Algorithmic Modeling and Complexity **Fall 2003**

Lecture 15: 21 Oktober

Lecturer: M. Veldhorst *Scribe: J.J. Franken*

Overview

This is the first lecture of a series about Distributed Algorithms (DA) and fault tolerance given by M. Veldhorst.

In distributed algorithms a number of hardware or software components cooperate in order to reach a common goal. The components operate independently and the only way to communicate is by sending messages from one component to another. The purpose of fault tolerance is to design distributed algorithms that reach the common goal even if some components behave erroneously.

In this lecture we will concentrate on modeling erroneous behavior of components and prove an impossibility result even for a very modest error model. In the next lectures we will see a number of problems for which fault tolerant algorithms do exist. The relation with different types of distributed systems is made explicit.

15.1 Fault Tolerant Distributed Algorithms

For many software systems correctness and availability of computing resources is essential. For example in medical systems, in systems for control of nuclear power plants, or in systems for control of air traffic and spacecraft. Computers might crash, or be infected with viruses, and not working properly. In real-time systems overload of resources might be considered as nonavailability.

One way of ensuring the correct behavior of systems with degrading components is of course to design software that is provably fault tolerant to some extent.

Another way is to ensure fault tolerance is to create redundancy by duplicating components. For example by incorporation of back-up systems. As soon as erroneous behavior of one component is detected, the component is replaced by a duplicate. If on the other hand, duplicates are used all the time, there might be an increase in the necessity of fault tolerant algorithms. If several duplicates measure the same physical quantity, then they should agree on the same number for it, even if their measurements differ (slightly). So one could consider the correct processing of erroneous data as an erroneous processing of correct data.

15.2 A Distributed System

In this section we will model distributed systems in order to design algorithms for them and to reason about the algorithms. Basicly a distributed system consists of a finite set P of processes and a communication system. We will first describe processes, and then the distributed system.

Each **process** $p \in P$

- has a read-only input register X_p,
- has an fixed identification,
- knows the fixed identifications of all other processes in P,
- has a write-only output register Y_p,
- runs forever,
- may send messages m to other processes in P by inserting them in M,
- may poll M whether M contains a message m sent to p, and if so, read m (and deletes it from M).

The program of a process p is a set of guarded commands where each guarded command consists of a finite sequence of statements and an associated condition. In a processing step p chooses nondeterministically a guarded command q of which the condition evaluates to true and executes the sequence of statements of q.

As long as Y_p has not been written we assume that it has the value b; b is assumed to be a value that p cannot assign to Y_p.

In order to reason about executions of a process we use the following concepts:

(internal) state: the process' values of the input registers, the output registers, and the internal variables (if any),

initial state: a state in which only the input values are known and in which the output register has value b,

decision state: a state where Y_p has a value different from b (because Y_p is write-only, we have that once p is in a decision state, it remains in it forever).

Actually, the program of p can be considered as a **transition system** that makes p to step from state to state.

Now we will concentrate on the concept of a distributed system. A distributed system consists of a finite set of processes and a communication system M. M contains the messages 'in transit'. A message has one destination (given as the identification of a process), one source (given as the identification of the sending process) and a contents. A message in M with destination p can be taken from M only by process p. Observe that messages in M all with destination p can be taken out from M in any order. p is unable to coerce a specific order. This means that if p wants to process messages in a certain order, it must keep a buffer of messages it took from M but that it still has to process. p is in full control of this buffer and hence is able to process messages in the buffer in any order it considers appropriate. Obviously this makes the design of a program of p rather difficult. In this series of lecture on distributed algorithms we will give no special attention to buffering messages, though in some sense we use it.

In order to reason about executions of a distributed system we use the following concepts:

configuration: (or global state) the set of internal states and the contents of M.

initial configuration: the set of initial internal states and an empty M.

decision state: a state where Y_p has a value different from b (because Y_p is write-only, we have that once p is in a decision state, it remains in it forever).

event: an event is a transition from one configuration γ to another configuration γ' (denoted as $\gamma \to \gamma'$) due to the execution of a series of statements associated with a guarded command of one process.

reachability: a configuration γ' is reachable from a configuration γ if there is a finite sequence of configurations $\gamma = \gamma_0 \to \gamma_1 \to \gamma_2 \to \cdots \to \gamma_i = \gamma'$).

execution: an execution is an infinite sequence of configurations ($\gamma_0 \to \gamma_1 \to \gamma_2 \to \cdots$) in which γ_0 is an initial configuration and $\gamma_i \to \gamma_{i+1}$ is an event.
Executions are infinitely long because we considered processes to run forever. This is done only for theoretical reasons.

Actually, the distributed system S can be considered as a **transition system** that makes S to step from configuration to configuration.

A nondeterministic distributed system is correct if it is correct for each execution[1]. This is rather cumbersome because there may be executions where an essential message is never picked up by some process because there always is another guarded command that evaluates to true. Hence, we want to restrict ourselves to so-called *fair* executions.

Definition 15.1 (fair execution) *An execution of a distributed algorithm is fair if for each process p and each guarded command c of p it holds that, once the condition of c evaluates to true, then after a finite number of steps the condition of c does not evaluate to true or the statements of c are executed by p.*

This is important for the communication between processes. It is reasonable to assume that each process p has a guarded command with the condition whether M contains a message for p. In a fair execution a message sent to p will eventually be picked up by p.

[1]This is different from the concept of correctness of nondeterministic algorithms in the theory of NP-completeness which is defined as correctness for at least one execution.

Observe that time hardly plays a role in the above definitions. Processes may run slowly of fast, messages sent in some order may be received in a different order, etc..

In this section we described asynchronous networks. There are other network models. For example, synchronous networks in which the aspect of time in an execution is incorporated, anonymous networks in which the identification of other processes is not known. For an introduction in distributed algorithms we refer to Tel [2].

15.3 Faults in the Model

In this section we will discuss the modeling of faults. For the design of fault tolerant distributed algorithms it is also important to describe what still performs correctly in the different fault models. We consider the following fault models:

initially dead: Some processes never start to execute their program, i.e., they are 'dead'. Life (that is not dead) processes execute their program correctly. It is not known which processes are dead.

crash: A number of processes may crash sometime during their exections. A crash process executes correctly until is crashes and halts it exection completely. The processes that do not crash, execute their program correctly. It is not known in advance which process will crash.

omissions: Some processes may skip statements during the execution of their program, but executed statements are executed correctly.

Byzantine: a Byzantine process shows arbitrary behavior (i.e.. an execution in which at least one step arises which leads from configuration C to a configuration C' which is not an event. A Byzantine process knows that it is Byzantine. Correct processes do not know what the Byzantine processes are.

An intially dead process is a special case of a crashed process; a crashed process is a special case of a process with omissions, and a process with omissions is a special case of a Byzantine process. So, when fault tolerance for Byzantine faults are achyieved, then fault tolerance for the other given fault models are achieved.

Observe that in these models the communication system is correct: the destination and the source of a message do not erroneously change, the contents of a message does not change, messages are not lost, and process p cannot take from M a message with destination q when $q \neq p$.

Definition 15.2 (A t-FAIL-robust-system) *A t-FAIL-robust-system (with one of the fault-models substituted for FAIL) with N processes is a system in which at least $N - t$ processes perform their local algorithm correctly and determine the right answer. The other processes may fail according to faults of type FAIL.*

In the current and next lectures we will deal with the question whether provably t-FAIL-robust algorithms can be designed. In robust algorithms the correct processes are able to cooperate correctly and are able to find a correct solution to a problem even if a number of processes fail permanently. To prove them correct, the precise type of faults should be known as well as (a bound on) the number of faulty processes.

So-called *stabilizing algorithms* obtain fault-tolerance in a different way. Stabilizing algorithms may be in an incorrect state but when no faults occur anymore, they move to a correct state eventually. So, when faults do not occur too frequently, stabilizing algorithms find a correct solution. Tel [2] gives attention to stabilizing algorithms.

15.4 The Consensus Problem

An important problem in distributed systems is that the processes find consensus about a set of inputs.

Given: an input bit: X_p for each process p,

Required: for each process p an output bit: $Y_p \in \{b, 0, 1\}$. If two processes p and q have written their output register, they have written the same value:

$$Y_p \neq b \text{ and } Y_q \neq b \text{ implies } Y_p = Y_q$$

In case faults do not occur, a distributed algorithm for the consensus problem is easy: each process p sends its input X_p to all other processes, and receives the input of all other processes. Then each process p determines deterministically the consensus value based on all inputs.
But now consider for example a system with one dead process. It is possible that all other processes are waiting for the receipt of the input of the dead process, and the algorithm never writes an output.

Above the consensus problem is not specified properly. A solution could be that independent of the input all processes write the zero bit. Actually we want to design a so-called t-FAULT-robust consensus algorithm:

Definition 15.3 *A t-FAULT-robust consensus algorithm for inputs X_p determines Y_p outputs where there are at most t faulty processes of type FAULT and that has the following three characteristics.*

fair termination: *In each fair execution of each non-faulty process p, p eventually writes its Y_p register.*

agreement: *Each configuration γ' reachable from an initial configuration γ satisfies*

$$Y_p \neq b \wedge Y_q \neq b \Rightarrow Y_p = Y_q \textit{ for all non-faulty processes } p \textit{ and } q$$

non-triviality: *for $v = 0$, respectively $v = 1$, there is an input and a reachable configuration in which some non-faulty process p writes the value v in Y_p.*

15.4.1 An impossibility result

In this section we will show an impossibility result Theorem 15.7 concerning crash-robust deterministic consensus algorithms.

This theorem considers a slightly weaker concept of a crash. If a process crashes while executing the sequence of statements of a guarded command, it still finishes the execution of the sequence. The impossibility result obviously still holds for stronger concept of a crash.

Deterministic here means that the program of each process is deterministic except for the order in which messages are taken from the communication system. Hence the execution of a guarded command can be considered to be triggered by the receipt of a message m, and the result in process p is determined by m and the state of p at the start of the execution. Actually, we consider the execution of the sequence of commands of g using message m in configuration γ as the application of a transition function applied to γ. This transition function is denoted by g_m. Obviously g_m is not necessarily applicable to each configuration.

With this description of what a deterministic distributed algorithm is, Theorem 15.7 extends automatically to nondeterministic consensus algorithms because correctness of nondeterministic distributed algorithms was defined as correctness for each execution.

Before we are going to prove the impossibility result, we need some tools about transition functions and configurations.

Definition 15.4 (applicable transition function) *A transition function g_m is applicable in configuration γ if in γ message m is in the message system M. Application of g_m leads to configuration $g_m(\gamma)$.*
A sequence $\sigma = (t_1, t_2, \ldots, t_k)$ of transition functions is applicable to configuration γ if t_1 is applicable to γ and for each i $(2 \leq i \leq k)$ t_i is applicable to $t_{i-1}(t_{i-2}(\cdots t_1(\gamma)\cdots))$. With $\delta = t_k(t_{k-1}(\cdots t_1(\gamma)\cdots)$ we say that $\sigma(\gamma) = \delta$, or $\gamma \rightsquigarrow^{\sigma} \delta$.

We state the following lemma without a proof.

Lemma 15.5 *Let σ_1 and σ_2 be two sequences of transition functions applicable in configuration γ. If the sets of processes of σ_1 and σ_2 are disjoint then $\sigma_1(\sigma_2(\gamma)) = \sigma_2(\sigma_1(\gamma))$. In other words: the final result is independent of the order of processing of the two independent sequences of transition functions.*

The next definitions are specific for the consensus problem.

Definition 15.6 (deciding, valence) *Process p decides on a value v if it writes v in Y_p.*
Configuration γ is 0-decided (1-decided) if for some process p the output register Y_p satisfies $Y_p = 0$ $(Y_p = 1)$.
Configuration γ is v-valent $(v = 0, 1)$ if for all decided γ' with $\gamma \rightsquigarrow \gamma'$ we have that γ' is v-decided.
Configuration γ is bivalent if there is a 0-decided γ_0 and a 1-decided γ_1 such that $\gamma \rightsquigarrow \gamma_0$ and $\gamma \rightsquigarrow \gamma_1$.
Configuration γ is univalent if either γ is 0-valent or γ is 1-valent.

Obviously a robust algorithm should guarantee that no configuration is reached that is 0-decided as well as 1-decided.

Theorem 15.7 (Fischer, Lynch, Paterson 1985)
There is no 1-crash-robust deterministic consensus algorithm.

Proof: This is proven by contradiction. Assume there is a 1-crash-robust deterministic consensus algorithm A. Then in each non-faulty process eventually a *decision state* is reached. The idea here is that an execution is constructed in which the transformation from a bivalent configuration to a valent configuration is postponed forever.

The proof consists of three parts. First we will show that at least one bivalent initial configuration does exist (Claim 15.8). Secondly we will show that we can move from bivalent configuration to bivalent configuration (Claim 15.9), and finally we will construct an infinite sequence of events in which we move to bivalent configurations forever and this infinite sequence is actually a fair execution.

Assume that there are N processes, identified by there index i $(1 \leq i \leq N)$.

Claim 15.8 *There is a bivalent initial configuration.*

Proof: We will prove this by contradiction; so suppose A has no bivalent initial configuration. Hence, an initial configuration will either always lead to a 0-decision, independent of the specific execution run, or always lead to a 1-decision (independent of the specific execution), even if some process crashes. We require the algorithm to be non-trivial. Thus there is an initial configuration $X = (X_1, \ldots, X_N)$ that will always lead to a 0-decision, and there is an initial configuration $X' = (X'_1, \ldots, X'_N)$ that willl always lead to a 1-decision. Then we can construct a sequence of initial configurations ($X = X^{(0)}, X^{(1)}, \ldots, X^{(k)} = X'$) for some $k \leq N$ where $X^{(j-1)}$ and $X^{(j)}$ differ in precisely one component.

$$\begin{array}{cccc} X_1^{(0)} & X_1^{(1)} & \cdots & X_1^{(k)} \\ X_2^{(0)} & X_2^{(1)} & \cdots & X_2^{(k)} \\ \vdots & \vdots & \cdots & \vdots \\ X_N^{(0)} & X_N^{(1)} & \cdots & X_N^{(k)} \end{array}$$

Because $X^{(0)}$ leads to a 0-decision, $X^{(k)}$ leads to a 1-decision, and no initial configuration is bivalent, there must be a j such that $X^{(j-1)}$ leads to a 0-decision and $X^{(j)}$ leads to a 1-decision, while $X^{(j-1)}$ and $X^{(j)}$ differ in precisely one component, say in component p.

Now let γ_0 be the initial configuration $X^{(j-1)}$ and let γ_1 be the initial configuration $X^{(j)}$. So the inputs of all processes are the same in configuration γ_0 and γ_1, except for the input of process p. Now suppose p is the only process that crashes. Let σ be an execution for γ_0 such that $\sigma(\gamma_0)$ leads to a 0-decision. Now $\sigma(\gamma_0)$ must be the same execution as $\sigma(\gamma_1)$ (because p crashes). Then either γ_0 or γ_1 is bivalent and hence we have a contradiction! ■

Claim 15.9 *Let Γ be a bivalent configuration of a distributed algorithm, and let $t = g_m$ be a transition function for process p (possibly m does not exist). Suppose t is applicable to Γ. Let $\mathcal{C}$*

be the set of configurations reachable from Γ without *applying* t *and let* $\mathcal{D} = t(\mathcal{C}) = \{t(\gamma) | \gamma \in \mathcal{C}$ *and* t *is applicable to* $\gamma\}$ *. Then* $\mathcal{D}$ *contains a bivalent configuration.*

Proof: For technical reasons we assume that if a guarded command of a process p polls the communication system M for a message, it may receive the non-existence answer even if M contains a message for p. With this extension, we have that if a transition function is applicable but not chosen to be executed, the transition function remains applicable. It more or less means that the receipt of a message can be delayed arbitrarily long. As a consequence we have that the transition function t which is applicable to Γ, is applicable to each $\gamma \in \mathcal{C}$.

We will prove the claim by contradiction. Assume $\mathcal{D}$ has no bivalent configuration. Each configuration in $\mathcal{D}$ is either 0-valent or 1-valent. We will first show that $\mathcal{D}$ contains a 0-valent configuration as well as a 1-valent configuration. For each set T of $N-1$ non-faulty processes with $p \in T$ there is a 0-valent configuration Γ_0 that is reachable from Γ using only processes in T. Γ_0 does exist but is not necessarily a configuration in $\mathcal{C}$. In case Γ_0 is in $\mathcal{C}$, we choose $\Delta_0 = t(\Gamma_0) \in D$ and Δ_0 must be 0-valent. If $\Gamma_0 \notin \mathcal{C}$, there must be a path of configurations from Γ to Γ_0 passing through $\mathcal{D}$. Let Δ_0 be the first configuration in $\mathcal{D}$ on this path. Because by assumption $\mathcal{D}$ does not contain bivalent configurations, this Δ_0 must be 0-valent.
In a similar way there is a configuration $\Delta_1 \in \mathcal{D}$ that is 1-valent, and Δ_1 is reachable from Γ using the same set T of processes.

So now we have a path $P_0 = (\Gamma = \gamma_0, \gamma_1, \ldots, \gamma_k)$ of configurations in $\mathcal{C}$ with $t(\gamma_k) = \Delta_0$, and we have a $P_1 = (\Gamma = \delta_0, \delta_1, \ldots, \delta_h)$ of configurations in $\mathcal{C}$ with $t(\delta_h) = \Delta_1$. These two paths have the first i configurations in common for some i. For each configuration $z \in P_0 \cup P_1$ the configuration $t(z)$ exists and $t(z)$ is in $\mathcal{D}$ and is either 0-valent or 1-valent. So it must be that there are two configurations z and z' that are neighbors in $P_0 \cup P_1$ and $t(z)$ is 0-valent and $t(z')$ is 1-valent. Let t' be the transition function such that $t'(z) = z'$. Let t and t' executed by processes p and p' using messages m and m', respectively. We consider two cases.
Case 1. Process $p' \neq p$. Then with lemma 15.5 the configuration $t'(t(z)) = t(t'(z)) = t(z')$ is 1-valent. Hence from the 0-valent configuration $t(z)$ a 1-valent configuration can be reached. Contradiction.
Case 2. Process $p' = p$. Then there must be a finite deciding execution run σ starting in configuration z and in which p is not involved, because otherwise A would not decide if p would crash. The fact that z and z' can be reached from Γ without using some process $q \neq p$ does not mean that q must crash; hence p might crash.
Let $z'' = \sigma(z)$ and $z'' \in \mathcal{C}$. Obviously σ can be applied to $t(z)$ and to $t(z')$. $t(z'') = \sigma(t(z))$ must be 0-valent because $t(z)$ is 0-valent. Moreover $t(t'(z'')) = \sigma(t(t'(z)) = \sigma(t(z'))$ is 1-valent. This contradicts the assumption that σ is a deciding execution run.
Hence in both cases we reach a contradiction. Thus $\mathcal{D}$ must contain a bivalent configuration. ∎

Now we will prove the theorem. Any deciding run on a bivalent initial configuration goes to a univalent configuration. Hence there must be an event (an application of a transition function) that goes from a bivalent to a univalent configuration. We will show that it is always possible that A runs in such a way that it avoids these deciding events, even in a fair execution. We will construct such an execution run. We therefor have to be precise in what order processes perform transition functions.

We keep a first-in-first-out queue QP of processes; initially QP contains all processes in arbitrary order $p_1, \ldots, p_N$. We also maintain a message queue QM in which messages are ordered according to the time they are sent. Initially QM is empty. When a process crashes it is automatically removed from QP.

The execution run is built up of stages $S_1, S_2, S_3, \ldots$. Stage S_i starts in configuration Γ_{i-1} and ends with a configuration Γ_i, which is the starting configuration of the next stage. We will construct stage S_i in such a way that Γ_i is bivalent provided Γ_{i-1} is bivalent. We start with a bivalent configuration Γ_0, which exists according to claim 15.8.

So assume that Γ_{i-1} is bivalent. In stage S_i the process p at the front of QP is removed from QP and added at the rear. If QM contains a message with destination p, p considers the transition function g_m with m the first message in QM with destination p (m is removed from QM). If QM contains no message for p, then p considers the transition function with an empty message. By claim 15.9 we know that there is an applicable sequence σ_i of transition functions ending with transition function g_m such that $\sigma(\Gamma_{i-1})$ is bivalent. In stage S_i this sequence σ_i is executed. This execution of σ_i does not affect the process queue QP, but it may change the message queue QM by inserting and/or removing messages. But we know that $\Gamma_i = \sigma_i(\Gamma_{i-1})$ is bivalent.

In this way the decision of algorithm A is postponed forever. The constructed execution is actually a fair execution. If QM contains a message for a process p this message will eventually be received by p due to the way a process and transition function is chosen at the beginnning of stages.

We conclude that there is fair execution of A in which A does not terminate. Contradiction. ■

In the next lectures we will show that the consensus problem can be solved assuming initially-dead faults, and present a number of problems that can be solved in the crash fault model.

References

[1] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of Distributed Consensus with One Faulty Process, *Journal of ACM* 32 (1985), pp 374-382.

[2] G. Tel, *Introduction to Distributed Algorithms*, Cambridge University Press, 1994.