Introduction

Para facilitar a construção de serviços de metadata altamente disponíveis, o Tango fornece aos programadores a abstração de uma estrutura de dados replicada na memória (como um mapa ou uma árvore) apoiada por um shared log, partilhado por todos os clientes. Esta ideia surgiu uma vez que os serviços onde a metadata está guardada encontram-se geralmente centralizados, não estando tão disponíveis. E também do facto dos serviços que já existem, e que garantem maior disponibilidade (ex: Zookeeper), não permitir tanta flexibilidade às estruturas de dados usadas.

Shared Log (baseado no CORFU)

O shared log usado pelo Tango é baseado numa versão modificado do CORFU, pelo que é possível executar as operações permitidas por esta interface. Os clientes podem adicionar entradas ao log, verificar a última entrada, ler uma entrada num offset específico e indicar que um determinado offset pode ser garbage collected. Também é importante realçar, que os clientes não trocam mensagens entre eles, isto é, todas as interações ocorrem através do shared log.

O shared log também é apoiado por um sequenciador centralizado, que serve essencialmente como um contador para saber a cauda atual do shared log. Este não é essencial para o funcionamento do Tango, mas serve mais como uma otimização. Como se usa chain replication, se vários clientes tentarem escrever no mesmo sítio, apenas um deles “vencerá” e irá conseguir escrever nesse offset, por isso mesmo sem sequenciador o shared log funciona. Caso o sequenciador falhe, qualquer cliente pode facilmente recuperar o seu estado usando a operação de slow check.

Tal como o CORFU original é assegurada ordem total no shared log e os clientes podem utilizar a chamada fill no caso de um cliente falhar após obter o offset para escrever na entrada do log.

Tango Object = View + History

Na base da arquitetura do Tango, está a definição de um Tango object, o Tango Object é uma estrutura de dados replicada no cliente, a partir do shared log, sendo que o seu estado existe sob duas formas:

1. um **histórico**: que é uma sequência ordenada das actualizações armazenadas de forma permanente no shared log;
2. **qualquer número de vistas**: que são cópias completas ou parciais da estrutura de dados - como uma tree ou um map - construídas a partir do shared log e armazenadas na memória dos clientes (ou seja, servidores de aplicações). As vistas são atualizadas nos clientes, conforme necessário, fazendo forwarding do histórico do objeto presente no shared log.

Um cliente modifica um Tango object dando append de um novo update ao histórico; e acede ao objeto sincronizando a sua vista local com o histórico do shared log.

Para além disso cada Tango Object tem de implementar obrigatoriamente a **apply** upcall, que altera a vista quando o Tango runtime a chama devido a novas entradas terem sido adicionadas ao shared log. A configuração da função apply, permite que cada cliente tenha vistas diferentes para o mesmo objeto. Por exemplo, um cliente pode ter uma “tree view” enquanto outro cliente pode ter uma “set view”, estando ambos a ler do mesmo log.

Adicionalmente, cada objeto tem uma interface externa de métodos de acesso específicos do objeto; por exemplo, o Tango Register expõe métodos de read/write, enquanto que poderíamos ter um Tango Map com os métodos get/put.

No entanto, estes métodos não mudam diretamente o estado do objeto em memória, em vez disso, guardam os seus parâmetros num update record que usa a função helper do tango, **update_helper**, de forma a dar append ao shared log.

Do mesmo modo, os métodos não leem diretamente o objeto, eles usam a função **query_helper**, que por sua vez reproduz os novos update records no shared log até ao seu fim (tail) e aplica-os ao objeto através da apply upcall antes de retornar.

A arquitetura do Tango (ou então como referido anteriormente caso se diga isso) garante as seguintes propriedades:

- Consistência: os clientes implementam RME, efetuando as escritas para o shared log e sincronizando o seu histórico para as leituras, garantindo linearizabilidade para as operações.
- Durabilidade: caso os clientes falhem, podem facilmente recuperar acedendo ao shared log e ao seu histórico.
- Histórico: os clientes podem aceder a estados mais antigos dos Tango objects, criando uma nova vista a partir do shared log. O Tango permite criar checkpoints no shared log para um objeto, que funcionam como uma imagem do objeto, de modo a não ser necessário consultar toda a lista de alterações ao objeto.
- Elasticidade (se calhar tirar daqui pq falamos na evaluation)

Transactions

(Slide 1)

Quanto às transações é usado optimistic concurrency control, ou seja, é adicionado um commit especulativo ao shared log e só depois é tomada a decisão. Adicionalmente, cada commit record contém um set de leitura: lista de objetos lidos na transação, juntamente com as suas versões, em que a versão é simplesmente o último offset do shared log em que o objeto foi modificado. Se não houver alterações aos objetos desde a última leitura, ou seja, nenhum dos objetos envolvidos na transação for obsoleto, é feito o commit senão aborta.

Assim, o Tango também consegue garantir atomicidade e isolamento, para as transações mesmo que estas envolvam vários objetos, guardando-os num único shared log.

As transações encontram-se delimitadas por BeginTX e EndTX. Sendo que BeginTX cria o contexto transacional localmente, enquanto o EndTX é responsável por adicionar o commit record ao shared log, e reproduz o log até ao ponto do commit, e de seguida é que é tomada a decisão.

(Slide 2)

Cada cliente que encontra um commit record, decide independentemente se deve abortar ou dar commit, comparando as versões de leituras dos seus objetos no readset, com as versões atuais dos objetos. Se a transação for confirmada, os objetos no conjunto de escrita são atualizados localmente com a função apply.

Para transações só com leituras não é adicionado o commit especulativo ao shared log, apenas é percorrido o log até à última entrada, e a decisão é feita localmente tornando as transações de leitura mais rápidas. No caso das transações apenas com escritas não é percorrido o log apenas é feito o commit.

Vamos apresentar um breve exemplo em que existe a Alice e o Bob que partilham a mesma conta bancária. Atualmente a conta tem 500$, o Bob quer depositar 100$ e a Alice quer levantar 500$. Decidem realizar essas operações ao mesmo tempo. O shared log já possuía algumas transações e ambos têm o estado da sua vista local atualizada. Começam a ser introduzidas as escritas mas o Bob acaba primeiro a transação, ou seja o commit record do Bob aparece primeiro na ordem total. O Bob percorre o log até ao seu commit e como verifica que não houve alterações efetua o depósito com sucesso. A Alice após terminar a transação também vai percorrer o log até ao seu commit, no entanto encontra alterações à conta bancária feitas pelo Bob, tendo de abortar o levantamento. Assim a conta bancária fica com 600$ após este exemplo.

Layered Partitions

Alguns clientes precisam apenas de uma parte dos objetos presentes no shared log, não é eficiente ter de percorrer todo o log. Para isso, a implementação do Tango mapeia cada objeto para uma stream. Ao recorrer à chamada readnext é possível obter a próxima entrada da stream no shared log sem ter de aceder às entradas pertencentes a outras streams. Cada cliente pode percorrer as streams que correspondem aos objetos que possui, percorrendo assim apenas a parte do log que lhe interessa. Resultando em layered partitioning, em que uma aplicação pode fragmentar o seu estado em múltiplos Tango objects, cada um instanciado num cliente diferente.

Para implementar os streams, cada cliente tem uma biblioteca que, internamente, guarda as streams na forma de linked list. De modo a construir a lista, cada entrada no log tem um header com o ID da stream e backpointers para a últimas k entradas da mesma stream, sendo necessárias N/K leituras para construir a lista em que N é o número total de entradas dessa stream. Quando o cliente inicia, a aplicação fornece um conjunto de IDs de streams de interesse para o cliente. Para cada stream a biblioteca procura a última entrada dessa stream no log e constrói a lista percorrendo o log pela ordem inversa através dos backpointers.

Uma alteração em relação à implementação original do CORFU, é quando o cliente faz append passa a enviar os IDs das streams ao sequencer, e este responde com os backpointers de cada stream para além do offset da próxima entrada como já acontecia na implementação original. Este processo permite guardar os k backpointers de cada stream nos headers dessa entrada no shared log. De modo a responder mais rápido ao cliente o sequencer também passa a guardar os k backpointers de cada stream.

Caso o sequencer falhe é necessário outro sequencer que tem de reconstruir o estado dos backpointers percorrendo o log pela ordem inversa. Os clientes também podem falhar após obter o offset e o processo é o mesmo que a implementação original do CORFU: qualquer cliente pode usar a operação fill para colocar junk values nesses offsets. No entanto as entradas com junk values não possuem headers com backpointers, mas o processo de construção da lista continua a funcionar. Só que quando todos os backpointers levam a junk values é necessário percorrer o log pela ordem inversa até encontrar a última entrada válida dessa stream.

Transactions over Streams

A chamada multiappend permite realizar transações que envolvem vários objetos, ou seja, afetam várias streams. Esta transação ocupa apenas uma entrada no shared log havendo apenas um commit. No entanto, existe a limitação do cliente não poder realizar transações que envolvem leituras de objetos que não tem cópia local. (no paper refere trabalho para futuro)

Como os clientes podem não ter vistas de todos os objetos, um cliente pode ver uma transação que afeta uma das suas streams de interesse mas também ter leituras de outras streams para os quais não possui uma cópia. Utilizando o mesmo processo das outras transações não permite que este cliente saiba o resultado da transação. Para evitar isso, o cliente que cria o commit também insere no shared log a decisão relativa a esse commit. Qualquer cliente que tenha uma vista dos objetos lidos na transação também pode inserir a decisão no log, caso o cliente que criou o commit ainda não o tenha feito quer seja por delay ou caso tenha falhado.

Quando nenhum cliente tem cópias do objeto lido e o cliente que criou a transação falha antes de adicionar a decisão, após um timeout, qualquer cliente pode reconstruir os objetos lidos na transação através do shared log e assim verificar se houve alterações ao objeto.

Evaluation

(Slide 1)

Nesta imagem, dois clientes cada um com uma vista para o mesmo objeto, sendo que um dos clientes só faz escritas e o outro só faz leituras. Podemos observar que o throughput diminui bastante quando as escritas começam, e depois ficam constantes com cerca de 40K ops/sec. No entanto, a latência das leituras aumenta por causa dos writes, e isto deve-se principalmente ao trabalho “extra” que o cliente que só faz leituras tem de fazer para estar atualizado com as escritas que vão sendo efetuadas.

(Slide 2)

Este gráfico mostra vários clientes com uma vista diferente cada , os quais só fazem leituras, com cerca de 10K reads/sec por cliente, enquanto existe uma carga constante de escrita de 10k writes/sec. Podemos ver que as leituras vão aumentando linearmente até chegar a um ponto em que o shared log fica saturado, ao usar um log apoiado por mais servidores é possível aumentar linearmente as leituras para os 18 clientes, no entanto ao adicionar mais clientes vai haver um ponto em que o log vai voltar a ficar saturado. Resumindo as leituras escalam linearmente até ao log estar saturado sendo possível usar um maior número de servidores para permitir mais leituras por segundo.

(Slide 3)

O gráfico da esquerda mostra que ao usar partições em que cada cliente realiza transações em que lê e escreve apenas numa stream também é possível escalar

linearmente o número de transações até à saturação do shared log. Sendo o gráfico idêntico ao do slide anterior.

No gráfico da direita são usados os mesmos 18 clientes e 18 servidores. São comparadas as transações normais do Tango com transações em que se usa o 2-phase-locking para bloquear os objetos do read set em vez de ter o comportamento normal do EndTX, podemos observar que o throughput diminui para ambas à medida que o número de transações entre objetos/streams aumenta, dado que consequentemente têm mais conflitos. Tal comportamento é esperado, mas demonstra que o Tango tem características semelhantes ao 2PL, mas o Tango tem a vantagem de não sofrer de problemas como deadlocks.

Conclusion

(Uma das maiores contribuições do Tango é permitir diferentes estruturas em memória partilhando o mesmo log. Podem haver 2 clientes em que um guarda uma árvore de ficheiros por ordem alfabética e outra a hierarquia das diretorias, isto permite realizar operações mais eficientes por exemplo listar os ficheiros começados pela letra B ou listar os ficheiros de uma dada diretoria.)

(O Tango permite vários use cases podendo ter todos os objetos do log replicados em cada cliente possivelmente cada um com uma vista diferente para certos objetos. As streams também permitem ter cada cliente com um objeto diferente ou haver partilha de apenas alguns objetos)