CSC2221: Introduction to Distributed Computing **Fall 2017**

Lecture 2: Fault-Tolerant Consensus

Lecturer: Faith Ellen *Scribe: Lily Li*

2.1 From Last Time...

The weighted minimum spanning tree has been solved.

2.2 Synchronous Message Passing Systems with Crash Failures

2.2.1 Formal Model

Definition 2.1 ***Crash failure:*** *if p_i crashes in round r all messages p_i sends previous rounds are delivered and an arbitrary subset of the messages which are supposed to be sent on round r are delivered in round $r+1$. This is represented by a* $\mathsf{crash}(i)$ *event. Afterwards, no* $\mathsf{comp}(i)$ *or* $\mathsf{crash}(i)$ *events. An* ***active process*** *is one which has not crashed. A* ***clean crash model*** *is one where no messages are sent during the round in which the process crashed. This is a stronger model than the previous one.*

Definition 2.2 *Let f be the maximum number of processors in the system that can fail. Then we call the system* ***f-resilient***

2.3 Consensus

2.3.1 The Consensus Problem

Suppose each processor p_i has special state components: the input x_i, the output y_i (aka **decision**). Initially x_i holds a value from some well-ordered set of possible inputs and y_i is undefined. Any assignment to y_i is irreversible. For the consensus problem we want:

Termination: In every admissible execution y_i is eventually assigned a value for any non-faulty processor p_i.

Agreement: if y_i and y_j are assigned, then $y_i = y_j$, for all non-faulty processors p_i and p_j.

Validity: The output is an input for some processor. More precisely, every processor gets an input of v then the output of all processors should be v.

At termination we want all processes which have not crashed to output y_i. This is the liveness condition. Agreement and validity are safety conditions. All three are needed for the consensus to be interesting. If you only have two out of the three, there exists a trivial algorithm.

Binary consensus has $x_0, ..., x_{n-1} \in \{0, 1\}$. Consensus is quite a benchmark problem in distributed computing.

2.3.2 Algorithm

This is going to be a f-resilient algorithm in a complete synchronous network. (How would we get a 0-resilient algorithm?)

Algorithm 1 f-resilient synchronous consensus in a complete network: code for processor p_i.

```
1: u_i ← x_i
2: ∀j ≠ i send u_i to p_j
3: for rounds 2, ..., f + 1 do
4:     v_i ← minimum value in p_i's in buffer's
5:     if v_i < u_i then
6:         u_i ← v_i
7:         for all j ≠ i do
8:             send u_i to p_j
9:         end for
10:    end if
11: end for
12: u_i ← minimum u_i and all values in p_i's in buffer
13: output u_i
```

Lemma 2.3 *If no process crashes during round r then all active processes have the same value of u_i at the end of round $r + 1$.*

Proof: Proof by contradiction. Suppose no crashes occur in round r and there exists two processor p_i and p_j such the value u_i of p_i is different then the value of u_j of p_j at the end of round $r + 1$. Further assume without loss of generality that $u_i < u_j$. If p_i had value u_i at the end of round r then p_i would have put u_i on the in buffers of all other processor, in particular p_j. At the start of round $r + 1$ processor p_j would update its value to u_i since $u_i < u_j$. Since this is not the case then p_i got value u_i from some other processor p_k during round $r + 1$. However p_k would have put u_i on the in buffer of p_j so p_j still should have value u_i. ■

Lemma 2.4 *If all active processes have the same value for u_i at the end of round $r$then u_i doesn't change in any later round.*

A proof is not really needed for this one. By analyzing the algorithm it is not hard to see that a processor only sends messages when it gets a smaller value from another processor. If all the processors have the same value, no processor will send (thus no processor will receive) any new messages.

Lemma 2.5 *(Agreement) All active processes p_i have the same value for u_i at the end of round $f + 2$ (and hence outputting the same value).*

Proof: By PHP there exists a round among the first $f + 1$ where no crash occurs. Then by the first lemma all processes will agree in the next round and (by the second lemma) all process will continue to agree in all subsequent rounds. ■

Lemma 2.6 *(Validity) At the end of every round, $\{u_0, ..., u_{n-1}\} \subset \{x_0, ..., x_{n-1}\}$*

Lemma 2.7 *(Termination) The algorithm terminates.*

The proof of validity is by induction. And you just have to look at the code for the Termination condition. Please consider an execution where $f+2$ are required. How about something like this: we have processors $p_1, ..., p_{f+2}$. The input to p_1 is 0 while the input to all other processors is 1. On the first round p_1 puts 0 onto p_2's input buffer. On round two p_1 sends the message and crashes. Generally on round $i+1$ processor i crashes but manages to send 0 to processor p_{i+1}. On the $f+1$ step. p_f sends 0 to p_{f+1}. On round $f+2$, processor p_{f+1} sets $u_{f+1} = 0$ and prepares to send 0 to p_{f+2}. Thus at the end of round $f+1$ processor p_{f+1} has value 0 and p_{f+2} has value 1. What happens when the crashes are clean? Only two rounds are needed.

2.3.3 Lower Bound on the Number of Rounds

Theorem 2.8 *Any f-resilient consensus algorithm for the synchronous model with $n \geq f+2$ processors requires $\geq f+2$ rounds (in the worst case).*

Proof: By lemma 2 and 3 (below) there exists a bivalent configuration in which $f-1$ processes have crashed and at most one process crashed per round. We are currently at the beginning of round $r \geq f$. The set of all messages that each active process receives in round r is the same for every execution starting at C (everyone is unsure of if they should decide 0 or 1) ■

Configuration C and C' are **indistinguishable** to process p_i if the local state of p_i is the same in both C and C'. Recall the **history** of an execution (its sequence of comp, del and crash) events. **Local history** of p_i consists of the events in the history that affect p_i, namely $\mathsf{comp}(i), \mathsf{del}(i), \mathsf{crash}(i)$. C (with execution α) and C' (with execution α') are indistinguishable to process p_i if $C \sim_{p_i} C'$ if p_i has the same local history in α and α', $\alpha \sim_{p_i} \alpha'$. This notation can be extended to a set of processes. From the book:

Definition 2.9 *Let α be an execution and p_i a processor. $\alpha|_{p_i}$ is the subsequence of* comp *and* del *events that occur in α at p_i together with the the state of p_i in the initial configuration of α.*

*Let α_1 and α_2 be two executions and let p_i be a processor which is non-faulty in both α_1 and α_2. Then α_1 is **similar** to α_2 with respect to p_i, denoted $\alpha_1 \overset{p_i}{\sim} \alpha_2$ if $\alpha_1|p_i = \alpha_2|p_i$.*

Note we can make some assumptions to simplify our proofs. First we can assume that each processor sends a message to every other processor during each round of execution. Further an execution is **failure sparse** if there is at most one crash per round. In the forgoing we assume only admissible, failure spare executions.

Lemma 2.10 *Let P be a set of processes and σ the history of an execution starting from C and all events in σ affect events in P. Suppose that for all $p_i \in P$ and all neighbors p_j of p_i, $\mathsf{inbuf}_i[p_ip_j]$ and $\mathsf{outbuf}_j[p_ip_j]$ are the same in C and C'. If $C \sim_P C'$, then σ can occur starting from C' and if σ is finite then $C\sigma \sim_p \sigma$.*

Lemma 2.11 *Any binary consensus algorithm has an initial configuration from which there are two executions which decide different values in which one execution crashes. (Equivalent: Every binary consensus algorithm has a bivalent configuration.)*

Proof: This is almost like one of those graph theory proofs. What you want is a chain of configurations in which adjacent configurations only differ by one processes. If you know the output of the first configuration and the last (by using validity) ■

Definition 2.12 *The **valence** of configuration C is the set of all values that are decided upon by a non-faulty processor in some configuration that is reachable from C in an (admissible failure sparse) execution that includes C.*

2.4 Valency Arguments

Let S be the set of executions. C is a **univalent** configuration (with respect to S) if all terminating executions in S starting from C output the same result v. In particular, all final configurations are univalent. Contrast this with **multivalent** configurations (with respect to S): where there are at least two terminating executions in S starting from C that output different results. Since we are talking about binary consensus, we care about bivalent configurations. Note: when we are looking a lower bounds it is ok to restrict your set S say to only executions which have at most one process crash per round. These are **failure sparse** executions.

Lemma 2.13 *Consensus algorithm A for n processors and f crash failures with $n \geq f + 2$ has a bivalent initial configuration.*

Proof: Suppose for a contradiction that all initial configurations are univalent. Then, since the input of all zeros is 0-valent and an input of all ones is 1-valent there must exist some two initial configurations I_0 and I_1 which only differ on their input to some processor p_i and such that I_0 is 0-valent and I_1 is 1-valent. Consider execution α with initial configuration I_0 where p_i crashes initially and no other processor crashes. Since I_0 is 0-valent, all processors except p_i output zero. However, let execution α' be the same execution as α but with initial configuration I_1. Since α and α' are indistinguishable to all processors except p_i, all processors except p_i must output 0. This contradicts the 1-valence of configuration I_1. ■

Lemma 2.14 *From every bivalent configuration C in which $< f-1$ processes have crashed, there is a round in which at most one process crashes and which results in a bivalent configuration.*

Proof: Let C be a configuration between rounds. Then $\mathsf{inbuf}_j[p_i p_j] = \emptyset$ for all active p_j. Suppose $< f - 1$ have crashed in C. Since $n \geq f + 2$, there are at least three active processes. Suppose C^B be a bivalent configuration. This proof is by contradiction and is going to make use of a couple of lemmas basically you want to construct some supposedly indistinguishable executions which output different values (thus resulting in a contradiction).

(We are going to give the proof in the book since that is the only one I have access to now. There might be some indexing errors, but just ignore these for the moment.) The formal proof is by induction, but since we have the above lemma we only need to concern ourselves with the inductive step.

Suppose fewer than $f - 1$ have crashed and suppose for a contradiction that in all subsequent rounds the configurations are univalent. Our initial $k - 1$ round bivalent execution is going to be α_{k-1}. Suppose that all one-round extensions of α_{k-1} is univalent. WLOG assume that the failure free extension of α_{k-1} is α_k. It is univalent, and in particular we can assume that it is 1-valent. Since α_{k-1} is bivalent there exists a one round extension with exactly one crash, call this β_k such that β_k is a 0-valent execution.

Suppose the crashed processor was p_i. On round k in execution β_k p_i was tried to send messages to processors $q_1, ..., q_m$ but failed. Alternatively observe that α_k is equivalent to the case when all message got to their intended recipients. There are intermediate executions where a subset of the messages intended for $q_1, ..., q_m$ got through. This is sort of similar to the phase transition we saw in the previous lemma. Thus there must exist two executions $\beta_k^{(j)}$ and $\beta_k^{(j+1)}$ (meaning that messages to $q_1, ..., q_j$ and $q_1, ..., q_{j+1}$ got through respectively) such that the former execution is 0-valent and the latter is 1-valent. We are going to extend both $\beta_k^{(j)}$ and $\beta_k^{(j+1)}$ one more round (there must be at least one more round because we have not exhausted the f crashes). Let these extensions be $\gamma_k^{(j)}$ and $\gamma_k^{(j+1)}$ where q_{j+1} crashes and do not send any messages. Then, with respect to all other processors, $\gamma_k^{(j)}$ and $\gamma_k^{(j+1)}$ are indistinguishable. Wait that's a problem because $\gamma_k^{(j)}$ followed from a 0-valent execution while $\gamma_k^{(j+1)}$ followed from a 1-valent execution but they are now indistinguishable. This is our contradiction. ■

There is one more lemma from the book before we can prove the theorem:

Lemma 2.15 *If α_{f-1} is an $(f-1)$-round execution of A that ends in a bivalent configuration then there exists a one-round extension of α_{f-1} in which some non-faulty processor has not decided.*

Proof: The last round is special since you can't just make another processor crash after this round. Thus you must do something clever. We start as before: we are at a bivalent execution α_{f-1} which is a $(f-1)$-round execution. If there exists an extension of α_{f-1} which is bivalent take that and we are done.

Suppose instead that all one round extensions are univalent with value v. Let α_{f-1} be such an extension with no crashes on the f^{th} round. Since α_{f-1} is bivalent, there must be some other execution whose extensions are bivalent or univalent with value $1-v$. Again, in the former case we are done. Suppose the latter case and let β_f be this extension. Note some processor p_i must have crashed during the execution and failed to send a message to a non-faulty processor p_j. We need to consider one more one-round extension of α_{f-1} before we can see the contradiction. Let γ_f be the same as β_f except that γ_f was successful in sending a message to a non-faulty processor p_k (note: since $n \geq f+2$ we can find at least two non-faulty processors, and further note: γ_f must also fail to send a message to p_j so it is possible that $\gamma_f = \beta_f$). Let us look at things from p_j and p_k's perspective. Since α_f and γ_f are indistinguishable to p_k (in both cases p_k got the message) p_k decides v. Similarly, since β_f and γ_f are indistinguishable to p_j (in both cases p_j did not get the message) p_j decides $1-v$. Hold on, this is the consensus problem so everyone needs to agree! We can't have p_j and p_k disagree especially since they are both non-faulty. ■

It is not too hard to see how the set of three lemmas from the book prove the original lower bound. It is basically an induction prove. We get the bivalent initial configuration, work through $f-1$ rounds maintaining a bivalent configuration and in one more round we still can't decide so we need at least $f+1$ rounds.

2.5 Homework

Prove that there is an f-resilient consensus algorithm for synchronous message passing model on a network which is $f+1$ connected.