Resumo SD

1) **Introdução** (Porquê distribuir um sistema? Quais os principais desafios da sua distriuição?)

O que é um **sistema distribuído**? É um sistema de componentes **software** ou **hardware** localizados em **computadores ligados em rede**, que comunicam e coordenam as suas acções através de **troca de mensagens**.
Exemplos: A rede informática de qualquer grande empresa. Monitores de pesquisa na web. Youtube. Redes sociais. Jogos em rede. Sistemas automáticos de trading financeiro. Computação em nuvem. Etc…

Protocolos da Internet como infraestrutura de suporte:

- **Passível de alteração** (**Inovação**): Mail, Web, Audio, VoIP, Video, IM, WebServices, Ethernet, GPRS, 802.11, Satélite, Bluetooth.
- **Dificil de alterar** (**Estabilidade**): TCP/UDP, IP.

Papel determinante da World Wide Web:

- Alteração do **padrão de utilização** dos serviços de telecomunicações;
- Desenvolvimento de standards de facto que permitiram criar **novas formas de trocar informação**: HTTP, HTML, XML, JSON;
- Criação de ambientes de **desenvolvimento simplificados**: PHP, Pearl, Java, Python, etc…

Evolução dos Sistemas Distribuidos:

- 1) Computação distribuida = Pouca mobilidade, pouco embebida;
- 2) Computação móvel (Em mais locais) = Muita mobilidade, pouco embebida;
- 3) Computação pervasiva (Em mais dispositivos) = Pouca mobilidade, muito embebida;
- 4) Computação ubíqua (Em todo o lado e todas as coisas - “A Internet das Coisas”) = Muita mobilidade, muito embebida.

Objetivos: Analisar as **arquiteturas e as soluções técnicas** que permitem desenvolver aplicações distribuídas, garantindo os **requisitos não funcionais** como a tolerância a faltas, segurança, reconfigurabilidade e escalabilidade.
Para tal, é necessário analisar os **problemas** que se colocam nos sistemas distribuídos e quais são as **soluções** para os ultrapassar.

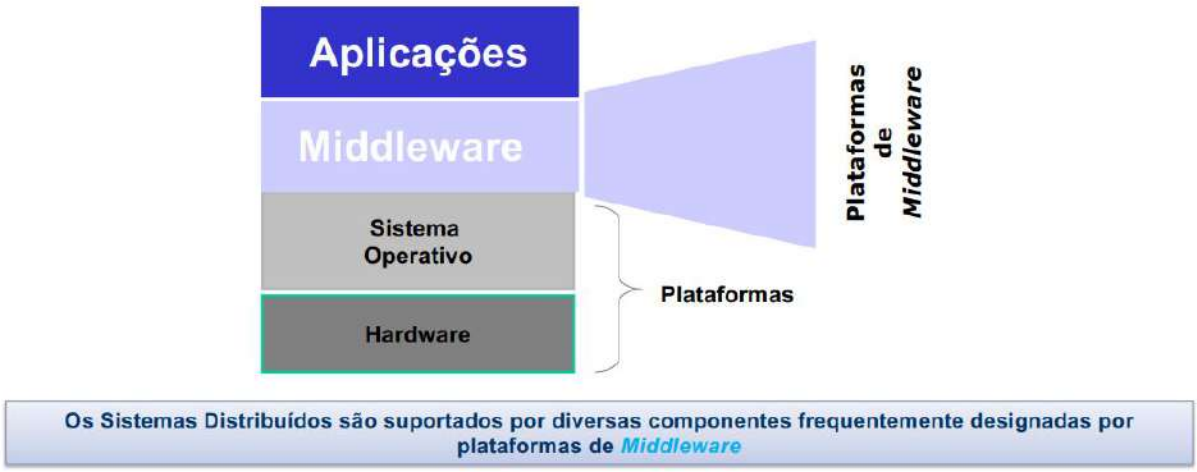
Razões para a distribuição:

- Distribuição geográfica;
- Extensibilidade, modularidade;
- Partilha de recursos;
- Maior disponibilidade;
- Maior desempenho;
- Outsourcing e comunicação em Nuvem.

Consequências e desafios:

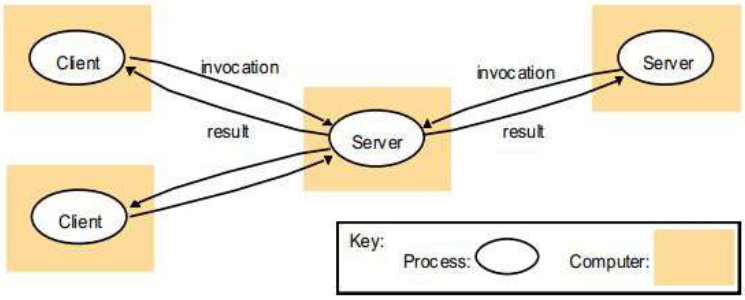
- Não há relógio global;
- Concorrência;
- Falhas independentes;
- Segurança.

Camadas de software: o middleware:

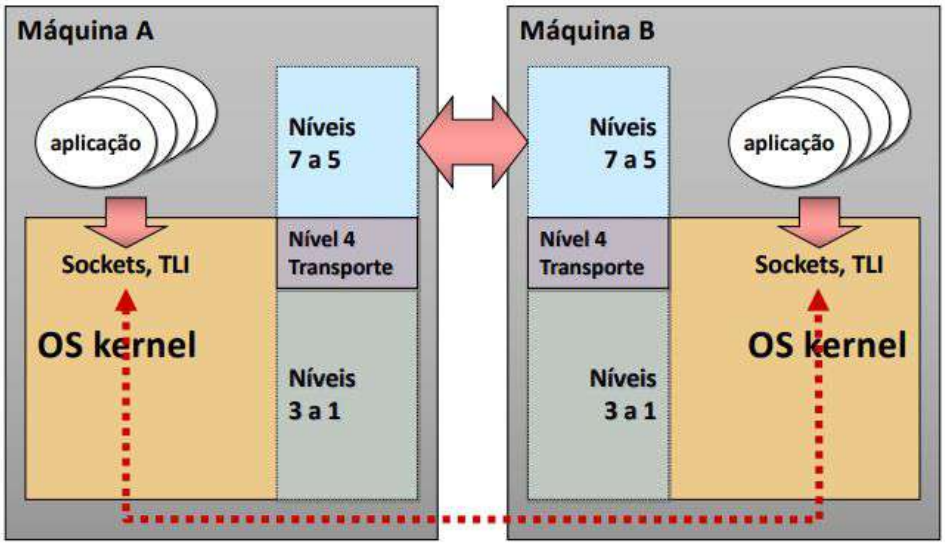


Arquitetura cliente-servidor:

- Servidores mantêm recursos e servem pedidos de operações sobre esses recursos;
- Servidores podem ser clientes de outros servidores;
- Simples e permite distribuir sistemas centralizados muito diretamente, mas pouco estável, pois está limitado pela capacidade do servidor e pela rede que o liga aos clientes.



Interfaces de comunicação:



Caracterização do canal de comunicação:

- Com ligação: Normalmente serve 2 interlocutores. Normalmente fiável, bidirecional e garante sequencialidade.
- Sem ligação: Normalmente serve mais de 2 interlocutores. Normalmente não fiável (perdas, duplicação, reordenação).
- Com capacidade de armazenamento em fila de mensagens: Normalmente com entrega fiável das mensagens.

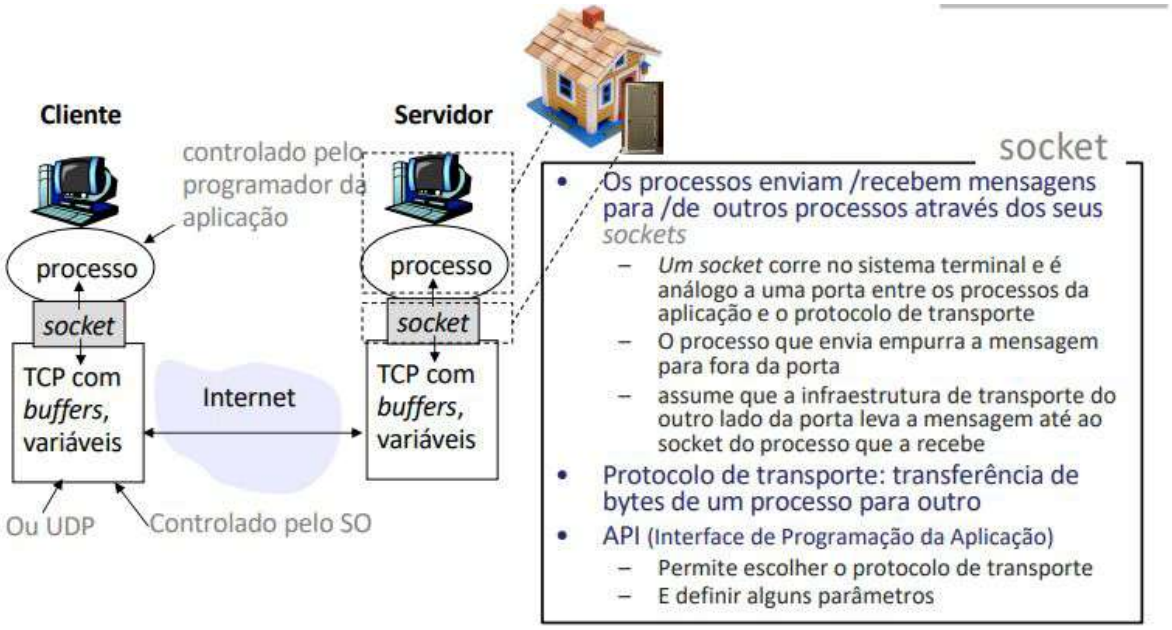
Portos = Extremidades de canais de comunicação:

- Possuem dois tipos de identificadores: O **do objeto do modelo computacional** e o **do protocolo de transporte**.

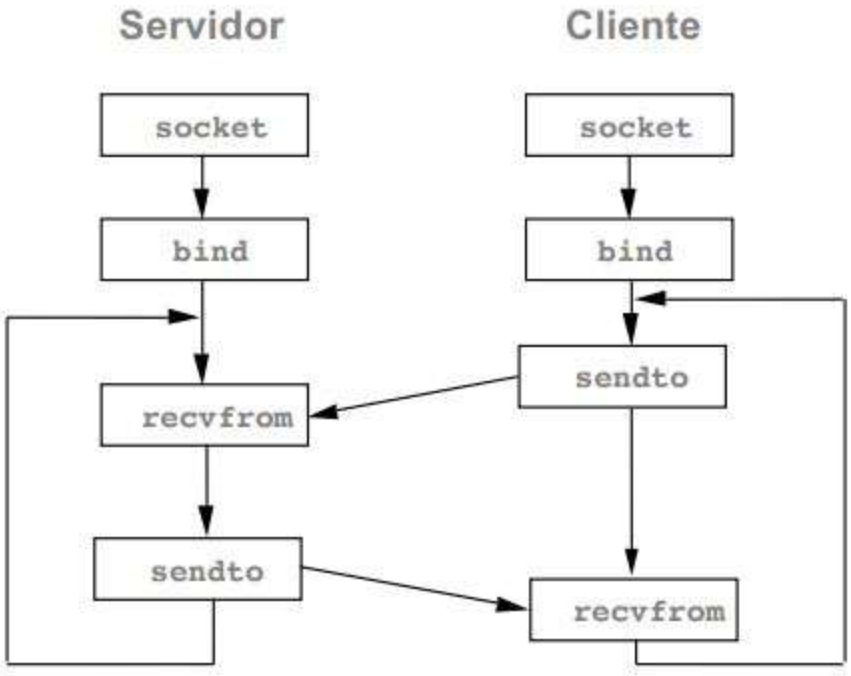
Interface sockets = Interface de programação para comunicação entre processos introduzida no Unix 4.2 BSD e standard POSIX:

- Objetivos: Independente dos protocolos. Compatível com o modelo de E/S do Unix. Eficiente.
- Dominio do socket (define a família de protocolos associado a um socket): **INET** (familia de protocolos Internet). **Unix** (comunicação entre processos da mesma máquina). **Etc…**
- Tipo do socket (define as características do canal de comunicação): **Stream** (canal com ligação, bidirecional, fiável, interface tipo sequência de octetos). **Datagram** (canal sem ligação, bidirecional, não fiável, interface tipo mensagem). **Raw** (permite o acesso direto aos níveis inferiores dos protocolos).

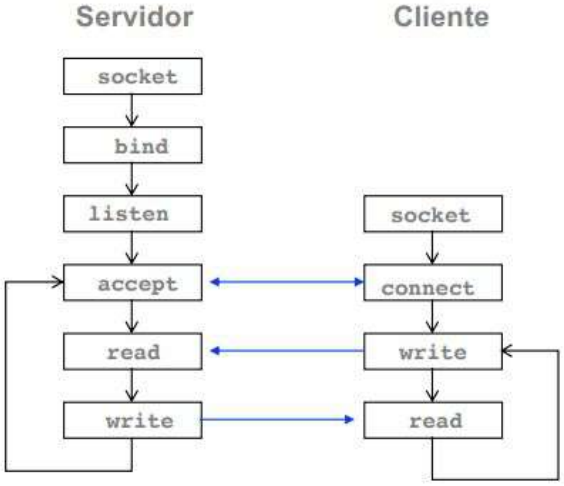
Sockets:

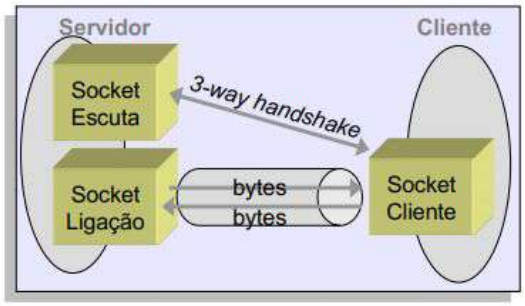


Sockets sem ligação (**UDP**):



Sockets com ligação (**TCP**):





Problemas da programação com sockets:

- É tornada **explícita** a **comunicação pela rede** (envio/receção de mensagens);
- É necessário o **marshalling/unmarshalling** (serialização/desserialização) da **informação entre sistemas** (potencialmente) **heterogéneos**: **Estruturas de dados no emissor - Stream de bytes da mensagem - Estruturas de dados do destinatário**; Necessário definir um **formato** para **representação da informação**; **Difícil** e **error-prone**.

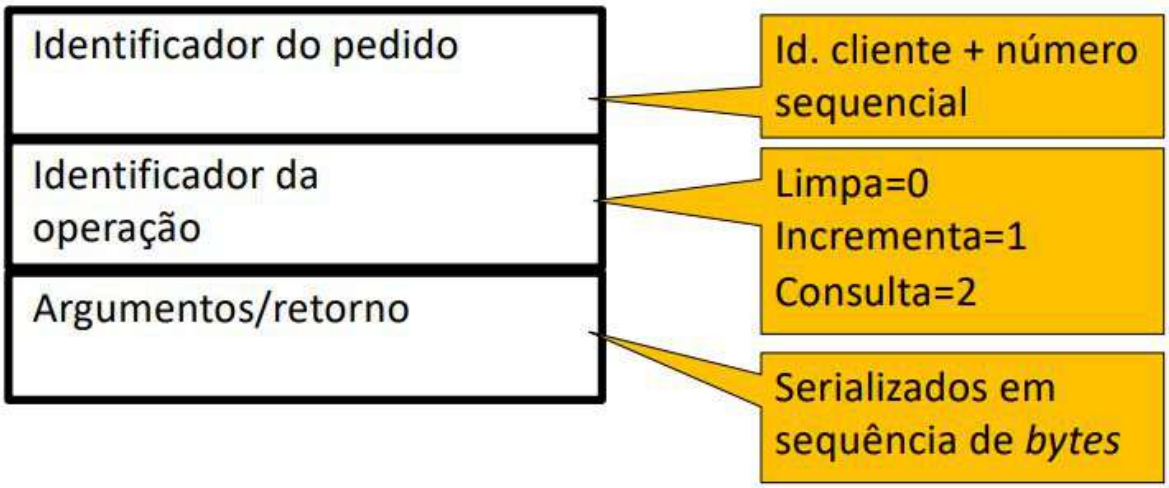
- Consequência: **Programação complexa, de baixo nível.**

2) **RPCs:** (Simplificar a programação de um Sistema Distribuído, parte A: Abstração do Sistema Distribuído como um sistema único, centralizado.)

2.1) Chamadas de Procedimentos Remotos (RPC):

O que levam as mensagens de pedido/resposta?

Exemplo:



Onde serializar os argumentos/retorno?

- **Converter estruturas de dados em memória** para **sequência de bytes** que possam ser **transmitidas pela rede** = encontrar uma forma de **emissor** e **recetor heterogéneos** interpretarem os dados **corretamente**.
- Máquinas **heterogéneas** representam tipos de formas diferentes: É necessário traduzir entre representação de **tipos do emissor** e de **tipos do recetor** ou usar um **formato canónico** na **rede**.
- **Marshaling** = serializar e traduzir para **formato canónico** (**Unmarshaling** é a operação inversa).

Problema da heterogeneidade?

- A **heterogeneidade** é a regra nos sistemas distribuidos;
- Os **formatos de dados** são **diferentes** nos **processadores** (little endian, big endian, apontadores, vírgula flutuante), nas **estruturas de dados geradas pelos compiladores**, nos **sistemas de armazenamento** (strings ASCII vs Unicode) e nos **sistemas de codificação**.

Modelo de faltas:

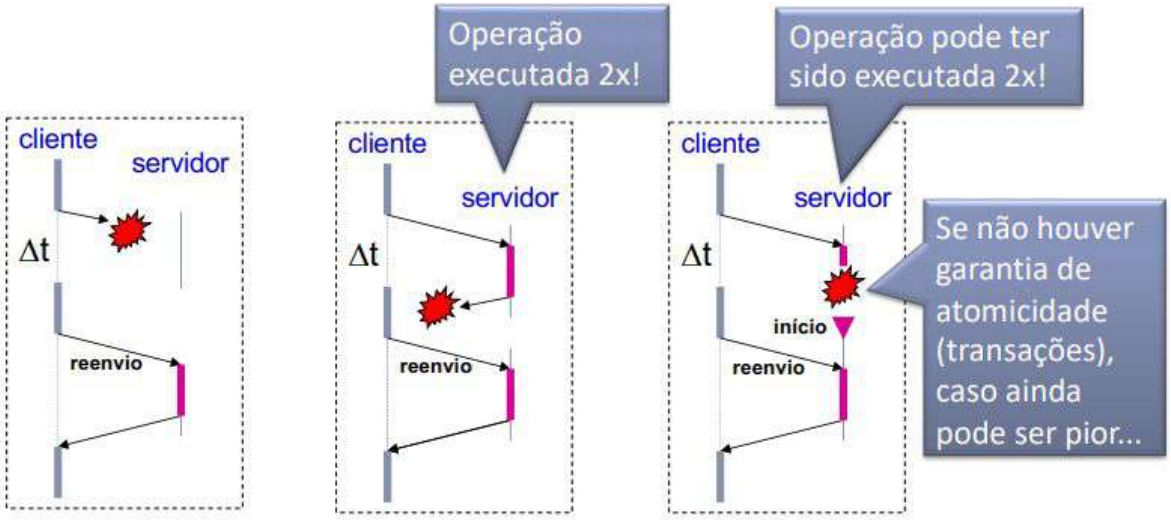
- Usando UDP para enviar mensagens (podem perder-se, chegar repetidas ou chegar fora de ordem);
- Processos podem falhar silenciosamente.

Timeout no cliente:

- Situação: Cliente envia pedido mas resposta não chega ao fim do timeout.
- Solução: Cliente deve, ou **retornar o erro**, ou **reenviar pedido**.

Timeout no cliente com reenvio:

- Quando a resposta chega após reenvio, o que pode ter acontecido?



Problema das execuções repetidas do mesmo pedido:

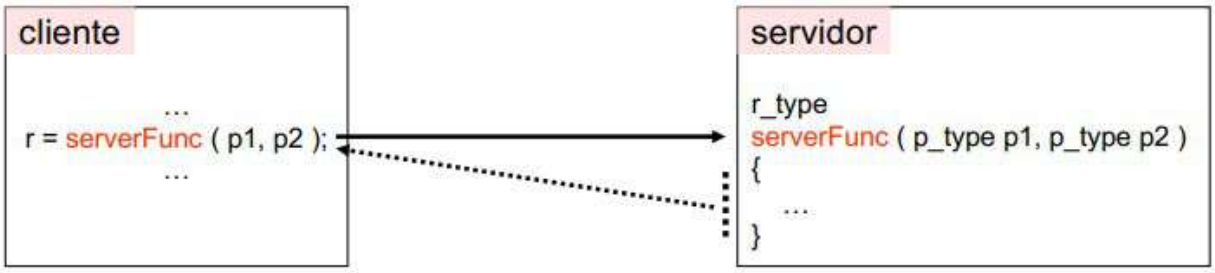
- Perde-se tempo desnecessariamente.
- Efeitos inesperado se a operação não for **idempotente**.

Idempotente = Operação que, se executada repetidamente, produz o **mesmo estado no servidor** e **devolve o mesmo resultado ao cliente**, do que **só executada uma vez**.

RPC = Modelo de programação da comunicação num sistema cliente-servidor:

- Objetivo: **Programação distribuída** com base em que clientes chamam procedimentos que se **executam remotamente no servidor**; **Abstração** do modelo **request-reply** pois existe um **pedido de execução** do procedimento e uma **resposta**.

- Visão do programador: O programador chama uma **função** (procedimento) aparentemente **local**. A função é executada remotamente no **servidor** (acendendo a dados mantidos no servidor).



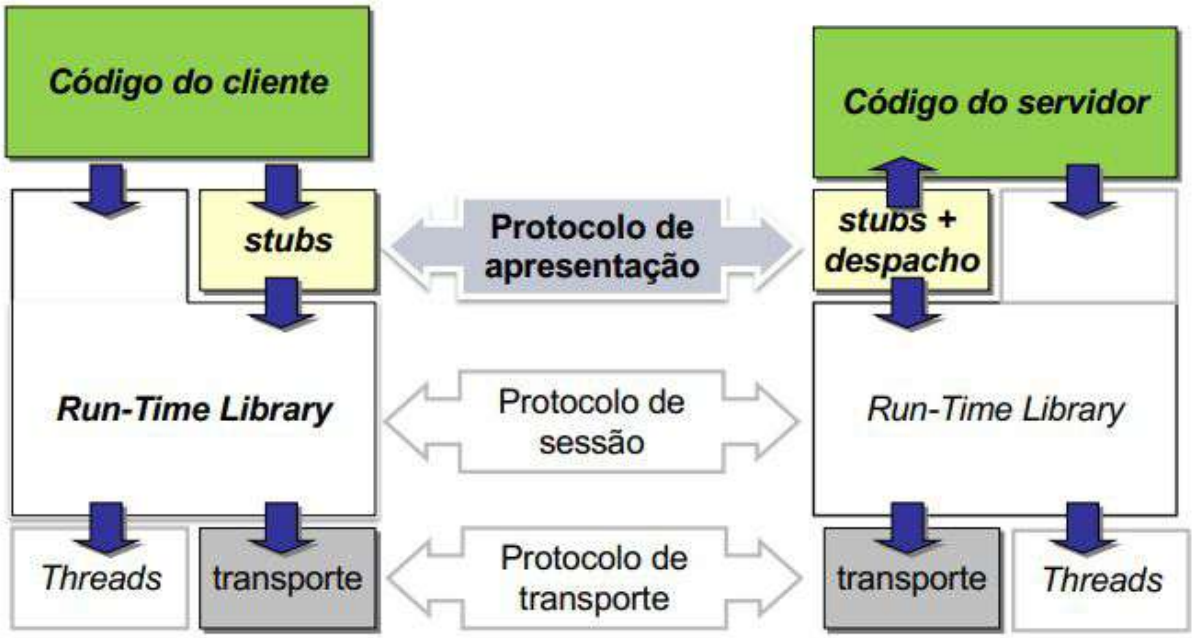
Programador preocupa-se apenas em programar a lógica de negocio. Desafios da distribuição são (quase) escondidos.

- Benefícios: Adequa-se ao fluxo de execução das aplicações; Simplifica tarefas fastidiosas e delicadas; Esconde diversos detalhes do transporte; Simplifica a divulgação de serviços (servidores).
  Fluxo de execução de uma chamada remota:

- Dificuldades que o RPC tem de resolver: Definir estrutura das mensagens (**específico**); Localizar o porto do servidor (**genérico**); Estabelecer canal de comunicação (**genérico**); Para cada pedido, criar mensagem de pedido (**específico**), converter e serializar parâmetros (**específico**), enviar, reenviar, filtrar duplicados (**genérico**) e vice-versa para a resposta (**específico**).
  Tudo isto é nos oferecido pelo RPC.
  Aspectos genéricos = Resolvidos por biblioteca de run-time;
  Aspectos específicos = Programador define a interface remota. Compilador gera código à medida.
- Estrutura:

Linguagem de descrição de interfaces remotas;
Stubs para adaptar dados de cada procedimento;
Biblioteca de run-time para o suporte genérico;
Gestor de nomes para localizar servidores.

- Tudo junto dá:

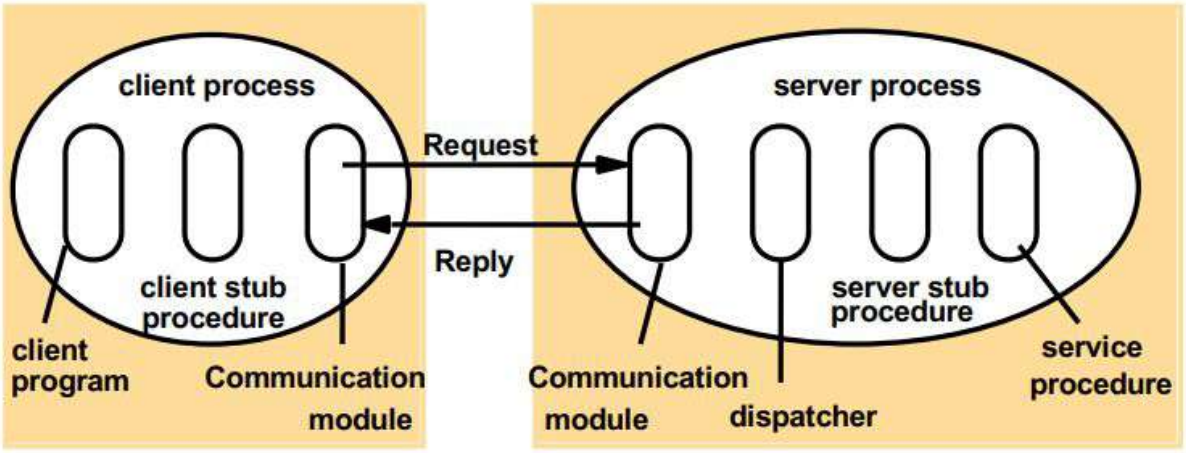


Estrutura do RPC: **Linguagem de descrição de interfaces remotas:**

- **RPCIDL =** Linguagem própria para descrição de interfaces. É uma linguagem declarativa e não descreve implementação.
  Esta linguagem permite definir **tipos de dados**, **protótipos de funções** e **interfaces remotas**.

Estrutura do RPC: **Stubs para adaptar dados de cada procedimento:**

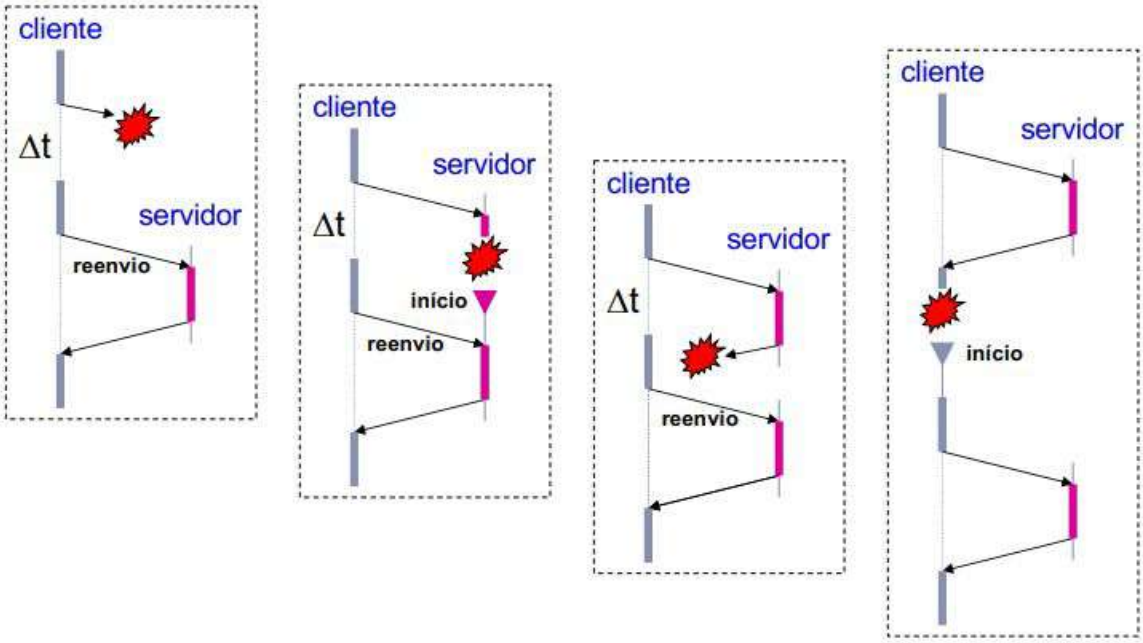
- Rotinas de adaptação (stubs): Cada função remota tem um **stub**, que é elemento chave para oferecer a ilusão de uma **chamada local**.



- Stubs cliente: Conversão de parâmetros; Criação e envio de pedido; Recepção e análise da resposta; Conversão de retorno.
- Stubs servidor: Recepção e análise do pedido; Conversão de parâmetros; Chamada da função local; Conversão de retorno; Criação e envio da resposta.

Estrutura do RPC: **Biblioteca de run-time para o suporte genérico:**

- Biblioteca de run-time = Suporta as operações genéricas do RPC:
  Localizar o porto do servidor, registar o servidor;
  Inicializar portos de comunicação;
  Estabelecer ligação entre cliente e servidor;
  Construir mensagens;
  Converter e serializar tipos primitivos de parâmetros;
  Enviar e receber mensagens;
  Implementar a **semântica de execução**.
- Semânticas de execução determinam o modelo de recuperação de faltas.
  Modelo de faltas:
  Perda, duplicação ou reordenação de mensagens;
  Faltas no servidor e no cliente;
  Possibilidade de servidor e cliente reiniciarem após as faltas.
  Perante estas faltas, como garantir uma dada semântica de execução?



- Semântica de execução do RPC:
  Sempre considerada na **ótica do cliente**;
  O modelo de faltas especifica que faltas podem ocorrer.
- <u>Semânticas de execução:</u>
  Talvez (maybe);
  Pelo-menos-uma-vez (at-least-once);
  No-máximo-uma-vez (at-most-once);
  Exatamente-uma-vez (exactly-once).

<u>Estrutura do RPC:</u> **Gestor de nomes para localizar servidores:**

- **Servidor** efetua o **registo**, **espera** por pedidos de criação de sessão, **espera** por invocações de procedimentos e **termina** a sessão.
- O **cliente** efetua o **binding** (estabelecimento da sessão-ligação ao servidor), **chama** os procedimentos remotos e **termina** a sessão.

<u>Em resumo:</u>

- A ideia chave do RPC é que fazer uma invocação **remota** deverá ser tão **simples para o programador**, como fazer uma invocação **local**.
- Desafios para garantir transparência: **Passagem de parâmetros** (dados têm de ser **serializados**); **Execução do procedimento remoto** (a rede falha… **tolerância a faltas** e notificação de faltas); **Desempenho** (depende em que grande medida da infraestrutura de comunicação entre cliente e servidor, e é sempre mais **lento** que uma invocação local).

- Infraestrutura de suporte ao RPC:
  No **desenvolvimento**, uma **linguagem de especificação de interfaces** (IDL) e um **compilador de IDL** (gerador de stubs).
  Na **execução**, uma biblioteca de suporte à execução do RPC (RPC run-time support) e um serviço de nomes.

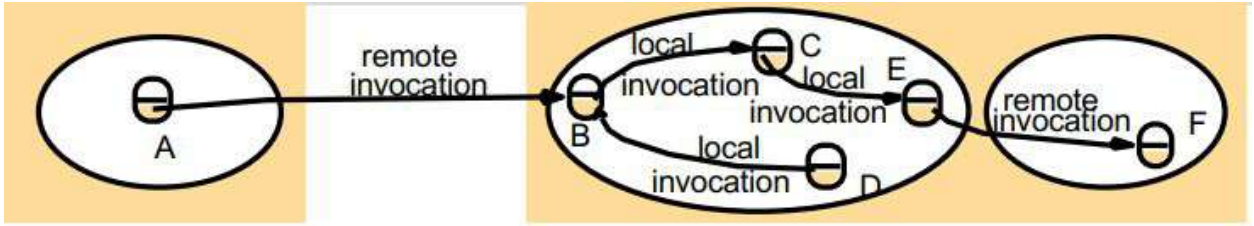
Invocação de métodos (em objetos) remotos (RMI):

- É uma **extensão** do conceito de **RPC**;
- **Num sistema de objetos** o **RPC** designa-se por **RMI**;
- Os principais sistemas são Corba, **RMI (Java Enterprise)** e Remoting (C# /.NET).

RMI vs RPC:

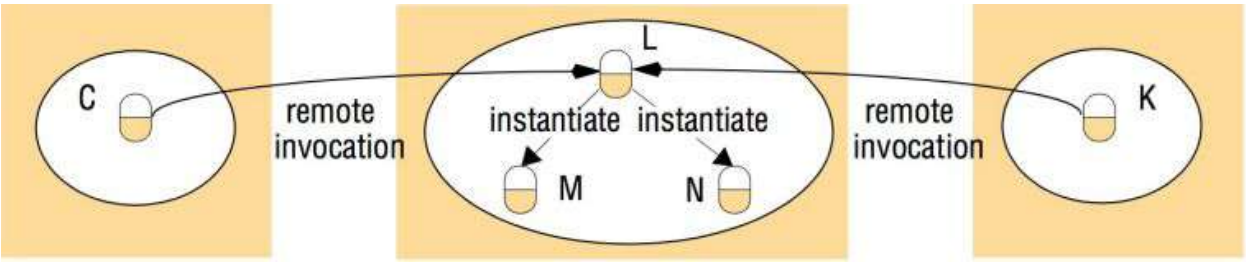
- Têm como **semelhanças** o uso de **interfaces** para definir os métodos invocáveis remotamente, em ambos existe um protocolo de invocação remota que oferece as mesmas **semânticas** e em ambos temos um **nível de transparência semelhante**.
- Têm como **diferenças** o facto de **em RMI** o programador ter ao seu dispor o **poder do paradigma OO**, as linguagens com objetos serem **fortemente tipificadas** e terem a **noção de interface** através de classes abstratas, a **passagem por referência** ser agora permitida e num servidor existirem vários **objetos remotos** e respetivas **interfaces remotas**.

Modelo de objetos distribuído:



- Objetos locais podem receber **apenas invocações locais**;
- Objetos remotos podem receber **invocações remotas e locais**.

Invocação de método:



- **Program cliente** tem referência para objeto remoto e chama método desse objeto.
- O **método** executa-se remotamente e pode **receber e retornar** referências remotas, bem como **resultar noutras invocações** a métodos noutros objetos (**locais ou remotos**) e levar à **criação de novas instâncias** de objetos (**na mesma máquina do objeto criador**).

2.2) GRPC:

- O **gRPC** é um RPC recente, que tem uma estrutura semelhante a qualquer RPC. É basicamente um RPC da era cloud, pois é dirigido a **serviços de cloud** em vez do clássico **cliente-servidor**.
- A **diferença essencial** é ter **um nível extra de indireção**:
  Em **cliente-servidor**, o **cliente** invoca um método de um **processo** a correr **numa máquina**.
  Nos **serviços cloud**, o **cliente** invoca um método de um **serviço** tipicamente implementado num número elevado de processos a correr **em máquinas distintas**.
- Muitas vezes os serviços lançam **containers**.
  Container = processo encapsulado num ambiente isolado que inclui todas as packages de software necessárias para o processo correr.
- O **gRPC** faz **outsourcing** de vários problemas para outros protocolos.

<u>Cenários de utilização do gRPC:</u>

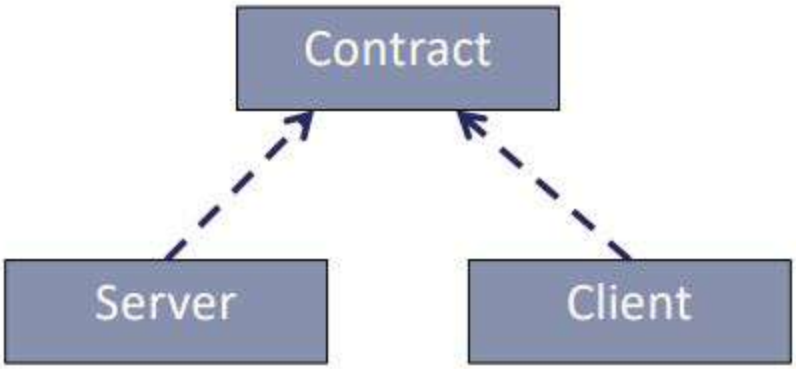
- Comunicação para Sistemas Distribuídos com requisitos de **baixa latência** e **elevada escalabilidade**;
- **Multi-linguagem**, **multi-plataforma**;
- **Exemplo de referência:** Clientes móveis a comunicar com servidores na nuvem.
- Protocolo **eficiente** e **independente da linguagem de programação**;
- Para a **Internet atual**, na qual quase todos os portos estão bloqueados, excepto para os portos Web (80, 443).

O Sistema gRPC usa uma **IDL** para definir os tipos de dados e as operações.
Disponibiliza também **ferramenta de geração de código** a partir do **IDL**, que trata da **conversão** de dados e da **gestão** da invocação remota.
Permite ter chamadas remotas **síncronas** e **assíncronas**.
Síncronas = esperam pela resposta;
Assíncronas = a resposta é verificada mais tarde.

Módulos:

- Contract = Definição da interface e tipos, na **linguagem protobuf**. Usa a ferramenta **protoc** para gerar código Java.
- Server = Implementação da interface do serviço.
- Client = Invocação do serviço.

Dependências entre os módulos:



Estrutura do gRPC: **Linguagem de descrição de interfaces remotas:**

- gRPC IDL: **Protocol Buffers:** Os protocol buffers (**protobuf**) fornecem uma IDL para **descrever mensagens e operações**. Isto permite **extender/modificar** os esquemas, mantendo compatibilidade com versões anteriores.
- O **protobuf** é um **formato canónico** (formato na rede está definido).
- A apresentação de dados tem uma **estrutura explícita**.
- O **compilador protoc** converte a IDL em código para uma grande variedade de linguagens de programação. Este pode ser chamado a partir de ferramentas, tal como o **Maven**: mvn generate -sources
- As **camadas mais baixas do gRPC** não dependem da **IDL**. Em teoria, é **possível usar alternativas ao protobuf** no **nível superior**.

Etiquetas protobuf:

- Todas as variáveis são fortemente tipificadas;

- São seguidas por um **número de etiqueta sequencial**, que é usado nas mensagens para **identificar** os diferentes campos. É necessário saber as **etiquetas** para conseguir interpretar a mensagem;
- Protobuf usa **condição explicita** devido às **etiquetas**. **Etiqueta** identifica o campo, não o seu tipo, **logo**, é preciso a definição do **protobuf** para reconstruir a **estrutura de dados** original. Um ficheiro **.proto** define os **objetos** e os **campos** desejados, bem como as operações que usam estas mensagens.

Operação gRPC: A definição de uma operação remota em **Protocol Buffers** permite apenas um **unico argumento** e um **unico resultado**:

- É preciso definir **tipo de dados** do **pedido**, bem como da **resposta**;
- Para passar múltiplos argumentos para uma operação, é necessário estarem agrupados num mesmo tipo de mensagem (o mesmo se aplica ao retorno de uma operação).

Tipos de dados do Protobuf:

- Tipos aninhados (permitem definir mensagens dentro de mensagens);
- Mapas (mapas associativos);
- Oneof (permite ter vários campos opcionais, sendo que apenas um pode ser definido - semelhante a union em C);
- Enums (alternativa a enviar strings repetidas pela rede com códigos de resultado ou mensagens de erros - o valor serializado contém apenas a etiqueta do campo);
- Service interfaces (RPC) (definição que permite o compilador de protocol buffers gerar a interface do service e stubs (cliente e service) adequados à linguagem escolhida.

Na **codificação de Protobuf**, quando cada **campo** está codificado em **binário**, é necessário incluir as **etiquetas** do ficheiro proto. Para **representar os campos**, existem **diferentes estratégias** para delimitar o início de um novo campo. Se o recetor **não conhece a etiqueta**, pode **ignorar** e **seguir para a próxima**!

Estrutura do gRPC: **Stubs para adaptar dados de cada procedimento:**

- **Stub = objeto usado pelo cliente**, que expõe a **interface do serviço**. A **invocação** de um **método no stub** vai serializar e traduzir os dados do pedido. Segue-se depois a **invocação remota**.
- É no **stub** que as **restrições** de **interface** e dos **tipos de dados** são verificadas.

Estrutura do gRPC: **Biblioteca de run-time para o suporte genérico:**

- gRPC run-time: Cabe à biblioteca de run-time a gestão do canal para realizar a chamada remota.
- Um **canal** é uma **ligação virtual** que liga o **cliente** ao **servidor**. Pode corresponder a um ou mais **sockets físicos** ao longo do tempo.
- O **transporte de dados** em **gRPC** é feito exclusivamente com **HTTP/2**.

Uma chamada remota gRPC é formada por um **nome de serviço e de método** (indicados pelo cliente), **meta-dados** (pares nome-valor), opcionalmente, e **zero ou mais mensagens de pedido** (usualmente apenas uma).
Uma chamada **termina quando o servidor responde** com **meta-dados**, opcionalmente, **zero ou mais mensagens de resposta** (usualmente apenas uma) e um **finalizador** (trailer), que indica se a chamada foi **OK** ou um **Erro**.

Estrutura do gRPC: **Gestor de nomes para localizar servidores:**

- **Resolução de nomes em gRPC:**
  - **gRPC** suporta o **DNS** como o serviço de nomes por omissão;
  - A definição de um **canal** (channel) gRPC usa a **sintaxe de URI**;
  - **Esquema comum:** dns:[//authority/]host[:port] onde port é 443 por omissão (HTTPS, seguro) ou 80 (HTTP);
  - Os **resolvers** contactam a **autoridade de resolução** e **devolvem o par** <endereço IP, porto>, juntamente com um **booleano** que indica se se trata do **servidor de destino** ou de um **balanceador de carga**.

- Resolver plug-ins: Os **resolvers** devem **contactar a autoridade** de **fazer a resolução** e **devolver** os **endereços** (IP e porto) e o **booleano**.
- A **API** permite que os **resolvers** monitorizem o **estado de uma extremidade de comunicação** e que devolvam resoluções atualizadas assim que necessário.

2.3) Web Services:

Que tecnologia usar para a comunicação remota?

- É necessário suportar a programação de comunicação entre entidades diferentes com **tecnologias diversas**.
- **Para isto é preciso:**
  Heterogeneidade das aplicações (Sistema Operativo, linguagem de programação, bibliotecas, etc…);
  Heterogeneidade dos dados;
  Heterogeneidade do transporte (protocolo de comunicação) (HTTP, e-mail, filas de mensagens, etc…).

Há portanto várias tecnologias candidatas:

- **Sun RPC**, que é antiquado, só suporta C e é pouco compatível com a Web e Internet atuais (usa portos arbitrários);
- **Java RMI**, que só suporta Java e o modelo de objetos distribuídos é complexo (uma entidade pode alojar objetos de outra entidade, e quem garante a integridade do estado do objeto?);
- **gRPC**, que é recente, o formato de dados binário dificulta inspeção e manipulação das mensagens e suporta apenas HTTP/2.

Concluindo, é necessária uma **tecnologia mais versátil, alinhada com os protocolos de comunicação da Internet e Web e independente de tecnologias específicas** (e de fornecedores).

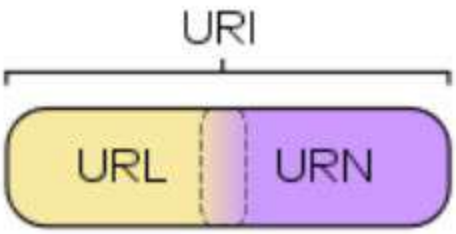
Web Service (ou serviço web, ou serviço) = Ponto de entrada numa aplicação, que encapsula uma funcionalidade (de negócio), que pode ser invocada remotamente. É definido através de uma interface, independente da implementação.

Motivação dos Web Services:

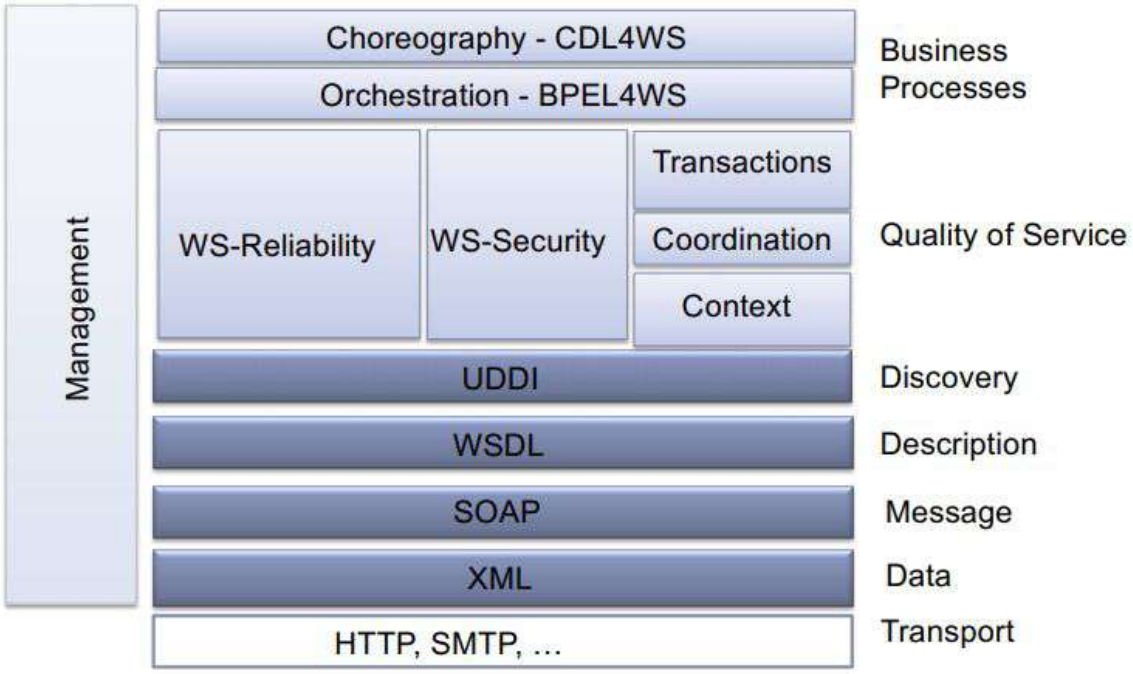
- Protocolo simples para garantir a interoperação entre plataformas de múltiplos fabricantes;
- Tratar da heterogeneidade de dados (com XML);
- Permitir a transferência de todo o tipo de dados (estruturas de dados, documentos estruturados, dados multimédia);
- Usar URL e URI como referências remotas;
- Eliminar a distinção de sistemas para transferência de documentos e para transferência de dados;
- Permitir utilizar RPC ou MOM (Message Oriented Middleware) (comunicação sincrona ou assíncrona);
- Usar protocolos de transporte amplamente conhecidos (HTTP, SMTP, MQ e outros).

Uniform Resource Identifiers (URIs):

- Standard de nomes de recursos na WWW;
- Sintaxe:
- Duas funções distintas: **Identificar univocamente um recurso na Internet** (**URNs**) e **localizar um recurso na Internet** (**URLs**).



Web Services standards completos (WS-*):



HyperText Transfer Protocol (HTTP):

- Protocolo de pedido-resposta do tipo RPC;
- Métodos remotos predefinidos: GET, HEAD, PUT, POST, etc…
- Permite parametrizar conteúdos (os pedidos dos clientes podem especificar que tipo de dados aceitam).

HyperText Markup Language (HTML):

- Essencialmente orientado à **apresentação** de informação, e não à **descrição dos dados**, pouco útil para os Web Services.

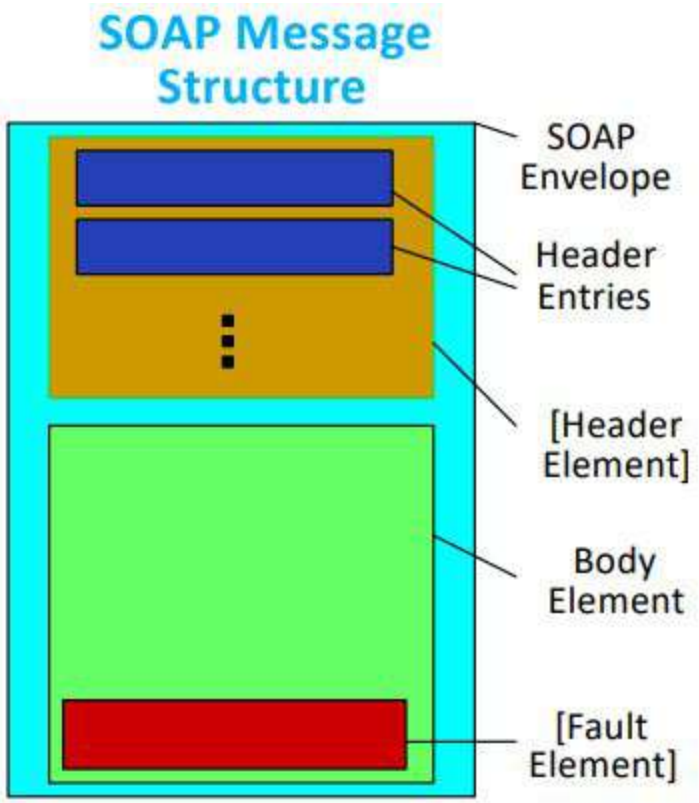
eXtensible Markup Language (XML):

- Linguagem de etiquetas, tal como HTML;
- Focada na **descrição** do conteúdo da informação, e não na sua **apresentação**.

Simple Object Access Protocol (SOAP):

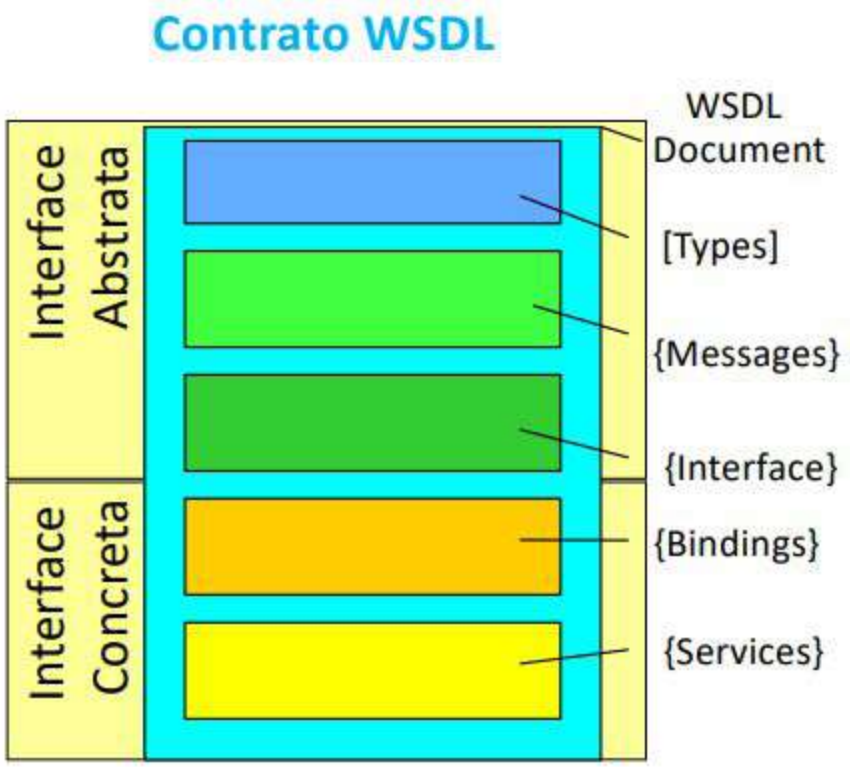
- Define estrutura de mensagens;

- Formato XML;
- Pode ser usado com praticamente todos os protocolos de transporte (HTTP, SMTP, TCP/IP, etc…).



Web Services Description Language (WSDL):

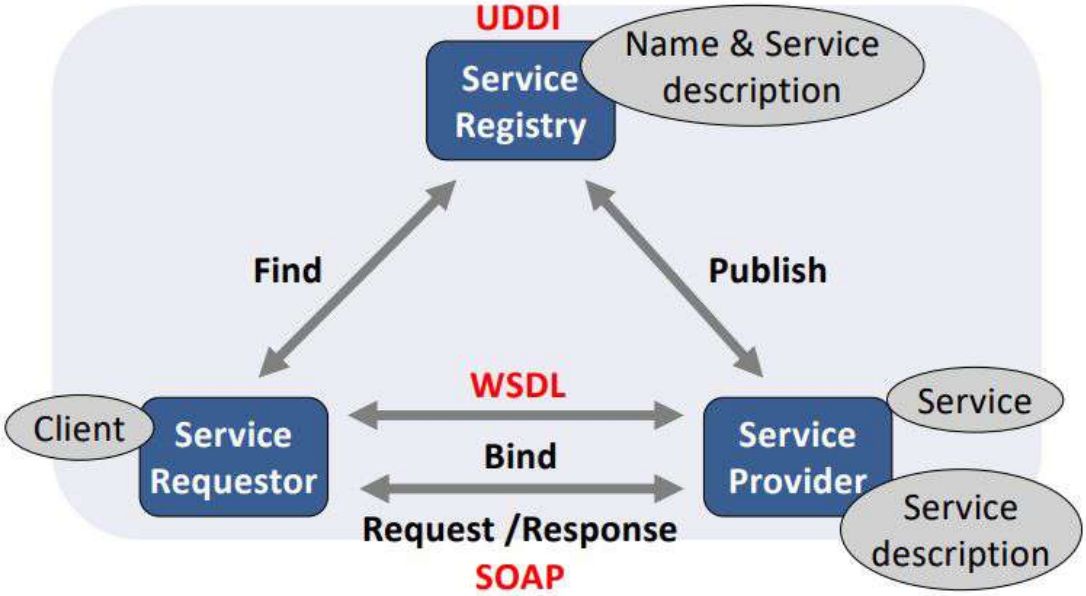
- **Acordo cliente-servidor** relativamente ao **serviço** (serve para gerar os stubs);
- Mais **flexível** que outros IDLs, para permitir vários tipos de interação (pedido-resposta e troca de documentos).



Universal Data Discovery and Integration (UDDI):

- Serviço de diretório para registro e pesquisa dos serviços disponíveis;
- Cliente pode procurar descrição de serviço (WDSL) por nome ou atributo, ou aceder diretamente ao URL.

Interação de um Web Service:



Desvantagens dos Web Services/SOAP:

- Obriga o uso de XML;
- Desempenho e escalabilidade limitados;
- Relativamente difícil de implementar;
- Normas WS-* muito complexas.

RESTful services (Implementação de serviços alternativa ao SOAP):

- REST standards base: Estilo arquitetural (não é protocolo) e tem foco na facilidade de **implementação e manutenção**, pois assume **HTTP** como transporte, assume modelo de interação **pedido-resposta** com base nos métodos HTTP e permite **diferentes representações dos dados** (XML, JSON, etc...).

Operações sobre recurso de dados em REST invocadas por HTTP.

Alternativas ao XML para representação de dados:

- JavaScript Object Notation (JSON): Tal como XML, representa registos e coleções de dados com flexibilidade, representa os dados em texto e tem uma representação explícita com etiquetas. Mas não tem (ainda) uma linguagem standard de definição de schemas.
  Tem como vantagens em relação ao XML, ser mais curto e ter uma notação mais próxima das linguagens de programação estilo C (melhor desempenho a

serializar/desserializar e melhor integração na linguagem, mais fácil para o programador).

REST dá ênfase nos dados e não na interface de um serviço.

Limitações da abordagem REST:

- Falta de contrato e de definição de interface gera ambiguidades (tanto na utilização do HTTP como no modelo de dados).

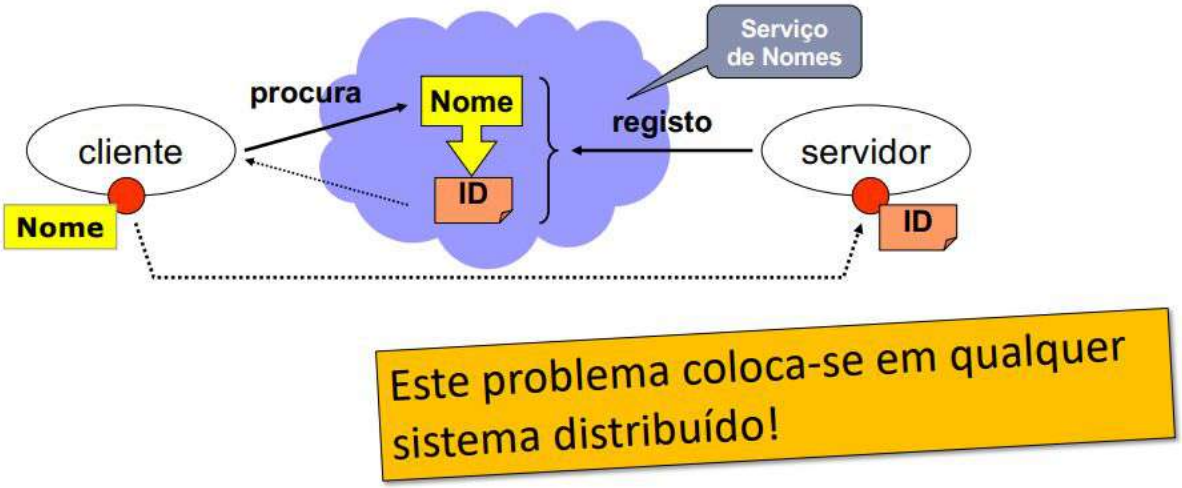
Conclusão (Web Services):

- **Para integrar aplicações,** é necessário suportar a programação de comunicação entre entidades diferentes com tecnologias heterogéneas (Java RMI limitado ao Java, gRPC limitado a HTTP/2, Sun RPC limitado ao C).
- Web Services - **tecnologia versátil:**

  Alinhada com os protocolos da Internet e Web;
  Independente de tecnologías específicas, suporta multiplos transportes;
  RESTful services (alternativa ao SOAP), de desenvolvimento mais rápido, pois adequa-se bem para oferecer acesso a dados mas tem limitações na representação de execução de operações.

2.4) Gestão de nomes:

Num sistema cliente-servidor, como pode um cliente descobrir o servidor?



Objetivo da gestão de nomes = Associar nomes a recursos (computadores, serviços, objetos remotos, ficheiros, utilizadores, etc…).

Nomes servem para:

- Localizar recurso sobre o qual se pretende atuar (**exemplo:** URL para abrir página);
- Partilhar recursos (**exemplo:** objeto remoto);
- Facilitar comunicação (**exemplo:** endereço de email)
- Associar atributos descritivos a recursos, e fazer procuras baseadas nesses atributos (**exemplo:** procurar impressora a cores na rede local).

Conceitos base:

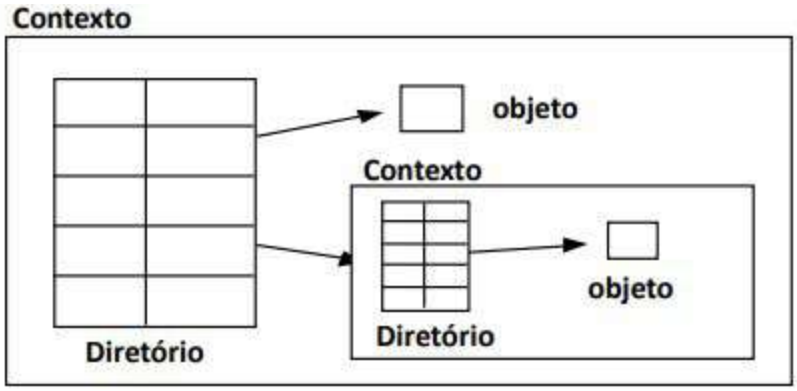
- **Nome =** mecanismo de descriminação de um objeto, usado por humanos, com carga semântica para os humanos e são sequências legiveis de caracteres;
- **Identificador =** também mecanismo de descriminação de um objeto, mas sob controlo de um sistema, sem carga semântica para os humanos e são sequências de bits;
- **Endereço =** identificador que permite encontrar diretamente o objeto (o endereço pode deixar de referenciar o objeto se este mudar de localização);
- **Espaço de nomes =** Conjunto de regras que define um universo de nomes admissíveis;
- **Autoridade =** Entidade que gere o recurso que suporta a implementação do objeto: Define as regras de gestão dos identificadores, deve garantir que as regras de gestão dos nomes são cumpridas e a autoridade pode ser delegada (hierarquias).

A partir do nome de um objeto existe uma cadeia de associações entre nomes, geralmente pertencentes a espaços de nomes diferentes.

Resolução de nome = a partir do nome obter o endereço respectivo.

Conceitos base:

- **Contexto =** Conjunto de associações pertencentes a um determinado espaço de nomes (define domínio em que se consideram válidos um conjunto de nomes).
- **Diretório =** Tabela(s) que materializa(m) as associações entre nomes e objetos de um Contexto (um Diretório também é um objeto que tem de ter um nome associado).



Propriedades de um espaço de nomes:

- Unicidade referencial;
- Âmbito de um nome;
- Homogeneidade de nomes compostos;
- Pureza de um nome.

Propriedades de um espaço de nomes: **Unicidade referencial:**

- Num determinado contexto, cada nome pode estar associado a apenas um objeto, caso contrário haveria ambiguidade referencial (onde não se poderia distinguir o objeto nem se poderia endereçá-lo).
- A situação inversa não é verdadeira. Isto é, um objeto pode estar associado a vários nomes.
- Os nomes simbolicos (alias) são normalmente nomes alternativos para um mesmo objeto no mesmo contexto.

Para garantir a unicidade referencial:

- **Atribuição de nomes globais:**

  Atribuição central = Latências elevadas, ponto único de falha (**exemplos:** endereços IP oficiais, endereços Ethernet);

  Atribuição local com difusão = Impraticável em larga escala, mas simples e prático em redes locais (**exemplo:** NetBios).
- **Soluções práticas para a atribuição de nomes globais:**

  Nomes não estruturados com grande amplitude referencial = podem ser atribuídos independentemente por qualquer contexto e podem ser gerados de forma pseudoaleatória ou mesmo totalmente aleatória (**exemplo:** GUID/UUID);

  Nomes hierárquicos = nomes globais compostos pela concatenação de nomes locais (**exemplo:** Números de telefone, nomes DNS, Nomes de Ficheiros).

Propriedades de um espaço de nomes: **Âmbito de um nome:**

- **Global (absoluto) =** tem o mesmo significado em todos os contextos onde o espaço de nomes é válido. São nomes independentes da localização do utilizador, simples de transferir entre contextos e difíceis de criar.
- **Local (relativo) =** tem diferentes significados consoante o contexto em que é usado. Permite a criação eficiente de nomes e é difícil transferir nome para outro contexto.

Propriedades de um espaço de nomes: **Homogeneidade de nomes compostos:**

- **Homogéneo =** formado por uma única componente, ou por várias componentes de igual estrutura e significado;
- **Heterogéneo =** formado por várias componentes com estruturas e significados diferentes.

Propriedades de um espaço de nomes: **Pureza de um nome:**

- **Impuro =** partes do nome são utilizadas para a sua resolução. Isto permite melhor desempenho e escalabilidade e torna a reconfiguração difícil.
- **Puro =** não contém nenhuma indicação que permita acelerar a sua resolução. Não ajuda a localizar o objeto. Permite flexibilidade de reconfiguração, mas é impraticável em larga escala.

Arquiteturas dos serviços de nomes:

- Funcionalidades:
  Registo das associações;
  Distribuição das associações;
  Resolução dos nomes;
  Resolução inversa.
- Requisitos:
  Larga escala e distribuição geográfica;
  Serviço que corre durante longo período;
  Necessidade de elevada disponibilidade;
  Isolamento de falhas;
  Suportar clientes que não confiam entre si;
  Heterogeneidade de nomes e protocolos.
- Componentes:
  Agente do serviço de nomes;
  Servidor de nomes;
  Base de dados de nomes.

Para o agente localizar o(s) servidor(es), há 3 opções:

- O endereço do servidor é fixo (well-known);
- Difusão periódica do endereço dos servidores;
- Pedido do cliente em difusão.

Modelos de resolução de nomes:

1) Baseada em difusão;
2) Iterativa;
3) Recursiva;
4) Iterativa controlada pelo servidor.

1) Resolução baseada em difusão:
   - Cliente envia nome a resolver em difusão para múltiplos servidores de nomes;
   - Quem souber responde.
   - Problemas: O que assumir quando ninguém responde? Escalavel para redes de larga escala?
2) Resolução iterativa:
   - Agente interage com vários servidores;
   - Esta é a resolução mais complexa para o agente, pois este tem de manter contexto de resolução;
   - É mais fácil de lidar com falhas.
3) Resolução recursiva:
   - Agente só interage com um servidor;
   - É a resolução mais simples para o agente;
   - É a resolução mais complexa para os servidores, pois têm de manter contexto de resolução;
   - Permite fazer caching nos servidores.
4) Resolução iterativa controlada pelo servidor:
   - Agente delega um servidor que faz resolução iterativa;
   - Combina algumas vantagens de ambos.

Serviços de nomes guardam informação essencial (recebem nome, devolvem endereço);
Serviços de diretório guardam informação mais rica (recebem pesquisas por atributo, devolver conjuntos de objetos).

Domain Name Service (DNS):

- Arquitetura para registo e resolução de nomes de máquinas da Internet;
- Exemplo de concretização: UNIX BIND;

- Espaço de nomes hierarquico e homogeneo;
- Ambito dos nomes global ou local;
- Cada contexto designa-se por dominio;
- Nomes impuros;
- Suporta resolução iterativa e recursiva controladas pelo servidor.

X.500:

- Norma que define arquitetura para um diretório de nomes em aplicações informáticas à escala mundial;
- Exemplo de concretização: Microsoft Active Directory, LDAP;
- Espaço de nomes global, hierarquico e homogeneo;
- Nomes impuros;
- Cada entrada é uma instância de uma classe (object class);
- Cada entrada da hierarquia é formada por um conjunto de atributos;
- O conjunto de classes define o esquema do espaço de nomes.

3) Tolerância a faltas:

(As **faltas** são a regra, não a exceção, e a **replicação** é a solução)

3.1) Tolerância a faltas ou confiabilidade (Dependability):

Sistema computacional:

- Um **sistema** tem uma especificação funcional do seu comportamento que define, em função de determinadas **entradas** e do seu **estado**, quais são as **saídas**;
- É formado por um **conjunto de subsistemas** (componentes);
- Tem um **estado interno**;
- Está sujeito a um **conjunto de entradas** ou **estímulos** externos;
- Tem um **determinado comportamento**: Produz **resultados** em função das **entradas** e do seu **estado interno**.

Comportamento de um sistema computacional:

- **Especificado**;
- **Observado** (Serviço cumprido ou serviço interrompido).

Sistema computacional determinístico = se as **saídas** e o **estado seguinte** forem uma **função** (determinística) dos **estímulos** e do **estado atual**.

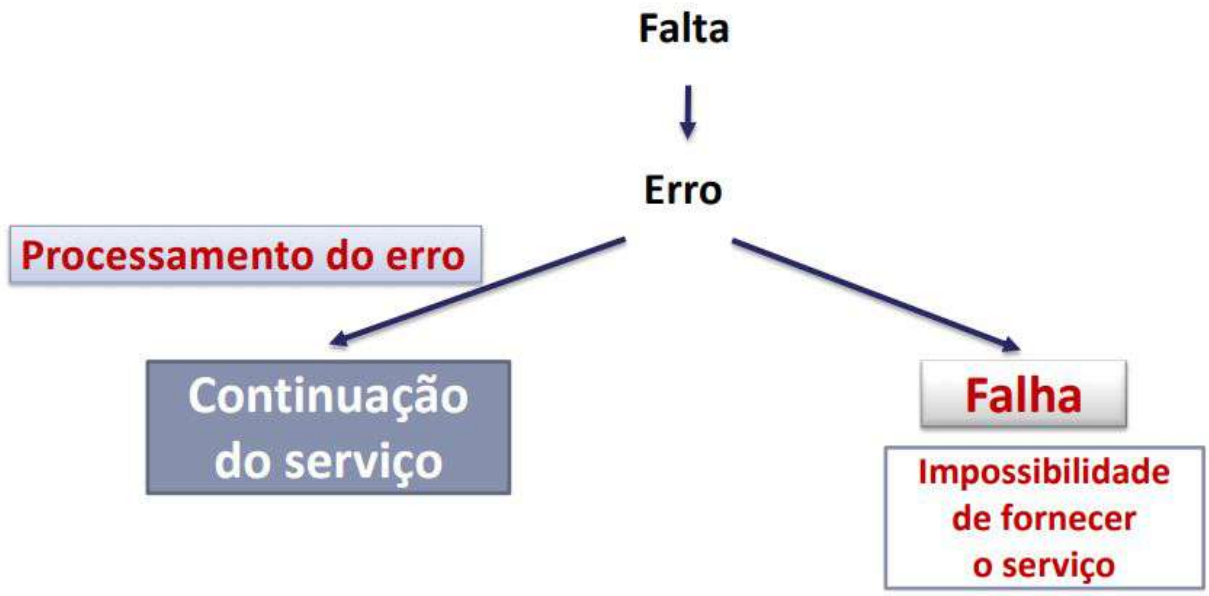
Falta (fault) = Acontecimento que altera o padrão normal de funcionamento de uma dada componente do sistema.

Erro (error) = Transição do sistema, provocada por uma falta, para um estado interno incorreto (Estado interno **inadmissível** ou Estado interno **admissível, mas não o especificado para estas entradas**).
Falha (failure) = Quando se desvia da sua especificação de funcionamento. Num determinado estado, o resultado produzido por uma dada entrada não corresponde ao esperado.

Em resumo:

- A **falta** é a **causa** de um erro;
- Uma **falha** é um **desvio do comportamento especificado** que ocorre devido a um **erro**;
- Falta -> Erro -> Falha.

Modelo de base da tolerância a faltas:



Falha de subsistema: Uma falha de subsistema é uma falta do sistema que o inclui. (Falta -> Erro -> Falha) -> (Falta -> Erro -> Falha)

Classificação de faltas: Isto permite **descrever** as suas caracteristicas.
Faltas podem ser classificadas por:

- **Causa;**
- **Origem;**
- **Duração;**
- **Independência;**
- **Determinismo.**

Tipos de faltas:

- Causa: **Falta física =** Por exemplo, fenomenos elétricos, mecânicos, etc…
  Falta humana:
  - **Acidental =** Por exemplo, mau desempenho, má operação, etc...
  - **Intencional =** Ataque premeditado.

- Origem: **Falta interna =** Por exemplo, dentro do subsistema, do programa, etc...
  Falta externa = Por exemplo, temperatura elevada, falta de energia, etc…

- Duração: **Faltas permanentes =** Mantêm-se enquanto não forem reparadas. São mais fáceis de detetar.
  Faltas temporárias ou transientes = Ocorrem apenas durante um determinado período, geralmente por influência externa. São difíceis de reproduzir e de detetar. Podem ficar reparadas imediatamente após terem ocorrido.

- Independência: **Faltas independentes =** Probabilidade de falta de uma componente é independente de outras componentes. É em geral uma boa aproximação no caso das faltas em hardware.
  Faltas dependentes ou de modo comum = Probabilidades elevadas de falta correlacionadas (por exemplo, faltas no software, etc…).

- Determinismo: **Faltas deterministicas =** Dependem apenas da sequência de entradas (inputs) e do estado. Repetindo essa sequência, reproduzimos a falta.
  Faltas não-deterministicas ("Heisenbugs") = Dependem de outros fatores não previsíveis. São difíceis de reproduzir, depurar.

Métricas para a tolerância a faltas:

- Permitem:
  Quantificar quão **tolerante a faltas** é um sistema;
  Comparar sistemas ou configurações diferentes.

Métricas de fiabilidade (reliability):

- Para **sistemas não reparáveis**, é usada a métrica **Mean Time To Failure (MTTF)**;
- Mede o tempo médio desde o instante inicial até à falha.

Métricas de fiabilidade em sistemas reparáveis:

- Em **sistemas reparáveis**, a fiabilidade é normalmente dada pelo tempo médio desde reinício correto até à proxima falha: **Mean Time Between Failures (MTBF)**
- É possível também calcular o tempo médio da reparação: **Mean Time To Repair (MTTR)**

  MTBF(horas) = (Horas a operar corretamente - Duração das falhas) / Número de falhas
  MTTR(horas) = Duração das falhas / Número de falhas
  Disponibilidade (availability) (% uptime) = MTBF / (MTBF + MTTR)

Melhoria da disponibilidade:

- **MTBF é uma medida básica da fiabilidade de um sistema:**
  Queremos que o **MTBF** seja o maior possível.
  O sistema é mais fiável se o tempo entre falhas for muito alto.
- **MTTR indica a eficiência da reparação:**
  Queremos que o **MTTR** seja o mais pequeno possível.
  O sistema é mais fiável se o tempo de reparação for pequeno.
- **Queremos disponibilidade a convergir para 100%:**
  Mas sabemos que nunca vamos lá chegar.

Classes de disponibilidade:

Classe de Disponibilidade = log10 [1 / (1 - D)] (Onde D é a disponibilidade)
A designação mais habitual é o **"número de noves"** (informal).

Tipo	Indisponibilidade (minutos/ano)	Disponibilidade D	Classe
Não gerido	52 560	90%	1
Gerido	5 256	99%	2
Bem gerido	526	99.9%	3
Tolerante a faltas	53	99.99%	4
Alta disponibilidade	5	99.999%	5
Muito alta disponibilidade	0,5	99.9999%	6
Ultra disponibilidade	0,05	99.99999%	7

1 ano tem 525 600 minutos

Modelos fundamentais:

- Antes de desenhar qualquer solução, é muito boa prática definir os **modelos fundamentais**! Ou seja, os **pressupostos**. Sem isso, a solução estará provavelmente errada (não fornecerá as propriedades desejadas).
- Três modelos fundamentais:
  Modelo de interação;

Modelo de faltas;
Modelo de segurança.

Modelo de interação:

- **Latência, que inclui:** Tempo de espera até ter acesso à rede + Tempo de transmissão da mensagem pela rede + Tempo de processamento gasto em processamento local para enviar e receber a mensagem.
- **Débito/largura de banda:** Quantidade de informação que pode ser transmitida simultaneamente pela rede.

- É o que pressupomos sobre o canal de comunicação, ou seja, Canal assegura ordem de mensagens? Mensagem pode chegar repetida? Jitter (que variação no tempo de entrega de uma mensagem é possível)?

- **Sistema síncrono =** aquele em que são garantidos os limites:
  - Cada mensagem enviada chega ao destino dentro de um tempo limite conhecido;
  - O tempo para executar cada passo de um processo está entre limites mínimo e máximo conhecidos;
  - A taxa com que cada relógio local se desvia do tempo absoluto tem um limite conhecido.
- **Sistema assíncrono =** Caso algum dos limites acima não seja conhecido.

- **Limites temporais:** Apesar de ser fácil estimar valores **prováveis** para os limites anteriores, conhecer limites **garantidos** nem sempre é possível!

Qualquer solução desenhada para sistemas assíncronos é correta num sistema síncrono.

- **Detecção de faltas de paragem num sistema síncrono:** Assume-se a existência de uma latência máxima entre nós da rede e um tempo máximo de processamento de cada mensagem. Quando os limites de tempo são ultrapassados, deteta-se a falta.
- **Detecção de faltas de paragem num sistema assíncrono:** Não é possível limitar a latência da rede ou o tempo de resposta do servidor. É impossível a detecção remota de falhas por paragem, mas é possível suspeitar das mesmas.

Modelo de faltas:

- Aqui é necessário identificar quais as faltas expectáveis, e em seguida, decidir quais são as faltas que vão ser recuperadas e quais são as faltas que não vão ser toleradas.
- **Taxa de cobertura =** Relação entre as faltas que têm possibilidade de ser recuperadas e o conjunto de faltas previsíveis.
- **As faltas que originam erros sem possibilidade de tratamento dão origem a falhas.**
- **Este modelo é muito mais complexo num sistema distribuído do que num sistema centralizado, pois vários componentes do sistema podem falhar.**

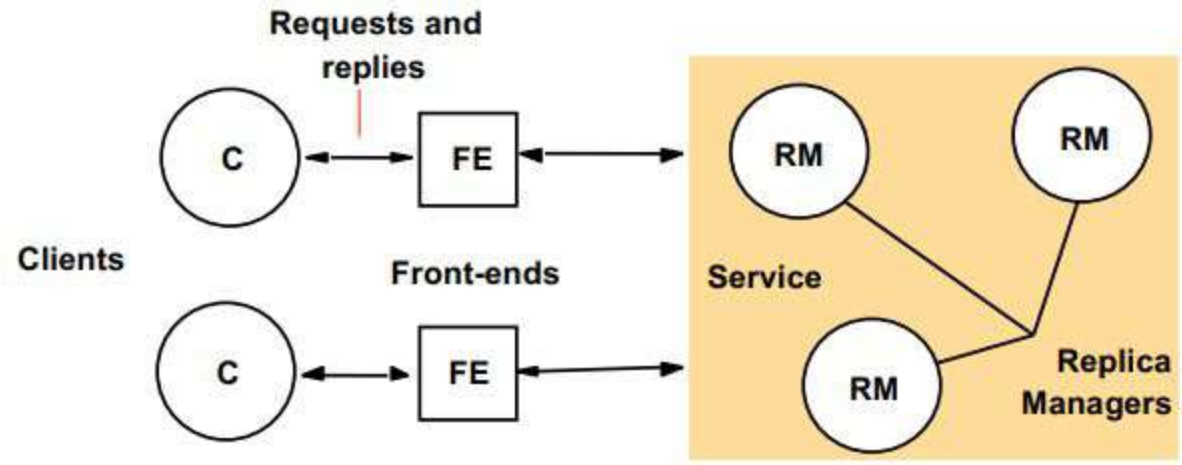
- Tipos de faltas:
  Silenciosas = Quando o componente para e não responde a nenhum estímulo externo;
  Arbitrárias/bizantinas = Quando qualquer comportamento do componente é possível (pior caso possível, mas útil para representar erros de software ou ataques).
  Por simplificação, é muitas vezes assumido que as faltas são silenciosas.

- Meios/políticas de tolerância a faltas:
  Qualquer meio de tolerância a faltas baseia-se na existência de um **mecanismo redundante** que possibilite que a função da componente comprometida seja obtida de outra forma.
  Redundância pode assumir diversas formas:
  Fisica ou espacial, com duplicação de componentes;
  Temporal, com repetição da mesma ação;
  De Informação, com algoritmos que calculam um estado correto com base no estado atual.

3.2) Replicação:

(Arquiteturas tolerantes a faltas em sistemas distribuídos)
Conceito simples: Manter **cópias** dos dados e do software do serviço em vários computadores.

Modelo arquitetural de base:



Benefícios da replicação:

- **Melhor disponibilidade - tolerância a faltas:**
  O sistema mantém-se disponível mesmo quando alguns nós falham, ou a rede falha (tornando alguns nós indisponíveis). Permite melhorar o **número de noves**
- **Melhor desempenho e escalabilidade:**
  Clientes podem aceder às cópias mais próximas de si (melhor desempenho). Algumas operações podem ser executadas apenas sobre algumas das cópias (distribui-se carga, logo consegue-se maior escalabilidade).

Requisitos da replicação:

- **Transparência de replicação:**
  Utilizador deve ter a ilusão de estar a aceder a um único objeto lógico. Objeto lógico este que é implementado sobre diferentes cópias físicas, mas sem que o utilizador se aperceba disso.
- **Coerência** (consistency)**:**
  Idealmente, um cliente que leia de uma das réplicas deve sempre ler **o valore mais recente** (mesmo que a escrita mais recente tenha sido solicitada sobre outra réplica).

Quantas faltas esperamos tolerar?

- Falhas silenciosas: Esperaríamos que **f+1 réplicas** tolerassem **f nós em falha**. Basta que **uma réplica correcta** responda para termos o valor correcto.
- Falhas arbitrárias/bizantinas: Entre as respostas que recebermos, **até um máximo de f podem ser erradas**. Logo, a única alternativa é recebermos **respostas iguais de pelo menos f+1 réplicas corretas**. Também esperaríamos que **2f+1 réplicas** tolerassem **f nós com falhas bizantinas**.
- **Apesar disto, na realidade normalmente precisamos de mais réplicas.**

As cinco fases de uma invocação num sistema replicado:

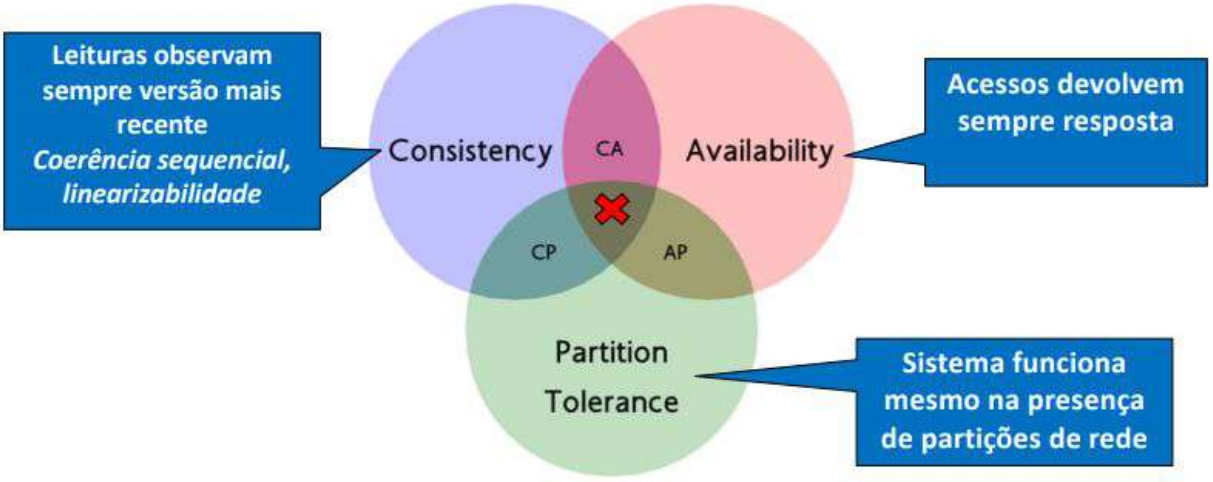
- **Pedido**: O front-end envia o Pedido **a um ou mais** gestores de réplica.
- **Coordenação**: Os gestores de réplicas coordenam-se para executarem o pedido **coerentemente**.
- **Execução**: Cada gestor de réplica **executa** o pedido.
- **Acordo**: Os gestores de réplicas **chegam a acordo** sobre o efeito do pedido.
- **Resposta**: **Um ou mais** gestores de réplica respondem ao front-end.

Pressupostos habituais:

- **Processos podem falhar silenciosamente** (não há falhas arbitrárias de processos).
- **Operações executadas em cada gestor de réplica não deixam resultados incoerentes caso falhem a meio.**
- **Replicação total** (cada gestor de réplica mantém cópia de todos os objetos lógicos).
- **Conjunto de gestores de réplica é estático e conhecido *a priori*.**

Limites da replicação:

- Idealmente, um sistema replicado deveria oferecer **Coerência forte** (Consistency), **Alta disponibilidade** (Avaliability) e **Tolerância a partições** (Partition tolerance). No entanto, segundo o **Teorema CAP**, qualquer sistema só pode alcançar 2 dessas 3 propriedades:



Ou seja, na presença de uma **partição de rede**, o **sistema replicado** precisa de **escolher** entre:

- Garantir que as respostas são **sempre coerentes** (retornar erro ou não responder quando não for possível garantir isso) - Foco na **coerência**;

- **Responder aos pedidos** de qualquer front-end sem dar erro (por vezes retornando valores desatualizados) - Foco na **disponibilidade**.

Tipos de replicação (exemplos):

- **Coerência fraca:** Gossip;
- **Coerência forte:** Primary-backup, Registo distribuído coerente, Replicação de máquina de estados.

4) Memória partilhada: (Simplificar a programação de um SD, parte B: tolerando faltas, usando replicação com coerência fraca)

4.1) Registos distribuídos (Memória Partilhada Distribuída):

Processador P3 (IAC) contém os **registos** genéricos R0 a R7
Processadores x86 contêm vários **registos** genéricos: rax, rbx, rcx, etc...
Ambos suportam a operação MOV, que **copia** para o primeiro elemento o conteúdo do segundo elemento.

Que operações se podem fazer sobre esses registos?
Leituras: MOV *destino*, registo
Escritas: MOV registo, *origem*

Registos:

- Podem ser generalizados para **variáveis** de diversos tipos.
- Leituras e escritas em registos são um mecanismo de programação bem conhecido.
- São fáceis de **replicar** para tolerância a faltas e desempenho.

Replicação Coerente e Disponível:

- **Coerência forte:**
  Leituras observam sempre versão mais recente.
- **Alta disponibilidade:**
  Qualquer acesso recebe uma resposta que não é “erro”.

Replicação ativa:

- Servidores fornecem um **serviço;**
- Clientes e servidores incluem uma **biblioteca** que executa o protocolo (papel semelhante aos stubs dos RPCs).

Protocolos de replicação ativa:

- As **réplicas** são todas idênticas e executam em paralelo o mesmo **serviço** como máquinas de estado determinísticas.
- Cliente envia mensagem às réplicas.
- Cada réplica executa a mensagem e envia a resposta.
- O cliente espera por um conjunto de respostas e retorna uma delas.

Registo distribuído:

- **Sistema replica apenas um registo:**
  Cada servidor contém uma réplica local do registo.
- **Suporta apenas duas operações:**
  Leitura do registo replicado**:** val = read();
  Escrita no registo replicado**:** ack = write(new_val).
- **Dadas duas operações de leitura ou escrita op1 e op2:**
  op1 **precede** op2 se terminar antes de op2 começar;
  op1 e op2 são **concorrentes** se nenhuma precede a outra.
- **Queremos coerência forte: coerência sequencial / registo atómico:**
  As operações parecem ter sido feitas por ordem ou com rigor:
  Leitura retorna um valor escrito concorrentemente ou o valor escrito pela última escrita que o precedeu;
  Se a **leitura L1** lê o valor da **escrita E1**, a **leitura L2** lê o valor da **escrita E2** e **L1** precede **L2**, então **E2** não precede **E1**.

<u>Modelos a assumir:</u>

- **Faltas silenciosas / crash:**
  Máquinas podem parar, mas não fazem mais nada. Será um modelo realista?
- **Sistema asíncrono:**
  Comunicação e processamento podem demorar um tempo arbitrário. Será um modelo realista?
- **Comunicação fiável:**
  Mensagens enviadas são sempre recebidas, desde que o remetente e o destinatário não falhem. Não há garantia de receção FIFO. Será um modelo realista?

Protocolo write-all-avaliable:

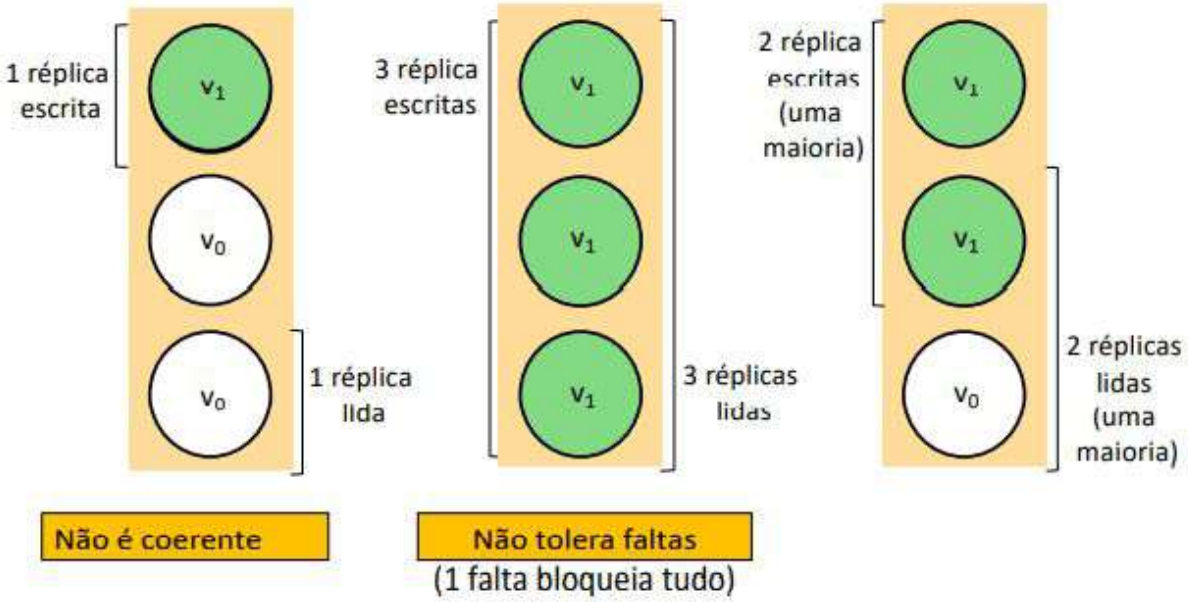
- **Escrita:** para escrever, o cliente:
  Envia pedido de escrita para todas as réplicas;
  Cada réplica que recebe o pedido, escreve o novo valor no seu registo local e responde “ack”;

Quando receber “ack” de uma réplica, cliente dá a escrita como terminada.

- **Leitura:** para ler, o cliente:
  Envia pedido de leitura para todas as réplicas;
  Cada réplica que recebe o pedido responde com o valor atual do registo local;
  Cliente espera pela primeira resposta e retorna-a.
- **Vantagens aparentes:**
  Grau de replicação **ótimo**, visto que f+1 réplicas toleram f faltas (isto é, f servidores pararem), tanto para efetuar escritas como para efetuar leituras, mas podem existir mais do que f+1 servidores.
  Operações **muito rápidas**, visto que basta receber a primeira resposta (que tipicamente chega da réplica mais próxima do cliente e/ou menos sobrecarregada) e cliente retorna.
- **Desvantagens aparentes:**
  Caso uma réplica não receba um pedido de escrita e depois responda à leitura, devido a mensagem atrasada ou falha da réplica, **réplica pode levar cliente a retornar leitura incoerente!**
  Mesmo quando não há falhas, se houver **múltiplos pedidos de escrita concorrentes**, **réplicas** podem ficar **incoerentes** e **leituras posteriores retornam valores diferentes!**

Protocolo Registo Coerente - versão básica:

- **De quantas réplicas ler/escrever?**



- **Sistema de Quóruns:**

Conjunto de **subconjuntos das réplicas** (**quóruns**), tal que **quaisquer dois subconjuntos** se **intersectam**.
Por exemplo: dadas N réplicas, qualquer quórum Q satisfaz |Q| > N/2

- **Modelo de faltas:**
  O mesmo (**Sistema assíncrono**, **Clientes/Servidores**, **Comunicação fiável**).
- **Registo Coerente:** Versão básica: um escritor / múltiplos leitores:
  Objetivo: Registo distribuído com coerência forte.
  Grau de replicação: N = 2f+1 => |Q| = f+1, pois qualquer quórum Q tem de satisfazer |Q| > N/2.
  Cada réplica guarda valor do **registo** e uma **tag** associada ao registo.
- **Réplicas:**
  Cada réplica guarda **val** (valor do registo) e **tag** (identifica a versão).
  Tag é composta (apenas) por **seq** (número de sequência da escrita que deu origem à versão.
  Dizemos que **tag1** é maior que **tag2** se e só se **seq1** > **seq2**.
- **Leituras:**
  Cliente envia read() para todas as réplicas. Aguarda por respostas de um quórum (ou seja, f+1 respostas). Seja maxVal o valor que recebeu associado à maior tag. Retorna maxVal.
  Réplica, ao receber **read()**, responde com **<val, tag>**.
- **Escritas:**
  Cliente executa leitura para obter a maior tag maxtag = <seq>. newtag = <seq+1>. Envia **write(val,newtag)** a todas as réplicas. Espera por acks de um quórum (ou seja, f+1 acks). Retorna ao cliente.
  Réplica, ao receber **write(v,t):**

```
se (t > tag) {
        val = v
        tag = t
}
Responde ack
```

- **Dúvidas frequentes:**
  Na leitura o cliente tem que esperar f+1 respostas e destas, escolhe sempre o valor com maxTag, mesmo que nas respostas exista uma maioria de réplicas com outra tag. O modelo não permitam que nos mintam (falta arbitrária), logo maxTag é mesmo a maior tag.
  Na escrita, uma escrita implica **sempre** uma fase de leitura para determinar o valor de maxTag, para garantir que a tag é maior do que a de todas as réplicas. À **fase de leitura** segue-se a **fase de escrita** com o valor de **newTag**.

- **O protocolo garante que** quando um cliente lê um registo e não está a decorrer nenhuma escrita concorrente no registo, o valor devolvido vai corresponder ao valor mais recente.
- **Observação chave:** cada **quórum de escrita** tem **pelo menos uma** réplica em comum com cada **quórum de leitura ou quórum de escrita**. Isto é garantido através de N = 2f+1 e |Q| = f +1

Protocolo Registo Coerente - versão completa:

- **Versão múltiplos escritores / múltiplos leitores:**
  Problema: dois escritores podem escolher o mesmo seq durante 2 escritas concorrentes;
  Solução: Tag passa a ser composta por **seq** (número de sequência da escrita que deu origem à versão) e **cid** (identificador do cliente que escreveu essa versão);
  Dizemos que tag1 é maior que tag2 se e só se: seq1 > seq2, ou (seq1 = seq2 e cid1 > cid2).
- **Leituras:**
  Cliente envia **read()** para todas as réplicas. Aguarda por respostas de um quórum (ou seja, f+1 respostas). Seja maxVal o valor que recebeu associado à maior tag. Retorna maxVal.
  Réplica, ao receber **read()**, responde com <val,tag>.
  É portanto igual ao da versão básica (só 1 escritor).
- **Escritas:**
  Cliente executa leitura para obter a maior tag maxtag = <seq,**cid**>. newtag = <seq+1, **meu_cid**>. Envia **write(val,newtag)** a todas as réplicas. Espera por acks de um quórum (ou seja, f+1 acks). Retorna ao cliente.
  Réplica, ao receber **write(v,t)**:

```
se (t > tag) {
        val = v
        tag = t
}
Responde ack
```

  As diferenças em relação ao da versão básica (só 1 escritor) são **cid** e **meu_cid** em maxtag e newtag.
- **Se houver uma ou mais escritas concorrentes com uma leitura, o valor devolvido pela leitura pode ser o da escrita mais recente ou o de uma das escritas concorrentes.**
- **Problema:**

Cliente que execute uma **sequência de leituras concorrentes com escrita(s)** pode obter resultados incoerentes por exemplo:
  - 1ª leitura devolve valor de uma escrita em curso;
  - 2ª leitura recebe respostas de réplicas desatualizadas.
- **Para corrigir o problema das leituras incoerentes, cliente ajuda a completar a escrita em curso ou incompleta (writeback, que no fim de cada leitura faz uma escrita na qual envia a última tag lida (a maior das que foram recebidas).**

Resultado: todas as leituras posteriores (não concorrentes) vão obter esse valor e não um valor anterior.

Protocolo Registo Coerente: variante com writeback:

- **Leituras:**
  Cliente envia **read()** para todas as réplicas. Aguarda por respostas de um quórum (ou seja, f+1 respostas). Seja maxVal o valor que recebeu associado à maior tag (maxTag). **WRITEBACK:** Envia **write(MaxVal, maxTag)** a todas as réplicas. Espera por acks de um quórum (ou seja, f+1). Retorna maxVal. Caso o valor mais recente seja de uma escrita que ainda não chegou a f+1 réplicas, assegura que essa escrita chega a esse número de réplicas.
  Réplica, ao receber **read()**, responde com <val,tag>.
- **Protocolo de escrita não muda.**

Vantagens do Registo Coerente:

- Primeiro protocolo que aprendemos que tolera faltas silenciosas em sistemas assíncronos;
- Réplica que falhe temporariamente e recupere está imediatamente pronta para participar (ficará naturalmente atualizada quando receber o próximo pedido de escrita).

Desvantagens do Registo Coerente:

- Várias réplicas para tolerar algumas faltas (Protocolo “caro”). Com quóruns de maioria, precisamos de 2f+1 réplicas para tolerar f faltas de réplicas.
- Leituras implicam respostas de múltiplas réplicas (em muitos sistemas, leituras são predominantes, logo o ideal seria permitir que leitura retornasse após resposta de uma réplica apenas).

<u>Variantes ao protocolo:</u>

Pesos variáveis:

- Cada réplica tem um peso não negativo (soma total de pesos é conhecida *a priori*).
- Um quórum passa a ser qualquer conjunto de réplicas tal que a soma do peso do quórum é superior a (peso total do sistema)/2.
- Permite dar maior peso a réplicas (mais fiáveis, com melhor conectividade ou com maior poder computacional).

Quóruns de leitura e escrita:

- O peso exigido para cada tipo de operação passa a ser distinto: read threshold (RT) para leituras, write threshold (WT) para escritas.
- Estes parâmetros têm de assegurar que: RT+WT > peso total do sistema, e que WT > peso total do sistema / 2.
- Interessante porque permite otimizar uma operação, à custa da outra. Por exemplo, em sistemas em que as leituras são mais frequentes, podemos ter RT << WT.

<u>Para além do Registo Coerente:</u>

- **Tolerância a f réplicas bizantinas:**
  Várias soluções disponíveis, normalmente baseadas em replicação ativa. Quóruns maiores. Necessário acautelar o pior caso (mesmo que o quórum contenha as f réplicas bizantinas, h+a também réplicas corretas em número suficiente no quórum. Normalmente N = 3f+1.
  Mensagens autenticadas para evitar que réplicas bizantinas enviem mensagens em nome de réplicas corretas.

4.2) Replicação Fracamente Coerente:

Replicação Fracamente Coerente: (A+P)

- **Alta disponibilidade:** Qualquer acesso recebe uma resposta que não é “erro”.
- **Tolerância a partições:** Sistema funciona mesmo na presença de partições de rede, ou seja, apesar de um número arbitrariamente alto de mensagens se perderem ou atrasarem.

Modelo do sistema:

- Múltiplas réplicas, múltiplos clientes;
- Cada réplica tem um **id numérico** = 0, 1, 2, 3, …
- Sistema **asíncrono**, onde podem ocorrer partições de rede;
- Rede assegura **entrega FIFO**;
- **Faltas silenciosas.**

Operações sobre o sistema replicado:

- **Queries (leituras);**
- **Updates:** Operações que modificam estado replicado. Podem ser operações **mais complexas** que simples write num registo (**por exemplo:** insert(element, map)).

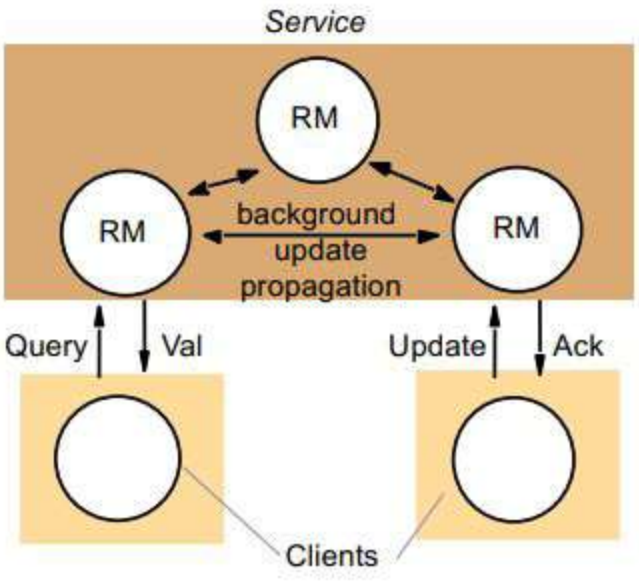
Esboço inicial de um protocolo: Interação cliente - réplica:

- **Cliente envia pedido de operação a uma réplica r,** por exemplo, à mais "próxima". Mesmo cliente pode contactar diferentes réplicas ao longo do tempo.
- **Réplica r que recebe o pedido do cliente:**
  Caso seja update, atribui-lhe um **Identificador** $u_{rep,seq}$**,** onde **rep** é o **identificador da réplica** e **seq** é o **número de sequência, incrementado para cada update emitido pela réplica.**

Esboço inicial de um protocolo: Propagação de updates:

- Feita em background, **assincronamente.**
- Replica A liga-se à Réplica B.
- A envia a B sequência de updates, na ordem pela qual A os mantém no seu log local.
- Pode ser usado algum mecanismo de filtragem para evitar que A envie updates que B já conhece.

Esboço inicial do protocolo: Visão geral:



Vantagens: - Alta **disponibilidade**, mesmo na presença de **partições de rede**;

- Acessos podem ser respondidos imediatamente pela réplica **mais próxima** do cliente (coordenação com o resto das réplicas feita em **background**);
- Vantagens muito importantes em **sistemas geo-replicados**.

Desvantagens: coerência fraca:

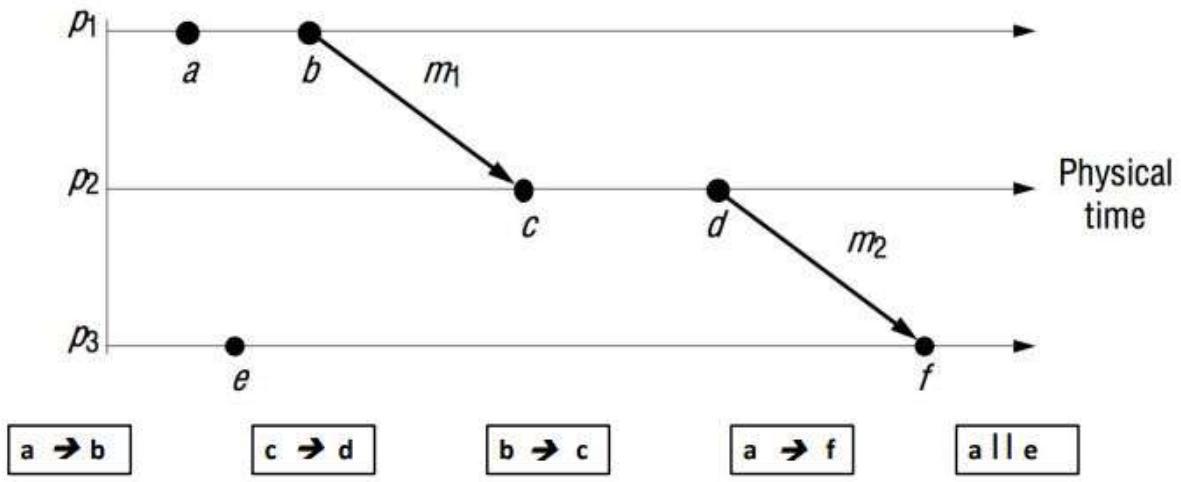
- Réplicas podem ter **vistas desatualizadas ou divergentes. Isto é,** uma réplica **não espera por coordenação** global antes de responder ao cliente, e as réplicas **não** executam necessariamente as operações na mesma **ordem.**
- **Temporariamente,** é possível a uma **réplica não ter os updates** que foram recentemente emitidos por outras réplicas. Esta sitação é resolvida quando as réplicas propagarem os updates entre si.
- É possível uma réplica **ordenar** certos updates de forma **diferente** de outra réplica.

Há aplicações que aceitam estas anomalias de incoerência, pois não são prejudicadas por isso.

Para resolver o problema da causalidade, normalmente basta que os processos cheguem a um acordo em relação à **ordem** com que ocorrem **certos eventos**, e não ao seu momento exato.
Para isso, podemos usar a relação **happens-before:**

1) Se **a** e **b** são eventos do mesmo processo, e **a** ocorre antes de **b**, então **a -> b**
2) Se **a** indica um evento envio de mensagem, e **b** o evento da receção dessa mesma mensagem, então **a -> b**.

Propriedades:
Transitividade: se **a -> b** e **b -> c**, então **a -> c**
Eventos concorrentes: nem **a -> b**, nem **b -> a**, então **a || b**
Exemplo:

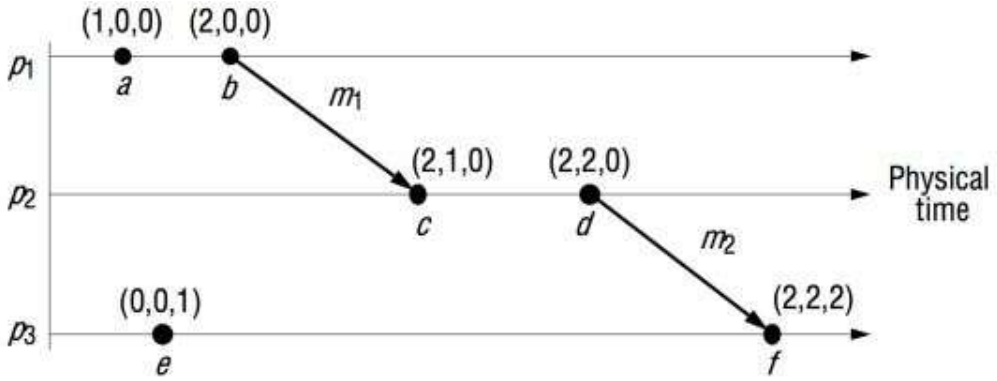


Implementação:

Timestamp vetorial V: vetor com N inteiros (para os N processos), em que cada processo i mantém o seu vetor V_i e envia-o em todas as mensagens;
Implementamos da seguinte forma:

1) Processo i coloca todos os elementos do seu vetor V_i a zero;
2) Sempre que ocorre um evento: $V_i[i] = V_i[i]+1$;
3) O vetor $t=V_i$ é incluído em todas as mensagens enviadas;
4) Quando um processo recebe uma mensagem, além de incrementar o seu contador (como fazemos no passo **2)**), atualiza o seu vetor: $Vi[j]=max(Vi[j],t[j])$ para j=1,2,...,N

Vi<Vj se pelo menos um elemento de Vi for menor e nenhum for maior que Vj



$x \rightarrow y$ então $V(x) < V(y)$

$V(x) < V(y)$ então $x \rightarrow y$

Neste exemplo, o evento (0,0,1) é **concorrente** com os que ocorrem nos outros processos

Conclusões:

- **Muitos protocolos de replicação otimista conseguem:**
  Prevenir muitas das anomalias de coerência que podem surgir com protocolos que oferecem garantias A+P;
  No exemplo acima vimos como prevenir duas importantes anomalias, mas sistemas mais recentes conseguem melhor;
  No entanto, outras anomalias continuam a poder ocorrer (ao contrário de sistemas com coerência forte).
- **Muitas aplicações aceitam coerência fraca para as estruturas de dados cuja coerência não é crítica.**

Gossip architecture: (Exemplo de um sistema replicado otimista)

- **Objetivo:** Oferecer sempre acesso rápido aos clientes, mesmo em situações de partições, sacrificando a coerência.
- As réplicas propagam os novos updates entre si periodicamente, em background (como se fosse o espalhar de um rumor, daí o nome **gossip**).

Funcionamento base:

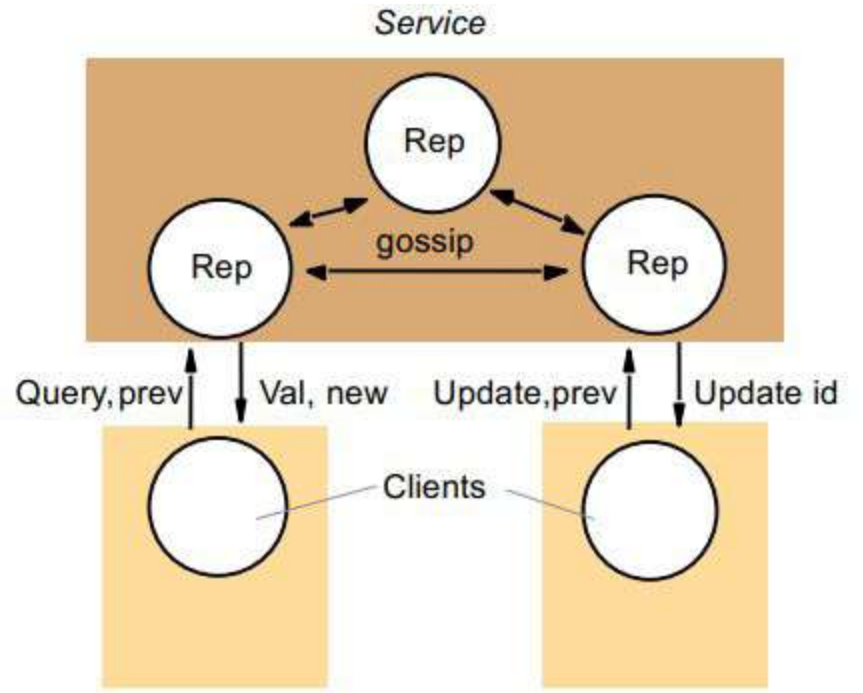
- Clientes enviam pedidos (leitura ou update) a uma réplica próxima.
- Réplicas propagam updates de forma relaxada, podendo ter vistas divergentes.

Coerência fraca, mas com duas garantias:

- Mesmo que cliente aceda a diferentes réplicas, os valores que lê são coerentes entre si (Cliente nunca lê um valor recente e depois um valor mais antigo);
- Estado de uma réplica respeita sempre a **ordem causal** entre updates (isto é, se um update **m2** depende de outro **m1**,réplica nunca executa **m2** sem antes ter executado **m1**).
- Opcionalmente, também permite acessos com coerência forte (Obrigam a maior coordenação, logo mais lentos e não toleram partições).

Algoritmo: interação cliente-réplica:

- Cada cliente mantém um **timestamp vetorial** chamado **prevTS** (Vetor de inteiros, um por cada réplica, que reflete a última versão acedida pelo cliente).
- Em cada pedido a uma réplica, cliente envia (pedido,**prevTS**).
- Réplica responde com (resposta,**newTS**): **newTS** é o timestamp vetorial que reflete o estado da réplica.
- Cliente atualiza **prevTS** com **newTS** (para cada entrada **i**, atualiza **prevTS[i]** se **newTS[i] > prevTS[i]**.



Update log: Util caso réplica já tenha recebido um update mas não o tenha podido executar pois falta receber/executar dependências causais. Nesse caso, o update está pendente no log mas ainda não foi executado.
Permite propagar updates individuais às restantes réplicas.

5) Coordenação:

(**Simplificar** a programação de um Sistema Distribuido, parte C - replicação com **coerência forte**)

5.1) Replicação de máquinas de estados:

Replicação Coerente e Disponível: (C+A)

- **Coerência forte:** Leituras observam sempre versão mais recente. Coerência sequencial, linearizabilidade, etc…
- **Alta disponibilidade:** Qualquer acesso recebe uma resposta que não é “erro”.

Replicação ativa: - Servidores fornecem um **serviço**;

- Clientes e servidores incluem uma **biblioteca** que executa o protocolo, eles desempenham um papel semelhante aos stubs dos RPCs.

Protocolos de replicação ativa:

- As **réplicas** são todas idênticas e executam em paralelo o mesmo **serviço** como máquinas de estado determinísticas;

- Cliente envia mensagem às réplicas;
- Cada réplica executa a mensagem e envia a resposta;
- O cliente espera por um conjunto de respostas e retorna uma delas.

Replicação de máquinas de estados:

- Solução **genérica** para a concretização de **serviços tolerantes a faltas**. Permite por exemplo replicar qualquer servidor de RPC (determinístico);
- Cada servidor é uma **máquina de estados**, definida por **variáveis de estado** e **comandos** atómicos;
- Todos os servidores seguem a mesma sequência de estados, se e só se:
    - **Estado inicial:** todos os servidores começam no mesmo estado;
    - **Acordo:** todos os servidores executam os mesmos comandos;
    - **Ordem total:** todos os servidores executam os comandos pela mesma ordem;
    - **Determinismo:** o mesmo comando executado no mesmo estado inicial gera o mesmo estado final.

Algoritmos:

- **Faltas silenciosas / crash:** Viewstamped replication, Paxos, **RAFT**;
- **Faltas bizantinas / arbitrárias:** PBFT, etc...

Algoritmo RAFT:

- **Modelo:** O mesmo dos registos distribuídos (**+ hipótese temporal fraca**):
  Sistema assíncrono: Mais hipótese temporal fraca: algumas coisas acontecem "a tempo";
  Clientes/Servidores: faltas por paragem (crash). No máximo **f** servidores param - grau de replicação **N = 2f+1**
  Comunicação fiável: Mensagens enviadas são recebidas desde que remetente e destinatário não falhem. Não assumimos ordem FIFO.
- **Ideias básicas:**
  Principal objetivo: ser **compreensível**! Importante, pois em informática a complexidade é (muito) má;
  Replicação ativa: Todas as réplicas executam os pedidos (como o registo distribuído). Mas uma das réplicas é **líder** (que pode mudar).
  Coerência forte: linearizabilidade.
- **Arquitetura:**
  Cliente-servidor;
  Algoritmo de consenso gere a atualização do **log replicado**;
  Máquina de estados executa as operações do log.

Linearizabilidade vs Coerência sequencial:

- **Coerência sequencial** (a do registo atómico distribuído):
  As operações **parecem** ter sido feitas por ordem (ou seja, cumprindo a especificação como se o sistema não fosse replicado).
- **Linearizabilidade:**
  As operações **parecem** ter sido feitas por ordem (ou seja, cumprindo a especificação se o sistema não fosse replicado). Essa ordem é coerente com a ordem **real** em que ocorreram.

Líder vs Servidores:

- **Líder tem a responsabilidade de gerir o log distribuído:**
  Recebe pedidos dos clientes;
  Replica-os nos outros servidores;
  Diz aos outros servidores quando podem passar os pedidos no log à máquina de estados (isto é, executá-los ou aplicá-los). Isto garante **safety**: se um servidor aplica a entrada X no indice i do log na sua máquina de estados, é garantido que nenhum servidor vai aplicar uma entrada diferente de X para o índice i.
- **Se o líder falha (crash), o outro tem de ser eleito.**

Ciclo de vida:

- **Cada servidor está num estado de 3: líder, seguidor, candidato:**
  Candidato: estando usado para eleger novo líder, os servidores estão pouco tempo nesse estado;
  Seguidores: passivos, apenas respondem a pedidos de líderes ou candidatos.
- **Tempo é dividido em períodos (terms):**
  Os períodos são numerados sequencialmente;
  RAFT garante que existe **apenas um líder** em cada período.

Comunicação nos servidores:

- **Comunicam usando RPCs:**
  RequestVote: iniciados pelos candidatos durante as **eleições**;
  AppendEntries: iniciados pelos líderes para **replicar** entradas do log e mostrar que estão vivos (**heartbeat**).
- **Quando um servidor chama um RPC:**
  Se ao fim de um tempo não receber resposta, repete o RPC. Os pedidos são todos idempotentes;
  Pode chamar em paralelo outros RPCs, por exemplo, um em cada um dos outros servidores.

Deteção da falha do líder:

- **Líder envia heartbeats periódicos:**
  Ou seja, chama o RPC AppendEntries periodicamente em todos os outros servidores, mesmo que não tenha pedidos de clientes.
- **Seguidor, se ao fim de um election timeout não recebe um heartbeat, então:**
  Assume que o líder falhou e começa uma eleição.

Eleição:

- **Seguidor incrementa o número do período, muda para estado candidato, vota em si próprio.**
- **Chama o RPC RequestVote em todos os servidores pedindo para votarem nele até:**
  1) **Vencer a eleição** (ou seja, obter **f+1 votos**):
     - Decisão de voto é first-come-first-served. Cada servidor vota no 1º que lhe pedir (depois não vota mais para o mesmo período).
     - Cada servidor vota no máximo em 1, logo só 1 pode vencer (no máximo há 2f+1 votos), garantindo safety na eleição: só 1 líder por período.
     - Com uma restrição extra para garantir safety na máquina de estados: o líder tem de estar **atualizado** relativamente ao período anterior (pelo menos f+1 servidores estão atualizados garantidamente).
  2) **Outro servidor vencer a eleição:**
     O que é demonstrado pela receção de um RPC AppendEntry com um número do período **maior ou igual** ao seu.
  3) **Ninguém vence a eleição:**
     Porque vários servidores passaram ao estado candidato ao mesmo tempo e os votos ficaram divididos.
     Todos os candidatos dão timeout e começam uma nova eleição (novo período).
- Os **timeouts** são aleatórios dentro de um intervalo para reduzir a probabilidade de empates, ou seja, de **ninguém vencer a eleição**.
- **Hipótese temporal fraca:** não acontece para sempre os atrasos serem maiores do que os timeouts.

Replicação do log:

- **Quando o líder recebe um pedido de um cliente:**
  Insere uma entrada com o pedido no fim da sua cópia local dos log;

Chama o RPC AppendEntries em paralelo para replicar a entrada nos outros servidores;
Quando a entrada está replicada em segurança (isto é, **em f+1 servidores**):
A entrada diz-se **confirmada** (committed);
O líder aplica a entrada à sua máquina de estados e retorna o resultado ao cliente;
O RAFT garante que as entradas confirmadas são **persistentes** (durable) e são executadas por todos os servidores.

- **Se o cliente envia pedido para seguidor, este reenvia para o líder.**
- **Dadas duas entradas confirmadas (committed) de logs diferentes mas mesmo índice e mesmo período, o RAFT garante que:**
  As duas entradas são iguais;
  Todas as entradas que as **precedem** são iguais.
- **Como?**
  Quando o líder cria uma entrada, num dado período, em dada posição, esta **nunca** muda.
  Seguidores fazer um teste de coerência simples:
  Líder envia em cada RPC AppendEntry o índice e período da entrada anterior do log;
  Seguidor verifica se é igual à entrada anterior do seu log. Se não for, não adiciona.
- **Quando um líder falha, pode deixar servidores incoerentes:**
  Líder chama RPCs com a entrada nos outros 2f servidores mas f RPCs são perdidos e f servidores executam => entrada fica confirmada.
  Líder falha e não chega a repetir os RPCs no f servidores em falta.
  f servidores têm a entrada confirmada no log, outros f não.
- **Resumidamente a solução é:**
  Eleição só elege **servidores atualizados** (que tenham no log todas as entradas já confirmadas);
  Novo líder envia as **entradas em falta** aos seguidores;
  Os seguidores não atualizados podem ter "erros": A verificação feita pelo RPC AppendEntries vai "andando para trás" até ao índice "correto". Depois, seguintes são "**corrigidos**" pelo valor correto do líder.

<u>Em Resumo:</u>

- **Cliente** envia pedidos para **líder**, que o insere numa entrada do log;
- **Líder** replica entrada nos **seguidores** ("consenso"), executa, responde;
- Se **líder** falha, elege-se outro, muda-se de período.

Consenso:

- **Problema do consenso =** Fazer um conjunto de processos chegarem a acordo sobre um dos valores propostos por cada um, **mesmo que alguns processos falhem.**
- **Impossibilidade:**
  Teorema FLP (Fischer-Lynch-Paterson): É impossível resolver consenso distribuído **num sistema assíncrono** se um processo puder falhar.
  Logo na prática, o **modelo** não pode ser puramente assíncrono.

Modelo:

- O mesmo dos registos distribuídos (**+ hipótese temporal fraca**):
  Sistema assícrono: **mais hipótese temporal fraca: algumas coisa acontecem "a tempo".**

Um **protocolo de replicação de máquinas de estados** (RAFT, Paxos, etc…) **resolve uma sequência de** consensos (consensos sobre os pedidos, eleição de líder, etc…), **e executa pedidos,** coisa que os **algoritmos de consenso** não fazem.

Blockchain:

- Termo **blockchain** tem dois sentidos: **estrutura de dados** e **sistema (distribuído)**;
- **Estrutura de dados -** log de blocos append-only;
- **Sistema distribuído -** computadores na Internet:
  Cada um contém uma **réplica** do log/cadeia de blocos;
  Executam um **algoritmo de consenso** para chegarem a acordo sobre o próximo bloco do log.

- **Máquinas de estados replicadas:**
  Originalmente as **transações** eram de dinheiro (criptomoeda);
  Ainda no Bitcoin, mas sobretudo no Ethereum, passaram a existir **transações** que são chamadas a funções de um objeto (smart contract) na gíria do Ethereum;
  Replicação de máquinas de estados!

- **Dois tipos de Blockchains:**
  Permissioned: computadores têm de ter permissão para fazer parte, logo os membros são conhecidos, logo **RAFT** e outros protocolos que referimos podem ser usados (em Hyperledger Fabric, por exemplo).

Permissionless: qualquer computador pode fazer parte, logo RAFT, etc… não funcionam. Bitcoin, Ethereum, usam consenso com **coerência fraca**.

5.2) Replicação passiva:

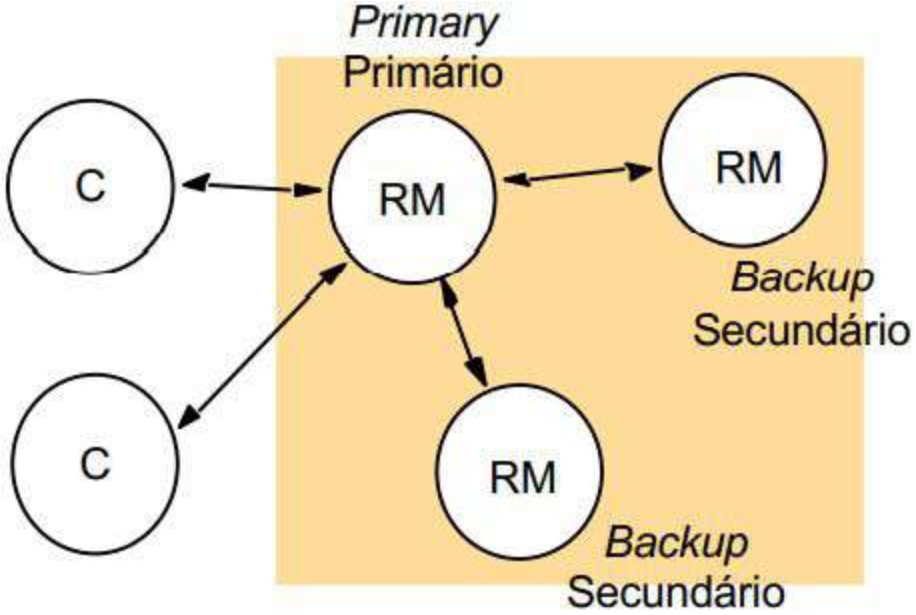
Replicação ativa vs passiva:

- **Replicação ativa** (registos atómicos distribuídos, replicação de máquinas de estados)**:** Caracterizada por todos os servidores executarem os pedidos;
- **Replicação passiva:** Caracterizada por existir um primário e um ou mais secundários. Só o primário executa os pedidos.
- **Diferença crucial:** Enquanto que no **RAFT**, e nos outros, **todos** os servidores executam os pedidos, na **replicação passiva** só o primário executa os pedidos.

Servidores: Um primário e **um ou mais secundários**;
Processos podem falhar silenciosamente, ou seja, não há falhas arbitrárias de processos.
Operações executadas em cada gestor de réplica são atómicas:

- Não deixam resultados incoerentes caso falhem a meio;
- Nos outros protocolos já assumíamos o mesmo.

Replicação total: Cada réplica mantém cópia de todos os objetos lógicos.
Conjunto de réplicas é estático e conhecido a priori.
Arquitetura (Replicação passiva):



Protocolo básico de replicação passiva:

- **Pedido:** Cliente envia pedido ao primário, usando semântica no-máximo-1-vez;

- **Coordenação: Primário trata dos pedidos** por ordem de chegada. No caso de um pedido repetido, devolve resposta já guardada. Ao termos apenas um primário, temos garantia de **linearizabilidade**;
- **Execução:** Primário executa pedido e guarda resposta;
- **Acordo:** Primário envia aos secundários (**novo estado**, resposta, id.pedido), ou seja, envia o **novo estado**, não o pedido. Secundário envia ACK.
- **Resposta:** Primário responde ao cliente.

Pressupostos para o protocolo primary-backup funcionar corretamente:

- **Sistema síncrono,** em particular, são conhecidos os limites máximos para:
  Tempo de transmissão de uma mensagem na rede (t_{max});
  Tempo de processamento de pedido;
  Taxa de desvio dos relógios locais.
- **A comunicação é fiável:** O transporte recupera de faltas temporárias de comunicação e não há faltas permanentes.
- **A rede assegura uma ordem FIFO na comunicação;**
- **Os nós só têm faltas por paragem (silenciosas);**
- **A semântica de invocação dos RPC é no-máximo-1-vez.**

Se houver múltiplos secundários (em simultâneo)?

- **Após detecção da falha do primário, secundários disponíveis elegem o novo primário**, o que é mais complicado, pois é necessário assegurar que todos os secundários elegem **o mesmo** primário.

Como medir os custos da replicação:

- **Grau de replicação:** Número de servidores usados para implementar o serviço;
- **Tempo de resposta (blocking time):** Tempo máximo entre um pedido e a sua resposta, no período sem falhas;
- **Tempo de recuperação (failover time):** Tempo máximo desde a falha do primário e o novo primário a receber pedido do cliente.
- **Objetivo: assumindo que f componentes podem falhar, minimizar as méricas acima.**

Custos da nossa solução:

- **Grau de replicação: f+1** réplicas toleram f faltas (ótimo);
- **Tempo de resposta** (blocking time)**: $4t_{max}$** (ignorando tempo de processamento);
- **Tempo de recuperação** (failover time)**: $P+3t_{max}$** (desde a falha do primário até novo primário receber pedido).

- **Disponibilidade:** MTTR = tempo de recuperação (médio vs máximo); Conhecendo MTBF do servidor primário, basta aplicar a fórmula **Disponibilidade = MTBF / (MTBF+MTTR)**

5.3) Transações em Sistemas Distribuídos:

Transações atômicas locais:

Sequência de leituras e escritas a **objetos partilhados** com outras transações.

Propriedades ACID: (fundamentais para perceber as transações)

- **Atomicidade** (Atomic)**;**
- **Coerência** (Consistent)**;**
- **Isolamento** (Isolated)**;**
- **Durabilidade** (Durable)**.**

Propriedades das transações ACID: Atomicidade:

Transação ou se executa **na totalidade ou não** se executa

Como garantir: O sistema tem de ser capaz de **repor** (rollback) a situação inicial no caso da transação abortar, registando cada **escrita** da transição num **log** (diário).

Diário do tipo undo log = Antes de realizar a escrita, regista (localização, valor original) no diário, e só depois realiza escrita. **Abortar transação:** percorrer o diário e repor os valores originais em cada localização afetada.

Diário do tipo redo log = Para cada escrita, regista (localização, novo valor) no diário, e não escreve na localização real durante a transação. **Confirmar transação:** percorrer o diário e aplicar cada novo valor.

Propriedades das transações ACID: Coerência:

Uma transação é uma **transformação correta** do estado. Assume-se que o conjunto das ações da transação não viola nenhuma das regras de integridade associadas ao estado. Isto requer que a transação seja um programa correto.

Propriedades das transações ACID: Isolamento:

Uma possibilidade seria executar todas as transações em série, uma de cada vez. Seria muito **ineficiente** não permitir concorrência, mas para isso tem de existir **isolamento** entre elas, ou seja, não interferência.

Assim, define-se a condição de **serializabilidade**. Considere a execução **concorrente** de transações.

As transações dizem-se **serializáveis** se existe uma execução sequencial (das mesmas transações) **equivalente** à execução concorrente, ou seja, em ambas

as execuções (concorrente e sequencial), as leituras devolvem o mesmo valor e os objetos escritos ficam com o mesmo valor.

Como gerir execuções concorrentes de transações:

- **Controlo de concorrência pessimista: sincronização em 2 fases estrita:** Cada objeto/grupo de objetos gerido por um **lock** (trinco) de leitura/escrita. Sincronização em duas fases estrita (**strict two phase locking** - 2PL):

  À medida que a transação vai lendo/escrevendo sobre objetos, vai **adquirindo** sucessivamente os respetivos **trincos** (primeira fase, "growing").

  Na terminação da transação (commit ou abort), **liberta os trincos** (segunda fase, "shrinking").

  Problemas:

  Uma solução pessimista envolvendo locks pode representar um **overhead elevado** e **limitar a concorrência**: Não é necessário considerar sempre o pior caso. Na prática, a probabilidade de duas transações acederem ao mesmo objeto simultaneamente é muitas vezes baixa.

  A sincronização com trincos pode ainda conduzir a **deadlock** (interbloqueio).

Soluções para o interbloqueio:

- **Prevenir:** Por exemplo, obrigar transações a adquirir todos os trincos **pela mesma ordem**, se conhecidos de antemão (muitas vezes não é possível).
- **Detectar e cancelar transações:** Há diferentes formas de detectar, seja **periodicamente** correr algoritmo de **procura de ciclos** sobre o grafo de "wait-for", seja por **timeout**, onde se parou muito tempo é porque há deadlock. Quando interbloqueio é detectado entre duas transações, **cancelar uma delas (ou ambas)** para libertar o(s) trinco(s) em causa.

Controlo de concorrência - Solução otimista:

- **Considera que conflitos são raros (optimismo):** Há várias soluções, inclusive optimismo + pessimismo;
- Solução com **backward validation -** 3 fases:
  1) **Trabalho:**
     Primeiro read/write num objeto causa cópia do último valor para um objeto provisório.
     Operações executadas sobre os valores provisórios.
  2) **Validação:**
     Quando recebido closeTransaction, verificar se a transação entra em conflito com outras.

Quando uma transação entra na fase de validação, recebe um **carimbo temporal** Tv.
A transação Tv é válida se é serializável em relação às que a precedem.
É serializável em relação a Ti se:

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v.
read	*write*	2.	T_v must not read objects written by T_i.
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i.

3) **Atualização:**
Se a transação for validada com sucesso, os valores provisórios são escritos nos objetos.

Propriedades das transações ACID: Durabilidade:

Escritas de transação confirmada devem **persistir**, mesmo quando ocorrem faltas expectáveis. Para tal, temos dados e logs mantidos em **armazenamento estável**, e quando há falha/recuperação:

Logs são analisados para determinar transações que estavam a ser confirmadas quando a falha ocorreu;

Essas transações são anuladas (undo log) ou completadas (redo log) antes de o sistema começar a executar novas transações.

Transações atómicas distribuídas:

- Cliente pretende executar transação que envolve operações **em múltiplos servidores.** Os servidores chamam-se **Participantes**, e cada **Participante** é capaz de executar e confirmar transações locais ACID (por exemplo, usa base de dados local).
- Como assegurar **transação distribuída correta** (em particular, atomicidade e isolamento)?

Suporte a transações locais:

- Cada participante usa uma base de dados (ou outra fonte transacional) que suporta:
  Começar transação local (**beginLocal**);
  Executar escritas e leituras no âmbito de uma transação local ativa;

Confirmar a transação (**commitLocal**) ou cancelar (**abortLocal**).

- **Muito importante**:

  Confirmar a transação nem sempre tem sucesso;

  Quando é que um pedido de confirmação pode resultar numa transação cancelada?

Transações distribuídas - Modelo de Faltas:

- **Distribuição implica lidar com** falta dos discos, máquinas e comunicações.
- Apenas se consideram **faltas silenciosas de paragem:** Não se consideram faltas bizantinas. Quando um processo p falha, outro é ativado e obtém o estado anterior de armazenamento persistente, **mascarando** as faltas de paragem.
- **Faltas temporárias de comunicação** toleradas pelos protocolos de transporte**:** Isto é, não se consideram faltas permanentes da rede (partições).
- Sistema **asíncrono.**

Como suportar transações distribuídas:

- **Coordenador** faz a gestão da transação distribuída;
- **Cliente** começa por pedir um identificador da transação distribuída ao coordenador;
- Depois de iniciada a transação distribuída, cliente envia invocações **diretamente aos participantes:** Cada **pedido leva o identificador** da transação distribuída.
- Ao receber pedido do cliente, os participantes pedem ao coordenador para se **juntarem à transação.**
- Depois de correr a transação distribuída, o **cliente pede ao coordenador para a fechar** (isto é, confirmar atomicamente).

Interação Cliente-Coordenador: API do Coordenador:

openTransaction() -> transID;

Inicia uma nova transação e atribui um **TID único**;

Este identificador é usado nas operações seguintes.

closeTransaction(transID) -> (commit, abort);

Termina a transação: Um resultado **commit** indica que a transação foi confirmada. Um resultado **abort** indica que foi cancelada.

abortTransaction(transID);

Cancela a transação.

Papel de cada participante: Invocações a cada **participante** levam o identificador da transação distribuída. Ao receber cada pedido, o **participante:**

- Verifica se já participa na transação distribuída.

- Se não: Ele inicia nova transação local, que fica associada à transação distribuída, e avisa coordenador: **joinTransaction(transID).**
- Executa pedido na transação local associada à transação distribuída.

Quando se chega ao closeTransaction:

- Cliente envia **closeTransaction(transID)** ao coordenador.
- Coordenador envia ordem de **commit** ou **abort** a **todos** os participantes que fizeram join à transação distribuída. Ele conhece **todos** os participantes envolvidos quando o cliente fecha a transação, e **insiste** até todos terem enviado um ACK.
- **Problema:** Não se permite a um dos participantes **abortar unilateralmente** a transação.

Transações distribuídas - Problemas a considerar:

- A **tomada de decisão** de cancelar ou confirmar uma transação é o problema mais complexo a resolver;
- Requer um **consenso** entre os diferentes participantes numa transação distribuída.

Diagrama de Estados do Coordenador:

- **A detecção de faltas de paragem é por timeout.** O coordenador pode logo tomar a decisão de abortar ou tentar contactar novamente os participantes.
- **Estado Inicial** envia **canCommit** a todos e vai para **Esperar.**
- Aí, se receber como input SIM de todos, envia **doCommit** a todos e vai para **Commit.** Caso receba um ou mais NÃO ou o temporizador expire, envia como output **doAbort** a todos e vai para **Abort.**
- Em **Abort** e em **Commit**, sempre que o temporizador expira, reenvia **doAbort** ou **doCommit**, respetivamente.

Diagrama de Estados do Participante:

- **Estado Inicial** se receber **canCommit** e $voto_i$ == SIM, output envia SIM ao coordenador e vai para **Preparado.** Caso **Estado Inicial** receba **canCommit** e $voto_i$ == NÃO ou caso o temporizador expire, output envia NÃO ao coordenador e vai para **Abort**.
- Em **Preparado**, se não recebeu mensagem, continua a esperar ou contacta coordenador. Caso receba **doAbort**, vai para **Abort**. Caso receba **doCommit**, vai para **Commit**.

Transações distribuídas: Tolerância a faltas no 2PC:

- **Não receção de mensagens:** Detectadas com um temporizador no Coordenador ou nos Participantes.
- **Timeout no Coordenador:**

  Estado Esperar: Não pode confirmar unilateralmente a transação, mas pode unilateralmente optar por cancelar a transação, se considerar que o atraso na resposta se deve a uma falta.

  Estados Abort e Commit: O coordenador não pode terminar a transação, ele tem que receber a confirmação de todos os Participantes. Pode repetir a mensagem previamente enviada.
- **Timeout num Participante:**

  Estado Inicial: Pode optar unilateralmente por abortar a transação, ou verifica o estado do Coordenador.

  Estado Preparado: Não pode progredir (depende da decisão do Coordenador que já influenciou, e a transação fica ativa e bloqueada até se saber essa decisão). Se os Participantes interagissem seria possível evoluir, obtendo a decisão do Coordenador que chegou a outros Participantes.

Recuperação depois de falta de paragem:

- **Recuperação do Coordenador:**

  Estados Inicial e Esperar: Repete as mensagens de Preparação para obter novamente a votação dos participantes.

  Estado Abort ou Commit: Se ainda não recebeu todas as confirmações, repete o envio da mensagem global previamente enviada.
- **Recuperação de um Participante:**

  Estado Inicial: Aborta unilateralmente a transação.

  Estado Preparado: Reenvia o seu voto (Sim ou Não) para o Coordenador.

Transações distribuídas - Problemas do 2PC:

- O protocolo é **bloqueante:** Obriga os Participantes a esperar pela recuperação do Coordenador, possivelmente quando estão no estado Inicial e de certeza no estado Preparado.
- **Não é possível fazer uma recuperação totalmente independente.** Depende do Coordenador.
- **Há alternativas não-bloqueantes**, sob modelos de faltas mais restritivos, e normalmente muito mais complexas.

6) Segurança:

(**(In)segurança** - e os **protocolos criptográficos** como solução)

6.1) Segurança e criptografia:

Segurança: o que é?

- **Confidencialidade:** só o emissor e o destinatário é que conseguem "**perceber**" a mensagem (o emissor **cifra** a mensagem, enquanto que o destinatário **decifra** a mensagem);
- **Autenticidade:** o emissor e o destinatário querem **confirmar a identidade** do interlocutor;
- **Integridade** das mensagens: o emissor e o destinatário querem garantir que qualquer **alteração** à mensagem é **detetada**;
- **Disponibilidade:** os serviços devem estar **acessíveis** e **disponíveis** para os utilizadores.

Princípios da criptografia:

- **A linguagem da criptografia:**
  m: texto **em claro** (plaintext)
  $K_A(m)$: mensagem **cifrada** com a chave K_A (ciphertext)
  $m = K_B(K_A(m))$
  Emissor e destinatário usam a **mesma chave** (**simétrica**): **K_S.**

Técnica criptográfica simples de chave simétrica:

- **Algoritmo cifra de César:**
  Cada letra do texto é **substituída** por outra, que se apresenta no alfabeto "à frente" dela um número fixo de posições (a chave **k** é esse número).

Técnica criptográfica ligeiramente mais sofisticada:

- Em vez de apenas 1, usamos **n alfabetos de substituição** (não necessariamente baseados em deslocamentos de k posições): $M_1, M_2, ..., M_n$, usados segundo um **padrão cíclico.** Para cada símbolo em claro, usar o próximo padrão de substituição de forma cíclica, para cifrar.
- **Chave:** os **n alfabetos** de substituição e o **padrão** cíclico.

Como quebrar uma solução criptográfica?

- **Ataque ciphertext-only:**
  Apenas tem acesso a mensagens cifradas.

Duas abordagens: **Ataque de força bruta** (testar todas as chaves possíveis, o que é muito difícil de decifrar mesmo com poucas chaves) e **Análise estatística** (por exemplo, frequência de certas letras e palavras num texto genérico).

- **Ataque known-plaintext:**
  Tem algum texto em claro correspondente a texto cifrado (por exemplo, quando se usa só um alfabeto, pode-se facilmente detectar os pares cifrados para algumas letras).
- **Ataque chosen-plaintext:**
  Consegue obter mensagens cifradas para qualquer mensagem em claro que ela possa gerar.

Standards de criptografia simétrica:

- **DES: Data Encryption Standard:**
  Usa chave simétrica de **56 bits**. Um ataque de força bruta consegue decifrar em menos de um dia. **3DES** é mais seguro, visto que cifra **3 vezes** com 3 chaves diferentes.
- **AES: Advanced Encryption Standard:**
  Veio **substituir** o DES. Usa chaves de **128, 192, ou 256 bits**. Um ataque de força bruta que demore 1 segundo no DES demora 149 triliões de anos no AES de 128 bits.

O principal problema da Criptografia de chave simétrica é conseguir o acordo entre o emissor e o destinatário relativamente à chave a usar.

Criptografia de chave pública:

- **Segue uma abordagem radicalmente diferente:**
  O emissor e o destinatário **não partilham a mesma chave**. Cada interveniente tem **um par de chaves:** Uma **chave pública**, conhecida por **todos**, e uma **chave privada**, conhecida **apenas pelo interveniente.**
- **Requisitos:**
  1) **É necessário um $K^+_B()$ e um $K^-_B()$, tais que $K^-_B(K^+_B(m)) = m$.**
  2) **Dada uma chave pública $K^+_B()$ terá de ser impossível calcular a chave privada $K^-_B()$.**

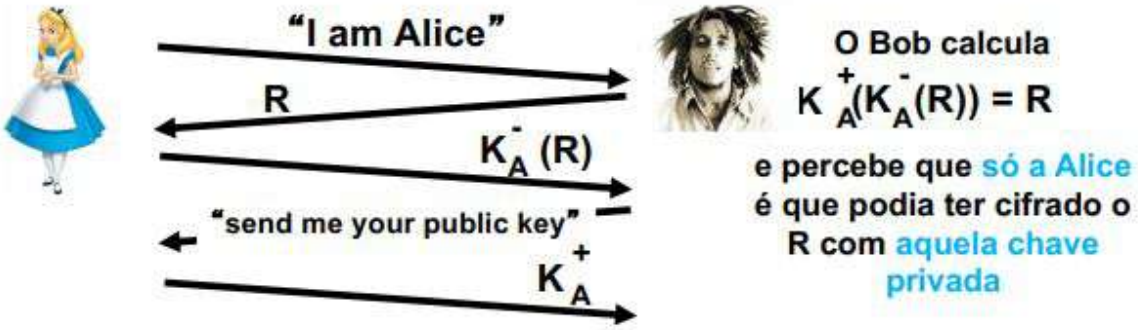
Algoritmos como o RSA, baseado em aritmética modulo-n, preenchem estes requisitos!

- E têm outra propriedade importante: **$K^-_B(K^+_B(m)) = m = K^+_B(K^-_B(m))$,** ou seja, **usar chave pública antes da privada dá o mesmo resultado que usar chave privada antes da pública.**

- Algoritmos de chave pública (**Diffie-Helman, Rivest-Shamir-Adleman ou RSA**) são **computacionalmente muito intensivos e lentos**: O DES é **pelo menos 100 vezes mais rápido** do que o RSA.
- Solução usada normalmente na prática: **chave pública -> chave simétrica**. Usa-se criptografia de chave pública para **estabelecer uma ligação** segura e definir uma segunda chave: a **chave simétrica da sessão**. Depois, essa chave simétrica vai ser usada para cifrar os dados da comunicação.

Autenticação:

- **Objetivo:** O Bob quer que a Alice **prove** a sua **identidade**.
- **Solução:** usar um número R (**nonce**), um número que é usado **apenas uma vez**.
    - **Problema:** é preciso ter chave simétrica;
    - **Solução:** Usa-se criptografia de **chave pública para autenticar** a Alice.



Ataque man in the middle: A Trudy faz-se passar por Alice (ao Bob) e por Bob (à Alice). **Difícil de detectar,** pois o Bob recebe tudo o que a Alice envia, e vice-versa. O **problema** é que a Trudy recebe todas as mensagens também!

Assinaturas digitais:

- **Técnica criptográfica** análoga às assinaturas feitas à mão;
- O emissor assina um documento digitalmente de forma a estabelecer-se como **criador/dono** do documento;
- O emissor **assina** a mensagem. O destinatário **verifica** a assinatura.
- **Propriedades:** ser **verificável** e **não falsificável** (o **destinatário** pode provar a qualquer pessoa que o **emissor**, e **ninguém** além dele (incluindo o destinatário), **assinou** aquele documento.
- O **emissor assina a mensagem m** cifrando-a com **a sua chave privada K_B^-**, criando uma mensagem "assinada", $K_B^-(m)$.

- **Verificação:** Se o emissor enviar **<mensagem m, assinatura $K_B(m)$>**, o destinatário pode usar a chave pública do emissor para verificar que foi ele que enviou a mensagem (só o emissor pode assinar com aquela chave privada).
  O destinatário **verifica** que o emissor assinou a mensagem, que mais ninguém assinou a mensagem, e que o emissor assinou **esta** mensagem, não outra qualquer.
- **Não repúdio:** O destinatário pode pegar na mensagem m e na assinatura $K_B(m)$ e provar em tribunal que foi o emissor a assinar a mensagem.
- **Problema:** É **computacionalmente dispendioso** cifrar com criptografia de chave pública mensagens grandes.
  O ideal era ter uma "impressão digital", pequena, de tamanho fixo, fácil de computar.
  Solução: aplicar uma **função hash H à mensagem m**, obtendo uma mensagem síntese ("digest") de tamanho fixo: H(m);
  Propriedade essencial: para qualquer m, só há um H(m).

Internet checksum: - Tem algumas propriedades interessantes como função hash (por exemplo, produz um digest fixo e curto - soma de 16-bit).

- Infelizmente, dada uma mensagem com um dado valor de hash, **é fácil encontrar outra** mensagem com o mesmo valor de hash.

Algoritmos de hash criptográficos:

- **MD5:** Muito inseguro, fácil obter colisões. RFC 1321, digest de 128 bits.
- **SHA-1:** Inseguro, já foram encontradas colisões. US standard [NIST, FIPS PUB 180-1], digest de 160 bits.
- **SHA-2 e SHA-3:** Seguros. US standards, digests de 224 a 512 bits.

Autoridade de certificação (AC)**:** - Dá a **garantia** de que uma chave pública pertence a uma entidade particular.

- O Bob **regista** a sua chave pública numa AC, mostrando uma **prova** da sua identidade;
- A AC cria um **certificado** que liga o Bob à sua chave pública;
- Esse **certificado é assinado pela AC**, e diz: "Esta é a chave pública do Bob".
- Quando a Alice quer obter a chave pública do Bob, ela obtém o certificado do Bob, verifica a assinatura do certificado usando a **chave pública da AC**, e usa a **chave pública do Bob** que está no certificado.

E-mail seguro:

- A Alice quer garantir **confidencialidade, autenticidade e integridade** das suas mensagens de e-mail;
- Para tal usa **3 chaves:** A sua **chave privada** para autenticar a mensagem, a **chave simétrica** para garantir confidencialidade na comunicação que se vai seguir, e a **chave pública do Bob** para enviar a chave simétrica de forma segura.

6.2) TLS, HTTPS e segurança operacional:

TLS/SSL: Secure Sockets Layer:

- Protocolo de segurança **implantado globalmente**: Suportado por quase todos os browsers e servidores web.
- **Oferece:** Confidencialidade, Integridade, Autenticação.
- Disponível para todas as aplicações TCP, através da interface **secure socket.**

SSL e TCP/IP:

- O SSL providencia uma API para a aplicação: Há bibliotecas/classes C e Java disponíveis.

- Temos alguns requisitos adicionais:
  Enviar stream de bytes e dados interativos;
  Queremos um conjunto de chaves para **toda a ligação** TCP;
  Queremos que a **troca de certificados** faça parte do protocolo: fase de handshake (“aperto de mão”).

Mini-SSL: um canal seguro simples:

- **Handshake:** a Alice e o Bob **trocam os seus certificados** para partilharem um **segredo** e usam as suas **chaves privadas** para se autenticarem.
- **Derivação de chaves:** a Alice e o Bob **criam um conjunto de chaves** a partir do segredo partilhado.

- **Transferência de dados:** Como temos um stream de bytes, os dados a ser transferidos são **divididos** em vários pedaços (os records).
- **Fecho da ligação:** São adicionadas **mensagens especiais** para fechar a ligação de forma segura.

Mini-SSL: Handshake: - Um handshake simples.
- **MS:** master secret**; EMS:** encrypted master secret.

Mini-SSL: Derivação de chaves:
- Considerado **pouco seguro usar a mesma chave** para operações criptográficas diferentes. Assim, usam-se **chaves diferentes** para autenticação de mensagens e para as cifrar.
- **4 chaves:**
  K_C = chave para cifrar dados do cliente para o servidor
  M_C = chave MAC para dados enviados do cliente para o servidor
  K_S = chave para cifrar dados do servidor para o cliente
  M_S = chave MAC para dados enviados do servidor para o cliente
- As chaves são **derivadas** a partir de uma função de derivação de chaves (KDF): Chaves criadas a partir do master secret (com alguma aleatoriedade adicional).

Mini-SSL: records:
- **Porque não cifrar os dados como um stream, à medida que se entregam ao TCP?**
  Onde colocaríamos o MAC?
  Se apenas no fim, só poderíamos verificar a integridade da mensagem quando todos os dados tivessem sido recebidos...
- **Assim, a stream é dividida numa série de records:**
  Cada record inclui um MAC;
  Assim o destinatário pode ir verificando cada record.

Mini-SSL: números de sequência:
- **Problema:** o atacante pode fazer **ataque de replay** com um record, ou pode re-ordenar records.
- **Solução:** colocar **número de sequência** no MAC:
  MAC = MAC(M_x, sequence || data);
  Nota: não se coloca um campo com o número de sequência, apenas se coloca no MAC.

Mini-SSL: nonces:

- **Problema:** o atacante pode fazer ataque de replay a **todos** os records de uma ligação.
- A Trudy pode capturar todas as mensagens entre a Alice e o Bob: No dia seguinte, pode começar uma nova ligação TCP com o Bob, enviando exatamente a **mesma sequência** de records.
- **Solução:** O Bob envia **nonces diferentes para cada ligação.** Assim as chaves passam a ser diferentes: As mensagens da Trudy não vão passar os testes de integridade do Bob.

Mini-SSL: informação de controlo:

- **Problema:** ataque de **truncagem** (o atacante forja um segmento de fecho de ligação TCP).
- **Solução:** adicionar **tipo** de record (**Tipo 0:** dados; **Tipo 1:** fecho de ligação).
- MAC = MAC(M_x, sequence || type || data).

Mini SSL -> SSL:

- Cliente e servidor podem suportar diferentes algoritmos criptográficos;
- Cliente e servidor podem decidir qual o algoritmo específico que querem usar antes da transferência de dados.

Cifras SSL:

- **Múltiplos** algoritmos suportados:
  Algoritmos de criptografia de chave pública (por exemplo, RSA);
  Algoritmos de criptografia simétrica (por exemplo, 3DES, RC4, AES);
  Algoritmo MAC.
- Na negociação, o cliente e o servidor chegam a **acordo** relativamente aos algoritmos a usar: O cliente oferece alternativas e o servidor escolhe uma.

SSL: objetivos do handshake:

1) **Autenticação do servidor;**
2) **Negociação para chegar a acordo relativamente aos algoritmos a usar;**
3) **Estabelecimento das chaves;**
4) **Autenticação do cliente (opcional).**

Handshake SSL:

1) **O cliente envia a lista de algoritmos que suporta, juntamente com o nonce do cliente;**

2) **O servidor escolhe de entre os algoritmos da lista e envia a sua escolha, o seu certificado, e o nonce do servidor;**
3) **O cliente verifica o certificado, extrai a chave pública do servidor, gera o pre_master_secret, cifra-o com a chave pública do servidor, envia;**
4) **O cliente e o servidor computam de forma independente as chaves MAC e de encriptação a partir do pre_master_secret;**
5) **O cliente envia o MAC de todas as mensagens de handshake;**
6) **O servidor envia o MAC de todas as mensagens de handshake.**

- Os 2 últimos passos **protegem o handshake;**
- O cliente normalmente oferece vários algoritmos, uns mais fortes e outros mais fracos (um ataque man-in-the-middle poderia apagar os mais fortes);
- Os 2 últimos passos **previnem** este ataque.

Protocolo SSL:

- **Record header:** tipo de conteúdo; versão do SSL; tamanho;
- **MAC:** inclui número de sequência, tipo e chave MAC M_x.

Formato do record SSL:

- **Content type =** 1 byte;
- **SSL version =** 2 bytes;
- **Length =** 3 bytes;
- **Data e MAC** cifrados (criptografía simétrica).

Firewall:

- Isola a rede de uma organização da Internet, definindo quais os pacotes que podem passar e quais os que devem ser bloqueados: Previne ataques de DoS (por exemplo, SYN flooding), acessos ilegais a dados internos, etc...
- **Filtros de pacotes "stateless":** filtragem feita pacote-a-pacote, com decisão de encaminhar ou deixar cair pacote baseada **apenas** nos endereços IP, portos TCP/UDP, flags, etc…
- **Filtros de pacotes "stateful":** além dos cabeçalhos, mantém também informação sobre o estado de cada ligação TCP, filtrando pacotes que não fazem sentido (por exemplo, pacotes enviados de uma ligação que não foi ainda aberta).
- **Gateways de aplicação:** filtra pacotes baseando-se não só nos campos IP/TCP/UDP mas também em dados da aplicação.

Firewall distribuída:

- Em centros de dados modernos, virtualizados, as políticas de controlo de acesso são **definidas centralmente** por um operador mas são **aplicadas de forma distribuída** ao nível do hypervisor: As políticas são implementadas no momento em que os pacotes entram/saem das interfaces de redes das VMs dos clientes, tipicamente usando o paradigma de **redes SDN** (Software-Defined Network).

Sistemas de deteção de intrusões:

- **Firewalls e filtros de pacotes:** Operam ao nível dos **cabeçalhos** dos pacotes e não fazem qualquer **correlação** entre sessões.
- **SIstema de deteção de intrusões (IDS):** Analisa os **conteúdos** do pacote ("deep packet inspection") e examina **correlações** entre pacotes.

IDS distribuído:

- Os IDS tradicionais recolhem periodicamente informação da rede (dos switches, routers, sensores) e **centralizam** a deteção de intrusões;
- Com os avanços nas redes programáveis os switches são agora ativos e a atividade de deteção passa a ser **distribuída**.

6.3) Autenticação e autorização distribuídas:

Autenticação = estabelecimento de uma **associação** entre uma identidade e um sujeito/entidade (o sujeito pode ser um humano, computador, serviço, etc…);
Autorização = especificação e verificação da **permissão** de um sujeito para aceder a um recurso (em informática a autorização é parte do **controlo de acesso** que inclui também a ação de permitir ou não dado acesso). A **autorização** e o **controlo de acesso** começam com a **autenticação.**

Proceso de Autenticação:

- Autenticação de **pessoas** é feita com base em:
  Algo que ela **sabe** (uma senha ou um pin por exemplo);
  Algo que ela **tem** (o cartão de cidadão por exemplo);
  Algo que ela **é** (impressão digita, retina, por exemplo).
  Estas 3 coisas são Credenciais de autenticação.
- Autenticação de **computadores/servidores** é feita com base em segredos (chaves criptográficas), e muitas vezes o resultado é uma **chave partilhada (secreta)** entre os participantes.

Autenticação:

- **Local:** Senha / palavra-passe, etc…
- **Remota:** Por exemplo iniciar sessão no Google ou no Fenix.

Autenticação remota básica:

- **Versão básica problemática:**
  Utilizador e sistema partilham um segredo (senha, por exemplo).
  O problema é que um atacante que escute a comunicação, fica a saber o segredo e/ou pode fazer **replay** da 3ª mensagem e personificar o utilizador.
- **Protocolo desafio-resposta:**
  O objetivo é evitar os problemas da versão básica;
  Utilizador e sistema partilham uma **função f**.
  Pode ser vulnerável a **replay,** dependendo da função.

Protocolo desafio-resposta: Versão 1:

- **Função f** partilhada = cifra simétrica e **chave secreta K.**
- **Segurança:** Proteção contra replay através do nonce. Autenticação pois só o utilizador e o sistema conhecem K.

Protocolo desafio-resposta: Versão 2:

- **Função f** partilhada = cifra assimétrica e **chave pública K_u;**
- **Segurança:** Proteção contra replay através do nonce. Autenticação pois só o utilizador conhece a **chave privada K_r** que faz par com K_u.

Autenticação na web:

- **Autenticação do servidor:**
  HTTPS (HTTP+TLS) suporta autenticação do servidor: A ideia é essencialmente a da **versão 2** acima, mas trocando **utilizador** por **servidor** e **sistema** por **cliente** (por exemplo browser). O **cliente** tem também de verificar a assinatura do **certificado** que contém a **chave K_u** do **servidor** (com a **chave pública da CA**).

Autenticação do cliente/utilizador:

- **Solução 1:** HTTPS suporta solução do **slide anterior** também para clientes, mas é pouco prático as pessoas terem e enviarem certificados.
- **Solução 2: Senha/password** depois de criada a sessão HTTPS. **É problemática** pois usamos dezenas de aplicações web. Má ideia deixar a gestão ao utilizador, pois usam senhas fracas, tomam nota e deixam à vista e usam as mesmas em várias aplicações.

Single Sign-On (SSO):

- **É a solução para o problema anterior:**
  Utilizador pode autenticar-se em apenas um **serviço de SSO;**

Autenticação é baseada num único conjunto de credenciais;
Várias aplicações partilham esse serviço de autenticação.

- **Resolve dois problemas, relacionados entre si:**
  Autenticação, como vimos acima;
  Gestão de identidades, pois apenas uma identidade por serviço de SSO.

SSO: Soluções anteriores:

- **Kerberos:** Talvez o primeiro SSO, um servidor para autenticação em várias aplicações. **Problema:** limitado apenas a um domínio.
- **Domain Controller:** Muito usado em ambiente empresarial. Autenticação em várias aplicações da empresa. **Problema:** ineficiente, baseado em XML.

OpenID Connect (OIDC):

- **Solução de SSO para aplicações web:**
  Resolve **autenticação** e **gestão de identidades**;
  Permite **desenvolvimento modular**: Programadores **delegam** a solução desses 2 problemas no serviço de SSO;
  Serviço fornecido por Google, Microsoft, Amazon, etc...
- **Funcionamento:** quando um utilizador quer fazer login numa aplicação, é redirecionado para um **fornecedor de identidade** (Authorization Server). Especificação do OIDC não quer saber como é feita a autenticação nesse servidor.

OIDC: funcionamento básico:

1) **Utilizador** pede à aplicação para começar a autenticação;
2) **Aplicação** redireciona utilizador para fornecedor de identidade, e **Fornecedor de identidade** autentica o utilizador e redireciona-o para a aplicação, levando um **ID token**;
3) **Aplicação** usa o **ID token** para pedir ao fornecedor de identidade a conclusão da autenticação. Obtém resposta correspondente.
4) **Aplicação** mostra ao utilizador página com informação de que está logged in.

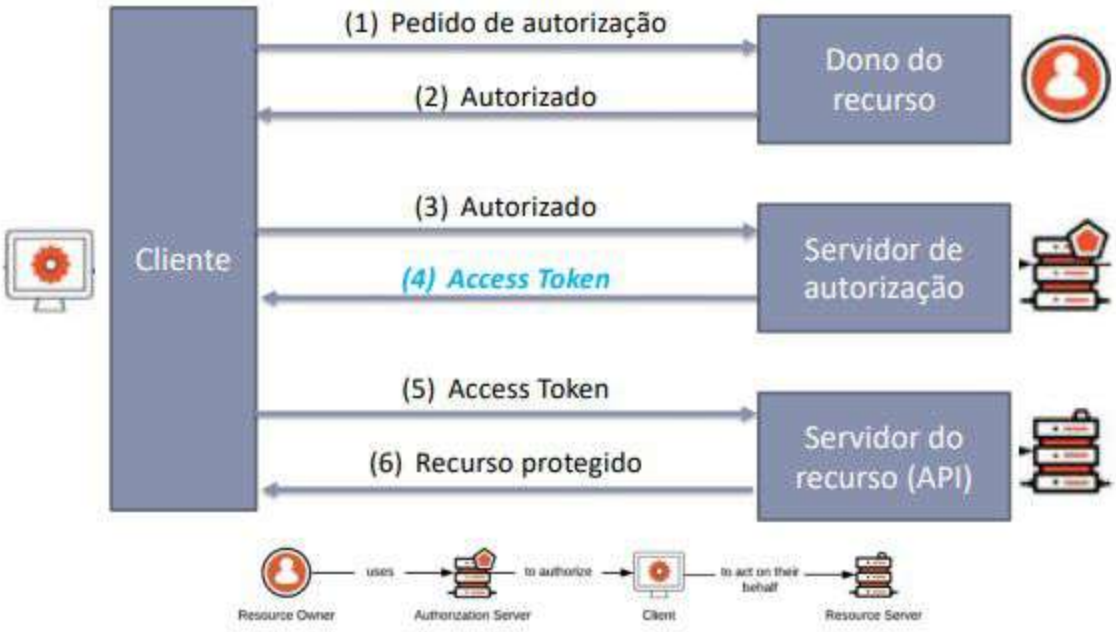
OIDC -> OAuth:

- Porque é que o **fornecedor de identidade** é designado Authorization Server na especificação do OIDC? Porque o OIDC é uma camada fina criada em cima de uma framework de **autorização** (não autenticação): **OAuth 2.0.**

OAuth 2.0:

- **Autorização:** especificação e verificação da permissão de um sujeito para aceder a recursos;
- **OAuth 2.0:** uma framework de autorização que permite a **clientes** usarem **recursos** em nome de **donos de recursos.**

OAuth 2.0: Funcionamento básico:



OAuth 2.0 -> OpenID Connect:

OAuth:

- **Cliente** pede autorização ao **dono do recurso** que o redireciona para o **servidor de autorização;**
- **Cliente** contacta o **servidor de autorização** que verifica a sua autorização -> **token;**
- **Cliente** fornece o **token** à **API;**
- **API** verifica o **token** e permite o acesso.

OIDC:

- **Utilizador** pede acesso à **aplicação** que o redireciona para o **servidor de autorização;**
- **Utilizador** autentica-se perante o **servidor de autorização** -> **token;**
- **Utilizador** fornece **token** à **aplicação;**
- **Aplicação** verifica o **token** e permite o acesso.

Processo igual se **dono de recurso = servidor do recurso/API = aplicação.**

Access Token é mais versátil do que o **ID Token** que serve apenas para autenticação.

OIDC: Comunicação:

- Baseada em **REST;**
- Endpoints do servidor de autorização (fornecedor de identidade):
  authorization_endpoint = URL para os utilizadores se autenticarem
  token_endpoint = URL onde as aplicações podem obter tokens
  userinfo_endpoint = URL onde as aplicações podem obter informação adicional sobre um utilizador

ID Token = Documento digital (JSON Web Token - JWT) **que contém:**

- **Claims:** informação sobre o utilizador que se está a autenticar;
- **Assinatura digital:** para verificar autenticidade e integridade do token. Gerada pelo servidor de autenticação com a sua chave privada e verificada usando a chave pública, fornecida pelo mesmo servidor;
- **Nonce:** para evitar ataques de replay;
- **Audiência:** CLIENT_ID definido pelo fornecedor de identidade quando a aplicação é registada. Objetivo é provar que o token foi de facto gerado para essa aplicação e não para outra.
- **Issuer:** identidade do fornecedor que gerou o token.

<u>Sumário:</u>

- **OpenID Connect** é uma solução de Single-Sign On moderna, para **aplicações web** e **ambientes de cloud:** Resolve autenticação e gestão de identidades e permite desenvolvimento modular (programadores delegam a solução desses 2 problemas no serviço de SSO).
- Implementada como uma camada fina criada em cima de uma framework de autorização: **OAuth 2.0**.