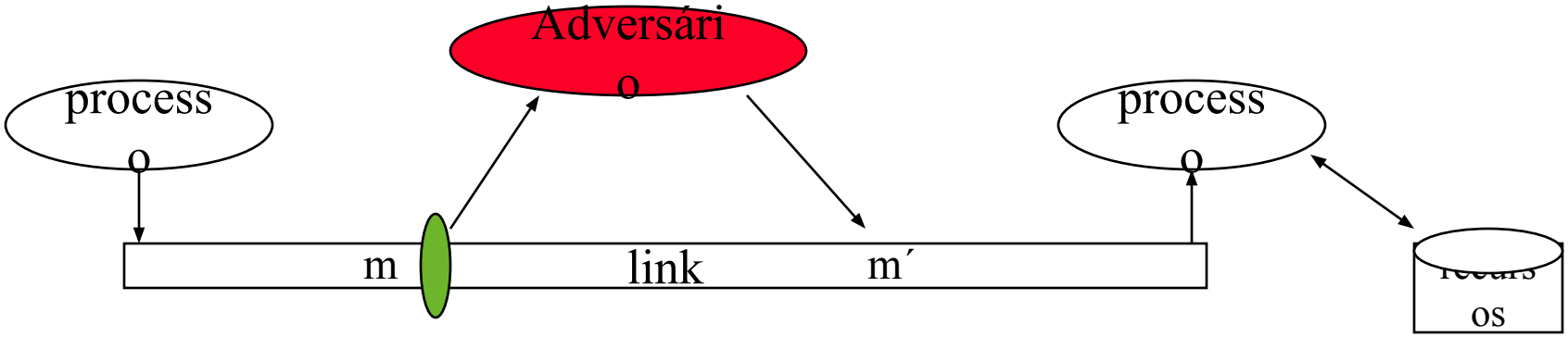
Segurança envolve os seguintes aspectos:

- *confidencialidade*: proteção contra revelação de dados a usuários não autorizados
- *integridade*: garantir que só o usuário autorizado irá modificar dados
- *disponibilidade*: garantir ao usuário autorizado acesso a dados/recursos
- *autenticação*: verificar se usuário é quem diz ser

Principais Riscos:

- interceptação de mensagens
- interrupção de serviço (*denial of service*)
- falsificação: (de dados, identidade, ou direito de acesso)
- duplicação e inversão de ordem de mensagens

Tipos de Ataques



Tipos de Ataques:

- Interrupção (mensagem é destruida ou ataque à disponibilidade do recurso)
- Interceptação (ataque à confidencialidade)
- Modificação (ataque à integridade)
- Fabricação (ataque à autenticidade)
-

Um canal (link) seguro garante:

- confidencialidade, integridade e unicidade da mensagem
- a autenticidade de ambos os processos envolvidos é garantida

Timing Model (de Sinconização)

Uma computação distribuída consiste de *um conjunto de sequências* de eventos e_{ij}, cada uma ocorrendo em um dos processos P_i.

$$Comp = \bigcup_{i=1..N} [e_{i1}, e_{i2}, e_{i3} \ldots]$$

Onde cada e_{ij} pode ser:

- qualquer evento interno a P_i
- Send(j, m) - envio de m para P_j
- Receive(k, m) – chegada de m de P_k

Se houvesse a noção global/consistente de tempo (com precisão infinita), poderia-se identificar exatamente a ordem de ocorrência de eventos na execução de um AD (p.ex.: colocando um timestamp global em cada evento)

No **Modelo Interleaving (intercalação)**, uma execução de um AD é representada como uma sequência global de eventos:

Exemplo de sequência:

1. P1 executa send(2,m)
2. P1 executa evento local
3. P1 executa receive(2,m2)
4. P2 executa evento local
5. P2 executa receive(1,m)
6. P1 executa send(1,m2)

Modelo de Interação e Sincronização

Problemas:

- É impossível identificar exatamente a ordem global (total) de eventos distribuídos (um problema relativistico), pois os relógios dos computadores precisariam estar perfeitamente sincronizados (impossível)
- O conhecimento da ordem exata de eventos de uma execução não fornece muitas informações sobre outras possíveis execuções do mesmo algoritmo

 Em compensação, em sistemas distribuídos apenas **a ordem parcial de ocorrência de eventos** de uma computação distribuída pode ser observada. Ordem parcial é definida pela **causalidade**.

Por isso, adota-se o **Modelo de Causalidade** (*partial order* ou *happened before*), ou causalidade potencial.

Modelo de causalidade: a ordem sequencial em cada processo define causalidade de eventos locais e eventos A:send(B,m) e B:receive(A,m) são causalmente dependentes

Modelo de causalidade potencial: difere do Modelo de Causalidade apenas pelo fato de que em cada processo também podem haver eventos locais concorrentes

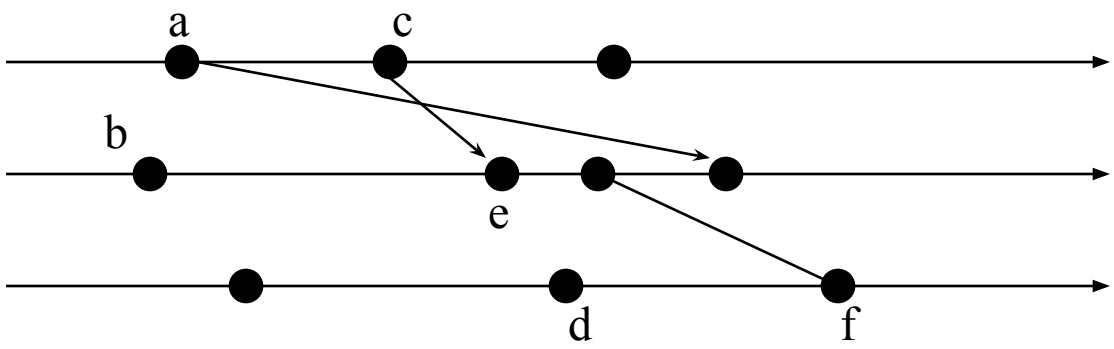
Relação "*Happened before*" entre Eventos

Def: evento *a ocorre antes de evento b* (notação: a $\rightarrow$ b) $\Leftrightarrow$

- a e b são eventos consecutivos no processo P e a terminou antes de b começar
- a= send(m) no processo P e b = receive(m) no processo Q
- a e b estão relacionados por transitividade:

  existem $e_1, e_2, .., e_N$ tal que $e_i \rightarrow e_{i+1}$, a = e_1, b= e_N

Def: eventos a e b são concorrentes (notação a || b) $\Leftrightarrow$ a $\not\rightarrow$ b e b $\not\rightarrow$ a



a → c **a || b a || d**

b → e **c || d e → f**

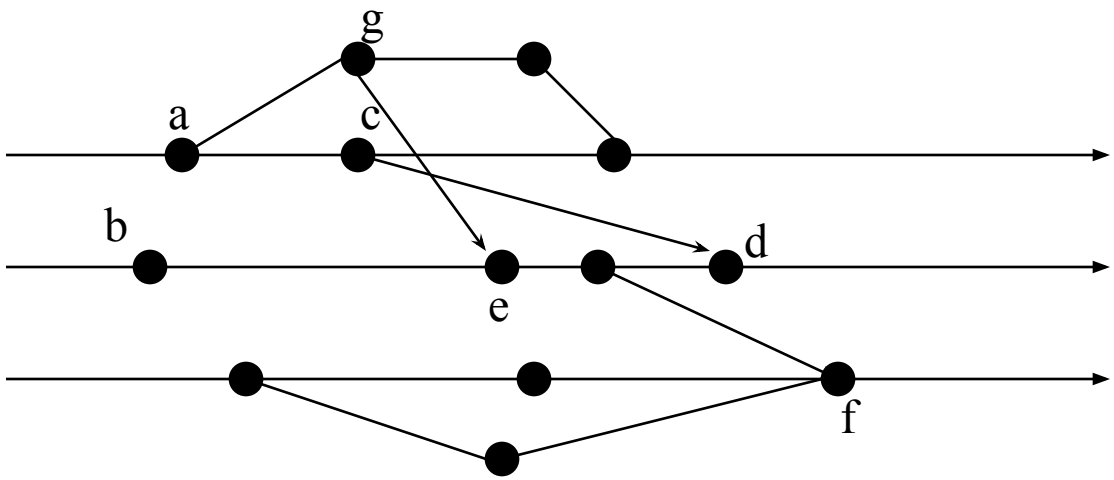
c → e **a → e**

Modelo de Causalidade Potencial

Mesmo que os eventos locais executem em ordem total, não é garantido que haja uma relação causal entre eles,

- p.ex. quando há threads concorrentes, ou então quando existe uma instrução receive de vários possíveis remetentes.

Quando precisamos caracterizar precisamente a dependência causal entre **todos** os eventos, então o modelo de causalidade potencial é mais apropriado. No entanto, devido a dificuldade de identificar causalidade local (interna) em um processo, esse modelo geralmente não é usado.



a → c **a || b g || d**

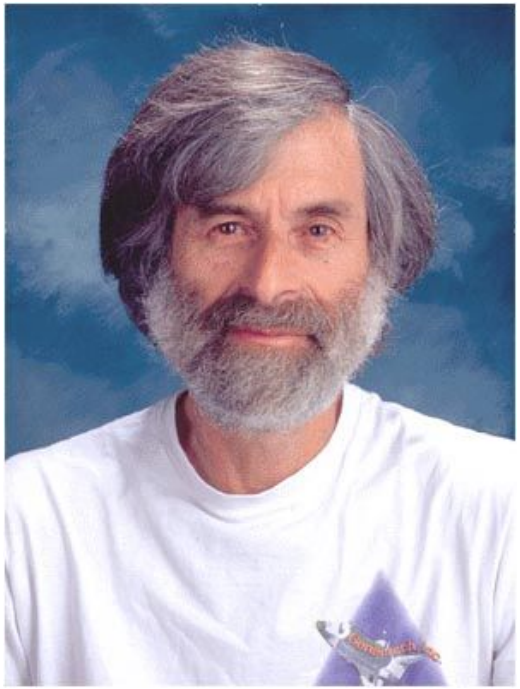
a → g **c || g e → f**

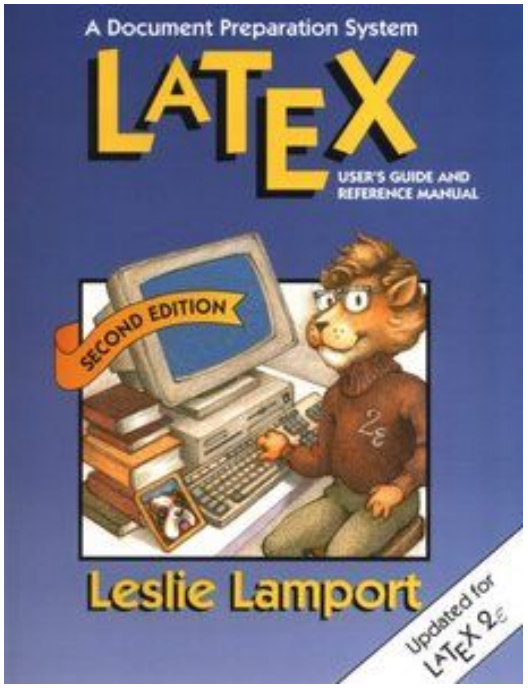
g → e **a → e**

Causalidade e Relógios Lógicos

Relógio Lógico

Proposto por Leslie Lamport no artigo: *Time, Clocks and the ordering of events in a Distributed System*. Communications of the ACM, 1978





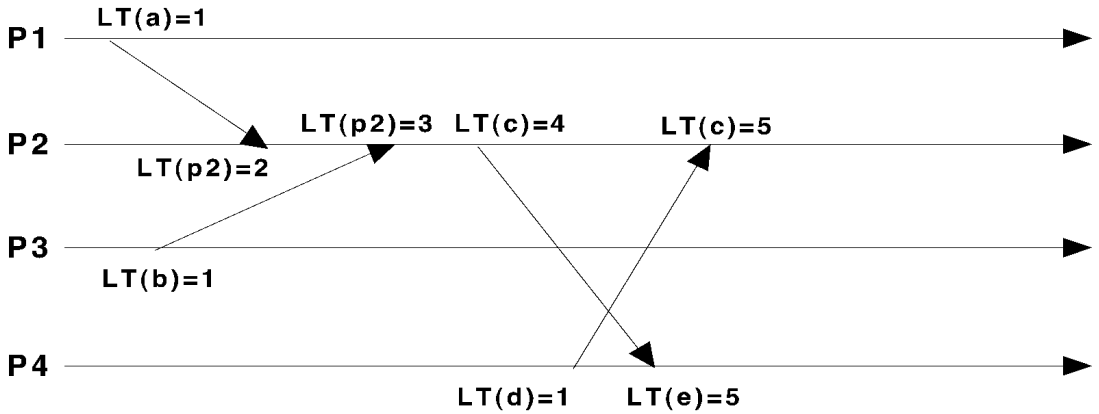
Laureado com o Prêmio Alan Turing em 2013

Relógio Lógico

Leslie Lamport propôs o conceito de relógio lógico que é consistente com a relação de causalidade entre eventos:

Idéia Central:

- cada processo p tem um contador local LT(p), com valor inicial 0
- para cada evento executado (exceto receive) faça LT(p) := LT(p) + 1
- ao enviar uma mensagem m, adicione o valor corrente LT(m):= LT(p)
- quando mensagem m é entregue a processo q, este faz LT(q) := max(LT(m),LT(q))+1



Consistência com a causalidade significa:

- se a → b então LT(a) < LT(b)
- se a || b então LT(a) e LT(b) podem ter valores arbitrários

Relógio Lógico (cont.)

O mecanismo proposto por Lamport **define uma ordem parcial consistente com** a dependência causal entre eventos**:**

se a → b, então (independentemente de a, b estarem ou não no mesmo processo), o valor LT(b) vai ser maior do que LT(a)

Apesar de não permitir a comparação de eventos concorrentes, o mecanismo pode ser estendido para impor uma ordem total (arbitrária) sobre todos os eventos.

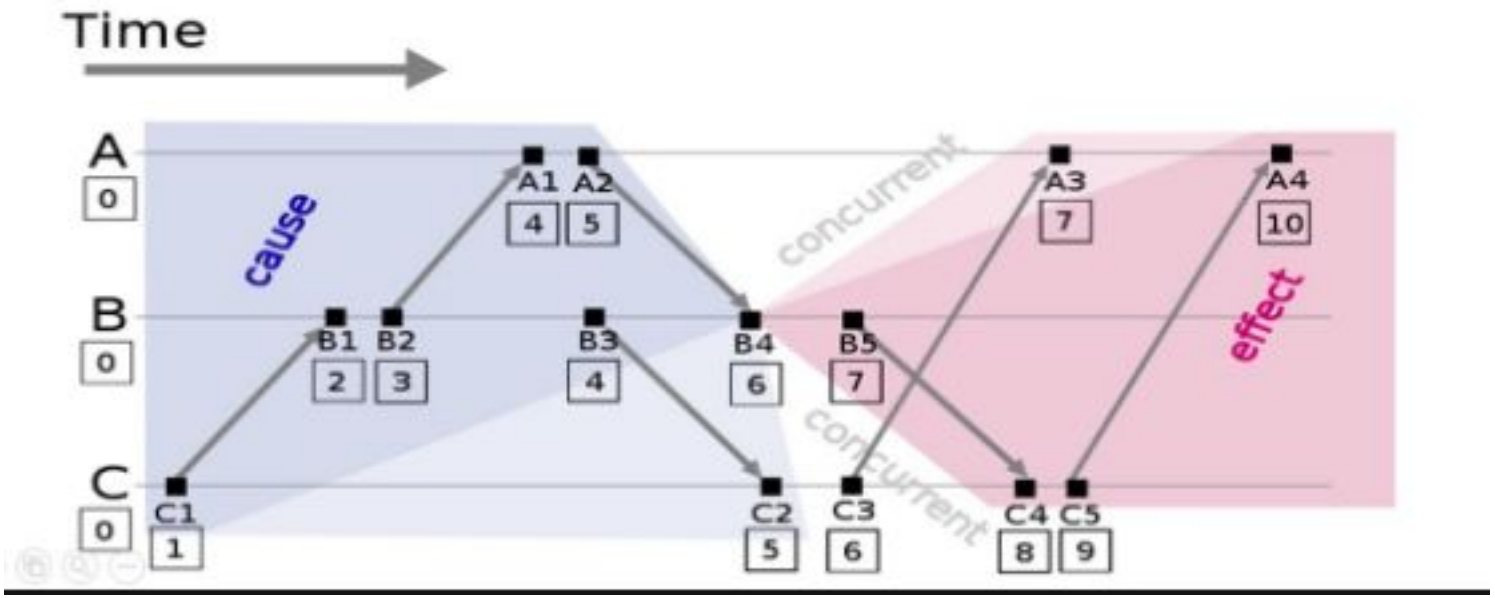
Para isto, basta adicionar o *process-ID* ao LT a fim de "desempatar" os casos em que o LT(a) = LT(b)

Outra vantagem é conhecer, para cada evento, o número de eventos (considerando todos os processos) que **potencialmente causaram/influenciaram** o evento.(i.e. a altura do evento).

Relógio Lógico (cont.)

Regiões de Causalidade e Efeito:

- Exemplo: do evento B4:



Modelo de Sistema para implementar o Relógio Lógico

Relógio Lógico é um exemplo de protocolo/mecanismo distribuído, que:

- não assume nada com relação a sincronismo e tempos de transmissão
- é tolerante às falhas de omissão (crash ou perda de mensagem)
- permite uso por um número variável de participantes
- impõe um overhead mínimo ao sistema
- está totalmente desvinculado do relógio real e mesmo assim pode ser usado para impor uma ordem total "consistente" ao conjunto de eventos distribuídos

Em quais condições ele não funciona?

- Só não é tolerante à corrupção de mensagens, em particular do time-stamp e da informação sobre remetente (falha arbitrária)
- E todos os processos precisam aderir ao protocolo.

Exercício para 1/04

Ler caps 1 & 2 do Livro (Kshemkalyani & Singhal)

Dar exemplos de:

- um sistema distribuidos que garante entrega FIFO, mas não garante entrega causal de mensagens;
- Uma coleção de estados locais gravados (não respeitando a causalidade) e que formam um estado global incorreto;

Obs: tente ser o mais formal/preciso possivel (usando a notação do livro)

Formas de Interação

Depende da unidade de processamento e distribuição: processo ou objeto.

Aspectos gerais:

- cada unidade define um espaço de nomes próprio e possui um estado
- unidades interagem entre sí de forma síncrona ou assíncrona.
- unidade invocadora possui uma referência para a unidade a ser invocada.

Referências:

- Chow, Johnson: Distributed Operating Systems and Algorithms, Seção 4.1
- G. Andrews: Paradigms for Process Interaction in Distributed Systems, ACM Computing Surveys 23(1), Março 91

Formas de interação:

1. Envio de mensagems
2. Comunicação de Grupo
3. RPC e outras formas de interação

Envio de Mensagens

É a forma de interação mais primitiva, pois:

- é mais parecida com o que efetivamente ocorre no nível físico e do SO
- é baseada no paradigma da E/S
- o programador precisa definir os tipos de mensagens a serem usadas
- o envio e espera por uma mensagem, bem como a construção e a interpretação da mesma ficam explícitos no programa

Principais componentes:

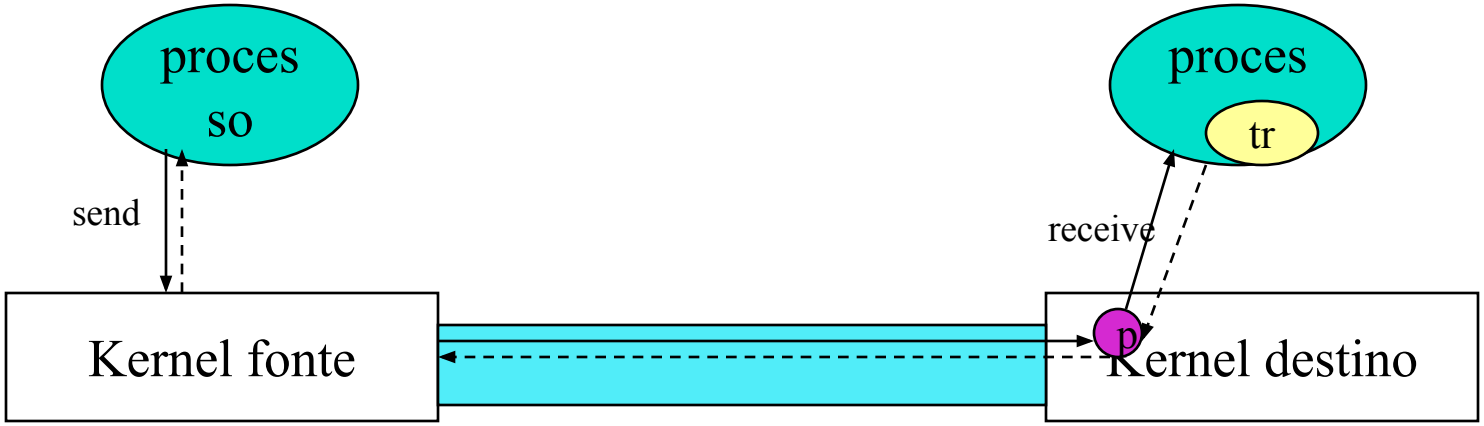
- Tipo (ou Formato) da Mensagem
- Endereço para recebimento de mensagens (porta)
- Referência para uma porta remota
- Semântica das primitivas send() e receive()
- Mecanismo de invocação do tratador da mensagem correspondente (*handler*)

Dependendo do sistema de apoio à execução (runtime system), a referência contém ou não explicitamente o endereço do processo destino.

Por exemplo:

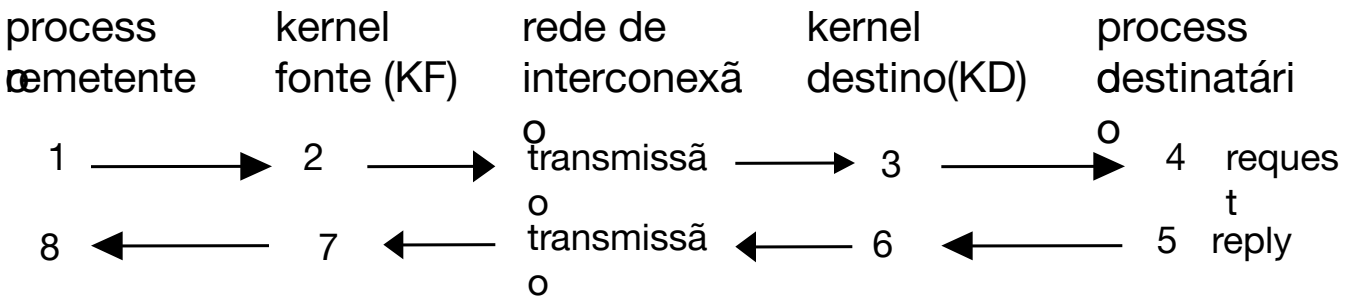
- Processos servidores declaram as portas nas quais podem receber mensagens junto ao runtime system (ou sistema operacional) .
- Processos clientes obtém esta informação (de alguma forma), instanciam as respectivas referências remotas e usam-nas para o envio de mensagens (Exemplo sockets).

Envio de Mensagens: Semântica



- Envio simples (Request Protocol)
- Envio com confirmação (Request-Reply Protocol)

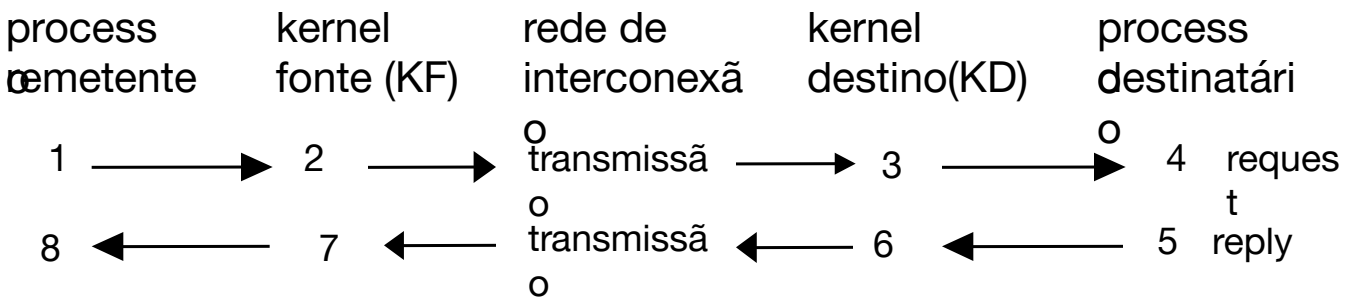
Mensagens: Sincronismo e Bufferização



Sincronização no envio de mensagens pode ocorrer em diferentes etapas da comunicação: entre o processo do usuáro e kernel, entre kernel fonte e destino ou entre processo fonte e destino.

Existem várias possibilidades de implementação de sincronismo nas primitivas de comunicação **send** e **receive.**

Mensagens: Sincronismo e Bufferização



Primitiva **receive** geralmente é bloqueante.

Na primitiva **send**: opções para desbloqueio do processo remetente**:**

- **não-bloqueante** (1-2): desbloqueio após cópia para o kernel KF
- **bloqueante** (1- transmissão): desbloqueio após transmissão para a rede
- **bloqueante-confiável** (1-3): desbloqueio após recebimento pelo kernel KD
- **bloquenate-explícito** (1-4): desbloqueio após entrega para processo destinatário
- **request-reply** (1-8): desbloqueio após processamento da mensagem pelo destinatário e envio de reposta ao remetente.

Usa-se buffers em diferentes etapas para compensar as diferentes taxas de envio e recebimento de mensagens. Conceitualmente, buffer poder ter tamanho 0 ou ∞.

Envio de Mensagem: Semântica

Síncrono (Symmetric Rendezvous):

Processos se sincronizam (bloqueiam) nas operações de send e receive P1 { ref a:b } P2 { port b }

send(a,m) ⟷ recv (b, m)

Ex: CSP, DP

Assíncrono unidirecional:

Processo cliente não é bloqueado, mas processo servidor fica bloqueado até chegada da mensagem.

P1 { ref a; bind(a,b) } P2 { port b }

send (a,m) → recv(b, m)

recv(a,resp) ← send(b, resp)

Ex: sockets

Assíncrono bidirecional (Call-back):

Toda interação consiste de um envio + resposta. O cliente fornece uma referência "c" para a qual a resposta deve ser enviada.

P1 { ref a:b; port c } P2 { port b; ref d }

a.send (a, m+c) → recv(b, m+d)

recv (c, resp) ← send(d, resp)

Ex: Regis

Endereçamento Direto

A forma de endereçamento é independente do sincronismo no envio de mensagens.

A questão central: O que é uma referência e como o remetente obtém uma para o destinatário ?

No **endereçamento direto**: remetente precisa conhecer o (endereço do) processo receptor para interagir com o mesmo.

Alternativas para o endereçamento direto:

1) Simétrico: o nome (ou identificação) do processo remoto é indicado explicitamente no send/receive em ambos os processos.

P: send(Q,m) ——————→ Q: recv(P, X)

Exemplos: Linguagens/Ambientes PLITS, DP, CSP, Occam

Prós & Contras:

(-) configuração do programa é estática e fixa no programa

(+) em tempo de compilação todas as referências remotas são resolvidas e o mapeamento é informado para o runtime/S.O. (

 Permite checagem de tipos, existência de processo par, e comunicação mais eficiente

Endereçamento Direto (cont.)

2) Assimétrico: similar ao anterior, com a diferença de que o processo receptor não precisa indicar explicitamente o cliente, mas pode receber mensagens de qualquer processo que conheça a sua porta.

Geralmente a primitiva **recv** permite ao receptor identificar o remetente (cliente) a fim de:

permitir o processamento concorrente de requisições (threads) e ter como associar as respostas as respostas às requisições correspondentes

Exemplos: sockets, SR

Prós & Contras:

(+) permite a implementação de servidores

(-) processos cliente precisam conhecer o endereço do servidor

(-) este endereço precisa ser indicado em cada send

Endereçamento Direto (cont.)

3) Orientado a conexão: dois processos criam uma (ou mais) conexões entre sí, totalmente independentes entre sí.

Em vez do nome do processo, usa-se a identificação da conexão como referência.

Exemplo:

- conexão TCP é caracterizada pela tupla (*IP-F, porta-F, IP-D, porta-D*)

Prós & Contras:

(+) Processos podem ter várias conexões independentes (fluxos independentes)

(+) As conexões podem se encarregar de: fragmentação, controle de fluxo, e retransmissão de fragmentos

(-) Antes da interação, os processos precisam criar as conexões

(-) Existe um custo associado à criação de uma conexão

(-) Conexão implementa uma comunicação entre pares (cada cliente precisa criar a sua conexão parao servidor)

Endereçamento Indireto

Endereçamento indireto: quando tanto o processo remetente como o receptor não precisam conhecer a identidade do outro processo.

Isto possibilita composição modular e flexível de componentes.

Existem as seguintes alternativas:

1) Mailbox = é uma estrutura de dados que armazena mensagens com entrega FIFO; mailboxes estão totalmente desvinculadas dos processos remetente e receptor. Estes mantém uma referência para uma mailbox comum.

(+) permite comunicação anônima, assíncrona e N para M

(+) permite comunicação multi-ponto (N processos-1 Mailbox) e multi-caminho (N Mailboxes entre dois processos)

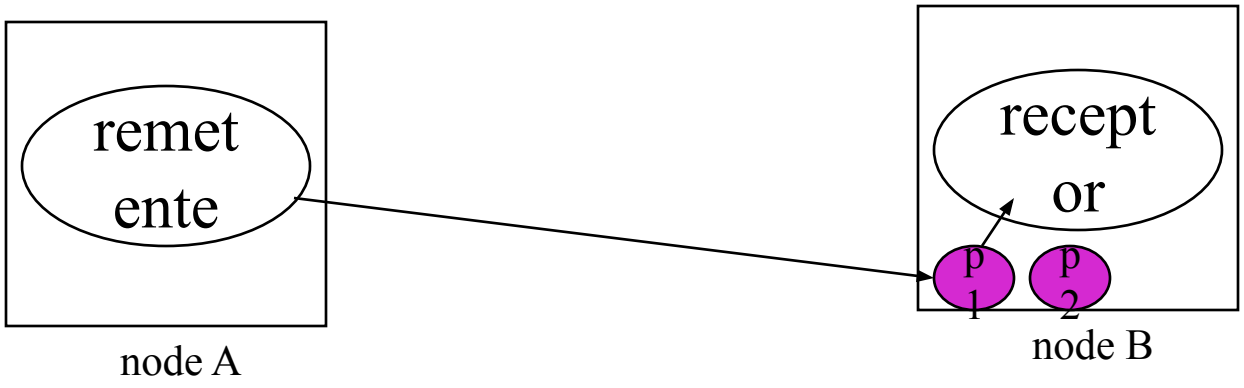
(-) por ser independente dos processos, fica difícil verificar se um programa distribuído realiza a interação desejada

Endereçamento Indireto

2) Porta = é uma mailbox associada a um processo receptor, e implementa uma entrega de mensagens FIFO.

(+) A porta é gerenciada pelo runtime system ou kernel (tamanho dinâmico) e um processo pode ter um número arbitrário de portas.

(-) quando portas não têm uma identificação global(+única) no sistema, referência consiste de (nodeID; portID), que precisa ser descoberta pelo processo remetente



Endereçamento Indireto (cont.)

3) Nomes locais:

- Linguagem de programação é estendida para permitir a declaração de nomes locais (port/ portref) tipados
- ligação de referências a portas é feito na etapa de configuração do sistema através de uma linguagem de configuração:

Exemplos: Conic, Regis, Durra

```
create P of main.exe("Hallo world")
link P.out -- Q.in
unlink P.in -- Q.out
```

Prós & Contras:

(+) escolha e conexão dos módulos de um sistema é feita através de uma **linguagem de configuração** (plug-and-run), independente da linguagem de programação

(+) a linguagem de configuração testa a compatibilidade entre portas e referências (e possivelmente outros atributos)

(+) certos ambientes possibilitam uma configuração dinâmica do sistema

(-) requer um runtime system especial, e não há sincronização entre configuração e execução

Endereçamento Indireto (cont.)

4) Serviço de nomes:

- mantém associações {nome,endereço}
- um processo servidor registra as suas portas junto a um servidor de nomes (SN
- processos clientes consultam o SN a fim de obter a referência para a porta do servidor.

Prós & Contras:

(-) requer mensagens adicionais ao SN (registro, consulta)

(-) SN é ponto central de falha e pode vir a ser um gargalo

(+) geralmente consulta ao SN só é feita antes do primeiro envio

(+) SN pode manter referências para vários servidores equivalentes e fazer uma distribuição dinâmica dos endereços de servidores para os clientes

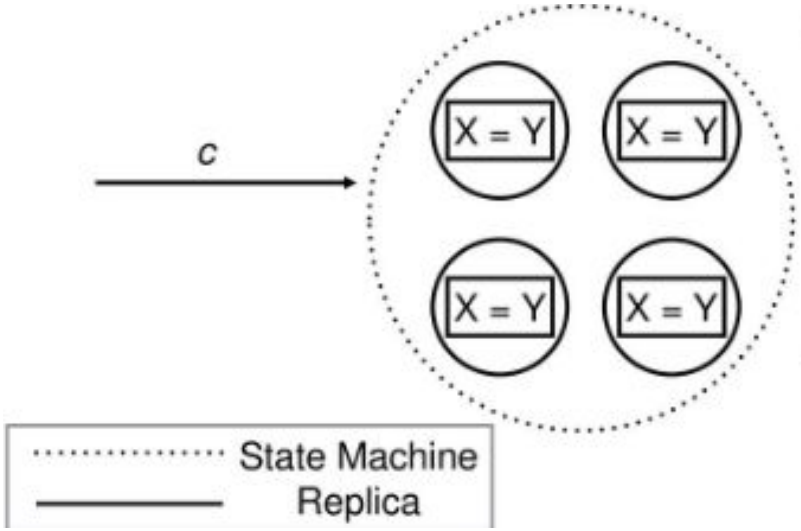
(+) um SN pode fazer a seleção do servidor através da comparação dos atributos requeridos e providos

Não confindir com serviço de diretório:

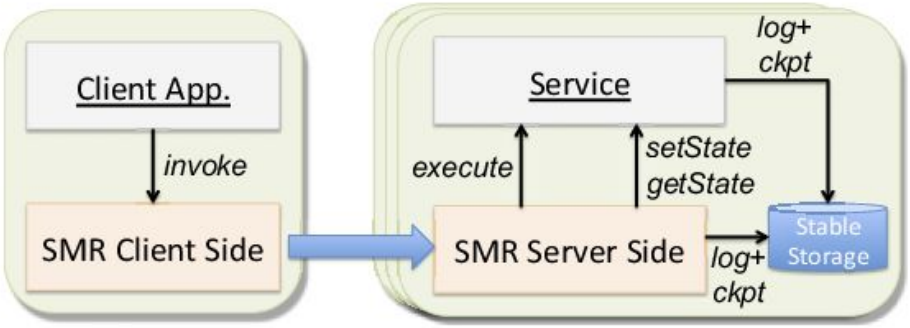
busca por um nome, a partir de atributos (yellow pages)

State Machine Replication

Uma forma de contornar falhas de fail-stop



- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine



Comunicação de Grupo e Multicasts

Em algumas aplicações há a necessidade de descrever a comunicação para um grupo de elementos (processos).

A noção de um grupo é essencial para aplicações distribuídas que:

- lidam com coordenação de atividades/ações distribuídas
- lidam com difusão de dados (exemplo: video-sob-demanda)
- Implemantam tolerância a falhas e alta disponibilidade através da replicação de dados/processamento.

Principal Objetivo: Liberar o desenvolvedor da aplicação da tarefa de gerenciar várias comunicações ponto-a-ponto para os membros do grupo.

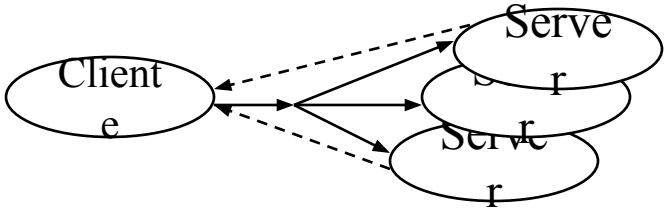
Principal vantagem:

- transparência de replicação

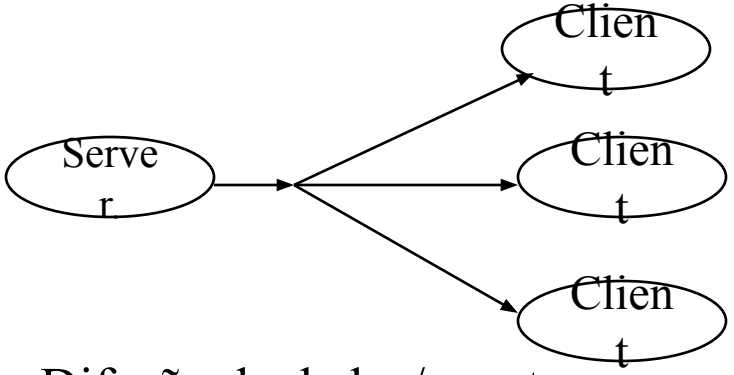
Serviço de grupo é um serviço de middleware que provê uma API e protocolos eficientes para:

- comunicação 1-para-N
- gerenciamnto de pertinência no grupo.

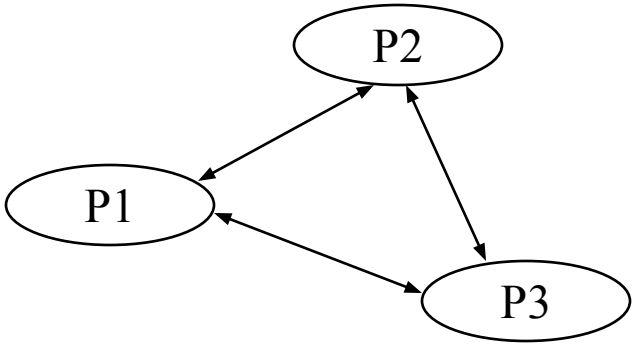
Usos de Multicasts



Servidores replicados
Exemplo: Tolerância à falhas



Difusão de dados/eventos
Exemplo: Streaming



Aplicações Multi-Peer
Exemplo: Instant Messaging

Comunicação de Grupo: Confiabilidade

Existem três categorias de requisitos de confiabilidade para comunicação de grupo:

a) Multicast não-confiável: o quanto mais membros receberem a mensagem, melhor.

- se ocorrerem falhas nos enlaces e/ou nos processos, alguns membros do grupo podem eventualmente não receber a mensagem de multicast

Exemplo: IP multicast (principal objetivo: eficiência da difusão)

b) Melhor dos esforços (best-effort multicast): garante-se que em algum momento futuro todos os destinatários conectados irão receber a mensagem

- se necessário, a mensagem pode ser replicada, e destinatários devem ser capazes de descartar mensagens repetidas
- destinatários não necessariamente precisam manter seus estados consistentes (p.ex. stateless servers)
- Remetente não precisa saber quem efetivamente recebeu a mensagem

 A entrega a todos é garantida através da retransmissão da mensagem (p.ex. Flooding)

Comunicação de Grupo

c) Confiável (reliable muticast): é necessário garantir que todos os membros do grupo mantenham os seus estados sincronizados

• para tal, toda requisição deve ser ou recebida por todos os servidores, ou então por nenhum deles (propriedade de atomicidade, *all-or-nothing*)

• para isto, cada membro do grupo precisa armazenar temporariamete toda mensagem recebida e executar um protocolo de confirmação (entre as réplicas) para decidir/confirmar se a mensagem pode ser entregue para a aplicação.

• quando o grupo muda (falha ou entrada de um membro):

- a entrega de mensagens deve ser consistente (todos os membros que permanecem ativos devem receber os multicasts pendentes),

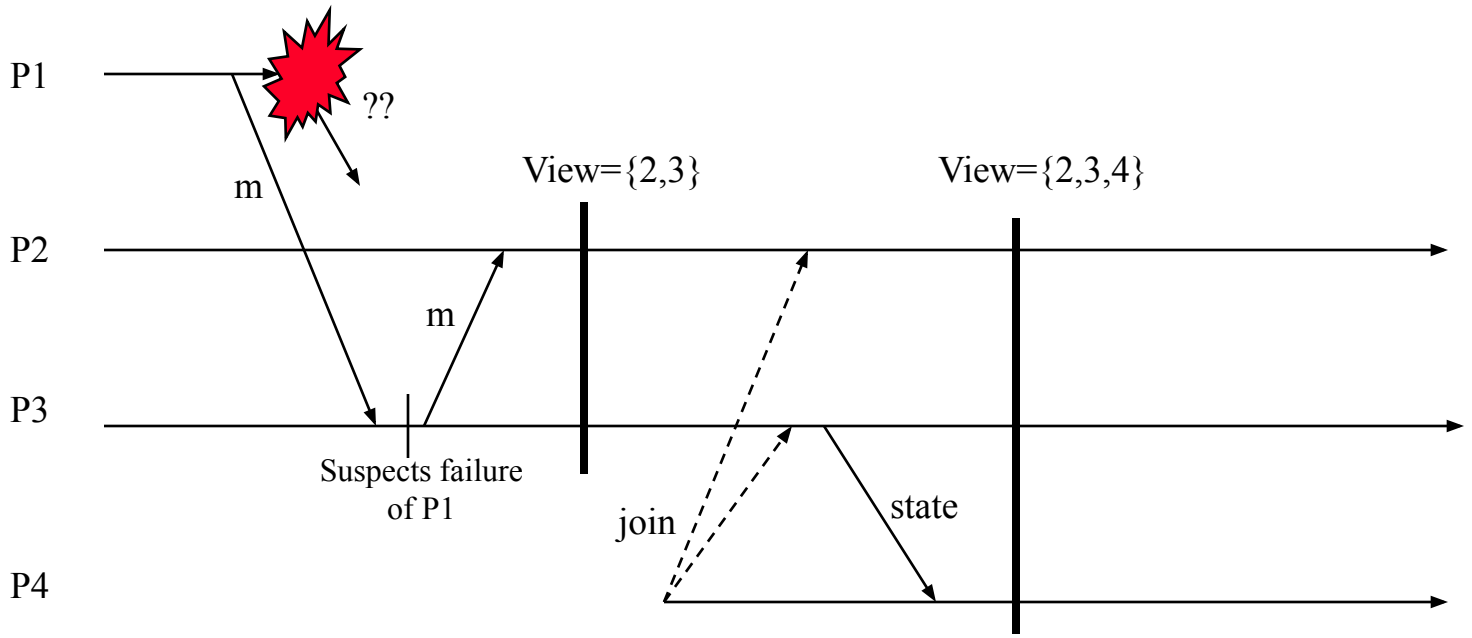
- os novos membros devem receber o estado dos membros antigos

d) **Sincronia Virtual (View-synchonous ou virtual synchony)**:

- Apenas os processos não-falhos estabelecem pontos de sincronização virtual (geralmente eliminando processos faltosos)
- Garante-se a entrega de todas as mensagens até a sincronização das visões

Comunicação de Grupo

Exemplo de uma entrega de multicasts em grupos dinâmicos.



Manutenção da Consistência

Para manter a consistência entre as réplicas, possivelmente também é necessário impor certa política de entrega de mensagens para a aplicação.

Idealmente, a entrega de um multicast **m** deveria ocorrer instantanemente (no momento do seu envio) para todas réplicas. Mas devido à ausência de relógios perfeitamente sincronizados e de diferentes tempos de transmissão, duas mensagens podem chegar em uma ordem diferente em duas réplicas

As possíves políticas de entrega de mensagens **m1** e **m2** multicast são:

FIFO: se m1 → m2 tiverem sido geradas pelo mesma fonte, então m1 será entregue antes de m2 em qualquer réplica

Causal: se m1 → m2 tiverem sido geradas em qualquer processo, então m1 será entregue antes de m2

Total: para qualquer m1 e m2, ou todas as réplicas recebem m1 antes de m2 ou vice-versa

Sincronia Virtual: se m tiver sido recebida por um membro ativo na troca de visão, então m será recebido por todos membros que participaram da troca de visão

Duas formas de Implementação

Grupo com um coordenador (sequenciador): *Ex: Amoeba*

- sequenciador recebe todas as requisições, atribui um número de sequência e re-transmite a mensagem para as demais réplicas
- todos membros confirmam o recebimento
- sequenciador eventualmente retransmite mensagens não confirmadas
- perda de mensagem pode ser detectada devido ausência de ACK ou envio de negACK

(+) ordem total pode ser obtida de forma simples

(+) falha (crash) do sequenciador requer uma eleição de um novo sequenciador e consulta ao log das retransmissões pendentes

Grupo homogêneo (Peer-to-Peer): *Ex: ISIS,Horus*

- qualquer membro que receber a requisição inicia o protocolo (*iniciador*)
- todos os membros ativos confirmam o recebimento, ou acusam perda
- qualquer membro pode retransmitir mensagens (ou o estado, p/ novos membros)

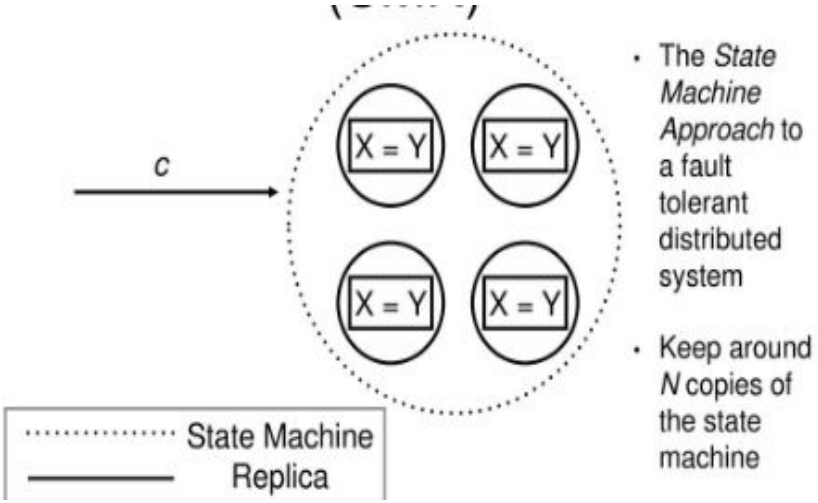
(-) protocolo é mais complexo e requer logs em todos os membros

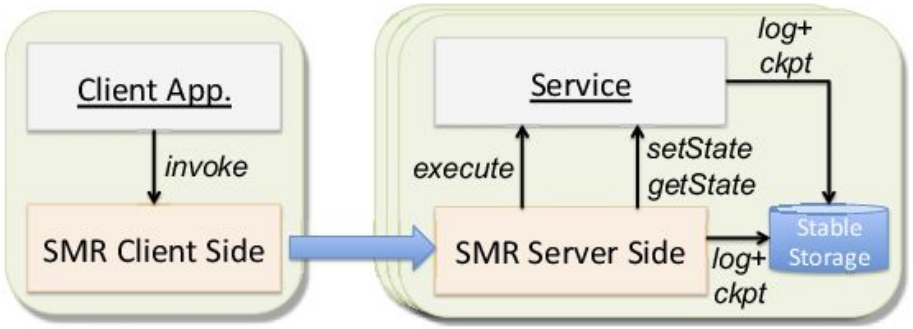
(-) ordem total é mais difícil de ser implementada

(+) não requer a eleição de um coordenador

Máquina de Estados Replicadas

- Objetivo: Manter N réplicas de processos com seus estados locais perfeitamente sincronizados
- Comunicação de grupo em total order para todas as réplicas
- Cada réplica recebe a mensagem de update a aplica sobre o seu estado local
- O update é determinístico, só depende da mensagem e do estado prévio





Outras Formas de Interação

Serviço de Eventos (Publish-subscribe):

Comunicação assíncrona, indireta de 1 para muitos

Processos publicam eventos tipados para um *Serviço de eventos (SE).* Outros processos podem registrar junto ao *Serviço de eventos* o interêsse em receber eventos de um certo tipo (*assinatura*).

Cada tipo de evento contém uma série de atributos tais como:

[ID do gerador, tipoEvento, parâmetros, timestamp]

O matching entre eventos/assinaturas é feito no SE e pode ser sintático (pelo tipo) ou por conteúdo, simples ou complexo.

 Paradigma adequado para redes dinâmicas e aplicações/serviços adaptativos e reativos.

Principais vantagens:

(+) Sistemas heterogêneos/abertos: processos não precisam conhecer os demais processos

(+) Desacoplamento total entre os processos que interagem

(+) Possibilidade de definição de eventos complexos

Exemplos:

- Event service de CORBA
- Jini (Sun)

Chamada Remota de Procedimentos/Métodos

O RPC foi proposto em 1985 por Birrell e Nelson[1]

A ideia principal:

O RPC permite ao programador chamar um procedimento, que é executado remotamente, como se fôsse um procedimento local.

Ao contrário do envio de mensagens, que segue o "paradigma de E/S" o RPC é uma interação síncrona (Request-Reply). É um mecanismo baseado no paradigma da chamada de procedimentos.

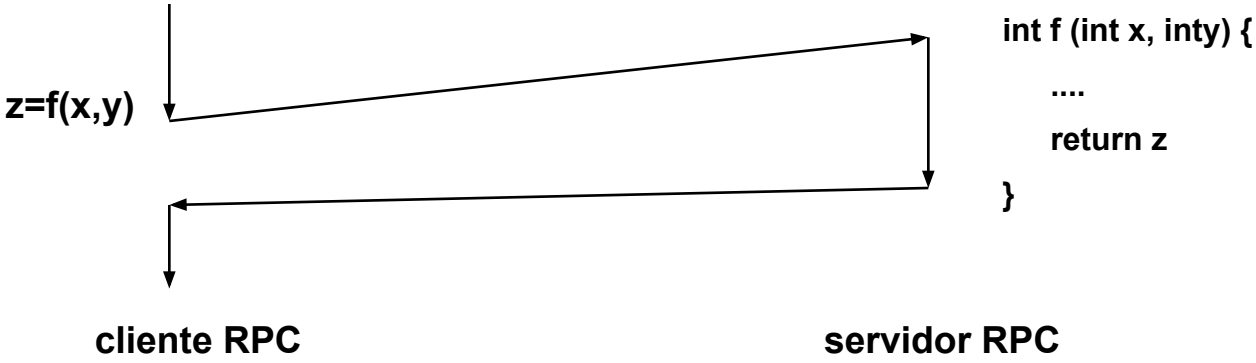
- permite uma melhor estruturação do programa distribuído
- faz um programa distribuído parecer um programa sequencial
- esconde vários detalhes relativos ao protocolo de comunicação sendo usado
- pode ser usado com diferentes protocolos e primitivas de comunicação

Atualmente, S.O.s e ambientes de programação disponibilizam o RPC como principal mecanismo de comunicação.

Exemplos: S.O. Amoeba, Java/RMI, CORBA,

(1) Birrel,A.D; Nelson B.J: Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, vol 2, no. 1, 1984.

A Ideia Central do RPC



Vantagens:

- dados são passados na forma de argumentos e resultados da função
- aparentemente não há diferenas entre chamada local e RPC

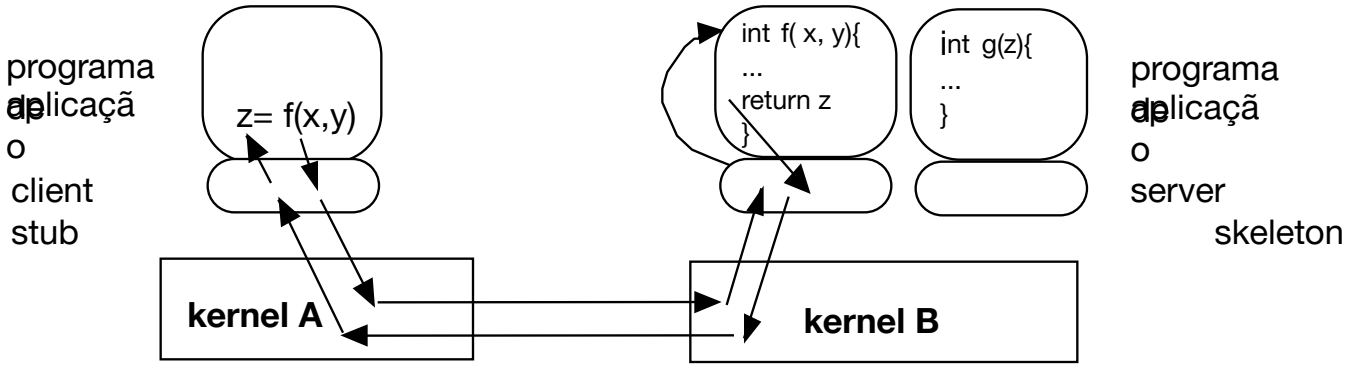
Questões:

- como o cliente fica conhecendo as funções remotas disponíveis?
- como o cliente escolhe o servidor quando há muitos servidores para função f ?
- quem empacota & desempacota argumentos e resultados de mensagens, e lida com os detalhes do protocolo de comunicação?
- como são tratados argumentos passados por referência?
- o que ocorre quando há perda de mensagens ou quebra do servidor?

Princípio do Funcionamento

O funcionamento do RPC é baseado em mecanismos similares aos usados na interação entre processo e kernel.

- programa de aplicação é ligado à procedimentos de interface (stubs)
- no lado servidor, a chegada de uma chamada remota causa a execução da função f como se esta fôsse um "interrupt handler"



cliente RPC

- aloca buffer
- empacota argumentos;
- pede envio mensagem;
- recebe resultado;
- desempacota resultado;
- retorna ao programa de aplicação

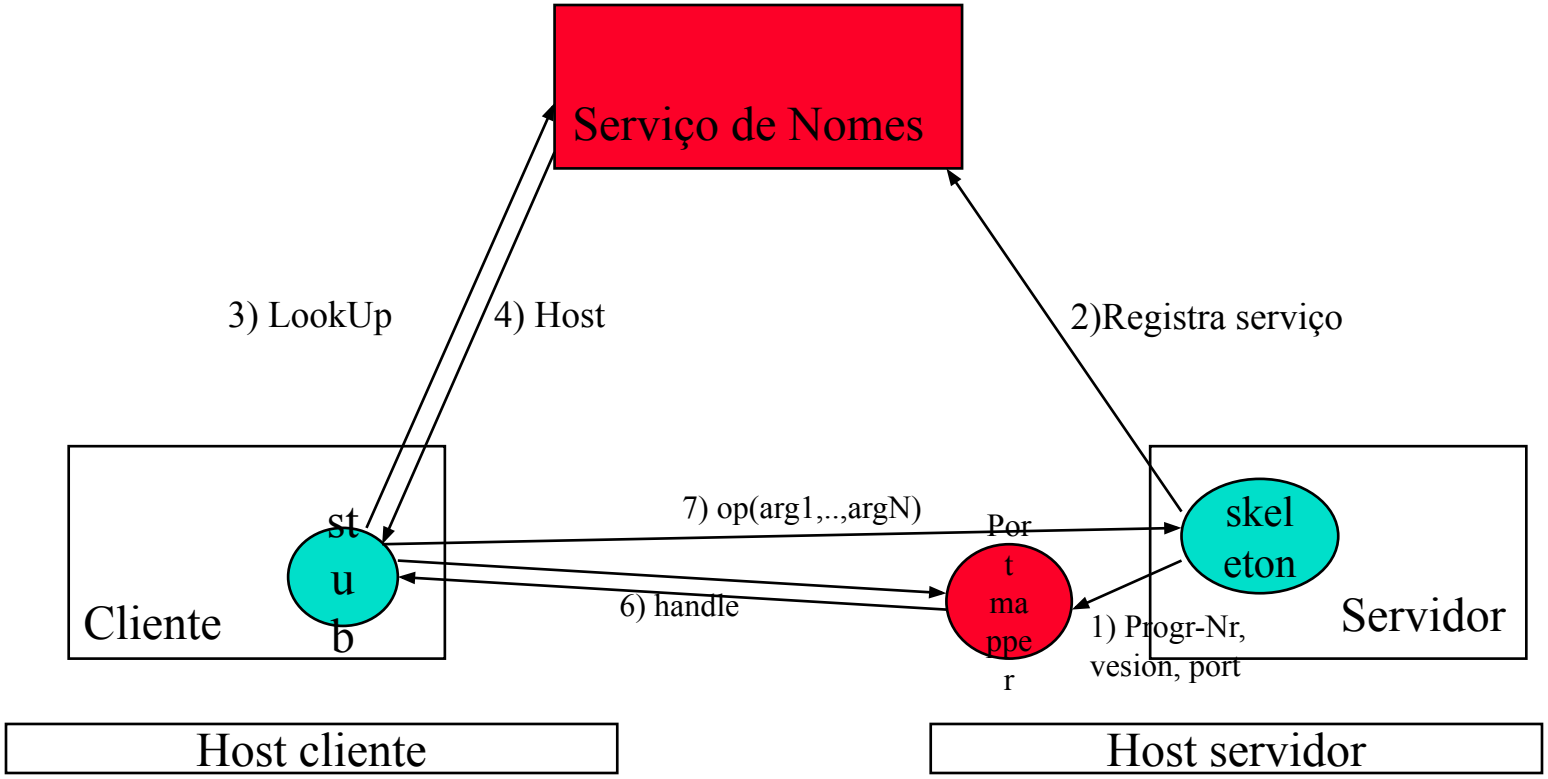
servidores RPC

loop_forever {

- recebe mensagem;
- desempacota argumentos;
- identifica e chama a função;
- empacota resultado
- envia mensagem

}

Binding entre Cliente e Servidor



Outras Formas de Interação

Espaço de Tuplas (Tuple Spaces)

Permite uma comunicação assíncrona, indireta e associativa e com persistência.

Todos os processos interagem através de um espaço de dados virtualmente compartilhado (o espaço de tuplas - ET).

Tupla = sequência de um ou vários tipos de dados, Exemplo: <“endler”, 1961>

Um ET pode conter tuplas dos mais diversos tipos.

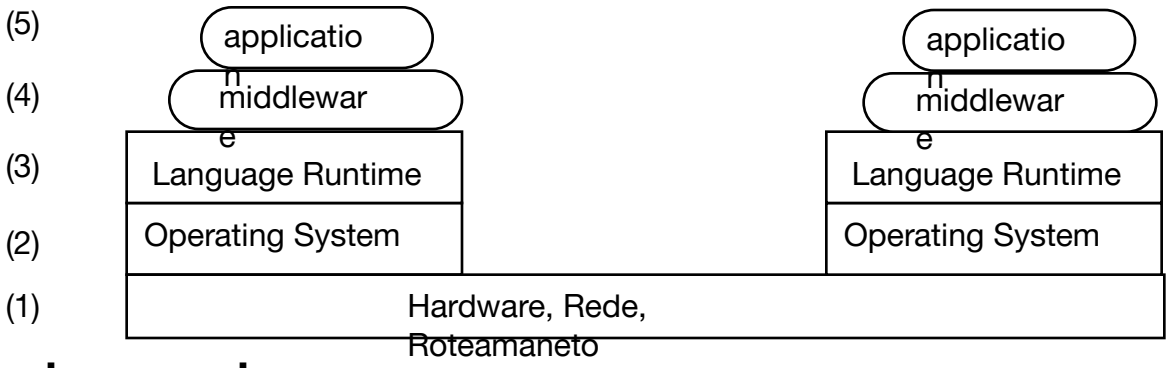
As primitivas:

- in (tupla) :: *adiciona uma tupla ao ET*
- out (padrão) :: *remove do ET qualquer tupla que satisfaça o padrão*
- read (padrão) :: *verifica se existe no ET uma tupla satisfazendo o padrão*

Exemplo de padrão: <string, 1961>

Exemplo de linguagens/ambientes: Linda, JavaSpaces, Tspaces, Agora

Modelo de Camadas



Função de cada camada:

(1) trata da transmissão de bits; controle de fluxo; integridade; roteamento

(2) comunicação fonte-destino; de um stream de bytes (IP, porta), Abstração: sockets

(3) modela interação entre elementos da linguagem de programação; como p.ex. processos ou objetos.

- implementa primitivas mais sofisticadas, mensagem estruturada, portas e referências remotas, e verificação de compatibilidade de tipos
- primitivas de criação (e mapeamento) de processos/threads em nós da rede
- torna heterogeneidade de SO transparente

(4) implementa serviços comuns a classes de aplicações: p.ex.: detecção de deadlocks, coordenação, eleição, comunicação de grupo confiável, suspeita de falha, transações atômicas.

(5) implementa a funcionalidade da aplicação e os problemas de distribuição/replicação ficam transparentes

Modelo Formal de Execução e Estado Distribuído

- Systema distribuído = conjunto de processos (P_1,P_2,..,P_N) + subsistema de comunicação
- Cada processo P_i é modelado como um sistema de transição, onde uma configuração do processo é denominada **estado**, e uma transição é um **evento**.
- Cada processo pode executar três tipos de eventos:
  - Evento interno
  - Evento de envio
  - Evento de recebimento

O sistema de transição de um SD é:

- Um conjunto de configurações, onde cada configuração consiste de:
  - estado de cada processo,e estado da rede, i.e. o conjunto de mensagens em trânsito
- Uma transição é um evento em um dos processos. Se for um evento de comunicação, modifica tb o estado da rede

Modelo Formal de Execução e Estado Distribuído

O *Estado global* de um SD com processos (P_1,P_2,..,P_N) e conjunto de canais unidirecionais C pode ser descrito como G=(S,L) onde:

S = {s_1,s_2,...s_N} e onde s_i é o estado do processo P_i (atribuição às variáveis do programa e o atual ponto de controle)

L_{ij} = {m_1,m_2,..,m_K} com i,j ∈ [1..M], conteúdo do canal C_{ij} ∈ C e m_i ∈ M (mensagens do programa)

Estado de um canal de comunicação é a sequência de mensagens no buffer (mensagens ainda em trânstito)

Ref: Chow & Johnson: seção 9.4

Modelo Formal de Execução e Estado Distribuído

O sistema começa com estado inicial $G^o = (S^o, L^o)$, onde $L_{ij}^o = \varnothing$ para cada C_{ij} (canais de comunicação vazios)

O sistema muda de estado através de eventos (executados no processo P_p) que:

- transformam o estado de s_p para s'_p e
- mudam o estado de no máximo um canal (entrada ou saida de P_p)
- Cada evento **e** no sistema é uma tupla:

$$e = (p, s, s', m, c)$$

onde p é o processo, $s, s' \in S$, $m \in M \cup \text{null}$ e $c \in C \cup \text{null}$

Obs: se m=null e c=null então **e** é evento interno**.**

- Os eventos ocorrem segundo o algoritmo sendo executado e o algoritmo é definido por todos os possíveis eventos.

Sistemas de Transição

Modelo natural de um sistema distribuído é um sistema de transição (C, →, I)

- C :: Um conjunto de configurações
- → :: uma relação de transição binária, γ → δ é uma transição de conf γ para conf δ
- I :: subcojunto de C (de configurações iniciais)
- Uma execução E de S é uma sequencia maximal (γ0, γ1, γ2,...) onde γ0 ∈ I, e ∀ i≥0, γi → γi+1

Sistemas de Transição

- Configuração terminal: é uma configuração para a qual não existe δ tal que $\gamma \rightarrow \delta$
- Configuração δ é alcançável a partir de γ: ($\gamma \Rightarrow \delta$) sse existe uma sequência $\gamma = \gamma 0, \gamma 1, \gamma 2, ..., \gamma k = \delta$, tal que $0 \leq i < k$, $\gamma i \rightarrow \gamma i+1$
- Configuração δ é alcançável: sse é alcançável de uma configuração incial

Algoritmos Distribuídos: Considerações Gerais

Além da corretude (p.ex. segurança, progresso, *fairness*, etc.), as seguintes propriedades são desejáveis:

- Simetria de papéis: todos os processos executam a mesma funcão(eventualmente com dados “particulares”, p.ex. PID)
- Robustez às falhas mais comuns: o quanto mais falhas realistas o “modelo de falhas” prever, melhor
- Genericidade: menor dependência possível das características da rede

  Por exemplo: Topologia arbitrária, Sistema assíncrono, ou comunicação não confiável
- Baixa complexidade de comunicação e espaço: menor possível número de trocas de mensagens, e/ou tamanho das mensagens
- Baixa complexidade de tempo: menor número de rodadas (para algorítmos síncronos), ou convergência rápida
- Simplicidade: quanto mais complexo for o algoritmo, maior é a dificuldade de:
  - Implementação e depuração
  - teste para todos os casos especiais
  - Entendimento e manutenção (p.ex. fazer extensões)

Algoritmos Distribuídos: Considerações Gerais

As premissas mais frequentes sobre o Modelo

- processos não falham ou têm apenas falhas fail-stop
- cada processo P tem conjuntos estáticos e bem definidos de processos dos quais recebe mensagens (IN_P) e para os quais envia mensagens (OUT_P) □ a topologia de interconexão é fornecida a priori e é fixa
- Canal de comunicação é seguro: não há duplicação, geração espontânea ou modificação das mensagens
- Canal de comunicação é FIFO: ordem de entrega igual à ordem de envio
    - Canal de comunicação é confiável: mensagens nunca se perdem
- tempo máximo de transmissão de mensagens é fixo e conhecido

Obs:

- Algoritmos determinísticos p/ modelos assíncronos geralmente não são robustos a qq tipo de falha
- Algoritmos deterministicos para modelos síncronos são capazes de tolerar falhas não triviais (bizantinas)
- Algoritmos auto-estabilizantes em modelos assíncronos: toleram falhas e possivelmente converge para o estado terminal

O Ambiente DAJ

DAJ -- A Toolkit for the Simulation of Distributed Algorithms in Java

- Desenvolvido por Wolfgang Schreiner (Univ. Linz)
- DAJ = biblioteca de classes Java (DAJ 1.0.2 para JDK 1.1)
- permite o desenvolvimento/visualização de algoritmos distribuídos
- modelo de interação é o envio de mensagens
- Programas podem ser executados como aplicações Java stand-alone ou como applets
- É um ambiente de desenvolvimento para pesquisa e ensino, disponível em http://www.risc.uni-linz.ac.at/software/daj/
- Contém vários exemplos de algoritmos

Manutenção da Consistência

Aplicações podem ter diferentes requisitos com relação ao grau de consistência (modelo de consitência) dos membros

consistência forte: é necessário que o estado das réplicas esteja sempre perfeitamente sincronizado (esquema replicação ativa): o grupo funciona como um único processo.

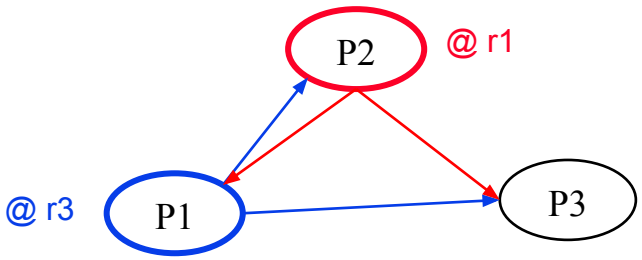
Sub-categorização:

- Consistência não uniforme = *para todos os processos corretos, a decisão precisa ser a mesma (p.ex. Entregar a mensagem)*
- Consistência uniforme = *se qualquer um dos membros toma uma decisão (e possivelmente falha logo após), então todos os demais membros precisam tomar a mesma decisão*.

  Esta última é necessária quando há eleitos colaterais (p.ex. Banco de dados replicado)

consistência fraca: basta que o estado das réplicas esteja quase sempre sincronizado (esquema primary- backup)

Tarefa “Esquenta” usando o Sinalgo



- Implementar o Relógio Lógico de Lamport para N processos estáticos (e todos com alcance de comunicação mútua),
- Cada processo (Pi) aleatoriamente gera uma mensagem m em uma rodada r, com a probabilidade Prob.
- Mostrar que toda entrega de mensagens entre um par Pi e Pj é FIFO
- Mostrar que:
  - Seja mensagem m_r emitida na rodada r e outra em m_q em rodada q, onde r < q, então $LT(m_r) < LT(m_q)$ e
  - que nada pode se dizer de sobre o LT de mensagens geradas na mesma rodada
- Discutir o que precisa ser feito para se gerar uma ordem total de todos os eventos.