CasperLabs Tech Spec

CasperLabs Development Team

Aug 26, 2020

Contents

We present the design for a new Turing-complete smart contract platform, backed by a Proof of Stake (PoS) consensus algorithm, and WebAssembly (Wasm). The intent is for this design to be implemented as a new permissionless, decentralized, public blockchain. The consensus protocol is built on Vlad Zamfir's correct-by-construction (CBC) Casper work. Here, we describe The Highway Protocol, a member of the CBC Casper family which is provably safe and live under partial synchrony. The computation model allows for efficient detection of when contract executions can be run in parallel and the block message format allows "merging" forks in the chain; so the platform avoids orphaning blocks. Rust is supported as the primary programming language for smart contracts because of its good support for compilation to wasm; however, the platform itself does not make assumptions about the source language, so libraries facilitating contract development in other programming language having wasm as a compile target are expected. Other features of the execution engine include: an account permissions model allowing for lost key recovery, and a permissions model to securely share state between accounts and/or contracts (without the need for expensive cryptographic checks). We also provide discussions of the economics of our proof-of-stake implementation and our token policies.

Disclaimer

By accepting this CasperLabs Tech Spec (this "**Whitepaper**"), each recipient hereof acknowledges and agrees that is not authorised to, and may not, forward or deliver this Whitepaper, electronically or otherwise, to any other person or reproduce this Whitepaper in any manner whatsoever. Any forwarding, distribution or reproduction of this Whitepaper in whole or in part is unauthorised. Failure to comply with this directive may result in a violation of applicable laws of any affected or involved jurisdiction.

Nothing in this Whitepaper constitutes an offer to sell, or a solicitation to purchase, the tokens native to the Casper blockchain ("**CLX**"). In any event, were this Whitepaper to be deemed to be such an offer or solicitation, no such offer or solicitation is intended or conveyed by this Whitepaper in any jurisdiction where it is unlawful to do so, where such an offer or solicitation would require a license or registration, or where such an offer or solicitation is subject to restrictions. In particular, any CLX to be issued have not been, and, as of the date of issuance of this Whitepaper, are not intended to be, registered under the securities or similar laws of any jurisdiction, whether or not such jurisdiction considers the CLX to be a security or similar instrument, and specifically, have not been, and, as of the date of issuance of this Whitepaper are not intended to be, registered under the U.S. Securities Act of 1933, as amended, or the securities laws of any state of the United States of America or any other jurisdiction and may not be offered or sold in any jurisdiction where to do so would constitute a violation of the relevant laws of such jurisdiction.

This Whitepaper constitutes neither a prospectus according to Art. 652a of the Swiss Code of Obligations (the "**CO**") or Art. 1156 CO nor a prospectus or basic information sheet according to the Swiss Financial Services Act (the "**FinSA**") nor a listing prospectus nor a simplified prospectus according to Art. 5 of the Swiss Collective Investment Schemes Act (the "**CISA**") nor any other prospectus according to CISA nor a prospectus under any other applicable laws.

The CLX are not expected to be instruments in an offer and sale which are subject to the jurisdiction or oversight of the U.S. Securities Exchange Commission (the "**SEC**"). In any event, however, CLX have not been approved or disapproved by, and are not expected to be approved or disapproved by, the SEC nor by the securities regulatory authority of any state of the United States of America or of any other jurisdiction, and neither the SEC nor any such securities regulatory authority has passed, or is expected to pass, upon the accuracy or adequacy of this Whitepaper.

The distribution of this Whitepaper and the purchase, holding, and/or disposal of CLX may be restricted by law in certain jurisdictions. Persons reading this Whitepaper should inform themselves as to (i) the possible tax consequences, (ii) the legal and regulatory requirements, and (iii) any foreign exchange restrictions or exchange control requirements, which they might encounter under the laws of the countries of their citizenship, residence or domi-cile and which might be relevant to the purchase, holding or disposal of CLX. No action has been taken to authorise the distribution of this Whitepaper in any jurisdiction in which such authorisation might be required.

No action has been or is intended to be taken by CasperLabs Networks AG and/or any of its affiliates in any jurisdiction that would or is intended to, permit a public sale or offering of any CLX, or possession or distribution of this Whitepaper (in preliminary, proof or final form) or any other sale, offering or publicity material relating to the CLX, in any country or jurisdiction where action for that purpose is required. Each recipient of this Whitepaper is reminded that it has received this Whitepaper on the basis that it is a person into whose possession this Whitepaper may be

lawfully delivered in accordance with the laws of the jurisdiction in which it is located and/or bound and it may not nor is it authorised to deliver this document, electronically or otherwise, to any other person. If the recipient receives this document by e-mail, then its use of this e-mail is at its own risk and it is the recipient's responsibility to take precautions to ensure that such e-mail is free from viruses and other items of a destructive nature.

Preliminary Nature of this Whitepaper

This Whitepaper is a draft and the information set out herein is of a preliminary nature. Consequently, neither CasperLabs Networks AG nor any of its affiliates assumes any responsibility that the information set out herein is final or correct and each of the foregoing disclaims, to the fullest extent permitted by applicable law, any and all liability whether arising in tort, contract or otherwise in respect of this Whitepaper. Neither this Whitepaper nor anything contained herein shall form the basis of or be relied on in connection with or act as an inducement to enter into any contract or commitment whatsoever. Recipients should note that the final structuring of CLX and the Casper blockchain is subject to ongoing technical, legal, regulatory and tax considerations and each is, therefore, subject to material changes. In particular, neither the applicability nor the non-applicability of Swiss financial market regulations on the CLX sale has not been confirmed by the Swiss Financial Market Supervisory Authority ("**FINMA**"). CasperLabs Networks AG and all its affiliates reserve the right to not assist in the completion of the software underlying CLX and the CasperLabs blockchain, to not participate in the issuance or creation of CLX or to change the structure of CLX and/or the Casper blockchain for any reason, each at its sole discretion.

Forward-Looking Statements

This Whitepaper includes "forward-looking statements", which are all statements other than statements of historical facts included in this Whitepaper. Words like "believe", "anticipate", "expect", "project", "estimate", "predict", "intend", "target", "assume", "may", "might", "could", "should", "will" and similar expressions are intended to identify such forward-looking statements. Such forward-looking statements involve known and unknown risks, uncertainties and other factors, which may cause the actual functionality, performance or features of the Casper blockchain and/or CLX to be materially different from any future functionality, performance or features expressed or implied by such forward-looking statements. Such forward-looking statements are based on numerous assumptions regarding the CasperLabs Networks AG's and/or any of its affiliates' present and future expectations regarding the development of the Casper blockchain and the associated software.

These forward-looking statements speak only as of the date of this Whitepaper. CasperLabs Networks AG and its affiliates expressly disclaim any obligation or undertaking to release any updates of or revisions to any forward-looking statement contained herein to reflect any change in CasperLabs Networks AG's and/or any of its affiliates' expectations with regard thereto or any change in events, conditions or circumstances on which any such statement is based.

Risk Factors

Furthermore, by accepting this Whitepaper, the recipient of hereof (the "**Recipient**") acknowledges and agrees that it understands the inherent risks associated with blockchain and distributed ledger technology, tokens and cryptocurrencies in general and the CLX in particular, including, but not limited to, those outlined hereinafter.

- **Risks associated with CasperLabs Networks AG's experience**: the Recipient is aware that CasperLabs Networks AG and its affiliates constitute a start-up group of companies. Inability of such companies to manage their affairs, including any failure to attract and retain appropriate personnel, could affect the completion and functionality of the Casper blockchain.
- **Risks associated with CLX relative value**: the Recipient understands and accepts that a purchaser of CLX may experience financial losses relative to other assets, including fiat currency and/or any other cryptocurrency (including any cryptocurrency used to acquire CLX). Potential purchasers and holders of CLX are urged to carefully review this Whitepaper and assess and understand the risk factors relating to the CLX and the Casper blockchain before acquiring CLX (when and if CLX become available).
- **Risks associated with (intellectual) property rights**: the Recipient understands and accepts that, due to a lack of originality of the software and to the immaterial character of the CLX, there may be no title of ownership in and to the intellectual property rights relating to CLX.

- **Risks associated with blockchain**: the Recipient understands and accepts that the smart contract, the underlying software application and software platform (i.e. the Casper blockchain) is still in an early development stage and unproven. The Recipient understands and accepts that there is no warranty that the process for creating the CLX and/or the Casper blockchain will be uninterrupted or error-free and acknowledges that there is an inherent risk that the software could contain weaknesses, vulnerabilities or bugs causing, inter alia, the complete loss of CLX. The Recipient understands and accepts that, after launch of the Casper blockchain, the smart contract and/or underlying protocols and/or the Casper blockchain and/or any other software involved may either delay and/or not execute a contribution due to the overall contribution volume, mining attacks and/or similar events.
- **Risk of weaknesses in the field of cryptography**: the Recipient understands and accepts that cryptography is a technology that evolves relatively fast over time. At the same time, methods and tools to decrypt, access and/or manipulate data stored on a distributed ledger or blockchain are highly likely to progress in parallel and in addition, new technological developments such as quantum computers may pose as of now unpredictable risks to the CLX and the Casper blockchain that could increase the risk of theft or loss of CLX (if and when CLX are created and/or issued).
- **Regulatory risks**: the Recipient understands and accepts that it is possible that certain jurisdictions will apply existing regulations on, or introduce new regulations addressing, distributed ledger technology and/or blockchain technology based applications, which may be contrary to the current setup of the smart contract or the CasperLabs Networks AG project and which may, inter alia, result in substantial modifications of the smart contract and/or the CasperLabs Networks AG project, including its termination and the loss of the CLX, if and when created and/or issued, or entitlements to receive CLX, for the Recipient.
- **Risks associated with abandonment / lack of success**: the Recipient understands and accepts that the creation of the CLX and the development of the Casper blockchain as well as the CasperLabs Networks AG project may be abandoned for a number of reasons, including lack of interest from the public, lack of funding, lack of prospects (e.g. caused by competing projects) and legal, tax or regulatory considerations. The Recipient therefore understands that there is no assurance that, even if the CLX/CasperLabs blockchain project is partially or fully developed and launched, the Recipient will receive any benefits through the CLX held by it (if and when created and/or issued).
- **Risks associated with a loss of private key**: the Recipient understands and accepts that CLX, if and when created and/or issued, will only be accessed by using a wallet technically compatible with CLX and with a combination of the Recipient's account information (address) and private key, seed or password. The Recipient understands and accepts that if its private key or password gets lost or stolen, the CLX associated with the Recipient's account (address) will be unrecoverable and will be permanently lost.
- **Risks associated with wallets:** the Recipient understands and accepts that CasperLabs Networks AG or any of its affiliates, employees, partners or advisors are in no way responsible for the wallet to which any CLX are transferred. The Recipient understands and agrees that it is solely responsible for the access and security of its wallet, for any security breach of its wallet and/or with any loss of CLX resulting from its wallet service provider, including any termination of the service by the wallet provider and/or bankruptcy of the wallet provider.
- **Risks associated with theft/hacks**: the Recipient understands and accepts that the smart contract, the website, the underlying software application and software platform (i.e. the Casper blockchain), during its development and after its launch, may be exposed to attacks by hackers or other individuals that could result in an inability to launch the Casper blockchain or the theft or loss of CLX. Any such event could also result in the loss of financial and other support of the CasperLabs Networks AG project impacting the ability to develop the CasperLabs Networks AG project and Casper blockchain.
- **Risks associated with mining attacks**: the Recipient understands and accepts that, as with other cryptocurrencies and tokens, if and when launched, the Casper blockchain is susceptible to attacks relating to validators. Any successful attack presents a risk to the smart contract, expected proper execution and sequencing of transactions, and expected proper execution and sequencing of contract computations.
- **Risks associated with a lack of consensus**: the Recipient understands and accepts that the network of validators will be ultimately in control of the genesis block and future blocks and that there is no warranty or assurance that the network of validators will perform their functions and reach proper consensus and allocate the CLX

to the Recipient as proposed by any terms. The Recipient further understands that a majority of the validators could agree at any point to make changes to the software and/or smart contracts and to run the new version of the software and/or smart contracts. Such a scenario could lead to the CLX losing intrinsic value.

- **Risks associated with liquidity of CLX**: the Recipient understands and accepts that with regard to the CLX, if and when created and/or issued, no market liquidity may be guaranteed and that the value of CLX relative to other assets, including fiat currency and/or any other cryptocurrency (including any cryptocurrency used to acquire CLX) over time may experience extreme volatility or depreciate in full (including to zero) resulting in loss that will be borne exclusively by the Recipient.
- **Risks associated with forking**: the Recipient understands and accepts that hard and soft forks as well as similar events may, inter alia, lead to the creation of new or competing tokens to the CLX, adversely affect the functionality, convertibility or transferability or result in a full or partial loss of units or reduction (including reduction to zero) of value of the Recipient's CLX (if and when created and/or issued).

CHAPTER 1

Blockchain Design

1.1 Communications

1.1.1 Node Discovery

Nodes form a peer-to-peer network, constantly communicating with each other to reach consensus about the state of the blockchain. A node is not necessarily a single physical machine, but it appears as a single logical entity to the rest of its peers by having a unique ID and address where it responds to requests.

Nodes periodically try to discover each other based on elements of the Kademlia protocol. Unlike the original Kademlia which was using UDP, nodes are using point-to-point gRPC calls for communication. According to this protocol every `Node` has the following properties:

- `id` is a Keccak-256 digest of the Public Key from the SSL certificate of the node
- `host` is the public endpoint where the node is reachable
- `discovery_port` is where the gRPC service implementing the `KademliaService` is listening
- `protocol_port` is where the gRPC service implementing the consensus related functionality is listening

The `KademliaService` itself has to implement only two methods:

- `Ping` is used by the `sender` to check if the callee is still alive. It also gives the callee the chance to update its list of peers and remember that it has seen the `sender` node.
- `Lookup` asks the node to return a list of Nodes that are closest to the `id` the `sender` is looking for, where distance between two Nodes based on the longest common prefix of bits in their `id`.

At startup the nodes should be configured with the address of a well known peer to bootstrap themselves from. To discover other nodes they can pick from multiple strategies:

- Perform one-time lookup on their own `id` by the bootstrap node (which doesn't know them yet) to receive a list of peers closest to itself. Recursively perform the same lookup with those peers to accumulate more and more addresses until there is nothing new to add.

- Periodically construct artificial keys to try to find peers at certain distances from `id` and perform a lookup by a random node.

1.1.2 Block Gossiping

Nodes propose Blocks in parallel by finding Deploys that can be applied independently of each other. Whenever a new Block is formed, it has to propagate through the network to become part of the consensus. This is achieved by nodes making calls to each other via gRPC to invoke methods on their `GossipService` interface which should be listening on the `protocol_port` of the `Node` that represents the peers in the network.

Principles

We call the method by which information is disseminated on the network *gossiping*. Gossiping means that when a node comes across new bits of information it will relay it to a selection of its peers, who do the same, eventually saturating the network, i.e. get to the point where everyone has been notified.

Nodes need three layers of information about Blocks to be able to participate in the consensus:

1. Block meta-data, e.g. parent relationships, validator weights, state hashes.
2. Deploys that were included in given Block.
3. Global State, to be able to run the Deploys, validate Blocks and build new ones on top of them.

Out of these only the top two are gossiped between nodes; —the Global State, they have to calculate themselves by running Deploys.

We have the following requirements for our gossiping approach:

- It should be efficient, i.e. minimize the network traffic while maximizing the rate at which we reach full saturation.
- Node operators should have a reasonable expectation that network traffic (a finite resource) will scale linearly with the amount of Deploys across the network while being less affected by the total number of nodes. This means the load should be distributed among the peers rather than create hot-spots.

To achieve these we have the following high-level approach:

- Gossip only the meta-data about the Blocks to minimize the amount of data transfer.
- Full Blocks can be served on demand when the gossiped meta-data is *new.*
- Nodes should pick a *relay factor* according to how much network traffic they can handle and find that many nodes to gossip to — nodes for which the information is *new*.
- Nodes should pick a *relay saturation* target beyond which point they don't try to push to new peers, so the last ones to get a message don't have to contact every other peer in futility.
- Nodes should try to spread the information mostly to their closer neighbors (in terms of Kademlia distance), but also to their farther away peers to accelerate the spread of information to the far reaches of the network.

Picking Nodes for Gossip

As we established in the *Node Discovery* section, the nodes maintain a list of peers using a Kademlia table. The table has one bucket for each possible distance between the bits of their IDs, and they pick a number k to be length of the list of nodes in each of these buckets that they keep track of.

In statistical terms, half of the nodes in the network will fall in the first bucket, since the first bit of their ID will either be 0 or 1. Similarly, each subsequent bucket holds half of the remainder of the network. Since k is the same for each

bucket, this means that nodes can track many more of their closest neighbors than the ones which are far away from them. This is why in Kademlia, as we perform lookups we get closer and closer to the best possible match anyone knows about.

In practice, this means that if we have a network of 50 nodes and a k of 10, then from the perspective of any given peer, 25 nodes fall into the first bucket, but it only tracks 10 of them; ergo it will never gossip to 15 nodes from the other half that it doesn't have the room to track in its table.

We can use this skewness to our advantage: say we pick a *relay factor* of 3; if we make sure to always notify 1 node in the farthest non-empty buckets, and 2 in the closer neighborhood, we can increase the chance that the information gets to those Nodes on the other half of the board that we don't track.

This is illustrated by the following diagram: The black actor on the left represents our node and the vertical partitions represent the distance from it. The split in the middle means the right half of the board falls under a single bucket in the Kademlia table. The black dots on the board are the nodes we track, the greys are ones we don't. We know few peers from the right half of the network, but much more on the left because it's covered by finer and finer grained buckets. If we pick nodes to gossip to evenly across the full distance spectrum, and the Node on the right follows the same rule, it will start distributing our message on that side of the board with a slightly higher chance than bouncing it back to the left.

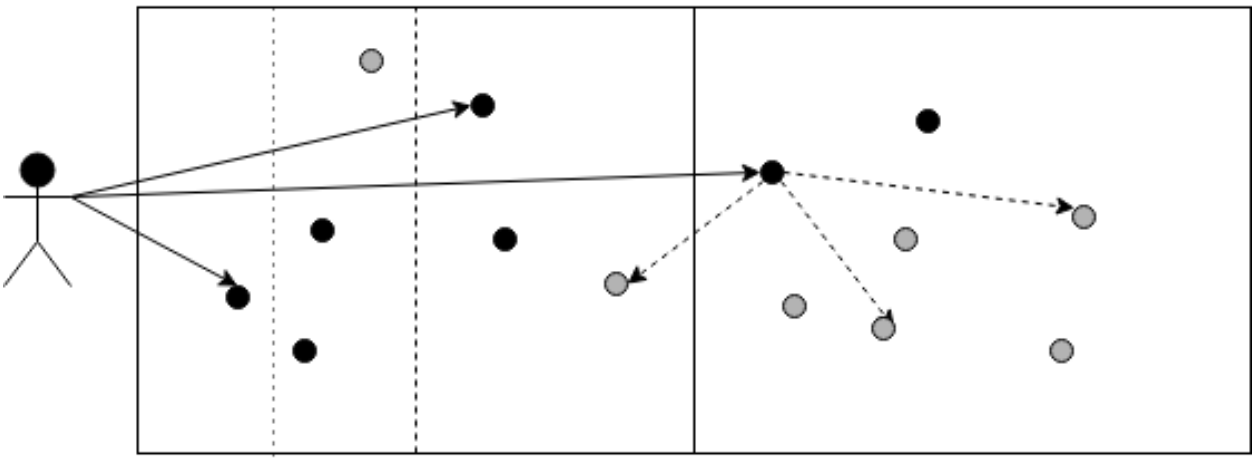

Fig. 1: Message propagation among peers

In terms of probabilities of reaching a grey node in the 2nd round, or just the message being on the right side of the board in the 1st or 2nd round of message passing, the gains are marginal and depend on how many peers there are in the Kademlia table. We could give higher weights to the buckets that reach the un-tracked parts of the network, but the effects will have to be simulated.

In practice the messages don't have to reach every node on the network. Achieving 100% saturation would be impractical as it would require a high level of redundancy, i.e. a node receiving the same message multiple times from different peers. Tracking who saw a message could bloat the message size or open it up to tampering. But even if a node isn't notified about a particular Block *right now*, it has equal chances of receiving the next Block that builds on top of that, at which point it can catch up with the missing chain.

Therefore nodes should have a *relay saturation* value beyond which they don't try to gossip a message to new nodes. For example, if we pick a *relay factor* of 5 and a *relay saturation* of 80% then it's enough to try and send to 25 nodes maximum. If we find less than 5 peers among them to whom the information was *new*, then we achieved a saturation beyond 80%. This prevents the situation when the last node to get the message has to contact every other node in a futile attempt to spread it 5 more times. Assuming that every node tracks a random subset of peers in the network, the saturation we observe in the nodes we try to contact is an approximation of the saturation in the whole network, with the accuracy depending on how many nodes we tried.

```
algorithm BlockGossip is
    input: message M to send,
           relay factor rf,
           relay saturation rs,
           kademlia table K
    output: number of messages sent s

    s <- 0
    P <- flatten the peers in K, ordered by distance from current node
    G <- partition P into R equal sized groups

    n <- an empty set to track notified peers
    m <- rf / (1 - rs) // the maximum number of peers to try to send to
    i <- 0 // the index of the current group

    while i < sizeof(G) and s <= rf and sizeof(n) < m:
        p <- a random peer in G(i)-n or None if empty
        if p is None then
            i <- i + 1
        else
            n <- n + p
            r <- the result of sending M to p, indicating whether M was new to p
            if r is true then
                i <- i + 1
                s <- s + 1

    return s
```

Fairness

We can rightfully ask how the gossip algorithm outlined above fares in the face of malicious actors that don't want to take their share in the data distribution, i.e. what happens if a node decides not to propagate the messages?

The consensus protocol has a built-in protection against lazy validators: to get their fees from a Block produced by somebody else, they have to build on top of it. When they do that, they have to gossip about it, otherwise it will not become part of the DAG or it can get orphaned if conflicting blocks emerge, so it's in everyone's interest for gossiping to happen at a steady pace.

What if they decide to announce *their* Blocks to *everyone* but never relay other Blocks from other nodes? They have a few incentives against doing this:

- If everybody would be doing it then the nodes unknown to the creators would get it much later and might produce conflicting blocks, the consensus would slow down.
- When they finally announce a Block they built *everyone* would try to download it directly from them, putting extra load on their networks; plus they might have to download extra Blocks that the node failed to relay before.
- If we have to relay to a 100 nodes directly, it could easily take longer for each of the 100 to download it from 1 node then for 10 nodes to do so and then relay to 10 more nodes each.

Having a *relay factor* together with the mechanism of returning whether the information was *new*, has the following purpose:

- By indicating that the information was new, the callee is signalling to the caller that once it has done the validation of the Block, it will relay the information; therefore the caller can be content that by informing this node it carried out the number of gossips it set out to do, i.e., it will have to serve the full Blocks up to R number of times.

- By indicating that the information wasn't new, the callee is signalling that it will not relay the information any longer, and therefore the caller should pick another node if it wants to live up to its pledge of relaying to R number of new nodes.

Nodes expect the ones which indicated that the Block meta-data was new to them to later attempt to download the full Blocks. This may not happen, as other nodes may notify them too, in which case they can download some Blocks from here, some from there.

There are two forms of lying that can happen here:

1. The callee can say the information wasn't anything new, but then attempt to download the data anyway. Nodes may create a disincentive for this by tracking each others' *reputation* and block nodes that lied to them.
2. The callee can say the information was new but not relay. This goes against their own interest as well, but it's difficult to detect. A higher relay factor can compensate for the amount of liars on the network.

Nodes may also use reputation tracking and blocking if they receive notifications about Blocks that cannot be validated or that the notifier isn't able to serve when asked.

Syncing the DAG

Let's take a closer look at how the methods supported by the `GossipService` can be used to spread the information about Blocks between nodes.

NewBlocks

When a node creates one or more new Blocks, it should pick a number of peers according to its *relay factor* and call `NewBlocks` on them, passing them the new `block_hashes`. The peers check whether they already have the corresponding blocks: if not, they indicate that the information is *new* and schedule a download from the `sender`, otherwise the caller looks for other peers to notify.

By only sending block hashes can we keep the message size to a minimum. Even block headers need to contain a lot of information for nodes to be able to do basic verification; there's no need to send all that if the receiving end already knows about it.

Note that the `sender` value is known to be correct because nodes talking to each other over gRPC must use two-way SSL encryption, which means the callee will see the caller's public key in the SSL certificate. The `sender` can only be a `Node` with an `id` that matches the hash of the public key.

StreamAncestorBlockSummaries

When a node receives a `NewBlock` request about hashes it didn't know about, it must synchronize its Block DAG with the `sender`. One way to do this is to have some kind of *download manager* running in the node which:

- maintains partially connected DAG of `BlockSummary` records that it has seen
- tries to connect the new bits to the existing ones by downloading them from the senders
- tracks which nodes notified it about each Block, to know alternative sources to download from
- tracks which Blocks it promised to relay to other nodes
- downloads and verifies full Blocks
- notifies peers about validated Blocks if it promised to do so

`StreamAncestorBlockSummaries` is a high level method that the caller node can use to ask another for a *way to get to the block it just shouted about*. It's a method to traverse from the *target* block *backwards* along its parents until every ancestor path can be connected to the DAG of the caller. It has the following parameters:

- `target_block_hashes` is typically the hashes of the new Blocks the node was notified about, but if multiple iterations are needed to find the connection points then they can be further back the DAG.
- `known_block_hashes` can be supplied by the caller to provide an early exist criteria for the traversal. These can, for example, include the hashes close to the tip of the callers DAG, forks, last Blocks seen from validators, and approved Blocks (i.e., Blocks with a high safety threshold).
- `max_depth` can be supplied by the caller to limit traversal in case the `known_block_hashes` don't stop it earlier. This can be useful during iterations when we have to go back *beyond* the callers approved blocks, in which case it might be difficult to pick known hashes.

The result should be a partial traversal of the DAG in *reverse BFS order* returning a stream of `BlockSummaries` that the caller can partially verify, merge into its DAG of pending Blocks, then recursively call `StreamAncestorBlockSummaries` on any Block that didn't connect with a known part of the DAG. Ultimately, all paths lead back to the Genesis; so eventually we should find the connection, or the caller can decide to give up pursuing a potentially false lead from a malicious actor.

The following diagram illustrates the algorithm. The dots in the graph represent the Blocks; the ones with a thicker outer ring are the ones passed as `target_block_hashes`. The dashed rectangles are what's being returned in a stream from one invocation to `StreamAncestorBlockSummaries`.

1. Initially the we only know about the black Blocks, which form our DAG.
2. We are notified about the white Block, which is not yet part of our DAG.
3. We call `StreamAncestorBlockSummaries` passing the white Block's hash as target and a `max_depth` of 2 (passing some of our known block hashes as well).
4. We get a stream of two `BlockSummary` records in reverse order from the 1st, and we add them to our DAG. But, we can see that the grandparents of the of the white Block are still not known to us.
5. We call `StreamAncestorBlockSummaries` a 2nd time passing the grandparents' hashes as targets.
6. From the 2nd stream we can see that at least one of the Blocks is connected to the tip our DAG, but there are again Blocks with missing dependencies.
7. We call `StreamAncestorBlockSummaries` a 3rd time and now we can form a full connection with the known parts of the DAG, there are no more Blocks with missing parents.

The following algorithm describes the server's role:

```
algorithm StreamAncestorBlockSummaries is
    input: target block hashes T,
           known block hashes K,
           maximum depth m,
           block summary map B
    output: stream of ancestry DAG in child to parent order

    G <- an empty DAG of block hashes
    Q <- an empty priority queue of (depth, hash) pairs
    A <- an empty list of ancestor block summaries
    V <- an empty set of visited block hashes

    for each hash h in T do
        push (0, h) to Q

    while Q is not empty do
```

(continues on next page)

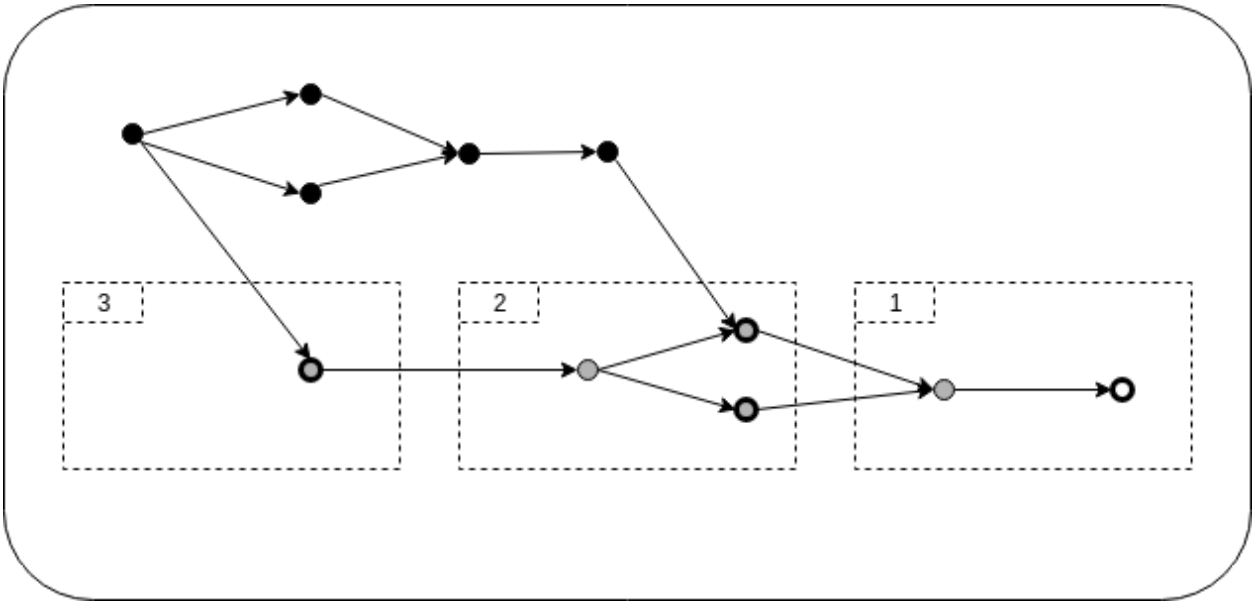

Fig. 2: Backwards traversal to sync the DAGs

(continued from previous page)

```
        pop (d, h) from Q
        if h in V then
            continue
        if h is not in B then
            continue
        b <- B(h)
        A <- A + b
        V <- V + h
        for each parent hash p of b do
            G <- G + (h, p)
            if d < m and p not in K do
                push (d+1, p) to Q

    return stream of A
```

And the next one depicts syncing DAGs from the client's perspective:

```
algorithm SyncDAG is
    input: sender node s,
           new block hashes N,
           block summary map B
    output: new block summaries sorted in topological order from parent to child

    G <- an empty DAG of block hashes
    A <- an empty map of block summaries
    m <- a suitable maximum depth, say 100

    function Traverse is
        input: block hashes H
        output: number of summaries traversed
        S <- s.StreamAncestorBlockSummaries(H, m)
        for each block summary b in S do
```

(continues on next page)

(continued from previous page)

```
            if b cannot be validated then
                return 0
            if b is not connected to H with a path or the path is longer than m then
                return 0
            if we see an abnormally shallow and wide DAG being built then
                // the server is trying to feed exponential amount of data
                // by branching wide while staying within the maximum depth
                return 0
            h <- the block hash of b
            A(h) <- b
            for each parent p of b do
                if p is not in B do
                    G <- G + (p, h)
        return sizeof(S)

    Traverse(N)

    define "hashes in G having missing dependencies" as
    hash h having no parent in G and h is not in B

    while there are hashes in G with missing dependencies do
        H <- the hashes in G with missing dependencies
        if Traverse(H) equals 0 then
            break

    if there are hashes in G with missing dependencies then
        return empty because the DAG did not connect
    else
        return A(h) for hashes h in G sorted in topological order from parent to child
```

`SyncDAG` needs to have some protection against malicious actors trying to lead it down the garden path and feeding it infinite streams of data.

Once `SyncDAG` indicates that the summaries from the `sender` of `NewBlocks` have a common ancestry with the DAG we have, we can schedule the download of data.

```
Q <- an empty queue of blocks to sync
G <- an empty DAG of block dependencies in lieu of a download queue
S <- a map of source information we keep about blocks
GBS <- the global block summary map
GFB <- the global full block map

function NewBlocks is
    input: sender node s,
           new block hashes N
    output: flag indicating if the information was new

    H <- find hashes h in N where h is not in GBS
    if H is not empty then
        push (s, H) to Q
        return true
    return false


parallel threads Synchronizer is
    for each (s, N) message m in Q do
        D <- SyncDAG(s, N, GBS)
```

(continues on next page)

(continued from previous page)

```
        for each block summary b in D do
            r <- if hash of b is in N then true else false
            ScheduleDownload(b, s, r)


function ScheduleDownload is
    input: block summary b,
           sender node s,
           relay flag r

    h <- the hash of b

    if h is in GBS then
        return

    if h is in S then
       S(h) <- S(h) with extra source s
       if r is true then
           S(h) <- S(h) with relay true
    else
       N <- the list of potential source nodes for b with single element s
       S(h) <- (b, N, r)
       if any parent p of b is in pending downloads G then
          add h as a dependant of p in G
       else
          add h to G without dependencies


parallel threads Downloader is
    rf <- the relay factor from config, say 5
    rs <- the relay saturation from config, say 0.8

    for each new item added to G or after an idle timeout do
        while we can mark a new hash h in G without dependencies as being downloaded␣
↪do
            (b, N, r) <- S(h)

            n <- a random node in N
            f <- the full block returned by n.GetBlockChunked(h)

            if f is valid then
                GFB(h) <- f
                GBS(h) <- b
                remove h from G

                if r is true then
                    K <- the current Kademlia table of peers
                    s <- the current node
                    BlockGossip(NewBlocks(s, h), rf, rs, K)
```

GetBlockChunked

Full Blocks containing all the deploys can get big, and the HTTP/2 protocol underlying gRPC has limits on the maximum message size it can transmit in a request body. Therefore, we need to break the payload up into smaller

chunks, transfer them as streams, and reconstruct them on the other side.

The caller should keep track of the data it receives and compare them to the `content_length` it got initially in the header to make sure it's not being fed an infinite stream of data.

In theory the method could return a stream of multiple blocks, but asking for them one by one from multiple peers as the notifications arrive to the node about alternative sources, should be favoured over downloading from a single source anyway.

StreamLatestMessages

When a new node joins the network it should ask one or more of its peers about the tips of their DAG, i.e. the Blocks on which they themselves would be proposing new blocks. This can be followed by an arbitrary number of calls to the `StreamAncestorBlockSummaries` until the new node has downloaded and partially verified the full DAG using the algorithms outlined above. It's worth cross-correlating the tips of multiple nodes to avoid being lied to by any single malicious actor.

The same method can also be utilized for pull-based gossiping by picking a random peer every now and then and asking for their tips of the DAG. This compensates for the diminishing returns of trying to reach 100% saturation by push based gossiping: as we reach full saturation it's becoming harder to find a node that doesn't have the information yet, so an increasing number of messages are sent in futility to peers that have already received an equivalent message from another peer; but for the small percentage of nodes that haven't yet been reached it becomes increasingly probable that any peer they communicate with has received the information.

Finally the following sequence diagram demonstrates the life cycle of Block propagation among nodes. The dashed blocks have been left unconnected for brevity but they do the same thing as the ones on the left side.

1.1.3 Deploy Gossiping

With certain consensus protocols, nodes can only produce blocks when its their turn to do so. To make sure that users can send their deploys to any node and see them included in the next available block on the network, regardless of which validator creates that block, nodes need to gossip about deploys between each other, independently of blocks.

The principles and mechanisms of deploy gossiping are exactly the same as the block gossiping depicted above.

A block may contain one or more deploys that a node already has, so when downloading a block only the deploy hashes and their execution results are included and the deploy body and header are excluded. A downloading node derives which deploys it is missing via the deploy hashes and downloads them separately.

When a node receives a deploy (either from a user or via deploy gossiping), it should inform some of its peers that it has the deploy available for download. The semantics are exactly the same as for *NewBlocks* (using the same algorithm).

Nodes use this method to retrieve one or more deploys from each other. Typically one deploy is streamed when deploy gossiping, but when a node determines it is missing one or more deploy bodies in a received block (derived from the deploy hashes in the block) it will request all of the deploys it is missing.

The semantics are very similar to *GetBlockChunked* but instead of containing a single item, the stream alternates between header chunks and contents, with each group corresponding to one deploy hash in the request.

1.2 Global State

1.2.1 Introduction

The "global state" is the storage layer for the blockchain. All accounts, contracts, and any associated data they have are stored in the global state. Our global state has the semantics of a key-value store (with additional permissions

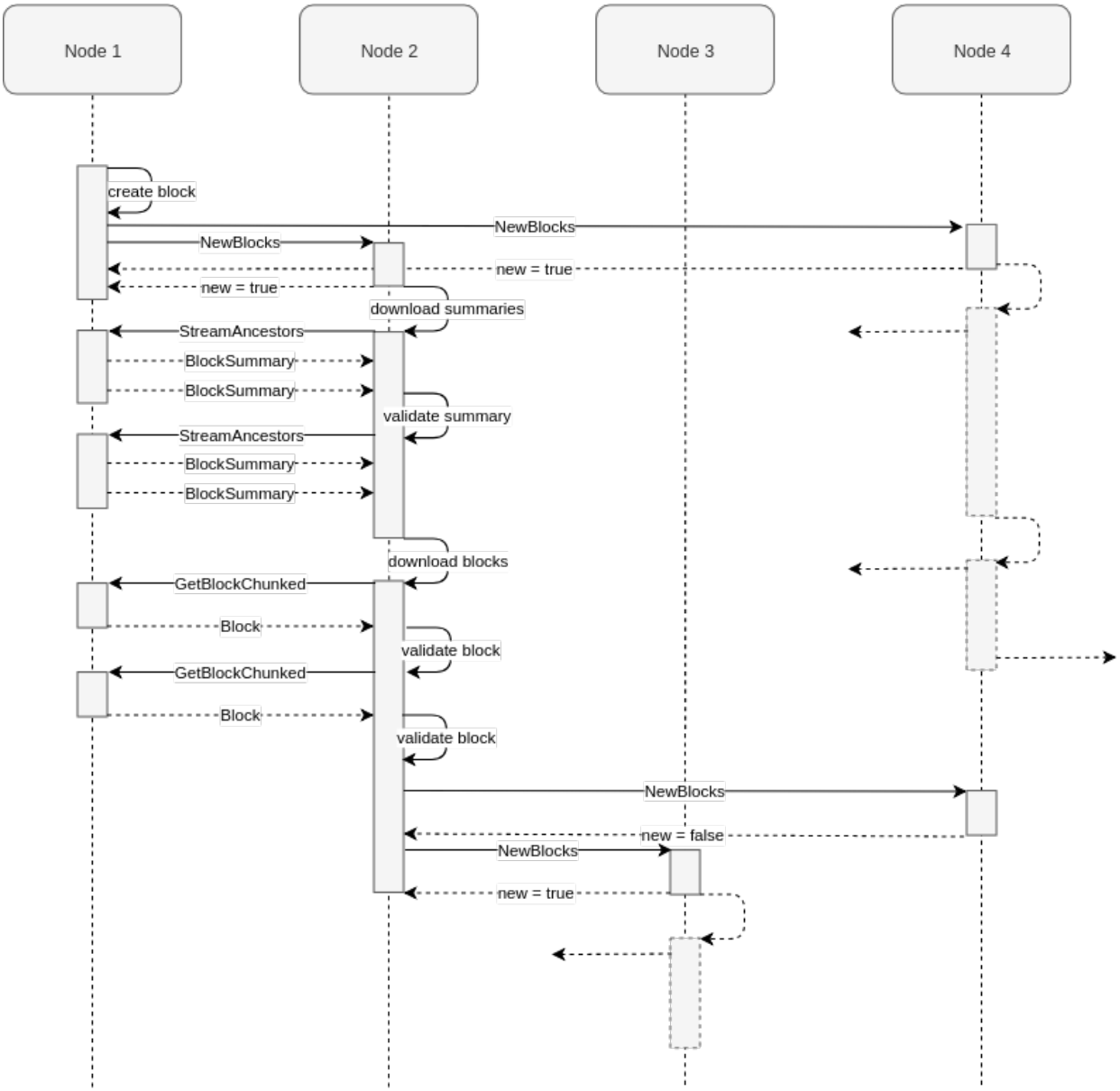

Fig. 3: Block Gossiping

logic, since not all users can access all values in the same way). Each block causes changes to this global state because of the execution of the deploys it contains. In order for validators to efficiently judge the correctness of these changes, information about the new state needs to be communicated succinctly. Moreover, we need to be able to communicate pieces of the global state to users, while allowing them to verify the correctness of the parts they receive. For these reasons, the key-value store is implemented as a *Merkle trie*.

In this chapter we describe what constitutes a "key", what constitutes a "value", the permissions model for the keys, and the Merkle trie structure.

1.2.2 Keys

A *key* in the global state is one of the following four data types:

- 32-byte account identifier (called an "account identity key")
- 32-byte immutable contract identifier (called a "hash key")
- 32-byte reference identifier (called an "unforgable reference")

We cover each of these key types in more detail in the sections that follow.

Account identity key

This key type is used specifically for accounts in the global state. All accounts in the system must be stored under an account identity key, and no other type. The 32-byte identifier which represents this key is derived from the `blake2b256` hash of the public key used to create the associated account (see *Accounts* for more information).

Hash key

This key type is used for storing contracts immutably. Once a contract is written under a hash key, that contract can never change. The 32-byte identifier representing this key is derived from the `blake2b256` hash of the deploy hash (see *Block Structure* for more information) concatenated with a 4-byte sequential ID. The ID begins at zero for each deploy and increments by 1 each time a contract is stored. The purpose of this ID is to allow each contract stored in the same deploy to have a unique key.

Unforgable Reference (`URef`)

This key type is used for storing any type of value except `Account`. Additionally, `URef`s used in contracts carry permission information with them to prevent unauthorized usage of the value stored under the key. This permission information is tracked by the runtime, meaning that if a malicious contract attempts to produce a `URef` with permissions that contract does not actually have, we say the contract has attempted to "forge" the unforgable reference, and the runtime will raise a forged `URef` error. Permissions for a `URef` can be given across contract calls, allowing data stored under a `URef` to be shared in a controlled way. The 32-byte identifier representing the key is generated randomly by the runtime (see *Execution Semantics* for for more information).

1.2.3 Values

A value stored in the global state is a `StoredValue`. A `StoredValue` is one of three possible variants:

- A `CLValue`
- A contract
- An account

We discuss `CLValue` and contract in more detail below, details about accounts can be found in *Accounts*.

Each `StoredValue` is serialized when written to the global state. The serialization format consists of a single byte tag, indicating which variant of `StoredValue` it is, followed by the serialization of that variant. The tag for each variant is as follows:

- `CLValue` is `0`
- `Account` is `1`
- `Contract` is `2`

The details of `CLType` serialization is in the following section. Using the serialization format for `CLValue` as a basis, we can succinctly write the serialization rules for contracts and accounts:

- contracts serialize in the same way as data with `CLType` equal to `Tuple3(List(U8), Map(String, Key), Tuple3(U32, U32, U32))`;
- accounts serialize in the same way as data with `CLType` equal to `Tuple5(FixedList(U8, 32), Map(String, Key), URef, Map(FixedList(U8, 32), U8), Tuple2(U8, U8))`.

Note: `Tuple5` is not a presently supported `CLType`, however it is clear how to generalize the rules for `Tuple1`, `Tuple2`, `Tuple3` to any size tuple.

Note: links to further serialization examples and a reference implementation are found in *Appendix B*.

CLValue

`CLValue` is used to describe data that is used by smart contracts. This could be as a local state variable, input argument or return value. A `CLValue` consists of two parts: a `CLType` describing the type of the value, and an array of bytes which represent the data in our serialization format.

`CLType` is described by the following recursive data type:

```
enum CLType {
   Bool, // boolean primitive
   I32, // signed 32-bit integer primitive
   I64, // signed 64-bit integer primitive
   U8, // unsigned 8-bit integer primitive
   U32, // unsigned 32-bit integer primitive
   U64, // unsigned 64-bit integer primitive
   U128, // unsigned 128-bit integer primitive
   U256, // unsigned 256-bit integer primitive
   U512, // unsigned 512-bit integer primitive
   Unit, // singleton value without additional semantics
   String, // e.g. "Hello, World!"
   URef, // unforgable reference (see above)
   Key, // global state key (see above)
   Option(CLType), // optional value of the given type
   List(CLType), // list of values of the given type (e.g. Vec in rust)
   FixedList(CLType, u32), // same as `List` above, but number of elements
                           // is statically known (e.g. arrays in rust)
   Result(CLType, CLType), // co-product of the the given types;
                           // one variant meaning success, the other failure
   Map(CLType, CLType), // key-value association where keys and values have the given␣
↪types
   Tuple1(CLType), // single value of the given type
   Tuple2(CLType, CLType), // pair consisting of elements of the given types
   Tuple3(CLType, CLType, CLType), // triple consisting of elements of the given types
```

(continues on next page)

(continued from previous page)

```
    Any // Indicates the type is not known
}
```

All data which can be assigned a (non-`Any`) `CLType` can be serialized according to the following rules (this defines the CasperLabs serialization format):

- Boolean values serialize as a single byte; `true` maps to `1`, while `false` maps to `0`.
- Numeric values consisting of 64 bits or less serialize in the two's complement representation with little-endian byte order, and the appropriate number of bytes for the bit-width.
  - E.g. `7u8` serializes as `0x07`
  - E.g. `7u32` serializes as `0x07000000`
  - E.g. `1024u32` serializes as `0x00040000`
- Wider numeric values (i.e. `U128`, `U256`, `U512`) serialize as: one byte given the length of the subsequent number (in bytes), followed by the two's complement representation with little-endian byte order. The number of bytes should be chosen as small as possible to represent the given number. This is done to reduce the size of the serialization in the case of small numbers represented within a wide data type.
  - E.g. `U512::from(7)` serializes as `0x0107`
  - E.g. `U512::from(1024)` serializes as `0x020004`
  - E.g. `U512::from("123456789101112131415")` serializes as `0x0957ff1ada959f4eb106`
- Unit serializes to an empty byte array.
- Strings serialize as a 32-bit integer representing the length in bytes (note: this might be different than the number of characters since special characters, such as emojis, take more than one byte), followed by the UTF-8 encoding of the characters in the string.
  - E.g. `"Hello, World!"` serializes as `0x0d00000048656c6c6f2c20576f726c6421`
- Optional values serialize with a single byte tag, followed by the serialization of the value it self. The tag is equal to `0` if the value is missing, and `1` if it is present.
  - E.g. `None` serializes as `0x00`
  - E.g. `Some(10u32)` serializes as `0x010a000000`
- A list of values serializes as a 32-bit integer representing the number of elements in the list (note this differs from strings where it gives the number of *bytes*), followed by the concatenation of each serialized element.
  - E.g. `List()` serializes as `0x00000000`
  - E.g. `List(1u32, 2u32, 3u32)` serializes as `0x030000000100000002000000030000000`
- A fixed-length list of values serializes as simply the concatenation of the serialized elements. Unlike a variable-length list, the length is not included in the serialization because it is statically known by the type of the value.
  - E.g. `[1u32, 2u32, 3u32]` serializes as `0x010000000200000003000000`
- A `Result` serializes as a single byte tag followed by the serialization of the contained value. The tag is equal to `1` for the success variant and `0` for the error variant.
  - E.g. `Ok(314u64)` serializes as `0x013a01000000000000`
  - E.g. `Err("Uh oh")` serializes as `0x00050000005568206f68`
- Tuples serialize as the concatenation of their serialized elements. Similar to `FixedList` the number of elements is not included in the serialization because it is statically known in the type.

  - E.g. `(1u32, "Hello, World!", true)` serializes as `0x010000000d00000048656c6c6f2c20576f726c64`

- A `Map` serializes as a list of key-value tuples. There must be a well-defined ordering on the keys, and in the serialization the pairs are listed in ascending order. This is done to ensure determinism in the serialization, as `Map` data structures can be unordered.
- `URef` values serialize as the concatenation of its address (which is a fixed-length list of `u8`) and a single byte tag representing the access rights. Access rights are converted as follows:

Access Rights	Serialization
`NONE`	0
`READ`	1
`WRITE`	2
`READ_WRITE`	3
`ADD`	4
`READ_ADD`	5
`ADD_WRITE`	6
`READ_ADD_WRITE`	7

- `Key` values serialize as a single byte tag representing the variant, followed by the serialization of the data that variant contains. For most variants this is simply a fixed-length 32 byte array. The exception is `Key::URef` which contains a `URef`, so its data serializes per the description above. The tags are as follows: `Key::Account` serializes as `0`, `Key::Hash` as `1`, `Key::URef` as `2`.

`CLType` itself also has rules for serialization. A `CLType` serializes as a single byte tag, followed by the concatenation of serialized inner types, if any (e.g. lists, and tuples have inner types). `FixedList` is a minor exception because it also includes the length in the type, however this simply means that the length included in the serialization as well (as a 32-bit integer, per the serialization rules above), following the serialization of the inner type. The tags are as follows:

`CLType`	Serialization Tag
`Bool`	0
`I32`	1
`I64`	2
`U8`	3
`U32`	4
`U64`	5
`U128`	6
`U256`	7
`U512`	8
`Unit`	9
`String`	10
`URef`	11
`Key`	12
`Option`	13
`List`	14
`FixedList`	15
`Result`	16
`Map`	17
`Tuple1`	18
`Tuple2`	19
`Tuple3`	20
`Any`	21

A complete `CLValue`, including both the data and the type can also be serialized (in order to store it in the global

state). This is done by concatenating: the serialization of the length (as a 32-bit integer) of the serialized data (in bytes), the serialized data itself, and the serialization of the type.

Contracts

Contracts are a special value type because they contain the on-chain logic of the applications running on the CasperLabs system. A *contract* contains the following data:

- a wasm module
- a collection of named keys
- a protocol version

The wasm module must contain a function named `call` which takes no arguments and returns no values. This is the main entry point into the contract. Moreover, the module may import any of the functions supported by the CasperLabs runtime; a list of all supported functions can be found in *Appendix A*.

Note: though the `call` function signature has no arguments and no return value, within the `call` function body the `get_named_arg` runtime function can be used to accept arguments (by ordinal) and the `ret` runtime function can be used to return a single `CLValue` to the caller.

The named keys are used to give human-readable names to keys in the global state which are important to the contract. For example, the hash key of another contract it frequently calls may be stored under a meaningful name. It is also used to store the `URef`s which are known to the contract (see below section on Permissions for details).

The protocol version says which version of the CasperLabs protocol this contract was compiled to be compatible with. Contracts which are not compatible with the active major protocol version will not be executed by any node in the CasperLabs network.

1.2.4 Permissions

There are three types of actions which can be done on a value: read, write, add. The reason for add to be called out separately from write is to allow for commutativity checking. The available actions depends on the key type and the context. This is summarized in the table below:

Key Type	Available Actions
Account	Read + Add if the context is the current account otherwise None
Hash	Read
URef	See note below
Local	Read + Write + Add if the context seed used to construct the key matches the current context

Permissions for `URef`s

In the runtime, a `URef` carries its own permissions called `AccessRights`. Additionally, the runtime tracks what `AccessRights` would be valid for each `URef` to have in each context. As mentioned above, if a malicious contract attempts to use a `URef` with `AccessRights` that are not valid in its context, then the runtime will raise an error; this is what enforces the security properties of all keys. By default, in all contexts, all `URef`s are invalid (both with any `AccessRights`, or no `AccessRights`); however, a `URef` can be added to a context in the following ways:

- it can exist in a set of "known" `URef`s
- it can be freshly created by the runtime via the `new_uref` function
- for called contracts, it can be passed in by the caller via the arguments to `call_contract`
- it can be returned back to the caller from `call_contract` via the `ret` function

Note: that only valid `URefs` may be added to the known `URefs` or cross call boundaries; this means the system cannot be tricked into accepted a forged `URef` by getting it through a contract or stashing it in the known `URefs`.

The ability to pass `URefs` between contexts via `call_contract` / `ret`, allows them to be used to share state among a fixed number of parties, while keeping it private from all others.

1.2.5 Merkle trie structure

At a high level, a Merkle trie is a key-value store data structure which is able to be shared piece-wise in a verifiable way (via a construction called a Merkle proof). Each node is labelled by the hash of its data; for leaf nodes —that is the data stored in that part of the tree, for other node types — that is the data which references other nodes in the trie. Our implementation of the trie has radix of 256, this means each branch node can have up to 256 children. This is convenient because it means a path through the tree can be described as an array of bytes, and thus serialization directly links a key with a path through the tree to its associated value.

Formally, a trie node is one of the following:

- a leaf, which includes a key and a value
- a branch, which has up to 256 `blake2b256` hashes, pointing to up to 256 other nodes in the trie (recall each node is labelled by its hash)
- an extension node, which includes a byte array (called the affix) and a `blake2b256` hash pointing to another node in the trie

The purpose of the extension node is to allow path compression. For example, if all keys for values in the trie used the same first four bytes, then it would be inefficient to need to traverse through four branch nodes where there is only one choice, and instead the root node of the trie could be an extension node with affix equal to those first four bytes and pointer to the first non-trivial branch node.

The rust implementation of our trie can be found on GitHub:

- definition of the trie data structure
- reading from the trie
- writing to the trie

Note: that conceptually, each block has its own trie because the state changes based on the deploys it contains. For this reason, our implementation has a notion of a `TrieStore` which allows us to look up the root node for each trie.

1.3 Execution Semantics

1.3.1 Introduction

The CasperLabs system is a decentralized computation platform. In this chapter we describe aspects of the computational model we use.

1.3.2 Measuring computational work

Computation is all done in a WebAssembly (wasm) interpreter, allowing any programming language which compiles to wasm to become a smart contract language for the CasperLabs blockchain. Similar to Ethereum, we use `Gas` to measure computational work in a way which is consistent from node to node in the CasperLabs network. Each wasm instruction is assigned a `Gas` value, and the amount of gas spent is tracked by the runtime with each instruction executed by the interpreter. All executions are finite because each has a finite *gas limit* that specifies the maximum

amount of gas that can be spent before the computation is terminated by the runtime. How this limit is determined is discussed in more detail below.

Although computation is measured in `Gas`, we still take payment for computation in *motes*. Therefore, there is a conversion rate between `Gas` and motes. How this conversion rate is determined is discussed elsewhere.

1.3.3 Deploys

A *deploy* represents a request from a user to perform computation on our platform. It has the following information:

- Body: containing payment code and session code (more details on these below)
- Header: containing
  - the *identity key* of the account the deploy will run in
  - the timestamp when the deploy was created
  - a time to live, after which the deploy is expired and cannot be included in a block
  - the `blake2b256` hash of the body
- Deploy hash: the `blake2b256` hash of the Header
- Approvals: the set of signatures which have signed the deploy hash, these are xsused in the *account permissions model*

Each deploy is an atomic piece of computation in the sense that, whatever effects a deploy would have on the global state must be entirely included in a block or the entire deploy must not be included in a block.

Phases of deploy execution

A deploy is executed in distinct *phases* in order to accommodate paying for computation in a flexible way. The phases of a deploy are payment, session, and finalization. During the payment phase, the payment code is executed. If it is successful, then the sessions code is executed during the session phase. And, finally (independent of whether session code was executed), the finalization phase is executed, which does some bookkeeping around payment.

In particular, the finalization phase refunds the user any unspent `Gas` originally purchased (after converting back to motes), and moves the remaining payment into the rewards pool for the validators. The finalization phase does not include any user-defined logic, it is merely upkeep for the system.

Payment code

Payment code provides the logic used to pay for the computation the deploy will do. Payment code is allowed to include arbitrary logic, providing maximal flexibility in how a deploy can be paid for (e.g., the simplest payment code could use the account's *main purse*, while an enterprise application may require deploys to pay via a multi-sig application accessing a corporate purse). We restrict the gas limit of the payment code execution, based on the current conversion rate between gas and motes, such that no more than `MAX_PAYMENT_COST` motes (a constant of the system) are spent. To ensure payment code will pay for its own computation, we only allow accounts with a balance in their main purse greater than or equal to `MAX_PAYMENT_COST`, to execute deploys.

Payment code ultimately provides its payment by performing a *token transfer* into the proof-of-stake contract's payment purse. If payment is not given or not enough is transferred, then payment execution is not considered successful. In this case the effects of the payment code on the global state are reverted and the cost of the computation is covered by motes taken from the offending account's main purse.

Session code

Session code provides the main logic for the deploy. It is only executed if the payment code is successful. The gas limit for this computation is determined based on the amount of payment given (after subtracting the cost of the payment code itself).

Specifying payment code and session code

The user-defined logic of a deploy can be specified in a number of ways:

- a wasm module in binary format representing a valid *contract* (Note: the named keys do not need to be specified because they come from the account the deploy is running in)
- a 32-byte identifier representing the *hash* or *URef* where a contract is already stored in the global state
- a name corresponding to a named key in the account, where a contract is stored under the key

Each of payment and session code are independently specified, so different methods of specifying them may be used (e.g. payment could be specified by a hash key, while session is explicitly provided as a wasm module).

1.3.4 Deploys as functions on the global state

To enable concurrent modification of *global state* (either by parallel deploys in the same block or parallel blocks on different forks of the chain), we view each deploy as a function taking our global state as input and producing a new global state as output. It is safe to execute two such functions concurrently if they do not interfere with each other, which formally can be defined to mean the functions *commute* (i.e., if they were executed sequentially, it does not matter in what order they are executed, the final result is the same for a given input). Whether two deploys commute is determined based on the effects they have on the global state, i.e. which operation (read, write, add) it does on each key in the key-value store. How this is done is described in *Appendix C*.

1.3.5 The CasperLabs runtime

A wasm module is not natively able to create any effects outside of reading / writing from its own linear memory. To enable other effects (e.g. reading / writing to the CasperLabs global state), wasm modules must import functions from the host environment they are running in. In the case of contracts on the CasperLabs blockchain, this host is the CasperLabs Runtime. Here, we briefly describe the functionalities provided by imported function. All these features are conveniently accessible via functions in the CasperLabs rust library. For a more detailed description of the functions available for contracts to import, see *Appendix A*.

- Reading / writing from global state
  - `read`, `write`, `add` functions allow working with exiting *URefs*
  - `new_uref` allows creating a new `URef` initialized with a given value (see section below about how `URef`s are generated)
  - `read_local`, `write_local`, `add_local` allow working with local keys
  - `store_function` allows writing a contract under a *hash key*
  - `get_uref`, `list_known_urefs`, `add_uref`, `remove_uref` allow working with the *named keys* of the current context (account or contract)
- Account functionality
  - `add_associated_key`, `remove_associated_key`, `update_associated_key`, `set_action_threshold` support the various *key management actions*

  - `main_purse` returns the *main purse* of the account

- Runtime flow and properties
  - `call_contract` allows executing a contract stored under a key (hash or `URef`), including passing arguments and getting a return value
  - `ret` is used by contracts to return a value to their caller (i.e. enables return values from `call_contract`)
  - `get_named_arg` allows getting arguments passed to the contract (either to session code as part of the deploy, or arguments to `call_contract`)
  - `revert` exits the entire executing deploy, reverting any effects it caused, and returns a status code that is captured in the block
  - `get_caller` returns the public key of the account for the current deploy (can be used for control flow based on specific users of the blockchain)
  - `get_phase` returns the current *phase* of the deploy execution
  - `get_blocktime` gets the timestamp of the block this deploy will be included in
- *Mint* functionality
  - `create_purse` creates a new empty purse, returning the `URef` to the purse
  - `get_balance` reads the balance of a purse
  - `transfer_to_account` transfers from the present account's main purse to the main purse of a specified account (creating the account if it does not exist)
  - `transfer_from_purse_to_account` transfer from a specified purse to the main purse of a specified account (creating the account if it does not exist)
  - `transfer_from_purse_to_purse` alias for the *mint's transfer function*

Generating `URefs`

`URefs` are generated using a cryptographically secure random number generator using the ChaCha algorithm. The random number generator is seeded by taking the `blake2b256` hash of the deploy hash concatenated with an index representing the current phase of execution (to prevent collisions between `URefs` generated in different phases of the same deploy).

1.4 Accounts

1.4.1 Introduction

An *account* is a cryptographically secured gateway into the CasperLabs system used by entities outside the blockchain (e.g., off-chain components of blockchain-based applications, individual users). All user activity on the CasperLabs blockchain (i.e., "deploys") must originate from an account. Each account has its own context where it can locally store information (e.g., references to useful contracts, metrics, aggregated data from other parts of the blockchain). Each account also has a "main purse" where it can hold CasperLabs tokens (see *Tokens* for more information).

In this chapter we describe the permission model for accounts, their local storage capabilities, and briefly mention some runtime functions for interacting with accounts.

1.4.2 The `Account` data structure

An `Account` contains the following data:

- A collection of named keys (this plays the same role as the named keys in a *stored contract*)
- A `URef` representing the account's "main purse"
- A collections of "associated keys" (see below for more information)
- "Action thresholds" (see below for more information)

1.4.3 Permissions model

Actions and thresholds

There are two types of actions an account can perform: a deployment, and key management. A deployment is simply executing some code on the blockchain, while key management involves changing the associated keys (which will be described in more detail later). Key management cannot be performed independently, as all effects to the blockchain must come via a deploy; therefore, a key management action implies that a deployment action is also taking place. The `ActionThresholds` contained in the `Account` data structure set a `Weight` which must be met in order to perform that action. How these weight thresholds can be met is described in the next section. Since a key management action requires a deploy action, the key management threshold should always be greater than or equal to the deploy threshold.

Associated keys and weights

As mentioned in the introduction, accounts are secured via cryptography. The *associated keys* of an account are the set of public keys which are allowed to provide signatures on deploys for that account. Each associated key has a weight; these weights are used to meet the action thresholds provided in the previous section. Each deploy must be signed by one or more keys associated with the account that deploy is for, and the sum of the weights of those keys must be greater than or equal to the deployment threshold weight for that account. We call the keys that have signed a deploy the "authorizing keys". Similarly, if that deploy contains any key management actions (detailed below), then the sum of the weights of the authorizing keys must be greater than or equal to the key management action threshold of the account.

Note: that any key may be used to help authorize any action; there are no "special keys", all keys contribute their weight in exactly the same way.

Key management actions

A *key management action* is any change to any of the permissions parameters for the account. This includes the following:

- adding or removing an associated key;
- changing the weight of an associated key;
- changing the threshold of any action.

Key management actions have validity rules which prevent a user from locking themselves out of their account. For example, it is not allowed to set a threshold larger than the sum of the weights of all associated keys.

Account security and recovery using key management

The purpose of this permissions model is to enable keeping accounts safe from lost or stolen keys, while allowing usage of capabilities of modern mobile devices. For example, it may be convenient to sign deploys from a smart phone in day-to-day usage, and this can be done without worrying about the repercussions of losing the phone. The recommended setup would be to have a low-weight key on the phone, only just enough for the deploy threshold, but not enough for key management, then if the phone is lost or stolen, a key management action using other associated keys from another device (e.g., a home computer) can be used to remove the lost associated key and add a key which resides on a replacement phone.

Note: that it is extremely important to ensure there will always be access to a sufficient number of keys to perform the key management action, otherwise future recovery will be impossible (we currently do not support "inactive recovery").

1.4.4 Creating an account

Account creation happens automatically when there is a *token transfer* to a yet unused *identity key*. When an account is first created, the balance of its main purse is equal to the number of tokens transferred, its action thresholds are equal to 1, and there is one associated key (equal to the public key used to derive the identity key) with weight 1.

1.4.5 The account context

A deploy is a user request to perform some execution on the blockchain (see *Execution Semantics* for more information). It contains "payment code" and "session code" which are contracts that contain the logic to be executed. These contracts are executed in the context of the account of the deploy. This means these contracts use the named keys of the account and use the account's local storage (i.e., the "root" for the local keys come from the account identity key).

Note: that other contracts called from the session code by `call_contract` are executed in their own context, not the account context. This means, an account (with logic contained in session code) can be used to locally store information relevant to the user owning the account.

1.4.6 Functions for interacting with an account

The CasperLabs rust library contains several functions for working with the various account features:

- `add_associated_key`
- `remove_associated_key`
- `update_associated_key`
- `set_action_threshold`
- `main_purse`
- `list_named_keys`
- `get_named_key`
- `add_named_key`

1.5 Block Structure

1.5.1 Introduction

A *block* is the primary data structure by which information about the state of the CasperLabs system is communicated between nodes of the network. We briefly describe here the format of this data structure.

1.5.2 Protobuf definition

Messages between nodes are communicated using Google's protocol buffers. The complete definition of a block in this format can be found on GitHub ; the description here is only meant to provide an overview of the block format; the protobuf definition is authoritative.

1.5.3 Data fields

A block consists of the following:

- a `block_hash`
- a header
- a body
- a signature

Each of these are detailed in the subsequent sections.

`block_hash`

The `block_hash` is the `blake2b256` hash of the header (serialized according to the protobuf specification).

Header

The block header contains the following fields:

- `parent_hashes`
  - a list of `block_hash`s giving the parents of the block
- justifications
  - a list of `block_hash`s givin the justifications of the block (see consensus description in part A for more details)
- a summary of the global state, including
  - the *root hash of the global state trie* prior to executing the deploys in this block (`pre_state_hash`)
  - the root hash of the global state trie after executing the deploys in this block (`post_state_hash`)
  - the list of currently bonded validators, and their stakes
- the `blake2b256` hash of the body of the block (serialized according to the protobuf specification)
- the time the block was created
- the protocol version the block was executed with

- the number of deploys in the block
- the human-readable name corresponding to this instance of the CasperLabs system (`chain_id`)
- the public key of the validator who created this block
- an indicator for whether this message is intended as a true block, or merely a *ballot* (see consensus description in part A for more details)

Body

The block body contains a list of the deploys processed as part of this block. A processed deploy contains the following information:

- a copy of the deploy message which was executed (see *Execution Semantics* for more information about deploys and how they are executed)
- the *amount of gas spent* during its execution
- a flag indicating whether the deploy encountered an error
- a string for an error message (if applicable)

Signature

The block signature cryptographically proves the block was created by the validator who's public key is contained in the header. The signature is created using a specified algorithm (currently only Ed25519 is supported), and is signed over the `block_hash` so that it is unique to that block.

1.6 Tokens

1.6.1 Introduction

CasperLabs is a decentralized computation platform based on a proof-of-stake consensus algorithm. Having a unit of value is required to make this system work because users must pay for computation and validators must have stake to bond. It is traditional in the blockchain space to name this unit of value "token" and we continue that tradition here.

This chapter describes how we define tokens and how they are used in our platform.

1.6.2 Token Generation and Distribution

A blockchain system will need to have a supply of tokens available for the purposes of paying for computation and rewarding validators for the processing of transactions on the network. A great deal of effort has been taken to ensure that no single individual or entity acquires 1%+ of the tokens from the onset. The tokens will be allocated in the following fashion (subject to applicable laws and regulations):

- 60% of total token supply will be sold to the public via validator sales and ongoing public sales.
- 16% reserved for developer & entrepreneur incentives, advisors and community managers via a DAO structure coupled with an AWS credits style system deployed over 5 years.
- 10% will be held by the CasperLabs network; any and all distributions will be commensurate with public release.
- 8% reserved for the team, vesting on a schedule similar to equity incentive programs over 3 years.

- 6% for advisors and strategic partners, distributed in a manner similar to the developer and community incentives.

In addition to the initial supply, the system will have a low rate of inflation, the results of which will be paid out to validators in the form of seigniorage, as described previously in this document.

The number of tokens used as a basis for calculating seigniorage and the above stated allocations is 10 billion.

1.6.3 Divisibility of tokens

Typically, a "token" is divisible into some number of parts. For example, Ether is divisible into 10^{18} parts (called Wei). To avoid rounding error, it is important to always represent token balances internally in terms of the number of indivisible parts. For the purposes of this document we will always work in terms of these indivisible units so as to not need to pick a particular level of divisibility for our "token" (as this detail is not important for the present description). We call the indivisible units which make up our token *motes*.

1.6.4 Mints and purses

A *mint* is a contract which has the ability to produce new motes of a particular type. We allow for multiple mote types (each of which would have their own mint) because we anticipate the existence of a large ecosystem of different tokens, similar to ERC20 tokens on Ethereum. CasperLabs will deploy a specific mint contract and it will manage the CasperLabs utility token (used for paying for computation and bonding onto the network). The mint also maintains all the balances for its type of mote. Each balance is associated with a `URef`, which acts as a sort of key to instruct the mint to perform actions on that balance (e.g., transfer motes). Informally, we will refer to these balances as *purses* and conceptually they represent a container for motes. The `URef` is how a purse is referenced from outside the mint.

The `AccessRights` of the *URefs* permissions model determine what actions are allowed to be performed when using a `URef` associated with a purse.

As all `URefs` are unforgable, the only way to interact with a purse is for a `URef` with appropriate `AccessRights` to be given to the current context in a valid way (see `URef` permissions for details).

The basic global state options map onto more standard monetary operations according to the table below:

Global State Action	Monetary Action
Add	Deposit (i.e. transfer to)
Write	Withdraw (i.e. transfer from)
Read	Balance check

We will use these definitions throughout this chapter as we describe the implementation and usage of tokens on the CasperLabs system.

1.6.5 The mint contract interface

A valid mint contract exposes the following methods (recall that many mint implementations may exist, each corresponding to a different "currency").

- `transfer(source: URef, target: URef, amount: Motes) -> TransferResult`
  - `source` must have at least `Write` access rights, `target` must have at least `Add` access rights
  - `TransferResult` may be a success acknowledgement or an error in the case of invalid `source` or `target` or insufficient balance in the `source` purse
- `mint(amount: Motes) -> MintResult`

  - `MintResult` either gives the created `URef` (with full access rights), which now has balance equal to the given `amount`; or an error due to the minting of new motes not being allowed
  - In the CasperLabs mint only the system account can call `mint`, and it has no private key to produce valid cryptographic signatures, which means only the software itself can execute contracts in the context of the system account

- `create() -> URef`
  - a convenience function for `mint(0)` which cannot fail because it is always allowed to create an empty purse
- `balance(purse: URef) -> Option<Motes>`
  - `purse` must have at least `Read` access rights
  - `BalanceResult` either returns the number of motes held by the `purse`, or nothing if the `URef` is not valid

1.6.6 Using purse `URefs`

It is dangerous to pass a purse's `URef` with `Write` permissions to any contract. A malicious contract may use that access to take more tokens than was intended or share that `URef` with another contract which was not meant to have that access. Therefore, if a contract requires a purse with `Write` permissions, it is recommended to always use a "payment purse", which is a purse used for that single transaction and nothing else. This ensures even if that `URef` becomes compromised it does not contain any more funds than the user intended on giving.

```
let main_purse = contract_api::main_purse();
let payment_purse = contract_api::create_purse();

match contract_api::transfer_purse_to_purse(main_purse, payment_purse, payment_
↪amount) {
    TransferResult::Success => contract_api::call_contract(contract_to_pay, payment_
↪purse),
    _ => contract_api::revert(1),
}
```

To avoid this inconvenience, it is a better practice for application developers intending to accept payment on-chain to make a version of their own purse `URef` with `Read` access rights publicly available. This allows clients to pay via a transfer using their own purse, without either party exposing `Write` access to any purse.

1.6.7 Purses and accounts

Every *Accounts* on the CasperLabs system has a purse associated with the CasperLabs system mint, which we call the account's "main purse". However, for security reasons, the `URef` of the main purse is only available to code running in the context of that account (i.e. only in payment or session code). Therefore, the mint's `transfer` method which accepts `URefs` is not the most convenient to use when transferring between account main purses. For this reason, CasperLabs supplies a transfer_to_account function which takes the public key used to derive the *identity key* of the account. This function uses the mint transfer function with the current account's main purse as the `source` and the main purse of the account at the provided key as the `target`. The transfer_from_purse_to_account function is similar, but uses a given purse as the `source` instead of the present account's main purse.

1.7 Appendix

1.7.1 A - List of possible function imports

The following functions can be imported into a wasm module which is meant to be used as a contract on the CasperLabs system. These functions give a contract access to features specific to the CasperLabs platform that are not supported by general wasm (e.g. accessing the global state, creating new `URefs`). Note that these are defined and automatically imported if the CasperLabs rust library is used to develop the contract; these functions should only be used directly by those writing libraries to support developing contracts for CasperLabs in other programming languages.

For an up to date description of exported functions please visit casperlabs_contract crate docs.

1.7.2 B - Serialization format

The CasperLabs serialization format is used to pass data between wasm and the CasperLabs host runtime. It is also used to persist global state data in the Merkle trie. The definition of this format is described in the *Global State* section.

A Rust reference implementation for those implementing this spec in another language can be found here:

- bytesrepr.rs
- cl_value.rs
- account.rs
- contract.rs
- uint.rs

Additionally, examples of all data types and their serializations are found in our main GitHub repository. These examples are meant to form a standard set of tests for any implementation of the serialization format. An implementation of these example as tests is found in our Scala code base.

1.7.3 C - Parallel execution as commuting functions

Introduction

The state of the CasperLabs system is represented by the *global state*. The evolution of this state is captured by the blockchain itself, and eventually agreed upon by all nodes in the network via the consensus mechanism. In this section we are concerned with only a single step of that evolution. We think of such a step as performing some "computation" that changes the global state. A *deploy* is a user request for computation, and contains two atomic units of computation: the payment code and the session code (the details of which are discussed elsewhere). For the purpose of this section, we think of each of these units as a (mathematical) function which takes the current global state as input, perhaps along with some other arguments, and produces a new global state as output. However, since the overall global state is ambient from the perspective of the session/payment code itself, the global state is not an explicit parameter in any user's source code, nor is there any explicit return value.

In this section we refine this idea of computation modeled as functions, and describe how it is used to enable parallel execution.

Computation as functions on the global state

As discussed in the introduction, we think of computation on the CasperLabs platform as being functions from the global state, G, to itself. Naturally, we can compose two such functions, to obtain another function. This corresponds to sequential execution. For example, you can think of the sequence `payment_code -> session_code` as

being the composition of two individual functions, capturing the effects of the payment and session codes, respectively. If there are smart contracts which are called during those execution phases, you could even break these down further into a sequence of those calls: `deployed_payment_wasm -> contract_a -> contract_b -> stored_session_code -> contract_c -> ...`. For notational purposes, we will call the set of functions $\{f \mid f : G \to G\} = End(G)$, meaning "endomorphisms of G."

While this simple model captures sequential execution, it does not model parallel execution. Parallel execution is important because it can enable the execution engine to run more than one deploy at the same time, possibly improving block processing times. Note: each deploy itself is still single-threaded; we will not support parallel execution within a single contract or deploy. This optimization is purely for the performance of the node implementation, not contract developers.

Computation as functions from G to $End(G)$

The problem with functions on the global state itself is they mutate the state, potentially causing problems if we wanted to apply two such functions at the same time. Therefore, we will instead think of computations as outputting a description of the changes to the global state that they would make if given the chance. Or phrased another way, the execution of a deploy will return a function that could be applied to the global state to obtain the post-state we would have obtained from running the computation while mutating the global state. The reason this helps is because we can apply multiple such functions to the same global state at the same time; they are pure functions that do not modify the global state. Thus we can execute multiple deploys in parallel and later combine their outputs (more on this later).

The way this is modeled in the rust reference implementation is via the `TrackingCopy`. Executing deploys (and the contracts they call) read/write from the `TrackingCopy` instead of the global state directly; the `TrackingCopy` *tracks* the operations and returns the `Transforms` which act on each key in the global state effected by the execution. Using the nomenclature from the theory, this collection of keys and transforms describes a function $f : G \to G$ which is an endomorphism on G, i.e. an element of $End(G)$.

An important note about the returned `Transforms` is there is exactly one `Transform` per key that was used during the execution. Initially, this may be unintuitive because a contract can use the same key multiple times, however, because each deploy executes sequentially, we can use the composition property discussed in the previous section to combine multiple sequential operations into a single operation. Consider the following example.

```
// an implementation of the function featured in the Collatz conjecture
let n = read_local("n");
let f_n =
    if n % 2 == 0 { n / 2 }
    else { 3 * n + 1 };
write_local("n", f_n);
```

The above function reads a local variable, performs a computation which depends on the current value of that variable, then writes an updated value. Suppose we execute this function on a global state where the value of the local key is `7`. Then the sequence of transforms on the global state would be `Read -> Write(22)` since `n` would be odd and thus `f_n` would be computed using the `else` case. From the perspective of state changes, we only need to keep the `Write(22)` transform because final state is the same as if we had also included the `Read` transform. In fact, by the same reasoning, we know that we only need to keep the last `Write`, whatever it happens to be, since it will be the final value on the key after the computation finishes. Notice that the resulting global state function does not exactly reproduce the original contract execution steps; it is a *reduced trace* where only the final effect on the global state is recorded[1]. In particular, this means applying the results of these executions is very fast relative to the original

[1] There is a special case of constructing reduced traces which is worth calling out explicitly. Suppose the initial value of a key in the global state is `X`, and after performing the execution, the transform for that key is `Write(X)`. Then it is valid to replace that transform with `Read`. This is because the computation acts like the identity function (i.e. the function which makes no changes) at this key, and therefore is equal to `Read`. Notably we cannot simply remove the transfrom from the map because the key was still used in some way during the computation. We must have a record of what keys were used to correctly detect when deploys commute (see the following sections for more details). Replacing a `Write` with a `Read` still has great benefits for parallel exectuion because reads do commute with one another, while writes do not. This optimization in the reduced traces is applied in our reference implementation.

execution (this will be importnat for how we use these traces in the next section). Also notice that the transforms which are produced depend on the initial state. This might be obvious since we are modeling compuation as functions $f : G \to End(G)$, so this statement is simply that the function really depends on its input. However, this is again an imporant concept to keep in mind when working with this model of computation. Going back to our example, if the value of the local key was `16` then the transform produced would be `Write(8)`, entirely different from the case where the initial value was `7`.

Constructing the post-state from parallel execution

Following from the previous section, we know that deploys execute to produce a `Map<Key, Transform>` which gives a summary (i.e. "reduced trace") of the effects the deploy would have had on each key in the global state (keys not present in the map are not effected). In the reference implementation we call this the `exec` phase. Since creating these maps does not mutate the global state, we can run as many of these as we want in parallel. However, after they have been run we need to actually produce a post-state, the new global state after applying the effects of the deploys (this will then be used as the pre-states for deploys in the following batch of executions). In the reference implementation, we call applying the collection of transforms to obtain a post-state the `commit` phase.

Before we can construct the post-state, we must know that one is well-defined. When working with parallel execution with a shared resource, you may encounter "race conditions". This is a situation where the outcome of a parallel computation depends on the order or timing of events, in particular when this timing is not explicitly controlled. Or phrased another way, parallelism with a shared resource is a lie and one of the processes will use the resource first, followed by the other one. A classic blockchain example of a race condition is a double spend (which under an accounts model, as opposed to UTXO, is the same as an overdraft on the account); one payer attempts to pay two payees at the same time without enough tokens to actually pay both. One payee or the other is not getting their tokens, depending on the order the transactions are processed.

In our simple model of computation where deploys are functions on the global state, this would correspond to functions that do not *commute*, that is to say, the order in which we apply the functions to the global state matters: $f \circ g \neq g \circ f$. Therefore, in order to prevent race conditions, we will only allow deploys to execute in parallel if they commute. Taking our more sophisticated model of computation, we have two deploys: $f : G \to End(G)$ and $g : G \to End(G)$, and we will only allow both be committed to the same pre-state G if $f(G) \circ g(G) = g(G) \circ f(G)$, i.e. the resulting maps of transforms commute.

We will discuss how to compute whether two maps of transforms commute in the next section. For now, we assume that run some set of deploys $d_1, d_2, d_3, \ldots$ in parallel against a fixed pre-state G to obtain a set of transform maps $T_1, T_2, T_3, \ldots$, then select only the transforms that commute $T_i, T_j, T_k, \ldots$ to apply to G, and thus obtain the post-state G'. The remaining deploys we can all run in parallel against G', again choosing the commuting ones to commit, obtaining G'', and so on. This final post-state is the same as if we had run all the deploys $d_1, d_2, d_3, \ldots$ in sequence against G, but perhaps faster (depending on how many could commute[2]) because we were able to run in parallel batches.

Detecting when maps of transforms commute

Two transform maps `m_1: Map<Key, Transform>` and `m_2: Map<Key,Transform>` commute if for all keys `k` which are present in both maps, the transforms `t_1 = m_1[k]` and `t_2 = m_2[k]` commute. Notably, if there are no such keys then the maps trivially commute. Two transforms `t_1:Transform` and `t_2: Transform` commute if:

- `t_1 == t_2 == Read`
- `t_1` and `t_2` are both of the same `Add*` transform variant (note they do not need to contain the same values within that variant)

[2] Recall that committing transforms is a very fast operation relative to execution, so it causes little overhead. The main overhead would come from executing the same deploy against multiple different starting states because it failed to commute multiple times. This can be mitigated by favoring including more expensive deploys in each committed batch.

where `Add*` is a placeholder representing any of the typed native add operations (`AddInt32`, `AddInt64`, `AddInt128`, `AddInt256`, `AddInt512`, `AddKeys`). And they do not commute otherwise. A short montra for this is: reads commute, adds commute, writes conflict. Note that writes *always* conflict, even if they are writing the same value. Consider the following example:

```
fn f() {
    let x = read_local("x");

    if x == 7 { write_local("x", 10); }
    else { write_local("x", 0); }
}

fn g() {
    let x = read_local("x");

    if x == 7 { write_local("x", 10); }
    else { write_local("x", 100); }
}
```

If the pre-state G has `local("x") == 7` then `f(G)` results in the transform `Write(10)`, and so does `g(G)`. However, if we compose `g(f(G))` then we obtain `Write(100)`, and if we compose `f(g(G))` then the result is `Write(0)` and hence the functions do not commute.

Handling Errors

The reason we can say "adds commute" in our rules is because mathematically addition is commutative. However, this relies on the infinite nature of the number line and real computers are finite. For example, if we considered the addition of three 8-bit numbers: 250, 3, and 5, any two of them can be added and they commute, but attempting to add all three results in an overflow error. Thus the final result depends on the order of addition:

- 250 + 3 + 5 = 253 (last addition does not happen due to the error)
- 250 + 5 + 3 = 255
- 3 + 5 + 250 = 8

Presently we circumvent this error by actually using modular arithmetic (wrapped addition as it is often called in computer science). Addition in modular arithmetic is still a commutative operation, so our theory holds together. In our example above 250 + 5 + 3 is always equal to 3, no matter what. However in the context of financial applications wrapping back to zero is an unexpected behavior. For this reason we use 512-bit numbers in our mint contract to represent balances, and the total number of token units (motes) available is less than `U512::MAX`, so overflow is impossible.

That said, this is not the only error which may arise due to the finite nature of computers. For example, the `AddKeys` transform is about adding elements to a map, which is a commutative operation as well (so long as none of the keys already existed in the map, then it is more akin to a write operation). Yet, this operation can also fail due to the physical machine being out of memory, thus once again meaning the order of additions could effect the final state of the map.

In a more powerful theory of parallel execution we could consider operations which fail. In this case we could say that transforms `t_1` and `t_2` commute if they are of the same addition type and the outcome of applying both to the input global state, G is not an error. This is a more complex rule because it requires doing some amount of computation during commutativity checking, whereas the previous theory was simple comparison. Yet, this theory might be worth pursuing because it solves the two problems we have listed here (overflow and out-of-memory), along with other problems that we presently cannot handle at all. For example, `Minus` could be introduced as a transform, and underflows could be handled using this refined commutativity rule. This has practical application in our system because it would mean transfers from the same source could commute if enough funds are available, whereas presently they will always be conservatively labeled as not commuting.

CHAPTER 2

Economics

The CasperLabs blockchain implements the first Proof-of-Stake network based on the CBC Casper family of protocols. The network is permissionless, such that anybody is allowed to become a validator, once they fulfill certain requirements. The blockchain hosts its own native token, designated by the symbol CLX, which serves many functions for the network's participants. It is used as deposit when becoming a validator. Users use CLX to fund their transactions, and validators receive rewards in CLX. New CLX is minted to incentivize validators, whereas bonded CLX is burned to punish faults.

Staking allows validators to receive rewards and share transaction fees. Validators are required to participate in the network by processing transactions; and proposing, validating, and storing blocks. The network's continuation and maintenance requires that validators adhere to the protocol, which is ensured by the network's incentive mechanism. This works by rewarding adherence to the protocol and punishing deviation from the protocol.

In order to stake in the network, prospective validators participate in bonding auctions for a limited number of validator positions, with winning bids becoming bonded as stakes. This approach strikes a balance between security and performance, since increasing the number of validators must weakly decrease network throughput with the present mainnet architecture, due to increased communication complexity.

Rewards are provided through a process called *seigniorage*. New tokens are minted at a constant rate and distributed to participating validators, similar to the block reward mechanism in Bitcoin. Unlike Bitcoin, validators don't have to wait until they mine a block in order to realize their rewards. In each block, seigniorage is paid to all participating validators. This ensures stable, continuous payments for honest validators, and eliminates the need to create pools.

A malicious validator, on the other hand, is punished through disincentives for various types of undesirable activities such as attacks on consensus, censorship and freeloading. The protocol is designed to penalize validators that engage in such activities by burning a part of their stake, referred to as slashing.

In addition to seigniorage payments, validators receive transaction fees paid by the users. Similar to Ethereum, computation is metered by gas, where each operation is assigned a cost in gas. However, unlike Ethereum, gas price is fixed in fiat terms, meaning the expenses of using or hosting a dapp is stable and predictable. Users do not have to choose a gas price when submitting their transactions, which greatly simplifies the user experience. Once a transaction is executed, used gas is calculated with the fixed gas price to calculate the transaction fee. Unlike seigniorage, transaction fees are collected by the block proposer, instead of being distributed to all participating validators.

2.1 Transaction Fees

When a user submits a transaction to perform some computation, the transaction uses up resources provided by network. To make cost accounting simpler, computational resources are denominated in a common unit of account called gas, as pioneered by Ethereum.

Once a transaction finishes its execution, the total gas it ends up using is retrieved and multiplied by the gas price, to calculate the final transaction fee. When it is a validator's turn to propose a block, the validator collects transactions from the transaction pool, executes them in a certain order, and publishes them in a new block. Fees in a block are collected by the block's proposer.

2.1.1 Gas Pricing

It is one of the goals of CasperLabs to maintain a certain level of predictability for users in terms of gas prices, and for validators in terms of transaction fees. Blockchains with unregulated fee markets are susceptible to high volatility in transaction fees, which get pushed up as demand rises and blocks become full.

To this end, as an initial step, Casperlabs is implementing a transaction pricing system that assigns fiat (dollar) prices to all relevant resources, such as bytes of storage, opcodes and standardized computation times for external functions. A successful implementation of this system requires a reliable on-chain feed of the CLX price in USD. To this end, CasperLabs utilizes the Chainlink network of oracles to aggregate a single price from major exchanges.

Normally, there would be no way to prioritize high value transactions with such a fixed price model. To mitigate this issue, we set our gas price at a level high enough to curb any congestion the network might face on a consistent basis, e.g. daily.

2.1.2 Flexible Payments

A major feature of Highway is the ability to implement arbitrary payment logic with *payment codes*. This has a number of advantages:

- It enables dapp maintainers to pay for the transactions of their users without additional complexity, greatly improving the onboarding of new users.
- It allows the platform to support all kinds of payment schemes that future businesses might want to adapt.
- It makes it possible to have multiple parties pay for a transaction, or contracts that pay for their own transactions.

Payment codes render Highway future-proof, that is, its economic layer has enough room for every possibility that one can imagine.

2.2 Staking and Delegation

2.2.1 Bonding Auctions

Because the present blockchain model relies on universal validation of blocks included in the consensus history, there is a sharp tradeoff between intuitive notions of decentralization/security and performance. Choosing the correct balance requires setting proper incentives, or even hard rules, for validator entry, in a manner that ensures performance remains acceptable without exposing the platform to oligopolistic capture.

An open auction with open bids, a simple, first-"price" resolution and a fixed number of slots solves the issue with imprecise control over the security/performance tradeoff, while offering prospective validators an easy to understand procedure for joining. There is no minimum stake and the security/performance tradeoff is handled by an automatic on-chain slot number adjustment mechanism.

2.2.2 Delegation

Users are able to delegate their own CLX to validators without having to become validators themselves. Validators, in turn, take a commission out of the rewards and transaction fees earned through delegated CLX. Delegated tokens are subject to the same unbonding rules as the tokens staked by validator from their own accounts. Further, delegated tokens can be used in bidding for validator slots and can be lost due to equivocations.

2.3 Slashing

The practical utility of a blockchain platform depends on its *safety* and *liveness*. A safe blockchain is one where users can expect valid transactions to eventually become recorded in the canonical history, or a linear sequence of finalized blocks. A live blockchain is one where this process can continue indefinitely, as long as there are validators to process, disseminate, and record the transactions in blocks. Actions by validators that constitute a threat to either the safety or the liveness of the blockchain are termed *faults*.

We can enforce compliance with certain features of the protocol, such as the fields expected to be populated in a block's metadata, as part of the programmatic protocol definition, and reject all blocks failing to satisfy the conditions as invalid, or faulty. However, some faults cannot be defined as properties of individual blocks, or directly prohibitied by the protocol specification. Rather, they must be incentivized by imposing costs for commission of faults. Direct incentivization of individual validators by these means is only possible with *attributable* faults, or faults that can be traced to an individual validator. *Slashing* is the term we use for such incentivization. Specifically, a *slashed* validator loses some, or all, of the stake, possibly resulting in ejection.

Currently, we only penalize equivocations, though this may change as we develop the platform.

Equivocation faults constitute a direct threat to the safety of the system by making it difficult to settle on a single canonical history of transactions. This reduces value of the system for both users and participants, since the value proposition of a blockchain is precisely that it must eventually finalize a unique history. Equivocation faults are attributable to individual validators and are subject to slashing. Moreover, slashing is necessary because it is not feasible to programmatically forbid validators from equivocating, as equivocation is not a property of a single block or a message. Because equivocations constitute a particularly serious threat to the expected operation of the blockchain and threatens its value to all users and validators, equivocations require a slash value of 1, without a limit imposed by minimum bond.

CHAPTER 3

Dapp Developer Guide

This guide is designed to support dapp developers getting started with the development of smart contracts on the CasperLabs blockchain in AssemblyScript or Rust. For Rust, there is a contract development kit that includes a runtime environment, reference documentation, and test framework. You can install our environment locally, create and test Smart Contracts with our Smart Contracts and Test Libraries, and use these libraries to build your own applications.

It is recommended you have prior knowledge about Unix-based operating systems, like GNU/Linux or macOS, and programming knowledge with

- Rust,
- AssemblyScript,
- JavaScript (optional),
- Python (optional).

The topics on the index include present and future documentation initiatives in our roadmap and are organized so that you will be able to:

- Understand what CasperLabs is building.
- Learn how to build and operate applications on the platform.
- Learn how to set up the CasperLabs environment locally.
- Work with our Contracts API to access our Rust resources.
- Learn how to create and test Smart Contracts with our libraries.

The motivation for our roadmap is inspired by feedback we are receiving from your recommendations. We hope you continue to provide your feedback as you embark on this journey with us – we look forward to building a decentralized future together.

Coding Standards and Review cover best practices and recommendations following our coding standards, code reviews and publishing your code to our GitHub.

3.1 Getting Help

CasperLabs makes available the following resources for you to connect and get support where you can:

- Connect live with members of our Engineering Team on our Discord Channel available to support you with the progress of your projects.
- Join the CasperLabs Community Forum that includes Technical discussions on using the CasperLabs features, obtain support, and pose questions to the CasperLabs core development team.
- Subscribe to CasperLabs Official Telegram channel for general information and update notifications about our platform.

If you have issues and bugs related to CasperLabs maintained smart contracts, you can file an issue on our main GitHub repository, or whichever repository the issue is related to.

Otherwise, if you...

- have questions that are not issues,
- need technical support,
- want to give feedback, or suggestions for improvement,
- or participate in a bounty,

you can use our Jira Service Desk.

3.1.1 Setting Up the Rust Contract SDK

The SDK is the easiest way to get started with Smart Contract development. This guide will walk you through the steps to get set up.

Prerequisites

Install Rust

Install Rust using `curl`

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

For more details follow the official Rust guide.

Install Google protobuf compiler

Linux with `apt`

```
sudo apt install protobuf-compiler
```

macOS with `brew`

```
brew install protobuf
```

For more details follow the official downloads page.

CasperLabs SDK

Available Packages

There are three crates we publish to support Rust development of Smart Contracts. These can be found on crates.io, and are:

- CasperLabs Contract - library that supports communication with the blockchain. That's the main library to use when writing smart contracts.
- CasperLabs Test Support - in-memory virtual machine you can test your smart contracts against.
- CasperLabs Types - library with types we use across the Rust ecosystem.

Technical Documentation for the Contract API

Each of the crates ships with API documentation and examples for each of the functions. Docs are located at https://docs.rs. For example, the contract API documentation for version 0.5.1 is located at: https://docs.rs/casperlabs-contract/0.5.1/casperlabs_contract/contract_api/index.html

Cargo CasperLabs

The best way to set up a CasperLabs Rust Smart Contract project is to use `cargo-casperlabs`. When you use this, it will set the project up with a simple contract and a runtime environment/testing framework with a simple test. It's possible to use this configuration in your CI/CD pipeline as well. Instructions are also available on GitHub.

```
cargo install cargo-casperlabs
```

Develop Smart Contracts

In this tutorial we will use `cargo-casperlabs`. The CasperLabs blockchain uses WebAssembly (Wasm) in its runtime environment. Compilation targets for Wasm are available for Rust, giving developers access to all the tools in the Rust ecosystem when developing smart contracts.

Create a new project

```
cargo casperlabs my-project
```

This step will create two crates called `contract` and `tests` inside a new folder called `my-project`. This is a complete basic smart contract that saves a value, passed as an argument, on the blockchain. The tests crate provides a runtime environment of the CasperLabs virtual machine, and a basic test of the smart contract.

Configure Rust for Building the Contract and Test

The project requires a specific nightly version of Rust, and requires a Wasm target to be added to that Rust version. The steps to follow are shown by running

```
cargo casperlabs --help
```

These steps can be performed by running

```
cd my-project/contract
rustup install $(cat rust-toolchain)
rustup target add --toolchain $(cat rust-toolchain) wasm32-unknown-unknown
```

Build the Contract

The next step is to compile the smart contract into Wasm.

```
cd my-project/contract
cargo build --release
```

The `build` command produces a smart contract at `my-project/contract/target/wasm32-unknown-unknown/release/contract.wasm`.

NOTE: It's important to build the contract using `--release` as a debug build will produce a contract which is much larger and more expensive to execute.

Test the Contract

The test crate will build the contract and test it in a CasperLabs runtime environment. A successful test run indicates that the smart contract environment is set up correctly.

```
cd ../tests
cargo test
```

The `tests` crate has a `build.rs` file: effectively a custom build script. It's executed every time before running tests and it compiles the smart contract in release mode for your convenience. In practice, that means we only need to run `cargo test` in the `tests` crate during the development. Go ahead and modify `contract/src/lib.rs`. You can change the value of `KEY` and observe how the smart contract is recompiled and the test fails.

3.1.2 Writing Rust Contracts on CasperLabs

Smart Contracts

This section explains step by step how to write a new smart contract.

Basic Smart Contract

The CasperLabs VM executes a smart contract by calling the `call` function specified in the contract. If the function is missing, the smart contract is not valid. The simplest possible example is an empty `call` function.

```
#[no_mangle]
pub extern "C" fn call() {}
```

The `#[no_mangle]` attribute prevents the compiler from changing (mangling) the function name when converting to the binary format of Wasm. Without it, the VM exits with the error message: `Module doesn't have export call`.

Using Error Codes

The CasperLabs VM supports error codes in smart contracts. A contract can stop execution and exit with a given error via the `runtime::revert` function:

```
use casperlabs_contract::contract_api::runtime;
use casperlabs_types::ApiError;

#[no_mangle]
pub extern "C" fn call() {
    runtime::revert(ApiError::PermissionDenied)
}
```

CasperLabs has several built-in error variants, but it's possible to create a custom set of error codes for your smart contract. These can be passed to `runtime::revert` via `ApiError::User(<your error code>)`.

When a contract exits with an error code, the exit code is visible in the Block Explorer.

Arguments

It's possible to pass arguments to smart contracts. To leverage this feature, use `runtime::get_named_arg`.

```
use casperlabs_contract::contract_api::runtime;

#[no_mangle]
pub extern "C" fn call() {
    let value: String = runtime::get_named_arg("value");
}
```

Storage

Saving and reading values to and from the blockchain is a manual process in CasperLabs. It requires more code to be written, but also provides a lot of flexibility. The storage system works similarly to a file system in an operating system. Let's say we have a string `"Hello CasperLabs"` that needs to be saved. To do this, use the text editor, create a new file, paste the string in and save it under a name in some directory. The pattern is similar on the CasperLabs blockchain. First you have to save your value to the memory using `storage::new_turef`. This returns a reference to the memory object that holds the `"Hello CasperLabs"` value. You could use this reference to update the value to something else. It's like a file. Secondly you have to save the reference under a human-readable string using `runtime::put_key`. It's like giving a name to the file. The following function implements this scenario:

```
const KEY: &str = "special_value";

fn store(value: String) {
    // Store `value` under a new unforgeable reference.
    let value_ref = storage::new_turef(value);

    // Wrap the unforgeable reference in a `Key`.
    let value_key: Key = value_ref.into();

    // Store this key under the name "special_value" in context-local storage.
    runtime::put_key(KEY, value_key);
}
```

After this function is executed, the context (Account or Smart Contract) will have the content of the `value` stored under `KEY` in its named keys space. The named keys space is a key-value storage that every context has. It's like a home directory.

Final Smart Contract

The code below is the simple contract generated by `cargo-casperlabs` (found in `contract/src/lib.rs` of a project created by the tool). It reads an argument and stores it in the memory under a key named `"special_value"`.

```
#![cfg_attr(
    not(target_arch = "wasm32"),
    crate_type = "target arch should be wasm32"
)]

use casperlabs_contract::{
    contract_api::{runtime, storage},
    unwrap_or_revert::UnwrapOrRevert,
};
use casperlabs_types::{ApiError, Key};

const KEY: &str = "special_value";

fn store(value: String) {
    // Store `value` under a new unforgeable reference.
    let value_ref = storage::new_turef(value);

    // Wrap the unforgeable reference in a value of type `Key`.
    let value_key: Key = value_ref.into();

    // Store this key under the name "special_value" in context-local storage.
    runtime::put_key(KEY, value_key);
}

// All session code must have a `call` entrypoint.
#[no_mangle]
pub extern "C" fn call() {
    // Get the first argument supplied to the argument.
    let value: String = runtime::get_named_arg("value")
    store(value);
}
```

Tests

As part of the CasperLabs local environment we provide the in-memory virtual machine you can run your contract against. The testing framework is designed to be used in the following way:

1. Initialize the context.
2. Deploy or call the smart contract.
3. Query the context for changes and assert the result data matches expected values.

TestContext

A `TestContext` provides a virtual machine instance. It should be a mutable object as we will change its internal data while making deploys. It's also important to set an initial balance for the account to use for deploys.

```
const MY_ACCOUNT: [u8; 32] = [7u8; 32];

let mut context = TestContextBuilder::new()
    .with_account(MY_ACCOUNT, U512::from(128_000_000))
    .build();
```

Account is type of `[u8; 32]`. Balance is type of `U512`.

Run Smart Contract

Before we can deploy the contract to the context, we need to prepare the request. We call the request a `Session`. Each session call should have 4 elements:

- Wasm file path.
- List of arguments.
- Account context of execution.
- List of keys that authorize the call. See: TODO insert keys management link.

```
let VALUE: &str = "hello world";
let session_code = Code::from("contract.wasm");
let session_args = runtime_args! {
    "value" => VALUE,
};
let session = SessionBuilder::new(session_code, session_args)
    .with_address(MY_ACCOUNT)
    .with_authorization_keys(&[MY_ACCOUNT])
    .build();
context.run(session);
```

Executing `run` will panic if the code execution fails.

Query and Assert

The smart contract we deployed creates a new value `"hello world"` under the key `"special_value"`. Using the `query` function it's possible to extract this value from the blockchain.

```
let KEY: &str = "special_value";
let result_of_query: Result<Value, Error> = context.query(MY_ACCOUNT, &[KEY]);
let returned_value = result_of_query.expect("should be a value");
let expected_value = Value::from_t(VALUE.to_string()).expect("should construct Value
↪");
assert_eq!(expected_value, returned_value);
```

Note that the `expected_value` is a `String` type lifted to the `Value` type. It was also possible to map `returned_value` to the `String` type.

Final Test

The code below is the simple test generated by `cargo-casperlabs` (found in `tests/src/integration_tests.rs` of a project created by the tool).

```
#[cfg(test)]
mod tests {
    use casperlabs_engine_test_support::{Code, Error, SessionBuilder,␣
↪TestContextBuilder, Value};
    use casperlabs_types::{RuntimeArgs, runtime_args, U512};

    const MY_ACCOUNT: [u8; 32] = [7u8; 32];
    // define KEY constant to match that in the contract
    const KEY: &str = "special_value";
    const VALUE: &str = "hello world";

    #[test]
    fn should_store_hello_world() {
        let mut context = TestContextBuilder::new()
            .with_account(MY_ACCOUNT, U512::from(128_000_000))
            .build();

        // The test framework checks for compiled Wasm files in '<current working dir>
↪/wasm'.  Paths
        // relative to the current working dir (e.g. 'wasm/contract.wasm') can also␣
↪be used, as can
        // absolute paths.
        let session_code = Code::from("contract.wasm");
        let session_args = runtime_args! {
            "value" => VALUE,
        };
        let session = SessionBuilder::new(session_code, session_args)
            .with_address(MY_ACCOUNT)
            .with_authorization_keys(&[MY_ACCOUNT])
            .build();

        let result_of_query: Result<Value, Error> = context.run(session).query(MY_
↪ACCOUNT, &[KEY]);

        let returned_value = result_of_query.expect("should be a value");

        let expected_value = Value::from_t(VALUE.to_string()).expect("should␣
↪construct Value");
        assert_eq!(expected_value, returned_value);
    }
}

fn main() {
    panic!("Execute \"cargo test\" to test the contract, not \"cargo run\".");
}
```

WASM File Size

We encourage shrinking the WASM file size as much as possible. Smaller deploys cost less and save the network bandwidth. We recommend reading Shrinking .wasm Code Size chapter of The Rust Wasm Book.

3.1.3 CasperLabs Contract DSL

The DSL is a way to help developers avoid boilerplate code. This boilerplate code is repetitive and well structured, enabling the creation of a DSL that uses macros to create this boilerplate code- which simplifies the creation and standardization of contracts.

The code will be a bit larger after the macros run and the boilerplate code is added into the contract. The macros operate like templates.

The macros work with Contract Headers (released as part of the 0.20 release). Contracts created prior to 0.20 will need to be upgraded to use the macros.

Table of Contents

Getting Started with the DSL

Since the DSL uses macros, it works like templates in the smart contract, so it's necessary to tell the Rust compiler where the macros are located for each smart contract. The aim of this guide is to describe how to configure the smart contract to use the DSL.

About the DSL

The DSL is designed specifically for Rust Smart Contrats.

- The `constructor_macro` creates the code that sets up the contract in the runtime and locates the contract in the runtime when execution begins (this is the deploy function that creates the entry point & stores the deploy hash stored under some function name in the runtime). Think of the function templated by the constructor macro as your `main` function, while the contract macro sets up the function definitions within the calls.
- The `contract_macro` generates the code for the headers for each of the entry points that use it.
- The `casperlabs_method` creates an entry point for any function in your contract.

Pre-Requisites - Set up the Rust SDK

Please use the Rust SDK to create your smart contract project before setting up the DSL.

Getting the Macros

The source code for the macros is located at GitHub. To import the macros, include a line in the `Cargo.toml` file in the `/contract` folder for your smart contract. The entry needs to appear in the `[dependencies]` section. This entry will import the macros into your project. There are a few sources for the macros.

From Crates.io

To use the crate available on crates.io include the following entry in the `Cargo.toml` file for the smart contract.

```
contract_macro = { package = "casperlabs_contract_macro", version = "0.1.0" }
```

From Github

To obtain the macros from Github, include this entry in `Cargo.toml`:

```
contract_macro = { git = "https://github.com/CasperLabs/casperlabs_contract_macro",␣
↪branch = "in_progress", package = "contract_macro"}
```

Local package

This example `Cargo.toml` entry uses a local path for the macros:

```
contract_macro = { path = "../../casperlabs-node/smart_contracts/contract_macro" }
```

Using the DSL

To use the DSL, simply add the following line to the `use` section of the contract.

```
use contract_macro::{casperlabs_constructor, casperlabs_contract, casperlabs_method};
```

This line can go after the last `use` line in the blank contract created by `cargo-casperlabs`.

Remember, if you are using the crates.io package, you may have to use the package as `casperlabs_contract_macro`. This depends entirely on how you import the package in your `Cargo.toml` file

Example Counter Contract

The following contract creates a counter in storage. Each time the contract is invoked, the counter is incremented by 1.

```
extern crate alloc;
use alloc::{collections::BTreeSet, string::String};

// import casperlabs contract api
use contract::{
        contract_api::{runtime, storage},
        unwrap_or_revert::UnwrapOrRevert,
};
// import the contract macros
use contract_macro::{casperlabs_constructor, casperlabs_contract, casperlabs_method};
use std::convert::TryInto;

// import casperlabs types
use types::{
        bytesrepr::{FromBytes, ToBytes},
        contracts::{EntryPoint, EntryPointAccess, EntryPointType, EntryPoints},
        runtime_args, CLType, CLTyped, Group, Key, Parameter, RuntimeArgs, URef,
};

const KEY: &str = "special_value";

// macro to set up the contract
```

(continues on next page)

(continued from previous page)

```
#[casperlabs_contract]
mod tutorial {
        use super::*;

// constructor macro that sets up the methods, values and keys required for the␣
↪contract.

        #[casperlabs_constructor]
        fn init_counter(initial_value: u64) {
            let value_ref: URef = storage::new_uref(initial_value);
            let value_key: Key = value_ref.into();
            runtime::put_key(KEY, value_key);
        }

// method macro that defines a new entry point for the contract.

        #[casperlabs_method]
        fn update_counter() {
            let old_value: u64 = key(KEY).unwrap();
            let new_value = old_value + 1;
            set_key(KEY, new_value);
        }

// method macro that defines a new entry point for the contract.

        #[casperlabs_method]
        fn get_counter_value() -> u64 {
            key(KEY).unwrap()
        }

        fn key<T: FromBytes + CLTyped>(name: &str) -> Option<T> {
            match runtime::get_key(name) {
                None => None,
                Some(maybe_key) => {
                    let key = maybe_key.try_into().unwrap_or_revert();
                    let value = storage::read(key).unwrap_or_revert().unwrap_or_
↪revert();
                    Some(value)
                }
            }
        }

        fn set_key<T: ToBytes + CLTyped>(name: &str, value: T) {
            match runtime::get_key(name) {
                Some(key) => {
                    let key_ref = key.try_into().unwrap_or_revert();
                    storage::write(key_ref, value);
                }
                None => {
                    let key = storage::new_uref(value).into();
                    runtime::put_key(name, key);
                }
            }
        }
}
```

Testing the Example Contract:

If you set up your contract using `cargo-casperlabs` you can test your contract using the local runtime.

Set up the runtime following the steps in the *testing* section of this guide to set up the runtime context.

The following test will check whether or not the tutorial contract is working properly:

```
#[cfg(test)]
mod tests {
    use test_support::{Code, SessionBuilder, TestContextBuilder};
    use types::{account::AccountHash, runtime_args, RuntimeArgs, U512};

    const MY_ACCOUNT: AccountHash = AccountHash::new([7u8; 32]);
    const KEY: &str = "special_value";
    const CONTRACT: &str = "tutorial";

    #[test]
    fn should_initialize_to_zero() {
        let mut context = TestContextBuilder::new()
            .with_account(MY_ACCOUNT, U512::from(128_000_000))
            .build();
        let session_code = Code::from("contract.wasm");
        let session_args = runtime_args! {
            "initial_value" => 0u64
        };
        let session = SessionBuilder::new(session_code, session_args)
            .with_address(MY_ACCOUNT)
            .with_authorization_keys(&[MY_ACCOUNT])
            .with_block_time(0)
            .build();
        context.run(session);
        let check: u64 = match context.query(MY_ACCOUNT, &[CONTRACT, KEY]) {
            Err(_) => panic!("Error"),
            Ok(maybe_value) => maybe_value
                .into_t()
                .unwrap_or_else(|_| panic!("{} is not expected type.", KEY)),
        };
        assert_eq!(0, check);
    }
}
```

Advanced Options

Once the base logic of the smart contract is in place, it's desirable to optimize the contract for the blockchain. This will require digging into the actual code that the DSL generates. This section will describe the steps to do this. Once the code has been expanded and then changed, make sure to remove the macros from the project configuration before saving the changes.

Debugging Contracts

It is possible to debug Rust contracts inside any IDE that supports breakpoints and watches. Make sure that the IDE supports Rust development and tools.

Expanding the Code

When the rust compiler encounters each of the macros, it 'expands' the code and adds additional lines of code for each of the macros. The resultant expanded code is then compiled to the wasm which can then be deployed to the blockchain.

The `cargo expand` tool will run the macros and append the boilerplate code to the contract without compiling the code.

Install the tooling with the following command:

```
cargo install cargo-expand
```

Run this command in the folder containing the smart contract code:

```
cargo expand
```

Example Simple Counter Contract

Running `cargo-expand` on the simple counter contract yeilds this output:

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std;
extern crate alloc;
use alloc::{collections::BTreeSet, string::String};
use contract::{
    contract_api::{runtime, storage},
    unwrap_or_revert::UnwrapOrRevert,
};
use casperlabs_contract_macro::{casperlabs_constructor, casperlabs_contract,␣
↪casperlabs_method};
use std::convert::TryInto;
use types::{
    CLValue,
    bytesrepr::{FromBytes, ToBytes},
    contracts::{EntryPoint, EntryPointAccess, EntryPointType, EntryPoints},
    runtime_args, CLType, CLTyped, Group, Key, Parameter, RuntimeArgs, URef,
};
const KEY: &str = "special_value";
fn __deploy(initial_value: u64) {
    let (contract_package_hash, _) = storage::create_contract_package_at_hash();
    let _constructor_access_uref: URef = storage::create_contract_user_group(
        contract_package_hash,
        "constructor",
        1,
        BTreeSet::new(),
    )
    .unwrap_or_revert()
    .pop()
    .unwrap_or_revert();
    let constructor_group = Group::new("constructor");
    let mut entry_points = EntryPoints::new();
    entry_points.add_entry_point(EntryPoint::new(
```

(continues on next page)

(continued from previous page)

```
        String::from("init_counter"),
        <[_]>::into_vec(box [Parameter::new("initial_value", CLType::U64)]),
        CLType::Unit,
        EntryPointAccess::Groups(<[_]>::into_vec(box [constructor_group])),
        EntryPointType::Contract,
    ));
    entry_points.add_entry_point(EntryPoint::new(
        String::from("update_counter"),
        ::alloc::vec::Vec::new(),
        CLType::Unit,
        EntryPointAccess::Public,
        EntryPointType::Contract,
    ));
    entry_points.add_entry_point(EntryPoint::new(
        String::from("get_counter_value"),
        ::alloc::vec::Vec::new(),
        CLType::Unit,
        EntryPointAccess::Public,
        EntryPointType::Contract,
    ));
    let (contract_hash, _) =
        storage::add_contract_version(contract_package_hash, entry_points,␣
↪Default::default());
    runtime::put_key("tutorial", contract_hash.into());
    let contract_hash_pack = storage::new_uref(contract_hash);
    runtime::put_key("tutorial_hash", contract_hash_pack.into());
    runtime::call_contract::<()>(contract_hash, "init_counter", {
        let mut named_args = RuntimeArgs::new();
        named_args.insert("initial_value", initial_value);
        named_args
    });
}
#[no_mangle]
fn call() {
    let initial_value: u64 = runtime::get_named_arg("initial_value");
    __deploy(initial_value)
}
fn __init_counter(initial_value: u64) {
    let value_ref: URef = storage::new_uref(initial_value);
    let value_key: Key = value_ref.into();
    runtime::put_key(KEY, value_key);
}
#[no_mangle]
fn init_counter() {
    let initial_value: u64 = runtime::get_named_arg("initial_value");
    __init_counter(initial_value)
}
fn __update_counter() {
    let old_value: u64 = key(KEY).unwrap();
    let new_value = old_value + 1;
    set_key(KEY, new_value);
}
#[no_mangle]
fn update_counter() {
    __update_counter();
}
fn __get_counter_value() -> u64 {
```

(continues on next page)

(continued from previous page)

```
    key(KEY).unwrap()
}
#[no_mangle]
fn get_counter_value() {
    let val: u64 = __get_counter_value();
    ret(val)
}
fn key<T: FromBytes + CLTyped>(name: &str) -> Option<T> {
    match runtime::get_key(name) {
        None => None,
        Some(maybe_key) => {
            let key = maybe_key.try_into().unwrap_or_revert();
            let value = storage::read(key).unwrap_or_revert().unwrap_or_revert();
            Some(value)
        }
    }
}
fn set_key<T: ToBytes + CLTyped>(name: &str, value: T) {
    match runtime::get_key(name) {
        Some(key) => {
            let key_ref = key.try_into().unwrap_or_revert();
            storage::write(key_ref, value);
        }
        None => {
            let key = storage::new_uref(value).into();
            runtime::put_key(name, key);
        }
    }
}
fn ret<T: CLTyped + ToBytes>(value: T) {
    runtime::ret(CLValue::from_t(value).unwrap_or_revert())
}
```

3.1.4 Testing Smart Contracts Locally

As part of the CasperLabs local Rust contract development environment we provide an in-memory virtual machine you can run your contract against. A full node is not required for testing. The testing framework is designed to be used in the following way:

1. Initialize the system (context).
2. Deploy or call the smart contract.
3. Query the context for changes and assert the result data matches expected values.

It is also possible to create build scripts with this environment and set up continuous integration for contract code.This environment enables the testing of blockchain enabled systems from end to end.

The TestContext for Rust Contracts

A `TestContext` provides a virtual machine instance. It should be a mutable object as its internal data will change with each deploy. It's also important to set an initial balance for the account to use for deploys, as the system requires a balance in order to create an account.

```
const MY_ACCOUNT: [u8; 32] = [7u8; 32];

let mut context = TestContextBuilder::new()
    .with_account(MY_ACCOUNT, U512::from(128_000_000))
    .build();
```

Account is type of `[u8; 32]`. Balance is type of `U512`.

Running the Rust Smart Contract

Before the contract can be deployed to the context, the request has to be prepared. A request is referred to as a `Session`. Each session call has 4 elements:

- A Wasm file path.
- A list of arguments.
- The account context for execution.
- The list of keys that authorize the call.

Here is an example of a prepared request:

```
let VALUE: &str = "hello world";
let session_code = Code::from("contract.wasm");
let session_args = runtime_args! {
    "value" => VALUE,
};
let session = SessionBuilder::new(session_code, session_args)
    .with_address(MY_ACCOUNT)
    .with_authorization_keys(&[MY_ACCOUNT])
    .build();
context.run(session);
```

Executing `run` will panic if the code execution fails.

Query and Assert

The smart contract creates a new value `"hello world"` under the key `"special_value"`. Using the `query` function it's possible to extract this value from the global state of the blockchain.

```
let KEY: &str = "special_value";
let result_of_query: Result<Value, Error> = context.query(MY_ACCOUNT, &[KEY]);
let returned_value = result_of_query.expect("should be a value");
let expected_value = Value::from_t(VALUE.to_string()).expect("should construct Value
↪");
assert_eq!(expected_value, returned_value);
```

Note that the `expected_value` is a `String` type lifted to the `Value` type. It was also possible to map `returned_value` to the `String` type.

Final Test

The code below is the simple test generated by `cargo-casperlabs` (found in `tests/src/integration_tests.rs` of a project created by the tool).

```
#[cfg(test)]
mod tests {
    use casperlabs_engine_test_support::{Code, Error, SessionBuilder,␣
↪TestContextBuilder, Value};
    use casperlabs_types::{RuntimeArgs, runtime_args, U512};

    const MY_ACCOUNT: [u8; 32] = [7u8; 32];
    // define KEY constant to match that in the contract
    const KEY: &str = "special_value";
    const VALUE: &str = "hello world";

    #[test]
    fn should_store_hello_world() {
        let mut context = TestContextBuilder::new()
            .with_account(MY_ACCOUNT, U512::from(128_000_000))
            .build();

        // The test framework checks for compiled Wasm files in '<current working dir>
↪/wasm'.  Paths
        // relative to the current working dir (e.g. 'wasm/contract.wasm') can also␣
↪be used, as can
        // absolute paths.
        let session_code = Code::from("contract.wasm");
        let session_args = runtime_args! {
            "value" => VALUE,
        };
        let session = SessionBuilder::new(session_code, session_args)
            .with_address(MY_ACCOUNT)
            .with_authorization_keys(&[MY_ACCOUNT])
            .build();

        let result_of_query: Result<Value, Error> = context.run(session).query(MY_
↪ACCOUNT, &[KEY]);

        let returned_value = result_of_query.expect("should be a value");

        let expected_value = Value::from_t(VALUE.to_string()).expect("should␣
↪construct Value");
        assert_eq!(expected_value, returned_value);
    }
}

fn main() {
    panic!("Execute \"cargo test\" to test the contract, not \"cargo run\".");
}
```

3.1.5 Deploying Contracts

Ultimately, smart contracts should be run on the blockchain. Once your smart contract is complete, it's time to deploy your contract to a blockchain. Deploying a contract has a few pre-requisites:

CasperLabs Client

The client software communicates with the network to transmit your deployments to the network. Clients exist for a variety of development platforms such as Python and Javascript. The official client is the Python Client It is also

possible to create a client to meet specific needs as well.

It's possible to use pre-built binaries or build from source. Both provide the casperLabs-client.

Ensure that your client matches the version of the network you intend to deploy to.

Using Binaries

- **Python**: casperlabs-client

Building from Source

Instructions

Make sure you have pre-requisites installed and you can build the casperlabs-client from source. If you build from source, you will need to add the build directories to your PATH.

Or, you can run the client commands from the root directory of the repo, using explicit paths to the binaries.

Token to Pay for Deployments

Blockchains are supported by infrastructure providers called "Validators". To use the Validator infrastructure, it's necessary to acquire token to pay for deployments (transactions).

Target Network

When sending a deploy, the client needs to know which host will receive the deployment. The host parameter is either a DNS name or IP address of the host.

Private Key

Blockchains use private key encryption to sign transactions. The private key used to sign the deployment must be the private key of the account that is being used to pay for the transaction.

Advanced Deployments

CasperLabs supports complex deployments. To learn more about what is possible visit GitHub.

Sending a Deployment to the Testnet

The easiest way to deploy a contract is to use an existing public network. CasperLabs provides a Testnet for this.

- Go to CasperLabs Clarity and create an account.
- Place your private key in a location that you can access during the deployment.
- Request tokens from the faucet to fund your account.

The Testnet is operated by external validators that can accept transactions at the following endpoints:

- deploy.casperlabs.io

- 18.219.70.138
- 62.171.172.72
- 52.88.90.57
- 116.203.69.88
- 35.158.200.94

The default port is 40401

Check the Client Version

There is an official Python client.

To check the client version run:

```
pip show casperlabs_client
```

If you want to send your deployments to an external network, use the latest released version of the client. If you are working off of dev, build the client locally and check the gitHash.

A Basic Deployment

As described above, a basic deployment must provide some essential information. Here is an example deployment using the Python client that will work with the basic contract we created using the Contracts SDK for Rust:

```
casperlabs_client --host deploy.casperlabs.io deploy \
    --session contract.wasm \
    --session-args '[{"name": "surname", "value": {"string_value": "Nakamoto"}}]' \
    --private-key account-private.pem \
    --payment-amount 10000000
```

If your deployment works, expect to see a success message that looks like this:

```
Success! Deploy 8428717f1cfc9cc5c047f503661e9c0fc2a495ead44305a807bead130cbd181f␣
↪deployed
```

Note: Each deploy gets a unique hash. This is part of the cryptographic security of blockchain technology. No two deploys will ever return the same hash.

Check the Deploy Status

The process of sending a deployment is separate from the processing of the contract by the system. Deployments queue up in the system before being added to a block. Therefore, it's important to check the status of the deployment. For example, if an incorrect key is used to sign the deployment, it's possible that the deploy does not process on the network, in spite of a successful deploy message. Deploy statuses can also be checked in the Clarity Explorer and also by the client.

```
casperlabs_client --host deploy.casperlabs.io show-deploy \
    8428717f1cfc9cc5c047f503661e9c0fc2a495ead44305a807bead130cbd181f

deploy {
  deploy_hash: "8428717f1cfc9cc5c047f503661e9c0fc2a495ead44305a807bead130cbd181f"
```

(continues on next page)

(continued from previous page)

```
...
  cost: 126165
}
status {
  state: PROCESSED
}
```

Check the Contract Executed Successfully

You can use the client's method `query-state` to see a specific named key of the account. It works the same as the `query` method from the testing framework.

```
casperlabs_client --host deploy.casperlabs.io query-state \
    --block-hash "f21fc0763279ad8349b0c0fce08e1ed678412d5e234a92e3063d4d5a35ee0739" \
    --type address \
    --key "0cc94662d68bd71b03083e38094f0b0e07a1bbb485969b6e68f21f4577fe928a" \
    --path "special_value"

string_value: "Nakamoto"
```

3.1.6 GraphQL

The CasperLabs node software includes a GraphQL console which you can use to explore the schema and build queries with the help of auto-completion. A GraphQL query looks at the blockchain on a single node. It is important to know which network you are querying when using a GraphQL interface.

To query the blockchain on Testnet, navigate to: CasperLabs Clarity.

Using GraphQL for Querying and Debugging Contracts

- View what graphs are available by clicking the DOCS and SCHEMA buttons on the right-hand side of the screen.
- Run a query, start typing "query" or "subscription" into the left-hand pane and see what the code completion offers up.

Note: The DOCS can be kept open on the right hand side to see what's available and closed when you finished your query.

For example:

You can use the following query to see 5 most recent ranks of the DAG:

```
query {
  dagSlice(depth: 5) {
      blockHash
  }
}
```

An example of querying what is stored at an account:

```
query {
  globalState(
    blockHashBase16Prefix: "The latest block hash"
    StateQueries: [
      {
        keyType: Address
        keyBase16: "Your Hex Key"
        pathSegments: []
      }
    ]
  ) {
    value {
      __typename
      ... on Account {
        pubKey
        associatedKeys {
          pubKey
          weight
        }
        actionThreshold {
          deploymentThreshold
          keyManagementThreshold
        }
      }
    }
  }
}
```

Using the "COPY CURL" button will return the equivalent pure HTTP/JSON command.

- Press the "play" button in the middle of the tool screen to see the query response.

For further details on GraphQL check out source code.

3.1.7 Execution Error Codes

As mentioned in Writing Rust Contracts, smart contracts can exit with an error code defined by an `ApiError`. Descriptions of each variant are found here and include the following sub-types:

- proof of stake errors
- mint errors
- custom user error code

An `ApiError` of one of these sub-types maps to an exit code with an offset defined by the sub-type. For example, an `ApiError::User(2)` maps to an exit code of `65538` (i.e. `65536 + 2`). You can find a mapping from contract exit codes to `ApiError` variants here.

3.1.8 Writing AssemblyScript Smart Contracts

CasperLabs maintains @casperlabs/contract to allow developers to create smart contracts using AssemblyScript. The package source is hosted in the main CasperLabs repository.

Installation

For each smart contract it's necessary to create a project directory and initialize it.

```
mkdir project
cd project
npm init
```

The `npm init` process prompts for various details about the project; answer as you see fit but you may safely default everything except `name` which should follow the convention of `your-contract-name`.

Then install assembly script and this package in the project directory.

```
npm install --save-dev assemblyscript@0.9.1
npm install --save @casperlabs/contract
```

Contract API Documentation

The Assemblyscript contract API documentation can be found at https://www.npmjs.com/package/@casperlabs/contract

Usage

Add script entries for assembly script to your project's `package.json`; note that your contract name is used for the name of the wasm file.

```
{
  "name": "your-contract-name",
  ...
  "scripts": {
    "asbuild:optimized": "asc assembly/index.ts -b dist/your-contract-name.wasm --
↪validate --optimize --use abort=",
    "asbuild": "npm run asbuild:optimized",
    ...
  },
  ...
}
```

In the project root, create an `index.js` file with the following contents:

```
const fs = require("fs");

const compiled = new WebAssembly.Module(fs.readFileSync(__dirname + "/dist/your-
↪contract-name.wasm"));

const imports = {
    env: {
        abort(_msg, _file, line, column) {
            console.error("abort called at index.ts:" + line + ":" + column);
        }
    }
};

Object.defineProperty(module, "exports", {
    get: () => new WebAssembly.Instance(compiled, imports).exports
});
```

Create an `assembly/tsconfig.json` file in the following way:

```
{
  "extends": "../node_modules/assemblyscript/std/assembly.json",
  "include": [
    "./**/*.ts"
  ]
}
```

Sample smart contract

Create a `assembly/index.ts` file. This is where the code for the contract has to go.

You can use the following sample snippet which demonstrates a very simple smart contract that immediately returns an error, which will write a message to a block if executed on the CasperLabs platform.

```
//@ts-nocheck
import {Error, ErrorCode} from "@casperlabs/contract/error";

// simplest possible feedback loop
export function call(): void {
    Error.fromErrorCode(ErrorCode.None).revert(); // ErrorCode: 1
}
```

If you prefer a more complicated first contract, you can look at example contracts on the CasperLabs github repository for inspiration.

Compile to wasm

To compile the contract to wasm, use npm to run the asbuild script from the project root.

```
npm run asbuild
```

If the build is successful, there will be a `dist` folder in the root folder and in it should be `your-contract-name.wasm`

3.1.9 Working with Ethereum Keys

The Casper platform supports two types of signatures for the creation of accounts and signing of transactions: secp256k1 and ed25519. Internally, the system does this is by representing the account as the hash of the public key + encryption type. By taking the hash of the public key and algorithm name, the likelihood of account collision is eliminated.

In this section we'll explore secp256k1, commonly known as Ethereum keys.

Key Generation

The CasperLabs client enables the creation of a new secp256k1 key-pair and the coresponding account hash.

```
$ mkdir secp256k1-keys

$ casperlabs_client keygen -a secp256k1 secp256k1-keys/
```

(continues on next page)

(continued from previous page)

```
Keys successfully created in directory: /tmp/secp256k1-keys

$ tree secp256k1-keys/
secp256k1-keys/
├── account-id-hex
├── account-private.pem
└── account-public.pem
```

Using Ethereum Keys in CasperLabs

It is possible to use existing secp256k1 keys. Ethereum private keys usually are presented in the form of hex string that represents 256 bits.

Example of the Ethereum private key.

```
18d763eef165a8334e32f8a7c6c6592f053d1716d779a0cad76ee5cddee79e8c
```

CasperLabs requires the private key to be in the PEM format, so it's required to do the conversion.

This is the example JS script to do it:

```
$ cat convert-to-pem.js
var KeyEncoder = require('key-encoder'),
    keyEncoder = new KeyEncoder.default('secp256k1');
let priv_hex = "18d763eef165a8334e32f8a7c6c6592f053d1716d779a0cad76ee5cddee79e8c";
let priv_pem = keyEncoder.encodePrivate(priv_hex, "raw", "pem");
console.log(priv_pem);

$ node convert-to-pem.js > eth-private.pem

$ cat eth-private.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIBjXY+7xZagzTjL4p8bGWS8FPRcW13mgytdu5c3e556MoAcGBSuBBAAK
oUQDQgAEpV4dVaPeAEaH0VXrQtLzjpGt1pui1q08311em6wDCchGNjzsnOY7stGF
tlKF2V5RFQn4rzkwipSYnrqaPf1pTA==
-----END EC PRIVATE KEY-----
```

The last missing element is obtaining the Account Hash. Because the internal representation of accounts is the hash of the public key and the algorithm, this hash is necessary in order to query the state. Creating a deploy and reading the header value is the easiest way to obtain this information.

```
$ casperlabs_client --host localhost deploy --private-key eth-private.pem --session␣
↪contract.wasm --algorithm secp256k1
Success! Deploy a124a6e68d804b4222cc12a3a9dfd1b64d97005897246820 95f87705614c54c8␣
↪deployed

$ casperlabs_client --host localhost show-deploy␣
↪a124a6e68d804b4222cc12a3a9dfd1b64d97005897246820 95f87705614c54c8
deploy {
  deploy_hash: "a124a6e68d804b4222cc12a3a9dfd1b64d97005897246820 95f87705614c54c8"
  header {
    account_public_key_hash:
↪"c1f358d347c3134d0eade886c8379ebf05788ee7fb713d657ce358b6cf3b35a6"
    timestamp: 1594220659169
    body_hash: "e6b3f3fcf2fb5d3ab1cc74a4fa0271b68c4f75a9273d6982a95ffb9ee4cdd4e7"
```

(continues on next page)

(continued from previous page)

```
  }
  approvals {
    ...
  }
}
status {
  state: PENDING
}
```

`account_public_key_hash` shows the Account Hash generated from `eth-private.pem`.

3.1.10 Solidity to Rust Transpiler

tl;dr

Introducing Caspiler - Transpile Solidity to Rust and access the cool features of Casper!

Smart Contracts at CasperLabs

The CasperLabs Virtual Machine runs smart contracts that compile to Webassembly. There are two ecosystems that provide compilation targets for webassembly: Rust and AssemblyScript. CasperLabs provides smart contract libraries to support development for both of these languages. The core development of the Casper Protocol is taking place in Rust, and as a result, there are many Rust tools that make rapid Smart Contracts development possible. It is widely recognized that most smart contracts in use today have been authored in Solidity for the EVM (Ethereum Virtual Machhine).

Solidity

Without any doubt, the existence and simplicity of Solidity is one of the key factors behind the growth of Ethereum. There is a large group of developers for whom Solidity is still the best tool for expressing their Smart Contract ideas. At CasperLabs we feel a strong connection with the Ethereum community, so we decided to include support for Solidity via a transpiler.

Transpiler

Transpiling is a well known process of turning code written in one high-level language into another high-level language. At the moment the most popular example is the TypeScript to JavaScript transpiler.

We have concluded that Solidity support is much easier and efficient to achieve by transpiling Solidity to Rust, rather than by compiling Solidity to WASM bytecode for the following reasons:

- Solidity features are easy to express in Rust, which is a much richer language.
- The shape of CasperLabs DSL is similar to Solidity.
- The CasperLabs Rust toolchain is something we want to leverage, rather than coding it from scratch.
- The CasperLabs execution model is different than Ethereum's, therefore it's easier to translate it on the language level, than on the bytecode level.

Solidity to Rust Migration

Having transpiler gives Smart Contract developers a powerful tool for the migration of the existing Solidity source code to Rust if they wish to use it.

Simple Example

Let's see how the Solidity code is being transpiled to the CasperLabs Rust DSL. There is almost one to one translation of the core components: `contract`, `constructor` and `method`.

Solidity

```
contract Storage {
     string value;

     constructor(string initValue) {
         value = initValue;
     }

     function getValue() public view returns (string) {
         return value;
     }

     function setValue(string newValue) public {
         value = newValue;
     }
}
```

CasperLabs Rust DSL

```
#[casperlabs_contract]
mod Storage {

    #[casperlabs_constructor]
    fn constructor(initValue: String) {
        let value: String = initValue;
        set_key("value", value);
    }

    #[casperlabs_method]
    fn getValue() {
        ret(get_key::<String>("value"));
    }

    #[casperlabs_method]
    fn setValue(newValue: String) {
        let value: String = newValue;
        set_key("value", value);
    }
}
```

ERC20

It is possible to transpile a complex Smart Contracts like ERC20 Token. Full example with tests can be found in this GitHub repository.

Deploying to Testnet.

Take a look at the deployment instructions in the dApp developer guide for details.

3.1.11 ERC-20 Tutorial

Most of smart contract developers are famillar with ERC-20 standard. It comes from the Ethereum ecosystem and it's a well established standard for building new smart-contract-based tokens. In this tutorial we will implement it for the CasperLabs platform.

The final code is available on our Github CasperLabs/erc20. If you haven't read Writing Rust Contracts on CasperLabs do it first.

Content of the tutorial:

Prepare

Before we start the development, let's prepare the repository first.

The code will be divided into 3 crates: `logic`, `contract` and `tests`. Let's start with generating a new project and applying a few changes.

Generate a new project using Cargo CasperLabs.

```
$ cargo casperlabs erc20-tutorial
$ cd erc20-tutorial
$ tree
.
├── contract
│   ├── Cargo.toml
│   ├── rust-toolchain
│   └── src
│       └── lib.rs
└── tests
    ├── build.rs
    ├── Cargo.toml
    ├── rust-toolchain
    ├── src
    │   └── integration_tests.rs
    └── wasm
        ├── mint_install.wasm
        ├── pos_install.wasm
        └── standard_payment.wasm

5 directories, 10 files
```

It's better to have one Cargo Workspace rather than separated crates. Create a new file `Cargo.toml` in the root directory.

```
# Cargo.toml

[workspace]

members = [
    "contract",
    "tests"
]

[profile.release]
lto = true
```

Now the project will have only one `target` directory placed in the root directory, so `tests/build.rs` must be adjusted. Change

```
const ORIGINAL_WASM_DIR: &str = "../contract/target/wasm32-unknown-unknown/release";
```

to

```
const ORIGINAL_WASM_DIR: &str = "../target/wasm32-unknown-unknown/release";
```

When building WASM file we don't want to build all crates, but only the `contract` one. Change

```
const BUILD_ARGS: [&str; 2] = ["build", "--release"];
```

to

```
const BUILD_ARGS: [&str; 4] = ["build", "--release", "-p", "contract"];
```

Finally remove `rust-toolchain` files from `contract` and `tests` crates and make just one in the root directory.

```
$ rm contract/rust-toolchain
$ mv tests/rust-toolchain .
```

Test the changes by compiling `contract` crate and executing `cargo test` on `tests` crate.

```
$ cargo build --release -p contract
$ cargo test -p tests
```

The source code shown in this tutorial is adapted from using the casperlabs solidity to rust transpilier. Previously, we would implement a separate `logic` crate to implement the ERC20 standard functionality. However, now with the new contract headers system, we can directly implement them into the `contract` crate.

ERC-20 Standard

The ERC-20 standard is defined in an Ethereum Improvement Proposal (EIP). Read it carefully, as it defines the methods we'll implement:

- balance_of
- transfer
- total_supply
- approve
- allowance
- transfer_from

- mint

Create a New Empty Smart Contract

Rust development with Casper is easy with the Rust SDK. Create a new contract by following these steps.

Include the Casper DSL

Contract development is easier with the DSL. Update the `Cargo.toml` with the DSL package.

Contract Initialization

When the contract is deployed it must be initialized with some values, this can be done with the help of the `casperlabs_constructor`. This also initializes the balance with the starting token supply.

```
#[casperlabs_constructor]
fn constructor(tokenName: String, tokenSymbol: String, tokenTotalSupply: U256) {
    set_key("_name", tokenName);
    set_key("_symbol", tokenSymbol);
    let _decimals: u8 = 18;
    set_key("_decimals", _decimals);
    set_key(&new_key("_balances", runtime::get_caller()), tokenTotalSupply);
    let _totalSupply: U256 = tokenTotalSupply;
    set_key("_totalSupply", _totalSupply);
}
```

We then also add a few helper functions to set, and retrieve values from keys. The `[casperlabs_method]` macro facilitates this. Notice that each of these helper functions reference each of the `set_key` definitions in the constructor.

```
#[casperlabs_method]
fn name() {
    ret(get_key::<String>("_name"));
}

#[casperlabs_method]
fn symbol() {
    ret(get_key::<String>("_symbol"));
}

#[casperlabs_method]
fn decimals() {
    ret(get_key::<u8>("_decimals"));
}

// write to storage
fn get_key<T: FromBytes + CLTyped + Default>(name: &str) -> T {
    match runtime::get_key(name) {
        None => Default::default(),
        Some(value) => {
            let key = value.try_into().unwrap_or_revert();
            storage::read(key).unwrap_or_revert().unwrap_or_revert()
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
// write to storage
fn set_key<T: ToBytes + CLTyped>(name: &str, value: T) {
    match runtime::get_key(name) {
        Some(key) => {
            let key_ref = key.try_into().unwrap_or_revert();
            storage::write(key_ref, value);
        }
        None => {
            let key = storage::new_uref(value).into();
            runtime::put_key(name, key);
        }
    }
}

fn new_key(a: &str, b: AccountHash) -> String {
    format!("{}_{}", a, b)
}
```

Total Supply, Balance and Allowance

We are ready now to define first ERC-20 methods. Below is the implementation of `balance_of`, `total_supply` and `allowance`. These are read-only methods.

```
#[casperlabs_method]
fn totalSupply() {
    ret(get_key::<U256>("_totalSupply"));
}

#[casperlabs_method]
fn totalSupply() {
    ret(get_key::<U256>("_totalSupply"));
}

#[casperlabs_method]
fn allowance(owner: AccountHash, spender: AccountHash) -> U256 {
    let key = format!("_allowances_{}_{}", owner, spender);
    get_key::<U256>(&key)
}
```

Transfer

Finally we can implement `transfer` method, so it's possible to transfer tokens from `sender` address to `recipient` address. If the `sender` address has enough balance then tokens should be transferred to the `recipient` address. TODO: Otherwise return the `ERC20TransferError::NotEnoughBalance` error.

```
#[casperlabs_method]
fn transfer(recipient: AccountHash, amount: U256) {
   _transfer(runtime::get_caller(), recipient, amount);
}


fn _transfer(sender: AccountHash, recipient: AccountHash, amount: U256) {
   let new_sender_balance: U256 = (get_key::<U256>(&new_key("_balances", sender)) -␣
↪amount);
```

(continues on next page)

(continued from previous page)

```
    set_key(&new_key("_balances", sender), new_sender_balance);
    let new_recipient_balance: U256 = (get_key::<U256>(&new_key("_balances",␣
↪recipient)) + amount);
    set_key(&new_key("_balances", recipient), new_recipient_balance);
}
```

Approve and Transfer From

The last missing functions are `approve` and `transfer_from`. `approve` is used to allow another address to spend tokens on my behalf.

```
#[casperlabs_method]
fn approve(spender: AccountHash, amount: U256) {
    _approve(runtime::get_caller(), spender, amount);
}

fn _approve(owner: AccountHash, spender: AccountHash, amount: U256) {
    set_key(&new_key(&new_key("_allowances", owner), spender), amount);
}
```

`transfer_from` allows to spend approved amount of tokens.

```
#[casperlabs_method]
fn transferFrom(owner: AccountHash, recipient: AccountHash, amount: U256) {
    _transfer(owner, recipient, amount);
    _approve(
      owner,
      runtime::get_caller(),
      (get_key::<U256>(&new_key(
            &new_key("_allowances", owner),
            runtime::get_caller(),
            )) - amount),
       );
}
```

Smart Contract Tests

In this section we'll use the CasperLabs Engine Test Support crate to test the ERC-20 smart contract against the execution environment that is equivalent to what CasperLabs uses in production. Here we will create 2 files that will set up the testing framework for the ERC20 contract.

The following is an example of a finished test.

```
#[test]
fn test_erc20_transfer() {
    let amount = 10;
    let mut token = ERC20Contract::deployed();
    token.transfer(BOB, amount, Sender(ALI));
    assert_eq!(token.balance_of(ALI), ERC20_INIT_BALANCE - amount);
    assert_eq!(token.balance_of(BOB), amount);
}
```

Remove `tests/src/integration_tests.rs` and create three files:

- `tests/src/erc20.rs` - sets up testing context and creates helper functions used by unit tests

- `tests/src/tests.rs` - contains the unit tests
- `tests/src/lib.rs` - required by rust toolchain. Links the other 2 files together

The `tests` crate has a `build.rs` file: effectively a custom build script. It's executed every time before running tests and it compiles `contract` crate in release mode for your convenience and copies the `contract.wasm` file to `tests/wasm` directory. In practice, that means we only need to run `cargo test -p tests` during the development.

Set up Cargo.toml

Define a `tests` package at `tests/Cargo.toml`.

```
[package]
name = "tests"
version = "0.1.1"
authors = ["Your Name here <your email here>"]
edition = "2020"

[dependencies]
casperlabs-contract = "0.6.1"
casperlabs-types = "0.6.1"
casperlabs-engine-test-support = "0.8.1"

[features]
default = ["casperlabs-contract/std", "casperlabs-types/std"]
```

Create ERC20.rs Logic for Testing

Set Up the Testing Context

Start with defining constants like method names, key names and account addresses that will be reused across tests. This initializes the global state with all the data and methods that the smart contract needs in order to run properly.

```
// tests/src/erc20.rs

pub mod account {
    use super::PublicKey;
    pub const ALI: PublicKey = PublicKey::ed25519_from([1u8; 32]);
    pub const BOB: PublicKey = PublicKey::ed25519_from([2u8; 32]);
    pub const JOE: PublicKey = PublicKey::ed25519_from([3u8; 32]);
}

pub mod token_cfg {
    use super::*;
    pub const NAME: &str = "ERC20";
    pub const SYMBOL: &str = "STX";
    pub const DECIMALS: u8 = 18;
    pub fn total_supply() -> U256 { 1_000.into() }
}

pub struct Sender(pub AccountHash);

pub struct Token {
```

(continues on next page)

(continued from previous page)

```
    context: TestContext
}
```

Deploying the Contract

The next step is to define the `ERC20Contract` struct that has its' own VM instance and implements ERC-20 methods. This struct should hold a `TestContext` of its own. The token contract hash and the erc20_indirect session code hash won't change after the contract is deployed, so it's handy to have it available. This code snippet builds the context and includes the compiled `contract.wasm` binary that is being tested. This function creates new instance of `ERC20Contract` with `ALI`, `BOB` and `JOE` having positive initial balance. The contract is deployed using the `ALI` account.

```
// tests/src/erc20.rs

// the contract struct
pub struct Token {
    context: TestContext
}

impl Token {

    pub fn deployed() -> Token {

        // Builds test context with Alice & Bob's accounts
        let mut context = TestContextBuilder::new()
            .with_account(account::ALI, U512::from(128_000_000))
            .with_account(account::BOB, U512::from(128_000_000))
            .build();

        // Adds compiled contract to the context with arguments specified above.
        // For this example it is 'ERC20' & 'STX'
        let session_code = Code::from("contract.wasm");
        let session_args = runtime_args! {
            "tokenName" => token_cfg::NAME,
            "tokenSymbol" => token_cfg::SYMBOL,
            "tokenTotalSupply" => token_cfg::total_supply()
        };

        // Builds the session with the code and arguments
        let session = SessionBuilder::new(session_code, session_args)
            .with_address(account::ALI)
            .with_authorization_keys(&[account::ALI])
            .build();

        //Runs the code
        context.run(session);
        Token { context }
    }
```

Querying the System

The above step has simulated a real deploy on the network. This code snippet describes how to query for the hash of the contract. Contracts are deployed under the context of an account. Since the deployment was created under thhe

context of `account::ALI`, this is what is queried here. The `query_contract` function uses `query` to lookup named keys. It will be used to implement `balance_of`, `total_supply` and `allowance` checks.

```
    fn contract_hash(&self) -> Hash {
        self.context
            .query(account::ALI, &[format!("{}_hash", token_cfg::NAME)])
            .unwrap_or_else(|_| panic!("{} contract not found", token_cfg::NAME))
            .into_t()
            .unwrap_or_else(|_| panic!("{} has wrong type", token_cfg::NAME))
    }

    // This function is a generic helper function that queries for a named key␣
↪defined in the contract.
    fn query_contract<T: CLTyped + FromBytes>(&self, name: &str) -> Option<T> {
        match self.context.query(
            account::ALI,
            &[token_cfg::NAME, &name.to_string()],
        ) {
            Err(_) => None,
            Ok(maybe_value) => {
                let value = maybe_value
                    .into_t()
                    .unwrap_or_else(|_| panic!("{} is not expected type.", name));
                Some(value)
            }
        }
    }

    // Here we call the helper function to query on specific named keys defined in␣
↪the contract.

    // Returns the name of the token
    pub fn name(&self) -> String {
        self.query_contract("_name").unwrap()
    }

    // Returns the token symbol
    pub fn symbol(&self) -> String {
        self.query_contract("_symbol").unwrap()
    }

    // Returns the number of decimal places for the token
    pub fn decimals(&self) -> u8 {
        self.query_contract("_decimals").unwrap()
    }
```

Invoking methods in the Contract

This code snippet describes a generic way to call a specific entry point in the contract.

```
    fn call(&mut self, sender: Sender, method: &str, args: RuntimeArgs) {
        let Sender(address) = sender;
        let code = Code::Hash(self.contract_hash(), method.to_string());
        let session = SessionBuilder::new(code, args)
            .with_address(address)
            .with_authorization_keys(&[address])
```

(continues on next page)

(continued from previous page)

```
            .build();
        self.context.run(session);
    }
```

Invoke each of the getter methods in the Contract.

```
    pub fn balance_of(&self, account: AccountHash) -> U256 {
        let key = format!("_balances_{}", account);
        self.query_contract(&key).unwrap_or_default()
    }

    pub fn allowance(&self, owner: AccountHash, spender: AccountHash) -> U256 {
        let key = format!("_allowances_{}_{}", owner, spender);
        self.query_contract(&key).unwrap_or_default()
    }

    pub fn transfer(&mut self, recipient: AccountHash, amount: U256, sender: Sender) {
        self.call(sender, "transfer", runtime_args! {
            "recipient" => recipient,
            "amount" => amount
        });
    }

    pub fn approve(&mut self, spender: AccountHash, amount: U256, sender: Sender) {
        self.call(sender, "approve", runtime_args! {
            "spender" => spender,
            "amount" => amount
        });
    }

    pub fn transfer_from(&mut self, owner: AccountHash, recipient: AccountHash,␣
↪amount: U256, sender: Sender) {
        self.call(sender, "transferFrom", runtime_args! {
            "owner" => owner,
            "recipient" => recipient,
            "amount" => amount
        });
    }
}
```

Create tests.rs File with Units

Unit Tests

Now that we have a testing context, we can use this context and create unit tests that test the contract code by invoking the functions defined in `tests/src/erc20.rs`. Add these functions to `tests/src/tests.rs`.

```
// tests/src/tests.rs

use crate::erc20::{Token, Sender, account::{ALI, BOB, JOE}, token_cfg};

#[test]
```

(continues on next page)

(continued from previous page)

```
fn test_erc20_deploy() {
    let token = Token::deployed();
    assert_eq!(token.name(), token_cfg::NAME);
    assert_eq!(token.symbol(), token_cfg::SYMBOL);
    assert_eq!(token.decimals(), token_cfg::DECIMALS);
    assert_eq!(token.balance_of(ALI), token_cfg::total_supply());
    assert_eq!(token.balance_of(BOB), 0.into());
    assert_eq!(token.allowance(ALI, ALI), 0.into());
    assert_eq!(token.allowance(ALI, BOB), 0.into());
    assert_eq!(token.allowance(BOB, ALI), 0.into());
    assert_eq!(token.allowance(BOB, BOB), 0.into());
}

#[test]
fn test_erc20_transfer() {
    let amount = 10.into();
    let mut token = Token::deployed();
    token.transfer(BOB, amount, Sender(ALI));
    assert_eq!(token.balance_of(ALI), token_cfg::total_supply() - amount);
    assert_eq!(token.balance_of(BOB), amount);
}

#[test]
#[should_panic]
fn test_erc20_transfer_too_much() {
    let amount = 1.into();
    let mut token = Token::deployed();
    token.transfer(ALI, amount, Sender(BOB));
}

#[test]
fn test_erc20_approve() {
    let amount = 10.into();
    let mut token = Token::deployed();
    token.approve(BOB, amount, Sender(ALI));
    assert_eq!(token.balance_of(ALI), token_cfg::total_supply());
    assert_eq!(token.balance_of(BOB), 0.into());
    assert_eq!(token.allowance(ALI, BOB), amount);
    assert_eq!(token.allowance(BOB, ALI), 0.into());
}

#[test]
fn test_erc20_transfer_from() {
    let allowance = 10.into();
    let amount = 3.into();
    let mut token = Token::deployed();
    token.approve(BOB, allowance, Sender(ALI));
    token.transfer_from(ALI, JOE, amount, Sender(BOB));
    assert_eq!(token.balance_of(ALI), token_cfg::total_supply() - amount);
    assert_eq!(token.balance_of(BOB), 0.into());
    assert_eq!(token.balance_of(JOE), amount);
    assert_eq!(token.allowance(ALI, BOB), allowance - amount);
}

#[test]
#[should_panic]
fn test_erc20_transfer_from_too_much() {
```

(continues on next page)

(continued from previous page)

```
    let amount = token_cfg::total_supply().checked_add(1.into()).unwrap();
    let mut token = Token::deployed();
    token.transfer_from(ALI, JOE, amount, Sender(BOB));
}
```

Configure lib.rs to run everything via cargo

Within the `tests/src/lib.rs` file, add the following lines. This tells cargo which files to use when running the tests.

```
#[cfg(test)]
pub mod tests;
#[cfg(test)]
pub mod erc20;
```

Run the Tests!

Run tests to verify they work. This is run via `bash`.

```
$ cargo test -p tests
```

3.1.12 Key Value Storage Tutorial

This tutorial walks through how to design a simple contract that creates a key that stores a CLType value. This example will show you how to store a u64, string, account hash, or U512 value.The code is available at: https://github.com/CasperLabs/casperlabs-kv-storage

This tutorial will also provide some insight into how to use the Casperlabs smart contract DSL and how contract headers work.

The Contract

Lets start by understanding the structure of the contract itself. Here we create a contract using the `casperlabs_contract` macro and name it `kvstorage_contract`. This is the name under which the contract package will be stored. The next macro we see is the `casperlabs_constructor`, since the a key-value contract is slightly stateless in nature, initialization is not required. However, because `casperlabs_constructor` is a required element, we simply create an empty function.

```
#[casperlabs_contract]
mod kvstorage_contract {

    #[casperlabs_constructor]
    fn init() {}

    #[casperlabs_method]
    fn store_u64(name: String, value: u64) {
        set_key(name.as_str(), value);
    }

    #[casperlabs_method]
```

(continues on next page)

(continued from previous page)

```
    fn get_u64(name: String) -> u64 {
        key(name.as_str())
    }

    #[casperlabs_method]
    fn get_string(name: String) -> String {
        key(name.as_str())
    }

    #[casperlabs_method]
    fn store_u512(name: String, value: U512) {
        set_key(name.as_str(), value);
    }

    #[casperlabs_method]
    fn store_string(name: String, value: String) {
        set_key(name.as_str(), value);
    }

    #[casperlabs_method]
    fn store_account_hash(name: String, value: AccountHash) {
        set_key(name.as_str(), value);
    }

    fn key<T: FromBytes + CLTyped>(name: &str) -> T {
        let key = runtime::get_key(name)
            .unwrap_or_revert()
            .try_into()
            .unwrap_or_revert();
        storage::read(key).unwrap_or_revert().unwrap_or_revert()
    }

    fn set_key<T: ToBytes + CLTyped>(name: &str, value: T) {
        match runtime::get_key(name) {
            Some(key) => {
                let key_ref = key.try_into().unwrap_or_revert();
                storage::write(key_ref, value);
            }
            None => {
                let key = storage::new_uref(value).into();
                runtime::put_key(name, key);
            }
        }
    }
}
```

Testing the Contract

The CasperLabs Contracts SDK supports local testing of smart contracts. This tutorial will cover how to test the u64 key-value function. This can be easily adapted it for other types also.

In order to test the contract, the value must be stored, and the contract has to be deployed. Here is some sample code for these steps:

```
impl KVstorageContract{
   pub fn deploy() -> Self {

       // build the test context with the account for the deploy

        let mut context = TestContextBuilder::new()
            .with_account(TEST_ACCOUNT, U512::from(128_000_000))
            .build();

       // specify the session code & build the deploy
        let session_code = Code::from("contract.wasm");
        let session = SessionBuilder::new(session_code, runtime_args! {})
            .with_address(TEST_ACCOUNT)
            .with_authorization_keys(&[TEST_ACCOUNT])
            .build();
        context.run(session);
        let kvstorage_hash = Self::contract_hash(&context, KV_STORAGE_HASH);
        Self {
            context,
            kvstorage_hash,
        }
    }

    // query the contract hash after the deploy is complete

    pub fn contract_hash(context: &TestContext, name: &str) -> Hash {
        context
            .query(TEST_ACCOUNT, &[name])
            .unwrap_or_else(|_| panic!("{} contract not found", name))
            .into_t()
            .unwrap_or_else(|_| panic!("{} is not a type Contract.", name))
    }

    // store the u_64 value in the global state

    pub fn call_store_u64(&mut self, name: String, value: u64) {
        let code = Code::Hash(self.kvstorage_hash, "store_u64".to_string());
        let args = runtime_args! {
            "name" => name,
            "value" => value,
        };
        let session = SessionBuilder::new(code, args)
            .with_address(TEST_ACCOUNT)
            .with_authorization_keys(&[TEST_ACCOUNT])
            .build();
        self.context.run(session);
    }
}
```

Write Unit Tests

With these functions in place, it's possible to start writing tests for the contract.

```
mod tests {
    #[test]
    fn should_store_u64() {
```

(continues on next page)

(continued from previous page)

```
        const KEY_NAME: &str = "test_u64";
        let mut kv_storage = KVstorageContract::deploy();
        let name = String::from("test_u64");
        let value: u64 = 1;
        kv_storage.call_store_u64(name, value);
        let check: u64 = kv_storage.query_contract(&KEY_NAME).unwrap();
        assert_eq!(value, check);
    }

   // A test to check whether the value is updated
   #[test]
    fn should_update_u64() {
        const KEY_NAME: &str = "testu64";
        let mut kv_storage = KVstorageContract::deploy();
        let original_value: u64 = 1;
        let updated_value: u64 = 2;
        kv_storage.call_store_u64(KEY_NAME.to_string(), original_value);
        kv_storage.call_store_u64(KEY_NAME.to_string(), updated_value);
        let value: u64 = kv_storage.query_contract(&KEY_NAME).unwrap();
        assert_eq!(value, 2);
    }
}
```

Running Locally

It's possible to run the unit tests locally- if you have set up the contract using cargo-casperlabs. The steps to set up the SDK are in the guide.

```
cargo test -p tests
```

Deploying to the Testnet and Interacting with the Contract

There is a standalone python cli application that you can use for the kvstorage contract. When working with the testnet, create an account in CLarity and fund it using the faucet. Download the private key and use the key to sign the deployment. It's possible to create keys using the python client as well.

**Note, that this client was designed specifically for this contract. **

Deploy the Contract

The first step is actually to deploy the compiled wasm to the network, if you are using the python kv-client you must use the command `deploy_kv_storage_contract`. Once the contract is deployed, the client will retrieve the contract session hash as well as the blockhash where the contract is deployed.

```
python cli.py deploy_kv_storage_contract -f
↪"29acb007dfa4f92fa5155cc2f3ae008b4ff234acf95b00c649e2eb77447f47ca" -p "../../kvkey.
↪private.key" -c "../target/wasm32-unknown-unknown/release/contract.wasm" -b True
```

Invoke an Entry Point & Set a value

Once the contract is deployed, we can create another deploy, which calls one of the entry points within the contract. To call an entry point, you must first know the name of the entry point and the session hash, which we retrieved from the previous step. The kv-client, has four distinct commands to set key-values for u64, String, U512 and AccountHash.

```
python cli.py insert_u64 -f
↪"29acb007dfa4f92fa5155cc2f3ae008b4ff234acf95b00c649e2eb77447f47ca" -p "../../kvkey.
↪private.key" -s "0e82027493b88db434e85f82f6bcf48a30e0c1db15cf55fb87b73461b8aef20b" -
↪k "test" -v 1 -b True
```

Query the Contract On Chain

Contracts can be executed under different contexts. In this example, when the contract is deployed, it runs in the context of a `Contract` and not a `Session`. This means that all stored keys are not stored under the account hash, but within the context of the contract. Therefore when we query to retrieve the value under a key, we are actually querying `AccountHash/kvstorage_contract/<key-name>` and not just `AccountHash/<key-name>`.

Reading a value is simple enough, after you insert a value, the command retrieves the block hash under which the value, is stored. Using that block hash, and the `read-key` command you can easily retrieve and value that was previously stored under a named key.

```
python cli.py read_key -bh
↪"cb08a634c9bbea695fbd92e2ddbeec6fe6a374db807b36fea35077a9c1d720df" -p
↪"29acb007dfa4f92fa5155cc2f3ae008b4ff234acf95b00c649e2eb77447f47ca" -k "test"
```

More information on the kv-client, is available via the `--help` command. There is detailed information on each of the commands available with the client.

NOTE: The session hash is retrieved from the the chain by using a simple time delay, if processing the deploy takes longer than expected, it is likely that the kv-client will error out and not retrieve the session hash. In such cases, you can retrieve the session hash using the python casperlabs_client.

You must first find the block hash for the block that contains your deploy. Once you have the requisite block hash, then you can use the python shell to retrieve the session hash

```
Import casperlabs_client
client = casperlabs_client.CasperLabsClient('deploy.casperlabs.io', 40401)
Session_code = client.queryState(<block-hash>, <account-hash>, "kvstorage_contract_
↪hash","address')
Session_hash = session_code.cl_value.value.bytes_value.hex()
```