Randomized Consensus

Sacrificing determinism

Randomization

The triumph of randomization?

Well, at least for distributed computations!

- No deterministic 1-crash-resilient solution to consensus
- t-resilient randomized solution to consensus for crash failures ($n \geq 2t+1$)
- Randomized solutions for consensus exist even for byzantine failures

Adversary

When designing fault-tolerant algorithms, we often assume the presence of a strong adversary

- It knows the local state of each process
- It knows the content of all sent messages
- It can select the next process to take a step
- It can select the message the process will receive

- It cannot prevent a message from being eventually received
- It cannot make more than t processes fail

Another advantage of free choice: completely asynchronous agreement protocols

Ben-Or, M. (1983)

Proceedings of the Second ACM Symposium on Principles of Distributed Computing, pp. 27-30.

Local coin

Aspnes, J. (2020). Notes on Randomized Algorithms
http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf

Ben Or’s randomized consensus algorithm

Soon after the FLP impossibility results appeared, people tried to find a way to circumvent it

Ben-Or gave the first algorithm to achieve consensus with probabilistic termination in an asynchronous model

The algorithm is correct if no more than t crash occur with $n > 2t$

The expected time to decide may be exponential

The randomized consensus problem

Every process has some initial value $x_p \in \{0, 1\}$, and must decide on a value such that the following properties hold:

- Agreement - No two processes decide differently
- Validity - If any process decides v, then v is the initial value of some process
- Termination - **With probability 1**, every correct process eventually decides

Model of the system

Set of n processes

At most **t < n/2** processes may crash (may stop to take steps)

Asynchronous environment

Reliable communication channels

Each process has access to a local coin-flip operation

when a process tosses its coin, it obtains 0 or 1 with probability ½

Asynchronous rounds and phases

The algorithm operates in rounds, each round has two phases

Reporting (voting) phase

each process P transmits its value x_p, and waits to hear from other processes

Decision (ratification) phase

if majority found, take its value; otherwise flip a coin to change the local value x_p

Process P: initial value x_p

```
for (r:=1; true; r:=r+1)                                        //  infinite rounds
        // Phase 1
        send the message (1, r, x_p) to all processes
        wait until n - t messages of type (1, r, *) are received  // including own value
        if k > n/2 messages have the same value v                 // majority
                send the message (2, r, v, D) to all processes    // positive intimation
        else
                send the message (2, r, ?) to all processes       // negative intimation
        // Phase 2
        wait until n - t messages of type (2, r, *)  arrive
        if there are k ≥ 1 D-messages (2, r, v, D)                // decide messages
                x_p := v
                if there are k > t  D-messages
                        decide v
        else
                x_p := fairCoinFlip()
```

Crash agreement
n > 2t

Phase results

At the end of phase 1 a process P

- proposes v if received a strict majority of reports v
- proposes ? otherwise

At the end of phase 2 a process P

- decides v if received $k \geq t+1$ proposals v $(v \neq ?)$
- adopts v for x_p if received $1 \leq k \leq t$ proposals v $(v \neq ?)$
- chooses a random value for x_p otherwise

The idea

If someone detected majority, then switch to the majority value

If enough processes detected majority, then decide

Otherwise, flip a coin // eventually, a majority of correct processes will flip in the same way

Agreement

1. At most one value can receive a majority of votes in the first phase of a round, so for any two messages (2; r; v; D) and (2; r; v’; D), v = v’

2. If some process sees t+1 (2; r; v; D) messages, then every correct process sees at least one (2; r; v; D) message

3. If every process sees a (2; r; v; D) message, every process votes for v in the first phase of round r+1 and every process that has not already decided decides v in round r+1

Validity

If all processes vote for their common input v in round 1, then all processes send (2; r; v; D) and decide v in the second stage of round 1

If there is no common input, then the decide value will be a value proposed by some process

Termination

Termination follows because if no process decides in round r, then each process either chooses its new preference based on the majority value v in a (2; r; v; D) message or it chooses its new preference randomly, and there is a nonzero probability that all of these random choices equal the unique first-phase majority value (or each other, if there is no majority value)

Aguilera, M., & Toueg, S. (2012). The correctness proof of Ben-Or's randomized consensus algorithm, *Distributed Computing*, Vol. 25, No. 5, pp. 371–381.

Termination

Lucky round - one in which processes toss coins that cause the termination of the algorithm (no matter what the adversary does)

Worst case - all n processes flip to the same side of the coin

$p = P(\text{lucky round}) \geq 2(1/2^n) = 1/2^{n-1}$

Termination

If the probability of success on a round (trial) is p and X is the number of rounds needed to get one success, then

- $P(X=k) = (1-p)^{k-1}\, p$ for k = 1,2,3, ...

  mean = 1/p

  $E(k) = 2^{n-1}$

  (shifted) geometric distribution

- $P(X \leq k) = 1 - (1-p)^k$ (cumulative) distribution function
- $P(X>k) = (1-p)^k$ $\lim_{k \to \infty} P(X > k) = 0$

Process P: Initial value x_p

step 0: r := 1

step 1: send the message (1, r, x_p) to all processes

step 2: wait until n - t messages of type (1, r, *) are received

if k> (n+t)/2 messages have the same value v

send the message (2, r, v, D) to all processes

else

send the message (2, r, ?) to all processes

step 3: wait until n - t messages of type (2, r, *) arrive

if there are k ≥ t+1 D-messages (2, *r, v, D*)

x_p := v

if there are k > (n + t)/2 D-messages

decide v

else

x_p := 1 or 0 each with probability ½

step 4: r := r + 1 and goto step 1

Bizantine agreement

n > 5t

Randomized byzantine generals

Rabin, M. (1983).

Proceedings of Twenty-Fourth IEEE Annual Symposium on Foundations of Computer Science, pp. 403-409.

Shared coin

Michael O. Rabin

1976 Turing award - For his paper "Finite Automata and Their Decision Problem," which introduced the idea of **nondeterministic** machines, which has proved to be an enormously valuable concept. This classic paper has been a continuous source of inspiration for subsequent work in this field.

Paper and award with Dana S. Scott

Why randomized

Achieve Byzantine Agreement with $n > 10t$.

Two Byzantine Agreement Protocols (BAP)

1. Termination in a fixed number R of rounds, but with a probability 2^{-R} of **error**.
2. Termination in an **expected** number of four rounds.

Authentication

Every message is authenticated by a digital signature.

- This requires that the processes in the system be supplied in advance, by a non-faulty dealer, with a directory of public keys.

A salient novel feature of the protocols is the use of randomly chosen secrets which are *shared* by the processes.

- This also requires that the processes be supplied in advance with certain information by the non-faulty dealer.

Shared coin = sequence of random 0-1 values, prepared and distributed by the trusted dealer.

Terminology

Let G_i, $1 \leq i \leq n$, be processes (the “Generals").

Every G_i has a message M_i, called G_i’s initial message.

To reach agreement, each G_i executes a program P_i called the Agreement Protocol.

P_i involves exchanging messages with other processes G_j, deciding what value to adopt as the common message, and deciding when to stop.

Terminology

If all the proper (non faulty) processes have the same initial message then the system will be called proper, otherwise faulty.

Each process G_i has a variable message(i), holding at any time G_i's current version of the message.

This variable holds, at the end of G_i's execution of P_i, G_i's version of the message, which will be one of the initial messages M_j or else the value "system faulty".

Byzantine Agreement Protocol

Processes G_i in a set are said to reach an agreement about the value of the message if each G_i stop and at the end of the execution of the protocol, message(i) = message(j) for all i,j.

We say that the proper processes have reached Byzantine Agreement if

1. All the proper processes reach agreement.
2. If all proper G_i have the same initial message M_i=M then the proper processes agree on M as the value of the message.

Phases

A phase in the computation of a proper G_i is the time interval between G_i's request for information from k other processes and the receipt of at least k - t replies.

The computation time required by G_i, once the information is received, is also included in the phase.

We assume that each process G_i has a local phase clock p(i) and that P_i assigns p(i) := p(i) + 1 at the end of each phase.

Lottery - global coin tosses

Coin tosses are precomputed by the trusted dealer who splits the results of each coin toss so that t+1 processors can determine the result, but t processors have no information.

It is a distributed object that delivers the same sequence of random bits b_1, b_2, . . . to each process.

Shamir, A., How to share a secret, CACM, Vol. 22 (1979), pp. 612-613.
https://www.youtube.com/watch?v=iFY5SyY3IMQ

Lottery

The Lottery procedure admits a parameter k, so that Lottery(k) is the k-th lottery round.

It is a procedure by which the proper processors can agree on a randomly chosen $s_k = 0,1$.

First protocol

- Constant time — R rounds
- Bounded error solution — 2^{-R}

BAP

```
Procedure BAP;                          // for G_i
begin
        message(i) := M_i;              // G_i's initial message
        for k = i to k = R do
        begin
                Polling(k);
                Lottery(k);
                Decision
        end
end;
```

The value of R is determined by the desired reliability : $1 - 2^{-R}$

BAP

1. **Polling** : G_i polls the other processes on their value of the message.
2. **Lottery:** G_i decide on a common random bit s_k.
3. **Decision**: G_i determines, using s_k, whether to adopt the plurality candidate version of the message obtained through Polling as his current version of the message.

Polling

Procedure Polling(k); // for G_i

begin

send σ_i(message(i),k) to all;

collect incoming σ_j(message(j),k) until n - t values received;

temp := most popular k-message; // the plurality candidate

count := count of multiplicity of most popular k-message;

end;

Lottery

Procedure Lottery(k); // for G_i

begin

// ask for E_j^k from all;

send E_i^k to all;

wait until t different values of E_j^k have arrived;

compute s_k from E_i^k and the available E_j^k;

end;

Decision

```
Procedure Decision;                                   // for G_i
begin
        s : = s_k;                                    // current shared  secret
        if (s=0 and count ≥ n/2) or (s=1 and count ≥ n-2t)
                message(i) := temp
        else
                message(i) := "system faulty"
end;
```

Correctness

All the proper processes will end their execution of BAP.

At the end of BAP, for all proper G_i

if (the system is proper and the initial message is M)

message(i) = M

else

message(i) = “system faulty” // with probability at least $1-2^{-R}$

Second protocol

- Bounded expected time 4 rounds
- Errorles solution

ETBAP

```
Procedure ETBAP;                                    // expected time BAP for G_i
begin
        k := 1;
        message(i) :=M_i
        inround := true;
        while inround                               // halt signal
        begin
                Polling(k);
                Lottery(k);
                Decisionproof                       // increases k by 1 each time
        end
end;
```

Decisionproof

```
Procedure Decisionproof;                                    // for G_i
begin
        s :=s_k
        if (s=0 and count ≥ n/2) or (s=1 and count ≥ n-2t)
                message(i) := temp                          // v
        else
                message(i) := "system faulty"
        if (s =0 and count ≥ n-2t)
                send σ_i("agreement reached on v") to all G_j;
        k := k +1
end;
```

Closefinish

```
Procedure Closefinish;                                  // for G_i
begin
        repeat
                If received new σ_j("agreement reached on v")
                        send σ_j("agreement reached on v") to all
        until t+1 communications with the same v are counted;
        message(i) := v;
        inround := false                                // interrupt to stop ETBAP
end;
```