

# Set Agreement

---

Lecture of December 7, 2009

Until very recently there were only two main object considered in distributed computing: registers and consensus. We have also seen that the consensus is impossible in a totally asynchronous shared memory system if at least one process can fail. This is because the read/write memory model can remain in a bivalent state for an arbitrarily long period (FLP impossibility result).

The generalization of the consensus we want to present in the lecture is a K-Set Agreement object, which is defined with the following properties:

- Validity: every value decided has been proposed
- Termination: every correct process eventually decides
- **Agreement:** at most  $k$  different values are decided

We can see that the first two properties are the same as the ones of the consensus object, but the relaxation lies in the Agreement property, which allows at most  $k$  different values to be decided (in consensus  $k = 1$ ).

## Impossibility results

Although *KSA* looks much simpler for implementation than Consensus, because of the relaxed agreement property, we will now see a few impossibility results which show the contrary.

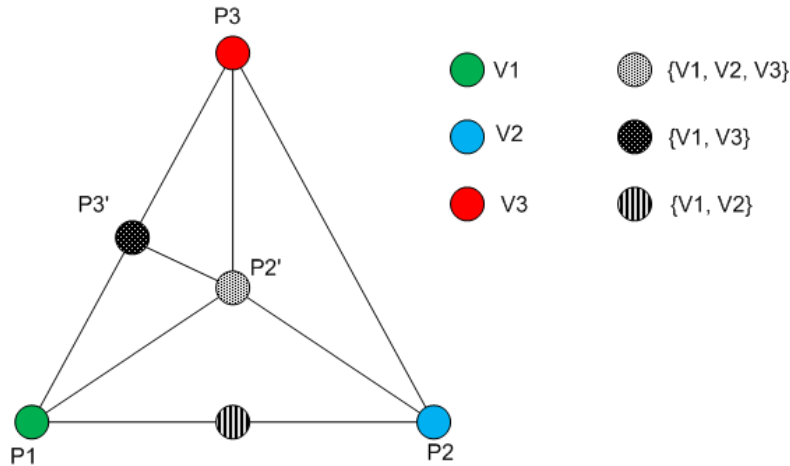
**Theorem 1.** *KSA is impossible in an asynchronous shared memory system with  $k+1$  processes, of which  $k$  might fail.*

As the general proof of the theorem would require a lot of time and since it is not the main topic of this lecture, we will now give just the intuition of the proof for  $k = 2$ . Let us consider a system of three processes:  $p_1, p_2, p_3$ . Furthermore, let us assume such a computational model where each process performs execution in rounds and the round of each process consists of the following two phases:

- writing a value to the shared memory
- taking a snapshot of the shared memory

After a process finishes the first round, it starts the second one and so on. Because of the asynchrony of processes, we have that not all processes see same values (for example, the fastest one will see only its own written value). We will now look deeply in what can happen in just one round  $r$ . Say that in the round  $r$  each process  $p_i$  has written a value  $v_i$ . Then, there exists a triangle  $T$  of the execution, where the values (colors) in the points of a triangle in some triangulation of  $T$  represent a value chosen locally in the round  $r$ . Corners of the triangle  $T$  can have only one value (e.g. the process in that execution was very fast so it saw only its own written value). The color of the point on an edge between two corners can be any of the colors of the two corners (e.g. this process is neither the slowest nor the fastest, so it

saw the two values in the snapshot). The third possibility is when the point is somewhere in the triangle, in which case the color of such a point is any color of the corners (e.g. this process is the slowest, so it saw all the values). One possible execution is shown in the Figure 1, where the sub-triangle  $P_1P_2'P_3'$  represents the situation when  $P_1$  is the fastest, then comes  $P_3$ , after which  $P_2$  takes the snapshot. Values of snapshots are represented in brackets when there are a few possible values.



Let us see if we can solve *KSA* at round  $r$ . Since every triangle in a triangulation of  $T$  is some possible execution of the algorithm (where the colors of the points are decided values in round  $k$ ), we are interested in asking if there exist no triangle in the triangulation which contains more than 2 colors (2-set-agreement in this concrete case). However, because of the two-dimensional case of *Sperner's lemma*, which states that for a given triangle  $T$  (where the corners of  $T$  are colored differently) and any triangulation of  $T$  (where the vertices of the triangles in a triangulation get colors as we described earlier), there exists at least one triangle in the triangulation whose vertices are colored with three different colors. In other words, if we adapt the lemma to our needs, we have that there always exists at least one execution where the processes decided on the three different values. Furthermore, it is possible to show that no matter how many rounds we execute, if we choose the sub-triangle with three colors to be new  $T$ , we have that the lemma still holds (as the triangles recursively have also some 3-colored sub-triangle), so there is always an execution where all the values are decided (none of the values is rejected).

The logical continuation is to see whether more processes will add us some power, so that we can implement *KSA* with more than  $k+1$  process, where  $k$  of them can fail. It is shown bellow that this is also impossible.

**Theorem 2.** *KSA is impossible in a system with  $n$  process and  $k$  failures.*

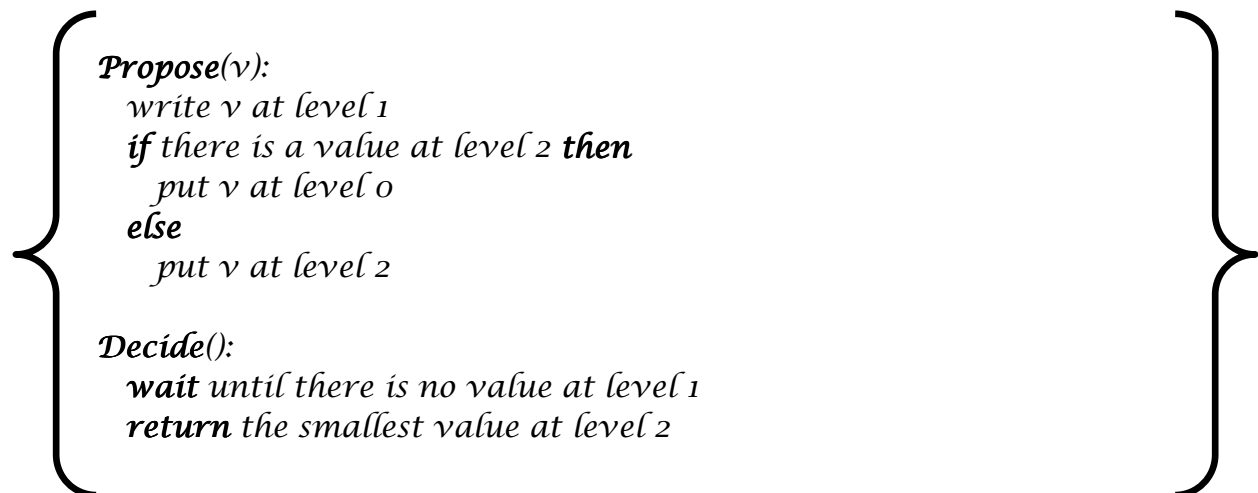
In order to show this, we will first show one interesting result known as Borowsky-Gafni simulation (*BGS*).

**Theorem 3.** Any task which can be solved  $k$  resiliently in a system of  $n$  process can be solved wait free in a system with  $k+1$  process.

This is because Borowsky and Gafni showed that  $n$  processes can be simulated by  $k + 1$  processes, where  $k$  of them can fail. For showing the simulation, we have to define a *Safe agreement* object, which has two operations *propose* and *decide* and is defined with the following properties:

- Validity: the value decided is one of the proposed values
- Agreement: no two different values are decided
- **Termination:**
  - Every correct process that invokes *propose* eventually returns from the invocation
  - Every correct process that invokes *decide* eventually returns from the invocation, unless some process fails while proposing

Safe agreement can be easily implemented in the following way:



```
Propose( $v$ ):  
  write  $v$  at level 1  
  if there is a value at level 2 then  
    put  $v$  at level 0  
  else  
    put  $v$  at level 2  
  
Decide():  
  wait until there is no value at level 1  
  return the smallest value at level 2
```

The *Validity* trivially holds. The *Agreement* can also be easily proven, because once some value is decided, it is the only value possible to be decided later. This is because once there are no values at level 1, all incoming proposals will read that level 2 is full so new values can be put only at level 0. Therefore, no other value can ever be decided since the set at level 2 will remain the same. The only place where a process can be blocked is obviously deciding method. However, this will happen only if a process fails after it writes its value at level 1, but before moving it to the level 0 or 2. This satisfies the *Termination* property, so the algorithm is correct.

Now we can show what *BGS* is all about. It consists of simulating the behavior of  $n$  processes by  $k+1$  in such a way that each of  $k + 1$  processes simulates the work of  $n$  threads. The only important thing is that  $i^{th}$  thread of each process needs to produce the same result in all processes. For this purpose we use *Safe agreement*, so that  $i^{th}$  thread in each process decides on the same value (or stays blocked). Even though from the *Termination* property of safe agreement some thread can stay blocked in while deciding, this can happen at most  $k$  times, so the remaining  $n - k$  threads can solve the problem as in the

way they would do it if  $k$  processes have failed. Let us see how would the *propose* method would look at the process  $i$  which proposes a value  $v$ .

***Propose***( $v$ ):

*create  $n$  threads and in each thread  $j$  do*

***while not finished***

*mutex*[ $i$ ] {

*sa*[ $j$ ].*propose*( $v$ )

}

*v<sub>j</sub> = sa*[ $j$ ].*decide*() // can block this thread

// do the part of the algorithm as would be done if

// process  $j$  has proposed value  $v_j$  in the solution

// with  $n$  processes and sets the value of *finished*

// when the algorithm is done

// (as in normal case with  $n$  processes)

There is a shared array of safe agreements *sa* and we consider that each process has its local mutex, which is used for controlling the access to the safe agreement array in one process. The simulation above shows how we can transform some problem where we need to propose a value, by simulating the execution of  $n$  processes at  $k + 1$  processes. At each process we create  $n$  threads after which we start looping until the decision is made, which is determined by the original algorithm. There are  $n$  safe agreements so that threads can decide between them on the value which should be chosen for passing to the original solution. Once this is decided, the value is passed to the original algorithm, which cannot make a difference between this execution (when the operations are invoked by threads) from the one when different  $n$  processes invoke operations. We know that the *decide* method can block a thread, but since no more than  $k$  threads can fail, that is at least  $n - k$  threads will remain correct at one process, which we know the original algorithm can handle since it is  $k$  resilient.

The *Theorem 2* follows directly from the *Theorem 1* and the *Theorem 3*. If there exists an algorithm which implements *KSA* with  $n$  processes (where  $k$  might fail), from *Theorem 3* that means there exists also an algorithm which implements *KSA* with  $k + 1$  processes. However, we know that this is impossible because of the *Theorem 1* (even if none of  $k + 1$  processes fails), so the *Theorem 2* holds.

As a conclusion, we have that the *KSA* is impossible in an asynchronous system with  $n$  processes, where  $k$  of them are faulty.

## **In the partially synchronous model**

The same approach we used for implementing the Consensus object can be used here as well. More precisely, we can adapt the algorithm used for the Consensus implementation to implement *KSA* object, provided that we have an access to *leader<sub>k</sub>* method (which can be implemented only in a system with some timing assumptions, i.e. partially synchronous).

*leader<sub>k</sub>* – returns a subset of processes of size  $k$  such that eventually the set is the same at every process and contains at least one correct process

With *leader<sub>k</sub>* construct, we can implement the KSA consensus with a few modifications of the consensus implementation which uses *leader*. We assume that processes share the array of registers  $T$ , which hold timestamp values, and the array of registers  $V$ , which hold  $(tsp, value)$  pairs. In addition, *highestTspValue* returns the *value* with the highest timestamp among all the pairs  $(tsp_i, value_i)$  in  $V$  and *highestTsp<sub>k</sub>* returns the  $k$  highest timestamps among all the timestamps in  $T$ . We then can implement the KSA as follows:

```
Propose( $v$ ):  
  while (true)  
    if  $i$  in leaderk() then  
       $T[i].write(ts)$   
       $val = highestTspValue()$   
      if ( $val = \perp$ ) then  
         $val = v$   
       $V[i].write(val, ts)$   
      if ( $ts$  in highestTspk()) then  
        return  $val$   
       $ts = ts + n$ 
```

The *termination* property holds, because *leader<sub>k</sub>* will eventually allow execution of only  $k$  processes, which means that they will be among the highest timestamps, so they will be able to decide. However, it is not so obvious that the agreement property also holds. If the decision of some process  $p$  is one of the first  $k$  decisions, then this doesn't violate the agreement property for sure. However, let us see what happens in other case, if the decision of a process  $p$  is not among the first  $k$  decisions. That means that when  $k$  processes have decided,  $p$ 's timestamp in  $T$  was not among the  $k$  highest. Consequently,  $p$ 's timestamp in register  $V$  was not among the highest  $k$  timestamps in  $V$ . Therefore, process  $p$  will set *val* to some of the already decided values. Therefore, the *agreement* property also holds.

Leaders play the role of failure detectors and what is interesting is that this *leader<sub>k</sub>* as defined above is the weakest failure detector which allows us to implement the KSA. However, there is an open problem concerning the sufficient (the weakest) base object which allows the implementation of KSA in a completely asynchronous system (just as we know that Compare&Swap is sufficient for implementing consensus).

## KSA and universality

We have seen that consensus object is universal in a sense that it is possible to wait-free implement any object using consensus as a base object (by replicating object to be implemented in every process and

ordering performed operations of the object by consensus objects). We will now see what kind of universality *KSA* offers.

*K-Vector Consensus (KVC)* is defined by the following properties:

- **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

It can be thought as a vector of  $k$  consensus objects. Each process proposes a value to a position in a vector and a consensus at that position takes care of returning only one non nil value at that position, among all the proposed values for that slot. The difference between the array of  $k$  consensus objects and  $k$ -vector consensus is that in the later can return to some process nil value as a decision at the position  $i$ , while some other process can get value  $v$  at the same position  $i$ .

It can be shown that *KVC* and *KSA* are equivalent objects. For that purpose, we will now show one possible implementation of the *KSA* using *KVC* as base object:

```

{
  Propose( $v$ ):
     $vect = kvc.propose(v, v, v, \dots, v)$ ;
    return some non nil value in  $vect$ 
}

```

It is obvious that the implementation above is correct (since there are at most  $k$  different values in  $k$ -vector consensus), so what we do next is implementation of *KVC* using *KSA* as base object, which is a bit more complicated. The argument of the *propose* method is a vector of non-nil values, and the output of the method is also a vector, which has exactly one non nil value.

```

{
  Propose( $vector$ ):
     $v\_d = ksa.propose(vector)$  // decided vector, at most k different vects
     $r[i].write(v\_d)$ 
     $c = snapshot(r)$ 
    position = the number of different non-nil elements in c
    value = the vector in c with the smallest weight
    return ( $position, value[position]$ )
}

```

By the definition of *KSA* we have that at most  $k$  different  $v\_d$  vectors will be returned when proposing a vector of values. In addition, we have that the vector will be some of the proposed vectors. Therefore, the *validity* property of *KVC* holds, because we return the element at some position (which is always smaller than  $k$ ) in some original vector. For checking whether the *agreement* property is satisfied, we will use one characteristic of an atomic snapshot. If we denote as  $C$ , a view that was returned by a *scan*

at a process  $p_i$ , we then have that each  $C_i$  is either a subset or a superset of some other view  $C_j$  ( $C_i$  can be equal to  $C_j$ ). With this characteristic we know that if the set  $c$  of vectors is of the same size, then the elements in it are also the same. Thus, the *value* determined as the vector in the view with the smallest weight will be the same (as the two vectors  $a$  and  $b$  have different weight if and only if at least one position  $a[i] \neq b[i]$ ) in each process which has the same number of elements in the snapshot view. With this, we can conclude that the *value[position]* value returned will be the same value in each process which has the same *position* value. This implies that the *agreement* property is also satisfied.

If we have *KVC* (or equivalently *KSA*), we will now show how we can build  $k$  objects (state machines) such that at least one of them is highly available (makes progress). We assume that there are  $n$  processes, each of them has a local copy of  $k$  state machines, as well as the list of commands for each state machine. There is also an infinite list of  $k$ -vector consensus objects which is shared between the processes. In proving the universality of the consensus object, we had that each of the processes proposes an action, which then waits to get the set of decided actions. This way every process executes the same set of actions in the same order. Here, in generalized universality, we have a similar idea. The difference is that this time each process has a vector  $v$  (of size  $k$ ) of actions it wants to execute ( $v[i]$  is the action for the  $i^{\text{th}}$  object). It proposes the vector to the *KVC* and we know that at one position  $i$  in the decided vector there will be a non-nil value and that action will be executed on object  $i$ . We will now give a few implementations, just to mention problems which exist in some of them. Additionally, we assume that  $(c, i)$  is the only entry in the decided vector where  $c$  is non-nil.

```

{
  First_try():
  while(true) do
    for  $j = 1$  to  $k$  do
       $com[j] = myCommands[j].next()$ 
       $kvc = kVectorConsensuses.next()$ 
       $(c, i) = kvc.propose(com)$ 
       $stateMachine[i].perform(c)$ 
}

```

The problem which exists in this implementation is that a process  $i$  can get non-nil value at position  $m$ , so it executes the action on the machine  $i$ . However, some other process can get nil value at position  $m$  (and not nil at some other position) so it will not be aware of the action which process  $i$  has performed on the state machine  $m$ .

The second implementation is based on sharing the update one process has performed, so that others are aware of the action it has performed.

```

Second_try():
  while(true)
    for j = 1 to k do
      com[j] = myCommands[j].next()
      kvc = kVectorConsensuses.next()
      (c, i) = kvc.propose(com)
      register.write(c, i)
      stateMachine[i].perform(c)
      read registers and perform stateMachine[j] if any

```

The problem in this implementation is that process  $i$  which got non nil at position  $m$  can be too slow, so that some other faster process (which has received nil value for machine  $m$ ) has found empty register for machine  $m$ , so it again doesn't have an accurate state of its local copy of the machine  $m$ .

Before we give the correct implementation of generalized universality, we will first define an *Abortable consensus* object. It has operation *propose(v)*, which proposes value  $v$ . The operation returns a pair  $(v, V)$  which means that value  $v$  was decided and, if the returned set  $V$  is empty, then we say that process **commits**  $v$ . Otherwise, we say that process **aborts** with  $v$  because of  $V$ .

*Abortable consensus* (AC) has the following properties:

- Validity: any value returned has been proposed
- **Agreement**: if a value  $v$  is decided then no other value is decided
- **Termination**: every correct process which proposes eventually decides and if all processes proposed the same value, then no process aborts

The implementation of AC is possible only with registers and is given bellow.

```

Propose(v):
  write v at level 1
  write V (which is the set of all values at level 1) to the 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 the V is the union of all values
  else
    return (v, V) where the V is the union of all values at level 2

```

We will use a shared list of  $k$ -vector abortable consensus objects in order to be sure that the action we want to execute on some state machine is known to the others before we really perform it. This way we



have solved the problem of unsynchronized local copies of the state machines. The whole algorithm is given below.

```

Third_try():
  while (true)
    for j = 1 to k do
       $com[j] = myCommands[j].next()$ 
       $kvc = kVectorConsensuses.next()$ 
       $kvac = kVectorAbortableConsensuses.next()$ 
       $(c, i) = kvc.propose(com)$ 
       $(vect[i], V[i]) = kvac[i].propose(c)$ 

    for each j = 1 to k except i do
       $(vect[j], V[j]) = kvac[j].propose(com[j])$ 

    for j = 1 to k do
      if V[j] is empty then
         $stateMachine[i].perform(vect[j])$ 
         $com[j] = myCommands[j].next()$ 
      else
         $com[j] = vect[j]$ 

```

As we have said, with the abortable consensus we are sure that if some process commits an action, all the processes that come later will get the information about the action. However, if there are many processes competing for the same machine, it is possible that an action will be aborted. Still,  $V$  will contain all the actions proposed and will be stored for proposal in the next iteration (the action might be executed or not in the next iteration, which depends again on the abortable consensus).

The safety of the algorithm above comes from the total order of operations: if a process performs an operation  $c$  on a state-machine  $m$  without having performed the operation  $c'$  on  $m$ , then no process performs  $c'$  on  $m$  without having performed  $c$ . This follows from the following two lemmas:

**Lemma 1.** *All commands executed come from an abortable consensus.*

**Lemma 2.** *Abortable consensus objects are executed in the same order by all processes.*

We can deduce that if one process is correct then at least one state machine progresses (this satisfies the *liveness* property) from the following two lemmas:

**Lemma 3.** *At least one abortable consensus commits in every iteration.*

**Lemma 4.** *Every correct process executes a command every two steps.*