Algorithmic Modeling and Complexity **Fall 2003**

Lecture 16: 24 October

Lecturer: M. Veldhorst *Scribe: M. van Es*

16.1 Overview

In the previous lecture we showed the nonexistence of a deterministic 1-crash-robust distributed algorithm for the consensus problem. Today we will try to answer two questions: (i) In what fault model does a robust deterministic distributed algorithm exist, and (ii) for which non-trivial problem does a t-crash robust distributed algorithm exist for some $t \geq 1$?

16.2 Introduction

Remember that there is no 1-crash-robust deterministic consensus algorithm ([3] and the previous lecture). Therefore, we can conclude the following about solving the consensus problem in distributed systems:

- It is necessary to use a more strict fault model (not allowing the crash fault).
- We must consider another fundamental problem to be used as a building block in fault-tolerant distributed systems; such another problem does exist for the crash model.
- We must consider randomization.
- It might help to use a weaker definition of termination.
- One could assume that the hardware supports some form of synchronization.

In this and the next lecture we will only deal with the first two and the last option.

16.3 Using a stricter fault model

The fault tolerant model used here is the *initially dead model.* A process is either initially dead and stays dead or it is always alife. Now consider the consensus problem assuming the four following restrictions:

- N processes, t are dead initially, but t is unknown
- $t < \frac{N}{2}$
- each process p_i has an input x_i $(1 \leq i \leq N)$

- all life processes should come up with the same binary value (the output).

We assume that the problem is not trivial: there are different inputs or different executions on the same inout that must lead to different outputs.

A dead process, naturally, does not send nor receives any messages. Hence, if a message is received by a process, the received message must originate from a life process. Now we present the consensus algorithm. It consists of $2 + \log N$ stages. Let P be the set of processes and let $L = \lceil \frac{N+1}{2} \rceil$.

```
1:  for  all   processes $p$:
    do   for   all processes $q \neq p$ :
               $p$ sends a message (containing an identification of $p$) to $q$
         endo
    endo
    for  all   processes $p$:
    do   $p$ receives $L - 1$ messages and ignores other messages sent to it in stage 1.
    enddo

for  $i \leftarrow 1$ to $\log N$ do
stage $i + 1$:  for all processes $q$:
                do $q$ creates a message $m$ with info of each $r \in Pred(q)$
                   $q$ sends $m$ to every other process.
                   $q$ waits until it receives a message from each $r \in Pred(q)$
                   $Pred(q) \leftarrow Pred(q) \cup \bigcup_{r\ \in Pred(q)} Pred(r)$.
                   $q$ stores the input values of all its new predecessors.
                endo
enddo
final:    for  all  processes $p$
          do   $p$ computes the knot $K$ of the induced subgraph of $G$ determined by $Pred(p)$.
               $p$ computes the consensus value using the input values of the processes in $K$.
          endo
```

A life process p receives in stage 1 precisely $L - 1$ messages and therefore knows of $L - 1$ other life processes. Let $G = (P, E)$ be the directed graph with processes as vertices and with edges $(i, j) \in E$ if process j has received and read in stage 1 a message from process i. Then, when a life process i has finished its stage 1, $indegree(i) = L - 1$. The dead processes x are isolated and have $indegree(x) = outdegree(x) = 0$.
Observe that the stages are not used as synchronization for the processes. As soon as a process has finished a stage, it proceeds with the next one. Actually, the messages are in some sense used for synchronization: a process does not proceed with the algorithm unless it has received enough messages.

G is a directed graph with strongly connected components.

Definition 16.1 *A graph $G = (P, E)$ is strongly connected iff there is a path from i to j for every $i, j \in P$. A strongly connected component of G is a maxim***al** *induced subgraph that is* *strongly connected.*[1]

There exists an algorithm that, given G, determines the strongly connected components in linear time (linear in the number of nodes and edges).

Considering again the graph G with strongly connected components, G can be viewed as a supergraph H in which the strongly connected components of G are the supernodes and there is an edge from one supernode to another if there is an edge from one vertex in the first supernode to a vertex in the second supernode. The graph of the supernodes is acyclic. Otherwise, for supernodes that form a cycle, these strongly connected components together also form a larger strongly connected component in G. This is not allowed, because the strongly connected components are defined as maximal subgraphs.

Definition 16.2 *The supergraph H is defined as follows. Suppose G has strongly connected components $V_1, \ldots, V_k$. Then $H = (\{1, 2, ..., k\}, F)$ where $F = \{(i, j) | i \neq j, \exists v_1 \in V_i$ and $\exists v_2 \in V_j$ with $(v_1, v_2) \in E\}$.*

Given G, H can be determined sequentially in linear time (linear in the amount of vertices and edges).

16.3.1 The knot subgraph

An induced subgraph K of G is called a *knot*, if K is a strongly connected component of G, K has ≥ 2 processes (i.e., nodes), and K has no incoming edges. By the construction of G it also holds that

$$|K| \geq L$$

Suppose there are two knots in G. These two knots have no nodes in common, and each of them contain at least L nodes. Hence the two knots together contain at least $2L$ nodes, leading to the following situation:

$$2L \geq 2 \cdot \frac{N+1}{2} = N + 1 > N$$

The number of nodes in two knots would yield a total number of nodes in G that is larger than N, thus there can be only one knot.

Lemma 16.3 *Let p be a life process. Then there is a life process $q \in K$ such that $q \to p \in E$.*

Proof: Let p be a life process with *indegree*$(p) = L - 1$. Then there are $q_1, \ldots, q_{L-1}$ such that $(q_i, p) \in E$. If $p \in K$, then by the definition of a knot, $q_i \in K$ for each i $(1 \leq i \leq L - 1)$.

Now suppose $p \notin K$. If one of the q_i is in K, the lemma holds. So assume none of the q_i is in K. The G contains at least L nodes not in K. K has itself at least L nodes. Hence, G contains at least $2L$ nodes. Contradiction. Hence one of the q_i must be in K. ■

[1]These are standard definitions in graph theory and can be found in many textbooks on graph theory and graph algorithms

16.3.2 Shortcuts

Now let us consider the stages 2 and further in the algorithm. The general idea all input values of the processes in K should be broadcasted to all the other life processes, such that all processes know of the processes in K and their input-values. The life processes will use them to compute a value (0 or 1) that is the consensus value. From the previous lemma we know that each life process is reachable from the knot K.

Suppose a path $p_1, p_2, \ldots, p_x$ exists. We add edges to G by shortcutting: add for each i $(1 \leq i \leq x-2)$ the edge (p_i, p_{i+2}). For each path of length two, the shortcut is added. If we repeat this shortcut for each path of length 2, $\log x$ times, each p_i has an incoming edge from p_j for each j $(1 \leq j < i)$. Observe that in adding edges by shortcutting, the number of edges in the graph may increase, but there is no change in the strongly connected components (considered as subsets of nodes).

Let

$$Pred(p) = \{q | q \rightarrow p \in G\}$$

so q is the 'predecessor' of p. Shortcutting means that all predecessors of q must become predecessors of p. The life process q makes a long message composed of:

- $id(q)$, its own identification
- x_q, the input-value of q
- $id(r)$, x_r for a $r \in Pred(q)$, the pair of the identification and the input-value of all its predecessors.

Process q "shouts" this message around to be picked up by all p of which q is a predecessor. Then it waits until it hears the shouts of all its own predecessors r. Note that this waiting is only successful because the crash fault is not allowed in this model. Then q updates het set of predecessors: $Pred(q) \leftarrow Pred(q) \cup \bigcup_{r\ \in Pred(q)} Pred(r)$.

Because the longest path in G has length at most N just before stage 2, $\log N$ repetitions of the updating described above, suffice to give each life process the input values of all processes in the knot.

Finally, the consensus value is computed independently and deterministically by all life processes using the input values of the processes in the knot. Hence we have shown the correctness of the following theorem.

Theorem 16.4 ***(Fischer, Lynch and Paterson 1985 [3])*** *The consensus problem can be solved if we have $t < \frac{N}{2}$ initially dead process.*

16.3.3 Efficiency

It is quite common to define the efficiency of distributed algorithms in terms of communication: what is the maximum number of bits sent through the network?

Exercise. Determine the efficiency of the consensus algorithm, measured as big O of a function of N (the number of processes). Assume that the identification and input of each process consists of $O(\log N)$ and 1 bit, respectively.

Exercise. The original paper of Fischer et al. [3] has a different algorithm for stage 2 and further:

```
2:  for all processes $p$:
    do $p$ creates a message $m$ with $(p, Pred(p), x_p)$
       $p$ sends $m$ to every other process.
       $Knowsof(p) \leftarrow \{p\} \cup Pred(p)$
       $Received(p) \leftarrow \{p\}$
       while not $Knowsof(p) \subseteq Received(p)$
       do    pick up a message $(r, Pred(r), x_r)$
             $Knowsof(p) \leftarrow Knowsof(p) \cup Pred(r)$
             $Received(p) \leftarrow Received(p) \cup \{r\}$
             $p$ stores vertex $r$, the incoming edges of $r$ and the input $x_r$.
       endo
    enddo
3:  for all    processes $p$
    do $p$ computes the knot $K$ of the induced subgraph of $G$ determined by $Knowsof(p)$.
       $p$ computes the consensus value using the input values $x_q$ of the processes $q \in K$.
    endo
```

Prove the following graph-theoretic lemma:

Lemma 16.5 *Let $G = (V, E)$ be a directed graph, $X \subseteq V$, $x \in X$ and C is the strongly connected component of G containing x. Suppose $(v, w) \in E$, $w \in X$ implies $v \in X$. Then $C \subseteq X$.*

Show finally that this stage 2 and 3 correctly finishes the consensus algorithm. Determine the efficiency of this modified algorithm (with the same assumptions as in the previous exercise).

16.3.4 Randomization

It is unfortunate that in the crash or Byzantine fault model, even a simple problem as consensus cannot be solved deterministically. The impossibility result of Fischer et al. [3] (see also previous lecture) shows only that at least one execution of one specific input may lead to postponement of a necessary decision. One could look for algorithms with many possible executions for each input (initial configuration) where the algorithm will decide (determine a correct output) on most executions. Thinking in this way, Bracha and Toueg [2] were able to design randomized consensus algorithms for the crash and Byzantine failure model, with at most $N/2$ and $N/3$ faulty processes, respectively, in which the probability of a terminating execution within k steps tends to 1 if k tends to ∞.

16.4 The renaming problem.

As stated in the introduction we will inspect a problem different from the consensus problem, to see whether there exists a non-trivial problem that can be solved in the crash fault model. The renaming problem is defined as follows: processes have different identifications, assign to the life processes new (different) identifications from a limited set of characters (e.g., a small number of bits)?

Given: for each process p, $id(p)$ is the identification of p, and $p \neq q$ implies $id(p) \neq id(q)$.

Determine: a $new(p)$, the new identification of p, for each life process p such that

$$id(p) \neq id(q) \text{ implies } new(p) \neq new(q) \text{ for } p \text{ and alife } q$$

Fault model: The used fault model is the crash model, and at most $t < \frac{N}{2}$ processes might crash (t is known).

In the algorithm process p maintains a name space V_p. Initially $V_p = \{id(p)\}$. During the algorithm V_p will grow and note that once $id(q)$ has been added to V_p, $id(q)$ will forever remain in V_p. Process p will receive name spaces of other processes, and it counts the number of times it receives a copy of its current name space. As soon as p has counted to $N - t$, the current name space of p is called stable.
When p is alife, then sometime V_p will become "stable". Even after it has become stable, the execution of the algorithm for process p proceeds and V_p may still grow, become unstable, and may become stable for a second (a third, ...) time.

A new identification will be assigned to p according to the rule

$$new(p) = (|V_p|, (\text{rank of } id(p) \text{ in } V_p))$$

with rank being an integer related to the size of V_p. If $id(p)$ is the largest element (name) in V_p, then its rank equals 0. If $id(p)$ is the second largest element in V_p, its rank will be 1, etc. The used V_p is the set V_p when it just has become stable for the first time. Now there could three possibilities for different processes p and q to which new names will be assigned:

1. $|V_p| \neq |V_q|$. Then $new(p) \neq new(q)$.
2. $|V_p| = |V_q|$ and $V_p \neq V_q$. The algorithm will be designed in such a way that this case never occurs.
3. $V_p = V_q$. Then the ranks of $id(p)$ and $id(q)$ are different, because (the identifications of) p and q are different.

The new names are chosen from a set of size at most $(N - \frac{1}{2}t)(t + 1)$.

16.4.1 Algorithm for renaming

```
Initially set $V_p = \{id(p)\}$ and the counter $c_p \leftarrow 0$
shout($V_p$)
while true
do  receive($V$)
    if   $V = V_p$
    then $c_p \leftarrow c_p + 1$
         if   $c_p = N - t$ and no $new(p)$ has been assigned
         then *$V_p$ is stable*
              $new(p) \leftarrow (|V_p|, \text{(rank of } id(p) \text{ in } V_p))$
         endif
    else if $V \subseteq V_p$   then ignore
    else if $V_p \subseteq V$   then $c_p \leftarrow 1$; $V_p \leftarrow V$
                                else $c_p \leftarrow 0$; $V_p \leftarrow V \cup V_p$
    endif
    shout($V_p$)
enddo
```

Now the question arises whether or not this algorithm terminates. This is not the same as the question whether it stops running, which it will not, but whether it will give new identifications (names) to all life processes.

Lemma 16.6 *Assume that the algorithm terminates.*
Let p and q have decided on a new name, based on a certain V_p and V_q, respectively, then $V_p \subseteq V_q$ or $V_q \subseteq V_p$ or both.

Proof: Process p has received V_p $N - t$ times from processes $r_1, r_2, \ldots, r_{N-t}$ and process q has received V_q $N - t$ times from processes $s_1, s_2, \ldots, s_{N-t}$. Because $t < \frac{N}{2}$ holds, one of the r_i and s_j is the same process; let this be process r. This process r has sent V_r to p which led to an increase of c_p. Process r also sent V_r' to q which led to an increase of c_q. This means

$$\begin{aligned} \mathrm{V}_r' &= \mathrm{V}_q \\ \mathrm{V}_r &= \mathrm{V}_p \end{aligned}$$

If r sends V_r' to q before it sends V_r to p: $V_r' = V_q$ and $V_r = V_p$ and $V_r' \subseteq V_r$ hold which implies $V_q \subseteq V_p$. If r sends V_r to p before it sends V_r' to q, then $V_p \subseteq V_q$. Hence the lemma holds. ■

With this lemma it is shown that the case 2 in the assignment of new names (i.e., the case $|V_p| = |V_q|$ and $V_p \neq V_q$) does not occur.

What if the algorithm runs forever and would not terminate (i.e., some life process never assigns a new name to itself).

Lemma 16.7 *Each life process p reaches a stable set at least once in each fair t-crash execution.*

Proof: Let p be a life process. The set V_p remains expanding while it is certain that always $|V_p| \leq N$. Hence there must be a largest name space $Vmax_p$ of process p. As soon as V_p has become this set $Vmax_p$, p shouted $Vmax_p$ to all life processes. Hence, from some moment onwards we have $Vmax_p \subseteq V_q$ for all life processes q (here we use the assumption of fair executions).

Claim 16.8 *For each life process q sometime $V_q = Vmax_p$.*

Proof: Suppose the claim does not hold. Then for some life process q we have that $Vmax_p$ is a proper subset of V_q at some moment. Then of course q shouts this V_q around, which is picked up by p. Then p expands V_p which was already the maximum sized set during the whole execution of p. Contradiction. ■

Thus each life process q sends once $Vmax_p$ to p. With $N - t$ life processes, $Vmax_p$ becomes a stable set. ■

Theorem 16.9 ***(Attiya et al., 1990 [1])*** *The renaming problem can be solved in a t-crash model in which $t < N/2$.*

References

[1] Attiya, H., A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, Renaming in an asynchronous environment, *Journal of the ACM*, volume 37, pages 524–548, 1990.

[2] Bracha, G. and S. Toueg, Asynchronous consensus and broadcast protocols, *Journal of the ACM*, volume 32, pages 824–840, 1985.

[3] Fischer, M.J., N.A. Lynch, and M.S. Paterson, Impossibility of distributed consensus with one faulty process, *Journal of the ACM*, volume 32, pages 374–382, 1985.