Gear Protocol technical paper

Abstract

This paper covers technical aspects of Gear Protocol, a new smart-contract engine for networks with untrusted code, whether it's a standalone bridged blockchains or Polkadot/Kusama parachains.

Gear allows to run Wasm programs (smart-contracts) compiled from many popular languages, such as Rust, C/C++ and more.

Gear smart-contract architecture utilizes advantages of an actor communication model, enables persistent memory for immutable programs and ensures very minimal, intuitive and sufficient api surface for blockchain context.

1. General outline

A short description of the Substrate blockchain framework is given in section (2).

An overview of Polkadot Network and Gear Protocol's role in it are described in section (3).

The distinctive characteristics and components of the Gear's network state are covered in section (4), details about how the Gear's network state evolves are provided in section (5).

For inter-process (or cross-contract) communications, an Actor model is used. How it is organized inside the Gear network state and what it brings to the table of decentralized computations is covered in section (6).

Using async/await pattern in asynchronous message communication is shown in (7).

Parallel message processing and efficient use of hardware resources ensured by Gear engine are described in (8).

A Wasm virtual machine used in Gear network and notable features introduced for VM implementation are covered in section (9).

The general Gear network internal workflow is briefed in (10).

Info about block producing and programs execution in the Gear blockchain network is shown in (11).

Balance transfers, Gas economy and DoS protection are covered in section (12).

Typical scenario of how Gear Protocol processes inputs from user interactions with program are outlined in section (13).

2. Substrate framework

The Substrate blockchain framework is an important component of the Polkadot network. It enables teams that want to create a new blockchain to not waste efforts on network implementation and consensus code by having to write it from scratch.

Substrate covers many aspects including - consensus mechanism (block finalization, validator voting system, fault-tolerance), networking layer (p2p connection, message sending and data replication functionality), full-node templates, database abstraction, client update mechanism through WASM (no hard forks) and other important modules. Technical description of those layers is beyond the scope of this paper. Refer to Substrate Documentation for more details.

Gear Protocol uses the Substrate framework under the hood. This helps to cover the most desired requirements for enterprise-ready decentralized projects - fault tolerance, replication, tokenization, immutability, data security and production-ready cross-platform persistent database.

Gear Protocol itself is implemented as a custom Substrate runtime introducing advanced native extensions (via host functions) for performance. Building a blockchain with Substrate allows Gear network to be deployed as a parachain on any compatible relay chain such as Polkadot and Kusama.

3. Polkadot Network

Polkadot is a next-generation blockchain protocol intended to unite multiple purpose-built blockchains, allowing them to operate seamlessly together at scale.

There are several components in the Polkadot architecture, namely: - Relay Chain - Parachains - Bridges

Relay Chain

Relay Chain is the heart of Polkadot, responsible for the network's security, consensus and cross-chain interoperability.

Parachains

Parachains are sovereign blockchains that can have their own tokens and optimize their functionality for specific use cases. Parachains must be connected to the Relay Chain to ensure interoperability with other networks. For this, parachains can pay as they go or lease a slot for continuous connectivity.

Bridges

A blockchain bridge is a special connection that allows Polkadot ecosystem to connect to and communicate with external networks like Ethereum, Bitcoin and others. Such connection allows transfer of tokens or arbitrary data from one blockchain to another.

Polkadot communication model and Gear role

Polkadot is not a smart contract platform by design, the relay chain itself does not not support smart contracts. The critical aspect of the Polkadot network is its ability to route arbitrary messages between chains. The use of these messages enables a negotiation channel between two parachains via sending asynchronous messages through it.

Gear Protocol's infrastructure for out-of-the-box dApp creation will allow any developer to build and run smart contracts easily, which takes much time, effort and cost on other platforms. But the cross-chain interoperability issues cannot be ignored. Becoming a parachain, Gear aims to improve the Polkadot/Kusama ecosystem, adjust to its needs, and move the dApp market forward.

Polkadot Relay Chain and Gear network ultimately speak the same language (asynchronous messages). Projects building on Gear Protocol can seamlessly integrate their solutions into the whole Polkadot/Kusama ecosystem.

Asynchronous messaging architecture allows networks powered by Gear Protocol to be an effective and easy-to-use parachains of Polkadot network:

1. Users deploy programs to the Gear network
2. Then, individual channels are established to popular parachains or bridges (there can be many and competing)
3. And the whole Gear parachain communicates through them

Such architecture allows driving the transition of the network between states and fits nicely to the whole network.

4. State

As any blockchain system, Gear network maintains distributed state. Runtime code compiled to Wasm becomes a part of the blockchain's storage state. Gear enables a defining feature of forkless runtime upgrades. The state is also guaranteed to be finalized if the finality gadget is used.

Storage state includes the following components:

- **Programs and memory** (includes program's code and its private memory)
- **Message queue** (global message queue of the network)
- **Accounts** (network accounts and their balances)

Programs and persistent memory

Programs (smart contracts) are first-class citizens in the Gear instance state.

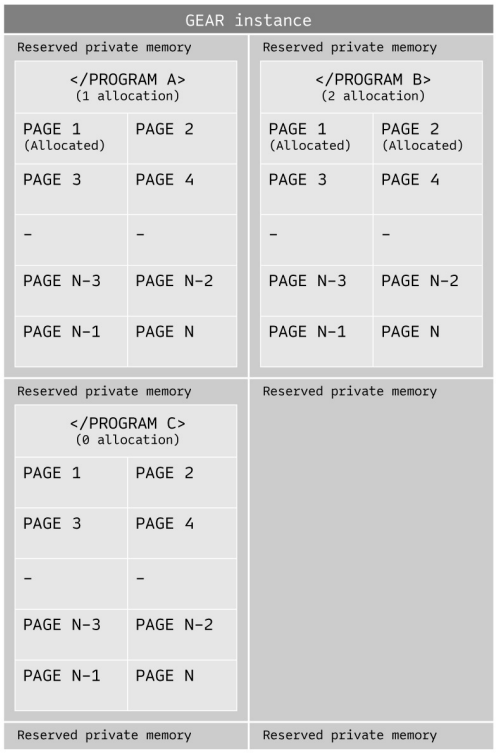
Program code is stored as an immutable Wasm blob. Each program has a fixed amount of individual memory which is reserved for a program during its initialization and persists between message-handling (so-called static area). Gear instance holds individual memory space per program and guarantees its persistence.

A program can read and write only within its own exclusively allocated memory space and has no access to the memory space of other programs. Individual memory space is reserved for a program during its initialization and does not require additional fee (included in the program initialization fee).

Programs can allocate more memory from the memory pool provided by a Gear instance. A program can allocate the required amount of memory pages in blocks of 64KB. Each additional memory block allocation requires a gas fee.

Each allocated memory block (64KB) is stored separately on the distributed database backend, but at the run time when a program accesses its memory, Gear node constructs continuous runtime memory and allows programs to run on it

without reloads.



Gear node uses lazy load technique so pages are brought into memory if the executing process demands them instead of loading all pages immediately. A program's state and its memory are saved each time after the program normally completes execution.

Utilizing a persistent memory is critical to ensuring the success of data-intensive dApps. The traditional approach of loading the program each time it needs to be addressed seems not optimized here. Decentralized applications with many-to-many relationships benefit from a persistent memory approach.

Message queue

Gear instance holds a global message queue. Using Gear node, users can send transactions with one or several messages to a particular program(s). This fills up the message queue. During block construction, messages are dequeued and routed to the particular program.

Accounts

For a public network, in order to be protected against DoS attacks a gas/fee for transaction processing is required. Gear provides a balance module that allows to store user and program balances and pay a transaction fee.

Regular balance transfer is performed inside the Substrate Balances module. Balance is transferred between users, program and validator accounts.

In addition to regular balance transfer, Gear network defines gas balance transfer that is used to reward validator nodes for their work and allows the network to be protected from DoS attacks.

In general, a particular Gear network instance can be defined as both permissioned and permissionless, public blockchain. In the permissioned scenario, no balance module is required.

5. State transition

Each system follows the rules according to which the state of the system evolves. As the network processes new input data, the state is advanced according to state transition rules. This input data is packed in atomic pieces of information called transactions.

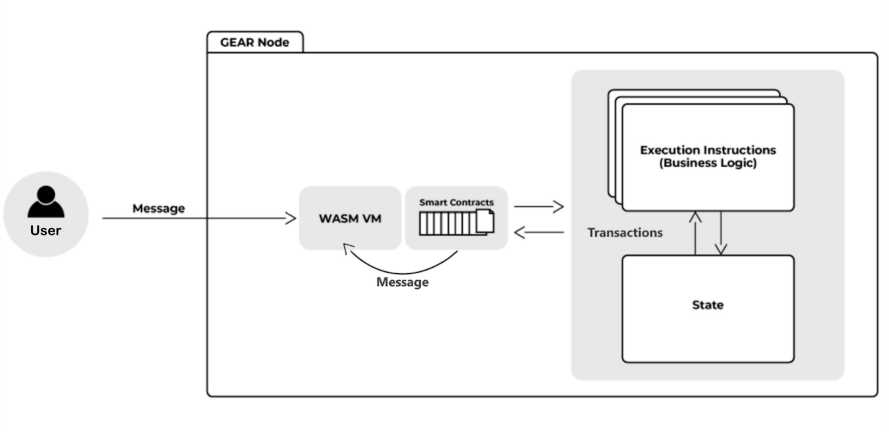
Gear nodes maintain and synchronize a transaction pool which contains all those new transactions. When any node (validator or not) receives a transaction, the node propagates the transaction to all connected nodes. For advanced reading how the transaction pool operates, refer to Substrate Documentation.

When a Gear validator node comes to produce a new block, some (or all) transactions from the pool are merged into a block and the network undergoes a state transition via this block. Transactions that were not taken in the last block remain in the pool until the next block is produced.

Gear Protocol supports the following types of transactions:

1. **Create a program** (user uploads new program(s) - smart-contracts)
2. **Send a message** (program or user fills the message queue)
3. **Dequeue messages** (validators (block producers) dequeue multiple messages, running associated programs)
4. **Balance transfers** (Gear engine performs user-program-validator balance transfers)

Message processing is performed in the reserved space of the block construction/import time. It is guaranteed that message processing will be executed in every block, and at least at some particular rate determined by current instance settings.



Create a program

Designated authorities (or any user for public implementation) of Gear network can propose a new program to be saved to the state. For public networks, a balance associated with a program is also provided. This new balance then constitutes the initial balance (Existential Deposit).

Send a message

End-users interact with programs via sending messages to the Gear network. Messages sent to the Gear network fill up the global message queue. This queue can be viewed as a runtime-driven transaction queue but with the guarantee that any message accepted into it will eventually be processed. Putting a message in the queue is not free and therefore a message is guaranteed to be dispatched.

Programs also exchange messages with each other. The result of the received message can be another message (reply) addressed to another program or a user or a designated behavior to be used for the next message it receives. A program can also send a message the execution result of which will be a creation of another program.

Dequeue messages

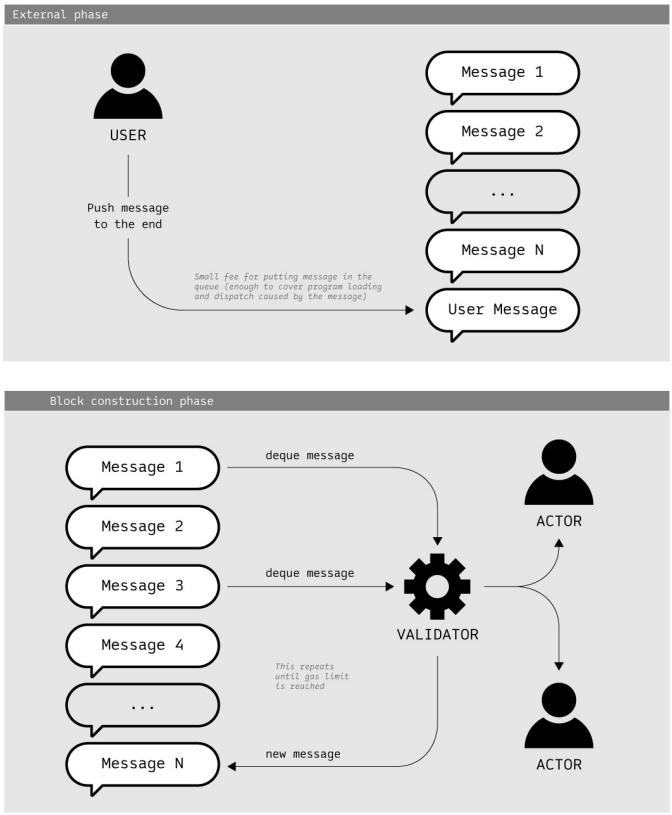
Validators can choose which messages to dequeue when it's their turn to produce the next block. It eliminates the need of each particular validator to maintain the full memory state. Dequeuing occurs only at the end of each block. During dequeuing, new messages can be generated. They can also be processed in this phase, but also can stay in the queue for the next block (and another validator).

Balance transfers

Regular balance transfers are performed inside the Substrate Balances module. Refer to the next chapter for more details.

Messages, blocks and events lifecycle

The picture below illustrates eternal lifecycle of Gear machinery. As actor model for communications dictates, nothing is shared, there are only messages:



6. Actor model for communications

One of the main challenges of concurrent systems is concurrency control. It defines the correct sequence of communications between different programs, and coordinates access to shared resources. Potential problems include race conditions, deadlocks, and resource starvation.

Concurrent computing systems can be divided into two communication classes:

- Shared memory communication — when concurrent programs communicate via changing the content of shared memory locations.
- Message passing communication — implies concurrent programs communication via messages exchanging. Message-passing concurrency is easier to understand than shared-memory concurrency. It is usually considered a more robust form of concurrent programming.

Typically, message passing concurrency has better performance characteristics than shared memory. The per-process memory overhead and task switching overhead is lower in a message passing system.

There are plenty of mathematical theories to understand message-passing systems, including the Actor model.

For inter-process communications, Gear Protocol uses the Actor model approach. The popularity of the Actor model has increased and it has been used in many new programming languages, often as a first-class language concept. The principles of the Actor model is that programs never share their state and just exchange messages between each other.

Actors post a message that appears at the end of the message queue. Some small fee is taken for putting messages in the queue, enough to cover program load and dispatch caused by the message. Messages are dequeued by validator nodes and this repeats until gas_limit is reached.

While in an ordinary Actor model, there is no guarantee on message ordering, Gear Protocol provides extra guarantees that the order of messages between two particular programs is preserved.

Using the Actor model approach provides a way to implement Actor-based concurrency inside program (smart-contract) logic. This can utilize various language constructs for asynchronous programming (for example, `Futures` and `async/await` in Rust).

7. Async-await support

Unlike classes, actors allow only one task to access their mutable state at a time, which makes it safe for code in multiple tasks to interact with the same instance of an actor.

Asynchronous functions significantly streamline concurrency management, but they do not handle the possibility of deadlocks or state corruption. To prevent deadlocks or state corruption, async functions should avoid calling functions that may block their thread. To achieve it, they use an *await* expression.

Currently, the lack of normal support of *async/await* patterns in the typical smart contract languages and frameworks brings a lot of problems for smart contract developers. Actually, achieving better control in a smart contract program flow is actually more or less possible by adding handmade functions. But the problem with many functions in a contract is that one can easily get confused — which function can be called at which stage in the contract's lifetime.

Gear Protocol natively provides arbitrary *async/await* syntax for any programs that simplifies development and testing and reduces the likelihood of errors in smart contract development. Gear Protocol's API also allows synchronous messages if the logic of the program requires it.

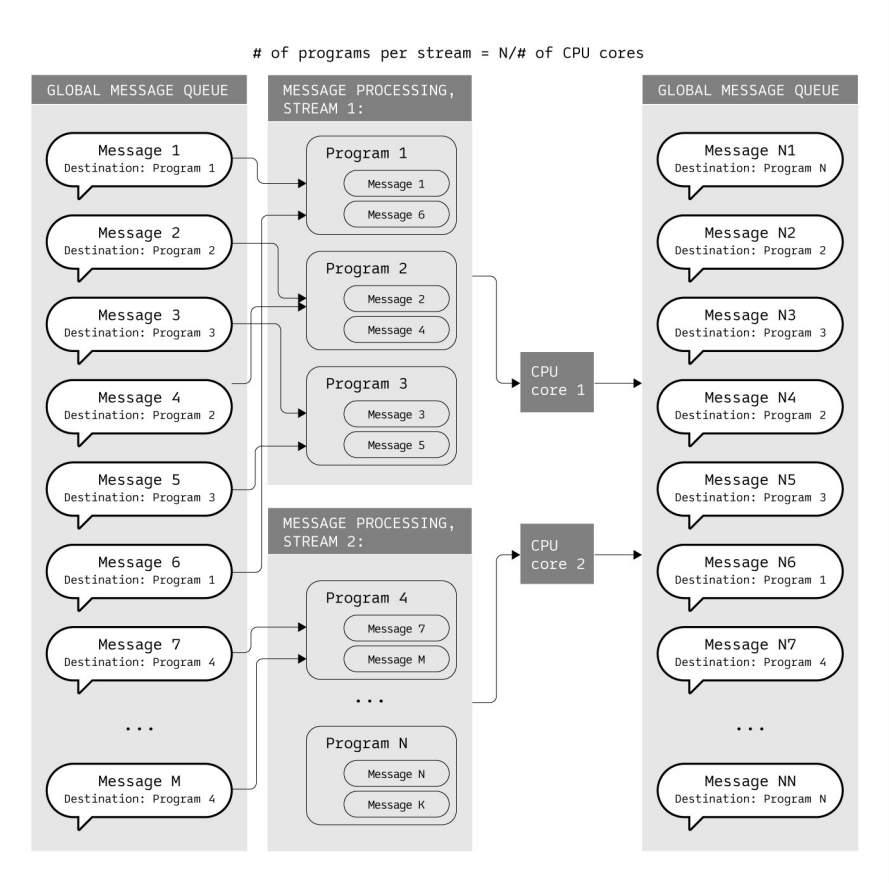
8. Memory Parallelism

Individual isolated memory space per program allows parallelization of message processing on a Gear node. Messages can be sorted into multiple streams in order to be processed in parallel. The number of streams is configured for the entire network and can be equal the number of CPU cores for typical validator nodes. Each stream contains messages intended for a defined set of programs. It relates to messages sent from other programs or from outside influences, for instance, user's transactions.

For example, given a message queue containing messages targeted to 100 different programs and Gear node running on a network where 2 streams of processing are configured. Gear engine uses a runtime-defined number of streams, divides total amount of targeted programs to number of streams and creates a message pool for each stream (50 programs per stream to be precessesed by 2 CPU cores of a validator node).

Programs are distributed to separate streams and each message appears in a stream where its targeted program is defined. So, all messages addressed to a particular program appear in a single processing stream.

In each cycle, a targeted program can have more than one message and one stream can process messages for multiple programs. The result of message processing is a set of new messages from each stream that is added to the global message queue, then the cycle repeats itself. The resultant messages generated during message processing are usually sent to other addresses (typically returning to the origin or to the next program).



9. Virtual machine (Wasm)

WebAssembly (abbreviated Wasm) is a standard developed by W3C Community Group, which defines a low-level binary code format for executable programs and can run as a standalone virtual machine. It has been designed and implemented in collaboration between all major competitors in its space.

Designed along with a complete mathematical and machine-verified formalisation, WASM significantly increases the speed of transactions, which contributes to improving efficiency. On top of this, WASM supports smart contracts that are written in different languages, which means that anyone can take an existing program or write a new one in a convenient language and compile it for execution in the Wasm virtual machine. This significantly increases application inclusivity compared to solutions based on domain specific languages.

Briefly, Wasm has the following advantages:

- Native speed. As it translates to actual hardware instructions.
- Portable. It can run on any actual hardware.
- Safe. Properly validated Wasm program cannot leave sandbox (guaranteed by specification).

The key to realizing Polkadot's vision of cross chain interconnection was found in Wasm: a generic and abstract machine specification that could mediate between blockchains with different runtimes (the chains' application logic).

Gear network instance uses Wasm under the hood. Any program written with a Gear Protocol is in Wasm format.

Security consideration

Wasm itself does not provide ambient access to the computing environment in which code is executed. Any interaction with the environment, such as I/O, access to resources, or operating system calls, can only be performed by invoking functions provided by the Wasm implementation that's embedded into a host environment and imported into a Wasm module.

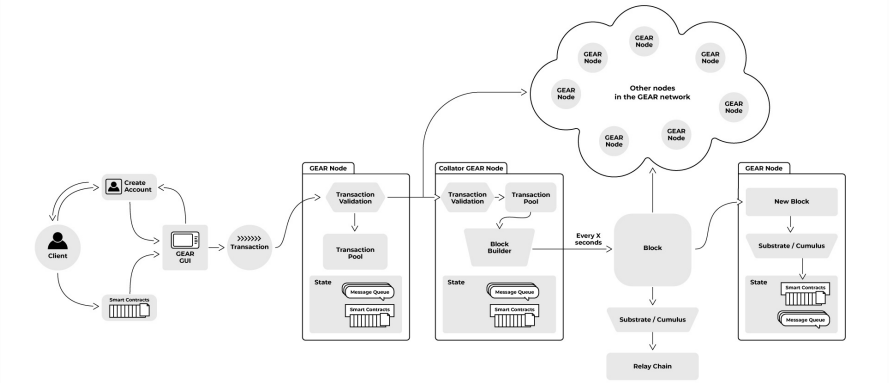
Such implementation defines how loading of modules is initiated, how imports are provided and how exports can be accessed. It takes responsibility for security policies suitable for a respective environment by controlling or limiting which functional capabilities it makes available for import.

Because Wasm is designed to be translated into machine code running directly on the host's hardware, it is potentially vulnerable to side channel attacks on the hardware level. When this is a concern, an embedded Wasm implementation may have to put suitable mitigations into place to isolate Wasm computations.

Researchers have previously reported possible vulnerabilities related to threads with shared memory. Gear is not the subject to these concerns by-design. As a concurrent computing system, Gear uses message-passing communication (Actor model) instead of shared-memory communication model. This mitages such security concerns.

10. The internal flow

This scheme represents on a high level of how the Gear network works:



Following decentralization principles, the Gear network consists of many computing devices - nodes that share a single global state.

A smart contract developer (Client) uploads a program (compiled to Wasm) to the network via a user interface that's provided by Gear - https://idea.gear-tech.io. To do that, a Client must have an account connected to Gear network and enough funds on it to pay a transaction fee.

When a user uploads a program, the interface sends a transaction to an arbitrary network Node. This node then validates the transaction and, if everything is ok, puts it in the transaction pool where it is shared across all nodes within the network.

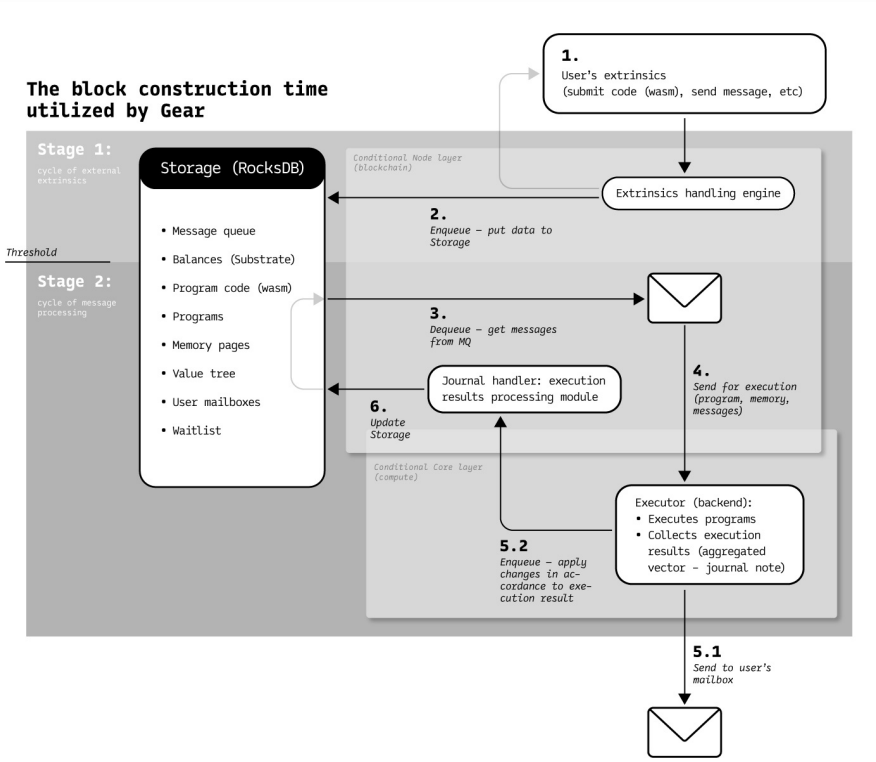
In Gear network, actors (users and programs) send transactions with one or several messages to each other. This fills up the global message queue. A special node in the network (collator) takes transactions to be processed in the next block. During block construction, messages are dequeued and processed within the reserved space of the block construction time. The suggested block is validated by other nodes and finalized in accordance with the consensus mechanism. The message processing results are saved in the state.

11. Program execution and block producing

Gear Protocol uses a standard Substrate Executive module that Initializes the block, executes extrinsics and finalizes the block. For more details refer to The execution of the Substrate runtime.

But Gear's value and uniqueness lies precisely in the ability of executing arbitrary Wasm smart contracts with parallelism.

This shema outlines Gear's part in the block construction workflow:



The process can be conditionally divided into two separate parts, both a blockchain (Node layer) and a computational layer (Core layer) where compute operations are performed.

The special mechanism on the Node layer runs cycle that collects extrinsics (related to new programs initialization or messages between actors) and fills the Storage that consists of:

- Message queue
- Submitted program code (wasm)
- Programs' memory pages (program state)
- Programs (program Ids, associated wasm and related memory pages, program state (is_active/on_pause/is_initialized))
- Actors balances
- Value tree (gas_spent/remained during subsequent messaging between actors)
- User's mailboxes
- Waitlist (messages on_pause in accordance to async/await logic)

As soon as all extrinsics for the current block have been taken from the transaction pool, a cycle of alternate retrieving messages, from the message queue and the program to which the message is addressed (as well as other related info such as balances, value tree etc), is started.

The executional component on the Core level receives a program and messages, initializes memory, arranges memory pages accordingly and using all the related information executes the program. As a result, the Executor gathers the execution results (new messages, errors etc), combines aggregated data and applies changes accordingly. Depending on the result, it can be applied as a message sent to a user's mailbox (reply) or returned back to the Node level from where it gets into the Storage. And the cycle repeats until the time allocated for the block runs out, or all messages are processed.

The way Gear node executes Wasm programs and processes messages having a conditional Node and Core levels fits well for a blockchain networks built on Gear Protocol. Moreover, this logic is written the way it can be applied not just with blockchains, but also reused for any other IT solutions.

12. Gear token economy and balance transfers

The token economy of networks powered by Gear Protocol is built around two main concepts: the base asset (Currency) and Gas.

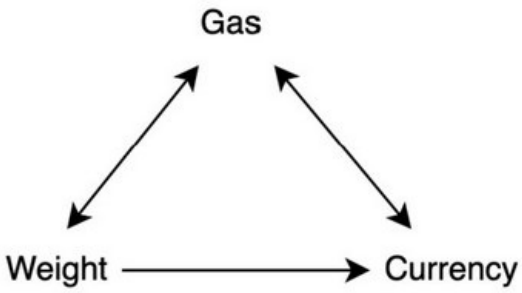
The base asset in the system is represented by the Gear network native token which is implemented by the standard Substrate Balances module. The latter, as the underlying currency, takes care of transfers, minting, burning etc.

External users' accounts are derived from the public keys that are generated by one of the supported cryptographic algorithms, which means that users can send on-chain transactions by signing them with their corresponding private key. Other actors (programs) may have accounts in the balances pallet - as long as there is some currency amount attributed to this actor. However, these account IDs are not derived from any public key, therefore there is no corresponding private key, so no extrinsics can be sent on behalf of these accounts.

The purpose of introducing Gas is twofold. Firstly, it limits the block time. Since programs in Gear are written in a Turing-complete language they may incur very long or even endless computations. Gas limit set for every external message helps to prevent potential DoS attacks. Secondly, it provides a mechanism for calculating message processing fees. The complexity of computations in Substrate is represented by the notion of Weight (1 weight == 10^-12 second). Gas in Gear is, in one sense, an extension of Weight as far as computations duration is concerned.

On the other hand, Gas extends the notion of Currency because it eventually defines a validator's award for processing messages.

The diagram below represent the idea that the three concepts are pairwise



equivalent:

Connecting lines mean that units of one entity can be converted into units of another entity. For instance, Gas can be purchased for Currency and then redeemed back and Weight is converted into Currency by Substrate's transaction-payment pallet (WeightToFee associated type).

Speaking categorically, this diagram must always commute in order to prevent arbitrage. For instance, Weight can be converted to Balance, and it is equal to a certain amount of Gas (in trivial case, the same nominal amount). At the same time Gas can be directly exchanged with Balance. We should make sure all possible conversions from one entity to another add up, regardless of the path taken along the edges of the diagram. If this property doesn't hold, it can potentially lead to an attacker taking advantage of the system

The simplest way to guarantee the requirement for this diagram to commute, is to set all the conversion functions to constants; more specifically, the Identity transform. It means we treat a unit of Gas as if it was a unit of Weight (that is, 10^-12 of a second) and it (nominally) costs the same as Currency in terms of base tokens. The latter can also always be arranged because we are free to choose the scale and precision of the token accordingly.

Gas limit as block time

As mentioned above, in Substrate computations duration is represented in units of Weight. 10^12 weight is 1 second. Naturally, it means that each block has a certain Weight capacity to fit in as many transactions as possible so that the overall computational cost doesn't exceed the limit. Each extrinsic's weight is roughly known upfront through benchmarking so that the block producer knows, with certain degree of accuracy, how many of those it can process in one block.

Reasoning about external transactions (extrinsics) is easy. For extrinsics that can be rather costly, and at the same time not affect the Message Queue at all, the weight is known upfront (for example, *set_code()*).

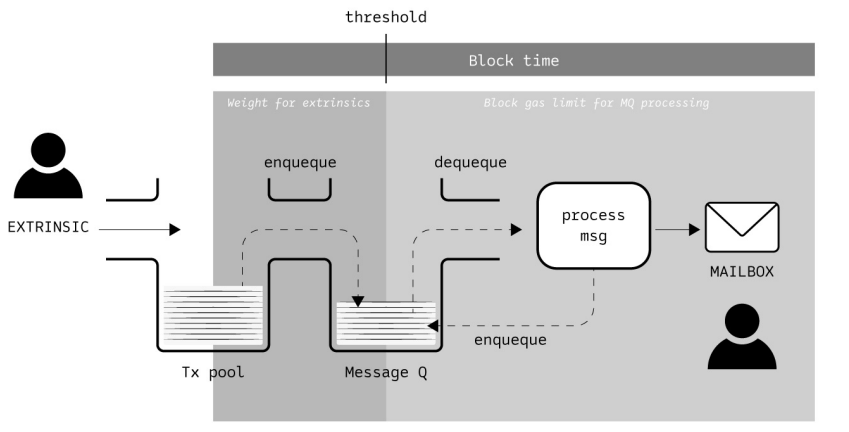
Things get more complicated when we add Gear-specific actions into the picture. The extrinsics can create a Message based on the input data and push it into the Message Queue. The required weight may still vary if a user wants to submit a very large program, but the weight will linearly depend on the extrinsic's size and is taken into account. But once it gets to processing messages, we have a problem. We won't know upfront how much time (in terms of Gas or Weight, which is, essentially, the same thing) a Message might take to process.

To mitigate this we introduce a dynamic remaining block gas limit (*gas_allowance*), which is updated every time a Message has been processed.

Furthermore, to decide whether we should put another Message to the current block, we take the next Message from the Queue independently on its *gas_limit* and during the processing if the current *gas_allowance* has been exceeded, return it back to be taken in another block.

It might be useful to think about this system as a pair of connected pools, where users' extrinsics get into the transaction pool (Tx pool) when a new block is being created by the block producer, who'll take as many extrinsics from the pool as possible to avoid exceeding the pre-configured time share allocated for extrinsics' processing.

This phase results in a number of new Messages having been pushed to the Message Queue. After those messages have already been added to the Queue, they start being popped and processed one by one. Upon processing, the result may be sent to the users' mailbox, or a number of newly spawned messages can go back to the Queue:



In our initial version, the threshold that determines the relative size of the total block timeshare allocated for each phase is static and set to 25%. However, in our later releases it will be made dynamic to account for the size of Tx pool and Message Queue to avoid overpopulation. If the Message Queue stays inflated while the Tx pool is empty, the threshold can be reduced and vice versa.

Balance transfer

Regular balance transfer is performed inside the Substrate Balances module. Balance is transferred between user, program and validator accounts.

In addition to regular balance transfer, the Gear network defines a gas balance transfer that is used to reward validator nodes for their work and allows the network to be protected from DoS attacks.

Gear node charges gas fee during message processing. The message processing algorithm is described below in details.

All interactions inside the Gear network are done via messaging. Messages on Gear Protocol have common interface with the following parameters:

- *source account*,
- *target account*,
- *payload*,
- *gas_limit*,
- *value*

There are five types of messages used in Gear's network:

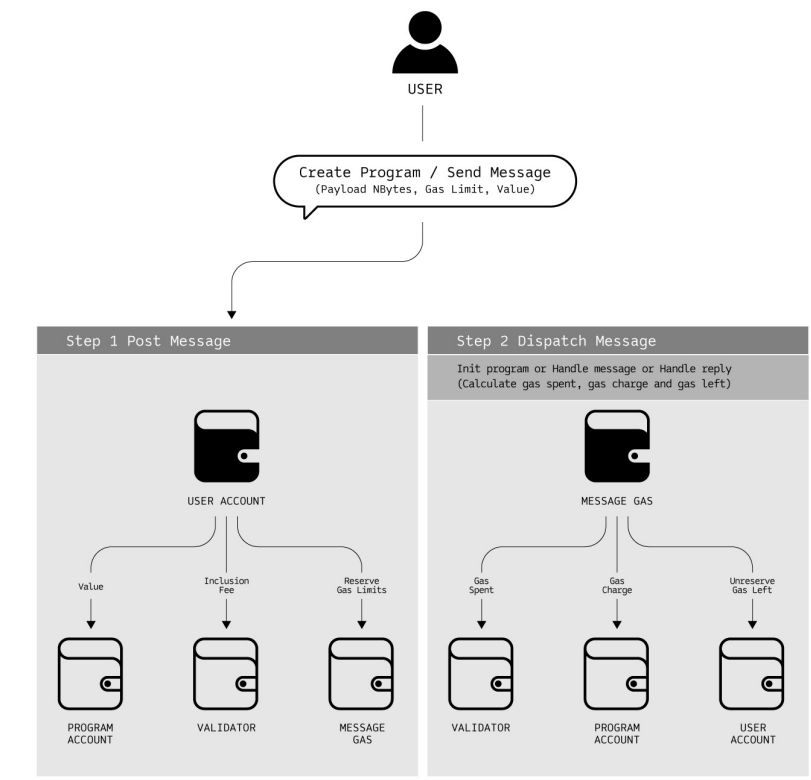
1. A special message from the user to upload a new program to the network. Payload must contain a Wasm file of the program itself. Target account must not be specified - it will be created as a part of Post message processing.
2. From user To program
3. From program To program
4. From program To user
5. From user To user

The last parameter of the send message function is a value to be transferred to a target account. In the special message of the initial program upload (#1), value will be transferred to a balance of the newly created account for the program.

Before the message processing step, some funds are reserved on the message initiator's account for paying a small processing fee. This is the standard inclusion fee for the Substrate framework and its size depends on different factors. Refer to Substrate Documentation for details.

Message processing consists of two steps:

1. Post message
2. Dispatch message



Post message

The Gear network tries to post a message into the message queue transaction pool. To do this, the local validator node verifies that the message initiator account has enough balance to cover sending of *value* and *gas_limit*.

For *Upload program* message type, validators verify that the message's *gas_limit* does not exceed the gas limit per block.

For messages between actors (#2-5, From - To types), the validators verify that the destination program is initialized (for messages addressed to a program) and that the message's *gas_limit* does not exceed the gas limit per block.

Gear Protocol introduces a *gas_limit* per block in order to ensure the block production/validation time doesn't stretch out indefinitely. Therefore, if a message (of any type) declares the gas_limit greater than that of an entire block, it won't be allowed in the queue.

After verification, if everything is ok, a validator with *block producer* role posts the transaction into the block, transfers *value* to the target account, transfers a small processing fee to the validator account and reserves a fee equal to *gas_limit* on a message initiator account (User's account).

Dispatch message

A message is dispatched by the network in the following way:

When uploading a program, the user specifies *gas_limit* and optionally *value* to be transferred to the program account. *gas_limit* serves as a maximum amount of gas to be spent on program initialization. For messages sent from programs, gas_limit is calculated automatically.

When a program is being initialized by a Gear node it consumes gas for both memory page allocation and per CPU instructions. It increments the internal counter - *gas_spent*.

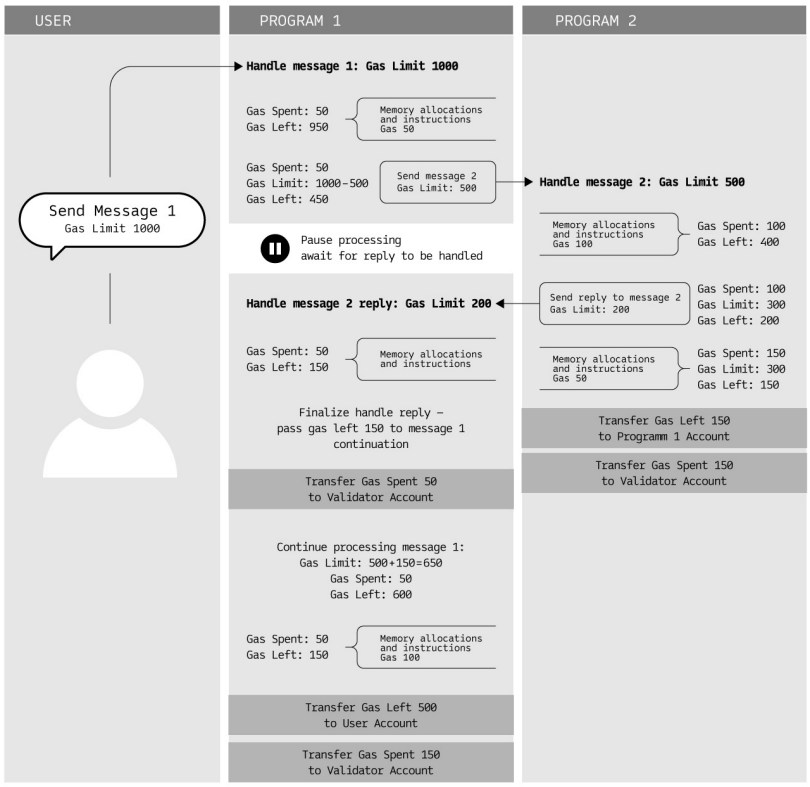
Gear nodes check that each increment of *gas_spent* value is lower than *gas_limit* that's specified in the initialization message. If it is more, then it stops program execution. This way *gas_limit* serves as a safeguard for user balance so no program will consume more than the user expects.

After the program has been initialized, *gas_spent* is charged against the user account towards the validator's account. The difference between *gas_limit* and *gas_spent* is *gas_left* and its value is unreserved on the message initiator account.

Gas fee is linear - 64000 gas per allocated memory page of size 64KB and 1000 gas per instrumented Wasm instruction.

For standard messages where the target is a program, all of the above gas consumption rules apply, as well as additional memory rent fees that are charged via an incrementing *gas_spent* value. This memory rent fee is similar to a memory allocation fee, but the price is lower than the one for initial allocation - 1000 gas per already allocated page. Memory rent is charged because each node has to load/save program memory from/to network state.

The picture below shows more detailed example of balance transfers:



Messages sent to a user appear in the user's mailbox. Keeping a message in the mailbox costs a fee. In order for the message to remain in the mailbox, the initiator of the message must specify the sufficient amount of gas_limit, some amount of which will be consumed each block.

The message remains in the mailbox until the value is claimed or message replied by the user or until the gas_limit specified for the initiator's message runs out. Once gas_limit has exhausted, the message is removed from the mailbox.

Not all incoming messages can be processed at one cycle and appear in a single block. It happens when gas required for all message producing exceeds block gas limit. In this case messages can appear in the next block (considering that more new messages are coming).

Gear nodes make a choice of which transactions with messages will end up putting messages in the queue. Messages from transactions with the highest fee are taken first. In this case, messages from transactions with the lowest fee can be delayed or even never end up in the processing queue. Deviation from the standard transaction handling algorithm may lead to unwanted economic circumstances.

A sequence of messages

Message initiator (user) can cause a sequence of messages where destination programs send messages down to other programs. A program can send messages to other programs in the future or during initialization.

gas_left available for further messages is a difference between *gas_limit* specified for the previous message and *gas_spent* for the previously processed message. *gas_spent* for all further messages is charged against the initiator's account towards the validator's account.

The *gas_limit* parameter in each next message can be either equal to entire *gas_left* from the previous message or be a custom value, but cannot be more than *gas_left* from the previous message (otherwise message processing will fail).

In a sequence, a source program can wait for reply from the program destination. Each wait during a block time costs gas that is charged against the initiator's account towards the validator's account.

This also relates to messages addressed to users and are wait in the mailbox. If a user replies to a message before *gas_limit* for the message has been exhausted, remaining *gas_left* of the last message is transferred to this user's balance. *gas_left* from the previous messages return to the message initiator's balance. If *gas_spent* has exceeded *gas_limit* while a message waits in the mailbox, the message is removed from the mailbox and the state.

System signals and gas reservation

The Gear Protocol ensures system and program's state consistency via introducing special handling mechanisms for potential issues and corner cases.

Gear actors have three common entry points - `init`, `handle`, `handle_reply`. Another special system entry point introduced by the Gear Protocol is `handle_signal`. It allows the system to communicate with programs if it is necessary to notify (signal) that some event related to the program's messages has happened. Only the system (Gear node runtime) can send signal messages to a program.

First of all, it can be useful to free up resources occupied by the program. A custom async logic in Gear implies storing `Futures` in a program's memory. The execution context of `Futures` can occupy some significant amount of memory in case of many futures. When a program sends a message and waits for a reply to be waked, the reply can not be received. So there might be the case that if the initial message in the waitlist runs out of gas or the gas amount is not enough to properly finish the execution, the program's state will be rolled back and `Future` will never be freed.

In this case, `Futures` remain in memory pages forever. Other messages are not aware about `Futures` associated with other messages. Over time, `Futures` accumulate in the program's memory so eventually a large amount of Futures limits the max amount of space the program can use.

In case a message has been removed from the waitlist due to gas constraints, the system sends a system message (signal) that is baked by an amount of **reserved gas**, which informs the program that it's message was removed from the waitlist. Based on this info, a program can clean up its used system resources (`Futures`).

For programs written using the Gear Protocol's `gstd` library, such signals can be sent to programs automatically under the hood when applicable. If a smart contract developer implements a program using `gcore` or Gear's syscalls, then such signals should be considered in the program's code explicitly.

With any other messages, the system message (signal) execution consumes gas. The Gear Protocol introduces a **gas reservation** feature which actually can be useful not only for system signals.

A program developer can provide a special function in the program's code which takes some defined amount of gas from the amount available for this program and reserves it. A reservation gets a unique identifier that can be used by a program to get this reserved gas and use it later. Programs can have different executions, change state and evaluate somehow, but when it is necessary, a program can send a message with this reserved gas instead of using its own gas.

The `gstd` library also provides a separate function for reserving gas specifically for system signal messages. It cannot be used for sending other regular cross-actor messages.

If a signal message appears, it uses gas specifically reserved for such kinds of messages. If no gas has been reserved for system messages, they are just skipped and the program will not receive them.

If gas has been reserved but no system messages occur within some defined period (amount of blocks), then this gas returns back from where it was taken. The same relates to gas reserved for non-system messages - gas returns back after a defined number of blocks or by program's command.

The important point here is that gas reservation can be used not only for system signals, but also for sending messages with someone else's gas. It can be useful in the case of big executions, the result of which should be a final message that will be sent using reserved gas from the initial message instead of using gas from the last sent message.

For example, let's consider some arbitrary game implementation. A user starts a game (sends a message to a program) and a program reserves some amount of gas from this message. Then other actors start sending messages to the program (registration of participation in the game). Once all is done (by timeout or number of participants threshold reached), a program sends a final message automatically (the game results) using reserved gas without the need for the user to manually interact with the program. It allows the program not to hang in the waitlist waiting for a reply and spending gas (being in the whitelist consumes gas every block). In this example, synchronous implementation can be more effective from the gas consumption perspective.

13. Typical scenario

Let's take a look how Gear machinery works when running imaginary program.

Gear allows any general-purpose language program compiled to Wasm to run, for example this `capacitor.rs`:

```
static mut CHARGE: u32 = 0;

static mut LIMIT: u32 = 0;
```

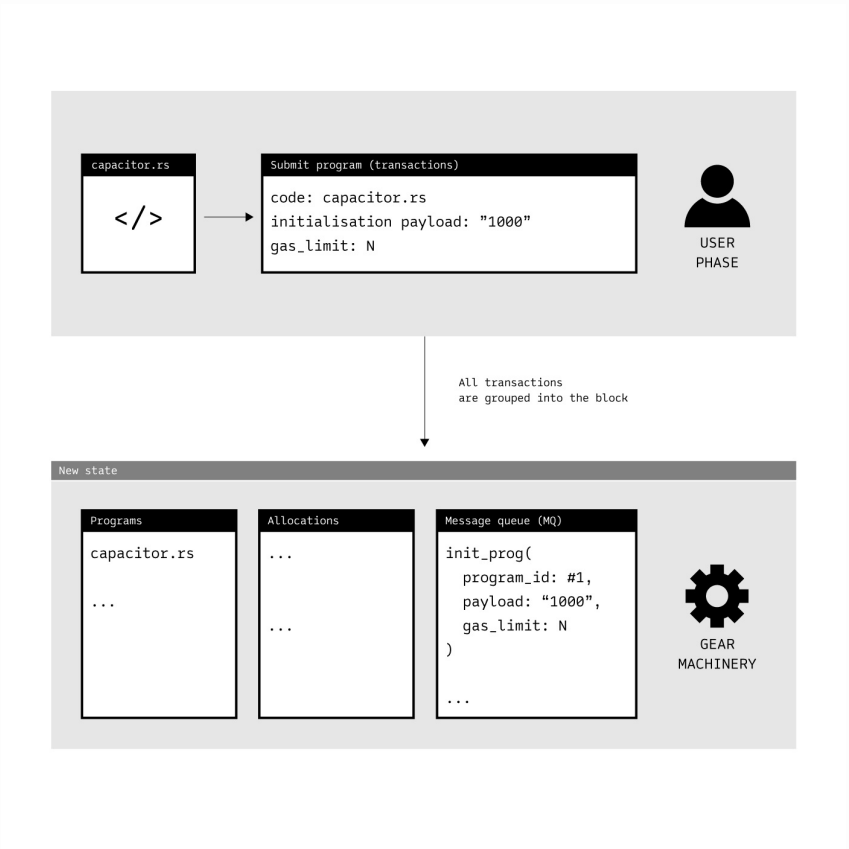
```rust
static mut DISCHARGE_HISTORY: Vec<u32> = Vec::new();

#[no_mangle]
pub unsafe extern "C" fn handle() {
    let new_msg = String::from_utf8(msg::load()).expect("Invalid
        message: should be utf-8");

    let to_add = u32::from_str(&new_msg).expect("Invalid number");

    CHARGE += to_add;

    if CHARGE >= LIMIT {
        DISCHARGE_HISTORY.push(CHARGE);
        msg::send(0.into(), format!("Discharged: {}",
        CHARGE).as_bytes(), 1000000000).unwrap();
        CHARGE = 0;
    }
}
```

For full example, refer to our test crate.

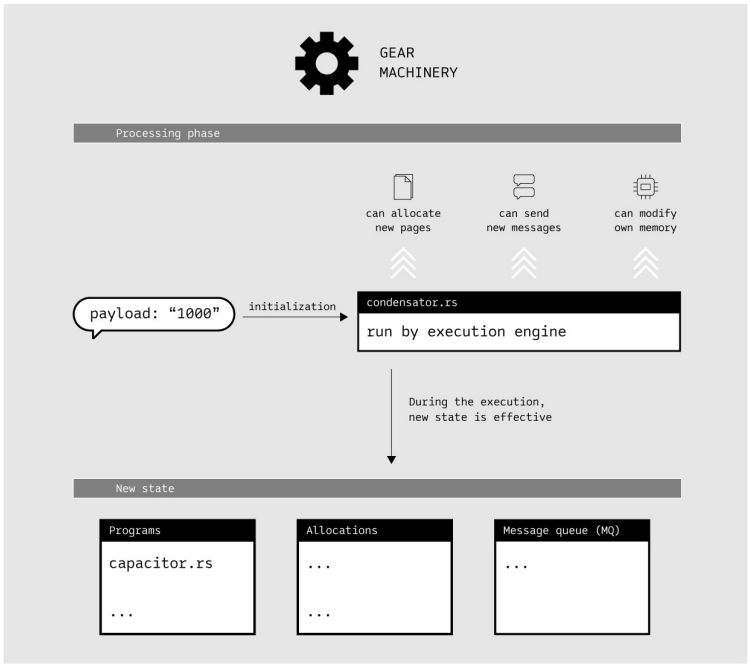
This is a simple program that “charges” with incoming messages and “discharges” when total amount of “charge” exceeds some limit provided in initialization.

Let’s take a look on series of diagrams illustrating how user creates such a program and then interacts with it:

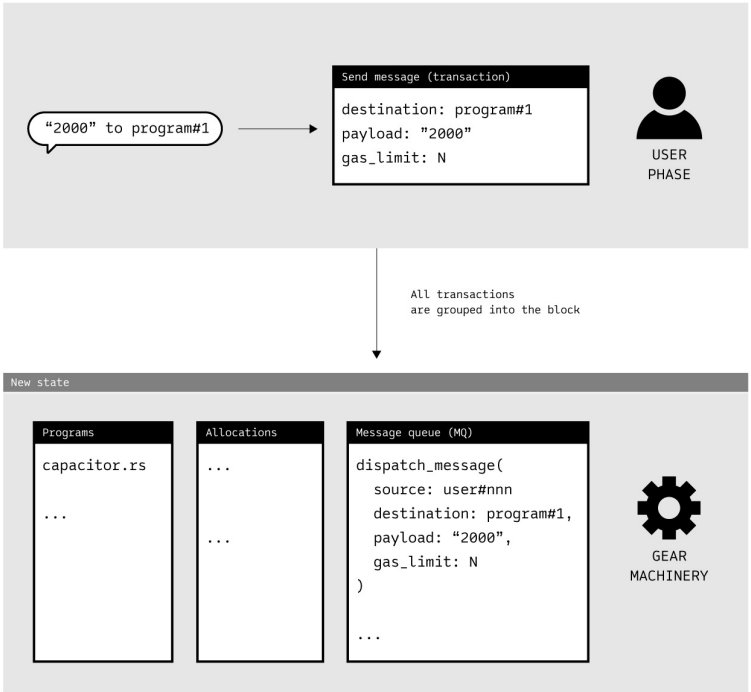
1. This is how the program is created - user just sends a transaction with program and initialization arguments:



2. Gear then processes this new input:



3. Once created, program can receive messages. For example, user can send “charge” of 2000 to it:



4. Gear then process this new input: