Escola Técnica Estadual de São Paulo

AUGUSTO FÉLIX TOMAZ DA COSTA

ELLEN VIDAL MARTINS

ITALO PEDRO LOPES SILVA

JÚLIA FERREIRA DA COSTA

LILLIAN NUNES DE OLIVEIRA

TRABALHO DE APS

Especificação Formal (Especificação de Interface e Comportamento) e Arquitetura de Sistemas Distribuídos

São Paulo

2021

AUGUSTO FÉLIX TOMAZ DA COSTA

ELLEN VIDAL MARTINS

ITALO PEDRO LOPES SILVA

JÚLIA FERREIRA DA COSTA

LILLIAN NUNES DE OLIVEIRA

TRABALHO DE APS

Especificação Formal (Especificação de Interface e Comportamento) e Arquitetura de Sistemas Distribuídos

Trabalho apresentado ao Curso de Desenvolvimento de Sistemas como requisito para obtenção de menção.

Orientador: Professor Luiz Ricardo

São Paulo

2021

Introdução

O uso dos *softwares* está cada vez mais presente no nosso cotidiano, e cada vez mais se mostra necessário o surgimento de meios os quais sejam capazes de tornar o uso destas entidades abstratas cada vez mais eficientes no mundo contemporâneo. Um desses meios é a computação paralela, que é capaz de tornar processamentos de dados mais rápidos, sendo distribuídos em várias máquinas ao redor de todo o mundo, o que por sua vez também é responsável por conectar o mundo todo a uma rede de informação. Formando assim uma computação de alto desempenho e mais eficiente.

Resumo

Apresentação do processo de estruturação e especificação da construção, e normatização arquitetônica dos sistemas distribuídos os quais fazem parte do nosso cotidiano em quase todos os hábitos e necessidades mundanas há quase 40 anos. Análise da estruturação da arquitetura e formação de sistemas que podem causar um grande prejuízo ao usuário ou proprietário se obtiverem alguma falha. Estudo de métodos para minimização de erros dos mesmos, como a especificação formal, que se divide em diversos tipos, tal como diversas linguagens de programação para modelar de forma correta e precisa o *software*, onde as especificações formais são nada mais do que métodos matemáticos utilizados para encontrar, entender e representar os requisitos e funcionalidades de um sistema distribuído, especificando sua interface ou até mesmo os diferentes comportamentos/estados que ele pode tomar ao longo do tempo.

Palavras chave: estruturação, especificação, e sistema.

,

Abstract

Presentation of the process of structuring and specifying the construction, and architectural standardization of distributed systems which have been part of our daily lives in almost all mundane habits and needs for almost 40 years. Analysis of the structure of the architecture and formation of systems that can cause great harm to the user or owner if they get any failure. Study of methods for minimizing their errors, such as the formal specification, which is divided into several types, such as several programming languages to model the software correctly and accurately, where the formal specifications are nothing more than what mathematical methods used to find, understand and represent the requirements and functionalities of a distributed system, specifying its interface or even the different behaviors/states it can take over time.

Keywords: structuring, specification, and system.

Sumário

1. ESPECIFICAÇÕES FORMAIS

Utilizando de métodos matemáticos, a especificação formal irá auxiliar o desenvolvedor de modo que ele entenda corretamente o que se é pedido, e de modo que ele possa desenvolver e estruturar os requisitos de forma correta, a fim de cometer o mínimo de erros e que consiga atingir as expectativas do usuário.

Existem diversos tipos de especificação formal para cada tipo de projeto necessário, como a especificação algébrica e a especificação baseada em modelos que serão retratadas posteriormente.

As especificações formais são métodos contidos em um conceito geral de arquitetura de *softwares*, os métodos formais. São meios criados a fim de construir um *software* com o mínimo de erros possível, tendo como base princípios matemáticos.

Entretanto, ela não é utilizada em todo e qualquer caso, pois apesar da precisão, ela toma um tempo bem maior do que muitos outros métodos, afinal, mudanças no mercado fizeram do prazo de entrega uma peça chave. Portanto, ela é comumente muito mais utilizada nos ditos anteriormente sistemas críticos, pois uma vez que eles falhem no seu propósito, podem custar muito dinheiro ao dono do *software*, ou até mesmo custar vidas. Um exemplo disso foi o projeto Ariane 5, um projeto espacial europeu, que por um erro no levantamento de requisitos e arquitetura do projeto de *software*, fez com que o mesmo explodisse após 40s de decolagem, causando um prejuízo enorme aos envolvidos.

Todavia, uma de suas principais qualidades (além da menor taxa de erros) é que o custo sai muito menor, afinal, não precisam bancar o custo de reparar erros que poderiam ser evitados. O que também pode pender muito ao seu lado, afinal, uma vez que os custos sejam menores, os ganhos serão maiores.

Na construção de um *software*, temos alguns passos que são seguidos em ordem decrescente de envolvimento do usuário, mas em ordem crescente de envolvimento do desenvolvedor, passos estes que são capazes de garantir um

software bem estruturado e que cumpra as necessidades do cliente de forma eficaz e com poucos custos. São eles:

DEFINIÇÃO DE REQUISITOS DO USUÁRIO

O levantamento de requisitos em si aqui faz a consulta ao usuário para entender o que se é pedido. Uma fase extremamente importante, afinal, ela é a base para se entender todos os pedidos necessários, além de que é o ponto que mais costuma causar erros no projeto finalizado (se não compreendido corretamente).

ESPECIFICAÇÃO DE REQUISITOS DO SISTEMA

Nesta etapa, deve se obter o entendimento dos requisitos, o que cada etapa deve fazer como chegar até ela, o que é necessário para se obtê-la, o quanto custará, etc. Provavelmente, neste ponto, o desenvolvedor terá (e se precisar, deve) retornar ao cliente, para que ele possa explicar novamente. Ainda nesta fase do projeto, a equipe do Cliente define os requisitos do sistema com o Analista de Sistemas, onde os Requisitos são divididos em dois grupos, Requisitos Funcionais (Regras de Negócio) e Requisitos Técnicos (Não Funcionais):

- **Requisitos Funcionais** são requisitos de negócios, de acordo com a necessidade da área que utilizará o sistema Cliente, tais como: definir as funcionalidades (cadastros, relatórios etc) *desig*n, as divisões e permissões entre diferentes usuários do Sistema, acesso de históricos e dados pelos usuários, inclusão e exclusão de certas informações no Sistema, opções de compartilhamento e troca de informações entre usuários, restrições do sistema além de outros possíveis itens dentro do desígnio comercial proposto pelo cliente.

- **Requisitos Não Funcionais** são desenhados por um analista de sistemas e avaliados pela área de tecnologia do cliente, pois define os valores técnicos do

software, tais como: prevenções de erros e falhas no sistema, aspectos antivírus, acessibilidade do Cliente ao sistema, armazenamento de dados, considerações de leis etc. Requisitos não funcionais podem ser divididos em 3 sub-requisitos:

1. **Requisitos de Produto:** confiabilidade, usabilidade e velocidade do *software*.

2. **Requisitos Organizacionais:** data de entrega, padrão de qualidade e à implementação do *software*.

3. **Requisitos Externos:** segurança, ética, privacidade e requisitos legais para o *software*.

Um alinhamento entre Cliente e Analista de Negócios é necessário para poder criar os Requisitos com sucesso, mesmo assim é imprescindível a aplicação de testes de aceite da área usuária, sendo necessárias eventualmente correções de erros, inclusões de novas funcionalidades e quaisquer outras situações que visem atender a necessidade do negócio do Cliente.

Para traduzir os pensamentos abstratos do cliente de maneira que todos do time de desenvolvedores possam entender pode ser utilizado o sistema *VORD* sigla para Definição de Requisitos Orientada a Ponto de Vista (*Viewpoint-Oriented Requirements Definition*).

Com os Requisitos determinados corretamente e aprovados pelo Cliente já é possível criar cronograma, definindo prazos, etapas de entregas, testes e aceite final do cliente.

PROJETO DA ARQUITETURA

O começo da modelagem do *software*, organizando os requisitos à necessidade e vontade do usuário, utilizando de diversos meios de representação clara para bom entendimento, tanto do desenvolvedor quanto de todos os

envolvidos. Possivelmente, o desenvolvedor terá de retornar à etapa 2, na especificação de requisitos do sistema.

PROJETO DE ALTO NÍVEL

O modelo do projeto final construído em linguagem de alto nível (a linguagem comum ao ser humano). Etapa esta construída e verificada pela especificação formal, como um detalhamento do que se é pedido e como deve ser criado.

NOÇÃO DE REQUISITOS DE SEGURANÇA

Outro tópico extremamente importante na criação de um sistema com o mínimo de falhas possível. São caracterizadas por requisitos básicos que mantêm o nível de segurança do *software*. A *ASVS* (*Application Security Verification Standard*) define os requisitos básicos de segurança de *software*, dentre eles estão:

• **Confidencialidade:** Garantir que somente pessoas autorizadas tenham acesso aos dados

• **Integridade:** Garantir que apenas pessoas autorizadas sejam capazes de alterar ou excluir dados, a fim de obter a máxima exatidão de dados possível.

• **Disponibilidade:** Os dados devem estar disponíveis ao usuário sempre que possível, sua avaliação deve ser tanto quantitativa (quantidade de dados) quanto qualitativa (qualidade do processo de verificação).

• **Autenticação:** Garantir que o usuário que está acessando o sistema seja legitimado.

• **Autorização:** Uma vez que o usuário seja legitimado e acesse o sistema, devemos garantir que ele possa acessar apenas os dados e realizar apenas as operações a qual ele tem permissão para fazê-lo.

• **Responsabilização:** Uma vez que todos os requisitos sejam cumprindo, temos que ter certeza que as ações daquele usuário são de unicamente sua responsabilidade.

Uma forma de garantir que todos esses requisitos sejam corroborados é utilizando do método de especificação *OWASP*.

SIL (ANÁLISE DO NÍVEL DE INTEGRIDADE)

Talvez um dos últimos tópicos a serem tratados neste quesito. Basicamente, tomar como prova se todos os feitios de segurança foram feitos corretamente, atendendo todos os requisitos. Para isso, existem várias formas utilizadas para identificar os erros e corrigi-los, inclusive, a especificação formal é uma forma intrinsecamente importante para se formar um sistema com o mínimo de erros possíveis em relação à segurança. Desta forma, principalmente a especificação algébrica deve ser um método muito confiável a fim de obter tal sucesso. Afinal, um sistema de segurança de grande porte será dividido em vários subsistemas a fim de minimizar os erros, a exemplo temos bancos de dados que guardam informações sobre vários usuários, estas informações devem poder ser acessadas em inúmeros locais, através dos famosos sistemas distribuídos, e para que os mesmos funcionem corretamente, deve-se ser utilizado da especificação algébrica (especificação de interfaces) para construir a relação entre o usuário e a máquina de forma efetiva, a fim de obter a informação de modo seguro e correto.

.

1. 1 ESPECIFICAÇÃO DE INTERFACE

Como dito anteriormente, existem diversas formas de especificação formal, cada uma para um tipo de projeto, e não é diferente com a especificação algébrica (especificação de interface). Interface é uma forma de comunicação entre as pessoas e o computador, ela pode ser entendida como uma abstração que estabelece a forma de interação da entidade com o mundo exterior, através da separação dos métodos de comunicação externa dos detalhes internos da operação. Uma interface também pode promover um serviço de tradução para entidades que não falam a mesma linguagem, como no caso de humanos e computadores. A interface é utilizada em diferentes áreas da ciência da computação e é importante no estudo da interação homem-máquina, no projeto de dispositivos de hardware, na especificação de linguagens de programação e também em projetos de desenvolvimento de *software*.

A especificação de interface é quando se define como o usuário pode interagir com o *software* por meio dos requisitos. Exemplo: se for um *software* para o lançamento de notas de uma escola, na especificação de interface será definido quem pode adicionar as notas no programa, quem pode velas, e etc.

Existem diferentes métodos de especificações, tal como diferentes linguagens, uma dessas é a especificação algébrica, comumente utilizada em sistemas de grande porte, que podem ser divididos em subsistemas, a fim de construí-los de forma independente e minimizar a taxa de erros. A especificação algébrica, também chamada de especificação de interface, é utilizada para a arquitetura correta da interface entre dois subsistemas deste sistema de grande porte.

Existem dois tipos de interface:

A **interface de usuário** que possibilita o usuário interagir com o computador ou algum outro dispositivo eletrônico.

Nesse sentido, refere-se também aos controles usados em um *software*, permitindo ao usuário interagir com o programa. Por exemplo, para digitarmos um texto no *Microsoft Word*, é preciso que o usuário interaja com a tela disponibilizada pela aplicação.

Basicamente, funciona da seguinte forma: O usuário interage com a interface, inserindo informações por meio de teclado e mouse. A partir disso, a interface irá interpretar os comandos do usuário, retornando alguma coisa na tela do computador.

E a **interface física** que se refere a um dispositivo físico, uma porta, ou conexão que interage com o computador ou qualquer outro dispositivo físico.

Um exemplo de interface física é o *Joystick* (manete de jogos). Através dele, é possível os jogadores interagirem com os games. O mouse também é uma interface física e outros periféricos.

Existem várias linguagens de programação a fim de simplificar este problema, como a *Larch, OBJ, LOTOS*, ou a mais famosa, *UML* (Linguagem de Modelagem Unificada).

Uma das linguagens mais usadas é a *Larch*, a qual tem como estrutura:

1. Introdução: Dada por *sort* que é o tipo de dado abstrato.

2. Imports: Abaixo do *sort*, temos as variáveis colocadas.

3. Descrição informal: Um texto curto escrito em linguagem de alto nível, explicando o objetivo do *software* e especificando as variáveis.

4. Declarações: Posteriormente, temos as declarações, onde são designados o domínio e contradomínio (interface dos subsistemas), isto é, a relação dos dois

conjuntos (subsistemas) em função de x. A mesma é dada pela sintaxe (a linguagem de programação escolhida).

5. Axiomas: Verdades absolutas necessárias para construir um teorema, esta que definem a semântica (definição, significado, simplificação) das operações necessárias para que a interface funcione corretamente através dos axiomas que caracterizam o comportamento da interface. Definem a semântica das operações.

Alguns componentes da sintaxe desta linguagem são:

- **Construtores:** Cria uma instancia do tipo abstrato de dados que estão sendo definidos. Isto é, declara as variáveis. EXEMPLO: Criar (*create*).

- **Modificadores:** Retornam estas variáveis já declaradas, mas com valores modificados. EXEMPLO: *Atrib*.

- **Consulta:** Desta vez, retornam o valor da instancia, ou seja, a instância já definida e modificada. Retorna o valor do dado que está sendo definido. EXEMPLOS: *LimU, LimS, Val*.

Um exemplo do uso da especificação algébrica seria num sistema de controle de tráfego aéreo, que não depende da alteração do estado do objeto.

1.2 ESPECIFICAÇÃO DE COMPORTAMENTO

Um dos problemas da especificação algébrica é que ela depende do estado do objeto, do seu comportamento, por isso ela pode não ser muito eficiente para casos específicos de *softwares*, e para isso, existe a especificação baseada em modelos, ou especificação de comportamento, que por sua vez expõe o estado do sistema, e é capaz de construir operação para caso de mudança de comportamento do objeto. Para esta, existem várias linguagens, como a *VDM* ou notação em B ou Z, ou novamente, o uso da linguagem *UML*.

A especificação de comportamento é como uma construção matemática que procura descrever o comportamento dos sistemas de forma rigorosa e abstrata, liberando assim a especificação de detalhes de implementação concreta. Com isso pode evitar problemas futuros como a não compreensão dos requisitos.

NOTAÇÃO EM Z

A notação em Z é uma notação madura, pois ela combina tanto a descrição formal e informal, quanto a destaques gráficos para representação. Assim como a especificação algébrica, ela detém de um nome e uma assinatura do esquema (declarações), tal como de um predicado, que é a construção do sistema em sintaxe de conjuntos (pertence a, está contido em, etc). Foi criada na década de 70, tendo como base a Teoria dos conjuntos.

Como exemplo a especificação de comportamento usando da notação em Z, temos, por exemplo, um *software* de uma bomba de insulina, a qual teriam variáveis de entrada (para ligar a bomba, fornecimento dos dados necessários para a dosagem, isto é, a glicose, etc) e variáveis de saída (como alarmes para o estado do usuário ou a dosagem necessária a ser fornecida), tal como os axiomas (verdades absolutas que não podem ser alteradas, como que a dosagem sempre deve ser menor ou igual à capacidade do reservatório, ou que nenhuma dose pode

exceder 4 unidades de insulina de uma vez, entre outras. Neste caso, o *software* iria obter uma primeira informação da glicose no sangue do usuário e, em seguida, uma segunda informação, se percebesse que a glicose aumentou dentre este intervalo de tempo (se o comportamento do objeto se alterou), o sistema iria enviar uma certa quantidade de insulina ao sangue do usuário.

NOTAÇÃO EM B

Criada também pela Universidade de Oxford, na década de 80, ela faz uso da Teoria de Conjuntos e predicados, onde divide a representação do sistema em:

- **Estado:** Instância das variáveis e o comportamento do objeto.
- **Invariantes:** Predicados lógicos aplicados nas variáveis, e determinação do estado verdadeiro.
- **Operações:** Consultam o estado e, se necessário, modificam o mesmo.

DIAGRAMA DE CASO DE USO (USE CASE)

Entretanto, outra forma de se modelar este sistema seria através do diagrama de estado (linguagem *UML*), que é construída na forma de um fluxograma, que definem e representam o objeto durante seu ciclo, tal como os estímulos que são capazes de mudar seu comportamento durante a trajetória do programa, através do famoso diagrama de *Use Case*, ou caso de uso.

Esta linguagem é construída a partir das seguintes estruturas:

- **Nome:** O nome de cada *Use Case*.
- **Descrições resumidas:** Uma curta descrição feita em linguagem de alto nível, descrevendo o objetivo do diagrama.

• **Fluxo de eventos:** Uma descrição do que cada *Use Case* deve fazer, de modo a qual o cliente deve ser capaz de entender.

• **Requisitos especiais**: Um texto que descreve cada requisito funcional e não funcional do *Use Case*.

• **Pré-condições:** Uma descrição inicial que define quando o *Use Case* deve começar a ser usado e quando não deve ser utilizado.

• **Pós-condições:** Uma descrição textual que descreve quando cada *Use Case* deve finalizar seu trabalho.

• **Pontos de extensão:** Pontos no fluxo do diagrama a quais podem ser inseridos novos comportamentos usando do relacionamento de extensão.

• **Relacionamentos:** Associações de comunicação, inclusão, etc. Relacionam os casos de uso.

• **Diagramas de atividades:** Uma representação através de um desenho de todo o fluxo de eventos.

• **Diagramas de caso de uso:** Diagramas que representam os relacionamentos de cada *Use Case*.

• **Outros diagramas**: Outras representações do *Use Cases*.

OUTRAS REPRESENTAÇÕES

De outra forma, podemos dividir em refinamento (estágio de obtenção dos requisitos e modelagem do projeto), síntese (o esqueleto do código começa a ser

montado) e prototipação e prova (o estágio necessário, a fim de, perceber se os requisitos foram atendidos, pode-se usar da prova de teoremas automatizados, que usa de uma descrição do sistema e de inferências lógicas para achar possíveis erros, ou até mesmo da prova de verificação de modelos, ainda na fase do refinamento, fazendo testes exaustivos em todos os estados do sistema a fim de obter o melhor resultado.

Para que ela seja construída de forma correta, deve-se levar em consideração também a relação entre os requisitos do sistema, como interfaces com o ambiente, noção de estados do sistema, noção de comportamentos, noção de requisitos de segurança, nível de integridade do *software*.

2. ARQUITETURA DOS SISTEMAS DISTRIBUÍDOS

Os sistemas distribuídos nasceram na quarta era da computação, com objetivo de encontrar um meio a qual fosse possível processar vários dados ao mesmo tempo de forma independente em qualquer lugar do mundo, onde temos vários componentes de *hardware* e *software* que operam, processando dados e transmitindo informações entre si de forma paralela, designando funções apenas através de mensagens enviadas entre si pela rede (a ligação entre dois ou mais sistemas, regida por uma interface, que é um meio físico que proporciona a conexão entre dois sistemas), conectando estas máquinas a uma rede de informação, de forma que fosse mais fácil trabalhar em sua modelagem (já que seria feito de forma independente). Além de que se uma falha ocorresse em uma dessas máquinas, não necessariamente todo o sistema estaria comprometido. Os componentes destes sistemas podem estar na mesma sala, no mesmo prédio, em algum lugar do mesmo país ou até mesmo em outro continente.

A exemplo, temos o sistema de *GPS*, internet, banco de dados de uma empresa, servidores de um jogo, etc. São usados para atendimento médico, uso comercial como entregas, gerenciamento ambiental ou até mesmo estudos científicos. Entretanto, para que funcione de forma eficiente, pode (e devem-se, afinal, estes sistemas costumam reger uma grande parcela de pessoas, e um erro pode acabar causando grandes prejuízos) utilizar da especificação algébrica para monitorar as interfaces dos subsistemas presentes neste sistema de grande porte. A exemplo de sistemas distribuídos temos a *internet, world wide web*, correios eletrônicos, sistemas de ficheiros distribuídos, máquinas multibancos, etc.

Agora que já introduzimos os conceitos básicos e necessários para um bom sistema distribuído, trazemos a arquitetura dos mesmos, isto é, a forma a qual iremos construí-los. A arquitetura de sistemas caminha de mãos dadas com a arquitetura de *software*, entretanto, enquanto a arquitetura de *software* trabalha com a organização dos modelos lógicos, a arquitetura de sistemas trabalha com a organização dos sistemas físicos (apesar de que a mesma não funciona sem os modelos lógicos).

Cada componente do sistema (seja este lógico ou físico, isto é, *software* ou *hardware*) será essencial para designar uma função do sistema, funções estas que são divididas em três:

- **Processamento:** Designa o comportamento do sistema distribuído, seus serviços ou funções.
- **Estado:** Maneja os dados do sistema
- **Interação:** Designa o relacionamento entre os componentes do sistema (interface), os famosos conectores.

Como dito anteriormente, podemos identificar nesses componentes que podem ser modelados a partir das especificações formais. Onde podemos utilizar a especificação de comportamento (baseada em modelos) para modelar os componentes que gerem o processamento, e utilizar da especificação de interface (algébrica) para gerir os componentes responsáveis pela interação.

Entretanto, há muitas formas de arquitetura para estes sistemas, cada uma para um objetivo específico.

2.1 CARACTERÍSTICAS DOS SISTEMAS DISTRIBUÍDOS

Sistemas distribuídos eficientes necessitam de uma bagatela de características importantíssimas, características estas como transparência, partilham de recursos, sistema assíncrono, concorrência e heterogeneidade.

TRANSPARÊNCIA

Uma das características mais importantes dele é a transparência, afinal, para o usuário e para o programador, ele deve ser visto como um sistema único. Esta característica se divide em vários tipos de transparências, são elas:

• **Transparência de acesso:** Torna possível com que o acesso aos recursos locais seja feitas a partir da mesma interface, isto é, da mesma operação.

• **Transparência de localização:** Isto é, torna possível o acesso aos recursos, sem que seja necessário o conhecimento da sua localização.

• **Transparência de concorrência:** Ou seja, o usuário não precisa ter acesso à concorrência do sistema em questão.

• **Transparência de replicação:** Quando um dado pode acabar apresentando falhas, este dado é duplicado para que se este caso venha a ocorrer haja um segundo ambiente a qual este componente possa ser utilizado, e a transparência de replicação se baseia no fato de que o usuário não deve ser capaz de saber que este dado se trata de uma cópia de um dado original.

• **Transparência de Falhas:** Torna possível que uma falha seja corrigida sem que o usuário perceba como isto aconteceu, como quando um *e-mail* é reenviado para o usuário devido a uma falha.

• **Transparência de Migração:** Isto é, permitir que o usuário possa mudar de localização sem que isto altere seu desempenho (a exemplo temos os telefones móveis).

• **Transparência de Desempenho:** Torna capaz a reconfiguração, ou atualização do sistema, sem que o usuário o perceba.

• **Transparência de Escalabilidade:** Permite a expansão do sistema, de que forma que a mesma passe despercebida do usuário, onde ele não consiga perceber como o mesmo foi feito.

PARTILHA DE RECURSOS

Uma das principais características de um sistema distribuído é a partilha de recursos, como por exemplo, vários computadores usarem uma mesma impressora ou base de dados, ou até mesmo um mesmo sinal de *Wi-Fi*. Esta característica pode ser prejudicada pelo tráfego excessivo de usuários, que podem causar a queda de latência ou deixar a aplicação inacessível por tempo limitado.

SISTEMA ASSÍNCRONO

Não existe um relógio global, o que significa que todas as máquinas que compõe o sistema distribuído podem trabalhar e funcionar em diferentes tempos de processamento, desta forma, os dados podem ser transmitidos intermutavelmente em um fluxo estável de dados. Por conta disso, temos a famosa queda de latência, que gera o irritante *lagg* nos jogos digitais.

CONCORRÊNCIA

Isto é, quando dois ou mais usuários executam processos, ou seja, necessitam da partilha de recursos, concorrendo ao acesso a este mesmo recurso. Como exemplo, temos várias pessoas tentando acessar um conteúdo de entretenimento na internet, fazendo com que a aplicação fique lenta, demore à carregar, ou muitas vezes até mesmo fazê-la “cair” (devido ao tráfego excessivo de usuários no sistema).

HETEROGENEIDADE

Pode funcionar independentemente do tipo de máquina, de *software*, de rede ou de linguagem de programação. Para resolver o problema da heterogeneidade, utiliza-se uma camada intermediária de *software*: *Middlewar*e.

2.2 TIPOS DE SISTEMAS DISTRIBUÍDOS

Anteriormente, citamos as formas as quais os componentes podem ser distribuídos, tais como a interação entre si. Entretanto, agora iremos falar sobre os tipos de sistemas as quais estas arquiteturas podem estar presentes, onde da mesma forma servem e fazem parte de tipos específicos de sistemas.

Existem vários tipos de sistemas distribuídos, entre eles:

- **Computação em cluster:** Uma computação homogênea (as máquinas devem ser parecidas), onde geralmente um único programa é executado em paralelo, e utilizam da mesma rede local (*LAN*).

- **Computação em grade:** Estas já são heterogêneas (as máquinas podem ter características diferentes e capacidades diferentes), podem ter conjuntos de empresas trabalhando nelas (compartilhando o processamento), todas conectadas em redes, independente do sistema operacional.

- **Computação em nuvem:** Da mesma forma que a computação em grade, esta se apresenta como heterogênea, e tem características bem parecidas com a mesma. Entretanto, diferentemente desta, a computação em nuvem é aberta. Isto é, qualquer uma pode fazer parte. Enquanto a computação em grade se apresenta essencialmente para empresas de modo fechado a elas.

- **Sistema de informação distribuída:** Disponibiliza que os dados sejam acessados em qualquer máquina, onde sua principal dificuldade se apresenta em construir a interface entre diferentes tipos de máquinas.

- **Sistemas pervasivos:** Um sistema muito instável, voláteis, pois dependem de uma bateria ou fontes de energia do tipo (como nos celulares).

2.3 ESTILOS ARQUITETÔNICOS DOS SISTEMAS DISTRIBUÍDOS

Existem várias formas as quais os componentes (módulo) de um sistema podem ser configurados. Ela é determinada pelos seus componentes, a conexão entre eles, como se deve dar estas interações e que tipo de dados são transmitidos. Onde a interface entre os componentes (comunicação que define uma aplicação) pode ser requerida (um componente pede a outro) ou fornecida (um componente fornece a informação à hoje). São elas:

- **Arquitetura em camadas:** Os componentes são divididos de forma hierárquica, isto é, em camadas, onde um componente da camada N, só poderá entrar em contato e requerer informação de um componente da camada N-1 (interface requerida), isto é, ele não poderá pedir informação à camada acima dele, para ela, tudo que poderá fazer será enviar informações (interface fornecida). São extremamente presentes nas famosas redes.

- **Arquiteturas baseadas em objeto:** Os componentes são datados como objetos, e conectados entre si por chamada remota. São principalmente muito utilizadas nas famosas arquiteturas cliente-servidor (arquitetura centralizada).

- **Arquiteturas centradas em dados:** Funciona como uma caixa de correios, onde componentes armazenam dados numa espécie de repositório de informações, enquanto outro componente recolhe estes dados. Portanto, são fracamente acopladas, afinal, estão fracamente conectadas entre si, a fim de que as informações, uma vez que um dos componentes se perca ou seja desligado, não sejam perdidas.

- **Arquiteturas baseadas em eventos:** O famoso sistema *publish-subscribe*, como uma rede social, onde apenas as pessoas inscritas no meio podem receber aqueles dados (a exemplo, temos os grupos privados do *Facebook*)**.**

ARQUITETURA CENTRALIZADA

Também chamada de arquitetura cliente-servidor, onde normalmente o cliente faz o pedido, requisição da informação, e o servidor então fornece esta informação ao cliente. Como um sistema de *streaming* (*Netflix, Amazon Prime Video*, etc).

Podemos dividi-la na camada de interface (relacionamento do cliente para com o servidor), o processamento (a requisição e fornecimento das informações e serviços requisitados) e o banco de dados (onde os serviços e dados são armazenados). São também divididas clientes gordos (*fat clients*), os quais fazem muito mais uso do sistema, e os clientes magros (*thin clients*), os quais fazem muito menos uso do sistema.

Ainda podem ser divididas em três camadas de servidor-cliente. Onde um servidor pode funcionar como um cliente da mesma forma. Recebendo o pedido do mesmo e, em seguida, requisitando as informações para uma terceira camada, onde por fim ele forneceria os serviços ao cliente da primeira camada.

ARQUITETURA DESCENTRALIZADA

Existem dois tipos de distribuição de componentes falando de arquitetura de sistemas distribuídos: a distribuição pode ser do tipo vertical, onde os componentes ficam em máquinas diferentes, isto é, cada componente possuem funções previamente definidas e distintas entre si, ou do tipo de distribuição horizontal, onde as funções serão divididas, o famoso sistema *peer-to-peer*, como o *Torrent*, que divide grandes arquivos em diversos tipos de *download*, este é o caso da arquitetura descentralizada. Diferentemente da arquitetura centralizada, aqui não temos a camada cliente-servidor pré-definida, mas sim o cliente será quem iniciar a requisição. Onde os lados podem ser alternados.

O tipo de sistema peer-to-peer (o tipo mais presente na arquitetura descentralizada), pode ser tanto estruturado (onde cada componente detém um código para identificar os componentes, a fim de procurar os arquivos corretos, tal

como seus dados) quanto não estruturadas (os dados são armazenados de forma aleatória, e a busca é feito da famosa “tempestade *broadcast*”, inundando a mesma, fazendo com que a mesma não tenha um desempenho tão bom).

Entretanto, existe outro tipo de arquitetura descentralizada do modo desestruturado, os *Super Pares*, que são um conjunto de dados que se unem, com uma espécie de *Gateway* (portão que intermedia a rede e um componente) que são os *SuperPares* (o componente com melhor conexão à rede), que se conecta aos outros próximos oferecendo uma melhor chance de identificação dos componentes.

ARQUITETURA HÍBRIDA

Possui tanto características da centralizada quanto da descentralizada, isto é, possui um arquivo central que contém o endereço dos componentes necessários a cumprir tal função, entretanto, o conteúdo, processamento e informações são distribuídas. Como aplicações de mensagens do tipo *WhatsApp* ou *Skype*.

Como os sistemas distribuídos colaborativos, onde o cliente procura por um dado, onde se dirige a um servidor que procura o conteúdo distribuído em diversos componentes (diversos nós), assim, cada um deles oferece os dados ao servidor que os redireciona de volta ao cliente.

2.4 DESAFIOS DOS SISTEMAS DISTRIBUÍDOS

Entretanto, além dos pontos citados acima, temos outros desafios que devem ser enfrentados a fim de se obter um sistema distribuído eficiente e consistente, desafios estes que podem se mostrar como verdadeiros problemas a serem enfrentados.

SEGURANÇA

Manter o nível de confiabilidade, garantir a integridade dos dados (proteção contra a corrupção ou alteração dos dados em questão), manter a disponibilidade do sistema (proteção contra interferências).

ESCALABILIDADE

A capacidade do sistema de se manter funcional independente do numero de usuários, isto é, independente do tráfego de usuários. Para isso, o *software* deve ser desenhado de forma que o numero de usuários não exija grandes alterações. E evitar algoritmos centralizados (os quais centralizam toda a operação em uma única maquina). Tal como controlar os custos e gerir o trabalho.

SISTEMA ABERTO

Tal como o próprio nome já dita, o sistema deve ser aberto para que se outros componentes sejam conectados a ele (sendo *hardware* ou *software*, como a

atualização, por exemplo) esses componentes possam conversar com aqueles que antes ali estavam. E para isso, devem ser conhecidas as especificações das interfaces destes componentes através de documentação.

TOLERÂNCIA A FALHAS

Um sistema tolerante a falhas implica que, no geral, uma falha em um componente do sistema, não deve comprometer o sistema como um todo. Para isso, deve haver:

- **Detecção das falhas:** Os dados corrompidos devem poder ser detectados de forma rápido e eficiente (além de que devem ser ocultados do usuário ao máximo). Esta detecção costuma ocorrer através da soma de verificações.

- **Localização das falhas:** O motivo da resposta ao pedido não ter chegado foi falha na rede ou falha no destino? E logicamente corrigi-la.

- **Mascaramento de falhas:** Uma falha pode ser mascarada através da replicação de dados, em mais de um ambiente, assim, se uma informação não chega, pode ser enviada a cópia dele presente em outro componente. Além de que, entre cada dois "*routers*" da *internet* (os famosos modens de *Wi-Fi*), devem sempre existir dois percursos.

3. CONSIDERAÇÕES FINAIS

Cada dia mais, sentimos a necessidade e presença dos *softwares* na nossa vida, presentes cada vez mais em áreas as quais uma falha pode comprometer uma vida, ou causar grandes custos. Assim, cada vez mais sentimos a necessidade de métodos e meios que garantam segurança e eficiência o suficiente. As especificações formais se baseiam especialmente nisso, diminuindo a margem de erros, além de que diminuem os custos do *software*, tendo que arcar menos com o custo de reparação.

Estamos tão acostumados com os *softwares* presentes nas nossas vidas que nem o percebemos, entretanto, este trabalho também tem como razão aos desenvolvedores que trabalham com a transparência desses sistemas.

Entender como tudo isto funciona, nos ajuda a entender melhor o mundo que nos rodeia. A enxergá-lo de outra forma, podendo ver além do oculto que ao nosso em torno existe.

4. BIBLIOGRAFIA

LIVROS

EMMERICH, Wolfgang. Engineering Distributed Objects. Edition 1st. New York, U.S.A. © John Wiley & Sons, 2000.

SOMMERVILE, Ian. Software Engineering. Edition 8th. Scotland, U.K. © Addison Wesley, 2006.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. Distributed Systems: Concepts and Design. Edition 3rd. London, England. © Addison-Wesley, 2001.

SITES

http://www.dsc.ufcg.edu.br/~jacques/cursos/apoo/html/plan/plan3.html,
Acesso em 06/ 07/ 2021, às 16h54min.

http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/uml/diagramas/usecases/usecases.html
Acesso em 06/ 07/ 2021, às 17h48min.

https://micreiros.com/os-diagramas-comportamentais-da-uml/,
Acesso em 07/ 07/ 2021, às 14h35min.

https://www.devmedia.com.br/como-usar-os-metodos-formais-no-desenvolvimento-de-software/31339, ***Acesso em 14/ 07/ 2021, às 12h19min.***

http://softwareseguro.blogspot.com/2012/10/requisitos-de-seguranca-de-software.html,
Acesso em 14/ 07/ 2021, às 13h24min.

https://blog.convisoappsec.com/requisitos-de-seguranca-asvs/, ***Acesso em 14/ 07/ 2021, às 13h35min.***

https://www.dnv.com.br/services/sil-nivel-de-integridade-de-seguranca--75360, ***Acesso em 14/ 07/ 2021, às 13h38min.***

*https://www.di.uminho.pt/~jfc/research/techrep/Campos93/node7.html***,**
Acesso em 14/ 07 /2021, às 17h48min.

https://www.cursosdeinformaticabasica.com.br/o-que-e-interface/ ***Acesso em 15/ 07/ 2021, às 20h36min.***