Rayana Vasconcelos de Sá Alencar

UTILIZAÇÃO DE TÉCNICAS DE *PROCESS MINING* EM SISTEMAS DE MIDDLEWARE ADAPTATIVOS

Trabalho de Graduação

Universidade Federal de Pernambuco
www.cin.ufpe.br

RECIFE
2017

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

Rayana Vasconcelos de Sá Alencar

UTILIZAÇÃO DE TÉCNICAS DE *PROCESS MINING* EM SISTEMAS DE MIDDLEWARE ADAPTATIVOS

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Nelson Souto Rosa*

RECIFE
2017

Não diga que a vitória está perdida se é de batalhas que se vive a vida.

—RAUL SEIXAS

Agradecimentos

Dedico esse trabalho a minha família. Meu pai Aécio e minha mãe Simone que são meus maiores torcedores, as minhas irmãs que sempre me ajudam da forma que podem, seja Rhayssa corrigindo meus textos e Rebeca na parte da computação. Agradeço aos meus colegas e amigos de turma que fizeram dessa jornada um ambiente mais amigável, especialmente ao meu namorado João Gabriel. Por fim, quero agradecer ao Centro de Informática por proporcionar uma educação de altíssima qualidade e ao professor Nelson Rosa por ter oferecido uma oportunidade de iniciação científica durante a minha graduação, pela orientação neste trabalho de conclusão de curso e por está sempre disponível para responder as minhas dúvidas.

Abstract

Distributed systems and IoT devices are increasingly present in people's lives. One of the main difficulties in building an application for these types of systems is due to the fact that they should be able to be executed on multiple devices that have different configurations. Middleware systems have the function of helping the development of this type of application, and this is possible because a middleware acts by concealing the different between the device environments, so the developer does not have to worry about those differences. In addition, changes may occur during the execution of an application, such as a network change where the application is connected; they require that systems are capable of adapting to new scenarios. It is possible to use process mining techniques to develop adaptive middleware systems. These techniques have gained considerable prominence because of its importance to organizations; the goal of process mining is to extract information from system records to better understand its operation. Thus, the purpose of this work is to use these techniques during the execution time of adaptive middleware systems in order to identify when it is necessary to adapt them.

Keywords: Middleware Systems, Distributed Systems, Internet of Things, Process Mining

Resumo

Os sistemas distribuídos e *IoT devices* estão cada vez mais presentes na vida das pessoas. Uma das principais dificuldades na construção de aplicações para esses tipos de sistemas deve-se ao fato de que elas podem ser executadas em vários dispositivos que possuem diferentes configurações. Os sistemas de middleware tem a função de ajudar no desenvolvimento desse tipo de aplicação. Isso é possível porque o middleware atua escondendo as diferenças existentes entre os ambientes dos dispositivos, de modo que o desenvolvedor não precise se preocupar com essas diferenças. Além disso, podem ocorrer alterações durante a execução de um aplicativo, como uma mudança de rede onde o aplicativo está conectado, tais mudanças tornam necessário o desenvolvimento de sistemas capazes de se adaptar a novos cenários. Para desenvolver sistemas de middleware adaptativos é possível utilizar técnicas de mineração de processos. Essas técnicas vêm ganhando bastante destaque devido a sua importância para organizações e têm como objetivo extrair informações de registros de sistemas para entender melhor seu funcionamento. Assim, a finalidade deste trabalho é utilizar dessas técnicas durante o tempo de execução de sistemas de middleware adaptativos para identificar quando é necessário adaptá-los.

Palavras-chave: Sistema de middleware, Sistemas distribuídos, *Internet of Things*, Mineração de processos

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

Sumário

1 Introdução

1.1 Contextualização e motivação

Middleware é uma infraestrutura de software que se encontra entre a aplicação e o sistema operacional, redes e hardwares adjacentes, que tem como finalidade ajudar no desenvolvimento e nas operações de sistemas distribuídos [2]. Uma das principais dificuldades ao se construir uma aplicação distribuída é conseguir fazer com que dispositivos em ambientes diferentes, sejam de sistemas operacionais, protocolos de redes, hardware, linguagem de programação e outros, consigam operar em conjunto. O middleware atua como uma espécie de máscara de heterogeneidade, tornando bem mais simples o desenvolvimento de software distribuídos ou multiplataforma.

Supondo-se que um desenvolvedor vai criar uma aplicação distribuída, se ele/ela tivesse que se preocupar com quais os tipos de sistemas operacionais podem executar a aplicação, se os dispositivos podem ser móveis ou não, em como vai ser um protocolo para as trocas de mensagens entre esses dispositivos, ou como a aplicação vai fazer uma requisição para o sistema. Sabemos que existem diferentes tipos de hardwares, softwares e redes em que a aplicação pode ser executada, se preocupar com todos esses casos ou até mesmo que seja com apenas alguns deles, o desenvolvedor teria um trabalho muito grande. Com um middleware, o engenheiro de software não vai precisar mais tratar esses problemas de heterogeneidade do ambiente, mas sim apenas como a interação da aplicação com o middleware.

Durante a execução de um sistema é esperado que ocorram eventos que afetem o seu funcionamento, por exemplo, em um sistema distribuído onde um dos dispositivos deixa de responder, o middleware poderia ter como solução esperar um tempo para fazer a requisição novamente ou encaminhar a requisição para outro dispositivo antes de mandar uma mensagem de erro ao usuário. Portanto, um middleware tem que ser tolerante a falhas, pois a existência desses cenários tornam-se necessárias adaptações para diminuir o impacto desses eventos no funcionamento da aplicação em geral [3].

Como visto, o middleware é uma camada de software que esconde a heterogeneidade dos dispositivos e a complexidade da distribuição, facilitando a vida dos desenvolvedores de

sistemas distribuídos. Além disso, essa camada vai ser responsável por monitorar, coordenar e ser tolerante a falhas, de forma transparente ao usuário durante a execução do sistema. Assim, como se tem estratégias para solucionar possíveis erros durante a execução da aplicação, existem também sistemas de middleware que são capazes de se adaptar internamente, com objetivo de sempre manter o mesmo nível de qualidade independente do cenário, eles são denominados de sistemas de middleware adaptativos.

1.2 Problema

Sabemos que os sistemas de middleware são utilizados principalmente em sistemas distribuídos, como a computação nas nuvens e na computação ubíqua, que cada vez mais estão presentes no cotidiano da sociedade. Ao desenvolver uma aplicação para esses ambientes, é necessário levar em conta os diferentes dispositivos existentes (smartphones, notebooks, tablets, desktops, televisores e outros) e os diferentes sistemas operacionais (Linux, macOS, Windows, Android, iOS, Windows Mobile, GNU e outros). Além disso, é preciso considerar os recursos que variam durante a execução de um sistema. Como exemplos desses recursos temos: a quantidade de memória disponível, a carga da bateria, a configuração, a conexão com uma rede de comunicação, as políticas de segurança diferentes que variam dependendo do domínio [4].

Conseguir lidar com tantas variáveis utilizando um sistema estático é bastante complicado. Sendo assim, com o objetivo de garantir que a qualidade dos serviços oferecidos por um middleware seja sempre a mesma, independentemente do ambiente e também para facilitar a tarefa dos desenvolvedores de aplicações, de forma que eles não tenham que se preocupar com os diferentes cenários que possam acontecer durante a execução do programa, surgiu a ideia de construir sistemas de middleware que se adaptem à mudanças do ambiente sem precisar interromper sua execução.

Para construir um middleware adaptativo é preciso responder 3 principais perguntas: **Como** será feita a adaptação ? **onde** o middleware precisará ser modificado ? **quando** ela deve ser realizada ? [5].

Existem duas formas principais de se responder como fazer adaptações de sistema, a primeira é através da adaptação paramétrica e a segunda é a adaptação composicional. A atualização de parâmetros apesar de funcionar bem, como no protocolo TCP, existe a limitação de não poder modificar algoritmos ou adicionar componentes durante o tempo de execução. Já a adaptação composicional, permite a troca de algoritmos e componentes em tempo de execução. Esse tipo de adaptação é bastante utilizado e as formas mais comuns de implementá-la é através de reflexão computacional, o desenvolvimento baseado em componentes e a programação orientada a aspectos [3].

Schmidt estrutura o middleware em quatro camadas como mostra a Figura 1.1 [6]. Essas camadas são: infraestrutura, distribuição, serviços comuns e serviços específicos.

Na camada de infraestrutura é onde ocorre a abstração do hardware e do sistema operaci-

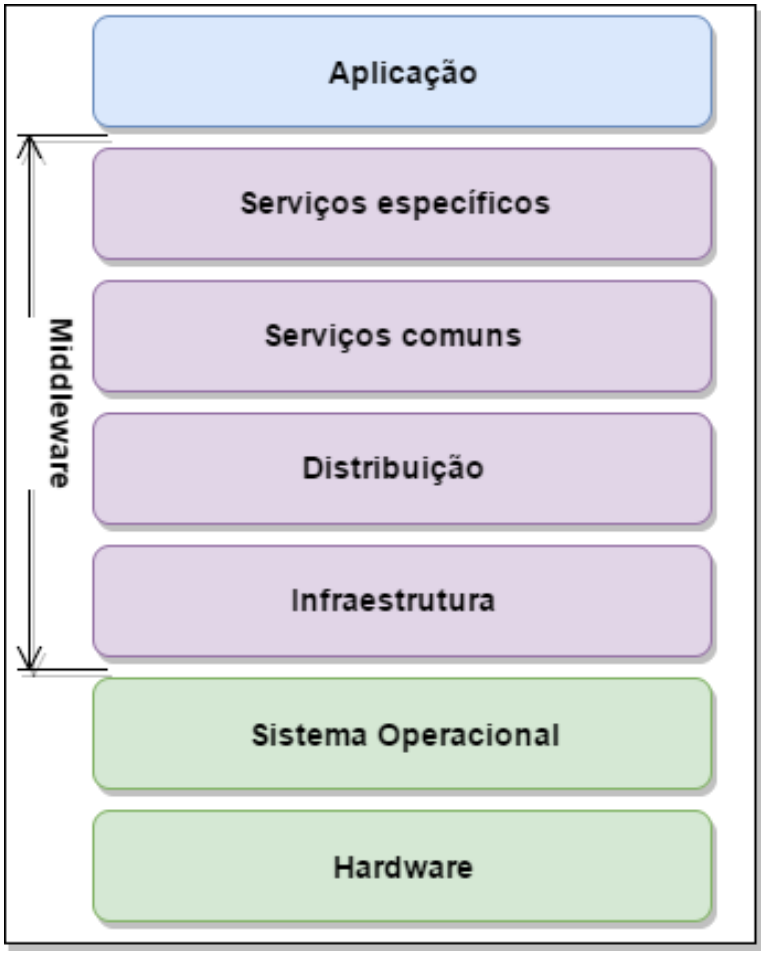

Figura 1.1: Camadas de um middleware.
Adaptado de [6].

onal. Já a camada de distribuição permite ao desenvolvedor escrever programas em alto-nível. Na camada de serviços comuns é onde ficam os serviços utilizados por aplicações de diversos domínios diferentes, como os serviço de nome e o serviço de tolerância a falha. Por fim, na camada mais próxima da aplicação, chamada de camada de serviços específicos é onde estão contidos os serviços necessários à aplicações de domínios específicos [7].

Como vimos, o middleware pode ser dividido em quatro camadas, responder em qual delas devem ocorrer as adaptações vai depender bastante da escolha do desenvolvedor. É possível também escolher a aplicação como o local para adaptar.

Quando o middleware é adaptado em tempo de execução, esta adaptação é dinâmica. Se ocorre alguma adaptação em tempo de desenvolvimento ou de compilação ou no de carregamento do programa, esta é chamada de adaptação estática. Quando a adaptação é estática fica mais fácil de garantir que uma modificação não irá alterar o comportamento do sistema, já a adaptação dinâmica suporta métodos de adaptação mais poderosos. Porém, modificar um sistema em tempo de execução torna mais difícil de se utilizar testes tradicionais e técnicas de verificação formal, técnicas essas que verificam a segurança e outra propriedades de correção [3].

1.3 Trabalhos relacionados

Existem diversos projetos com diferentes implementações de middleware adaptativo, dentre eles podemos citar dois projetos: o dynamicTAO [4] e o MidARCH [1].

1.3.1 DynamicTAO

O dynamicTAO surgiu com a ideia de se criar um ambiente que fornecesse os requisitos necessários para modificar um middleware. Nesse projeto, decidiu-se não implementar um *Object Request Broker (ORB)* do zero, mas sim utilizar o TAO que é um tipo de ORB *open source*, portável e flexível.

O TAO ORB é estático durante sua execução, portanto foi necessário utilizar o conceito de reflexão computacional para modificar dinamicamente as suas estratégias, que são definidas antes da execução, com isso surgiu o dynamicTAO.

Para identificar quando é necessário uma modificação no middleware, foi criado um serviço chamado 2K *Monitoring Service* que pode ser acoplado e desacoplado ao dynamicTAO em qualquer momento. Esse serviço vai monitorar a execução do sistema, guardando cinco tipos de informações sobre uma requisição do cliente: o endereço da máquina do cliente, o nome do objeto que está sendo monitorando, o nome da operação feita pelo objeto, o *timestamp* e a duração do lado do servidor. Os arquivos gerados no monitoramento são guardados em um arquivo do sistema ou em um banco de dados. Existem ainda duas interfaces de acesso às informações coletadas, uma para publicar e a outra para receber consultas do usuário. Como exemplo de consulta que a interface recebe, temos: "Quando foi a última chamada à operação A no objeto X?" [4].

1.3.2 MidARCH

O MidARCH é uma framework de middleware e um ambiente de execução especialmente projetado para ajudar no desenvolvimento e na execução de sistemas de middleware adaptativos.

O framework é formado de doze tipos diferentes de componentes, que são definidos de acordo com *Remoting Patterns* [8], e quatro tipos diferentes de conectores que permitem a adaptação. Além disso, eles possuem uma descrição do comportamento em *Communicating Sequential Processes (CSP)*.

O ambiente de execução é responsável por executar o middleware, fazer verificações estáticas e dinâmicas e se necessário modificar o sistema. Ele consiste em dois elementos principais, o *Execution Engine* e o *Adaptation Manager*, como podemos ver na Figura 1.2.

O *Execution Engine* é responsável por orquestrar a execução dos componentes e conectores. Nesse elemento são gerados os grafos estruturais e comportamentais a partir da arquitetura definida no framework. O RabbitMQ[1] é o middleware responsável por coletar informações sobre

[1] http://www.rabbitmq.com/

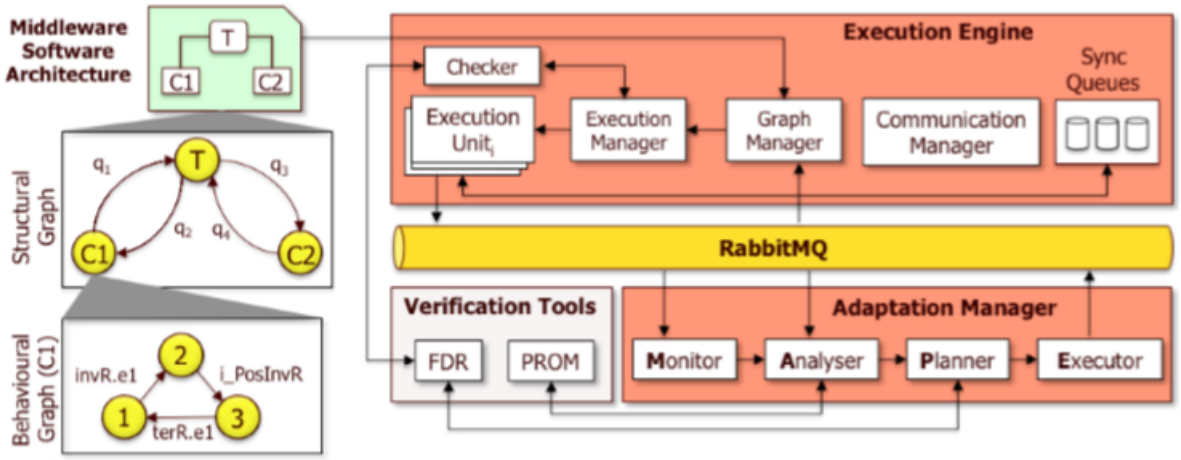

Figura 1.2: Execution environment do MidARCH [1].

o sistema e passar para o *Adapatation Manager*.

O *Adaptation Manager* implementa a arquitetura MAPE-K proposta pela IBM e é responsável por fazer adaptações quando necessárias. Ademais, esse elemento utiliza o FRD[2] e o ProM[3] para fazer uma análise no middleware e encontrar algum erro. O FDR analisa os grafos gerados no *Execution Engine* e o ProM o comportamento do middleware. Depois de feitas as análises, se os resultados indicam que é necessária alguma modificação no middleware, elas serão feitas de acordo com o que foi planejado para as propriedades que foram violadas [1].

1.4 Solução

Nesses últimos anos, além do aumento do desenvolvimento de sistemas distribuídos e de *Internet of Things* (IoT), houve também o crescimento do destaque e da importância de mineração de processos.

Existem diversos sistemas de informações utilizados para gerar logs de eventos de algum processo. Esses logs geralmente contém informações como: o nome da atividade realizada (*activity name*), em qual momento ela foi realizada (*timestamp*), a qual instância do processo ela pertence (*process id*) e quem realizou a atividade (*resource name*). A mineração de processos é o nome que se dá as análises feitas nestes logs.

O objetivo de minerar os logs gerados por uma sistema é de descobrir informações, monitorar e melhorar o processo do qual o log foi extraído. É possível gerar, por exemplo, uma rede de Petri a partir do log extraído de um sistema. A importância disso é que muitas vezes um sistema é modelado para funcionar de uma maneira, mas durante sua execução ele apresenta um comportamento diferente. Com a mineração de processos é possível entender e encontrar possíveis problemas que existem em um processo [9].

[2]https://github.com/jgrapht/jgrapht
[3]http://www.promtools.org

Como vimos, para se construir um middleware adaptativo é importante responder a três principais perguntas: **Como** será feita a adaptação?, **quando** ela deve ser realizada ? e **onde** o middleware precisará ser modificado ?. Existem diferentes formas de responder a pergunta de como será feita adaptação, dentre elas temos a reflexão computacional, desenvolvimento baseado em componentes e a programação orientada a aspectos.

O objetivo deste trabalho é construir um módulo capaz de responder **quando** é preciso fazer uma adaptação em tempo de execução de um middleware. Para implementar um módulo capaz de responder a essa pergunta é necessário monitorar o seu funcionamento e identificar um comportamento não desejável. Por isso, decidimos investigar como utilizar técnicas de mineração de processo, como foi proposto pelo MidARCH, para identificar quando é necessária fazer uma adaptação no sistema.

1.5 Estrutura do Trabalho

Este trabalho é composto por cinco capítulos. O primeiro capítulo é a introdução, nele foi abordado uma explicação sobre sistemas de middleware, a necessidade de sua adaptação, a descrição do problema que desejamos resolver, como esse problema já foi abordado por outras pessoas e a nossa proposta de estudo.

No segundo capítulo é apresentada a ferramenta de mineração de processo utilizada na solução e a descrição da arquitetura MAPE-K. No terceiro capítulo é explicado com detalhes a solução implementada e onde ela se encontra na arquitetura geral do middleware. No quarto capítulo, temos a explicação e os resultados dos testes realizados para medir o *overhead* de se utilizar essa técnica para implementação de um middleware adaptativo. Por fim, temos o último capítulo que conclui o trabalho e apresenta propostas de melhorias.

2

Background

O objetivo deste trabalho é utilizar técnicas de mineração de processos para identificar *quando* realizar adaptação de sistemas de middleware. Para um melhor entendimento da solução proposta, este capítulo apresenta uma breve explicação sobre o ProM que é a ferramenta de mineração utilizada. Também apresentamos a arquitetura que o middleware deve implementar para utilizar a solução proposta e, por fim, descrevemos a Lógica Temporal Linear utilizada para descrever as propriedades checadas na mineração de processos.

2.1 ProM

Existem diversas ferramentas capazes de realizar mineração, por exemplo: ARIS PPM[1], HP BPI[2] e ILOG JViews[3]. As ferramentas possuem padrões diferentes de construção e leitura de log, o que torna bastante difícil utilizar mais de uma delas numa mesma base de logs. Para resolver esse problema, pesquisadores da Universidade Tecnológica de Eindhoven desenvolveram uma framework *open-source* chamado de ProM.

O ProM é independente de plataforma, foi desenvolvido em Java e atualmente se encontra na Versão 6.6. Nesse framework existem diversas técnicas de mineração de processos que são classificados em três categorias: *Discovery*, *Conformance* e *Extension*. Essas técnicas estão no formato de *plugins* e qualquer desenvolvedor Java pode adicionar novos *plugins* no ProM [10].

As técnicas de *Discovery* são as que recebem como entrada um log de evento e apenas com isso conseguem criar um modelo de processo, exemplos de *plugins* que pertencem a essa categoria são: *Alpha algorithm*, *Genetic mining* e *Fuzzy Miner*.

Fazem parte da categoria de *Conformance* as técnicas que recebem como entrada a descrição de um modelo de processo e um log, o objetivo delas é verificar o quanto as informações presente no log correspondem aos comportamentos presente no modelo, exemplos de *plugins* pertencente a essa categoria são: *LTL-Checker* e *Conformance Checker*.

[1] http://www.ariscommunity.com
[2] https://softwaresupport.hpe.com/document/-/facetsearch/document/KM00883435
[3] https://www.ibm.com/developerworks/downloads/ws/jviews/index.html

As técnicas de *Extension* são as que recebem como entrada um modelo de processo e seu log, esse tipo de técnicas tem como finalidade encontrar melhorias para o modelo, o *Discovery of the Process Data-flow (Decision-Tree Miner)* é um exemplo de *plugin* que pertence a essa categoria [11].

2.2 MAPE-K

Em 2011 a IBM propôs uma nova arquitetura de software conhecida como *Monitoring, Analysis, Planning, Execution and Knowledge (MAPE-K)* [12]. Essa arquitetura foi desenvolvida para diminuir as dificuldades presentes no monitoramento, principalmente em tempo de execução de sistemas, que a cada ano se tornavam mais heterogêneos e distribuídos. A solução encontrada foi chamada de *autonomic computing*, são sistemas que conseguem automonitorar seu comportamento e que implementam essa arquitetura [13]. A arquitetura é composta de 4 módulos: Monitoramento, Análise, Planejamento e Execução. Todos os módulos possuem conhecimento sobre o comportamento do sistema e quais são as suas metas (*goals*), como mostrado na Figura 2.1 [12].

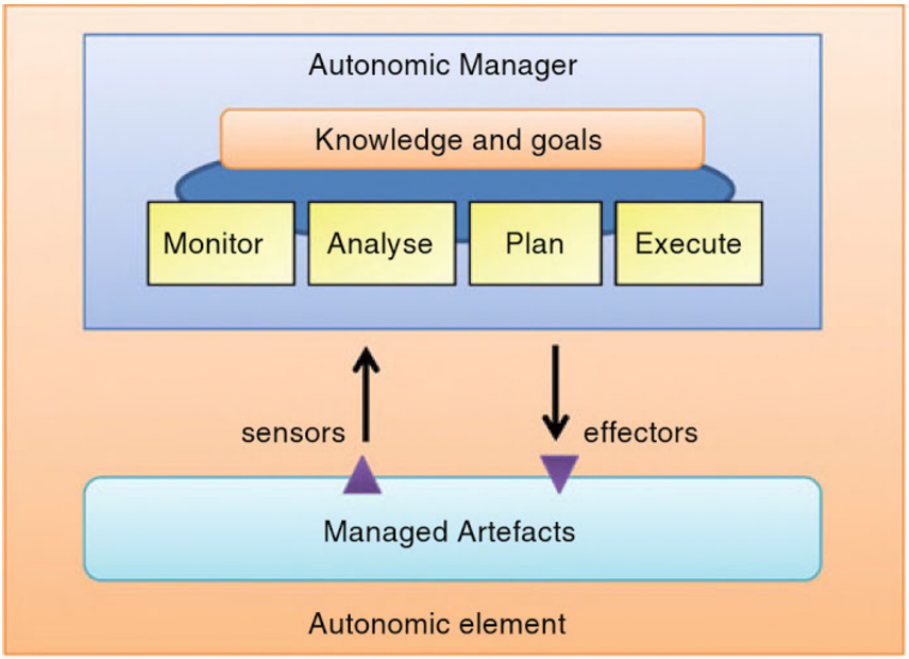

Figura 2.1: A arquitetura de referência da IBM.
Fonte: LALANDA; MCCANN; DIACONESCU (2013)

Na Figura 2.1 existem diversos *sensors*, que são responsável por coletar informações do sistema. O módulo de Monitoramento, como o próprio nome sugere, vai receber essas informações de tempos em tempos, determinados pelo usuário e gerar um log com as informações recebidas.

O módulo de Análise vai receber o log gerado no monitoramento. A partir disso, é feita uma análise das informações para verificar se o comportamento do sistema está de acordo com as

propriedades previamente definidas pelo usuário. Se em algum momento o analisador encontrou alguma violação das propriedades, isso indica que o sistema não está funcionando como deveria.

No módulo de Planejamento são definidas quais modificações devem ser feitas no sistema para ele voltar a funcionar com a qualidade esperada pelo usuário. Depois de definidas como serão feitas as modificações, o módulo de Execução é responsável por realizá-las através de componentes chamado *effectors*. Esse ciclo se repete durante todo o tempo de vida do software.

Se um sistema de middleware for implementado seguindo a arquitetura MAPE-K, o módulo de análise vai ser responsável por responder **quando** adaptar e o módulo de planejamento é responsável por responder **como** vai ser feita a adaptação [5].

2.3 Lógica Temporal Linear

Todo sistema quando executado possui um conjunto de estados e é importante checar se essas configurações estão corretas, pois somente verificar a saída fornecida por um sistema não garante que o seu comportamento está de acordo com o que foi planejado. Pode haver, por exemplo, um estado de *deadlock* que não ocorreu durante os testes, mas ele está presente na execução. Para isso, existe a Lógica Temporal Linear (LTL), que é um formalismo lógico para especificar propriedades temporais. Essa lógica é bastante utilizada em checagem e especificações de modelos de sistemas.

Lógica temporal é uma extensão da lógica proposicional, capaz de matematicamente representar estados de um sistema. A sintaxe é composta por proposições atômicas (verdadeiro ou falso), conectores booleanos ("e", "ou", negação, "se e somente se", "se então" e outros) e operadores temporais modais ("eventualmente", "sempre", "até" e outros) [14].

Para entender melhor como a Lógica temporal funciona, imaginemos que exista um sistema que possui diversos métodos, dentre eles um método *a* que deve sempre ser executado. Suponha que o log da Figura 2.2 é um log coletado durante a execução do sistema e que desejamos verificar se a propriedade LTL " <> *a*" é verdadeira. Essa propriedade significa "eventualmente a atividade *a* acontece" e ela vai ser verdadeira nesse log porque a quarta atividade foi *a*. Caso *a* não aparecesse em nenhum momento, ela seria falsa, e se aparecesse em mais de um momento, ela ainda continuaria verdadeira. Se o retorno da propriedade for falso, significa que o sistema se comportou de uma maneira não esperada .

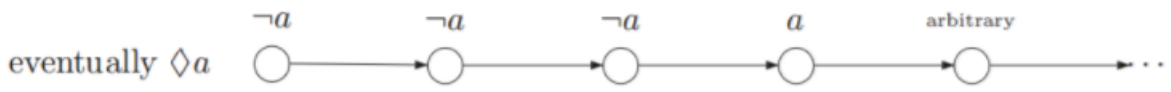

Figura 2.2: Comportamento do sistema por um período de tempo.
Fonte: BAIER; KATOEN (2008)

3
Solução

Neste capítulo será abordado como foi desenvolvida a solução, sua arquitetura e o fluxo de funcionamento. Como explicado no capítulo anterior, a ferramenta ProM possui diversos *plugins* que executam algum algoritmo de mineração. Neste projeto, utilizamos o *plugin* LTL-*checker*, que verifica a porcentagem de *traces* de um log em que uma dada propriedade LTL é verdadeira. Como o ProM foi desenvolvido em Java, esse projeto também utilizou a mesma linguagem.

Para desenvolvimento desta solução utilizamos um middleware desenvolvido em Java e um sistema que já conta com o módulo de monitoramento. Esse módulo é capaz de gerar logs dos eventos do middleware com as seguintes informações: nome da atividade relacionada, quem realizou a atividade, tipo de evento e o *timestamp*. Os logs gerado por esse módulo é utilizado na nossa nossa solução que é responsável por analisá-los, utilizando o ProM, e o resultado dessa análise é passado para o módulo de planejamento.

3.1 Requisitos

A solução que fornecemos nesse projeto é uma possível forma de responder a pergunta de quando se torna necessária a adaptação, ela vai ser o módulo de análise da arquitetura MAPE-K. Para isso, essa solução deve ser capaz de converter os logs, que são fornecidos através do módulo de monitoramento, para um formato que a ferramenta de análise aceite. É também necessário que esse módulo consiga se comunicar com o ProM, que é o responsável por fazer as análises dos logs. Além disso, essa solução precisa ser rápida, pois como o objetivo é melhorar um middleware em tempo de execução não é aceitável uma solução que leve muito tempo para executar.

A saída da análise realizada por este módulo deve seguir para o módulo de planejamento e caso seja necessário uma modificação deve seguir para o de execução. Na Figura 3.1 temos uma visão geral da arquitetura.

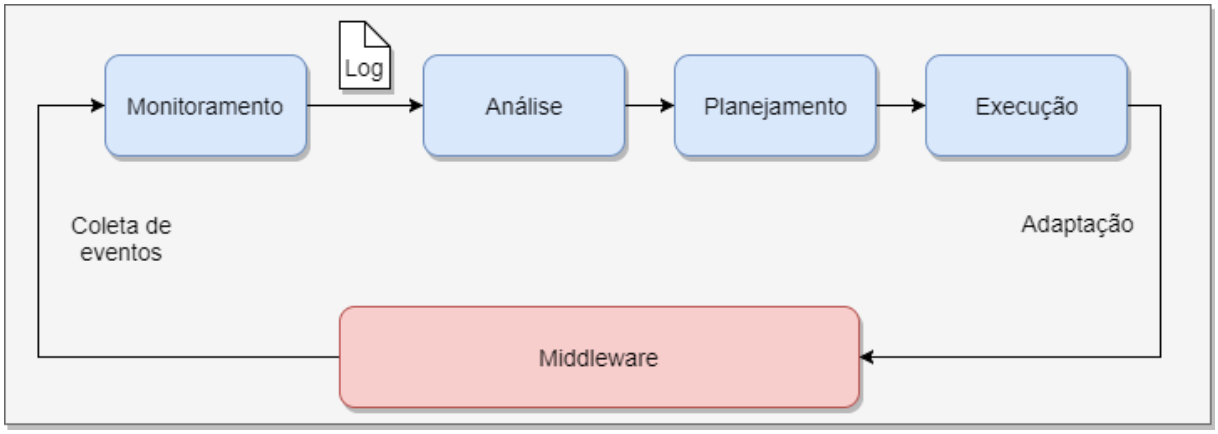

Figura 3.1: Arquitetura geral

3.2 Módulo de Análise

Esse projeto se encontra no módulo de análise da arquitetura. Além de receber como entrada o log gerado na fase de monitoramento, o usuário da solução também deve definir quais as propriedades LTL que ele deseja checar. A saída fornecida na análise vai ser as porcentagens de *traces* que satisfazem as propriedades e o módulo de planejamento vai receber essas informações. Na Figura 3.2 temos uma visão interna da arquitetura do módulo de análise e sua interação com o ProM.

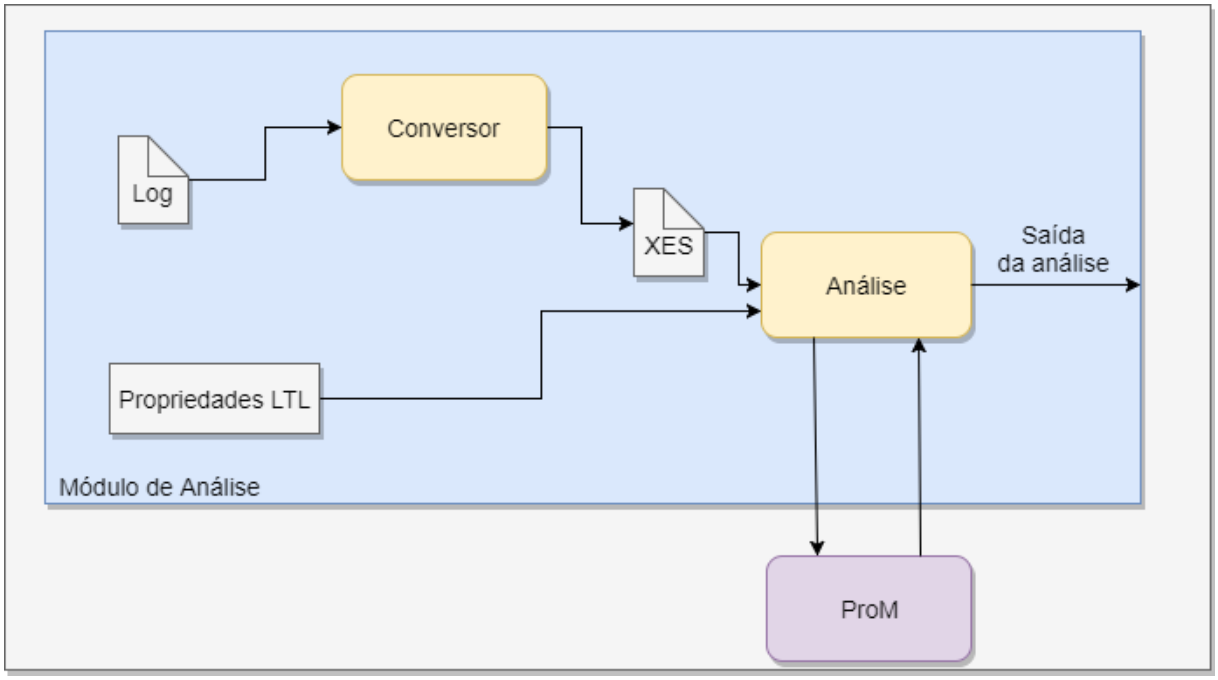

Figura 3.2: A arquitetura do módulo de análise e sua comunicação com o ProM.

Para entender melhor como funciona esse módulo, podemos separar a explicação em duas partes. Na primeira parte, temos a conversão do log para o formato do arquivo de entrada, que vai ser encaminhado para a ferramenta de mineração de processos, no nosso caso o ProM. Depois

que o arquivo está pronto vem a segunda parte, onde será feita a verificação das propriedades, previamente definidas pelo usuário. A saída fornecida nessa etapa deve ser repassada para o módulo de planejamento do middleware.

3.2.1 Conversor

Um middleware adaptativo deve gravar, idealmente, o início e o fim de todas as ações realizadas pelo middleware. Os logs gerados pelo módulo de monitoramento devem conter esses registros. Nossa solução assume que os logs gerados são expostos como um texto, onde cada linha corresponde a um evento e tenha informações como: nome da atividade, tipo de evento e timestamp. É preciso então converter o log fornecido para o formato que o ProM entenda.

O ProM aceita como entrada somente arquivos nos formatos *Magic eXtensible Markup Language* (MXML), *Comma-Separated Values* (CSV) e *eXtensible Event Stream* (XES). Como o XES é um padrão de arquivo criado especificamente para mineração de processos decidimos utilizar esse formato.

Na Figura 3.3 temos um trecho de um log fornecido pelo modo de monitoramento de um middleware. Em cada linha temos informações sobre um evento separadas por ponto e vírgula. Essas informações incluem quem executou a atividade, o nome da atividade, o tipo de evento e o *timestamp*.

Para converter o log para o formato XES são necessárias mais duas informações: saber qual o nome da atividade que inicia um *trace* e quem a executa. Essas informações devem ser fornecidas pelo usuário para que o conversor seja capaz de identificar quando um *trace* começa e termina. Importante destacar que não é garantido que o arquivo esteja em ordem de execução, pois muitos desenvolvedores utilizam métodos assíncronos para gravar os eventos no log. Então, foi necessário ordenar as informações, fornecidas pelo log, antes de criar o arquivo XES final.

Na Figura 3.4 temos um trecho de um um arquivo XES que foi gerado pelo conversor utilizando o log da Figura 3.3. O nome da atividade que inicia o *trace* é *receive* e quem a executa é o *ServerRequestHandler*.

```
ServerRequestHandler;receive;start;2016-05-12T15:11:16.850-0300
NamingProxy;bind;start;2016-05-12T15:11:24.998-0300
Requestor;invoke;start;2016-05-12T15:11:25.004-0300
Marshaller;marshall;start;2016-05-12T15:11:25.008-0300
Marshaller;marshall;complete;2016-05-12T15:11:25.026-0300
```

Figura 3.3: Trecho de um log antes da conversão para XES.

```
<trace>
    <event>
        <string key="resource:value" value="ServerRequestHandler"/>
        <string key="concept:name" value="ServerRequestHandler_receive_start"/>
        <string key="lifecycle:transition" value="start"/>
        <date key="time:timestamp" value="2016-05-12T15:11:16.850-0300"/>
    </event>
    <event>
        <string key="resource:value" value="NamingProxy"/>
        <string key="concept:name" value="NamingProxy_bind_start"/>
        <string key="lifecycle:transition" value="start"/>
        <date key="time:timestamp" value="2016-05-12T15:11:24.998-0300"/>
    </event>
    <event>
        <string key="resource:value" value="Requestor"/>
        <string key="concept:name" value="Requestor_invoke_start"/>
        <string key="lifecycle:transition" value="start"/>
        <date key="time:timestamp" value="2016-05-12T15:11:25.004-0300"/>
    </event>
```

Figura 3.4: Trecho de um log depois da conversão para XES.

3.2.2 Chamada do ProM

Depois de gerar o log no formato XES, temos que seguir para a próxima etapa que é responsável por checar se as propriedades LTL, escolhidas pelo usuário, são satisfeitas . O usuário precisa informar quais as propriedades LTL e os nomes das atividades que serão verificadas.

Nesta segunda etapa, as propriedades e os parâmetros escolhidos pelo usuário são passados para um método responsável por gerar as fórmulas LTL no formato que a ferramenta de mineração compreende. Se o usuário utilizar, por exemplo, a propriedade "eventualmente acontece a atividade A" mais a propriedade de que "a atividade inicial é A", passando como parâmetro a atividade *ServerRequestHandler_receive_start*, a consulta gerada que vai ser passada para o ProM possui a seguinte fórmula: `" <>(?act{ServerRequestHandler_receive_start})";"(?act1{ServerRequestHandler_receive_start})"`

É possível perceber que os nomes das atividades no arquivo XES da Figura 3.4 estão diferentes daquelas da Figura 3.3 que foram geradas pelo módulo de monitoramento. A ferramenta ProM quando vai checar uma propriedade, ela verifica apenas se existe alguma atividade em cada *trace* com o nome especificado, e o tipo do evento (*start* ou *complete*) não é analisado. Portanto essas modificações nos nomes foram feitas para ajudar a verificar se as atividades de um log começam (*start*) e terminam (*complete*).

Se um usuário quer checar a propriedade "sempre acontece a atividade *receive* do *ServerRequestHandler* ". Ao invés de utilizar a propriedade LTL "eventualmente acontece a atividade A", utiliza-se a propriedade que checa se "eventualmente acontece a atividade A e então a atividade B", e como parâmetro, o nome da atividade A vai ser *ServerRequestHandler_receive_start* e da atividade B será *ServerRequestHandler_receive_complete*. Dessa forma teremos o retorno relativo à quantas vezes a atividade foi inicializada e terminada.

Existe uma quantidade muito grande de propriedades LTL, porém algumas não fazem

sentido para o nosso contexto, por exemplo, a propriedade que checa se existe um estado antes de um determinado tempo. Atualmente, nossa solução aceita 10 propriedades LTL listadas na Tabela A.1.

É possível aumentar a quantidade de propriedades aceitas pela nossa ferramenta de forma bem simples. Existe uma classe Java no projeto onde cada método dessa classe é responsável por retornar uma propriedade LTL específica. Esses métodos recebem como parâmetros os nomes das atividades e o retorno deles é uma *String* que adiciona esses parâmetros a fórmula LTL. Lembrando que a propriedade deve obedecer a gramática da lógica temporal linear utilizada no ProM [15].

No código abaixo temos o exemplo de dois métodos que retornam propriedades LTL. A primeira é a propriedade "eventualmente a atividade A acontece" e a segunda propriedade é "sempre que atividade A acontece então depois acontece a atividade B, não precisa ser imediatamente depois".

```
     public String eventually_activity_A(String inputA){
return "\" <>(?act{"+ inputA+ "})\"";
}
     public String always_when_A_then_eventually_B
(String inputA,String inputB){
return "\"[](( (?act1{"+ inputA+ "}) -> <>(?act2{"+inputB+"})))\"";
}
```

No ProM existe um *plugin* chamado LTL-*checker*. Esse *plugin* extrai o log de um modelo e recebe um arquivo que contém propriedades LTL. O log será analisado para saber se o sistema possui essas propriedades. O retorno da análise é a porcentagem da quantidade de *traces* do processo que satisfazem cada fórmula que foi passada para a ferramenta.

Para entender melhor o LTL-*Checker* temos o seguinte exemplo: Seja um log com 9 *traces* e a propriedade LTL " <>*Order Delivery*" que equivale a "eventualmente *Order Delivery*", ou seja, se nos *traces* em algum momento aconteceu a atividade de nome *Order Delivery*. Se nesse log apenas em 9 dos 13 *traces* a propriedade é verdadeira, o *plugin* retorna 0,69, isto é, 69% dos *traces* contém essa propriedade.

Nesta etapa, responsável por checar as propriedades LTL no log gerado, utilizamos uma aplicação chamada de LTL*Miner*[1]. Essa aplicação foi desenvolvida em Java e foi criada com objetivo de gerar e descobrir fórmulas LTL baseadas em um template de entrada. O *LTLMiner* modificou o *plugin* LTL-*checker* para funcionar como uma biblioteca *stand alone* [16].

Assim com o LTL*Miner*, conseguimos obter as porcentagens de *traces* que satisfazem as propriedades escolhidas pelo usuário. Foi preciso modificar o tipo do retorno do LTL*Miner* para retornar um vetor de inteiros onde cada posição é equivalente ao retorno de uma propriedade. Além disso, foi preciso tratar também casos em que a ferramenta retorna um valor vazio para uma determinada propriedade. Isso acontece caso alguma atividade de entrada na propriedade

[1] https://github.com/TKasekamp/LTLMiner

não exista no log, neste caso o valor de retorno passou a ser 0, pois nenhum trace do log obedece a essa propriedade. Por fim, o vetor de saída deve ser passado para o módulo de planejamento da arquitetura.

4
Avaliação

No capítulo anterior foi explicado como funciona a solução proposta. Neste capítulo queremos entender melhor como ela pode impactar no tempo de execução do middleware.

4.1 Experimentos

A métrica que escolhemos estudar foi do tempo que a solução proposta neste projeto demora para realizar sua computação. Esse métrica é chamada de *tempo de resposta* e é composta pelo tempo de converter um log para o formato XES, criar a consulta para a ferramenta de mineração e o tempo de computação da ferramenta.

Para entender melhor como o tempo da solução pode variar, escolhemos estudar alguns fatores para saber como eles afetam a métrica. Os fatores escolhidos foram: o tamanho do log, as propriedades LTL escolhidas pelo usuário e a quantidade de propriedades que vão ser checadas.

Todos os experimentos foram realizados no mesmo equipamento, um notebook com Windows 10, 8 GB de RAM, 64 bit e processador Intel Core i7.

4.1.1 Tamanho do log

Neste experimento, queremos testar o quanto o fator *tamanho do log* influencia no tempo de resposta da solução. Para isso, utilizamos três diferentes logs, o primeiro tinha apenas 644 *traces*, o segundo 1383 *traces* e o último tinha 2764 *traces*. Além disso, para realizar este teste utilizamos a propriedade *eventually_activity_A* que significa "eventualmente uma atividade A acontece".

Foram feitas 2000 execuções. A média do tempo de execução para cada propriedade é mostrada na Figura 4.1. O desvio padrão para o primeiro log foi de 13,6 ms, para o segundo foi de 60,02 ms e para o último foi de 123,51 ms.

É possível perceber que a quantidade de *traces* interfere muito no tempo de resposta da solução. Portanto, é importante o usuário que for utilizar a solução ter o cuidado de no módulo de monitoração do middleware escolher um tamanho ideal de log para passar para o módulo de

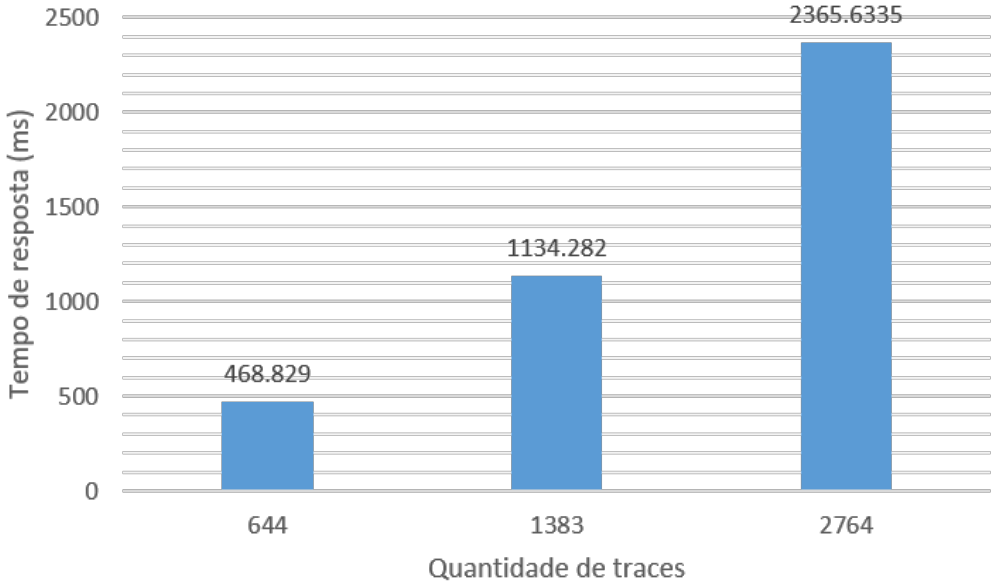

Figura 4.1: Tempo de resposta para cada log.

análise. Pois, dependendo do contexto do middleware, dois segundos apenas nesse módulo é bastante tempo.

4.1.2 Propriedades LTL

Neste experimento, queremos saber se o tempo gasto para receber uma resposta do módulo de análise depende de qual propriedade LTL o usuário escolheu checar. Escolhemos 3 diferentes propriedades para esse experimento. As propriedades escolhidas foram: *eventually_activity_A, eventually_activity_A_and_eventually_activity_B* e *eventually_activity_A_then_B_then_C*.

A três propriedades foram escolhida porque suspeitávamos que o tempo necessário de processamento por parte da ferramenta de mineração seria diferentes para cada uma delas. A primeira propriedade diz que "eventualmente uma atividade A acontece", então para checar se isso acontece é preciso encontrar o primeiro caso em que a atividade aparece em todos os *traces* do log. A segunda propriedade diz que "as atividades A e B acontecem, não necessariamente uma seguida da outra", nessa propriedade é preciso checar se existe o nome de duas atividades em cada trace. Por fim, a última propriedade diz que "a atividade A acontece, depois acontece a atividade B e depois acontece a atividade C, não necessariamente uma seguida da outra", então é necessário checar se existem as três atividades em todos os *traces* de um log.

O log utilizado nesse experimento contém 1383 *traces* e foram feitas 2000 execuções. A média do tempo das execuções para cada propriedade é mostrada na Figura 4.2. O desvio padrão para a primeira propriedade foi pequeno, de apenas 60,02 ms, já o da segunda propriedade foi um pouco maior, mas ainda continuou pequeno, 65,40 ms, e a terceira propriedade obteve o menor desvio padrão de 53,17 ms.

Como suspeitávamos o tempo de execução foi menor para a propriedade um e maior para a propriedade três. Contudo, mesmo essa terceira propriedade sendo a mais lenta das três testadas, foram necessário em média, apenas, 1,20 segundos para analisar todo o log que contém mais de mil *traces*.

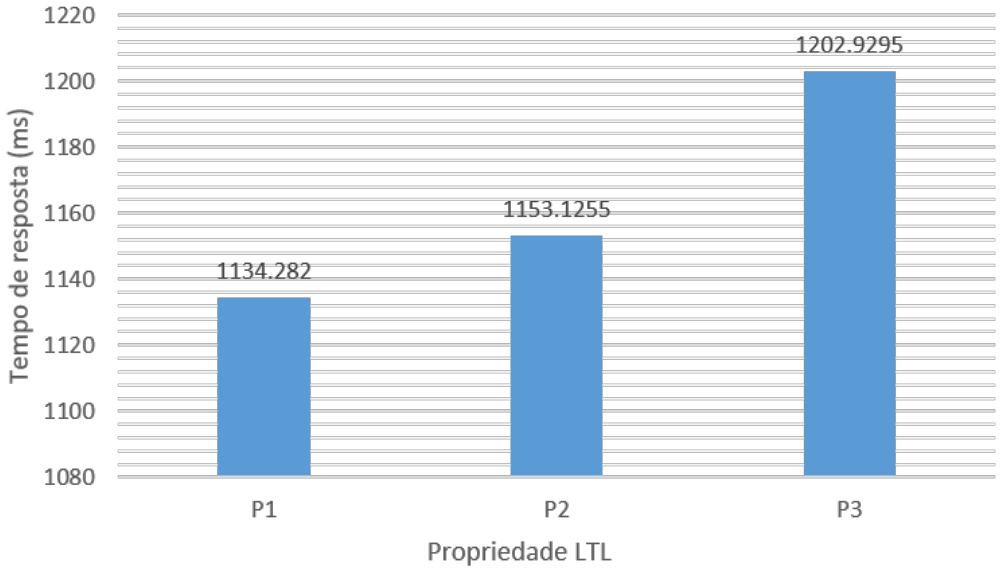

Figura 4.2: Tempo de resposta para cada propriedade.

4.1.3 Quantidade de Propriedades LTL

O último experimento realizado foi para identificar o quanto influencia o usuário pedir para analisar mais de uma propriedade LTL em um mesmo log. Para este experimento, utilizamos um log com 1383 *traces* e foram feitas 2000 execuções. Para o primeiro caso, escolhemos passar apenas uma propriedade a *eventually_activity_A*, no segundo caso passamos duas propriedades a *eventually_activity_A e eventually_activity_A_and_eventually_activity_B* e, no último caso, passamos três propriedades *eventually_activity_A* , *eventually_activity_A_and_eventually_activity_B* e *eventually_activity_A_then_B_then_C*.

A média do tempo das execuções para cada caso é mostrada na Figura 4.3. O desvio padrão para cada caso foi respectivamente 60,02 ms, 50,20 ms e 47,48 ms. A diferença de tempo para o caso dois e para o caso três foi muito pequena. Esse resultado não era o esperado, visto que o caso três está analisando três propriedades.

Na lógica LTL é possível obter uma propriedade através de combinações de outras propriedades. Observando o tempo de execução de P3 da Figura 4.2 e do caso 2 da Figura 4.3 temos que os tempos de execuções e a quantidade de propriedades analisadas foram diferentes. Sabemos que a propriedade P3 da Figura 4.2 é *eventually_activity_A_then_B_then_C*, é possível obter a mesma fórmula analisando duas propriedades a *eventually_activity_A_then_activity_B* e a *eventually_activity_B_then_activity_C*. Portanto o usuário deve ter cuidado ao escolher a

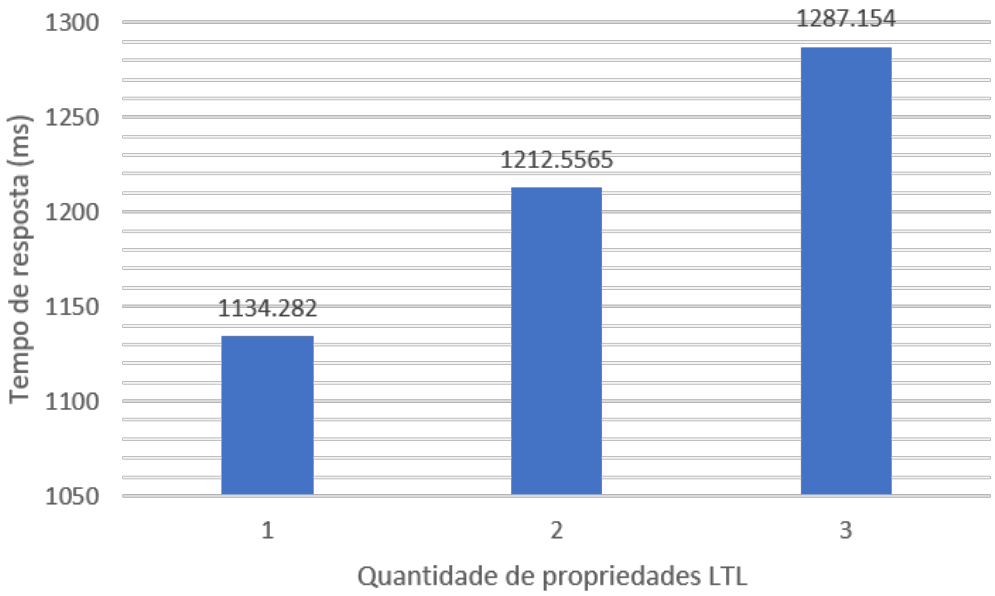

Figura 4.3: Tempo de resposta por quantidade de propriedades LTL.

propriedade LTL que ele precisa analisar, pois em alguns casos talvez seja mais vantajoso unir duas propriedades mais simples do que utilizar uma mais complexa.

4.2 Conclusão dos experimentos

Os resultados obtidos nos experimentos mostraram que o tempo gasto pelo módulo de análise depende de todos os três fatores analisados, principalmente o tamanho do log. As médias dos tempos de respostas obtidos nos experimentos ficaram entre 1 e 2 segundos, esses são valores altos para sistemas que são adaptados em tempo de execução, contudo, antes de descartar essa solução é importante levar em conta o contexto que o middleware se encontra e qual vai ser o *overhead* total que o sistema de adaptação vai acrescentar no tempo total de execução do middleware. A utilização da adaptação pode se tornar uma fração pequena do tempo total da execução do middleware, como acontece no MidARCH onde o tempo total gasto pela adaptação representa apenas 0,598% do tempo total [1].

5

Conclusão

Neste trabalho foi apresentado como se utilizar técnicas de mineração de processos em um middleware adaptativo, com objetivo de identificar quando é necessária uma modificação no sistema. Para entender a solução, foi explicado que o desenvolvimento de sistemas de middleware adaptativos têm uma importância muito grande para a computação, visto que, cada vez mais as aplicações se encontram distribuídas e nos mais variados tipos de dispositivos. Vimos também que é um tema bastante explorado, porém a utilização de técnicas de mineração de processos para esse tipo de situação ainda é uma ideia nova.

Ademais, apresentamos uma explicação sobre a ferramenta utilizada para fazer mineração de processos e a arquitetura que um middleware adaptativo deve seguir para utilizar a solução proposta. Além disso, foram realizados experimentos para entender melhor como esta solução pode impactar no tempo de execução do middleware.

Os experimentos mostraram que o *overhead* de utilizar a solução proposta para implementação de middleware adaptativos depende de muitos fatores como o contexto do middleware e tamanho do log. O usuário deve definir parâmetros que podem pesar nesse tempo, como o tamanho do log que vai ser passado para módulo de análise, as quantidades e tipos de propriedades que vão ser analisadas pela ferramenta de mineração.

5.1 Sugestões para trabalhos futuros

Como trabalho futuro, seria interessante implementar o módulo de planejamento que deve receber como entrada o vetor que o módulo de análise retorna com os valores referente a cobertura de cada propriedade em um log. Para isso, é preciso fazer um estudo para entender qual valor mínimo que deve ser retornado pela ferramenta de mineração, ou seja, a porcentagem de traces que devem satisfazer a uma determinada propriedade. Como sabemos qual é a propriedade que não está sendo satisfeita torna-se fácil identificar o que precisa ser adaptado.

Referências

[1] Nelson Rosa and David Cavalcanti. Building adaptive middleware using software architecture as enabling technology. Technical report, Universidade Federal de Pernambuco,Centro de Informática, 2017.

[2] Richard E. Schantz and Douglas C. Schmidt. *Middleware for Distributed Systems*. John Wiley Sons, Inc., 2007.

[3] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[4] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[5] N. Rosa. Middleware adaptation through process mining. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 244–251, March 2017.

[6] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, June 2002.

[7] Luiz Gustavo Couri Nogara. *Um Estudo Sobre Middlewares Adaptáveis*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2007.

[8] M. Völter, M. Kircher, and U. Zdun. *Remoting patterns: foundations of enterprise, Internet and realtime distributed object middleware*. Wiley series in software design patterns. John Wiley, 2005.

[9] W. M. P. van der Aalst, V. Rubin, H. M. W. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87, 2008.

[10] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. *The ProM Framework: A New Era in Process Mining Tool Support*, pages 444–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[11] H.M.W. Eric Verbeek and R. P. Jagadeesh Chandra Bose. *ProM 6 Tutorial*. 2010.

[12] P. Lalanda, J.A. McCann, and A. Diaconescu. *Autonomic Computing: Principles, Design and Implementation*. Undergraduate Topics in Computer Science. Springer London, 2013.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing, Jan 2003.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[15] H. de Beer. *The LTL Checker Plugins: A Reference Manual*. Eindhoven University of Technology, Eindhoven, 2004.

[16] Tõnis Kasekamp. *Discovering LTL-Based Business Rules From Event Logs*. PhD thesis, UNIVERSITY OF TARTU, 2015.

Apêndice

A

Apêndice

Tabela A.1: Listas de propriedades implementadas.

Propriedade	**Significado**
eventually_activity_A	Eventualmente acontece a atividade **A**.
always_when_A_then_eventually_B	Sempre que a atividade **A** acontece depois acontece a atividade B, não precisa ser imediatamente depois.
eventually_activity_A_and_eventually_activity_B	A atividade **A** e a atividade **B** acontecem, em qualquer ordem.
eventually_activity_A_next_activity_B	A atividade **A** acontece e imediatamente depois acontece a atividade **B**.
eventually_activity_A_then_activity_B	A atividade **A** acontece e depois acontece a atividade **B**, não precisa ser imediatamente depois.
eventually_activity_A_next_activity_B_next_activity_C	A atividade **A** acontece, depois acontece a atividade **B** e depois acontece a atividade **C**, precisa ser uma seguida da outra.
eventually_activity_A_then_B_then_C	A atividade **A** acontece, depois acontece a atividade **B** e depois acontece a atividade **C**, não precisa ser imediatamente depois.
eventually_activity_A_or_eventually_B	Acontece a atividade **A** ou a atividade **B**.
is_activity_of_first_state_A	A atividade inicial é **A**.
Precedence(A,B)	A atividade **B** só vai ser executado se a atividade **A** for executado antes, não precisa ser **B** logo depois de **A**.