A Probabilistic Nanopayment Scheme for Golem

Paweł Bylica, Łukasz Gleń, Piotr Janiuk, Aleksandra Skrzypczak, and Artur Zawłocki

imapp, contact@golemproject.net

November 11, 2015

Abstract

We consider a setting where a payment is made by a single payer to a possibly large group of recipients, each receiving only a small sum in the order of $0.01. For such small sums transaction fees are relatively large even if we consider cryptocurrencies instead of bank-based transactions.

Both payers and recipients are expected to repeatedly take part in many payments, but subsequent payments of a single payer may have different groups of recipients. Also, we assume that payments result from activities carried out in a decentralised peer-to-peer network, so we would like to avoid solutions relying on any trusted third party. We cannot therefore use existing solutions that require a central server or assume that a payer makes a series of payments to a single recipient (Bitcoin micropayment channels).

Instead, we propose a probabilistic payment scheme in which only one recipient randomly chosen from a group of candidates is rewarded in a single payment. Our solution is based on Ethereum and is thus decentralised and avoids relying on a trusted third party. We describe an Ethereum smart contract implementing a lottery used to reward recipients and calculate the cost of running it. Depending on the variant of the protocol, this cost is either proportional to the logarithm of the number of recipients or constant. In both cases, the cost of running a lottery with 1 000 participants is similar to the cost of direct payments to 20 recipients.

1 Introduction

This work is part of research conducted for the Golem project[1], currently under development at imapp.[2] The aim of the Golem project is to create a global prosumer market for computing power, in which producers may sell spare CPU time of their personal computers and consumers may acquire resources for computation-intensive tasks. In technical terms, Golem is designed as a decentralised peer-to-peer network established by nodes running the Golem client software. For the purpose of this paper we assume that there are two types of nodes in the Golem network: **requester nodes** that announce computing tasks and **compute nodes** that perform computations (in the actual implementation nodes may switch between both roles). A requester node partitions a task into multiple subtasks, specifies payment for each subtask and recruits a group of compute nodes, each of which downloads and completes one subtask. This is again a simplifying assumption, since in principle a single compute node could complete multiple subtasks. Results of each subtask are sent back to the requester node. After the requester node collects the results of all subtasks, it performs the payment step.[3] This last step is the main focus of the paper.

Golem needs to handle a variety of tasks with different sizes and decomposable to a different degree. In particular:

- Tasks can be partitioned into a few or thousands of subtasks.
- Task value can range from one to thousands of dollars.

[1] http://golemproject.net

[2] http://www.imapp.pl

[3] In this scenario we assume that the compute nodes do not fail and eventually deliver the results and the requester node is willing to perform the payment step. The first assumption may be implemented for example by ensuring each subtask is computed by more than one node (in parallel or in sequence) and the second one may be enforced by a suitable reputation system for the nodes, but this is outside the scope of this paper.

- Payment for a single subtask may be as low as a fraction of cent ($0.01 is roughly the cost of electricity consumed by an average workstation using 200 watts for 15 min, assuming the price of electricity is $0.20 for 1 kWh).

The main requirement for any payment solution used in Golem is that it should be decentralised and not depend on any trusted authorities (banks, brokers).

We should not expect that after a payment for a subtask is made, another payment with the same payer and recipient will occur in a short time. That is, it may not be possible to accumulate payments made by one payer and transfer the larger sum using a single transaction. On the other hand, we may consider methods that assume recurring payments to a single recipient: a compute node does not necessarily have to be paid a tiny sum for each subtask but instead may be paid larger sums once in a while. The obvious requirement is that in the long run the node's income should approach the sum of subtask values that this node has computed.

The last requirement for a payment solution is that the transaction fees should not exceed 1% (up to an order of magnitude). Thus for example a payment for a task worth $1 and performed by 10 compute nodes should incur no more than few cents of transaction fees in total. This requirement rules out traditional methods such as bank-based transactions, credit card transactions or PayPal, since transaction fees they incur are in the order of $0.1, plus a percentage of the transaction value (see e.g. [11]), and paying to 10 compute nodes would multiply the fees tenfold.

It seems more promising to look at available payment schemes based on digital currencies such as Bitcoin [3], which we do in Section 2. In Section 3 we describe Ethereum, a platform for developing decentralised applications, and consider payment methods based on Ethereum. The bulk of the paper is devoted to one of these methods: the lottery payment scheme. In Sections 4 and 5 we describe the lottery protocol, in Section 6 we compare the lottery with other methods cost-wise, in Section 7 we analyse its probabilistic properties. In Section 8 we discuss some details of the lottery implementation. Section 9 contains concluding remarks. Technical details are included in the Appendices.

2 Micropayment Schemes for Cryptocurrencies

We first consider a naive solution in which the requester pays each node participating in the computation with a separate Bitcoin transaction. Although Bitcoin transaction fees are lower than for bank-based transactions, a transaction smaller than 0.01 BTC ($\sim$ $2.5 at the moment of this writing) requires a fee of 0.0001 BTC ($\sim$ $0.025) to discourage "dust" transactions that bloat the Bitcoin blockchain [5]. Such a fee may thus be higher than the transaction value.

A Bitcoin transaction may include many outputs, each to a different recipient, which seems a good match for one-to-many payments. We can therefore consider making a single Bitcoin transaction with a separate output for each of the participants. In this case, the flat fee of 0.00001 BTC for "dust" transactions applies only once to the whole transaction, so this is not an issue. However, the size of the transaction in bytes grows with each output and Bitcoin also has fees for large transactions: the default charge is 0.0001 BTC per 1 000 bytes. Using the formula for approximating transaction size from [5]:

$$\texttt{transaction_size} = 148 * \texttt{number_of_inputs} + 34 * \texttt{number_of_outputs} + 10$$

we may estimate that a transaction with 100 recipients will cost at least $0.1 and the cost will rise as the coins become fragmented and require a large number of inputs. However, as fees in Bitcoin are market-based rather than hard coded, miners may at some point start requiring higher fees for processing transactions [13]. According estimates in [2], miners should require a fee of at least 0.0032 BTC ($\sim$ $0.8) for including each 1 000 bytes in a block), to compensate for the fact that a larger block is more likely to be orphaned. Finally, we should point out that receivers of "dust" payments will bear the cost of spending the tiny coins they receive, due to higher fees for transactions with large number of inputs. This may discourage potential users from participating in Golem. For these reasons, Bitcoin transactions with multiple outputs may be suitable for tasks with a relatively high value (say above $10) or small number of participants. For other scenarios a different payment scheme has to be used.

For completeness we also mention Bitcoin micropayment channels ([4]), but this solution does not fit our setting as it assumes that a payer makes a series of payments to one recipient. There is also a number

of custom micropayment solutions such as Coinbase Tip [8](no longer active), or ChangeTip [7] but they rely on a trusted third-party that processes transactions off-chain, which we definitely want to avoid in Golem.

2.1 Probabilistic Payment Schemes

We next turn our attention to probabilistic payment schemes. The idea is that instead of paying $0.01 directly, the payer issues a "lottery ticket" for a lottery with $1 prize with a 1/100 chance of winning. The expected value of such ticket is $0.01. The advantage is that on average only one ticket in 100 will lead to an actual transaction. Such scheme is proposed e.g. in [16] and [20]. A possible implementation of this scheme may use a cryptographic hash function [9], for example SHA-3 [18]. First, both the payer and the recipient draw (or choose) their numbers n_P and n_R, respectively. Both parties initially keep their numbers secret. Then the recipient reveals $\texttt{hash}(n_R)$, the hash of n_R. In the next step the payer reveals $\texttt{hash}(n_P)$. Finally, both parties reveal their numbers (in any order) and use them to decide if the recipient receives the prize: the recipient wins if and only if $n_P = n_R \mod N$, where $1/N$ is the probability of winning. Note that even if the recipient learns n_P before revealing n_R, she cannot change her choice of n_R since she committed to the original value by revealing $\texttt{hash}(n_R)$ and it is considered unfeasible to find another number with the same hash.

Obviously, this probabilistic procedure does not guarantee that a Golem node is fairly remunerated if it only takes part in a small number of tasks. However, as the number of tasks increases, the node's real income from lottery rewards will approach the amount it would receive being paid for each task. For the requester node the situation is similar, but the probabilistic effects may be more problematic: suppose a requester node just joined Golem and now has to pay for its first task to 100 other nodes with tickets giving each receiver 1/100 chance of winning the total task value. Now, since each ticket is evaluated independently, there is a 37% chance that the requester will not have to pay anything but also a substantial 26% chance that at least two tickets win, and almost 2% chance that at least 4 tickets win. Therefore the requester should be prepared to pay 4 times more than expected, which may discourage users from joining Golem.

We can modify this scheme to make it more predictable for the requester by ensuring that among the tickets issued to reward the participants of a single task, exactly one ticket is winning. For example, if there are 100 participants then each of them has 1/100 chance of winning, as before, but the requester has guarantee that task value will have to be paid only once. In other words, after the task is completed the requester will have to organise a lottery for the participants, in which exactly one participant wins.[4] The drawback of this approach is that it requires a protocol that involves a large group of participants: the payer and all the recipients. Such a protocol is more complex than in a one-to-one scenario where each lottery ticket is evaluated separately.

In the rest of this paper we describe and compare several possible payment schemes (most of them briefly) in the context of Ethereum platform.

3 Ethereum

Ethereum [10] is a decentralised, blockchain-based cryptocurrency system, but also a platform for development of distributed applications. In Ethereum it is possible to implement arbitrary complex rules required by a payment scheme as **smart contracts** in a high-level programming language (we use Solidity [17], in this paper). This would allow us to extend payment rules in the future, for example by including fees for using proprietary software in a software-as-a-service model on compute nodes. Moreover, Ethereum does not suffer from coin fragmentation. This makes Ethereum our platform of choice for implementation of micropayment schemes.

Ethereum has a notion of global state which is stored in the blockchain. This state is, essentially, a collection of accounts, each with its unique address and balance of **Ether** (Ethereum's currency). An account may store some data in a persistent storage and may have contract code associated with it.

The global state is changed by **transactions**. Each transaction has a sender address and a recipient address and transfers a specified amount of Ether between these addresses. If the recipient's account

[4] This can be easily generalised to a constant number n of winners ($n > 0$) for a lottery, each rewarded with $1/n$ of the task value.

is associated with contract code, the contract is run as a result of the transaction. The transaction may carry additional data, accessible by the contract. A contract may call another contract, which may call yet another one, but this chain of execution has to start with a transaction sent by an external actor (a user). A contract has no way of calling any external service outside Ethereum.

Contracts are executed by the Ethereum Virtual Machine (EVM) [21]. Each EVM instruction consumes certain amount of gas which reflects a computational cost of processing the instruction by Ethereum nodes. Gas has to be purchased with Ether by the user who wishes to run a contract. Paying for gas is the analogue of transaction fees in Bitcoin. Gas price is market-based: each transaction specifies the maximum price its sender is willing to pay for a gas unit and miners may prioritise transactions based on this information. So to estimate the cost in USD of executing a contract we have to take into both the current average gas price and the price of Ether. On the other hand, in order to compare different payment schemes implemented a Ethereum contracts we can simply compare their cost in gas units.

Ethereum does not solve all our problems. For example, it provides reliability but not confidentiality, since all data in Ethereum state is public (so that everyone may verify all transactions). This may be an obstacle for implementing certain more complicated payment schemes. For example, a protocol for a lottery may require the requesting node and all compute nodes to choose secret values that will be used later to determine the lottery winner. Those values have to be produced outside Ethereum, which gives the requesting node a possibility to reveal its secret value to a chosen compute node making the lottery unfair.

Computation model of Ethereum is deterministic: the outcome of any transaction is always the same when it is performed in a given global state. This limits the possibility of generating random values in Ethereum. A common solution is to rely on future blockchain data as a source of randomness. For example, a timestamp or a hash of the header of some future block may be used to seed a random number generator.[5]

Ethereum contracts do not have any mechanism to schedule actions (such as calling another contract or reading a timestamp) for execution at later time. Therefore any payment scheme using Ethereum will rely on users willing to perform transactions required by the scheme's protocol. In particular, users have to be online during at least some phases of the protocol. They also must have enough Ether to pay for their transactions, which may be a problem for payment protocols that require not only the requester node but also compute nodes to take actions. This is especially severe in protocols with many participants, any of which can stop the protocol. Fortunately, mechanisms for providing economic incentives for users to take actions (deposits, rewards) can be easily implemented in smart contracts.

3.1 Infrastructure of Ethereum Contracts

Some of the problems listed above may be solved by smart contracts provided by Ethereum community independent of Golem. For example, there exist preliminary implementations of RANDAO (RANdom generating Decentralised Autonomous Organisation) which generate random numbers from users' input. Users are paid for providing inputs with money paid by users that consume random numbers. The two RANDAO implementations we are aware of are [14] (compatible with obsolete POC5 version of Ethereum) and [15] (more up-to-date).

Another smart contract that can be useful for our purposes is the Ethereum Alarm Clock [1] that allows users to schedule a contract call for a specified future block. Users pay for scheduling their actions and get paid for executing other users' actions on time.

Both the RANDAO contracts and the Alarm Clock contract rely on a two-sided network effect: users on the demand side (requesting random numbers or scheduling message calls) rely on users on the supply side (providing input or executing scheduled calls), but users on the supply side will participate only if there is enough users on the demand side to provide adequate financial incentive. At this point it is hard to tell whether the userbase for these contracts will be big enough to make them useful.

In Section 8 we discuss problems with solutions that produce random values based solely on user input, such as RANDAO.

In future we may benefit from other solutions developed by members of Ethereum community. We also keep in mind that Ethereum still evolves and new features may appear in core Ethereum.

[5] The hash of a future block header depends on future transactions and the output of a proof-of-work algorithm, and thus may be treated as a pseudorandom value, see [21], Sect. 4, on how it is computed.

3.2 Payment Schemes in Ethereum

In this section we consider several payment schemes that can be implemented in Ethereum:

Direct transfer (with batching). This is a naive approach in which the requesting node makes a transfer to each compute node participating in the task. Simple transfer costs 21 000 gas units for every user, which is about $0.001 at the time of this writing.[6] We also consider a variant that uses batching[6] to pack all payments in one transaction. In this case the cost of sending ether is 9 000 gas units per node plus the cost of sending all data to the contract (min. 2 000 gas per node). With current gas prices this method gives transaction fees below 5% for payments of $0.01 or more. To handle tasks with lower payments to a single recipient and to account for fluctuations of gas price we have to find a better method.

Subaccounts. Another approach is to create a smart contract that keeps in its storage account balances of every Golem node, as a map with user addresses as keys and ether balances as values. To receive payments, users must first register their subaccounts in the contract, which is an one time investment of approximately 40 000 gas. In the payment step, the payer sends a list of recipients and payment values to the contract, and the contract increments each recipient's subaccount balance by the specified payment value. Users can transfer ether from their subaccounts to "real" Ethereum accounts at any time, covering the cost of the transfer. In this scenario, the cost of a single transaction is 5 000 gas units per recipient (for modifying the storage) plus the cost of sending all data to the contract and additional integrity checks (see Section 6 for more accurate cost estimates).

Lottery. This is a probabilistic approach in which the requester organises a lottery in which only one compute node wins. In this scenario the cost is generated mainly by the size of data that needs to be transferred to the contract. This approach is described in Section 4 and its cost is estimated in Section 6.

We also mention two additional payment schemes based on Ethereum but do not attempt to estimate their cost:

Micropayment channels. Micropayment channels in context of Ethereum are described in [6]. They allow the user to send transactions off-chain. Unfortunately, as in the case of Bitcoin micropayment channels [4] this solution is designed for many 1-1 microtransactions between two given parties. The cost of opening a channel is quite large (at least the cost of adding 5 new storage fields 20 000 gas each) and channel endpoints might never cooperate again. Additionally, a micropayment channel requires a security deposit, so if a requester has to open many channels the deposits may sum up to a large amount.

Bank-set approach. Most micropayments protocols for P2P networks require some sort of a centralised institution (bank) that is in charge of approving transactions [12]. One idea is to replace this institution with a set of peers (an approach similar to the Karma protocol [19]) that keep track of transactions and user accounts. The requester pays the total amount of ether for each task into one joint Ethereum contract. If a user wants to pay out her share from her account, the nodes in the bank-set vote for or against the payout. Unfortunately it is not clear how to choose bank-set representatives, how Ethereum contract should know current representatives, how should they vote and how to design a reward/punishment mechanism for representatives. At the moment, this solution seems to be too complicated to be practical.

4 Lottery

Our proposed solution allows the payer to avoid making separate transactions to each of the participants. Instead, the whole task value is paid to only one participant, chosen randomly. This way, paying for the task requires only a small number of transactions, independent of the number of compute nodes involved.

The idea is to organise a lottery for payees, with the reward equal to the total value of the task. The lottery has only one winner, other participants get nothing. For i-th participant the probability of winning is set to v_i/v, where v_i is the payment due to this participant for computing a subtask and v

[6] As of 13 Oct. 2015 average gas price is $\sim$ 55 Gwei $= 5.5 \cdot 10^{-8}$ eth, and 1 eth is $\sim$\$0.6.

is the value of the whole task. Thus the expected value of the outcome for this participant is $(v_i/v)\, v = v_i$. If the participants take part in lotteries for a large number of tasks, in the long run they may expect the same income as with direct payments for each subtask.

Below we describe the lottery protocol on a fairly abstract level. We fill in the details in Section 5 and Appendix B.

Lottery contract. Lotteries are handled by a single Ethereum contract which stores information about all the lotteries that are in progress. Participants (the payer and payees) send messages to this contract to start the lottery and claim the reward. The basic protocol consists of two types of messages: `lotteryInit`, sent by the payer to establish a lottery after the task is completed, and `lotteryWinner`, sent by any participant (presumably the winner) after the winner is determined to transfer the reward to the winner's account and end the lottery.

Starting the lottery. To start a new lottery, the payer creates a lottery description L and calculates its hash $h(L)$. The description contains a unique lottery identifier, Ethereum addresses $a_1, \ldots, a_n$ of each payee and payment values $v_1, \ldots, v_n$ due to each payee, possibly together with some other data. The payer then sends an `lotteryInit` message to the contract to announce the lottery. The message contains $h(L)$ and also transfers the task value from the payer to the contract (in Ethereum, each contract is associated with an account). The contract stores $h(L)$ in the Ethereum persistent storage. Since L contains a unique lottery identifier, $h(L)$ is also a unique value and may be used as a key when storing and retrieving various lottery data to/from Ethereum persistent storage.

The payer also announces the lottery description L to the Golem network, so that every interested node[7] can verify that its payment v_i has the correct value and check that $h(L)$ is indeed written to the Ethereum storage.

Determining the winner. The winner of the lottery can be uniquely determined from its description L and some random value R that is not known to any party at the moment the lottery is started. We assume that L and the timestamp t_0 of the moment at which $h(L)$ is written to the Ethereum storage determine another timestamp t_R of some future moment at which R will be revealed to everyone. From this moment on, R will also be available to the lottery contract.

A standard way to implement this rather abstract assumption in a blockchain setting is to use the hash of some future block as a random value. When the lottery is initialised the contract increments the number of the current Ethereum block by a fixed number, determining the number of a deciding block. The hash of this future block will be used to determine the lottery winner. There is a problem though with implementing this simple solution in Ethereum (see Section 4.2 for details).

Claiming the reward. Lottery reward may be claimed by sending a `lotteryCheck` message to the lottery contract, with the full lottery description L. The contract then calculates $h(L)$ and checks that it exists in the storage, which indicates that the lottery has started but the reward has not been claimed yet. If t_R has elapsed and R is available, the contract computes the winner's address from L and R and transfers the reward from the contract's account to the winner. At this point, the contract removes $h(L)$ from the storage.

The problem with the basic protocol is that the size of L is proportional to the number of payees and whoever sends the `lotteryCheck` message has to pay for sending L and processing it by the contract. Taking into account that sending a byte of data in a message costs 68 gas units we estimate that sending and processing the message will require at least 2 000 gas units per payee. This would defeat the whole lottery approach, which was proposed in order to avoid paying transaction fees proportional to the number of payees.

Fortunately, in order to verify that a_i is the winner the contract does not need to examine whole L. If the list of payees with their payment values is stored in a data structure called **Merkle tree** then it is enough to send and examine an amount of data proportional to the logarithm of the number of payees

[7] The lottery description must be sent to all lottery participants. It can be also sent to other nodes that are interested in watching the lottery contract and potentially take part in the lottery as a third party (for example, capture the hash of the deciding block if the payer does not do it on time or reveal that a node claiming to be the winner is cheating).

(see Appendix B for details). Therefore instead of sending full lottery description with `lotteryCheck` we can send the address a_i and just enough data to verify that this is the winner's address.

4.1 Optimistic Approach

If there is a lot of payees, the cost of `lotteryCheck` may still be high even when only a part of the lottery description is required to compute the winner. We propose an extension of the basic protocol that allows the parties to avoid sending the `lotteryCheck` message altogether. Instead, a payee can claim to be the winner by sending a new `lotteryWinner` message containing only $h(L)$. The contract cannot verify the claim and only checks that $h(L)$ exists in the storage. Then the contract stores additional data along with $h(L)$, namely the address a_i of the claimed winner and a computed deadline timestamp t_d which determines when the reward may be paid to a_i.

Claim verification. Until the deadline t_d elapses anyone can reveal the true winner by sending a `lotteryCheck` message with winner address a_i and enough data to validate it. Sending a `lotteryCheck` message after a `lotteryWinner` message should only happen if the payee that claimed to be the winner is caught cheating, which we hope will occur rarely (hence the name "optimistic approach"). To encourage peers to reveal cheaters, and to punish dishonest participants, we provide a mechanism of deposits. A payee sending the `lotteryWinner` message must transfer a deposit which is paid back if no one protests before the deadline elapses. This deposit is a reward for proving that the payee claiming to be the winner is cheating.

Paying out the reward. After the deadline t_d elapses, anyone may send a message `lotteryPayout` with $h(L)$ as the argument. If $h(L)$ exists in the storage and the address of the payee claiming to be the winner is set, the reward is transferred to this address and the lottery data is erased from the storage.

Under the assumption that all participants are honest, the cost of a lottery is constant, independent of the number of payees. An important property of the protocol described above is that after the random value is revealed, the payee who determines to be the winner may calculate the cost of a `lotteryCheck` message and decide whether to perform the basic protocol by sending `lotteryCheck` and obtain the reward immediately (modulo the time required by the Ethereum network to confirm the transaction), or to follow the extended protocol by sending the `lotteryWinner` with a deposit, wait until the deadline elapses and send the `lotteryPayout` message to get the lottery reward.

4.2 The Problem of 256 Past Blocks

Recall that we planned to use the hash of a future Ethereum block (deciding block) to determine who wins the lottery. More precisely, we assume that when the lottery is started the contract computes the number of the deciding block and stores this number together with other lottery data. Later on, when a `lotteryCheck` is sent the contract checks if a block with the required number has already been produced and if so, uses its hash to calculate who is the winner. Now, a problem with this approach is that in Ethereum, contracts can only access hashes of the last 256 blocks. Since a block is created roughly each 15 seconds, this gives us only about an hour during which the hash value is directly available from the contract. There is no possibility of scheduling a message for sending at a later time in Ethereum, other than relying on a third-party contract (for example Alarm Clock Contract [1]), so during this hour some party has to explicitly send either a `lotteryCheck` message that will use the hash value to compute the winner, or some other message that will prompt the contract to store the hash value for later use.

Capturing the hash of the deciding block. To address this issue we extend the protocol with a `lotteryCaptureHash` message the result of which is to store the hash of the deciding block. To encourage sending this message soon after the deciding block is generated we require that the sender of a `lotteryInit` message deposits an additional amount on the contract's account, together with the lottery reward. The deposit is returned to the payer if the payer sends a `lotteryCaptureHash` message within 128 blocks after the deciding block is generated. Between 128 and 256 blocks after the deciding block anyone may send this message and claim the payer's deposit.

As a last resort, if the hash of the deciding block is not stored and at least 256 more blocks are generated, a `lotteryCaptureHash` message will store the hash of the unique available block with the number

congruent modulo 256 to the number of the original deciding block. In this case, the payer's deposit will remain in the contract's account. Thus instead of trying to retrieve the lost hash we allow the winner to be changed.

If we could overcome the problem of 256 blocks then a payer deposit and `lotteryCaptureHash` message would be pointless and protocol would become simpler. An alternative solution would be to use a separate contract for providing hashes of past blocks. See Section 8 for more remarks on alternative sources of randomness.

4.3 Lottery Agents

We have to account for a situation when the lottery winner goes offline for a longer period of time and is unable to claim the reward or simply does not want or cannot pay for sending a `lotteryCheck` or `lotteryWinner` message. In such cases a third party may serve as **lottery agent** sending messages instead of the winner while keeping part of the reward. In particular, after a fixed amount of time elapses since the time when the random value R is revealed (the time when the deciding block is produced) and nobody has claimed to be a winner, anyone may send the `lotteryCheck` message which will transfer part of the reward, say 10%, to the sender of the message and the rest to the winner, and will erase the contract from Ethereum storage.

5 Lottery Protocol Specification

The lottery contract is implemented in Solidity [17], a high-level language compiled to machine code of the Ethereum Virtual Machine [21]. A contract in Solidity consists of a number of functions, each function is a entry point to the contract code. A message sent to the contract identifies a function to be called and specifies values of the function's arguments. Below we list Solidity functions that correspond to messages in the lottery protocol. The following data types of arguments are used:

- `uint` (an alias for `uint256`) is the type of 256 bit unsigned integers,
- `bytes32` is the type of fixed-size arrays of 32 bytes,
- `address` is the type of 160 bit Ethereum addresses.

In all the functions, the first parameter (`bytes32 lotteryHash`) is used to specify the lottery on which the function operates. Money is denominated in wei (1 wei $= 10^{-18}$ ether) and represented as `uint` values.

Contract functions. The following functions of the lottery contract can be invoked by users. The cost of each function is estimated based on the contract code in Appendix C.

- `function lotteryInit(bytes32 lotteryHash)`

  The sender Initialises a lottery. Lottery value is transferred to the lottery contract and a structure with lottery data is created in the storage. `lotteryHash` is the value used as a key to retrieve lottery data from the contract storage (see Appendix B for information on how it is computed).

  Estimated cost: 65 000 gas.

- `function lotteryWinner(bytes32 lotteryHash)`

  The sender claims to be the winner.

  Estimated cost: 30 000 gas.

- `function lotteryCaptureHash(bytes32 lotteryHash)`

  The message saves the hash of maturity block if it is possible. The payer should send this message within 128 blocks since maturity in order to get back the payer deposit. Later, but within 256 blocks, anyone else can send this message and get the payer deposit as a reward.

  Estimated cost: 35 000 gas.

- `function lotteryCheck(bytes32 lotteryHash, uint256 uid, address winner, uint32 rangeStart, uint32 rangeLength, bool[] path, bytes32[] values)`

  The sender specifies an address and the data required to verify that it is the winner's address. If the address is verified, the reward is transferred and the lottery data is erased from the storage.

  Estimated cost: $40\,000 + 2\,700 \cdot \log_2(N)$ gas.

- `function lotteryPayout(bytes32 lotteryHash)`

  The sender wants to receive her claimed reward after deadline elapsed. Transfers the reward and erases lottery data from the storage.

  Estimated cost: 30 000 gas.

LotteryData struct. For every lottery that is in progress the following Solidity structure is stored in the Ethereum storage:

```
struct LotteryData {
    uint value;
    uint maturity;
    uint deadline;
    uint randVal;
    address payer;
    address winner;
}
```

The meaning of the fields is as follows:

- `value` is the lottery reward (in wei),
- `maturity` is the number of the deciding block,
- `deadline` is the timestamp at which the user that claimed to be the winner may collect the reward,
- `randVal` is the random value determined from the hash of the deciding block,
- `payer` is the address of the user that created the lottery,
- `winner` is the address of the user that claimed to be the winner.

The contract will also have an `uint` variable `golemDeposit` which will store the total amount collected in the contract from commissions and lost deposits. This amount will be eventually redeemed by equity token holders (all Golem application owners).

The syntax of Solidity is similar to JavaScript. In particular, if lotteries is a mapping of uints to structs of type LotteryData then the statement

```
lotteries[lotteryHash].deadline = t
```

will update the field `deadline` in the `LotteryData` struct for a lottery with hash `lotteryHash`. One difference from JavaScript is that in Solidity a `mapping` is a total function, in particular `lotteries` initially maps all possible key values to a `LotteryData` struct with all fields initialised to 0. To check if a lottery data has been initialised we will check if `lotteries[lotteryHash].value` is nonzero.

Protocol. In the description of messages below the semantics of the **Preconditions** and **Effects** clauses is that if all the conditions specified in **Preconditions** are satisfied then the contract's state is changed as described in **Effects**. Otherwise, the message has no effect on the state of the contract.

1. The payer negotiates payment with every payee independently. Let v_i be the amount that i-th payee expects after this step and let a_i be the address of the i-th payee.

2. The payer calculates the transaction value $v = \sum_i v_i$, calculates the probability of winning for each payee and composes a lottery description L containing these data (see Section B for details). The payer then sends L to every payee. The payees check that L is correct. In particular, they may verify that the expected lottery reward for the i-th payee is equal to v_i.

3. The payer sends a `lotteryInit` message to the contract with the lottery hash $h(L)$ as the argument. The lottery hash can be calculated by anyone who knows L and is used as a key for storing/retrieving lottery data throughout the protocol (see Section B for information how $h(L)$ is computed).
   Preconditions: no `LotteryData` struct is associated with $h(L)$ in the contract's storage. This is equivalent to the condition

   ```
   lotteries[lotteryHash].value == 0
   ```

   Effects: The message transfers the lottery value v and the payer deposit to the contract and stores the `LotteryData` struct for the new lottery. The field `value` is set to 10/11 of the value transferred by the `message` and and the remaining 1/11 of this value is treated as the deposit. The field `maturity` is initialised with argument m and `payer` is initialised with the sender's address. The remaining fields are initialised with zeros.

   ```
   lotteries[lotteryHash] = LotteryData(value, block.number + maturity,
                                        0, 0, msg.sender, 0)
   ```

   At this point any payee can calculate the hash of the lottery on its own and can verify if it exists in the contract's storage. It can be done by querying the (locally available) Ethereum state.

4. Anyone (not only a payer) can send a `lotteryCaptureHash` message with argument $h(L)$ to write the hash of the deciding block as the lottery seed.
   Preconditions:

   (a) The value field of the `LotteryData` for the lottery has been set (meaning that the lottery has been initialised but not finalised yet).

   ```
   lotteries[lotteryHash].value != 0
   ```

   (b) The lottery `randVal` has not been set yet.

   ```
   lotteries[lotteryHash].randVal == 0
   ```

   (c) The maturity has elapsed (the stored maturity number is less than current pending block number).

   ```
   lotteries[lotteryHash].maturity < block.number
   ```

   Effects:

   (a) If the difference of the current block number and maturity is not greater than 128 then the hash of the deciding block is written as the `randVal` and the payer deposit is transferred to the payer.

   ```
   lottery.randVal = random(lottery.maturity);
   lottery.payer.send(calculatePayerDeposit(lottery.value));
   ```

   (b) If the difference of the current block number and maturity is greater than 128 but not greater than 256 then the hash of the deciding block is written as the `randVal` the payer deposit is transferred to the message sender.

   ```
   lottery.randVal = random(lottery.maturity);
   msg.sender.send(calculatePayerDeposit(lottery.value));
   ```

   (c) If the difference of the current block and maturity is greater than 256 then the hash of the block with number maturity+$k \cdot 256$, where k is the greatest possible integer, is written to the contract as the `randVal` and the payer deposit remains in the contract (a contract owner gets it).

   ```
   lottery.randVal = random(changeMaturity(lottery.maturity));
   golemDeposit += calculatePayerDeposit(lottery.value);
   ```

5. Anyone (not only a payee) can send a `lotteryWinner` message to claim being the winner. The message contains the lottery hash and requires to transfer a winner deposit.
   Preconditions:

   (a) The maturity stored for the lottery has been set.

   ```
   lotteries[lotteryHash].maturity != 0
   ```

   (b) The maturity has elapsed.

   ```
   lotteries[lotteryHash].maturity < block.number
   ```

   (c) Nobody has claimed to be the winner yet.

   ```
   lotteries[lotteryHash].winner == 0
   ```

   Effects:

   (a) The contract sets the claimed winner to the sender's address and sets deadline.

   ```
   lottery.winner = msg.sender;
   lottery.deadline = now + deadline;
   ```

   (b) If the `randVal` is not set than calls `lotteryCaptureHash` message.

   ```
   lotteryCaptureHash(lotteryHash)
   ```

6. Anyone (not only a payee or the payer) can send a `lotteryCheck` message. The message contains the lottery hash and the data required to compute the winner (see Section B.1 for details).
   Preconditions:

   (a) The maturity stored for the lottery has been set

   ```
   lotteries[lotteryHash].maturity != 0
   ```

   (b) The maturity has elapsed.

   ```
   lotteries[lotteryHash].maturity < block.number
   ```

   Effects:

   (a) If the `randVal` is not set than call `lotteryCaptureHash` message.

   ```
   lotteryCaptureHash(lotteryHash)
   ```

   (b) If 28800 blocks has been added to the blockchain after the deciding block (this would take approx. 5 days) then 10% of the transaction value v is transferred to the sender (which is treated as a lottery agent) and the transaction value is reduced to 90% of its previous value.

   ```
   msg.sender.send(calculateAgentProvision(lottery.value));
   ```

   (c) If the winner field is set to 0 in the lottery data than the transaction value is sent to the winner.

   ```
   winner.send(lottery.value);
   ```

   (d) If the winner field is equal to the winner address then the transaction value and the winner deposit are transferred to the winner.

   ```
   winner.send(lottery.value + calculateWinnerDeposit(lottery.value));
   ```

   (e) If the winner field is not equal to the winner address then the transaction value is transferred to the winner, winner deposit is transferred to the sender.

   ```
   msg.sender.send(calculateWinnerDespoit(lottery.value));
   ```

   (f) The lottery is erased from the contract.

   ```
   delete lotteries[lotteryHash]
   ```

7. Anyone can send a `lotteryPayout` message to the contract. The message contains the lottery hash.
   Preconditions:

   (a) The deadline has elapsed.

Method	Constant cost	Cost per user	Cost for 10 users	Cost for 100 users	Cost for 1 000 users
Batched transfers	21 000	11 000	130 000	1.1 mln	11.2 mln
Subaccounts	21 000	8 000	100 000	840 000	8.2 mln
Lottery (check)	140 000	$(\log_2(n)/n)2\,700$	150 000	155 000	165 000
Lottery (optimistic)	155 000	0	155 000	155 000	155 000

Table 1: Transaction costs in gas for different payment methods.

```
lotteries[lotteryHash].deadline < now
```

(b) Somebody has claimed to be a winner (winner isn't set to zero).

```
lotteries[lotteryHash].winner != 0
```

Effects:

(a) The transaction value and the deposit are transferred to the claimed winner.

```
lottery.winner.send(lottery.value + calculateWinnerDeposit(lottery.value))
```

(b) Transaction is completed and the lottery data is erased from the contract.

```
delete lotteries[lotteryHash]
```

Storage size optimisation. In the final version of the contract we optimise the storage size of `LotteryData`. The values of both maturity and deadline can be kept in the same field of the struct, as they have the same size and are never required at the same time: deadline is only set when someone sends `lotteryWinner` message and at this point maturity is no longer needed. Moreover, we can keep payer address and winner address in the same field. Payer address is only required for returning payer deposit. When winner address is being set, the random value must already be fixed and the deposit is already returned or lost. This allows us to limit size of final structure from 6 fields to 4.

Additionally, we can pack whole struct in two 256-bit words. Lottery value will be kept in one word and all the other values can be packed in one storage word: 8 bytes are enough for block numbers in maturity/deadline field, 4 bytes to keep random value (see remarks in Appendix B) and 20 bytes to keep payer/winner address. Writing a 256-bit word to the storage costs 20 000 gas, so using 2 words for `LotteryData` instead of 5 allows us to cut the cost of `initLottery` in half.

When estimating gas use (and cost in ethers) of the messages we assume the optimisation has been done, even though for simplicity in Appendix C we show contract code before optimisation.

6 Cost Comparison

In this section we compare transaction fees in payment schemes considered in Section 3.2 for varying number of users. For each scheme we estimate the total amount of gas required to pay for a single task. For calculating the required amount of gas we take into account the following operations:

- sending a message to a contract (21 000 gas),
- transferring data to a contract (68 gas for one nonzero byte),
- transferring ether to other account in a contract (9 000 gas),
- writing new storage entry (20 000 gas for each nonzero 256-bit word),
- modifying existing storage entry (5 000 gas for each 256-bit word),
- computing SHA-3 hash (222 gas for hashing a 256-bit word).

Other operations have negligible cost. The costs are summarised in Table 1 and shown in graphical form in Figure 1.

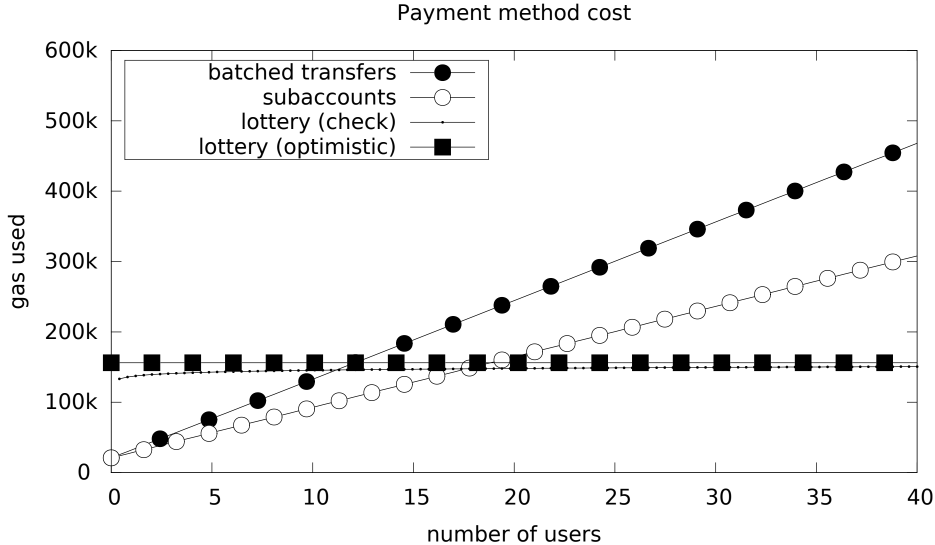

Figure 1: Transaction costs for different payment methods, in graphical form.

Note that when using direct transfers, only the payer bears the transaction costs, while other payment methods also incur costs for the recipients: in the subaccounts approach recipients pay for transferring ether to their accounts and in the lottery approach the winner pays for claiming the reward. To facilitate comparison between payment methods we assume that the recipients will negotiate higher payments to compensate for additional costs and thus effectively all transaction costs will be covered by the payer.

In the subaccounts approach the recipients decide when to transfer ether from their subaccounts managed by the payment contract. Each such transfer requires approximately 35 000 gas and we assume that users on average delay the transfer long enough that the transaction fee for a single task does not exceed 1 000 gas. Thus per-user cost in the subaccount approach is estimated at 8 000 gas (2 000 for passing the recipient's address and payment value to the contract, 5 000 for updating the balance and 1 000 as the amortised cost of transferring ether from the subaccount).

In both variants of the lottery approach, payer fees are constant ($\sim$ 100k gas for initialising the lottery and capturing the hash of the deciding block) independent on the number of participants. The winner is free to decide which variant to choose. If the number of participants does not exceed 128 then sending `lotteryCheck` is the cheapest option, for larger number of participants the optimistic variant has lower fees. There is however a trade-off: optimistic approach has lower transaction fees but the winner has to wait approximately 24 hours after sending the `lotteryWinner` message to collect the reward.

Note also that for a task with 1 000 participants it is less expensive to run 10 lotteries (one for each 100 participants and with 1/100 chance of winning 1/10 of the total task value) than to pay using batched transfers or subaccounts.

6.1 Costs in Fiat Currency

We may also give corresponding costs in USD, using the gas/ETH and ETH/USD rates at the time of this writing.[8] As the ETH/USD rate changes quickly (and will probably do so in the future, taking into account the history of Bitcoin price), transaction costs in any fiat currency should be taken as very rough estimates, so we round the values up to the nearest power of ten. In this approximation Table 1 collapses to just two rows, as shown in Table 2.

6.2 So, Which Payment Method Should I Use to Pay for My Task?

Assuming the payer wants to keep transaction fees on the level of 1%, direct transfers or subaccounts are appropriate for tasks in which each participant is to be paid at least \$0.1, for example a task worth \$1

[8] As of 13 Oct. 2015, average gas price is approximately $5.5 \cdot 10^{-8}$ ETH and one ether costs approximately \$0.6

Method	Cost for 10 users	Cost for 100 users	Cost for 1 000 users
Batched transfers/subaccounts	\$0.01	\$0.1	\$1
Lottery (either variant)	\$0.01	\$0.01	\$0.01

Table 2: Transaction costs in USD for different payment methods, order-of-magnitude estimates.

computed by 10 users or a task worth \$1 000 computed by 10 000 users.

A lottery must be used for tasks in which participants are to be paid less than \$0.1. It may also be used if there are at least 20 participants and the payer wants to minimise transaction fees even at the cost of using a more complex and probabilistic payment method. Note that the constant cost of the lottery approach is low enough to make it suitable for tasks with total value of \$1, which is the minimum task value Golem will need to handle (see Section 1).

When comparing costs of various methods we should take into account another factor besides gas requirements. Namely, if given a choice users will probably prefer to participate in tasks for which payment is made with deterministic methods and the outcome is more predictable. For similar reasons, lotteries with less participants will be preferred, as they are more likely to provide a reward. If the compute nodes are in short supply, lottery rewards will have to be higher than task value paid with batched transfers or subaccounts method, to encourage nodes to participate. This may drastically limit the applicability of the lottery method.

7 Probabilistic Analysis of the Lottery Approach

In this section we examine some probabilistic characteristics of the lottery approach. A crucial property, from the point of view of a Golem user, is that as the user participates in more and more lotteries, her income approaches the expected value. That is, the probability that the sum of rewards will be significantly different from the expected value is getting lower as the number of attempts grows. More formally, let us assume for simplicity that all lotteries have the same reward v and the same probability of winning p. For a fixed user, the expected value of the reward from a single lottery is $v \cdot p$ and, since lotteries are independent, the expected value of the sum of rewards from n lotteries is $n \cdot v \cdot p$. Let X_n be a random variable representing the sum of outcomes from n lotteries. From the law of large number it follows that for any positive ϵ

$$\lim_{n \to \infty} \Pr\left(|X_n - n \cdot v \cdot p| > \epsilon\right) = 0.$$

Users may want to know how fast will their income converge to its expected value. More specifically, they may ask questions of the kind:

> What is the probability that I receive at least 80% of the expected income after 100 lotteries?

The probability that the user wins exactly k lotteries out of n is given by the probability mass function of the binomial distribution:

$$f(k, n, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

Therefore, the probability that the user wins at least k lotteries out of n is

$$F(k, n, p) = \sum_{i=k}^{n} f(i, n, p) = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}$$

Suppose the user participates in $n = 100$ lotteries, each with $p = 1/10$. The expected number of lotteries won is $p \cdot n = 10$. In order to get at least 80% of the expected income the user has to win at least $80 \cdot p \cdot n = 8$ lotteries, the probability of which is $F(8, 100, 0.1) \approx 0.79$.

In Figure 2 we plot the values of $F(0.08, n, 0.1)$ and $F(0.09, n, 0.1)$ which represent the probability that the user receives at least 80% and at least 90% of the expected income, respectively, after n lotteries, each with 10 participants. The reader may check for example that getting 80% of the reward with probability 0.9 requires approximately 300 lotteries while getting 90% of the reward with the same probability requires 1 400 lotteries.

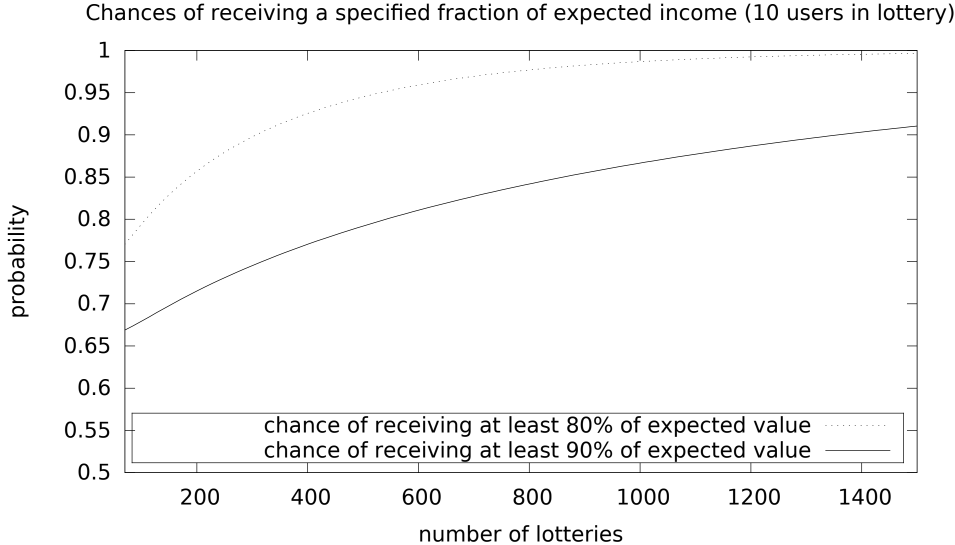

Figure 2: A probability of receiving at least 80% or 90% of expected income after n lotteries with 10 participants

8 Source of Randomness for the Lottery Implementation

In our lottery implementation we use the hash of a future block as the source of randomness. This solution has two drawbacks. First, there is a risk that the random value may be manipulated by Ethereum miners. That is, a miner may decide whether to publish a newly found block after looking at the block hash and calculating the winning address. Clearly, if the lottery reward is small compared to the reward for finding a new block then it is not profitable to withhold a block to change the odds of winning a lottery. Thus one way to circumvent this problem is to limit the maximum lottery reward and in case the task value exceeds this limit, split payees into several groups and organise a separate lottery for each group, with a fraction of the task value as the reward.

The second problem is due to the fact that only the hashes of past 256 blocks are available in the contract so we have to complicate the protocol and introduce the mechanism of deposits to encourage the participants to capture the hash of the deciding block.

Let us consider some alternatives to using block hashes as the source of randomness.

8.1 The RANDAO Approach

Let us consider a protocol in which the winner is determined using only data provided by a group of participants, in a peer-to-peer environment, without using any external sources (including hashes of future block) and without relying on any trusted third party. That is, each participant reveals a piece of data and once all data is public the winner is determined in a provable way.

The protocol needs to deal with situations in which some participants fail to reveal their data. This may be due to communication failure or may be done on purpose. Assume one participant delayed revealing her data until all other data are revealed. This participant has now full information required to determine who the winner is, and may decide to quit the protocol after deciding that its outcome is unfavourable.[9]

In this situation the procedure may be either (1) to determine the winner using only the available data or (2) to repeat the whole protocol without the participant who quit. The first case means that input from all participants is not required for completing the protocol which may make the whole procedure vulnerable to various denial-of-service attacks that prevent participants from providing their data.

[9] We assume the participants commit to their data before the protocol starts, for example using hash functions, so changing the data at this point is not an option.

In the second case, participants may collude to affect the outcome. Suppose there is a group of participants cooperating to increase the chance of one of them winning. Now, one of the group may delay revealing the last piece of data and quit if the winner would not be a member of the group, and the procedure would need to be repeated. Obviously, such behaviour increases the chances that a group member will finally turn out to be the winner and thus makes the whole protocol unfair.[10]

Note that participants may be discouraged from quitting the lottery for example by losing deposits they have to make earlier (this is the solution adopted in RANDAO implementations [14] and [15]). This would make the whole lottery protocol more complicated and would require some further design decisions, for example how big should the deposit be, what if a participant does not have money to pay the deposit and so on.

8.2 A Contract for Storing hashes of Past Blocks

We also consider implementing our own custom contract for storing hashes of past blocks. The idea is to consider only hashes of blocks with numbers divisible by 100 (for instance) and remember last 100 (for instance) such hashes. This means that a fresh block hash will be available roughly each 25 min (time to generate 100 blocks) and the contract may reproduce hashes from the past 100 days (time to generate 10 000 blocks). The contract stores an array of 100 hashes and has two functions: `feed()` for writing the hash of the most recent block with number divisible by 100, and `get(blocknum)` which returns the hash for the latest block with the number divisible by 100 and less or equal to `blocknum`.

```
contract OneHundredHashes {

    bytes32[100] hashes;

    function feed() external {
        uint8 index = uint8((block.number / 100) % 100);
        hashes[index] = block.blockhash(block.number - block.number % 100);
    }

    function get(uint blocknum) external returns (bytes32) {
        uint8 index = uint8((blocknum / 100) % 100);
        return hashes[index];
    }
}
```

The cost of a call to `feed` is 26 000 gas (21 000 for a transaction and 5 000 for updating storage) and `feed` should be called at least every 100 blocks, that is, roughly 20 000 times a year. With the current price of gas and Ether, the cost of running the contract is approx $20 per year. The cost of contract initialisation and storing first 100 hashes is negligible (well below $1). The cost of calling `get` is slightly above 21 000 gas (reading from storage is cheap compared to writing).

For the contract to function properly, `feed` needs to be called once in every 25 min, by users or by a service integrated with Golem.

9 Conclusion

In this paper we present several approaches to one-to-many micropayments in the context of the Golem project. We require solution that is decentralised, provides secure transactions without any trusted authorities and, additionally, does not assume that a payer makes a series of micropayments to a single recipient. On the other hand, we do assume that one recipient receives a series of micropayments from many payers. The main problem is how to design the payment scheme to keep transaction fees in the order of 1% or less.

Cryptocurrencies provide decentralised and secure transaction platform, so we first consider basing our solution on Bitcoin, by far the most widely used cryptocurrency. It turns out that transaction fees

[10] We may consider a modification of the protocol in which the order in which the participants reveal their data is fixed in some fair way. However, there will still be a possibility that the last participant to reveal her data is a member of the group.

in Bitcoin are too high to guarantee a safety margin against possible changes of Bitcoin price in fiat currencies, especially if there are 100 or more transaction recipients. To make things worse, there are reasons to expect that Bitcoin transaction fees will be higher in the future (see Section 2 for details).

We then turn our attention to Ethereum. A straightforward approach—batching many direct transfers in a single Ethereum transaction—is cheaper than in Bitcoin, but still within the same order of magnitude. Transaction fees may be somewhat reduced if payments for a single recipient are accumulated on this recipient's "subaccount" managed by an Ethereum contract. The cost of all these straightforward methods is proportional to the number of task participants, not to the task value. This means that there is penalty for distributing tasks to larger number of participants: we estimate that if a payment for computing a single subtask is lower than $0.1 then transaction costs exceed 1% of the task value (with the costs of gas and Ether at the time of this writing).

A big advantage of Ethereum is that it allows us to use smart contracts to implementing more complex payment schemes better suited to Golem setting and with lower transaction fees. In particular, we propose a probabilistic payment scheme that uses lotteries to remunerate task participants. The cost of our solution is either constant, independent from a transaction value and the number of participants, or proportional to the logarithm of the number of participants, depending on the variant of the payment protocol chosen by the participant that wins the lottery. In practice, costs of both variants are very similar even for thousands of participants. The constant term in the cost of the lottery is low, making the lottery cheaper than the "subaccounts" method (our best $O(n)$ method) for tasks with at least 20 participants.

In practice, even if the lottery is superior to all $O(n)$ methods in terms of the gas use, Golem users may prefer to use a method with higher fees but with a predictable outcome. If compute nodes always have the option to choose a task with deterministic payment, this may effectively make lotteries nonviable.

It may be also interesting to consider a generalisation of the lottery payment protocol that handles lotteries with an arbitrary (but fixed) number of winners. For example, a payment for a task with 1 000 participants could be made with a lottery that has 10 winners, giving each participant 1/100 chance of winning instead of 1/1 000. Such a generalised lottery would be less expensive than running 10 separate lotteries for each 100 participants, but would probably require a more complex protocol. This could be a subject of future work.

We think that the proposed lottery mechanism can be used not only in the Golem project but also in other one-to-many micropayments scenarios in the future.

References

[1] Merriam, Piper. "Ethereum Alarm Clock", http://www.ethereum-alarm-clock.com/.

[2] Andresen, Gavin. "Back-of-the-envelope calculations for marginal cost of transactions", https://gist.github.com/gavinandresen/5044482.

[3] Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008, https://bitcoin.org/bitcoin.pdf.

[4] "Working with micropayment channels", *Bitcoinj website*, https://bitcoinj.github.io/working-with-micropayments.

[5] "Bitcoin Transaction Fees Explained", February 2014, http://bitcoinfees.com/.

[6] Buterin, Vitalik. "Scalability, Part 1: Building on Top", *Ethereum Blog*, September 17, 2014, https://blog.ethereum.org/2014/09/17/scalability-part-1-building-top/.

[7] *ChangeTip website*, https://www.changetip.com/.

[8] "Shutting Down Coinbase Tip Button", *The Coinbase Blog*, February 10, 2015, https://blog.coinbase.com/2015/02/10/shutting-down-the-coinbase-tip-button/.

[9] Menezes, Alfred J., et al. *Handbook of Applied Cryptography*, 1997.

[10] Buterin, Vitalik, et al. "Ethereum White Paper", updated September 30, 2015, https://github.com/ethereum/wiki/wiki/White-Paper.

[11] Board of Governors of the Federal Reserve System, Press Release, June 29, 2011, http://www.federalreserve.gov/newsevents/press/bcreg/20110629a.htm.

[12] Jain, Mohit, Siddhartha Lal and Anish Mathuria. *Peer-to-Peer (P2P) Micropayments: A Survey and Critical Analysis*, technical report, DA-IICT, Gandhingar, India, 2008, http://www.dgp.toronto.edu/~mjain/P2P_Micropayment-2008.pdf.

[13] Kaşkaloğlu, Kerem. "Near Zero Bitcoin Transaction Fees Cannot Last Forever", in *The International Conference on Digital Security and Forensic (DigitalSec2014)*, 91–99, June, 2014, http://sdiwc.net/digital-library/near-zero-bitcoin-transaction-fees-cannot-last-forever.html.

[14] McKinnon, Dennis. "RANDAO repository", https://github.com/dennismckinnon/Ethereum-Contracts/tree/master/RANDAO.

[15] Qian, Yaucai. "RANDAO repository", http://github.com/randao.

[16] Rivest, Ronald. "Electronic Lottery Tickets as Micropayments", in *Financial Cryptography*, 307–314, Springer Verlag, 1997, https://people.csail.mit.edu/rivest/pubs/Riv97b.pdf.

[17] Reitwiessner, Christian, et al. "Solidity: Trustless, decentralised, smart contracts for Ethereum", web page, https://ethereum.github.io/solidity/.

[18] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION, *SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions*, FIPS PUB 202, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[19] Vishnumurthy, Viviek, Sangeeth Chandrakumar and Emin Gün Sirer. "KARMA: A Secure Economic Framework for Peer-to-Peer Resource Sharing", in *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003, http://cs.brown.edu/courses/csci2950-g/papers/karma.pdf.

[20] Wheeler, David. "Transactions Using Bets", in *Proc. of the Fourth Cambridge Workshop on Security Protocols*, 82–92, Springer, 1996.

[21] Wood, Gavin. "Ethereum: A Secure Decentralized Generalised Transaction Ledger", 2014, http://gavwood.com/paper.pdf.

A Dictionary

- **Contract.** Contract refers to the smart contract, EVM code in Ethereum blockchain which services micropayments.
- **Deadline.** Deadline is a number of a block when transaction value is payed out if only one payee claims to be the winner and no one protests. After the deadline protests are not taken into account. Deadline should correspond to 24 hours, ie. 86400 seconds.
- **Lottery agent.** User that claims that other user is a winner. She may pay the cost of sending lotteryCheck message for 10% of transaction value. Lotteries agent may take part only in old lotteries where nobody is claiming to be a winner for a long time (∼5 days).
- **Maturity.** Maturity is a number of a block when the random number is fetched and a winner is determined. After maturity payee can verify who is the winner.
- **Message.** Message is a data sent from Ethereum account to a contract.
- **Payer deposit.** It is a value paid by payer in order to urge him to send another message in the next step. The estimate of deposit value is 10% of transaction value.

- **Random number, random number source.** It is an independent random number which determines a winner in a lottery. In Golem we consider hashes of blocks as sources of randomness. We recall that Ethereum allows to read hashes of only 256 previous blocks, so random value is a hash of a block and determines a winner of a lottery. Primarily it is the hash of maturity block but if the hash of maturity block is not accessible (maturity is too old, more than 256 blocks away) we take a hash of the nearest block which number matches maturity + k * 256.
- **User Identity.** It is an address identifying an account (user) in Ethereum.
- **Winner deposit.** It is a value payed by payee which claims to be the winner or by lottery agent. The estimate of deposit value is 50% of a transaction value.

B Lottery Description

Here we show how to store lottery data in a Merkle tree, so that the contract can verify that a specified payee is indeed the winner in the number of steps proportional to the logarithm of the number of payees. We assume a fixed cryptographic hash function [9] (for example SHA-3 [18]) that will be used for the payment protocol. For a binary string B, $\texttt{hash}(B)$ will denote the result of the hash function applied to B. For a sequence $B_1, ..., B_n$ of binary strings, $\texttt{hash}(B_1, ..., B_n)$ will denote the the result of the hash function applied to the concatenation of $B_1, ..., B_n$. In the concrete Ethereum implementation SHA-3 with 256-bit output will be used.

In the following we assume that the task computation may be divided to at most 2^S subcomputations or, in other words, that the payment to each payee is a multiple of $V/2^S$, where V is the value of the whole task. A large value of S will allow us to split the task in a fine grained subtasks but will make the lottery description bigger.

Let N be the number of payees and, for each i in $\{1, ..N\}$, let r_i be such that the value due to the i-th payee is

$$v_i = r_i \cdot \frac{V}{2^S},$$

that is

$$r_i = 2^S \cdot \frac{v_i}{V}.$$

Since $\sum_i v_i = V$, we also have $\sum_i r_i = 2^S$.

For each i in $\{1, ..., N\}$, let R_i denote the $\sum_{j<i} r_i$.

Given a number x in $\{0, 1, ..., 2^S - 1\}$, the winner of the lottery is the unique index i in $\{1, ..., N\}$ such that

$$R_i \leq x < R_i + r_i.$$

Example (with $S = 3$, $N = 4$)**.** *Let* $r_1 = 2$, $r_2 = 3$, $r_3 = 1$ *and* $r_4 = 2$. *Then we have* $R_1 = 0$, $R_2 = 2$, $R_3 = 5$ *and* $R_4 = 6$.
Now,

$$\begin{array}{ll} 1 \textit{ wins if} & 0 \leq x < 2 \\ 2 \textit{ wins if} & 2 \leq x < 5 \\ 3 \textit{ wins if} & 5 \leq x < 6 \\ 4 \textit{ wins if} & 6 \leq x < 8 \end{array}$$

Definition (lottery description)**.** *A **lottery description** L is a data structure that contains all relevant data for a lottery instance, such as the address of the Golem node that announces the lottery, a timestamp (which together uniquely identify a lottery), task value V, the list of payee addresses $a_1, \ldots, a_N$ and the corresponding list $p_1, \ldots, p_N$ of probabilities of winning the lottery for each of the payees. L may also contain some other data required by the implementation, the exact details are not relevant. We assume that all parties agree on the format used for lottery descriptions and are able to verify that L is valid. By $B(L)$ we denote the binary representation of L (i.e. encoding of L as a sequence of bits).*

Definition (payment list)**.** *A **payment list** for L is a sequence $P(L) = ((a_1, R_1, r_1), \ldots, (a_N, R_N, r_N))$ where the values $R_1, \ldots, R_N$ and $r_1, \ldots, r_N$ are computed from task value V and probabilities $p_1, \ldots, p_N$ as described above. To restrict our attention we assume that R_i and r_i are 32 bit words (thus $S = 32$). The definitions below can be adjusted to other values of S in a straightforward way.*

B.1 Lottery Verification

Now, in order to verify that a_i is the winner of a lottery described by L for a given random value R, one has to make sure that $P(L)$ contains a tuple (a_i, R_i, r_i) such that $R_i \leq R < R_i + r_i$ holds. This can be done by proposing the tuple (a_i, R_i, r_i) and:

1. checking that it satisfies the required inequalities,
2. proving that it is an element of $P(L)$.

(1) is trivial and (2) can be done by iterating over $P(L)$ until (a_i, R_i, r_i) is found. However, doing so in the lottery contract would require sending whole $P(L)$ in a message (or half of $P(L)$ on average, if the triples in $P(L)$ are sorted) which would incur a substantial cost for the sender. Fortunately, the verification may be performed in the number of steps and with the size of data proportional to the logarithm of N by encoding information in $P(L)$ in a **Merkle tree**, that is a full binary tree in which every inner node is labelled by the hash of the labels of its children.

In the following we make use of the standard identification of binary trees with nonempty prefix-closed sets of sequences over $\{0, 1\}$. That is, $T \subseteq \{0, 1\}^*$ is a binary tree if $T \neq$ and for every $n \in \{0, 1\}^*$ and $b \in \{0, 1\}$, if $nb \in T$ then $n \in T$. Here, the empty sequence is the root of T and $n0$ and $n1$ are the left and the right child of s,respectively. T is **full** if every inner node has two children, that is if for every $n \in T$ we have $n0 \in T \iff n1 \in T$. A **labelled binary tree** is a binary tree T and a function l from T to some fixed set of labels. Finally, a Merkle tree is a **labelled full binary tree** (T, l) with the labelling function $l : T \to \{0, 1\}^{256}$ satisfying:

$$l(n) = \texttt{hash}(l(n0), l(n1))$$

for every $n \in \{0, 1\}^*$ such that $n0 \in T$.

A pleasant property of Merkle trees is that to prove that a given tree contains a node with specific label w we only need to examine the amount of data proportional to $\log(d)$, where d is the depth of the specified node.

Let us fix a Merkle tree T and a node $n = b_1, \ldots, b_d$ with label $w = l(n)$ and let $w_1, \ldots, w_d$ be a sequence of 256 bit words such that for each $i \in \{1, \ldots, d\}$, if $b_i = 0$ then w_i is the label of the right child of the node $b_1, \ldots, b_{i-1}$ and otherwise w_i is the label of its left child (that is, $w_i = l(b_1, \ldots, b_{i-1}, 1 - b_i)$).

Let $h_0, \ldots, h_d$ denote the labels of the nodes on the path from the root to n. That is, h_0 is the label of the root, $h_d = w$ and in general, $h_i = l(b_1, \ldots, b_i)$. From the property of Merkle trees we have, for every $i \in \{1, \ldots, d\}$:

- if $b_i = 0$ then $h_{i-1} = \texttt{hash}(h_i, w_i)$
- if $b_i = 1$ then $h_{i-1} = \texttt{hash}(w_i, h_i)$

This means that given n, w and $w_1, \ldots, w_d$ we may compute the values of h_i, starting from $h_d = w$ and climbing the tree up to h_0 which is the label of the root. Since hash is assumed to be collision resistant ([18]), finding n, w and $w_1, \ldots, w_d$ for which the above procedure yields the given hash value h_0 is considered extremely difficult. Therefore, for all practical purposes, proposing the values $n, w, w_1, \ldots, w_d$, computing the hash h_0 and checking that it is equal to the label in the root of a Merkle tree T is considered a valid proof of n being a node in T labelled with w.

The above definitions are illustrated in Figure 3 which shows a Merkle tree with a distinguished node $n = 010$. The label w of n, together with labels w_3, w_2 and w_1, are used to compute hashes $h_3, \ldots, h_0$.

Now, going back the lottery setting, a Merkle tree for a lottery description L is a Merkle tree with minimal height such that for every (a_i, R_i, r_i) in $P(L)$ there exists a leaf labelled with $\texttt{hash}(a_i, R_i, r_i)$. This definition allows many different Merkle trees for a given L, so we assume that all lottery participants agree on a common algorithm that builds a "canonical" tree $M(L)$ for each L, so that every payee may

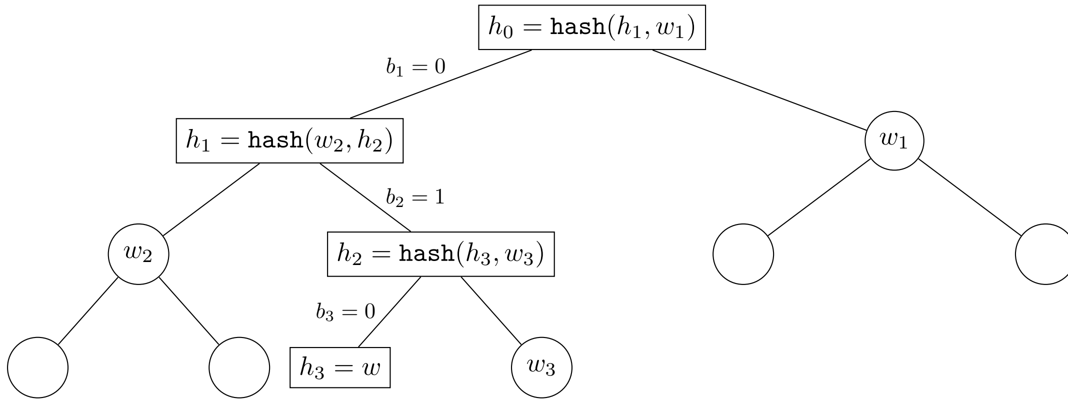

Figure 3: An example Merkle tree.

construct $M(L)$ after receiving L. Alternatively, the sender may send a tree for L together with L to each payee. Note that the height of any Merkle tree for L, that is the maximum depth of any leaf, is equal to ceil($\log_2 P$).

A **hash of** L is defined as $h(L) = \texttt{hash}(\texttt{hash}(B(L)), h(M(L)))$, where $B(L)$ is the binary encoding of L and $h(M(L))$ is the label of the root of $M(L)$.

A **winner certificate** $C = (a, R, r, b_1, \ldots, b_d, w_1, \ldots, w_d)$ consists of

- an Ethereum address a of the proposed winner,
- S-bit words R and r encoding respectively the beginning and the length of the range within which the S-bit random value must fall,
- a sequence $b_1, \ldots, b_d$ of bits, with $d \geq 0$,
- a sequence $w_1, \ldots, w_d$ of 256-bit words.

Given C we define the sequence $h_d, h_{d-1}, \ldots, h_0$ of 256-bit words as follows:

$$h_d = \texttt{hash}(a, R, r)$$

$$h_{i-1} = \begin{cases} \texttt{hash}(h_i, w_i) & \text{if } b_i = 0 \\ \texttt{hash}(w_i, h_i) & \text{if } b_i = 1 \end{cases}$$

Let x be a S-bit random value. C is **valid** for x if

$$R \leq x < R + r \quad \text{and} \quad \texttt{hash}(\texttt{hash}(B(L)), h_0) = h(L).$$

A Solidity implementation of the algorithm that checks the validity of a winner certificate is coded in Appendix C as the function `checkCertificate` with the following signature:

```
function checkCertificate(uint32 rand, bytes32 lotteryHash, WinnerCertificate cert)
returns (bool)
```

The argument `rand` is the random value (x in the description above, with precision $S = 32$), `lotteryHash` is $h(L)$ and `cert` is a struct encoding a winner certificate.

C Contract Code

It's an Ethereum contract written in Solidity. Take into account that is a simple version of lottery with more fields in `LotteryData` struct for more clarity.

```
contract Lottery {
```

```
uint golemDeposit; // golem account for commissions

uint constant maturity = 10; // blocks
uint constant agentMaturity = 28800; // blocks
uint constant deadline = 86400; // seconds

struct LotteryData { // in real-life this struct can fill 2 storage words
    uint value;
    uint maturity;
    uint deadline;
    uint32 randVal;
    address payer;
    address winner;
}

struct WinnerCertificate {
    uint256 uid; // lottery id
    address winner; // winner's address
    uint32 rangeStart; // beginning of the range
    uint32 rangeLength; // length of the range
    bool[] path; // encoding of the leaf as a sequence b_1,...,b_d
    bytes32[] values; // values w_1,...,w_d
}

mapping (bytes32 => LotteryData) lotteries;

function lotteryInit(bytes32 lotteryHash) {
    LotteryData lottery = lotteries[lotteryHash];
    if (lottery.value != 0)
        return;
    uint payerDeposit = calculateInitialPayerDeposit(msg.value);
    uint value = msg.value - payerDeposit;
    lotteries[lotteryHash] = LotteryData(value, block.number + maturity, 0, 0,
                                         msg.sender, 0);
}

function lotteryCaptureHash(bytes32 lotteryHash) {
    LotteryData lottery = lotteries[lotteryHash];
    if (lottery.value == 0)
        return;

    if (lottery.randVal != 0 )
        return;

    if (block.number <= lottery.maturity)
        return;

    if (lottery.maturity + 128 <= block.number)
        lottery.payer.send(calculatePayerDeposit(lottery.value));
    else if (lottery.maturity + 256 <= block.number)
        msg.sender.send(calculatePayerDeposit(lottery.value));
    else
        golemDeposit += calculatePayerDeposit(lottery.value);

    lottery.randVal = random(lottery.maturity);
```

```
    lottery.maturity = 0;
    lottery.payer = 0;
}

function lotteryWinner(bytes32 lotteryHash) {
    LotteryData lottery = lotteries[lotteryHash];
    if (lottery.value == 0)
        return;

    if (msg.value < calculateWinnerDeposit(lottery.value))
        return;

    if (lottery.winner != 0 && block.number <= lottery.maturity)
        return;

    lottery.winner = msg.sender;
    lottery.deadline = now + deadline;

    if (lottery.randVal == 0) {
        lotteryCaptureHash(lotteryHash);
    }
}

function lotteryPayout(bytes32 lotteryHash) {
    LotteryData lottery = lotteries[lotteryHash];
    if (lottery.winner == 0)
        return;
    if (now <= lottery.deadline)
        return;

    uint deposit = calculateWinnerDeposit(lottery.value);
    lottery.winner.send(lottery.value + deposit);
    delete lotteries[lotteryHash];
}

function checkCertificate(uint32 rand, bytes32 lotteryHash, WinnerCertificate cert)
                            internal returns (bool) {
    // Check if random val falls into the range
    if (rand < cert.rangeStart ‖ rand >= cert.rangeStart + cert.rangeLength)
        return false;

    // Initially, h is the value stored in the leaf (h_d)
    bytes32 h = sha3(cert.winner, cert.rangeStart, cert.rangeLength)

    // Update h with hashes h_{d-1}, ..., h_0
    for (uint i = cert.path.length; i > 0; i-) {
        if (cert.path[i-1] == false)
            h = sha3(h, cert.values[i-1]);
        else
            h = sha3(cert.values[i-1], h);
        }
    // Mix with uid
    h = sha3(cert.uid, h);

    return h == lotteryHash;
}
```

```
function lotteryCheck(bytes32 lotteryHash, uint256 uid, address winner, uint32 rangeStart,
            uint32 rangeLength, bool[] path, bytes32[] values) {
    WinnerCertificate memory winnCert = WinnerCertificate(uid, winner, rangeStart,
                        rangeLength, path, values);
    LotteryData lottery = lotteries[lotteryHash];
    if (lottery.value == 0 || block.number <= lottery.maturity)
        return;

    if (lottery.randVal == 0) {
        lotteryCaptureHash(lotteryHash);
    }

    if (!checkCertificate(lottery.randVal, lotteryHash, winnCert)) {
        return;
    }

    if (lottery.winner != 0) {
        if (msg.sender != winner) {
            msg.sender.send(calculateWinnerDeposit(lottery.value));
            winner.send(lottery.value);
        } else {
            winner.send(lottery.value + calculateWinnerDeposit(lottery.value));
        }
    } else {
        if (msg.sender != winner && block.number > lottery.maturity + agentMaturity) {
            // Sender is lottery agent
            msg.sender.send(calculateAgentCommission(lottery.value));
            winner.send(lottery.value - calculateAgentCommission(lottery.value));
        } else {
            winner.send(lottery.value);
        }
    }

    delete lotteries[lotteryHash];
}

function changeMaturity(uint maturity) internal constant returns(uint) {
    uint begMod = (block.number - 256) % 256;
    uint matMod = maturity % 256;
    maturity = (block.number - 256) + matMod - begMod;
    if (begMod > matMod) {
        maturity += 256;
    }
    return maturity;
}

function random(uint maturity) internal constant returns(uint32) {
    if (block.number - maturity > 256)
        maturity = changeMaturity(maturity);

    return uint32(block.blockhash(maturity));
}

function calculateWinnerDeposit(uint val) internal constant returns (uint) {
    return val / 2;
```

```
    }

    function calculateInitialPayerDeposit(uint val) internal constant returns (uint) {
        return val / 11;
    }

    function calculatePayerDeposit(uint val) internal constant returns (uint) {
        return val / 10;
    }

    function calculateAgentCommission(uint val) internal constant returns(uint) {
        return val / 10;
    }

}
```