Secure Blockchain-Based Voting at Party Conventions

Roland Bernhard Reif*

Abstract

This paper illustrates the design and implementation of a smart contract for secure voting to be used, e.g. at party conventions. This implementation supports both voting on motions and electing candidates running for office. Depending on their roles, participants have different privileges. The smart contract ensures that statues and the rules of procedure are automatically followed. The code is published in a public repository[1].

1 Introduction

Elections are the cornerstone of liberal democracies and the trust of citizens in their integrity is essential for the acceptance of election results. In the past two decades, classical paper ballots have seen the rise of alternative voting systems, including hybrid approaches (e.g. Scantegrity [1]) and pure online voting (e.g. OVIS [2]). The cryptographic primitives used in blockchain systems offer the building blocks to enhance online voting systems by better ensuring that requirements (e.g. integrity, reliability, uniqueness, etc.) for voting systems are met.

The novel approach of this paper is the implementation of a smart contract for the specific requirements of party conventions. The author has been a member of a political party for over half a decade and participated in various roles (guest, member, alternate delegate, delegate, admin[2]) in dozens of party conventions. Traditional paper-based voting can be quite time-consuming as votes have to be collected and counted manually. When delegates pass/reclaim their voting rights to/from the (alternate) delegate of their choice it can be quite complicated to locate this person in the pool of hundreds of convention participants. It is also hard for the admin(s) to ensure that certain rules are strictly followed, e.g. that any (alternate) delegate may only hold a maximum of two voting rights at a time and that voting rights are only passed to eligible participants. These and many other problems will be solved by this project.

2 Related Work

In this section, we provide a very brief selection of related work to hint at the vast number of scientific papers regarding voting on blockchains. Also we briefly relate our project to these works.

A systematic review of current research trends and advantages and challenges of Blockchain based voting systems has been performed by Taş et Tanrıöver in [3], where they analyzed 63 scientific papers. In a similar way Jafar et al. in [4] have looked at the state of the art and provide basic explanations about influences blockchain technologies could have on e-voting systems. According to their research blockchain technology can make voting more accessible, can stop illegal voting and make it easy to verify election results. We will see, that these prospects hold for this paper.

None of the papers analyzed in the reviews and no other paper examined by the author handles the specific requirements for party conventions.

Hjálmarsson et al. [5] have identified three

*ro_reif@live.concordia.ca

[1]https://github.com/RBReif/secureVoting

[2][3]

main aspects when creating a smart contract for any voting system:

1. Identifying the roles involved in elections (admins and voters).
2. Processing elections (creation, voter registration, etc.).
3. Actual voting by voters.

This project follows and extends this approach, e.g. by implementing more granular roles and allowing for two different election processes (depending on whether it is a vote on a motion or an election of people into an office).

3 Requirements for Party Convention Voting System

The following requirements are mainly (not fully) based on the rules of procedure and statues of the author's party[4]. The scope of this project is to implement a smart contract for party conventions to facilitate voting processes that fulfill the following requirements.

- There exists exactly one voting right per delegate and this is initially held by the respective delegate.
- Delegates can transfer their voting rights to to other delegates or to alternate delegates. These can transfer these voting rights further.
- Each (alternate) delegate can hold a maximum of two voting rights at the same time.
- Delegates can reclaim their original voting rights.
- Per voting, each voting right can be cast only once.
- Any party member has the right to create a regular motion.
- The answer options for regular motions are "yes", "no", and "abstain".
- If and only if a regular motion receives the explicit support of at least 10 delegates or of at least 30 party members, the motion will be called to a vote.
- Admins can create elections.
- Any party member can run as a candidate in an election.
- Admins can close elections and regular motions.
- Anyone (including guests) can see the results of motions after voting has ended.

Such a smart contract can be easily adapted for similar purposes, e.g. for annual general meetings of companies, etc.

4 Design

4.1 Entities in the Voting Process

Two main classes of entities exist in the context of party conventions: participants and voting processes. Voting processes are either regular motions about content with three possible answer options "yes", "no" or "abstain". Or they are elections of candidates to party offices with the options "abstain" and all party members,

[4] https://www.fdp-bayern.de/sites/default/files/2021-12/FDP%20Bayern%20Satzung%20%2880LPT%29.pdf and https://www.fdp-bayern.de/sites/default/files/2022-04/FDP%20Bayern%20WAO%20%2881LPT%29.pdf

who are running for this office. Even though their are differences in the creation processes and in the way these voting procedures are started, they still share a lot of similar functionality when it comes to their conduction (e.g. who is allowed to cast a vote), their closing and the counting of cast votes. Hence, the author made the design decision to represent both voting processes in the same class (`Voting`).

4.2 Stages in the Voting Process

Votings can be in different stages and in each stage different actions are possible. A design decision for three stages is plausible. One stage represents the period before voting has started (`PREPARED`), one stage represents the period during voting (`OPEN`) and the last stage represents the final state after voting has ended (`CLOSED`). Figure 1 illustrates the three stages in the voting process on a regular motion. First a party member creates the motion. Then further party members can support it. Once enough delegates ($\geq$ 10) or enough party members ($\geq$ 30) support it, the stage is changed to `OPEN` and eligible voters can cast their votes. The final stage is reached once all votes were cast or once an admin closes the voting process. Then anyone can take a look at the results.

The voting process for elections is identical for the `OPEN` and the `CLOSED` stages. Yet such votings can only be created by admins. During the `PREPARED` stage party members can nominate themselves or other party members as candidates. After a certain time admins can transition the stage of the election to `OPEN`.

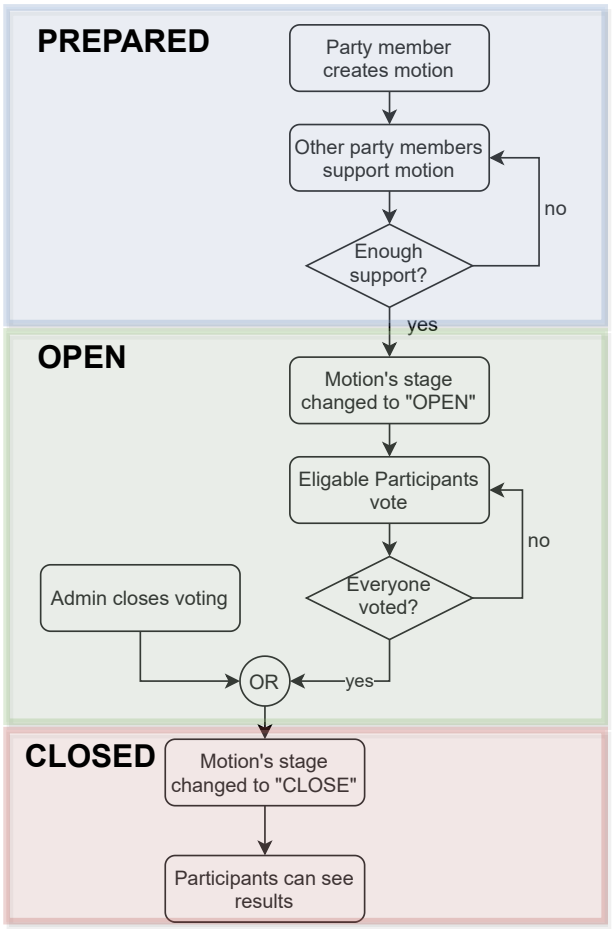

Figure 1: Process of voting on a motion.

5 Implementation

The smart contract is implemented in Solidity[5] to be run on Ethereum[6] or one of its testnets.

5.1 Global State

Globally we need to keep track of all participants and all votings. The source code for global vari-

[5] https://soliditylang.org/
[6] https://ethereum.org/en/

ables is depicted in Figure 2. All participants are stored in both a mapping from the participants addresses to themselves (`participants`) and in a dynamic array (`participantsArray`). This allows an efficient lookup with `partipants` and also allows the iterate over all participants with `participantsArray`. The later will be needed when a `DELEGATE` wants to find and reclaim a transferred voting right. In addition, we also store a counter of all voting rights (= number of delegates including admins). This is more efficient than to always iterate over all participants and count those with voting rights. All votings regardless of their stage are stored in a dynamic array.

```
uint public votingRightsCounter = 0;
mapping(address => Participant) public participants;
address[] public participantsArray;
Voting[] public votings;
```

Figure 2: Globally kept state.

5.2 Structs

Figure 3 shows the source code for the three structs that are used to represent the two kinds of entities we elaborated on in Chapter 4.1. A participant holds a role (`GUEST`,`MEMBER`, `ALTERNATEDELEGATE`, `DELEGATE`, or `ADMIN`). This is implemented with an Enum and hence `GUEST` equals 0 and `ADMIN` equals 4. Each role includes all previous roles, e.g. each `ALTERNATEDELEGATE` is also a `MEMBER` and also a `GUEST`. An active voting right is represented by a non-zero address in the two `votingRightXfrom` variables. When a new `DELEGATE` is added `votingRight1from` is set to the delegate's own address. This makes it possible that we always keep track of who is the original owner of a voting right. This allows us that this owner can reclaim a transferred right. Votings are identified by an `ID`. The `text` variable holds the question to be voted on (e.g. "Should we legalize Cannabis?" or "Who should be elected Chairperson?"). `vtype` differentiates the type of `Voting` (either `MOTION` or `ELECTION`) and `stage` differentiates the current stage. To prevent double-voting, we keep track of all voting rights that have been used already (`voters`). We also keep track of all party members that express their support of a regular motion in `initialSupporters`.

The actual voting options that voters can vote for are stored in a dynamic array (`options`). Each option is represented by an substantiation of the additional struct **Option** and contains a text (e.g. "yes", "no", "abstain", or the name of a candidate) and a counter for votes cast in favor of this option.

```
struct Participant {
    ConventionRole role;
    address votingRight1from;
    address votingRight2from;
}
struct Option{
    string text;
    uint voteCount;
}
struct Voting{
    uint id;
    string text;
    VotingType vtype;
    VotingStage stage;
    Option[]   options;
    address[]  voters;
    address[]  initialSupporters;
}
```

Figure 3: Entities represented by structs.

	Guest	Member	Alt.-Delegate	Delegate	Admin
addParticipant()					X
createElection()					X
startElection()					X
closeVoting()					X
reclaimVotingR()				X	X
transferVotingRX()			X	X	X
vote()			X	X	X
createMotion()		X	X	X	X
supportMotion()		X	X	X	X
runInElection()		X	X	X	X
seeResults()	X	X	X	X	X

Figure 4: Public Functions and allowed callers.

5.3 External Functions

Various internal helper functions are implemented to e.g. check if there are any open motions or to actually transfer voting rights between (alternate) delegates, to check if a motion has enough supporters to be opened, and so on. For the sake of simplicity, this paper will focus on the external functions. An overview of which role is allowed to execute which function is provided in Figure 5.3.

Admin Functions

To add a new convention participant an admin can call `addParticipant(role, addr)` containing the role of the new participant (a value between 0 and 4) and the participant's address. If there are open votings or if the participant was already added the smart contract aborts. Otherwise the function can succeed. If `role` is *geq* `DELEGATE` the participant's `votingRight1from` is set to its own address and the global `votingRightsCounter` gets incremented. The new participant gets inserted into the respective global stage (`participants` and `participantsArray`). Figure 5 depicts the respective source code.

```
function addParticipant(ConventionRole role_, address addr_ )external{
    require(participants[msg.sender].role == ConventionRole.ADMIN,
    "Only admins can add new convention participants.");

    require(participants[addr_].votingRight1from==address(0)
    && participants[addr_].votingRight2from==address(0),
    "The Participant was already added and holds voting rights." );

    require(noOpenVotings(),
    "Can not add new participants while voting is going on.");

    Participant memory newParticipant;
    newParticipant.role = role_;
    if (role_ >= ConventionRole.DELEGATE){
        newParticipant.votingRight1from =addr_;
        votingRightsCounter++;
    }
    participants[addr_]  = newParticipant;
    participantsArray.push(addr_);
}
```

Figure 5: Implementation of addParticipant()

With `createElection(txt): id` admins can create new elections in the `PREPARED` stage. The `txt` input parameter will be the text of the voting (e.g. "Who should be elected Chairperson?). Only one `Option` is added on position 0 in the `options` array with the text "abstain". The function returns the index of the newly created election in the global `votings` array.

Admins can use `startElection(id)` to change the stage of an election to `OPEN`. With `closeVoting(id)`, admins can close votings (of both types). `id` represents the index of the respective `Voting` in the global `votings` array.

Delegate Function

Figure 6 shows the source code for what happens when a delegate calls `reclaimVote(): success` to reclaim the delegate's voting right. After having checked that the calling person is a `DELEGATE` and that he neither holds already two voting rights nor does he already hold his own voting right, the smart contract locates the caller's voting right. Then the voting right is transferred back to its original owner. A boolean return value indicates successful execution.

```
function reclaimVote() external returns(bool){
    require(participants[msg.sender].role >=ConventionRole.DELEGATE,
    "Only DELEGATEs can reclaim their original voting right.");

    require(participants[msg.sender].votingRight1from == address(0)
    || participants[msg.sender].votingRight2from == address(0),
    "Reclaimer already holds 2 active voting rights");

    require(participants[msg.sender].votingRight1from != msg.sender
    && participants[msg.sender].votingRight2from != msg.sender,
    "Reclaimer already holds his/her original voting right");

    for (uint i=0; i< participantsArray.length;i++){
        if(participants[participantsArray[i]].votingRight1from==msg.sender){
            transfer(participantsArray[i],msg.sender,1);
            return true;
        }
        if(participants[participantsArray[i]].votingRight2from==msg.sender){
            transfer(participantsArray[i],msg.sender,2);
            return true;
    }   }
```

Figure 6: Implementation of reclaimVote()

Alternate Delegate Functions

To transfer a voting right in the first place, an (alternate) delegate can call `transferVotingRightX(towhom)`. After ensuring the right roles at sender and receiver side, the smart contract enforces that the sender actually has a voting right to send and that the receiver currently holds less than two voting rights. Additionally we do not allow the transfer of voting rights during open votings.

In an `OPEN` voting process any participant with the role of `ALTERNATEDELEGATE` or higher has the right to cast their votes. This can be done by calling `vote(votingid, optionid)` where `votingid` is the index of the voting in `votings` and `optionid` is the index of the option one wants to choose (e.g. 0 for "abstain"). Obviously one is only allowed to vote, if one holds at least one voting right that was not cast in this particular voting.

For each not previously cast voting right, the counter for the selected option is incremented and the address of the voting right is added to the `voters` attribute to prevent double-voting. As soon, as all votes have been cast, the voting process is closed automatically. The source code for this function is illustrated in Figure 7.

Member Functions

New motions can be created by party members via calling `createMotion(txt):id`. This will create a new motion with the provided text and the three standard options ("abstain", "yes", "no"). The new motion will be added to the global `votiongs` array and its index is returned.

Party members can call `supportMotion(votingid)` and if they have not supported this motion before and if the motion is currently in the `PREPARED` stage, then the caller will be added as a supporter of this motion. If there are now enough supporters for the motion to be called to a vote, the stage is changed to `OPEN`.

Any party member has the right to run for office by calling `runInElection(electionID, name)` with the ID of the respective election and the member's unique name. The respective election needs to be in the `PREPARED` stage and the candidate is not allowed to be added multiple times. A new `Option` will be created with the candidates name stored in the `text` variable.

```
function vote(uint Votingid_, uint optionid_) external{
    require(participants[msg.sender].role
        >=ConventionRole.ALTERNATEDELEGATE,
    "Only (ALTERNATE)DELEGATEs can cast votes.");

    require(votings[Votingid_].stage == VotingStage.OPEN,
    "Voting is not open for voting.");

    require(participants[msg.sender].votingRight1from!=address(0)
    || participants[msg.sender].votingRight2from!=address(0),
    "No active voting rights held by sender");

    require(participants[msg.sender].votingRight1from!=address(0)
    && notContainedInArrayAddr(votings[Votingid_].voters,
                participants[msg.sender].votingRight1from)
    || participants[msg.sender].votingRight2from!=address(0)
    && notContainedInArrayAddr(votings[Votingid_].voters,
                participants[msg.sender].votingRight2from),
    "Vote(s) were already cast");

    if(participants[msg.sender].votingRight1from!=address(0)
    && notContainedInArrayAddr(votings[Votingid_].voters,
                participants[msg.sender].votingRight1from)){
        votings[Votingid_].voters.push(
            participants[msg.sender].votingRight1from);
        votings[Votingid_].options[optionid_].voteCount++;
    }

    if(participants[msg.sender].votingRight2from!=address(0)
    && notContainedInArrayAddr(votings[Votingid_].voters,
                participants[msg.sender].votingRight2from)){
        votings[Votingid_].voters.push(
            participants[msg.sender].votingRight2from);
        votings[Votingid_].options[optionid_].voteCount++;
    }

    if(votings[Votingid_].voters.length >= votingRightsCounter){
        votings[Votingid_].stage=VotingStage.CLOSED;
}  }
```

Figure 7: Implementation of vote()

Guest Function

If a voting process is in the `CLOSED` stage, anyone can request the result via `seeResult(votingid):result`. The returned string will list all options and the amount of votes they received.

6 Evaluation

This smart contract enforces **Uniqueness**, as any voting right can only be used once per voting. **Data Integrity** is implemented as it is impossible to alter a vote once it is cast. **Eligibility** is enforced, as only (alternate) delegates can hold voting rights. **Robustness** is achieved in the sense, that any delegate can reclaim their voting rights without having to actually contact/locate the current holder. **Reliability** is achieved in the sense, that anyone can be sure that any participant can hold a maximum of two votes. In addition administrative steps are automated as much as possible and do not rely on human interaction if not needed. **Verifiability** of the final result is given, as anyone can call the function to see the final results. **Fairness** is partly achieved. It is achieved in the sense, that the results function can only be executed once a voting is closed. However, previous votes are recorded on the blockchain and might be seen by technologically-sophisticated convention participants. **Practicality** in its functionalities is this smart contract's main strength as it has uniquely been developed to fulfill the specific requirements of party conventions. Yet, compared to paper based solutions this system might be more costly as transaction fees (gas) is required. Also, the creation of Ethereum wallets for each participant is surely an overhead.

Voter Anonymity and **Receipt-Freeness** are in conflict with **Accountability** (meaning that party members can see how their delegates cast their votes). Both can not be fulfilled at the same time. This smart contract has settled at a compromise. It does not directly store who voted how by separately storing who already voted (*voters* attribute in *Motion*) without storing how they voted. The *voteCount* attribute in *Option* only stores the current number of votes cast for this option. Yet, the casting of votes as a transaction is obviously

stored on the blockchain.

7 Conclusion

In this paper the author has designed and implemented a secure and practical voting system on the Ethereum blockchain for conventions held by e.g. political parties, companies or other institutions that require the election of officials as well as voting over motions. Different roles allow for the calling (and successful execution) of different functions as defined in the respective rules of procedure. The smart contract limits the trust of participants in admins, as certain administrative tasks are performed automatically by the smart contract. For example, motions are closed automatically after everyone has voted.

A middle man / trusted third party is no longer needed, e.g. for the creation of new motions. Delegates with voting rights can temporarily pass on their voting right. But with one click (the call of a function) they get their voting right returned.

Future work may incorporate more complicated forms of elections, e.g. secrecy/anonymity. In addition, the concept of fairness could be extended by implementing a more elaborate commitment scheme with hiding functionality of cast votes that are only revealed after voting has ended. Further optimization (by reducing required gas) in the source code may be possible.

References

[1] D. Chaum, A. Essex, R. Carback, J. Clark, S. Popoveniuc, A. Sherman, and P. Vora, "Scantegrity: End-to-End Voter-Verifiable Optical-Scan Voting," *IEEE Security & Privacy*, vol. 6, no. 3, pp. 40–46, 2008.

[2] G. Ofori-Dwumfuo and E. Paatey, "The Design of an EElectronic Voting System," *Research Journal of Information Technology*, vol. 3, no. 2, pp. 91–98, 2011.

[3] R. Taş and m. z. Tanrıöver, "A Systematic Review of Challenges and Opportunities of Blockchain for E-Voting," *Symmetry*, vol. 12, no. 8, 2020. [Online]. Available: https://www.mdpi.com/2073-8994/12/8/1328

[4] U. Jafar, M. J. A. Aziz, and Z. Shukur, "Blockchain for Electronic Voting System—Review and Open Research Challenges," *Sensors*, vol. 21, no. 17, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/17/5874

[5] F. . Hjálmarsson, G. K. Hreiðarsson, M. Hamdaqa, and G. Hjálmtýsson, "Blockchain-Based E-Voting System," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 983–986.