The CHT Play

An Informal Note on the Necessary Part of the Proof that Ω is the Weakest Failure Detector for Consensus

Felix C. Gärtner[1] Rachid Guerraoui[2] Petr Kouznetsov[2]

[1] Laboratory for Dependable Distributed Systems, Aachen University
[2] Distributed Programming Laboratory, EPFL

Abstract

This note gives a high level and informal account of the necessary part of the proof that Ω is the weakest failure detector to implement consensus with a majority of correct processes. The proof originally appeared in a widely cited but rarely understood paper by Chandra, Hadzilacos and Toueg. We describe it here as a play in five acts, preceded by a prologue and followed by an epilogue.

The Prologue

Consensus and leader election are two fundamental abstractions in distributed computing. Consensus provides processes with the ability to agree on a common value. We consider here a variant of leader election, denoted by Ω, through which the processes eventually agree on a common correct leader. Proving that Ω is the weakest failure detector to implement consensus with a majority of correct processes goes through exhibiting two algorithms:

1. an algorithm that implements consensus with a majority of correct processes using Ω (sufficient part);
2. an algorithm T (called a transformation or a reduction algorithm) that, using any algorithm A that implements consensus with some failure detector $\mathcal{D}$, implements Ω (necessary part).

The necessary part, often called the "CHT proof" (or simply "CHT ") (for Chandra, Hadzilacos and Toueg [1]) pioneers a new reduction style in the theory of distributed computing. It is however quite long and rather involved. That is maybe why it is rarely understood.

We have tried in this note to give a high level description of CHT. Clearly, we often sacrificed rigour for intuition: sometimes on purpose and sometimes not. We give a description of the main components of the proof and we consider a simple case of the proof while abstracting away a trickier in the appendix. Even our very informal description requires however five acts. Moreover, the use of the first of these acts is getting visible only at the very end of the proof. We thus encourage the reader to continue reading even when it seems as though the proof does not make progress anymore. Catching a glimpse of the elegance and cleverness of the CHT proof will be your reward at the end.

Before delving into CHT, we first recall here some of its underlying elements. As we recalled above, CHT is about constructing algorithm T using consensus algorithm A (itself using failure detector $\mathcal{D}$). We say few words here about what is, a priori, needed to be noticed about algorithms A and T.

- About algorithm A (which is given):

  1. *A uses a failure detector $\mathcal{D}$ in a message passing system.* A failure detector is a distributed abstraction that provides processes with information about the status (crashed or not) of other processes in the system. Every process p_i has a module of that failure detector and p_i can query that module at any time. The information obtained from this query is used as an input by p_i to

its algorithm automaton, as we discuss below. Algorithm A is a set of deterministic distributed automata, one per process. At every step of any run of A, exactly one process p_i triggers its algorithm automaton: we say that p_i executes a step. A correct process is one that executes an infinite number of steps. (A process that crashes simply stops triggering its automaton.) The input of the automaton is (1) p_i's state, (2) a message that p_i has received from some other process, as well as (3) the value obtained by p_i from its local failure detector module. The output of the automaton is (4) a new state as well as (5) a message that p_i broadcasts. Communication is supposed to be reliable in the sense that if p_i sends a message, then this message is eventually received by every correct process.

2. *A implements consensus.* The problem consists for the processes to propose values and to decide on the same final value. More precisely, every correct process will decide on a value (*termination* property of consensus); no two correct processes will ever decide differently (*agreement*); and any value decided is a value proposed (*validity*).

- About algorithm T (which is to be constructed):

  1. *T needs to implement* Ω. In other words, T is a leader election algorithm which should ensure that, eventually, all correct processes permanently elect the same correct process. The existence of both (1) algorithm T and (2) an algorithm that implements consensus using Ω is what derives the fact that Ω is the weakest failure detector to implement consensus.
  2. *T is not restricted to using A as a black-box.* In other words, we are not trying here to implement Ω out of a consensus abstraction. At first glance, using consensus as a black-box, we could indeed easily implement a leader election scheme which ensures that the processes do all elect the same leader: the processes would each propose its identity and, using consensus, we would get the same identity as a decision. This is a strictly weaker variant of Ω because there is no guarantee that the leader is correct. On the other hand, if the goal was simply for every correct process to elect a correct process, each would simply output itself. The challenge that T needs to face is to have the processes eventually and permanently *agree* on the very same *correct* leader. Algorithm T achieves this by using A's automata at *every* process, as we overview in the following.

We give here an overview of algorithm T. The basic idea underlying T is to have each process locally *simulate* the overall distributed system in which the process executes several runs of A. The processes then use the outputs provided by these runs to extract the leader process.

Every process simulates, locally, runs of algorithm A by launching threads that mimic the behavior of every other process in the system running algorithm A (Fig. 1). But how can one process p_i simulate the overall system executing several runs of A? Basically, every process p_i feeds algorithm A with a set of proposed values, one for each process of the system, i.e., p_i pretends to be every other process and proposes values to A's runs. In fact, process p_i proposes all possible combinations as we discuss in Act 2. Then all automata composing A are triggered locally by p_i, which locally emulates, for every simulated run of A, the states of all processes as well as the emulated buffer of exchanged messages.

Crucial elements that are needed for the simulation are (1) the values from failure detectors that would be output by $\mathcal{D}$ as well as (2) the order according to which the processes are taking steps. For these elements, which we call the stimuli of algorithm A, process p_i *exchange* information with the other processes, as we will discuss later (Act 1). It is important to notice that the output of $\mathcal{D}$ is not restricted in any way. In fact, this output can be *any value* that encodes some information about failures.

Every process p_i locally uses the series of (consensus) decisions obtained by A in order to *extract* the leader process (Acts 3-5). The idea is to analyze which combination of values proposed to consensus leads to which decision values. In short, the process that is elected leader (i.e., extracted) is the one, the proposed value of which, makes a difference in the pattern of decision values.

Act 1: The Exchange

The processes periodically query their failure detector modules of $\mathcal{D}$ and send the output to all other processes. As a result, every process knows more and more of the other processes' failure detector outputs and

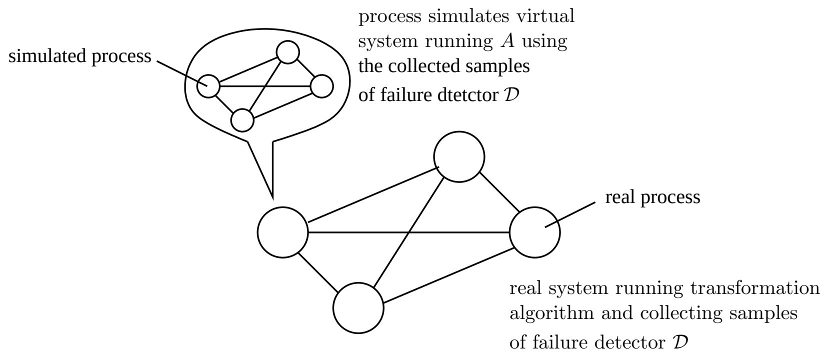

simulated process
process simulates virtual system running A using the collected samples of failure dtetctor $\mathcal{D}$
real process
real system running transformation algorithm and collecting samples of failure detector $\mathcal{D}$

Figure 1: Processes simulate runs of consensus in their own simulation environment taking stimuli from "reality".

temporal relations between them.

All this information is pieced together in a single data structure, a directed acyclic graph (DAG). This DAG has some special properties which follow from its construction as we will discuss later. What is important to see for now is that the DAG can be used to derive simulation stimuli to A: it contains activation schedules and failure detector outputs for the processes to execute steps of A's instances. In fact, every path through this DAG is one such possible stimulus which can be used as an input to A in the simulation environment.

An example is depicted in Figure 2 in which the vertex $[p_1, d_4]$ (meaning that the failure detector at p_1 responded with d_4) is added to the DAG after having received $[p_2, d_2]$ and $[p_3, d_3]$. The DAG is transitively closed. Thus, every suffix of a path through the DAG is again a path through the DAG.

Of course, these schedules will always be finite, so they can be used to simulate only finite runs of A in the simulation environment (this means that A may not terminate for some processes). But this is not too bad. As long as the processes continue to exchange information, the DAG becomes larger and larger and provides more and more simulated terminating runs of A.

When the processes communicate, they send their own version of the DAG to each other. When receiving such a DAG, a process integrates it into its own DAG (it forms the union). In this way, the intersection of all correct processes' DAGs grows without bound too: without knowing exactly how large it is, the processes construct an ever-increasing "common sub-DAG".

$$\begin{tikzcd}
 & {[p_2, d_2]} \arrow[dr] & \\
{[p_1, d_1]} \arrow[ur] \arrow[rr] \arrow[dr] & & {[p_1, d_4]} \\
 & {[p_3, d_3]} \arrow[ur] &
\end{tikzcd}$$

Figure 2: Construction of the DAG.

Act 2: The Simulation

Let's forget the point about the common sub-DAG and the special DAG properties for a moment and just recall that the DAG can be used to extract stimuli for running A in a simulation environment. So let's assume that some process p_i has just constructed a new version G of its personal DAG. There are many paths through G and each path corresponds to an activation schedule for A. Process p_i simply starts a simulation environment *for every path* through G. Note that this might be a Herculean task in reality, but we are interested here in proving the existence of an algorithm, not necessarily an efficient one. Since G is finite, the number of paths through G is also finite and so we are not demanding the impossible.

So let's consider one such path π through G. We will use π to stimulate A in the simulation environment. But A is an algorithm which solves consensus, and so we need to provide input values for A. Which ones should we choose? the answer is simple: don't choose; rather run even more instances of A! So for the path π we run not one simulation environment, but rather $n+1$ simulation environments at the same time. (Remember that n is the number of processes in the system: $p_1, \ldots, p_n$.) The initial values given to each of these instances follow a certain pattern: in the first instance, all processes propose 0; in the second instance, process p_1 proposes 1 and all others propose 0; in the third instance, processes p_1 and p_2 propose 1, and all others propose 0, etc. In the final $(n+1)$-th instance, all processes propose 1. The input vectors to the instances are denoted $I_0, I_1, \ldots, I_n$, where I_i makes processes p_1 up to p_i propose 1 and the rest (processes p_{i+1} up to p_n) propose 0.

To recall: whenever its DAG changes, every process p_i runs a simulation environment for *every path through the DAG* and *for every input vector* $I_0, \ldots, I_n$. Each simulation is run "up to the end", i.e., until it runs out of activation stimuli from π found in the current DAG. When all simulations are finished, then p_i reconsiders incoming messages, constructs a new DAG and starts the simulations all over again from scratch. As mentioned above, this is a lot of work, but it is not impossible.

Act 3: The Tagging

At the end of the simulations (i.e., before constructing a new DAG and re-starting the simulations again), process p_i looks at the results of the runs of A. In some simulation environments, A might not have terminated since the length of the input stimulus is not sufficiently large (the emulated processes have not been activated often enough). But in others, A might have terminated. In fact, since correct processes keep on exchanging messages with each other, there is eventually a path through the DAG in which such processes will regularly appear. So it is just a matter of waiting long enough, and launching new simulations, until there is some simulation stimulus which leads A to termination. Since A is a consensus algorithm, it gives us the decision value for the particular run of the simulation (either 0 or 1).

Process p_i now looks at all simulations which started from the same initial input vector. For example, it considers all simulations which started from I_0 and looks at the decision values of those simulations which have terminated. Of course, the decision value will be 0 for input vector I_0 since all processes proposed 0 and the decided value must be a proposed value (this is mandated by the validity property of consensus). Similar for I_n, only 1 can be decided. But for other input vectors like I_1 and I_{n-1} both 0 and 1 can be the result.

For every input vector $I_0, \ldots, I_n$, p_i determines the set of all decided values and "tags" the input vector with them. An input vector which is only tagged with 0 is called 0-*valent*. An input vector tagged only with 1 is called 1-*valent*. Otherwise, the input vector is called *bivalent*. (We disregard the case where none of the simulated runs decides, i.e., where there are no tags at all attached to an input vector; this happens only finitely many times.)

Now look at the list of input vectors $I_0, \ldots, I_n$: certainly, and as we pointed out, I_0 is 0-valent, but what about the others? Let's assume that I_1 and I_2 are 0-valent too, but not all input vectors can be 0-valent because I_n is 1-valent. This means that there exists some index i in the sequence of input vectors where I_{i-1} is 0-valent and I_i is either 1-valent or bivalent. Index i is called a *critical index.* Interestingly, the critical index can be used to make an educated guess about some correct process in the system.

Act 4: The Stabilization

An important fact to observe now is that there is a time after which the critical index does not change anymore and this index is the same at all processes. Since I_0 will always remain 0-valent, the critical index cannot decrease below 1; it will eventually stabilize to a value which is at least 1. Furthermore, all correct processes doing the simulation will stabilize to the same critical index. This is because the DAG is the only input to the simulations, and so the DAG alone inevitably determines the critical index. But as noted earlier, correct processes exchange and update their views of the DAG periodically, which implies a common ever-growing sub-DAG of all correct processes. Eventually, the common sub-DAG will be sufficiently large to allow A to terminate on the stimuli extracted from it. Hence, all correct processes will “stabilize” on the tags they attach to the input vectors: for example, if some process tags I_3 with 0, then eventually every other correct process will do the same (this tag is sticky, i.e., it will not vanish). As a result, once an input vector has become bivalent, it cannot switch back to 1-valent or 0-valent anymore. So the critical index cannot shift to the right, it can only shift to the left (toward I_0). Moreover, the critical index cannot shift left forever, it must stop at some point. And this point is eventually computed by every correct process. So while there is no way for the processes to “know” that their critical index has stabilized (unless it is 1 of course), stabilization will eventually happen.

Act 5: The Extraction

The final step is now to see what a critical index has to do with the identity of a correct process. This is maybe the point which is most difficult to understand, so what we start by showing that a critical index has *some* relation to a correct process. If you can see that in some cases the critical index gives useful hints about a correct process, it might be easier for you to believe that it may do the same in the other (more complicated) cases.

The *simple* case we consider is, besides the assumption that the stabilization to a critical index i has taken place, that I_{i-1} is 0-valent, and that I_i is 1-valent. The claim now is that is that p_i is a correct process.

Assume by contradiction that p_i is faulty. In this case, p_i stops participating in the collective building of the DAG. So at some point, the DAG is being extended without adding vertexes with respect to p_i anymore. Now the special property of transitive closure of the DAG is important: consider any path π through the DAG in which p_i has stopped to participate. Since at some point p_i has stopped contributing to the DAG, there is a suffix π' of π in which there is no vertex regarding p_i. Because of transitive closure, π' is also a path through the DAG.

Because π' is also a path through the DAG, there must be a simulated run of A performed with π' as input stimulus for the input vector I_i. Since we assume that the critical index has stabilized, I_i remains 1-valent, and so running A on any extension of π' results in deciding value 1. To summarize, A has a run c_1 in which processes start from I_i, p_i takes no steps at all and the decision value is 1.

Now look at the same stimulus applied to A starting from I_{i-1}. Since I_{i-1} is 0-valent, the outcome of A must be 0. To summarize, A has a run c_2 in which processes start from I_{i-1}, p_i takes no step at all and the decision value is 0. But from the point of view of all the processes apart from p_i, the executions c_1 and c_2 are indistinguishable: the only difference is p_i’s initial value that changes in both cases, but no process knows that value as p_i takes no step. Algorithm A is deterministic, so it should yield the same decision value in both cases, which it doesn’t. This is the contradiction. So p_i must be a correct process.

The above line of reasoning shows that there is some non-trivial information about correct processes in the critical index. We have argued only for the case when I_i is 1-valent. The case when I_i is bivalent is a little trickier and involves reasoning about devices called *hooks* and *forks* (this is probably the place in the proof where readers might surrender). We will do some hand waving here and just ask the reader to believe that this part of the proof works. For those who still want to understand this final part of the proof, The appendix A provides a multi-page, high-level overview over this part of the proof.

The Epilogue

So let’s put all CHT pieces together: we are given a distributed system, a failure detector $\mathcal{D}$, and an algorithm A which uses $\mathcal{D}$ to solve consensus. We design an algorithm T where the processes:

- periodically query $\mathcal{D}$;
- exchange the results in the form of DAGs;
- use the DAGs to locally simulate a large number of runs of A with all possible input vectors of proposal consensus values; (Every process runs these simulations, not necessarily the same set, but eventually every simulation which is done by one correct process is also performed by all other correct processes);

- use the consensus decisions obtained from these runs to tag the input vectors;
- and finally use these tags to identify a critical index and elect the corresponding leader process.

To conclude, let's point out some extensions of CHT.

- CHT considers a distributed system model where the processes communicate by exchanging messages through reliable channels (e.g., in Act 1). The result has been extended in [8] to a distributed system model where the processes communicate through registers. The result was also revisited in models with more sophisticated shared objects than registers, first in [9], and later in [5]. With these objects, the weakest failure detector to implement consensus is strictly weaker than Ω.
- The variant of consensus considered in CHT is sometimes called *strong* consensus: no process can decide 0 (resp. 1) if they all propose 1 (resp. 0) (this is fundamental in Act 3). This contrasts a weaker variant of consensus where we would require only that algorithm A has a run where 1 is decided and a run where 0 is decided. Considering such a weak variant impacts the result as pointed out in [1] and [6].
- Finally, it is important to recall that CHT is *only* the *necessary* part of the proof that Ω is the weakest failure detector for consensus. The *sufficient* part goes through exhibiting an algorithm that implements consensus using Ω. Variants of such algorithms have been proposed in the literature [2, 4, 7, 10]. All these algorithms assume however a majority of correct processes, which is indeed necessary if Ω is the only failure detector available. Determining the weakest failure detector for consensus without the correct majority assumption has more recently been studied in [3].

Acknowledgements

Although now intended for teaching purposes, the writing of this note was indirectly motivated by (independent) discussions with Hagit Attiya, Eli Gafni, Nancy Lynch, Sergio Rajsbaum and Michel Raynal.

References

[1] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[2] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[3] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical Report IC/2003/77, EPFL, December 2003. Availabe at http://icwww.epfl.ch/publications/.

[4] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288 – 323, 1988.

[5] Rachid Guerraoui and Petr Kouznetsov. On failure detectors and type boosters. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, October 2003.

[6] Rachid Guerraoui and Petr Kouznetsov. The gap in circumventing the consensus impossibility. Technical report, EPFL, ID:IC/2004/28, January 2004. Available at http://icwww.epfl.ch/publications/.

[7] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[8] Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG'94)*, volume 857 of *LNCS*, pages 280–295. Springer Verlag, 1994.

[9] Gil Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 100–109, August 1995.

[10] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, 1988.

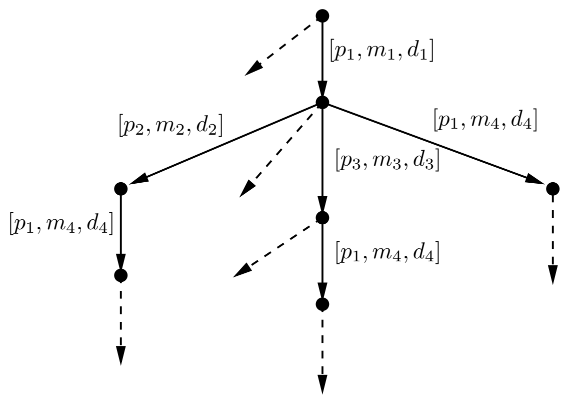
A Handling a Bivalent Critical Index

In this appendix we fill a gap in the proof of the main part of this document. The basic question we answer here is the following: how can the identity of a correct process be determined if, for some stabilized critical index i, the input vector I_{i-1} is 0-valent and the vector I_i is bivalent? In contrast to the case studied before (where I_i was 1-valent), the correct process is not necessarily p_i. The proof is slightly more complicated than before. In fact, the proof for this part needs almost as many pages (and even more figures) to explain than the entire proof which was presented up to now. In some sense, we are advising the readers to read this appendix only if absolutely necessary.

A.1 Simulation Tree

Remember that a process runs a simulation (using the given consensus algorithm A) for all input vectors and for all paths through the current version of its DAG. Eventually, (provided the process does not crash) the result of this is a huge set of simulated runs of A. Let's only consider those runs which start with the (bivalent) input vector I_i (where i is the critical index). We can combine all of them into a *simulation tree.* The root of the tree is the starting configuration. Every edge in the tree corresponds to a step taken by some process and every vertex corresponds to a configuration resulting from the steps leading to it. Runs of consensus which "share a prefix" (i.e., start in the same way) have the same prefix in the tree. Basically, the tree represents all choices of next steps during the simulation, much like a game tree in chess for example.

A step consists of three items and is represented as $[p, m, d]$. The value p denotes which process executed the step, m identifies which message that process consumed (if any) and d represents the output of the failure detector during this step. For example, the simulation tree using the input of the DAG in Fig. 2 could look as follows:



Note that there are many more choices that have not been represented in this tree: for example, the DAG will consist of increasingly many possible samples to choose from. Also there is a choice of the message m which a process should consume in the simulation. (In fact, a simulated process can consume an "empty" message in a step: this models arbitrary message delays.) So one "step" in the DAG offers multiple choices (and hence path continuations) in the simulation tree.

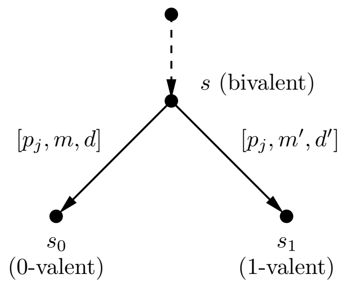
The individual states of the tree also carry a valency: if in that state the consensus algorithm has terminated with a decision of 0 or 1, then that state is 0-valent or 1-valent, respectively. For states, in which the consensus algorithm has not yet terminated, we have the following rule: if a state s has a followup state (a descendant in the tree) which is 1-valent, then s is also 1-valent. Similarly for 0-valent descendant states. However, if s has both a 1-valent and a 0-valent descendant state, then s is *bivalent.* Recall that the starting state of the simulation tree (the root) must be bivalent.

A.2 Determining a Correct Process: Hooks and Forks

As mentioned before, the correct process is in general not p_i anymore (where i is the critical index) but some other process. The idea is to use here certain patterns in the simulation tree to identify a correct process. These patterns are called *forks* and *hooks.* In the following, we first look at how hooks and forks allow the processes to determine a correct process. Then we argue that every simulation tree for I_i contains at least one hook or one fork which does not vanish. Let's consider forks first (this is the easier case).

A.2.1 Forks

The following picture identifies a *fork*. A fork consists of a bivalent state s from which two *different* steps by the *same* process p_j are possible which, on the one hand, lead into a 0-valent state s_0 and, on the other hand, to a 1-valent state s_1:

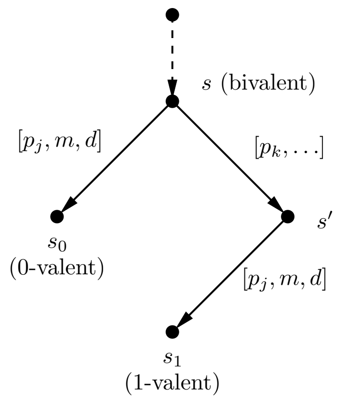


The point to note now is that if we can find a fork in the simulation tree, then p_j must be a correct process. To see this, assume that p_j is faulty. Like in the main part of this text, we can now exploit a property of the DAG (its transitive closure). Starting from state s_0, there is a continuation π_0 in the simulation tree in which p_j never takes a step. Since s_0 is 0-valent, the processes decide 0 in π_0. Similarly, starting from state s_1 there is a continuation π_1 in which p_j also does not take any step, and in which the other processes take exactly the same steps as in π_0. Since s_1 is 1-valent, the processes decide 1.

There is a contradiction leaking here: the only difference between π_0 and π_1 is the state of p_j. But p_j is crashed and so processes should come to the same decision in π_0 and π_1, not different ones. So p_j must be correct.

A.2.2 Hooks

Let's look at hooks now. A *hook* in the simulation tree has the following structure:



It consists of a bivalent state s, a state s' which is reached by executing a step of some process p_k, and two states s_0 and s_1 reached by executing *the same* step of process p_j. Additionally, state s_0 must be 0-valent and s_1 must be 1-valent (or vice versa; the order does not matter here).

If we can find a hook in the simulation tree, then we're done: in this case, p_k is a correct process. The reason for this can be derived by similar arguments as in the case of a fork: if p_k were faulty, then we could construct extensions of s_0 and s_1 in which p_k takes no step, but where all other processes take the same steps. Hence, they should reach the same decision. But the valencies of s_0 and s_1 are different — a contradiction! So p_k must be correct.

A.3 Existence of Hooks and Forks

If we have found a hook or a fork in the simulation tree, then we are lucky: we can extract the common correct leader. If there are many hooks and forks in the simulation tree, then we need to prevent different processes from looking at different hooks and outputting different values. This is done by defining the notion of a "first" hook or fork. Basically this is possible since the vertexes of the simulation are countable: the processes can then select the hook or fork with the "smallest" root state s and extract from it the leader.

But what if all newly computed hooks or forks disappear from the simulation tree and never re-appear? In fact, this can't happen. Eventually, and as we argue below, there will be a hook or fork in the simulation tree which does not go away. Take an *infinite* bivalent simulation tree. We consider a hypothetical algorithm which goes through the simulation tree. The algorithm terminates only when a hook or a fork has been found. The algorithm proceeds as follows:

```
$s := \langle$root of simulation tree$\rangle$
**while** *true* **do**
    $j := \langle$choose the next correct process in a round robin fashion$\rangle$
    $m := \langle$choose the oldest message for process $p_j\rangle$
    **if** $\langle s$ has descendant $s'$ (possibly $s = s'$) such that, for some $d$,
            $s'$ extended with $[p_j, m, d]$ is a bivalent vertex of the tree $\rangle$
    **then** $s := s'$ extended with $[p_j, m, d]$
    **else exit**
**end**
```

Basically the algorithm locates a *fair path* through the simulation tree, i.e., a path in which all correct processes get scheduled infinitely often and every message sent to a correct process is eventually consumed. Additionally, this fair path goes through bivalent states only.

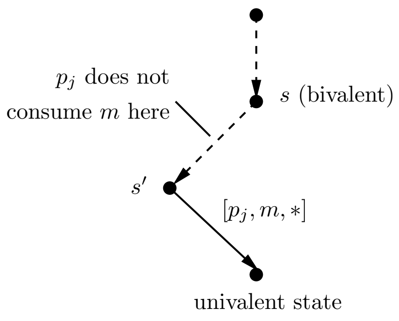
There are two questions to ask now: (1) Does this algorithm always terminate? (2) If it terminates, where is the hook or the fork?

A.3.1 Terminating the Infinite Simulation Tree

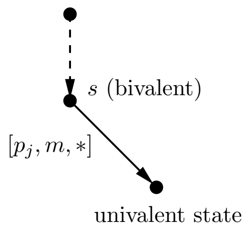
Suppose, by contradiction, that the algorithm does not terminate. That is, we can locate an *infinite* path in the simulation tree that goes through bivalent states only. Moreover, the path corresponds to a fair run of the consensus algorithm A: every correct process takes an infinite number of steps in the run and eventually receives every message sent to it. Note that a state in which some process decides a value $v \in \{0, 1\}$ cannot be bivalent, otherwise the agreement property of consensus would be violated. In this case, we end up with a fair run of algorithm A in which no process ever decides: a contradiction with the termination property of consensus.

A.3.2 Identifying a Fork or a Hook

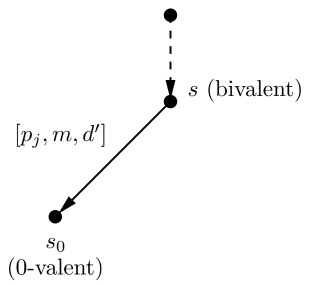
So we have seen that at some point the algorithm terminates. What condition holds at that point? To find out, we simply have to negate the condition of the **if** statement. The negation reads as follows: state s has no descendant in the tree in which p_j takes a step consuming message m and where the resulting state is bivalent. In other words, any step of p_j consuming message m brings any descendant of s (including s itself) to either a 1-valent or a 0-valent state. Let's look at some extension of s in which p_j takes a step at some descendant s'. We can depict this as follows:



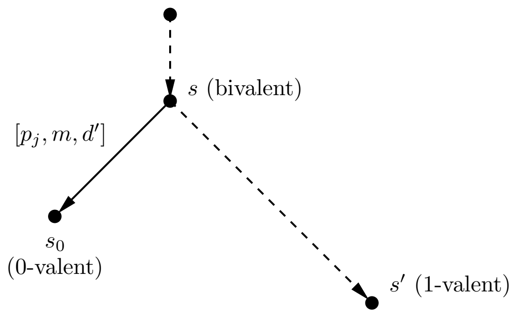
Since the descendant can be state s itself, the following is also a legal tree:



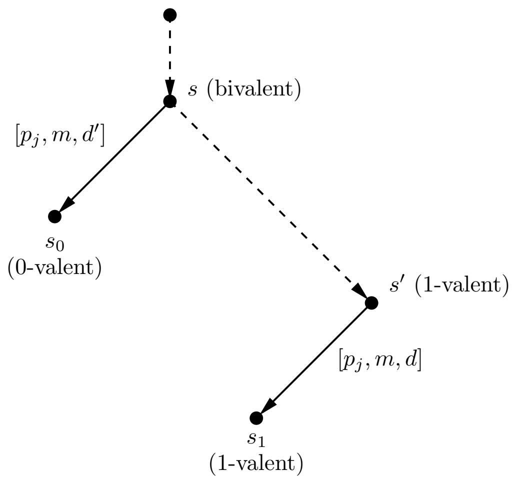
Without loss of generality, we assume that some step $[p_j, m, d']$ brings s to a 0-valent state. That is, our simulation tree contains the following subtree:



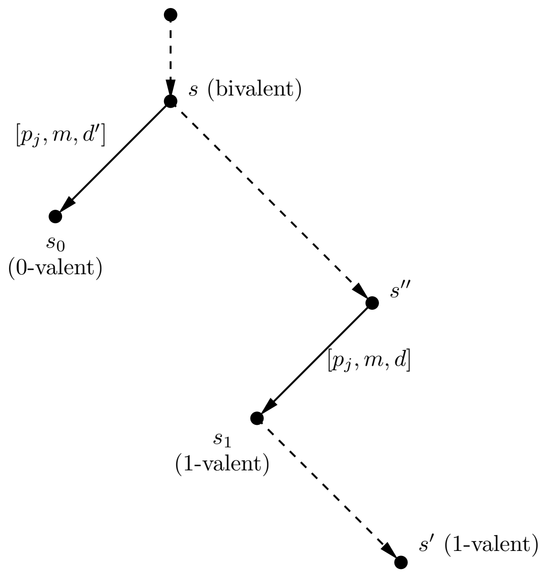
Since s is a bivalent state, it must have some descendant s' which is 1-valent. This can be depicted as follows:



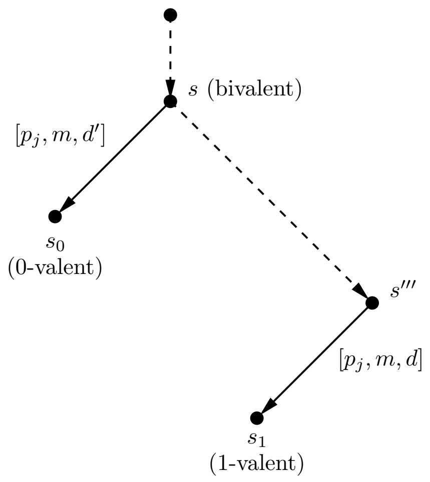
Let's assume that p_j takes no step between s and s' in which it consumes message m. This means that m is still unconsumed in state s' and some step of the form $[p_j, m, d]$ is applicable there. Since s' is 1-valent, the step $[p_j, m, d]$ applied to s' results in a 1-valent state. So we have the following situation:



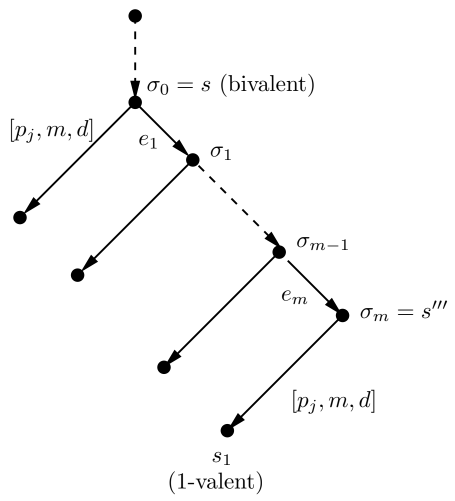
We can also come up with a situation like this even if p_j takes a step between s and s' in which it consumes m. So assume that p_j takes a step consuming m in some intermediate state, say, s'', on the path from s to s'. The resulting state cannot be 0-valent. Why? Because we know that s' is 1-valent, and so the resulting state must be at least bivalent! But it cannot be bivalent because the termination condition of the algorithm which identified s disallowed bivalent followup states of actions of p_j. So overall the situation then looks like this:



In both cases, our simulation tree contains the following subtree where s''' is either s' or s'':



We claim that within this structure there must be a fork or a hook somewhere. The argument is quite simple: the path between s and s''' consists of a finite sequence of steps $e_1, e_2, \ldots, e_m$. Since p_j does consume m between s and s''', the very same step $[p_j, m, d]$ can be applied in the first state following s on the way to s''', i.e., after executing step e_1. Similarly, it can be applied after executing step e_1 and e_2. In fact, it can be executed in *every* state on the way from s to s'''. We denote the corresponding intermediate states by $\sigma_0 = s, \sigma_1, \ldots, \sigma_{m-1}, \sigma_m = s'''$.



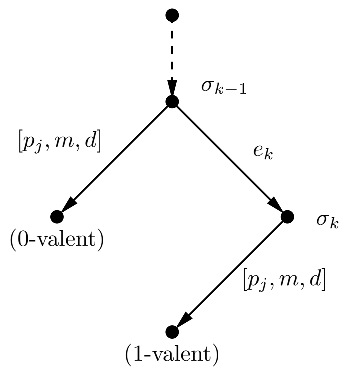
The important point here is to see that, for any $k = 0, \ldots, m$, the step $[p_j, m, d]$ applied to σ_k results in a 0-valent or a 1-valent state. This is because the algorithm which identified s terminated and the termination condition stated that there is *no descendant of s* which is bivalent after applying that ominous step of p_j.

Let $k \in \{0, \ldots, m\}$ be the lowest index such that $[p_j, m, d]$ brings σ_k to a 1-valent state. We know that such an index exists, since s_1 is 1-valent and all such resulting state are either 0-valent or 1-valent.

Now we have the following two cases to consider: (1) $k = 0$, and (2) $k > 0$.

Let's assume that $k = 0$, i.e., $[p_j, m, d]$ applied to s brings it to a 1-valent state. But we know that there is a step $[p_j, m, d']$ that brings s to a 0-valent state. That is a fork is located! As a result, we identified p_j as a correct process.

If $k > 0$, we have the following situation:



That's a hook! If e_k was executed by process p_k, then we have identified p_k as a correct process following the argument about hooks above.

Thus, a bivalent infinite simulation tree has at least one *finite* subtree that allows us to compute a single correct process.