UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
PROGRAMA DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Peter Michael Claes Krause

Biblioteca para Replicação Máquina de Estados em Elixir

Florianópolis, Santa Catarina
2022

Peter Michael Claes Krause

Biblioteca para Replicação Máquina de Estados em Elixir

Monografia submetida ao Programa de Graduação em Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do título de Bacharelado em Sistemas de Informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis, Santa Catarina
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Krause, Peter Michael Claes
Biblioteca para Replicação Máquina de Estados em Elixir / Peter Michael Claes Krause ; orientador, Odorico Machado Mendizabal, 2022.
61 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistema de Informação, Florianópolis, 2022.

Inclui referências.

1. Sistema de Informação. 2. Sistemas Distribuídos. 3. Replicação Ativa. 4. Erlang. 5. Elixir. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. III. Título.

Peter Michael Claes Krause

Biblioteca para Replicação Máquina de Estados em Elixir

O presente trabalho em nível de bacharelado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Frank Augusto Siqueira, Dr.

Prof. Maicon Rafael Zatelli, Dr.

Coordenação do Programa de Graduação
em Sistemas de Informação

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, Santa Catarina, 2022.

Dedico este trabalho ao Koda, meu melhor amigo. Que
um dia eu possa te ver de novo.

AGRADECIMENTOS

Primeiramente, agradeço minha família. Meus avós Johanna, Ruth e Octávio, meus pais Luciana e Michael, meus irmãos Guilherme e Laura, e meu cunhado Edson. Meu mais profundo obrigado pela estrutura, pelo amor e pelos horizontes.

À minha namorada Carolina, pelo carinho, amor e suporte a todo momento. Por cada surpresa e por cada bilhete nesses anos de trabalho e faculdade. Que sortudo eu sou por poder namorar você.

As amizades que fiz na faculdade: Samuel, Gabriel, José, Ubiratan e Raphael. Obrigado pelas conversas, esforço e camaradagem ao longo da graduação. Vocês são parte integral deste diploma.

Aos amigos anteriores ao curso: Cairê, Gabriel, Thales, Yan e Arthur. Carrego as memórias e a parceria de vocês a todo momento. Amo-os de coração.

Agradeço meu orientador Odorico, por topar orientar este trabalho e apoiar a construção do mesmo durante tantas orientações.

Por fim, à Universidade Federal de Santa Catarina, pela infraestrutura e corpo docente, essenciais na minha formação.

“Gutta cavat lapidem”
Provérbio Latim

RESUMO

Aplicações distribuídas, como servidores modernos, lidam com uma crescente demanda por disponibilidade e velocidade. Uma das maneiras de amplificarmos a disponibilidade de um serviço é a adoção da estratégia de Replicação Máquina de Estado. Paralelamente a estas exigências de tolerância a falhas e desempenho, presenciamos uma retomada do uso de linguagens de programação funcional no desenvolvimento de aplicações, por garantir tanto segurança quanto escalabilidade no processo de desenvolvimento. Neste cenário, identificou-se o Elixir, uma linguagem de programação funcional baseada na plataforma de desenvolvimento Erlang, cuja máquina virtual BEAM oferece recursos para a construção de aplicações de larga escala, altamente disponíveis, e tolerantes a falhas. Apesar de triunfarem em aplicações monolíticas, os componentes da BEAM oferecem um grau adicional de complexidade quando direcionados à montagem de sistemas distribuídos. O objetivo deste trabalho foi a codificação de uma biblioteca de código aberto que permita a adoção da estratégia de Replicação Máquina de Estado com simplicidade, ao passo que endereça as fraquezas apresentadas pela BEAM no âmbito de aplicações distribuídas tolerantes a falhas. A utilidade do presente projeto foi comprovada por testes com um *benchmark* artificial simulador de cargas. Este teste de desempenho comparou latência e vazão de um servidor simples, um servidor com réplicas em consenso manualmente configuradas e um servidor que utiliza a biblioteca para abstrair toda a configuração necessária na segunda aplicação. Ao agir como uma fina camada entre as réplicas e a implementação do protocolo de consenso, foi verificado que a adição da biblioteca não sacrificou o desempenho da aplicação replicada em consenso, ao passo que reduziu o labor da configuração das instâncias distribuídas e suas interações.

Palavras-chave: 1. Sistemas Distribuídos. 2. Tolerância a falhas. 3. Replicação de Máquina de Estados. 4. Erlang. 5. Elixir.

ABSTRACT

Modern applications, such as web servers, face a growing demand for speed and availability. One way to address such challenges is the adoption of the State Machine Replication strategy, granting further availability to the target service. Alongside these demands, the functional programming paradigm is making a return to mainstream software development, for its scalable and safe nature. Amidst this scenario, comes Elixir, a functional programming language built on top of the Erlang development platform, whose underlying virtual machine offers powerful resources for the construction of highly-available and fault-tolerant applications that thrive while maintaining a high throughput. Although the out-of-the-box components offered by the BEAM virtual machine are battle-tested and reliable, they offer an additional level of complexity when configured to operate in a distributed environment. The purpose of this work was the the building of an easy-to-use library capable of abstracting the non-trivial work required to setup state machine replicas in a distributed Erlang environment. The library usage was verified through a synthetic benchmark test, where it was possible to see that the performance constraints imposed by the underlying consensus protocol were not increased nor decreased, proving that the library acted as a thin layer capable of reducing the configuration work it set out to do without greatly interferring in performance.

Keywords: 1. Distributed Systems. 2. Fault Tolerance. 3. State Machine Replication. 4. Erlang. 5. Elixir.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
BEAM	*Bogdan's Erlang Abstract Machine*
CPU	*Central Processing Unit*
FIFO	*First In, First Out*
IP	*Internet Protocol*
OTP	*Open Telecom Platform*
PID	*Process Identifier*
RME	Replicação Máquina de Estados
RPC	*Remote Procedure Call*

SUMÁRIO

1 INTRODUÇÃO

O funcionamento da civilização moderna se tornou dependente dos sistemas digitais que o integram. Usuários humanos contam com o funcionamento de aplicativos diversas vezes ao dia, seja para realizar uma transação bancária, participar de video conferências, acompanhar notícias, assistir filmes, ou outras atividades (COULOURIS *et al.*, 2011). É imprescindível que estes mesmos sistemas ofereçam serviço de maneira estável durante qualquer momento do dia ou localização geográfica de seus usuários, visto que a incapacidade de oferecer tal serviço pode incorrer em prejuízo monetário (ROGERS, 2020). Para mitigar os efeitos de falhas em um sistema, é possível configurá-lo para que opere de forma distribuída. Isto é, ao invés de um sistema central, pode-se ter um sistema granular com níveis parciais de falha e sucesso, visto que falhas localizadas não acarretam queda total do serviço. Neste sentido, um sistema distribuído deve satisfazer certos requisitos para ser considerado tolerante a falhas: alta disponibilidade, fazendo com que funcione corretamente pela maior quantidade de tempo possível; confiabilidade, reduzindo a quantia de defeitos durante a execução; e integridade, mantendo o estado do sistema ou recuperando-o independente de falhas (TANENBAUM; RENESSE, 1985).

No contexto de sistemas distribuídos, uma conhecida estratégia para assegurar consistência forte é a de replicação ativa por Replicação Máquina de Estados (RME) (SCHNEIDER, 1990; GUERRAOUI; SCHIPER, 1996). Nesta estratégia, um sistema adere a uma série de regras de transição de estado, junto de uma ferramenta que possa garantir consenso em relação às mensagens recebidas de clientes, para se ter uma série de réplicas de um serviço principal com estado idêntico. Em caso de falhas, é possível utilizar-se do estado de uma réplica para recuperar o estado perdido com confiança. Apesar de ser norteada por princípios simples, a implementação desta estratégia demanda entendimento dos problemas não-triviais que ocorrem em ambientes distribuídos.

De forma paralela a esta demanda, a indústria de software observa uma retomada no uso de linguagens funcionais de programação (BETTS, 2020). Os atributos fundamentais do paradigma, como imutabilidade, recursividade, e ausência de efeitos colaterais, são restrições que promovem um desenvolvimento mais seguro e escalável. Não somente isso, a peça fundamental deste paradigma – a função – permite que a composição de objetos, vista no paradigma orientado a objetos, seja substituída pela composição de funções. Funções são mais fáceis de serem orquestradas de forma distribuída do que objetos, sendo possível invocá-las de maneira remota ou aplicá-las em conjuntos de dados distribuídos. Esta mesma facilidade é um dos pontos centrais da revolução do *BigData* (DEAN; GHEMAWAT, 2008). Sem efeitos colaterais ou variáveis mutáveis, junto da facilidade em distribuir funções remotamente, o paradigma

naturalmente facilita a construção de programas concorrentes. Dentre muitas ferramentas funcionais, uma se destaca pela afinidade com sistemas distribuídos: o Erlang. Uma plataforma de desenvolvimento feita para solucionar os problemas da indústria de telecomunicações na década de 80 (ARMSTRONG, 2007a), cujo enfoque de sua máquina virtual *Bogdan's Erlang Abstract Machine* (BEAM) em suportar abstrações de concorrência fez com que a plataforma pudesse ser utilizada também na indústria de software. Com a transição da indústria de software de sistemas *desktop* para servidores remotos (CHEN, 2007), os pontos fortes do Erlang correspondem totalmente com as demandas dos servidores modernos: aplicações capazes de suportar grandes números de requisições simultâneas, e ser facilmente configurado para operar de forma distribuída entre locais diferentes.

Apesar de resolver a maioria dos problemas clássicos enfrentados na programação concorrente (ARMSTRONG, 2007b), as ferramentas da plataforma de desenvolvimento não contam com uma portabilidade segura ou de fácil uso quando postas em um ambiente distribuído. Em tempos mais recentes, se desenvolveu o Elixir, uma linguagem moderna construída sobre os pilares do Erlang, usufruindo de uma sintaxe melhorada sem perder os recursos tolerantes a falha da linguagem que a antecede, mas que ainda conta com os mesmos desafios na construção de sistemas distribuídos.

Neste trabalho, a interseção entre a implementação da estratégia de RME com o uso da plataforma de desenvolvimento Erlang expõe um ponto franco da máquina virtual e suas ferramentas. Para preencher esta lacuna, propõe-se uma biblioteca de código aberto, capaz de ser configurada com facilidade e sem demasiado entendimento do funcionamento interno de estratégias de replicação ou consenso entre mensagens, ao passo que toma responsabilidade pela orquestração de instâncias distribuídas da máquina virtual, mantendo ambas complexidades fora do escopo do desenvolvedor usuário.

1.1 MOTIVAÇÃO

A criação e gerência de um *cluster* de instâncias BEAM tolerante a falha é uma tarefa desafiadora, visto que a BEAM não oferece estruturas de replicação de estado de uma instância de máquina virtual, tampouco ferramentas de cópia ou réplica de uma instância da mesma. Não obstante, a conexão e contato entre múltiplos nodos deve ser feita através do uso de módulos Erlang puros, que, na prática, são verbosos e crípticos. Por último, a máquina virtual também não trata eventos problemáticos, como uma partição de rede, por padrão. O desenvolvedor deve abordar a possibilidade deste tipo de falha em sua própria implementação.

1.2 OBJETIVOS

O objetivo deste trabalho é desenvolver uma biblioteca em Elixir que permita o fácil emprego da estratégia de Replicação de Máquina de Estados. Com esta finalidade em mente, enumeraram-se os seguintes objetivos específicos:

1. Investigar a máquina virtual BEAM em busca de pontos de melhoria em relação às suas primitivas de Tolerância a Falhas.
2. Avaliar a facilidade em configurar a biblioteca, tomando a qualidade e simplicidade de sua *Application Programming Interface* (API) do ponto de vista de um usuário.
3. Avaliar o desempenho da biblioteca proposta em comparação com outras propostas de Replicação de Máquina de Estados. Para isso, pretendemos comparar o desempenho de serviços. Isto será feito com três aplicações distintas: um servidor simples, sem réplicas; um servidor com réplicas em consenso, porém configurado manualmente; e por último um servidor que utiliza a biblioteca para gerenciar e comunicar-se com suas réplicas.
4. Publicar a biblioteca em um repositório público sob uma licença de código aberto.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado da seguinte forma: no Capítulo 2 serão abordados os fundamentos teóricos basilares para a composição deste trabalho. O Capítulo 3 trata sobre trabalhos relacionados. No Capítulo 4, são apresentados os detalhes do desenvolvimento da biblioteca. O Capítulo 5 apresenta a avaliação dos objetivos enumerados na Seção 1.2. Por último, o Capítulo 6 apresenta considerações finais sobre o projeto e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta monografia, dois conceitos são primais para o entendimento da biblioteca desenvolvida e seu funcionamento interno:

- A linguagem de programação Erlang e sua máquina virtual, que moldam o caráter funcional do Elixir e oferecem abstrações úteis na construção de mecanismos tolerantes a falhas.
- A estratégia de Replicação de Máquina de Estados, uma forma de conferir alta disponibilidade com consistência forte a um sistema distribuído.

2.1 ERLANG E ELIXIR

Erlang é uma plataforma de desenvolvimento direcionada para a construção de sistemas escaláveis, confiáveis e com alta disponibilidade de serviço. Apesar de ter sido concebida para solucionar os problemas de larga escala enfrentados pelos sistemas de telecomunicações dos anos 80 (ARMSTRONG, 2007a), o emprego da plataforma não está restrito somente a indústria de telecomunicações, visto que estes desafios, no final das contas, eram problemas relacionados a concorrência, distribuição, tolerância a falha, escalabilidade e alta disponibilidade, fazendo com que a plataforma se mantenha relevante no desenvolvimento de *software* moderno.

Podemos sumarizar os atributos mais notáveis da plataforma Erlang em três partes: A máquina virtual, a linguagem de programação, e as ferramentas de suporte oferecidas pela plataforma. As seguintes seções entram em maior detalhe a respeito de cada uma destas partes, além de uma breve introdução ao Elixir.

2.1.1 A máquina virtual

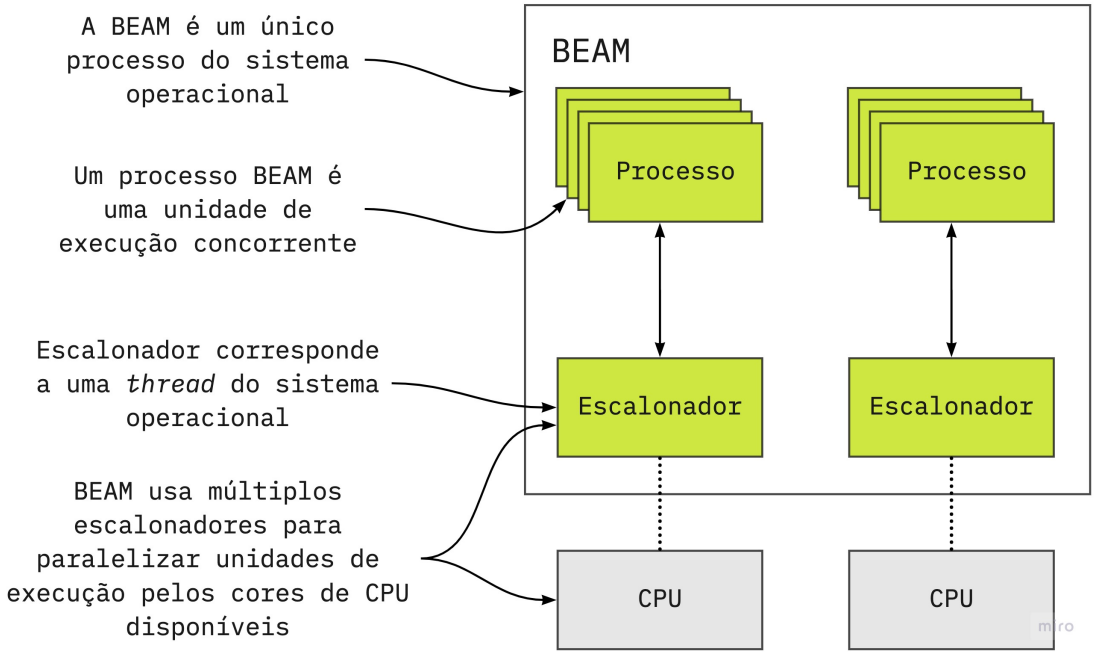
O cerne da plataforma de desenvolvimento é sua máquina virtual, conhecida como BEAM. O maior motivo para sua existência é o desejo dos criadores da plataforma por um suporte para construção de sistemas concorrentes, de forma que um usuário da ferramenta não precise recorrer a blocos de baixo nível do sistema operacional ou similar para compor trechos de código paralelos ou concorrentes.

Na máquina virtual, qualquer trecho de código é executado por um processo Erlang, e não por um processo do sistema operacional. Processos Erlang executam de forma sequencial, são identificados por um *Process Identifier* (PID) e orquestrados por Escalonadores. Um sistema típico Erlang consegue executar centenas de milhares ou até milhões destes processos simultaneamente. Isto é possível dado o modelo de concorrência (ARMSTRONG, 2007b) adotado pela máquina virtual, sumarizado nos seguintes pontos:

- **Processos leves**: processos Erlang são inicializados ocupando espaços dinamicamente alocados na pilha, que crescem conforme necessário. Além disto, por não serem processos do sistema operacional, os Escalonadores podem realizar trocas de contexto baratas visto que realizam estas trocas em pontos controlados na máquina virtual, ao contrário de um processo do sistema operacional que necessitaria salvar o estado completo do processo antes de realizar a troca de contexto.
- **Processos isolados**: processos Erlang não compartilham memória (ARMSTRONG, 2007a). Por causa disto, dispensa o uso de estruturas de sincronização como *semáforos* ou *mutex*.
- **Comunicação assíncrona**: processos Erlang se comunicam de forma assíncrona e seguem o modelo de mensagens do Ator (HEWITT *et al.*, 1973). Um processo Erlang envia mensagens a outro PID. O processo recipiente se encarrega de definir seu comportamento perante a chegada de uma mensagem.

Escalonadores estão associados a *threads* do sistema operacional, responsáveis por orquestrar e agendar a execução de processos Erlang. Possuem comportamento preemptivo e, por padrão, oferecem uma janela de tempo para a execução de cada processo bastante curta, garantindo que nenhum processo de longa duração bloqueie a execução do resto do sistema. A Figura 1 retrata o gerenciamento de processos Erlang e a interface da mesma com o sistema operacional, em alto nível.

Figura 1 – Concorrência na máquina virtual BEAM



Adaptado de: (JURIC, 2019)

2.1.1.1 Estrutura de um processo Erlang

Processos Erlang não compartilham memória. Para que trechos de execução distintos partilhem conhecimento, precisam se comunicar por troca de mensagens assíncronas. Estas mensagens, quando enviadas para um processo Erlang, são armazenadas em uma fila seguindo a ordem *First In, First Out* (FIFO). O destinatário consome estas mensagens na ordem que elas chegam, e uma mensagem só pode ser removida da fila quando for consumida. O tamanho da caixa de mensagens é limitada apenas pela memória disponível.

2.1.1.2 Instâncias distribuídas

Na plataforma de desenvolvimento é possível instanciar múltiplas instâncias da máquina virtual, que passam a ser chamadas de nodos. Dessa forma, cada nodo corresponde a uma instância BEAM com um nome associado a ela. Múltiplas instâncias da BEAM constituem um *cluster* de nodos. Processos localizados em nodos diferentes ainda podem se comunicar trocando mensagens, visto que seu PID pode ser exposto pela máquina virtual que o hospeda. O funcionamento de cada instância permanece o mesmo descrito na Seção 2.1.1: processos Erlang executam trechos de código, orquestrados por Escalonadores preemptivos executando como *threads* do sistema operacional.

2.1.2 A linguagem de programação

O Erlang é uma linguagem de programação funcional, compilada, de tipagem forte e dinâmica, que executa na máquina virtual BEAM. Dados são imutáveis e não existem referências a variáveis, salvo atributos constantes. Modificação de listas, mapas, ou quaisquer estruturas de dados, resulta no uso de outro espaço de memória para acomodar o novo valor. Tipos comuns como `int`, `float`, entre outros, são suportados no Erlang. Também há suporte para mapas, tuplas e listas utilizando os tipos *map*, *tuple* e *list* respectivamente. Uma documentação detalhada da API pode ser consultada em (ORG, 2022).

A seguir são apresentados alguns tipos específicos da linguagem ou tipos de dados comuns em linguagens funcionais.

- **Átomos**: Átomos no Erlang/Elixir são constantes cujo valor é seu próprio nome. Usados para enumerar valores, especialmente úteis para identificar mensagens trocadas por processos Erlang, átomos são declarados adicionando o prefixo “:” em um nome, como, por exemplo, `:error`. Dois átomos são equivalentes se seus nomes são iguais.

- **PID**: O identificador de um processo Erlang possui um tipo reservado no Erlang/Elixir. A estrutura de um PID é composta de três números inteiros separados por pontos, como, por exemplo: `#PID<0.14.19>`. O primeiro número do PID armazena o número do nodo que armazena este processo. O segundo número armazena os primeiros 15 *bits* do número do processo, utilizado para indexá-lo na memória da plataforma. O terceiro número do PID armazena os *bits* 16, 17 e 18 do número do processo.

2.1.3 As ferramentas da plataforma

O desenvolvimento de aplicações Erlang é indissociável das ferramentas de suporte embutidas na BEAM. Estes componentes são agrupados em um módulo conhecido como *Open Telecom Platform* (OTP). Apesar do nome novamente relacionado a indústria de telecomunicações, estas ferramentas têm um caráter de propósito geral relacionadas a construção de sistemas concorrentes e distribuídos.

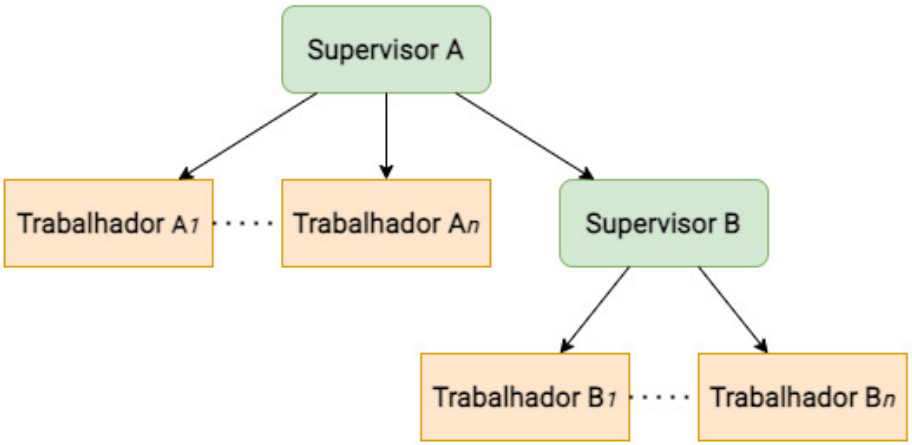
Os artifícios oferecidos pela OTP incluem ferramentas de distribuição e versionamento, conexão de instâncias BEAM distribuídas, e pacotes de código-fonte de pronto uso. Dado o foco do presente trabalho, será explicado a seguir apenas o último destes artifícios: os pacotes de código-fonte contendo abstrações utilizáveis na construção de sistemas.

2.1.3.1 Supervisor

Supervisores são processos genéricos responsáveis por administrar o funcionamento de outros processos. Esta administração inclui inicializar, reiniciar ou terminar um processo. Processos supervisionados ou inicializados por um supervisor são considerados filhos deste supervisor. Processos que não supervisionam outro processo são considerados processos Trabalhadores. A Figura 2 ilustra uma Árvore de Supervisão. No Erlang/Elixir, uma Árvore de Supervisão se refere à relação hierárquica entre processos Supervisores e seus filhos, sendo eles Trabalhadores ou Supervisores.

Dois processos na BEAM podem se conectar através de um *link*. O efeito deste *link* é fazer com que, caso um dos processos termine sua execução, o outro processo receba um *exit signal*, representado por uma mensagem que notifica o PID do processo terminado e uma razão de seu término. Processos que encerram de forma saudável enviam um *exit signal* contendo o Átomo `:normal`. Porém, caso o processo tenha sido encerrado devido um erro inesperado, o *exit signal* irá conter o Átomo `:exit`. Por padrão, um *exit signal* contendo `:exit` causará o término do outro processo conectado. Comportamentos específicos para o tratamento de saídas de processos podem ser feitos utilizando o mecanismo de captura de saída (*trap exit*), onde um processo conectado a outro por um *link* não será encerrado caso receba um *exit signal* de `:exit`.

Figura 2 – Árvore de supervisão



Fonte: Autor

O módulo do Supervisor encapsula o mecanismo descrito acima. Por padrão, irá capturar a saída de todos seus processos filhos e não será encerrado caso algum destes encontre um erro inesperado. O Supervisor ainda pode ser encerrado por outras razões, como a queda total de um sistema, porém não pela falha de algum processo filho. Quando um Supervisor é encerrado, seus processos filhos também são. O comportamento mais frequente de um Supervisor, dado um erro fatal em um de seus processos filho, é comandar a reinicialização do processo terminado. Existem algumas políticas de reinicialização pré-definidas que o Supervisor pode configurar. São elas:

- *One-for-one*: quando um processo filho é terminado, apenas este processo é reinicializado. Na Figura 2, se o processo *A1* for terminado, ele será o único reinicializado.
- *One-for-all*: quando um processo filho é terminado, todos os processos filhos são terminados e reinicializados. Na Figura 2, se o processo *A1* for terminado, todos os filhos do Supervisor A, inclusive o Supervisor B, serão reinicializados.
- *Rest-for-one*: quando um processo filho é terminado, todos os processos que tenham sido inicializados após o processo terminado serão reinicializados. Na Figura 2, se o processo *B2* for terminado, todos os processos de *B2* até *Bn* serão reinicializados. O processo *B1* será o único não reinicializado.

2.1.3.2 Servidor genérico

O servidor genérico é um processo Erlang especializado em servir outros processos. Sua capacidade se restringe a armazenar um estado e oferecer maneiras

de consultar ou alterar este estado. O módulo OTP define uma série de *callbacks* que devem ser implementadas no código-fonte da aplicação, reduzindo drasticamente o *boilerplate* necessário para a configuração manual de um processo Erlang com a mesma capacidade. Dada a natureza de dados imutáveis do Erlang/Elixir, o conteúdo de um servidor genérico é mantido através de um laço de repetição recursivo que envia seu estado atual para a próxima iteração do *loop*. A vantagem do uso do servidor genérico é sua fácil configuração de chamadas assíncronas, o conteúdo exposto nestas respostas e o como será mantido o seu estado através das repetições do laço.

Requisições enviadas a um servidor genérico podem ser respondidas de forma síncrona ou assíncrona. Estas duas formas de interação são chamadas *calls* e *casts*, respectivamente. O valor retornado por ambos *call* e *cast* devem aderir a um formato pré-definido: uma tupla cujo primeiro elemento é um átomo, e o segundo elemento o conteúdo da resposta. Para requisições síncronas, o átomo da tupla deve ser `:reply`, enquanto requisições assíncronas devem retornar o átomo `:noreply` como primeiro elemento da tupla de resposta. Um servidor genérico também pode definir *timeouts* para o período de espera de uma requisição. No caso de não haver nenhuma requisição no período, o servidor genérico pode comandar seu próprio término.

2.1.4 Elixir

Elixir é uma linguagem de programação funcional que herda todas as propriedades vistas no Erlang: compilada, de tipagem forte e dinâmica, e executada na BEAM (JURIC, 2019). Dada a natureza críptica do código-fonte Erlang, o Elixir surge com o propósito de descomplicar a sintaxe ofuscada do Erlang, mantendo o acesso às abstrações e facilidades da máquina virtual subjacente descritos nas Seções 2.1.1 e 2.1.3. Uma documentação detalhada de sua API e seus artifícios pode ser consultada em (TEAM, 2022). Código-fonte Elixir também é intercambiável com código-fonte Erlang, dado que ambas linguagens compilam para *bytecode* Erlang. De modo geral, o Elixir diminui a barreira de entrada para o ecossistema Erlang. Sendo mais idiomático do que o Erlang, oferece mecanismos mais concisos e modernos, tornando o desenvolvimento de sistemas que executam na BEAM mais fácil e popular.

2.1.4.1 Structs

O Elixir, sendo uma linguagem funcional, não oferece suporte a objetos. No entanto, podemos definir mapas com chaves pré-definidas e, possivelmente, valores *default* destas chaves. Isto pode ser feito utilizando-se *structs*. Um *struct* leva o nome do módulo em que é definido, e pode ser instanciado como um mapa qualquer. A palavra-chave para definir um *struct* é `defstruct`, que recebe uma lista de átomos que define as chaves desta estrutura. No trecho abaixo, é declaro um *struct* `Example.Person`, cujas chaves são `first_name` e `last_name`. Em seguida, cria-se uma instância deste

struct. Depois, um acesso correto de uma de suas chaves. Logo após este acesso correto, demonstra-se um acesso errado do *struct*.

```
defmodule Example.Person do
    defstruct [:first_name, :last_name]
end

iex> person = %Example.Person{first_name: "John", last_name: "Doe"}
%Example.Person{first_name: "John", last_name: "Doe"}

iex> person.first_name
"John"

iex> person.age
** (KeyError) key :age not found in: %Example.Person{first_name: "John",
last_name: "Doe"}.
```

2.1.4.2 Notação de funções

Funções são mencionadas utilizando a seguinte notação: `Módulo.função/aridade`. Quando uma função não possui parâmetros, sua aridade na notação é 0 (zero). Por exemplo, o seguinte módulo, `Hello`, possui apenas uma função: `world`. Esta função será referenciada como `Hello.world/0`.

```
defmodule Hello do
    def world() do
        "Hello, world!"
    end
end
```

2.1.4.3 *Pattern-matching*

Na BEAM, e por consequência no Elixir, o operador = realiza um trabalho mais complexo do que apenas atribuição. Conhecido como operador *match*, este transforma a expressão inteira em uma equação, fazendo com que os valores do lado esquerdo do sinal de igualdade sejam combinados com os elementos ao lado direito do operador, constituindo um padrão. Ambos exemplos a seguir constituem válidas operações de *match*: `idade = 1` e `{"João", idade} = {"João", 25}`. No primeiro exemplo, a variável idade é correspondida com o valor 1. No segundo exemplo, a variável idade é atribuída o valor 25, visto que este foi o padrão estabelecido pelo operador. Em outras plataformas, esta operação é também chamada desestruturação de variáveis. No Elixir,

podemos aplicar esta técnica em assinaturas de funções.

```
defmodule Greeter do
    def greet(name, "it"), do "Ciao, #{name}!"
    def greet(name, "pt-br"), do "Oi, #{name}!"
    def greet(name, language), do: "#{name}, default hi!"
end
```

É possível estabelecer quais padrões de *match* os parâmetros de uma função devem se adequar para esta ser executada. Ao nível de desenvolvimento, isto reduz drasticamente a necessidade de controles de fluxo, como `if`s, para que o código-fonte saiba deduzir qual caminho de execução seguir.

2.1.4.4 Meta-programação

Um dos mais importantes benefícios do Elixir é sua capacidade de redução de código *boilerplate*. Além da sintaxe moderna em comparação ao Erlang, o Elixir oferece *macros* que tornam a escrita de código ainda mais fluída e simplificada. Inspirado em *macros* da linguagem Lisp (WINSTON; HORN, 1986), *macros* no Elixir atuam durante o tempo de compilação e alteram a *Abstract Syntax Tree* (AST), manipulando o trecho de código de entrada e produzindo um trecho de saída alternativo. Este artifício não-trivial é uma peça chave na sintaxe "açucarada" do Elixir e de muitas de suas funções nativas.

O exemplo a seguir retrata um módulo `Example.Logger`, cuja *macro* `__using__` pode ser declarada em outros módulos para que estes "injetem" em sua AST os conteúdos que residem dentro da *macro*. Esta injeção ocorrerá durante a execução do programa, e o módulo `Example.Notifier` irá adquirir a função `log/1` como se houvesse herdado-a. Em seguida, demonstra-se o resultado em um *shell*.

```
defmodule Example.Logger do
    def __using__ do
        quote do
            def log(msg) do
                IO.puts("#{__MODULE__}: #{msg}")
            end
        end
    end
end

defmodule Example.Notifier do
    use Example.Logger
end
```

```
iex> Example.Notifier.log("Hello, world!")
"Example.Notifier: Hello, world!"
```

A manipulação da árvore sintática possui ferramentas mais complexas, permitindo a construção de predicados mais refinados. O seguinte exemplo (SCHOENFELDER, 2018) demonstra a construção do bloco *while*, inexistente no Elixir/Erlang, mas comum em linguagens orientadas e objeto. Em seguida, um exemplo de sua invocação.

```
defmodule Example.Macro do
    defmacro while({:=,_[i|_]}=init, predicate, do: [{:->,_,_}] = body) do
      quote location: :keep, generated: true do
        whilef = fn whilef, acc ->
          unquote(i) = acc
          if unquote(predicate) do
            acc2 =
              case unquote(i) do
                unquote(body)
              end
            whilef.(whilef, acc2)
          else
            acc
          end
        end
        whilef.(whilef, _ = unquote(init))
      end
    end
end

iex> j = while n = 1, n < 5 do n ->
   >    n + 1
   > end
4
```

2.1.5 Desempenho

Quando comparamos o Elixir com outras ferramentas contemporâneas, podemos ver em detalhe como as decisões arquitetônicas da plataforma BEAM reverberam em ambos pontos positivos e negativos. Uma publicação feita em janeiro de 2020 (STRESSGRID, 2020) pela empresa Stressgrid, especializada em realizar teste de carga em servidores, conduziu uma série de testes com Elixir, Go e Node visando coletar algumas das métricas mais comuns de avaliação de desempenho: taxa de

transferência, utilização de *Central Processing Unit* (CPU) e número de conexões. Neste teste, o Elixir se comporta de maneira satisfatória, mantendo latência baixa durante a bateria de testes e conseguindo atender o máximo de conexões usadas no teste.

O ponto fraco notável do Elixir e da BEAM se torna evidente com muitos clientes simultâneos. O uso da CPU aumenta vertiginosamente, chegando próximo dos 95%, ao passo que os competidores se mantém entre 10% e 25%. Isto se deve ao funcionamento da máquina virtual e sua maneira de escalonar processos: por serem leves, baratos e fáceis de manejar, a máquina virtual se apodera de quantas *threads* precisar para orquestrar o processamento concorrente do Elixir. Ainda sim, apesar de o recurso ser apoderado pela máquina virtual, esta mantém o funcionamento de maneira constante e serve os clientes durante toda duração do teste.

2.2 TOLERÂNCIA A FALHAS E ESTRATÉGIAS DE REPLICAÇÃO

Falhas durante a execução de um sistema podem causar inconsistências ou interrupção do sistema. No caso das falhas causarem defeito, além de causar avarias e prejuízo financeiro (ROGERS, 2020), requerem alguma maneira para que o sistema volte a funcionar corretamente, seja outro programa responsável por sua reinicialização, ou um esforço humano para resolver a situação. A capacidade de um sistema em manter sua execução mesmo se um ou mais de seus componentes falhou é denominada tolerância a falhas. A busca por um sistema tolerante a falhas, portanto, visa construir um sistema de tal forma que falhas inesperadas durante a execução do programa não acarretem o desvio na execução normal do serviço (WILFREDO, 2000). Sistemas centralizados são particularmente vulneráveis neste aspecto, visto que são suscetíveis a um único ponto de falha e seu desligamento provoca uma queda total de serviço (BUDHIRAJA *et al.*, 1993). Um dos propósitos em se ter um sistema distribuído é dispersar o funcionamento do servidor em uma quantia de processos, máquinas ou redes de tal forma que problemas inesperados, como uma queda de energia, causem a falha de um ou poucos membros que compõe o sistema.

2.2.1 Replicação

Uma das maneiras de se tornar um sistema tolerante a falhas é criar réplicas deste sistema em servidores físicos diferentes, de forma que a falha de uma instância ocorra de forma isolada, sem comprometer o funcionamento das outras réplicas. Um servidor replicado adquire a capacidade de funcionar mesmo se um número limitado de réplicas estiver fora de serviço (BUDHIRAJA *et al.*, 1993). Para que as réplicas sejam utilizáveis em casos de falha, deve-se mantê-las atualizadas conforme o serviço tem seu estado alterado. A maneira com que se mantêm as réplicas atualizadas pode se dar de pelo menos duas formas: replicação passiva e replicação ativa. Na abordagem de replicação passiva, uma única réplica recebe requisições do cliente, propaga o novo estado para as demais réplicas, e então responde à requisição. Na replicação ativa, todas as réplicas exercem o mesmo papel, processando as requisições dos clientes em uma mesma ordem total.

2.2.2 Replicação Máquina de Estados

A replicação ativa, também é conhecida como RME (SCHNEIDER, 1990; LAMPORT, 1978). Nesta abordagem, o servidor é definido como uma máquina de estados, cujo estado é composto por variáveis que podem ser alteradas por *comandos*. Clientes interagem com o servidor através de comandos que leem ou modificam o estado do servidor (SCHNEIDER, 1990).

Um ponto central da RME é garantir que todas as réplicas partam de um mesmo estado inicial e, após processar comandos de clientes, realizam uma transição que as mantenham com um estado idêntico. Para aderir a esta característica, comandos devem seguir uma natureza *determinística*. Isto é, o estado futuro de qualquer réplica depende apenas do seu estado atual e da saída gerada pelo comando. Visto que comandos devem ser entregues de forma sequencial a todas as réplicas para estas atingirem o mesmo estado de forma consistente (HERLIHY; WING, 1990; GUERRAOUI; SCHIPER, 1996), o sistema deve recorrer a um artifício que garanta esta ordenação de comandos. Ao adotar a ordenação de entrega, o sistema passa a satisfazer as propriedades de *ordem* e *acordo* propostas em (SCHNEIDER, 1990). A ordenação da entrega de comandos pode ser alcançada através de duas formas:

- **Processos sequenciadores**: interceptador de comandos que implementa garantias de entrega antes de repassar a sequência ordenada de mensagens para as réplicas.
- **Protocolos de consenso**: algoritmos que acordam em uma mensagem a ser entregue a todas as réplicas com base em uma interação conjunta das mesmas.

Apesar da aplicabilidade de ambas as soluções, no presente trabalho optou-se por utilizar o protocolo de consenso Raft (ONGARO; OUSTERHOUT, 2014), tanto por ser uma alternativa menos complexa do que o protocolo Paxos (LAMPORT, 1998), quanto pelo fato de ter sido encontrada uma biblioteca de código aberto que implementa o Raft no ecossistema Erlang. Esta biblioteca é mantida por uma organização renomada (RABBITMQ, 2007), trazendo confiabilidade ao código.

2.2.3 Raft

Raft é um algoritmo de consenso utilizado para garantir a ordenação total de mensagens necessária para a implementação da estratégia de RME. Em resumo, este algoritmo resolve o problema que surge quando múltiplos servidores necessitam concordar em um único estado mesmo em face de falhas. O Raft opera elegendo uma instância líder em um dado *cluster*. Esta instância líder é responsável por aceitar requisições de clientes e gerenciar a integridade de estado dos servidores. Informação flui de forma unidirecional, vinda do líder para os servidores.

Projetado para ser compreensível, é equivalente ao algoritmo de consenso Paxos (LAMPORT, 1998) em termos de desempenho, porém decompõe sua imposição de consenso em três tópicos relativamente independentes que fundamentam sua execução: eleição do líder, replicação de *logs* e funcionamento seguro. O Raft trata o tempo como uma sequência de períodos chamados *terms*. Os *terms* iniciam com a eleição de um líder, e então partem para um período de votação dos membros a fim de determinar se há um consenso com base na mensagem adquirida de um cliente. Uma

série de propriedades definidas pelo algoritmo são respeitadas durante sua execução. Estas são:

- **Eleição segura**: no máximo um líder é eleito a cada *term*.
- **Líder *append-only***: um líder jamais sobrescreve ou deleta registros de seu *log*, apenas acrescenta entradas.
- **Correspondência de *logs***: duas entradas de dois *logs* com índices e *terms* idênticos garantem que todas as entradas de ambos *logs*, até estas duas, serão correspondentes.
- **Completude de líder**: se uma entrada de *log* é gravada em um dado *term*, esta entrada também estará presente nos *logs* dos líderes de todos os *terms* futuros.
- **Integridade de máquina de estados**: se um servidor gravou uma entrada em seu *log* com um certo índice, então nenhum outro servidor irá gravar outra entrada em seu *log* para aquele mesmo índice.

As instâncias que participam de um *cluster* no Raft podem assumir um de três papéis:

- **Candidato**: uma instância que pode ser eleita líder. Entra neste estado quando não recebe contato do Líder após um certo tempo. A instância Candidata pode ser eleita líder, ou retornar a ser Seguidor.
- **Seguidor**: participante passivo, responde chamadas *Remote Procedure Call* (RPC) do líder para lançar seu voto a respeito do consenso da atual mensagem. Pode responder de maneira afirmativa caso o estado do Candidato esteja apropriado em termos da corretude do consenso, ou responder negativamente caso haja inconsistência de *terms* em relação ao último comando processado ou inconsistência de índices. Também pode responder de forma positiva ao quórum e adicionar ou remover parte de seu *log* para se acomodar ao comando proposto.
- **Líder**: responsável central pela proposição de comandos aos outros servidores, com objetivo de atingir um consenso. Em caso de haver uma maioria de respostas afirmativas vindas dos Seguidores, transiciona o sistema para o novo estado

Pelo fato do presente projeto utilizar o Raft como um pacote de terceiro, seu uso das primitivas é restringido a apenas algumas invocações principais da biblioteca *Ra* (RABBITMQ, 2017). Ou seja, a biblioteca Pistis não se encarrega de realizar a eleição de líder, garantir a replicação dos *logs* ou impor o funcionamento seguro dos participantes do algoritmo. As funções utilizadas pelo presente projeto são detalhadas na Seção .

3 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos relacionados ao tema abordado nessa monografia. Em especial, foram selecionadas pesquisas que tratam de tolerância a falhas em Erlang e transparência na replicação de serviços. As seguinte seções abordam estes tópicos.

3.1 TOLERÂNCIA A FALHAS EM ELIXIR OU ERLANG

3.1.1 Implementação de Replicação em Cadeia em Erlang

Em (FRITCHIE, 2010), é demonstrado um estudo de caso da implementação do protocolo de Replicação em Cadeia em um armazenamento de chave-valor distribuído chamado Hibari[1]. Os autores destacam a dificuldade implícita em construir sistemas distribuídos, especialmente no que se refere à aplicação da teoria para implementações de sistemas comerciais, visando aumento da disponibilidade do serviço.

A estratégia de Replicação em Cadeia (VAN RENESSE; SCHNEIDER, 2004) (do inglês *Chain Replication*) organiza um conjunto de réplicas logicamente encadeadas e implementa um esquema *Mestre/Escravo*, de forma que o servidor *mestre* esteja no início da cadeia, seguido pelos servidores *escravos*.

O *mestre* é responsável por executar requisições e por propagar atualizações do seu estado ao resto da cadeia. Consistência forte (*i.e.*, *linearizabilidade*) é garantida desde que: (i) mudanças de estado sejam corretamente processadas pelo servidor *mestre* e propagadas ordenadamente respeitando a ordem da cadeia, e (ii) operações de leitura são processadas pela última instância da cadeia. Em caso de falha na propagação pela cadeia, a instância faltosa é removida da cadeia e a atualização de estado é enviada adiante, enquanto as instâncias adjacentes são conectadas uma a outra. Quando esta instância for reiniciada, ela será adicionada ao final da cadeia e terá o seu estado clonado da instância acima. Enquanto estiver em fase de recuperação, requisições de clientes serão respondidas pela instância acima.

O Hibari implementa replicação em cadeia utilizando primitivas de envio de mensagem assíncrona oferecidas pelo Erlang. Dentre elas, destaca-se o uso do `gen_server`, que facilita a composição de um nodo topo de cadeia (*mestre*) responsável por instanciar o próximo nodo da cadeia. Por utilizarem as primitivas do `gen_server`, estes nodos são automaticamente adicionados a uma árvore de supervisão, cujo propósito é isolar falhas, sem comprometer o resto do sistema.

Detecção de falhas de partição é alcançada com a inclusão de um processo independente por servidor responsável por monitorar *n* réplicas onde o Hibari estiver operando. Uma verificação periódica testa se transmissões de um nodo Erlang foram

[1] Hibari: `https://github.com/hibari/hibari`

capturadas por todas outras réplicas neste ambiente. Se uma dada réplica n_1 capturou o pacote enquanto n_2 não o fez, é levantada a possibilidade de uma partição estar ocorrendo em n_2. A medida remediadora é feita pelo mesmo processo que percebeu a falha, abortando a execução do servidor primário e notificando a aplicação através de um alarme. O servidor primário entra em um estado de bloqueio e a situação requer a resolução por um administrador humano.

O armazenamento distribuído chave-valor Hibari (FRITCHIE, 2010) compreende uma estratégia de replicação distinta do presente trabalho, mas usufrui de abstrações comuns oferecidas pela máquina virtual subjacente, como primitivas de envio e recebimento de mensagem, supervisão de processos e o servidor genérico `gen_server`. Os problemas e desafios relatados pelos autores se aplicam a todos ambientes de computação distribuída, e não a limitações da linguagem. Entretanto, a implementação de cadeia de réplicas, recuperação e detecção de falhas são responsabilidade do programador. Assim como em (FRITCHIE, 2010), pretendemos utilizar recursos nativos da linguagem para prover tolerância a falhas, mas a nossa proposta visa esconder do programador detalhes sobre estratégias de replicação, tornando o processo de desenvolvimento mais fácil e menos suscetível à erros.

3.1.2 Replicação Máquina de Estados em escala global

Em (ENES *et al.*, 2020), os autores apresentam um novo protocolo de RME, Atlas, feito para implementação de sistemas de larga escala. Destaca-se a capacidade de clientes de não depender de um único líder pré-definido, visando oferecer serviço independente de distância geográfica.

A principal necessidade por um novo protocolo para sistemas globais é a realidade moderna onde servidores são executados em diferentes locais do globo, de forma a oferecer alta disponibilidade e menor latência, visto que clientes conseguem acessar o servidor mais próximo deles. O problema visto em protocolos como Paxos (LAMPORT, 1998) e Raft (ONGARO; OUSTERHOUT, 2014) é a necessidade de um líder ou um quórum que determine a ordem dos comandos entregues às réplicas – em sistemas globais, a latência na comunicação os participantes na ordenação de mensagens distantes fisicamente deteriora a performance do grupo de réplicas.

Os autores identificam o nível de tolerância a falha oferecidos por protocolos RME comuns como sendo demasiados em sistemas distribuídos geograficamente. Os autores partem do pressuposto que desastres naturais que possam causar perda total do *data center* são raros, enquanto que é possível lidar com *downtime* planejados reconfigurando o site indisponível fora do sistema (LAMPORT, 1978; SHRAER *et al.*, 2012). Além disto, interrupções temporárias de um *data center* tem tipicamente uma duração curta (LIU *et al.*, 2016). Por este motivo, o número de falhas de sites concorrentes, em um sistema distribuído geograficamente, é baixo (CORBETT *et al.*,

2012).

A etapa de comparação de desempenho do projeto apresenta resultados satisfatórios: o protocolo EPaxos (MORARU *et al.*, 2013), similar ao Atlas por ser um protocolo *leaderless*, tem uma latência maior que o Atlas devido ao fato de não seguir as premissas fundamentais do Atlas que o confere maior resolução de conflitos na etapa de coordenação de comandos. Ao ser comparado com o protocolo FPaxos (HOWARD *et al.*, 2017), verificamos os mesmos resultados vistos anteriormente: menor latência dos serviços, em todas as etapas de teste com número de falhas e tamanho da carga crescente. O Atlas possui uma menor taxa de operaçoes que estes protocolos em cenários onde a taxa de conflito é muito grande (>90%).

Conclui-se com a observaçao do fato do protocolo Atlas ser pioneiro no quesito de ser *leaderless* e parametrizado pelo número de falhas toleradas concorrentemente. Visto que também é feito para específicos cenários de sistemas distribuídos em larga escala pelo planeta, a atual resiliência dos data centers permite um alívio nas premissas de tolerância a falha seguidas por outros protocolos de consenso. Garantindo uma redução de latência para clientes, mantendo a estabilidade e desempenho do sistema em caso de expansão da rede de servidores e ultrapassando os protocolos competidores, a simples premissa fundamental do Atlas garante uma gigantesca disponibilidade para cenários cada vez mais comuns onde servidores se espalham pelo globo. No presente trabalho, o foco estará na configuração da estratégia de RME, com muito menos enfoque nas peculiaridades do protocolo de consenso subjacente.

3.1.3 Análise de aplicações Erlang tolerantes a falhas

Em (NYSTROM, 2009), os autores enumeram as ferramentas tolerantes a falhas oferecidas pelo Erlang e como podem ser colocadas em prática na construção de sistemas para então concluir com a construção de um identificador destas primitivas com base na semântica do projeto. Iniciando com uma extensiva recapitulação dos básicos da linguagem, os autores ressaltam o modelo de falha adotado pelas primitivas do Erlang, o *fail-stop*, onde partes faltosas de um sistemas tem sua execução interrompida e esta mesma falha é visível pelo resto do sistema sem comprometê-lo.

Com este princípio em mente, partem para a revisão das ferramentas mais comuns na construçao de mecanismos tolerantes a falha na plataforma Erlang: o servidor genérico (`gen_server`) e o supervisor.

Os autores descrevem, na seção seguinte, exemplos de árvores de supervisão onde é possível haver outros problemas que não seja um estado faltoso sendo propagado para o resto do sistema. Nestes exemplos, o fator comum entre eles é o prolongamento da árvore de supervisão sem devidos supervisores como nodos processos filhos. De forma geral, caso um processo trabalhador instancie outros processos, é importante garantir que o erro de seus filhos propague através dele até o nodo su-

pervisor. Processos trabalhadores incapazes de propagar seu erro ou de processos filhos adiante na árvore tendem a causar falhas gravíssimas na árvore de supervisão.

Enumera-se em seguida uma série de convenções de design vitais para construção de um sistema propriamente isolado e tolerante a falha. De forma resumida, estas convenções procuram garantir que processos trabalhadores sejam ou monitorados por um supervisor, ou propaguem sua falha até um supervisor, sem pausas no meio. Processos supervisores intermediários devem garantir uma um número de tentativas de reinicialização igual a zero a fim de reduzir o número máximo de falhas. Supervisores do topo da árvore de supervisão devem possuir um tempo limite de terminação de filhos infinito, de forma que o atraso de qualquer processo filho não interfira com a reinicialização comandada pelo topo da árvore de supervisão.

O trabalho em questão apresenta uma ferramenta extratora e analisadora da árvore de supervisão de uma aplicação com base na sua árvore semântica abstrata (*AST*). Após uma extensa explicação sobre seu funcionamento interno, são apresentados os resultados dos experimentos, demonstrando que a ferramenta é capaz de analisar com alto grau de acerto a qualidade da árvore de supervisão extraída. A ferramenta perde bastante de sua acurácia quando utilizada em projetos em fase inicial do desenvolvimento pelo fato do código-fonte ser ainda muito breve e, muitas vezes, incompleto.

Apesar de compartilhar da plataforma Erlang e dos componentes OTP, a proposta do nosso trabalho é oferecer um componente facilmente configurável que:

- Atua durante a execução de um sistema, ao invés do tempo de compilação.
- Confere alta disponibilidade e tolerância a falhas através do emprego da estratégia de RME, ao invés de analisar sua própria árvore de supervisão internamente.

3.2 ESTRATÉGIAS PARA REPLICAÇÃO TRANSPARENTE

3.2.1 Biblioteca para replicação transparente de serviços

Em (PEREIRA *et al.*, 2019), propõe-se uma solução transparente para a configuração e gerenciamento de réplicas seguindo a estratégia de replicação de máquina de estados. Através uma API enxuta, baseada em anotações, o desenvolvimento da aplicação pode se concentrar na lógica da aplicação ao invés de preocupar-se com detalhes relacionados ao protocolo de consenso ou gerência das réplicas.

A biblioteca em questão atinge o objetivo norteada por alguns pontos chave:

- *Desacoplamento entre protocolo de consenso e servidor*: A biblioteca separa a aplicação do módulo responsável por realizar a entrega do consenso. Este aspecto é o que confere ao desenvolvedor a capacidade de não se preocupar com a

configuração de protocolo de consenso, coordenação de réplicas ou aprendizado sobre o protocolo de consenso.

- *Compartilhamento de protocolo de consenso*: Visando otimizar o consumo de recursos pelo componente responsável pelo protocolo de consenso, apenas uma instância se responsabiliza pela ordenação de requisições externas e o redirecionamento para as réplicas alvo. Essa decisão de projeto permite reduzir os recursos que seriam gastos pelo emprego de múltiplos protocolos de consenso.
- *Registro de serviços em tempo de execução*: Serviços só podem ser conectados ou desconectados do sistema através de dois comandos com a respectiva função. Ao invocar qualquer um deste comandos, a biblioteca se responsabiliza pela remoção ou adição do serviço às replicas.

A biblioteca é estruturada de tal forma que requisições de clientes são processadas por um módulo único. Este mesmo módulo, capaz de abstrair os detalhes do protocolo de consenso, também é responsável por despachar estas requisições para as réplicas. Outro componente se especializa em ordenar a criação de mais réplicas se necessário. A criação de réplicas é feita através de reflexão do código fonte e também é executada por um componente separado.

A API exposta pela biblioteca fornece um conjunto de anotações que servem para identificar comandos RME. Sendo anotações, estas podem ser adicionadas ao sistema apenas durante o desenvolvimento da aplicação e não durante a execução. Além destas anotações, uma série de classes abstratas definem o comportamento geral dos componentes personalizáveis da biblioteca, cabendo ao desenvolvedor providenciar especializações que implementem a solução desejada.

Testes da biblioteca demonstraram que, para cargas de trabalho menores, a diferença de performance entre aplicações que usam e aplicações que não usam a biblioteca é imperceptível. Dado que a carga enviada aos servidores cresce, serviços que usam da biblioteca limitam a vazão no processamento de requisições. Isto é esperado, dado a sobrecarga gerada pela minimização do esforço do desenvolvedor em prol do fácil uso. Além disto, foi identificado que infraestruturas onde nodos computacionais puderam compartilhar recursos tiveram uma melhor distribuição de carga e a vazão cumulativa para diferentes aplicações usando a biblioteca torna-se atrativa.

3.2.2 Simplificação de Replicação Máquina de Estados

Em (DENIZ ALTINBÜKEN, 2012), é apresentado o OpenReplica, um serviço baseado em Paxos com objetivo de prover suporte para definição, configuração e sincronização de réplicas em sistemas distribuídos.

O OpenReplica toma responsabilidade pela criação e manutenção de instâncias de serviços de coordenação ou réplicas no sistema em que forinserido. De natureza

orientada a objetos, o OpenReplica opera com máquinas de estado definidas pelo desenvolvedor para transformar-las em máquinas de estados replicadas tolerantes a falha uma vez em execução. Oferecendo *proxies* dos objetos replicados, membros do sistema conseguem interagir com o mesmo através de uma API única como se fossem objetos locais.

De forma a aproveitar disponibilidade e flexibilidade o máximo possível, o OpenReplica oferece mudanças dinâmicas das réplicas ou de *acceptors* durante tempo de execução. Clientes interagindo com um serviço adotando o OpenReplica podem integrar infraestruturas existentes através de *name servers* implementados pelo OpenReplica de forma a direcionar requisições diretamente às réplicas.

O OpenReplica também é capaz de agrupar réplicas e nós coordenadores com base em sua localização física, sendo capaz de reduzir números de falhas catastróficas de forma significativa através de estratégias ambiciosas de posicionamento das réplicas.

Os resultados mostrados na fase de teste do OpenReplica são satisfatórios e evidenciam a robustez da biblioteca. Utilizando as métricas do uso do ZooKeeper (HUNT *et al.*, 2010), a comparação de latência, espaço de memória e número de réplicas instanciadas frente a carga, ambos serviços são equivalentes em seus resultados. A taxa de transferência do sistema que emprega o OpenReplica é severamente inferior ao do ZooKeeper. É necessário ter em mente, no entanto, que o OpenReplica tem seu código-fonte gravado em Python, enquanto que o ZooKeeper, gravado majoritariamente em Java, C e C++, é também otimizado para processar lotes de operação de cada vez. Este detalhe de otimização é deixado de lado no OpenReplica visto que seu objetivo é outro ao do ZooKeeper.

No presente trabalho, não se planeja aprofundar a configuração a nível de redes tal qual o OpenReplica, da mesma foram que as *proxies* de objetos também serão descartadas do projeto.

3.2.3 Replicação Transparente através de meta-programação

Em (UGLIARA *et al.*, 2017), os autores utilizam capacidades nativas da linguagem de programação Cyan para tornar enxuta a construção de réplicas operacionais para um dado sistema. Os autores partem do princípio que *frameworks* famosos de replicação, apesar de facilitar a comunicação e ordenação de comandos entre réplicas, carecem da capacidade de equipar o desenvolvedor com maneiras enxutas de definir métodos de transição de estado. Para resolver este problema, a linguagem Cyan oferece recursos de meta-programação que facilitam a declaração de transições de estado bem definidas. Para demonstrar o uso, os autores integram estas premissas com a biblioteca Treplica (VIEIRA; BUZATO, 2008), usada para construir uma estrutura de replicação ativa para aplicações distribuídas.

O recurso central da linguagem Cyan empregado pelos autores são meta-objetos, declarações estáticas no código-fonte Cyan substituídos por implementações concretas Java durante a execução. Transições de estado emitidas para réplicas de um dado sistema são expressas por meta-objetos cunhados de ações. Estas ações têm um comportamento similar a uma classe em linguagens orientadas a objetos, onde herdam definições de classes utilizadas na estratégia de replicação em questão. No caso dos autores, estes meta-objetos herdam de componentes expostos pela biblioteca Treplica. Após definir um meta-objeto, métodos deste mesmo componente devem receber anotações para que o compilador Cyan substitua este método com uma classe concreta em tempo de execução. As primitivas de transparência oferecidas pela linguagem Cyan demandam um entendimento de organização de pacotes e da própria linguagem para que funcionem de maneira adequada. Apesar de cumprir seu propósito, a configuração da replicação foca apenas nas transições de estado por utilizar outra ferramenta — Treplica (VIEIRA; BUZATO, 2008) — que auxilie nesta abstração. No presente trabalho, planeja-se reduzir grandemente a necessidade de entender a fundo a plataforma ou a linguagem para que seja possível configurar a estratégia de RME.

3.3 SUMÁRIO DOS TRABALHOS RELACIONADOS

Tabela 1 – Pontos principais dos trabalhos relacionados

Trabalho	**Estratégia**	**Transparência?**	**Linguagem**
(FRITCHIE, 2010)	Replicação em cadeia	Não	Erlang
(ENES *et al.*, 2020)	RME	Não	Nenhuma
(NYSTROM, 2009)	Nenhuma	Não	Erlang
(PEREIRA *et al.*, 2019)	RME	Sim	Java
(DENIZ ALTINBÜKEN, 2012)	RME	Sim	Python
(UGLIARA *et al.*, 2017)	RME	Sim	Cyan (Java)
(KRAUSE, 2022)	RME	Sim	Elixir

Fonte: Autor

Os trabalhos coletados neste capítulo compartilham os conceitos de tolerância a falhas, transparência e a plataforma BEAM com o presente projeto. Apesar de a estratégia RME estar presente em quatro (PEREIRA *et al.*, 2019; DENIZ ALTINBÜKEN, 2012; UGLIARA *et al.*, 2017; ENES *et al.*, 2020) destes trabalhos, nenhum implementa a mesma utilizando a BEAM. De forma análoga, os projetos que utilizam a BEAM ou utilizam outra estratégia de replicação (FRITCHIE, 2010) ou (NYSTROM, 2009) focam em conceitos basilares de tolerância a falhas, sem se preocupar com a implementação de uma estratégia visando tolerar falhas em tempo de execução. Este trabalho foca na intersecção entre estes trabalhos: uma solução transparente para o ecossistema Erlang/Elixir, com foco na abstração da gerência das réplicas distribuídas e o consenso das mensagens enviadas a elas.

4 BIBLIOTECA

Este capítulo compreende a biblioteca desenvolvida para o presente projeto. Aborda as decisões de projeto tomadas em sua construção, uma visão geral de sua organização e funcionamento interno, e por fim a API para ser possível utilizá-la.

4.1 VISÃO GERAL

4.1.1 Nomenclatura

Na mitologia grega, Pistis é a divindade que personifica boa-fé, confiança e confiabilidade (GRIMALDI, 1957). Com estas qualidades em mente, o projeto foi batizado de Pistis, dado que os atributos desta figura são também qualidades buscadas em um sistema tolerante a falhas.

4.1.2 Propósito

Como visto no Capítulo 1, a construção de um *cluster* que executa na BEAM é uma tarefa desafiadora por si só. Somado ao desafio de implementar um protocolo de consenso e moldar o sistema para que este concretize a estratégia de RME, o projeto lida com diversos problemas para que estes elementos sejam regidos em harmonia. A biblioteca Pistis replica um servidor em réplicas distribuídas facilmente configuráveis, fortemente consistentes e tolerantes a falha que aderem à estratégia de RME. Mensagens de clientes tem o consenso garantido pelas réplicas através do protocolo Raft (ONGARO; OUSTERHOUT, 2014), cuja implementação também é abstraída do usuário final.

Foi escolhido o protocolo de consenso Raft por ser uma alternativa menos complexa que o protocolo Paxos. Além disso, uma biblioteca de código aberto mantida pela organização RabbitMQ (RABBITMQ, 2017) fornece as funções basilares do protocolo Raft para o ecossistema Erlang, tornando-a compatível com o código-fonte do projeto. Por abstrair tanto os detalhes do protocolo de consenso quanto as primitivas distribuídas do Erlang, o uso da biblioteca Pistis não demanda conhecimento prévio sobre RME ou Raft, requerendo do usuário final apenas a definição da máquina de estados.

4.2 ARQUITETURA

A aplicação Pistis possui um supervisor central dinâmico e diversos módulos especializados que passam a adicionar processos filhos a este supervisor central dinâmico. Esta dinamicidade é importante dado que a biblioteca pode demandar a

supervisão de processos adicionais caso haja mais réplicas do que o esperado e/ou falhas ocasionais demandem a supervisão inesperada de outros processos.

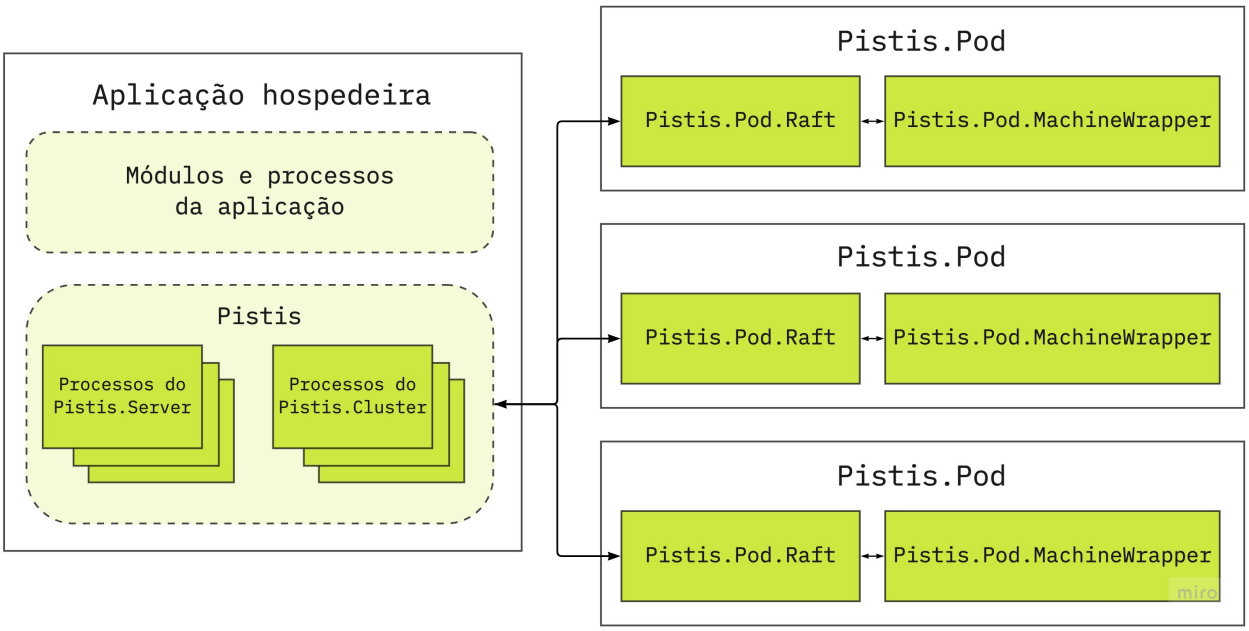
Um princípio norteador no desenvolvimento da biblioteca é a busca pela transparência da biblioteca. Para isso, deve demandar o mínimo de configuração possível, assim como esconder do usuário de que maneira as réplicas são geridas e orquestradas, ou como o protocolo de consenso é assegurado. Os seguintes contextos definem a biblioteca:

1. `Pistis.Server`: define a primitiva do repasse de comandos vindos do cliente e o mecanismo de processamento de requisições.
2. `Pistis.Pod`: define a estrutura que cerca a máquina de estados e sua interação com o protocolo de consenso Raft.
3. `Pistis.Cluster`: define os mecanismos de criação e manutenção do *cluster* distribuído da biblioteca.
4. `Pistis.Machine`: define o protocolo de uma máquina de estado e seu comportamento. Este componente, que corresponde a lógica da aplicação, deve ser implementado pelo usuário.

4.2.1 Elementos da máquina virtual

Os módulos da biblioteca são traduzidos para processos durante tempo de execução, e estes se encontram na instância BEAM da aplicação do usuário. Para as réplicas, são designadas novas instâncias distribuídas da máquina virtual. Dentro destas, o componente Raft vive em forma de um processo, cuja responsabilidade é se comunicar com outras réplicas, para atingir um consenso, quanto repassar os comandos para o invólucro da máquina de estados. A Figura 3, a seguir, retrata o mapeamento dos elementos da máquina virtual (processos e nodos) para o contexto da biblioteca.

Figura 3 – Nodos e processos no contexto da biblioteca

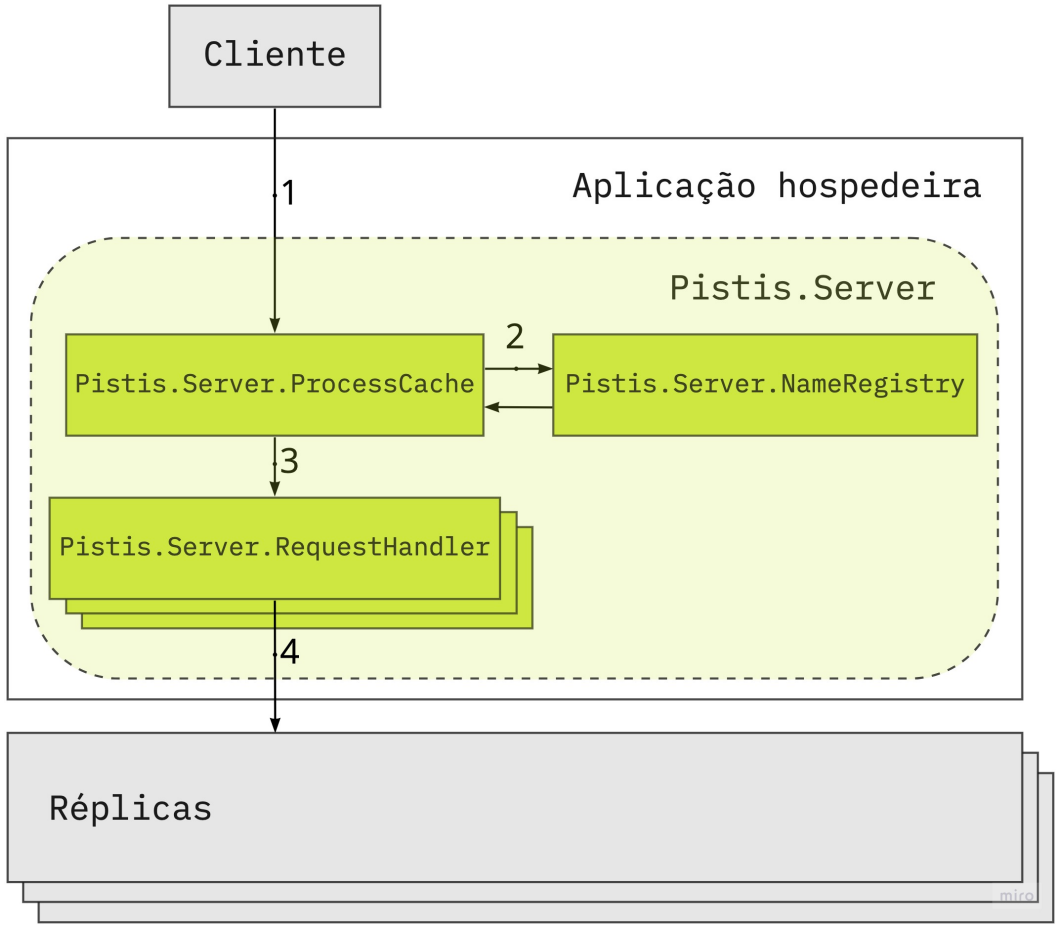


Fonte: Autor

4.2.2 Interação cliente-servidor

A biblioteca Pistis se posiciona como um interceptador entre cliente e servidor. Para isto, o usuário da biblioteca deve garantir que comandos do cliente sejam repassados ao servidor através da primitiva `Pistis.Server.send_request/1`. Desta forma, a biblioteca pode realizar o consenso da mensagem e atualizar o estado das máquinas de forma determinística. A função `Pistis.Server.send_request/1` aceita um parâmetro, que será a mensagem do cliente. Este parâmetro será envelopado em uma estrutura formal e enviado para ser processado pelo *cluster* distribuído. O retorno desta função é uma resposta consentida por todas as réplicas do *cluster*. O funcionamento interno do módulo `Pistis.Server` é ilustrado na Figura 4:

Figura 4 – Interação cliente-servidor



Fonte: Autor

A biblioteca processa requisições de clientes com um mecanismo de servidores genéricos sob demanda, garantindo que aumentos súbitos no número de requisições serão atendidos pela biblioteca. Ao receber um comando vindo de um cliente, a biblioteca procura um processo atendedor de requisição, o `Pistis.Server.RequestHandler`, para lidar com esta nova requisição. Caso não haja nenhum disponível no *cache*, uma nova instância de um `Pistis.Server.RequestHandler` é criada. Estes processos atendedores de requisição são encerrados caso não recebem a uma requisição em até 1 (um) minuto, garantindo que a *pool* de processos não irá crescer demasiadamente. O módulo `Pistis.Server.RequestHandler` é responsável por enviar o comando para ser processado pela camada de consenso do Raft.

4.2.3 Implementação do protocolo de consenso

O protocolo de consenso Raft foi implementado no presente projeto utilizando-se a biblioteca Erlang de código aberto Ra (RABBITMQ, 2017). A biblioteca Ra traz funcionalidades prontas consigo, como: eleição de líder, replicação de *logs* e compactação de *logs*, mas a biblioteca Pistis oculta a necessidade do usuário final invocar

estas primitivas. Apesar de ser um artefato puro Erlang, a compatibilidade com o Elixir permite o uso direto da biblioteca no código fonte do Pistis. As funções da biblioteca Ra em uso pelo Pistis são:

- `:ra.start/0`: comando invocado para cada uma das réplicas iniciar uma instância do Raft localmente.
- `:ra.process_command/2`: forma central para o processamento de comandos vindos do cliente. Parâmetros desta função são o endereço BEAM de alguma réplica e a requisição, respectivamente.
- `:ra.members/1`: maneira de visualizar os membros Raft atualmente conectados. Parâmetro da função é uma tupla contendo o nome do *cluster* e o endereço de um membro qualquer.
- `:ra.start_cluster/4`: função utilizada para a criação do *cluster* Raft. Recebe como parâmetro um átomo que dita o tipo do *cluster*, o nome do *cluster*, o módulo da máquina de estados e uma lista de endereços BEAM.
- `:ra.add_member/2`: forma dinâmica para se adicionar um novo membro ao *cluster* Raft. Parâmetros são a atual instância líder do *cluster* Raft e o endereço BEAM do novo membro a ser adicionado.
- `:ra.start_server/5`: maneira de tornar um novo membro Raft ciente do atual líder e vice-versa para então integrar o novo membro ao esquema do protocolo de consenso.

4.2.4 Gerência do *cluster*

A gerência das réplicas BEAM é regida pelo módulo `Pistis.Cluster.Manager`, que se responsabiliza pela instanciação e conexão das réplicas. O estado do *cluster* é mantido utilizando-se dois módulos complementares: `Pistis.Cluster.State` e `Pistis.Cluster.StateStorage`. O módulo `Pistis.Cluster.StateStorage` é um servidor genérico responsável por expôr capacidades simples de leitura e escrita de seu estado interno: uma instância do *struct* `Pistis.Cluster.State`. Este, por sua vez, armazena o líder Raft, todos os membros do *cluster*, e quaisquer instâncias que falharam em se conectar ao *cluster*. Seu detalhamento interno é visto a seguir:

```
defmodule Pistis.Cluster.State do
  defstruct [:leader, :members, :failures]

  @type raft_member :: tuple()
  @type t :: %__MODULE__{
    leader: raft_member(),
```

```
    members: list(raft_member()),
    failures: list(raft_member()),
  }
end
```

Um servidor genérico, expresso no módulo `Pistis.Cluster.ConnectionRetrier`, é executado a cada 3 (três) segundos e verifica se existem quaisquer membros que falharam em se conectar ao *cluster* em um primeiro momento. Caso hajam quaisquer destes, este processo inicia uma tentativa de adicionar cada uma destas instâncias falhas de maneira dinâmica ao *cluster*, visando reestabelecer a conexão total de todas as réplicas. Caso uma nova instância seja criada e adicionada ao *cluster*, o estado das réplicas é transmitido para a nova instância, garantindo que esta terá um estado equivalente ao resto do serviço.

De forma análoga ao módulo `Pistis.Cluster.ConnectionRetrier`, o módulo `Pistis.Cluster.StateRefresher` é um servidor genérico que atualiza o estado do *cluster* a cada 2 (dois) segundos, através da compração entre o estado do *cluster* Raft e o que está armazenado no módulo `Pistis.Cluster.StateStorage`.

4.2.5 Definição da máquina de estados

Para que a biblioteca funcione, a peça fundamental a cargo do usuário é a definição da máquina de estados que será replicada. Utiliza-se a anotação `@behaviour`, sendo a maneira Elixir de se definir um protocolo, equivalente a uma `interface` em Java ou C#. O protocolo em questão vive no módulo `Pistis.Machine`, demonstrado a seguir:

```
defmodule Pistis.Machine do
  alias Pistis.Machine.{Request, Response}
  @type machine_state :: any()
  @callback initial_state() :: machine_state()
  @callback process_command(Request.t(), machine_state()) :: Response.t()
end
```

As anotações `@callback` definem assinaturas de funções que devem ser implementadas pelos módulos que concretizem este protocolo. Sendo um módulo enxuto, delega ao desenvolvedor a implementação de duas funções:

- `Pistis.Machine.initial_state/0`: deve retornar o estado inicial da máquina de estados. Isto pode ser um dicionário, uma lista, um número inteiro, ou qualquer forma de representação de estado que expresse o estado inicial da aplicação.
- `Pistis.Machine.process_command/2`: deve ser implementada para todos tipos de mensagem que podem ser recebidas de um cliente. O primeiro parâmetro

desta função será um *struct* `Pistis.Machine.Request`, enquanto o segundo parâmetro será o estado atual da máquina de estados. O retorno desta função deve ser um *struct* `Pistis.Machine.Response`.

4.2.5.1 Pistis.Machine.Request e Pistis.Machine.Response

Em busca de uma definição clara e tipagem forte, ambas requisição de cliente e resposta de servidor foram expressas em dois *structs* distintos. Para armazenar a requisição de um cliente, o struct `Pistis.Machine.Request` possui apenas um campo: `body`. `Pistis.Machine.Response`, por outro lado, possui dois campos, `response` e `state`, que armazenam a resposta do servidor e o estado atual das máquinas, respectivamente. O trecho abaixo inspeciona ambas estas estruturas:

```
defmodule Pistis.Machine.Request do
  defstruct [:body]
  @type t() :: %__MODULE__{body: any()}
end

defmodule Pistis.Machine.Response do
  defstruct [:response, :state]
  @type t() :: %__MODULE__{response: any(), state: any()}
end
```

4.2.5.2 Exemplos de máquinas de estado

Abaixo estão listados exemplos de estruturas de dados simples que implementam o protocolo `Pistis.Machine`. Por conta disso, são compatíveis com a biblioteca.

```
defmodule Example.KVStore do
  alias Pistis.Machine.{Request, Response}
  @behaviour Pistis.Machine

  @impl Pistis.Machine
  def initial_state(), do: %{}

  @impl Pistis.Machine
  def process_command(%Request{body: {:get, key}}, current_state) do
    %Response{response: Map.get(current_state, key), state: current_state}
  end

  def process_command(%Request{body: {:put, key, value}}, current_state) do
    %Response{response: :ok, state: Map.put(current_state, key, value)}
```

```
  end
end
```

O trecho acima ilustra uma máquina de estados que opera como um mapa. Seu estado inicial é um mapa vazio e os dois comandos que interagem com esta máquina são uma leitura e uma escrita. A tupla que identifica o comando do escrita carrega uma variável `key`, usada para realizar a leitura desta chave no estado da máquina. A tupla que captura o comando de escrita contém as variáveis `key` e `value`, que representam a chave e o valor que serão inseridos no mapa. A resposta para o comando de leitura é o valor correspondente àquela chave no mapa, enquanto a resposta para o comando de escrita é um átomo `:ok`.

```
defmodule Example.Stack do
  alias Pistis.Machine.{Request, Response}
  @behaviour Pistis.Machine

  @impl Pistis.Machine
  def initial_state(), do: []

  @impl Pistis.Machine
  def process_command(%Request{body: {:push, item}}, state) do
    %Response{response: :ok, state: [item | state]}
  end

  def process_command(%Request{body: {:pop}}, []) do
    %Response{response: :empty_list, state: []}
  end

  def process_command(%Request{body: {:pop}}, state) do
    [head | tail] = state
    %Response{response: head, state: tail}
  end
end
```

O trecho acima ilustra uma máquina de estados que opera como uma pilha. Seu estado inicial é uma lista vazia, e os dois comandos que esta máquina de estados lida com são os de inserção e remoção, expressos por `:push` e `:pop`, respectivamente. Para o comando `:pop`, é realizado um *pattern-matching* na assinatura da função de forma que não haja tentativa de remoção de itens da lista caso essa estiver vazia. Caso a lista não esteja vazia, o comando executa normalmente. A resposta para o comando de inserção é um átomo `:ok`, enquanto a resposta para o comando de remoção é o item do topo da pilha, ou um átomo `:empty_list` caso a lista estiver vazia.

4.2.6 Configurações

A biblioteca Pistis conta com quatro variáveis configuráveis, três destas opcionais. Listadas abaixo:

- `machine`: armazena o módulo alvo para a replicação.
- `cluster_size` (opcional): um número inteiro positivo maior que 3 (três). Dita quantas réplicas serão instanciadas e/ou conectadas pela biblioteca. O valor padrão é 3 (três).
- `cluster_boot_delay` (opcional): um número inteiro positivo maior que zero. Dita os milissegundos de tempo de espera durante o tempo de subida do *cluster*. A redução deste tempo pode fazer com que réplicas falhem em se conectar ao *cluster* durante o primeiro *boot* do sistema, porém um tempo de espera muito grande faz com que a aplicação demore demais para estar disponível. O valor padrão é 4000 (quatro mil).
- `known_hosts` (opcional): uma lista de átomos que representam endereços de instâncias BEAM para serem conectados. O valor padrão é uma lista vazia.

Um exemplo de arquivo de configuração:

```
import Config

config :pistis, machine: Example.KVStore
config :pistis, cluster_size: 4
config :pistis, cluster_boot_delay: 4500
config :pistis, known_hosts: [:"foo@10.10.1.2", :"bar@10.10.1.3"]
```

Seguindo este exemplo, a biblioteca utilizará o módulo `Example.KVStore` como máquina de estados. O tempo de espera do servidor para inicializar a instância Raft em cada uma das máquinas será de 4,5 segundos. A aplicação irá se conectar às instâncias BEAM `foo` e `bar`, que executam nos *Internet Protocol* (IP)s `10.238.82.82` e `10.238.82.85`, respectivamente, e fará destas instâncias réplicas do servidor. A aplicação irá instanciar duas instâncias BEAM, visto que o tamanho desejado do *cluster* é maior que os *hosts* conhecidos. Haverão quatro réplicas no total.

5 AVALIAÇÃO

Neste capítulo são avaliados os Objetivos 2 e 3 propostos no início do projeto. O primeiro aspecto a ser avaliado é a usabilidade geral da biblioteca Pistis, no que concerne a complexidade e tempo gasto para adicionar a biblioteca a uma aplicação existente. O segundo aspecto é a sobrecarga causada na aplicação que utiliza a biblioteca em comparação com outras aplicações que não a utilizam.

5.1 DESCRIÇÃO DE USO

Visando disponibilizar o projeto em forma de código aberto, a biblioteca Pistis está disponível na plataforma Hex[1], o gerenciador de pacotes mais popular do ecossistema Erlang. A plataforma Hex auxilia na hospedagem de pacotes de código aberto, incluindo o seu versionamento, documentação e licenciamento. O pacote[2] da biblioteca pode ser acessado e baixado sem necessidade de autenticaçao da plataforma.

5.1.1 Instalação

Para instalar, testar, compilar ou distribuir uma aplicação Elixir, utilizamos a ferramenta Mix, embutida na instalação da linguagem. Esta ferramenta é utilizada através da linha de comando e facilita a execução de tarefas rotineiras e/ou relacionadas ao processo de implantação de forma geral. Para instalarmos qualquer biblioteca, deve-se adicionar uma tupla contendo o nome registrado na plataforma Hex e a versão desejada do artefato no arquivo `mix.exs`, presente em qualquer aplicação Elixir. O exemplo a seguir ilustra o arquivo `mix.exs` gerado automaticamente junto da aplicação `Example`. Adicionamos a tupla do artefato à lista retornada pela função `deps`.

```
defmodule Example.MixProject do
  use Mix.Project

  def project do
    [
      app: :example,
      version: "0.1.0",
      elixir: "~> 1.12",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end
```

[1] https://hex.pm/
[2] https://hex.pm/packages/pistis

```
  def application do
    [
      extra_applications: [:logger],
      mod: {Example.Application, []}
    ]
  end

  defp deps do
    [
      {:pistis, "~> 0.1.10"}
    ]
  end
end
```

Em seguida, devemos atualizar o projeto com a recém-adicionada dependência e compilá-lo. Para isto, deve-se estar no diretório do projeto e então executar:

- `$ mix deps.get`, para baixar a dependência
- `$ mix compile`, para compilar o projeto

5.1.2 Configuração

Com a biblioteca instalada como dependência do projeto, deve-se criar ou atualizar o arquivo `config/config.exs` e definir ao menos o módulo da máquina de estados. Outras variáveis de configuração podem ser definidas neste arquivo, como ilustra a Seção 4.2.6. Com as variáveis de configuração estabelecidas, o último passo é adicionar o módulo `Pistis.Core.Entrypoint` a lista de processos supervisionadas da aplicação alvo, como ilustrado no exemplo a seguir:

```
defmodule Example.Application do
  @moduledoc false
  use Application

  @impl true
  def start(_type, _args) dor
    children = [Pistis.Core.Entrypoint]
    opts = [strategy: :one_for_one, name: Example.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

5.1.3 Análise da usabilidade

A biblioteca pode ser instalada como qualquer dependência do ecossistema Elixir/Erlang, e sua adição à aplicação se dá de maneira comum quando comparada com outras bibliotecas do ecossistema. As configurações do sistema são em sua maioria opcionais, demandando do usuário apenas a implementação da máquina de estados desejada. Esta, por sua vez, pode ser implementada com os artifícios mais básicos da linguagem Elixir, sem demandar linhas de código demasiadas, conhecimento específico sobre o protocolo de consenso ou prática com o ambiente distribuído da BEAM.

5.2 DESEMPENHO

Para avaliar o custo da biblioteca Pistis, foram executados testes de desempenho com duas outras aplicações Elixir na procura de comparar os resultados obtidos em cada uma das soluções.

5.2.1 Ambiente de teste

Para executar os testes em um ambiente capaz de suportar cargas significativas, as máquinas geradoras de carga e hospedeiras da aplicação criadas e utilizadas através da plataforma Emulab (WHITE, 2002).

5.2.1.1 Especificações da máquina

As instâncias que hospedaram a aplicação e suas réplicas foram do tipo `d430`. As especificações estão listadas abaixo:

- **Arquitetura**: x86_64
- **Número de *cores***: 8
- **Threads** por CPU: 2
- **Memória *RAM***: 4096MB
- **Sistema Operacional**: Ubuntu 20.04

5.2.1.2 Cenários de teste

Diferentes pareamentos de número de clientes e número de réplicas foram utilizados para se obter métricas o mais abrangentes e realistas quanto possível. Cada uma das configurações de número de clientes / número de réplicas por servidor está listado a seguir:

Tabela 2 – Cenários de teste no Emulab

	3 réplicas	6 réplicas	9 réplicas
2 clientes	`2c3r`	`2c6r`	`2c9r`
4 clientes	`4c3r`	`4c6r`	`4c9r`
8 clientes	`8c3r`	`8c6r`	`8c9r`
16 clientes	`16c3r`	`16c6r`	`16c9r`
32 clientes	`32c3r`	`32c6r`	`32c9r`
64 clientes	`64c3r`	`64c6r`	`64c9r`
128 clientes	`128c3r`	`128c6r`	`128c9r`
256 clientes	`256c3r`	`256c6r`	`256c9r`
512 clientes	`512c3r`	`512c6r`	`512c9r`
1024 clientes	`1024c3r`	`1024c6r`	`1024c9r`

5.2.2 Aplicações alvo

Criou-se três aplicações de funcionalidade similar para obter a referência do desempenho. Estas aplicações expõem uma rota HTTP que receberá requisições `GET` de um gerador de carga, no endereço `http://<host>:<porta>/benchmark`. Através desta requisição HTTP, a aplicação irá executar suas operações de leitura e/ou escrita enquanto atualiza um contador a respeito das métricas do teste. As aplicações instrumentadas [3] para o teste são:

- `plain-benchmark`: um servidor que opera sem protocolo de consenso. Escuta requisições na porta 5454.
- `raft-benchmark`: um servidor cujas réplicas aderem ao protocolo Raft através da biblioteca `Ra`. Configuração de réplicas feita manualmente. Escuta requisições na porta 5455.
- `pistis-benchmark`: um servidor que utiliza a biblioteca Pistis, portanto não requer trabalho adicional para aplicar o protocolo de consenso tampouco a gerência de réplicas distribuídas. Escuta requisições na porta 5456.

5.2.3 Métricas

Para o teste, duas métricas foram acompanhadas e produzidas pelos testes ao longo de 5 (cinco) minutos de carga sintética:

- **Latência**: nesta métrica, cada processo cliente realiza uma requisição para o servidor alvo contendo um comando de escrita ou leitura escolhido aleatoriamente. A duração do tempo de resposta do servidor, em milissegundos, é observada utilizando a função `:os.system_time/1`. Após o término da requisição, o tempo

[3] https://github.com/peterkrauz/pistis-benchmarks

gasto é armazenado em uma nova linha de um arquivo `txt`, e o processo hiberna por 50 (cinquenta) milissegundos antes de repetir esta ação. Apenas um dos processos possui permissão para registrar a latência, e esta escolha é realizada aleatoriamente.

- **Vazão**: nesta métrica, o servidor instancia um processo assíncrono que registra em um arquivo `txt` a diferença entre a quantia de operações, leitura ou escrita, registradas na última medição com a quantia de operações atual. O processo hiberna por 1 (um) segundo antes de repetir esta ação.

5.2.4 Resultados

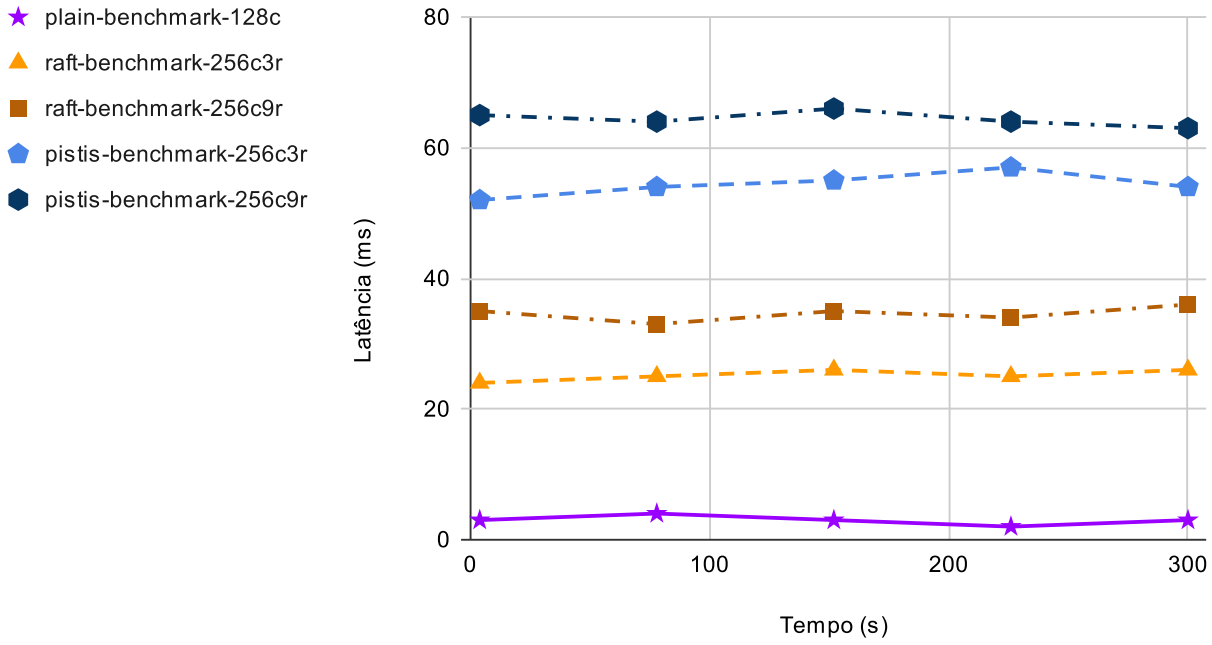


Figura 5 – Latência correspondente a quatro mil comandos, temporalmente

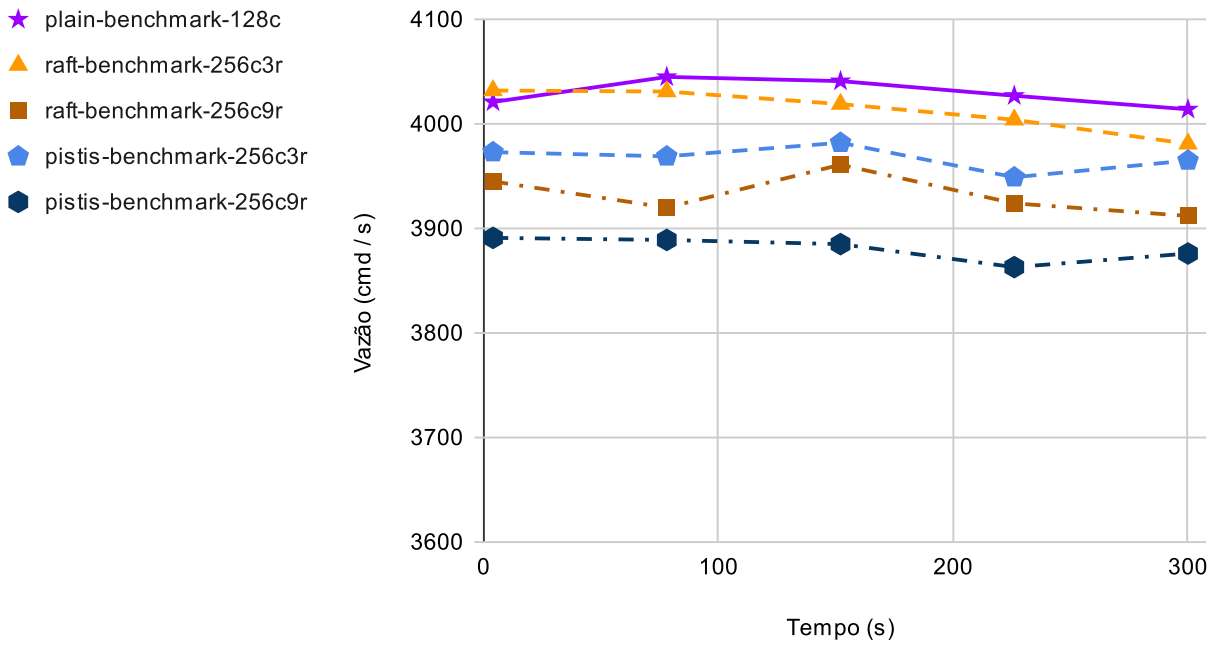


Figura 6 – Vazão em torno de quatro mil comandos, temporalmente

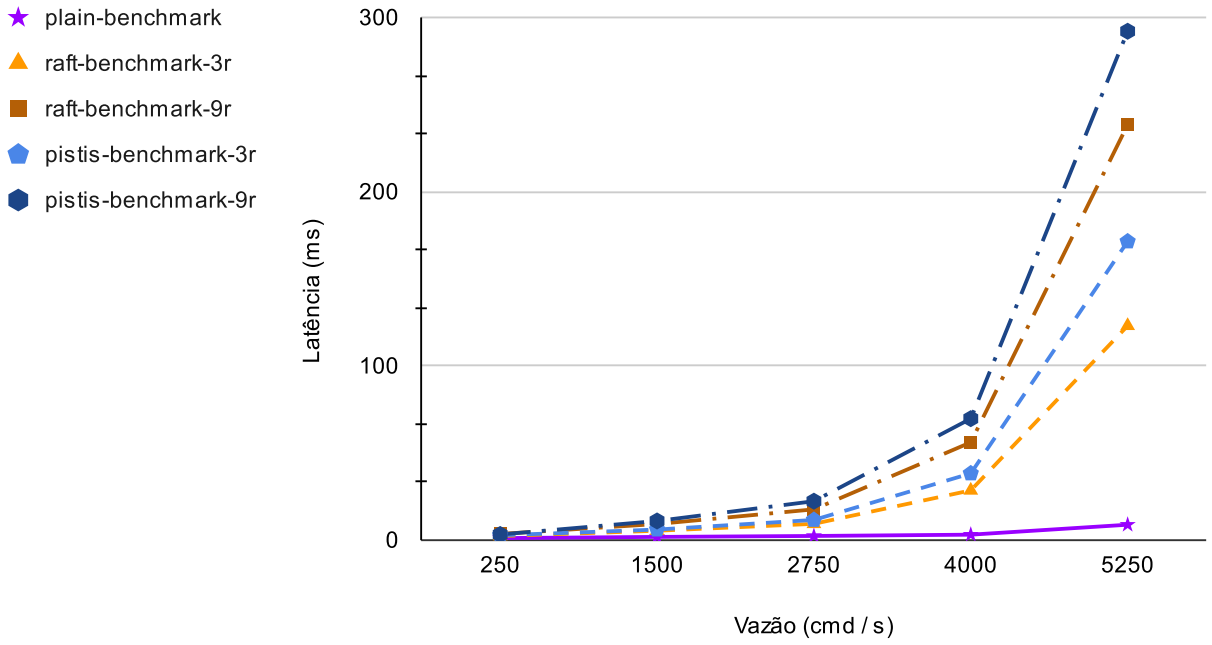


Figura 7 – Saturação das aplicações

Com base nos dados coletados pelos teste de vazão (Figura 6) e latência (Figura 5), assim como o de saturação (Figura 7), é possível afirmar que a adição da biblioteca não sacrificou o desempenho quando comparada com a versão manualmente configurada do protocolo de consenso e réplicas distribuídas. A gerência das réplicas pela biblioteca é eficaz, visto que a adição de novas réplicas não incrementou

a latência demasiadamente nem diminuiu a vazão do servidor. Quando comparadas com o servidor sem réplicas nem protocolo de consenso, as aplicações com réplicas distribuídas em consenso é visivelmente menos performática, o que é esperado dado o esforço gasto na execução do protocolo de consenso e troca de mensagens entre as instâncias distribuídas. No aspecto de tolerância a falhas, testes manuais foram feitos durante o desenvolvimento da biblioteca para verificar os devidos processos eram reinicializados quando tinham sua execução interrompida a força. Por causa disso, não foram realizados testes instrumentados direcionados aos atributos de tolerância a falha.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma biblioteca de código aberto, cujo objetivo de composição de réplicas distribuídas fortemente consistentes deve ser alcançado com fácil configuração e razoável custo no desempenho da aplicação hospedeira. A biblioteca se responsabiliza pela gerência de instâncias distribuídas da máquina virtual Erlang enquanto impõe um estrito consenso nas mensagens enviadas ao servidor. A biblioteca recorre a outra biblioteca de código aberto para a implementação do protocolo de consenso, mas simplifica a interação com a mesma por uma API enxuta e idiomática.

No quesito de usabilidade, o projeto demanda pouco esforço para ser configurado, usando uma ferramenta de gerenciamentos de pacotes para a distribuição e instalação do biblioteca, enquanto suas variáveis de ambiente são poucas e opcionais em sua maioria. O contrato que deve ser cumprido pelo desenvolvedor usuário – a implementação da máquina de estados – é compreensível e clara. Apesar disso, não foi possível implementar a biblioteca como um interceptador completamente transparente. Porque o repasse das mensagens do servidor para a implementação do protocolo de consenso teve de ser feito de forma imperativa – ou seja, invocando primitivas da implementação do protocolo de consenso diretamente –, a biblioteca demanda que o usuário final adicione invocações do repasse da mensagem.

Do ponto de vista de desempenho, a adição da biblioteca, ao passo que abstrai a configuração não-trivial do ambiente distribuído e o consenso entre as réplicas, não sacrificou a latência ou vazão da aplicação quando comparada à aplicação com réplicas distribuídas em consenso gerida manualmente. Ocasionais aumentos de latência foram vistos durante os testes, mas o atraso agregado – 10 a 20 milissegundos – não prejudicou a aplicação. O custo da biblioteca, portanto, é aceitável, visto que para facilitar o desenvolvimento e fazê-lo de forma prática, é natural que o desempenho da aplicação hospedeira diminua.

6.1 TRABALHOS FUTUROS

Melhorias deste trabalho visam ampliar a segurança na gerência das réplicas. Para isto, a adição de um novo módulo especializado em verificar a saúde do componente *Raft* e da máquina de estados atrelada faria com que falhas independentes em alguma das duas partes fossem capturadas e sinalizadas para o gerenciador do *cluster* tomar ação.

Um ponto adicional de melhoria é o desacoplamento entre a biblioteca e o protocolo de consenso. O suporte de múltiplos protocolos de consenso é desejável, mas demandaria a construção de uma camada de abstração adicional na biblioteca, entre o despacho das mensagens e o protocolo de consenso. Esta camada eliminaria a dependência entre o projeto e as primitivas concretas do Raft.

Além disso, o módulo `:slave`[1], utilizado pela biblioteca para a instanciação de nodos BEAM, teve sua depreciação anunciada recentemente. Para manter a biblioteca utilizando os componentes mais atualizados e seguros possível da plataforma Erlang, a desconexão deste módulo na biblioteca e implementação do módulo substituto `:peer`[2] seria um passo fundamental.

Por último, é possível reduzir os passos necessários para a implantação da biblioteca caso a biblioteca instanciar o processo `Pistis.Core.Entrypoint` internamente uma única vez. Hoje, este módulo deve ser adicionado pelo usuário final à lista de processos filhos da aplicação Elixir – um passo que não demanda muito esforço. Porém, é um passo adicional, e poderia ser feito internamente pela biblioteca, com a devida configuração.

[1] https://www.erlang.org/doc/man/slave.html
[2] https://erlang.org/documentation/doc-13.0-rc1/lib/stdlib-4.0/doc/html/peer.html

REFERÊNCIAS

ARMSTRONG, Joe. A History of Erlang. *In*: PROCEEDINGS of the Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California: Association for Computing Machinery, 2007a. (HOPL III), 6–1–6–26. DOI: 10.1145/1238844.1238850. Disponível em: https://doi.org/10.1145/1238844.1238850.

ARMSTRONG, Joe. Erlang — Software for a Concurrent World. *In*: PROCEEDINGS of the 21st European Conference on ECOOP 2007: Object-Oriented Programming. Berlin, Germany: Springer-Verlag, 2007b. (ECOOP '07), p. 1. DOI: 10.1007/978-3-540-73589-2_1. Disponível em: https://doi.org/10.1007/978-3-540-73589-2_1.

BETTS, Thomas. **The Resurgence of Functional Programming**. QCon. 2020. Disponível em: https://www.infoq.com/news/2020/11/functional-programming/.

BUDHIRAJA, Navin; MARZULLO, Keith; SCHNEIDER, Fred B.; TOUEG, Sam. The primary-backup appproach. *In*: DISTRIBUTED systems. [*S.l.*: *s.n.*], 1993. P. 199–216.

CHEN, Tom. The Web is Everywhere [Note from the Editor-in-Chief]. **IEEE Communications Magazine**, IEEE, v. 45, n. 9, p. 16–16, 2007.

CORBETT, James C. *et al.* Spanner: Google's Globally-Distributed Database. *In*: 10TH USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). Hollywood, CA: USENIX Association, out. 2012. P. 261–264. Disponível em: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Distributed Systems: Concepts and Design**. [*S.l.*]: Addison-Wesley Publishing Company, mai. 2011.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. *In*: COMMUNICATIONS of the ACM. [*S.l.*: *s.n.*], 2008. P. 107–113.

DENIZ ALTINBÜKEN, Emin Gün Sirer. **Commodifying Replicated State Machines with OpenReplica**. [*S.l.*], 2012.

ENES, Vitor; BAQUERO, Carlos; REZENDE, Tuanir França; GOTSMAN, Alexey; PERRIN, Matthieu; SUTRA, Pierre. State-machine replication for planet-scale systems. *In*. DOI: 10.1145/3342195.3387543. Disponível em: https://doi.org/10.1145/3342195.3387543.

FRITCHIE, Scott Lystig. Chain replication in theory and in practice. *In*. DOI: 10.1145/1863509.1863515. Disponível em: https://doi.org/10.1145/1863509.1863515.

GRIMALDI, William M. A. A Note on the Pisteis in Aristotle's Rhetoric, 1354-1356. **The American Journal of Philology**, JSTOR, v. 78, n. 2, p. 188, 1957. DOI: 10.2307/291828. Disponível em: https://doi.org/10.2307/291828.

GUERRAOUI, Rachid; SCHIPER, André. Fault-tolerance by replication in distributed systems. *In*: RELIABLE Software Technologies — Ada-Europe '96. [*S.l.*]: Springer Berlin Heidelberg, 1996. P. 38–57. DOI: 10.1007/bfb0013477. Disponível em: https://doi.org/10.1007/bfb0013477.

HERLIHY, Maurice P.; WING, Jeannette M. Linearizability: a correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems**, Association for Computing Machinery (ACM), v. 12, n. 3, p. 463–492, jul. 1990. DOI: 10.1145/78969.78972. Disponível em: https://doi.org/10.1145/78969.78972.

HEWITT, Carl; BISHOP, Peter; STEIGER, Richard. A universal modular ACTOR formalism for artificial intelligence. *In*: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence. [*S.l.*: *s.n.*], 1973. P. 235–245.

HOWARD, Heidi; MALKHI, Dahlia; SPIEGELMAN, Alexander. Flexible Paxos: Quorum Intersection Revisited. *In*: FATOUROU, Panagiota; JIMÉNEZ, Ernesto; PEDONE, Fernando (Ed.). **20th International Conference on Principles of Distributed Systems (OPODIS 2016)**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. (Leibniz International Proceedings in Informatics (LIPIcs)), 25:1–25:14. DOI: 10.4230/LIPIcs.OPODIS.2016.25. Disponível em: http://drops.dagstuhl.de/opus/volltexte/2017/7094.

HUNT, Patrick; KONAR, Mahadev; JUNQUEIRA, Flavio Paiva; REED, Benjamin C. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *In*: USENIX Annual Technical Conference. [*S.l.*: *s.n.*], 2010.

JURIC, Saša. **Elixir in Action**. [*S.l.*]: Manning Publications, 2019.

LAMPORT, L. **Time, clocks, and the ordering of events in a distributed system.** Communications of the ACM, ACM, 1978.

LAMPORT, Leslie. The part-time parliament. **ACM Transactions on Computer Systems**, Association for Computing Machinery (ACM), v. 16, n. 2, p. 133–169, mai. 1998. DOI: 10.1145/279227.279229. Disponível em: https://doi.org/10.1145/279227.279229.

LIU, Shengyun; VIOTTI, Paolo; CACHIN, Christian; QUÉMA, Vivien; VUKOLIC, Marko. XFT: practical fault tolerance beyond crashes. *In*: OSDI'16: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. [*S.l.*: *s.n.*], 2016. P. 485–500.

MORARU, Iulian; ANDERSEN, David G.; KAMINSKY, Michael. There is more consensus in Egalitarian parliaments. *In*: PROCEEDINGS of the Twenty-Fourth ACM Symposium on Operating Systems Principles. [*S.l.*]: ACM, nov. 2013. DOI: 10.1145/2517349.2517350. Disponível em: https://doi.org/10.1145/2517349.2517350.

NYSTROM, Jan. **Analysing fault tolerance for erlang applications**. Place of publication not identified: Uppsala University, 2009. ISBN 978-91-554-7532-1.

ONGARO, Diego; OUSTERHOUT, John K. In Search of an Understandable Consensus Algorithm. *In*: GIBSON, Garth; ZELDOVICH, Nickolai (Ed.). **USENIX Annual Technical Conference**. [*S.l.*]: USENIX Association, 2014. P. 305–319. Disponível em: http://dblp.uni-trier.de/db/conf/usenix/usenix2014.html#OngaroO14.

ORG, Erlang. **Official Erlang Documentation**. [*S.l.*: *s.n.*], 2022. https://www.erlang.org/docs. Online; accessed 09 March 2022.

PEREIRA, Paola Martins; DOTTI, Fernando Luis; MEINHARDT, Cristina; MENDIZABAL, Odorico Machado. A library for services transparent replication. *In*. DOI: 10.1145/3297280.3297308. Disponível em: https://doi.org/10.1145/3297280.3297308.

RABBITMQ. **Launch of RabbitMQ open source Enterprise Messaging**. RabbitMQ. 2007. Disponível em: https://www.rabbitmq.com/resources/RabbitMQ_PressRelease_080207.pdf.

RABBITMQ. **Ra**. [*S.l.*]: GitHub, 2017. https://github.com/rabbitmq/ra.

ROGERS, James. Google lost $1.7M in ad revenue during YouTube outage, expert says. **Fox Business**, 2020. Disponível em: https://www.foxbusiness.com/technology/google-lost-ad-revenue-during-youtube-outage-expert.

SCHNEIDER, F. B. **Implementing fault-tolerant services using the state machine approach: A tutorial**. ACM Computing Surveys (CSUR): ACM, 1990.

SCHOENFELDER, Paul. **Functional Imperative Programming With Elixir**. 2018. Disponível em: http://bitwalker.org/posts/2018-03-18-functional-imperative-programming-with-elixir. Acesso em: 30 jun. 2022.

SHRAER, Alexander; REED, Benjamin; MALKHI, Dahlia; JUNQUEIRA, Flavio. Dynamic reconfiguration of primary/backup clusters. *In*: USENIX ATC'12: Proceedings of the 2012 USENIX conference on Annual Technical Conference. [*S.l.*: *s.n.*], 2012. P. 39.

STRESSGRID. **Benchmarking Go vs Node vs Elixir**. 2020. Disponível em: https://stressgrid.com/blog/benchmarking_go_vs_node_vs_elixir/. Acesso em: 15 jun. 2022.

TANENBAUM, Andrew S.; RENESSE, Robbert van. Distributed operating systems. *In*: ACM Computing Surveys. [*S.l.*: *s.n.*], 1985.

TEAM, The Elixir. **Official Elixir Documentation**. [*S.l.*: *s.n.*], 2022. https://elixir-lang.org/docs.html. Online; accessed 11 March 2022.

UGLIARA, Fellipe Augusto; VIEIRA, Gustavo Maciel Dias; OLIVEIRA GUIMARÃES, José de. Transparent Replication Using Metaprogramming in Cyan. *In*: PROCEEDINGS of the 21st Brazilian Symposium on Programming Languages. [*S.l.*]: ACM, set. 2017. DOI: 10.1145/3125374.3125375. Disponível em: https://doi.org/10.1145/3125374.3125375.

VAN RENESSE, Robbert; SCHNEIDER, Fred B. Chain Replication for Supporting High Throughput and Availability. *In*: 91–104. USENIX Symposium On Operating Systems Design and Implementation (OSDI). [*S.l.*: *s.n.*], 2004.

VIEIRA, Gustavo Maciel Dias; BUZATO, Luiz Eduardo. Treplica: Ubiquitous replication. *In*: 26TH Brazilian Symposium on Computer Networks and Distributed Systems. Rio de Janeiro, Brasil: [*s.n.*], 2008.

WHITE, B. et al. **An integrated experimental environment for distributed systems and networks**. Proc. of the Fifth Symposium on Operating Systems Design e Implementation.: USENIX Association, 2002.

WILFREDO, Torres. **Software Fault Tolerance: A Tutorial**. [*S.l.*], 2000.

WINSTON, P H; HORN, B K. LISP. Second edition, jan. 1986. Disponível em: `https://www.osti.gov/biblio/7203980`.

APÊNDICE A – CÓDIGOS DO PROJETO

A.1 BIBLIOTECA

O código fonte pode ser encontrado em: https://github.com/peterkrauz/pistis.

A.2 TESTES DE DESEMPENHO

O código fonte utilizado para os testes de desempenho pode ser encontrado em: https://github.com/peterkrauz/pistis-benchmarks.

ANEXO A – ARTIGO DO PROJETO

Biblioteca para Replicação Máquina de Estados em Elixir

Peter Michael Claes Krause[1]

[1]Departamento de Informática e Estatística – Universidade

Federal de Santa Catarina – Floriánopolis – SC – Brasil

peter.krause@grad.ufsc.br

Abstract. *Pistis is an open-source library that abstracts the creation and management of distributed, strongly-consistent state machine replicas for the Elixir/Erlang platform. It uses Raft as a consensus protocol in order to ensure total ordering of all client requests and so be able to enforce active replication through the State Machine Replication technique, while also tackling cluster formation and guaranteeing the health of each replica. It demands from the user just a concrete implementation of the desired state machine.*

Resumo. *Este trabalho desenvolveu Pistis, uma biblioteca de código-aberto que abstrai a composição de um cluster de réplicas distribuídas fortemente consistentes para o ecossistema Elixir/Erlang. A biblioteca recorre ao protocolo de consenso Raft para garantir a ordenação total das mensagens e implementar a estratégia de Replicação Máquina de Estados, ao passo que se responsabiliza pela gerência do cluster e da saúde das instâncias que o integram, demandando do usuário apenas uma implementação concreta da máquina de estados desejada.*

1. Introdução

Servidores modernos enfrentam uma crescente demanda por disponibilidade. Uma maneira de ampliar a disponibilidade de um servidor é a criação de réplicas do mesmo, substituindo um ponto de falha central por uma quantia de réplicas que podem ser acionadas para lidar com requisições de clientes em caso de falha. Para manter estas réplicas atualizadas e úteis, deve-se garantir que seu estado seja atualizado conforme o servidor.

A estratégia Replicação Máquina de Estados é uma maneira de garantir consistência forte entre as réplicas. Paralelamente, a indústria de desenvolvimento de software presencia uma retomada no uso de linguagens funcionais no desenvolvimento convencional, pelo fato das restrições do paradigma proporcionarem escalabilidade e segurança no desenvolvimento de aplicações.

No ambiente funcional, a linguagem Erlang sobressai no quesito de tolerância a falha: sua máquina virtual foi desenvolvida para facilitar a construção de sistemas concorrentes, distribuídos e tolerantes a falha. Mesmo assim, instâncias distribuídas da máquina virtual requerem severo trabalho manual para serem configuradas, e a adoção de uma estratégia de replicação ativa requer conhecimento teórico prévio.

Com esta interseção em mente, foi desenvolvido Pistis, uma biblioteca que abstrai a implementação da Replicação Máquina de Estados, enquanto cuida da gerência de instâncias distribuídas da máquina virtual da plataforma Erlang.

2. Trabalhos relacionados

Em [Enes, et al. 2020; Pereira, et al. 2019; Altinbüken, D. 2012; Ugliara, et al. 2017], foram desenvolvidas soluções para estratégias de replicação transparentes. Cada trabalho propõe uma solução suficientemente desacoplado do artifício responsável pela ordenação total das mensagens, seja por anotações ou atuando como *proxies* do servidor.

Em [Fritchie, S. 2010; Nystrom, J. 2009], são explorados aspectos da plataforma de desenvolvimento Erlang, mas sem relação com a estratégia Replicação Máquina de Estados. Em [Fritchie, S. 2010], é implementada a estratégia de Replicação em Cadeia, enquanto em [Nystrom, J. 2009], um avaliador de tolerância a falha com base na árvore sintática da aplicação.

Ambos grupos exploram algo em comum com o presente trabalho. No entanto, a interseção entre estes é exatamente o objetivo da biblioteca Pistis: a dificuldade em lidar com artifícios distribuídos da máquina virtual da plataforma Erlang, enquanto oculta do usuário os detalhes não-triviais de implementação da estratégia Replicação Máquina de Estados. A tabela abaixo enumera as principais características dos trabalhos relacionados em comparação com o presente projeto.

Tabela 1. Pontos principais dos trabalhos relacionados

Trabalho	Estratégia	Transparência?	Linguagem
(FRITCHIE, 2010)	Replicação em cadeia	Não	Erlang
(ENES *et al.*, 2020)	RME	Não	Nenhuma
(NYSTROM, 2009)	Nenhuma	Não	Erlang
(PEREIRA *et al.*, 2019)	RME	Sim	Java
(DENIZ ALTINBÜKEN, 2012)	RME	Sim	Python
(UGLIARA *et al.*, 2017)	RME	Sim	Cyan (Java)
(KRAUSE, 2022)	RME	Sim	Elixir

3. Fundamentação teórica

Dois pilares teóricos embasam o presente projeto: o ecossistema Erlang/Elixir e a estratégia Replicação Máquina de Estados.

3.1. Erlang e Elixir

Erlang é uma plataforma de desenvolvimento direcionada para a construção de sistemas escaláveis, confiáveis e com alta disponibilidade de serviço [Armstrong, J. 2007a]. Ainda que concebida para solucionar as dores da indústrias de telecomunicações, os mesmos desafios podem ser encontrados na indústria de software, fazendo com que seus atributos se mantenham relevantes atualmente.

O cerne da plataforma é sua máquina virtual. Dentro desta, *threads* são mapeadas em escalonadores que gerenciam processos Erlang. Estes são versões leves, alocadas dinamicamente na pilha, cuja troca de contexto é barata, portanto, frequente [Armstrong, J. 2007b; Juric, S. 2019]. Estes processos não compartilham memória, e por causa disso eliminam a necessidade do uso de estruturas de sincronização como *mutex* e semáforos. Processo devem se comunicar por troca de mensagens e o fazem de

maneira assíncrona, seguindo o modelo de mensagens do Ator [Hewitt, C. et al. 1973]. A plataforma de desenvolvimento também inclui módulos de código de pronto que encapsulam comportamentos padrões.

A linguagem de programação Elixir foi construída sobre os pilares do Erlang. Herda seu caráter funcional, mas carrega uma sintaxe modernizada e menos verbosa. Conta com *macros* que podem ser desenvolvidos para a facilitação do desenvolvimento, além de uma interoperabilidade total com o Erlang. Qualquer módulo Elixir pode invocar funções de módulos Erlang

3.2. Replicação Máquina de Estados

Na busca por um sistema tolerante a falhas, a capacidade do mesmo em suportar erros de forma localizada, ao invés de uma falha total, é desejável. Uma das maneiras de aplicar este conceito é criar réplicas do servidor e garantir que as mesmas estejam atualizadas para que, em caso de falha, sejam utilizáveis. A Replicação Máquina de Estados é uma forma de se replicar um servidor ativamente. Isto é, todas as réplicas processam a requisição de um cliente antes de produzir uma resposta para a requisição. Desta maneira, se garante que o servidor é fortemente consistente.

A estratégia RME, em específico, visualiza o servidor como uma máquina de estados. Todas as requisições de clientes são vistas como comandos que podem ler ou alterar o estado do servidor. Estes comandos devem ser determinísticos, de forma que seja possível assegurar que todas as réplicas, ao receber um mesmo comando, passarão pela mesma transição de estado e, por fim, terão um estado idêntico [Schneider, F. 2009]. Para garantir que as réplicas recebam sempre o mesmo comando, um artifício ordenador de mensagens deverá ser utilizado. No presente projeto, utilizou-se o protocolo de consenso Raft, através de uma implementação Erlang de código-aberto mantida pela organização RabbitMQ.

4. Pistis

O objetivo da biblioteca é abstrair tarefas complexas do usuário, como a gerência de um *cluster* ou a ordenação total das mensagens enviadas ao servidor. Com esta transparência em mente, a biblioteca se posiciona como um interceptador entre cliente e máquina de estado, demandando uma implementação concreta da máquina de estados, para ser possível replicá-la em instâncias distribuídas, e também que o usuário final defina trechos de código que repassem os comandos ao processador da biblioteca.

A biblioteca agrupa suas funcionalidades em quatro módulos que trabalham em conjunto para resolver o problema descrito acima: *Pistis.Cluster*, *Pistis.Server*, *Pistis.Pod* e *Pistis.Machine*. Estes módulos tratam a gerência do *cluster*, o atendimento de requisições e despacho de mensagens aos clientes, o envelopamento da máquina de estados de forma genérica e a definição do protocolo da máquina de estados, respectivamente. O diagrama abaixo ilustra como a biblioteca coordena a interação de processos e instâncias distribuídas da máquina virtual em sua execução:

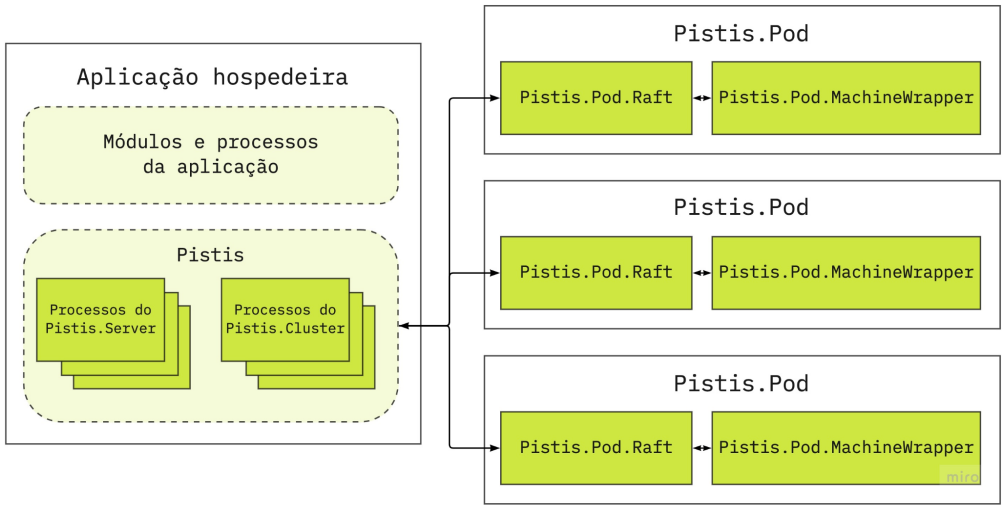


Figura 1. Processos e instâncias da máquina virtual no contexto da biblioteca

Dentre seus quatro módulos, o de maior interesse ao usuário é o *Pistis.Machine*, cuja função é expor um contrato da máquina de estados que deve ser implementado pela aplicação usuária.

4.1. Pistis.Machine

O módulo demanda a implementação de duas funções: *initial_state/0* e *process_comand/2*.

A função *initial_state* deverá retornar o estado inicial da máquina de estados. Este pode ser um dicionário, uma lista, um número inteiro, ou qualquer representação de estado.

A função *process_command* deve ser implementada para cada comando que a máquina de estados possa receber. O primeiro parâmetro desta função irá conter o conteúdo da requisição do cliente envelopada em um *struct* especializado, o *Pistis.Machine.Request*. O segundo parâmetro será o estado atual da máquina de estados. A função deve retornar um *struct Pistis.Machine.Response*, cujo conteúdo fica a critério da implementação do usuário.

5. Resultados

A avaliação da biblioteca se deu de duas formas: a facilidade em instalar e configurar a biblioteca, e o desempenho de uma aplicação hospedeira que utiliza a biblioteca.

5.1. Avaliação de usabilidade

A instalação da biblioteca é realizada utilizando o gerenciador de pacotes *mix*, ferramenta principal deste tipo de tarefa no ambiente Elixir. A instalação é feita como qualquer biblioteca do ecossistema: 1) adicionar o artefato a lista de dependências; 2) executar um comando no terminal para baixar os pacotes e compilar o projeto.

Após instalada, a biblioteca é configurada em duas etapas: 1) adicionar o módulo *Pistis.Core.Entrypoint* na lista de processos filhos da aplicação hospedeira; 2) fornecer

um arquivo *config.exs* com variáveis de configuração da biblioteca. O arquivo precisa definir apenas uma variável: *:machine*, cujo valor deve ser o nome do módulo que armazena a máquina de estados da aplicação cliente.

Variáveis de configuração não-obrigatórias incluem: 1) *:cluster_size*, um número inteiro que dita quantas réplicas serão instanciadas; 2) *:cluster_boot_delay*, número inteiro expresso em milissegundos tempo de espera que o servidor irá tomar antes de instanciar e conectar-se com as réplicas; e 3) *:known_hosts*, uma lista de endereços de máquinas virtuais Erlang. A biblioteca irá se conectar a estes *hosts* e utilizá-los para abrigar réplicas.

5.2. Avaliação de desempenho

Para avaliar o desempenho, produziu-se um *benchmark* sintético através da plataforma Emulab [White, B. et al. 2002] com uma licença acadêmica. Instâncias on-demand foram instanciadas para simular clientes geradores de carga ou servidores. Experimentos foram conduzidos com um série de combinações de número de clientes e número de servidores. Cada instância possuía 4096MB de memória RAM, 2 threads por CPU, 8 cores de processamento e um sistema operacional Ubuntu 20.04.

As aplicações alvo foram: 1) um servidor simples, sem réplicas, chamado de *plain-benchmark*; 2) um servidor com réplicas fortemente consistente utilizando Raft, porém configurado manualmente, chamado de *raft-benchmark*; e 3) uma aplicação que utilizou a biblioteca Pistis, portanto conta com réplicas fortemente consistentes, porém sem demandar trabalho adicional de configuração, chamado de *pistis-benchmark*.

A seguinte Figura mostra os resultados do teste de saturação:

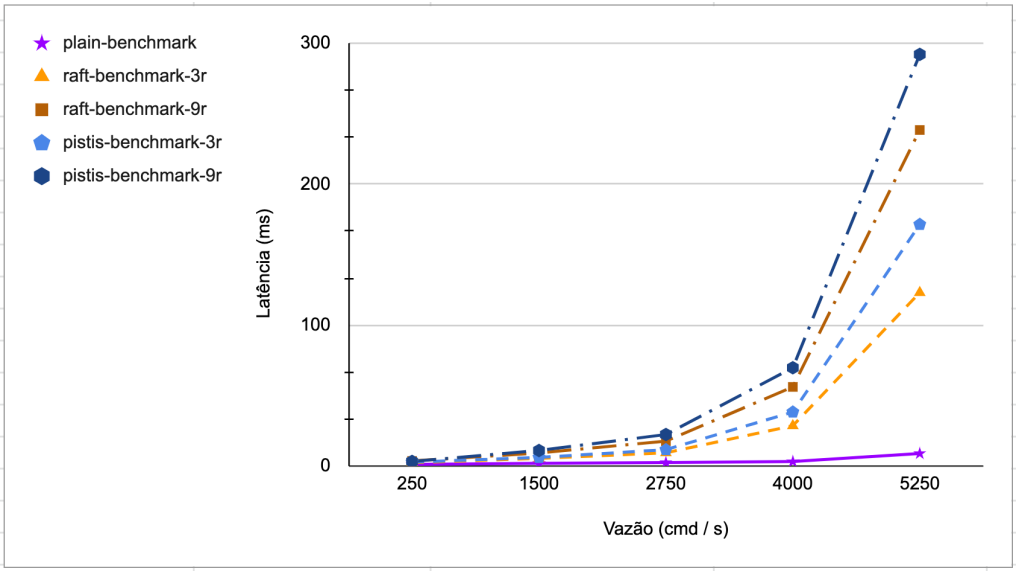


Figura 2. Teste de saturação das aplicações

6. Conclusão

Este trabalho apresentou uma biblioteca de código aberto, cujo objetivo de composição de réplicas distribuídas fortemente consistentes deve ser alcançado com

fácil configuração e razoável custo no desempenho da aplicação hospedeira. A biblioteca se responsabiliza pela gerência de instâncias distribuídas da máquina virtual Erlang enquanto impõe um estrito consenso nas mensagens enviadas ao servidor. A biblioteca recorre a outra biblioteca de código aberto para a implementação do protocolo de consenso, mas simplifica a interação com a mesma por uma API enxuta e idiomática

Sua instalação e configuração demanda a quantia usual de etapas quando comparada com outras bibliotecas comuns, visto que o processo de adição do artefato, instalação de pacotes, e definição de variáveis de ambiente são processos comuns a serem realizados na interação com pacotes de código de terceiros.

Do ponto de vista de desempenho, a biblioteca provocou um aumento na latência, mas esta latência adicional se deu em uma quantia aceitável, visto que a biblioteca facilitou o desenvolvimento do usuário e manteve as propriedades desejadas da replicação ativa.

7. Trabalhos futuros

- Incremento da robustez das instâncias distribuídas. Adicionar um processo assíncrono que verifique a saúde tanto do módulo de consenso, quanto da máquina de estados.
- Troca de módulo depreciado. O módulo *:slave*, utilizado pela biblioteca, teve sua depreciação anunciada enquanto o projeto era desenvolvido. O substituto recomendado pela plataforma Erlang é o módulo *:peer*, que conta com outras funções para instanciação de instâncias distribuídas.
- Desacoplamento do módulo de consenso. A biblioteca Pistis se tornou altamente dependente do protocolo Raft durante o desenvolvimento. Um módulo intermediário tornaria a biblioteca mais flexível, independente do algoritmo de consenso, e permitiria uma fácil adoção de outras alternativas ordenadoras de mensagem.

Referências

Fritchie, Scott Lystig. (2010) Chain replication in theory and in practice. In: Erlang '10: Proceedings of the 9th ACM SIGPLAN workshop on Erlang.

Enes, Vitor; Baquero, Carlos; Rezende, Tuanir França; Gotsman, Alexey; Perrin, Matthieu; Sutra, Pierre. (2020) State-machine replication for planet-scale systems. In: EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems.

Nystrom, Jan. (2009) Analysing fault tolerance for erlang applications. Place of publication not identified: Uppsala University, 2009. In: Uppsala: Acta Universitatis Upsaliensis, 2009.

Pereira, Paola Martins; Dotti, Fernando Luis; Meinhardt, Cristina; Mendizabal, Odorico Machado. (2019) A library for services transparent replication. In: SAC '19: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing.

Altinbüken, Deniz, Emin Gün Sirer. (2012) Commodifying Replicated State Machines with OpenReplica.

Ugliara, Fellipe Augusto; Vieira, Gustavo Maciel Dias; Oliveira Guimarães, José de. (2017) Transparent Replication Using Metaprogramming in Cyan. In: Proceedings of the 21st Brazilian Symposium on Programming Languages.

Hewitt, Carl; Bishop, Peter; Steiger, Richard. (1973) A universal modular ACTOR formalism for artificial intelligence. In: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence.

Armstrong, Joe. (2007a) A History of Erlang. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California: Association for Computing Machinery, 2007.

Armstrong, Joe. (2007b) Erlang — Software for a Concurrent World. In: PROCEEDINGS of the 21st European Conference on ECOOP 2007: Object-Oriented Programming. Berlin, Germany: Springer-Verlag, 2007.

Juric, Saša. Elixir in Action. [S.l.]: Manning Publications, 2019

Schneider, F. B. (2009) Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR): ACM, 1990.

White, B. et al. (2002) An integrated experimental environment for distributed systems and networks. Proc. of the Fifth Symposium on Operating Systems Design. Implementation.: USENIX Association, 2002.