Developing a Distributed Distributed Consensus Protocol Consensus Protocol

Alec Grieser
ahg3de@virginia.edu

December 12, 2015

1 Motivation

Many of the problems facing the future of Bitcoin today, including most notably the problem of scaling Bitcoin as it sees increased adoption by the general public, must contend with an equally vexing preliminary obstacle, which is that any significant change to the Bitcoin protocol can result in unwanted forks in the block chain, which can undermine the currency and lead to problems in the system. We have already seen some changes to protocol as it is, but we can expect more changes in the future both from proactive decisions to change the protocol to make improvements to it and also reactive decisions to plug bugs and fix vulnerabilities. Therefore, it is important that even as the exact parameters of the Bitcoin protocol change that there be consensus on what the latest version is and what the latest version should be.

It is this problem that the research done here is attempting to address. We can boil down the more abstract problem posed above—how do we maintain consensus on the protocol being used—into a few different sub-problems:

- How is the protocol specified in a clear, unambiguous way?
- How do updates to the protocol get propagated through the network?
- How can interested parties—miners, full nodes, users, etc.—be certain that an update has been accepted?
- How does the community decide to accept an update to the protocol?

In answering this, one response could involve a centralized party (for example, the developers of the Bitcoin Core code) that makes all of the decisions regarding the future of the currency. However, in the spirit of Bitcoin, it would seem more appropriate if we are to find a distributed solution to this problem. As Bitcoin is perhaps the greatest distributed consensus protocol ever deployed, in the end, what we really are searching for is a distributed consensus protocol that determines the exact parameters of another distributed consensus protocol.

2 Specification

Software specification is a non-trivial software engineering problem in no means peculiar to Bitcoin. The specification for Bitcoin is currently a little dispersed, and nothing that I could find was strictly official. There were a couple community attempts at creating documentation for the protocol, such as the documentation offered at Bitcoin Wiki, a developer reference on the Bitcoin.org website, and an open source GitHub project called Bitcoin-spec that is putting together a comprehensive PDF specification written in LaTeX. Now, while these may in fact suffice for a formal specification for this class, I want to propose here what may be a better design decision for what Bitcoin appears to need today. Now, specification is a difficult Software Engineering problem, and I don't propose to have a general solution, but I do have a proposal that I think would work for Bitcoin at least to the extent that is necessary for this project. But before I go into my proposed solution, I would like to explain a few properties that I believe that the specification should have if it is to be useful to the community:

- **Modular design** - This specification, as much as possible should be broken into semi-autonomous pieces that allow for proposals to be made that only affect one small part of the protocol. For example, one should be able to easily make a proposal that only affects the maximum block size without having to worry about the change affecting (explicitly) the other parts of the protocol.
- **Comprehensive** - The entirety of the protocol should be included in this document so that there are no hidden "unwritten rules" that underpin the system but aren't explicitly part of the protocol.
- **Flexible** - The format should be able to handle arbitrary changes to the protocol. This is to guard against some fatal flaw/Achilles' heel in the protocol from being baked in due to the rigidity of the specification document itself.
- **Unambiguous** - There should be no difference of opinion as to what the underlying protocol means and therefore what is necessary to produce a "correct" implementation. The language of mathematics produces perhaps the clearest and least ambiguous way of expressing certain notions that is known to humanity (except perhaps Lojban), and it should be used as much as possible.
- **Disseminable** - Perhaps this is taken as a given in the digital age, but it should be easy to publish and acquire the specification document, so it should be in some format that is easy to send back and forth. Formally, any piece of information expressible as binary bits can be sent between computers, but certain formats will lend themselves more to widespread publishing than others.
- **Verifiable** - This property is more particular to Bitcoin and other distributed systems than traditional systems in need of formal specifications, but there should be a mechanism for nodes in the network to verify that a particular specification document is the real one actually being used in the network rather than an ersatz imitation presented by a malicious node.

Now, if one examines the properties listed above, one thing that may pop out is that two properties in particular—the requirements that the specification be flexible and that it be unambiguous—seem a little contradictory, especially if the solution to the ambiguity problem is mathematics, a "language" that is, if powerful, not nearly as expressive as natural language, even if it does have the property that it can generally be agreed on what it means by large segments of the population. I acknowledge that this tension does exist, which is part of the reason why software engineering is hard, but there is a compromise to be had between the two positions. In short, what is best written in mathematics can be written in mathematics and what is best left to natural language can be written in natural language. For example, if we wish to specify the maximum block size or a hashing requirement or a schedule for mining rewards, we note that each of these are either numeric or determined by mathematical formulae, so this may be best left to mathematics. However, if we wish to add semantics or explanation to the protocol, spicing it up with natural language is probably the best bet due to the difficulty that one might encounter when trying to convey that kind of information through mathematical techniques.

2.1 Proposed format

With this in mind, I turn now to my proposed format. I will note that my way is not the only technique that could meet all of the criteria outlined above, and there may be other, competing sets of criteria under which my proposed format is unsatisfactory but other formats may be more appropriate. But I do think that this format offers some advantages that make it worth giving consideration. The format is thus: a JSON file can be used to define all of the different components (objects, if you will) involved in the Bitcoin protocol and describe their interaction. Fields in the objects can include, if appropriate, description fields, which allow for the designer to add arbitrary English strings to explain more fully a concept; size fields, which can specify how big (in bytes) a particular object can be; sub-objects, such as, for transactions, the lists of transaction inputs and outputs as well as associated elements like the locking and unlocking scripts, etc.; and more domain-specific fields for things like script, which may want to add fields to specify the affect an instruction will have on the stack or what arguments an instruction may take.

I will go into more detail as to how this format ticks the boxes outlined above, but I first wanted to give some examples as to what the specification might look like to give the reader a more intuitive feel for the proposal. With that in mind, here is what some snippets of a final specification might look like:

```
{
 "transaction": {
  "description":
   "one event transferring value from one set of outputs to another",
  "fields": {
   "inputs": {
     "description": "list of incoming txn_inputs containing source of value",
     "type": "list<txn_input>",
     ...
   },
   "outputs": {
```

```
      "description":
       "list of outgoing txn_outputs containing destination of value",
      "type": "list<txn_output>",
      ...
    },
    "lock_time": {
      "description": "block height or timestamp when transaction is final",
      "size": 4,
      ...
    },
    ...
  },
  "max_size": 100000,
  ...
},
"txn_input" : {
  "description":
   "one input into a transaction taking value from another transaction",
  "fields": {
    "prev_tx_hash": {
      "description": "hash of transaction containing the consuming output",
      "type": "hash256",
      "formula": "hash256(transaction)",
      ...
    },
    "tx_out_index" : {
      "description": "position of consuming transaction in source transaction",
      "type": "uint32_t",
      ...
    },
    "script_sig": {
      "description": "unlocking script for appropriate transaction output",
      "type": "script",
      ...
    },
  },
  ...
},
"txn_output": {
  "description": "out output from a transaction storing transferred value",
  "fields": {
    "value": {
      "description": "transferred value to output",
      "type": "uint64_t",
      ...
    },
    "script_pub_key": {
      "description": "locking script of transaction output",
      "type": "script",
      ...
    },
    ...
  },
  ...
```

```
},
"block": {
 "description": "batch of transactions and metadata for the block chain",
 "fields": {
  "header": {
   "description": "important metadata for this block",
   "type": "block_header",
   ...
  },
  "transactions": {
   "description": "list of transactions included in this block",
   "type": "list<transaction>",
   ...
  },
  ...
 },
 "max_size": 1000000,
 ...
},
"block_header": {
 "description": "header of block information for creating the block chain",
 "fields": {
  "version": {
   "description": "block version number",
   "type": "uint32_t",
   ...
  },
  "hash_prev_block": {
   "description": "hash of previous block in the chain",
   "type": "hash256",
   "formula": "hash256(block_header)",
   ...
  },
  "merkle_root": {
   "description": "hash of Merkle tree root of included transactions",
   "type": "hash256",
   ...
  },
  "nonce": {
   "description": "arbitrary integer",
   "type": "uint32_t",
   ...
  },
  ...
 },
 ...
},
"script": {
 "instructions": [
  {
   "word": "OP_DUP",
   "description": "duplicate the top of the stack",
   "opcode": 118,
   "args": "",
```

```
    "input": "x",
    "output": "x x",
    ...
   },
   {
    "word": "OP_HASH160",
    "description": "produce 160-bit hash of top of the stack",
    "opcode": 170,
    "args", "",
    "input": "x",
    "output": "RIPEMD-160(SHA-256(x))",
    ...
   },
   {
    "word": "",
    "description": "pushes data to stack equal in size to opcode",
    "opcode": ">= 1 || <= 75",
    "args": "x",
    "input": "",
    "output": "x",
    ...
   },
   ...
  ],
 },
 ...
}
```

From above, we can see the basics of how this specification document may be structured. We can define objects by their fields, and for fields we can specify types varying from standard C/C++ types (like `uint32_t`) or specific Bitcoin types that would have to also be defined elsewhere (like `hash256` for a 32-byte hash, probably double SHA-256 given that this is Bitcoin). Special attributes like `max_size` can be used to determine the maximum size of an object, so if one wanted to propose a change to the protocol increased, for example, the maximum block size, all one would have to do is produce a new proposal that changes that number (equal to 1,000,000 in the above sample mini-specification) to something larger or even a formula. Likewise, if one wanted to update the hashing algorithm to, for example, SHA-3 if SHA-256 gets broken, then one could produce a specification that simply changes the hash algorithm used in the instructions for the `OP_HASH256` instruction.

One final note about the example above: it doesn't, at present, attempt to differentiate between the different types of things that could be defined, e.g., larger units like blocks or transactions, smaller types like `uint32_t`, or something more abstract like the script language definition. In the final protocol specification, it may be useful to distinguish between these different kinds of things, so I don't claim that the specification above is necessarily in its final form, but I do think it serves as a useful, non-trivial starting point for developing a final specification for the entire protocol.

2.2 Fitness of solution

So, does the JSON format have the properties we wanted it to have when we began discussing what would make a good specification system? Well, let's consider them in turn:

- **Modular design** - This was, in some sense, the most compelling reason for choosing something like JSON. Essentially, specifications for different parts of the protocol can be separated into different objects. Then changes can be made by changing the appropriate object (and possibly fixing some changes to related objects that may be necessary to handle ripples). But the exact parameters of the system are still fairly well separated into different, independent sections of the document, which was the goal.

- **Comprehensive** - No technical limitation should be able to stop the JSON document from covering the entire protocol except maybe that one could argue that some meta-knowledge of the system might be necessary to understand how the entire system fits together. I grant that this could be somewhat difficult, but also by extending the system to include more, including possibly a section where one just writes up how these different parts fit together, one can contain even this meta-knowledge in the specification itself.

- **Flexible** - The JSON format can be extended to handle any situation by adding additional JSON objects to the list and by creating new fields for existing ones as needed. It may be helpful to have an additional section in the JSON specification which puts some structure on the content of an object for clarity or comprehensiveness, but these strictures would be part of the protocol rather than part of the technical limitations of the JSON format.

- **Unambiguous** - By encoding mathematics as a string, the JSON format can, if it chooses to, incorporate completely unambiguous mathematical language, which is certainly a plus. There is also the possibility to include clarifying remarks through something like the `description` field on an object, which may, along with the formal mathematics, clear up ambiguity, but we do have to be careful with that, as one could just as easily introduce ambiguity or contradictions through natural language.

- **Disseminable** - Another particularly strong aspect of using JSON is that there is already a fair amount of web infrastructure dedicated to do parsing, creating, and transmitting JSON objects, with several internet APIs already making use of it, and JSON handling functionality available in most popular languages either as a first party citizen or in popular libraries. This should make the friction for figuring out how the document will be sent out somewhat less than it otherwise would be.

- **Verifiable** - I will go into this in more detail in the next section, but the idea is that this ends up being just a text file that we then have to come to consensus over, which means that there has to be a way for someone in possession of this file that they have received themselves can determine that this is really the file that everyone else is referring to. This is not a trivial requirement, but it is possible through some standard cryptographic techniques (involving hashes) that I will discuss more in the next session.

So the proposed solution does meet a fair number of the requirements that we have set for it. The question is then how would we structure network communication so that we can both change the contents of the document and maintain consensus on what the most recent version of this document is. This will be the topic for the next section.

2.3 Sidebar: Using binary or source as the specification

One question that the reader may have is why we don't just use the binary or source code as the specification itself (and then we can use the hash of the code/binary in the following sections). While this would work in the sense that if we could get everyone to agree to use a particular version, then there would be some amount of consensus on what protocol would be used, and then the following sections on updates to the protocol could be reduced to submitting and agreeing to patches to the code. And to a certain extent, this is essentially how the Bitcoin network reaches consensus on its own protocol currently, with a large percentage just agreeing to use the Bitcoin core code.

However such a system would have the following problems. First, if the code in question is made to be platform-specific, then this will exclude anyone on other platforms from joining the network, which may limit slightly limit the number of full nodes. Right now, Linux remains a pretty popular solution for servers, and even those using OS X or Windows servers can at least run a virtual machine containing Linux, but there is no guarantee that this will be the case in the future, so it may be smart to "future-proof" the specification by making it platform-agnostic.

Secondly, a specification system that uses the actual source or binary as the specification would suffer from the deficit that the entire network would be committing itself to a particular implementation of the underlying protocol, which means that if there is some problem in the code that remains un-noticed until it starts to cause issues, then the whole network will experience the problem at once. For example, if the Bitcoin core code fairs poorly under high-traffic (because it doesn't load balance well, for example) and begins to drop transactions accidentally or crash, then if traffic increases on the network overall, the entire system could begin dropping transactions and become disconnected or spotty. Furthermore, if there is a security vulnerability in the core code, then an attacker who figures it out could infect potentially all of the nodes. Depending on the severity of the vulnerability, this may allow an attacker to double spend on the network successfully, gain an advantage in mining blocks, deny service to certain individuals (effectively making their Bitcoin value-less), or even bring down the entire network.

This would, of course, be unfortunate, so it should probably be the goal of the Bitcoin community in general to have a few different clients, each of which has a significant share of the nodes, to prevent this attack, though any attempt to do so would have to be careful to make sure that all of the different versions could inter-operate and produced identical inter-node messages. This isn't an easy challenge, but it is probably better than having one client monopolize all of the CPUs. If the code itself becomes the specification, then this is impossible to manage, but it is possible with my proposed implementation-agnostic solution.

3 Maintaining Consensus

Essentially, we have no solved—or proposed a solution—for finding an object on which consensus can meaningfully be created by trying to map the amorphous and abstract notion of a protocol onto bits that can be passed back and forth. The next hurdle would then be to figure out a way by which nodes may maintain agreement and notice when they have to upgrade themselves because of a change to the protocol as well as be sure that an upgrade is necessary. The problem of determining whether a change should be accepted or rejected by the community will be tackled later, but for now assume that this is possible and all that we want to ensure now is that the change will be propagated through the network.

When tackling this problem, I will consider several problems that together add up to the larger problem that we are trying to solve. This includes (1) verification of the current version used by the network by any node in a reliable way and (2) determination of a node that it needs an upgrade. Once the node knows it needs to upgrade, it can then request one from whatever vendor it got its original software, either automatically or manually depending on how intricate a client may be. It would then be on the vendors to upgrade their software to meet the new specification, which is not a trivial task, but it is, at its heart, divorced from the one of setting the specification itself.

3.1 Network version determination

Once we have a specification document, there are several different places in the protocol that one can embed information about this document in the network, Two places that are pretty handy candidates are the version fields for both block headers and transactions. Each of these are currently 4 bytes, so neither could contain full 256 bit hashes at the moment, but one could embed checksums generated from the applying SHA-256 (or, in true Bitcoin spirit, double SHA-256) to the specification file in these fields as something like the last 2 bytes, for example. In the future, one could imagine adding an additional field to the block header or transaction format that allows for full hashes of the specification to be held in addition to the version field (or in lieu of it even), but that would be a fairly significant change that probably isn't worth pursuing right now.

Now, a checksum would not be enough to strongly guarantee security if it were used alone. There are only 4.29 billion integers that are expressible with only 32 bits, so even if all 4 bytes of the version number are used to encode the checksum, a potential attacker could probably brute force a collision in a reasonable amount of time and attempt to convince you that the protocol is something other than the actual one used by the rest of the network. However, the checksum only has to signify that something has changed in the protocol rather than expressing the change itself, so even if we limit the number of bits used in the version parameter to 16, there is still only a 1 out of 65,536 chance that two particular protocol hashes will share the same checksum value, though one should be mindful of the birthday paradox that would expect to find a collision half of the time even if there are only (roughly) 256 different checksums created. This isn't the best, but it will probably suffice until the protocol is updated to use a larger spot for the protocol hash if all it has to do is signal to nodes who join the network midstream that something has changed.

What will really be used to guarantee version accuracy is that the full specification

hash has to be included in the blockchain for it to be valid, and the value of this hash can't be changed by would be attackers (without a large amount of computing power). The exact way that this hash is included in the blockchain will be specified later, but for now, it is enough to know that it exists and that there is a particular transaction (in a particular block) where the node can look. If the node receives bogus data from a would-be malefactor, it can request this node for the transaction where the new specification has been included. If it doesn't get a response from this node (or others in the network), it can be sure that the specification is false and there is no need to upgrade.

3.2 Node Upgrades

The final piece of this puzzle is then the mechanism by which the node verifies the version of the software that it itself is currently using. For this, there are two basic options: (1) clients, including light-weight clients, can include the specification document itself, or (2) they can include only the specification hash. Hopefully, full nodes at the least would include the full document, as possession of this document will allow the node owner to investigate the contents of the document, which will be helpful in the next section on how the network protocol is to be changed, but all that is strictly necessary to verify that the current client is on the latest version is the hash of the specification, which I suspect would be the method most Bitcoin light clients would use given that it is significantly less data to have to store locally.

So the order of events would thus be this:

1. A node wakes up after being dormant and missing information.
2. It updates its own information and notices that blocks/transactions are running an unrecognized version.
3. The node then requests for the latest version data from its neighbors including a full hash in the blockchain.
4. The node then knows to request an update from its software vendor for a newer version of the client that can handle the new protocol.

As a final note, I should remark that none of this solves the problem of verifying that the software one receives does what is advertised, and it would be trivial for a mischievous attacker to write some malware into a Bitcoin client. The attacker, if they then also followed the rules about including the specification hash or document, would remain undetected just as before. Verifying that *software* is correct and virus-free is a hard security problem that I am not proposing to solve. All that my solution claims to do is to provide a way for an honest node to monitor the system and determine whether their software nominally meets the specification that the network is currently working with.

4 Changing the Protocol

While the preceding ideas are perhaps interesting or at least useful from a pure software engineering point of view and and perhaps useful if one were to design a system where power

and decisions were made at the top by a controlling few, they don't even attempt to answer the question as to how one might go about getting the network to agree to changes in a publicly-accessible way. This is the subject for the rest of the proposal, and it will involve creating a way for the users of the system within the network to express preference as to how the protocol should change and evolve.

The heart of the proposal is this: the users of the system should be the ones who have ultimate control over the direction of Bitcoin itself. Ultimately, it is the users who will determine if Bitcoin lives or dies anyway, as if the cryptocurrency fails to meet their needs, they can always move on to a different system which better suits them. If we wish to get the user's opinions, we could do so in a couple of different ways. For one, we may want to try and give one vote to every user and have votes tallied on new changes by nodes in the network. But then we have the problem of figuring out how many people there are and how many control multiple address, etc., not necessarily an impossible task with Bitcoin, but an error-prone one usually at best. One could imagine requiring some kind of either secret but centralized system or distributed but open method of identification proving identity, but this introduces serious concerns involving either centralization (and trust) or a risk at losing anonymity, a key feature (rightly or wrongly) for many users.

So rather than adding to the system some separate test for identity, it would perhaps be preferable if we could instead find some already limited but fairly widely distributed quantity within the Bitcoin ecosystem to substitute as a form of quasi-identity. Now, there are only a few different candidates, as many elements of the Bitcoin system are designed to be numerous rather than limited. For example, one can create an unlimited number of public addresses, or create an unlimited number of transactions, so neither addresses nor transactions can be used for this purpose. There are a limited number of blocks, in that there is only about 1 created every 10 minutes, but using blocks as votes has its own problems. For one, there are only a handful of mining pools, so if those pools either colluded together or simply naturally arose on agreement on certain issues, they could easily override the wishes of the general populace. More generally, the number of miners within the Bitcoin system will tend towards a vast minority of the users as it becomes more popular and mining becomes more specialized. So if we wish to make sure that majority of the community has a say, then it probably won't work.

The next logical element of the system to attempt to use would then be Bitcoin itself, with one vote given for each bitcoin (or, equivalently, for each satoshi). There are several advantages and disadvantages to such a system, but the basic advantage to be gained is that there can only be a finite number of them in the world at any given time, they have intrinsic value and are thus hard to acquire if one wants to spam the system, and possession of bitcoin, in some sense, proves that one has stake in the system and can be viewed as kind of analogous to stocks in a corporation. In corporate settings, it is usually expected that one will get one vote for each share, so it shouldn't surprise us if one bitcoin is one vote in this system. I therefore propose a system where one can publicly declare that one is setting aside a certain amount of bitcoin (agreeing not to spend it during the voting procedure) and along with that, publicly broadcast a vote for a new protocol.

So, if there is a way to use Bitcoin to vote, we will try and use that. Then the system has to solve a few problems:

- How does one propose changes and thus call for a vote?
- How exactly are these votes recorded on the blockchain?
- What is needed to "win" a vote?
- How do we determine that a vote has ended?
- How are the results from the vote reported?

I will attempt to address all of these questions in the forthcoming sections, but I will not address these strictly in order. The most technically interesting question involves the problem of recording the vote in the blockchain. I will thus address this first, which will also be useful in determining the way votes will be proposed in the blockchain. The rest of the protocol ends up being more about constructing a workable "constitution"-type structure rather than solving technical problems and will thus be handled last. Finally, I will go over some potential problems with this system and attempt to provide useful analysis on the system as a whole.

4.1 Casting one vote

The essential strategy used here is to have a two-phase voting procedure, the first phase of which will feature publishing a commitment to the blockchain so that everyone can vote in secret. Then there will be a second phase where the commitment pre-images are revealed so that the votes can be counted from the public ledger.

So, where should this information go? Well, there isn't actually that much space for the information to go within a transaction, as almost all of it is set by the protocol, and proposals to use the blockchain to do things like pass messages (for example, the Ombuds project) have to essentially hack around these inherent limitations in the protocol to pass along interesting information.

One potential place for this that I propose would be the locking and unlocking scripts. I propose this for two different reasons. First, these scripts can be greatly extended, at least in theory, so there is room here to add the new data. But secondly, by placing this information here, we can enforce through technical means requirements such as that, in order to make the vote meaningful, the bitcoin used as part of the vote can't be spent until after the election concludes and the votes are revealed.

So, what should the locking script look like? Well, as a preliminary proposal, one script that looks promising might be a type of locking script known as a transaction puzzle. (See Bitcoin Wiki.) In general with transaction puzzles, the approach is to publish a hash to the blockchain and require that someone produce the pre-image of this hash to unlock the given transaction output. In our case, we will want to publish a hash that commits a vote to the chain so that one has to reveal the vote in order to get back the original coin. With that in mind, here is how such a locking script might look:

```
OP_HASH256
OP_DATA SHA-256(SHA-256(vote_id || specification_hash || nonce))
OP_EQUAL
```

Above, the second line of the script is what contains the commitment itself. There are three parts that I have identified as useful to have: (1) the `vote_id` is an ID to group of all of the votes cast in the same election together to help with counting votes after the reveal, and it is published to the blockchain by the person who initially called for a vote; (2) the `specification_hash` is the hash (let's say double SHA-256) of new protocol specification that this vote is supporting; and (3) a random `nonce`, 32 bytes in length, serving as a way for the person casting their vote to make finding the pre-image non-trivial. Note that we are taking double SHA-256 of the vote to create the commitment—I don't claim that there is any technical reason why one should do this, but the `OP_HASH256` opcode in script performs double SHA, so it's easier to go with the grain and use this as the hash rather than using single SHA, for example.

Now, once this transaction is created, it should be clear that one would need to publish a transaction where the unlocking script is the following to get back the value set aside as the vote for the duration of the election:

```
OP_DATA vote_id || specification_hash || nonce
```

This reveals the vote, which means once everyone publishes their commitment pre-images, all of the votes can be tallied and a winner declared. Furthermore, this unlocking script has the property that one has to know the nonce for it to be successful, which means that it is (we think) very computationally difficult for any attacker to determine the vote and unlock the transaction pre-maturely. These are all great properties, and it looks like this script has real potential.

However, there is one fatal flaw to this locking-/unlocking-script pair. In the transaction where one reveals the pre-image, one probably will want to reassign the coin used during the vote back to the original owner. But the transaction unlocking script contains no signature, so once the pre-image is revealed, all secret information needed to make a valid transaction spending the coin has been leaked, so a would-be attacker can create a new transaction sending the bitcoin to them rather than the real owner. (This would look identical to a double-spend attack on the network, and the attacker is guaranteed to be able to get their transaction included in blocks faster, but neither is the real owner—and the attacker may be willing to pay higher fees.)

To overcome this, I propose the following modification: combine the above transaction puzzle script with the standard Pay-to-Pubkey-Hash script. Here is what that would look like:

```
OP_HASH256
OP_DATA SHA-256(SHA-256(vote_id || specification_hash || nonce))
OP_EQUALVERIFY
OP_DUP
OP_HASH160
OP_DATA pubkey_hash (public_address)
OP_EQUALVERIFY
OP_CHECKSIG
```

As one can see, the first three lines of this new locking script are exactly the locking script earlier proposed, and the remaining lines are exactly the Pay-to-Pubkey-Hash script that is currently the most commonly used Bitcoin locking script. Then the appropriate unlocking script would be the following:

```
OP_DATA signature
OP_DATA pubkey
OP_DATA vote_id || specification_hash || nonce
```

Then just as before, the vote is revealed publicly to be tallied, but this time, an attacker would have to be able to generate a valid signature for another transaction if they wanted to redirect this new transaction somewhere else. I would also add that this system means that while secrecy of the underlying vote during the first phase is something that this system would allow, the presence of additional hurdles locking a transaction means that it is not strictly necessary, so anyone who votes is free to publish their nonce and their preference, and then everyone can verify that they are telling the truth. This may not actually be a positive (there are some benefits to a completely secret election in that it is harder for the election to be biased by early votes in one way), but it is an interesting feature of this system.

4.2 Proposing a change

Now that we have a mechanism for creating a transaction that has with it some information (in the above discussion, a vote commitment and then pre-image reveal), we can modify these scripts to include signal to the network that a vote is being called where a specific new protocol is being proposed. Our basic strategy in this will essentially be to modify the script by removing the hashing component so that the information being passed can be read both when the locking script and when the unlocking script are published to the blockchain.

So, here is what the locking and unlocking scripts would look like. Note that the person proposing the change gets to choose the `vote_id` value, and it is this value that all valid nodes in the network wishing to vote will have to concatenate with the specification hash and nonce. The only requirement is that this number be unique for each election so that votes can be easily separated and tallied. Also, the `specification_hash` here is the hash of whatever the new protocol specification will be should the network agree to it, which will also be decided by the person proposing the vote. The final parameter, `vote_type`, I will go into more detail later, but it will determine how long the voting period will be, when the voting starts, how many people are necessary to participate, and what exactly is required for the new proposal to succeed.

So, this leads naturally to the following locking script:

```
OP_DATA vote_id || specification_hash || vote_type
OP_EQUALVERIFY
OP_DUP
OP_HASH160
OP_DATA pubkey_hash (public_address)
OP_EQUALVERIFY
OP_CHECKSIG
```

Note that the actual specification document is not included in the information published, with the idea being that it is on the proposer to send out through other means (such as standard message passing, online fora and discussion boards, carrier pigeon, etc.), but this way, anyone who receives a supposed specification document can verify that the document is actually what is being proposed by comparing its hash to the one that has been published in the blockchain.

Now, we have most of the technical elements laid out for working proposing new specifications and for voting on them on the block chain, but it still remains to be seen how we can fit those together to form a coherent system. To do this, there need to be a set of rules put in place to determine what exactly is needed to call for a vote for a new proposal, and once a vote is called, what exactly is needed to determine if it has been accepted or not, including the time for elections, the minimum participation rate, and the total margin of victory required.

When voting was discussed, there was no minimum amount (except perhaps the global minimum for the amount of value that has to be placed in a transaction) placed in a single vote needed for it to count, and this isn't necessarily the best decision (if one wanted to prevent spammers, for example), but it does however also make some modicum of sense not to attempt to limit the franchise to only those with a certain level of wealth if we wish to keep as many people eligible to vote as possible (and therefore less likely to become effectively centralized). However, when it comes to proposing new specifications for the Bitcoin protocol, due to the effect that a proposal can have on the system with more transactions being added to vote and nodes being forced to tabulate winners, along with the general desire to make changes fairly rare in order to keep the currency and protocol stable, there is a strong incentive to limit the number of proposals to change the system that get a vote.

For this reason, it should be the case that there is some minimum amount that can be in a proposal transaction if it is to be the case that these are relatively rare. An effective limit may be something like 100 BTC, a significant amount that one wouldn't put on the line lightly, but is also not an impossible quantity to raise. Furthermore, there is no reason why these 100 BTC have to be used in a single transaction to call for a vote. One could imagine splitting this burden across multiple in order to crowd-fund a new proposal, which might help democratize the system.

When it comes to election timing and victory margin, I propose that there should be at least two different tracks under which changes can be made based on the urgency of a proposal. The main difference would be time given for an election and the margin of victory required. For a standard vote, I propose that there should probably be about a week (1008 blocks) between a vote being called for and the first stage of the vote being called. Then there should be about a week (again, 1008 blocks) for interested nodes to produce the transactions containing commitments. Then, after that, there should be another week for nodes to publish transactions revealing their votes. Following that, a winner can be declared. For the new specification to prevail, it seems fair to require that there is a minimum participation rate expressed in terms of the total amount of bitcoin used in the vote. The exact best numbers for this would require more thought, but something expressed either in terms of some percentage of the total number of bitcoin available—something like 1%—or in absolute terms like 100,000 BTC should work. This requirement shows that there is some agreement of those in the network that this is an idea that has been given thought and consideration by the community. Furthermore, a super-majority of the votes in the range of 60% should be necessary to change the protocol in order to encourage stability in the face of a close vote. Alternatively, one could combine these into a single requirement that in order for a new proposal to be adopted, it should have the backing of a super-majority of those who vote as well as votes totaling at least some target amount (e.g., 100,000 BTC or 1% of all

bitcoin that have been created).

The second track would then be reserved for emergencies—situations such as what happened with the `OP_RETURN` bug where a failure to correct the protocol could result in serious problems for the network. In this situation, the turn around time for votes would be much shorter—something like a day rather than a week—but the decrease in time is exchanged for an increase in votes. In this case, one would require something like 90% of all votes cast to support the change, as would be the case for security updates. This aims to address problems of the whole process being too rigid for certain kinds of updates while the standard voting procedure aims to remain fairly rigid to avoid having changes made to the system too often.

Each of these types of votes can be given a code within the protocol specification that is published along with the `vote_id` and `specification_hash` parameters. The network would then use this to determine which of the above voting procedures exactly is being used with any particular proposal. In any case, following the end of the election, there should be a block mined in one of the blocks following the end of the voting that announces the winner for the network to verify within the Coinbase parameter. This is the way in which there can be a publicly visible way that nodes in the network can later use to verify the current protocol version.

As a final remark in this system, one might note that this whole procedure is rather involved and has a few different moments when we could imagine tweaking the system or perhaps a place where a design bug would need to be fixed. To handle this, we could create a distributed distributed distributed consensus protocol consensus protocol consensus protocol to make sure consensus is reached by the network on what the exact parameters of the distributed distributed consensus protocol consensus protocol is... but that seems a bit much and leads to problems of infinite regression. ("It's consensus protocols all the way down.") To get around this, we can actually include the specification for *this part* of the protocol within our general specification document in addition to specifications of the elements of the protocol that are already in use. Then changing our voting structure can be done using the same infrastructure for changing the Bitcoin protocol proper. Then one runs into issues of meta-changes where one might imagine certain groups of users attempting to gain an advantage by bending the rules not only of the system but also of the rules that are then used to change the rules to make the system. But the alternative is either a proscribed set of rules that never change or an infinite sequence of protocols designed to keep lesser protocols in line, so given this backdrop, including the protocol-changing part of the protocol within the protocol specification itself seems like the least of all of the evils.

This completes a description of the newly proposed voting system that can be used to change the protocol in a decentralized way that can then be verified by any node in the network and thus allow for automatic updates that may not have been possible in the current system. I have attempted to summarize some of the advantages of this system so far, but there are several possible sources for problems that are certainly worth addressing. Before concluding, I will touch on these to give a more complete view of the proposal.

4.3 What could go wrong?

One major concern that may occur to the reader is whether the miners would go along with such a proposal and whether miners would agree to include transactions they disagree with within a block. To this, I would respond that miner participation is a major concern, as they are the ones who ultimately control the currency, in some respect. There is, however, also a feeling among some that the miners are primarily interested in decisions being made that further the network's health and continuation, and are often agnostic as to the exact solution (at least according to the founders of Ombuds).

But even if miners care and actively attempt to subvert the system by not including the votes they don't like, this still might not be as big a problem as may be supposed. Part of the impetus for including a week between voting rounds in the standard voting procedure is to allow for there to be a large number of opportunities for even small miners to get a block or two in within the pertinent period. This way even if a minority of miners disagree with a change, hopefully at one player will and can be counted on to include the transactions that the other miners are neglecting. Now, if the miners *universally* agree that a change to the protocol would be detrimental, then there would be little recourse for those who go against the miners, as their votes would never be included by anyone. But perhaps if *none* of the miners agree to a change, perhaps it shouldn't be added to the protocol given that this could encourage miners to leave the network or discourage new miners from coming online, so this may not be the worst feature of this system. If one likes, one could think of this as a natural check on the system by which miners are given a little more say on the way the election goes than an average user, but this matches their additional importance over average users.

A more serious problem with this system may lie in the one-bitcoin, one-vote policy that underpins the system. On the one hand, as has already been stated, this policy makes a modicum of sense if we think of Bitcoin as a corporate endeavor and owners of the coin as stock holders, and the Bitcoin does serve as a kind of "proof of stake" in the system and does stop spammers from joining the system. But we have to be careful here, as the natural consequence then becomes that the wealthy become *de facto* more powerful than others. While this has interesting social consequences in and of itself, it also has direct consequences when it comes to the degree to which this protocol remains decentralized as opposed towards tending towards centralization. For various reasons, capital is naturally conglomerative in that those with money can invest in the kinds of ventures that will allow for them to acquire more capital. So, in that sense, we may expect money to concentrate itself in the hands of a few. If power is then correlated with money, there is then an additional problem to the usual ones that worry those who are concerned about wealth inequality in that now votes and influence would also be concentrated in the few.

But on the other hand, there is one check that the general populace has that hasn't existed with many currency systems in the past. Namely, the *ad hoc* nature of Bitcoin and the lack of backing from any particular state means that at any time, the users can switch to another cryptocurrency or back to traditional, government-issued currency. While this transition wouldn't be costless, it would be considerably less expensive than a lot of currency transitions in the past and possible in ways that it just hasn't been possible in the past. And the exodus of users from the currency shouldn't be considered an element of first resort against centralization if those with power begin to bend the rules in self-interested

ways, but it should serve as a compelling reason for those with a stake in the system to make decisions that benefit the community as a whole rather than as simply optimizing the currency for themselves.

5 Conclusion

Ultimately, some solution to the problems of software upgradeability and protocol consensus will have to be reached by the Bitcoin network if it is to have any sort of longevity into the future. The needs of future users cannot possibly be expected to match up perfectly with those of the users today, and we will want a way to fix the problems with the system that are opaque to us now but become more obvious with age. To do this in a way that does not fork the blockchain and fracture the system so it becomes unworkable will be difficult, and I have proposed my solution. If adopted, it would put the future of the system in the hands of those who use it, for better or for worse, rather than in the hands of the few core developers who currently set what is in the core code and what gets excluded. This philosophy is ultimately more consistent with the founding principles of the Bitcoin project than current proposals being thrown around so far and hopefully would successfully make the protocol resilient to the needs of future users as well as the present. But maintaining a currency is hard and the future unsure, and ultimately, only time will tell as to whether a system built like Bitcoin to run in such a decentralized way can survive or whether it will be unable to keep itself consistent and run aground without a centralized source of power leading the way.