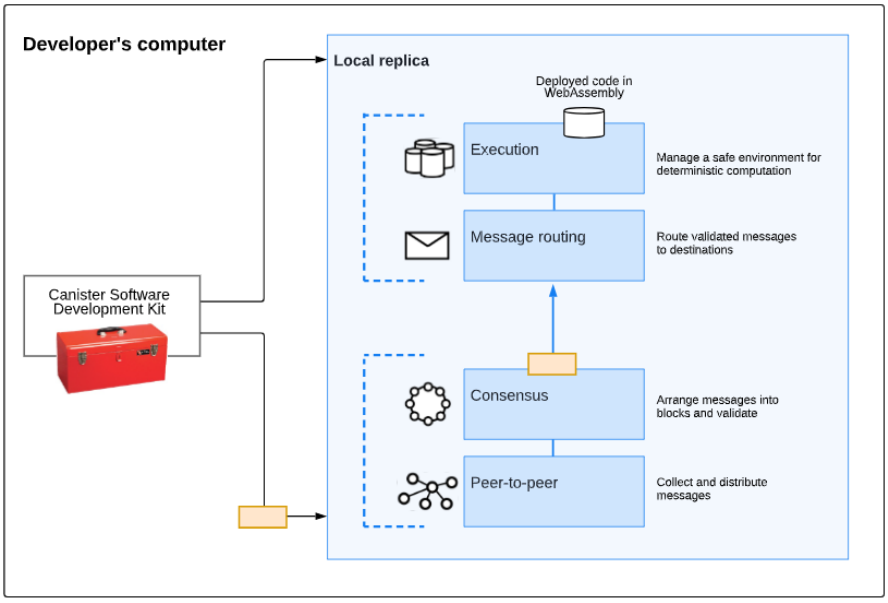
Local execution environment

Date	
Slides	
Courses	IDP
Lecture video link	

Introduction

IC SDK is the SDK created by DFINITY for creating and managing canisters. A local canister execution environment is a setup on a developer's local machine that allows them to run and test canisters. The SDK provides various different commands to take care of the local execution environment. The diagram shows the things that are running using the Canister Development Kit. Canister Development Kit (CDK) is an adapter used by the IC SDK that provides a programming language with the features necessary to create and manage canisters.



There are four major components of the local replica that is created:

- A **peer-to-peer** (P2P) networking layer that collects and advertises messages from users, from other nodes in its subnet blockchain, and from other subnet blockchains. Messages received by the peer-to-peer layer are replicated to all of the nodes in the subnet to ensure the security, reliability, and resiliency.

- A **consensus** layer that selects and sequences messages received from users and from different subnets to create blockchain blocks that can be notarized and finalized by Byzantine Fault Tolerant Consensus forming the evolving blockchain. These finalized blocks are delivered to the message routing layer.
- A **message routing** layer that routes user- and system-generated messages between subnets, manages the input and output queues for dapps, and schedules messages for execution.
- An **execution environment** that calculates the deterministic computation involved in executing a smart contract by processes the messages it receives from the message routing layer.

There are two types of subnets you can deploy locally:

1. Application Subnet
2. System Subnet

If you're running the system subnet, you can deploy all the NNS canisters. If you want to develop and deploy the application canisters for your dapp, you have to deploy it on the application subnet. You can still install the NNS canisters in the application subnet. However, all of the NNS canisters might not work perfectly as the system subnets have special characteristics:

- No cycles accounting takes place.
- More generous per-call instruction limit.
- More generous Wasm module size limit.

Open questions

1. **When we are doing dfx start, how is it building the Replica on the developer machine and what are the processes that are being running?**

Deep-dive into dfx commands.

There are many commands in dfx that are used to start and manage the local canister execution environment. In this section, we would conduct an in-depth study to understand what is actually happening when these commands are used.

Whenever we deploy anything on our local machine using dfx, it is running everything as it should in the mainnet. i.e. it runs the networking, consensus etc.

Now the question is how is this being done through the local machine?

lets investigate the dfx start command and try to investigate what it is doing. Going through the source code of the command (sdk/src/dfx/src/commands/start.rs)

High level overview of what is happening:

1. **Logging and Configuration**: If the application is not running in the background, it logs the version of the dfx (the IC's development tool) it's using. It then retrieves the project configuration and creates a network descriptor, which is a representation of the network configuration.
2. **Checking Previous Process**: It checks if there's a previous process running for the local server. If there is, it likely throws an error or performs some cleanup.
3. **Cleaning State**: If the `clean` option is set, it cleans up the state of the project.
4. **Creating Directories and paths**: It creates necessary directories for the network and writes the network ID if it doesn't exist.
5. **Configuring Adapters**: It configures the Bitcoin adapter and the canister HTTP adapter if they are enabled.
6. **Replica Configuration**: It creates a configuration for the replica. This includes setting the subnet type, log level, and artificial delay, as well as enabling the Bitcoin adapter and the canister HTTP adapter if they are configured.
7. **Emulator or Replica**: Depending on the `emulator` option, it either creates an emulator configuration or a replica configuration.
8. **Starting Actors**: It starts various actors (**Actix framework**). Depending on whether the emulator is enabled, it either starts an emulator actor or a Bitcoin adapter actor, a canister HTTP adapter actor, and a replica actor. It also starts an ICX proxy actor, which is responsible for forwarding and managing browser requests.
9. **Running the System**: It runs the Actix system, which starts the execution of all the actors.
10. **Cleanup**: After the system has run, it removes the socket files for the Bitcoin adapter and the canister HTTP adapter if they exist.

TLDR:

The `dfx start` command starts the `replica` and the `ICX proxy` The `ICX proxy` id responsible for forwarding and managinging the local browser reques the local browser requests and acts as the boundary node work.

Next question: What is being run in the `replica` ?

```
t#[context("Failed to start replica actor.")]
pub fn start_replica_actor(
    env: &dyn Environment,
    replica_config: ReplicaConfig,
    local_server_descriptor: &LocalServerDescriptor,
    shutdown_controller: Addr<ShutdownController>,
    btc_adapter_ready_subscribe: Option<Recipient<BtcAdapterReadySubscribe>>,
    canister_http_adapter_ready_subscribe: Option<Recipient<CanisterHttpAdapterReadySubscribe>>,
) -> DfxResult<Addr<Replica>> {
    // get binary path
    let replica_path = env.get_cache().get_binary_command_path("replica")?;
    let ic_starter_path = env.get_cache().get_binary_command_path("ic-starter")?;

    setup_replica_env(local_server_descriptor, &replica_config)?;
    let replica_pid_path = local_server_descriptor.replica_pid_path();

    let bitcoin_integration_config = if local_server_descriptor.bitcoin.enabled {
        let canister_init_arg = local_server_descriptor.bitcoin.canister_init_arg.clone();
        Some(BitcoinIntegrationConfig { canister_init_arg })
    } else {H
        None
    };

    let actor_config = replica::Config {
        ic_starter_path,
        replica_config,
        bitcoin_integration_config,
        replica_path,
        shutdown_controller,
        logger: Some(env.get_logger().clone()),
        replica_pid_path,
        btc_adapter_ready_subscribe,
        canister_http_adapter_ready_subscribe,
    };
    Ok(Replica::new(actor_config).start())
}
```

This starts the replica using the replica binaries. The `Replica::new(actor_config).start()` line is where the replica actor is actually created and started.

On further investigations on the replica actor (sdk/src/dfx/src/actors/replica.rs), what it does is:

1. Defines the replica structure and implementation

2. in the implementation, it creates a replica thread, and in that thread they run the `replica` binary.
3. There are a lot of other steps that is taking place, which is not relevant to us.

Next question? what does this `replica` binary have? For this, we have to explore the `ic` source code.