Generalized Universality

Eli Gafni[1] and Rachid Guerraoui[2]

[1] UCLA
[2] EPFL

Abstract. *State machine replication* reduces distributed to centralized computing. Any sequential service, modeled by a state machine, can be replicated over any number of processes and made highly available to all of them. At the heart of this fundamental reduction lies the so called *universal consensus* abstraction, key to providing the illusion of single shared service, despite replication.

Yet, as universal as it may be, consensus is just one specific instance of a more general abstraction, *k-set consensus* where, instead of agreeing on a unique decision, the processes may diverge but at most k different decisions are reached. It is legitimate to ask whether the celebrated state machine replication construct has its analogue with $k > 1$. If it did not, one could question the aura of distributed computing deserving an underpinning Theory for 1, the unit of multiplication, would be special in a field, distributed computing, that does not arithmetically multiply.

This paper presents, two decades after k-set consensus was introduced, the generalization with $k > 1$ of state machine replication. We show that with k-set consensus, any number of processes can emulate k state machines of which at least one remains highly available. While doing so, we also generalize the very notion of consensus universality.

Keywords: State machine replication, k-set consensus, universality.

1 Introduction

One of the most fundamental constructs of distributed computing is the *replicated state machine* protocol [11]. It essentially makes a distributed system emulate a, highly available, centralized one using a *consensus* abstraction [6]. Making the approach efficient by allowing a system to run with little overhead while the system's components are well behaved, and nevertheless not let it commit to a mistake while experiencing faults, has been a trust of distributed computing [4].

How does state machine replication work? A computing service is modeled as a state machine that executes commands deterministically. Processes hold each a copy of this state machine, to which they issue commands. To provide the illusion of sharing a single state machine, the processes use consensus. Each instance of consensus is used to decide which command to execute next and hence make sure all commands are executed on the state machine copies in the same order: this,

J.-P. Katoen and B. König (Eds.): CONCUR 2011, LNCS 6901, pp. 17–27, 2011.

together with the very fact that the state machine is deterministic, implies that all its copies keep the same state. (It was later shown how the choice of which proposed commands to execute next can be made fair [8].) Consensus is said to be *universal* [8] in the sense that its availability implies the fair availability of any shared service.

Yet, as universal as it may be, consensus is just the specific case of a more general abstraction: *k-set consensus* [5], where the number of decisions that can be output by processes is more than 1 but at most k. That realization of Chaudhuri in 1990, the challenge she posed, whether k-set consensus is solvable in a system where the number of processes is larger than k, and her quote of Saks about "smelling like Sperner", has resulted, three years later, in the discovery of the connection between distributed computing and algebraic-topology [9,2,10].[1]

Given the importance of the state machine replication construct and the cornerstone role of consensus, it is natural to ask what form of construct we get if we generalize consensus to k-set consensus. Not being able to generalize state machine replication, and the actual universality of (1-set) consensus, to the case where $k > 1$, would be frustrating and would somehow reveal a hole in the theory of distributed computing.

We show in this paper that k-set consensus is, in a precise sense, *k-universal*: with k-set consensus, we can implement k state machines with the guarantee that at least one machine remains highly available to all processes. In other words, whereas consensus reduces distributed computing to centralized computing, i.e., "1-computing", k-set consensus is the generalization that reduces distributed computing to "k-computing".

"Practical" applications might be foreseen. Multiple state machines, implementing different services, one of which is guaranteed to progress, is better than one state machine that does not progress. This could be the case if consensus cannot be reached but 2-set consensus can: a shared memory system of 3 processes and 1 failure, or a system that provides some weak compare-swap primitive that allows for two winners. In fact, multiple state machines, even implementing the same service, may provide for an interesting alternative behavior to a classical state machine replication scheme at the time when the system is not stable. Instead of blocking like a single machine will do, in our case at least one machine will progress. There is of course the danger that the state machines diverge from each other but many applications can tolerate divergence of view for a while. When the system stabilizes, these divergent views may be reconciled to continue with what is effectively a single view of the system. k read-write processes proceeding wait-free.

The rest of the paper is organized as follows. We first recall below the basic state machine replication construct, then we present the properties of generalized state machine replication and finally our protocol.

[1] The connection has been symmetric so far: whatever one has proved using pure algorithmic implementation arguments had the analogue in algebraic-topology.

2 State Machine Replication: The Classics

2.1 Model

We assume here that processes can exchange information by reading and writing from shared memory cells, as well as agree on common decisions. More precisely, we assume a basic *read-write* shared memory model augmented with *consensus* objects [1].

Processes can be *correct*, in which case they execute an infinite number of steps of the algorithm assigned to them, or they *crash* and stop any activity. We consider an *asynchronous* system, meaning we make no assumption neither on process relative speeds nor on the time needed to access shared read-write memory cells or consensus objects.

The way consensus is used is simple: processes propose inputs and get back outputs such that the following properties are satisfied.

1. *Validity:* any output is the input of some process.
2. *Agreement:* all outputs are the same;
3. *Progress:* any correct process that proposes an input gets back an output;

2.2 Protocol

The algorithm underlying state machine replication is depicted in Figure 1. It is *round-based* as we will explain below.

Every process maintains locally a copy of the state machine as well as an ordered list of commands, denoted respectively *sm* and *comList* in Figure 1. The state machine is deterministic: the same command executed on different copies of the state machine in the same state, leads to copies that are also in the same state .

The processes typically have different lists of commands, say requests coming from different users of a web service modeled by the state machine. For the sake of presentation simplicity, we assume here that every process has an infinite such list of commands. A process picks one command at a time from its list; we also say that the process *issues* the command. As we will explain, the process does not issue the next command until it managed to execute the previous command on its state machine. Of course, the challenge for the protocol is to execute commands on the various copies of the state machine in the same order. This is where the consensus abstraction comes to play.

Consensus objects form a list, denoted by *ConsList* in Figure 1, and exactly one object of the list is used in each *round* of the protocol. Processes go round-by-round, incrementally, in each round proposing a command to the consensus object of the round and executing the command returned by that consensus object. Crucial to the correctness of the protocol lies the very fact that, in any given round, the consensus instance used by all the processes is the same shared object.

Basically, every process p proposes the next command it wants executed to the next consensus object. This, in turn, returns a command, not necessarily

```
local data structures:
1  sm                                          (* a copy of the state machine *)
2  comList                                     (* a list of commands *)
3  passed := true      (* determines if the process executed its previous command *)

shared data structure:
4  ConsList                                    (* a list of shared consensus objects *)

forever do:
5  if passed then com1 := comList.next()       (* pick the next command *)
6  cons := ConsList.next()                     (* pick the next consensus object *)
7  com2 := cons.propose(com1)                  (* agree on the next command *)
8  sm.execute(com2)                            (* execute the agreed upon command *)
9  if com2 = com1 then passed := true
10 else passed := false                        (* test if own command passed *)
```

Fig. 1. State machine replication

that proposed by p, but one proposed by at least some process. The command returned to a process p is then executed by p on its state machine: we simply say that p *executes* the command. To ensure that every process executes the commands in their original order, no process issues its next command unless it has executed its previous one.

2.3 Correctness

The correctness of the protocol of Figure 1 lies on four observations.

1. *Validity:* If a process q executes command c, then c was issued by some process p and q has executed every command issued by p before c. This follows from the facts that (a) a consensus object returns one of the inputs proposed (*validity* property of consensus), i.e., one of the commands issued by a process and (b) a process does not issue a new command unless it has executed the previous one it issued.
2. *Ordering:* If a process executes command c without having executed command c', then no process executes c' without having executed c. This follows from the facts that (a) the processes execute the commands output by the consensus objects, (b) the consensus objects are invoked by the processes in the same order and (c) each such object returns the same command to all processes (*agreement* property of consensus).
3. *Progress:* Every correct process executes an infinite number of commands on the state machine. This follows from the facts that (a) there is no wait statement in the algorithm and (b) every invocation to consensus by a correct process returns a command to that process (*progress* property of consensus).

It is important to notice at this point that this simple protocol does not guarantee fairness. Consensus objects could always return the commands proposed by the

same process, i.e., there is no obligation for a consensus object to be fair with respect to the processes of which it selects the input. Fairness could however easily be ensured by having the processes help each other. Namely, when a process issues a command, it first writes it in shared memory before proposing it to consensus. Processes would now propose sets of commands (their own and those of others) to consensus; accordingly, a consensus would return a set; the set would be the same at all processes which would execute the commands in the same deterministic order. For presentation simplicity, we omit fairness and helping.

3 Generalized State Machine Replication

Basically, with consensus, a state machine can be replicated over any number of processes and made highly available to all those processes. This makes of consensus a *universal* abstraction for any sequential service can be modeled as a state machine accessed by any number of processes. In a sense, with consensus, computing on several distributed computers is reduced to computing on a single, highly available, one.

In the following, we generalize this idea to show that, with k-set consensus, we can replicate k state machines one of which at least one is highly available. (The one that remains highly available is unknown in advance, for otherwise this would boil down to classical state machine replication.) In some sense, we show that k-set consensus is k-*universal*. We first give below the model underlying general state machine replication, then we define the properties we seek it to ensure before diving into the details of our protocol.

3.1 Properties

Here, we assume k state machines replicated over all processes of the system. The processes have each at their disposal a list of k-vectors of commands that they *issue* and seek to *execute* on their local copies of the k state machines: a command issued at entry j of a vector is to be executed on machine sm$[j]$. Again, for presentation simplicity, we assume the lists of commands are infinite.

A generalized state machine replication protocol satisfies the following properties:

1. *Validity:* If a process q executes command c on state machine sm$[i]$, then c was issued by some process p at entry i (of p's command vector), and q has executed every command issued by p before c at entry i.
2. *Ordering:* If a process executes command c on state machine sm$[i]$ without having executed command c' on sm$[i]$, then no process executes c' without having executed c on sm$[i]$.
3. *Progress:* There is at least one state machine sm$[i]$ on which every correct process executes an infinite number of commands.

It is easy to see that for the case where $k = 1$, these properties correspond exactly to those of state machine replication.

3.2 K-set Consensus

We assume here a standard read-write memory but now augmented with the k-set consensus abstraction [5]. We assume k-set consensus in its *vector* form [3]: It takes as input a k-vector of non-nil values, and returns, to each process, a k-vector composed of nil values and exactly one non-nil value among those proposed. We simply call this abstraction the *consensus vector*. It ensures the following properties:

1. *Validity:* any non-nil value returned at some entry i of an output vector is the input of some process at entry i of an input vector.
2. *Agreement:* Any two non-nil values returned to any two processes at the same entry of the output vectors are the same.
3. *Progress:* Every correct process that proposes an input (vector) gets back an output (vector) and every output contains exactly one non-nil value.

It is important to notice that the agreement property above does not prevent one process from getting a non-nil value returned at entry i and nil at entry j, whereas another process is getting some non-nil value at entry j and nil at entry i.

3.3 From 1 to k

To get a sense of the technical difficulty behind our generalization, consider first a naive protocol resulting from (a) replacing, in Figure 1, the consensus abstraction with the consensus vector one, and (b) having, in every round r, a process p executes on state machine sm[i] the command obtained at position i from the consensus vector, if any, i.e., if p obtains nil at position i in r, then p does simply not execute anything on state machine sm[i] in round r.

Clearly, such a protocol would guarantee liveness (progress): at least one state machine will remain highly available since the consensus vector will return at least one non-nil value and at least one command will be executed in every round. Yet, safety (ordering) will be violated as we illustrate now through a simple two-round execution of this naive protocol.

- Round 1. Assume p obtains a command c at position 1 (after proposing its initial command vector): p will then accordingly execute c on sm[1]. In the meantime, assume process q obtains a command $c' \neq c$ at position 2 and accordingly executes c' on sm[2].
- Round 2. Assume p obtains a command at position 2 and accordingly executes that command on its state machine sm[2]. This would violate ordering for p ignores that q already executed c' on sm[2] in round 1.

Intuitively, the issue should be sorted out by having every process announce what command it has executed before proceeding to the next round: say q would need to notify p that q has executed c' on sm[2] in round 1. This notification is not trivial for it needs to be synchronized with the action where p needs itself to execute a command on sm[1].

To sort out this issue, *adopt-commit* objects [7] come in handy. These can be implemented in a standard asynchronous read-write memory. We recall below the specification of such objects before explaining how they are used in our context.

3.4 Vectors of Adopt-Commit Objects

The specification of an adopt-commit object is as follows. Every process proposes an input value to such an object and obtains an output value, either in a *committed* or *adopted* status. (One could model such an output as a pair, combining a value and a bit depicting the status committed or adopted of that value). The following properties are satisfied:

1. *Validity:* The output value of any process is an input value of some process.
2. *Agreement:* If a committed value is returned to a process, then no different output value (committed or adopted) can be returned to any other process.
3. *Progress:* Every correct process that proposes an input value obtains an output value.
4. *Commitment:* If no two input values are different, then no output value can be adopted. (It is necessarily committed).

We use a vector of adopt-commit objects at each round, and this vector acts as a synchronization *filter* through which processes go, after passing the consensus vector and before actually executing commands on their state machines. Each process, after obtaining an output from the consensus vector, goes through the vector of adopt-commit objects. (In a specific order we explain below). In short, a process only executes commands that are committed. Those adopted are kept for next round.

3.5 Protocol

Our generalized state machine replication protocol is depicted in Figure 2. We denote the list of consensus vectors by *ConsVectList*, the list of adopt-commit vectors by *AConsVectList* and the list of vectors of commands available to a process by *comVectList*. A process can pick the next element in a list using function *next()* and also recall the last element picked in a list using function *current()*. Processes do not add items in those lists during the execution of the protocol.

The protocol proceeds in rounds. In every round, the initial vector of commands is denoted by *comVect*, the one resulting from the vector of consensus objects is denoted by *comVect1* (this one might contain nil values) and the one resulting from the vector of adopt-commit objects is denoted by *comVect2* (this one contains values in an adopted or committed status). If the latter vector returns a command that is committed (resp. adopted) to a process p, we say that p *commits* (resp. *adopts*) the command.

```
local data structures:
1  smVect[]                                          (* a vector of state machines *)
2  comVectList                                       (* a list of command vectors *)
3  for j := 1 to k do: comVect[j] :=                 (* pick the first command vector *)
comVectList[j].next()

shared data structures:
4  ConsVectList                                      (* a list of consensus vectors *)
5  AConsVectList                                     (* a list of adopt-commit vectors *)

forever do:
6  consVect := ConsVectList.next()                   (* pick the next consensus vector *)
7  comVect1 := consVect.propose(comVect);            (* decide a new vector of commands *)
8  aconsVect := AConsVectList.next()                 (* pick the next adopt-commit vector *)

9  for i := 1 to k do:
10   if comVect1[i] ≠ nil then:
11     comVect2[i] := aconsVect[i].propose(comVect1[i])   (* exploit success first *)

12 for i := 1 to k do:
13   if comVect1[i] = nil then:                      (* try to commit old commands *)
14     comVect2[i] := aconsVect[i].propose(comVect[i])

15 for i := 1 to k do:
16   if older(comVect2[i],comVect[i]) then sm[i].execute(comVect[i])  (* catch-up *)
                                                     (* keep the command for next round *)
17   if adopted(comVect2[i]) then comVect[i] := comVect2[i]
18   else
19     sm[i].execute(comVect2[i])
20     if comVect2[i] := comVectList[i].current()
21       then comVect[i] := comVectList[i].next()
22       else comVect[i] := comVectList[i].current()
23     add(comVect[i],comVect2[i])                   (* remember the committed command *)
```

Fig. 2. Generalized state machine replication

For presentation simplicity, we assume that a process can test if a command was adopted simply using a function *adopted(c)*, a process can encode in a command c' the fact it has committed c, simply by writing $add(c', c)$, and the process can check that fact by simply testing if $older(c, c')$.

Two main ideas underly our generalized state machine replication protocol (Figure 2):

1. *Exploit success first.* To ensure *liveness*, a process p, at round r, accesses first the adopt-commit object corresponding to the non-nil value (i.e., the command) returned by the consensus vector at r (lines 10–11 in Figure 2). Subsequently, p proceeds to the rest of the entries at which is was returned nil and proposes the original commands to the consensus vector (lines 13–14 in Figure 2).

This ensures that at least one process will commit a command in every round. Indeed, for an adopt-commit object not to commit a command, it has to be concurrently invoked with at least two distinct values. The first process p to return from any of the adopt-commit object, by virtue of being first, must commit the command. No process q can prevent p from committing by proposing a distinct command concurrent with p, as then, q's command was not returned there, and q already went through the adopt-commit object of its returned command, contradicting the fact that p was the first to return from any adopt-commit object.

2. *Remember commitments.* To ensure *safety*, a process p might need to execute two commands on the same machine in the same round. A process p might indeed adopt a command c in round r for entry i, then commit another command c' in round $r+1$ for that same entry i. This might happen if another process q committed c at r and then moved to propose and commit c' at $r+1$. In this case, p should execute c and then c', both in round $r+1$. Should p execute c' without having executed c, p would violate safety.
In our protocol, when q commits a command c in round r, then moves to round $r+1$ with a command c', q encodes in c' the fact that c was committed before c' (line 23 in Figure 2): hence, in round $r+1$, p will decode that information from c', then execute c before c' (line 16 in in Figure 2). In fact, p executes c even if it only adopts c' in round $r+1$.
It is important to notice here that c cannot "get lost" as every process that did not commit c in round r must have adopted c at round r. Hence, all proposed values to the adopt-commit object at entry i at round $r+1$, which are not c, are commands which encode the very fact that c has been committed at round r.

4 Correctness

Theorem 1. *If a process q executes command c on state machine $sm[i]$, then c was issued by some process p at entry i (of its command vector), and q has executed every command issued by p before c at entry i.*

Proof. There are exactly two places of the algorithm of Figure 2 where a process p can execute a command c on its state machine $\mathrm{sm}[i]$: at line 19 and line 16. For p to execute a command c on $\mathrm{sm}[i]$ at line 19, p must have obtained c from an adopt-commit object at entry i ($comVect2[i]$). For p to execute a command c on $\mathrm{sm}[i]$ at line 16, some process p' must have obtained c from an adopt-commit object at entry i ($comVect2[i]$) and added it to the command vector at entry i (line 23). In both cases, some process p'' must have proposed c to an adopt-commit object at entry i, and hence must have obtained c from a consensus vector at entry i. In turn, some process p''' must have proposed c to that vector at entry i and hence must have issued the command at entry c. It remains to show now that p has executed on $\mathrm{sm}[i]$ every command c' issued by p before c at entry i. The only place where a process p issues a new command at entry i in the algorithm of Figure 2 is at line 20. By the preliminary test performed in

that line, this happens only if the command executed by p at entry i was issued by p for rank i. Hence, p cannot execute a new command at entry i which is issued by p, without having executed the previous command issued by p.

Theorem 2. *If a process executes command c on state machine $sm[i]$ without having executed command c' on $sm[i]$, then no process executes c' without having executed c on $sm[i]$.*

Proof. Assume a process p executes, on its state machine sm$[i]$, command c at some round r. If this happens at line 16, then some process q has committed c through an adopt-commit object $acons Vect[i]$ in round $r-1$ (line 18). Assume furthermore that process p did not execute c' before c. This means that no adopt-commit object $acons Vect[i]$ has returned c' in a committed status at round $r' < r-1$. Assume now that p executes on sm$[i]$, command c at round r at line 19. This means that p has committed c through an adopt-commit object $acons Vect[i]$ in round r (line 18). Assume furthermore that process p did not execute c' before c. This means that no adopt-commit object $acons Vect[i]$ has returned c' in a committed status at round $r' < r$. Hence, no process q can execute c' without having executed c on state machine sm$[i]$.

Lemma 1. *If a process p commits command c in round r on state machine $sm[i]$, then every process which finishes round $r+1$ executes c on state machine $sm[i]$.*

Proof. Assume process p commits command c in round r on state machine sm$[i]$. By the specification of adopt-commit, aconsVect$[i]$ returns command c (either in a committed or adopted status) to all processes that invoked it in round r. Hence, all processes which start round $r+1$ either (a) executed c on sm$[i]$ in round r and start round $r+1$ with a command c' such that c' encoded the commitment of c (line 23), or (b) start round $r+1$ with command c itself (line 17). In both cases, any process that did not execute c in round r will, in round $r+1$, either commit c and execute it (line 19) or learn about c having been committed and execute it (line 16).

Theorem 3. *An infinite number of commands are executed on at least one state machine at all correct processes.*

Proof. Assume at least one process is correct. Assume by contradiction that there is a round at which no process executes a command on a state machine. This means that no adopt-commit object returns a committed command. Given that the protocol of Figure 2 has no wait statement, every adopt-commit object must have had two different concurrently proposed values. This means that all processes obtained different values from the consensus vector. Consequently, all processes started at different adopt-commit objects. This is in contradiction with the fact that every adopt-commit object has two different concurrent proposals. Hence, at least one process commits a command on at least one machine in every round. This follows from the order according to which processes access adopt-commit objects. By Lemma 1, all correct processes execute a command on at

least one machine every two rounds. Hence, there is at least one state machine on which all correct processes execute an infinite sequence of commands.

5 Concluding Remarks

When k-set consensus was introduced [5], as creative feat as it was, it was formulated in the "wrong" way. The question "what the analogue of (consensus) state machine replication is?" could not be imagined, as k-set referred to multiple values. When [3] equated k-set consensus with vector consensus, the question of generalizing state machine replication started to make sense. In retrospect, generalized state machines replication as presented here is so simple, that it begs the question "why so long?"

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
2. Borowsky, E., Gafni, E.: Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. In: STOC (1993)
3. Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., Travers, C.: The k-simultaneous consensus problem. Distributed Computing 22(3), 185–195 (2010)
4. Lamport, L.: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
5. Chaudhuri, S.: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Information and Computation 105, 132–158 (1993)
6. Fischer, M., Lynch, N., Paterson, M.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
7. Gafni, E.: Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. In: PODC (1998)
8. Herlihy, M.: Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems 11(1), 124–149 (1991)
9. Herlihy, M.P., Shavit, N.: The Topological Structure of Asynchronous Computability. Journal of the ACM 46(6), 858–923 (1999)
10. Saks, M., Zaharoglou, F.: Wait-Free k-set consensus is Impossible: The Topology of Public Knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
11. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1987)