Abstract—Cryptocurrencies, of which Bitcoin and Ethereum are the most popular, are based on an underlying abstraction known as blockchain. The blockchain itself is built to offer two different but complementary abstractions: a distributed tamper-proof ledger and consensus. The properties of blockchains consensus are significantly different from classical consensus algorithms in the distributed computing literature. Moreover, those properties diverge depending on the type of blockchain and the underlying consensus algorithm. In this dissertation, we explore how to build a consensus service on top of existing blockchains, and how the different guarantees provided by the latter can be exploited by applications. Due to blockchain properties, the API provided by a blockchain based consensus service is more complex than the one provided by implementations of traditional consensus algorithm such as Paxos, PBFT or Raft. Blockchain Consensus Service (BCS), allows applications to balance performance and consistency i.e., applications can change from a costly strong consistent consensus execution to a weakly consistent and more efficient one. With this, we provide a Byzantine fault tolerant consensus service with a classic API that offers incremental consistency and a distributed tamper-proof ledger.

I. INTRODUCTION

In 2008, Satoshi Nakamoto presented Bitcoin [1], a peer-to-peer version of electronic cash with the objective of eliminating the need for a trusted third party in online transactions. Bitcoin is one of the most popular online transaction systems. It works with digital coin and the key technology behind it is the blockchain. The blockchain itself is built to offer two different but complementary abstractions: a distributed tamper-proof ledger and consensus. The blockchain is an ever-growing chain of blocks. This chain works as a ledger because it is unique and stores all the valid transactions.

Distributed applications need to keep a consistent state across replicas or need replicas to come to an agreement about operations. Because most distributed applications need agreement across replicas and the fact that many distributed systems problems are equivalent to consensus, consensus plays a key role in the development of applications. There are a few consensus protocols like Paxos [2], Raft [3] and Practical Byzantine Fault Tolerance (PBFT) [4] that, despite being relatively simple in the normal case, is very hard to understand in the presence of faults. One thing that is more difficult than understanding the algorithms is to implement them correctly. Blockchain is not trivial to understand but its usability can be. This happens despite the eventual complexity of the underlying consensus algorithm and cryptographic methods used. Because of this, we believe that a consensus service based on a blockchain system can be relatively simple to implement.

Our problem is: how to offer a classical consensus interface on top of blockchain consensus. We want to provide a classical consensus interface so that a large range of applications can use blockchain to solve consensus. Also, the methods from a classical consensus interface are well known by the community since it has been used for years in several different protocols. Besides being simpler, blockchains like the one used in Bitcoin, also admit inconsistencies for some undetermined time. These inconsistencies make room for speculation. Speculation allows applications to use the time that they are waiting for consensus replies to do useful work. By doing this, applications can save time and be more responsive and available, increasing user satisfaction.

The main issue that we need to overcome in order to provide a consensus service based on blockchain is how to handle blockchain’s fork problem. Forks occur when two or more nodes in the blockchain create different versions of the chain of blocks creating several alternative histories of transactions. How can we manage this problem in an acceptable way? Fortunately for us forks happen with low probability (2%) which allows for real systems to function. We present the BCS, a service implemented on top of blockchains that allows applications to choose the level of consistency (or trust) that they want for every single execution of consensus by choosing between a weak and a strong consistent result.

The main motivation for BCS is the idea of joining the properties of traditional consensus systems and blockchain systems to provide a new system with a large and interesting range of properties that this two types of systems could not provide alone.

The properties that blockchain offers are very interesting and they are not easy to provide. The properties may vary depending on the type of blockchain but the most usual are: anonymity, decentralization, persistency and auditable. But beyond the properties, what is interesting in the blockchain is the distributed tamper proof ledger. The ledger, together with cryptographic methods make the system reliable.

A system that is able to provide the properties of consensus together with the properties of blockchains is a system that; is decentralized, tolerates Byzantine faults,
is auditable and persistent. In other words, is a system that applications can trust and will be developed a small cost of adding an abstraction layer to a blockchain system. To the best of our knowledge, BCS is the first system that provides both consensus and blockchain properties with a simple consensus API. The API allows already existing applications to migrate from a traditional consensus algorithm to a consensus service based on blockchain gaining a new set of guarantees.

The main contribution of this paper is BCS: a new approach on consensus services taking advantage of already existent blockchain infrastructure, providing the properties of consensus and blockchain at the same time while offering a simple API and handling forks in a practical manner.

II. BACKGROUND AND RELATED WORK
A. Consensus

Consensus is one of the fundamental problems in distributed systems. Many important distributed systems problems are reducible or equivalent to consensus. Examples are atomic broadcast and State Machine Replication (SMR) [5].

There are different variations of consensus in the literature, like binary consensus, multi-value consensus or vector consensus and others. All variations of consensus can be defined in terms of three properties; validity, agreement and termination. Normally the difference between them is the definition of the validity property [5].

We are particularly interested in the vector consensus variation (first presented in [6]) because that is the one that we will try to provide in our solution. Here the validity property is called vector validity and its definition is the following [5]:

• Vector validity: Every correct process that decides, decides on a vector $V$ of size $n$:
  - $\forall p_i :$ if $p_i$ is correct, then either $V[i]$ is the value proposed by $p_i$ or $\perp$;
  - at least $(f + 1)$ elements of $V$ were proposed by correct processes.

• Agreement: No two correct processes decide differently.

• Termination: Every correct process eventually decides.

Consensus is a problem simple to explain and it is straightforward to solve if, and only if, there are no failures. A more interesting challenge is to solve it in the presence of faults. In fact, in an environment where faults exist, consensus may be impossible to solve. It has been proven that in an asynchronous system where even a single process may crash it is impossible to solve consensus (the FLP impossibility [7]). Despite the FLP impossibility, there are numerous algorithms that solve consensus in the presence of faults [2], [3], [4], [8] and [9]. This is possible because the authors of such algorithms work around the impossibility. [10] presents a survey about techniques that are used to work around the FLP impossibility.

We are particularly interested in speculative methods because our solution is based on blockchain and the most popular blockchain consensus algorithm Proof-of-Work (PoW) has a problem. The problem is that PoW admits inconsistencies for some undetermined time. One way for applications to deal with these inconsistencies is through speculative executions. The use of speculation needs to be careful because it introduces new and difficult challenges [8] [11] [12].

B. Blockchain

A blockchain is composed by a chain of blocks. Each block is composed by a timestamp, a nonce, a hash of the previous block and a list of all the new transactions that were not stored in the previous blocks [13]. Miners are nodes in the network that have the function to mine (“create”) new blocks, producing a PoW. PoW is the process of finding a nonce that, when hashed together with the content of the block produces a hash starting with a specific number of zeros. This hash proves that the history of transactions has not been tampered with.

There are two types of blockchains: permissionless blockchains like Bitcoin’s blockchain where everyone can participate, and permissioned blockchains where participants are controlled, have identities and form a consortium [14].

Permissionless blockchains are normally based on proof-of-“something” like PoW and Proof-of-Stake (PoS). We focus only on PoW and PoS because those two are the most used and studied types of proof. In abstract, PoW is very simple, one entity proves to another entity or entities that it performed some computation. In Bitcoin, this proof is done through the calculation of a hash and the process of calculation this hash is called mining [1].

In practice, many problems may occur during the calculation of the PoW as well as problems related to the PoW concept itself. The most relevant problems are forks and the 51% attack.

A normal fork is, in simple words, when two different chains exist. Normal forks occur when two different miners do the PoW for two different blocks at the same time. A set of processes accepts the block from one miner while the others accept the block from a different miner. An undefined number of normal forks may occur, and multiple concurrent chains may exist at the same time. Processes chose the longest chain as the correct one because it is the one with more work done and eventually only the longest chain will remain. Forks area problem because they cause inconsistencies in the system. They cause the existence of more than one concurrent and competing chains. Nodes in different points of the network will observe a different order of transactions.

Because forks are the main problem in our solution we will try to understand the probability of that event.
In [15] and [16] both empirical and theoretical estimates are made. In [15] the theoretical probability of a fork is 1.741% and the empirical value is 0.3%. In [16] the theoretical probability of a fork is 1.78% and the empirical value is 1.69% that was optimized to 0.78% with an improvement proposed in the paper. Those measurements and calculation where made with specific models for PoW blockchain and under the normal case (i.e., as far as the authors known, there was not an attacker in the system).

Permissioned blockchain was created because certain types of applications (e.g., banking application) can not have anonymous user and information can not be public. In this type of blockchain, nodes need to be authenticated by the consortium of members. With authentication, information is only available for selected nodes and every member of the system is well known. Due to the fact that PoW and PoS blockchains store the entire history of transactions and do not require authentication from users, they can not be used to build this type of blockchain. For these reasons, permissioned blockchains use traditional consensus protocols or equivalents, like Paxos or an implementation of PBFT [4] to order transactions. Hyperledger fabric, for example, uses an implementation of PBFT [14].

III. Solution

Our solution is targeted for permissionless blockchain system because as we have seen in Section II-B they provide properties that permissioned blockchains do not provide, and pose a more difficulty solution due to the possibility of forks.

A. System Model

The system is composed by: Applications that are the clients of our system, Wrappers, Blockchain Clients and the blockchain itself. Each Application has a Library and is connected to one Wrapper. Every Wrapper is connected to one client Application and to one Blockchain Client. The Blockchain Client is connected to one Wrapper and to peers of the blockchain network. Next, we present the assumptions about our system:

- Communications: we assume that interprocess communication and remote communications are done using authenticated perfect links. Authenticated perfect links are defined in Section 2.4.6 of [17] and provide three properties: reliable delivery, no duplication, authenticity.
- Membership: BCS is publicly accessible. Still, to participate, Applications must have a blockchain account and get registered in a Certification Authority (CA). All clients with a blockchain account have a key pair (a public key and a private key). When clients perform an operation by sending a transaction to the blockchain, the transaction must be signed using the private key. To validate that a particular transaction was created by a member of the system (not just someone with a blockchain account), clients have access to a certificate signed by a trusted CA, which has a list of public keys of all the members in the system. Whenever a Wrapper receives a transaction, it uses the certificate to validate the client’s public key and verify that it is indeed a member of the system.
  - Fault Tolerance: if a set composed by Application, Library, Wrapper and Blockchain Client has a crash or any other type of fault (Byzantine); it is assumed that there are always enough correct processes such that we can obtain a Byzantine quorum. That is, the number of faults that the system tolerates is limited. This eliminates the needs for a fault detection mechanism. In a system with N Applications (and respective Wrapper and Blockchain Client) our system tolerates \( F = \lfloor (N-1)/3 \rfloor \) faults. We assume that the set composed by Application, Library, Wrapper and Blockchain Client can be seen as a single process in the system and thus fail simultaneously and in the same way.
  - Consensus Instance: the system assumes that clients know the identifiers of the consensus instances that they want to participate. The identifiers are used to distinguish between different consensus instances.
  - Checkpoint: the system assumes that there is a checkpointing mechanism relative to its own state. Checkpointing is a mechanism that periodically saves the state of the blockchain relevant to our system. Thus, when a new client enters the system, his Wrapper can query this checkpoint mechanism to know the state of the system at the time the checkpoint was made. After obtaining a state through the checkpoint, the Wrapper completes its state by reading the remaining blocks until it reaches the same point of the remaining Wrappers.

Checkpointing is important for two reasons. First, it prevents the potentially time-consuming reading of the Blockchain for each of the new system clients. Second, it allows Wrappers not to save information about the system from the beginning of time.

A fairly simple method to avoid the need for a checkpoint is to force Wrappers to read the entire blockchain from the beginning of time (beginning of time for our system, and not the entire blockchain) and save their state. Reading the blockchain can take days and the amount of information related to the system can be quite a lot.

- Forks: the system assumes a low probability of forks. The system also assumes that hard forks do not happen and that the soft forks have a very high probability of being resolved in the N blocks that are closer to the head of the chain (more about forks in Section III-G).

B. Target Applications

Based on the system model presented in Section III-A and the restrictions caused by the way BCS handles forks
A fork can be seen as speculative execution that needs to be aborted because it caused inconsistencies among replicas. Based on this, applications that use our system need to be able to deal with these potential inconsistencies. One of the inspirations for our idea on how to handle forks was a system called Correctables presented in [18].

Correctables [18] makes incremental consistency guarantees available to the applications. By offering the same abstraction, we target the same class of applications identified in [18], such as a ticket selling system and an ad serving system.

However, our strong consistency model is not final as the one presented in [18] due to the non-zero (but very small) probability that we can observe a fork longer than N blocks. Therefore, applications must be able to deal with this low probability event. Examples of applications that, despite this problem, could use our system are stock management applications. When the stock is very high, the application can simply use the weak consistent results. When the stock is low the application may choose to wait for the stronger consistent results. Even if the stronger consistent results turn out to be inconsistent and for example, the application sells more units of a product than the ones that are available this is the same thing that already happens in real-world applications. For example, it is public knowledge that flight companies sell more tickets than the ones that are available to maximize profits in case a passenger does not show up to the flight. When there are more passengers than seats, they compensate the passenger that do not get to go on the plane.

C. Architecture

To implement a consensus service that provides a classical consensus API on top of existing blockchains; we propose the architecture defined in Figure 1.

The architecture is composed of the following components:

- **Application**: It is the client of the BCS service. It uses the API provided by the Library to interact with the system.
- **API**: Bridges the gap between the application and the consensus algorithm. This API has, among others, the ⟨Propose,v⟩ and ⟨Decide,d⟩ methods or similar. We will discuss the API in Section III-D.
- **Library**: The Library provides the API and manages communication between the Application and the Wrapper. It is responsible for the translation of operations from the Application to the Blockchain. For example, translating a propose invocation into a blockchain transaction with the necessary information. It is also responsible for informing the client when decisions are available.

- **Wrapper**: The Wrapper has the function of executing the main algorithm. It is responsible for forwarding the transactions sent by the Library with the information of operations to the Blockchain. The Wrapper reads the blockchain and sends the results from the execution of the algorithm to the Library.
- **Blockchain Client**: The Blockchain Client is a peer of the Blockchain network and is responsible for committing the transactions to the Blockchain, to be connected to other blockchain peers and to receive the chain of blocks that compose the Blockchain.
- **Blockchain**: The Blockchain is responsible for solving consensus and store the history of all transactions. It consists of a set of nodes that, typically, are similar to the Blockchain Client (Section II-B).

We will now discuss the proposed architecture, together with its strengths and weaknesses. The justification for the Application, the Blockchain Client and the Blockchain itself are very simple. We need an Application because it is the client of our system and we need it to invoke operation, otherwise the system would do nothing. We use a blockchain because the main idea behind our work is to leverage its properties. We use a Blockchain Client because it is the simplest way to interact with the Blockchain network.

The reason for the Library component is the following: we decided to add this extra layer to the Application because it allows us to keep the information necessary to create transactions with the owner of this information (the Application) and at the same time not force the Application to create the transactions themselves. This is important for two reasons. First, the information needed to create transactions is very sensible (it contains the private key of the blockchain account) and because of this, it
can not leave the client. Second, if the applications are forced to create transactions, our system would not be compatible with already existing applications. Having the Library component, allows us to offer a simple consensus API. It is much simpler to use an already developed library than to modify the Applications in order for them to create the blockchain transactions. Still, developers who want to manage transactions themselves can ignore the Library and communicate directly with the Wrapper.

The Wrapper is the middleware responsible for transforming the consensus abstraction that the Blockchain provides for transactions into a generic consensus service.

The strongest point of this architecture is that the separation between the Library and the Wrapper. By doing this separation we allow two things. The first is that it is possible to run the Library and the Wrapper in different machines, which means that the Application does not need to have the blockchain in its machine. The second is: in a system that is not subject to Byzantine faults, the Wrapper can serve multiple Libraries (see Section III-A).

The strongest point of this architecture is that the separation between the Library and the Wrapper. By doing this separation we allow two things. The first is that it is possible to run the Library and the Wrapper in different machines, which means that the Application does not need to have the blockchain in its machine. The second is: in a system that is not subject to Byzantine faults, the Wrapper can serve multiple Libraries (see Section III-A).

The weakest points about this architecture are: the Wrapper needs to run in the same machine as the Blockchain Client which means that it needs to run in the machine that has the entire blockchain (depending on the Blockchain system that is used, it can consume up to 1TB of disc space, and when the initial synchronization is occurring it also consumes a decent amount of memory). The Wrapper and the Blockchain Client need to run on the same machine because for security reasons the implementations of Blockchain clients that exist do not support secure remote connections. This happens because using a remote connection would increase the attack surface making the system more vulnerable. They do not support remote connection to force all the accesses to be made locally.

The fact that the Wrapper needs to run in the same machine as the Blockchain Client causes another issue. If we want to tolerate Byzantine faults, the Application, the Library, the Wrapper and the Blockchain Client need to be the same entity. This means that they will all fail in the same way and at the same time. This assumption simplifies our design. In a Byzantine, system components can not trust each other and because of this, if we separated the Library from the Wrapper, the libraries would need to talk to a Byzantine quorum of Wrapper because it could not trust in a single one. This would increase the complexity of the system.

We saw in Section III-A that our system tolerates $F = \lfloor (N - 1)/3 \rfloor$ faults. When we refer a fault, the components that are included are: the Application, the Library, the Wrapper and the Blockchain Client.

D. API

We will now present BCSs API.

1) (join,gId): allows clients (the applications) to join a group in the system.
2) (leave,gId): allows clients (the applications) to leave a group the system.
3) (propose, id, gId, v): allows clients (the applications) to propose a value to a consensus instance. It is a propose that receives the value to be proposed, the id of the consensus instance and the group that the instance and the application belong to.
4) (decide, d): callback that informs the application about a decision. It returns the respective decided value $d$ with the accepted values. The vector contains; the id of the groups, the id of the instance, the values and meta information about the values. It contains a list of the members of that instance, the number of the blockchain block that contains the last accepted value (it is the point where the decision was made) and for each value, it contains the id of the client that proposed it.
5) (decideFinal, d): callback that informs the application about a final decision. It was made by the second reading of the wrapper and provides an extra level of consistency.

E. Algorithm

This section introduces the core algorithm of our system. The algorithm is executed by clients and wrappers. Although Wrappers do not communicate with each other, their decisions must be the same because they all access to the same blockchain and the blockchain ensures the order of the transactions in the system (except in the exceptional case of forks).

F. Execution at a Participant

Next we explain in a very simple way the sequence of steps that occur at a participant of the system. Briefly, the steps are:

1) The Application invokes (Join,gId) to join the groups with gId in the system.
2) The Library creates a transaction with the information that its application wants to join the system and sends this transaction to the Wrapper.
3) The Wrapper forwards the transaction to the Blockchain Client.
4) The Blockchain Client commits the transaction to the Blockchain.
5) The Blockchain Client replies to the Wrapper with the transaction hash. The Wrapper forwards the hash to the Library and from the Library the hash
The idea behind the concept of having groups in the system is to allow the system to have different types of applications running concurrently. This allows different applications with their replicas execution consensus on our system in a concurrent way.

An important detail is the beginning of the reading by the Wrapper (i.e. the block in which they begin reading). The size of Bitcoins blockchain is more than 170GB and the size Ethereums blockchain is more than 1TB. Because of the size of the Blockchain and the fact that transactions that the system commits to the blockchain will be included at the head of the blockchain, all the blocks that were added before the beginning of our system are irrelevant to us. For those two reasons, the reading should start at the block that was the head of the blockchain when our system first started. In practice the reading can start in any block, it’s just a matter of configuring the system and having all the Wrappers start reading at the same block.

### G. Forks

The main challenge in our approach was how to deal with forks. Forks pose a problem to the resolution of consensus because when they occur, they cause different Wrappers to read different blocks that can have different orders in the transactions.

There are two main mechanisms for The first

The most popular mechanism for dealing with forks in the blockchain requires that: to confirm a transaction as permanent, it is necessary to wait for a certain number of blocks, say N blocks, to be added to the blockchain [19]. That is, a transaction t included in the block with number X is confirmed when the block with number $X + N + 1$ is included in the blockchain. In the Bitcoin blockchain, the value typically used for N is 6 [20], in Ethereum the recommended value is 12 [20].

Therefore, in concrete, we deal with forks by having the Wrapper reading the blockchain twice. The first reading always reads the head of the blockchain while the second reading is a fixed number of blocks behind. This works because forks have a low probability of occurrence and they happen at the head of the blockchain. The fact that the second reading is not done at the head of the blockchain makes it less probable to be affected by forks (i.e. it will be less affected by those forks cause by two miners producing different blocks at the same time). The second reading is not immune to forks, but they already have a low probability at the head of the chain and the more the second reading is behind, the lower the probability of it to be affected by a fork.

It is also important to clarify the difference between the state of the blockchain and the state of our system. The state of the blockchain is composed by: transactions, clients, the balance of clients’ accounts, the order of transactions, among others. While the state of our system is composed by: the consensus instances, the members of each instance and the system’s clients (the Applications). The state of the system depends on the state of the blockchain because of the order of the transactions. Thus,
an inconsistency in the state of the blockchain (forks), leads to an inconsistency of the state of our system.

Note that the checkpoint that we talked about in this section and the checkpoint that we explain in Section III-A are different. The checkpoint in Section III-A refers only to our system and it is a checkpoint on the state of the system (which applications, which members of each application, which instances of incomplete consensuses among others ...). The checkpoint referred in this section to solve forks is done on the blockchain as a whole and would continue to compel the Wrapper to read the whole blockchain to obtain the state of the system. These two types of checkpoint are compatible and have the potential to be used together.

H. Discussion

The goal of this dissertation was to take advantage of the properties offered by Blockchain systems and at the same time provide the properties of Byzantine consensus. Both [21] and [22] list the properties provided by a Blockchain system. We have seen how the system works, now we will see the properties that the system offers. Because or system uses Blockchain it also provides same of the Blockchain properties. The properties that our system provide that derivative directly from Blockchain are:

- **Decentralization**: Blockchains have their own distributed consensus algorithm that maintains consistency in the network eliminating the need for a central entity that ordered the transactions. Because of this, and because we did not add any centralization point to the system, our system is also decentralized.

- **Persistence**: It is nearly impossible to delete or rollback transactions in the blockchain. To do it an attacker would need to own some serious computational resources. Because the operations in our system are done through transactions, our system also has persistency.

- **Auditability**: It is possible to audit a blockchain because transactions can be easily tracked and verified. Again, because the operations in our system are done through transactions, our system also has auditability. In fact, in the absence of a fork, it fairly simple for anyone that knows our algorithm to read the blockchain and check how every decision was made.

Note that our system, as well as Blockchains, is auditable, distributed and has persistency all thanks to the distributed tamper-proof ledger that is the core of the Blockchain.

We know that consensus is defined by three properties, agreement, validity and termination. Therefore, in order for our system to offer a consensus service, we need to provide those properties. In a permissioned Blockchain there is no problem in assuring them but in permissionless blockchain forks pose a problem. We have seen in Section II-B that the theoretical probability of forks is below 2%. Our solution in a permissionless Blockchain can provide those properties with more than 98% probability. Also, with the second reading, the probability of a fork is even small and gets close to zero. We can think that the 2% probability of something to go wrong in the first reading, and even lower for the second reading, is a fair price to pay for the extra properties that the Blockchain provides and that traditional consensus system does not provide.

Because the system provides a vector consensus service we define agreement, validity and termination in the same way as in Section II-A. Therefore, if we have N processes (Application, Library, Wrapper and Blockchain Client we have already stated in Section III-A are a single process) and tolerate f failures, we have the following:

- **Vector validity**: Every correct process that decides, decides on a vector V of size n:
  - \( \forall p_i : \text{if } p_i \text{ is correct, then either } V[i] \text{ is the value proposed by } p_i \text{ or } \perp; \)
  - at least \( (f + 1) \) elements of \( V \) were proposed by correct processes.

- **Agreement**: No two correct processes decide differently.

- **Termination**: Every correct process eventually decides

Our system provides vector validity because every correct Wrapper in the system collects more than \( (N + f)/2 \) proposals. The missing values to complete the total number of N, are the \( \perp \) values. In practice, we only collect the correct values from the blockchain.

In the absence of a fork, every Wrapper reads the same blocks from the blockchain, and all correct Wrappers will interpret the blockchain in the same way, so they will all decide the same ensuring agreement.

In our system, in the absence of a fork, all the operations from correct processes will reach the Blockchain and eventually every correct Wrapper will read them. The only case where our system may not decide is in the presence of fork, namely when the Wrapper reads the wrong blocks from the wrong chain and those blocks do not contain the transactions with the proposals (or the transactions are not in the same order). Then the fork is solved and the transactions with the proposal are included in blocks that the Wrapper had already read (the first time). The second reading will fix this problem and all the Wrappers will have the same final decisions.

Note that, while improbable, forks that are longer than the number of blocks that the second reading is behind the first can happen. If this happens, Wrappers will read the wrong blocks from the wrong chain and those blocks may not contain the transactions with the proposals (or the transactions may not be in the same order). Then the fork is solved and the transactions with the proposal are included in blocks that the Wrapper had already read (the first and the second readings) and the Wrapper will never read those proposals, the system will stop, and no decision is ever made for that consensus instance.
This chapter presents the evaluation of our solution. Recall that, the objective is to provide a consensus service based on a blockchain system. With this in mind, our evaluation was designed to answer the following questions:

1) Is it possible to have a consensus system based on a permissionless blockchain system?
2) In what way do forks impact the system? What is the frequency of incorrect first decisions and incorrect final decisions?
3) What is the cost of waiting for a final decision? How long is the wait between the initial decision and the final decision? How often does the initial and final decision differ?

To help us have a better understanding of our system and to provide some clues about the answers to these questions, we thought of some metrics. The metrics are the following:

- Transactions request latency: the amount of time it takes for a transaction created by a client operation to reach the blockchain and for the transactions hash to be returned to the client has a form of confirmation.
- Initial decision latency: the amount of time elapsed on a client since it proposed a value for a consensus instance until he receives the initial decision from that instance.
- Final decision latency: the amount of time that passed on a client since it proposed a value for a consensus instance until he receives the final decision from that instance.
- Number of blocks per decision: the number of blocks that were created since the block that contains the first value in the decisions until the block of the last value included in the decision.
- Wrong decisions: the number of times that a single client received a final decision different from the initial one. This can also be seen as the number of times that a single client was affected by a fork.
- Wrong final decisions: the number of times that for the same instance, at least one client received a different decision from the other thus violating agreement. This can also be seen as the number of times that a single client was affected by a fork longer than N, the defined number for the second reading, blocks.
- Instances with a wrong decision: the number of instances where at least one client receive a different initial decision.
- Number of clients that received a wrong decision per instance: the number of clients that received a wrong decision in an instance with a wrong decision. We will use this metric to understand how many clients were affected by the fork that caused the wrong decision.

The idea with the evaluation is to have a notion about the impact of dealing with forks and the general performance of the system. Normally this type of evaluation is done to compare different solutions, but we know that our solution is not comparable with, for instance, a Paxos, Raft or SMR implementation. Those system are capable of execution the equivalents of our consensus instance under a second ([3], [23]), while our system is limited by the blockchain block generation rate (in the order of tens of seconds).

A. Test Application

As our use case application, we implemented a simple ticket selling system inspired in Listing 5 of [18]. The application sells tickets to events (e.g. concerts, movies etc...) and is available to vendors. Vendors can create events with a certain amount of tickets and can sell tickets to those events.

In the application, every client knows the existence of one event and how many tickets that event has. The tickets in the application are all equal. The Application proposes to the system the number of tickets that it wants to sell. When the decision is delivered, the application checks if the number of tickets that remains is above zero. If the remaining amount of tickets is above zero then all the replicas of the application, that successfully proposed a value, can sell the tickets that they requested. If there are not enough ticket to fulfill all the requests in the vector, the application sells the tickets by order. This way a replica of the application can know if it can or not sell the tickets. The application has a threshold on the number of tickets and when that threshold is reached the application waits for the finals decisions before deciding about selling the tickets. For a stock of tickets above the threshold, applications can safely sell the tickets without waiting for a final decision thus improving latency.

B. Test Description

We tested our system in the Ethereum test network, Rinkeby [24]. In this network, blocks are generated at an average rate of one block every 15 seconds. Because blocks are mined in a semi-trusted environment instead of the public environment of the Ethereum main network, the block generation rate does not have large variations with a standard deviation of half a second [24].

We executed two experiments, one with 7 clients and another with 14 clients. We choose to start with 7 because we thought that for our minimum number of clients we would like to tolerate more than 1 fault. With seven clients, the system is able to handle 2 faults. Also, seven was the maximum amount clients that we were able to run in the machine that we used for development.

The experiments were run in a Supermicro Superworkstation 7039a-i with 132 Gb of memory ram (four units of DDR4-2666, Registered ECC with 32 Gb). The disc is an Intel DC S4500 Enterprise series SSD, 480GB, 2.5” TLC SATA-6Gb. It has two 4U active CPU heat sink and two CPUs Intel Xeon Silver 4114, LGQ 3647, 2.2GHz, 10C/20T.
In this machine, we had only 220GB of disk available. With this amount of disk available, the maximum number of clients that we were able to execute was 14. This happens because each client needs to have its own copy of the blockchain.

In Ethereum, every transaction has a transaction fee (so does Rinkeby). The creators of the transaction define the maximum amount of ether that they are willing to pay [13]. Due to transaction fees, the creators of the blockchain blocks give priority to transactions with the highest transaction fees. Because of this, in our system, and due to the fact that we had access to a large amount of Rinkeby ether, we decided to pay a for transactions a value substantially higher than the average value paid in the network. According to [24] the gas price is at 1 Gwei (1 Gwei corresponds to 0.000000001 ether) and the gas limit of 0.000009 Gwei. For our transactions in the system, we defined the gas limit to 0.0001 and gas price at 1.0001 Gwei. Another detail is that Rinkeby is an Ethereum test network so the number of transactions in it is very low when compared to the main Ethereum network. Most of the blocks in Rinkeby do not reach their limit. These two facts make it very likely that the transactions in our system were included in the block to be created. Another transaction was not prioritized over ours.

In the two experiments, we executed 24 runs of 100 instances of consensus for a total of 2400 instances of consensus. Every time that we started a run, clients had a new Ethereum account with the initial balance of 0.05 ether. Starting with new accounts for every execution allow us to test the system without us having to worry about nonces of the transactions (see Section II-B). All transactions had the same destination address (0x816641acb67a07517059d62655c7f0d02732b39c) meaning that all transactions in our system were sent to the same account. In [24] is possible to search by the destination address to view all the transactions that our system created during the experiment and even during development and testing.

C. System Performance

Table I shows the raw values measured. From Table I we can observe some interesting facts about our system. The first is that the difference in seconds between the delivery of the initial decision and the final decision is around 180 seconds. This large gap makes total sense and due to the fact that the reading of the blockchain that finds the final decisions is 12 blocks behind the initial reading. Note that, the blockchain only creates one block at a time. On average, the Rinkeby network takes \(15 \times 12 = 180\) seconds to generate 12 blocks.

An observation is that the time for the decisions to be delivered did not increase when the number of clients increased, in fact, it appears to stay stable. Unfortunately, we can not be sure about this because tests with a greater number of clients are necessary to prove if this observation is valid or not. Another problem with this observation is that for a larger number of clients, the number of blocks per decision will increase. This will happen for two reasons. One is that with more clients, the number of clients that are a little late will increase and the values that they propose will appear later in the blockchain. The second one is that for a large number of clients, it is impossible for all the values needed to make the decisions to be included in the same block because blocks have a limit in the number of transactions that they can contain. This means that a decision whose values have to fit in two blocks instead of just one will take at least two times the time needed to generate a block \((2 \times 15 = 30\) seconds\). If the decision occupies three blocks it will take at least three times the time needed to generate a block \((3 \times 15 = 45\) seconds\) and so forth.

The problem with the number of blocks that a decision occupies can already be observed in our measurements. Although they are rare, we observed decisions that occupied more than one block, this raised the average number of blocks per decision to a value bigger than one and potentially having an impact on the time measured.

In our system, considering the number of blocks per decision to be around 1.2 and that the block generated rate is 15 seconds we have that the expected value to be near \(1.2 \times 15 = 18\) seconds. The observed values are a little higher than the theoretical calculation. But if we take into account the standard deviations of the measures we can see that the observed time, for both the initial and final decisions, is not far from what can be expected.

D. Forks Measures

After we have seen and discussed the performance of our system we will know discuss the metrics about the forks. Table II presents the values obtained in our experiments.

By looking at the values in Table II we observe that by doubling the number of clients and obviously doubling the total number of decisions made by clients, the number of wrong decisions also doubled. It makes perfect sense that with more clients we would measure more wrong decisions but what is interesting is that the percentage of wrong decisions per total decisions made remained stable (with 7.8 \% for 7 clients and 7.1\% for 14). We can also observe that the number of instances with at least one wrong decision grew very little. We explain this by the fact that despite having more clients (which is more propitious to having more wrong decisions), soft forks in the network have nothing to do with the number of clients that our system has. The frequency of soft forks in the network is independent of the number of clients in BCS. Having more clients will cause the number of wrong instances to increase because we have more clients that can be affected by a soft fork. Analyzing the values for the wrong clients per instance, we see that normally a wrong decision affects more than one client. This means that the fork that caused it affected other nodes in the network.
Finally we can observe that the number of wrong final decisions is 0. This means that throughout the entire set of tests the second reading in our system was always correct. The second reading is always correct because we used the value for transaction confirmation in the Ethereum blockchain (as discussed in Section III-G).

V. CONCLUSION

This document presents BCS, a blockchain based consensus service, that provides incremental consistency to applications. Furthermore, our solution bridges the gap between classic consensus systems and blockchain systems. Incremental consistency allows our system to work around the fork problem of permissionless blockchains.

A blockchain based consensus service allows the existence of a highly scalable system. We know this is true because of the existence of large-scale blockchain applications such as Bitcoin and Ethereum. The use of Blockchain also provides distributed tamper proof ledger that stores all the results from the execution of consensus. To the best of our knowledge, there is no other consensus service that provides a distributed tamper proof ledger.

BCS is able to deal with forks, tolerates Byzantine faults, and works despite the existence of fork.

REFERENCES


### TABLE I: Latencies Raw values

<table>
<thead>
<tr>
<th>Number of clients</th>
<th>Initial Decision</th>
<th>Final Decision</th>
<th>Block Generation</th>
<th>Number of blocks per decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>23.67 ± 8</td>
<td>205 ± 8.2</td>
<td>15 ± 0.5</td>
<td>1.27 ± 0.47</td>
</tr>
<tr>
<td>14</td>
<td>21.74 ± 8.2</td>
<td>203 ± 9.4</td>
<td>15 ± 0.5</td>
<td>1.14 ± 0.36</td>
</tr>
</tbody>
</table>

### TABLE II: Fork related metrics

<table>
<thead>
<tr>
<th>Number of clients</th>
<th>Clients decisions</th>
<th>Wrong initial decisions</th>
<th>Wrong final decisions</th>
<th>Instances with wrong decisions</th>
<th>Wrong clients per instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>16 091</td>
<td>1 262</td>
<td>0</td>
<td>477</td>
<td>2.64 ± 1.34</td>
</tr>
<tr>
<td>14</td>
<td>33 877</td>
<td>2 407</td>
<td>0</td>
<td>505</td>
<td>4.77 ± 2.55</td>
</tr>
</tbody>
</table>