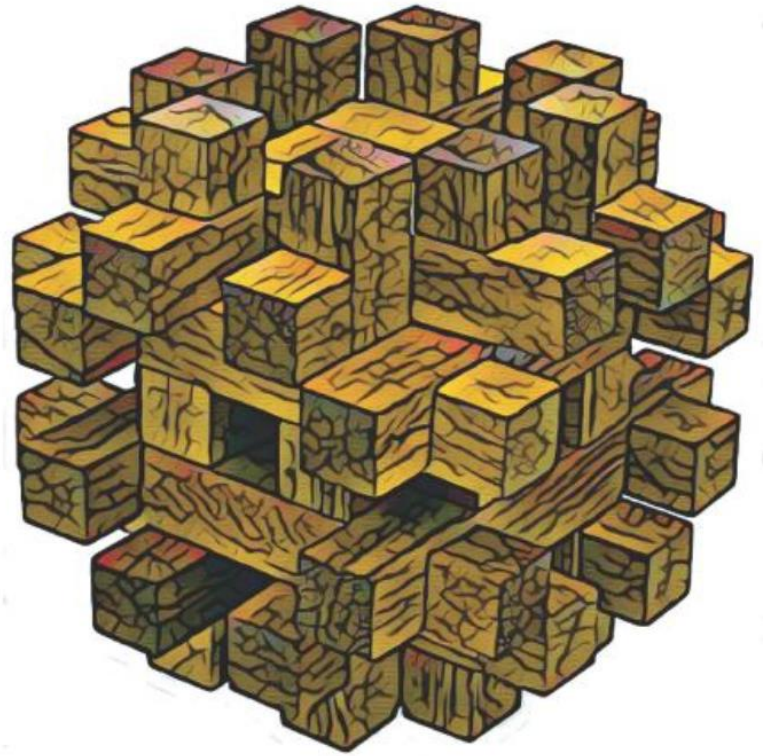
Distributed Systems

Maarten Van Steen & Andrew S. Tanenbaum



3th Edition – Version 3.03 - 2020

Capítulo 3

Processos

Quinta-Feira, 07 de Julho de 2022

INTRODUÇÃO AOS THREADS

IDÉIA BÁSICA

Construção de processadores virtuais em software, no topo de processadores físicos:

- Processador: Provê um conjunto de instruções junto com a capacidade de executar automaticamente uma série destas instruções.
- Thread: Um processador de software mínimo em cujo contexto uma série de instruções podem ser executadas. Salvar o contexto de uma thread implica em parar a execução atual e salvar todos os dados em uso, de forma que seja possível retornar ao mesmo estágio em um momento seguinte.
- Processo: Um processador de software em cujo contexto um ou mais threads podem ser executadas. Executar uma thread significa executar uma série de instruções em um contexto daquela thread.

CHAVEAMENTO DE CONTEXTO

CONTEXTOS

- Contexto de Processador: Uma coleção mínima de valores armazenados nos registradores de um processador usados para execução de uma série de instruções (p.e., stack pointer, endereço de registradores, contador de programa, etc).
- Contexto de Thread : Uma coleção de valores armazenados em registradores e memória, usados na execução de uma série de instruções (isto é, context de processador, estados).
- Contexto de Processo: Uma coleção minima de valores armazenados em registradores e memória, usados para execução de uma thread (isto é., contexto de thread, mas também pelo menos os valores dos registradores da MMU).

CHAVEAMENTO DE CONTEXTO

OBSERVAÇÕES

1. Threads dividem o mesmo espaço de endereçamento. O chaveamento de contexto de Thread pode ser feito de forma totalmente independente do sistema operacional.
2. Chavemento de processo é geralmente (de alguma forma) mais custoso pois envolve o SO, trapping para o kernel ...
3. A criação e destruição de uma thread é muito menos custosa do que fazer o mesmo para um processo.

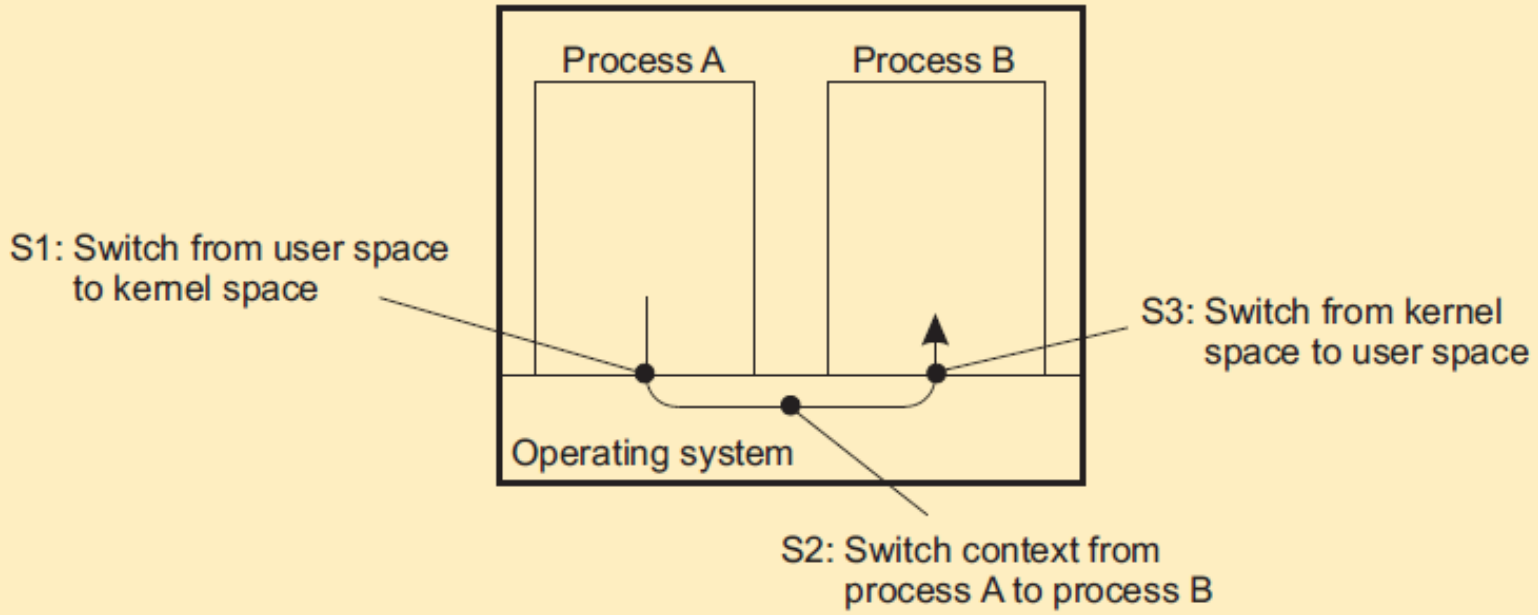
POR QUE USAR THREADS

ALGUMAS RAZÕES SIMPLES

- Evitar bloqueamentos desnecessários: um processo single-threaded vai bloquear quando estiver fazendo E/S; em um processo multi-threaded, o Sistema Operacional pode chavear a CPU para outro thread no processo.
- Explorar paralelismo: os threads de um processo multi-threaded pode ser escalonado para rodar em paralelo em um multi-processador ou processador multi-core.
- Evitar chaveamento de processo: estruture a aplicação não como uma coleção de processos, mas através de multiplos threads.

EVITAR CHAVEAMENTO DE PROCESSOS

EVITAR TROCA DE CONTEXTO CUSTOSA ..



TRADE-OFFS (compromissos)

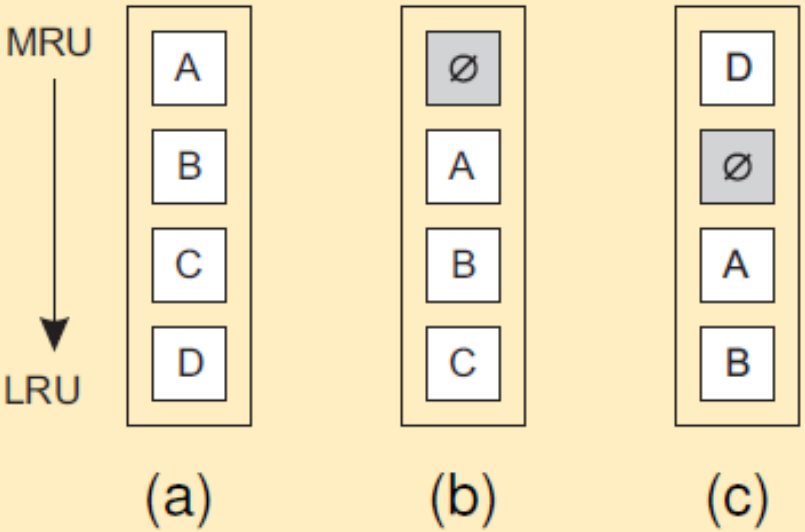
- Threads usam o mesmo espaço de endereçamento: mais propenso a erros
- Nenhum suporte do SO/HW para previnir que as threads usem memória uma das outras
- Chaveamento de contexto de thread pode ser mais rápido que o chaveamento de contexto de processo.

CUSTO TROCA DE CONTEXTO

CONSIDERE UM HANDLER SIMPLES MANUSEADOR DE INTERRUPÇÃO DE RELÓGIO

- Custos diretos: chaveamento em si e a execução do código do handler
- Custos indiretos: outros custos, notadamente causados pela bagunça na cache

O QUE UMA TROCA DE CONTEXTO PODE CAUSAR: CUSTOS INDIRETOS



a) Antes do chaveamento de contexto

b) Depois do chaveamento de contexto

c) Depois de acessar o bloco D

THREADS E SISTEMAS OPERACIONAIS

QUESTÃO PRINCIPAL

- Deveria o kernel de SO prover suporte a threads, ou este suporte deve ser implementado em pacotes de nível usuário?

O QUE UMA TROCA DE CONTEXTO PODE CAUSAR: CUSTOS INDIRETOS

- Todos processos podem ser completamente manuseados dentro de um único processo → implementações podem ser extremamente eficientes.
- Todos serviços providos pelo kernel são feitos em nome do processo no qual a thread reside → se um kernel decide bloquear uma thread, então o processo todo será bloqueado.
- Threads são usadas quando existe muitos eventos externos: threads bloqueiam com base em eventos → se um kernel não pode distinguir threads, como ele pode suportar a sinalização de eventos para eles ?

THREADS E SISTEMAS OPERACIONAIS

SOLUÇÃO NO KERNEL

A ideia completa é implementar o pacote de suporte a threads dentro do kernel. Isto significa que todas operações retornam como chamadas de sistema:

- Operações que bloqueiam uma thread não são mais um problema: o kernel escalona outra thread disponível dentro do mesmo processo.
- O manuseio de eventos externos é simples: o kernel (que pega todos eventos) escalona uma thread associada com o evento.
- O problema é (ou era) a falta de eficiência devido ao fato que cada operação na thread requer um trap para o kernel.

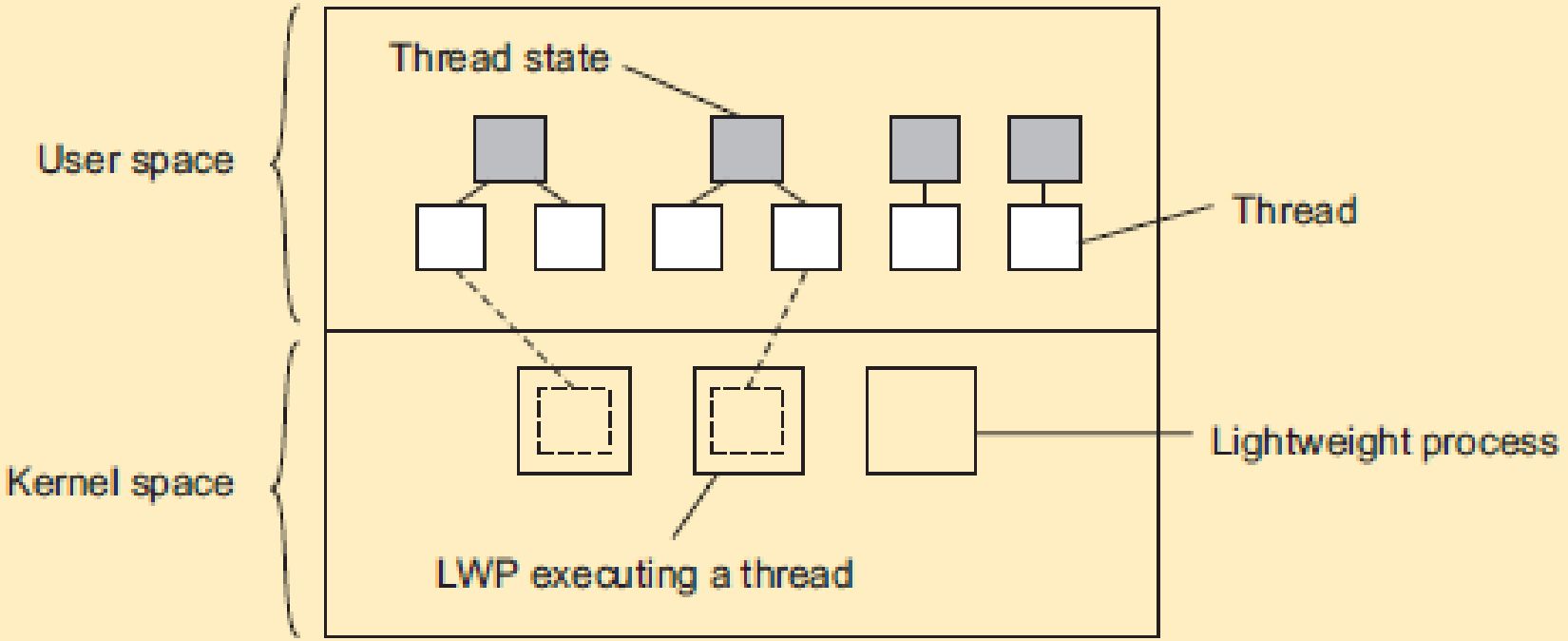
CONCLUSÃO – MAS ..

- Tente uma mistura de threads nível usuário e nível kernel em um mesmo conceito, contanto que o ganho de desempenho não implique em aumento da complexidade.

PROCESSOS LIGHTWEIGHT

IDEIA BÁSICA

Introduzir uma abordagem de threads de dois níveis: lightweight processes que podem executar threads nível usuário.



PROCESSOS LIGHTWEIGHT

PRINCÍPIO OPERACIONAL

- Uma thread nível usuário faz uma system call → o LWP que está executando esta thread, bloqueia. A thread se mantém presa (bound) ao LWP.
- O kernel pode escalonar outro LWP mesmo tendo uma thread presa a ele. Nota: esta thread pode chavear para qualquer outra thread rodável que esteja no espaço de usuário.
- Se uma thread realiza uma chamada de operação bloqueante em nível usuário → ocorre chaveamento de contexto para outra thread rodável, (presa ao mesmo LWP).
- Quando não há threads para serem escalonadas, o LWP pode permanecer desocupado, e pode até ser removido (destruído) pelo kernel.

NOTA

- Este conceito foi virtulamente abandonado – é somente thread nível usuário ou nível kernel.

USANDO THREADS LADO CLIENTE

MULTITHREADED WEB CLIENT

Escondendo latências da rede:

- Web browsers escaneiam páginas HTML no recebimento, e descobre que mais páginas (files) são necessárias de serem buscadas.
- Cada arquivo é buscado por uma thread separada, cada uma realizando uma requisição HTTP.
- Conforme os arquivos chegam, o browser vai mostrando (displays) eles.

MULTIPLAS CHAMADAS REQUISIÇÕES-RESPOSTA PARA OUTRAS MÁQUINAS (RPC)

- Um cliente realiza várias chamadas ao mesmo tempo, cada um por uma thread diferente.
- Ele então espera até que os resultados sejam retornados.
- Nota: se chamadas são feitas a diferentes servidores, Podemos ter aceleração linear.

CLIENTES MULTITHREADED – AJUDA ?

PARALELISMO EM NÍVEL THREAD: TLP

Se *ci* denota uma fração de tempo para que exatamente *i* threads estejam sendo executadas simultâneamente.

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

com *N* sendo o máximo número de threads que (podem) executar ao mesmo tempo

MEDIDAS NA PRÁTICA

Um Web browser típico tem um valor entre 1.5 e 2.5 → threads são primariamente usadas para a organização lógica de browsers.

USANDO THREADS LADO SERVIDOR

MELHORA DE DESEMPENHO

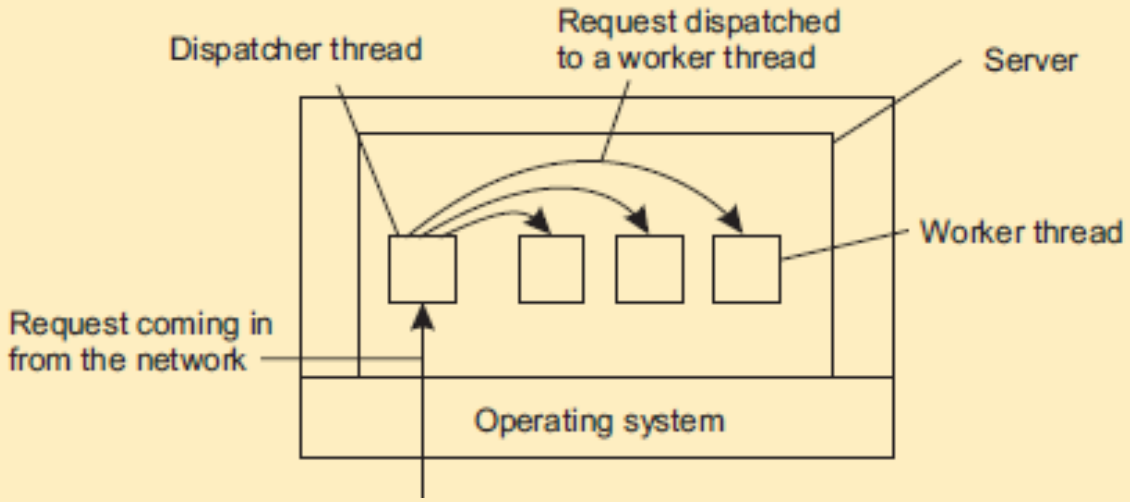
- Iniciar uma thread é mais custoso que iniciar um novo processo.
- Ter um servidor single-threaded proíbe o escalamento para um sistema multiprocessor.
- Como nos clientes: esconda a latência da rede reagindo a próxima requisição enquanto a anterior está sendo atendida.

MELHOR ESTRUTURA

- Maioria dos servidores tem altas demandas de E/S. Usando simples, chamadas bloqueantes bem entendidas simplifica a estrutura toda.
- Programas multi-thread tendem a ser menores, mais simples de entender devido a fluxo de controle simplificado.

POR QUE MULTITHREAD É POPULAR: ORGANIZAÇÃO

MODELO DISPATCHER / WORKER



VISÃO GERAL

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

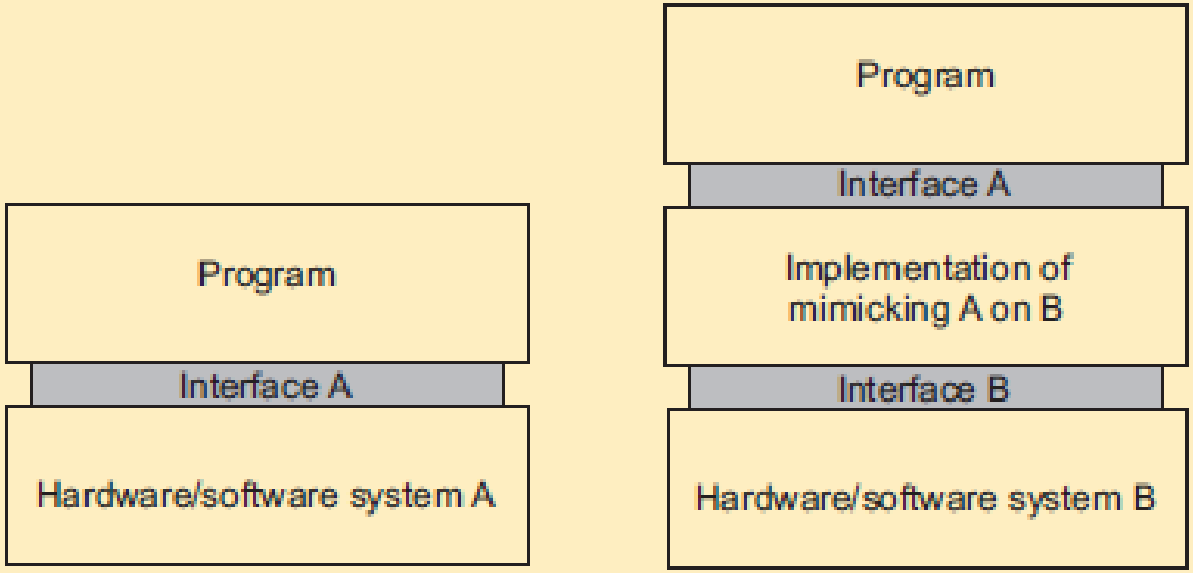
VIRTUALIZAÇÃO

OBSERVAÇÃO

Virtualization é importante:

- Hardware muda mais rápido que software
- Facilita a portabilidade e a migração de código
- Isolamento de components em falha ou atacados

PRINCÍPIO: IMITANDO INTERFACES



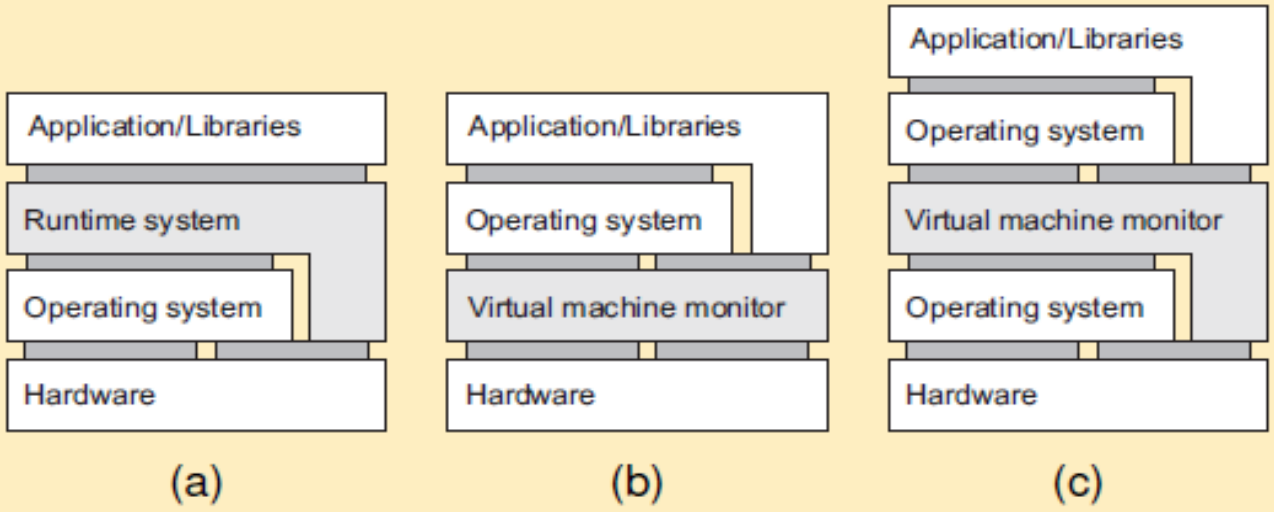
IMITANDO INTERFACES

4 TIPOS DE INTERFACES EM 3 DIFERENTES NÍVEIS

1. Instruction set architecture (i): o conjunto de instruções de máquina, com dois subconjuntos:
   - Instruções privilegiadas: permitidas para serem executadas somente pelo SO.
   - Instruções Gerais: podem ser executadas por qualquer programa.
2. System calls (ii) são oferecidas por um SO.
3. Library calls (iii), conhecidas como API (application programming interface (iv))

FORMAS DE VIRTUALIZAÇÃO

(a) PROCESSO VM, (b) VMM NATIVO, (c) VMM HOSTED



DIFERENÇAS

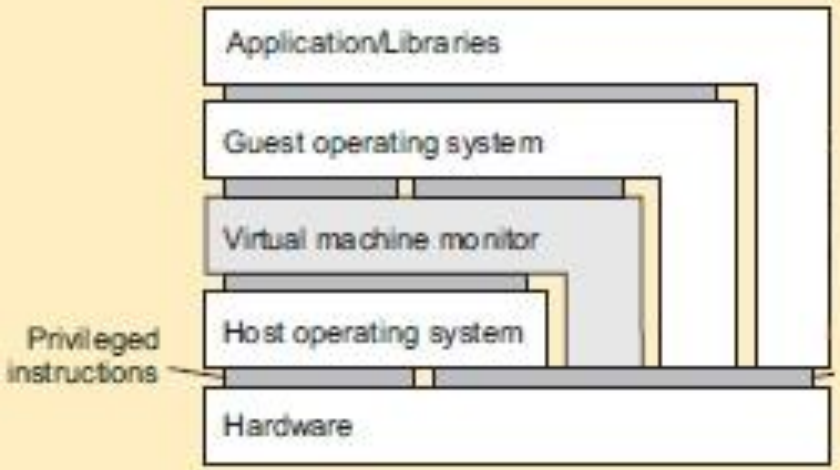
(a) Conjunto separado de instruções, um interpretador / emulador rodando no topo do SO

(b) Instruções de baixo nível (low-level), junto com um SO mínimo (bare-bone)

(c) Instruções de baixo nível, as delegando o trabalho para um SO completo

VMs DESEMPENHO

REFINANDO A ORGANIZAÇÃO



- Instrução privilegiada: se e somente se executada em modo usuário –pode causar um trap para o SO
- Instrução não privilegiada: o restante

INSTRUÇÕES ESPECIAIS

- Instrução sensível a controle: pode afetar a configuração da máquina (p.ex. afetando o registro de relocação ou tabela de interrupção).
- Instrução sensível a comportamento: efeito é parcialmente determinado por context (p.ex. POPF seta uma flag que habilita interrupção, mas somente em modo Sistema)

CONDIÇÃO PARA VIRTUALIZAÇÃO

CONDIÇÃO NECESSÁRIA

- Para muitos Computadores convencionais, um monitor de máquina virtual pode ser construído se um conjunto de instruções sensíveis para aquela máquina é um subconjunto de instruções privilegiadas.

PROBLEMA: condição nem sempre é satisfeita

- Pode existir instruções sensíveis que são executadas em modo usuário sem causar um *trap* para o SO

SOLUÇÕES

- Emular todas instruções
- *Wrap* instruções sensíveis não privilegiadas para desviar o controle para a VMM (*virtual machine monitor*)
- Para-virtualização: modifique o SO impedindo instruções sensíveis não privilegiadas ou tornando elas não sensíveis (i.e. mudando o contexto)

VMs E CLOUD COMPUTING

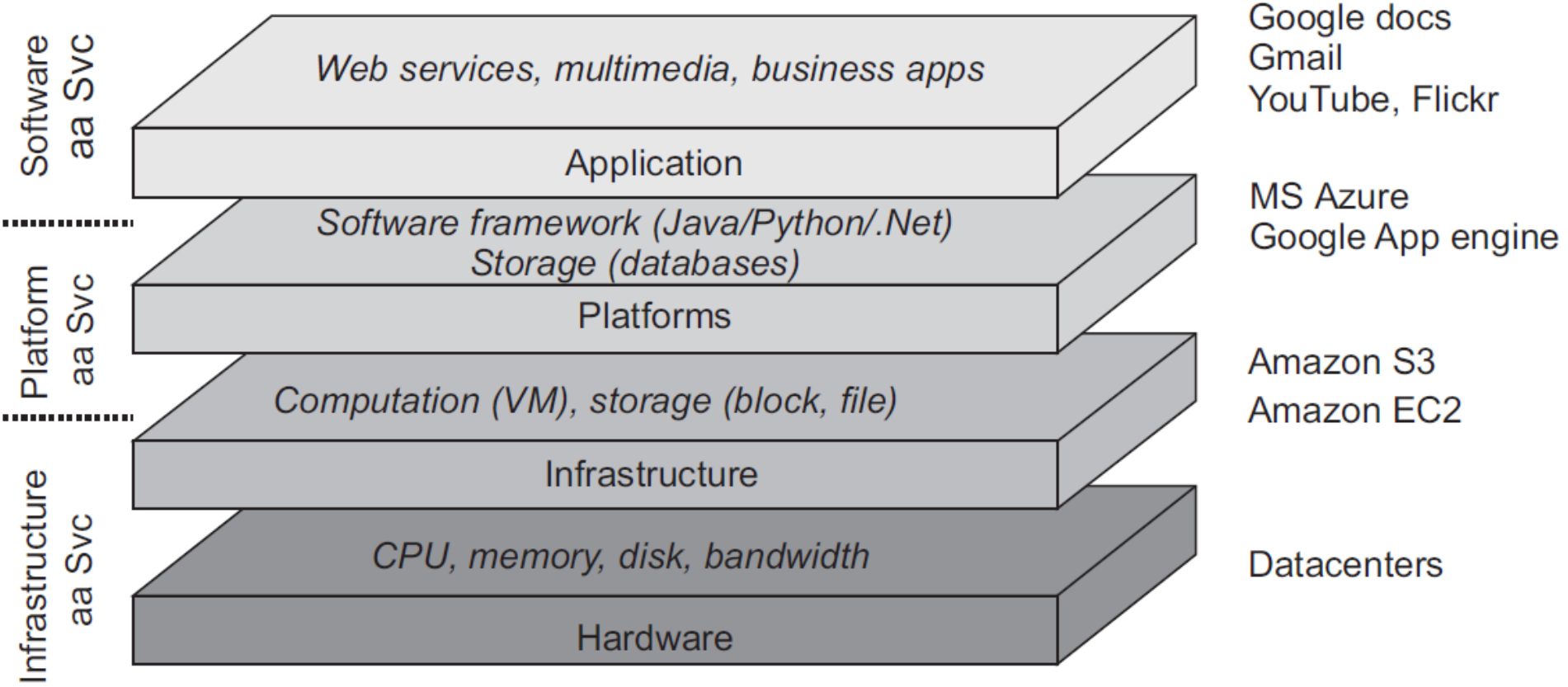
3 TIPOS DE SERVIÇOS EM CLOUD

- Infrastructure-as-a-service: cobrindo a infrestrutura básica
- Platform-as-a-service: cobrindo serviços em nível Sistema
- Software-as-a-service: contendo a aplicação em sí

IaaS

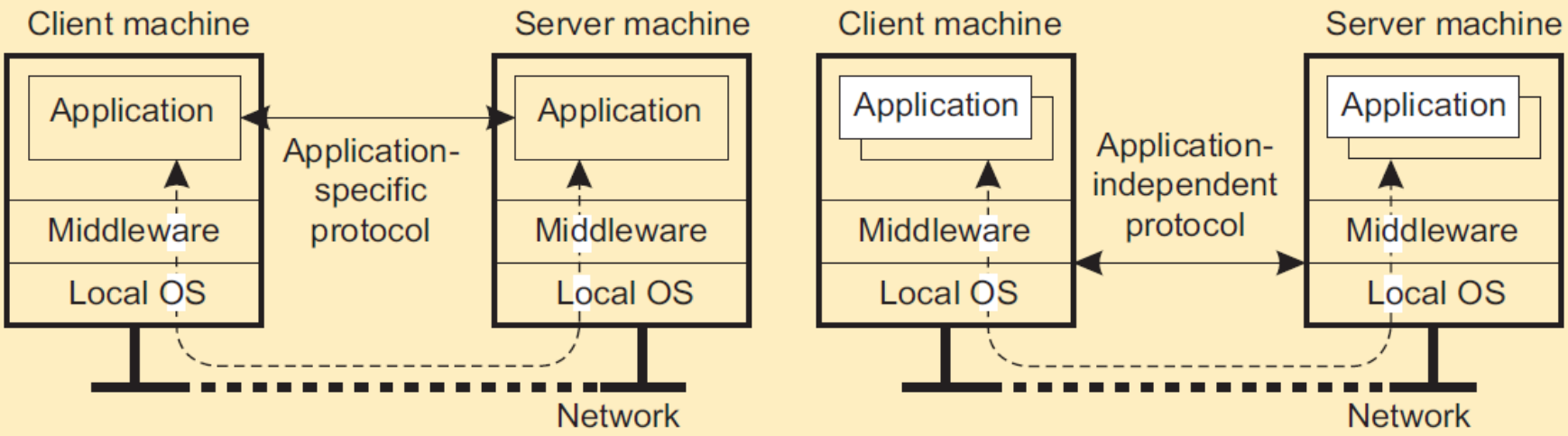
- Ao invés de alugar uma máquina física, um provedor de serviços em núvem vai alugar uma VM (ou VMM) que pode possívelmente estar compartilhando uma máquina física com outros clientes → com quase isolamento total entre clientes (embora o isolamento de desempenho possa nunca ser atingido)

COMPUTAÇÃO EM NÚVEM (CLOUD)



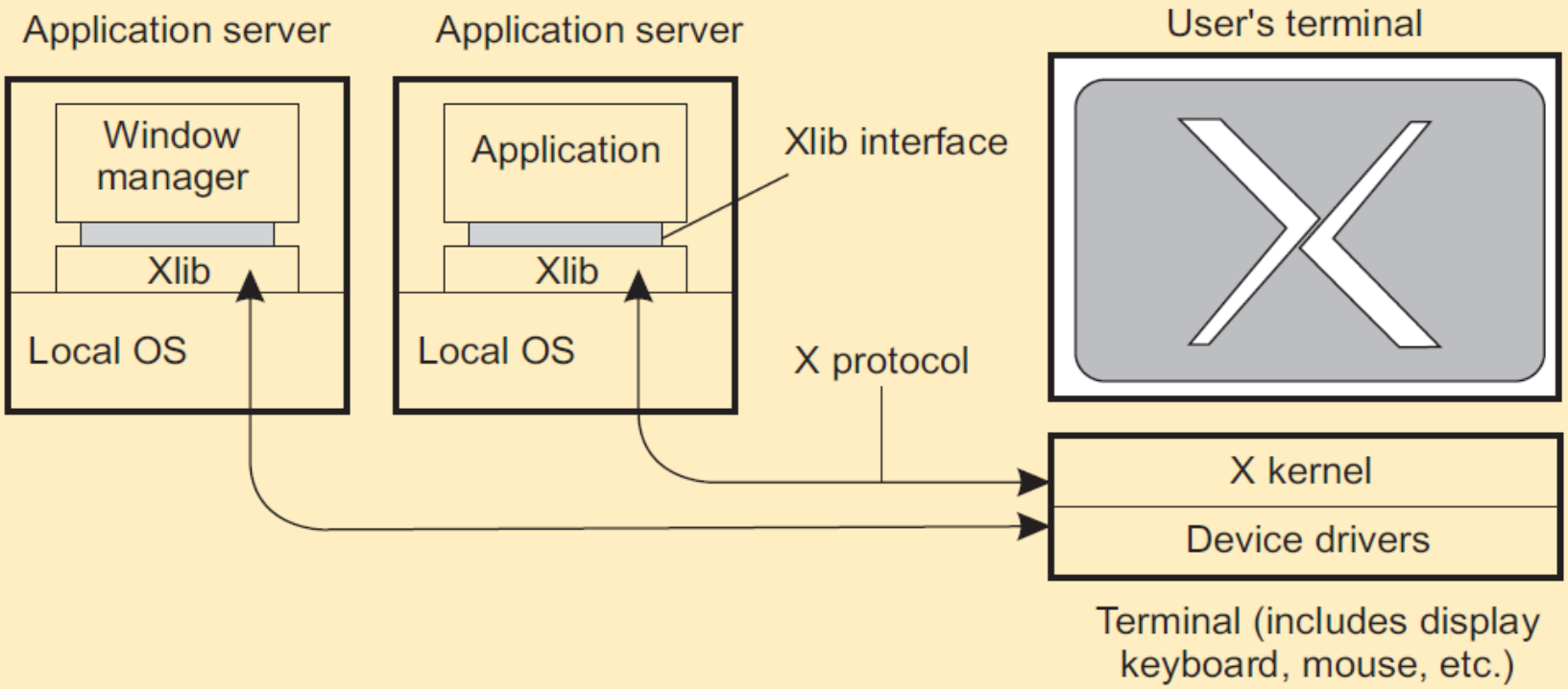
INTERAÇÃO CLIENTE-SERVIDOR

DISTINÇÃO ENTRE SOLUÇÕES EM NÍVEL APLICAÇÃO E EM NÍVEL MIDDLEWARE



EXEMPLO SISTEMA X-WINDOW

ORGANIZAÇÃO BÁSICA



X CLIENTE E SERVIDOR

- A aplicação age como um cliente para o X-kernel, que roda como um servidor na máquina do cliente

MELHORANDO X WINDOW

OBSERVAÇÕES PRÁTICAS

- Existe frequentemente uma separação entre a lógica de controle e os commando na interface de usuário
- Aplicações tendem a operarem sincronismo apertado com o X-kernel

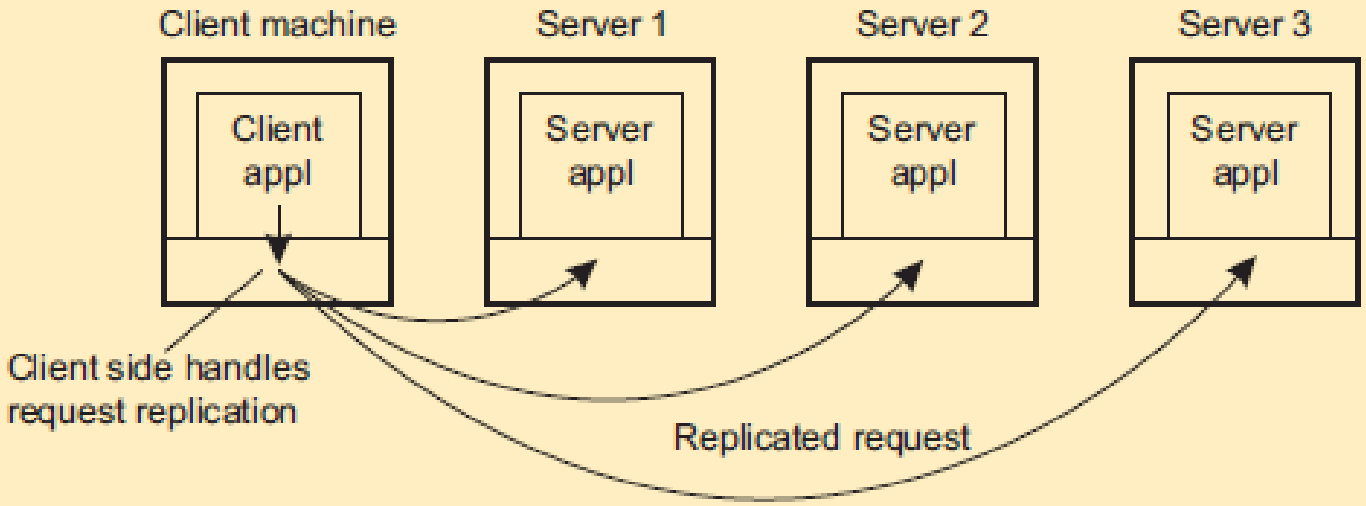
ABORDAGEM ALTERNATIVA

- Deixar as aplicações controlarem o display completamente, até o nível de pixel (p.ex. VNC)
- Prover somente poucas operações de display no alto nível (dependente de drivers de video local), permitindo operações de display mais eficientes

SOFTWARE LADO CLIENTE

GERALMENTE CUSTOMIZADO PARA TRANSPARÊNCIA DISTRIBUÍDA

- Transparência no acesso: stubs no lado cliente para RPCs
- Transparência localidade / migração: deixa o software do lado cliente manter a localização atual
- Transparência replicação: múltiplas invocações manuseadas pelo stub cliente



- Transparência falha: pode frequentemente ser colocado somente do lado cliente (se tentarmos mascarar falhas no servidor e na comunicação)

SERVIDORES ORGANIZAÇÃO GERAL

MODELO BÁSICO

Um processo implementando um serviço específico em nome de uma coleção de clientes. Espera por requisições entrantes vindas de um cliente e subsequentemente garante que a requisição é atendida, e na sequência espera pela próxima requisição entrante.

SERVIDORES CONCORRENTES

DOIS TIPOS BÁSICOS

- Servidor interativo: servidor manipula uma requisição antes de atender a próxima requisição
- Servidor concorrente: usa um dispatcher, que pega uma requisição entrante e então passa para um thread/processo separado atender.

OBSERVAÇÃO

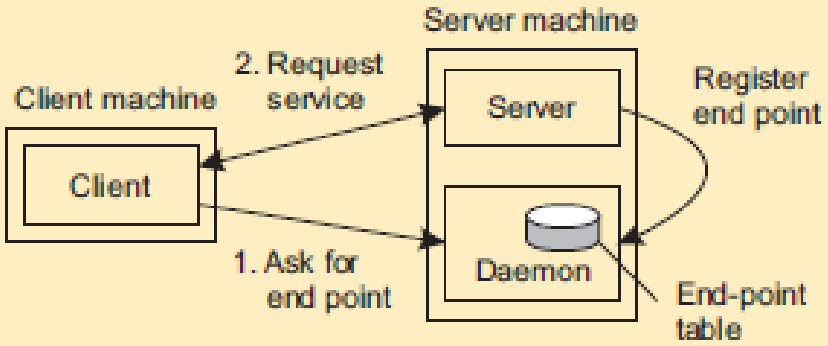
- Servidores concorrentes é a norma: eles podem facilmente manipular múltiplas requisições , notavelmente na presença de operações bloqueantes (disco ou outros servidores)

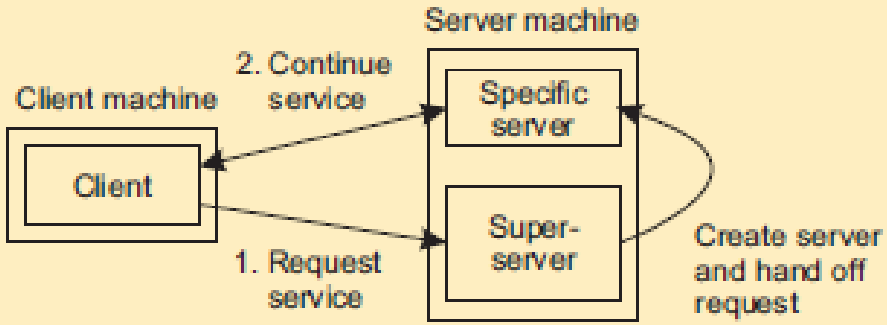
CONTATANDO UM SERVIDOR

OBSERVAÇÃO: A MAIORIA DOS SERVIÇOS SÃO ATRELADOS A UM PORTO ESPECÍFICO

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)

ATRIBUIÇÃO DINÂMICA DE UM "END POINT"





COMUNICAÇÃO OUT-OF-BAND (FORA DE ORDEM)

QUESTÃO

É possível interromper um servidor uma vez que ele tenha aceitado uma requisição de serviço (ou esteja em processo de aceitar)

SOLUÇÃO 1: USE UM PORTO SEPARADO PARA DADOS URGENTES

- Servidor com uma thread/processo separada para mensagens urgentes
- Mensagem urgente → requisição associada é colocada em espera
- Nota: necessário suporte do SO para escalonamento baseado em prioridade

SOLUÇÃO 2: USE AS FACILIDADES DA CAMADA DE TRANSPORTE

- Exemplo: TCP permite mensagens urgentes na mesma conexão
- Mensagens urgentes podem ser capturadas usando técnicas de sinalização dos SO´s

SERVIDORES E ESTADO

SERVIDORES STATELESS

Nunca mantém informações precisas sobre o estado de um cliente depois de ter realizado o atendimento de uma requisição

- Não registra se um arquivo foi aberto (simplesmente fecha ele novamente após o acesso)
- Não promete invalidar a cache de um cliente
- Não mantém informações sobre seus clientes

CONSEQUÊNCIAS

- Clientes e servidores são completamente independentes
- Inconsistências de estado devido a quedas do servidor são reduzidas
- Possíveis perdas de desempenho por causa de coisas que não podem ser antecipadas (ex. um cliente que precisa antecipar prefetch de blocos)

QUESTÃO

Comunicação orientada a conexão serve para stateless design ?

SERVIDORES E ESTADO

SERVIDORES STATEFUL

Mantém o estado de seus clientes

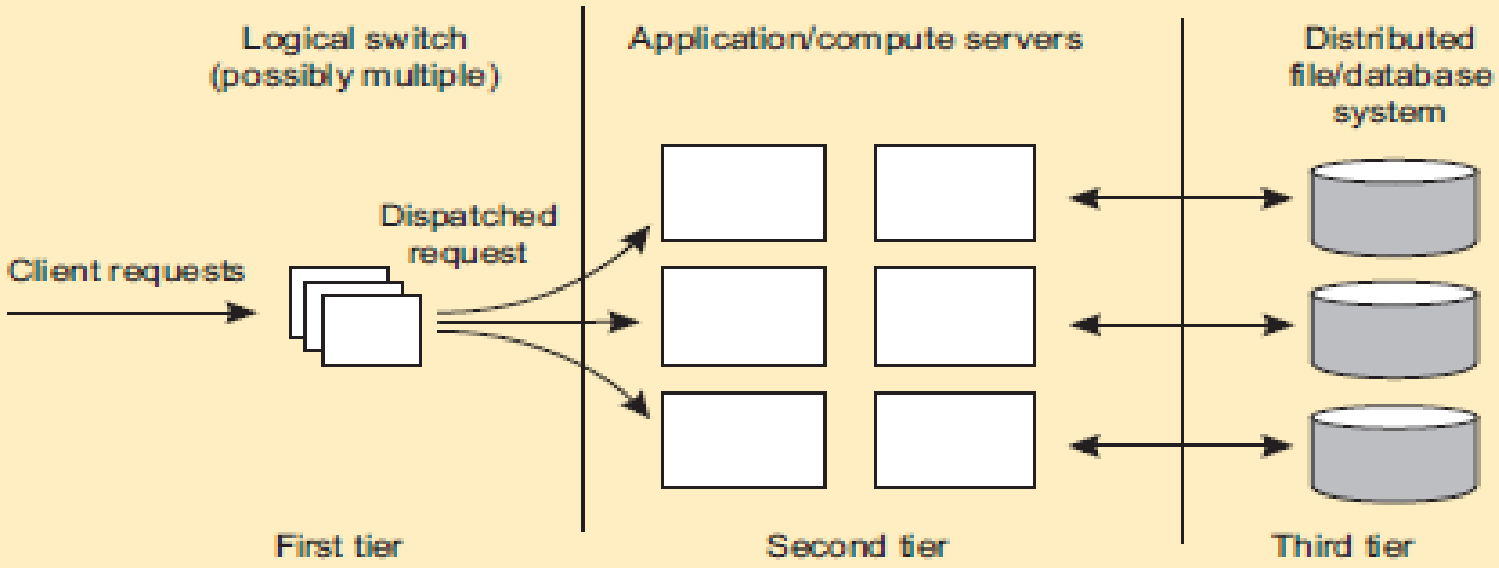
- Registra que um arquivo foi aberto de forma que é possível fazer *prefetch*
- Sabe quais dados estão na cache do cliente, e permite a manutenção de cópias locais de dados compartilhados

OBSERVAÇÃO

O desempenho de um servidor *stateful* pode ser bem alto, desde que os cliente permitam cópias locais. Na verdade, confiabilidade nem sempre é o maior problema

TRÊS DIFERENTES TIERS

ORGANIZAÇÃO COMUM



ELEMENTO CRUCIAL

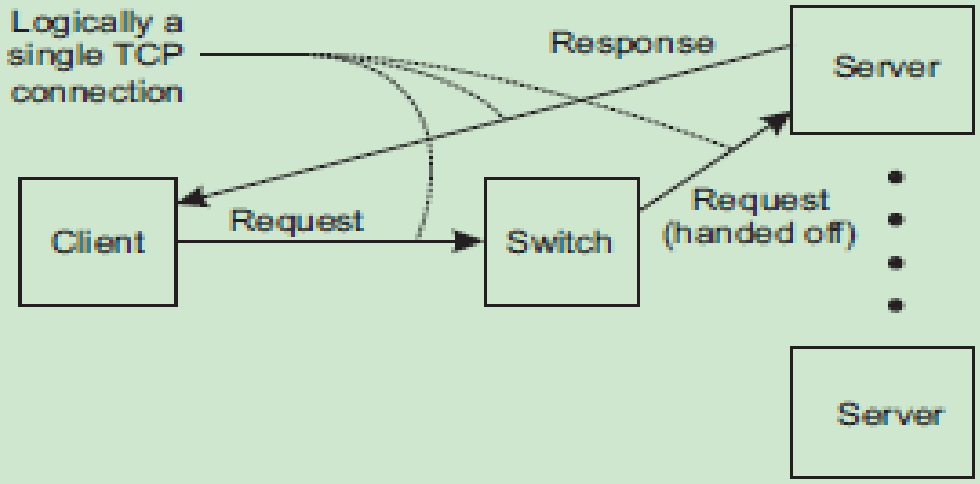
O primeiro tier é geralmente responsável por passar requisições para um servidor apropriado: request dispatching

HANDLING REQUISIÇÕES

OBSERVAÇÃO

- Ter a primeira camada (tier) manuseando toda comunicação de/para o cluster pode levar a bottleneck

SOLUÇÃO: TCP HANDOFF

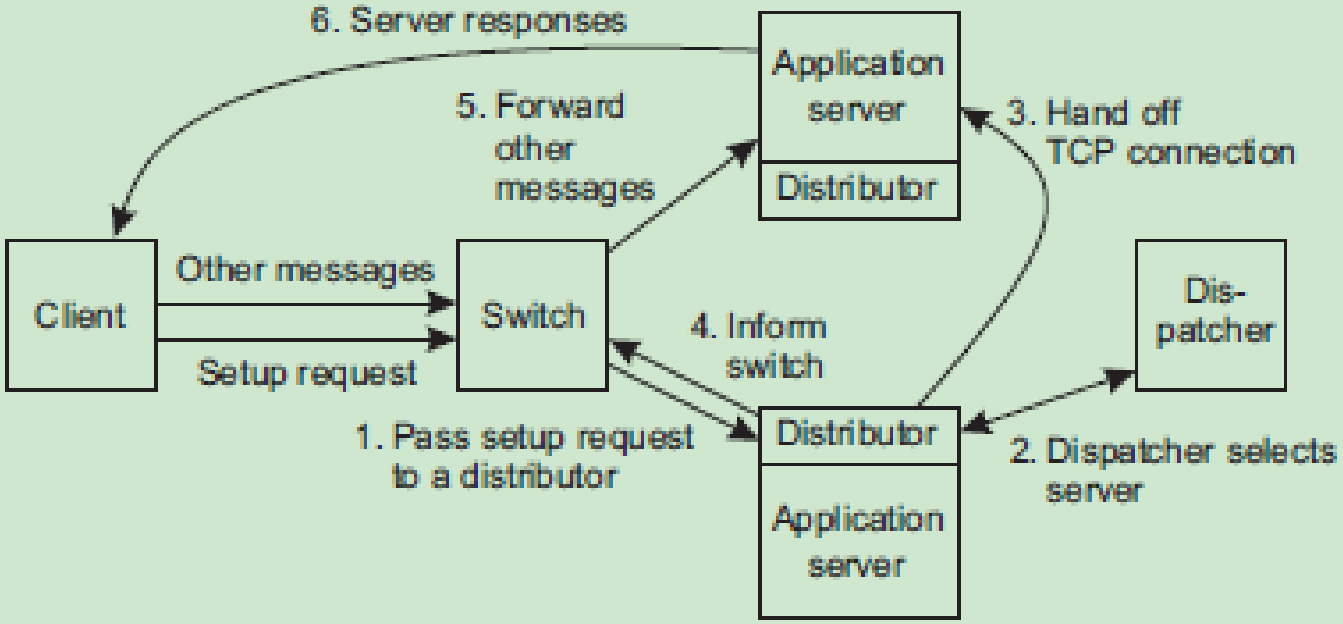


CLUSTER DE SERVIDORES

O FRONT-END PODE FACILMENTE FICAR SOBRECARREGADO: ATENÇÃO ESPECIAL PODE SER NECESSÁRIA

- Chaveamento na camada de transporte: o front-end passa as requisições TCP para um dos servidores, considerando algumas medidas de desempenho na decisão
- Distribuição baseada em conteúdo: front-end lê o conteúdo das requisições e seleciona o melhor servidor

SOLUÇÃO: TCP HANDOFF



SERVIDORES ESPALHADOS PELA INTERNET

OBSERVAÇÃO

Espalhar servidores na internet pode introduzir problemas administrativos. Isto pode ser minimizado usando data centers de um mesmo provedor de núvem

DISPATCH DE REQUISIÇÃO: SE LOCALIDADE É IMPORTANTE

Abordagem comum: use DNS

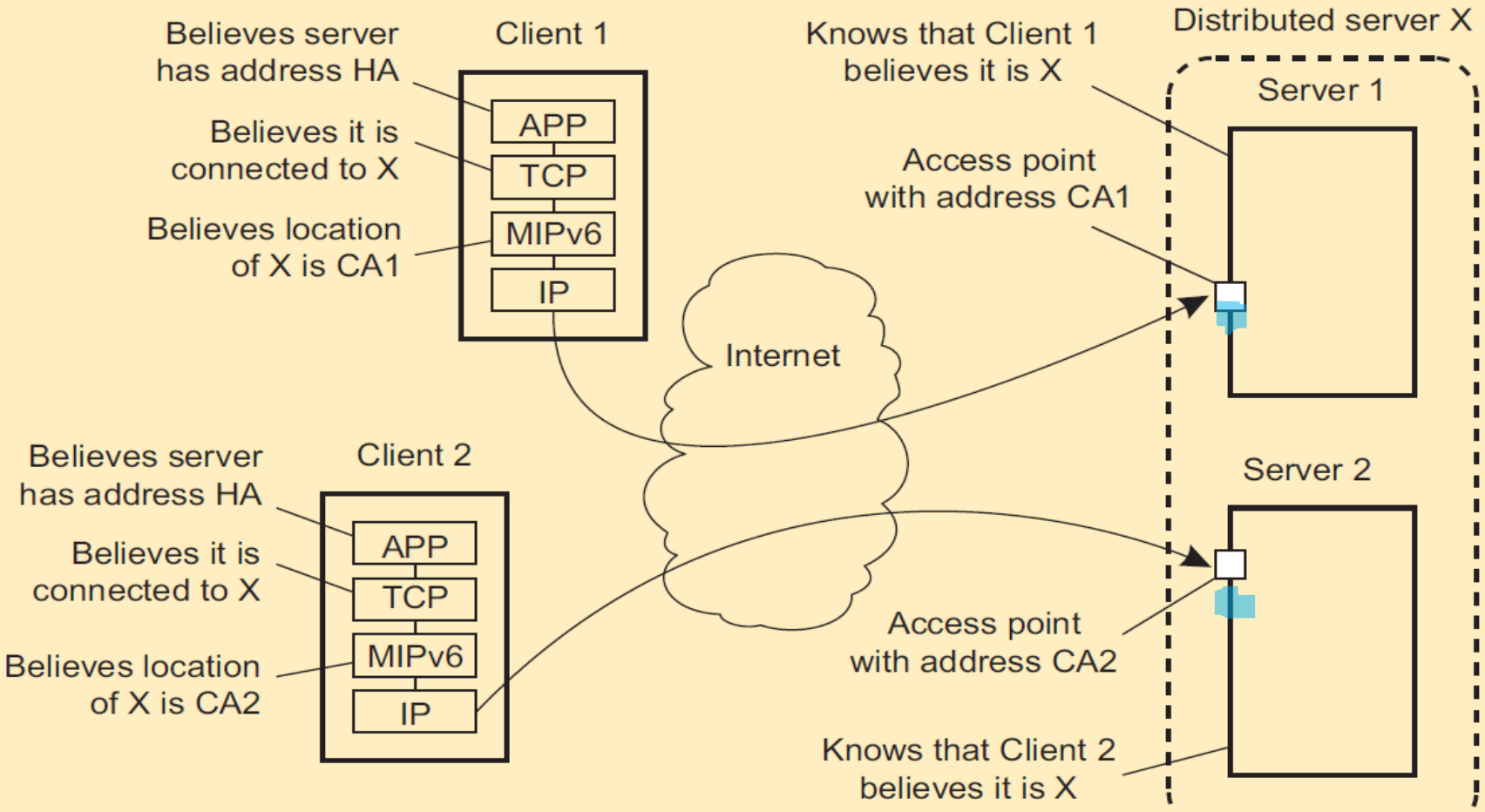
- Cliente procura um serviço específico através do DNS – o endereço do cliente é parte da requisição
- Servidor DNS mantém referência para servidores replica para requisitar serviços e retornar endereço do servidor mais local

TRANSPARÊNCIA NO CLIENTE

Para manter o cliente sem noção da distribuição, deixe o DNS resolver em nome do cliente. Problema é que o resolvedor pode na verdade estar longe do local do cliente atual

SERVIDORES DISTRIBUÍDOS COM ENDEREÇOS IPv6 ESTÁVEIS

TRANSPARÊNCIA ATRAVÉS DE IP MÓVEL



- *CA* (Care of Address)
- HA (home address)

SERVIDORES DISTRIBUÍDOS DETALHES DE ENDEREÇAMENTO

ESSÊNCIA: CLIENTES QUE POSSUEM UM IPv6 MÓVEL PODEM TRANSPARENTEMENTE SET UP UMA CONEXÃO PARA QUALQUER PEER

- Cliente C sets up uma conexão IPv6 *home address* (*HA*)
- *HA* é mantido por um agente local (*home agent*) (*network level*) que entrega a conexão para um registrador que toma conta do endereço *CA* (*Care of Address*)
- *C* pode então aplicar uma otimização de rota através do *forwarding* de pacotes para o endereço *CA* (i.e. sem o *hand off* através do agente local)

SISTEMAS DISTRIBUÍDOS COLABORATIVOS

- Servidor origem mantém um endereço local (*home*), mas *hands off* conexões para endereços de *peers* colaboradores → servidor origem e *peer* aparecem como um servidor

EXEMPLO PLANETLAB

ESSÊNCIA

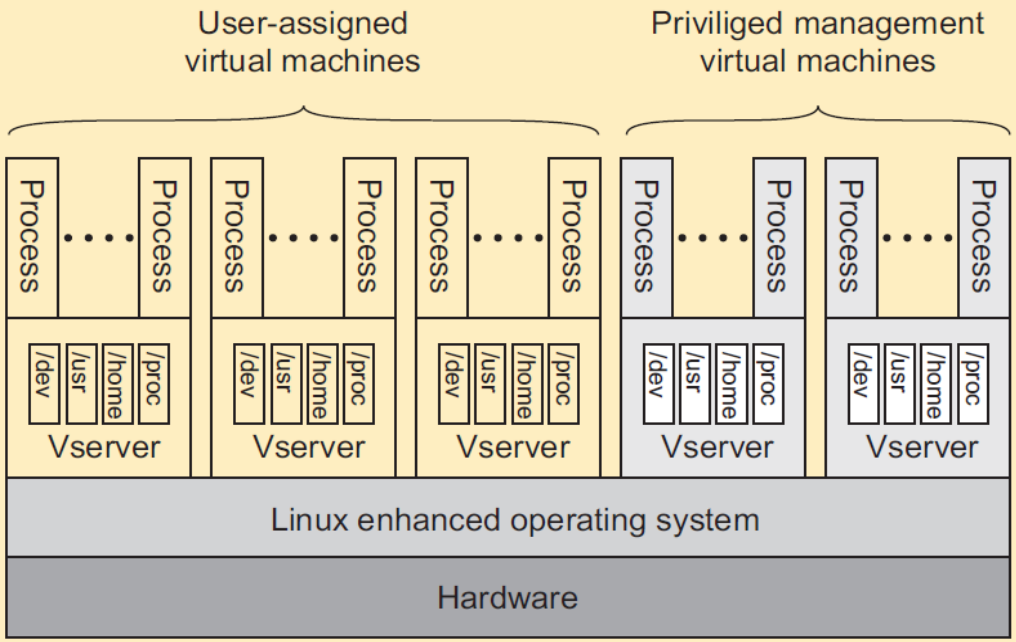
Diferentes organizações contribuem com máquinas, que depois são subsequentemente compartilhadas para vários experimento

PROBLEMAS

Precisamos garantir que diferentes aplicações distribuídas não atrapalhem umas as outras → virtualização

PLANETLAB ORGANIZAÇÃO BÁSICA

OVERVIEW



VSERVER

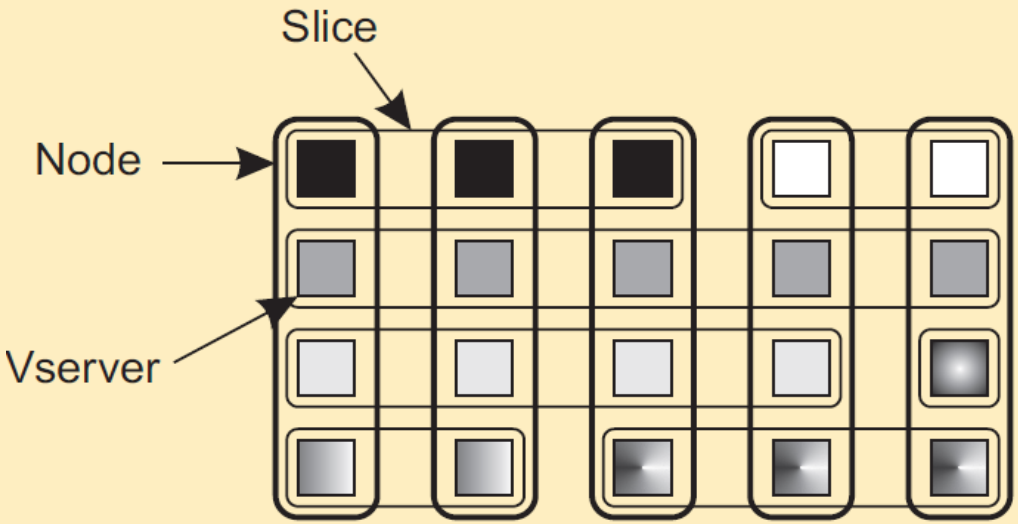
Ambientes independentes e protegidos com suas p´roprias bibliotecas, versões de servidores e etc. Aplicações distribuídas são designadas a uma coleção de vservers distribuídos através de múltiplas máquinas

PLANETLAB VSERVERS E SLICES

OVERVIEW

- Cada Vserver oper em seu próprio ambiente (`chroot`)
- Melhorias no Linux incluem ajustes próprios de ID´s de processos (ex. `init` ID 0)
- Dois processos em diferentes Vservers podem ter o mesmo user ID, mas isto não implica o mesmo usuários

SEPARAÇÃO LEVA A SLICES

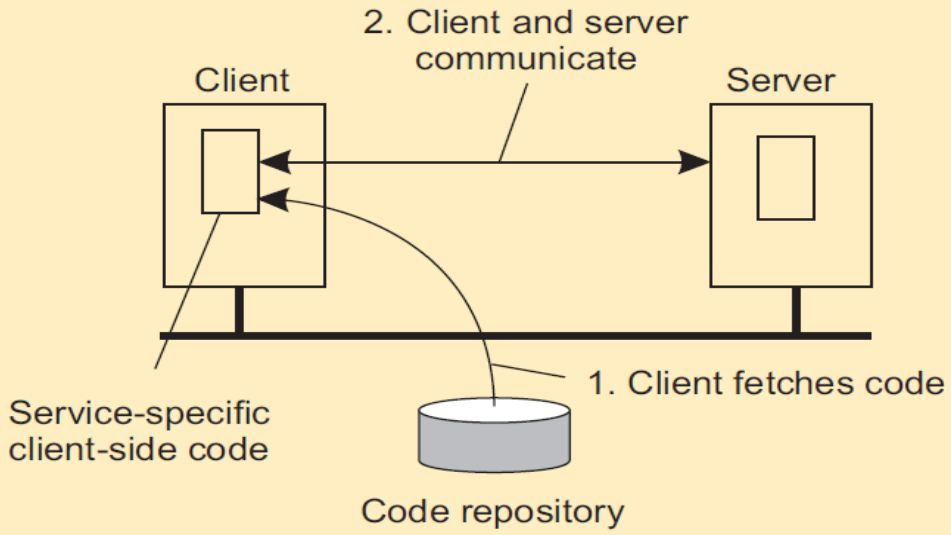


RAZÕES PARA MIGRAR CÓDIGO

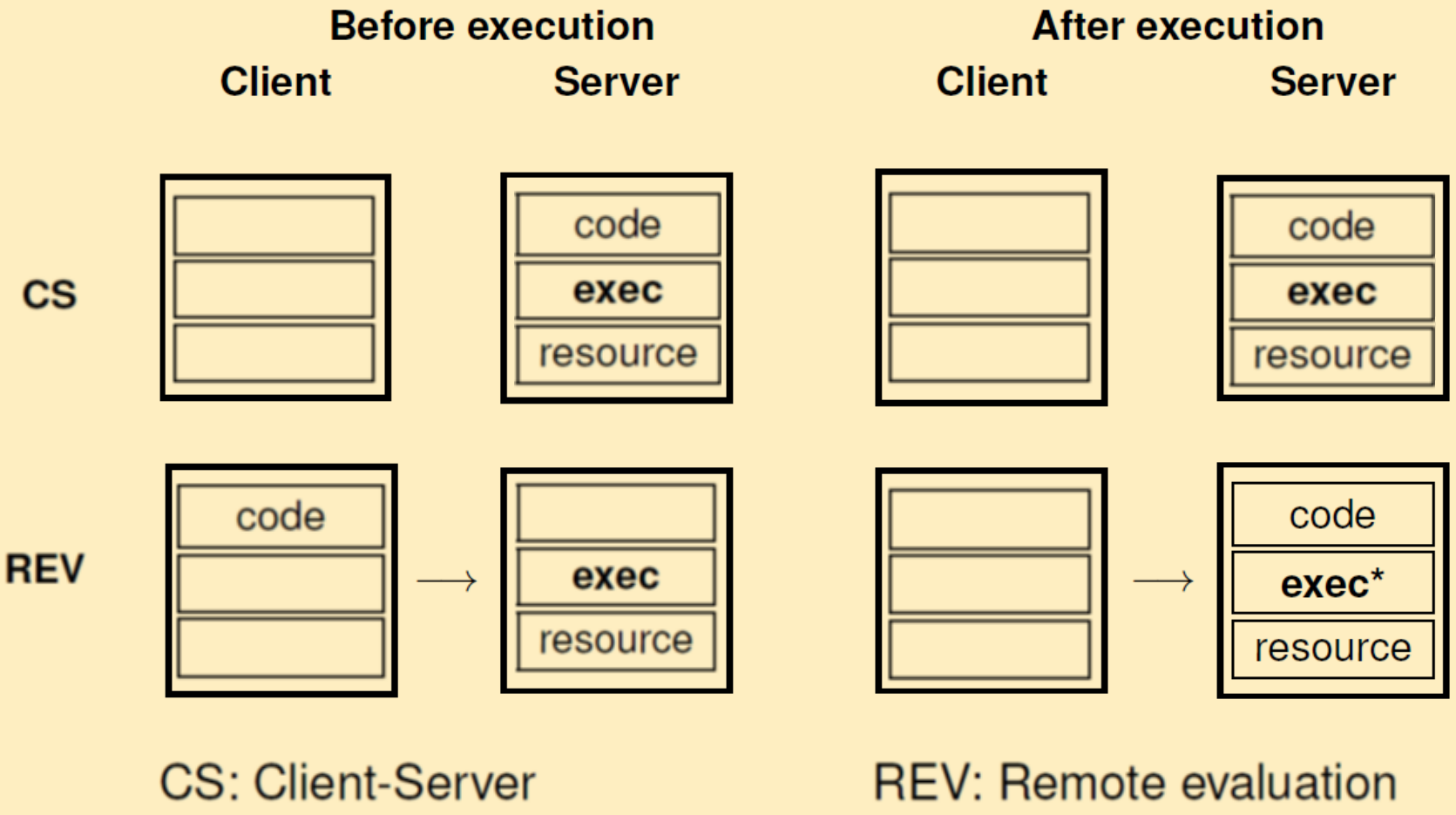
DISTRIBUIÇÃO DE CARGA

- Garantir que servidores em data centers estão suficientemente carregados (para minimizar desperdício de energia)
- Minimizar comunicação para garantir que computações estão próximas dos dados (pense em computação móvel)

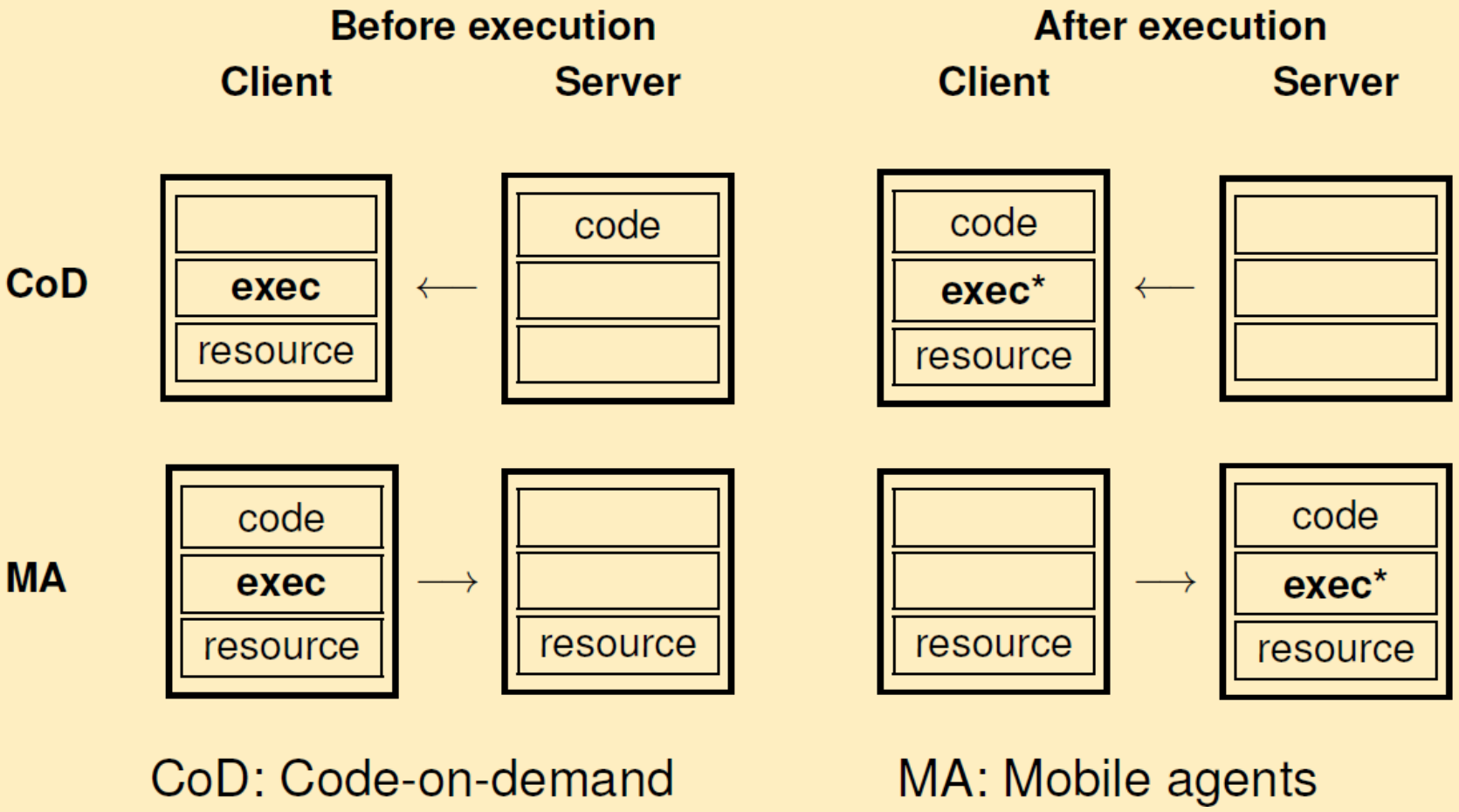
FLEXIBILIDADE: MOVER CÓDIGO PARA CLIENTE QUANDO NECESSÁRIO



MODELOS PARA MIGRAÇÃO DE CÓDIGO



MODELOS PARA MIGRAÇÃO DE CÓDIGO



MOBILIDADE FORTE E FRACA

COMPONENTES OBJETO

- Segmento de código: contém o código atual
- Segmento de dados: contém o estado
- Estado execução: contém contexto de thread executando o código do objeto

MOBILIDADE FRACA: MOVE SOMENTE O CÓDIGO E SEGMENTO DE DADOS (E REBOOT EXEC.)

- Relativamente simples, especialmente se código é portável
- Distingue code shipping (push) de code fetching (pull)

MOBILIDADE FORTE: MOVE COMPONENTE, INCLUINDO ESTADO DA EXECUÇÃO

- Migração: move o objeto todo de uma máquina pra outra
- Cloning: inicia um clone, e ajusta ele no mesmo estado de execução.

MIGRAÇÃO EM SISTEMAS HETEROGÊNEOS

PROBLEMA PRINCIPAL

- A máquina alvo pode não ser adequada para executar o código migrado
- A definição de contexto de processo / thread / processador é altamente dependente do hardware local, SO e runtime system

ÚNICA SOLUÇÃO: MÁQUINA ABSTRATA IMPLEMENTADA EM DIFERENTES PLATAFORMAS

- Linguagens interpretadas, efetivamente tendo sua própria VM
- VMM´s

MIGRAÇÃO DE MÁQUINA VIRTUAL

MIGRANDO IMAGENS: TRÊS ALTERNATIVAS

1. Fazer push de páginas de memória para uma nova máquina e enviar as que forem modificadas mais tarde durante o processo de migração
2. Parar a máquina virtual; migrar memória e iniciar uma nova VM
3. Deixar uma nova VM puxar (pull in) novas páginas sob demanda: processos iniciam imediatamente na nova VM e copiam páginas de memória sob demanda.

DESEMPENHO DA MIGRAÇÃO DE VM´s

PROBLEMA

- Uma migração completa pode levar algumas dezenas de segundos. Precisamos também saber que durante a migração, um serviço estará completamente indisponível por múltiplos segundos

MEDIÇÕES RELACIONADAS A TEMPO DE RESPOSTA DURANTE A MIGRAÇÃO DE VM´s