Examen Master IFI, 2ème année, Parcours CSSR
2010/2011
Tous documents autorisés

F. Baude et al.

29 Novembre 2010, Durée 3 heures

Exercice 1 (3 points)

Design a probe-echo algorithm to count the number of processes in an anonymous network whose topology is a strongly connected graph. The algorithm starts when an initiator node sends out a probe to each of its neighbors, and ends when the initiator receives an echo from each neighbor. When the algorithm terminates, the initiator of the algorithm should figure out the count. Briefly argue why your solution will work.

Cet exercice étant facile et plus ou moins déjà traité, le soin avec lequel l'algorithme proposé sera écrit et analysé sera pris en compte dans la note.

Exercice 2 (5 points)

Voici un algorithme schematique qui fait un CIC (communication induced checkpointing). Protocole étudié dans le cours. Seul le pseudo-code de la réception de message et la reprise sont montrés, `local_state` représente l'état local du processus.

```
Receive message from process i on process j with piggybacked checkpoint index N
 if N>current_checkpoint then
   send (local_state,j,N) to stable storage
   current_checkpoint = N
 endif
transmit message to the application
END

Recovery of global state N (executed by stable storage)
 for all processes i
  send to process i the state: (local_state,j,N)
END
```

Questions:

1. Que fait-on lorsque le processus `i` veut prendre un checkpoint local? (écrire le pseudo code de `Process i takes a local checkpoint`) – environ 2 lignes de code
2. Comment calcule t-on l'index N du meilleur état à restaurer? Dit autrement: quelle est la meilleure valeur de N à envoyer à la procédure `Recovery`?
3. Que se passe-t-il dans l'algorithme ci-dessus lorsque dans la procédure de réception, l'index N reçu vaut `current_checkpoint-2`?
   Cela est-il optimal?

- Si oui, expliquer brièvement pourquoi?
- Sinon, proposez une amélioration de l'algorithme ci-dessus.

4. L'algorithme ci-dessus ne traite pas le cas des messages en transit. Proposer une extension des 2 procédures ci-dessus permettant de restaurer les messages en transit

Exercice 3 (4 pts)

Répondez aux questions suivantes à propos de l'agorithme d'Exclusion Mutuelle Distribuée de Ricart et Agrawala:

1. Représentez sur un un diagramme de séquence les echanges de message d'une exécution possible de l'agorithme pour le scénario suivant:
   - le sysème comporte 4 noeuds distribués N1, ..., N4
   - à t=t0, le noeud N1 réclame le droit d'entrer en section d'exclusion mutuelle (S.E.M.)
   - à t=t1, t1>t0, les noeuds N2 et N3, qui on déjà recu la demande de N1, demandent à leur tour (simultanément) à entrer dans la S.E.M.
   - à t=t2, t2>t1, N1 recoit le dernier message lui permettant d'entrer en S.E.M.
   - à t=t3, t3>t2, N1 recoit les deux messages envoyés par N2 et N3 à t=t1
2. Prouvez formellement, par une suite de déductions logiques en partant d'hypothèses pertinentes (comme nous l'avons fait en cours), que l'une des propriétés d'exclusion mutuelle vues en cours notées ME1, ME2 ou ME3), au moins, est bien respectée par cet algorithme.

Exercice 4 (8 pts)

- 1. Décrivez briévement ce que sont les propriétés de *safety (sureté)* et *liveness (vivacité)*.

- 2. Quelles sont les propriétés d'un " Eventually Leader Detector? Lesquelles sont des propriétés de *safety (sureté)* et *liveness (vivacité)* ? Selon vous, est-il possible d'implémenter une abstraction de Leader Election en utilisant un Eventually Perfect Failure Detector ? Oui/Non, expliquez pourquoi ? (mentionnez bien les propriétés qui seraient violées et/ou satisfaites)

- 3. Quelles sont les propriétés d'un "Reliable Broadcast " dit "régulier" (celui vu en cours) ? Donnez la signification de chaque propriété et classez les en safety/liveness.

  La Figure 1 présente un algorithme de reliable broadcast dit "uniforme" (souvenez vous, au dernier cours, vous avez vu la différence entre uniforme et régulier au niveau du consensus...l'idée est la même):

  – (a) Expliquez moi quelle est la différence entre un Reliable Broadcast régulier et un Reliable Broadcast uniforme ? (mentionnez et argumentez bien les propriétés)
  – (b) Expliquez moi au travers de deux diagrammes d'espace-temps la différence entre les deux.
  – (c) Décrivez moi l'idée derrière l'algorithme en Fig. 1.

- 4. Comparez les deux propriétés d'ordre causal suivantes:

  - (a) Si un processus délivre les messages m1 et m2, et m1 $\rightarrow$ m2, alors le processus doit délivrer m1 avant m2.
  - (b) Si un processus délivre un message m2, alors il a du délivrer tous les messages m1 tel que m1 $\rightarrow$ m2.

  Sont-elles équivalentes? Si non, laquelle est la plus forte. Laquelle est une propriété de *safety (sureté)* et laquelle est une propriété de *liveness (vivacité)* ? Laquelle devrait être utilisée dans une spécification d'une abstraction de Causal Order Broadcast?

- 5. Quelle sont les propriété d'un "Uniform Consensus" ? Donnez la signification de chaque propriété et classez les en safety/liveness. La Figure 2 présente un algorithme de "Flooding Uniform Consensus". Pensez vous qu'il est possible d'optimiser cet algorithme afin de sauver une étape de communication, en d'autres termes, que tous les processus corrects décident après N-1 étapes de communication (N étant le nombre de processus participant à l'algorithme) ? Considerez le cas simple où il n'y a que deux processus et argumentez.

Algorithm 3.4 All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast (urb).

Uses:
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$).

function canDeliver(m) **returns** boolean **is**
return (correct $\subseteq$ ack$_m$);

upon event $\langle$ *Init* $\rangle$ **do**
delivered := pending := $\emptyset$;
correct := Π;
forall m **do** ack$_m$:= $\emptyset$;

upon event $\langle$ *urbBroadcast* | m $\rangle$ **do**
pending := pending $\cup$ {(self, m)};
trigger $\langle$ *bebBroadcast* | [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver* | p_i, [DATA, s_m, m] $\rangle$ **do**
ack$_m$:= ack$_m$ $\cup$ $\{p_i\}$;
if ((s_m, m) $\notin$ pending) **then**
pending := pending $\cup$ {(s_m, m)};
trigger $\langle$ *bebBroadcast* | [DATA, s_m, m] $\rangle$;

upon event $\langle$ *crash* | p_i $\rangle$ **do**
correct := correct $\setminus\{p_i\}$;

upon exists $(s_m, m) \in$ pending **such that** canDeliver(m) $\wedge$ $m \notin$ delivered **do**
delivered := delivered $\cup$ $\{m\}$;
trigger $\langle$ *urbDeliver* | s_m, m $\rangle$;

Figure 1: All-Ack Uniform Reliable Broadcast

Algorithm 5.3 Flooding Uniform Consensus

Implements:
UniformConsensus (uc).

Uses:
BestEffortBroadcast (beb);
PerfectFailureDetector ($\mathcal{P}$).

upon event $\langle$ *Init* $\rangle$ **do**
correct := Π; round := 1; decided := $\perp$; proposal-set := $\emptyset$;
for $i = 1$ **to** N **do** delivered[i] := $\emptyset$;

upon event $\langle$ *crash* | p_i $\rangle$ **do**
correct := correct $\setminus$ $\{p_i\}$;

upon event $\langle$ *ucPropose* | v $\rangle$ **do**
proposal-set := proposal-set $\cup$ $\{v\}$;
trigger $\langle$ *bebBroadcast* | [MySet, 1, proposal-set] $\rangle$;

upon event $\langle$ *bebDeliver* | p_i, [MySet, r, newSet] $\rangle$ **do**
proposal-set := proposal-set $\cup$ newSet;
delivered[r] := delivered[r] $\cup$ $\{p_i\}$;

upon (correct $\subseteq$ delivered[round]) $\wedge$ (decided = $\perp$) **do**
if round = N **then**
decided := *min* (proposal-set);
trigger $\langle$ *ucDecide* | decided) $\rangle$;
else
round := round + 1;
trigger $\langle$ *bebBroadcast* | [MySet, round, proposal-set] $\rangle$;

Figure 2: Flooding Uniform Consensus