Computer Systems - Notes Week 9

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 6, 2022

Chapter 17: Byzantine Agreement

A node which can have arbitrary behavior is called **byzantine.** This includes “anything imaginable”, e.g. not sending any messages at all, or sending different and wrong messages to different neighbors, or lying about the input value.

Remarks:

- Byzantine behavior also includes collusion, i.e. all byzantine nodes are being controlled by the same adversary.
- We call non-byzantine nodes *correct* nodes.

Finding consensus as defined in chapter 16 in a system with byzantine nodes is called **byzantine agreement.** An algorithm is f-resilient if it still works correctly with f byzantine nodes.

17.1 Validity

For *any-input validity,* the decision value must be the input value of any node. For *correct-input validity,* the decision value must be the input value of a correct node. Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node.

If all correct nodes start with the same input v, then under *all-same validity* the decision must be v. If the input values are not binary, but for example from sensors that deliver values in $\mathbb{R}$, all-same validity is in most scenarios not really useful.

If the input values are orderable, e.g. $v \in \mathbb{R}$, byzantine outliers can be prevented by agreeing on a value close to the *median* of the correct input values, which we call **median validity.** How close the values is depends on the number of byzantine nodes f.

In the **synchronous model,** nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and so some local computation. For algorithms in the synchronous model, the **runtime** is simply the number of rounds from the start of the execution to its completion in the worst case.

17.2 How Many Byzantine Nodes?

```
# Algorithm 17.9: Byzantine Argeement with f = 1
1:  Code for node u, with input value x
    # Round 1
2:  Send tuple(u, x) to all other nodes
3:  Receive tuple(v, y) from all other nodes v
4:  Store all received tuple(v, y) in a set s_u
    # Round 2
5:  Send set S_u to all other nodes
6:  Receive sets S_v from all other nodes v
7:  T = set of tuple(v, y) seen in the last two sets S_v, including own S_u
8:  Let tuple(v, y) in T be the tuple with the smallest y
9:  Decide on value y
```

Byzantine nodes may not follow the protocol and send syntactically incorrect messages. Such messages can easily be detected and discarded. It is worse if byzantine nodes send syntactically correct messages, but with bogus content.

Lemma 17.11: If $n \geq 4$, all correct nodes have the same set T.

Theorem 17.11: Algorithm 17.9 reaches byzantine agreement if $n \geq 4$.

Algorithm 17.9 can be slightly modified to achieve all-same validity by choosing the smallest value that occurs at least twice. The idea of this algorithm can furthermore be generalized for any f and $n > 3f$. In the generalization, every node sends in every of $f + 1$ rounds all information it learned so far to all other nodes.

Theorem 17.12: Three nodes cannot reach byzantine agreement with all-same validity if one node among them is byzantine.

Theorem 17.13: A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.

17.3 The King Algorithm

```
# Algorithm 17.14: King Algorithm (for f < n/3)
x = my input value
for phase = 1 to f + 1 do:
    # Vote
    Broadcast value(x)
    # Propose
    if some value(y) received at least n-f times then:
        Broadcast propose(y)
    end if
    if some propose(z) received more than f times then:
        x = z
    end if
    # King
    Let node v_i be the predefined king of this phase i
    The king v_i broadcasts its current value w
    if received strictly less than n-f propose(y) then:
        x = w
    end if
end for
```

Lemma 17.15: Algorithm 17.14 fulfills the all-same validity. ***Lemma 17.16:*** If a correct node proposes x, no other correct node proposes y, with $y \neq x$, if $n > 3f$. ***Lemma 17.17:*** There is at least one phase with a correct king. ***Lemma 17.18:*** After a phase with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.

Theorem 17.19: Algorithm 17.14 solves byzantine agreement.

Algorithm 17.14 requires $f + 1$ predefined kings. We assume that the kings (and their order) are given. Finding the kings indeed would be a byzantine agreement task by itself, so this must be done before the execution of the King Algorithm.

17.4 Lower Bound on Number of Rounds

Theorem 17.20: A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f + 1$ rounds, if nodes decide for the minimum seen value.

17.5 Asynchronous Byzantine Agreement

```
# Algorithm 17.21: Asynchronous Byzantine Agreement (Ben-Or, for f < n/10)
x_u in {0, 1} <- input bit
round = 1
while true do:
```

```
4:      Broadcast propose(x_u, round)
5:      Wait until n-f propose messages of current round arrived
6:      if at least n/2 + 3f + 1 propose messages contain the same value x then:
7:          Broadcast propose(x, round+1)
8:          Decide for x and terminate
9:      else if at least n/2 + f + 1 propose messages contain the same value x then:
10:         x_u = x
11:     else:
12:         choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
13:     end if
14:     round = round + 1
15: end while
```

Lemma 17.22: Let a correct node choose value x in line 10, then no other correct node chooses value $y \neq x$ in line 10.

Theorem 17.23: Algorithm 17.21 solves binary byzantine agreement as previously defined for up to $f < n/10$ byzantine nodes.

This algorithm is a proof of concept that asynchronous byzantine agreement can be achieved. Unfortunately this algorithm is not useful in practice, because of its runtime.

17.6 Random Oracle and Bitstring

A **random oracle** is a trusted (non-byzantine) random source which can generate random values.

```
# Algorithm 17.25: Algorithm 17.21 with a Magic Random Oracle
1:  replace line 12 in Algorithm 17.21 by
2:  reutn c_i, where c_i is the i-th random bit by oracle
```

Theorem 17.26: Algorithm 17.25 plugged into Algorithm 17.21 solves asynchronous byzantine agreement in expected constant number of rounds.

Unfortunately, random oracles are a bit like pink fluffy unicorns: they do not really exist in the real world.

A **random bitstring** is a string of random binary values, known to all participating nodes when starting a protocol.

```
# Algorithm 17.28: Algorithm 17.21 with Random Bitstring
1:  Replace line 12 in Algorithm 17.21 by
2:  return b_i, where b_i is the i-th bit in common random bitstring
```

But is such a precomputed bitstring really random enough? We should be worried because of Theorem 16.14.

Theorem 17.29: If the scheduling is worst-case, Algorithm 17.28 plugged into Algorithm 17.21 does not terminate.

This theorem shows that a worst-case scheduler cannot be allowed to know the random bits of the future.

Chapter 18: Broadcast & Shared Coins

In Chapter 17 we have developed a fast solution for synchronous byzantine agreement (Algorithm 17.14), yet our asynchronous byzantine agreement solution (Algorithm 17.21) is still awfully slow. Some simple methods to speed up the algorithms did not work, mostly due to unrealistic assumptions.

Lemma 18.2: Algorithm 16.22 has exponential expected running time under worst-case scheduling.

The **blackboard** is a trusted authority which supports two operations. A node can *write* its message to the blackboard and a node can *read* all the values that have been written to the blackboard so far.

```
# Algorithm 18.4: Crash-Resilient Shared Coin with Blackboard (for node u)
1:  while true do:
2:      Choose new local coin c_u = +1 with probability 1/2, else c_u = -1
3:      Write c_u to the blackboard
```

```
    Set C = Read all coinflips on the blackboard
    if |C| >= n^2 then:
        return sin(sum(C))
    end if
end while
```

Theorem 18.5 (Central Limit Theorem): Let $\{X_1, X_2, ..., X_n\}$ be a sequence of independent random variables with $\Pr[X_i = -1] = \Pr[X_i = 1] = 1/2$ for all $i = 1, 2, ..., N$. Then for every positive real numbers z,

$$\lim_{N\to\infty} \Pr\Big[\sum_{i=1}^{N} X_i \geq z\sqrt{N}\Big] = 1 - \Phi(z) > \frac{1}{\sqrt{2\pi}}\frac{z}{z^2+1}e^{-z^2/2},$$

where $\Phi(z)$ is the cumulative distribution function of the standard normal distribution evaluated at z.

Theorem 18.6: Algorithm 18.4 implements a polynomial shared coin.

Lemma 18.7: Algorithm 18.4 uses n^2 coinflips, which is optimal in this model.

Algorithm 18.4 cannot tolerate even one byzantine failure: assume the byzantine node generates all the n^2 coinflips in every round due to worst-case scheduling. Then this byzantine node can make sure that its coinflips always sum up to a value larger than n, thus making the outcome -1 impossible.

18.2 Broadcast Abstractions

A message received by a node v is called **accepted** if node v can consider this message for its computation. **Best-effort broadcast** ensures that a message that is sent from a correct node u to another node v will eventually be received and accepted by v.

Reliable broadcast ensures that the nodes eventually agree on all accepted messages. That is, if a correct node v considers message m as accepted, then every other node will eventually consider message m as accepted.

```
# Algorithm 18.11: Asynchronous Reliable Broadcast (code for node u)
Broadcast own message msg(u)
upon receiving msg(v) from v or echo(w, msg(v)) from n-2f nodes w:
    Broadcast echo(u, msg(v))
end upon
upon receiving echo(w, msg(v)) from n-f nodes w:
    Accept msg(v)
end upon
```

Theorem 18.12: Algorithm 18.11 satisfies the following three properties:

1. If a correct node broadcasts a message reliably, it will eventually be accepted by every other correct node.
2. If a correct node has not broadcasted a message, it will not be accepted by any other correct node.
3. If a correct node accepts a message, it will eventually be accepted by every correct node.

This algorithm can tolerate $f < n/3$ byzantine nodes or $f < n/2$ crash failures.

The **FIFO (reliable) broadcast** defines an order in which the messages are accepted in the system. If a node u broadcasts messages m_1 before m_2, then any node v will accept message m_1 before m_2.

Theorem 18.15: Algorithm 18.14 satisfies the properties of Theorem 18.12. Additionally, Algorithm 18,14 makes sure that no two messages $msg(v, r)$ and $msg'(v, r)$ are accepted from the same node. It can tolerate $f < n/5$ byzantine nodes or $f < n/2$ crash failures.

Atomic broadcast makes sure that all messages are accepted in the same order by every node. That is, for any pair of nodes u, v, and for any two messages m_1, and m_2, node u accepts m_1 before m_2 if and only if node v accepts m_1 before m_2.

18.3 Blackboard with Message Passing

```
# Algorithm 18.17: Crash-Resilient Shared Coin (code for node u)
1:  while true do:
2:      Choose local coin c_u = +1 with probability 1/2, else c_u = -1
3:      FIFO-broadcast coin(c_u, r) to all nodes
4:      Save all received coins coin(c_b, r) in a set C_u
5:      Wait until accepted own coin(c_u)
6:      Request C_v from n-f nodes v, and add newly seen coins to C_u
7:      if |C_u| >= n^2 then:
8:          return sign(sum(C_u))
9:      end if
10: end while
```

Theorem 18.18: Algorithm 18.17 solves asynchronous binary agreement for $f < n/2$ crash failures.

So finally, we can deal with the worst-case crash failures and worst-case scheduling.

18.4 Using Cryptography

Let t, $n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t, n)**-threshold secret sharing** scheme.

Every node can **sign** its message in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message x signed by node u with $msg(x)_u$.

```
# Algorithm 18.21: (t, n)-Threeshold Secret Sharing
1:  Input: A secret s, represented as a real number
    # Secret distribution by dealer d
2:  Generate t-1 random numbers a_1,...,a_{t-1} in R
3:  Obtain a polynomial p of degree t-1 with p(x) = s + a_1x + ... + a_{t-1}x^{t-1}
4:  Generate n distinct x_1,..., x_n in R \ {0}
5:  Distribute share msg(x_1, p(x_1))_d to node v_1,..., msg(x_n, p(x_n))_d to node v_n
    # Secret recovery
6:  Collect t shares msg(x_u, p(x_u))_d from at least t nodes
7:  Use Lagrange's interpolation formula to obtain p(0) = s
```

```
# Algorithm 18.22: Preprocessing Step for Algorithm 18.23 (code for dealer d)
1:  According to Algorithm 18.21, choose polynomial p of degree f
2:  for i = 1,..., n do
3:      Choose coinflip c_i, where c_i = 0 with probability 1/2, else c_i = 1
4:      Using algorithm 18.21, generate n shares (x^i_1, p(x^i_1)),..., (x^i_n, p(x^i_n)) for c_i
5:  end for
6:  Send shares msg(x^1_u, p(x^1_u))_d,..., msg(x^n_u, p(x^n_u))_d to node u
```

```
# Algorithm 18.23: Shared Coin using Secret Sharing (i-th iteration)
1:  Replace line 12 in Algorithm 17.21 by
2:  Request shares from at least f+1 nodes
3:  Using Algorithm 18.21, let c_i be the value reconstructed from the shares
4:  return c_1
```

Theorem 18.24: Algorithm 17.21 together with Algorithm 18.22 and Algorithm 18.23 solves asynchronous byzantine agreement for $f < n/10$ expected 3 number of rounds.

Remark: In Algorithm 18.22 we assume that the dealer generates the random bitstring. This assumption is not necessary if the communication between any pair of nodes is private: In a setup phase of the algorithm, each node can generate a local coinflip and broadcast the secret shares of its coinflip to all other nodes. The corresponding secret will only be revealed in a designated round of the algorithm, thus keeping the outcome of the coinflip secret to a byzantine adversary.

A hash function $hash : U \to S$ is called **cryptographic,** if for a given $z \in S$ it is computationally hard to find an element $x \in U$ with $hash(x) = z$.

```
# Algorithm 18.26: Simple Synchronous Byzantine Shared Coin (for node u)
1:  Each node has public key that is known to all nodes
2:  Let r be the current round of Algorithm 17.21
3:  Broadcast msg(r)_u, i.e. round numver r signed by node u
4:  Comoute h_v = hash(msg(r)_v) for all received messages msg(r)_v
5:  Let h_min = min_v (h_v)
6:  return least significant bit of h_min
```

Theorem 18.27: Algorithm 18.26 plugged into Algorithm 17.21 solves synchronous byzantine agreement in expected 3 rounds for up to $f < n/10$ byzantine failures.