FDS12:
Consensus hierarchy and universal construction

So far…

Read-write registers cannot solve:

- Wait-free consensus
- Wait-free set agreement
- 1-resilient consensus
  - ✓Can be generalized to k-resilient k-set agreement

Today

- Combining registers with stronger objects
  - ✓Consensus and test-and-set (T&S)
  - ✓Consensus and queues
- Universality of consensus
  - ✓Consensus can be used to implement any object
- Consensus number
- Message-passing in shared memory

Test&Set atomic objects

Exports one operation test&set() that returns a value in {0,1}

Sequential specification:

The first atomic operation on a T&S object returns 1, all other operations return 0

2-process consensus with T&S

Shared objects:

T&S TS

Atomic registers R[0] and R[1]

Upon propose(v) by process p_i (i=0,1):

```
R[i] := v
if TS.test&set()=1 then
        return R[i]
else
        return R[1-i]
```


3-process consensus with T&S?

Assume A solves consensus among three-processes p_0, p_1, p_2, using registers and T&S objects

Consider the *critical bivalent* run R of A: every one-step extension of R is univalent (HW: show that it exists)

W.L.O.G., assume that

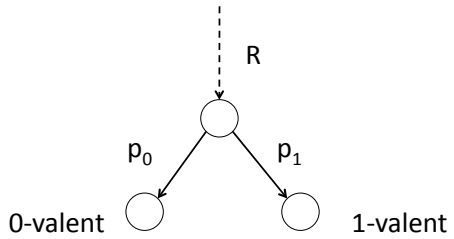
- $R.p_0$ is 0-valent
- $R.p_1$ is 1-valent

We establish a case where some process cannot distinguish a 0-valent state from a 1-valent one

3-process consensus with T&S?

If p_0 and p_1 access different objects at the end of R, or p_0 and p_1 access the same *register* in R, then we come back to the read-write case (p_0 or p_1 cannot decide in some solo extension)

Thus, p_0 and p_1 are about to access the same T&S object



3-process consensus with T&S

Suppose p_0 and p_1 access the same T&S object

- ✓p_2 cannot distinguish $R.p_0$ and $R.p_1$ in a solo extension (T&S returns 0 and all other objects have the same states) => p_2 can never decide

=> T&S and registers cannot (wait-free) solve 3-process consensus

FIFO Queues

Exports two operations enqueue() and dequeue()

- enqueue(v) adds v to the end of the queue
- dequeue() returns the first element in the queue
  (LIFO queue returns the last element)

2-process consensus with queues

Shared:

Queue Q, initialized (winner,loser)

Atomic registers R[0] and R[1]

Upon propose(v) by process p_i (i=0,1):

```
R[i] := v
if Q.dequeue()=winner then
        return R[i]
else
        return R[1-i]
```

3-process consensus with queues?

- Let A solve consensus among p_0, p_1, p_2, using registers and queues
- Similarly, there exists a critical run R in which the same queue is about to be accessed by p_0, p_1, p_2
- Suppose R. p_0 is 0-valent, R. p_1 is 1-valent, and p_0 and p_1 access the same queue
  - ✓The decision is "encoded" in the queue
  - ✓But the queue can only be accessed with dequeue() and enqueue()
  - ✓At least one process is confused (Homework: finish the argument)

=> Consensus power of a queue is 2 (similar for stacks)

But why consensus is interesting?

Because it is universal!

- If we can solve consensus among N processes, then we can *implement* any object shared by N processes
  - ✓T&S and queues are universal for 2 processes
- A key to implement a generic fault-tolerant service (replicated state machine)

What is an *object* ?

Object O is defined by the tuple (Q,O,R,σ):

- Set of states Q
- Set of operations O
- Set of outputs R
- Sequential specification σ, a subset of OxQxRxQ:
  - ✓(o,q,r,q') is in σ ⇔ if operation o is applied to an object in state q, then the object *can* return r and change its state to q'
  - ✓Total on OxQ (defined for all o and q)

Deterministic objects

- An operation applied to a *deterministic* object results in exactly one (output,state) in RxQ, i.e., σ can be seen a function OxQ -> RxQ
- E.g., queues, counters, T&S are deterministic
- Unordered set (put/get) – non-deterministic

Example: queue

Let V be the set of possible elements of the queue

Q=V* (all sequences with elements in V)

O={enq(v)$_{v \text{ in } V}$,deq()}

R=V U {Ø} U {ok}

σ(enq(v),q)=(ok,q.v)

σ(deq(),q.v)=(v,q)

σ(deq(), Ø)=(Ø, Ø)

Implementation: definition

A distributed algorithm A that, for each operation o in O and for every p_i, describes a concurrent procedure o_i using base objects

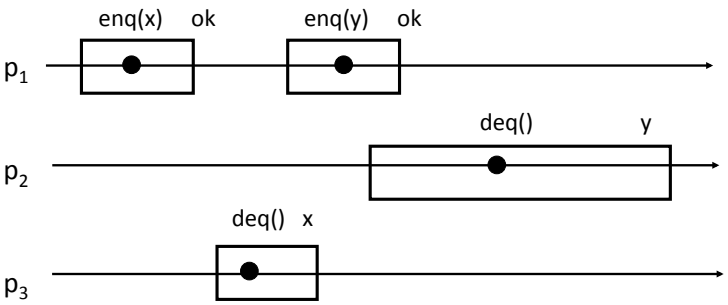
A run of A is *well-formed* if no process invokes a new operation on the implemented object before returning from the old one (we only consider well-formed runs)

Implementation: correctness

A (wait-free) implementation A is correct if in every well-formed run of A

- Wait-freedom: every operation run by p_i returns in a finite number of steps of p_i
- Linearizability ≈ operations "appear" instantaneous (the corresponding *history* is *linearizable*)

Linearization



p_1-enq(x); p_1-ok; p_3-deq(); p_3-x;
p_1-enq(y); p_1 –ok; p_2-dequeue(); p_2-y

Universal construction

Theorem 1 [Herlihy, 1991] If N processes can solve consensus, then N processes can (wait-free) implement every object O=(Q,O,R,σ)

A moment of meditation

Suppose you are given an unbounded number of consensus objects and atomic read-write registers

You want to implement an object O=(Q,O,R,σ)

How would you do it?

Universal construction: idea

Every process that has a pending operation does the following:

- Publish the corresponding *request*
- Collect published requests and use consensus instances to serialize them: the processes agree on the order in which the requests are executed
- Processes agree on the order in which the published requests are executed

Universal construction: variables

Shared abstractions:
N atomic registers R[0,…,N-1], initially Ø
N-process consensus instances C[1], C[2], …

Local variables for each process p_i:
integer *seq*, initially 0
// the number of p_i's requests executed so far
integer *k*, initially 0
// the number of batches of
// all requests executed so far
sequence *linearized*, initially empty
//the serial order of executed requests

Universal construction: algorithm

Code for each process p_i: implementation of operation op

```
seq++
R[i] := (op,i,seq)                      // publish the request
repeat
        V := read R[0,…,N-1]            // collect all requests
        requests := V-{linearized}      //choose not yet linearized requests
        if requests≠Ø then
            k++
            decided:=C[k].propose(req)
            linearized := linearized.decided
            //append decided request in some deterministic order
until (op,i,seq) is in linearized
return the result of (op,i,seq) in linearized
        // using the sequential specification σ
```


Universal construction: correctness

- Linearization of a given run: the order in which operations are put in the *linearized list*
  - ✓ Agreement of consensus: all *linearized* lists are related by containment (one is a prefix of the other)
- Real-time order: if op1 precedes op2, then op2 cannot be linearized before op1
  - ✓Validity of consensus: a value cannot be decided unless it was previously proposed

Universal construction: correctness

- Wait-freedom:
  - ✓Termination and validity of consensus: there exists k such that the request of p_i gets into *req* list of every processes that runs C[k].*propose(req*)

Another universal abstraction: CAS

Compare&Swap (CAS) stores a *value* and exports operation CAS(u,v) such that:

- If the current value is u, CAS(u,v) replaces it with v and returns u
- Otherwise, CAS(u,v) returns the current value

A variation: CAS returns a boolean (whether the replacement took place) and an additional operation read() returns the value

N-process consensus with CAS

Shared objects:
CAS CS initialized Ø
// Ø cannot be an input value

Code for each process p_i (i=0,…,N-1):
v_i := input value of p_i
v :=CS.CAS(Ø,v_i)
if v = Ø
return v_i
else
return v

M-consensus object

M-consensus stores a value in {Ø} U V and exports operation propose(v), v in V:

For 1st to Mth propose() operations:

- If the value is Ø, then propose(v) sets the value to v and returns v
- Otherwise, returns the value

All other operations do not change the value and return Ø

M-process consensus with M-consensus

Immediate: every process p_i simply invokes C.propose(input of p_i) and returns the result of it

(M+1)-consensus using M-consensus?

Impossible: (M+1)-th process is confused

Consensus number

An object O has consensus number k (we write cons(O)=k) if

- k processes can solve consensus using registers and any number of copies of O
- but k+1 processes cannot

If no such number k exists for O, then cons(O)=∞

(k=cons(O) is the maximal number of processes that can be perfectly synchronized using copies of O and registers)

Consensus numbers

- cons(register)=1
- cons(T&S)=cons(queue)=2
- …
- cons(N-consensus)=N
  - ✓N-consensus is N-universal!
- …
- cons(CAS)=∞

Open questions

- Robustness

  Suppose we have two objects A and B, cons(A)=cons(B)=k

  Can we solve (k+1)-consensus using registers and copies of A and B?
- Can we implement an object of consensus power k shared by N processes (N>k) using k-consensus objects?

What about message passing?

Message-passing

- Which results for shared memory can be translated into message-passing models?
- Consider a network where every two processes are connected via a reliable channel
  - ✓no losses, no creation, no duplication

Implementing message-passing

Theorem 1 A reliable message-passing channel between two processes can be implemented using two 1W1R registers

Corollary 1 Consensus is impossible to solve in an asynchronous message-passing system if at least one process may crash

Implementing shared memory

Theorem 2 A 1W1R regular register can be implemented in a (reliable) message-passing model where a majority of processes are correct

Implementing a 1W1R register

Upon write(v)
  t++
  send [v,t] to all
  wait until received [ack,t] from a majority
  return ok

Upon read()
  r++
  send [?,r] to all
  wait until received {(t',v',r)} from a majority
  return v' with the highest t'

Implementing a 1W1R register, contd.

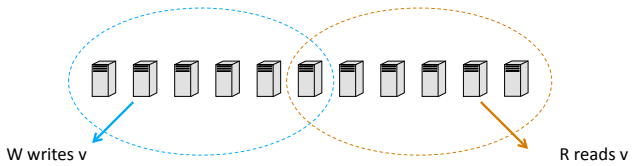
Upon receive [v,t]
  if $t>t_i$ then
    $v_i := v$
    $t_i := t$
    send [ack,t] to the writer

Upon receive [?,r]
  send $[v_i,t_i,r]$ to the reader

A correct majority is necessary

Otherwise, the reader may miss the latest written value

The quorum (set of involved processes) of any write operation must intersect with the quorum of any read operation:



There is more to this

- Adaptive algorithms
- Non-uniform computing models
- Failure detection
- Transactional memory

Check http://www.net.t-labs.tu-berlin.de/~petr/ for more information