

Computação Distribuída

Processos, Threads e Migração

CHAPTER 3

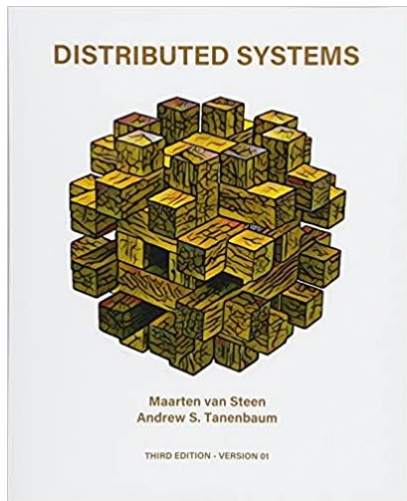
PROCESSES

Vladimir Rocha (Vladi)

CMCC - Universidade Federal do ABC



2007



2017

Disclaimer

- Estes slides foram baseados nos do professor **Emilio Franceschini** para o curso de Sistemas Distribuídos na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Estes slides foram adaptados daqueles originalmente preparados (e gentilmente cedidos) pelo professor **Daniel Cordeiro, da EACH-USP** que por sua vez foram baseados naqueles disponibilizados online pelos autores do livro "Distributed Systems", 3ª Edição em: <https://www.distributed-systems.net>.

Agenda

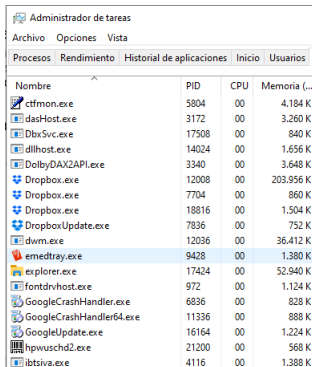
- Introdução à processos e threads
- Threads em sistemas distribuídos
- Clientes
- Servidores
- Migração de código

Agenda

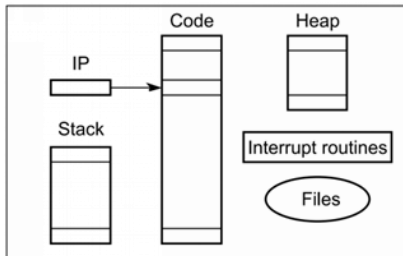
- Introdução à processos e threads
- Threads em sistemas distribuídos
- Clientes
- Servidores
- Migração de código

Processos

- Uma das abstrações mais importantes de um SO
 - Representam a **execução** de um programa
 - Execuções simultâneas de um programa são representadas por diversos processos
- Por segurança, os **espaços de memória** de cada processo são **isolados**
- Evita problemas que seriam causados por ataques deliberados ou por bugs
- Cada processo é uma **linha de execução independente** escalonada pelo SO

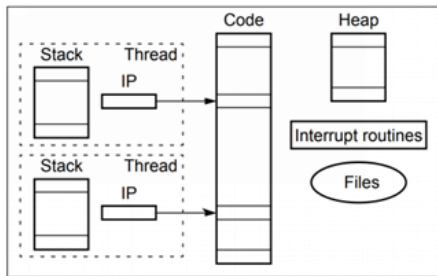


Administrador de tarefas			
Arquivo Opciones Vista			
Procesos	Rendimiento	Historial de aplicaciones	Inicio Usuarios
Nombre	PID	CPU	Memoria (...)
ctfmon.exe	5804	00	4,184 K
dasHost.exe	3172	00	3,260 K
DbxSvc.exe	17508	00	840 K
dllhost.exe	14024	00	1,656 K
DolbyDAX2API.exe	3340	00	3,648 K
Dropbox.exe	12008	00	203,956 K
Dropbox.exe	7704	00	860 K
Dropbox.exe	18816	00	1,504 K
DropboxUpdate.exe	7836	00	752 K
dwm.exe	12036	00	36,412 K
emedtray.exe	9428	00	1,380 K
explorer.exe	17424	00	52,940 K
fontdrvhost.exe	972	00	1,124 K
GoogleCrashHandler.exe	6836	00	828 K
GoogleCrashHandler64.exe	11336	00	888 K
GoogleUpdate.exe	16164	00	1,224 K
hpwuschd2.exe	21200	00	568 K
ibtsiva.exe	4116	00	1,388 K

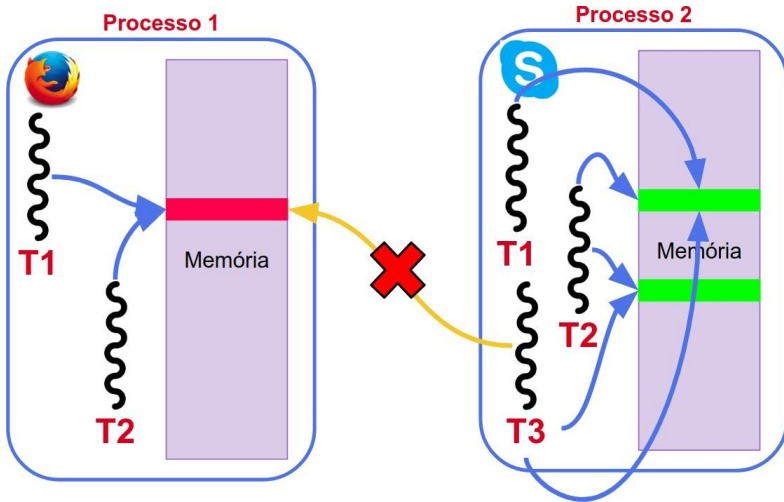


Threads

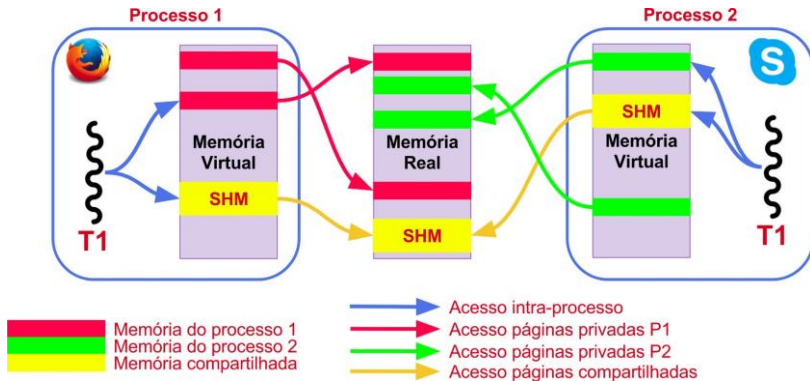
- Frequentemente um mesmo processo precisa fazer **mais de uma coisa por vez**. Ex.: Navegador de Internet
- Da mesma maneira que processos fornecem múltiplas linhas de execução em uma máquina, threads permitem **múltiplas linhas de execução em um só processo**
- Como efetivamente todas as threads são o mesmo processo, **todas têm acesso ao mesmo espaço de memória** e a todos os recursos disponíveis neste processo



Processos vs. Threads - Versão Simplificada



Processos vs. Threads - Versão (menos) Simplificada



Processos vs. Threads

- SO provê mecanismos para dividir o tempo do processador entre processos e threads, escalonando-os nas unidades de processamento disponíveis
 - Bloqueios por alguma operação de E/S causam uma troca do processo/thread pelo próximo na fila
- **Trocas de contexto entre threads são baratas**
 - Basta trocar o IP e mais alguns registradores
- **Trocas de contexto entre processos são mais caras**
 - Exigem troca da tabela de páginas, troca de IP, troca de rotinas de manuseio de interrupções, ...
- Ainda assim há momentos que o uso de processos pode ser preferível. Exemplo: Google Chrome

Processos vs. Threads

Benchmark	Operating System	Operation	Time per Op
Spawn New Process	NT	spawnl()	12.0 ms
	Linux	fork()/exec()	6.0 ms
Clone Current Process	NT	NONE	N/A
	Linux	fork()	1.0 ms
Spawn New Thread	NT	CreateThread()	0.9 ms
	Linux	pthread_create()	0.3 ms
Switch Current Process (20 runnable processes)	NT	Sleep(0)	0.010 ms
	Linux	sched_yield()	0.019 ms
Switch Current Thread (20 runnable threads)	NT	Sleep(0)	0.006 ms
	Linux	sched_yield()	0.019ms

Spawn: cria e executa

Agenda

- Introdução à processos e threads
- **Threads em sistemas distribuídos**
- Clientes
- Servidores
- Migração de código

Threads em sistemas distribuídos

Razões para utilizar várias threads:

- Evita bloqueios: para programas com uma thread, caso haja uma chamada a IO, o SO bloqueará a thread.
- Explora o paralelismo: as threads, em um programa multi-thread, podem ser executadas em paralelo em diversos processadores ou cores.
- Evita troca de contexto entre processos: leva à programação concorrente com memória compartilhada.

Threads no cliente

Cientes web multithreaded — escondem a latência da rede:

- Navegador analisa a página HTML sendo recebida e descobre que *muitos outros arquivos devem ser baixados*.
- Cada arquivo é baixado por uma thread separada; cada uma realiza uma requisição HTTP (bloqueante)
- A medida que os arquivos chegam, o navegador os exibem.

Múltiplas chamadas requisição-resposta (RPC) para outras máquinas

- Um cliente faz várias chamadas simultâneas, cada uma em uma thread diferente
- Ele espera até que todos os resultados tenham chegado.
- Obs: se as chamadas são a servidores diferentes, você pode ter um **speed-up linear**

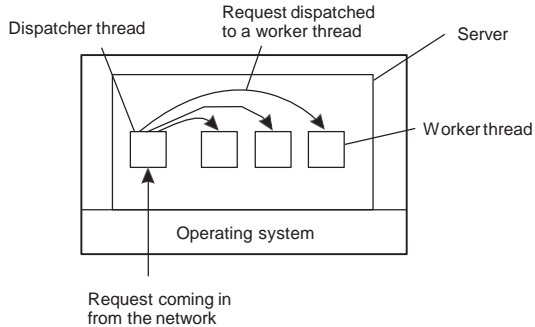
Threads no servidor

Melhorias no desempenho

- Iniciar uma thread é **muito** mais barato do que iniciar um novo processo
- Ter servidores single-threaded impedem o uso de sistemas que possuem vários processadores (multi-core)
- Tal como os clientes: **esconda a latência da rede** reagindo à próxima requisição enquanto a anterior está enviando sua resposta.

Threads no servidor

Modelo despachante / operário (dispatcher/worker)



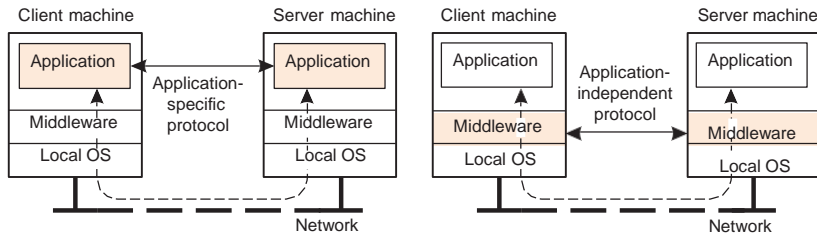
Agenda

- Introdução à processos e threads
- Threads em sistemas distribuídos
- **Clientes**
- Servidores
- Migração de código

Clientes: interação com servidor

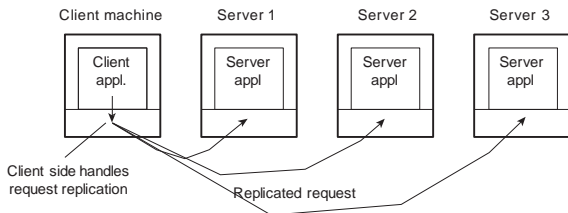
Cliente se comunica com o servidor

Usando protocolos de aplicação ou de middleware



Cientes: transparência de distribuição

- **transparência de acesso:** *stubs* do cliente para requisições RPC
- **transparência de localização/migração:** deixa o software cliente manter o controle sobre a localização atual
- **transparência de replicação:** múltiplas requisições são gerenciadas pelo stub do cliente:



- **transparência de falhas:** podem geralmente ser responsabilidade só do cliente (que tenta esconder falhas de comunicação e do servidor)

Agenda

- Introdução à processos e threads
- Threads em sistemas distribuídos
- Clientes
- **Servidores**
- Migração de código

Servidores: organização geral

Modelo básico

Um processo que implementa um serviço específico em nome de uma coleção de clientes.

Ele espera pela requisição de um cliente, garante que a requisição será tratada e, em seguida, passa a esperar pela próxima requisição.

Servidores: organização geral

Dois tipos básicos:

Servidores iterativos o servidor trata uma requisição antes de atender a próxima

Servidores concorrentes usa um despachante (*dispatcher*), que pega uma requisição e repassa seu tratamento a uma *thread*/processo separado

Observação

É mais comum encontrarmos **servidores concorrentes**: eles podem tratar múltiplas requisições mais facilmente, principalmente se for necessário realizar operações bloqueantes (em discos ou outros servidores).



Servidores *stateless*

Não mantém informação exata sobre o status de um cliente após ter processado uma requisição:

- Não guarda se um arquivo foi aberto (simplesmente fecha-o e abre de novo se necessário)
- Não promete invalidar o cache do cliente
- Não rastreia os seus clientes

Servidores e estado

Servidores *stateless*

Não mantém informação exata sobre o status de um cliente após ter processado uma requisição:

- Não guarda se um arquivo foi aberto (simplesmente fecha-o e abre de novo se necessário)
- Não promete invalidar o cache do cliente
- Não rastreia os seus clientes

Consequências

- Clientes e servidores são completamente independentes (evita coordenação)
- **Inconsistências de estado** devido a problemas no cliente ou servidor são reduzidas
- Possível **perda de desempenho**. Um servidor não pode antecipar o comportamento do cliente (ex: *prefetching*)



Servidores com estado (*stateful*)

Guardam o status de seus clientes:

- Registram quando um arquivo foi aberto para realização de *prefetching*
- Sabem quando o cliente possui cache dos dados e permitem que os clientes mantenham cópias locais de dados compartilhados

Servidores e estado

Servidores com estado (*stateful*)

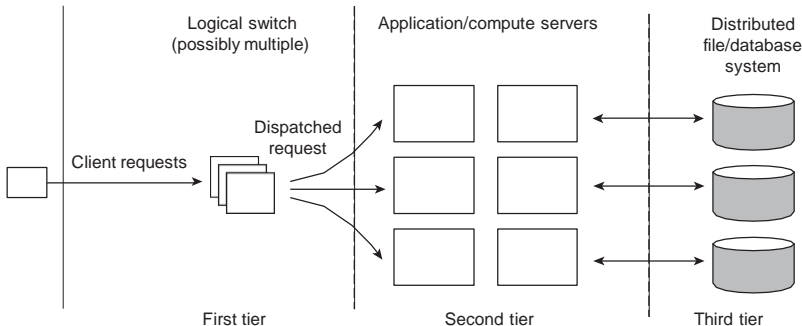
Guardam o status de seus clientes:

- Registram quando um arquivo foi aberto para realização de *prefetching*
- Sabem quando o cliente possui cache dos dados e permitem que os clientes mantenham cópias locais de dados compartilhados

Observação:

O desempenho de servidores *stateful* pode ser extremamente alto, desde que seja permitido que os clientes mantenham cópias locais dos dados. Nesse caso, **confiabilidade não é o maior problema e sim a inconsistência.**

Aglomerados de servidores: três camadas diferentes



A primeira camada (*frontend*) é responsável por interagir com clientes e repassar requisições para um servidor apropriado (*backend*).

O logical switch pode ser um proxy (componente intermediário que encaminha requisições) conhecido como *load balancer*.

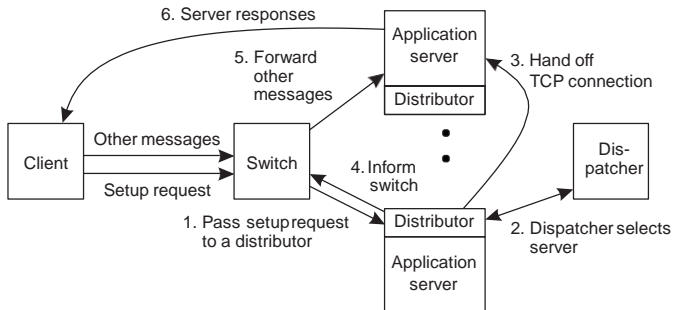
O *load balancer* utiliza diversas estratégias (round-robin, ociosidade, sessão, etc.) para encaminhar a requisição ao servidor adequado.

Aglomerados de servidores: tratamento de requisições

O frontend (switch do slide anterior) fica sobrecarregado facilmente: medidas precisam ser tomadas

Switch da camada de transporte: Frontend simplesmente redireciona o mensagem a um dos servidores backend (considerando métricas de desempenho)

Distribuição: frontend seleciona o melhor servidor backend de acordo ao conteúdo da mensagem recebida



Agenda

- Introdução à processos e threads
- Threads em sistemas distribuídos
- Clientes
- Servidores
- Migração de código

Migração de código

- Abordagens para realização de migração de código
- Migração em sistemas heterogêneos

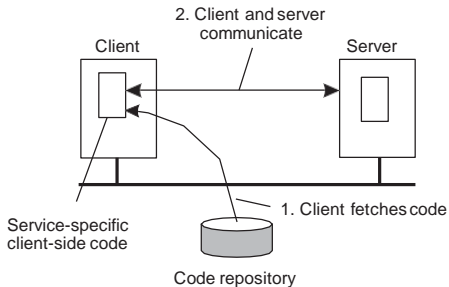
Migração de código: razões para migrar

Distribuição de carga

Assegurar que os servidores do data center estão suficientemente carregados (e.g. para prevenir perda de energia)

Minimizar a comunicação assegurando que os computadores estão próximos aos dados

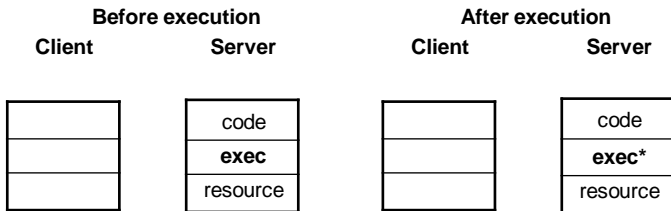
Flexibilidade: mover o código ao cliente quando necessário



Migração de código: contexto



CS



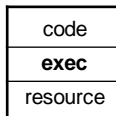
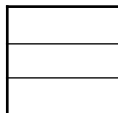
CS: Client-Server

Migração de código: contexto

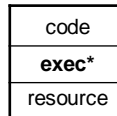
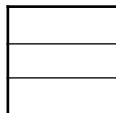


CS

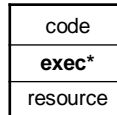
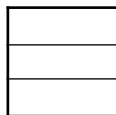
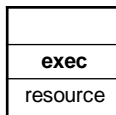
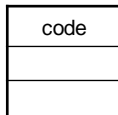
Before execution
Client Server



After execution
Client Server



REV



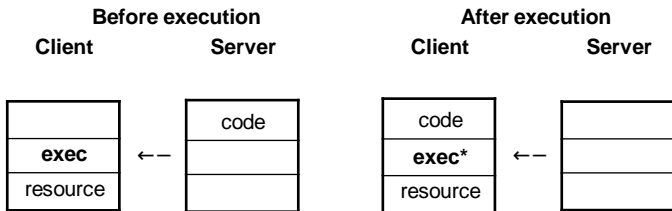
CS: Client-Server

REV: Remote evaluation

Migração de código: contexto



CoD

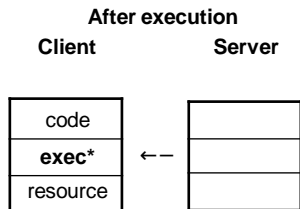
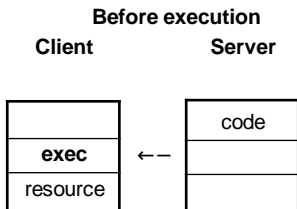


CoD: Code-on-demand

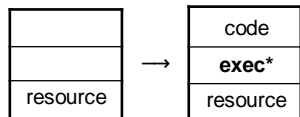
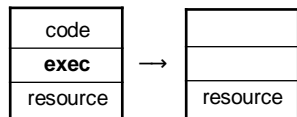
Migração de código: contexto



CoD



MA



CoD: Code-on-demand

MA: Mobile agents

Migração em sistemas heterogêneos

Problema principal

- A máquina destino pode não ser adequada para executar o código migrado
- A definição de contexto de thread/processo/processador é **altamente dependente do hardware, sistema operacional e bibliotecas locais**

Solução

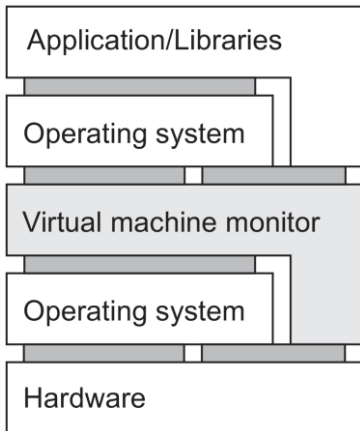
Usar alguma **máquina abstrata** que é implementada nas diferentes plataformas:

- Linguagens interpretadas, que possuem suas próprias MVs
- Uma MV que represente um SO completo
- Um container que represente um SO completo

Migração em sistemas heterogêneos

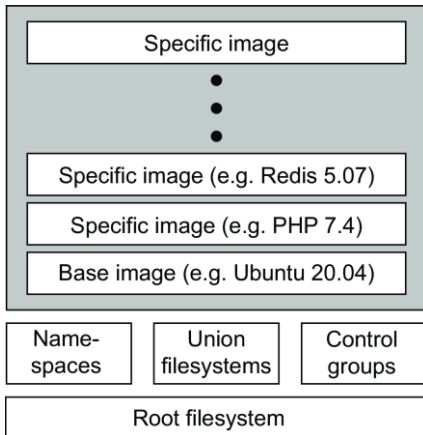
Máquina virtual de um SO

VirtualBox [2007]

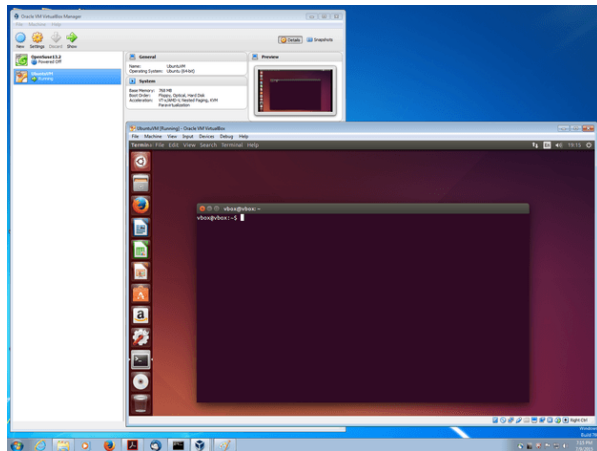


Container de um SO

Docker [2013]



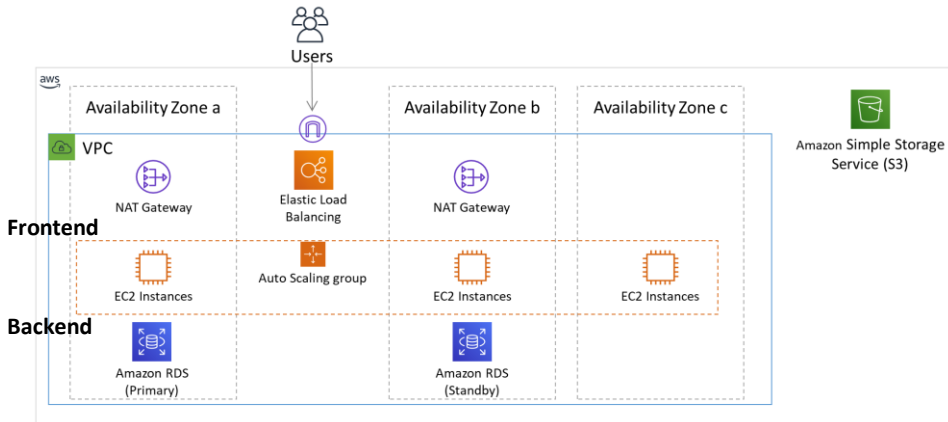
Migração em sistemas heterogêneos



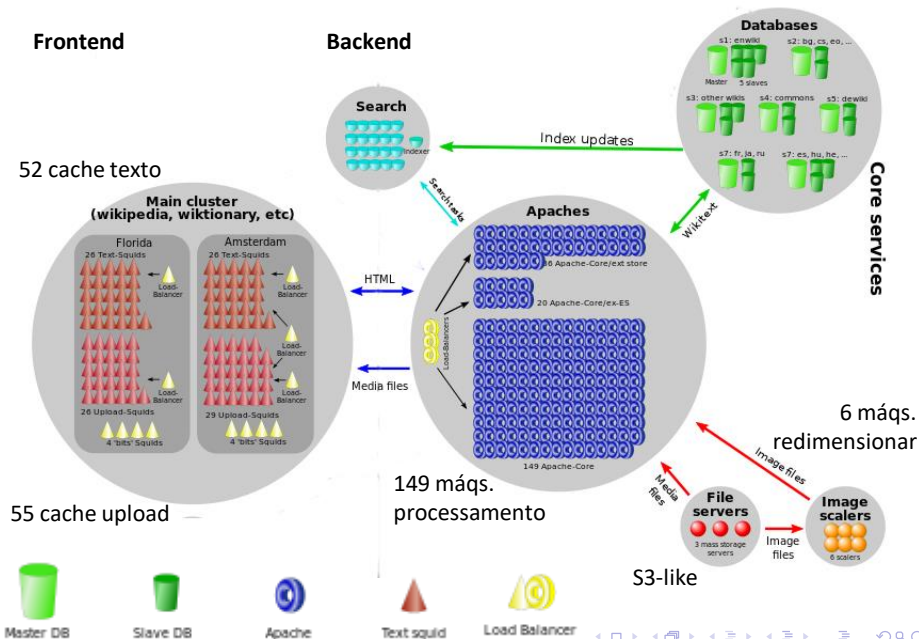
Vídeo de como instalar Ubuntu no Virtualbox (opcional)
https://www.youtube.com/watch?v=i_mNaqvrBe0

Servidores Caso1: Arquitetura Resiliente (AWS)

<https://aws.amazon.com/pt/blogs/architecture/building-resilient-well-architected-workloads-using-aws-resilience-hub/>



Servidores Caso2: Arquitetura do Wikimedia



Conceitos adquiridos

- Threads no servidor (dispatcher/worker).
- Stateless e Stateful.
- Load balancer.
- Virtualização.
- Frontend e Backend.