Pontifícia Universidade Católica
do Rio de Janeiro

Juarez da Silva Bochi

Programação Paralela no Banco de Dados Chave-Valor Redis

Dissertação de Mestrado

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador: Profª. Noemi de La Rocque Rodriguez

Rio de Janeiro
Setembro de 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Juarez da Silva Bochi

Programação Paralela no Banco de Dados Chave-Valor Redis

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Profª. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática — PUC–Rio

Prof. Roberto Ierusalimschy
Pesquisador — PUC-Rio

Profª. Ana Lúcia de Moura
Pesquisador — PUC-Rio

Prof. Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico — PUC-Rio

Rio de Janeiro, 03 de Setembro de 2015

Juarez da Silva Bochi

Graduou–se em Engenharia Elétrica pela Universidade Federal do Rio grande do Sul (UFRGS). Atualmente trabalha com desenvolvimento de *software* na Globo.com.

Ficha Catalográfica

Bochi, Juarez da Silva

Programação Paralela no Banco de Dados Chave-Valor Redis / Juarez da Silva Bochi; orientador: Noemi de La Rocque Rodriguez. — 2015.

62 f. : il. (color); 30 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Redis. 3. Lua. 4. Programação Paralela. 5. Modelos de Concorrência. 6. Modelo M:N. 7. Bancos de Dados. I. Rodriguez, Noemi de La Rocque. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

À globo.com, pelo apoio financeiro.
Aos professores e colegas da PUC-Rio, por todos os ensinamentos.
A todos meus amigos e familiares, pela amizade.
Aos amigos Leandro Ribeiro e Jean Emer, pela ajuda na revisão do texto.
À minha orientadora Noemi, pela inspiração, paciência e incentivo.
À Isabela, minha melhor amiga e companheira, por toda compreensão e carinho.

Resumo

Bochi, Juarez da Silva; Rodriguez, Noemi de La Rocque. **Programação Paralela no Banco de Dados Chave-Valor Redis**. Rio de Janeiro, 2015. 62p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Redis é um banco de dados chave-valor de código livre que dá suporte à avaliação de scripts Lua, mas sua implementação utiliza apenas uma tarefa de sistema operacional. Scripts longos são desencorajados porque a avaliação do código é bloqueante, o que pode causar degradação de desempenho para os demais usuários. Através da aplicação do modelo de concorrência $M:N$, que combina tarefas de nível de sistema operacional com tarefas do nível de usuário, adicionamos no Redis a capacidade de execução de scripts em paralelo, permitindo que todos os núcleos do servidor sejam explorados. Com a utilização de corotinas Lua, implementamos um escalonador capaz de alocar e suspender a execução de tarefas de nível de usuário nos núcleos disponíveis sem necessidade de alteração do código dos scripts. Este modelo permitiu proteger o programador das complexidades naturais do paralelismo como sincronização no acesso a recursos compartilhados e escalonamento das tarefas.

Palavras–chave

Redis; Lua; Programação Paralela; Modelos de Concorrência; Modelo M:N; Bancos de Dados.

Abstract

Bochi, Juarez da Silva; Rodriguez, Noemi de La Rocque (Advisor). **Parallel Programing in the Redis Key-Value Datastore**. Rio de Janeiro, 2015. 62p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Redis is an open source key-value database that supports Lua programming language scripts, but it's implementation is single threaded. Long running scripts are discouraged because script evaluation is blocking, which may cause service levels deterioration. Applying the $M{:}\,N$ threading model, which combines user and operating system threads, we added to Redis the ability of running scripts in parallel, leveraging all server cores. With the use of Lua coroutines, we implemented a scheduler able to allocate and suspend user-level tasks in the available cores without the need of changing scripts' source code. The M:N model allowed us to protect the programmer from the natural complexities that arise from parallel programming, such as access to shared resources synchronization and scheduling of tasks into different operational system threads.

Keywords

Redis; Lua; Parallel Programming; Threading models; M:N threading model; Databases.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Existem poucos artigos na literatura que investigam modelos de paralelismo *multi-core* em linguagens dinâmicas (Skyrme et al., 2014). Normalmente, estas linguagens se limitam a expor a API de baixo nível do sistema operacional para criação de *threads* e processos, deixando o programador sujeito às armadilhas do compartilhamento de memória e à complexidade das primitivas de sincronização, como *locks* e semáforos. Este trabalho contribui com a análise e implementação do modelo multi-tarefa $M{:}\,N$, ou híbrido, para uma linguagem de *script* em uma aplicação real, com objetivo de contornar algumas destas dificuldades.

Mais especificamente, exploramos o modelo $M{:}\,N$, que combina tarefas de usuário e de sistema operacional, no banco de dados chave-valor Redis[1], possibilitando a execução paralela de *scripts* Lua. Redis é um banco de dados em memória, *single threaded*, não relacional e de alto desempenho que suporta execução bloqueante de *scripts* Lua. Através da aplicação do modelo $M{:}\,N$, permitimos a execução paralela de *scripts* de uma forma que abstrai as complexidades do uso compartilhado de memória e do escalonamento de múltiplas tarefas. Nossas modificações da aplicação viabilizam o uso de todos os núcleos do servidor simultaneamente, através da execução em paralelo de múltiplos *scripts*, sem necessidade de modificação do código de *scripts* existentes.

A implementação padrão do banco de dados Redis não é multi-tarefa e portanto não há processamento paralelo de outros comandos durante a execução de qualquer comando ou *script*, mesmo que o servidor possua mais de um núcleo. Esta característica da implementação do Redis inviabiliza a execução de *scripts* de longa duração em cenários práticos com múltiplos clientes. Um único cliente executando *scripts* bloqueantes de longa duração pode degradar o atendimento dos demais clientes.

Implementamos o modelo multi-tarefa $M{:}\,N$ através da combinação de múltiplas *threads* de sistema operacional com corotinas Lua, de uma forma que possibilita a paralelização de *scripts* sem necessidade de modificação do

[1] `http://redis.io`

código existente. Os *scripts* são executados sem bloquear os demais clientes e os núcleos extras do servidor são utilizados sem nenhuma alteração na semântica dos *scripts* para interação com o banco de dados. O escalonador implementado permite que os *scripts* de usuário sejam suspensos e retomados de acordo com a disponibilidade de núcleos e do banco de dados sem que o programador precise ceder explicitamente o controle da linha de execução ou que tenha que se preocupar com o uso de memória compartilhada.

Neste trabalho, descrevemos como implementamos este modelo e introduzimos novos comandos no banco de dados Redis para execução de *scripts* de maneira assíncrona compatível com os *scripts* não bloqueantes. A manutenção dos comandos para execução de *scripts* bloqueantes é justificada porque em alguns cenários é desejável que o código execute de maneira atômica. Também avaliamos a utilidade do sistema desenvolvido, através de aplicações reais criadas para explorar três grandes áreas de aplicação que visualizamos para nosso sistema:

- ***scripts* de longa duração**: São *scripts* que fazem alto uso de CPU e que não poderiam ser executados de forma bloqueante. Escolhemos desenvolver um sistema de detecção de anomalias e o implantamos para monitorar um *datacenter* com centenas de máquinas.
- **aceleração de *scripts***: Neste cenário, o objetivo é diminuir o tempo de execução de uma tarefa explorando o paralelismo. Para representar um caso extremo deste tipo de aplicação, implementamos um algoritmo paralelo de quadratura adaptativa, que calcula a integral definida de uma função através da soma da área de trapézios.
- **processamento em *tempo real***: Neste caso, desejamos processar um fluxo de dados contínuo sem bloquear o recebimento de informação. Para validar esta situação, criamos um detector de tópicos populares para a rede social Twitter.

Esta dissertação é dividida da seguinte maneira: O capítulo 2 expõe a motivação para se reavaliar os modelos de programação multi-tarefa e embasa a teoria por trás do modelo de paralelismo adotado. No capítulo 3, descrevemos o banco de dados Redis, a linguagem Lua e como o Redis a utiliza. No capítulo 4, apresentamos as modificações que foram feitas para aplicar o modelo $M{:}\,N$ na paralelização de *scripts*. O capítulo 5 apresenta as aplicações desenvolvidas e o resultado dos testes. Finalmente, no capítulo 6, encerramos com a conclusão.

2
Modelos multi-tarefa

2.1
Motivação

O *"free-lunch"* (Sutter, 2005) do aumento de velocidade de processamento através da aceleração do *clock* dos processadores acabou já há alguns anos. Conforme pode ser visto na figura 2.1 (McKenney, 2011), a frequência das CPUs está estagnada há mais de uma década. Desde o fim do aumento da frequência dos processadores, a aceleração de programas está condicionada ao bom aproveitamento de todos os núcleos disponíveis. A programação paralela é necessária e, apesar da dificuldade de se produzir código correto com compartilhamento de memória (Ousterhout, 1996), os programadores muitas vezes não possuem alternativas.

Figura 2.1: Evolução da frequência do *clock* de CPUs Intel, em MHz

Decompor tarefas em etapas que possam ser executadas em paralelo é uma das fontes de complexidade essencial (Brooks Jr., 1987) da programação paralela. Outra fonte de complexidade, acidental, da programação paralela é a criação e alocação destas subtarefas em múltiplas tarefas de sistema

operacional. Este mapeamento de atividades em diferentes tarefas de sistema operacional é necessário para que haja paralelismo e aproveitamento dos vários núcleos do computador, uma vez que cada tarefa de sistema operacional não roda em mais de um núcleo simultaneamente. Este escalonamento exige muitos cuidados para uma implementação eficiente e correta. O acesso a recursos compartilhados, como memória e execução de operações de E/S, introduz dificuldades para o programador. A adoção de certos modelos de programação, linguagens ou bibliotecas pode auxiliar nestas dificuldades.

Neste capítulo, analisaremos os modelos de concorrência multi-tarefa, comparando as vantagens e desvantagens de cada um quanto à facilidade de programação, capacidade de paralelismo e eficiência na criação e na troca de contexto entre tarefas. Também mencionaremos os padrões de implementação e as abstrações de linguagens mais comuns em cada modelo.

2.2 Modelos de concorrência multi-tarefa

Tarefas ou *threads* são linhas de execução ou sequências de instruções independentes, cada um com sua própria pilha de execução. Um escalonador, seja de sistema operacional ou no nível de usuário, pode alocar estas tarefas em um ou mais núcleos do processador. Geralmente, os modelos multi tarefa são classificados em tarefas no nível de sistema operacional, tarefas no nível do usuário e modelo híbrido. Descreveremos cada um destes modelos a seguir.

2.2.1 O modelo 1:1: tarefas de nível do sistema operacional

O modelo predominante para concorrência é a criação de uma *thread* de sistema operacional para cada tarefa independente que se deseja criar. Por esta razão, esse modelo é conhecido como 1:1. Todas as tarefas são criadas, gerenciadas, sincronizadas e finalizadas pelo núcleo (*kernel*) do sistema operacional.

Este modelo é o de implementação mais simples para as aplicações, uma vez que a coordenação das *threads* é feita pelo sistema operacional. Outra vantagem deste modelo é que o sistema operacional pode alocar as *threads* em núcleos diferentes do computador, possibilitando que elas executem em paralelo. Quando uma *thread* executa uma operação bloqueante, o sistema operacional também pode suspender a tarefa e alocar outras *threads* para execução. A preempção, ato de suspender temporariamente uma tarefa para executar outra rotina, não necessita cooperação da tarefa. A pilha de execução

da tarefa é mantida e a troca de contexto é feita de maneira transparente para a aplicação.

Uma das desvantagens deste modelo está no custo relativamente alto de criação e troca de contexto entre *threads*. Geralmente, o modelo é vantajoso quando é necessário criar um pequeno número de *threads* independentes que rodem por longos períodos (McCracken, 2002). Além disto, este modelo clássico de programação multi-tarefa com preempção expõe o programador a algumas armadilhas. A combinação da preempção com o compartilhamento de memória é perigosa e requer o uso de primitivas de sincronização como *locks* e semáforos para o funcionamento correto da aplicação. O domínio desses mecanismos é raro entre programadores de aplicações, que são forçados a lutar contra *deadlocks*, condições de corrida e não determinismo.

Existem algumas abstrações e primitivas de linguagem que amenizam as dificuldades do paralelismo tradicional multi-tarefa com compartilhamento de memória. Estruturas de dados atômicas, como as presentes na linguagem Java, permitem que variáveis sejam compartilhadas sem a necessidade do controle de exclusão mútua manual. O uso de memória transacional (Herlihy et al., 2003) é outra maneira de se executar código de forma que outras tarefas não consigam ver estados intermediários, permitindo o compartilhamento de variáveis sem necessidade de sincronização. A implementação de memória transacional por *software* permite que isto seja feito em *hardware* comum e está disponível, por exemplo, na linguagem Clojure (Hickey, 2008). Passagem de mensagens e suas abstrações, como canais sequenciais (Hoare, 1978), implementados na linguagem Go[1], e os atores de Erlang (Virding et al., 1996) e Scala (Haller e Odersky, 2007) também evitam a necessidade de sincronização ao eliminar o compartilhamento de memória. Estes recursos não são tão comuns em linguagens dinâmicas. Além disso, interpretadores de algumas destas linguagens, como Python e Ruby, utilizam um *lock* global do interpretador (GIL, do inglês *Global Interpreter Lock*). Este mecanismo de sincronização é adotado para proteger estruturas internas da linguagem, como aquelas usadas para gerenciamento de memória, que não são *thread-safe*[2]. Isto acaba dificultando a obtenção de paralelismo nestas linguagens.

O padrão de projeto *thread pool* evita o alto custo de criação e destruição de tarefas do modelo 1:1. Este padrão é bastante comum e está disponível na linguagem Java, como uma das classes de `Executors`. A aplicação inicializa um conjunto de trabalhadores (*workers*) e envia tarefas (tasks) para estes trabalhadores através de uma fila. Cada *worker* é associado a uma *thread*

[1] `http://golang.org/`
[2] `https://wiki.python.org/moin/GlobalInterpreterLock`

de sistema operacional e pode, portanto, executar trabalho em paralelo. As tarefas (*tasks*), neste caso, não devem ser confundidas com tarefas (*threads*) de sistema operacional ou de usuário. Elas não possuem pilha própria de execução e são sempre executadas atomicamente, do início até o fim. Ao contrário de *threads*, que podem sofrer preempção pelo sistema operacional, o *worker* não pode suspender estas tarefas temporariamente para execução de outra tarefa. Portanto, embora o número de tarefas possa superar o número de trabalhadores neste padrão, como as tarefas não possuem linha de execução própria, o padrão de *thread pool* não foge do modelo 1:1. De qualquer forma, o padrão é bastante útil e abstrai boa parte da complexidade do escalonamento de tarefas para o programador.

2.2.2 O modelo $N:1$: tarefas de nível de usuário

Neste modelo, a aplicação faz o escalonamento das tarefas e as tarefas não são sequer identificadas pelo sistema operacional. O modelo se chama $N:1$ porque a aplicação executa N tarefas de nível de usuário em uma única *thread* de sistema operacional. As tarefas são gerenciadas no nível de usuário e o sistema operacional enxerga apenas um processo monolítico. Sendo assim, este estilo é o mais portátil, uma vez que não depende do *kernel* do sistema operacional. Apesar do escalonamento ser uma complexidade adicional para a aplicação, o escalonador pode ser otimizado de acordo com as necessidades da aplicação, ao invés de se contar com um escalonador genérico como é o do sistema operacional.

Outra diferença neste modelo é que as *threads* precisam ceder explicitamente o controle, já que não é possível implementar preempção no nível de usuário. Tipicamente se tem um custo inferior de criação de novas tarefas e a troca de contexto é mais barata, porque não há necessidade de se entrar em modo privilegiado do *kernel*. A falta da preempção traz a grande vantagem de se eliminar a necessidade de sincronização. Neste modelo de programação multi tarefa não preemptiva ou colaborativa, só há troca de contexto quando uma linha de execução cede explicitamente o controle. A troca de contexto é determinística e em pontos pré-definidos nas tarefas de nível de usuário. Portanto, não há necessidade de sincronização adicional através de *locks* ou semáforos.

Como desvantagem, já que o sistema operacional vê uma única *thread*, não é possível rodar paralelamente tarefas distintas em núcleos diferentes do processador. Outra desvantagem é que toda a aplicação é bloqueada caso uma tarefa execute uma operação bloqueante, por exemplo, uma operação de E/S. Desta forma, operações bloqueantes precisam ser emuladas com operações não

bloqueantes, o que nem sempre é possível (McCracken, 2002). Esta emulação deve ser feita pela aplicação, o que é um ônus para o programador. Ao utilizar uma biblioteca externa, por exemplo, ele precisa se certificar de que ela não é bloqueante. Mesmo que a biblioteca seja capaz de fazer operações de forma assíncrona, deve fazer isto de forma compatível com o escalonador da aplicação.

Devido a estas restrições, este modelo é bastante explorado em conjunto com a orientação a eventos, que tipicamente utiliza operações de entrada e saída não bloqueantes. Ao invés de se bloquear o fluxo de execução durante uma destas operações, um *loop* principal da aplicação executa continuamente e toda vez que uma operação de E/S é necessária, uma função de retorno (*callback*) é registrada para ser executada assincronamente quando a operação for concluída. A cada iteração do *loop* principal, o sistema operacional é consultado para que se determine as operações concluídas e as funções de retorno (*callback*) são chamadas.

Do ponto de vista do programador, no entanto, é mais simples raciocinar sobre operações de E/S síncronas ou bloqueantes, já que a pilha de execução e o contexto são preservados, como se a operação fosse executada instantaneamente. O modelo de programação de eventos é menos natural (Behren et al., 2003) para o programador. A inversão de controle faz com que diversos blocos de *callback* sejam aninhados, com um efeito no código conhecido coloquialmente como *"callback hell"*, ou gerenciamento manual da pilha (Adya et al., 2002). Abstrações como *Futuros* (Walker et al., 1990) e *Promessas* (Liskov, 1988), assim como o operador *await* do C# (Microsoft, 2014), aliviam este problema, ao preservar a pilha de execução.

Corotinas, como as implementadas em Lua (Moura e Ierusalimschy, 2009), também permitem implementar o modelo de tarefas de nível de usuário com operações de E/S não bloqueantes, mas de forma mais natural para o programador. Uma corotina pode ceder o controle (*yield*) e manter o seu contexto e pilha de execução quando o controle lhe for devolvido. Para emular operações de E/S não bloqueantes, ao invés de registrar uma função de retorno, a corotina pode ceder o controle enquanto o resultado não estiver disponível. Através deste mecanismo, uma biblioteca de E/S pode expor uma API que faz *yield* automaticamente em todas as operações de forma transparente para o cliente, liberando a corotina para a troca de contexto sempre que necessário. Isto permite que o programador raciocine sobre seu código como se ele fosse executado sequencialmente e de maneira síncrona.

2.2.3 O modelo híbrido $M{:}\,N$

Os modelos descritos anteriormente fazem o gerenciamento de tarefas de nível de sistemas operacional ou de nível de usuário. No modelo de *threading* híbrido, ou $M{:}\,N$, um escalonador de aplicação controla a execução de M tarefas de nível de usuário em N *threads* de nível de sistema operacional, com $M > N$. Este modelo é o mais flexível, combinando as vantagens dos dois modelos, mas é o de implementação mais complexa, já que depende do trabalho combinado do escalonador do sistema operacional e da aplicação. Além disso, reintroduz a necessidade de sincronização caso as tarefas de usuário façam uso de memória compartilhada. Como vantagens, combina os benefícios da velocidade e baixo custo de criação e troca de contexto das *threads* a nível de usuário com a possibilidade de execução em paralelo das *threads* de sistema operacional.

Scheduler Activations (Anderson et al., 1991) são um exemplo deste modelo, provendo *threads* de nível de *kernel* com flexibilidade de *threads* de usuário, escalonáveis pela aplicação. Esta estratégia foi adotada no sistema operacional NetBSD, mas abandonada devido à complexidade de implementação.

Um outro exemplo deste modelo é o mecanismo *User-Mode Scheduling* (Microsoft, 2014) introduzido no Windows 7. Este mecanismo permite que aplicações gerenciem o escalonamento de suas *threads*, cada uma com sua própria pilha de execução. Com a troca de contexto mais leve, há um aproveitamento mais eficiente dos processadores quando há um grande volume de tarefas, mas o desenvolvedor precisa implementar um escalonador próprio.

Os atores da linguagem Scala (Haller e Odersky, 2009), por combinarem primitivas de *threads* e de eventos, também podem ser vistos como exemplo do modelo $M{:}\,N$. Uma vantagem neste caso é que o escalonador é abstraído pela biblioteca da linguagem. Um ator pode suspender a pilha de execução ao invocar `receive`, como no modelo de *threads*, e manter a pilha ao invocar `react`, como no modelo baseado em eventos.

A biblioteca `luaproc` da linguagem de programação Lua (Skyrme et al., 2008) é outro exemplo deste modelo. A biblioteca implementa um escalonador que gerencia a execução de múltiplas tarefas criadas pelo usuário. Ao utilizar os comandos definidos pela biblioteca para troca síncrona de mensagem, as tarefas são automaticamente suspensas ou retomadas de acordo com a disponibilidade de executores e das mensagens, de forma transparente para o programador. A complexidade de implementação do modelo $M{:}\,N$ é abstraída para o programador, permitindo que ele se beneficie das vantagens do modelo:

paralelismo multi tarefa, tarefas com baixo custo e simplicidade de raciocínio em cima do código propiciado pelo uso de corotinas.

2.3 Conclusão

Não podemos afirmar que algum dos modelos apresentados é superior aos demais. Cada um deles possui suas vantagens e desvantagens. Na tabela 2.1, resumimos as características de cada um dos modelos de concorrência multi-tarefa avaliados.

Tabela 2.1: Comparação dos modelos de concorrência multi-tarefa

	Escalonador	**Paralelismo**	**Custo tarefas**	**Preempção**
1:1	Sistema operacional	Sim	Alto	Sim
N:1	Usuário	Não	Baixo	Não
M:N	Híbrido	Sim	Baixo	Sim

O modelo $M:N$ é ainda pouco explorado em relação a aplicabilidade e limites e nossa contribuição se dá na implementação deste modelo em um sistema real. Como veremos neste trabalho, apesar das dificuldades inerentes à implementação de um escalonador no nível da aplicação, através deste modelo, é possível criar um sistema que permite ao usuário explorar o paralelismo multi-core com uma linguagem dinâmica sem sobrecarregar o programador com as complexidades naturais da preempção ou da programação orientada a eventos.

Nesta dissertação, descrevemos a implementação do modelo $M:N$ em um cenário controlado, onde só há um tipo de E/S. Alteramos o banco de dados Redis para permitir que *scripts* Lua executem de forma concorrente e paralela. Ao transformar os *scripts* em corotinas que cedem o controle de execução automaticamente ao acessarem o banco de dados e que retomam automaticamente quando o resultado está disponível, abstraímos as complexidades do escalonamento de tarefas para o programador. No capítulo seguinte, descreveremos o banco de dados Redis, sua implementação e relação com a linguagem de *script* Lua, antes de abordarmos as modificações realizadas.

3
Redis e scripts Lua

3.1
Redis

Redis é um banco de dados remoto chave-valor com armazenamento em memória. Frequentemente é visto como um servidor de estruturas de dados, por permitir operações eficientes em diversos tipos abstratos. O Redis é bastante popular em diversas áreas de aplicação, como principal meio de armazenamento ou como meio auxiliar mais rápido para outros bancos de dados (*cache*). Sua implementação é na linguagem C e é *single-threaded*. Isto é, atende todos os clientes com uma única *thread* de execução do sistema operacional através de uma arquitetura baseada em *loop* de eventos. É um servidor em rede, que recebe requisições de clientes através de uma porta TCP/IP ou de um *Unix socket*. Os comandos permitidos no Redis são abstrações simples que operam sobre cinco estruturas de dados: *strings*, listas, conjuntos, conjuntos ordenados e dicionários. Uma das características principais do servidor é a baixa latência, obtida principalmente pelo armazenamento principal em memória, sem garantia de durabilidade.

Existem duas formas opcionais de persistência no Redis[1]: arquivo de *journal* em disco (AOF) e *snapshots periódicos* (RDB). Nenhuma dessas formas provê garantia total de durabilidade, já que os comandos dos clientes são avaliados e os resultados retornados para o cliente antes de serem salvos. Para manter o tempo de resposta baixo, o Redis não espera que uma operação seja escrita no arquivo de *journal* em disco para considerá-la executada. Isso significa que o Redis, assim como outros bancos de dados não relacionais (*NoSQL*), não dá suporte completo a transações ACID, porque não há garantia de durabilidade. Quando a funcionalidade de *snapshot* é ativada, os dados são gravados periodicamente de forma assíncrona em disco. Esta gravação do *snapshot* em disco é a única situação na qual a implementação padrão do banco de dados faz uso de outra *thread* de sistema operacional além da *thread* principal.

[1] `http://redis.io/topics/persistence`

Para alta disponibilidade, um servidor Redis pode operar como réplica de um servidor mestre, agindo no modo escravo em *hot stand-by* (Tanenbaum e Steen, 2006). Neste caso, todos os comandos executados no mestre são replicados para o escravo. Para manter a latência de atendimento dos clientes baixa, o servidor mestre pode aceitar um comando e enviar a resposta para o cliente antes de o comando ser replicado para os escravos. Uma consequência negativa desta decisão é que, caso haja uma falha no servidor mestre, uma réplica poderá ficar desincronizada. Isto é, não há garantia de consistência entre as réplicas. Quado o Redis detecta a inconsistência do escravo, pode forçar uma re-sincronização total. Os servidores escravo também podem atender clientes, mas somente aceitam operações de leitura. Assim, também é possível melhorar o desempenho do sistema através da replicação ao se dividir a carga de leitura em múltiplos processos.

A última versão do Redis, 3.0, dá suporte a um modo de operação em *cluster* no qual o banco de dados opera com múltiplos servidores mestre que aceitam comandos de escrita, operando simultaneamente. Cada mestre pode ter uma ou mais réplicas. Se mais de uma réplica identificar que o mestre falhou, qualquer uma destas réplicas pode iniciar um processo de eleição. A réplica vencedora é então promovida para mestre. As chaves são fragmentadas entre os servidores através do algoritmo de *consistent-hashing* (Karger et al., 1997). Um cliente não precisa saber o nodo que contém a chave que deseja acessar. O cliente pode acessar qualquer servidor, mas, caso o servidor não seja responsável pela chave requisitada, o servidor retorna um erro para o cliente indicando qual outro nodo é responsável por esta chave. Deste modo, é possível armazenar mais conteúdo no sistema do que a memória disponível em um único servidor.

Por ser um banco de dados em memória com bom desempenho, o Redis é frequentemente utilizado como *cache*. É possível configurar um tamanho máximo de memória para o servidor e as políticas de liberação de memória. Por padrão, o algoritmo utilizado é uma aproximação de LRU (do inglês *Least Recent Used*, Menos Recentemente Utilizado), em que as chaves mais antigas, não acessadas há mais tempo, são apagadas primeiro.

O Redis dá suporte ao padrão de troca de mensagens PUB/SUB (do inglês *Publisher/Subscriber*, ou Publicador/Assinante). Neste padrão, ao invés de mensagens serem enviadas diretamente para receptores, publicadores enviam mensagens em tópicos e assinantes se inscrevem nos tópicos de interesse. Este padrão tem o benefício de desacoplar os remetentes dos destinatários, permitindo que publicadores possam enviar mensagem sem necessidade de conhecer os destinatários. Já os destinatários, isto é, assinantes de tópicos, não

precisam conhecer todos os rementes e podem filtrar as mensagens de interesse de acordo com seus tópicos. Neste contexto, o Redis faz o papel de *broker* para troca das mensagens, recebendo as mensagens e repassando para os assinantes interessados. A implementação do banco não provê persistência das mensagens e somente os assinantes conectados recebem as mensagens publicadas, sem possibilidade de consultarem mensagens anteriores.

Como o banco de dados tem uma implementação *single-threaded*, os comandos do sistema não são executados de forma concorrente e são naturalmente serializados. Assim, há garantia natural de atomicidade, isolamento e consistência no sentido ACID de todas as operações executados no Redis, apesar de não haver garantia de durabilidade. Essas garantias são válidas inclusive no com o uso de *scripts*, códigos em Lua enviados pelos usuários.

O código Lua dos *scripts* é avaliado de forma bloqueante e atômica. Ou seja, o atendimento de outros clientes é suspenso enquanto um *script* está executando. O fato do Redis não permitir a execução paralela de *scripts* simplifica a implementação do servidor e também permite que *scripts* executem múltiplos comandos de maneira atômica. No entanto, este comportamento impossibilita a execução de scripts de longa duração em ambientes com múltiplos usuários. Mesmo que os *scripts* façam apenas operações de leitura, o atendimento dos demais usuários é interrompido durante a avaliação destes *scripts*, o que pode acarretar uma degradação inaceitável do serviço.

3.2 Scripts Lua no Redis

Objetivo do suporte a scripts no Redis

Desde a versão 2.6, o Redis dá suporte à execução de *scripts* Lua submetidos pelo cliente. O primeiro motivo[2] para a introdução do suporte a *scripts* foi economia de banda e redução de latência. Muitas vezes, uma aplicação necessita executar múltiplos comandos para ler o banco de dados, processar alguma informação e escrever o resultado de volta no sistema. Este tráfego de ida e volta de dados entre o cliente e o servidor pode ser evitado ao se mover o processamento para o servidor, eliminando a latência do envio de múltiplos comandos e respostas na rede. Esta não é uma ideia nova e é largamente explorada em outros ambientes distribuídos, como por exemplo, no uso de *stored procedures* em bancos de dados relacionais.

O segundo motivo é que normalmente o gargalo do banco de dados Redis tende a ser E/S, não CPU. Mesmo nos casos em que a CPU é exigida, a maior

[2] `http://oldblog.antirez.com/post/redis-and-scripting.html`

parte do uso de CPU geralmente está no processamento do protocolo de rede, não nas operações sobre as estruturas de dados. Desta forma, outro benefício da execução de *scripts* é amenizar o gargalo de E/S, já que a lógica da aplicação remota passa a ser executada no servidor, fazendo com que menos comandos tenham que ser transmitidos pela rede.

Por fim, o terceiro motivo da introdução dos scripts foi permitir a execução atômica de operações para as quais o Redis não provê um comando específico. Apesar de existirem mais de 150 comandos na última versão estável do Redis, praticamente todos são abstrações simples em cima das estruturas de dados disponibilizadas. Cada uma destas operações executa de forma atômica, mas de acordo com a necessidade da aplicação, é necessário compor múltiplos comandos do Redis. Para executar estes diversos comandos de forma atômica, antes da introdução dos scripts, era necessário utilizar transações, fonte de complexidade adicional do ponto de vista do programador. Os comandos para manipulação de transações ainda estão disponíveis no banco de dados, mas seu uso é desencorajado pela dificuldade de serem usados corretamente. O Redis considera que uma transação falhou, e a reverte, somente quando uma chave monitorada explicitamente é modificada durante a transação.

Para ilustrar a complexidade do uso de transações no Redis, mostramos a seguir o código em Ruby necessário para incrementar um contador apenas se ele existir. O teste de existência da chave é necessário porque o comando `INCR` cria a chave com valor 1 caso ela não exista.

```
redis.watch :counter
if redis.exists :counter
  redis.multi
  redis.incr :counter
  redis.exec
else
  redis.unwatch
end
```

Este código, que utiliza os comandos `WATCH`, `MULTI`, `UNWATCH` e `EXEC` para controle da transação. Observamos que o comando `UNWATCH` não é necessário após o comando `EXEC`. Quando o comando `EXEC` é invocado, todas as chaves deixam de ser monitorados. Este código é equivalente ao seguinte *script* Lua:

```
if redis.call("exists", KEYS[1]) == 1 then
    return redis.call("incr", KEYS[1])
end
```

Com o uso de *scripts*, a combinação de comandos é muito mais simples, já que não há necessidade de se preocupar com condições de corrida. Os *scripts* são executados atomicamente, mesmo quando fazem chamadas a vários comandos. Desta forma, *scripts* podem ser vistos como comandos customizados, além dos nativos do servidor.

Interface de execução de scripts para os clientes

O Redis é um servidor acessível via rede e permite que clientes executem comandos[3] através de um protocolo próprio[4]. A implementação padrão do Redis também disponibiliza uma biblioteca para a construção de clientes com a linguagem *C* e uma interface de linha de comando REPL (do inglês *Read Eval Print Loop*). Existem bibliotecas desenvolvidas por terceiros disponíveis para construções de clientes em diversas linguagens[5].

O ponto de entrada para execução de *scripts* é um comando chamado `EVAL` que recebe como argumento o corpo de uma função Lua para execução. O Redis também permite que um *script* seja carregado e executado múltiplas vezes sem necessidade de reenvio do corpo da função Lua. Para isto, utiliza como identificador do *script* o *hash* SHA-1 do corpo da função. Não é permitido definir nomes para as funções. Abaixo, listamos todos comandos relacionados a *scripting* na implementação padrão:

`EVAL script numkeys key [key ...] arg [arg ...]`: Executa um *script* Lua no servidor. O comando recebe como argumentos o código do *script*, a quantidade de argumentos que são chaves do banco de dados, as chaves e demais argumentos para execução do *script*.

`EVALSHA sha1 numkeys key [key ...] arg [arg ...]`: Executa um *script* Lua pré-definido no servidor através dos comandos `EVAL` ou `SCRIPT LOAD`. Utiliza o *hash* SHA-1 do corpo do código do *script* como identificador e os demais argumentos são tratados como no comando `EVAL`.

`SCRIPT EXISTS script [script ...]`: Testa a existência de um *script* do servidor. Isto é, testa se ele já foi executado ou definido anteriormente.

`SCRIPT FLUSH`: Remove todos os scripts do cache do servidor

`SCRIPT LOAD script`: Define um script no servidor e retorna o seu identificador (*hash* SHA-1).

`SCRIPT KILL`: Interrompe a execução do script.

O apêndice A demostra uma sessão com estes comandos.

[3] `http://redis.io/commands`
[4] `http://redis.io/topics/protocol`
[5] `http://redis.io/clients`

Interface com o banco de dados para os scripts

Os *scripts* executados pelo Redis rodam em uma espécide de *sandbox*, com acesso restrito aos módulos padrão de Lua e sem direito de criar variáveis globais. O módulo `os`, por exemplo, não é disponibilizado para evitar que *scripts* acessem diretamente o sistema de arquivos. Os scripts têm acesso a um módulo exposto pelo banco de dados chamado `redis`. É este módulo que permite a interação com o banco de dados. As funções disponiblizadas por ele são:

- `call`, para execução de comandos Redis.
- `pcall`, para execução de comandos de modo protegido, retornando uma *string* com a descrição do erro em caso de falha.
- `log`, para mensagens de log.
- `sha1hex`, para cálculo do *hash* sha1.
- `error_reply`, para retorno de erro.
- `status_reply`, para retorno com *status* de sucesso.

3.3 Estados Lua para execução dos scripts

Lua é uma linguagem *extensível*, ou seja, é possível criar módulos em C que alteram o comportamento da linguagem. Também é uma linguagem de *extensão*, isto é, pode ser embutida em uma aplicação, permitindo que código em Lua amplie o comportamento da aplicação. De acordo com estas definições, Lua é usada no banco de dados Redis das duas maneiras, mas principalmente como uma linguagem de *extensão* da aplicação Redis. Apesar de expor aos *scripts* uma linguagem estendida com adição de bibliotecas, como, por exemplo, bibliotecas para manipulação de estruturas no formato `json` e `msgpack` e funções definidas em C para que os *scripts* possam interagir com a aplicação, os *scripts* são utilizados principalmente para estender o comportamento da aplicação.

Lua permite que múltiplos ambientes de execução, chamados de estados Lua, co-existam de forma isolada em uma mesma aplicação. Para que isso seja possível, Lua não mantém nenhuma variável ou estado global em variáveis da biblioteca. Todo estado é encapsulado em uma estrutura chamada `lua_State` e cada `lua_State` possui uma *thread* Lua de execução com sua própria pilha de execução. A interação entre C e a API de Lua é feita principalmente através de uma pilha (*stack*) de valores. É possível criar *threads* Lua (não de sistema operacional) adicionais em um mesmo `lua_State` através do

uso de corotinas, que possuem uma pilha de execução independente da *thread* Lua principal.

A implementação padrão de Redis cria um único estado Lua na sua inicialização e o utiliza para interpretar todos os comandos executados por quaisquer clientes. Cada *script* é criado como uma função global deste `lua_State`. É possível passar argumentos para os *scripts* através de dois tipos de variáveis: `ARGV` e `KEYS`. `KEYS` são argumentos especiais que devem ser utilizados para especificar as chaves do banco de dados que o script irá acessar. Quando em modo *cluster*, o sistema valida que as chaves passadas para o *script* através do argumento `KEYS` pertencem de fato ao servidor.

A linguagem Lua permite que variáveis globais sejam criadas dentro de funções. Para evitar que *scripts* interfiram uns com os outros através do uso compartilhado de variáveis globais, o Redis impõe uma restrição na criação destas variáveis. Esta restrição é criada através da modificação da *metatable* da tabela da variável de ambiente, impedindo a inserção de novas chaves nesta tabela.

A função `call` disponibilizada para os *scripts* é definida em C. Quando essa função é invocada pelo *script*, o sistema utiliza um cliente Redis virtual para execução do comando solicitado. Quando o resultado do comando é obtido, ele é convertido dos tipos nativos do Redis para um valor ou tabela Lua e finalmente retornado para o *script*. Quando o script retorna, os valores Lua retornados pelo *script* são serializados para o protocolo de rede e enviados para o cliente real que acionou a execução do *script*.

Figura 3.1: Diagrama de sequência para execução do comando EVAL.

A figura 3.1 é um diagrama do fluxo de execução de um *script*. Quando um usuário executa um comando EVAL, dispara um evento de E/S no Redis. O *loop* de eventos identifica que o comando é um comando de execução de *script*

e aciona a função `evalGenericCommand`. Esta função prepara o *script*, definindo-o como uma função Lua, e insere os argumentos do *script* na pilha do estado Lua. O banco de dados inicia o *script* através da chamada da API Lua `lua_call`.

Para interagir com o banco de dados, o *script* invoca a função Lua `redis.call`, que, por sua vez, invoca a função C do Redis chamada `luaRedisCallCommand`. Dentro desta função, o comando do banco de dados é executado e o resultado inserido na pilha do estado Lua. O resultado é sempre convertido para os tipos da linguagem. Como está ilustrado no diagrama, caso o retorno seja numérico, a conversão é feita através da função `lua_pushnumber`. A função C retorna então o controle para o estado Lua e o *script* prossegue a execução. O *script* pode fazer mais chamadas à função `call`, repetindo o ciclo.

Quando o *script* termina de executar, o controle retorna para a função `evalGenericCommand`. Esta função pega o resultado da pilha do estado Lua e o serializa para o protocolo Redis através da função `luaReplyToRedisReply`. Finalmente, o resultado é enviado via rede para o usuário. Durante todo este processo todos os usuários estão bloqueados e todas estas etapas são executadas de maneira síncrona.

4
Concorrência de scripts Lua

4.1
Motivação

Apesar da implementação padrão de *scripts* do Redis atender bem todos os objetivos iniciais descritos na seção 3.2, ela apresenta algumas limitações. Como já mencionado, todos os clientes são bloqueados enquanto um *script* está sendo processado. Como consequência, não é recomendável executar *scripts* de longa duração. Em certos cenários, mesmo quando o tempo de execução ultrapassa poucos milisegundos, seja porque os *scripts* executam muitos comandos no Redis ou por que fazem uso intensivo de CPU, a degradação no atendimento dos demais clientes não é tolerável. É principalmente por este motivo que a empresa Twitter não utiliza *scripts* nas suas dezenas de milhares de instâncias do banco de dados Redis (Yue, 2014). A possibilidade de execução de *scripts* de forma não bloqueante é uma funcionalidade requisitada com certa frequência nos fóruns do banco de dados[1]. Um dos objetivos deste trabalho é atender esta demanda, introduzido comandos que permitem que o usuário execute *scripts* de forma não bloqueante. Do ponto de vista do cliente, o comando continua sendo executado de maneira aparentemente bloqueante, com a resposta sendo retornada depois de certo tempo. Isto continua sendo perfeitamente aceitável, no entanto, adicionamos a capacidade do servidor processar comandos de outros usuários neste período. Também adicionamos a capacidade de execução de múltiplos *scripts* em paralelo.

Em alguns casos, a atomicidade dos *scripts* pode ser desejada. O *script* pode precisar a garantia de que nenhum outro cliente alterará as chaves que está manipulando. Em outros casos, esta garantia não é necessária e pode ser vantajoso que a interação dos *scripts* com o banco de dados seja assíncrona, de forma concorrente com outros *scripts* e clientes para que o banco de dados seja liberado para processar outros comandos. Como existem estes dois cenários possíveis de uso, optamos por permitir que o usuário possa escolher se o *script* será avaliado com ou sem necessidade de atomicidade e isolamento, isto é, de

[1] `https://groups.google.com/forum/#!topic/redis-db/qIvehBvufLM`

forma síncrona ou assíncrona.

A utilização de *scripts* concorrentes traz o benefício adicional do aproveitamento dos múltiplos núcleos do servidor. *Scripts* que fazem grande uso de CPU podem ser executados no servidor, sem risco de impacto para outros usuários. O paralelismo pode ser obtido através da criação de várias tarefas de sistema operacional, que potencialmente podem executar *scripts* distintos. Os *scripts* em si podem ser vistos como tarefas de nível de usuário, podendo ser suspensos quando interagem com o banco de dados, liberando as tarefas de sistema operacional para o processamento de outros *scripts*. Esta implementação segue o modelo multi-tarefa $M{:}\,N$ e será discutida em detalhes na seção 4.3. Na seção a seguir, discutimos a interface para acionamento dos *scripts*.

4.2 Interface para o usuário

Disponibilizamos os seguintes comandos no banco de dados para que o usuário possa invocar *scripts* de forma assíncrona:

`EVALASYNC script numkeys key [key ...] arg [arg ...]`: Executa um *script* Lua de maneira assíncrona. Os argumentos são idênticos aos de seu par `EVAL`.

`EVALSHAASYNC sha1 numkeys key [key ...] arg [arg ...]`: Executa um *script* Lua pré-definido no servidor de maneira assíncrona. Assim como seu par `EVALASYNC`, utiliza como identificador do *script* o *hash* SHA-1 retornado por `SCRIPT LOAD`.

Estes dois comandos são o ponto de entrada para as modificações desenvolvidas no Redis neste trabalho e têm comportamento muito similar aos seus pares `EVAL` e `EVALSHA`, já descritos na seção 3.2. A única diferença dos novos comandos é que sua execução é não bloqueante e a interação dos *scripts* com o banco de dados é assíncrona.

A execução é dita assíncrona em contraposição à execução bloqueante, que é síncrona. Quando um *script* é avaliado pelo comando `EVAL`, executa todos seus comandos de forma aparentemente instantânea, sem entrelaçamento com comandos de outros clientes. Na execução assíncrona, entre duas chamadas sucessivas do *script* para o banco de dados, comandos de outros clientes podem ser executados. Isto é, a execução de um *script* de forma assíncrona não é atômica e os comandos acionados por ele são executados de maneira entrelaçada com os comandos invocados por outros clientes.

Do ponto de vista do código, no entanto, as chamadas continuam aparentemente bloqueantes e síncronas. Quanto o *script* tenta interagir com o banco de dados, o escalonador suspende a execução do *script* e, quando o resultado

está disponível, continua a execução, mantendo a mesma pilha de execução, mas sem, no entanto, bloquear o servidor. Do ponto de vista do cliente que aciona o *script*, também não há diferenças aparentes. O comando retorna o resultado depois que a execução termina. Não é possível para um cliente isolado executando comandos sequenciamente distinguir os *scripts* assíncronos dos comandos padrão do Redis.

Não introduzimos nenhuma alteração para um único *script* usufruir diretamente do assincronismo. Futuros (Walker et al., 1990), por exemplo, permitiriam a execução de múltiplos comandos em paralelo por um único *script*. Nosso objetivo foi manter a compatibilidade total com os *scripts* síncronos. Também optamos por não introduzir um modo híbrido de execução, em que apenas alguns trechos do código sejam executadas de forma síncrona. Este tipo de funcionalidade poderia ser útil, mas essa necessidade não surgiu durante o desenvolvimento das aplicacações do capítulo 5.

Outra decisão de projeto que tomamos foi não criar nenhuma interface para os *scripts* se comunicarem diretamente. Toda comunicação deve ser feita através do próprio banco de dados. Não vimos necessidade de os *scripts* trocarem mensagens porque o Redis já possui estruturas de dados de lista e canais PUB/SUB que podem ser usados como filas de mensagens. Através destas estruturas é possível fazer os *scripts* se comunicarem entre si ou com outros clientes do banco de dados.

Optamos também por não implementar compartilhamento de memória entre os *scripts* porque as chaves do banco de dados já fazem este papel naturalmente. O banco de dados provê os comandos necessários para acesso atômico às estruturas de dados que disponibiliza e assim evitamos a necessidade de sincronização explícita nos *scripts*, que seria necessária devido à existência de preempção de tarefas no modelo híbrido $M:N$. Sob o ponto de vista dos *scripts*, o banco pode ser considerado como uma espécie de *tuple-space*, através do qual os scripts podem interagir.

4.3 Implementação

Nossa implementação segue o modelo de *threading* híbrido $M:N$ descrito no capítulo 2. Cada *script* representa uma das M tarefas de nível de usuário e criamos um *pool* de N trabalhadores para executar estas tarefas. Cada *worker* é uma *thread* de sistema operacional e uma estratégia de escalonamento simples distribui as tarefas.

Na seção 3.3, vimos como os *scripts* são definidos e executados como funções em um estado Lua global na implementação padrão do Redis. As

principais alterações que fizemos foram a criação de múltiplos estados Lua ao invés de um único estado global e a execução de *scripts* em corotinas ao invés de funções. As corotinas Lua (Moura e Ierusalimschy, 2009) são *threads* de nível de usuário com preempção colaborativa, o que permite suspender os *scripts* quando eles interagem com o banco de dados.

A linguagem Lua permite a criação de múltiplas corotinas em um estado Lua, possibilitando arquiteturas multitarefa sem preempção, isto é, modelo $N{:}\,1$. A execução de cada corotina Lua não pode ser feita por tarefas de sistema operacional distintas, mas a API em C da linguagem também permite que sejam criados múltiplos estados Lua isolados que podem executar em paralelo, desde que a execução de funções ou corotinas em cada estado seja iniciada por tarefas de sistema operacional diferentes. Como os estados são isolados, não há compartilhamento de variáveis e toda comunicação entre os estados deve ser intermediada pela aplicação. Utilizando esta API, bibliotecas podem implementar *multithreading* preemptivo. (Ierusalimschy, 2013) sugere uma implementação deste modelo utilizando troca de mensagens para comunicacão entre processos Lua. A biblioteca luaproc (Skyrme et al., 2008) é uma evolução desta implementação. Nossa implementação é bastante próxima à implementação da biblioteca `luaproc`. Nós criamos múltiplos estados Lua e um escalonador no nível de aplicação controla múltiplas tarefas de nível usuário, de acordo com o modelo $M{:}\,N$. Diferentemente da biblioteca `luaproc`, no entanto, na nossa implementação não há interação direta entre os estados Lua. A comunicação é feita sempre através do banco de dados.

A implementação padrão do Redis utiliza um *loop* de eventos que processa continuamente comandos invocados pelos clientes. Um *script* é tratado como um único comando, então o *loop* de eventos só trata qualquer outra requisição ao terminar de processar todo o *script*. Para permitir a execução de *scripts* de forma assíncrona, ainda dentro do *loop* de eventos, armazenamos qual foi o comando chamado (`EVALASYNC` ou `EVALASHASYNC`), assim como os argumentos do comando, em uma estrutura denominada `task` e a inserimos em uma fila global. A seguir, associamos um estado Lua à tarefa recém criada. Caso exista, utilizamos um `lua_State` livre e em caso contrário criamos um novo para execução do *script*.

Workers são criados na inicialização do banco de dados e buscam continuamente tarefas para execução da fila global. Quando um *worker* obtém uma tarefa da fila, executa a corotina que está definida no estado Lua até o *script* retornar o resultado ou tentar interagir com o banco de dados. Esta estratégia é similar ao padrão de *thread pool*; no entanto, neste caso, as tarefas podem ceder o controle e serem suspensas. Isto acontece quando o *script* tenta

acessar o banco de dados através de uma chamada a `call` e o *worker* não consegue obter o *lock* global para execução do comando. Neste caso, o *worker* suspende a execução do *script* e reinsere a tarefa na fila global. Quando um *worker* obtém uma tarefa suspensa, adquire o *lock* para execução do comando do banco de dados e depois faz `resume` do `lua_State`, continuando a execução do *script*. Outras estratégias de escalonamento além da obtenção forçada do *lock* na segunda tentativa foram testadas, mas nos nossos testes tiveram desempenho inferior. Testamos, por exemplo, a criação de um *worker* exclusivo para interação com o banco de dados em uma fila independente.

Este *lock* global foi necessário porque as estruturas de dados internas do Redis **não** são *thread-safe* e por isso precisamos garantir que dois comandos não sejam executados simultaneamente. Se o *loop* de eventos tratasse a requisição de um cliente enquanto um *script* Lua executa um comando em outra *thread* de sistema operacional, as estruturas de dados do Redis poderiam ser corrompidas. Para fazer esta sincronização, usamos exclusão mútua entre os *workers* e o *loop* de eventos, que continua rodando em uma única *thread* de sistema operacional.

Também adicionamos exclusão mútua em algumas estruturas de dados globais, como *seed* de números aleatórios e horário do servidor. Outro ponto de sincronização necessário foi no *buffer* de escrita para os clientes. É feita exclusão mútua durante a serialização do resultado de cada *script* com o *loop* de eventos que envia o *buffer* para cada cliente. Com a ferramenta Valgrind[2], certificamos que nenhuma outra estrutura de dados global poderia ser acessada em paralelo, gerando potenciais condições de corrida. A ferramenta verifica se endereços de memória compartilhada são acessados por *threads* distintas sem o uso de *locks* e também verifica se os *locks* são adquiridos e liberados na ordem correta.

Quando o *script* retorna um valor, serializamos o resultado e o enviamos para o cliente. A tarefa é então excluída e o estado Lua liberado para reutilização. A reutilização de estados Lua evita o custo relativamente alto de criação destas estruturas (Skyrme et al., 2008).

O diagrama da figura 4.1 resume como se dá a execução de um comando assíncrono. Diferentemente do diagrama da figura 3.1, a execução de `lua_call` é feita pelo *worker*, fora do *loop* de eventos principal. Adicionalmente, quando o *script* precisa interagir com o banco de dados ele cede a vez (`yield`) e o escalonador suspende sua execução. Quando o comando é executado por outro *worker*, o *script* é reiniciado. O passo a passo abaixo descreve este processo em maiores detalhes.

[2] `http://valgrind.org/`

1. Um cliente executa um comando `EVALASYNC`
2. O *loop* de eventos aciona uma função C responsável pelo comando, que lê o script e os argumentos passados pelo cliente (`evalGenericCommand`).
3. O banco cria uma *task* para o *script* e armazena os comandos passados pelo cliente
4. *Worker* obtém tarefa, define uma corotina Lua no `lua_State` e a invoca.
5. O *script* Lua invoca a função `redis.call`
6. O comando e os argumentos passados para a função `redis.call` são salvos. A função C `redis.call` retorna `LUA_YIELD`.
7. A função C que invocou o *script* Lua recebe como retorno `LUA_YIELD`.
8. A tarefa é suspensa e reinserida no final da fila.
9. Um outro *worker* obtém a tarefa novamente.
10. O *worker* obtém o *lock* de exclusão mútua e executa o comando pendente.
11. O *worker* devolve o controle para a função C `redis.call`, retomando a corotina.
12. A função de continuação, finalmente converte o resultado para o Lua e devolve o controle para o *script*.

Figura 4.1: Diagrama de sequência para execução do comando EVALASYNC.

13. Se o script invocar a função redis.call novamente, retorna para o passo 6.

14. O script termina, e o controle volta para a função C que invocou a corotina Lua.

15. O resultado é serializado e enviado para o cliente.

Nas subseções a seguir, descrevemos em maior detalhe alguns pontos que exigiram atenção para manter a corretude do banco de dados em todos os cenários.

4.3.1 Encerramento do servidor

Quando o servidor é encerrado ou quando um usuário executa o comando SCRIPT FLUSH, que limpa todos os *scripts* registrados, uma tarefa terminadora (*poison pill*) é inserida para cada *worker* existente. Quando um *worker* recebe uma destas tarefas especiais, ele encerra o *Lua state* e a *pthread* onde roda. Isto também garante que as tarefas assíncronas não serão interrompidas antes de retornarem e que nenhum comando `EVALASYNC` ficará pendente.

4.3.2 Compartilhamento de scripts entre estados

O Redis permite que um *script* seja carregado através do comando `SCRIPT LOAD`. Este comando retorna o *hash* SHA-1 do *script* como seu identificador. O *script* pode então ser avaliado através do comando `EVALASHA`. Internamente, na implementação padrão do Redis, quando um *script* é definido, o banco de dados cria uma função global no estado Lua com o identificador igual ao *hash* SHA-1 do seu código prefixado por `f_`. Quando o usuário invoca o comando `EVALASYNCSHA`, o banco verifica se a função com nome apropriado existe no estado Lua e então a invoca com os parâmetros fornecidos pelo usuário.

A introdução de múltiplos Lua States faz com que seja necessário definir o mesmo *script* em diferentes estados Lua. Quando o comando `EVALASYNCSHA` é invocado e executado em um estado Lua diferente daquele em que o *script* foi definido a primeira vez, não basta testar se a função com o nome apropriado existe. Para contornar este problema, incluímos o seguinte passo na chamada de um *script* através de seu *hash*: quando a função não existe no Lua State, verificamos se o *script* existe no dicionário global dos *scripts* disponíveis. A implementação padrão mantém este dicionário para que novas réplicas do banco possam sincronizar os *scripts* existentes. Caso o *script* realmente

esteja carregado, a função é re-definida no Lua State desejado. Existe uma necessidade de sincronização adicional para acesso deste dicionário, já que os *workers* só possuem controle de exclusão mútua na execução de comandos do banco de dados.

4.3.3 Replicação

Vimos anteriormente que o Redis dá suporte à replicação para aumentar a disponibilidade do sistema. Neste modo de operação, uma única instância *master* recebe todas as escritas e envia para uma ou mais instâncias *slave* os comandos com argumentos que foram executados. A replicação é feita com a garantia de que os comandos serão executados na mesma ordem nas instâncias *slave*, de modo que todas sempre atinjam o mesmo estado final.

Na implementação padrão do Redis, os comandos síncronos `EVAL` e `EVALSHA` são replicados como comandos comuns. O próprio *script* e seus argumentos são re-interpretados nas réplicas. Para que isso não cause inconsistência no estado das réplicas, os *scripts* devem ser determinísticos, no sentido que, dado um estado inicial e os parâmetros de entrada, os *scripts* devem sempre fazer as mesmas operações, resultando em um mesmo estado final. É importante observar que todos os demais comandos do Redis são naturalmente determinísticos. *Scripts* que não executaram operações de escrita, assim como os comandos nativos de leitura, não são replicados porque não alteram este estado global. Diversas regras foram adicionadas ao banco de dados para tentar garantir a idempotência dos *scripts*:

- Não é permitido que scripts façam comandos de escrita depois de gerar um número pseudo-aleatório através da biblioteca `random`.
- Alguns comandos do Redis podem determinar que uma chave expire automaticamente depois de um certo intervalo de tempo, relativo à execução do *script*. Para que o tempo de processamento do *script*, que pode variar de uma máquina para outra, ou de uma execução para outra, não insira não determinismo, o horário de execução de cada comando é transferido entre a instância *master* e as réplicas. Além disso, todos os comandos invocados a partir do *script* são interpretados como se houvessem sido executados no mesmo *timestamp* de inicialização do *script*.
- Depois de executarem qualquer comando de escrita, os scripts não podem mais ser interrompidos através do comando `SCRIPT KILL`.

Esta série de regras mostraram na prática ser suficientes para garantir a sincronização dos comandos bloqueantes, mas não são suficientes para garantir a replicação correta dos novos comandos `EVALASYNC` e `EVALSHAASYNC`. Como os comandos não são executados de maneira atômica, a concorrência introduz um não determinismo natural. Se um *script* estiver sendo executado concorrentemente com comandos ou *scripts* de outros clientes, a ordem em que os comandos do *script* e dos demais clientes será processada é indeterminada. Desta forma, a replicação do *script* na réplica não pode ser feita simplesmente reavaliando o *script*, já que o estado do sistema poderia ser alterado durante a execução do *script*.

Para ilustrar este problema, suponha o seguinte exemplo, que transfere 10 unidades da chave A para a chave B caso a chave A possua valor suficiente, isto é, maior que 10.

```
if tonumber(redis.call('get', 'A')) > 10 then
    redis.call('decrby', 'A', 10)
    redis.call('incrby', 'B', 10)
end
```

E um segundo script de define o valor de A como 0:

```
redis.call('set', 'A', 0)
```

Se a variável `A` possuir o valor 10 e `B` possuir o valor 0 e os dois *scripts* forem invocados simultaneamente através do comando `EVALASYNC`, diversos estados finais (`A`, `B`) podem ser obtidos: (0, 0), (0, 10) ou (-10, 10). Cada réplica poderia atingir um destes estados distintos, causando inconsistência entre elas.

Para fazer a replicação dos comandos de avaliação assíncrona de *scripts* deterministicamente, ao invés de executar os *scripts* nos *slaves*, ao invés de adicionarmos mais regras ou tentarmos contornar o problema, adotamos a solução mais simples e natural: replicamos as chamadas ao banco de dados na ordem em que são realizadas, independentemente do *script* de origem. Por exemplo, o servidor poderia enviar para a réplica a seguinte ordem de operações, ao invés dos dois *scripts* anteriores: SET A 0; DECRBY A 10; INCRBY B 10, resultando sempre no estado final (-10, 10). Esta solução é menos propensa a falhas que a solução adotada pela implementação padrão do banco de dados, mas é potencialmente mais cara, uma vez que uma quantidade grande de comandos pode ser executada por um *script*.

As restrições descritas acima foram criadas para garantir que as réplicas do banco atinjam o mesmo estado final após a execução dos *scripts*. Como fazemos a replicação da mescla de comandos executados por *scripts* assíncronos com os demais comandos executados na mesma ordem de entrelaçamento

em que os comandos foram executados originalmente no servidor mestre, há garantia de que as réplicas sempre atingirão o mesmo estado final, sem necessidade de imposição das regras impostas para os *scripts* bloqueantes.

4.3.4 Controle de timeout

Como o servidor não executa outros comandos enquanto um *script* está sendo interpretado, o Redis oferece um mecanismo de *timeout* para que, caso um script demore muito para finalizar ou entre em *loop* infinito, se possa interromper a execução dele através do comando SCRIPT KILL.

Não estendemos essa funcionalidade de *timeout* para os scripts assíncronos, uma vez que eles não deixam o servidor ocupado. Também não estendemos o comando SCRIPT KILL para interrupção de comandos assíncronos devido à falta de uma forma de identificar cada *script* unicamente.

4.3.5 Testes automatizados

O Redis possui uma extensa suíte de testes de integração automatizados que verifica todo o comportamento funcional do banco de dados. Os testes são escritos na linguagem de programação TCL, que dá suporte à orientação a eventos. Processos servidores são criados e múltiplos clientes assíncronos executam comandos nestes servidores.

Tomamos a devida preocupacão de manter os testes existentes corretos e também adicionamos testes de integração adicionais para os comandos implementados. Basicamente, os novos testes verificam que os comandos `EVALASYNC` e `EVALSHAASYNC` se comportam corretamente de forma isolada e concorrente.

4.4 Processamento de streams de dados

A introdução da execução paralela de *scripts* no servidor abre a possibliidade de exploração de um outro paradigma, o de processamento de *streams* de dados em tempo real. Trabalhos recentes nesta área, como MillWheel (Akidau et al., 2013) e Storm (Toshniwal et al., 2014) permitem o processamento de um fluxo de informação através de grafos acíclicos direcionados (DAG, na sigla em inglês) de processamento. Os usuários destes sistemas podem especificar o código de execução de cada nó e a topologia do grafo. É possível criar um sistema equivalente, embora não distribuído, no Redis ao explorarmos o suporte

do modelo PUB/SUB combinado com a avaliação de *scripts* Lua assíncronos introduzida neste trabalho.

Visualizamos cada *script* como um vértice da DAG de processamento. A comunicação entre *scripts*, representada pelas arestas do grafo, é feita através de canais PUB/SUB. Cada *script* pode ter múltiplas entradas, se inscrevendo (SUB) em múltiplos canais. As saídas são feitas através da publicação (PUB) por parte do *script* em um ou mais canais.

Na implementação padrão do Redis, *scripts* Lua já possuem a capacidade de publicar mensagens, da mesma forma que usuários comuns, através do comando `PUBLISH`, cujos parâmetros são o nome do canal e a mensagem. Qualquer usuário pode se increver em um canal através do comando `SUBSCRIBE`, que aceita como argumento o nome de um ou mais canais. O Redis se encarrega de republicar todas as mensagens recebidas para os assinantes incritos no canal. Depois de se inscrever em um canal, o cliente entra em modo assinante e passa a receber as mensagens assincronamente. O cliente em modo assinante não deve invocar mais comandos, a não ser comandos de inscrição ou desinscrição em canais.

Para construir o grafo de processamento que desejamos, é necessário garantir que as mensagens sejam entregues para todos os nós, mas a implementação padrão do Redis não provê esse tipo de garantia para os assinantes. Não há persistência das mensagens: clientes não podem recuperar mensagens que foram publicadas enquanto não estavam conectados. Este problema pode ser contornado ao se mover os assinantes para dentro do servidor. *Scripts*, no entanto, não podem se inscrever em canais na implementação padrão do Redis. Isto não foi implementado devido à incompatibilidade da natureza assíncrona do envio das mensagens e da execução síncrona e bloqueante dos *scripts*.

4.4.1 Interface

Com o suporte a *scripts* assíncronos que introduzimos neste trabalho, poderíamos permitir que *scripts* executassem o comando `SUBSCRIBE` e entrassem em modo assinante, recebendo as mensagens de forma assíncrona. Optamos, no entanto, por facilidade de uso e de implementação, em adicionar um comando para inscrever *scripts* em canais. O comando foi batizado como `SSUBSCRIBE` e permite inscrever um *script* como assinante de um canal. O banco de dados executa o *script* de forma assíncrona quando qualquer *publisher* publica uma mensagem no canal. Também adicionamos o comando `SUNSUBSCRIBE` para desinscrever *scripts* de canais. A listagem abaixo demonstra o uso destes comandos em uma seção no terminal. Os comandos são

entrados após a seta > e o símbolo ↪ representa uma queda de linha.

Primeiramente, carregamos um *script* e recebemos o seu identificador SHA-1. O *script* incrementa a chave `chamadas` toda vez que é executado.

```
127.0.0.1:6379> SCRIPT LOAD 'redis.call("incr", "chamadas")'
"c1f1242a3e4613130654d96ae85b88457d6ce4c8"
```

Verificamos então que o contador na chave `chamadas` não possui valor associado:

```
127.0.0.1:6379> GET chamadas
(nil)
```

Quando executamos o *script* pela primeira vez através do seu identificador, o valor da chave passa a ser 1.

```
127.0.0.1:6379> EVALSHA
   ↪ c1f1242a3e4613130654d96ae85b88457d6ce4c8 0
(nil)
127.0.0.1:6379> GET chamadas
"1"
```

Inscrevemos então o *script* no canal `channel`.

```
127.0.0.1:6379> SSUBSCRIBE
   ↪ c1f1242a3e4613130654d96ae85b88457d6ce4c8 channel
OK
```

Quando publicamos a mensagem "message"neste canal, verificamos que o contador `chamadas` é incrementado.

```
127.0.0.1:6379> PUBLISH channel message
(integer) 1
127.0.0.1:6379> GET chamadas
"2"
```

Finalmente, desinscrevemos o *script* do canal e verificamos que o contador não é alterado, confirmando que o *script* não foi executado novamente.

```
127.0.0.1:6379> SUNSUBSCRIBE
   ↪ c1f1242a3e4613130654d96ae85b88457d6ce4c8 channel
(integer) 1
127.0.0.1:6379> PUBLISH channel message
(integer) 0
127.0.0.1:6379> GET chamadas
"2"
```

4.4.2 Implementação

A implementação destes dois novos comandos se resume a algumas alterações pontuais. Criamos um tipo novo de *subscriber* para representar *scripts*, ao invés de clientes na rede. Toda vez que um cliente publica uma mensagem em um canal que possui um assinante do tipo *script*, o banco de dados cria uma tarefa de execução assíncrona de *scripts* e a adiciona na fila para execução. O Redis executa o *script* passando como primeiro argumento o nome do canal e como segundo argumento a mensagem.

Como foi descrito nesta seção, através destes dois comandos, é possível construir grafo de processamento no Redis com Lua. Na seção 5.3, descrevemos uma pequena aplicação desenvolvida para validar este cenário de uso.

5
Resultados experimentais

Analisaremos os benefícios das modificações realizadas do ponto de vista de três cenários de uso:

- ***scripts* de longa duração**: São *scripts* que fazem alto uso de CPU e que não poderiam ser executados de forma bloqueante. Comparamos o desempenho da execução assíncrona com a implementação original na execução de um *script* para multiplicação de matrizes. Também desenvolvemos e avaliamos um sistema para detecção de anomalias.
- **aceleração de *scripts***: Neste cenário, o objetivo é diminuir o tempo de execução de uma tarefa explorando o paralelismo. Para representar este tipo de aplicação, implementamos um algoritmo paralelo de quadratura adaptativa, que calcula a integral definida de uma função através da soma da área de trapézios.
- **processamento em *tempo real***: Neste caso, desejamos processar um fluxo de dados contínuo sem bloquear o recebimento de informação. Para validar esta situação, criamos um detector de tópicos populares para a rede social Twitter.

As medidas de tempo de execução, exceto quando mencionado, foram realizados em uma instância do tipo c4.8xlarge da Amazon Web Services[1]. São máquinas virtuais com 36 vCPU Intel Xeon E5-2666 v3 (Haswell) de 2.9GHz e 60 Gb de memória RAM. O sistema operacional utilizado foi Amazon Linux AMI 2015.03[2], baseado em Red Hat 4.8.2-16. O Redis foi compilado com gcc 4.8.2 e utilizamos um *pool* de 15 *workers*[3] para tarefas assíncronas

[1] `https://aws.amazon.com/ec2/instance-types/`
[2] `https://aws.amazon.com/amazon-linux-ami/`
[3] Como a máquina possui 36 núcleos virtuais, poderímos potencialmente executar até 35 *workers* em paralelo, mas optamos por manter alguns núcleos ociosos.

5.1 Execução de scripts de longa duração

Scripts de longa duração bloqueiam o servidor, impedindo que outros clientes sejam atendidos. Neste primeiro cenário, temos por objetivo permitir que múltiplos *scripts* sejam executados em paralelo sem degradação significativa do atendimento dos demais clientes.

5.1.1 Benchmark básico

Nosso primeiro teste procura determinar qual a perda de desempenho devido ao custo adicional de sincronização que introduzimos para possibilitar a execução assíncrona. Executamos 10000 requisições através de 50 clientes concorrentes em cada teste através do comando de `redis-benchmark` da própria aplicação. O primeiro *script* representa um caso mínimo, que não interage com o banco e apenas retorna o valor 0. O segundo *script* faz apenas uma chamada ao banco, incrementando um contador. Os resultados, medidos em milhares de requisições por segundo, estão relatados na tabela 5.1.

Tabela 5.1: Comparação de desempenho de *scripts* bloqueantes

Script	**Implementação padrão**	**EVAL**	**EVALASYNC**
Retorna 0	132 kreq/s	114 kreq/s	92 kreq/s
Incremento	103 kreq/s	101 kreq/s	73 kreq/s

Observamos um taxa de 14% inferior de requisições na utilização do comando `EVAL` no primeiro teste e 2% inferior no segundo. Esta perda de desempenho em relação à implementação padrão é devida aos mecanismos de sincronização introduzidos no *loop* de eventos da aplicação. Na utilização do comando asssíncrono, os números são respectivamente 31% e 29%. A diferença na perda é levemente inferior (2%) devido ao custo de criação e escalonamento das tarefas. Nestes casos extremos, não há nenhum ganho na utilização de *scripts* assíncronos.

Os benefícios da execução assíncrona são mais aparentes em casos mais típicos, em que os *scripts* fazem múltiplas chamadas ao banco de dados e utilizam mais CPU. Como exemplo mínimo deste caso, desenvolvemos um *script* para múltiplicar matrizes quadradas, que armazenamos em estruturas de dados de lista do Redis. Este exemplo pode parecer artificial, mas o armazenamento de grandes matrizes não é incomum no Redis. Na globo.com, uma estrutura similar é utilizada para armazenar a matriz de preferências de usuários por vídeos (Pereira et al., 2014). O código está listado abaixo.

```
local A = redis.call('LRANGE', KEYS[1], 0, -1)
local B = redis.call('LRANGE', KEYS[2], 0, -1)
local result = KEYS[3]
redis.call('DEL', result)
local n = math.sqrt(#A)

for j = 0,n - 1 do
    for i = 0,n - 1 do
      local v = 0
      for k = 0, n - 1 do
        local Ai, Aj = k, j
        local Bi, Bj = i, k
        local product = A[Ai + (Aj * n) + 1] *
                        B[Bi + (Bj * n) + 1]
        v = v + product
      end
      redis.call('rpush', result, v)
    end
end
return n
```

Executamos este *script* com o comando `EVAL`, com o comando `EVALASYNC` e com o comando `EVAL` na implementação padrão, sem nossas modificações. A figura 5.1 ilustra a quantidade de requisições por segundo, medida novamente com o aplicação `redis-benchmark` com 50 clientes paralelos, variando a dimensão das matrizes quadradas de 2 até 100. A diferença de desempenho entre o comando `EVAL` do Redis com nossas modificações e o da implementação padrão é de apenas 2.6% no pior caso e seria indistinguível no gráfico e por isso incluímos na figura apenas o desempenho dos comandos `EVAL` e `EVALASYNC` com nossas modificações. Para matrizes pequenas, o taxa de requisições por segundo atendida pelos comandos é similar (diferença abaixo de 5% para matrizes de dimensão 2). Quando o custo computacional cresce, com o aumento do tamanho das matrizes, a quantidade de requisições/s atendidas com o comando `EVALASYNC` passa a ser bastante superior (9,5 vezes maior para uma matriz de dimensão 100).

Na figura 5.2, plotamos a variação no número de requisições atendidas na múltiplicação de matrizes 100x100 à medida que se aumenta o tamanho do *pool* de *threads*. Observamos uma aceleração praticamente linear até 10 núcleos.

Figura 5.1: Comparativo de desempenho na multiplicação de matrizes, variando o tomanho da matriz

5.1.2 Detecção de anomalia

Os testes descritos na subseção anterior demonstram superioridade do comando `EVALASYNC` quando há uso de CPU significativo, mas em um exemplo artificial. Nesta subseção, discutiremos uma aplicação real desenvolvida para validar o caso de uso de scripts de longa duração no Redis. Implementamos um detector de anomalias em tempo real para séries temporais. A detecção de anomalias consiste em encontrar padrões de dados que não se encaixam no comportamento esperado e tem diversas aplicações, como detecção de intrusão e fraudes. Uma das grandes dificuldades da construção de um algoritmo para detecção de anomalias é que uma anomalia é, por definição, um comportamento inesperado. Isto torna o algoritmo difícil de ser treinado. Normalmente, utilizam-se métodos não supervisionados de treinamento (Chandola et al., 2009).

Nossa implementação foi desenvolvida visando a detecção de anomalias em *datacenters*. O algoritmo escolhido, no entanto, é genérico, e pode ser utilizado em outros contextos.

Abaixo, listamos algumas das dificuldades apresentadas na detecção de anomalias neste tipo de aplicação:

– O sistema deve ser capaz de armazenar um grande volume de informação: Datacenters comerciais frequentemente possuem milhares de servidores e cada servidor pode ser representado por uma série temporal multi-

Figura 5.2: Desempenho na multiplicação de matrizes variando a quantidade de núcleos.

dimensional (CPU, uso de memória, etc). Devido à sazonalidade natural da utilização das aplicações em um data center, que pode variar de acordo com a hora do dia ou com o dia da semana, se requer que pelo menos o período de algumas semanas seja armazenado. Sendo assim, o volume total de dados coletados pode facilmente chegar na ordem de milhões de pontos.

– Os algoritmos para detecção de anomalia confiáveis exigem um esforço computacional razoável, mas com um baixo tempo de resposta.

– O processamento para detecção de anomalias não pode aumentar significativamente a latência da inserção de métricas.

Implementação

Construímos uma aplicação para demonstrar como nossa modificação do Redis é capaz de satisfazer esse conjunto de requisitos. O código desta aplicação está disponível em `https://github.com/jbochi/mgof`. Embora a aplicação seja compatível com a implementação padrão do Redis, que permite a coleta e armazenagem de um grande volume de métricas com uma baixa latência, quando alguma série temporal é processada em busca de anomalias, o bloqueio durante a execução do *script* impacta na latência da aplicação para coleta de dados. Só é praticável utilizar esta aplicação com a nossa versão de Redis com suporte a *scripts* concorrentes.

Apesar do alto volume de informação que necessita ser armazenado pelo sistema, os clientes da aplicação estão tipicamente interessados em saber apenas se existe uma anomalia ou não em uma dada métrica ou servidor. Uma aplicação externa ao banco de dados necessitaria acessar toda a série temporal para fazer o processamento dos dados, mas os *scripts* Lua permitem que esse grande volume de informação seja processado, retornando uma resposta sumarizada (presença de anomalia ou não). Isto minimiza o tráfego de dados na rede. Nossa modificação do Redis permite que isto seja feito de forma não bloqueante, não interferindo na coleta de dados.

O algoritmo escolhido (Wang et al., 2011) para nossa implementação divide a série temporal em um conjunto de janelas de tempo configurável. Cada valor da série é dividido em um número também configurável de faixas, por padrão 10. Para se detectar a presença de anomalia na janela de tempo atual, o algoritmo compara a distribuição dos pontos em cada faixa na última janela com as distribuições das janelas anteriores. A janela é considerada anômala se sua distribuição não puder ser estatisticamente "explicada"pela distribuições anteriores através de um teste de chi quadrado.

A interface da aplicação tem apenas duas entradas. A primeira é a função de inserção de um ponto, que recebe como argumentos métrica, valor e timestamp. A segunda, detecção de anomalia, recebe como único argumento o nome da métrica e retorna se a anomalia está presente ou não. A detecção de anomalia também aceita parâmetros opcionais de configuração, como tamanho das janelas, número de faixas para classificação dos valores da série e nível de confiança mínimo para indicar que uma anomalia está presente.

Para executar a detecção da anomalia, registramos no banco de dados um *script* que aceita como argumento a métrica a ser analisada. Este *script* verifica o valor máximo e mínimo da série e classifica os pontos da última janela no número de faixas configurado. A seguir, calcula a distribuição dos pontos nessas faixas e compara essa distribuição com a distribuição das janelas anteriores. Para evitar que a distribuição de cada janela tenha que ser recalculada, o *script* armazena a distribuição de cada janela e a quantidade de vezes que cada janela foi utilizada para classificar outras janelas. Quando a janela atual tem uma distribuição similar à distribuição ocorrida em um número mínimo de janelas, o *script* retorna que a janela atual não é anômala. A quantidade de vezes que uma distribuição deve ocorrer para ser considerada normal também é configurável e por padrão é 1. Quando não existe uma distribuição anterior similar, o *script* retorna que a anomalia foi detectada.

Escolhemos armazenar as séries temporais como conjuntos ordenados do Redis. Cada elemento do conjunto possui um `score` que é utilizado

na ordenação dos elementos. Utilizamos como `score` da série temporal o `timestamp` da métrica. O valor de cada elemento do conjunto deve ser único. Por isso, utilizamos como valor uma tupla com o `timestamp` e valor da métrica. No Redis, não existe uma outra estrutura ordenada, como uma fila, que permita valores duplicados.

Resultados

Implantamos esta aplicação para monitorar um conjuntos de poucas centenas de instâncias de bancos de dados utilizados no portal globo.com. Utilizamos como métricas a porcentagem de uso de CPU e de memória de cada instância de banco de dados. Durante nossos testes, identificamos uma instância que estava operando de forma anômala antes de qualquer incidente ser relatado, comprovando a eficácia do algoritmo.

Em termos de desempenho, a aplicação analisa uma janela de uma hora de uma série temporal de um mês, com um ponto a cada minuto, em 530ms em um *laptop* de uso pessoal[4]. A aplicação mantém em *cache* computações intermediárias e requisições subsquentes são processadas em 90ms, em média. Embora sejam tempos relativamente baixos, não é apropriado bloquear o servidor por períodos tão longos caso se esteja monitorando muitas séries temporais distintas.

5.2 Aceleração da execução de scripts

Neste caso, temos como objetivo a aceleração da execução de um *script* individual. O objetivo é dividir a tarefa e executar diversos *scripts* em paralelo ao invés de um único *script* sequencial com objetivo de diminuir o tempo total de execução. Para simular este cenário, implementamos um algoritmo de quadratura adaptativa (Andrews, 1991), que aproxima a integral de uma função através da soma da área de trapézios. Este não é um caso de uso real de aceleração de *scripts* no Redis porque não há necessidade de interação do *script* com o banco de dados. No entanto, é um algoritmo que exige muito processamento e por isso permite avaliar a possibilidade de aceleração de *scripts* e divisão de tarefas.

Inicialmente o algoritmo estima a integral da função como um único trapézio. Para verificar se a estimativa é adequada, o algoritmo divide o trapézio em dois e verifica se a nova estimativa está suficientemente próxima a anterior. Caso a diferença esteja acima da tolerância, subdivide o trapézio.

[4]Mac Book Pro Mid 2012 com 4 processadores 2.5 GHz Intel Core i5 e memória de 16 GB 1600 MHz DDR3 e sistema operacional Mac OS X 10.10.3.

O algoritmo é executado recursivamente para cada novo trapézio até se obter uma estimativa aceitável em todos os trapézios. Nossa implementação, listada no anexo B, aceita que um ou mais *scripts*, com mesmo código, rode em paralelo. As regiões do gráfico que precisam ser subdivididas e recalculdas são armazenadas em uma fila compartilhada pelos *scripts* no Redis.

Para medir a real aceleração do algoritmo, também implementamos uma versão puramente sequencial em Lua, que não utiliza a fila no Redis e guarda a lista de regiões a ser calculada em memória. O tempo de execução deste código diretamente pelo interpretador Lua ou no Redis é aproximadamente igual e levemente inferior ao do *script* paralelo rodando em uma única *thread*, seja através do comando `EVAL` ou `EVALASYNC`. É claro que não é nosso objetivo acelerar algo que pode ser executado fora do banco de dados e esta medida foi tomada apenas para se verificar o custo da comunicação através do Redis.

Figura 5.3: Comparação de desempenho no cálculo de integral definida.

Os resultados estão apresentados na figura 5.3. Utilizando como referência o tempo de execução do *script* bloqueante, vemos que é possível acelerar a execução de um *script* através da utilização de múltiplos comandos em paralelo. Obtemos, inclusive, aceleração real do programa. Isto é, conseguimos um tempo de execução inferior ao melhor código sequencial fora do Redis. Este exemplo demonstra que a aceleração de *scripts* é viável dentro do banco de dados.

5.3 Processamento de dados em tempo real

Para explorar o cenário de processamento de *streams*, criamos uma aplicação que contabiliza os *hashtags* mais populares na rede social Twitter. Nesta rede social, os usuários publicam mensagens de texto (ou *tweets*) de até 140 caracteres. Para facilitar a pesquisa e a descoberta de conteúdo, os usuários frequentemente marcam suas mensagens com *hashtags*. Estas etiquetas (*tags*) são palavras precedidas do caracter tralha (*hash*). O Twitter exibe na sua interface os tópicos mais comentados no momento de acordo com estes marcadores. Não é possível, no entanto, visualizar os tópicos mais comuns em uma dada região geográfica.

A rede social disponibiliza uma API para consumo de um canal com todos os *tweets* de uma região geográfica de interesse. Cada mensagem neste canal é composta pelo texto do *tweet* e por metadados, como localização e nome do usuário. Para processar esse *stream* de dados no Redis, criamos uma aplicação externa que se registra no *stream* do Twitter e publica todas as mensagens em um canal PUB/SUB do Redis.

O grafo de processamendo deste canal está descrito na figura 5.4. Nesta figura, os *scripts* estão representados como vértices com formato oval, os canais como arestas direcionadas e as chaves do Redis como retângulos.

O *script* "Conta mensagens"da figura 5.4 é carregado e inscrito no canal "twitter"através dos seguintes comandos:

```
SCRIPT LOAD "redis.call('incr', 'messages')"
SSUBSCRIBE e2e5237dbb8a0827abff6e678f2f6db0254e77f5
   ↪ twitter
OK
```

O *script* "Extrai texto", que cria um novo canal cujo conteúdo é apenas o texto das mensagens, sem seus metadados, é criado da seguinte maneira:

```
SCRIPT LOAD "redis.call('publish', 'twitter_text',
   ↪ cjson.decode(ARGV[2]).text)"
"32dbd2f474982a61f883f8bb9919a7a217e83af9"
SSUBSCRIBE 32dbd2f474982a61f883f8bb9919a7a217e83af9
   ↪ twitter
OK
```

Para criar um canal de *hashtags* a partir do canal de texto, utilizamos o *script* "Extrai hashtags", registrado da seguinte maneira:

Figura 5.4: Exemplo de uma DAG para processamento de stream de tweets, contabilizando as hashtags mais populares.

```
SCRIPT LOAD "for hash_tag in ARGV[2]:gmatch('#%S+')
   ↪ do redis.call('publish', 'hash_tags', hash_tag)
   ↪ end"
"70099b9f70780c7cc34852a3879aa046308040df"
SSUBSCRIBE 70099b9f70780c7cc34852a3879aa046308040df
   ↪ twitter_text
OK
```

Finalmente, para contabilizar a quantidade de *tweets* para cada *hashtag*, registramos o *script* "Conta hashtags":

```
SCRIPT LOAD "redis.call('zincrby', 'hashtags', 1,
   ↪ ARGV[2])"
"8a76441d0653d1842565729a867c4a8254d3c092"
SSUBSCRIBE 8a76441d0653d1842565729a867c4a8254d3c092
   ↪ hash_tags
OK
```

A quantidade de ocorrências de cada *hashtag* é gravada em uma estrutura de conjuntos ordenados. As cinco *hashtags* mais frequentes podem ser listadas através de um comando do Redis na chave de resultados:

```
ZREVRANGEBYSCORE hashtags +inf 0 WITHSCORES LIMIT 0 5
 1) "#KCA"
 2) "49"
 3) "#VoteFifthHarmony"
 4) "30"
 5) "#Vote1DUK"
 6) "16"
 7) "#VoteLuanSantana"
 8) "3"
 9) "#AmanhaTalitaAraujoNoPaparazzo"
10) "2"
```

É possível criar um *script* monolítico que receba como argumento o texto de cada *tweet* e compute a frequência de cada *hashtag*. Preferimos decompor o problema em *scripts* menores que fazem uso dos canais para se comunicarem. Como vantagem dessa abordagem, clientes externos poderiam se registrar nos canais intermediários, que contém os textos e *hashtags* pré-processados.

O código que publica o *stream* no Redis, carrega os *scripts* e imprime as *hashtags* mais populares está disponível publicamente em `https://github.com/jbochi/twitter-stream`. Executamos estes comandos no dia 28 de março de 2015 e a hashtag mais popular no Brasil era "#KCA", relativa ao prêmio *Kids' Choice Awards*. Esta *hashtag* só passou a ser exibida pelo Twitter como um *trending topic* depois de aproximadamente uma hora. Rodamos novamente estes *scripts* no dia 02 de agosto de 2015 e as *hashtags* mais populares eram novamente relacionadas à televisão e entretenimento: "#MPN"(concurso "Meus prêmios Nick"), "#MTVHottest"(Concurso do canal MTV), "#Los33"(um filme que estava estreiando no Brasil nesta data), "#TataWerneck"(atriz), "#ArthurAguiarNaDançaDosFamosos"(cantor em um programa de televisão).

Em nossos testes com o mesmo *laptop* utilizado no teste anterior, conseguimos processar 2050 mensagens/s com tempo de resposta médio de 24ms. Não podemos fazer uma comparação de desempenho com *scripts* bloqueantes ou com a implementação padrão do Redis porque somente os *scripts* assíncronos tem a capacidade de interagir com os canais PUB/SUB.

Nesta aplicação, utilizamos diferentes *scripts* para processar um mesmo canal de entrada, sem que o publicador precise ter ciência dos consumidores. Com o *script* que extrai *hashtags*, também validamos que é possível publicar

mais de uma mensagem para uma única mensagem de entrada. Outras topologias também são possíveis. Por exemplo, podemos publicar mensagens em mais de um canal de saída ou inscrever o mesmo *script* em diferentes canais de entrada.

6
Conclusão

O objetivo principal deste trabalho foi investigar a adequação do modelo multi-tarefa $M{:}\,N$ em uma aplicação real de largo uso. Nossa contribuição se dá na implementação deste modelo no banco de dados Redis, adicionando o suporte a *scripts* paralelos de uma forma simples para o programador, que abstrai as dificuldades do escalonamento e da sincronização de tarefas. Conseguimos atingir este objetivo mantendo a compatibilidade com os *scripts* bloqueantes. Demonstramos a utilidade das extensões implementadas nos cenários de detecção de anomalias, aceleração de *scripts* e processamento de *streams*. Este resultado reforça o potencial de utilização do modelo multi-tarefa $M{:}\,N$ como uma ferramenta para possibilitar o paralelismo em linguagens dinâmicas de maneira eficiente e simples do ponto de vista do programador.

Em relação à implementação mono-tarefa padrão do banco de dados, observamos no nosso sistema ganhos significativos nos cenários explorados pelas aplicações desenvolvidas. No entanto, a implementação do modelo proposto introduziu necessidade de sincronização no processamento dos eventos de rede do Redis, o que causou perda de desempenho em alguns casos de uso comuns. Após o desenvolvimento deste trabalho, um *branch* de desenvolvimento do banco de dados Redis foi criado para permitir a liberação de memória de forma não bloqueante[1]. Essa implementação introduziu uma *thread* adicional para liberação de memória e, para que isso fosse possível, removeu o compartilhamento de objetos dos *buffers* de rede de cada cliente e de todos os comandos Redis, substituindo o compartilhamento por cópia dos dados. Essa alteração viabiliza a implementação de *scripts* assíncronos sem a necessidade de sincronização adicional que tivemos que introduzir para escrita na rede e potencialmente mais simples. No total, alteramos o código fonte do banco de dados em 12 arquivos e um total de 913 linhas de código em locais sensíveis.

Apesar dos resultados satisfatórios nos cenários propostos, em nossa implementação, existe contenção do *lock* global uma vez que todos os *workers* concorrem por ele para alteração de qualquer chave no Redis. Um possível trabalho futuro é implementar *locks* de menor granularidade, bloqueando

[1] `http://antirez.com/news/93`

apenas as chaves que forem acessadas. Tal implementação é agora possível, uma vez que chaves Redis não mais compartilham dados internamente.

Outra melhoria anunciada no banco de dados[2] após o desenvolvimento de nossas alterações foi a possibilidade de depuração de *scripts* e replicação de *scripts* através da replicação do seu efeito e não do *script*, assim como fizemos com os *scripts* bloqueantes. Seria desejável que os *scripts* assíncronos também pudessem ser depurados da mesma forma.

A motivação deste trabalho era abstrair a complexidade da programação paralela em um pequeno nicho. Apesar das modificações no banco de dados terem sido pontuais, a implementação e depuração do programa para garantir um correto funcionamento do banco de dados em qualquer situação foi bastante díficil. Tivemos que lidar com diversas condições de corrida e experimentar estratégias diferentes de sincronização e escalonamento para obter um bom desempenho. Esta dificuldade só reforça nossa crença de que é necessário buscar abstrações mais simples e eficientes para que a programação paralela deixe de ser uma verdadeira arte e entre no alcance de um público maior de desenvolvedores.

[2] `http://antirez.com/news/97`

7
Referências Bibliográficas

ADYA, A. et al. Cooperative task management without manual stack management. In: **Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference**. Berkeley, CA, USA: USENIX Association, 2002. (ATEC '02), p. 289–302. ISBN 1-880446-00-6. Disponível em: <http://dl.acm.org/citation.cfm?id=647057.713851>.

AKIDAU, T. et al. Millwheel: Fault-tolerant stream processing at internet scale. **Proc. VLDB Endow.**, VLDB Endowment, v. 6, n. 11, p. 1033–1044, ago. 2013. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2536222.2536229>.

ANDERSON, T. E. et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 25, n. 5, p. 95–109, set. 1991. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/121133.121151>.

ANDREWS, G. R. **Concurrent Programming: Principles and Practice**. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991. ISBN 0-8053-0086-4.

BEHREN, R. von; CONDIT, J.; BREWER, E. Why events are a bad idea (for high-concurrency servers). In: **Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9**. Berkeley, CA, USA: USENIX Association, 2003. (HOTOS'03), p. 4–4. Disponível em: <http://dl.acm.org/citation.cfm?id=1251054.1251058>.

BROOKS JR., F. P. No silver bullet – essence and accidents of software engineering. **IEEE Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 4, p. 10–19, abr. 1987. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.1987.1663532>.

CHANDOLA, V.; BANERJEE, A.; KUMAR, V. Anomaly detection: A survey. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 41, n. 3, p. 15:1–15:58, jul. 2009. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1541880.1541882>.

HALLER, P.; ODERSKY, M. Actors that unify threads and events. In: **Proceedings of the 9th International Conference on Coordination Models and Languages**. Berlin, Heidelberg: Springer-Verlag, 2007. (COORDINATION'07), p. 171–190. ISBN 978-3-540-72793-4. Disponível em: <http://dl.acm.org/citation.cfm?id=1764606.1764620>.

HALLER, P.; ODERSKY, M. Scala actors: Unifying thread-based and event-based programming. **Theorical Computer Science**, Elsevier Science Publishers Ltd., Essex, UK, v. 410, n. 2-3, p. 202–220, fev. 2009. ISSN 0304-3975. Disponível em: <http://dx.doi.org/10.1016/j.tcs.2008.09.019>.

HERLIHY, M. et al. Software transactional memory for dynamic-sized data structures. In: **Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2003. (PODC '03), p. 92–101. ISBN 1-58113-708-7. Disponível em: <http://doi.acm.org/10.1145/872035.872048>.

HICKEY, R. The Clojure programming language. In: **Proceedings of the 2008 Symposium on Dynamic Languages**. New York, NY, USA: ACM, 2008. (DLS '08), p. 1:1–1:1. ISBN 978-1-60558-270-2. Disponível em: <http://doi.acm.org/10.1145/1408681.1408682>.

HOARE, C. A. R. Communicating sequential processes. **Communications of the ACM**, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359585>.

IERUSALIMSCHY, R. **Programming in Lua**. 3rd. ed. [S.l.]: Lua.Org, 2013. 347 p. ISBN 859037985X, 9788590379850.

KARGER, D. et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: **Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1997. (STOC '97), p. 654–663. ISBN 0-89791-888-6. Disponível em: <http://doi.acm.org/10.1145/258533.258660>.

LISKOV, B. Distributed programming in Argus. **Communications of the ACM**, ACM, New York, NY, USA, v. 31, n. 3, p. 300–312, mar. 1988. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/42392.42399>.

MCCRACKEN, D. Posix threads and the Linux kernel. In: **Proceedings of the Ottawa Linux Symposium**. Ottawa, Canada: Ottawa Linux Symposium, 2002. p. 330–337.

MCKENNEY, P. E. Is parallel programming hard, and, if so, what can you do about it? 2011.

MICROSOFT. **C# Programming Guide**. 2014. Disponível em: <http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx>.

MICROSOFT. **User-Mode Scheduling**. 2014. Disponível em: <https://msdn.microsoft.com/en-us/library/windows/desktop/dd627187(v=vs.85).aspx>.

MOURA, A. L. D.; IERUSALIMSCHY, R. Revisiting coroutines. **ACM Transactions on Programming Languages and Systems**, ACM, New York, NY, USA, v. 31, n. 2, p. 6:1–6:31, fev. 2009. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/1462166.1462167>.

OUSTERHOUT, J. Why threads are a bad idea (for most purposes). In: **Presentation given at the 1996 Usenix Annual Technical Conference**. San Diego, CA, USA: USENIX Association, 1996. v. 5.

PEREIRA, R. et al. Cloud based real-time collaborative filtering for item–item recommendations. **Computers in Industry**, Elsevier, v. 65, n. 2, p. 279–290, 2014.

SKYRME, A.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Exploring Lua for concurrent programming. v. 14, n. 21, p. 3556–3572, 2008.

SKYRME, A.; RODRIGUEZ, N.; IERUSALIMSCHY, R. Scripting multiple CPUs with safe data sharing. **Software, IEEE**, v. 31, n. 5, p. 44–51, Sept 2014. ISSN 0740-7459.

SUTTER, H. The free lunch is over: A fundamental turn toward concurrency in software. **Dr. Dobb's journal**, v. 30, n. 3, p. 202–210, 2005.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems: Principles and Paradigms (2nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

TOSHNIWAL, A. et al. Storm@twitter. In: **Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2014. (SIGMOD '14), p. 147–156. ISBN 978-1-4503-2376-5. Disponível em: <http://doi.acm.org/10.1145/2588555.2595641>.

VIRDING, R.; WIKSTRöM, C.; WILLIAMS, M. **Concurrent Programming in Erlang (2nd Edition)**. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996. ISBN 0-13-508301-X.

WALKER, E. F.; FLOYD, R.; NEVES, P. Asynchronous remote operation execution in distributed systems. In: IEEE. **Distributed Computing Systems, 1990. Proceedings., 10th International Conference on**. Paris, France, 1990. p. 253–259.

WANG, C. et al. Statistical techniques for online anomaly detection in data centers. In: IEEE. **Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on**. Dublin, Ireland, 2011. p. 385–392.

YUE, Y. **How Twitter Uses Redis To Scale - 105TB RAM, 39MM QPS, 10,000+ Instances**. 2014. Disponível em: <http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html>.

A
Exemplo de sessão no terminal

Através da interface de linha de comando do Redis, é possível fazer experimentação e testes não automatizados. Um exemplo de sessão no terminal demonstrando os comandos básicos do Redis e também os de scripting está abaixo. Os comandos executados estão em negrito e o retorno está em fonte padrão.

```
% ./redis-cli
redis> ping
PONG
redis> SET nome juarez
OK
redis> GET nome
"juarez"
redis> incr mycounter
(integer) 1
redis> incr mycounter
(integer) 2
redis> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2
   ↪ key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
redis> EVAL "return redis.call('get','mycounter')" 0
"2"
redis> EVAL "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
redis> GET foo
"bar"
redis> SCRIPT LOAD "return 1"
"e0e1f9fabfc9d4800c877a703b823ac0578ff8db"
redis> SCRIPT EXISTS
   ↪ e0e1f9fabfc9d4800c877a703b823ac0578ff8db
1) (integer) 1
```

```
redis> EVALSHA e0e1f9fabfc9d4800c877a703b823ac0578ff8db 0
(integer) 1
redis> SCRIPT FLUSH
OK
redis> SCRIPT EXISTS
   ↪ e0e1f9fabfc9d4800c877a703b823ac0578ff8db
1) (integer) 0
redis> EVALSHA e0e1f9fabfc9d4800c877a703b823ac0578ff8db 0
(error) NOSCRIPT No matching script. Please use EVAL.
```

B
Algoritmo de quadratura adaptativa no Redis

```
local func = function(x)
    if (x < 0.01 or x > 0.99) then
        return 0
    end
    return math.sin(3/x) * math.sin(5/(1-x)) +
           math.pow(x, math.tan(math.pi/2 - x)) * math.sqrt(
              ↪ x) +
           math.sin(1/x) +
           2 * math.sin(1/(1-x))
end

local key_range_list = 'ranges'
local key_total_area = 'area'
local threads_running_key = 'threads_running'

local ERROR_TOLERANCE = 1e-10

local area_estimate = function(func, a, b)
  return (b - a) * (func(b) + func(a)) / 2.0
end

local good_area_estimative = function(func, a, b)
  local middle = (a + b) / 2.0
  local estimative = area_estimate(func, a, b)
  local better_estimative = area_estimate(func, a, middle) +
     ↪  area_estimate(func, middle, b)
  local error = math.abs(estimative - better_estimative)
  return error < ERROR_TOLERANCE, better_estimative
end

local push_range = function(a, b)
  redis.call('rpush', key_range_list, cmsgpack.pack({a, b}))
end
```

```
local pop_range = function()
  local range = redis.call('lpop', key_range_list)
  if range then
    return cmsgpack.unpack(range)
  end
end

local total = 0

if redis.call('incr', 'threads_running') == 1 then
  -- First thread starting
  redis.call('del', key_range_list)
  push_range(0, 1)
  redis.call('set', key_total_area, 0)
end

local range
while true do
  if not range then
    range = pop_range()
  end
  if not range then
    local last = redis.call('decr', 'threads_running') == 0
    local result = redis.call('incrbyfloat', key_total_area,
      ↪ total)
    if last then
      return(result)
    else
      return("other is running")
    end
  end
  local a, b = range[1], range[2]
  local ok, estimative = good_area_estimative(func, a, b)
  if ok then
    total = total + estimative
    range = nil
  else
    local middle = (a + b) / 2.0
    push_range(a, middle)
    range = {middle, b}
```

```
    end
end
```