Replicação de servidores Memcached com Repcached

Arquitetura master/slave em servidores memcached com libevent

Fabiano Araujo
Universidade La Salle
UNILASALLE
Canoas, Brasil

***Abstract*—Este artigo apresenta a utilização de Repcached como solução para replicação de Memcached, aproximando a solução de um ambiente tolerante a falhas por replicação.**

***Index Terms*—cache, replication, memcached, repcached, distributed systems**

I. Introdução

Um sistema tolerante a falhas tem a capacidade de fornecer alta disponibilidade com consistência para seu uso em situações onde parte de sua funcionalidade ou de seus componentes param funcionar corretamente, estão defeituosos ou em falha total, de modo que o reparo possa ser realizado durante a falha sem a interrupção total ou parcial do sistema. [1]

O maior exemplo onde sistemas tolerante a falhas são necessários para prevenir problemas e acidentes são os aviões. Neles, a comunicação entre cada componente é crucial e portanto tolerante a falhas onde é possível que um componente assuma a funcionalidade do outro em caso de falha completa ou perda do mesmo. Em alguns fabricantes é possível se ter quatro conjuntos de três processadores trabalhando paralelamente com o sistema de voto para que os comandos de navegação, conhecidos como *fly-by-wire* cheguem ao seu resultado ideal. [2]

Em sistemas distribuídos a tolerância a falhas pode ser relacionada à transações bancárias, da necessidade de atomicidade, até a replicação de servidores de cache, da necessidade de disponibilidade.

Servidores de cache são utilizados por diversos sistemas distribuídos que consumimos diariamente, inclusive em sistemas voltados para o entretenimento, como o Netflix, e sua principal finalidade é armazenar dados que não precisam ser atualizados constantemente e tornar-los disponível para acesso com uma grande velocidade de leitura, reduzindo custos de operação mecânica em discos ou tempo de execução de comandos.

Neste estudo é demonstrada a replicação, uma das técnicas de tolerância a falhas, em servidores Memcached [3], comumente utilizados por sistemas distribuídos, como Facebook ou Redit. Utilizando Docker [4], os resultados obtidos são referentes ao experimento de parada e reinicialização de um elemento com defeito, neste caso um servidor de cache e o comportamento da aplicação o qual realiza a leitura dos dados periodicamente.

II. Referencial Teórico

Para o melhor entendimento do estudo, a compreensão básica dos princípios de tolerância a falhas é parte fundamental da compreensão da motivação da técnica de replicação. Junto dos conceitos simples, a compreensão da utilização de servidores cache e a arquitetura *master/slave* também é fundamental para o entendimento completo de como os dados transitam entre os servidores e a razão pela qual a leitura é realizada, em geral, apenas em um tipo de servidor

A. Tolerância a falhas

Os principais componentes para a criação de um sistema tolerante a falhas podem ser resumidos em quatro tópicos diferentes: Disponibilidade, confiança, segurança e mantenabilidade [1].

Disponibilidade é um dos principais conceitos a qual sistemas distribuídos podem ser exigidos principalmente em estruturas *microservices*, cujo, assimilando com o anteriormente demonstrado modelo de aviação aonde os componentes são disjuntos e podem funcionar independentes, o sistema maior é dividido em pequenos módulos e estes módulos possuem um ecossistema específico para cada um deles, onde a falha de um não impacta de forma imediata o sistema como um todo.

Assim a disponibilidade é um dos principais fatores a serem analisados neste estudo, visto que trata-se de servidores cache o qual a informação encontra-se para leitura especificamente neles. A discussão sobre a migração de sistemas de dados entre RDBMS (*Relational Database Management System*) e NoSQL (*Not-only SQL*), não é o objetivo deste artigo.

Confiança, por vezes confundida com disponibilidade, refere-se à propriedade que um sistema tem para rodar continuamente sem uma falha [1]. Difere-se da disponibilidade no ponto em que a disponibilidade é o mero fato de que o sistema está acessível e a confiança é a relação de frequência de erros sobre o tempo que impactam o acesso ao sistema. Por exemplo, se um *host* desliga por um milissegundo a cada hora, pode-se dizer que o mesmo tem uma disponibilidade alta mas uma confiança pequena. Por outro lado, se um sistema para por duas semanas em um mês determinado do ano, pode-se dizer que o mesmo tem uma disponibilidade um pouco abaixo dos 100%, porém a confiança é alta.

Quanto a segurança, é tendencioso pensar apenas no fato da segurança invasiva dos dados. No entanto, quando se tratando de tolerância a falhas, a segurança é a certeza de que quando um sistema se encontra em falha crítica nada catastrófico irá acontecer [1]. Citando a aviação, devido ao modelo adotado deste o lançamento das missões Apollo da NASA, a forma com que os dados são processados e outros componentes que possam assumir o mesmo papel em caso de falha, garantem que a aviação seja um dos sistemas mais seguros de transporte. [2]

Por fim, a mantenabilidade se refere a agilidade de reparo de um componente ou sistema em caso de falha. Com as recentes evoluções na área de inteligência artificial, é possível visualizar a utilização de *deep learning* para eventuais correções sem que sejam necessárias intervenções humanas nestes sistemas.

É possível ainda definir os níveis de falha nos chamados modelos de falhas que se dividem em *crash failure*, *omission failure*, *timing failure*, *response failure* e *arbitrary failure*.

Dentre todos, o mais arriscado, difícil de se diagnosticar sem verificação ativa de desenvolvedores é o *arbitrary failure*, isto porque se caracteriza a falha aonde o dispositivo está enviando valores incorretos, porém todo seu funcionamento está correto. Normalmente se identificam por falhas de regras de negócios ou de erros de desenvolvimento.

A utilização de replicações de servidores cache tem como objetivo diminuir a ocorrência dos *crash failures*, quando a aplicação ou servidor trava ou para de responder.

B. Servidores cache

Servidores cache podem ser utilizados com diversos tipos de dados, mas seu principal objetivo é ser um servidor específico com dados que frequentemente são acessados e que não possuem armazenamento permanente. Dentre os tipos de servidores cache dois se destacam pela facilidade de utilização, o Redis e Memcached [3].

Estes são apenas exemplos de softwares de cache, onde nada impede que o carregamento de informações em espaço de memória possa ser implementado em demanda específica, sendo também uma forma de cache.

Fundamentalmente, no entanto, Redis não necessariamente é um servidor cache. Originado depois do Memcached, o Redis possui as características de um banco não relacional com as possibilidades de armazenamento de dados em RAM, o que permite uma leitura muito mais rápida e resposta para a solicitação em tempo incomparável com RDBMS normais. Sua principal diferença ao Memcached é a possibilidade de persistir os dados de forma nativa.

O Memcached por sua vez não possui funcionalidade de tipificação de dados armazenados ou mesmo a persistência em disco físico, todos os dados ficam armazenados em RAM e, por isto, possui uma necessidade maior em relação aos cuidados de tolerância a falhas, visto que uma vez o software ou dispositivo serem desligados, as informações são perdidas e novo reabastecimento dos dados deve ser realizado.

Em ambos os softwares, os valores são armazenados na estrutura chave-valor, onde o Redis armazena o tipo de informação junto dos dados permitindo operações complexas de ordenação ou estrutura de filas simplesmente encadeadas, do contrário do Memcached que armazena a estrutura em valor *string* apenas.

O Redis possui, nativamente, a possibilidade de distribuição de processamento em clusters e replicação de servidores com estrutura *master/slave*. O Memcached por outro lado, não oferece nenhum dos dois. A solução mais próxima de replicação é a utilização de Repcached [5]

O cache também é implementado em menores versões como nos navegadores atuais que possuem cache local de arquivos estáticos como JavaScripts, CSS e imagens. Além deste cache local, é possível também o controle de balanceamento de acesso de dados por outros servidores de cache interligados ou não, como demonstra a figura 1.

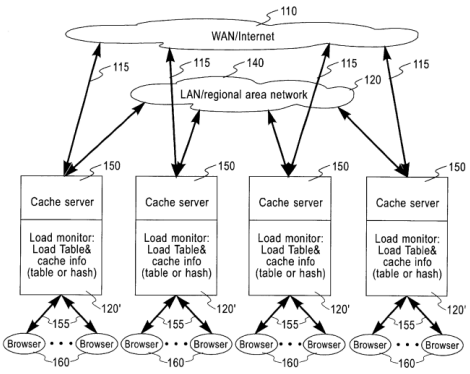

Fig. 1. Arquitetura de caches [6]

A figura 1 ainda demonstra fatores a serem abordados pela replicação de servidores e omite a estrutura *master/slave* de alguns softwares. O exemplo da figura 1, no entanto, mostra a relação de conectividade padrão de algumas aplicações, como os navegadores.

Visualizando a imagem como uma forma de fluxo para os servidores externos, onde este estariam disponíveis via WAN (servidor de base de dados, por exemplo) os servidores de caches estão dispostos com a finalidade de permitir acesso mais rápido às informações previamente carregadas e são conectados em uma rede LAN para comunicarem entre si.

III. Implementação

O experimento realizado para demonstração da ferramente se dispôs entre dois *containers* Docker [4], obtidos de uma imagem com todos as bibliotecas necessárias para execução de dois servidores Repcached [5] isolados.

A replicação fica por parte da definição cíclica dos IPs de servidor *master* e *slave*. Por vezes a definição da dependência ficou confusa e foi obtido o entendimento de que o a definição real entre *master* e *slave*, nesta ocasião, refere-se especificamente à escolha do desenvolvedor ou cliente de utilização de um IP.

Utilizando imagens pré definidas dos *containers*, algumas modificações foram realizadas para que pudesse ser exibida

de forma coerente a comunicação entre os servidores nos *containers*. Isto porque os exemplos encontrados e fundamentados nas próprias imagens utilizadas como exemplo utilizam do próprio *host* de um *container* Docker com o próprio *container*, não deixando clara as limitações da comunicação entre dois *containers*.

Da imagem *yrobla/docker-repcached* [7], foram retirados os arquivos que criavam a possibilidade de logins com administradores, por não ser relevante ao experimento e, principalmente, foi postergada a inicialização do *daemon* Memcached. Mesmo que o projeto possua um nome diferente, Repcached, o mesmo é uma adaptação do Memcached adicionando apenas um parâmetro para identificação do IP do servidor cujo terá replicação dos dados.

A postergação da inicialização se fez necessária pelo contexto de utilizar dois *containers* Dockers. Em sua natureza, cada *container* possui um IP local cujo, por padrão, é definido por um dispositivo virtual de rede. Este dispositivo funciona como um DHCP entre *containers* atribuindo os IPs assim que o *container* for iniciado.

No código inicial no entanto era necessário definir o IP do servidor *slave* antes da inicialização. Porém como a inicialização é o evento que é atribuído um IP na rede virtual interna de *containers* Docker, não era possível definir a atribuição, gerando uma falha no procedimento pois o mesmo adotava o IP 127.0.0.1 caso não fosse especificado.

Exposta a porta 11211, padrão do servidor memcached, então, é inicializado o *container* e acessado o mesmo utilizando as diretivas *exec -it –entrypoint /bin/bash* para que se tenha acesso à um *shell* onde é possível iniciar o servidor memcached.

Iniciando os dois *containers* foi então verificado os IPs internos definidos pelo comando *docker network inspect bridge*. Tomando nota dos IPs atribuídos, os serviços memcached foram iniciados em ambos os *containers* apontando o parâmetro *-x* para o IP do *container* oposto. Por isto a atribuição de *master* e *slave* se tornou cíclica.

Tal comportamento permite que caso o *slave* caia e retorne em um momento posterior, o mesmo possua todos os dados de *master* e vice-versa. Diferente de estruturas como Redis que possui *slaves* como apenas leitura e propagação direta para *master*. Da forma como o Repmemcached é organizado ambos possuem a mesma atribuição

Omitir a definição do IP ou tornar a atribuição do IP pelo parâmetro *-x* para 127.0.0.1 inviabiliza o fluxo, sendo necessário reiniciar todo o processo em *master* caso o *slave* caia.

A estrutura utilizada no experimento pode ser visualizada na figura 2.

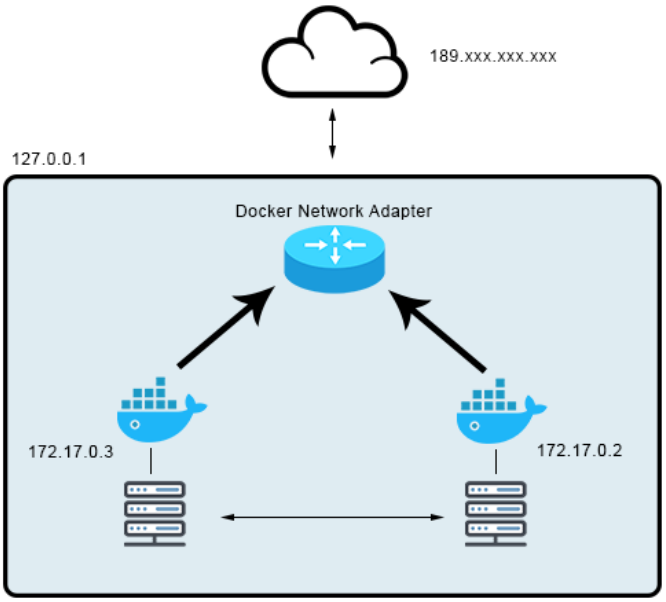

Fig. 2. Ambiente de experimento

IV. Resultados

Foi possível realizar a replicação de ambientes Memcached com Repcached com sucesso entre *containers* Docker. Uma vez iniciados os *containers*, foi identificado os IPs dos mesmos e finalmente iniciados os serviços via *nohup*, uma vez que os parâmetros estavam melhor dispostos se explicitamente inseridos

Foram nomeados cache1 e cache2 respectivamente e tratada a conectividade de uma aplicação PHP com o cache2, sendo este ponto de entrada de obtenção e escrita da aplicação. Em termos de utilização, pode-se afirmar que a aplicação interagiu diretamente apenas com o cache2 e que devida a funcionalidade de replicação, o cache1 foi alimentado de forma semelhante.

No momento do ensaio de uma queda, para que a aplicação pudesse continuar a consumir de um servidor Memcached, foram atribuídos dois IPs de servidores com pesos distintos. O peso do servidor, no contexto de aplicação PHP, segundo a sua documentação define a prioridade com que a conectividade irá acontecer, assim o servidor com peso maior terá prioridade e uma vez que não for possível conectar, tentara o segundo mais próximo [8]. Não é definida a forma de utilização caso os servidores tiverem o mesmo peso.

Em código PHP, é possível definir servidores de replicação dos dados, porém nada condiz com a replicação de informação integral que o Repcached se propõe. Das diretivas do PHP, o que se tem de replicação é a replicação imediata da chave salva pelo próprio script PHP, porém chaves criadas por terceiros não serão replicadas visto que não comunicação entre os servidores.

A replicação em código PHP também não permite que os dados armazenados sejam recuperados em caso de queda de um dos servidores, assim servindo apenas para a replicação em servidores com finalidades distintas ou que possuam replicação não conjunta, isto é, entre outros dois servidores que não se comunicam diretamente.

A aplicação contou com um endpoint em AJAX que demonstrava a informação armazenada diretamente por *telnet*,

utilizando das diretivas, uma vez estabelecida conexão com Memcached de *get* e *set*.

Em outro experimento, foi realizada a conexão entre três Dockers, no intuito de aumenta a complexidade cíclica para três servidores. O experimento no entanto falhou pelo motivo de funcionalidade da biblioteca libevent. Dado o ambiente como demonstrado na figura 3, uma entrada no servidor *slave 1* não estaria armazenado do *slave 2*, mas apenas no *master*, pois o *slave 2* está tanto recebendo a informação como enviando para outro servidor que não se comunica diretamente com o ele, no caso, *master*.

Tal conclusão inviabiliza a utilização de Repmemcache em ambientes que demandam grande complexidade de tolerância a falhas ou que necessitem de um controle de alta disponibilidade visto que conforme seus autores, o Memcached não foi construído como um servidor de alta disponibilidade.

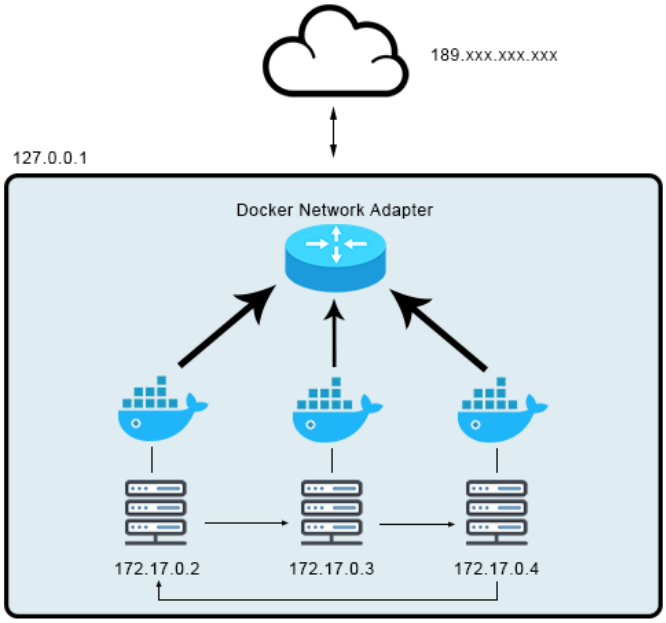

Fig. 3. Ambiente de experimento com três replicações

V. Conclusão

Por meios padrão, não é possível obter a replicação do ambiente Memcached, visto que o mesmo não possui instruções nem foi moldado para tal. Sua natureza de armazenamento de dados apenas em RAM também não prevê qualquer persistência de dados físicos que pudessem justificar a adaptação de outro serviço como leitura dos seus dados.

Assim o uso do Repcached permite a criação de um servidor *master* e um *slave* diretamente conectados, mas não permite a criação de *clusters* de informação. A replicação também não acontece em termos de processamento mas apenas de indicação de lugar na memória em outro endereço de rede além do servidor *master*, não configurando assim o comportamento de *cluster*.

A definição de que o Memcached não possui alta disponibilidade impacta diretamente no consumo atual da informação, dada a vasta capacidade tecnológica que outra ferramentas tem neste assunto, como Redis. Cabe aos desenvolvedores e analistas identificarem a utilização do Memcached detalhadamente para evitar perigos de inviabilidade de comunicação por parte de falta de replicação.

Referências

[1] M. v. S. Andrew S. Tanenbaum, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[2] J. Herman. Fault-tolerance in avionics systems. [Accessed: 18- Nov- 2018]. [Online]. Available: https://cs.unc.edu/~anderson/teach/comp790/powerpoints/fault_tolerance.pptx

[3] Packet. Memcached. [Accessed: 18- Nov- 2018]. [Online]. Available: https://memcached.org/

[4] Docker. Docker. [Accessed: 15- Nov- 2018]. [Online]. Available: https://www.docker.com/

[5] K. Inc. Repcached. [Accessed: 15- Nov- 2018]. [Online]. Available: http://repcached.lab.klab.org/

[6] P. S.-L. Y. Kevin Michael Jordan, Kun-Lunk Wu, "Load balancing cooperating cache servers by shifting forwarded request," Patent US 6 438 652, 2002.

[7] Yrobla. Base docker image to run a repcached server. [Accessed: 10- Nov- 2018]. [Online]. Available: https://github.com/yrobla/docker-repcached

[8] PHP. Memcached::addservers. [Accessed: 18- Nov- 2018]. [Online]. Available: http://php.net/manual/en/memcached.addservers.php