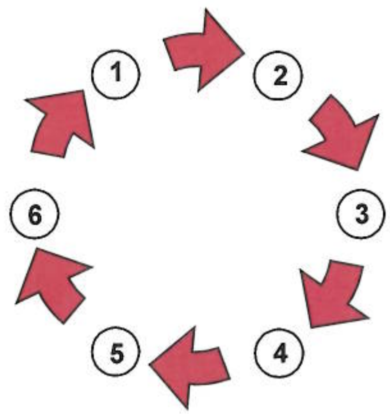
Specification for Leader Election in a Ring

Given a Ring R of Processes with Unique Identifiers (uid's)



Let $n(i) = dst(out(i))$, the next location

Let $p(i) = n^{-1}(i)$, the predecessor location

Let $d(i,j) = \mu k \geq 1.\ n^k(i) = j$, the distance from i to j

Note $i \neq p(j) \Rightarrow d(i,p(j)) = d(i,j)-1$.

Specification, continued

Leader (R,es) == ∃ ldr: R. (∃e@ldr. kind(e)=leader) &
(∀i:R. ∀e@i. kind(e)=leader ⇒ i=ldr)

Theorem ∀ R:List(Loc). Ring(R)
∃ D:Dsys(R). Feasible(D) &
∀es: ES. Consistent(D,es). Leader(R,es)

Decomposing the Leader Election Task

Let LE(R,es) == $\forall$i:R.

1. $\exists$e. kind(e)=rcv(out(i), <vote,uid(i)>)

2. $\forall$e'. kind(e)=rcv (in(i), <vote,u>) $\Rightarrow$
   (u>uid(i) $\Rightarrow$ $\exists$e'.kind(e')=rcv (out(i),<vote,u>))

3. $\forall$e'. [(kind(e')=rcv(out(i), <vote,uid(i)>)) $\vee$
   $\exists$e. (kind(e)=rcv(in(i), <vote,u>)& (e < e' & u> uid(i)))]

4. $\forall$e@i. kind(e)=rcv(in(i),uid(i)). $\exists$e'@i. kind(e')=leader

5. $\forall$e@i. kind(e)=leader. $\exists$e@i. kind(e)=rcv (in(i), <vote,uid(i)>)

Realizing Leader Election

Theorem $\forall \mathtt{R:List(Loc)}\,.\,\mathtt{Ring(R)}$

$\exists \mathtt{D:Dsys(R)}\,.\,\mathtt{Feasible(D)}\,.$

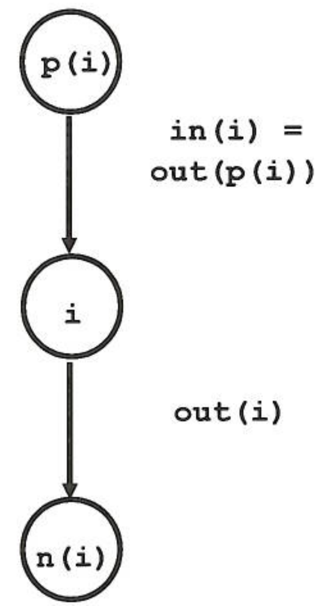
$\forall \mathtt{es:Consistent(D, es)}\,.\,\left(\mathtt{LE(R, es)} \Rightarrow \mathtt{Leader(R, es)}\right)$

Proof: Let $\mathtt{m} = \mathtt{max}\,\{\mathtt{uid(i)} \mid \mathtt{i} \in \mathtt{R}\}$, then $\mathtt{ldr} = \mathtt{uid}^{-1}(\mathtt{m})$.

We prove that $\mathtt{ldr} = \mathtt{uid}^{-1}(\mathtt{m})$ using three simple lemmas.

Intuitive argument that a leader is elected

1. Every i will get a vote from predecessor for the predecessor.
2. When a process i gets a vote u from its predecessor with $u > \mathrm{uid}(i)$ it sends it on.
3. Every rcv is either vote of predecessor $\mathrm{rcv}_{\mathrm{in(i)}}$ for itself or a vote larger than process id before.
4. If a process gets a vote for itself, it declares itself ldr.
5. If a processor declares ldr it got a vote for itself.



Lemmas

Lemma 1. $\forall \mathtt{i} : \mathtt{R}.\ \exists \mathtt{e} \mathbin{@} \mathtt{i}.\ \mathtt{kind(e)} = \mathtt{rcv}\,(\mathtt{in(i)}, <\mathtt{vote}, \mathtt{ldr}>)$

By induction on distance of `i` to `ldr`.

Lemma 2. $\forall \mathtt{i}, \mathtt{j} : \mathtt{R}.\ \forall \mathtt{e} \mathbin{@} \mathtt{i}.\ \mathtt{kind(e)} = \mathtt{rcv}\,(\mathtt{in(i)}, <\mathtt{vote}, \mathtt{j}>)\,.$

$$(\mathtt{j} = \mathtt{ldr} \vee \mathtt{d(ldr, j)} < \mathtt{d(ldr, i)})$$

By induction on causal order of `rcv` events.

Lemma 3. $\forall \mathtt{i} : \mathtt{R}.\ \forall \mathtt{e}' \mathbin{@} \mathtt{i}.\ (\mathtt{kind(e')} = \mathtt{leader} \Rightarrow \mathtt{i} = \mathtt{ldr})$

If $\mathtt{kind(e')} = \mathtt{leader}$, then by property 5, $\exists \mathtt{v} \mathbin{@} \mathtt{i}.\mathtt{rcv}\,(\mathtt{in(i)}, <\mathtt{vote}, \mathtt{uid(i)}>)\,.$

Hence, by Lemma 2 $\mathtt{i} = \mathtt{ldr} \vee (\mathtt{d(ldr, i)} < \mathtt{d(ldr, i)})$

but the right disjunct is impossible.

Finally, from property 4, it is enough to know

$\exists \mathtt{e}.\mathtt{kind(e)} = \mathtt{rcv}\,(\mathtt{in(ldr)}, <\mathtt{vote}, \mathtt{uid(ldr)}>)$

which follows from Lemma 1.

QED

Realizing the clauses of LE(R,es)

We need to show that each clause of `LE(R,es)` can be implemented by a piece of a distributed system, and then show the pieces are compatible and feasible.

We can accomplish this very logically using these Lemmas:

- Constant Lemma
- Send Once Lemma
- Recognizer Lemma
- Trigger Lemma

Leader Election Message Automaton

state $me : \square$; initially $uid(i)$

state $done : B$; initially $false$

state $x : B$; initially $false$

action $vote$; precondition $\neg done$

  effect $done := true$

  sends $[msg(out(i), vote, me)]$

action $rcv_{in(i)}(vote)(v) : \square$;

  sends if $v > me$ then $[msg(out(i), vote, v)]$ else[]

  effect $x :=$ if $me = v$ then $true$ else x

action $leader$; precondition $x = true$

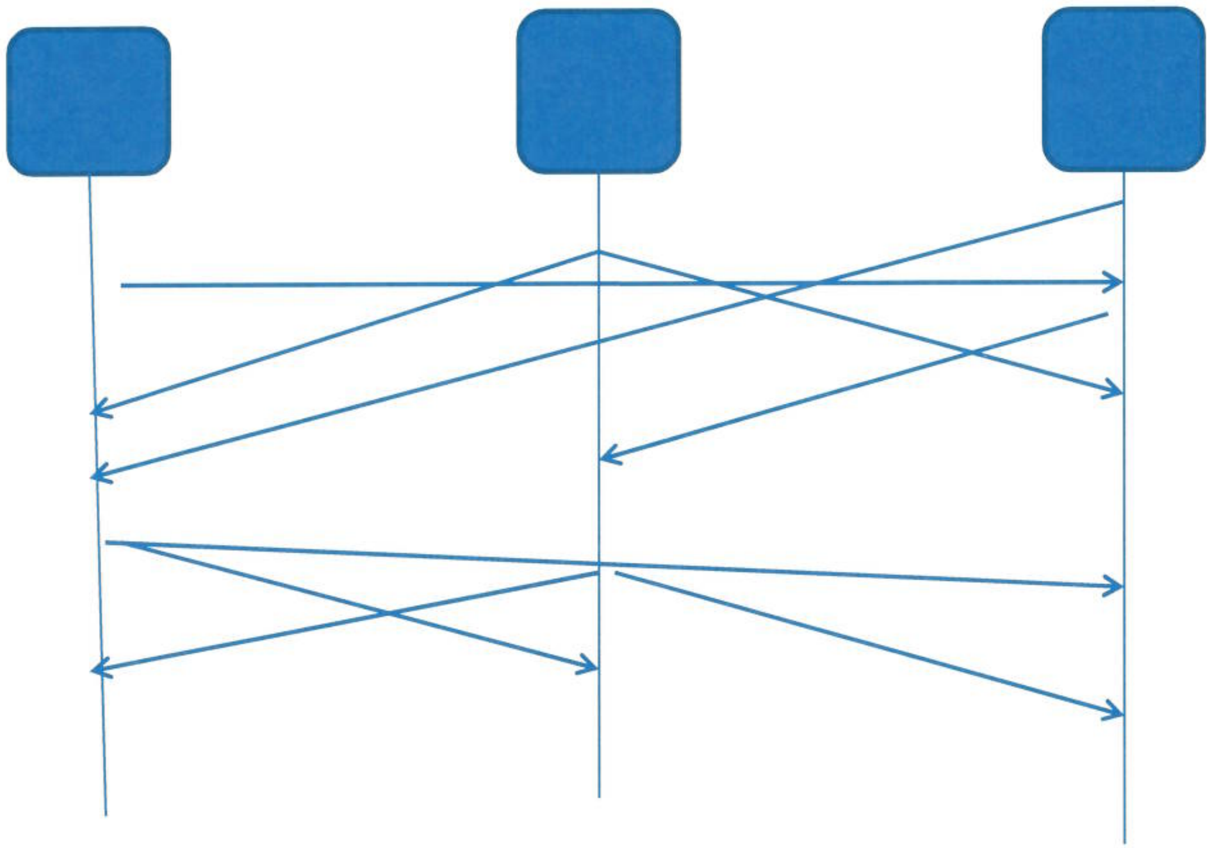
only $rcv_{in(i)}(vote)$ affects x

only $vote$ affects $done$

only $\{vote, rcv_{in(i)}(vote)\}$ sends $out\ (i), vote$

Requirements of Consensus Task

Use asynchronous message passing to decide on a value.



Logical Properties of Consensus

P1: If all inputs are unanimous with value v, then any decision must have value v.

All v:T. (If All e:E(Input). Input(e) = v then
All e:E(Decide). Decide(e) = v)

Input and Decide are event classes that effectively partition the events and assign values to them. The events are points in abstract space/time at which "information flows." More about this just below.

Logical Properties continued

P2: All decided values are input values.

All e:E(Decide). Exists e':E(Input).

e' < e & Decide(e) = Input(e')

We can see that P2 will imply P1, so we take P2 as part of the requirements.

Event Classes

If X is an event class, then E(X) are the events in that class. Note E(X) effectively partitions all events E into E(X) and E-E(X), its complement.

Every event in E(X) has a value of some type T which is denoted X(e). In the case of E(Input) the value is the typed input, and for E(Decide) the value is the one decided.

Further Requirements for Consensus

The key safety property of consensus is that all decisions agree.

P3: Any two decisions have the same value.

This is called agreement.

All e1,e2: E(Decide). Decide(e1) = Decide(e2).

Specific Approaches to Consensus

Many consensus protocols proceed in rounds, voting on values, trying to reach agreement. We have synthesized two families of consensus protocols, the 2/3 Protocol and the Paxos Protocol families.

We structure specifications around events during the voting process, defining E(Vote) whose values are pairs <n,v>, a ballot number, n, and a value, v.

Properties of Voting

Suppose a group G of n processes, Pi, decide by voting. If each Pi collects all n votes into a list L, and applies some deterministic function f(L), such as majority value or maximum value, etc., then consensus is trivial in one step, and the value is known at each process in the first round – possibly at very different times.

The problem is much harder because of possible failures.

Fault Tolerance

Replication is used to ensure system availability in the presence of faults. Suppose that we assume that up to f processes in a group G of n might fail, then how do the processes reach consensus?

The TwoThirds method of consensus is to take n = 3f +1 and collect only 2f+1 votes on each round, assuming that f processes might have failed.

Example for f = 1, n = 4

Here is a sample of voting in the case T = {0,1}.

0	0	1	1	inputs

0 _11	_011	001_	00_1	collected votes
1	1	0	0	next vote

00_1	001_	0_11	_011
0	0	1	1

where f is majority voting, first vote is input

Specifying the 2/3 Method

We can specify the fault tolerant 2/3 method by introducing further event classes.

E(Vote), E(Collect), E(Decide)

E(Vote): the initial vote is the <0,input value>, subsequent votes are <n,f(L)>

E(Collect): collect 2f+1 values from G into list L

E(Decide): decide v if all collected values are v

The Hard Bits

The small example shows what can go wrong with 2/3. It can waffle forever between 0 and 1, thus never decide.

Clearly if there is are decide events, the values agree and that unique value is an input.

Can we say anything about eventually deciding, e.g. liveness?

Liveness

If f processes eventually fail, then our design will work because if f have all failed by round r, then at round $r+1$, all alive processes will see the same $2f+1$ values in the list L, and thus they will all vote for $v' = f(L)$, so in round $r+2$ the values will be unanimous which will trigger a decide event.

Example for f = 1, n = 4

Here is a sample of voting in the case T = {0,1}.

0	0	1	1	inputs
0 01_	001_	001_	_011	collected votes
0	0	0	1	next vote

000_	00_1	0_01	_001
0	0	0	0

where f is majority voting, first vote is input, round numbers omitted.

Safety Example

We can see in the f = 1 example that once a process Pi receives 2/3 unanimous values, say 0, it is not possible for another process to over turn the majority decision.

Indeed this is a general property of a 2/3 majority, the remaining 1/3 cannot overturn it even if they band together on every vote.

Safety Continued

In the general case when voting is not by majority but using f(L) and the type of values is discrete, we know that if any process Pi sees unanimous value v in L, then any other process Pj seeing a unanimous value v' will see the same value, i.e. v = v' because the two lists, Li and Lj at round r must share a value, that is they intersect.

Synthesizing the 2/3 Protocol from a Proof of Design

We can formally prove the safety and liveness conditions from the event logic specification given earlier.

From this formal proof of design, **pf**, we can automatically extract a protocol, first as an abstract process, then by verified compilation, a program in Java or Erlang.

The Synthesized 2/3 Protocol

Begin r:Nat, decided_i, vote_i: Bool,
r = 0, decided_i = false, vi = input to Pi; vote_i = vi

Until decided_i **do**:
1. r := r+1
2. **Broadcast** vote <r,vote_i> to group G
3. **Collect** 2f+1 round r votes in list L
4. vote_i := majority(L)
5. **If** unanimous(L) **then** decided_i := true

End

Abstract Process Model

$M(P)$ == (Atom List) X (T + P)
$E(P)$ == (Loc X M(P)) List
$F(P) = M(P) \rightarrow (P \times E(P))$

It is easy to show that *M and E are continuous type functions* and that *F is weakly continuous. Thus for*

Process == corec(P. F (P))
Msg == M(Process) and Ext == E(Process)
we conclude *Process is a subtype of F(Process),*
Process $\subseteq$ *Msg* $\rightarrow$ *Process X Ext*

A Fundamental Theorem of about the Environment

The Fischer/Lynch/Paterson theorem (FLP85) about the computing environment says:

it is not possible to deterministically guarantee consensus among two or more processes when one of them might fail.

We have seen the possibility of this with the 2/3 Protocol which could waffle between choosing 0 or 1. The environment can act as an adversary to consensus by managing message delivery.

The Environment as Adversary

In the setting of synthesizing protocols, I have shown that the FLP result can be made constructive (CFLP). This means that there is an algorithm, env, which given a potential consensus protocol P and a proof pf that it is nonblocking can create message ordering and a computation based on it, env(P,pf), in which P runs forever, failing to achieve consensus.

Definitions

P is called effectively nonblocking if from any reachable global state s of an execution of P and any subset Q of n − t nonfailed processes, we can find an execution from s using Q and a process P in Q which decides a value v.

Constructively this means that we have a computable function, wt(s,Q) which produces an execution and a state s in which a process, say P decides a value v.

Constructive FLP

Theorem (Constructive FLP): Given any deterministic effectively nonblocking consensus procedure P with two or more processes tolerating a single failure, we can effectively construct a non-terminating execution of it.