Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Martin Steffen, Immo Grabe

Verteilte Algorithmen

Wintersemester 2003/04 — **Serie 9** — 14. Januar 2004

Thema : **(Aufgaben mit Lösungshinweisen)**

Ausgabetermin: **14. Januar 2004**

Abgabe: **19. Januar 2004**

Aufgabe 1 (Peterson (3 Punkte)) Bearbeiten Sie Aufgabe 15.4 aus [**?**].

Solution: Peterson's Algorithm for leader election in a ring network under asynchronous execution is given on page 482 in [**?**]. As the other algorithms we have seen, the election mechanism is based on an total ordering on user ids and the largest id is used to pick the leader. The decisive insight, which also distinguishes this algorithms from the others seen so far is, that the only thing required is that exactly one process is picked. If one does not (unnecessarily) insits on the largest process being chosen, one need not "report back" to the largest process; this economizes messages. So the trick is that the algorithm does not sieve out processes until the "largest" one remains, but rather the *tokens* carrying the ids, until the largest token survives; this then identifies the leader as the process on which it currently resides.

Sieving out a token mean comparing it with it's two "neighbor tokens", i.e., its left and right token. The smaller ones disappear from the game, only the largest of the three survives the comparison. Of course the tokens as data cannot compare themselves; it must be done by the processes, and to avoid "back messages", one does the following trick: instead of finding out whether *a process* is a local maximum by accepting messages from his (surviving) neighbor processes, which requires a bidirectional ring, a process sends its id alone the ring for a distance of 2. This allows a process to determine the maximum id for 3 neighboring tokens and accepting it as new id, which obviosly breaks the linkage between original maximun id a a given process[1]

In order to get an answer (which, btw., is P_6) one can simply write down the tokes and proceed as follows: find each lokal maximum. Move it one step clockwise (ignoring relay nodes). This determines the survivors. Then continue with only the survivors until one (= the maximum) token remains).

Of course this explanation is a certain simplification of what actually happens in the asyynchronous network, but not much so. From the perspective of one processor node, the

[1] Another way to intuitively understand the $n \log n$-message complexitity even if it's an algorithm for *uni*directional rings is that intuitively the information flow and thus the sieving-out process works *bidirectional.*

behavior "*looks*" synchronous: the (active) process simply takes exactly two messages, makes its comparison, and continues with the next "round". □

Aufgabe 2 (Asynchroner Broadcast (6 Punkte)) Bearbeiten Sie Aufgabe 15.15 aus [**?**], d.h., beweisen Sie die Korrektheit des *AsynchBcastAck* aus Abschnitt 15.3.

Solution:

The algorithm *AsynchBcastAck*, sketched on page 499 of [**?**], is an simple extension of the spanning tree algorithm by some value to be broadcasted —a trivial extension— plus a convergecast "phase".

For correctness, we have to show *safety* and *liveness.*

Safety: for safety, the trick as always is to find the right (inductive) invariants. Let's first think what we intend a final result to be "safety". The algorithm does, one could say, three things:

1. build up the spanning tree,
2. send around the value to be broadcast, and
3. "convergecast" the acks.

For the last point, there is no obvious safety property as end result, which can be used as specification.[2] To generalize the first point we state:

1. at each reachable point, the "graph of the parent-pointers" T form a tree with root i_0 which is a subgraph of the network graph $G = (V, E)$.
   - for the base case, the statement obviously holds: all parent pointers are initialized to $\perp$
   - In the inductive step
2.

$i = i_0 \vee parent \neq 0)$ then $bcast \in C_{i,j} \vee bcast \in send_j \vee parent_j \neq \perp$, for all $j \in nbrs$.

$i = i_0 \vee parent \neq 0) \wedge reported = true$, then $ack \in send(i)_j \vee ack \in C_{i,j} \vee j \in acked$ (for all $j \in nbrs_i$

Aufgabe 3 (Knotenanzahl im Netz (4 Punkte)) Bearbeiten Sie Aufgabe 15.20 aus [**?**], d.h., geben Sie einen Algorithmus an der die Anzahl der Knoten im asynchronen Netz ermittelt. Der Korrektheitsbeweis ist *nicht* erforderlich.

Solution: The idea is that, in much the same way that the broadcast algorithm was an adaptation of the simple spanning tree algorithm in that it piggy-backed the value to distribute in the outgoing exploration message, we can here use the flooding back messages (the "echo") to report back information about the number of nodes. Since the outgoing

[2] Of course one can use auxiliary safety properties for the proof and for the intermediate steps, but the acks themselves seem to have no real importance after the algorithm has terminated.

exploration messages build up a *tree*, each node can simply collect the number of descendants it recieves via its neighbors, and relay it to its parent, adding 1. We can use the echoing acklowledgement-messages for this purpose, but we have to be careful to count only those coming from descendants in the tree, not all.

```
// exercise 15.2

...

States:
  ..
  n = 1 : Nat                          // we start assigning each node weight 1


Transitions:
   send(m)_{i,j}                    // output
     precondition:
         m is first on send(j)
     effect:
         remove first element of send(j)

   receive("search")_{i,j}        // input, exploration phase
     effect:
        if    val = ⊥
        then  val := w;
              parent := j;
              for all k ∈ nbrs - {j} do
                  add(``search'') to send(k)
        else add "ack(0)" to send(j) // don't count along this edge

    receive("ack", m)                // input, echo
        effect:
             ack := ack ∪ {j}
             n   := n + m

    report_i (for i ≠ i_0) // internal
      precondition:
         parent not = ⊥
         acked    = nbrs - {parent}   // all our neighbors have reported
         reported = false             // but we ourselven not yet
      effect:
         add "ack(n)" to send(parent)
         reported := true             // and we are through



    report_i (for i = i_0)     // root of the tree
        precondition:
           acked = nbrs
           reported = false
        effect
           reported := true
```

What we cannot do is to postpone the acklowledgement-messages, i.e., send an acklowledgement only if all neighbors have acklowledged, i.e., lower the degree of parallelism. This, however, would lead to *deadlocks* in the algorithm.