Model Checking Distributed Consensus

Michele Tatarek

University of Genova

m.tatarek@gmail.com

Joint work with Giorgio Delzanno and Riccardo Traverso

July 22, 2014

Overview

Motivations

- Consensus is one of the most difficult problems considered in the area of distributed algorithms
- There are very few basic algorithms for distributed consensus, most of them based on the Paxos metaphor proposed by Lamport
- Paxos is defined for asynchronous systems where the number of interleavings grows exponentially with the number of processes
- Formal methods can help in designing correct solutions
- We apply PROMELA/Spin to validate two case-studies: Paxos and Raft

Distributed consensus: definition

Assumptions:

- Distributed processes communicate using asynchronous channels.
- Initially every process proposes a value.

Goal: Eventually, every process must agree on the **same** value.

Distributed consensus: definition

Assumptions:

- Distributed processes communicate using asynchronous channels.
- Initially every process proposes a value.

Goal: Eventually, every process must agree on the **same** value.

The FLP impossibility result

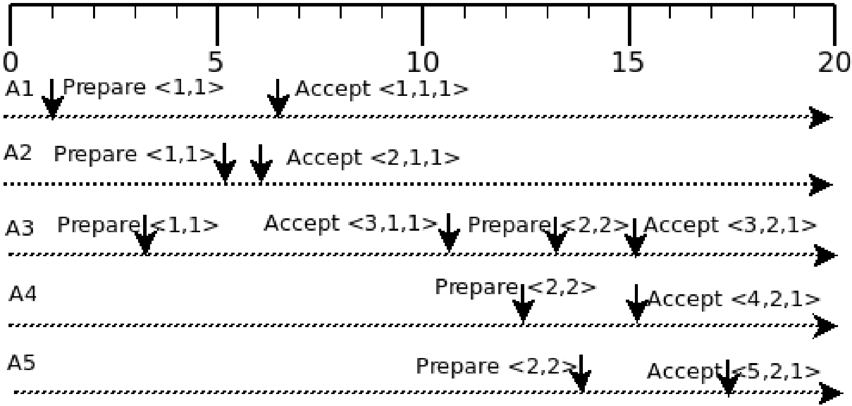
- Solving consensus in an asynchronous system with at least one faulty process is impossible!
- Intuitively is impossible to tell a very slow process from a crashed one.

One partial solution: the Paxos algorithm

- We assume fail-stop crashing model.
- Possibly non terminating.
- Round associated with messages used as timestamps to identify old requests.
- three-phase leaderless algorithm divided into three roles.
  - Proposers that can propose values for consensus.
  - Acceptors that vote those values.
  - Learners that eventually learn the chosen value.

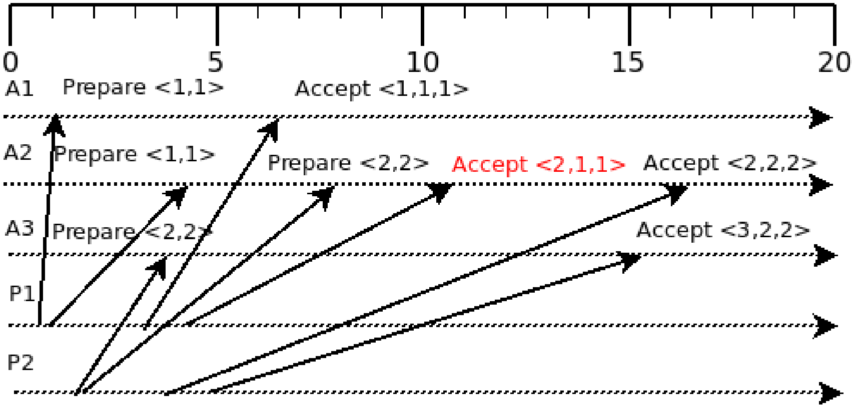
The acceptors point of view: received messages

If a previous value was already chosen, 2nd proposer will find and use it. Acceptors will update their round number accordingly.



A worse case: two proposers competing

Previous value is not yet chosen, 2nd proposer does not see it.
New proposer chooses its own value, older proposals are ignored.



Expected properties

- When a value is chosen by a learner, then no other value has been chosen/accepted in previous rounds of the protocol.
- Whenever a value is chosen by the learner, any successive voting always selects the same value but with larger round identifiers.
- Proposers values comes into play only if all values received by acceptors are undefined, i.e. there are no circulating values detected.

Overview

The PROMELA language

- A PROtocol MEta LAnguage to specify multi-threaded and distributed programs.
- Input language for the SPIN model checker that features user defined data structures, thread definitions, channels, and CSP-like guarded commands
- Processes and requirements in temporal logic are compiled into Büchi automata

System Specification

Proposers filled with initial round numbers and proposed values.
Acceptors identified by ascending numbers for addressing.
Only one value is proposed by each proposer for simplification.

```
init
{ atomic{
   run proposer(1,1); run proposer(2,2); ...
   run acceptor(0); run acceptor(1); ...
   run learner();
  };
 }
```

Definitions and Data Structures

Constants

```
#define ACCEPTORS 3
#define PROPOSERS 5
#define MAJ (ACCEPTORS/2+1)// majority
#define MAX (ACCEPTORS*PROPOSERS)
```

Channels

```
chan prepare = [MAX] of { byte, byte };
chan accept  = [MAX] of { byte, byte, short };
chan promise = [MAX] of { mex };
chan learn   = [MAX] of { short, short, short };
```

The proposer model

```
proctype proposer(int round; short myval) {
    short hr = -1, hval = -1, tmp;
    short h, r, v;
    byte count;
    bprepare(round);
    do
     :: rec_p(round,count,h,v,hr,hval);
          % on promise increment count

     :: send_a(round,count,hval,myval,tmp);
          % check majority on count and send accept
    od }
```

The acceptor model: accept

```
proctype acceptor(int i) {
    short rnd = -1, vrnd = -1, vval = -1;
    do
      :: rec_p(i,rnd,vrnd,vval);
            % on prepare:
            %   send promise and update rnd,vrnd,vval

      :: rec_a(i,rnd,vrnd,vval);
            % on accept:
            %  update vrnd, vval and send learn
      od }
```

The learner model

```
proctype learner() {
    byte mcount[MAX];

    do
      :: rec_l(mcount);
         % on learn:
           % check majority on mcount[rnd]
    od
}
```

Encoding Correctness

```
read_l(id,rnd,lval,lastval,mcount)=
 d_step {
   learn??id,rnd,lval ->
   if
    :: mcount[rnd-1] < MAJ -> mcount[rnd-1]++;
    :: else  fi;
   if
    ::mcount[rnd-1] >= MAJ   ->
      if :: (lastval >= 0 && lastval != lval) ->
               assert(false);
         :: (lastval == -1) -> lastval = lval;
         :: else fi
    :: else fi;
   id = 0; rnd = 0; lval = 0
  }
```

Overview

Cut-off theorems

Theorem

The safety property stated before holds for the model with multiple learners if and only if there exist no execution that violates the safety assertion in the model with a single instance of the `learner` *proctype.*

Learners just observe the situation in a single loop...

Cut-off theorems

Theorem

The safety property stated before holds for the model with multiple learners if and only if there exist no execution that violates the safety assertion in the model with a single instance of the `learner` *proctype.*

Learners just observe the situation in a single loop...

Theorem

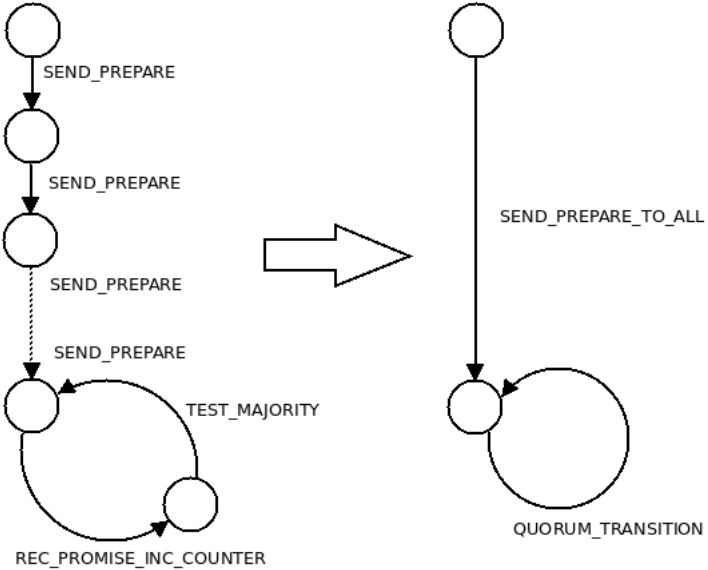
If for a given value of the parameter MAJ the safety property stated before holds for two proposers (with distinct values), then it holds for any number of proposers.

By induction on the round number, only the value associated with the latest highest round number is kept.

Normalisations

- Messages are kept ordered in channels by using ordered send (!!) and random receive (??) operators.
- Variables are reset after use to bring the automaton to its original state and reduce the state-space.

The proposer automaton revisited



The updated proposer model

The new proposer

```
proctype proposer(short crnd; short myval) {
 short aux, hr = -1, hv = -1;
 short rnd;
 short prnd,pval;
 byte count=0,i=0;
 mex pr;

 d_step{ bprepare(crnd); }/*automaton-level atomicity*
 do
 :: qt(i,pr,count,hr,hv,myval,crnd,aux);
 od
}
```

Quorum Transition

```
qt(i,pr,count,hr,hv,myval,crnd,aux) =
  atomic{
   occ(i,pr,count,hr,hv,crnd);
   test(count,hr,hv,myval,crnd,aux);
   hv= -1;   hr = -1; count =0; aux=0;
  }
```

Channel Inspection

```
occ(i, pr, count, hr, hv, crnd) =
  d_step{
   do
   i=0;
   ::  i < len(promise) ->
      promise?pr;  promise!pr;
      if  ::pr.rnd==crnd-> count++;
             if
             ::pr.prnd>hr-> hr=pr.prnd;  hv=pr.pval;
             ::else  fi;
          ::else  fi;
      i++;
::  else ->/*vars  reset*/
      pr.prnd=0;  pr.pval=0;  pr.rnd=0;  i=0;
      break;
   od;
   }
```

Majority Test

```
test(count,hr,hv,myval,crnd,aux) =
  if
    ::count>=MAJ ->
      aux=(hr<0 ->myval : hv);
      baccept(crnd,aux);
      break;
    ::else
   fi;
```

Read-only Prepare Channel

Channels are treated as a common blackboard messages are not removed after reading to model duplication.
New Acceptor:

```
proctype acceptor(int id) {
 short crnd = -1, prnd = -1, pval = -1;
 short aval,rnd;
 do
   ::d_step {
      prepare??<eval(id),rnd> ->
      if
       :: (rnd>crnd)  -> crnd=rnd;
       :: else
     fi;
     rnd = 0
   }
   ...
 }
```

The two model are equivalent w.r.t. exchanged messages

- A sequence of a reading followed by a sending in a proposers can be permuted since the communication model is asynchronous.
- For the same reason, we can group all sends issued by proposers in a single atomic broadcast step.
- Receive operations cannot be grouped together in the first model because they alter the internal proposer state.
- The effect of a single receive operation followed by an update and a test on *count* is the same as a quorum transition applied directly on the channel.

Overview

Experimental analysis of the first model: some results

First Model

Prop	Acc	Max	Maj	Time (sec)	App	States	Unsafe	OutOfMem
2	2	4	2	0.006		790		
2	3	6	2	0.285		65091		
2	4	8	3	5.97		744224		
2	5	3	2	2.55		377295	X	
2	5	10	3	61.5		5186311		X
2	6	12	3	35	X	$7*10^6$	X	

Optimized Model

Prop	Acc	Max	Maj	Time (sec)	App	States	Unsafe	OutOfMem
2	2	4	2	$<$0.01		61		
2	3	6	2	$<$0.01		926		
2	4	8	3	0.02		4421		
2	5	10	3	0.4		91956		
2	6	12	4	2.74		473261		
2	7	14	4	71.9		9762358		
2	8	16	5	799		28208534		X
2	8	16	5	52.3	X	7525860		
2	8	16	4	60.3	X	8550176		

And more Instances can be Fully Verified

Prop	Acc	Max	Maj	Time (sec)	App	States	Unsafe	OutOfMem
5	3	15	2	>3600		$>1.22*10^8$		X
5	3	15	2	50	X	7949467		
5	3	15	1	2.17	X	340445	X	
6	2	12	2	25.9		4397176		
7	2	14	2	558		71298752		
8	2	16	2	2370		$1.4984*10^8$		X
8	2	16	2	70.4	X	8045888		
8	2	16	1	17.4	X	2189799	X	

Overview

Conclusions

- Modelling for an automaton-based model checker requires having to think about the automaton underlying the model.
- Spin careful parameters setting along with accurate abstractions are the keys to improve experimental results.
- Induction over execution paths can possibly prove more cut-off theorems (acceptors? Under consideration).
- Other consensus algorithms can possibly be analysed with the same techniques (Raft? Work in progress)

References and...questions?

Leslie Lamport (1998)
The Part-Time Parliament
ACM Transactions on Computer Science 16(2), 133 – 169.

Leslie Lamport (2001)
Paxos made simple
SIGACT News 32(4), 51 – 58.

D. Ongaro and J. Ousterhout (2014)
In Search of an Understandable Consensus Algorithm
In proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14).

G.Delzanno, M.Tatarek and R. Traverso (2014)
Model Checking Paxos in Spin
In proceedings of the 5th International Symposium on Games, Automata, Logics and Formal Verification(GandALF '14), to appear.