Specification of the Cardano shell

Kristijan Šarić

06.02.2019

This document is a high-level specification of how the pieces that make Cardano shell work together and an attempt to define a general, secure and simple way to combine them, while introducing the reader with the specifics of how they work. *Beware, work in progress!*

Contents

List of Figures

List of Tables

1 Introduction

The introduction. WIP!

2 IPC and communication between Daedalus and Node

Currently, the *Daedalus* and the *Node* (this is what I'm calling the node, thus the uppercase) communicate via **IPC for reaching a consensus which port to use for the JSON API communication, since we have a lot of issues of ports being used** . **IPC** (Inter Process Communication) is a set of methods which processes can use to communicate - `https://en.wikipedia.org/wiki/Inter-process_communication`.

The actual communication right now is being done by the *spawn* function, pieces of which can be found here. The part of the code which adds the handle id which they will use to communicate via environment variable "NODE_CHANNEL_FD" here.

Currently, the *Deadalus* starts the *Node* (we will ignore the *Launcher* for now, since it complicates the story a bit).

The initial simplified communication protocol can be seen on Figure 1.

When the *Daedalus* calls and starts the *Node*, it also opens up the **IPC** protocol to enable communication between the two. First, the *Node* sends the message **Started** back to the *Daedalus* to inform him that the communication can begin. After that, *Daedalus* sends the message **QueryPort** to the *Node*, and the *Node* responds with the free port it found using **ReplyPort PORTNUM** that is going to be used for starting the HTTP "server" serving the *JSON API* which they can then use to communicate further.

The communication is bi-directional, on Windows it is using **named pipes**.

We can easily generalize this concept. We can say that *Daedalus* is the *Server*, and that the *Node* is the *Client*. Since the communication is bi-directional, we can say that either way, but we can presume that the *Server* is the process which is started first.

What does this bring us? It brings us the ability to **decouple** our implementation from the specific setting where *Daedalus* is the *Server* and the *Node* is the *Client*. There are some ideas that we might switch the order of the way we currently run these two, so we can freely implement that either way. For example, when we write tests for this communication protocol, we need to both start the server and the client and then check their interaction. It also removes any extra information that we might need to drop anyway. What we are focusing here is the **communication protocol** and not it's actors. We don't really care who says to whom, we are interested in seeing what is being said - *how the whole protocol works*.

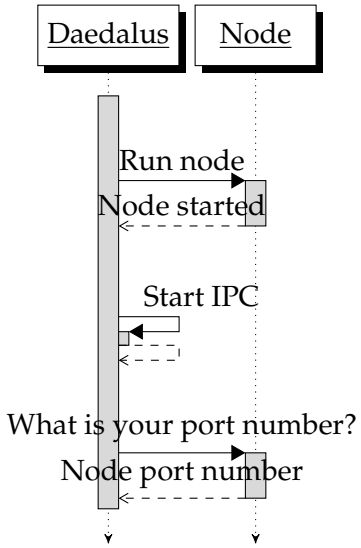

Figure 1: Message protocol for current IPC implementation.

2.1 IPC simple communication

Here we will take a look at the simplest possible communication. **Ping** and **Pong**. The *Server* sends the **Ping** and the *Client* responds with a **Pong**.

We can take a look at this very simple protocol on Figure 2.

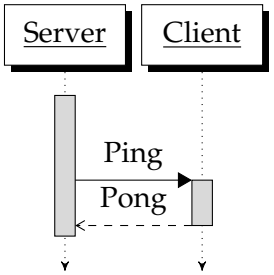

Figure 2: IPC message protocol for Ping/Pong.

Very simple transformation rules can be applied here.

$$Ping \Longrightarrow Pong \tag{1}$$

$$\frac{\text{Ping}}{\text{Pong}} \tag{2}$$

And that's it. Let's now take a look at a more involved case where we care about exceptional situations.

2.2 IPC simple communication with exceptions

Here we will take a look at the simplest possible communication. **Ping** and **Pong**. The *Server* sends the **Ping** and the *Client* responds with a **Pong**. We also consider all the **exceptional situations**.

What can go wrong when we send the message? For example, we need to consider what actor are we currently looking at. Is it the *Client* or the *Server*? The socket could be closed, the process could be shut down, the message could take too long to respond, the format could be wrong (yay, types) and so on. Since we are not interested in the low-level details, we can generalize these exceptional situations into more general messages that we can enumerate.

We can simplify all of this by saying there is an exception *MessageSendFailure* that can be used on **both sides**. This simplifies a lot of things for us, including different exceptional situations we might reach. The result can be seen here.

$$(Ping \land \neg MessageSendFailure) \Longrightarrow Pong \tag{3}$$

$$\frac{Ping \qquad \overline{\neg MessageSendFailure}}{Pong} \tag{4}$$

The other situation just halts the protocol, which we can observe as bottom.

$$\frac{\text{MessageSendFailure}}{\bot} \tag{5}$$

And this is a simplified way we can observe any exception situation in this communication.

2.3 IPC protocol communication with exceptions

Here we will take a look at a more complex communication where we actually have the full communication between parties - *Server* and *Client*.

When the communication starts, the *Client* sends the **Started** message to the *Server* indicating that it's now ready for the communication. After that, the *Server* sends the **QueryPort** message, which is then sent to the *Client* where the *Client* selects over which port will the *REST JSON* communication continue using **ReplyPort PORT**.

What can go wrong when we send the message? The same thing as in the previous section.

We can simplify all of this by saying there is an exception *MessageSendFailure* that can be used on both sides. This simplifies a lot of things for us, including different exceptional situations we might reach. The result can be seen here.

$$Ping \land \neg MessageSendFailure \implies Pong \tag{6}$$

$$Started \land QueryPort \land \neg MessageSendFailure \implies ReplyPort \tag{7}$$

The other situation just halts the protocol, which we can observe as bottom.

$$\frac{Ping \qquad \overline{\neg MessageSendFailure}}{Pong} \tag{8}$$

$$\frac{\dfrac{Started}{QueryPort} \qquad \overline{\neg MessageSendFailure}}{ReplyPort\{\mathbf{port} \mid port < 65535 \land port > 0\}} \tag{9}$$

$$\frac{\text{MessageSendFailure}}{\bot} \tag{10}$$

And that is our full communication protocol. It can be seen on Figure 4.

The state machine diagram that can be used to represent this can be seen here.

We can then consider a simple transition function for the client.

$$\begin{aligned} &\mathsf{fullProtocol} \in \mathsf{ServerMessage} \to \mathsf{ClientMessage} \\ &\mathsf{fullProtocol\ Ping} \quad\ = \mathsf{Pong} \\ &\mathsf{fullProtocol\ QueryPort} = \mathsf{ReplyPort\ 0} \end{aligned}$$

So what we really have, when speaking about actual state transitions is an asynchronous interface between two "devices" (of any sort). So we can use two channels between the two "devices" (in our case the server and the client). One channel for communicating via server-client direction and the other one for the client-server direction.

On both of these we can use a simple 2-phase handshaking protocol.

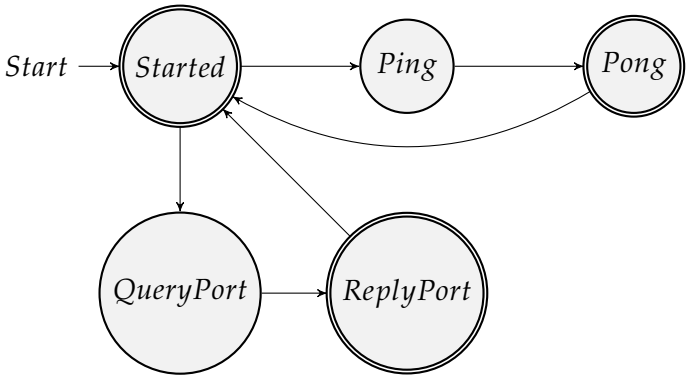

Figure 3: Full protocol FSM.

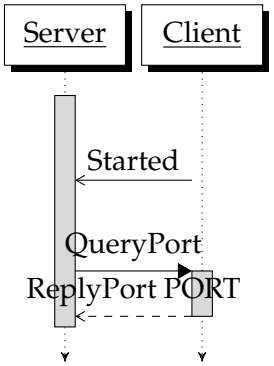

Figure 4: Message protocol for the full IPC implementation.

Of course, we can build on top of that, and actually implement a **FIFO** structure between the two, that is identical to the mechanism we will actually use in the implementation, which are named pipes (bi-directional). There are some specifics for Windows when using node.js, but we can ignore that in the spec. For bidirectional communication, you need two FIFOs, one for the server-client and the other for client-server.

We can imagine looking at just one process and seeing the input **FIFO** and the output **FIFO**. That simplifies the story and the specification, observe it as an input queue of messages and an output queue of messages for *a single process*, and simply composing two of these together yields a full communication. We can imagine that the process in this protocol is a simple *mapping* we mentioned above which just converts a **InMsg** to a **OutMsg** from the input **FIFO** to the output **FIFO**. The example of the communication can be as presented on the next state transition diagram.

For example, for the **Ping-Pong** protocol:

$$\begin{bmatrix} inQueue = [] \\ outQueue = [] \end{bmatrix} \Rightarrow \begin{bmatrix} inQueue = [\textbf{Ping}] \\ outQueue = [] \end{bmatrix} \Longrightarrow \begin{bmatrix} inQueue = [] \\ outQueue = [\textbf{Pong}] \end{bmatrix}$$

For example, for the **QueryPort-ReplyPort** protocol:

$$\begin{bmatrix} inQueue = [] \\ outQueue = [] \end{bmatrix} \Rightarrow \begin{bmatrix} inQueue = [\textbf{QueryPort}] \\ outQueue = [] \end{bmatrix} \Longrightarrow \begin{bmatrix} inQueue = [] \\ outQueue = [\textbf{ReplyPort}] \end{bmatrix}$$

3 Update mechanism

Currently, the *Daedalus* and the *Node* (this is what I'm calling the node, thus the uppercase) communicate via **JSON API** once they have settled in on a port via which to communicate (see here). First of all, we need to understand that the blocks in the blockchain contain the version of *Daedalus* (the frontend). We can say that *Daedalus*, also known as *frontend* is the *Server*, and that the *Node*, also known as *backend* is the *Client*, which are the same things under different names. We can imagine that each block can contain a version of the frontend, which is essentially a hash signature from the installer. That is something that can change in the future, but we can simplify our life by imagining that what the blockchain contains is the link for the installer (which, when simplified, it does).

Let's start simple. Let's take the blockchain and the version into consideration. First of all, we can consider what we have in production, since that is something we can base our assumptions on:

- there are **101**(which is the number of epochs at the time of writing this) **epoch**s in the **blockchain**
- there is 21600 **slot**s in an **epoch**
- each **slot** *may* contains a **block**
- there could be 21600 **block**s in an **epoch**, if all **slot**s have a **block**
- each **block** *may* contain a **frontend version**
- when a **hard fork** occurs, the update system stops working and the client needs to download the new frontend manually, in our current versions we have that covered since the protocol version 1 and 2 will contain the information about the update

We can remove other details for now and simply focus on this simple scenario. The very simple representation can be seen on Figure 5.

We then consider how to describe such a system. Since we can observe computation/digital systems as state machines, we proceed to do so. We can imagine a simple system that only deals with the blocks and slots in a very simple manner. We have the blockchain which is the collection of blocks (which then contain transactions, but we omit that since we are not really interested in them right now). The node which we run is a simple machine which reads those blocks into (local) memory so it can know at what state the whole blockchain is. Given that, we can simply imagine the node syncing blocks each "tick" (a unit of time which we are not interested in, but serves as a snapshot of state of our system in some interesting moments). We can observe such synchronization as seen here:

$$\begin{bmatrix} node\_state = 0 \\ blockchain\_state = 234 \end{bmatrix} \Rightarrow \begin{bmatrix} node\_state = 6 \\ blockchain\_state = 234 \end{bmatrix} \Rightarrow \begin{bmatrix} node\_state = 23 \\ blockchain\_state = 234 \end{bmatrix}$$

In this case, we have a situation where the node synced 6 blocks after the first transition, and it synced further 17 blocks on the second transition. The blockchain state remains constants for the purpose of simplification, it would ordinarily increase.

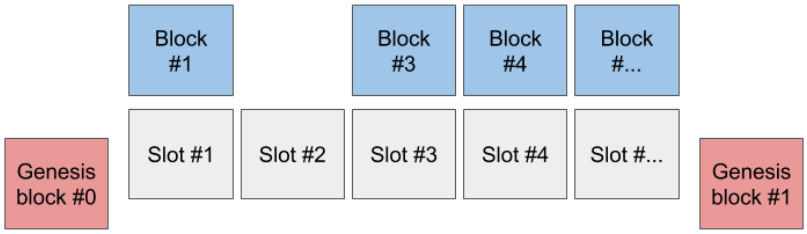

Figure 5: Empty blockchain without any notions of a version.

From there we can add the versions of the frontend installer, as seen on Figure 6.

The somewhat enriched structure requires of us to summon additional states into the process, since we do need some way of describing what happens to the machine itself with these new changes. Or, to put it simply, we need to add information about installer versions on blocks in order to check how it all fits together. We can simplify the idea by first imagining that we only have one update in the whole blockchain, and add additional installers as we further refine the idea. We can imagine that we have the installer on block 22, and that the installer is updated once we sync up to that point, as seen here:

$$\begin{bmatrix} node\_state = 0 \\ blockchain\_state = 234 \\ installer\_version\_block = 22 \\ latest\_installer\_version = 0 \end{bmatrix} \Rightarrow \begin{bmatrix} node\_state = 6 \\ blockchain\_state = 234 \\ installer\_version\_block = 22 \\ latest\_installer\_version = 0 \end{bmatrix} \Rightarrow \begin{bmatrix} node\_state = 23 \\ blockchain\_state = 234 \\ installer\_version\_block = 22 \\ \textbf{latest\_installer\_version = 22} \end{bmatrix}$$

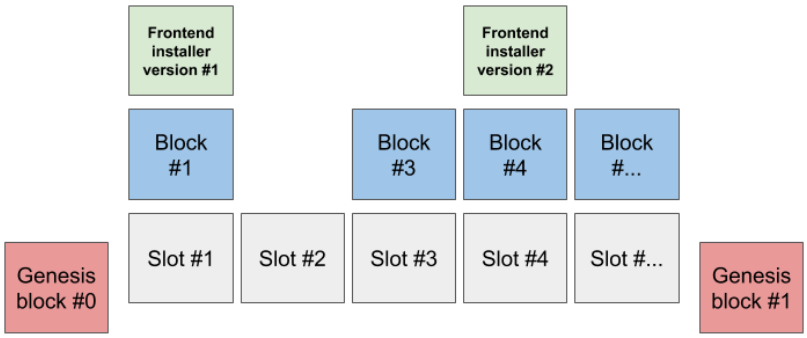

Figure 6: Blockchain with installers on specific blocks.

As you can imagine, this fits very nicely into testing, for example **state-machine-quickcheck** (or similar), using Haskell.

We can advance such idea by increasing the number of updates in the blockchain and by asssigning different versions of the installers to each update.

4 Update mechanism with Launcher

A simple communication between the frontend and the blockchain (backend) can be described as seen on Figure 7.

The specifics of how this works are a bit tricky. We use the **cardano-launcher** also known simply as **Launcher** is something we require so we can have control over the (Electron) Daedalus process and to be sure it shuts down correctly. The installers are different on different platforms:

- on Windows we download and use the installer directly
- on Mac we use the pkg file, which we open using an external program
- on Linux we use a custom script called the *update-runner*, which we build using Nix

For now, we can abstract over that and say that each platform has it's own specifics. Let's take a look at some of the key functions we will use:

$$\mathsf{uniqueKeys} \in \forall k(k \in \mathit{Ordered}).(k \mapsto v) \to \mathbb{P}\ k$$

$$\mathsf{blockchainContents} \in \mathsf{Blockchain} \to (\mathsf{Epoch} \mapsto [\mathsf{Slot}])$$

$$\mathsf{blockchainEpochs} \in \mathsf{Blockchain} \to \mathbb{P}\ \mathsf{Epoch}$$
$$\mathsf{blockchainEpochs} = \mathsf{uniqueKeys} \circ \mathsf{blockchainContents}$$

$$\mathsf{fetchUniqueUpdatesFromBlockchain} \in \mathsf{Blockchain} \to \mathbb{P}\ \mathsf{InstallerVersion}$$

If we take a look at the typical state transition of such a system, we can easily imagine something like this.

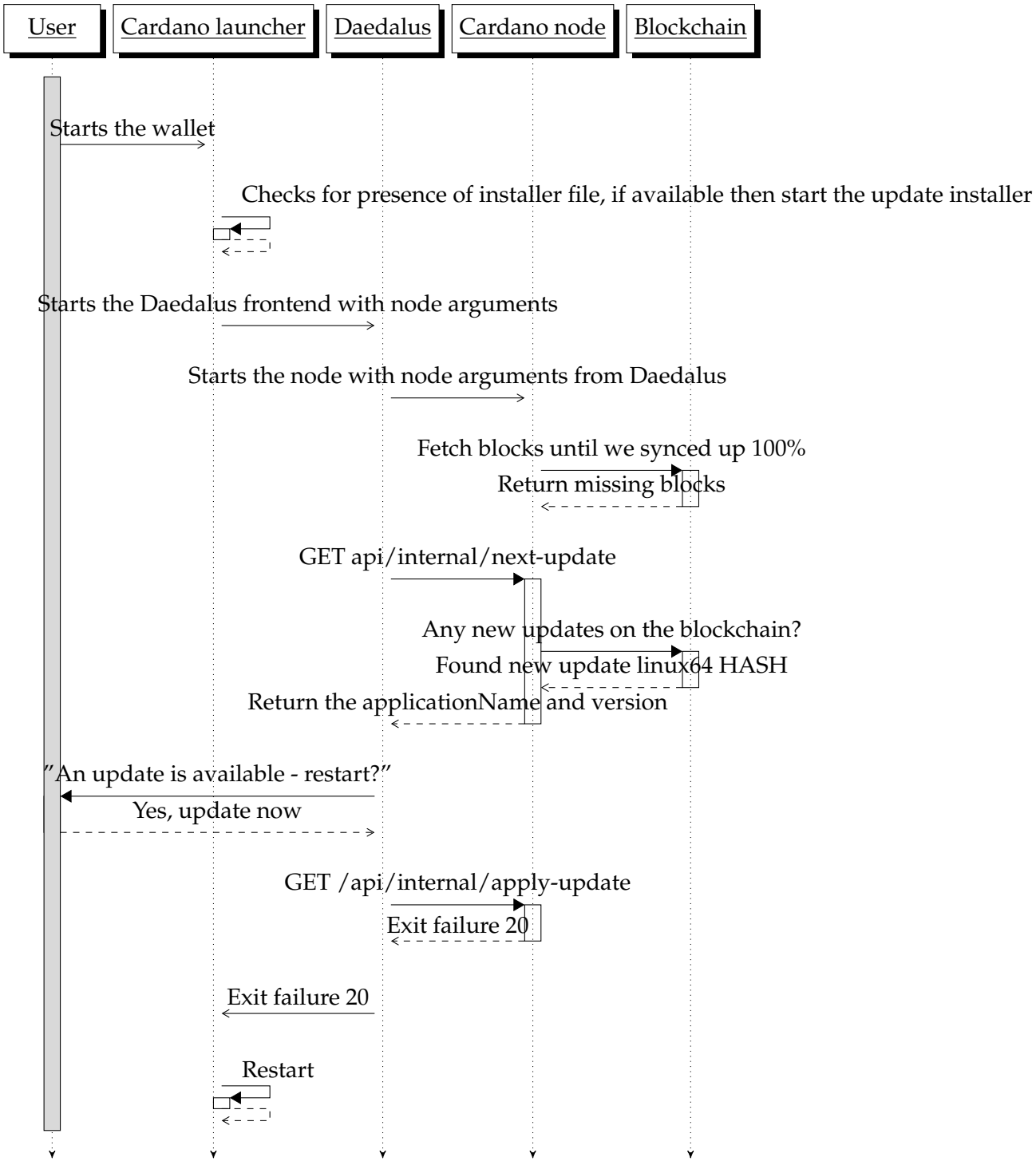

Figure 7: Update system protocol