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 4. Februar 2004

Serie 12

Thema : Endsemestertest (Einzelabgabe) (Aufgaben mit Lösungshinweisen)

Ausgabetermin: 4. Februar 2004

Abgabe: 11. Februar 2004

Aufgabe 1 (MST (6 Punkte)) Bearbeiten Sie Aufgabe 15.38(a) aus [1].

Solution: The *SimpleMST*-algorithm is described at page 522. It can be seen as a "adaptation" of the *SynchGHS*(cf. [1, p. 66]) to the asynchronous setting. Of course the algorithm is not identical to the synchronous one. A particular difference is that in the synchronous setting, there is not absorb/merge distinction.

The starting point for this exercise is the (asynchronous) GHS. Basically the only thing to change is to achieve that mwoe is determined (more or less) synchronously. In the *GHS*, a component (or rather a process at the border of the component) surveys its neighbors at any time by sending out the test-messages. This leads (sometimes) to the delay of the answer.

Here, the process at level k does not send out the test *unless* at least the local environment of the process is at the same level. That's still a bit "less synchronous" than the *SynchGHS*, where it is guaranteed that the whole network across component boundaries is at the same level. Here this is not possible, but if we only probe neighbors which have the same level, then at least we don't have to wait for the answer. To be always up-to date, each process sends informs his neighbors each time it reaches one new level. □

Aufgabe 2 (MST + leader election (4 Punkte)) Bearbeiten Sie 15.40 aus [1]

Solution: The problem is stated at [1, page 523], respectively the *STtoLeader*-algorithm at page 501. In the code below, we assume that we are given the undirected, unrooted *spanning tree* as part of the *branches*-variables.

```
lsttoleader_i // p.501
messages:
signature:
states:
  branches ⊆ nbrs // contains the unrooted tree
  send[j] : queue of Nat, for all neighbors, intialy empty
  has_elected ⊆ branches = ∅
transitions:
  //-------------------------------------------------------------
  send(m)_{i,j}                      // standard trans'ns
    precondition:  m is first on send(j)    // using the output
    effect:        remove m from send(j)    // send buffers
  //-------------------------------------------------------------
  convergecast_i
    precondition:
      branches = {j}                        // i is a leaf
      reported = false
    effect:
      send[j] := send[j] :: elect(uid)      // send my uid
      reported := true                      // elect only once
  //-------------------------------------------------------------
  convergecast_i
    precondition:
      reported = false
      |branches|  > 1                        // i is not a leaf
      branches \ has_elected  = {j}          // one has  not elected
    effect:
      send[j] := send[j] :: elect(uid)      // send my uid
      reported := true
  //-------------------------------------------------------------
  convergecast_i
    precondition:
      branches = {}                          // degenerate tree
      reported = false
    effect:
      reported := true                      // elect only once
      leader   := i
  //-------------------------------------------------------------
  receive(elect(uid'))_{j,i}  //
    effect:
      has_elected := has_elected + j        // one more has elected
      if    reported = false
      then if   branches \ has_elected = {p} // one more left
           then send[p]  := send[p] :: elect(uid);
                reported := true
            else skip
      else                                // I have already reported
                                          // which means that I must have
                                          // sent my elect to j, which
                                          // must have been the last open
                                          // branch.
           if uid > uid'
           then leader := true
           else skip
  //-------------------------------------------------------------
```

□

Aufgabe 3 (Logical time für *CountMoney* (6 Punkte)) Bearbeiten Sie Aufgabe 18.5 aus [1], d.h., entwickeln Sie, ausgehend von dem Geld-Transfersystem und mit Hilfe logischer Uhren schrittweise den *CountMoney*-Algorithmus.

Solution: The exercise *CountMoney* ask to explicitely carry out the transformations for the logical time and for one of the applications using the time stamps, namely on the example of the *CountMoney*-algorithm. We start with the simple banking algorithm from Figure 1. The transfer functionality is modeled by some non-deterministic internal action, which, depending on the state, chooses some amount of money and one process to transfer the money to. Next we add *Lamport's clocks* to the basic bank algorithm (cf. Figure 2).

```
bank_system$_i$
types:
messages:

signature:
  input:    receive$_{i,j}(m)$, j $\in$ nbrs
  output:   send(m)$_{i,j}$,   j $\in$ nbrs
  internal: transfer$_{i,j}$,  j $\in$ nbrs

states:
  money  : Nat = m_$i$;                       // some initialization
  send   : [1..n] -> (queue of Nat);

transitions:
  //--------------------------------------------------------------
  send(m)$_{i,j}$                        // standard trans'ns
    precondition:  m is first on send(j)   // using the output
    effect:        remove m from send(j)   // send buffers
  //--------------------------------------------------------------
  receive(m)$_{i,j}$  // we receive new money
    effect:
            money := money + m;            // update the balance
  //---------------------------------------------------------------
  transfer$_{i,j}$
    precondition:  (m,j) $\in \varphi$(state) // some condition
    effect:                                   // non-det. autonomous
            money   := money - m;             // sending of money
            send[j] := send[j] :: m;

tasks:
  { send(m)$_{i,j}$  | j $\in$ nbrs } +
  { transfer$_{i,j}$ | j $\in$ nbrs }
```

Figure 1: Bank system

The extension is straightforward.

Finally the *CountMoney*-algorithm (cf. Figure **??**). Conceptually, the algorithm consists of two phases (both are required in the exercise):

1. an agreement,[1] which deadline is taken when to count the money, and
2. the "snapshot" procedure proper.

As for the agreement, the code below uses a *predefined* sequence of points in time:

$$1 \quad 2 \quad 4 \quad \ldots 2^n \ldots$$

[1] Here we do not deal with a full-blown agreement problem. It is possible, indeed required, that the processes share some common knowledge initially about which start time(s) to take. The problem is that fixing one particular start time may lead to a situation, where this point is already past, for some process.

```
bank_system$_i$   // additional clocks (Lamport)
types:      Clock = Nat;
messages:

signature:
  input:    receive$_{i,j}(m)$, j $\in$ nbrs
  output:   send(m)$_{i,j}$,  j $\in$ nbrs
  internal: transfer$_{i,j}$, j $\in$ nbrs

states:
  money : Nat = m_$i$;                        // some initialization
  send  : [1..n] -> (queue of Nat);
  clock : Clock = 0;

transitions:
  //------------------------------------------------------------
  send(m,c)$_{i,j}$                         // standard trans'ns
    precondition:  m is first on send(j)   // using the output
    effect:        remove m from send(j);  // send buffers
                   clock := clock + 1;     // Lamport' clock
  //------------------------------------------------------------
  receive(m,c)$_{i,j}$                           // we receive new money
    effect:
            money := money + m;            // update the balance
            clock := max(clock,c) + 1;     // Lamport's clock
  //-------------------------------------------------------------
  transfer$_{i,j}$
    precondition:  (m,j) $\in \varphi$(state) // some condition
    effect:                                    // non-det. autonomous
            money   := money - m;              // sending of money
            send[j] := send[j] :: m;
            clock   := clock + 1;              // also here we clock

tasks:
  { send(m)$_{i,j}$ | j $\in$ nbrs } +
  { transfer$_{i,j}$ | j $\in$ nbrs }
```

Figure 2: Bank system + Lamport

One problem is to assure, as said, that the point agreed upon does not lie in the past. If some process intends to do some money count, it wants to do so at the "next" appropriate point in time. It must therefore find out what the times of the other processes are. By the time he knows the other may have continued to count up their clocks![2] Fairness can never assure that a *particular* (finite) deadline is not missed.

Anyway, one way out would be that once a process learns that there is a counting procedure going on with a deadline in the future, it should stop transferring money around, lest to count up the time until the counting money is acknowledged, and then the time has passed once more. This way of dealing with the problem —stop the actions— is not what is wanted; the free transfer of money should not be restricted by the need of counting.

Another way to deal with the problem is that given the predefined sequence of unboundedly increasing points in time

$$t_0 \quad t_1 \quad \dots$$

each process performs the count-money book-keeping *for all* t_i in parallel.

In Figure **??** we (somewhat wrongly) assume some predefined, globally-agreed time t_{fix}. A process whose time is strictly less than t_{fix} can just take part in the transfer game, updating the *money*-variable as it sends and receives transfers. If the point is reached, for the process, it is time to count the money. For the process, it means: all the money it currently has plus the one the will sooner or later arrive and which has not yet been counted by the sender. According to the conventions in [1], the time stamp of an event depends on the value of the clock *after* the event has happened, and furthermore money sent or received at the *exact* logical time t_{fix} is counted at the process to which the time belongs (it's not "transit money").

A process starts counting money if in one step it reaches or crosses the time line t_{fix}. This, of course, is faulty, as we cannot be sure (and fairness does not help here) that the fixed deadline has not yet passed. If a process, for some reason, has passed the timeline already, then the following error occurs. The sender process, after the time line, transfers the money, i.e., puts it on the outgoing queue and just counts the money after subtracting the amount. Therefore this should count as "transit money". The receiver, however, cannot know that the sender has *not* counted it, because it is no "transit money" in his eyes.

```
count_money$_i$    //  count  money  with  _fixed_  deadline  t
types:       Clock = Nat;
             Time  = Clock * Pid            // Lamport: lex. order
messages:
            transfer of Nat * Clock;

signature:
  input:     receive$_{i,j}(m)$, j ∈ nbrs
             revision$_i$
  output:    send(m)$_{i,j}$,  j ∈ nbrs
             balance(m)$_i$                // send balance to env.
  internal:  transfer$_{i,j}$, j ∈ nbrs
```

[2] As an aside: note that doing a broadcast + convergecast "synchronizes" the clocks to some extent.

```
states:
  t           : Time = $t_{fix}$                // predefined time
  money       : Nat = m_$i$;                         // some initialization
  send        : [1..n] -> (queue of Nat);
  clock       : Clock = 0;
  counting[j] : notyet + ongoing + done = notyet   // 3 stages of counting, per neighbor
                                                   // notyet is universal for all neighbors,
                                                   // however
transitions:
  //-------------------------------------------------------------------------
  send(m,c)$_{i,j}$                               // standard trans'ns
    precondition:  m is first on send(j)   // using the output
    effect:        remove m from send(j);  // send buffers
                   clock := clock + 1;     // Lamport' clock.
                                           // This transition does not change
                                           // the money -> we don't check whether
                                           // we cross the time line.
  //---------------------------------------------------------------------
  receive(m,c)$_{i,j}$                            // we receive new money
    effect:
            clock := max(clock,c) + 1;     // Lamport's clock
            if   $t_{fix} \leq$ (clock,i) // we are after the line
            then if     counting[j] = notyet
                 then  counting[j'] := ongoing, $\forall$ j' $\in$ nbrs
                                           // we must copy money to balance only once
                       balance     := money; // freeze our own current money
                                           // as of now (before the update!)
                  else skip                 // so far, the new money has
            fi;                             // not entered the books.
                                            // if we are before the line -> balance = 0
                                            // otherwise: balance is the amount before that
                                            // reception and counting is on.
            if (c,j) < $t_{fix}$ $\wedge$                // money sent  before the line
               counting[j] = ongoing       // accumulate transit money;
            then balance := balance + m;
            else counting[j] := done      // we are through with j, since the queues
                                          // are fifo and the times are monotone.
            money := money + m;           // we must not forget the general bookkeeping...


  //------------------------------------------------------------
  transfer$_{i,j}$
    precondition:  (m,j) $\in \varphi$(state)      // some condition
                                                 // m = 0 counts as dummy
    effect:                                      // non-det. autonomous
            clock   := clock + 1;                // also here we clock
            if $t_{fix} \leq$ (clock,i)   // we are after the time line
            then if   counting[i] = notyet       // we have crossed it right now
                 then counting[j'] := ongoing, $\forall$ j' $\in$ nbrs
                      balance      := money -m;  // freeze the money, after
                                                 // the update! The value of money
                                                 // is before the time line. The amount
                                                 // m will be counted at the receiver.
                  else
                      skip
            else skip;                        // time has not yet come
            money   := money - m;             // don't forget to bookkeep and
            send[j] := send[j] :: (m,(i,clock)); // send the timestamped money indeed
  //------------------------------------------------------------
  balance$_i(m)$                             // announce the result
    precondition:
      m = money;
      counting[j] = done, $\forall$ j $\in$ nbrs // all transit money has arrived
    effect:
      clock := clock + 1;

tasks:
  { send(m)$_{i,j}$ | j ∈ nbrs } +
  { transfer$_{i,j}$ | j ∈ nbrs }
```

One way out is that each process performs the accounting procedures for an "infinite" number of predefined time lines while the local time proceeds. This means we have to generalize the above process such such that there are balances for a countable sequence of ever-growing time-lines $t_0, t_1, t_2 \ldots$. In the above algorithm with a fixed time, we modeled the termination and output simply by requiring, that the process has reached the predefined deadline and especially that it has received all transit money from all neighbors.

Now we have to be more careful. If a process has received all transit money belonging to time t_r, this does guarantee that all other processes did not start after this deadline, leading to the error described above. So a process cannot send out it's balance value for t_i unless it is sure that all others in the network had been able to *meet this deadline.* In order to find out, it sends around a wave of broadcast/convergecast messages. Each process answers whether it has a valid balance wrt. proposed date.

```
count_money$_i$    //  count money
types:      Clock = Nat;
            Time  = Clock * Pid              // Lamport: lex. order
messages:
           transfer of Nat * Clock;

signature:
  input:    receive$_{i,j}$(m), j ∈ nbrs
            revision$_i$
  output:   send(m)$_{i,j}$,  j ∈ nbrs
            balance(m)$_i$                      // send balance to env.
  internal: transfer$_{i,j}$,  j ∈ nbrs

states:
  t              : Nat -> Time                         // growing sequence of time lines
  money          : Nat = m_i;                       // some initialization
  balance        : Nat -> (Nat + undef)  = undef, for all r \in Nat
  send           : [1..n] -> (queue of Nat);
  clock          : Clock = random numer;               // some init value
  counting[r][j] : notyet + ongoing + done = notyet    // 3 stages of counting, per deadline and
                                                       // per neighbor
                                                       // notyet is universal for all neighbors,
                                                       // however
transitions:
  //-----------------------------------------------------------------------
  send(m,c)$_{i,j}$                          // standard trans'ns
    precondition:  m is first on send(j)   // using the output
    effect:        remove m from send(j);  // send buffers
                   clock := clock + 1;     // Lamport' clock.
                                           // This transition does not change
                                           // the money -> we don't check whether
                                           // we cross the time line.
  //-------------------------------------------------------------------
  receive(m,c)$_{i,j}$                          // we receive new money
    effect:
      newclock := max(clock,c) + 1;         // Lamport's clock
      ∀ k ∈ Nat.            // this only _looks_ like an
                                                 // infinite loop :-) If we loop from zero
                                                 // upward, then comes some k' s.t.
```

```
                                                // t(k') > (newclock,i). By assumption,
                                                // all k'' > k, are even larger; hence
                                                // we can break out of the loop here.
                                                // The loop can be further optimized by
                                                // realizing that k's belonging to
                                                // past time points need not be rechecked,
                                                // but that's finetuning...
            if     (clock,i) < t(k) ≤ (newclock,i) // we just crossed t(k)
            then  counting[k][j'] := ongoing, ∀ j' ∈ nbrs
                                     // we must copy money to balance only once
                   balance[k]    := money; // freeze our own current money
                                          // as of now (before the update!)
                                          // if we never _cross_ the line, balance[k]
                                          // will remain undefined.
                    else skip             // so far, the new money has
            fi;                             // not entered the books.
                                          // if we are before the line -> balance = 0
                                          // otherwise: balance is the amount before that
                                          // reception and counting is on.
          if (c,j) < t(k) ∧                      // money sent  before the line
             counting[k][j] = ongoing       // accumulate transit money;
          then balance[k] := balance[k] + m; // note that balance[k] on the lhs is not undefined he
          else counting[k][j] := done      // we are through with j and t(k), as the queues
                                        // are fifo and the times are monotone.
    end loop;
    money := money + m;           // we must not forget the general bookkeeping...
    clock := newclock;
//-------------------------------------------------------------
transfer$_{i,j}$
  precondition:  (m,j) ∈ φ(state)      // some condition
                                               // m = 0 counts as dummy
  effect:                                      // non-det. autonomous
    newclock   := clock + 1;
    ∀ k ∈ Nat                     // again,  no real danger here
      if   (clock,i) < t(k) ≤ (newclock,i)     // we crossed line t(k)
      then counting[k][j'] := ongoing, ∀ j' ∈ nbrs
           balance[k]      := money - m;           // freeze the money, after
                                               // the update! The value of money
                                               // is before the time line. The amount
                                               // m will be counted at the receiver.
    end loop;
    money   := money - m;                 // don't forget to bookkeep, to
    clock   := newclock;                  // update the clock, and to
    send[j] := send[j] :: (m,(i,clock)); // send the timestamped money indeed
//-------------------------------------------------------------
finish-k$_i$:                            // aux. transition
    precondition:
      counting[k][j] = done, ∀ j ∈ nbrs // all transit money has arrived
    effect:
      counting[k][i] = done;
      clock := clock + 1;
//-------------------------------------------------------------
broadcast$_i$                                // try to get time $t_k$ through
                                             // This will work similar than the
                                             // asychronous broadcast with ack.
                                             // We have to be careful, however, that
                                             // other processes will use the same procedure
                                             // such that we don't mess up different
                                             // broadcasts/convergecasts...
  precondition:
    k is minimal s.t.
      counting[k][i]= done
  effect:
    clock := clock + 1;
    send[j] := send[j] :: (candidate(k), i, clock),   for all neighbors j
```

```
                                                  // the identity must be attached, because
                                                  // many broadcasts will be done at the
                                                  // same time.
   //--------------------------------------------------------------------------------
   receive(candidate(k),j',c)$_{j,i}$          //
     effect:
       if    bcast[k][j'] = false                  // I have not yet heard of the broadcast
                                                  // for time k, started by j'
       then parent[k][j'] := j;             // for convergecast
            bcast [k][j'] := true
            send[j] := send[j] :: (candidate(k,j'), clock),  for all neighbors j \ parent[k][j']
       else send[j] := send[j] :: (ack(k,j'), clock)  // immedatly do the convergecast
   //--------------------------------------------------------------------------------
   receive(ack(k,j'))$_{j,i}$                 // the convergecast, initiator j', time k
     effect:
        acked[k][j'] := acked[k][j'] + j;
   //--------------------------------------------------------------------------------
   converge$_i$                // similar to report of simple bcast with acks'
     precondition:
       parent[k][j'] $\neq$ null
       acked[k][j'] = nbrs - parent[k][j']
       counting[k]  = done        // I have met deadline k!
     effect
        send[j] := send[j] :: (ack(k,j'), clock), j = parent[k][j]
   //--------------------------------------------------------------------------------
    balance(m)$_i$
      precondition:
        reported = false
        parent[k][i] = null;   // I'm one root for time k
        acked[k][i]  = nbrs    // all my neighbors have answered
      effect:
        reported := true;     // I'm done! And I don't do a broadcast as root later!
                              // I will participate in other processes broadcast/
                              // convergecast round, of course.
   //--------------------------------------------------------------------------------


 tasks:
   { send(m)$_{i,j}$ | j $\in$ nbrs } +
   { transfer$_{i,j}$ | j $\in$ nbrs }
```

Aufgabe 4 (Unlogische Zeit (4 Punkte)) Bearbeiten Sie Aufgabe 18.4(a). Geben Sie dabei für das Weglassen *jeder* der Bedingungen für logische Zeit eine Transformation an.

Solution: Let us first recapitulate the 4 conditions for logical time.

uniqueness: each event gets a unique timestamp.

local causality: locally for one process, the timestamps for each are strictly increasing.

communication causality: matching send and receive-events are strictly ordered.

finiteness: for each value from the time domain, there are finitely time values strictly smaller.

Let's address the conditions in the order given.

1. We can drop the *uniqueness* by taking the same setting as in Lamport's implementation but extend the time domain by a universally smallest time event, say $\perp$. All processes start with this value as time stamp for the first event, then this violates the uniqueness requirement, but leaves all other conditions unharmed. Indeed, this idea works also for Welch's solution and, as it seems, for any implementation of logical time.

   Another simple idea is to take Lamport's clocks and omit the process idea from the time values. This clearly destroys uniqueness of the timestamps. By counting up the clock as before, and by updating one own clock when learning that one "lags behind" upon message reception, the conditions for strict monotonicity are still met. Obviously, also the finiteness requirement is not destroyed.

2. Here the problem is to keep uniqueness, which for part 3 is guaranteed by the different process id. A possible route simply make rather weak restrictions on the time domain, namely to require a *quasi-ordering,* only. [1] is not explicit what should be expected of the time domain beyond the 4 conditions of logical time.[3]

   If one takes Lamport's clocks, pairs of local clock value and process id, and if one uses as order on $\mathit{Clock} \times \mathit{Procid} = \mathbb{N} \times \mathbb{N}$ simply the order relation on `Clock`, i.e., one simply does not use the process id as "tiebreaker", then one obtains a quasiordering, antisymmetry is gone. The other 3 conditions of logical time still hold.[4] Another strategy could be to count up upon reception, but when on counts for an non-reception, one allows to go back in the count. To avoid going back indefinitely, one could take the smallest unused number so far.

3. To violate causality for communication, is simple. Using Lamport's clocks as starting point, we attack the *strictness* part of the definition. We count up *locally* as before, but for the receive-event, we do not *count up wrt. transmitted timestamp.* This means, upon reception, the new clock value is the maximum from the transmitted clock value and the incremented local clock value. Note that especially the uniqueness requirement still hold. The worst that can happen is the the clock values of sending and reception are *identical* (in case the receiver lags behind), but then the process id can be used as tiebreaker.[5]

4. That's trivial. Given a time domain T, we may simply add one extra value, say ∞ with $t < \infty$ for all t from T. Since T must be infinite,[6] we immediately obtain a violation of the finiteness assumption: taking a *infinite* sequence α, a *logical* time assignment assigns to each of the infinite many events of α some $t \in T$.

[3] Those have of course, some implications on the "causality relation". For instance to be strictly increasing implies to be in some "strict order relation".

[4] To be precise, the finiteness condition hinges on the fact that there are only finitly many processes. For Lamport's and Welch's implementation, this assumption is not needed!

[5] Note that this works only if a process cannot send messages to itself!

[6] The conditions for strict monotonicity imply this.

Since already the change the underlying time domain does the trick, one firstly can use any logical time implementation which consequently never reaches the additional value ∞, and secondly, the other 3 conditions are trivially still hold, since ∞ is a time stamp which is never used in an assignment nor in an implementation. Note that a time assignment from an execution into the time domain is required to be *injective,* but not *surjective.*

Another solution could be to use a dense time domain, for instance the rational numbers.

□

References

[1] Nancy Lynch. *Distributed Algorithms.* Kaufmann Publishers, 1996.