sikoba RESEARCH

An Introduction to Asynchronous Binary Byzantine Consenus

Aleksander Kampa
ak@sikoba.com

15 March 2021
(draft version)

Abstract

This document provides an introduction to consensus in asynchronous environments. It provides some historical background, shows how common coins can be implemented using threshold cryptography and introduces basic consensus mechanics. A recent protocol by Tyler Crain [1] is then explained in detail, and it is also shown under which conditions the Adversary would be able to break the protocol.

This paper also shows that Crain's protocol can be easily adapted to the weakly Byzantine setting, where only a subset of faulty nodes are Byzantine.

1 The Consensus Problem

1.1 Origins

An aircraft typically has several sensors to measure external condition such as temperature. Sensors can fail, but a failure does not mean that they simply stop working. Sometimes, faulty sensors start sending out seemingly random data, at the risk of significantly biasing information that is crucial for the pilot. To study possible solutions to this problem, a project called SIFT was founded by NASA in the late 1970s. This is how the field of study of Byzantine systems started. The first research paper on SIFT, entitled "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control" [2] was published in October of 1978.

A bit later, around 1980, the *Byzantine Generals Problem* was defined by Lamport, Shostak and Pease [3]. In computer science, the expression *State Machine Replication* is used to describe the general method of reaching consensus in a distributed system [4].

1.2 Advantages of Distributed Systems

There are two main advantages of distributed systems over centralised systems: distributed systems can tolerate failures and can provide resistance to fraud.

Let's consider a traditional centralised banking system. Banks usually maintain at least one fully functional backup site, with databases on the backup sites being continually synchronised with the primary site. Should their primary computer center fail, the bank must be able to quickly switch to a backup site. There may be specific legal requirements as to how far a backup site has to

be from the primary site, and how often switchover tests have to be run. Another problem with centralised systems is the potential for fraud. A single programmer may be able to alter critical data in a database. Therefore, complex security protocols have to be put in place.

In a distributed system, up to a certain proportion of processors (typically one third) can fail without compromising the system. In addition, the potential for fraud is much more limited, as is would require a significant subset of the processes to cooperate.

1.3 Basic setting

We cite from Bracha [5]: "The system consists of n processes that communicate by sending *messages* in which no messages are lost or generated. We assume a reliable message system in which no messages are lost or generated. Each process can directly send messages to any other process, and can identify the sender of every message it receives. Up to t of the processes are *faulty* and may deviate from the protocol. A protocol is called *t-resilient* if it satisfies the agreement and validity requirements in the presence of up to t faulty processes."

In this paper, we will often write *node* instead of *process*. Note that we can obtain reliable messaging by asking the recipient for an acknowledgement, and by re-sending the message as many times as necessary.

1.4 Properties of Consensus

"In the consensus problem, each process p start with a local binary value (..) and the processes have to decide on a common value. A *consensus protocol* that solves the consensus problem terminates when each correct process makes an irreversible *decision* on some value." [5]

The basic properties of consensus are the following:

- **Termination**: all correct processes eventually decide (in an expected finite number of steps)
- **Agreement**: all correct processes decide on the same value
- **Validity**: if all correct processes propose the same value, then they decide on that value

Another way to express validity is to say that the value that correct processes decide on must have been proposed by at least one correct process.

1.5 Crash failures versus Byzantine behaviour

A crash failure occurs when a process simply stops participating in the protocol. This corresponds to an irreversible computer crash.

While a crash failure is a simple affair, Byzantine behaviour means that the process can act in an arbitrary way, which includes all types of malicious behaviour. In fact, as a worst-case scenario, we must assume that all Byzantine processes are controlled by an *Adversary* whose goal is to prevent consensus to occur.

Clearly, if we want a distributed system that is resilient against all possible failures, including malicious behaviour, it has to be able to tolerate Byzantine processes.

1.6 Synchronous versus Asynchronous

"In a *synchronous* system, processes run in lock step, and messages sent in one step are received in the next." [5] This means that there is a bounded time delay for message transmission. If a

process does not receive any message from another process after a certain delay, it means that no message was sent, which provides valuable information.

In asynchronous systems, there are no predefined bounds on the time a message takes to reach it recipient.

It is easier to develop consensus protocols in synchronous settings, and such protocols tend to reach consensus faster than protocols tolerant of asynchrony. In addition, under normal circumstances, synchrony assumptions may appear to hold "almost always". It is clear, however, that any synchrony assumptions may fail in extreme circumstances. A truly resilient distributed system must, in our view, tolerate a fully asynchronous network.

1.7 From multi-valued to binary

It may seem that dealing with only binary consensus is quite limiting, as most real-world distributed systems will need to agree on potentially complex data structures, such as transactions or sets of transactions. However, efficient protocols to reduce multi-valued consensus to binary have been proposed. Without going into details, we can therefore say that in the end it always boils down to binary consensus.

1.8 Asynchronous Byzantine Binary Consensus as a Fallback

Some protocols are designed to reach consensus quickly in favourable circumstances, when a large majority of processes is correct. In such settings, the (slower) Byzantine consensus is only used as a fallback, to be called only when conditions deteriorate. One example is provided in [6].

1.9 Fundamental Results for Asynchronous Systems

The very earliest research on consensus protocols did not consider fully asynchronous environments. For example, in "The Byzantine Generals Problem" by Pease, Shostak and Lamport (1980) [7], one of the assumptions is that "the absence of a message can be detected." When adding signed messages to this assumption, it can be shown that a distributed system can tolerate *any* number of Byzantine processes, although this requires an very large number of communication rounds.

However, asynchronous were studied soon thereafter. Some important results for asynchronous systems are as follows:

- Fischer, Lynch and Paterson (1983) [8] : deterministic consensus protocols are impossible even in the case of only one fail-stop process.
- Bracha and Toueg (1983) [9]: any consensus protocol can tolerate at most $t < n/2$ fail-stop or $t < n/3$ Byzantine processes
- Ben-Or (1983) [10] : consensus protocol tolerating $t < n/2$ fail-stop or $t < n/5$ Byzantine processes
- Bracha (1987) [5] : consensus protocol tolerating up to $t < n/3$ Byzantine processes
- Bitcoin (2008) [11] : consensus protocol tolerating up to $t < n/2$ adversarial processing power (which is similar to Byzantine processes) if the requirement of round finality is dropped.

2 Common Coins a.k.a. Random Global Coins a.k.a. Random Beacons

2.1 The Need for Randomness

From a well-known result by Fischer, Lynch and Paterson, cited above, we know that binary consensus in an asynchronous setting cannot always be achieved by a deterministic process even if only a single node is crash-prone. A deterministic process is one in which no random steps are taken, and all correct processes decide after at most R rounds, with R being a constant that is known in advance.

That is not to say that deterministic consensus can never be reached. Indeed, when only a small number of faulty nodes are assumed, and when honest nodes tend to have similar initial values, consensus may still be achieved in most cases. The FLP result simply means that there will always be *some* borderline cases when no agreement can be reached if just one node crashes at the wrong time.

Therefore, to solve the consensus problem in the asynchronous setting, randomised protocols must be used. In such protocols, random steps are sometimes taken, and the termination is probabilistic: every correct node decides with probability 1, but the number of rounds is not bounded.

Early randomised protocols used local randomness, where every process would perform a coin toss in each round. These protocols were designed to ensure termination when all correct processes would obtain the same result of their coin toss, which happens with a probability of $1/2^{(n-t)}$. Using this technique, Bracha [5] was the first to present a randomised protocol tolerating up to $t < n/3$ Byzantine processes[1], which is optimal. However, the expected number of steps to reach consensus is proportional to $2^{(n-t)}$, which is exponential in n.

2.2 Shared Coins

The number of steps can be reduced by increasing the probability that the processes obtain the same coin toss result in a given round. The algorithms used to increase this probability are called *shared coin* algorithms, and the probability that all processes generate the same output is called the *agreement parameter*.

Using a polynomial shared coin algorithm, Aspnes and Herlihy [12] were the first to obtain a polynomial-step $O(n^4)$ consensus protocol in 1990. Ideally, of course, we would want a perfect shared coin, i.e. one with agreement parameter equal to 1. This was initially thought to be impossible, and in their paper Aspnes and Herlihy wrote that "implementing an unbiased shared coin is provably impossible in an asynchronous system." By 2007, Attiya and Censor [13] had managed to reduce the number of steps to $O(n^2)$ - much better, but still not very practical.

The breakthrough came by using threshold cryptography, a relatively recent field that started to be developed only during the 1990s. As we will see, threshold cryptography allows to generate almost-perfect common coins with limited communication overhead.

2.3 Threshold Signatures

In the context of a *threshold cryptosystem* [14], a (m, M)-*threshold signature scheme* is one where at least m parties out of M are required for creating a signature. Given some data $\mathfrak{D}$, the typical procedure is that each party i wishing to participate in the signing generates an individual signature:

$$sig_i(\mathfrak{D})$$

[1] his paper was submitted in 1984 but published only in 1987

Combining at least m such signatures from different parties, the threshold signature can then be computed by anyone:

$$SIG(\mathfrak{D})$$

Note that the threshold signature cannot be obtained when strictly less than m individual signatures are available. The signature $SIG(\mathfrak{D})$ is the same no matter which m parties participated in the signing process. It is therefore impossible to know which parties have participated in the signature by only looking at $SIG(\mathfrak{D})$.

A very important feature of threshold signatures, and indeed of most cryptographic signatures, is their quasi-randomness. Taking some arbitrary data, it is practically impossible to predict anything about the resulting signature[2].

2.4 Common Coins: Extracting Randomness from Threshold Signatures

A shared coin algorithm with a perfect agreement parameter (i.e. equal to 1) is usually called a *Common Coin*, a *Random Global Coin* or a *Random Beacon*.

In order to generate a common coin, the processes must have a way to independently generate the same sequence of different values or strings $COIN[1], COIN[2], COIN[3], ...$ at each round, without communicating with other processes. This can be achieved very simply: the protocol specifies a random string or number $SEED$, and we define $COIN[r]$ as a string that is a simple function of the round number r, for example:

$$COIN[r] \leftarrow \text{"\#\{}SEED\text{\}\_\#\{}r\text{\}"}$$

In a blockchain setting, we can add the block number b to ensure that the coins in each block are different:

$$COIN[r] \leftarrow \text{"\#\{}SEED\text{\}\_\#\{}b\text{\}\_\#\{}r\text{\}"}$$

Finally, again in a blockchain setting, the previous block hash $HASH[b-1]$ can be added. This eliminates the possibility of coins being computed in advance:

$$COIN[r] \leftarrow \text{"\#\{}SEED\text{\}\_\#\{}HASH[b-1]\text{\}\_\#\{}b\text{\}\_\#\{}r\text{\}"}$$

The threshold signature of the COIN value of each round can then be used to generate a common coin, as that signature is pseudo-random. The first (or second, third, ...) bit of the signature can then be taken as the common coin for each round. In Crain's paper, the signature itself is hashed first, which adds a second randomisation step, and then the first bit of that hash is taken. The process is as follows:

[2]All natural signature schemes will have this property, although one could imagine artificial signature schemes that do not satisfy it completely, for example by pre-pending a fixed string or by modifying the natural signature in some predictable way. But even in that case, such signatures will always have a quasi-random component.

Table 1: Common Coin Computation

Node i in **Round** $r > 0$

compute $COIN[r]$

broadcast $sig_i(COIN[r])$

wait until at least $n - t$ values $sig_x(COIN[r])$ are received

compute the threshold signature $SIG(COIN[r])$,
then get the first bit of the cryptographic hash of that signature:

$c[r] \leftarrow first\_bit(hash(SIG(COIN[r])))$

3 Consensus Mechanics

We continue to place ourselves in an asynchronous setting, with processes/nodes seeking to reach binary agreement in an adversarial environment.

3.1 Adversarial Setting

We are in a setting with n nodes, which are all well known and identified by their public keys[3]. Of these n nodes, t are faulty. Among the faulty nodes, $t' \leq t$ are Byzantine. Let us denote this setting $ABS(t, t')$. We can distinguish three main cases:

- when $t' = 0$, we are in a Crash-Prone setting.
- when $0 < t' < t$ we are in a Weakly Byzantine setting (as defined in [15]).
- when $t' = t$ we are in a Byzantine setting.

These settings correspond to different powers of the Adversary. In all setting, the Adversary can make up to t nodes crash, at any time and in any sequence. In Byzantine settings, the Adversary can also make up to t' nodes behave in a Byzantine, i.e. arbitrary, manner.

The Adversary is also given control over the message delivery. In addition, the adversary may be given additional powers, such as the ability to see the content of the messages. In general, we will assume that the Adversary is not able to impersonate nodes or fake messages, as the computational overhead of signing messages is small. There are signature-free protocols, which would be necessary if the Adversary was computationally unbounded, which is not a realistic setting.

3.2 Zugzwang

In any adversarial setting, up to t nodes can simply fail, stop sending any messages and cease to participate in the consensus protocol. This means that correct nodes will generally have to make decisions after receiving $(n - t)$ messages in any step of the protocol where nodes receive messages, as there is no guarantee that more messages will ever be received. This is what we call the Zugzwang number, which is the same in all adversarial settings.

Correct nodes may wait a bit longer after having received $(n - t)$ messages, to allow for the possibility of receiving further messages which might expedite consensus.

[3] Although this may seem restrictive, the higher-level consensus protocol for which the ABBC is the fallback may include ways for nodes to join or be removed from the network

Zugzwang is a German word used in games such as chess to denote a forced move, usually one that is detrimental to the player making that move. We use it here to signify that a node may be forced to proceed with limited information.

3.3 Trust Threshold

At certain steps of the consensus protocol, we want to make sure that at least one of the messages received has been sent by a correct node. This is the case when at least $t' + 1$ messages with the same value are received, as there are at most t' Byzantine nodes. The Trust Threshold number, denoted by τ, is therefore:

$$\tau = t' + 1 \tag{1}$$

In the fully Byzantine setting, we have $\tau = t + 1$.

3.4 Overlap

Consider a set S with n elements, and two of its subsets, S_1 and S_2, with n_1 and n_2 elements respectively.

When $n_1 + n_2 \leq n$ then $S_1 \cap S_2 = \varnothing$ is possible.

But when $n_1 + n_2 > n$, then $|S_1 \cap S_2| \geq n_1 + n_2 - n$. This is the minimal overlap, which we will simply call ***overlap***.

As we will see shortly, the overlap is an important concept in consensus protocols. When two correct nodes receive n_1 and n_2 messages respectively, the overlap $n_1 + n_2 - n$ represents the minimum number of messages that both nodes have received from the same processes. Often, we will want the overlap to to be at least equal to τ.

3.5 Commitment Threshold and Lower Bound for n

At certain steps of the consensus protocol, we want to make sure that if a correct node receives at least λ messages with value B, then no other correct node could have received λ or more messages with value $\neg B$. We will call λ a ***commitment threshold***. Assuming λ exists, we must obviously have $\lambda \leq t - n$ because of Zugzwang, so with $\alpha \geq 0$ we define:

$$\lambda(\alpha) = n - t - \alpha \tag{2}$$

We can then define the ***lower bound for n*** as the smallest number of nodes $\nu(t, t')$ such that $\lambda(0)$ exists in the adversarial setting $ABS(t, t')$.

Using Overlap

If two correct nodes each receive $\lambda(\alpha)$ messages with identical values B_1 and B_2 respectively, we want the overlap between to be more than the number of Byzantine nodes, i.e. at least equal to τ. That way, we can be sure that $B_1 = B_2$. We must therefore have:

$$overlap = 2\lambda(\alpha) - n = 2(n - t - \alpha) - n = n - 2t - 2\alpha > t' \tag{3}$$

This implies:

$$n > 2t + t' + \alpha \tag{4}$$

This is equivalent to:

$$\lambda(\alpha) > t + t' + \alpha \tag{5}$$

Using Views

If we want one node to see $\lambda(\alpha)$ identical messages B, while maximising the number of messages $\neg B$ that can be received by another node, the Adversary could create the following configuration:

View 1, with a maximum of B values

- t' Byzantine nodes with B)
- $\lambda(\alpha) - t'$ honest nodes with B) messages with $B : \lambda(\alpha)$
- α honest nodes with $\neg B$) messages with $\neg B : \alpha$
- (delayed: t honest nodes with $\neg B$)

To maximise the number of messages with $\neg B$ seen by another node, the adversary makes Byzantine change the value of their message and delays messages with value B from t honest nodes. The result is:

View 2, with a maximum of $\neg B$ values

- t' Byzantine nodes with $\neg B$)
- $t + \alpha$ honest nodes with $\neg B$) messages with $B : \lambda(\alpha) - t - t'$
- $\lambda(\alpha) - t - t'$ honest nodes with B) messages with $\neg B : t + t' + \alpha$
- (delayed: t honest nodes with B)

We see that the Adversary was able to create two extreme "views" of the situation that are $(t+t')$ apart [4]. We therefore find the same result as when using the overlap method. The value of $\lambda(\alpha)$ (number of messages with B in View 1) must be strictly greater than $t+t'+\alpha$ (number of messages with $\neg B$ in View 2):

$$\lambda(\alpha) > t + t' + \alpha \tag{6}$$

And we again obtain:

$$n > 2t + t' + \alpha \tag{7}$$

Conclusion

Therefore, when $\alpha = 0$:

$$n = \nu(t, t') = 2t + t' + 1 > 2t + t' \tag{8}$$

This reduces to the well-known $n > 3t$ limit in the Byzantine setting. Most consensus protocols make use of the SND concept, which therefore provides the natural lower bound on the number of correct processes required for consensus.

[4] a consensus protocol may be able to prevent Byzantine nodes from sending different messages to different nodes, here we deal with the generic worst-case scenario

4 The Crain20 Asynchronous Byzantine Consensus Process

In this section, we will consider a specific binary protocol, based on the paper entitled "A Simple and Efficient Asynchronous Randomized Binary Byzantine Consensus Algorithm" by Tyler Crain, published in early 2020 [1]. The presentation of the protocol has been modified to make it simpler to understand.

4.1 Notations

There are $n = 3t + 1$ processors, of which at most t are Byzantine.

$r \in \{0, 1, 2, ...\}$ denotes a round number.

Instead of $\{0, 1\}$, we use $\{T, F\}$ as the binary values of the consensus protocol. That way, there is no confusion between round numbers and consensus values.

$B \in \{T, F\}$ denotes an arbitrary binary value. Given the value B, we use $\neg B$ to denote its opposite. Some readers may know this as $!B$ or $\sim B$ or even $\overline{B}$.

4.2 The Adversary

In our model, the Adversary has wide-ranging powers:

- the adversary controls up to t processors;
- the adversary controls message scheduling and delivery;
- the adversary can see the contents of all messages sent.

There are however some limits to the Adversary's power:

- the adversary cannot create fake messages, and is unable to impersonate correct nodes;
- all messages must eventually be delivered.

By encrypting messages send between nodes, it is possible to hide messages from the adversary. As we will see, this can help to speed up consensus under the most adversarial circumstances, but does carry a computational overhead.

4.3 Description of the Consensus Protocol

Table 2 shows the consensus process from the point of view of a single node.

Note that in wait(r), for $r \geq 0$ a node's own message is also counted.

Re set(0): nothing prevents a node to wait for more than $(n - t)$ values, in which case two values could be received $(t + 1)$ times.

Re ini(r): again, a node could wait and receive more than $(n - t)$ values, we will see later that this does not constitute a problem.

Re decide(r): "deciding" in round r means that the final estimate of that round is broadcast only once, and remains valid for all future rounds.

Table 2: Crain20 Consensus Protocol ($n > 3t$)

Round 0		
	start	broadcast initial value $e(0)$
	wait(0)	wait until $(n-t)$ values $e_x(0)$ are received
	set(0)	set $\varepsilon(0)$ to a value received by at least $t+1$ nodes
	proof(0)	compile proof for round 0
	bcast(0)	broadcast round 0 estimate $\varepsilon(0)$ together with proof
Round $r > 0$		
	wait(r)	wait until $(n-t)$ values $\varepsilon_x(r-1)$ are received
	ini(r)	set initial estimate $e(r)$ if value B received $(n-t)$ times : $e(r) \leftarrow B$ else : $e(r) \leftarrow \varnothing$
	coin(r)	participate in computing common coin $c(r)$
	set(r)	set final estimate $\varepsilon(r)$ if $e(r) = \varnothing : \varepsilon(r) \leftarrow c(r)$ else : $\varepsilon(r) \leftarrow e(r)$
	proof(r)	compile proof for round r
	decide(r)	if $e(r) = c(r)$ then <u>decide</u>
	bcast(r)	broadcast current round estimate $\varepsilon(r)$ with proof

4.4 Three Definitions

Table 3 provides three definitions that will be useful later. While these definitions refer to a specific node i, we may use $PINI(B)$, $PHOLD(r, B)$ and $OPEN(r)$ when the context is clear.

Note on PINI(B)

A correct node following the protocol exactly will only be able to have a single value received $(t+1)$ times after receiveing $(n-t)$ values in wait(0) ...

... the node may however receive more $e_x(0)$ messages later, and be able to prove $PINI(\neg B)$

... and if that node received a valid value $\varepsilon_x(0) = \neg B$ in wait(1), then it will also be able to prove $PINI(\neg B)$

Note on PHOLD/OPEN

First, note that after receiving $(n-t)$ values in wait(r), any node will be in exactly one of the three states: OPEN, PHOLD(T) or PHOLD(F).

Then, note that if a node's final estimate in round $(r-1)$ was B, then after receiving $(n-t)$ values in wait(r), that node can only be be in the states OPEN and PHOLD(r, B).

Finally, note that if a node with $\varepsilon_i(r-1) = B$ waits for additional messages, it may also be able to prove $PHOLD_i(r, \neg B)$ in addition to $OPEN(B)$.

PHOLD(T) and PHOLD(F) cannot hold in the same round

It is clear that no two nodes in the same round can have PHOLD(T) and PHOLD(F), whether correct or Byzantine.

Table 3: Three Definitions

$PINI_i(B)$	has $(t+1)$ valid $e_x(0) = B$ values *PINI allows to prove that B is the initial value of at least one correct node*
$PHOLD_i(r, B)$	has $(n-t)$ valid $\varepsilon_x(r-1) = B$ values *PHOLD allows to prove that a node can override the value of the global coin in round r, if necessary*
$OPEN_i(r)$	does not have $(n-t)$ identical valid $\varepsilon_x(r-1)$ values *This means that the node has received $\varepsilon(r-1)$ values for both T and F and is unable to override the value of the global coin*

4.5 Defining Proof

We are now ready to provide a concise definition of what is required in proof(r).

Table 4: Proof Requirements

$VALID_i(r, B, proofs)$	Has there been a round $r' \leq r$ such that the value of the common coin $c(r')$ was $\neg B$? if **no** : requires $PINI_i(B)$ if **yes** : requires $PHOLD_i(r', B)$

As before, we will simply write $VALID(r, B)$ when the context is clear. Note that there is no common coin in round 0, therefore $VALID(0, B)$ requires $PINI(B)$. More importantly, note that this definition is *recursive*, in that proofs in a given round may require proofs from previous rounds.

In order to make things much more explicit, and to help with understanding, table 5 lists these validity rules in much more detail.

Again, notice the recursive nature of the proof requirements.

To override a common coin value in a given round r, it suffices to show $(n-t)$ final estimates $\varepsilon(r)$ that are different from the common coin - that condition is simple and easy to understand.

The condition that allows a process to accept the round's common coin value B is a bit less intuitive:

- if there was no previous round in which the common coin was $\neg B$, all we do is check that $PINI(B)$ holds, i.e. that the value B is the initial value of at least one correct node.
- however, if there was a previous round $r' < r$ in which the common coin was $\neg B$, then we must be able to explain why the node in question did not set its final estimate to the common coin's value in that round r'. In other words, we need a proof that the node was allowed to override the value $\neg B$ in round r', otherwise how could we have gotten to round r? This proof is provided by $PHOLD(r', \neg B)$.

Table 5: Detailed Proof Requirements ($r > 0$)

When $c(r) = T$

$VALID(r, T) = confirm(r, T)$

Has there been a round $r' < r$ such that
the value of the common coin $c(r')$ was F ?

if **no** : requires $PINI(T)$

if **yes** : requires $PHOLD(r', T)$

$VALID(r, F) = override(r, F)$

if **yes** : requires $PHOLD(r, F)$

When $c(r) = F$

$VALID(r, F) = confirm(r, F)$

Has there been a round $r' < r$ such that
the value of the common coin $c(r')$ was T ?

if **no** : requires $PINI(F)$

if **yes** : requires $PHOLD(r', F)$

$VALID(r, T) = override(r, T)$

if **yes** : requires $PHOLD(r, T)$

4.6 Demonstrating Validity

To get some intuition as to how the protocol works, let's assume that all honest nodes have the same initial value. Without loss of generality, let's choose T as that value.

Round 0

All correct nodes send T in start. Because there are at most t Byzantine nodes, no node can obtain (t+1) values F in wait(0), and therefore no node will be able to send a valid estimate F in bcast(0). As a result, all valid $\varepsilon_x(0)$ values must be T.

Byzantine nodes cannot interfere, all they can do is attempt to slow down progress by delaying bcast(0).

Round 1

In Round 1, all correct nodes set e(1) to T in ini(1), as we have seen that all $\varepsilon_x(0)$ values must be T. The nodes then participate in discovering the value of the common coin c(1).

With probability 50%, c(1) will be found to be T. In that case, all correct nodes decide on that T, broadcast that decision and stop.

Let's now assume c(1) is found to be F.

In that case, the protocol requires all correct nodes to maintain $\varepsilon(1)$ at T. Does this meets the validity requirement $VALID(1, T)$? Yes it does, as it simply requires PHOLD(1,T) which holds. So T is valid.

But could a Byzantine node broadcast a valid value F? Let's look at $VALID(1, F)$. As we have no previous common coin with value T, we would need $PINI(F)$ to hold - which we know is not possible.

So the only possible value that any node, Byzantine or not, can set $\varepsilon(1)$ to is T.

Round 2

All correct nodes set $e(2)$ to T in ini(r), then participate in uncovering the common coin $c(2)$.

With probability 50%, c(2) will be found to be T. In that case, all correct nodes decide on that T, broadcast that decision and stop.

And if $c(2)$ is F, we will again find that the only valid estimate of $\varepsilon(2)$ for all nodes is T.

etc.

By now it should be clear that:

- Consensus will be reached in the first round when the common coin is T;
- As long as the common coin is F, all processes must maintain their estimate at T;
- the Adversary can do nothing to prevent consensus, expect slowing down message delivery.

The expected number of rounds to reach consensus is:

$$\sum_{n=1}^{\infty} \frac{n}{2^n} = 2$$

4.7 Ensuring Progress

We must ensure that correct nodes can always make progress from one round to another. We can never rely on Byzantine nodes for that: the Adversary can always make all Byzantine nodes simply stop participating in the protocol.

Round 0 : Because there are at least $(n-t)$ correct nodes, every correct node eventually receives $(n-t)$ values in $wait(0)$. Because $(n-t) \geq 2t+1$, one of these value will be received at least $t+1$ times, thus ensuring progress in round 0.

Round 1 : It is easy to verify that progress is also ensured in round 1.

All we have to do now is to show that if honest nodes have progressed to round r, with all necessary proofs provided until that round, then they can also progress to round $r+1$. This is done, somewhat fastidiously, in tables 6 to 9. Note that we also cover situations in which a process decides to wait for more than $(n-t)$ values in $wait(r)$, showing that this does not pose a problem.

To help unerstand these tables, let's take table 6, which assumes that the value of the common coin in round r was T, and that a given correct node set its final estimate to $\varepsilon(r)$ to T. Then, in round $(r+1)$, that node can be in one of three configuration: $PHOLD(T)$, $OPEN$ or $PHOLD(F)$, and in each of these the common coin $c(r+1)$ can be either T or F. That gives us 6 configurations, and we need to show that the node in question can provide a proof of its final estimate in all of these configurations.

For example, in table 6 point ❶ we deal with the case $PHOLD(r+1, T)$ and $c(r+1) = T$. The protocol requires the final estimate to be set to $\varepsilon(r+1) = T$, so we need to prove $confirm(r+1, T)$. We find that this is actually equivalent to $confirm(r, T)$, which holds via $\varepsilon(r) = T$ and by the induction hypothesis.

Table 6: Progressing from $c = T$ and $\varepsilon = T$ $(r > 0)$

Round r $\quad c(r) = T$ and $\varepsilon(r) = T$

This means that $confirm(r, T)$ holds

Round $r + 1$

$PHOLD(r + 1, T)$

❶ $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$

if no F before, need $PINI(T)$
if F before, need $PHOLD(r', T)$

... this is equivalent to $confirm(r, T)$
... which holds via $\varepsilon(r) = T$

❷ $c(r + 1) = F$ ■ $\varepsilon(r + 1) = T$

we need $override(r + 1, T)$, which we have

$OPEN(r + 1)$

❸ $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$ ... see (1) above

❹ $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$

the previous coin was T so we need $PHOLD(r, F)$

... which we have because we have received
... at least one valid $\varepsilon_x(r) = F$

$PHOLD(r + 1, F)$

This should normally not happen, requires receiving more than $(n - t)$ values in $wait(r + 1)$

❺ $c(r + 1) = T$ ■ $\varepsilon(r + 1) = F$

we need $override(r + 1, F)$, which we have

❻ $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$ ... see (4)

Table 7: Progressing from $c = F$ and $\varepsilon = F$ $(r > 0)$

Round r $\quad c(r) = F$ and $\varepsilon(r) = F$

This means that $confirm(r, F)$ holds

Round $r + 1$

$PHOLD(r + 1, T)$

This should normally not happen, requires receiving more than $(n - t)$ values in $wait(r + 1)$

1. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$

the previous coin was F so we need $PHOLD(r, T)$

... which we have because we have received
... valid values $\varepsilon_x(r) = T$

2. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = T$

we need $override(r + 1, T)$, which we have

$OPEN(r + 1)$

3. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$ ... see (1) above

4. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$

if no T before, need $PINI(F)$
if T before, need $PHOLD(r', F)$

... this is equivalent to $confirm(r, F)$ which holds

$PHOLD(r + 1, F)$

5. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = F$

we need $override(r + 1, F)$, which we have

6. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$ ... see (4)

Table 8: Progressing from $c = T$ and $\varepsilon = F$ $(r > 0)$

Round r $\quad c(r) = T$ and $\varepsilon(r) = F$

This means that $override(r, F)$ holds

Round $r + 1$

$PHOLD(r + 1, T)$

This should normally not happen, requires receiving more than $(n - t)$ values in $wait(r + 1)$

1 $\quad c(r + 1) = T \quad \blacksquare \quad \varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$

if no F before, need $PINI(T)$
if F before, need $PHOLD(r', T)$

... this is equivalent to $confirm(r, T)$ which holds
... because we have received valid values $\varepsilon_x(r) = T$

2 $\quad c(r + 1) = F \quad \blacksquare \quad \varepsilon(r + 1) = T$

we need $override(r + 1, T)$, which we have

$OPEN(r + 1)$

3 $\quad c(r + 1) = T \quad \blacksquare \quad \varepsilon(r + 1) = T$

we need $confirm(r + 1, T)$ ... see (1) above

4 $\quad c(r + 1) = F \quad \blacksquare \quad \varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$

the previous coin was T so we need $PHOLD(r, F)$

... which we have via $\varepsilon(r) = F$

$PHOLD(r + 1, F)$

5 $\quad c(r + 1) = T \quad \blacksquare \quad \varepsilon(r + 1) = F$

we need $override(r + 1, F)$, which we have

6 $\quad c(r + 1) = F \quad \blacksquare \quad \varepsilon(r + 1) = F$

we need $confirm(r + 1, F)$ ... see (4)

Table 9: Progressing from $c = F$ and $\varepsilon = T$ $(r > 0)$

Round r $c(r) = F$ and $\varepsilon(r) = T$

This means that $override(r, T)$ holds

Round $r + 1$

$PHOLD(r + 1, T)$

1. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

   we need $confirm(r + 1, T)$

   we had F in the previous round, so need $PHOLD(r, T)$
   ... which we have by $\varepsilon(r) = T$

2. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = T$

   we need $override(r + 1, T)$, which we have

$OPEN(r + 1)$

3. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = T$

   we need $confirm(r + 1, T)$ ... see (1) above

4. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

   we need $confirm(r + 1, F)$

   if no T before, need $PINI(F)$
   if T before, need $PHOLD(r', F)$

   ... this is equivalent to $confirm(r, F)$ which holds
   ... because we must have received a valid $\varepsilon_x(r) = F$

$PHOLD(r + 1, F)$

This should normally not happen, requires receiving more than $(n - t)$ values in $wait(r + 1)$

5. $c(r + 1) = T$ ■ $\varepsilon(r + 1) = F$

   we need $override(r + 1, F)$, which we have

6. $c(r + 1) = F$ ■ $\varepsilon(r + 1) = F$

   we need $confirm(r + 1, F)$ ... see (4)

4.8 The Path To Decision

Let's assume that we are in round r and no correct node has yet decided. Correct nodes as a group can be in one and only one of three general configurations:

- COR-OPEN(r): all correct nodes have an initial estimate $e(r) = \varnothing$
- COR-SET(r,T): at least one correct node has $e(r) = T$, all others having $e(r) = \varnothing$
- COR-SET(r,F): at least one correct node has $e(r) = F$, all others having $e(r) = \varnothing$

Note that if some of the correct nodes deviate slightly from the protocol and wait for more than $(n - t)$ values in $wait(r)$, they may have a choice between setting $e(r) = \varnothing$ and $e(r) = B$ for some value B. This is not a problem, as they are expected to choose a definite value in $ini(r)$. Note that configurations COR-SET(r,T) and COR-SET(r,F) are mutually exclusive.

We define similar configurations for Byzantine nodes:

- BYZ-OPEN(r): all Byzantine nodes have an initial estimate $e(r) = \varnothing$
- BYZ-SET(r,T): at least one Byzantine node has $e(r) = T$, all others having $e(r) = \varnothing$
- BYZ-SET(r,F): at least one Byzantine node has $e(r) = F$, all others having $e(r) = \varnothing$

Again, note that BYZ-SET(r,T) and BYZ-SET(r,F) are mutually exclusive.

4.8.1 COR-OPEN(r)

Let's assume we are in context COR-OPEN(r), i.e. no correct node has set an initial estimate in round r.

Without loss of generality, we assume that the Byzantine nodes are in contexts BYZ-OPEN(r) or BYZ-SET(r,T)[5].

With a probability of 50%, $c(r) = T$. In that case, All correct nodes will set $\varepsilon(r) = T$, and no Byzantine node can prove $\varepsilon(r) = F$ and override that value.

In round r+1, all correct nodes set $e(r + 1) = T$.

Then, with a probability of 50%, $c(r + 1) = T$ and all correct nodes decide T

Else, if $c(r + 1) = F$, it can be easily seen that all correct nodes maintain T and no Byzantine node can override.

etc.

We therefore see that consensus is reached in the first round after r when the coin is T

Bottom line: from a COR-OPEN(r) configuration, there is a probability of at least 50% that consensus will be reached an expected 2 rounds later, i.e. at expected round $r + 2$

4.8.2 COR-SET(r,B)

We now assume context COR-SET(r,B), meaning that at least one correct node has set an initial estimate in round r, without loss of generality let this value be T. We know that no other node, correct or Byzantine, can have set its initial value to F.

[5] if the Adversary was able to correctly guess the value B of the common coin in round r, it would try to put the Byzantine nodes into context BYZ-SET(r,¬B) in order to avoid consensus. Without this knowledge, the Adversary has to choose between BYZ-SET(r,T) and BYZ-SET(r,F).

With probability 50%, we will have $c(r) = T$, so all correct nodes with an initial estimate T will decide. All other nodes will set their final estimate to T, as none can override the coin value.

In round $(r + 1)$, all correct nodes set their initial estimate to T. We have two cases:

If $c(r + 1) = T$ then all remaining correct nodes (if any) decide T and consensus is reached.

If $c(r + 1) = F$, all correct nodes maintain T as their final estimate. As to Byzantine nodes, we can easily verify that they are unable to $confirm(r + 1, F)$ and must also either set $\varepsilon(r + 1) = T$ or stop participating in the protocol. Therefore, all nodes will again start round $r + 2$ with initial estimates of T.

etc.

So again it is clear that all nodes will maintain the estimate T and decide that value as soon as the coin becomes T.

Bottom line: from a COR-SET(r,B) configuration, there is a probability of 50% that consensus will be reached an expected 2 rounds later, ie at expected round $r + 2$.

4.8.3 Expected Number of Rounds to Consensus

In round 1, correct nodes are in one of the configurations COR-OPEN(1) or COR-SET(1,B). It follows that consensus is reached in an expected 2 rounds after round 1, i.e in round 3.

5 Strategy of the Adversary

In order to prevent consensus, the Adversary needs four things:

- The correct nodes cannot all have the same initial value in round 0, as there is no available adversarial strategy in this scenario[6];
- The ability to see the round 0 initial values $e(0)$ of all correct nodes;
- Full control of message delivery scheduling;
- The ability to know the next 2 coin values.

Note that it is sufficient to see the initial values of the correct nodes only in round 0. Because the Adversary is able to schedule message delivery, and correct nodes will follow the protocol precisely, the adversary will always know the estimates of all correct nodes in subsequent rounds.

5.1 Round 0 Strategy

We assume that the correct nodes do not start with the same initial estimate in round 0. It is then easy to see that, if the Adversary can see these initial estimates, it can cause correct nodes to have any combination of estimates $\varepsilon(0)$. This can be done as follows.

1. The Adversary waits for all $(n - t)$ correct nodes to broadcast their initial values in *start*. At least $(t + 1)$ correct nodes will have broadcast the same initial estimate B, while at least one correct node will have broadcast $\neg B$.
2. The Adversary then causes all t Byzantine nodes to broadcast $e(0) = \neg B$. As a result, $(t+1)$ initial values $\neg B$ are being broadcast.

[6] the only possibility for the Adversary to prevent consensus would be to cause the common coin to always take the same value, which is impossible

3. In wait(0), the Adversary can then choose to deliver $(t+1)$ initial values of either B or $\neg B$ to every correct node, and t of any other value, thus forcing that node to set $\varepsilon(0)$ to the desired value.

Even if the Adversary only has a total of $(t+1)$ initial values B, and there are $2t$ values $\neg B$, this is sufficient to force a correct node to set $\varepsilon(0) = F$. The Adversary delivers $(t+1)$ values B and t values $\neg B$, after which the Adversary stops further deliveries until that node broadcasts its round 0 estimate. Any messages delivered afterwards will not affect that node's round 0 estimate.

5.2 Strategy for Subsequent Rounds

In order to remain in control at the beginning of a given round r, the Adversary needs to:

- start with a suitable configuration of $\varepsilon(r-1)$ estimates;
- know the values of $c(r)$ and $c(r+1)$ before staring the delivery of messages in $wait(r)$.

There are really only two possibilities: the next coin will be either the same, or different.

When the two next coins are the same, the Adversary makes the correct nodes override the coin value in round r as follows:

Table 10: Case B $\rightarrow$ B

Round	Coin	estimate	N1	N2	N3	Z
r	**B**	e	¬B	¬B	¬B	∅
		ε	¬B	¬B	¬B	B
r+1	**B**					

We have used a scenario with 4 nodes, of which three are correct (N1, N2 and N3) and one is Byzantine (Z). This can easily be extended to the case of $(2t+1)$ correct and t Byzantine nodes.

When the coins change between rounds, the Adversary makes the correct nodes follow the next coin, the one in round r, which will make it possible to later override the coin in round $(r+1)$. This is how this works:

Table 11: Case B $\rightarrow$ ¬B

Round	Coin	estimate	N1	N2	N3	Z
r	**B**	e	∅	∅	∅	¬B
		ε	B	B	B	¬B
r+1	**¬B**					

In the Appendix, we show a sequence of rounds in which the Adversary manages to prevent consensus by applying the concepts that were just presented.

6 Crain20W: an ABBC protocol in a Weakly Byzantine Setting

The Crain20 protocol relies on two essential points:

1. Trust threshold:
   in set(0), $\varepsilon(0)$ will be set to a value that is the initial value of at least one correct node.
2. Commitment threshold:
   in ini(r), if some node sets its initial estimate $e_i(r)$ to a value B that is not $\varnothing$, then no other node, whether correct or Byzantine, can set its estimate to $\neg B$.

It is therefore immediately clear that it can be adapted to the Weakly Byzantine setting, with only one very simple modification, as shown in table 12. This protocol requires $n \geq \nu(t, t') = 2t + t' + 1$ nodes (cf 3.5).

Table 12: Crain20W Consensus Protocol ($n > 2t + t'$)
Difference from Crain20

Round 0	
set(0)	set $\varepsilon(0)$ to a value received by at least $\lfloor \frac{n-t}{2} \rfloor + 1$ nodes

In fact, it would be enough to require $t' + 1$ similar values in set(0) but it makes sense to require correct nodes to set their round 0 final estimate to the value received most often.

Versions

15 March 2021: first draft version published on http://research.sikoba.com

References

[1] Tyler Crain. A simple and efficient asynchronous randomized binary byzantine consensus algorithm. *CoRR*, abs/2002.04393, 2020.

[2] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

[3] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[4] State machine replication. `https://en.wikipedia.org/wiki/State_machine_replication`.

[5] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[6] Aleksander Kampa. Proof-of-decision: Achieving fast and timeout-resistant consensus in asynchronous byzantine environments. `http://research.sikoba.com/`, 2019.

[7] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[8] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In *PODS*, pages 1–7. ACM, 1983.

[9] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *PODC*, pages 12–26. ACM, 1983.

[10] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC*, pages 27–30. ACM, 1983.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[12] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, Sep 1990.

[13] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. In *STOC*, pages 155–164. ACM, 2007.

[14] Threshold cryptosystem. `https://en.wikipedia.org/wiki/Threshold_cryptosystem`.

[15] Aleksander Kampa. One-step consensus in weakly byzantine environments. `http://research.sikoba.com/`, 2019.

A Preventing Consensus: The Adversary in Action

In this example, we assume that the distributed system consist of 4 nodes, of which 3 are correct (node N1, N2 and N3) and one is Byzantine (Z). Not all of the correct nodes have the same initial estimate in round 0. Given that the Adversary always makes all the correct nodes take the same value, this approach can be applied to any number of nodes.

Table 13: Preventing Consensus

Round	Coin	estimate	N1	N2	N3	Z
0		e	0	0	1	1
		ε	0	1	1	1
1	**0**	e	1	1	1	∅
		ε	1	1	1	0
2	**0**	e	∅	∅	∅	1
		ε	0	0	0	1
3	**1**	e	0	0	0	∅
		ε	0	0	0	1
4	**1**	e	0	0	0	∅
		ε	0	0	0	1
5	**1**	e	0	0	0	∅
		ε	0	0	0	1
6	**1**	e	0	0	0	∅
		ε	0	0	0	1
7	**1**	e	∅	∅	∅	0
		ε	1	1	1	0
8	**0**	e	∅	∅	∅	1
		ε	0	0	0	1
9	**1**	e	∅	∅	∅	0
		ε	1	1	1	0
10	**0**	e	1	1	1	∅
		ε	1	1	1	0
11	**0**	e	1	1	1	∅
		ε	1	1	1	0
12	**0**	e	1	1	1	∅
		ε	1	1	1	0
13	**0**	e	1	1	1	∅
		ε	1	1	1	0
14	**0**	e	∅	∅	∅	1
		ε	0	0	0	1
15	**1**	e	∅	∅	∅	0
		ε	1	1	1	0
16	**0**	e	1	1	1	∅
		ε	1	1	1	0
17	**0**	e	1	1	1	∅
		ε	1	1	1	0
18	**1**					