Oracle

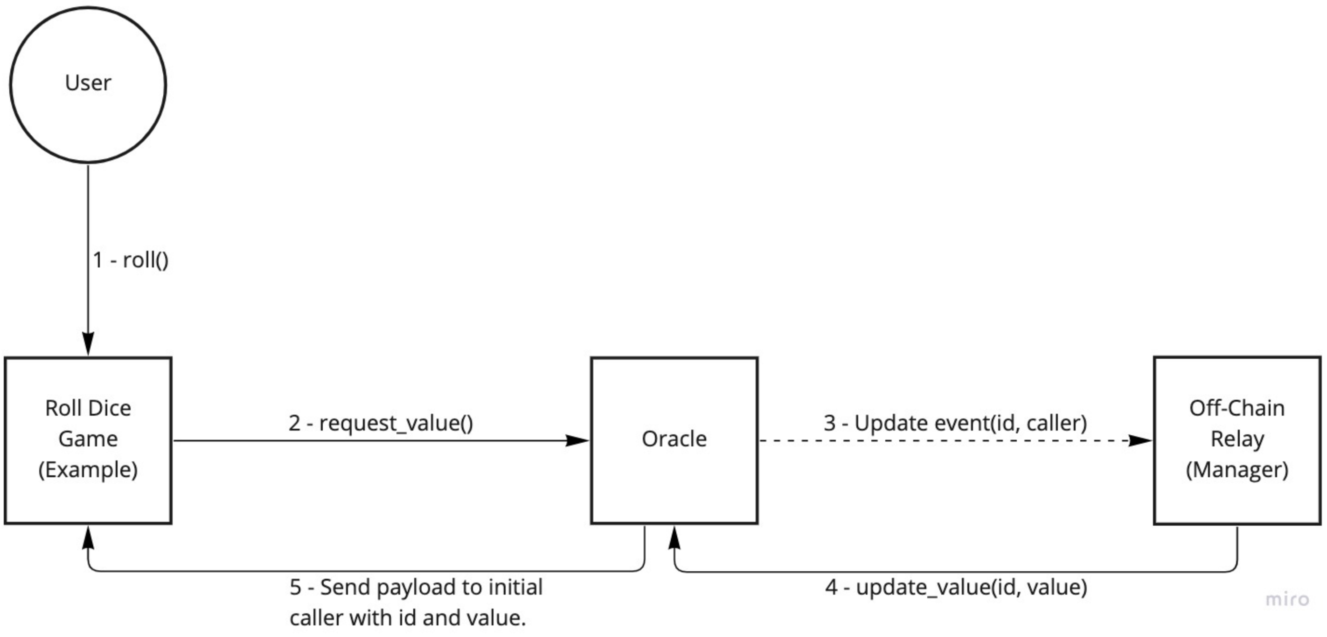
Logic

Every time the caller contract wants to get the off-chain value, he must send specific action called `RequestValue`. Then internally oracle will manage request queue(increased by nonce) and emit new event for off-chain relay. When off-chain service processed new event,

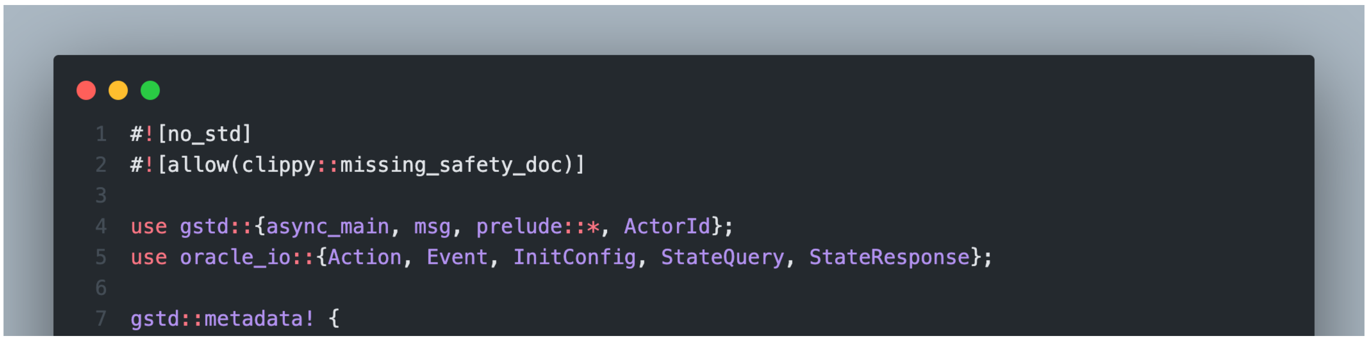
```
UpdateValue {
    id: u128,
    value: u128
}
```

action must be called by manager(relay). Oracle owner may specify any manager and change him over-time. `UpdateValue` forms payload to initial caller program and execute it, then request id gets removed from `requests_queue`. Callee program must handle payload ahead of basic action handler by oracle program id. Example payload layout: `id: u128, value: u128`.

Architecture

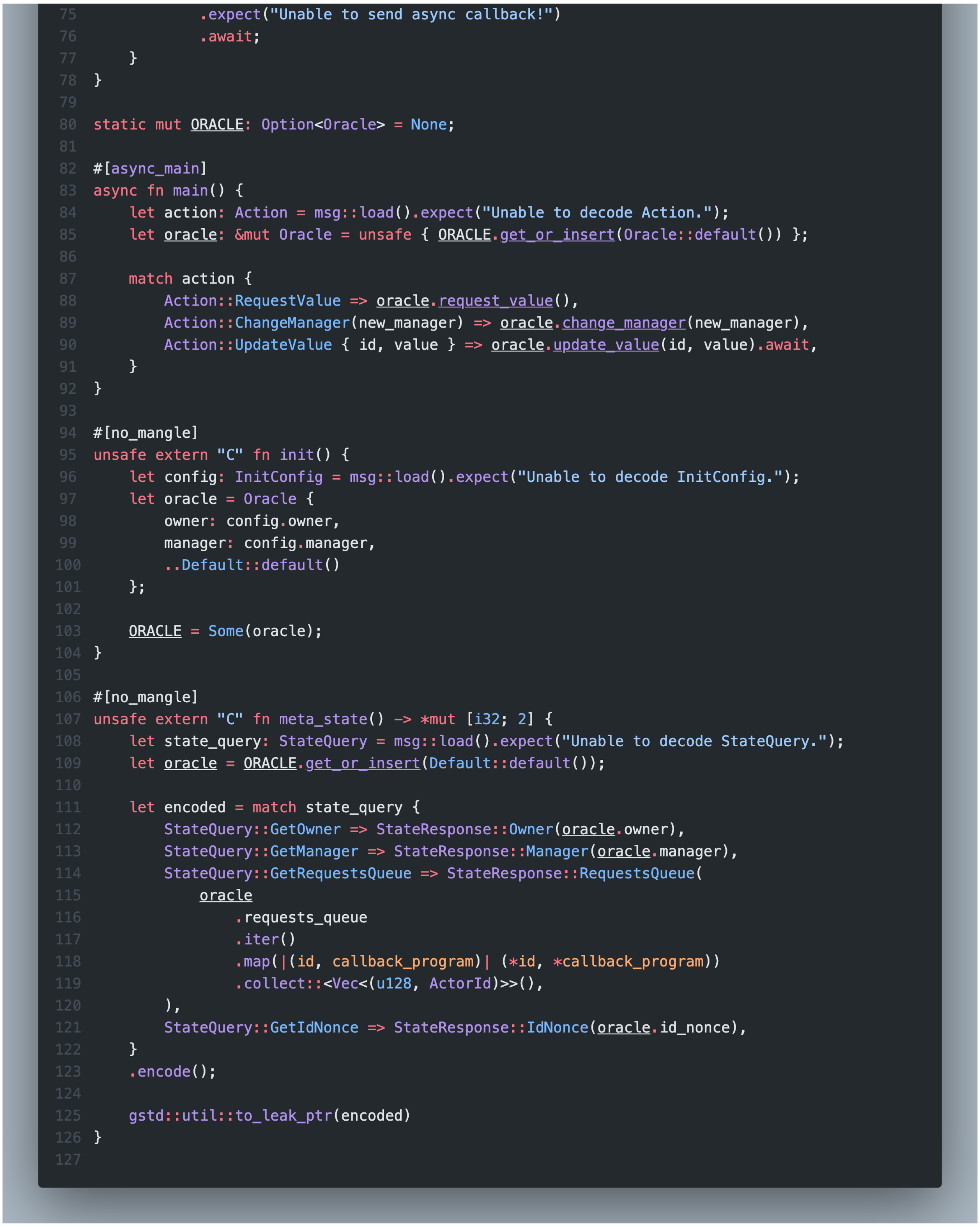

User
1 - roll()
Roll Dice Game (Example)
2 - request_value()
Oracle
3 - Update event(id, caller)
Off-Chain Relay (Manager)
4 - update_value(id, value)
5 - Send payload to initial caller with id and value.
miro

Structs and functions



```
#![no_std]
#![allow(clippy::missing_safety_doc)]

use gstd::{async_main, msg, prelude::*, ActorId};
use oracle_io::{Action, Event, InitConfig, StateQuery, StateResponse};

gstd::metadata! {
```

```rust
    title: "Oracle",
    init:
        input: InitConfig,
    handle:
        input: Action,
        output: Event,
    state:
        input: StateQuery,
        output: StateResponse,
}

#[derive(Debug, Default)]
pub struct Oracle {
    pub requests_queue: BTreeMap<u128, ActorId>,
    pub owner: ActorId,
    pub manager: ActorId,
    pub id_nonce: u128,
}

impl Oracle {
    pub fn request_value(&mut self) {
        self.id_nonce = self.id_nonce.checked_add(1).expect("Math overflow!");
        let id = self.id_nonce;

        let program = msg::source();

        if self.requests_queue.insert(id, program).is_some() {
            panic!("Invalid queue nonce!");
        }

        // Emit request with id from queue
        msg::reply(
            Event::NewUpdateRequest {
                id,
                caller: program,
            },
            0,
        )
        .unwrap();
    }

    pub fn change_manager(&mut self, new_manager: ActorId) {
        if msg::source() != self.owner {
            panic!("Only owner allowed to call this function!");
        }

        self.manager = new_manager;

        msg::reply(Event::NewManager(new_manager), 0).unwrap();
    }

    pub async fn update_value(&mut self, id: u128, value: u128) {
        if msg::source() != self.manager {
            panic!("Only manager allowed to call this function!");
        }

        let callback_program = *self
            .requests_queue
            .get(&id)
            .expect("Provided ID not found in requests queue!");

        if self.requests_queue.remove(&id).is_none() {
            panic!("Provided ID not found in requests queue!");
        }

        // Callback program with value
        let _callback_result = msg::send_for_reply(callback_program, (id, value).encode(), 0)
```



```rust
            .expect("Unable to send async callback!")
            .await;
    }
}

static mut ORACLE: Option<Oracle> = None;

#[async_main]
async fn main() {
    let action: Action = msg::load().expect("Unable to decode Action.");
    let oracle: &mut Oracle = unsafe { ORACLE.get_or_insert(Oracle::default()) };

    match action {
        Action::RequestValue => oracle.request_value(),
        Action::ChangeManager(new_manager) => oracle.change_manager(new_manager),
        Action::UpdateValue { id, value } => oracle.update_value(id, value).await,
    }
}

#[no_mangle]
unsafe extern "C" fn init() {
    let config: InitConfig = msg::load().expect("Unable to decode InitConfig.");
    let oracle = Oracle {
        owner: config.owner,
        manager: config.manager,
        ..Default::default()
    };

    ORACLE = Some(oracle);
}

#[no_mangle]
unsafe extern "C" fn meta_state() -> *mut [i32; 2] {
    let state_query: StateQuery = msg::load().expect("Unable to decode StateQuery.");
    let oracle = ORACLE.get_or_insert(Default::default());

    let encoded = match state_query {
        StateQuery::GetOwner => StateResponse::Owner(oracle.owner),
        StateQuery::GetManager => StateResponse::Manager(oracle.manager),
        StateQuery::GetRequestsQueue => StateResponse::RequestsQueue(
            oracle
                .requests_queue
                .iter()
                .map(|(id, callback_program)| (*id, *callback_program))
                .collect::<Vec<(u128, ActorId)>>(),
        ),
        StateQuery::GetIdNonce => StateResponse::IdNonce(oracle.id_nonce),
    }
    .encode();

    gstd::util::to_leak_ptr(encoded)
}

```