APUVS, Blatt 9

Jan Fajerski and Kai Warncke and Magnus Müller

18. Januar 2011

Aufgabe 9.1

Die Variable `participant` ist für die Funktion des Algorithmus nicht essentiell. Er würde auch funktionieren, wenn man das gesamte `if`-Statement durch

```
send <election, i> to successor;
```

ersetzt. Die Variable `participant` verhindert, dass nebenläufige Wahlvorgänge laufen. Kommt der zweite Wahlvorgang beim Initiator der ersten Wahl an, dann wird dieser nicht weiter propagiert.
Der Algorithmus im Foliensatz 11 auf Seite 11 ist jedoch nicht korrekt. Durch den `else`-Zweig `j=i` kann es dazu kommen, dass nicht der Prozess mit der höchsten ID zum Leader gewählt wird. Ausserdem kann es passieren, dass zur gleichen Zeit zwei unterschiedliche Knoten für unterschiedliche Leader stimmen, was nach Voraussetzung nicht passieren darf.
Im folgenden Beispiel starten 2 Knoten den Wahlalgorithmus (Knoten 2 und kurz danach Knoten 1) (siehe 1). Bekommt nun Knoten 2 die `election`-Nachricht von Knoten 1, sendet Knoten 2 eine `<elected, 2>`-Nachricht (siehe 2). Auch Knoten 3 aktzeptiert die Wahl und leitet die Nachricht weiter. Kurz darauf kommt jedoch die Nachricht `<election, 3>` wieder bei 3 an (von der von 2 initierten Wahl) und 3 sendet seinerseit eine `<elected, 3>`-Nachricht. Es werden also 2 unterschiedliche Leader gewählt (siehe 3), was nach Vereinbahrung am Anfang des Foliensatz 11 nicht zulässig ist.

Aufgabe 9.2

Die minimale Zeit T' ist in Abhängigkeit von i bei N Prozessen wie folgt definiert:

$$\begin{aligned} T'(N) &= T \\ T'(i) &= T'(i+1) + T_{PROCESS} + T_{TRANS} \end{aligned}$$

Der höchste Prozess (also derjenige mit der höchsten Prozessnummer) wird nie die den Ausführungszweig in dem die Zeit T' gewartet wird abarbeiten, da er nie eine Antwort

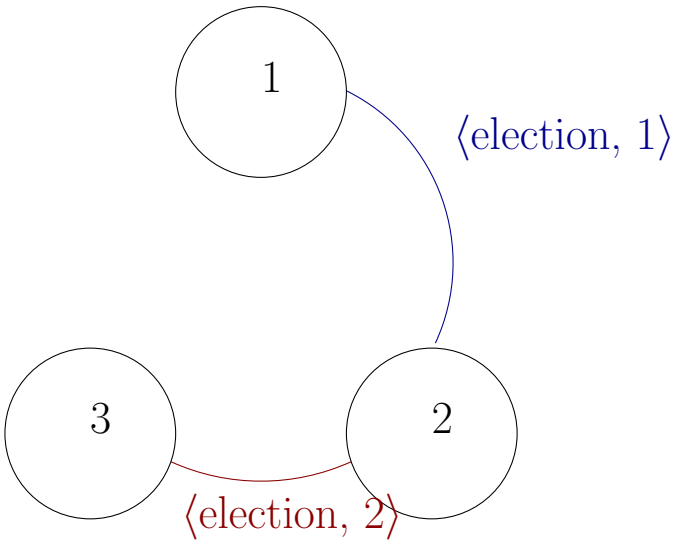

Abbildung 1: Knoten 2 startet eine Wahl und kurz danach auch Knoten 1

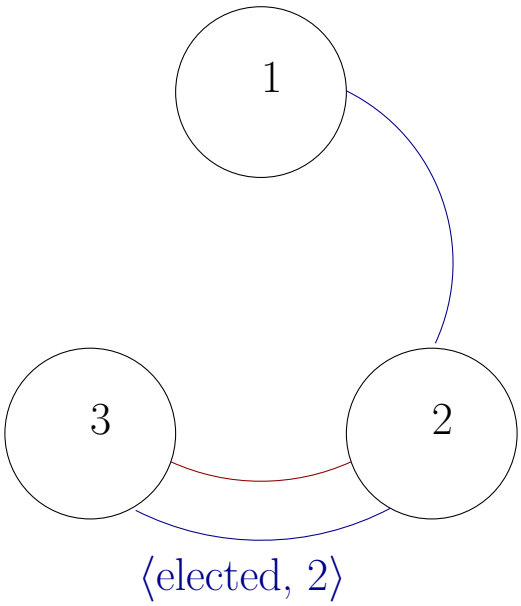

Abbildung 2: Der falsche else-Zweig wird ausgeführt

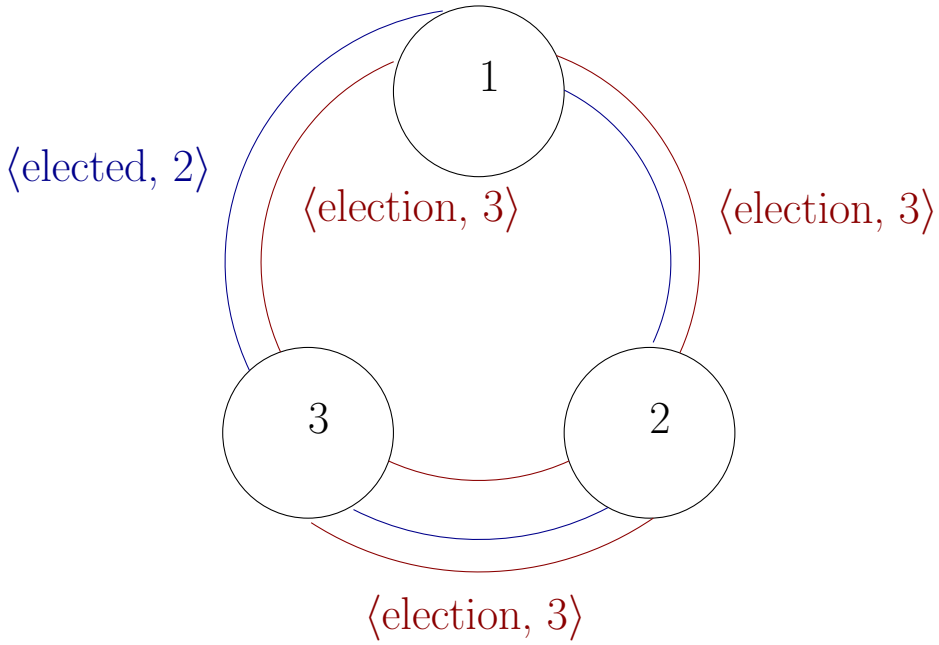

Abbildung 3: Zwei Knoten haben unterschiedliche Leader gewählt

auf seine e-Nachricht bekommt. Darum kann man T' des höchsten Prozesses einfach auf T setzen.
Der darunterliegende Prozess muss mindestens diese Zeit, plus die Zeit die Nächsthöhere zum Versenden der eigenen e-Nachricht benötigt ($T_{PROCESS}$) und der Nachrichtenlaufzeit der c-Nachricht benötigt, warten.

Aufgabe 9.3

Der unter `http://de.wikipedia.org/w/index.php?title=Echo-Algorithmus&oldid=50781161` beschriebene Algorithmus implementiert nicht den `echo`-Algorithmus. Beispielsweise zählen die Knoten, dir rot sind nicht die eingehenden `explore`-Nachrichten mit und der `Initiator` beendet sich sobald er *einen* `echo` Marker erhalten hat. Da wir die Funktion des Algorithmus nicht ergründen können, können wir nur davon ausgehen, dass die Farben nötig sind, da sie den Programmablauf beeinflussen.

Aufgabe 9.4

Jeder Knoten (ausser der `Initiator`) darf ausfallen *nachdem* er ein Token an seinen Vaterknoten geschickt hat. Fällt einer dieser Knoten vorher aus, terminiert der Algorithmus nicht, da jeder Knoten von jedem Nachbarn ein `Token` erwartet. Fällt der `Initiator` aus wird nie ein `Leader` gewählt. Dieser darf also nicht ausfallen.

Aufgabe 9.5

```
  received = 0;
  father = null;
  children = []; // liste der kinder

Initiator:
  forall (q in neighbours) do
    send <token> to q;
  while (received < #neighbours) {
    receive msg from q;
    if (msg == <child>) {
      children = [q | children];
    }
    received = received + 1;
  }
  decide

Participant:
  receive <token> from neighbour q;
  father = q;
  received = received + 1;
  forall (q in neighbours, q is not father) do
    send <token> to q;
  while (received < #neighbours) do {
    receive msg from c;
    if (msg == <child>){
      children = [c | children];
    }
    received = received + 1;
  }
  send <child> to father;
```