Análise de dependabilidade em um sistema de integração

Tarcísio Batista de Freitas Junior

[1]Departamento de Ciência da Computação - Universidade de Brasília

{tarcisiodf23@gmail.com}

Resumo. *Artigo final do trabalho da disciplina de "Dependabilidade de Sistemas Computacionais", ministrada pelo professora Genaína Nunes no segundo semestre de 2019. O foco do trabalho é a modelagem e análise de um sistema de integração contínua[1], responsável por otimizar as etapas de um processo de construção e exposição de um sistema. Os dados probabilísticos foram criados randomicamente, com o intuito de simular desenvolvedores e sistemas de diferentes níveis e complexidade. Foram utilizadas técnicas de modelagem e análise ensinada em aula, tal qual o uso da ferramente PRISM juntamente com o uso da linguagem PCTL, para que seja feita uma análise do que pode ser trabalhado no sistema para evitar erros e agilizar o lançamento de um sistema.*

1. Introdução

De acordo com Laprie and B. Randell[5], dependabilidade é definida como a eficácia de um sistema de entregar um serviço confiável. É uma especialidade de sistemas que possuem o foco nas áreas de computação, e devido a importância de quem o utiliza e de onde o software é executado, a dependabilidade de sistemas tem sido uma área que aumentou o seu campo de pesquisas desde a sua implementação até a fase de testes de softwares[7].

Neste trabalho, serão aplicadas técnicas probabilísticas para análise de sucesso de sistemas de integração contínua. O sistema foi analisado através de diagramas de estado e sequência, para posterior checagem probabilística no PRISM [2], que como já citado anteriormente, utiliza-se da linguagem PCTL para confirmação de pontos importantes da dependabilidade.

É um ponto extremamente importante a ser ressaltado, que os dados utilizados são randômicos para simulação de diferentes níveis de times de programação, com complexidades variáveis de sistemas a serem desenvolvidos. De acordo com a arquitetura modelada pela equipe de infraestrutura, esse sistema pode ser mais preciso, ou até mesmo mais genérico, dependendo das necessidades de cada equipe[1].

O trabalho se organiza de maneira simplória, apresentada através de seções, aonde a segunda apresenta uma introdução teórica sobre um sistema de integração contínua, seu funcionamento de modo geral, tais quais suas características. Na seção 3, é apresentada a metodologia apresentada no trabalho para análise. Na seção 4 são apresentadas as propriedades avaliadas. Na seção 5, é apresentado o resultado. Por último, a conclusão juntamente com a referência bibliográfica.

2. Integração Contínua

Para se entender os pontos analisados no trabalho, é necessário uma introdutória teórica sobre o assunto, aqui apresentada.

A integração contínua é uma técnica cada vez mais utilizada, de se centralizar os códigos de desenvolvedores , com intuito de, a cada alteração a ser feita no código, em determinada *branch*, *scripts* são executados para que ocorram testes, encontrar e investigar *bugs* mais rapidamente, melhorar a qualidade do software e otimizar o tempo para validação de novas alterações ou correções, com a finalidade de buildar e já expor o novo compilado ao cliente.

De acordo com as necessidades de cada equipe de desenvolvimento, são criados scripts que permitem e validam qualidade do código, erros sintáticos, padrões não seguidos, ou outros técnicas que favorecem a melhoria da qualidade do código. Além do controle de qualidade, outra grande importância da integração contínua é a automação de geração de compilados para teste do desenvolvedor e análise do cliente. Uma das ferramentes mais conhecidas de integração contínua hoje é o Jenkins[3], mostrado abaixo:

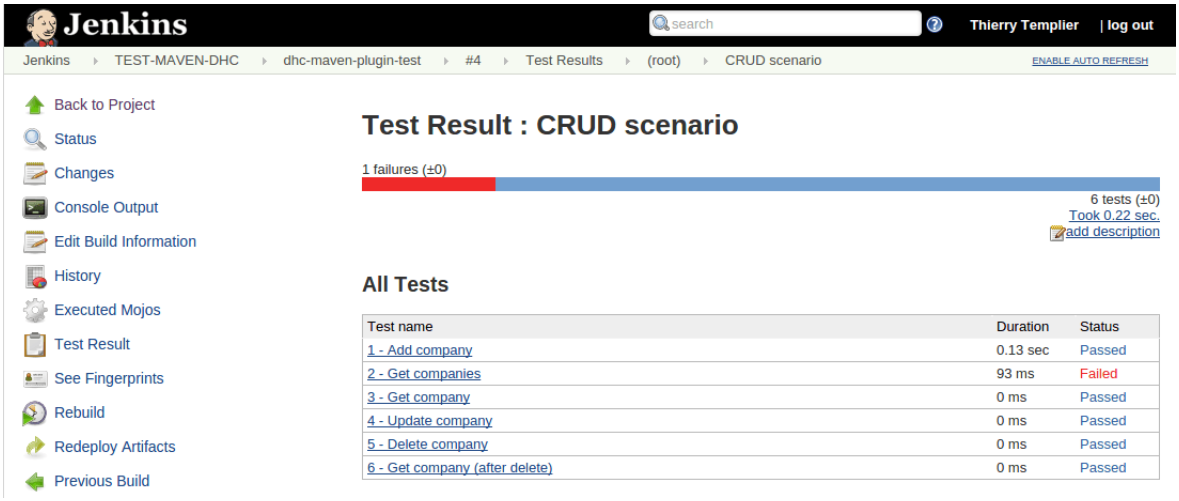

Figura 1. Sistema de CI Jenkins[3]

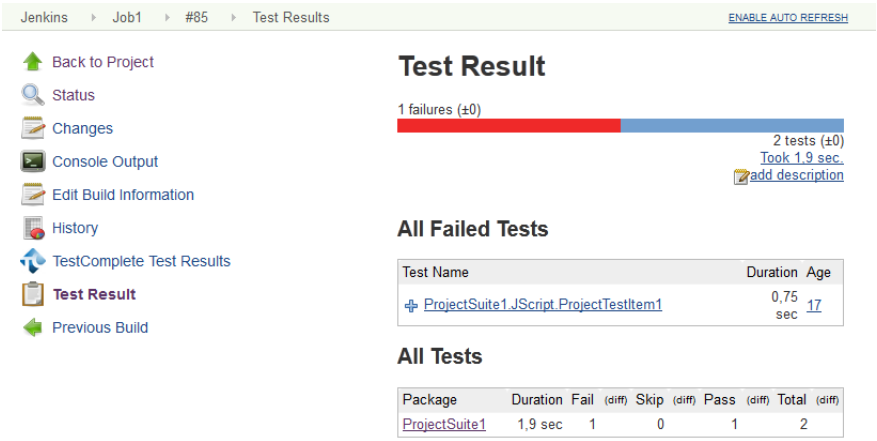

Figura 2. Resultado de testes dos Jenkins

2.1. Requisitos Técnicos

Para que o sistema de integração contínua consiga atingir o estado mostrado no diagrama, é necessário que saber quais pontos são utilizados na arquitetura do mesmo. Como mos-

Figura 3. Deploy realizado pelo Jenkins apés testes passarem com sucesso

trado no diagrama de sequência mostrado na imagem 7, o primeiro ponto é o requisito de sistema, que chega com as especificações necessárias do sistema. O próximo passo é a atualizadação , por parte do desenvolvedor, do código a ser alterado. Pra que isso aconteça, é calculada a probabilidade do desenvolvedor estar com acesso à rede, possibilitando que ele se conecte ao servidor do versionador de código, aqui utilizando o GIT. A segunda necessidade técnica é que o servidor GIT esteja no ar e com acesso à rede. Por último, o terceiro requisito técnico é que o servidor de *build* e *deploy* esteja no ar e conectado à rede. Para o estado final de sistema deployado, o código deve passar pelos testes do sistema[7], além do mesmo possuir espaço em disco pra lançar a nova versão do sistema.

2.2. Fases do processo

Para análise do trabalho, os dados e os valores probabilísticos são provindos do seguinte pelas seguintes fases :

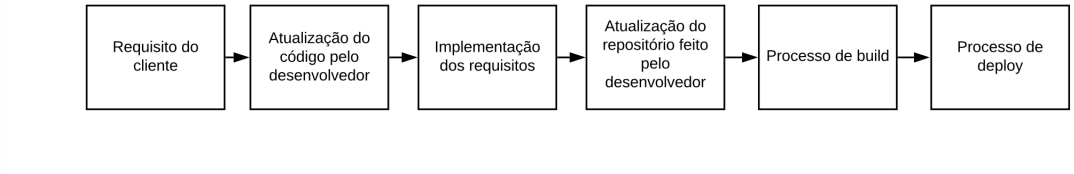

Figura 4. Processo de análise de dados de um sistema de integração contínuo

Como já visto no diagrama de sequência, o processo inicial começa do requisito recebido pelo desenvolvedor. As *features* são identificadas e começa o processo de implementação. Nesse processo, são analisadas as probabilidades do sistema chegar ao seu estado final. Portanto, os dados de interesse são as chances de possíveis erros dados por problemas na qualidade do código- o que pode ser evitado utilizando-se boas práticas[6], ou na conjuntura da infraestrutura.

2.3. Fluxo da integração contínua

O fluxo da CI é simplificado pela imagem 5.

3. Metodologia

A atividade inicial foi a escolha do tema, juntamente com sua modelagem e arquitetura utilizando o PRISM. Para isso, foram utilizados valores randômicos que acredito ser uma estimativa genérica para uma boa quantidade de espaço amostral.

Como parte do projeto, foram desenhados diagramas de estado e sequência do modelo a ser analisado no artigo. É mostrado abaixo o diagrama de sequência gerado para demonstrar a ordem de execução do fluxo :

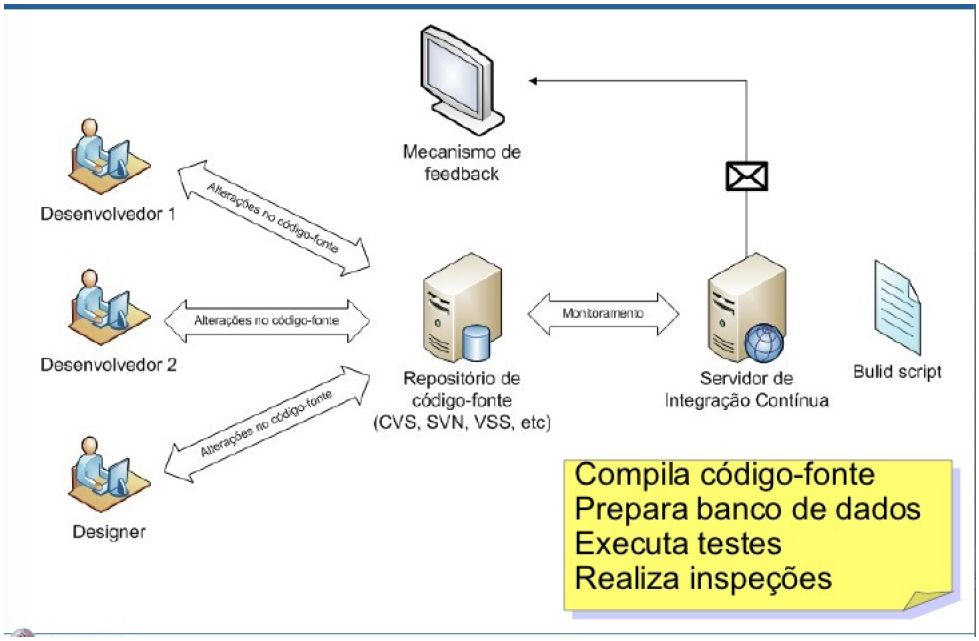

Figura 5. Fluxo de Sistema de integração contínua

Com o diagrama de sequência, houve a criação do modelo no PRISM. Para checagem dos valores probabilísticos, utilizou-se a linguagem PCTL. As seções 4 e 5 mostram como ocorre a checagem [4].

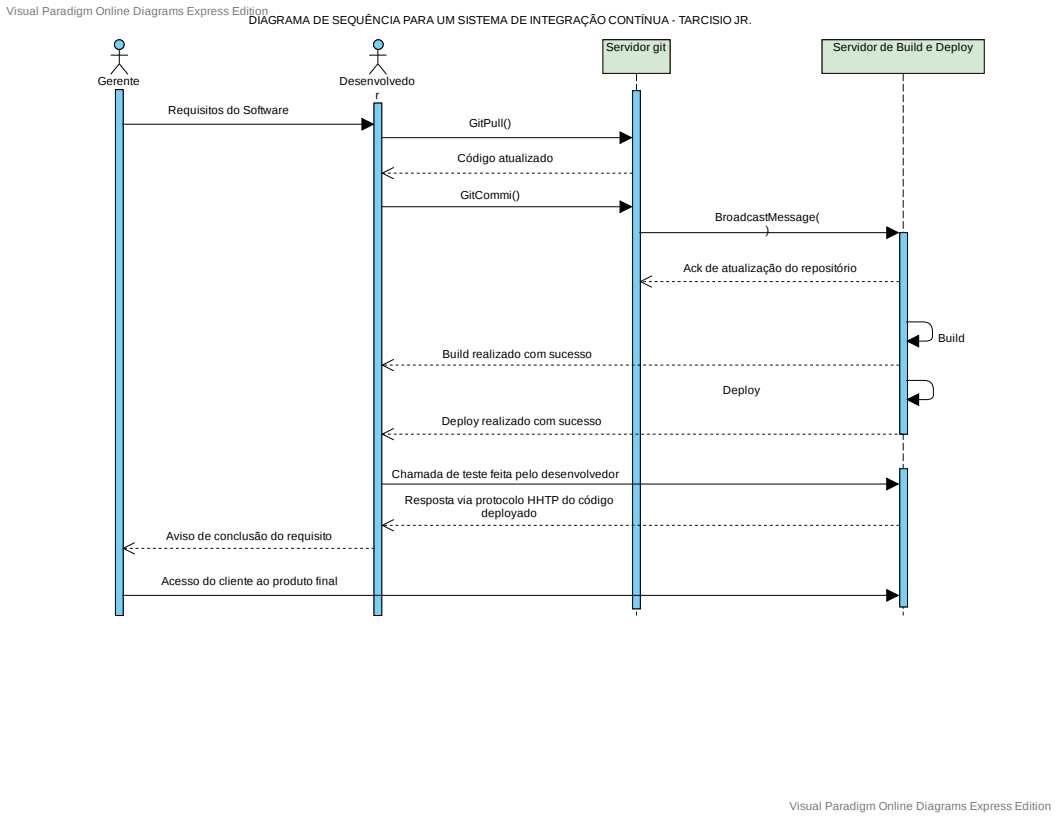

Figura 6. Diagrama de sequência do sistema de integração continua

4. ProProst

Com intuito de uma descrição preliminar das propriedades escritas em PCTL do sistema de integração contínua, alguns indicadores foram selecionados. Nas tabelas a seguir estão listados os indicadores selecionados e na tabela 2 a especificação preliminar com sua propriedade correspondente para checagem.

Fase	**Indicador**	**Definicao**	**Forma de obtencao**
Atualização do código	1)Avaliação de desatualização	Avalia se o código do desenvolvedor está atualizado e se o mesmo possuir internet.	Coloca-se o valor de probabilidade do computador estar com conexão, juntamente com o a necessidade de atualização do código.
Checagem de respositório.	2) Avaliação de repositório.	Avalia para saber se o gerenciador de código está no ar e com acesso à rede, para atualização.	Pega-se o valor da probabilidade da máquina do controlador de versões estar no ar e com acesso à rede.
Build e Deploy	3) Build e deploy do sistema	Builda, gerando o compilado do novo código, e faz o deploy para teste do desenvolvedore posterior uso do cliente.	Calcula-se a probabilidade da máquina de build e deploy estar no ar, com a chance do sistema já ocupado. Por último, calcula-se a chance de erro sintático no código.

Indicador 1	
ProProst	O sistema tem um comportamento de continuidade de execução do fluxo com 99% de chances.
Propriedade	P>=0.99(computerHasInternetAccess)
Indicador 2	
ProPost	O sistema tem um comportamento de continuidade de execução com a possibilidade do servidor git estar no ar e com conexão com internet.
Propriedade	P = ? [(26/27 : serverIsOn) * (0.99 : serverHasInternetAccess)]
Indicador 3	
ProPost	O Sistema tem continuidade e chega ao seu estado final de deploy do sistema. Para isso, é analisado se o servidor de build e deploy está no ar, com conexão com internet, com espaço para armazenamento para o código compilável, e que o código não possua erros sintáticos.
Propriedade	P = ? [(32/33: serverIsOn) * (0.99 : serverHasInternetAccess) * (0.935: serverIsNotBusy) * (0.95: noErrorOnCode)]

5. Simulação usando PRISM

Com o objetivo de fazer uma análise baseando-se nos diagramas de sequência e de estado mostrados anteriormente, foi-se modelado o sisstema de integração contiínua no PRISM. Na análise de sensibilidade , a porcentagem de confiabilidade oscilava bastante.

Abaixo é mostrado parte do código do PRISM utilizado para modelagem :


```
// Trabalho de dependabilidade de software
// Objetivo: trabalho visa modelar um sistema de integração contínua.
dtmc

const double git_chance_turn_off_server = 1/27;
const double continuous_chance_turn_off_server = 1/19;
const double build_chance_turn_off_server = 1/33;

const double sintatic_error_on_code = 1/7;

module hasInternet

        // A maquina possui conexao com a internet
        // 1 - erro de conexao
        // 2 - conexao com sucesso
        hasInternetAccess : [0..2] init 0;

        [] hasInternetAccess = 0 -> 0.01:(hasInternetAccess'=1) + 0.99:(hasInternetAccess'=2);
        [] hasInternetAccess = 1 -> 1:(hasInternetAccess'=1) ;
        [connected] hasInternetAccess = 2 -> 1:(hasInternetAccess'=2);

endmodule

module developerComputerHasInternet = hasInternet [hasInternetAccess = computerHasInternetAccess] endmodule

module gitServerOnAir

        // 1 - com problema
        // 2 - servidor git no ar
        gitServerStatus: [0..2] init 0;

        [] gitServerStatus = 0 & (hasInternetAccess = 2) -> git_chance_turn_off_server:(gitServerStatus'=1) + (1 - git_chance_turn_off_server):(gitServerStatus'=2);
        [] gitServerStatus = 0 & (hasInternetAccess = 1) -> 1:(gitServerStatus'=1);
        [] gitServerStatus = 1 -> 1:(gitServerStatus'=1);
        [onAir] gitServerStatus = 2 -> 1:(gitServerStatus'=2);

endmodule

module continousServerOnAir = gitServerOnAir [gitServerStatus = continuousServerStatus, git_chance_turn_off_server = continuous_chance_turn_off_server,gitOnAir = continousOnAir] endmodule

module updateCodeAndCommit

        // 1 - problema no update e commit
        // 2 - updateCode and commit com sucesso
        updateCodeAndCommit: [0..2] init 0;

        [onAir] updateCodeAndCommit = 0 & (computerHasInternetAccess = 2) & (gitServerStatus = 2 ) -> 1:(updateCodeAndCommit'=2);
        [] updateCodeAndCommit = 0 & (computerHasInternetAccess = 1) | (gitServerStatus = 2 ) ->  1:(updateCodeAndCommit'=2);
        [] updateCodeAndCommit = 1 -> 1:(updateCodeAndCommit'=1);
        [commited] updateCodeAndCommit = 2 -> 1:(updateCodeAndCommit'=2);

endmodule

module buildOnAir = gitServerOnAir [gitServerStatus = buidServerStatus, git_chance_turn_off_server = build_chance_turn_off_server ,gitOnAir = buildOnAir] endmodule
module buildDeploy

        // 0 - buidServerStatus está no ar, entao verifica erro sintatico.
        // 1 - verifica erro sintatico no codigo
        // 2 - verifica erro de falta de capacidade.
        // 3 - buildado com sucesso.
```

Figura 7. Parte do código de modelagem utilizando o PRISM

Para a simulação, foi considerado uma confiabilidade de cada componente de 100%, visto que erros de confiabilidade sistemas computacionais como Jenkins, são extremamente baixos, e por isso, em boa parte dos estudos, o erro(que no caso deveria ser instrumental), acaba sendo menosprezado.

Propriedade	Descrição	Resultado
P = ? [F (hasInternetAccess = 2)]	Desenvolvedor com acesso à internet,	99%.
P = ? [F (gitServerStatus = 2)]	Servidor do git está disponível e online.	95,34%
P = ? [F (updateCodeAndCommit = 2)]	Atualizar o códito e commitar com sucesso.	95,38%
P = ? [F (buildStatusServer =4)]	O sistema foi deployado com sucesso.	70,4%

O modelo desenvolvido no PRISM tem uma alteração muita sensível nos valores determinados para o útimo módulo(build e deploy), como mostrado na próxima seção.

5.1. Resultados Obtidos

Com os resultado mostrado através do PRISM, é perceptível que o módulo mais sensível acaba se tornanndo o de build e deploy. Nele se encontram 3 ações que podem gerar erro no alcance para se alcançar o estado final do diagrama.

E a principal delas é a checagem de erros pelo teste do sistema integrado. O valor padrão é de 14%, deixando em aproximadamente 70% a chance do estado final ser alcançado. Se utilizássemos um valor de confiabilidade baixo pra esse modulo, ele se tornaria menor que 50%, quase que inviabilizando o uso do sistema, que tem como função

primordial a otimização do tempo e ganho de performance das equipes de desenvolvimento.

6. Conclusão

Deve-se ter muito cuidado ao configurar o sistema de integração contínua. Um fator extremamente relevante é que a equipe de infraestrutura deve estar bem alinhada com os desenvolvedores[6], pois se as regras determinadas de testes não forem bem explicadas, podem ocorrer erros desnecessários.

Outro ponto importante manutenção contínua do máquina física do servidor. Deve-se configurar de modo a modularizar os compilados, tanto qunto seus logs, para que o melhor uso possível do *hardware* disponível para o servidor.

Referências

[1] https://aws.amazon.com/pt/devops/continuous-integration/.

[2] www.prismmodelchecker.org.

[3] https://jenkins.io/.

[4] Lars Grunske. “Specication Patterns for Probabilistic Quality Properties”. Em: (2008), pp. 31–40.

[5] J.C. Laprie e B. Randell. “Fundamental concepts of dependability”. Em: (2001).

[6] Robert C. Martin. *Código limpo: Habilidades Práticas do Agile Software*. 1th. Alta Books, 2009.

[7] Marcos Danilo Chiodi Martins. *Testes de software*. 1th. Estácio, 2016.