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$_C HANNEL_F D$"*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 communcation 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.

Since the communication is bi-directional, currently the communication is using **files**.

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*.

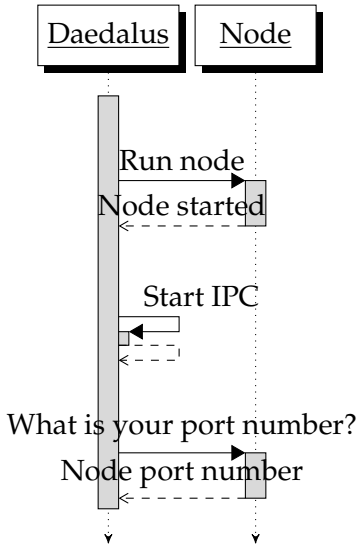


Figure 1: Message protocol for current IPC implementation.

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.

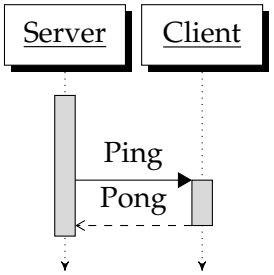


Figure 2: IPC message protocol for Ping/Pong.

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.

```
fullProtocol ∈ ServerMessage → ClientMessage
fullProtocol Ping      = Pong
fullProtocol QueryPort = ReplyPort 0
```

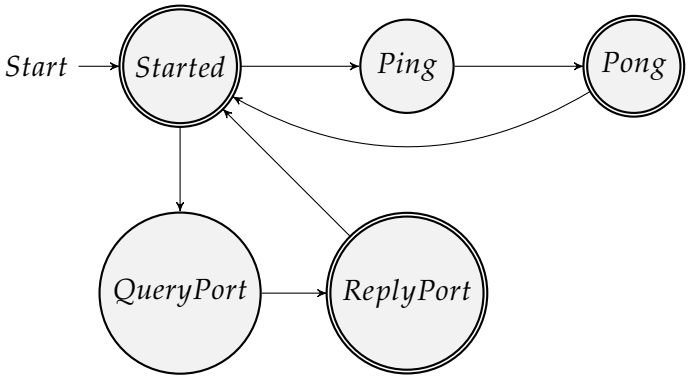


Figure 3: Full protocol FSM.

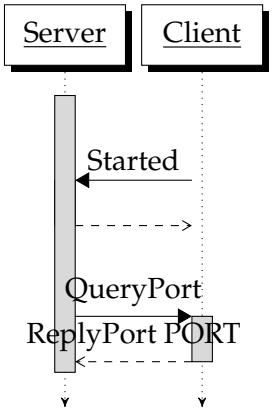
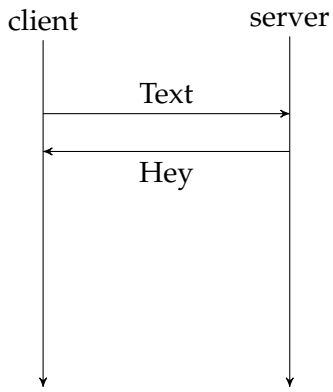


Figure 4: Message protocol for the full IPC implementation.

3 Update mechanism

First of all, we need to understand that the blocks in the blockchain contain the version of *Daedalus* (the frontend). 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.



Let's see how that looks like: