Gear technical

Abstract

This paper covers technical aspects of Gear, a new Polkadot/Kusama parachain and smart-contract engine for networks with untrusted code.

Gear provides an engine for Polkadot/Kusama parachains and standalone bridged blockchains. Gear allows to run WebAssembly programs (smart-contracts) compiled from many popular languages, such as C/C++, Rust 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 Substrate blockchain framework is given in section (2).

An overview of Polkadot Network and Gear'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).

Balance transfers, Gas economy and DoS protection are covered in section (6).

For inter-process (or cross-contract) communications, actor model is used. How it is organized inside the Gear state and what it brings to the table of decentralized computations is covered in section (7).

Using async/await pattern in asynchronous message communication is shown in (8).

Parallel message processing and efficient use of hardware resources ensured by Gear engine are described in (9).

Typical scenario of how Gear processes inputs from user interactions with program are discussed in section (10).

A WebAssembly virtual machine used in Gear network and notable features introduced for VM implementation are covered in section (11).

Some of the use cases where Gear can be used most effectively are given in section (12).

2. Substrate framework

Substrate blockchain framework is an important component of Polkadot Network. It allows every team creating a new blockchain not to waste efforts for implementation of the networking and consensus code from scratch.

Substrate covers both - replication (networking layer) and fault-tolerance (consensus mechanism). Technical description of those layers is beyond the scope of this paper. Refer to Substrate Documentation for more details.

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

Gear itself is implemented as a custom Substrate runtime. It also introduces advanced native extensions (via host functions) for performance.

Using Substrate allows to quickly connect Gear instances as parachain/parathread into the Polkadot/Kusama network.

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

The critical aspect of Polkadot network is its ability to route arbitrary messages between chains. Using this messages is as simple as it can be: negotiate the channel between two parachains and send asynchronous messages through it.

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

Asynchronous messaging architecture allows Gear to be an effective and easy-to-use parachain of Polkadot network:

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

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

4. State

As any blockchain system, Gear maintains distributed state. Runtime code compiled to WebAssembly becomes part of the blockchain's storage state.

Gear ensures one of the defining features - forkless runtime upgrades. The state is also guaranteed to be finalized if finality gadget is used.

Storage state components

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

Programs

Programs 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 memory which persists between message-handling (so-called static area).

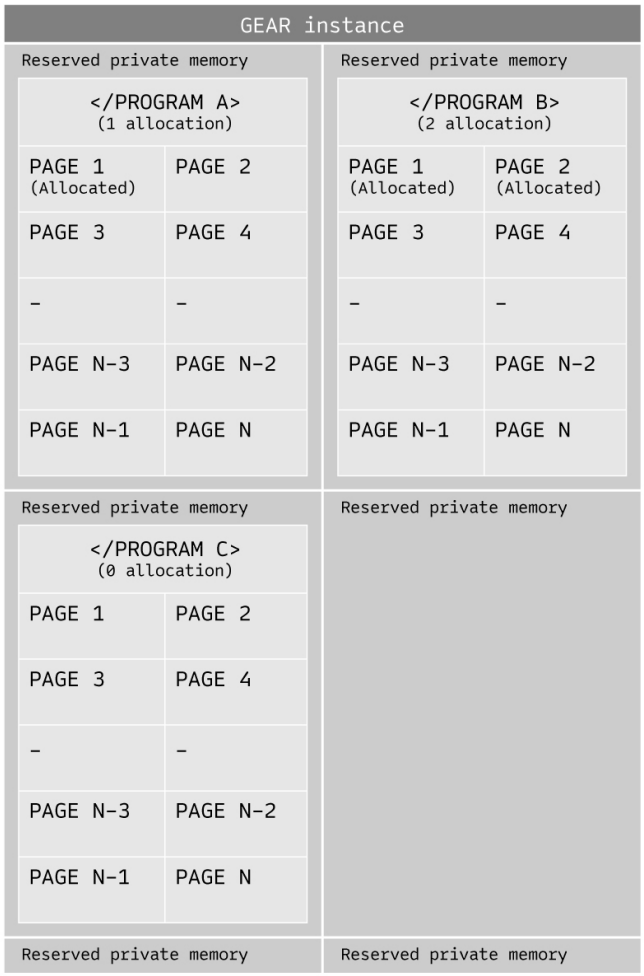
Programs can allocate more memory from the memory pool provided by a Gear instance. A particular program can read and write only within exclusively allocated to it memory.

Memory

Gear instance holds individual memory space per program and guarantees it's persistence. A program can read and write only within its own 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).

A program can allocate the required amount of memory in blocks of 64KB. Each memory block allocation requires gas fee. Each page (64KB) is stored separately on the distributed database backend, but

at the run time, Gear node constructs continuous runtime memory and allows programs to run on it without reloads.



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, protection against DOS attacks always requires gas/fee for transaction processing. Gear provides a balance module that allows to store user and program balances and pay a transaction fee.

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 producing.

Gear supports the following types of transactions:

1. **Create a program** (user uploads new programs - smart-contracts)
2. **Send a message** (program 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 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 and as a result, send messages to 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.

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).

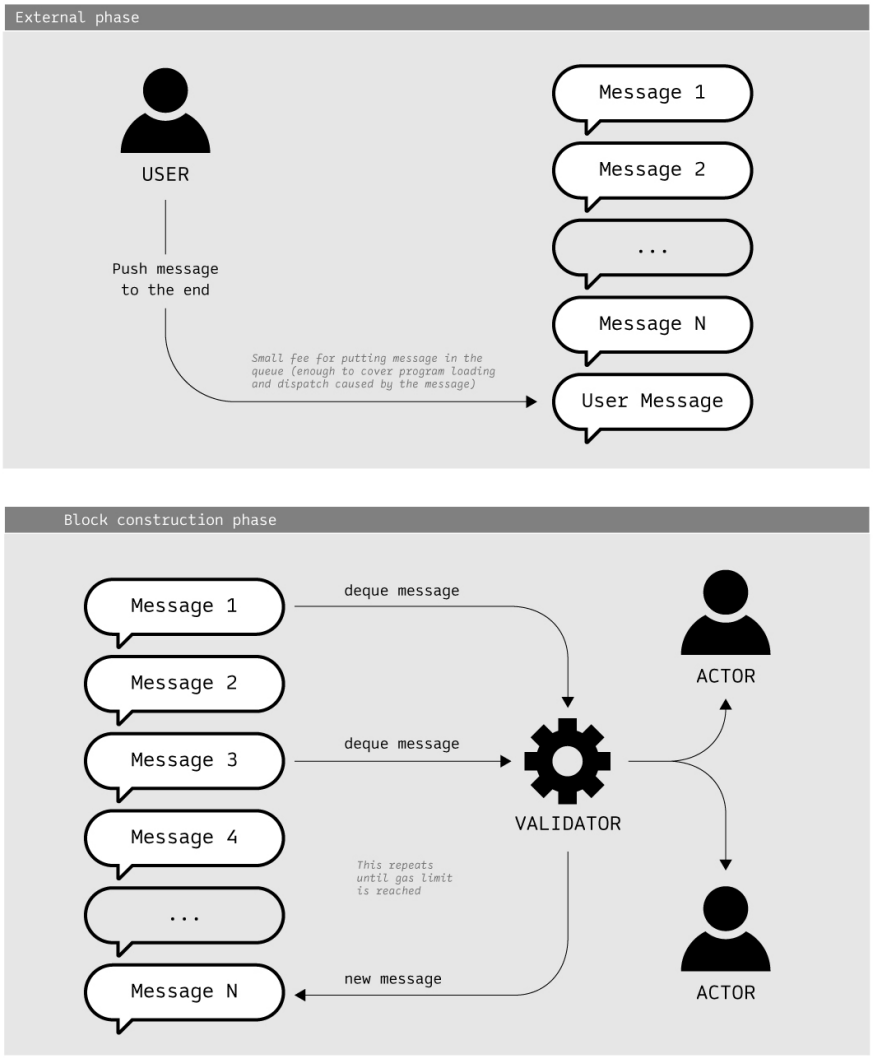
Balance transfers

Regular balance transfers are performed inside 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. Messages with destination “system” end up in the

event log to be inspected in the user space.



6. Balance transfers and gas economy

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 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.

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

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

- source account,

- target account,
- payload,
- gas_limit
- value

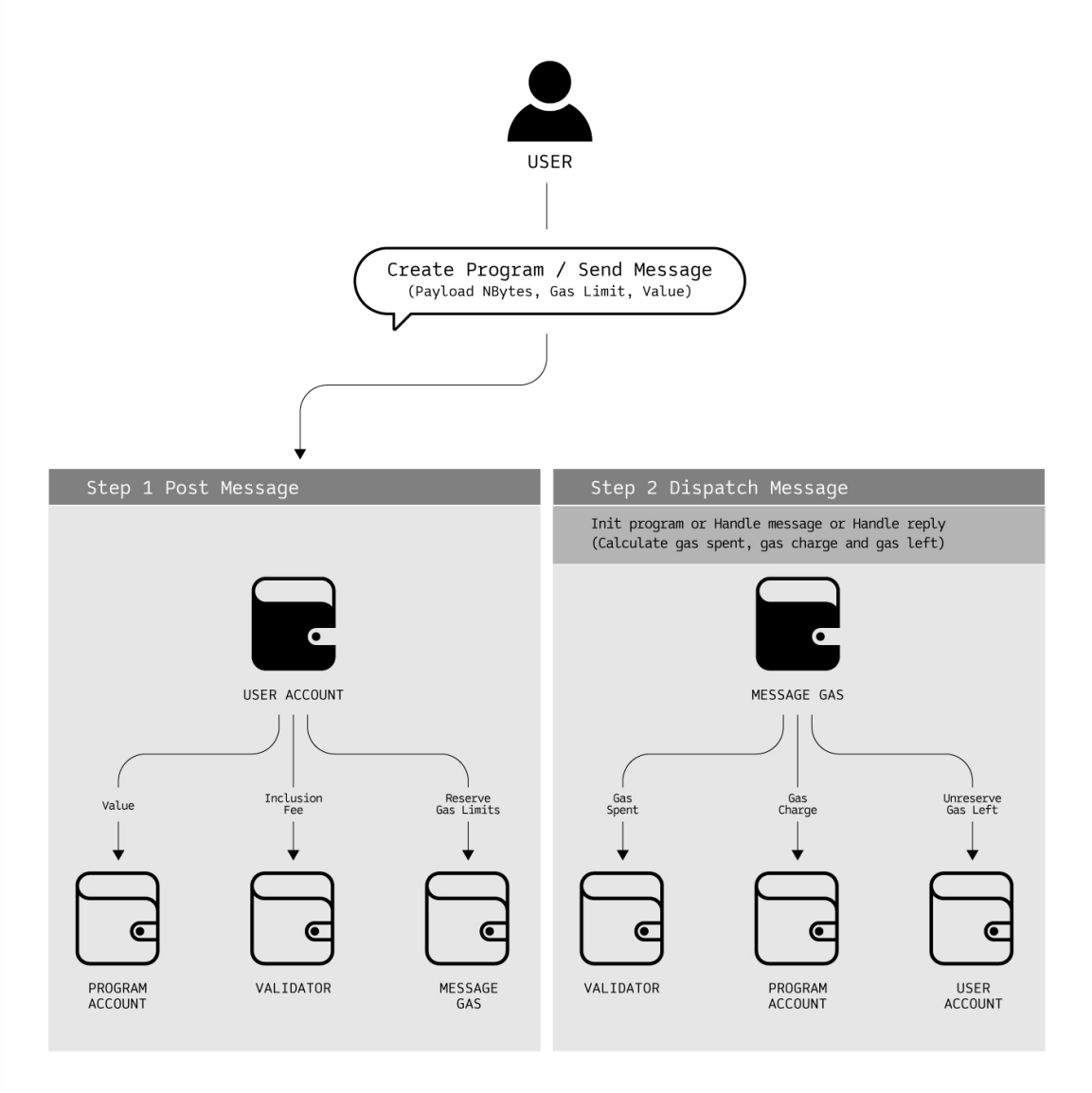
There are five types of messages used in Gear network:

1. A special message from 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 processing message post.
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, 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 initiator account for paying a small processing fee. This is the standard inclusion fee for Substrate framework, its size depends on different factors. Refer to Substrate Documentation for details.

Message processing consists of two steps:



Step one

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

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

For messages between actors (*From* - *To* types), the validator verifies 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.

The reason why Gear introduces a *gas_limit* per block is 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 small processing fee to the validator account and reserves fee equal to *gas_limit* on a message initiator account (User’s account).

Step two

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

When uploading a program, 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.

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

Gear node checks on each increment of *gas_spent* that its value is lower than *gas_limit* specified in the initialization message and 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.

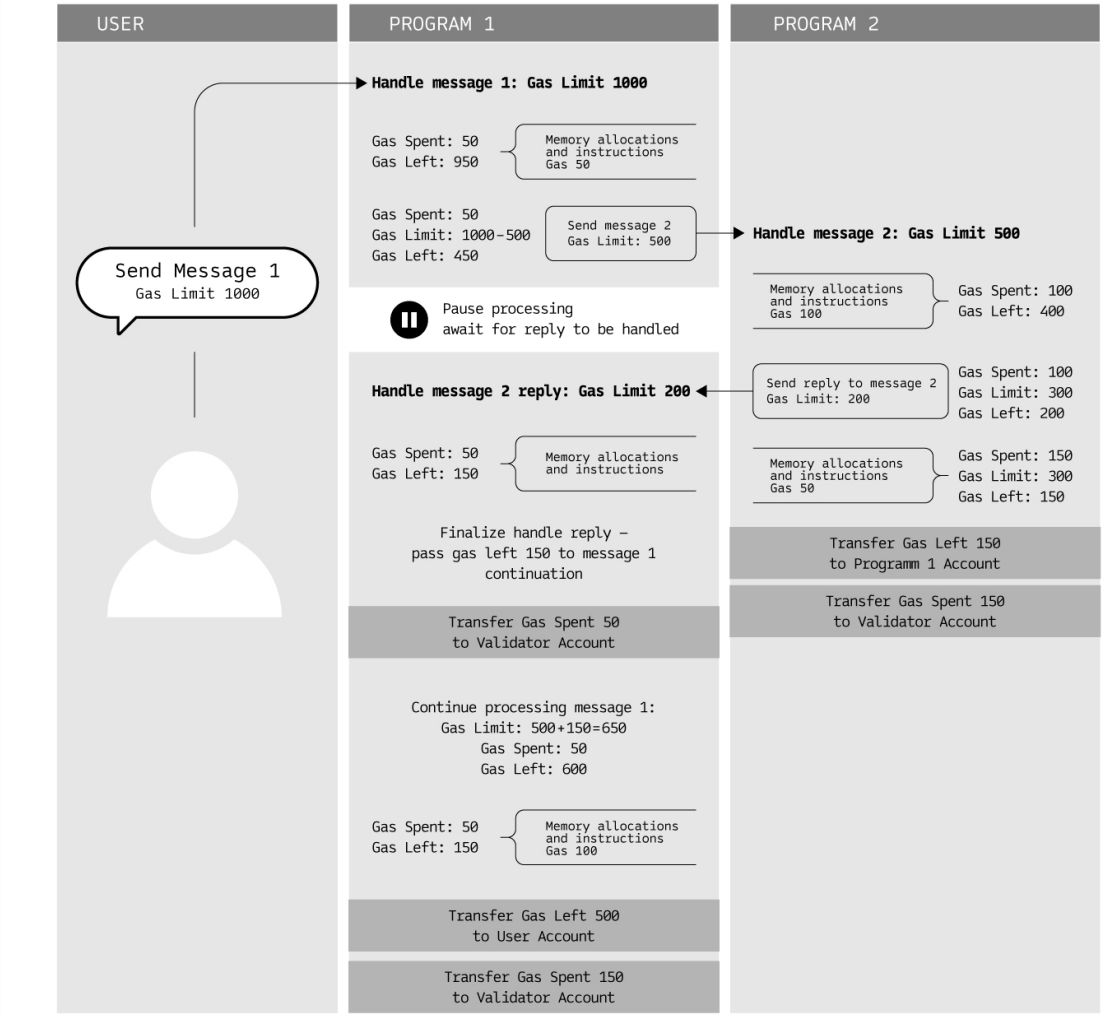
After the program has been initialized, *gas_spent* is charged against the user account towards the validator’s account.

The difference between *gas_limit* and *gas_spent* is *gas_left* and its value is unreserved on the 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 appliy as well as additional memory rent fee is charged via incrementing *gas_spent* value. This memory rent fee is similar to 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 node has to load/save program memory from/to network state.

The picture below shows more detailed example of balance transfers:



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.

7. 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 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 first-class language concept. The principles of Actor model is that programs never share any state and just exchange messages between each other.

While in an ordinary actor model, there is no guarantee on message ordering, Gear makes some extra guarantees that order of messages between two particular programs is preserved.

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

8. 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 avoid 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* pattern in the typical smart contracts code 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 (in Solidity smart contracts). 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 natively provides arbitrary *async/await* syntax for any programs. It greatly simplifies development and testing and reduces the likelihood of errors in smart contract development. Gear API also allows to use synchronous messages by simply not using *await* expression if the logic of the program requires it.

9. Memory Parallelism

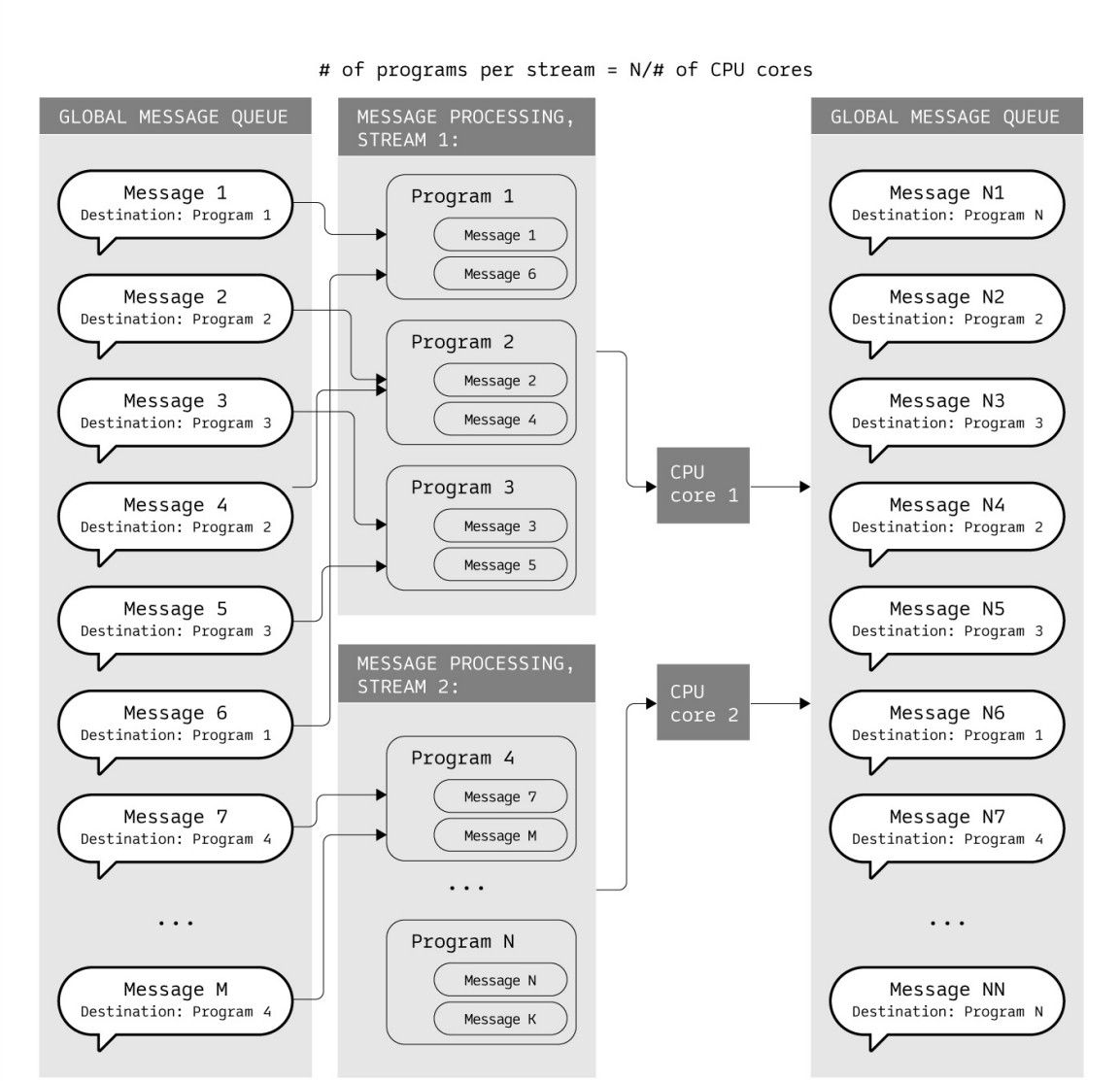
Individual isolated memory space per program allows parallelization of message processing on a Gear node. Number of parallel processing streams equals the number of CPU cores. Each stream processes messages intended for a defined set of programs. It relates to messages sent from other programs or from outside (user's transactions).

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

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 processes messages for plenty of programs. The result of message processing is a set of new messages from each stream that is added to the message queue, then the cycle repeats. The resultant

messages generated during message processing are usually sent to another address (return to origin or to the next program).



10. Typical scenario

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

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

```
static mut CHARGE: u32 = 0;

static mut LIMIT: u32 = 0;

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

#[no_mangle]
pub unsafe extern "C" fn handle() {
```

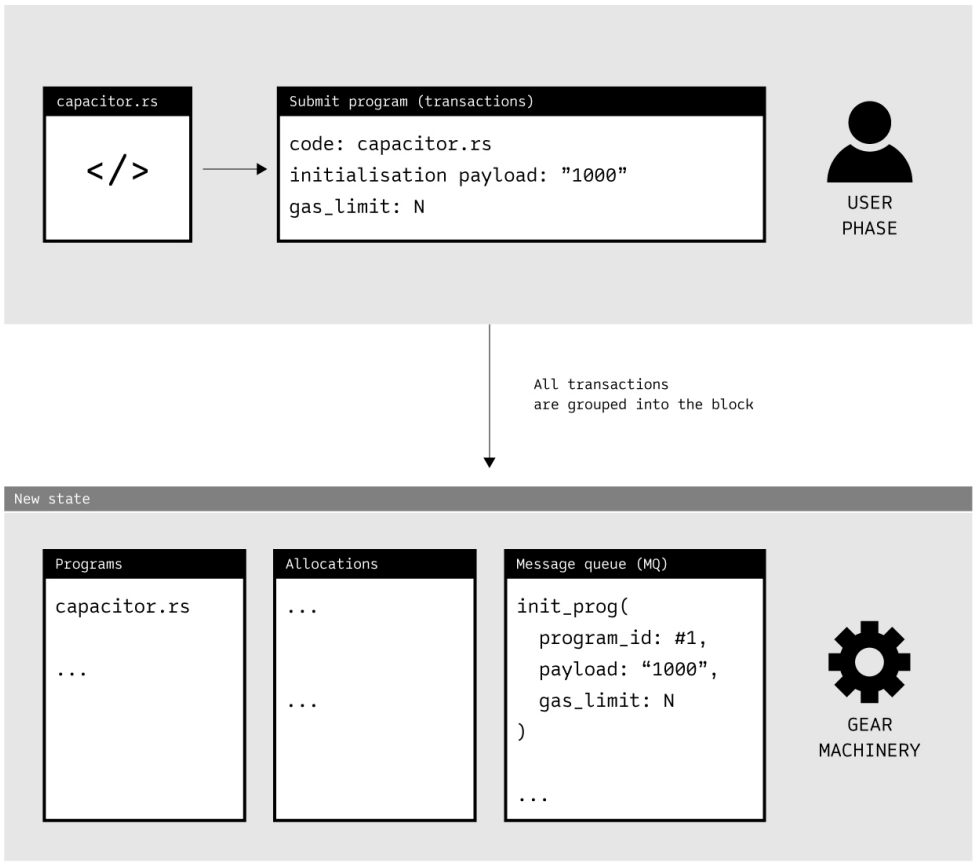
```
    let new_msg = String::from_utf8(msg::load()).expect("Invalid
        message: should be utf-8");

    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);
        CHARGE = 0;
    }
}
```

For full example, refer to our test crate.

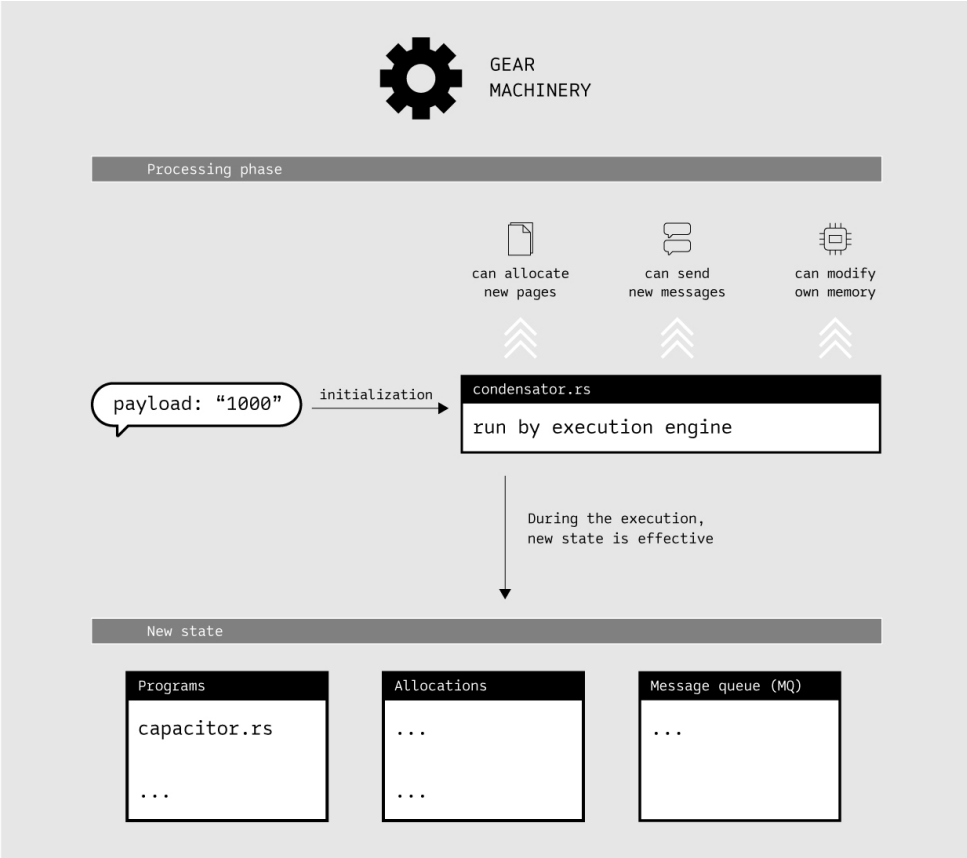
This is a simple program that “charges” with incoming messages and “discharges” when total amount of “charge” exceeds some limit provided in initialization.

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

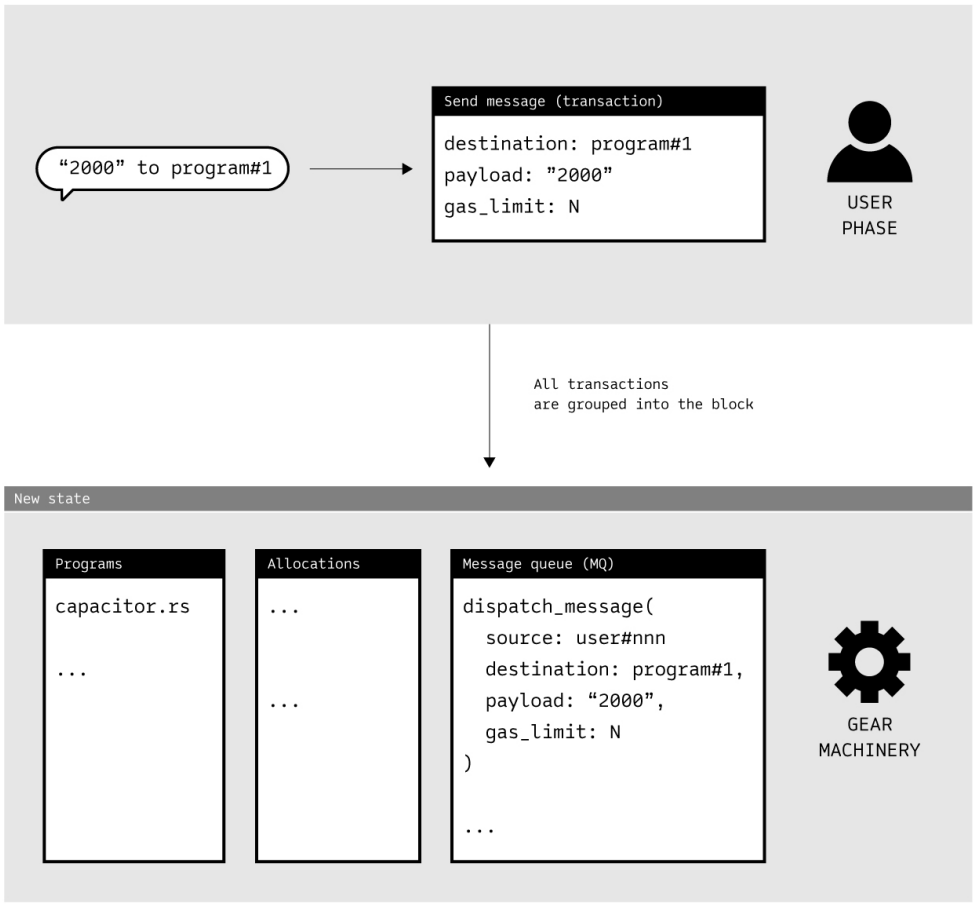
1. This is how the program is created - user just sends a transaction with program and initialization arguments:



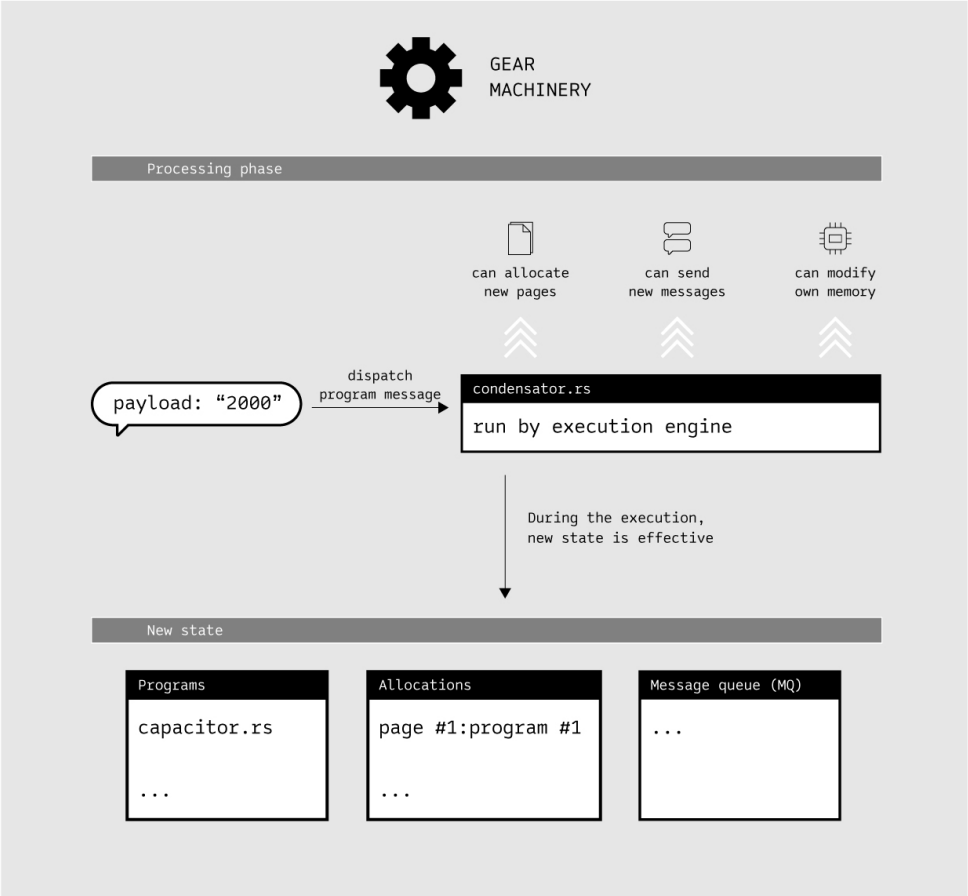
2. Gear then processes this new input:



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



4. Gear then process this new input:



11. Virtual machine (WebAssembly)

Gear uses WebAssembly (or Wasm) under the hood. Any Gear program is in WebAssembly format.

WebAssembly is a code format for deploying programs. In Gear context, any smart-contract is a WebAssembly program.

WebAssembly has the following advantages:

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

WebAssembly is a global industrial technology, remarkable for many reasons:

- It has been designed and implemented in collaboration between all major competitors in its space.
- It has been designed and released along with a complete mathematical, machine-verified formalisation.

12. Applicability and use cases

Micro/nanoservices

Years ago, software development moved from monolithic to microservice architecture. Addressing complexities of previous approach related to tightly coupled, interconnected code. In recent years, nanoservices have become mainstream to address known microservice complexity.

Any general-purpose language nanoservice function compiled to Wasm can be uploaded as a program to Gear. Gear will execute the code to process a certain task within necessary timeframe. Gear provides automatic seamless scalability according to the number of tasks necessary to process. You only pay for the resources your function consumes.

In interaction with other nanoservices in Gear, you get an applicable architecture out of the box. Just focus on the pieces of code that matter to you, and Gear handles the rest.