Set-Agreement
(Generalizing Consensus)

R. Guerraoui

Consensus

Processes propose each a value and ***agree*** on one of those values

Every process invokes ***propose()*** with a (proposed) input parameter value and eventually return a (decided) value

Consensus

Validity: every value decided has been proposed

Agreement: no two different values are decided

Termination: every correct process that proposes a value eventually decides

Consensus

Consensus is ***impossible*** in an ***asynchronous*** shared memory system (***registers***)

FLP (Dijkstra 2001): A ***read/write*** memory model can remain in a bivalent state for an arbitrarily long period if we have no control over the ***scheduling of the processes***

K-set-agreement

Every process invokes propose() with a (proposed) parameter value and eventually return a (decided) value

Validity: every value decided has been proposed

Agreement: at most k different values are decided

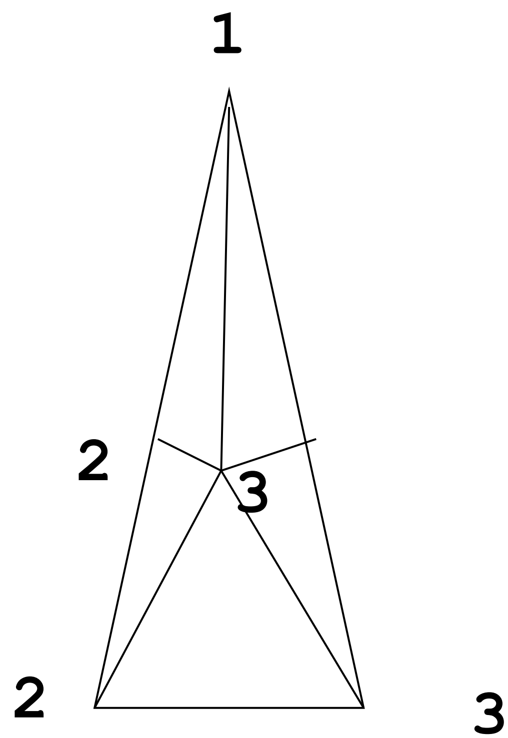
Termination: every correct process eventually decides

K-set-agreement

K-set agreement is wait-free impossible in an asynchronous shared memory system (registers) with k+1 processes

HS,BG,SZ 93 (Godel prize 2004)

K-set-agreement (Sperner)



Sperner's Lemma: at least one triangle has three colors

K-set-agreement

K-set-agreement is wait-free impossible in a system with n processes and k failures

BG: Any (colorless) task that can be solved k resiliently in a system of n processes can be solved wait free in a system of k+1 processes

Safe agreement

- A weak form of consensus with two functions propose(v) and decide()

- When a process invokes propose(v) we say it proposes (v)
- When a process returns v’ from decide() we say it decides v

Safe agreement

- Validity: the value decided is one of the values proposed
- Agreement: no two different values are decided

- Termination: (a) every correct process that invokes propose() eventually returns from the invocation and (b) every correct process that invokes decide() eventually returns from the invocation unless some process fails while proposing

Safe agreement algorithm

propose(v)

- write v at level 1
- if there is a value at level 2, put v at level 0
  - else write v at level 2

decide()

- wait until there is no value at level 1
- return the smallest value at level 2

From k-resilency to wait-freedom

propose(v)

- // for all j from 1 to n

- while(true)
- - mutex(propose_j(v))
- - v_j=decide()
- - return(v_j)

Consensus

Consensus can be implemented with little synchrony (eventual leader) – or with a strong object (C&S)

Using consensus, processes can implement any shared object: universal construction

K-set-agreement

Leader(): returns a process such that eventually the same correct process is returned to all

Leader-k(): returns a subset of processes of size k such that eventually the set is the same and contains at least one correct process

Consensus algorithm (functions)

- To simplify the presentation, we assume two functions applied to Reg[1,..,N]
  - highestTsp() returns the highest timestamp among all elements Reg[1].T, Reg[2].T, .., Reg[N].T
  - highestTspValue() returns the value with the highest timestamp among all elements Reg[1].V, Reg[2].V, .., Reg[N].V

Consensus algorithm

- propose(v): while(true)
  - if leader() then
    - Reg[i].T.write(ts);
    - val := Reg[1,..,n].highestTspValue();
    - if val = ⊥ then val := v;
    - Reg[i].V.write(val,ts);
    - if ts = Reg[1,..,n].highestTsp()
    - then return(val)
    - ts := ts + n

K-set-agreement algorithm (functions)

- To simplify the presentation, we assume two functions applied to Reg[1,..,N]
  - highestTsp() returns the highest timestamp among all elements Reg[1].T, Reg[2].T, .., Reg[N].T
  - highestTspValue_k() returns the k values with the highest timestamp among all elements Reg[1].V, Reg[2].V, .., Reg[N].V

K-set-agreement

- propose(v): while(true)
  - if leader_k() then
    - Reg[i].T.write(ts);
    - val := Reg[1,..,n].highestTspValue();
    - if val = ⊥ then val := v;
    - Reg[i].V.write(val,ts);
    - if ts in Reg[1,..,n].highestTsp_k()
    - then return(val)
    - ts := ts + n

K-vector consensus (Afek et al)

- K-set agreement is equivalent to a k-vector consensus (kVectCons) object
- Every process invokes kVectCons with propose(kVect) and returns a vector of size k

K-vector consensus

- Validity: any non nil element returned at position i has been proposed at position i
- Agreement: no two non-nil elements returned at the same position are different
- Termination: Every correct process that proposes eventually returns, and any vector returned has exactly one non-nil element

From k-vector consensus to k-set

- propose_k(v):
  - (vect) = propose_SkVect(v,v,..v)
  - let v be the non nil value in vect
  - return(v)

From k-set to k-vector

- We first go through a simple version of k-vector consensus (kS-vector) where the processes propose a value and return a consensus vector (with the same properties as vector consensus)

From k-set to k-Svector

- propose_kSVect(v):
  - v = propose_k(v)
  - Reg[i].write(v);
  - snap = Reg.snapshot()
  - let j be the number of non-nil values in snap and v the smallest value in snap
  - return(j,v)

From k-set to k-vector

- propose_SkVect(v):
  - v = propose_k(v)
  - Reg[i].write(v);
  - snap = Reg.snapshot()
  - let j be the number of non-nil values in snap and v the smallest value in snap
  - return(j,v)

From k-Svector to k-vector

- propose_kVect(vect):
  - (j,vect) = propose_kSVect(vect)
  - return(j,vect(j))

Universality [Lamport 77]

- Using consensus, processes can implement any shared object

Universality [Lamport 77]

- Assume an infinite list of requests available to each process:
  - *commands* accessed through *next()*
- Assume a state machine object of which each process holds a copy:
  - sM accessible through *perform*()
- Assume an infinite list of consensus objects shared by the processes:
  - *Consensus* accessed through *next()*

Universality [Lamport 77]

- Algorithm
  - while(true)
  - c = commands.next()
  - cons = Consensus.next()
  - c’ = cons.propose(c)
  - sM.perform(c’)

Universality

- Safety (total order): if a process performs request c without having performed c', then no process performs c' without having performed c. This follows from the use of consensus objects in the same order by all the processes.

- Liveness: if at least one process is correct, then the state machine progresses (executes an infinite number of steps). This follows from the liveness of consensus

What form of universality with set-agreement?

What about several state machines of which at least one progresses

Can we implement k < n state machines?

Implementing k state machines implies solving k-set agreement

K-set agreement

- K-set agreement: a function propose() through which a process proposes a values and decides a value

- Validity: the value decided is one of the values proposed
- Agreement: at most k different values are decided
- Termination: every correct process that proposes eventually decides

Implementing k state machines implies solving k-set agreement

Are these problems equivalent?

Yes

Generalized universality

- Using consensus, processes can implement a shared state machine that makes progress

- Using k-set agreement, processes can implement k state machines of which at least one makes progress

k state machines

- Assume k state machines, sM(i), each process holding a copy of each one, accessible through *perform*()
- Assume k infinite list of commands available to each process:
  - *commands(j)* accessed through *next()*
- Assume an infinite list of safe agreement objects shared by the processes:
  - *sCons* accessed through *next()*

Generalized universality (2)

- Use a list of k-vector consensus objects (kVectCons) to execute the commands on the k state machines

Universality [Lamport 77]

- Algorithm
  - while(true)
  - - c = commands.next()
  - - cons = consensus.next()

  - - c' = cons.propose(c)
  - - sM.perform(c')

Generalized universality?

- Algorithm
  - while(true)
  - - for j = 1 to k: com(j) = commands(j).next()
  - - kVectC = kVectCons.next()

  - - (c,i) = kVectC.propose(com)
  - - sM(i).perform(c)

Generalized universality?

- Algorithm
  - while(true)
  - - for j = 1 to k: com(j) = commands(j).next()
  - - kVectC = kVectCons.next()

  - - (c,i) = kVectC.propose(com)
  - - Register.write(c,i)
  - - sM(i).perform(c)
  - - Read Registers and perform on sM(j') if any

Abortable consensus

- When a process invokes propose(v) we say it proposes (v)
- When a process returns (v,V) from propose() we say it decides v; values in V are said to be returned
  - If V is empty, we say the process commits v. Else we say it aborts with v because of V.

Abortable consensus

- Validity: any value returned has been proposed
- Agreement: if a value v is decided then no other value is decided
- Termination: (a) every correct that proposes eventually decides and (b) if all processes propose the same value then no process aborts

Abortable consensus

propose(v)

- write v at level 1
- write V, the set of all values at level 1, at level 2
- If all V at level 2 are the same singleton v
  - then return(v)
- else, if there is some singleton V = v, then

return (v,V) where V is the union of all values

else return(v,V) where V is the union of all values at level 2

Generalized universality

- Use a list of k-vector consensus objects (kVectCons)

as well as …

- a list of k-vector abortable consensus (kVectACons)

Generalized universality (step 0)

Algorithm

- newCom = commands.next()
- while(true)
- - kVectC = kVectCons.next()
- - kVectAC = kVectACons.next()
- …

Generalized universality (step 1)

Algorithm (cont'd)

- …
- (c,i) = kVectC.propose(newCom)
- …

Generalized universality (step1-2)

Algorithm (cont'd)

- …
- (c,i) = kVectC.propose(newCom)
- (vect(i),V(i)) = kVectAC(i).propose(c)
- …

Generalized universality (step1-2-2’)

Algorithm (cont’d)

- …
- (c,i) = kVectC.propose(newCom)
- (vect(i),V(i)) = kVectAC(i).propose(c)
- for j = 1 to k except i:
  - (vect(j),V(j)) = kVectAC(j).propose(newCom(j))

  …

Generalized universality (step 3)

Algorithm (cont'd)

…

for i = 1 to k

- If V(i) is empty then
  - sM(i).perform(vect(i))
  - newCom(i) = commands(i).next()
- else
  - newCom(i) = vect(i)

Generalized universality (step 3)

for i = 1 to k

- if V(i) empty then
  - if vect(i) > newCom(i) then
    - sM(i).perform(newCom(i))
  - sM(i).perform(vect(i))
  - newCom(i) = commands(i).next()

- else
  - if some element v in V(i) > vect(i) then
    - sM(i).perform(v)
    - newCom(i) = commands(i).next()

Generalized universality (safety)

Total order: if a process performs command c on state machine j without having performed c' on j, then no process performs c' on j without having performed c.

This follows from:

- Lemma 1: all commands executed come from abortable consensus
- Lemma 2: abortable consensus objects are executed in the same order by all processes

Generalized universality (liveness)

- Liveness: if one process is correct, then at least one state machine progresses.

This follows from the following:

- Lemma 3: At least one abortable consensus commits in every iteration

- Lemma 4: Every correct process executes a command every two steps