Proposta de um algoritmo para *checkpoints* particionados

Everaldo de Avila G. Junior[1], **Odorico Machado Mendizabal**[2]

[1]Centro de Ciências Computacionais
Universidade Federal do Rio Grande (FURG) – Rio Grande – RS – Brasil

[2]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

everaldo.junior@furg.br,odorico.mendizabal@ufsc.br

Resumo. *Estratégias de replicação permitem que serviços mantenham-se operacionais apesar da ocorrência de falhas. Para aumentar disponibilidade de sistemas replicados é necessário que réplicas faltosas possam se recuperar. Normalmente, uma réplica em recuperação obtém um checkpoint de alguma réplica correta e processa os novos comandos não contidos no checkpoint. Portanto, réplicas corretas precisam salvar periodicamente uma imagem de seus estados em uma unidade permanente. Para garantir consistência, este procedimento de checkpointing exige a interrupção momentânea do serviço, causando atraso no atendimento de requisições. Este trabalho apresenta uma proposta para execução de requisições em paralelo à criação de checkpoints, reduzindo a sobrecarga ocasionada pelo procedimento de checkpointing.*

1. Introdução

Uma maneira de prover tolerância a falhas é através de replicação. Dessa forma, mesmo que algumas réplicas falhem, réplicas corretas mantêm a execução do serviço. Contudo, é necessário que uma maioria de réplicas corretas estejam em execução. Para aumentar a disponibilidade do sistema mecanismos de recuperação permitem que réplicas faltosas retornem a seu comportamento correto. Para isto, técnicas de durabilidade como *checkpointing*, *logging* e transferência de estados devem ser integradas ao sistema replicado.

Checkpoints são imagens do sistema, representando o estado da aplicação em um determinado instante de tempo. Durante a execução normal réplicas devem armazenar seus estados (*checkpoints*) periodicamente. Assim, na ocorrência de uma falha e eventual recuperação, a réplica em recuperação retoma sua execução a partir de algum *checkpoint*. Porém, enquanto uma réplica estiver em processo de *checkpointing* novas requisições ficam impedidas de executar, causando degradação no desempenho [Bessani et al. 2013]. Para aliviar a sobrecarga causada pelo processo de *checkpointing*, estratégias que permitam a execução de requisições em paralelo à criação de *checkpoints* são necessárias [Castro and Liskov 2000], [Bessani et al. 2013].

Neste trabalho é apresentada uma estratégia de *checkpointing* que visa permitir a execução de requisições em paralelo à realização de *checkpoints*. O algoritmo proposto explora o particionamento do estado da aplicação.

2. Algoritmo para *checkpoints* particionados

O algoritmo proposto divide o estado da aplicação em partições. Desta maneira, enquanto é criada a imagem de uma partição, são impedidos de executar somente os comandos

que operam sobre esta partição. Operações sobre dados nas demais partições podem ser executados em paralelo com o procedimento de *checkpointing*.

Em nosso algoritmo um comando é representado por c_i, onde i representa o identificador do comando. Os identificadores assumem valores monotônicos crescentes, ou seja, $i = 0, 1, 2, ...$ Assume-se que um comando pode operar sobre uma única variável (leituras e escritas atômicas). O estado da aplicação S é equivalente à união das partições, representada por $P = P_1 \bigcup \ldots \bigcup P_n$, onde P_j indica a partição j. Logo, $\bigcup_{j=1}^{n} P_j = S$. Considera-se que cada variável x da aplicação está contida em apenas uma partição ($x \in P_j$). Logo, $\forall A, B \in P, A \neq B \Rightarrow A \cap B = \varnothing$.

Comandos pertencem a duas classes: *comando executável*, que consiste em operações de leitura ($r(x) : v$) ou escrita ($w(x, v)$) sobre uma variável e *comando de recuperação*, que inclui operações para consultar até qual comando foi salvo em um *checkpoint* ($cp\_id(p)$), requisitar o *checkpoint* de uma partição ($s\_cp(p)$) e requisitar o *log* ($s\_log()$).

Um *checkpoint* $\mathcal{C}$, é representado pela união dos *checkpoints* de todas as partições do estado da aplicação. Mais precisamente, $\mathcal{C} = \{\mathcal{C}_1^k \cup \ldots \cup \mathcal{C}_n^z\}$, onde os índices superiores $(k, \ldots, z)$ representam o identificador do último comando executado e persistido no *checkpoint* e os índices inferiores $(1, \ldots, n)$ identificam a partição.

Além do conjunto de *checkpoints*, o algoritmo mantém um *log* dos comandos executados. Este registro é utilizado para recuperação, pois uma réplica em recuperação busca o estado mais atual (obtida por algum *checkpoint*) e reexecuta o *log* de comandos. Assim, a réplica atinge o estado consistente com as demais réplicas em execução. O *log* é um conjunto finito de comandos $\mathcal{L} = \{c_i, c_{i+1}, ..., c_w\}$. Sempre que um *checkpoint* é realizado o *log* é apagado ($\mathcal{L} = \varnothing$), pois as modificações realizadas por estes comandos foram armazenadas no *checkpoint*.

Durante sua execução, uma réplica comporta-se conforme o Algoritmo 1. Inicialmente, a réplica instala seu estado inicial e inicializa o *log* de comandos (linhas 1–2). O vetor de *marcações* ($\mathcal{M}$) (linha 3) indica quando o *checkpoint* de uma partição deve ser realizado. Cada posição de $\mathcal{M}$ indica a quantidade de modificações realizadas em cada partição. Quando a quantidade de alterações exceder um limite pré-determinado (*max_mod*), o *checkpoint* de uma determinada partição deve ser realizado. A réplica aguarda a chegada de requisições (linha 4), representadas pelo comando c. Se o comando c for uma requisição de identificadores de *checkpoint*, a réplica executa o procedimento $Envia\_ids\_cp()$ (linha 6), descrito no Algoritmo 2. Caso c seja uma requisição de *log* (linha 8), a réplica envia o *log* de comandos (linha 9). Caso c seja uma requisição de estado (linha 11), então a réplica envia o *checkpoint* referente à partição solicitada (linha 12). Caso c seja um comando executável, a réplica verifica se não existem requisições pendentes armazenadas no $buffer$ (linha 14), isto é, comandos que não puderam operar em suas respectivas partições devido à execução de algum *checkpoint*. Se houverem comandos aguardando por execução, estes são executados (linhas 14–18). Então, para a execução de c, primeiramente é identificada a partição sob a qual o comando operou (linha 20). Se não há processo de *checkpointing* em execução sobre a partição p_j (linha 21), o comando é executado, o vetor de *marcações* é atualizado e a execução é registrada no *log* (linhas 22–24). Caso contrário, o comando é inserido no $buffer$ (linha 26). O estado da partição $\mathcal{C}_n^k$ é salvo sempre que o valor de $\mathcal{M}[n]$ exceder max_mod (linha 29).

Algoritmo 1 Execução

1: $\mathcal{C} \leftarrow \mathcal{C}_1^0 \cup \mathcal{C}_2^0 \cup \ldots \cup \mathcal{C}_n^0$;
2: $\mathcal{L} \leftarrow \varnothing$;
3: $\mathcal{M}[n] \leftarrow [0, 0, ..., 0]$;
4: Quando recebe(c)
5: **if** c == cp_id **then**
6: Envia_ids_cp(); {Enviar o maior id de um comando que modifica a partição i}
7: **end if**
8: **if** (c == s_log) **then** {Envia o log de comandos}
9: $send\_log(\mathcal{L})$;
10: **else**
11: **if** c == $s\_cp[i]$ **then**
12: $send\_cp(\mathcal{C}_i^{cmd})$; {Envia os dados da partição i}
13: **else**
14: **for** c_b in $buffer$ **do**
15: $p_j \leftarrow particao(c_b)$; {atribui a p_j o número da partição acessada por c_b}
16: executa(c_b); {Executa a requisição}
17: $\mathcal{M}[p_j]$ + +; {Incrementa o marcador da partição}
18: $\mathcal{L} \leftarrow \mathcal{L} \cup c_b$; {registra no log}
19: **end for**
20: $p_j \leftarrow particao(c_b)$; {atribui a p_j o número da partição acessada por c}
21: **if** ¬ ($\mathcal{C}_{id}$_em_checkpoint) **then**
22: executa(c); {Executa a requisição}
23: $\mathcal{M}[p_j]$ + +; {Incrementa o marcador da partição}
24: $\mathcal{L} \leftarrow \mathcal{L} \cup c$; {registra no log}
25: **else**
26: $buffer \leftarrow buffer \cup c$; {adiciona o comando no buffer}
27: **end if**
28: **if** $\mathcal{M}[n] > max\_mod$ **then**
29: $store(C_{p_j}^{\#c})$; {armazena a partição}
30: **end if**
31: **end if**
32: **end if**

O Algoritmo 2 descreve o procedimento de envio dos identificadores de cada partição. Para todas as partições do estado (linhas 2–5) a réplica envia uma tupla (id, i, cmd) (linha 3), onde id é o identificador da réplica, i é o índice da partição e cmd representa o número de sequência do último comando que operou sobre a partição de índice i. Por fim, é incrementado o iterador sobre as partições (linha 4).

Algoritmo 2 Envia_ids_cp

1: $i \leftarrow 0$
2: **for** $\mathcal{C}_i^{cmd} \in \mathcal{C}$ **do** {Para todas as partições...}
3: $send(id, i, cmd)$; {envia ids da réplica, da partição e último comando executado}
4: i++;
5: **end for**

O procedimento de recuperação ocorre conforme a Figura 1. Assumindo que o estado da aplicação é dividido em 2 partições, os últimos *checkpoints* armazenados em r_1 e r_2 são $\mathcal{C}$ = $\{C_1^8 \cup C_2^4\}$ e $\mathcal{C}$ = $\{C_1^4 \cup C_2^6\}$, respectivamente. Considere que r_1 está em processo de recuperação. Ao requisitar os últimos identificadores de comandos processados pelo *checkpoint*, através das requisições $cp\_id(1)$ e $cp\_id(2)$, a réplica r_1 receberá de r_2 as tuplas $(2, 1, 4)$ e $(2, 2, 6)$. Ao receber as tuplas, a réplica r_1 compara os identificadores dos últimos comandos que executaram sobre as partições 1 e 2. Como r_1 possui um *checkpoint* mais recente da partição 1 (identificador = 8), não é necessário requisitar esta partição. Contudo, o *checkpoint* da partição 2 armazenado por r_1 é mais antigo que o *checkpoint* de r_2. Dessa forma, r_1 requisita somente o *checkpoint* da partição 2, através do comando $s\_cp(2)$. A partir da recepção da imagem do *checkpoint* 2, r_1 instala os *checkpoints* mais atuais e processa o *log* de comandos.

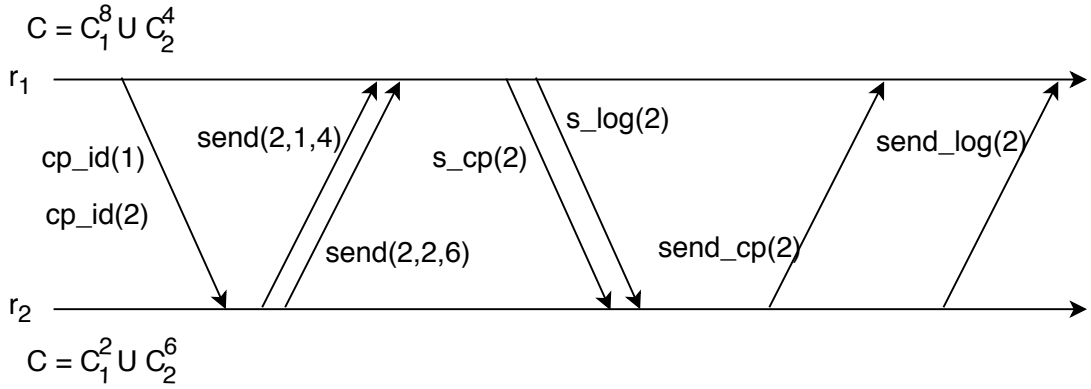

Figura 1. Exemplo de requisição de estados.

3. Conclusão

Neste trabalho foi apresentado um algoritmo para *checkpointing* particionado. Dessa forma, apenas partes do estado da aplicação são bloqueadas durante a criação de um *checkpoint*, permitindo que alguns comandos possam executar em paralelo com a escrita do *checkpoint*. Como trabalho futuro pretende-se avaliar o desempenho do algoritmo, comparando com a estratégia tradicional, ou seja, sem particionamento. Espera-se obter ganho na vazão do serviço durante a realização de *checkpoints*, uma vez que esta estratégia reduz a contenção nas operações de *checkpointing*. Pretende-se avaliar o algoritmo com relação a quantidade de partições do estado, isto é, identificar até que ponto a estratégia oferece ganho de desempenho através do particionamento do estado.

Referências

Bessani, A. N., Santos, M., Felix, J., Neves, N. F., and Correia, M. (2013). On the efficiency of durable state machine replication. In *USENIX Annual Technical Conference*, pages 169–180.

Castro, M. and Liskov, B. (2000). Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 19. USENIX Association.