Resumo Científico Completo: Aplicações e Serviços de Computação em Nuvem

Universidade do Minho | Mestrado em Engenharia Informática Disciplina: Aplicações e Serviços de Computação em Nuvem

Índice

1. Introdução
2. Virtualização e Máquinas Virtuais
3. Contentores e Docker
4. Kubernetes e Orquestração
5. Armazenamento Distribuído
6. Provisioning e Automação
7. Monitorização de Sistemas
8. Arquiteturas Distribuídas
9. Modelos de Serviço Cloud
10. DevOps e Infraestrutura como Código
11. Benchmarking e Análise de Performance
12. Conclusão

Introdução

A Computação em Nuvem transformou fundamentalmente a forma como as organizações gerem infraestrutura, desenvolvem software e entregam serviços. Este resumo científico apresenta uma síntese rigorosa dos conceitos, tecnologias e práticas essenciais que suportam as arquiteturas modernas de aplicações distribuídas em ambientes cloud.

A disciplina "Aplicações e Serviços de Computação em Nuvem" aborda três camadas de abstração fundamentais:

1. **Infraestrutura**: Como recursos físicos são abstratos (virtualização, contentores)
2. **Plataforma**: Como aplicações são orquestradas e escaladas (Kubernetes, Ansible)
3. **Serviços**: Como valor é entregue aos utilizadores (IaaS, PaaS, SaaS)

Cada camada introduz desafios distintos que exigem soluções específicas, discutidas ao longo deste documento.

Virtualização e Máquinas Virtuais

Fundamentação Teórica

A virtualização é uma abstração fundamental que permite criar múltiplas representações lógicas (máquinas virtuais) a partir de um único recurso físico (servidor). Esta abstração resolve problemas críticos de

infraestrutura.

Problema Original (Pré-2000):

- Cada aplicação necessitava de servidor dedicado
- Subutilização significativa de recursos (tipicamente 15-30% de CPU)
- Custos elevados de infraestrutura e espaço em data center
- Gestão heterogénea (diferentes modelos de CPU, interfaces)

Solução: Os Três Pilares da Virtualização

1. Consolidação e Eficiência

A virtualização permite que múltiplos sistemas operativos e aplicações corram no mesmo hardware físico. Mecanismo: Time-slicing de CPU, alocação de memória por VM. Benefício quantificável: Aumento de utilização de 15% → 70-80% em data centers modernos.

2. Isolamento e Segurança

Cada VM é isolada através do hypervisor, garantindo que:

- Uma VM comprometida não afeta outras
- Recursos dedicados não são competidos entre VMs (sem contenção de desempenho)
- Vulnerabilidades são confinadas

Mecanismo: Hardware-based isolation (TLB, page tables, memory protection rings).

3. Abstração e Transparência

A heterogeneidade do hardware é abstraída. Uma aplicação desenvolvida para arquitetura x86 corre em qualquer servidor x86 sem modificações. Benefício: Portabilidade total, simples para utilizadores finais.

Arquitetura do Hypervisor

Um hypervisor é o software fundamental que intermedia acesso de VMs ao hardware físico.

Responsabilidades do Hypervisor:

1. **Gestão de CPU**: Interceta e traduz instruções privilegiadas do SO convidado
2. **Gestão de Memória**: Virtualiza espaço de endereços através de "shadow page tables"
3. **Gestão de I/O**: Cria abstrações de dispositivos virtuais (discos, NICs)
4. **Gestão de Rede**: Implementa bridges virtuais ou NAT para conectividade

Classificações Importantes

Virtualização Completa vs. Paravirtualização

Aspeto	Virtualização Completa	Paravirtualização
Modificação SO Convidado	Nenhuma	Necessária (hooks)

Aspeto	Virtualização Completa	Paravirtualização
Tradução Instruções	Completa (Intel VT-x/AMD-V)	Parcial
Desempenho	Bom (com hardware support)	Excelente
Portabilidade	Máxima	Limitada
Exemplos	VirtualBox, VMware	Xen

Tipos de Hypervisor

Tipo 1 (Bare Metal):

- Instalado diretamente no hardware
- Funciona como "micro kernel" otimizado
- Desempenho máximo
- Exemplos: VMware ESX, KVM
- Caso de uso: Data centers de produção

Tipo 2 (Hosted):

- Instalado como aplicação sobre SO de propósito geral
- Overhead do SO anfitrião reduz desempenho
- Flexibilidade máxima
- Exemplos: VirtualBox, Parallels
- Caso de uso: Desenvolvimento local

Contentores e Docker

Motivação e Problema Original

O Problema: "Mas Funcionava na Minha Máquina"

Antes dos contentores, o ciclo de vida do desenvolvimento enfrentava desafios crónicos:

1. **Dependency Hell**: Conflitos entre versões de bibliotecas
2. **Inconsistência de Ambientes**: Dev, Teste, Produção eram diferentes
3. **Peso das VMs**: Cada VM ocupa 50GB+, arranca em segundos/minutos

Conceito Fundamental de Contentor

Um contentor é um ambiente virtual leve que:

- Agrupa processos e recursos (CPU, memória, disco)
- Isola aplicação do sistema anfitrião
- Partilha o kernel do SO anfitrião (diferença crítica com VMs)
- Empacota código + dependências + configuração

Benefícios:

1. **Isolamento com Leveza**: Sem overhead de kernel completo
2. **Consistência**: "Build once, run anywhere"
3. **Portabilidade**: Funciona em laptop, servidor físico, cloud
4. **Eficiência**: Múltiplos contentores por servidor

Tecnologia Subjacente (Linux Kernel)

Os contentores utilizam três mecanismos do kernel Linux:

1. Namespaces (Isolamento)

Particionam recursos globais do kernel para cada contentor:

- **PID Namespace**: Contentor vê próprios PIDs
- **Network Namespace**: Interface de rede virtual própria
- **Mount Namespace**: Sistema de ficheiros próprio

2. Control Groups (cgroups) - Gestão de Recursos

Limitam quantidade de recursos que cada contentor pode utilizar:

- **CPU**: Limitar a 1 core, mesmo se servidor tem 8
- **Memória**: Limite máximo de 512MB
- **I/O**: Limitações de banda de disco

3. SELinux (Segurança)

Adiciona controlo de acesso obrigatório (MAC) para garantir isolamento.

Kubernetes e Orquestração

Motivação: Do Contentor à Frota

Executar 1 contentor é trivial. Aplicações modernas requerem dezenas ou centenas de contentores que precisam:

- Distribuição automática por múltiplos servidores
- Replicação para alta disponibilidade
- Auto-reparação (reiniciar contentores que caem)
- Descoberta de serviços
- Balanceamento de carga
- Escalabilidade horizontal automática

Arquitetura Kubernetes

Kubernetes é um sistema operativo distribuído para contentores. Arquitetura mestre-escravo.

Control Plane (Cérebro)

1. **API Server**: Ponto de entrada, fonte da verdade do cluster
2. **etcd**: Base de dados distribuída, armazena todo estado
3. **Scheduler**: Coloca pods nos nós considerando requisitos
4. **Controller Manager**: Garante estado desejado (ReplicaSet, Deployment, etc.)

Nodes (Músculos)

Em cada nó corre:

1. **Kubelet**: Agente que assegura contentores estão a executar
2. **Container Runtime**: Executa contentores (Docker, containerd, rkt)
3. **Kube-proxy**: Gerencia rede, implementa serviços

Objetos Fundamentais

Pod: Unidade mais pequena no Kubernetes. Pode ter múltiplos contentores que partilham rede.

Deployment: Objeto de alto nível que define número de réplicas desejadas. Kubernetes assegura que sempre corre o número correto.

Service: Abstração para descoberta de serviços. Fornece IP estável e DNS, balanceia tráfego entre pods.

Armazenamento Distribuído

Desafios Fundamentais

O armazenamento em sistemas distribuídos enfrenta desafios únicos:

1. **Replicação**: Manter dados consistentes entre múltiplos nós
2. **Tolerância a Falhas**: Sobreviver a falhas de nós, discos
3. **Escalabilidade**: Gerenciar volumes crescentes de dados
4. **Eficiência**: Otimizar para latência vs. throughput vs. espaço

Separação de Dados e Metadados

Padrão: Manager-Worker

- **Manager/Metadata Servers**: Gerem namespace, localização de ficheiros
- **Data Nodes**: Armazenam dados

Vantagens da Separação:

- Metadados são pequenos (cabem em memória)
- Dados são volumosos (distribuídos em múltiplos discos)
- Escalabilidade diferenciada
- Operações de ficheiro são atómicas centralizadas

Disponibilidade de Dados: Replicação vs. Erasure Codes

Replicação: Múltiplas cópias exatas. Overhead 3x, tolerância a 2 falhas, leitura rápida.

Erasure Codes: Dados divididos em k fragmentos + m blocos de paridade. Overhead 1.5x, tolerância a 3 falhas, leitura requer reconstrução.

Quando Usar:

- **Replicação**: Dados "quentes", requisitos de baixa latência
- **Erasure Codes**: Dados "frios", grandes volumes, baixo custo

---

Provisioning e Automação

Desafios da Gestão Manual

Infraestruturas modernas com dezenas/centenas de servidores exigem automação. Abordagem manual é impraticável.

Problemas:

1. Repetitivo e moroso
2. Heterogéneo
3. Evolução constante
4. Propenso a erros (configuration drift)

Paradigmas: Imperativo vs. Declarativo

Imperativo (Shell Scripts): Define **como** fazer algo (passo a passo).

- Não idempotente
- Frágil
- Difícil diagnosticar

Declarativo (Ansible): Define **o que** queremos (estado final desejado).

- Idempotente
- Robusto
- Diagnostico claro

Ansible: Infraestrutura como Código

Ansible implementa paradigma declarativo com:

- **Agentless**: Usa SSH
- **YAML**: Linguagem simples, legível
- **Idempotente**: Apenas mudanças necessárias
- **Modular**: Roles promovem reutilização

Monitorização de Sistemas

Fundamentação Teórica

Sistemas modernos são "caixas negras" que geram milhões de eventos por segundo. A monitorização transforma "ruído" em "sinal".

Arquitetura de Monitorização em 4 Camadas

1. **Observação**: Recolhe eventos brutos
2. **Recolha**: Agrega e normaliza eventos
3. **Análise**: Processa, indexa, armazena dados
4. **Apresentação**: Visualiza em dashboards, alertas, relatórios

Trade-offs: Event-driven vs. Sampling

Event-Driven: 100% precisão, overhead muito alto **Sampling**: Precisão parcial, overhead baixo

Escolha depende do contexto e requisitos de performance.

Crítica das Médias

As médias são enganadoras. Exemplo:

- Servidor A: [50, 50, 50, 50, 50] → Média 50ms
- Servidor B: [10, 10, 10, 10, 200] → Média 48ms

Servidor B parece melhor, mas 20% dos pedidos são 10x lentos.

Métricas Corretas: Percentis (P50, P95, P99), ECDF, visualizações.

Arquiteturas Distribuídas

Padrões Fundamentais de Distribuição

1. Replicação: Múltiplas cópias idênticas. Para disponibilidade e escalabilidade de leitura.

2. Particionamento (Sharding): Dados/funcionalidade divididos. Para escalabilidade de escrita.

3. Orientação a Serviços: Sistema dividido em serviços independentes. Para modularidade e escalabilidade.

Arquitetura Multi-tier

Típico: Presentation (stateless) → Application (state transitório) → Data (stateful)

Desafio: Replicação de componentes stateful é exponencialmente mais complexa que stateless.

Modelos de Serviço Cloud

Princípio Fundamental: Partilha de Responsabilidades

À medida que subimos na abstração, delegamos mais responsabilidades ao fornecedor.

IaaS: Cliente gerencia aplicações e SO. Fornecedor: infraestrutura. **PaaS**: Cliente gerencia apenas aplicações. Fornecedor: plataforma completa. **SaaS**: Fornecedor gerencia tudo. Cliente apenas usa.

OpenStack: Nuvem Privada IaaS

Software open-source para criar nuvem IaaS privada com controlo total.

Componentes: Nova (compute), Cinder (storage), Swift (object), Neutron (network), Glance (images).

Vantagens Privada vs. Pública: Controlo total, conformidade regulatória, potencial menores custos em escala.

DevOps e Infraestrutura como Código

DevOps como Síntese

DevOps une Desenvolvimento e Operações, rompendo silos tradicionais.

Objetivo: Entregar valor mais rápido e fiável.

Infraestrutura como Código (IaC)

Infraestrutura definida em código versionado:

- **Reprodutibilidade**: Mesmo código = Mesmo resultado
- **Versionamento**: Git para histórico completo
- **Automação**: Deploy em minutos, não semanas
- **Documentação**: Código é documentação

Ciclo DevOps: Plan → Code → Build → Test → Deploy → Operate → Monitor → Feedback

Cada fase contribui para velocidade e fiabilidade.

Benchmarking e Análise de Performance

Os Três Pilares de um Benchmark

1. **Workload**: Conjunto de pedidos (traces reais vs. sintéticas)
2. **Environment**: Hardware e software documentado
3. **Metrics**: O que medimos

Latência vs. Throughput

Relação Intuitiva Errada: L = 1/T **Relação Real**: Depende de carga (3 fases):

- Fase 1 (Idle): Ambas melhoram
- Fase 2 (Near Capacity): Throughput sobe, latência começa a subir
- Fase 3 (Overload): Throughput estagna, latência dispara

Little's Law: Latência = (Fila + Tempo Serviço) / Throughput

Crítica das Médias (Novamente)

Padrões que as médias escondem:

- Long Tail: Alguns pedidos muito lentos
- Degradação: Performance piora ao longo do tempo
- Bimodalidade: Dois modos de comportamento

Métricas Corretas: Percentis, desvio padrão, ECDF, histogramas.

Conclusão

A Computação em Nuvem é um ecossistema complexo que exige compreensão em múltiplas camadas:

Mensagens Centrais

1. **Tudo é Trade-off**: Não existe solução perfeita. Cada escolha envolve compromissos.
2. **Escalabilidade é Difícil**: Escalar horizontalmente é fácil em teoria, mas complexo em prática (estado, consistência, rede).
3. **Observabilidade é Crítica**: Sem monitorização, é impossível diagnosticar problemas em sistemas complexos.
4. **Automação é Essencial**: Operações manuais não escalam. IaC e DevOps são fundamentais.
5. **Conhecimento de Padrões**: Arquiteturas distribuídas existem por razões. Compreender padrões permite tomar decisões informadas.

A Computação em Nuvem não é apenas sobre tecnologia, é sobre permitir que organizações entreguem valor aos utilizadores de forma mais rápida, fiável, e eficiente.

Documento Final: Resumo Científico Completo Universidade do Minho | Engenharia Informática
Baseado em conteúdo lecionado e padrões de avaliação (2022-2024)