UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

NELSON ANTONIO ANTUNES JUNIOR

Permitindo Maior Replicabilidade de Experimentos em Ambientes Distribuídos com Nodos de Baixa Confiabilidade

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Weverton Cordeiro
Co-orientador: Prof. Dr. Luciano Paschoal Gaspary

Porto Alegre
Julho de 2017

“I love deadlines.
I love the whooshing noise they make as they go by.”

— DOUGLAS ADAMS, The Salmon of Doubt

AGRADECIMENTOS

Gostaria de agradecer a ambos, meu orientador, Prof. Weverton Cordeiro e co-orientador, Prof. Luciano Gaspary, por me darem a oportunidade de estudar e desenvolver em cima de uma área que tem sido a de meu maior interesse, sistemas distribuídos. Desculpem-me por qualquer inconveniência causada pela minha personalidade orgulhosa, costumo me manter fechado quando os resultados de meus esforços estão abaixo do que eu esperava.

Obrigado aos meus pais, Maria Medianeira e Nelson Antunes, por se preocuparem comigo e estarem sempre dispostos a ajudar. Obrigado por tudo que fizeram por mim, principalmente por cuidarem de mim com tanto carinho.

E obrigado à Caroline Morinel, por me dar apoio durante todo o processo de pesquisa e realização do projeto; por sempre querer o meu bem e ter dedicado tanto do seu tempo para me fazer feliz; por me motivar em momentos onde imaginei que não seria mais capaz de finalizar este trabalho.

RESUMO

A reprodução de experimentos representa uma das melhores formas de se comprovar a eficácia de propostas científicas nas mais diversas áreas do conhecimento. Na computação, a replicabilidade de experimentos é particularmente favorecida pela popularidade dos repositórios públicos, bem como pelo compartilhamento de código fonte e de dados de entrada de publicações científicas. As dificuldades da replicação de experimentos surgem em áreas com ambientes de execução imprevisíveis e voláteis. Máquinas falhando e problemas com comunicação de redes correspondem a esta descrição e são comumente encontrados em ambiente distribuídos. Para superar estes problemas, um sistema distribuído capaz de rodar experimentos em ambientes distribuídos de baixa confiabilidade sem intervenção manual será desenvolvido, onde computadores que falharem durante sua execução sejam substituídas por outras funcionais. Comparada a outras propostas, sua inovação é a capacidade de manter o contexto de máquinas falhas, similar a *checkpoints*, e permitir que seus substitutos recuperem os dados salvos. Essa funcionalidade deve aprimorar a replicabilidade do experimento desde que a disponibilidade do sistema também virtualmente crescerá. Como desafios, ainda é necessário desenvolver um sistema que permita a execução de experimentos distribuídos, para então ser possível aplicar o protocolo de recuperação. Resultados obtidos durante a avaliação do protótipo em um ambiente instável demonstram uma menor variação e maior precisão entre múltiplas tentativas de um mesmo experimento, com um desvio padrão de 1.6% da média e precisão de 95.7%, comparado com os resultados sem o uso do sistema, com um desvio de 25% da média e precisão de 72%.

Palavras-chave: Ambientes Distribuídos. Sistemas Distribuídos. PlanetLab. Experimentação. Replicabilidade.

ALLOWING BETTER EXPERIMENTS' REPLICABILITY ON DISTRIBUTED ENVIRONMENTS WITH LESS RELIABLE NODES

ABSTRACT

One of the best ways to prove the efficiency of scientific proposals on any field of knowledge is by the reproduction of experiments. The experiments' replicability is particularly favorable by the popularity of the public repositories within computing, as well by sharing source code and input data in scientific papers. The challenges of replicating experiments appears in fields with environments susceptibles to volatile and unpredictable executions. Machines crashing and the troubles of network communications match this description and are commonly found in distributed environments. To overcome this problems, a distributed system capable of running experiments in these less reliable systems without manual intervention will be developed, where the computers that fail during execution shall be swapped with the functional ones. Compared to other proposed solutions, its innovation is the capability to keep the context of failed machines, similar to checkpoints, and allow their substitute to recover the saved data. This feature shall enhance the experiments' replicability since the system's availability will also virtually increase. The challenge comes with the development of a distributed experiments' execution system from the ground up, so we will be able to apply our own recovering protocol. Results obtained during the evaluation of the prototype in an unstable environment show a smaller variation and greater precision between multiple attempts of the same experiment, with a standard deviation of 1.6% from the mean and accuracy of 95.7%, compared to the results without the use of the system, with a deviation of 25% from the mean and accuracy of 72%.

Keywords: Distributed Environment. Distributed Systems. PlanetLab. Experimentation. Replicability.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACM	Association for Computing Machinery
VIM	International Vocabulary Metrology
SSH	Secure Shell
SCP	Secure Copy
SFTP	SSH File Transfer Protocol
XML	Extensible Markup Language
RPC	Remote Procedure Call
DSL	Domain Specific Language
API	Application Programming Interface
IP	Internet Protocol
URL	Uniform Resource Locator
SQL	Structured Query Language
ACL	Access Control List
DNS	Domain Name Server
P2P	Peer-to-Peer

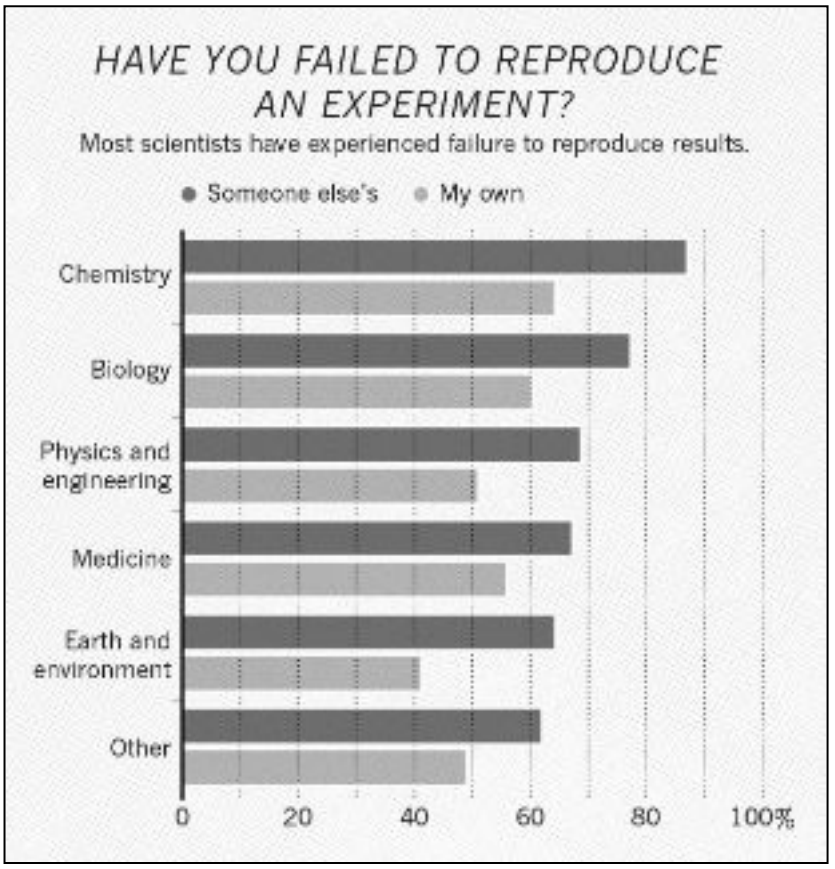
SUMÁRIO

1 INTRODUÇÃO

A reprodução de um experimento é a melhor forma que temos de atestar sua autenticidade, onde em termos gerais, replicá-lo seria sua reprodução feita por terceiros. Porém, a crise da replicação afetou a credibilidade de experimentos que pesquisadores, incluindo os próprios autores, não alcançaram resultados similares em tentativas posteriores à publicação original [BAKER 2016]. A Figura 1.1 mostra a proporção de falhas em tentativas de reprodução de experimentos por cientistas, onde os resultados foram separados por campo do conhecimento, e as cores indicam se foi um experimento próprio ou não. Em média, metade das reproduções próprias falharam, enquanto que dois terços das reproduções efetuada por terceiros falharam. Método aplicado, métricas utilizadas, condições de operação, dados de entrada e de saída estão entre os requisitos mínimos presentes em uma publicação para garantir que ela seja reproduzível. Contudo, caso haja algum erro humano por parte do autor, seja a própria intenção de adulteração ou seleção de resultados, ou até mesmo "sorte", uma sequência específica de eventos que passou despercebida, os mesmos resultados nunca serão alcançados em novas tentativas, e são publicações com estas características que mancharam a credibilidade de todos os outros experimentos.

Para indicar a credibilidade de um publicação, a ACM [2017] determinou que artigos podem ser avaliados de três formas quanto a sua validade, onde todas dependem da precisão (não-variação de resultados) entre múltiplas tentativas: **repetibilidade**, precisão medida pela equipe original no mesmo local de teste, com os mesmos procedimentos, métricas e condições de operação; **replicabilidade,** precisão medida por uma equipe diferente no mesmo ou em outro local, com os mesmos procedimentos, métricas e condições de operação; e **reprodutibilidade**, precisão medida por uma equipe diferente com procedimentos e métricas diferentes, normalmente desenvolvidos por esta equipe. Essas definições diferem da norma internacional *International Vocabulary Metrology* (VIM) na adição do termo intermediário entre repetibilidade e reprodutibilidade. Diferentemente das outras ciências, os pesquisadores da computação precisam apenas do código fonte e dos dados de entrada de um experimento para ser capaz de replicá-lo em seu próprio ambiente, são neles que os métodos e as métricas usadas estão definidas. O fácil compartilhamento destes artefatos é uma vantagem presente na ciência computacional e favorecida pela existência de repositórios públicos, onde podem ser divulgados a qualquer um que tenha interesse de utilizá-los e aprimorá-los.

Figura 1.1 - Gráfico contendo proporção de falhas em experimentos reproduzidos por pesquisadores de diferentes áreas



Fonte: BAKER, Monya. 2016.

1.1 Definição do Problema

Computação também é suscetível a falhas de replicação: versionamento de *software* modificam eficiência e compatibilidade entre códigos; concorrência e sincronização dependem de políticas de escalonamento e desempenho dos recursos utilizados; redes de computadores dependem de uma infraestrutura dinâmica e, em um ambiente real, fora do controle do utilizador, com a possibilidade de perda da conexão a qualquer momento. Estes são exemplos de desafios encontrados em um **ambiente distribuído** e que afetam o resultado entre diversas execuções das aplicações instanciadas neles, problemas que dificultam a replicabilidade de experimentos. Um ambiente distribuído é um conjunto de múltiplas máquinas, de um par a milhares, que estão a disposição do usuário, mas não possuem nenhuma funcionalidade em conjunto além do acesso a elas. Criar, manter e gerenciar uma arquitetura que pode variar de duas a centenas de máquinas não é uma tarefa simples, cada máquina presente em um neste ambiente é uma nova variável que afeta o conjunto. Pesquisadores não deveriam ter que lidar com estes contratempos infraestruturais quando este

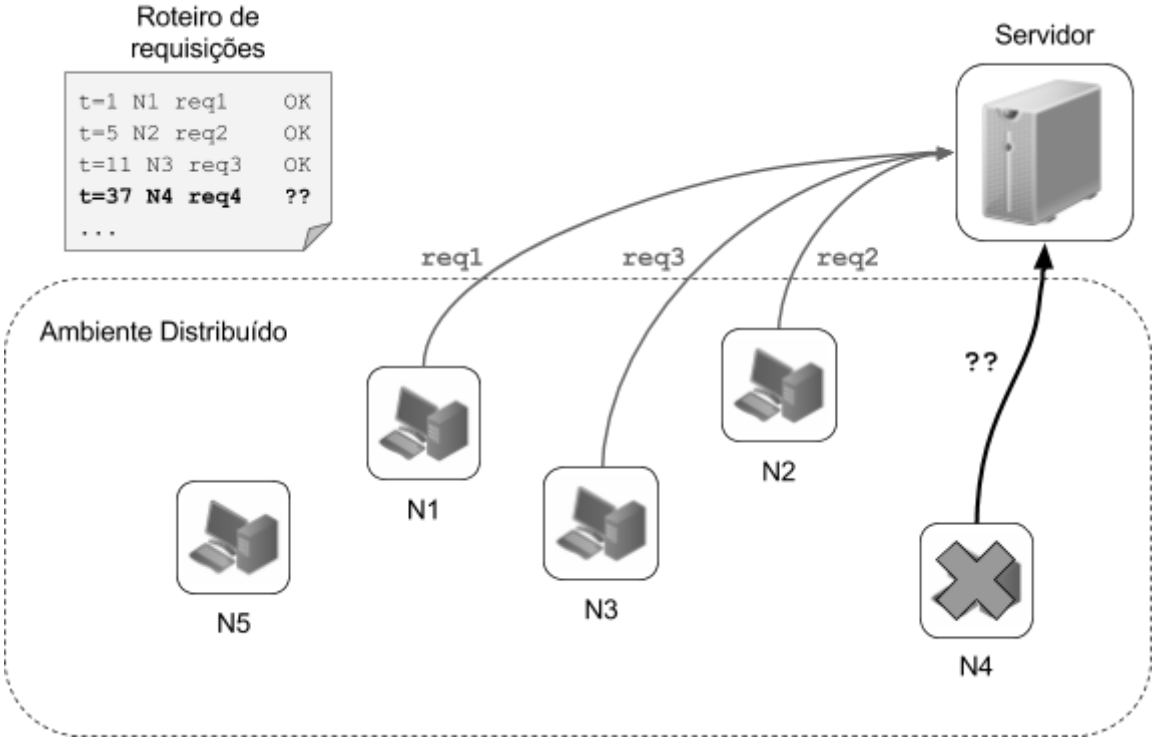
não é o foco de suas pesquisas, mas sim, apenas com o uso dos benefícios que este ambiente proporciona. Ambientes distribuídos permitem que um único experimento rode em múltiplas máquinas diferentes simultâneamente, onde sua carga de trabalho pode ser dividida entre todas. Este método, conhecido como computação distribuída, é uma das formas de se aprimorar o desempenho de um experimento, com computadores responsáveis por dividir as tarefas para outros e reunir os resultados no final de seu processamento. Outra caso de uso aos ambientes distribuídos é a simulação de sistemas baseados em redes, como avaliar a capacidade de um novo protocolo ou uma nova arquitetura com *n* servidor e *m* clientes realizando requisições dado um traço a ser seguido, para *n* e *m* números naturais quaisquer. A melhor forma de autenticar estas propostas é através de experimentação em um ambiente distribuído real. **Sistemas distribuídos** adaptam estes ambientes reais e fornecer local de trabalho preparado para o uso dos usuários com suas aplicações. De acordo com Tanenbaum [TANENBAUM et al. 2007], *"Um sistema distribuído é uma coleção de computadores independentes que aparecem ao seus usuários como um sistema único e coerente"*. Qualquer forma de computadores conectados via rede trabalhando em conjunto e usado como uma única entidade pode ser considerado um sistema distribuído. A partir dessa definição, podemos incluir sistemas como *grids* e *clusters,* bem conhecidas na área de computação de alto-desempenho por possuir sistemas com comunicação e processamento de alta velocidade e arquiteturas próprias para prover tais capacidades, ou infraestruturas *cloud*, onde usuários não precisam saber qual máquina foi reservada por determinado tempo, apenas seus recursos disponíveis e o meio de acessá-la.

Aplicações com longos períodos de atividade são as mais afetadas por ambientes instáveis, pois a probabilidade de ocorrer uma falha a durante a execução é um risco a correr. Em uma execução local, uma interrupção inesperada da máquina pode ser considerada como uma falha, pois não é mais possível contatá-la ou utilizá-la e qualquer execução em andamento não será finalizado. Em um sistema distribuído, a falha de uma das máquinas ocasiona o mesmo efeito nela, mas não tem uma consequência pré-definida no todo, e por isso é denominada como falha parcial [GUERRAOU et al. 2006]. Falhas parciais são impossíveis de definir sua causa apenas ao detectá-las: se foi o processo que gerencia a máquina que parou de funcionar, ou um problema de comunicação com tal máquina. Sistemas distribuídos precisam fornecer uma alta disponibilidade, para que não haja a interrupções durante sua utilização. Para isso, ele deve ser capaz de tratar todas as falhas que um ambiente distribuído é propenso a gerar, ou as aplicações instanciadas nele também estão suscetíveis às falhas.

Por questões de demonstração dos efeitos das falhas de ambientes distribuídos, suponhamos que um novo protocolo de comunicação entre computadores *NP* (Novo Protocolo) seja desenvolvida, um que seja diferente dos já existentes como TCP e UDP, e, com objetivo de autenticar a eficácia deste protocolo, devemos avaliar sua execução em um

ambiente distribuído real. Para isso, um sistema será desenvolvido, onde cada máquina *Ni* irá enviar uma sequência de requisições a um servidor usando este protocolo, onde *Ni* é o identificador da máquina com um número natural *i* único entre todos identificadores. Esse sistema permitirá analisar a vazão dos dados, tempo de resposta e outras variáveis desejadas pelo pesquisador quanto ao desempenho do protocolo. Para criar uma avaliação reproduzível, é necessário padronizar os dados de entrada, que nesse caso seria a sequência de requisições *reqi* que cada máquina *Ni* deve realizar ao servidor em um dado momento *t*, onde *reqi* é o identificador da requisição com um número natural *i* único entre todos identificadores. Iniciando a execução de todas as máquinas sincronizadamente, em um momento *t* entre 5 e 11 chegaremos na representação da Figura 1.1.1. Nela podemos ver a situação recém descrita, onde um roteiro de requisições está sendo seguido pelas máquinas do ambiente e enviando requisições ao servidor. Neste caso, as requisições *req1*, *req2* e *req3* foram enviadas pelas máquinas *N1*, *N2* e *N3*, respectivamente.

Figura 1.1.1 - Diagrama demonstrando o efeito de uma falha durante a execução de um experimento em um ambiente distribuído



Fonte: Elaborada pelo autor

Caso em algum momento entre as requisições *req1* e *req4* a máquina *N4* pare de funcionar por uma falha de sistema, a sua execução do experimento será interrompida, e quaisquer requisições futuras que tenha essa máquina como origem não serão realizadas. Na

Figura 1.1.1, pode-se ver esta situação, onde não há como enviar a *req4* ao servidor, pois sua máquina de origem está desativada devido à falha. Consequentemente, o resultado da avaliação não será fiel ao roteiro base, e execuções seguintes dessa mesma avaliação resultaram em saídas diferentes, sem contar com falhas que também podem ocorrer às outras máquinas nas novas tentativas. A avaliação descrita do protocolo *NP* é incapaz de alcançar um bom grau de repetibilidade e, consequentemente, replicabilidade. Como a pesquisa com ambientes distribuídos existe desde o início da década de 1980, propostas para auxiliar a experimentação no campo já foram publicadas.

1.2 Revisão do Estado da Arte

Gush (GENI *User Shell*) [ALBRECHT et al. 2010], anteriormente conhecido com *Plush* (*PlanetLab User Shell*) [ALBRECHT et al. 2007], é um *framework* para gerenciar aplicações em plataformas de ambientes distribuídos, com um alto nível de configuração via arquivos XML (*Extensible Markup Language*) ao definir sua aplicação e capaz de gerenciar as máquinas remotas como recursos a serem descobertos e alocados dinamicamente, sejam declaradas por credenciais para conexões SSH (*Secure Shell*) ou usando APIs (*Application Programming Interface*) de plataformas suportadas. Aplicações possuem um ou mais blocos de execução, onde comandos *shell* e métodos de sincronização distribuída da própria ferramenta podem ser montados sequencialmente pelo usuário, e diversas configurações desses blocos destinadas a múltiplas máquinas de um mesmo experimento. Arquivos para executar a aplicação devem ser fornecidos, pois ele será enviado a cada uma das máquinas alocadas. A perda de sinal de uma das máquinas remotas em tempo de execução podem gerar três reações: se nenhuma outra máquina está disponível para substituir a perdida, a aplicação é abortada; caso contrário, dependendo da configuração inicial do experimento, após a substituição a aplicação pode, ou ser reiniciada em todas suas instâncias do experimento, ou apenas ser iniciada na nova instância da máquina disponível para acompanhar as demais. O controlador, como é definido a máquina onde são executadas as operações de gerência do *framework*, não pode falhar, ou todo o sistema para de funcionar. É possível utilizar um controlador *backup*, recebendo as informações de forma replicada, mas depois que o original parar de funcionar, ele não pode voltar depois como um *backup* daquele que o substituiu, por que todas as informações recebidas durante sua indisponibilidade não são recuperáveis após sua reativação.

Splay [LEONINI et al. 2009] é uma *framework* desenvolvida para avaliar viabilidade de algoritmos distribuídos, onde serão executados em um ambiente controlado. Cada máquina remota do sistema possui um *daemon* que deve ser conectado ao controlador, que é a máquina que gerencia todas as operações. No controlador estão presentes diversos módulos

independentes, como base de dados e interfaces *Web* ou por linha de comando. O ambiente de execução é considerado controlado devido ao *daemon* da ferramenta, instanciado em cada máquina, que delimita seus recursos e fornece a biblioteca para utilizar apenas os recursos disponíveis, obrigando programação com a linguagem do sistema, *Lua*. A execução remota se baseia em RPC (*Remote Procedure Call*) através de XML, ou seja, não há transferência de arquivos envolvidos, apenas execução de algoritmos. Seguindo a premissa de que não haja um sistema *Splay* já instanciado, é necessário preparar ele por completo, instalando tanto o controlador quanto todas as máquinas remotas que irão rodar o *daemon*. Não há menções de tolerância a falhas na *framework*, apenas no caso onde caso desejasse tal funcionalidade em uma das aplicações de exemplo, ela foi adicionada no código da aplicação, e não fornecida pela ferramenta.

Dois projetos similares e que surgiram na mesma época e instituição são *EXPO* [RUIZ et al. 2013] e *EXECO* [IMBERT et al. 2013]. Modularidade de bibliotecas, suporte nativo à plataforma *Grid5000*, utilização da aplicação *Taktuk* são suas principais características. *Taktuk* [CLAUDEL et al. 2009], uma aplicação externa a estas duas recém citadas, permite que a camada de envio de experimentos seja extremamente eficiente, graças seus algoritmos de divisão de trabalho e paralelização em topologia de árvore, ao invés de estrela, na comunicação SSH entre máquinas. A diferença entre as ferramentas está na forma que experimentos são definidos e instanciados: *EXPO* adota um arquivo de configuração com sua DSL (*Domain Specific Language*, termo usado para linguagens criadas para resolver um problema em específico, neste caso, configuração); *EXECO* funciona através de uma biblioteca *Python*, sendo possível configurar máquinas destinos “programaticamente”, como dito na pesquisa. Falhas na experimentação serão apenas gravados no relatório final do experimento, não havendo uma política de recuperação delas. Suas execuções são delimitadas a apenas comandos SSH, sem a capacidade de envio de arquivos auxiliares.

Além de ferramentas, alternativas como algoritmos de seleção das máquinas menos propensas a falhas [COSTA et al. 2015] são propostas que melhoram a replicabilidade de experimentos e não estão limitados a ambientes específicos. Como vantagem dessa solução, ela pode ser integrada em outras propostas já existentes. Utilizando análise do tempo de resposta entre máquinas de um conjunto do ambiente distribuído, um grupo que não ultrapassa do limiar estabelecido é selecionado e formado seu grafo, contendo tais valores coletados. Diversas estratégias de seleção são aplicadas sobre esse grupo baseando-se em teoria de grafos, onde um provável subgrafo ótimo seria selecionado e avaliado quanto sua eficácia. Apesar dos resultados positivos, a execução meramente escolhe a melhor opção, mas não quer dizer que a melhor opção obtida não conterá falhas.

A lista de soluções continua com outras opções similares às citadas ou para nichos específicos de problemas na computação distribuída (*e.g. Neko, Weevil, Mace, NEPI, OMF,*

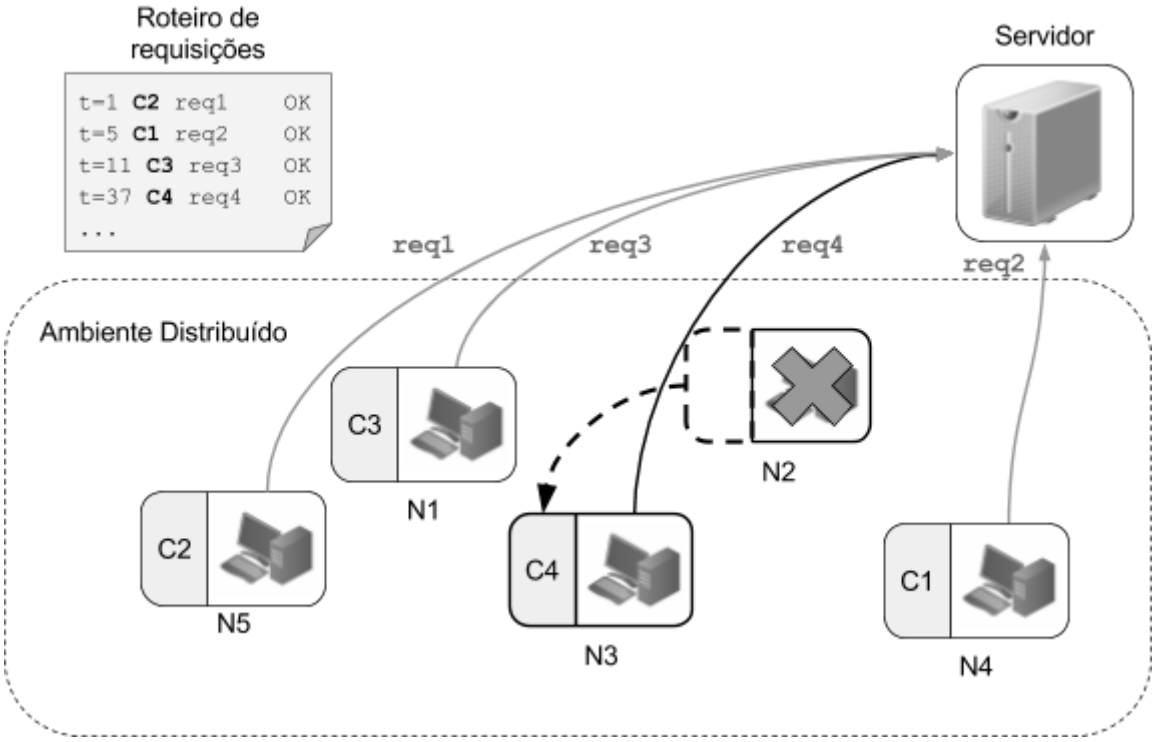
Fabric, Ansible, Rundeck, Dkron), mas **nenhuma** trata o problema citado na Seção 1.1. Apesar de *Gush* oferecer tolerância a falhas automatizada nas execuções de remotas, é incapaz de resolver o problema. A máquina perdida será reiniciada sem nenhuma informação adicional, logo, sua execução não estará mais sincronizada com as demais. Como resultado, todas as operações já realizadas pela máquina perdida serão repetidas, onde no problema descrito geraria uma inconsistência ainda maior nos resultados: o número final de requisições será maior que o esperado, pois elas serão duplicadas e acontecerão em momentos diferentes dos esperados.

1.3 Objetivo

O objetivo deste trabalho será de elaborar um sistema distribuído capaz de executar experimentos em ambientes de baixa confiabilidade sem intervenção manual, onde máquinas que pararem de funcionar durante a experimentação sejam substituídas por outras disponíveis. Como inovação, essa solução deve salvar o progresso de execução por máquina, para que, em caso de sua falha, uma nova seja definida como sua substituta com o mesmo progresso transcorrido que a máquina perdida. Esta proposta resolveria tanto o problema de descontinuidade demonstrado no exemplo da Seção 1.1 quanto ao problema das soluções já existentes, onde as requisições já realizadas não serão repetidas. Assim, o progresso de um experimento não estaria atrelado à máquina onde é executado, mas a uma representação da máquina. Essa representação pode ser chamada de **contexto**, e máquinas serão designadas para instanciar os contextos requisitados pelo experimento.

Será usado o mesmo exemplo da avaliação do protocolo *NP* da Seção 1.1 para demonstrar o efeito da proposta. Requisições serão relacionadas a um contexto *Ci*, onde *Ci* é o identificador do contexto com um número natural *i* único entre todos identificadores. Cada máquina deve ser capaz de identificar que contexto representa e executar suas requisições, como é demonstrado na Figura 1.3.1. Caso uma máquina do experimento falhe durante sua execução, o seu contexto deve ser instanciado em uma outra máquina disponível. Na Figura 1.3.1, podemos ver esse evento acontecer entre as máquinas *N2* e *N3*, onde o contexto *C4* agora é representado pela última citada. Como o progresso é baseado no contexto, a máquina *N3* será capaz de continuar recuperando as informações necessárias sobre o contexto *C4*, requisições posteriores, como a *req4*, não serão perdidas.

Figura 1.3.1 - Diagrama demonstrando o efeito de uma falha durante a execução de um experimento usando a proposta descrita em um ambiente distribuído



Fonte: Elaborada pelo autor

O restante deste trabalho está organizado como segue: O Capítulo 2 tratará da proposta de arquitetura para resolver o problema descrito neste Capítulo. O Capítulo 3 contém a descrição de tecnologias utilizadas e com elas implementam a arquitetura do Capítulo anterior e os resultados obtidos por tal implementação dada uma avaliação em um ambiente distribuído instável. Finalmente, o Capítulo 4 fecha o trabalho apresentando considerações finais, com discussões sobre limitações da arquitetura proposta, casos reais de estudo a serem testados com a solução e possíveis trabalhos futuros que poderiam melhorar o campo de experimentos que podem ser tratados pela proposta.

2 EXPEEKER - UMA ARQUITETURA PARA REPLICAR EXPERIMENTOS

A arquitetura desenvolvida é descrita, de forma breve, na Seção 2.1. Os principais desafios encontrados durante o seu desenvolvimento foram: (1) definir os requisitos do experimento que será executado pela arquitetura; (2) detectar falhas nos trabalhadores; (3) alocar trabalhadores para a execução de um experimento e sua inicialização; e (4) manter o progresso do contexto na nova máquina quando sua anterior falhar. Eles serão discutidos durante a Seção 2.2.

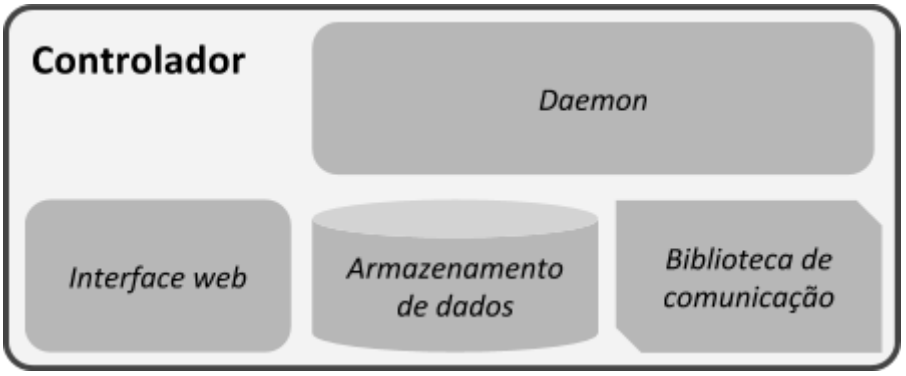
2.1 Visão Geral da Arquitetura

Uma arquitetura centralizada será a base da propostas, composta por duas categorias de máquinas: **controlador** e **trabalhador**. O controlador é o centro e reuni as funcionalidades principais de gerência e interação com a arquitetura. Ele distribui os experimentos através dos trabalhadores do sistema e os executa. O trabalhador é a máquina onde os experimentos são instanciados. Nas Subseções 2.1.1 e 2.1.2 serão descritos os componentes que fazem parte de cada uma destas máquinas, representadas pelas Figuras 2.1.1.1 e 2.1.2.1, respectivamente, o controlador e o trabalhador.

2.1.1 Controlador

O componente de **armazenamento de dados** se baseia em *data stores*, ou seja, independente da tecnologia usada (*e.g.* banco de dados ou sistemas de arquivos), um meio onde dados podem ser mantidos e recuperados quando necessários. Toda informação necessária para o gerenciamento do sistema será mantido neste módulo: dados de trabalhadores e experimentos. Deve ser acessível pelo *daemon*, pela interface *web* e pelos trabalhadores, mas cada um com acesso limitado ao necessário para seu funcionamento. Usuários interagem com o sistema através da **interface *web***, um serviço *web* acessível via navegadores. O acesso *web* simplifica a gerência realizada ao enviar e receber dados, sendo que o controlador pode estar instanciado em máquinas diferentes da estação de trabalho do usuário. Através deste módulo, novos trabalhadores e experimentos são adicionados ao armazenamento de dados, e o estado dos trabalhadores e as execuções de experimentos podem ser consultadas.

Figura 2.1.1.1 - Diagrama representando o controlador e seus componentes



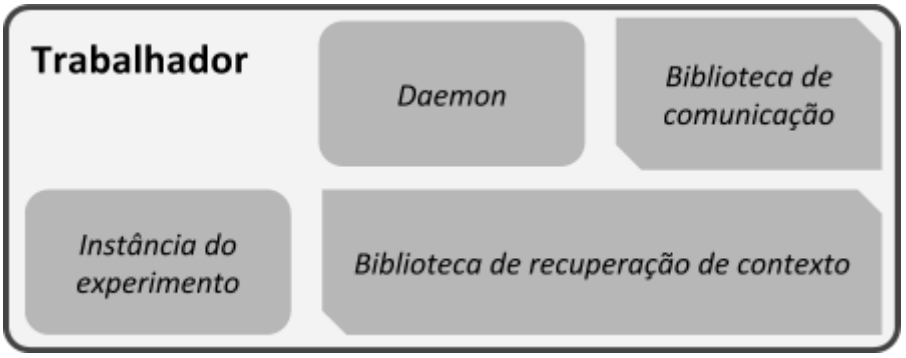
Fonte: Elaborada pelo autor

A gerência e o processamento do controlador são realizados pelo componente ***daemon***, onde seu nome indica seu funcionamento, um processo em andamento em plano de fundo sem a necessidade de interações com o usuário. Suas funcionalidades podem ser listadas como: (1) envio e instalação dos arquivos necessários para a inicialização de novos trabalhadores, assim que registrados no armazenamento de dados; (2) detecção de falhas nos trabalhadores, para então reagir de acordo com o estado deste trabalhador perdido; (3) alocação de trabalhadores, envio dos arquivos e parâmetros do experimento às máquinas e notificação do seu início, quando todos os trabalhadores estiverem preparados. O último componente a ser abordado é a **biblioteca de comunicação**, composta por todas as funções que o *daemon* necessita para se conectar aos trabalhadores e acessá-los. Para isso, funções devem ser implementadas para permitir ao *daemon*: (1) acessar comandos *shell* remotamente; (2) transferir arquivos entre controlador e trabalhador; (3) receber informação sobre o estado de conexões; e (4) notificar o início de experimentos.

2.1.2 Trabalhador

A **instância do experimento** representa a execução local de experimentos alocados a esse trabalhados. Para o experimento ser recuperável, ele deve usar as funções disponibilizadas pela **biblioteca de recuperação de contexto**, pois é nela que o padrão a ser seguindo é definido. Da mesma forma, a biblioteca fornece os meios do *daemon* de recuperar e aplicar o contexto da/na atual instância de experimento, mas como forma de comandos como “salvar” ou “carregar”, sem lidar diretamente com os dados.

Figura 2.1.2.1 - Diagrama representando o trabalhador e seus componentes



Fonte: Elaborada pelo autor

Similar ao controlador, o trabalhador também possui os componentes ***daemon*** e **biblioteca de comunicação**. O primeiro é responsável pelo controle dos experimentos localmente, garantindo seu funcionamento até o fim de sua execução e salvando seus contextos com biblioteca de recuperação. Manter o controlador informado do seu estado de atividade também é responsabilidade *daemon*. A biblioteca é o meio de comunicação entre o trabalhador e o controlador, onde suas funções devem: (1) gerenciar o estado de conexão; (2) receber notificações para iniciar experimentos; e (3) recuperar e armazenar o contexto do/no armazenamento de dados do controlador.

2.2 Desafios da Proposta

Um **experimento** é definido como a aplicação a ser executada pelo sistema. Para isso, é necessário um arquivo compactado composto por um executável e quaisquer outros arquivos que o experimento necessite durante sua execução. O arquivo compactado facilita o armazenamento e a transferência de arquivos entre computadores. Porém, não é qualquer tipo de experimento executável que será capaz de aproveitar a capacidade de recuperação do sistema. Um **indicador de progresso** do experimento é a única forma de continuar a execução quando uma nova máquina assumí-la. O experimento não só deve ser capaz de medir sua progressão, mas estar preparado para continuar uma execução interrompida. É função do usuário descobrir como implementar tal capacidade ao seu experimento, seja por uma variável de tempo ou mantendo as informações processadas até então. Além disso, experimentos podem possuir mais que um único modo de execução, onde duas máquinas devem agir de modo diferentes, definidas pelo seu parâmetro de entrada. Assim, cada um desses modos de operação são definidos como **papéis**, e é através deles que serão definidos os parâmetros que tal papel do experimento receberá em sua inicialização e o número de instâncias que são necessárias para ele.

Falhas em sistemas distribuídos não são problemas triviais, e podem ser interpretadas de diversas maneiras. Elas podem ser divididas em duas categorias iniciais: falhas **transientes** e falhas **permanentes**. As falhas transientes ocorrem durante um período de tempo finito, ou seja, após o fim desse período, ela deixa de existir. Geralmente essas falhas estão relacionadas com a comunicação entre computadores, onde um alto tráfego de rede impede a chegada de pacotes, por exemplo. Contudo, o computador remoto não encerra suas operações por problemas de conexão, e isso deve ser tratado pela sistema para preservar sua consistência. Falhas permanentes afetam a execução da máquina, interrompendo-a por definitivo. Desde falha do próprio processo em execução ao desligamento da máquina, sua comunicação com o sistema só será retomado através da reinicialização local quando reconectada. ***Heartbeats*** permitem que falhas transientes e permanentes sejam detectadas durante a execução do sistema. Seu funcionamento consistem em um envio constante de mensagens da máquina em questão, informando seu estado de atividade e sua conexão ao sistema, onde respostas às mensagens são necessárias para informar o mesmo à máquina de origem. A detecção da falha baseia-se no **tempo de resposta** das mensagens, identificado como uma falha quando o limite estabelecido pela aplicação é ultrapassado. Para implementar o sistema de *heartbeats*, o controlador ficará responsável por gerenciar todos *heartbeats* e verificar o estado funcional do sistema, detectando falhas em trabalhadores. Ao mesmo tempo, um processo local em todos trabalhadores deve enviar os *heartbeats* e receber suas confirmações. Para manter a consistência do sistema em falhas transientes, quando os *heartbeats* não forem detectados no controlador, a máquina de origem não receberá nenhuma confirmação e deverá cessar operações, para posteriormente ser reconectada ao sistema. Não há motivo para manter o funcionamento de uma máquina quando não há garantias que seu contexto ou execução estão sendo corretamente tratados.

O primeiro passo para executar um experimento em um ambiente distribuído é a alocação dos recursos necessários, nesse caso sendo as máquinas do ambiente. Para uma máquina poder executar o experimento, ela deve estar, no mínimo, conectada ao sistema e em funcionamento, e infere-se isto através dos *heartbeats*, descritos na Subseção 2.1.2. Porém, uma máquina conectada não significa que ela seja capaz de rodar o experimento. Experimentos podem utilizar muito do poder computacional e a execução de múltiplos em uma mesma máquina podem afetar um ao outro no escalonamento de recursos. Por este motivo, o **estado de disponibilidade** deve ser informada junto com o *heartbeat*, onde cada máquina informará sua capacidade de executar ou não um experimento. Com esta informação, o número de máquinas requeridas pelo experimento são alocadas, de acordo com suas disponibilidades, e o experimento enviado e executado por elas.

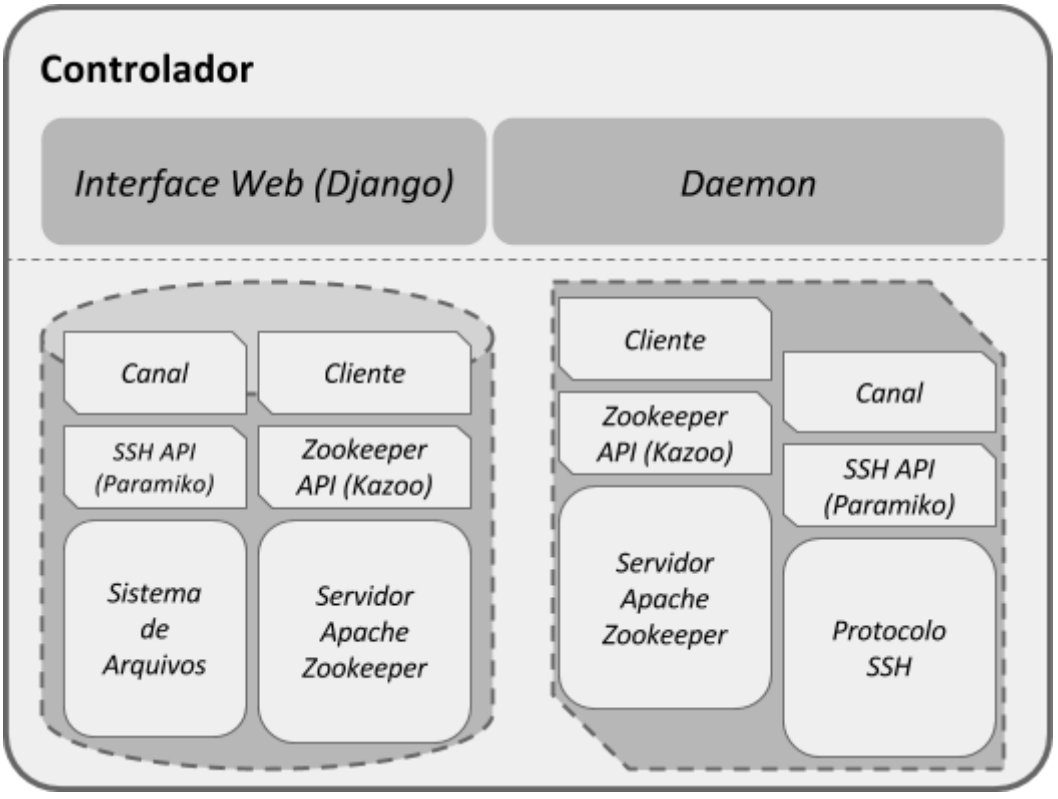
Informações mantidas em cada uma das máquinas correm o risco serem irrecuperáveis após falhas. Consequentemente, a melhor forma de evitar esse problema é manter esta

informação em algum local seguro durante seu uso. Como o envio e consulta constante dessas informações remotamente seria muito caro quanto a recursos, devido ao tempo de resposta dessas requisições afetando a execução do experimento e ao tráfego de dados, esse controle de informações pode ocorrer periodicamente. Essa é a estratégia por trás de ***checkpointing***, onde imagens (*snapshots*) do estado atual de um sistema são tiradas para que, no caso do *checkpoint*, o sistema seja capaz recuperar-se através de uma reversão (*rollback*) à última imagem registrada. Assim, a recuperação de contexto de execuções do experimento é possível, onde cada máquina envia periodicamente o conjunto de informações de todos experimentos em execução. Quando um trabalhador for perdido, as novas instâncias dos experimentos instanciados recuperarão as imagens armazenadas para continuar suas execuções baseadas nelas.

3 IMPLEMENTAÇÃO PROTOTÍPICA

O Capítulo 3 descreve como o sistema desenvolvido no Capítulo 2 foi implementado. A linguagem escolhida para implementação do projeto foi *Python* 2.7.13. Dois motivos incentivaram o uso dessa linguagem: instalação de pacotes de terceiros simplificada através do pacote *PIP*, que vem nativamente desde a versão 2.7.9; e a unificação entre todos os componentes da arquitetura. O código da implementação está disponível via *GitHub*[1]. As Seções 3.1 a 3.4 descrevem cada um dos componentes do controlador, presentes na Figura 3.1, começando com o armazenamento de dados na Seção 3.1, composto pelo servidor *Apache Zookeeper* em conjunto com o sistema de arquivos. A Seção 3.2 irá tratar como o Protocolo SSH e novamente o *Zookeeper* permitem a interação entre controlador e trabalhador. A Seção 3.3 descreve como o *daemon* é capaz de gerenciar sua múltiplas funcionalidades simultaneamente. A Seção 3.4 demonstra as capacidades da interface *web* implementada, onde dados do sistema são adicionados e consultados. Finalmente, a Seção 3.5 irá descrever a implementação do trabalhador brevemente, pela sua simplicidade.

Figura 3.1.1 - Diagrama demonstrando, com mais detalhes, como os componentes foram implementados no protótipo



Fonte: Elaborada pelo autor

[1] https://github.com/naajuniorr/expeeker

3.1 Armazenamento de Dados

O controlador precisa de um meio para armazenar os dados de todos os componentes que serão usados no sistema. Apesar de um banco de dados ser uma solução simples para este problema, uma solução mais adaptável aos trabalhos futuros foi escolhida. Entre as duas opções encontradas, *Apache Cassandra* [WANG et al. 2012] e ***Apache Zookeeper*** [HUNT et al. 2010], a segunda apresentava um ambiente que além de oferecer armazenamento dos dados de configuração do sistema, também pode notificar as modificações destes dados. O *Apache Zookeeper* é um servidor de código aberto capaz de manter *metadados* e providenciar sincronização em sistemas distribuídos. Dados são mantidos em nodos (*znodes*) como texto binário, com o limite padrão de 1 MB por nodo, e são organizados em uma nomenclatura hierárquica, similar a um sistema de arquivos, logo, o nome deve ser único entre todos os nodos filhos de um mesmo nodo pai. A limitação de espaço dos *znodes* obrigou a usar o **sistema de arquivos** do sistema operacional para armazenar os arquivos dos experimentos, armazenando no *Zookeeper* a sua localização apenas, para ser recuperável pela aplicação quando for usado enviado pelo trabalhadores. Todos os outros dados, como trabalhadores, são mantidos completamente em nodos do servidor, sendo os experimentos a única exceção.

Znodes podem assumir duas características opcionais quando criados: efemeridade, um nodo que não pode ter filhos, mas que existe enquanto a conexão do cliente que o criou se mantiver com o *Zookeeper*; e sequencialidade, o valor atual de um contador de dez dígitos é fixado ao final do nome no nodo deste tipo, e este valor deve ser maior que todos os outros nodos sequenciais do o mesmo nodo pai. *Watch* (vigilância) é um gatilho de uso único fornecido pelo servidor que deve ser configurado durante requisições de consulta ou modificação em nodos. Ele pode ser baseado nas alterações do nodo, como quando ele é criado, alterado ou removido, em alterações nos filhos de um nodos, são criados ou removidos. Cada vez que o *watch* é engatilhado, é necessário reativá-lo para que novas modificações sejam notificadas, por isso é chamado “de uso único”.

O **cliente** do controlador serve como um intermediário entre as aplicações de interface e *daemon* e a API do *Zookeeper*, o ***Kazoo***. O pacote *Kazoo* é API que permite a interação da linguagem *Python* e o servidor *Zookeeper*. A comunicação com o servidor consiste em operações simples, como criar, recuperar e editar nodos e seus valores, mas com a biblioteca intermediária “cliente”, funções realizam operações mais complexas do que fornecidas pelo pacote, como adicionar os dados de um trabalhador ou experimento ao servidor por exemplo. O cliente também trata o fato do *Zookeeper* poder armazenar dados apenas no formato texto em nodos, garantido que dados sejam transformados quando enviados e retornado no modelo usado na implementação.

3.2 Biblioteca de Comunicação

A biblioteca de comunicação pode ser dividida em 4 conjuntos de funções: (1) notificação de eventos; (2) gerenciamento de *heartbeats*; (3) comandos *shell* remotos; e (4) transferências de arquivos. Os dois primeiros são possíveis com o servidor *Zookeeper*, enquanto que os dois restantes podem ser feitos com o protocolo SSH. Para notificar eventos, a criação de um nodo no *Zookeeper* pode ser usado como um gatilho de *watches*, além de permitir que qualquer recuperação do experimento verifique que ele já foi iniciado, pois esse nodo será mantido no servidor. Assim, trabalhadores que receberam experimentos devem esperar pela criação deste nodo de sinalização para poderem executar o experimento, sendo notificados pelo *Zookeeper* quando esse evento ocorrer. A verificação de *heartbeats* é feita através dos nodos efêmeros do *Zookeeper,* criados por cada um dos trabalhadores conectados ao sistema. A existência deste nodo indica uma conexão ativa, já que o *Zookeeper* exige a conexão constante com o servidor através de seu cliente. Logo, o tratamento de envio *heartbeats* e recebimento de confirmações são realizadas pelo servidor, onde o limite de tempo de resposta aceito deve ser configurado. O protótipo utiliza o valor de 30 segundos como seu limite máximo e mínimo entre *heartbeats*. Após esse tempo, o conexão será desfeita, e o nodo será removido pelo próprio servidor. Através do cliente, o *daemon* tem acesso ao estado atual de um trabalhador verificando seu nodo de conexão.

Para poder instalar dependências necessárias para a execução de nossa solução em trabalhadores, acesso a comandos *shell* e transferência de arquivos devem ser suportados pelo sistema. O protocolo SSH fornece ambas capacidades, e é o método usado pelo sistema. O pacote ***Paramiko*** disponibiliza um cliente SSH dentro da linguagem *Python*, com suporte a transferência de arquivos via protocolo SFTP (*SSH File Transfer Protocol*). Infelizmente, após testes, percebeu-se que a maioria das máquinas utilizadas não ofereciam suporte ao protocolo SFTP, mas apenas ao protocolo **SCP** (*Secure Copy*), onde um pacote de mesmo nome permite seu uso através do *Python*. Uma biblioteca intermediária **Canal** foi criada para não ser necessário sempre instanciar ambos clientes na implementação, pois o uso de um normalmente está relacionado com o uso do outro.

3.3 Daemon[2]

Duas linhas de execução são necessárias para executar as funcionalidades do *daemon*, já que a verificação de *heartbeats* precisa estar sempre ativa, enquanto que suas operações que necessitam acessar o trabalhador, como recuperação de trabalhadores, devem ser feitas *on*

[2] O código base usado para desenvolver *daemons* na linguagem *Python* foi desenvolvida por Matthew Vliet, e está disponível em https://gist.github.com/mvliet/7649806

demand, quando elas forem requisitadas. Para isso, a verificação será realizada por um processo filho do principal, que ficará responsável por conectar-se aos trabalhadores. A verificação é feita periodicamente em intervalos de 30 segundos, assim, o intervalo de tempo que uma desconexão pode ser notada pelo *daemon* pode variar entre 30 e 60 segundos, somando o tempo de verificação com o tempo de desconexão com o *Zookeeper*.

Para gerenciar as operações com trabalhadores, será utilizada uma **fila de tarefas** mantida no *Zookeeper*, onde cada tarefa será identificada por um nodo sequencial, definindo sua ordem de inserção e permitindo seguir o conceito FIFO (*First In, First Out*) com a ordenação dos nodos pelo seu nome. A tarefa irá representar as ações que o *daemon* deve realizar, e assim que finalizada, seu nodo será removido da fila e a próxima será realizada. *Watches* são usados para notificar a chegada de novas tarefas, recuperando suas informações e reativando o *watch,* para estar preparado para novas adições. Enquanto não houver tarefas, essa linha de execução se manterá em espera da notificação do *watch*. O controlador aceita três tarefas: (1) instalação de trabalhador, que engloba a instalação do *software* de controle necessário para manter a conexão com sua máquina e suas dependências; (2) envio e execução de experimento, onde é necessário alocar o número de trabalhadores necessários, transferir o arquivo do experimento e iniciar sua execução; e (3) recuperação de trabalhadores perdidos, onde uma tentativa de reconexão e reativação é feita, e em caso de falha, um novo trabalhador será alocado para substituir execução na máquina perdida, realizando as mesmas operações da tarefa 2 recém descrita. Os desafios presentes na segunda e terceira tarefa já foram desenvolvidas na Seção 2.2, logo, suas soluções serão aplicadas aqui. A **gerência de recursos** é a segunda linha de execução do *daemon*. Sua função é verificar todos os trabalhadores do sistema e recuperá-los nos casos de desconexão.

3.4 Interface Web

Através do *framework* ***Django***, foi possível criar uma interface *web* que poderia rodar nossa biblioteca "cliente", já que o processamento da página é feito em *Python*. O *framework* possui uma grande curva de aprendizagem e, por este motivo, foi usado o mínimo possível dela, para não retirar o recursos do objetivo original do projeto. Foram implementadas três funcionalidades similares para cada um dos dois tipos dados que podem ser adicionados através da interface: operações de (1) adicionar, (2) listar e (3) consultar os dados de trabalhadores e experimentos. As telas de adição são compostas por formulários que recuperam os dados do usuário, como visto na Figura 3.4.1. Com trabalhadores, a primeira caixa de texto recupera uma lista de endereços separados por "novas linhas", a segunda recupera o usuário, a terceira recupera a chave privada em texto e a última, a senha. Então, para cada endereço, uma tarefa para instalar o trabalhador é posta na fila do *Zookeeper* através

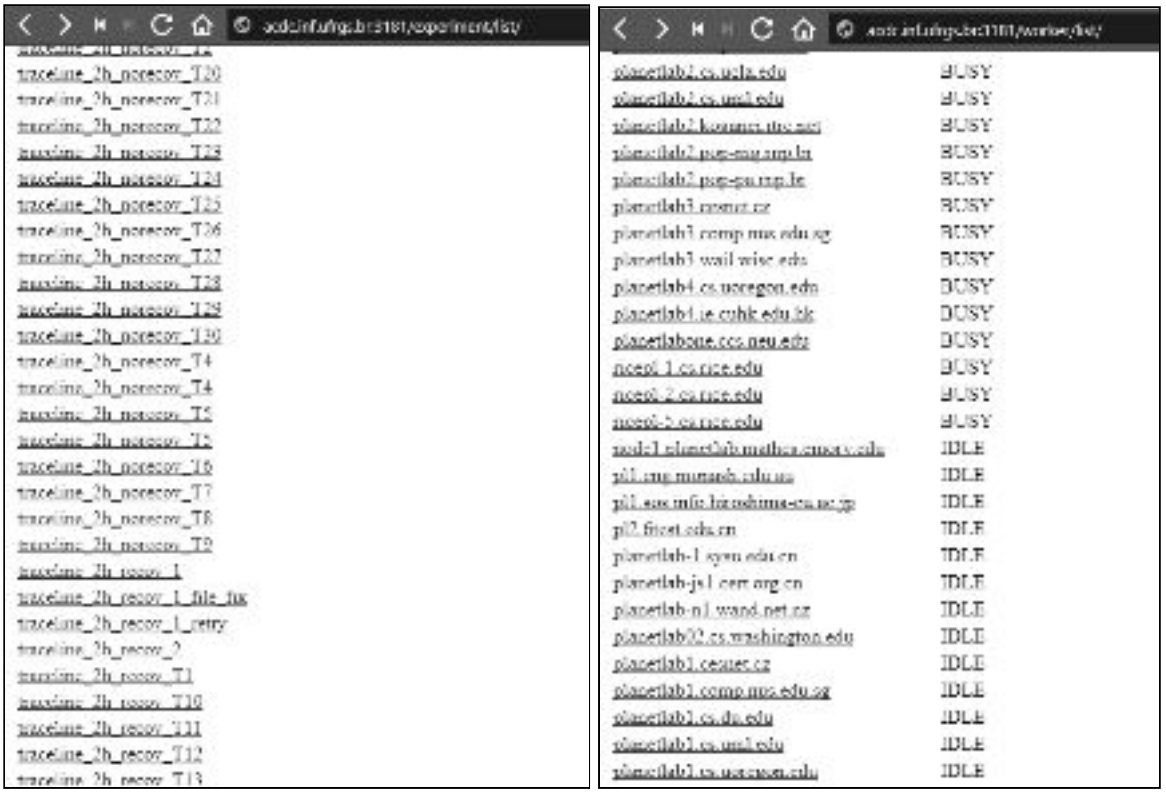
do cliente. A autenticidade dessas credenciais são verificadas apenas durante a execução da tarefa, quando a tentativa de conexão é feita.

Figura 3.4.1 - Janelas de adição de experimento (esq.) e trabalhadores (dir.)

Fonte: Capturada pelo autor

A janela de adição de experimentos recebe o nome do experimento, um arquivo contendo o experimento em formato de compactação *gzip*, uma caixa que indica se o experimento utiliza a classe *Snapshot* ou se deverá ser rodado via comando *shell* e, por último, cinco conjuntos de formulários de papéis, onde no mínimo um deve estar preenchido. Nos papéis, são definidos seu nome, parâmetros (ou comando *shell*, caso a caixa indicativa de *snapshot* esteja desmarcada) e o número de atores para este papel. O arquivo do experimento é salvo em uma pasta dentro do diretório do controlador e seu caminho mantido com os dados do experimento. Este conjunto de dados será enviada ao *Zookeeper* como uma tarefa de adicionar experimento quando o botão *Submit* for pressionado, realizando uma requisição *POST* ao servidor.

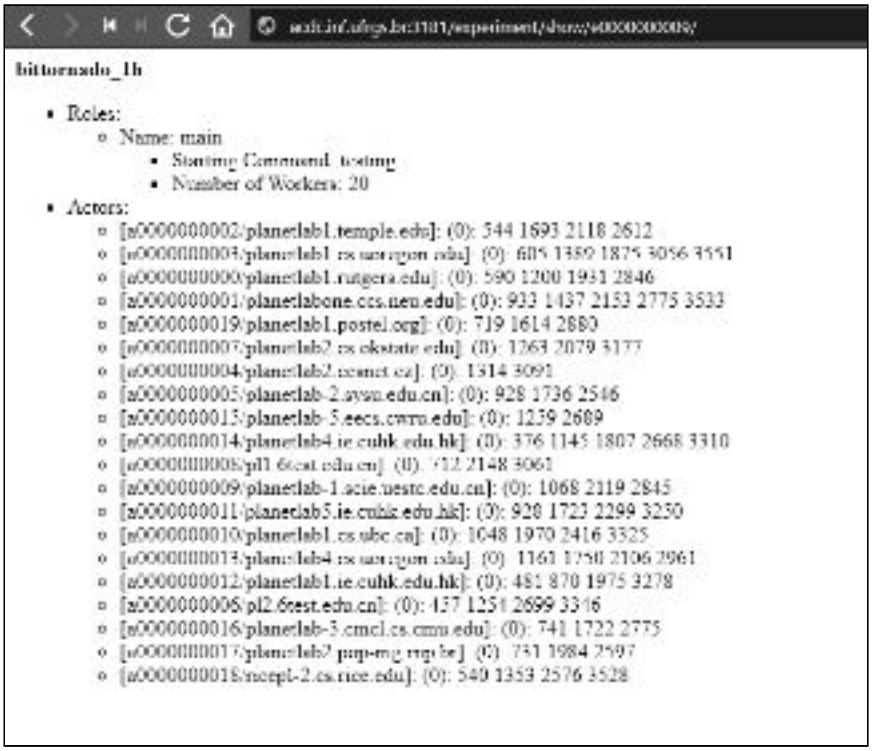
Figura 3.4.2 - Janelas contendo listagem de experimentos (esq.) e trabalhadores (dir.)



Fonte: Capturada pelo autor

As janelas de listagem e consulta são mais simples, por precisarem apenas mostrar uma lista de itens ou apenas um deles em mais detalhes. Para as janelas de listagem, demonstradas na Figura 3.4.2, são usadas os métodos para recuperar todas instâncias dos dados armazenados. A janela à esquerda lista os nomes dos experimentos que em algum momento foram adicionados ao sistema, enquanto que a janela à direita lista os *hostnames* dos trabalhadores registrados, seguido do seu estado atual de disponibilidade (*idle* para ocioso e *busy* para ocupado). Ao clicar em um destes itens, a página será redirecionada para a janela de consulta, demonstradas nas Figuras 3.4.3 e 3.4.4. Os métodos usados nas consultas recuperam o identificador passado via URL pela página de listagem, sendo eles o *hostname* do trabalhador e o identificador do experimento, e retorna suas informações. A Figura 3.4.3 representa a janela de consulta de um experimento, onde é descrito quais papéis foram declarados e as saídas de cada trabalhador que o executou. A Figura 3.4.4 representa a janela de consulta de trabalhador, onde dados como seu estado, tempo ativo, data da última conexão e desconexão com o sistema e número de falhas ocorridas são demonstradas.

Figura 3.4.3 - Janelas contendo consulta de experimento



Fonte: Capturada pelo autor

Figura 3.4.4 - Janelas contendo consulta de trabalhador

Fonte: Capturada pelo autor

3.5 Trabalhador

Trabalhadores são considerados como ocupados quando um experimento estiver em execução neles, e é função do *daemon* indicar ao controlador seu estado atual. Seu estado é representado a partir do nodo efêmero que representa sua conexão no *Zookeeper*, como já descrito na Seção 3.2. Dependendo onde esse nodo for criado, ele indicará o estado do trabalhador: se for filho dos nodos ociosos, é um trabalhador ocioso, se for filho de um nodo ocupado, é um trabalhador ocupado, e caso não esteja em nenhum dos dois, é um trabalhador desconectado. Assim, o controlador consegue determinar com facilidade quais trabalhadores estão disponíveis para alocação ou não, apenas recuperando os filhos do nodo ocioso. Logo, o

trabalhador deve ser capaz de modificar seu estado atual de acordo com a chegada e finalização de experimentos, removendo o nodo do estado anterior e instanciando o novo. Umas versão simplificado da biblioteca “cliente”, utilizando o pacote *Kazoo* para estabelecer conexão com o *Zookeeper* no controlador, apenas com as funções necessárias pelo trabalhador (Subseção 2.1.2) fará parte da biblioteca de comunicação.

Uma segunda linha de execução do *daemon* deve utilizar *watches* para verificar a designação de novos experimentos, feitas durante a alocação do trabalhador. Quando notificado, os dados e arquivos do experimento devem ser carregados e preparado para execução. Assim, um segundo *watch* deve ser ativado para verificar o início do experimento e, quando recebido, o experimento será executado. O *daemon* manterá sua execução até que seja terminado, e sua saída retornada ao *Zookeeper*. Erros de execução no experimento não são tratados, apenas reportados ao controlador como erro gerado. Como não é possível averiguar se a origem do erro é de código mal implementado ou falha do sistema, a melhor opção é deixar o usuário analisar a saída e tomar as devidas providências.

A biblioteca de recuperação de contexto é disponibilizada através da classe *Snapshot*, logo, o experimento deve ser implementado herdando a classe original e utilizando ela e seus métodos para salvar e consultar as variáveis de progresso e escrever o código do experimento. Da mesma forma, a classe fornece métodos que permite o *daemon* realizar a recuperação ou armazenamento do contexto do experimento em execução. Os dados são salvos no *Zookeeper*, onde, com a referência do contexto e do experimento, qualquer trabalhador será capaz de recuperá-lo. O *daemon* periodicamente armazenará o contexto durante sua execução, em intervalos de 30 segundos.

4 AVALIAÇÃO EXPERIMENTAL

O ambiente escolhido para instanciar os trabalhadores foi o *PlanetLab* [PETERSON et al. 2006], um *testbed,* ambientes que permitem avaliação e reprodução de experimentos científicos, de sistemas distribuídos, mantido através do trabalho conjunto de instituições ao redor do mundo. Cada usuário tem acesso a múltiplas máquinas virtuais. A instituição que ajudar fornecendo suas próprias máquinas virtuais, terá direito de acesso à plataforma através de um usuário com seu *slice*, uma coleção de máquinas virtuais. A virtualização permite que uma máquina física seja representada como uma ou mais máquinas virtuais. Em cada instância virtual, os recursos físicos serão divididos e/ou limitados, com os objetivos de aumentar a capacidade de computação distribuída e de manter a segurança do sistema, garantindo que apenas os recursos que foram designados à virtualização sejam utilizados pelos usuários das máquinas.

A fragilidade do *PlanetLab* destaca os problemas em ambientes distribuídos. Em tentativas de conexão SSH com um *slice* contendo 377 máquinas registradas, apenas 70 delas foram capaz de responder ao comando, as restantes ultrapassaram do tempo de resposta de 30 segundos, negaram conexão, ou não foram encontradas. O resultado apresenta uma disponibilidade menor que 20% que se mantém em múltiplas tentativas ao longo de dias. Os próprios responsáveis pela conduta do ambiente [SPRING et al. 2005] afirmam que ele não é capaz de reproduzir experimentos em execuções únicas, mas que múltiplas execuções é possível alcançar resultados significativos estatisticamente, e que tal instabilidade deve ser considerada como uma funcionalidade para testar sistemas instanciados neste ambiente, não como uma falha.

O principal sistema operacional do ambiente é Linux Fedora 8, com casos de Linux Fedora 12 e Linux Centos 6, com o endereçamento de 32-bits sendo maioria. Em ordem decrescente de frequência, os processadores mais usados são das famílias: Intel Xeon, Intel Core 2 Duo, Intel Pentium D, Intel Core 2 Quad, Intel i5 e AMD Opteron. A memória RAM varia entre 1 GB e 16 GB, sendo aproximadamente 4 GB a moda. A máquina usada como controlador é um servidor com quatro processadores AMD Opteron 6276, totalizando 64 núcleos a 2.3 GHz de frequência, e 64 GB de memória RAM. Seu sistema operacional é o *Linux Ubuntu* 16.04.2 LTS com *kernel 4.4.0-66-generic*. As versões de *softwares* e bibliotecas usados são: *Kazoo* 2.4, *Paramiko* 2.2.1, *SCP* 0.10.2, *Django* 1.10.5, *PIP* 9.0.1 e *Apache Zookeeper* 3.4.9. O Capítulo 4 irá descrever como foi possível avaliar a eficácia do protótipo em um ambiente real de testes, o *PlanetLab*. A proposta de avaliação e seus objetivos são apresentados na Seção 4.1, e seus resultados e análise, discutidos na Seção 4.2.

4.1 Caso de Avaliação: Precisão de Requisições

Um sistema cliente-servidor será simulado, similar ao exemplo citado no Capítulo 1, onde um servidor HTTP deverá receber requisições POST de clientes em tempos determinados por um arquivo como roteiro a ser seguido. O servidor é a única forma de unificar as requisições dos diferentes trabalhadores em um relógio único, gerando um arquivo similar ao roteiro, por isso ele deve anotar o *timestamp* das requisições POST recebidas. *Timestamps* de desconexão são definidas pelo controlador quando notadas em sua gerência de recursos. Trabalhadores seguirão localmente o arquivo contendo o roteiro com os instantes que uma requisição deve ser enviada, enviando-las assim que acreditarem ser o instante correto. É mantido, através da recuperação de contexto, o tempo percorrido pelo trabalhador até então, onde o seu substituto deve seguir a partir do último contexto salvo, consequentemente, percorrendo o roteiro até seu fim. Como não é viável reproduzir avaliações com tempos longos de execução, como semanas, falhas são injetadas para estimular os efeitos no sistema quando estas são ou não tratados. As injeções seguem um arquivo próprio com *timestamps*, que deve ser seguido pelos trabalhadores, onde em um tempo determinado pelo roteiro a execução completa do trabalhador, não apenas da instância do experimento, será suspendida. Falhas provocadas pelo próprio ambiente também estarão presentes durante a execução das avaliações e são incluídas no registro de falhas da avaliação. Exemplos de possíveis falhas de ambientes já foram citadas durante a descrição do desafio de detecção de falhas na Seção 2.2. O objetivo desse caso de teste é de verificar a capacidade do protótipo implementado manter a precisão de um arquivo de traços ao longo do tempo baseada nos *timestamps* em que o servidor recebe as requisições. Como qualquer uma das propostas do estado da arte analisadas são incapazes de aplicar tal experimento sem prejudicar seus resultados com duplicação de requisições ou são incapazes de rodar no ambiente selecionado, a avaliação será uma comparação da arquitetura *Expeeker* tratando e não tratando as falhas detectadas em trabalhadores.

Como parâmetros da avaliação, tem-se: o número de trabalhadores que devem ser alocados; o número de trabalhadores que devem estar disponíveis como substitutos; e o tempo de duração do teste, definido pelos roteiros a serem seguidos. Como parâmetros da implementação, tem-se: o intervalo entre armazenamento de contextos; o tempo para o controlador considerar um trabalhador como desconectado; e o intervalo de verificação do estado atual dos trabalhadores feito pelo controlador. Com isso, os testes serão feitos com 50 máquinas durante o período de 2 horas e 24 horas, onde 20 máquinas ficarão disponíveis como substitutos caso as titulares sejam incapazes de serem reconectadas. Todos os parâmetros da arquitetura estão definidas como 30 segundos cada, como já mencionado nas Seções 3.2, 3.3 e 3.5. Para cada uma das baterias, o roteiro possui uma distribuição diferente

das requisições. O roteiro de 2 horas possui 2.915 requisições distribuídas uniformemente ao longo do tempo. A injeção de falhas é distribuída aleatoriamente, com um total de 24 falhas cobrindo 48% das 50 máquinas do experimento, ou seja, uma falha em 24 das máquinas alocadas. O roteiro de 24 horas possui 47.114 requisições distribuídas por duas curvas normais em sequência, com centros em 6 e 18 horas. Essa distribuição foi escolhida para simular momentos onde haveriam picos de requisições de clientes a um servidor, e, quando houver falhas no ambiente, a disparidade entre resultado e o roteiro será mais visível nestes trechos. A injeção de falhas é distribuída por uma curva normal com centro em 12 horas, onde o motivo para tal decisão é de ser perceptível a diferença entre o primeiro e o segundo pico de requisições quanto à precisão dos resultados. São injetadas 99 falhas com uma cobertura de 96% das máquinas.

4.2 Resultados

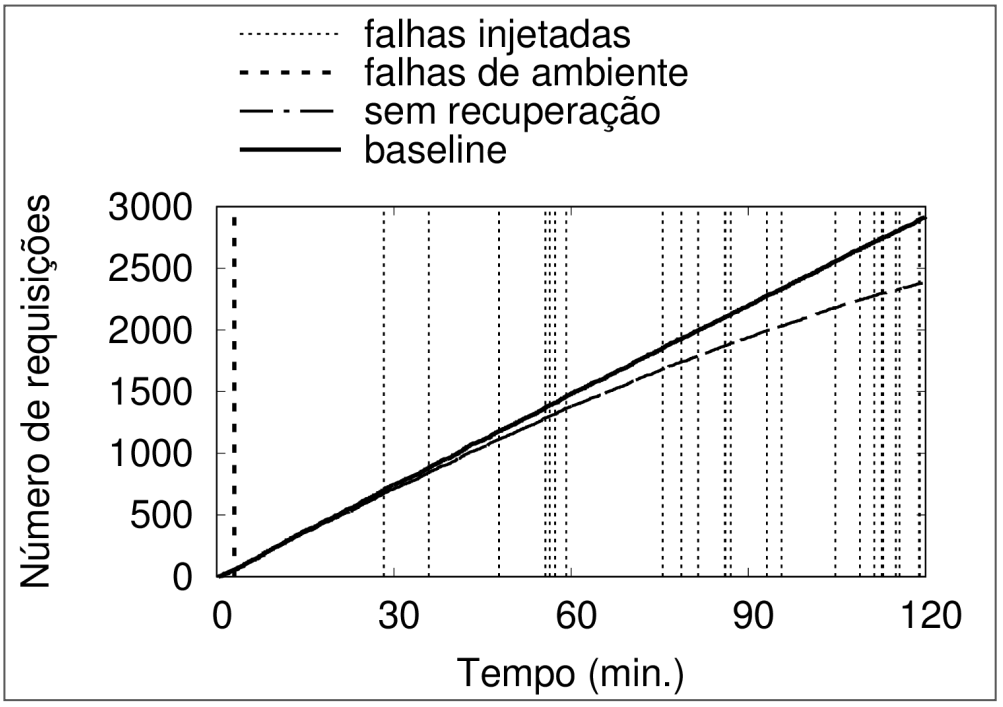
Primeiramente descreveremos como devem ser lidos os gráficos apresentados nesta Seção. Seu objetivo é de apresentar a precisão ao comparar os resultados obtidos em execuções no ambiente de avaliação com determinadas configurações, representadas pela linha tracejada, com o *baseline*, representada pela linha constante. O *baseline* deve ser considerado como um limite teórico, onde todas as requisições foram realizadas no seu tempo exato e nenhum pacote foi perdido durante a execução do experimento. Qualquer variação de tempo ou falha de envio das requisições fará com que os resultados afastem-se do *baseline*. Os momentos de falha em trabalhadores durante a execução foram notados pelo controlador nos momentos demonstrados pelas linhas pontilhadas. Quanto maior o número de falhas neste ambiente de execução sem recuperação, o número de requisições não enviadas de acordo com o tempo aumentará proporcionalmente, sendo representada pelo grau de divergência entre as linhas. Pela solução com recuperação manter o número de trabalhadores sempre o mais próximo do exigido durante a configuração do experimento, a diferença e o grau de divergência dos resultados em relação ao *baseline* deveriam ser as menores possíveis e, no pior caso, onde as falhas não podem ser tratadas, iguais a diferença e grau de divergência do resultado sem tratamento.

4.2.1 Bateria de 2 horas

A Figura 4.2.1.1 contém um caso base para comparação, como os resultados deveriam ser representados caso as falhas do ambiente não fossem tratadas durante a execução. Das 2.915 requisições esperadas, foram recebidas 2.380, aproximadamente 82% do esperado. 25 falhas ocorreram durante a execução, onde 24 foram injetadas. Apenas 25 máquinas

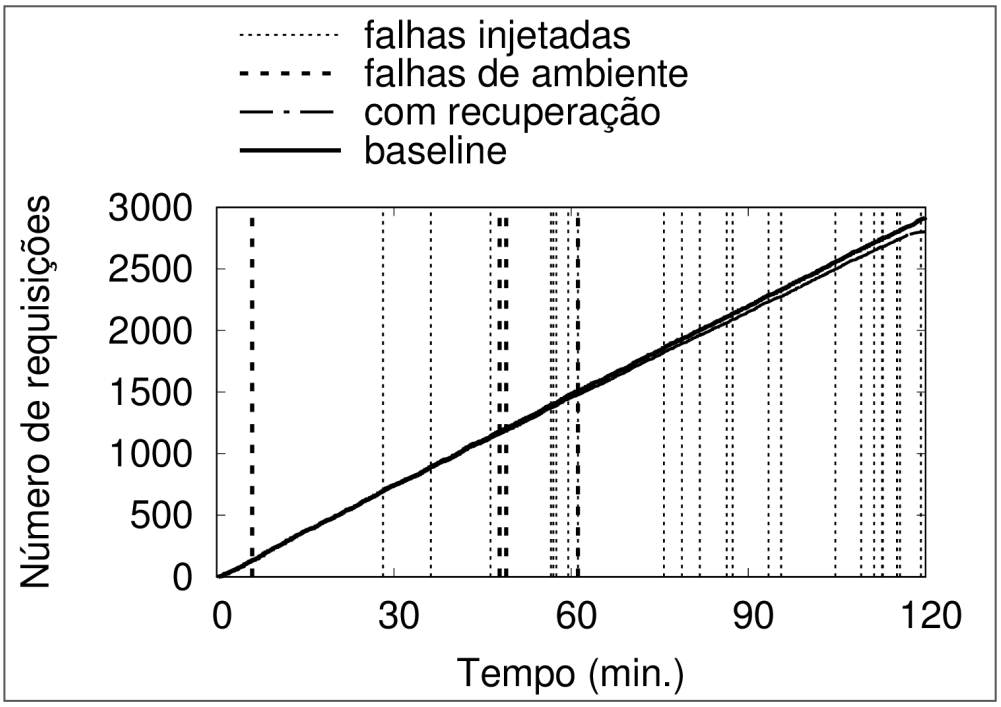
terminaram a execução, como o esperado de acordo com o número máquinas em execução e falhas. A Figura 4.2.1.2 apresenta os resultados obtidos em uma das execuções da simulação no ambiente com o uso da recuperação de falhas. 2.803 requisições foram recebidas, 96% do esperado. 28 falhas ocorreram durante a execução, sendo 4 delas ocasionadas pelo ambiente. Mesmo assim, o sistema terminou sua execução em todas 50 máquinas e o tratamento das falhas se aproximou mais ao *baseline*, replicando com uma precisão maior o roteiro.

Figura 4.2.1.1 - Gráfico contendo resultados de uma avaliação em um ambiente sem recuperação de falhas, baseado em número de requisições ao longo de 2 horas



Fonte: Elaborada pelo autor

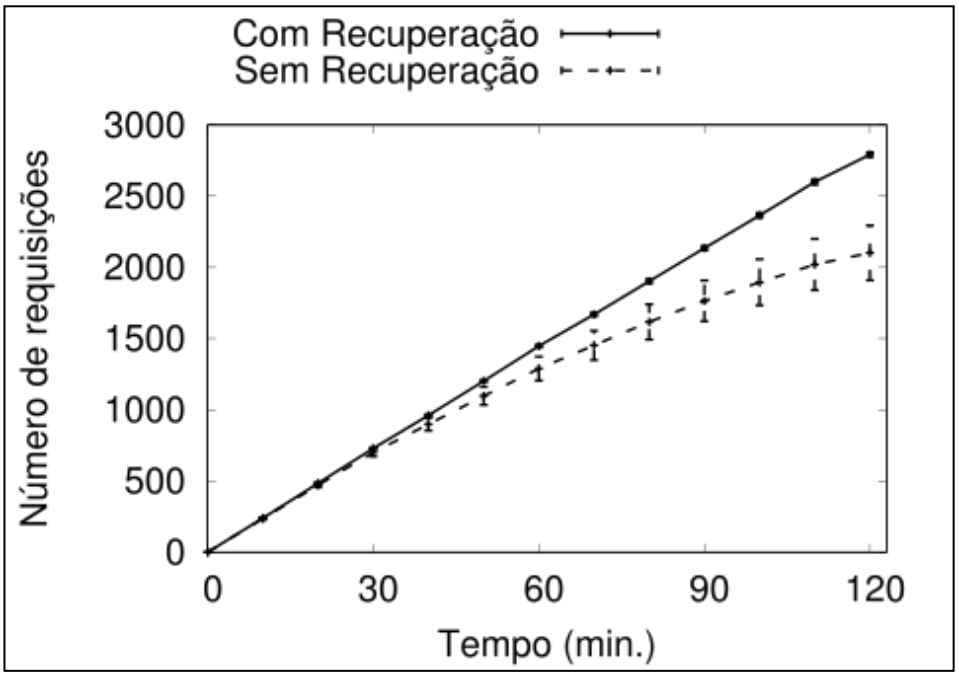
Figura 4.2.1.2 - Gráfico contendo resultados de uma avaliação em um ambiente com recuperação de falhas, baseado em número de requisições ao longo de 2 horas



Fonte: Elaborado pelo autor

Para comprovar a eficácia do protótipo, foram realizados 30 execuções do experimento de 2 horas em cada uma das duas configurações. Com todos resultados coletados, a média das requisições e seu intervalo de confiança com precisão de 95% foram calculados em períodos de 10 minutos, gerando o gráfico apresentado pela figura 4.2.1.3 e os dados que compõe a tabela 4.2.1.1. O intervalo de confiança no instante de 120 minutos dos experimentos com recuperação é uma variação de apenas 16 requisições para mais ou para menos da média, enquanto que o para sem recuperação, esse intervalo aumenta para uma variação de 192 requisições para mais ou para menos. A consistência dos resultados obtidos com recuperação demonstram como o protótipo, e, consequentemente, a arquitetura desenvolvida, foram capazes de solucionar o problema proposto: permitir a maior replicabilidade de experimentos realizados em ambientes com nodos de baixa confiabilidade.

Figura 4.2.1.3 - Gráfico contendo a média dos resultados de 30 avaliações em ambientes com e sem recuperação de falhas, baseado em número de requisições ao longo de 2 horas



Fonte: Elaborada pelo autor

Tabela 4.2.1.1 - Tabela contendo a média dos resultados de 30 avaliações em ambientes com e sem recuperação de falhas, baseado em número de requisições ao longo de 2 horas

Tempo (min.)	**Sem Recuperação**			**Com Recuperação**		
	Média (μ)	*Lim. Inf. do Int. de Conf.*	*Lim. Sup. do Int de. Conf.*	*Média (μ)*	*Lim. Inf. do Int. de Conf.*	*Lim. Sup. do Int. de Conf.*
10	236	229.5	242.4	239	235	242.9
20	470	454.7	485.2	484	480	487.9
30	700	671.5	728.4	729	723.1	734.8
40	898	853.3	942.6	961	954.9	967
50	1097	1033.6	1160.3	1202	1194.1	1209.8
60	1289	1205.1	1372.8	1448	1437.6	1458.3
70	1452	1348.7	1555.2	1670	1656.4	1683.5
80	1617	1493.4	1740.5	1903	1887.8	1918.1
90	1765	1621.7	1908.3	2135	2119.8	2150.1
100	1895	1733.6	2056.3	2365	2348.1	2381.8
110	2020	1840.4	2199.5	2600	2582.4	2617.5
120	2102	1910	2293.9	2792	2775.8	2808.1

Fonte: Elaborada pelo autor

A Tabela 4.2.1.2 apresenta dados sobre falhas ocorridas durante a experimentação, medidas de forma acumulada a cada intervalo de 10 minutos. As falhas são classificadas entre injetadas (aquelas inseridas para avaliar a robustez da solução proposta) e falhas de ambiente (aquelas que naturalmente ocorrem, e que são o objetivo primário deste trabalho). Observe que enquanto um dos experimentos não teve nenhuma falha de ambiente, outro teve até 60 falhas, o que mostra a natureza imprevisível e aleatória das mesmas. Quanto às falhas injetadas, algumas delas não puderam sê-lo devido a alguma falha de ambiente que já havia ocorrido no nodo no qual a falha deveria ter sido injetada.

Tabela 4.2.1.2 - Tabela contendo número de falhas injetadas e de ambiente detectadas durante as 30 avaliações em um ambiente com recuperação

Tempo (min.)	**Falhas Injetadas**				**Falhas de Ambiente**			
	Média (μ)	*Desv. Pad. (σ)*	*Mínimo*	*Máximo*	*Média (μ)*	*Desv. Pad. (σ)*	*Mínimo*	*Máximo*
10	1	0	1	2	3	6.9	0	34
20	2	0	2	3	4	6.8	0	34
30	5	0	4	6	7	9.1	0	34
40	6	0	6	8	8	11.7	0	52
50	7	0	7	9	10	12.5	0	52
60	12	1	10	14	12	15	0	60
70	14	0	13	16	14	16.4	0	60
80	15	0	14	17	14	16.5	0	60
90	20	1.4	16	23	14	16.8	0	60
100	21	1.4	16	24	14	16.9	0	60
110	21	1.4	16	24	14	17	0	60
120	21	1.4	16	24	14	17.1	0	60

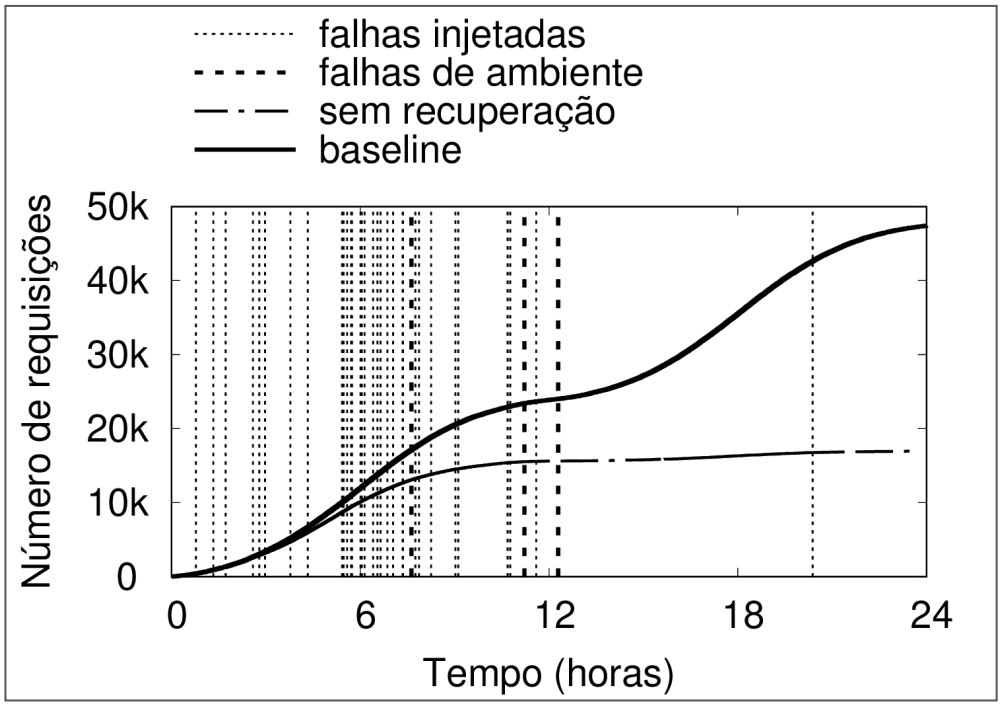
Fonte: Elaborada pelo autor

4.2.2 Bateria de 24 horas

O objetivo das execuções de 24 horas é verificar o efeito de falhas ao longo de durações maiores. Infelizmente, realizar uma bateria de múltiplos testes com tanto tempo não foi possível, então aqui serão apresentados os resultados de execuções únicas em cada uma das configurações: com e sem recuperação. A Figura 4.2.2.1 apresenta o resultado da

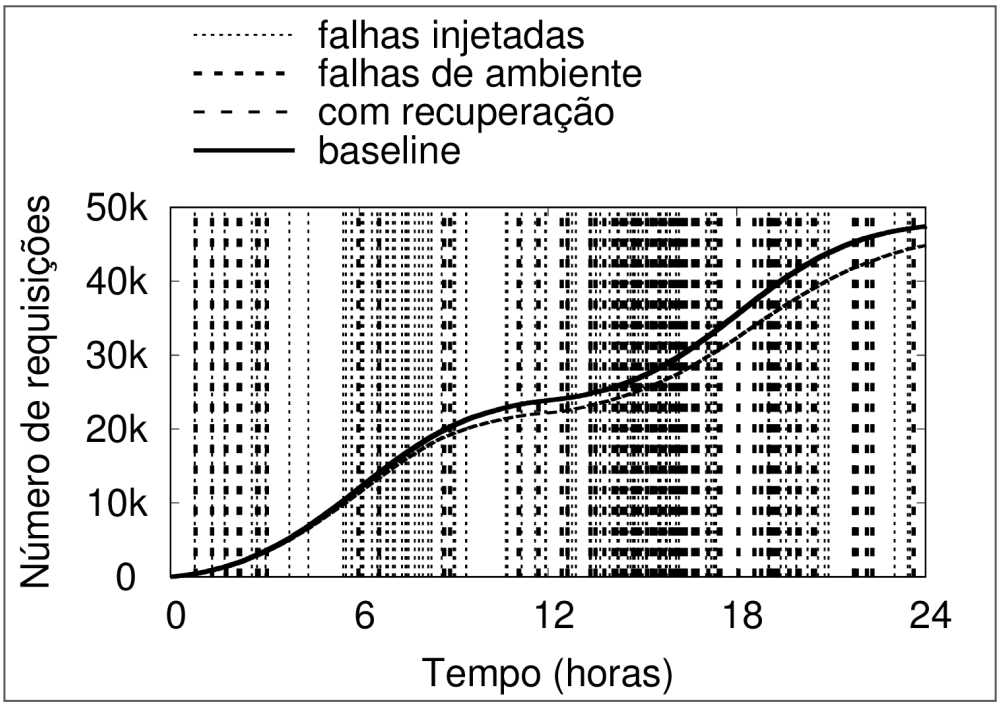
execução em um ambiente sem recuperação de trabalhadores. Com o número maior de falhas concentrados mais no centro da execução, ao invés de ao longo de toda ela como nos resultados anteriores, é possível ver o seu efeito no número de requisições, onde seu crescimento para a partir do ponto de a maioria dos trabalhadores pararam de funcionar. Inclusive é possível ver isso com o número reduzido de falhas após a metade do experimento. Das 47.114 requisições esperadas, apenas 16.951 foram recebidas, 35.7% do total. Das 48 falhas totais, 15 foram causadas pelo ambiente, logo, apenas 2 trabalhadores terminaram a execução. A Figura 4.2.2.2 demonstra os resultados obtidos utilizado a recuperação de trabalhadores. Os resultados indicam a eficácia do sistema, onde mesmo sob 333 falhas, onde 239 são do ambiente, foi capaz de realizar 44.863 das requisições do roteiro, 94.6% do total. Nem todas as máquinas foram capaz de finalizar a execução, 6 não foram corretamente recuperadas, indicando que o protótipo ainda possui problemas a serem corrigidos. Nota-se o número excessivo de falhas do ambiente, comprovando a instabilidade do *PlantetLab*, aproximadamente 10 falhas por hora.

Figura 4.2.2.1 - Gráfico contendo resultados de uma avaliação em um ambiente sem recuperação de falhas, baseado em número de requisições ao longo de 24 horas



Fonte: Elaborada pelo autor

Figura 4.2.2.2 - Gráfico contendo resultados de uma avaliação em um ambiente com recuperação de falhas, baseado em número de requisições ao longo de 24 horas



Fonte: Elaborada pelo autor

5 CONSIDERAÇÕES FINAIS

A arquitetura *Expeeker* é capaz de recuperar trabalhadores falhos e o armazenar seus contextos, garantindo um grau de replicabilidade maior aos experimentos, apesar das adversidades de ambientes distribuídos. Os resultados obtidos pelo protótipo implementado foram apresentados no Capítulo 4, e a sua capacidade de replicar experimentos, dados os recursos necessários, foi comprovada. Contudo, ainda há espaço para evolução do sistema, adicionando as funcionalidades à arquitetura, citadas nas Seções seguintes. O escopo deste projeto sempre foi grande, e funcionalidades planejadas tiveram que ser postas de lado pelo tempo disponível para a finalização deste trabalho. A Seção 5.1 irá discutir 2 propostas que serão adaptadas para usar o *Expeeker*, avaliando se é possível alcançar novos resultados com o protótipo implementado. A Seção 5.2 descreverá as limitações ainda existentes na arquitetura, e as formas que elas podem ser tratadas com o sistema para superá-las, restando apenas a avaliação dessas propostas para comprovar sua eficácia. Por último, a Seção 5.3 irá tratar de novas funcionalidades que auxiliam na criação de um ambiente mais propício a execução de experimentos, como a sincronização entre múltiplos trabalhadores.

5.1 Discussão sobre Casos de Estudo

Para avaliar as capacidades do *Expeeker* com experimentos reais, dois casos de estudo foram selecionados. Ambos são experimentos com longos períodos de execução e com progresso a ser mantido ao longo de sua duração. Com o *Expeeker*, revisitar estas propostas procurando avaliar a diferença de resultados com os apresentados em suas publicações seria um ótimo indicativo da eficácia da arquitetura e de sua utilidade em pesquisas. Caso os resultados sejam satisfatórios, também deseja-se avaliar durações maiores que as já alcançadas pelas propostas, com o intuito de verificar capacidade das soluções selecionadas em situações que eram inviáveis anteriormente.

O exemplo apresentado no Capítulo 1 e o caso de teste da Seção 4.1 foram baseados em um destes trabalhos, mas simplificado para focar apenas na avaliação da arquitetura *Expeeker*. O artigo original utiliza para sua experimentação um sistema baseado em solicitação de identidades em redes P2P (*Peer-to-Peer*), necessário para simular um ataque *Sybil*, definido como o forjamento de múltiplas identidade com objetivos maliciosos sobre tal rede, como alterar mensagens entre seus componentes. O objetivo de tal simulação era de avaliar uma solução de baixo custo energético e processual para tratar tal ataque [CORDEIRO et al. 2012]. Máquinas de uma ambiente distribuído devem seguir um arquivo de traços para simular dito sistema, onde apenas algumas delas realizam tentativas de um ataque *Sybil* no provedor de identidades, enquanto que outras seriam apenas usuários de tal rede, e a solução

deveria ser capaz de identificar e conter os ataques, reduzindo os recursos utilizados para processar as requisições recebidas. A experimentação foi realizada também no ambiente *PlanetLab*, mas sua duração foi limitada a 2 horas para evitar perdas de máquinas durante experimentação, já que este é um ambiente instável e falhas afetariam a repetibilidade dos resultados obtidos. A arquitetura *Expeeker* seria usada similarmente ao caso de teste da Seção 4.1, com o progresso do traço sendo salvo e sua execução mantida pelas máquinas substitutas.

O segundo caso de teste utiliza um ambiente distribuído não para a simulação de um sistema, mas como um extrator de dados. A proposta se baseia na reconstrução de sessões de usuários em redes P2P [CORDEIRO et al. 2014], onde imagens periódicas dos usuários presentes são coletadas, mas por limitações destas redes, essas imagens são incompletas, contendo apenas parte do conjunto completo, e há um tempo de espera entre as requisições para recuperar essa informação. A solução do problema utiliza múltiplas imagens obtidas do mesmo instante por diferentes máquinas para burlar limitações, e, com elas, reconstruir a imagem completa da rede unindo todas as imagens. Porém, dependendo da quantidade de imagens, ainda é possível que esta visão esteja incompleta, e este é o principal desafio a ser superado pelo trabalho: reconstruir a sessão estatisticamente com diferentes números de imagens ao longo de várias capturas, verificando quantas são necessárias para se alcançar resultados mais próximos ao verdadeiro estado da rede observada. O ambiente escolhido para realizar a experimentação é novamente o *PlanetLab*, e o desafio é manter todas as máquinas continuamente extraindo as imagens de sessões, onde perdas destas máquinas ocorrem durante a execução, decrementando o número imagens coletadas por instantes ao longo do tempo e, consequentemente, prejudicando sua qualidade final. Deseja-se utilizar a arquitetura *Expeeker* para garantir o funcionamento de todas as instâncias de máquinas extratoras enquanto o experimento estiver em execução, mantendo a noção de qual e quando a próxima imagem deve ser capturada.

5.2 Limitações da Arquitetura

O *Expeeker* não é capaz de fornecer um ambiente totalmente tolerante a falhas ou compatível com todos os experimentos em ambientes distribuídos. Limitações foram estabelecidas durante a apresentação deste trabalho, mas não quer dizer que propostas de soluções para tais problemas não foram elaboradas, apenas não foram implementadas e postas em avaliação. Por esse motivo, sem sua eficácia validada, não foram adicionadas como características definitivas da proposta. O *Expeeker* requer que o controlador mantenha-se ativo em todos os momentos, por ser o ponto central da arquitetura e sua atividade constante é fundamental. Quando qualquer tipo de falha atingir este componente, todas os trabalhadores conectadas a ele serão desativados, como descrito na Subseção 2.2. Não é possível continuar

suas execuções, por não existir outro controlador para controlar a consistência da arquitetura. Uma maior disponibilidade do controlador pode ser alcançada através da sua replicação em diversas máquinas, obrigando que todos os dados também sejam replicados constantemente, para não serem perdidos por uma falha. Uma das instâncias do controlador seria o líder, executando todas as funcionalidades já definidas, e as demais seriam auxiliares, verificando o estado do controlador líder via *heartbeats* e realizar um algoritmo de eleição de um novo líder em caso da desconexão do atual. Por sua vez, o líder será responsável pela recuperação dos auxiliares, pois suas falhas impediriam a recuperação de um novo líder. Para evitar problemas de conexão afetando múltiplos controladores, como tráfego ou bipartição de rede, a melhor opção seria instanciar cada um em redes diferentes.

Já abordado na Subseção 2.1.1, instâncias de experimento não podem se comunicar com outras existentes de forma direta. Por ser um ambiente alocado dinamicamente, não há como saber quais máquinas farão parte do experimento em um dado momento durante sua execução. Apesar da lista inicial ser estabelecida durante a inicialização do experimento, falhas de máquinas alteram essa lista, pois suas substitutas não fazem parte dela. No caso do exemplo durante o Capítulo 1, a entidade Servidor não poderia ser alocada pelo sistema, pois o endereço para o qual as requisições devem ser enviadas é fixa desde o início de sua execução. Por esse motivo, experimentos com comunicação entre outras instâncias do mesmo experimento é impossível, sendo sua solução atual determinar previamente a máquina destino e instanciar ela manualmente, passando seu endereço como parâmetro de execução às máquinas alocadas. Comunicação entre trabalhadores pode ser resolvida com a implementação de um serviço similar a um DNS (*Domain Name Server*), onde a referência ao contexto retornaria seu atual trabalhador, assim, permitindo a comunicação entre as máquinas. Esta seria uma funcionalidade fornecida pelo controlador por já possuir a lista de trabalhadores alocados para cada contexto, onde trabalhadores poderiam requisitar tal informação quando necessário. Todos os efeito de falhas entre trabalhadores conectados ainda precisam ser analisadas mais a fundo em múltiplos casos de uso, para verificar as necessidade de recuperação e tratamento que tal evento exigiria.

5.3 Trabalhos Futuros

Novas funcionalidades já estão planejadas para o sistema, além da realização das avaliações dos estudos de casos e suprimento das limitações ainda existentes na arquitetura. Ela visam cobrir experimentos ou conveniências que a implementação atual não é capaz sozinha. A primeira delas surgiu da capacidade do próprio sistema ser programável pelo pesquisador, onde ele poderia implementar seus próprios limiares e parâmetros de controle. Por exemplo, permitir a configuração de qual a condição para um trabalhador ser considerado

como disponível ou indisponível à alocação ou atividade e, para isso, permitir o uso de variáveis de execução do trabalhador ou do próprio experimento para determinar tal condição, como a degradação de seu desempenho. Assim, heurísticas seriam definidas a partir da necessidade do pesquisador para gerar o melhor ambiente possível para a execução do experimento desejado.

Outra funcionalidade a ser adicionada vem de um problema existente no campo da computação distribuída, mas principalmente presente quando as máquinas estão geograficamente distantes. O problema referido é a dessincronia de relógio entre máquinas distribuídas, onde duas máquinas dificilmente terão seus relógios com o mesmo horário, independente de fuso horário. Em uma ambiente como o *PlanetLab*, há diferenças superiores a 30 minutos, por exemplo, devido ao fuso horário local de cada máquina. A solução que o *Expeeker* forneceria é o uso da hora de apenas uma origem, nesse caso, do controlador. Ao iniciar um trabalhador, a hora do controlador seria requisitada, e o momento que a resposta for recebida, será anotado seu tempo. Somando a hora do controlador com a diferença de tempo desde a resposta, temos a hora sincronizada como resultado. Finalmente, como relógios não são a única forma possível de sincronização, primitivas como semáforos e barreiras também auxiliam no desenvolvimento de experimento, e seriam ótimas adições ao sistema.

REFERÊNCIAS

TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed systems: principles and paradigms**. Prentice-Hall, 2007.

GUERRAOUI, Rachid; RODRIGUES, Luís. **Introduction to reliable distributed programming**. Springer Science & Business Media, 2006.

PENG, Roger D. Reproducible research in computational science. **Science**, v. 334, n. 6060, p. 1226-1227, 2011.

DRUMMOND, Chris. Replicability is not reproducibility: nor is it good science. 2009.

ACM, Artifact Review and Badging. Disponível em: <http://www.acm.org/publications/policies/artifact-review-badging>. Acesso em 3 de julho de 2017

BAKER, Monya. 1,500 scientists lift the lid on reproducibility. **Nature**, 2016. Disponível em: <http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970?WT.mc_id=SFB_NNEWS_1508_RHBox>. Acesso em 3 de julho de 2017

PETERSON, Larry; ROSCOE, Timothy. The design principles of PlanetLab. **ACM SIGOPS operating systems review**, v. 40, n. 1, p. 11-16, 2006.

SPRING, Neil et al. Using PlanetLab for network research: myths, realities, and best practices. **ACM SIGOPS Operating Systems Review**, v. 40, n. 1, p. 17-24, 2006.

GARRETT, Thiago; DUARTE JR, Elias P.; BONA, Luis CE. Estratégias de Seleção de Nodos no PlanetLab para Execução de Experimentos. **SBRC**. 2011.

COSTA, Lauro Luis; BONA, Luis CE; DUARTE JR, Elias P. Melhorando a Precisão e Repetibilidade de Experimentos no PlanetLab. **SBRC**. 2015

HANNAH, Adrian. Fabric: a system administrator's best friend. **Linux Journal**, v. 2013, n. 226, p. 3, 2013.

HOCHSTEIN, Lorin. Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way. O'Reilly Media, Inc., 2014.

MERKEL, Dirk. Docker: lightweight linux containers for consistent development and deployment. **Linux Journal**, v. 2014, n. 239, p. 2, 2014.

ALBRECHT, Jeannie R. et al. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In: **LISA**. 2007. p. 1-19.

ALBRECHT, Jeannie; HUANG, Danny Yuxing. Managing distributed applications using gush. In: **International Conference on Testbeds and Research Infrastructures**. Springer, Berlin, Heidelberg, 2010. p. 401-411.

LEONINI, Lorenzo; RIVIÈRE, Étienne; FELBER, Pascal. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). In: **NSDI**. 2009. p. 185-198.

RUIZ, Cristian C. et al. Managing large scale experiments in distributed testbeds. In: **Proceedings of the 11th IASTED International Conference**. 2013. p. 628-636.

IMBERT, Matthieu et al. Using the EXECO toolkit to perform automatic and reproducible cloud experiments. In: **Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on**. IEEE, 2013. p. 158-163.

CLAUDEL, Benoit; HUARD, Guillaume; RICHARD, Olivier. TakTuk, adaptive deployment of remote executions. In: **Proceedings of the 18th ACM international symposium on High performance distributed computing**. ACM, 2009. p. 91-100.

URBAN, Peter; DÉFAGO, Xavier; SCHIPER, André. Neko: A single environment to simulate and prototype distributed algorithms. In: **Information Networking, 2001. Proceedings. 15th International Conference on**. IEEE, 2001. p. 503-511.

QUEREILHAC, Alina et al. NEPI: An integration framework for network experimentation. In: **Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on**. IEEE, 2011. p. 1-5.

WANG, Yanyan et al. Automating experimentation on distributed testbeds. In: **Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering**. ACM, 2005. p. 164-173.

KILLIAN, Charles Edwin et al. Mace: language support for building distributed systems. In: **ACM SIGPLAN Notices**. ACM, 2007. p. 179-188.

RAKOTOARIVELO, Thierry; JOURJON, Guillaume; OTT, Max. Designing and orchestrating reproducible experiments on federated networking testbeds. **Computer Networks**, v. 63, p. 173-187, 2014.

SPINELLIS, Diomidis. Service orchestration with rundeck. **IEEE Software**, v. 31, n. 4, p.

16-18, 2014.

BUCHERT, Tomasz et al. A survey of general-purpose experiment management tools for distributed systems. **Future Generation Computer Systems**, v. 45, p. 1-12, 2015.

HUNT, Patrick et al. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: **USENIX annual technical conference**. 2010. p. 9.

WANG, Guoxi; TANG, Jianfeng. The nosql principles and basic application of cassandra model. In: **Computer Science & Service System (CSSS), 2012 International Conference on**. IEEE, 2012. p. 1332-1335.

SANDVE, Geir Kjetil et al. Ten simple rules for reproducible computational research. **PLoS computational biology**, v. 9, n. 10, p. e1003285, 2013.