# DS - Linc (Linc Is Not Centralized)

Simon Fernandez

May 3, 2018

# Chapter 1

# Agent deployment

## 1.1 Topology

The chosen topology is a random topology. The network must only be connected. All algorithms are designed so that they can be applied to any kind of topology.

## 1.2 Nodes

There are two types of nodes in LINC.

- **Agents :** They store the data, receive the commands from users, forward the messages across the topology. They can be linked to any kind of node.
- **Users :** Where the user is. Commands sent by the user are sent to this node and then sent to the topology. User nodes are entry nodes of the topology. Users can only be linked to Agents. Inter-user links are forbidden.

## 1.3 Communications

Because the topology is random and unknown, sending a message from node A to node B is not trivial. In LINC I implemented a message passing protocol similar to the protocols used in Internet networks or Local networks.

Each node has a list of its immediate neighbours (in the `nodes()` variable). When sending a message to an immediate neighbour, the node just sends the message to the node. But when a node A wants to send a message to a distant node B, it must know where to send the message. I decided to forbide broadcasting messages not used for routing protocols. So, in order to find where the target node B is, the node A first broadcasts {`whereis, B`}. When a node receives this kind of message, if it knows how to reach B, it replies with {`path, By, B, Dist`} meaning "I (noted By) know how to reach B in Dist hops". When receiving a `path` message, a node updates a list of known distant nodes: now it knows that it can reach B in Dist+1 hops by sending the message By.

Each node only keeps the shortest path, and updates it when receiving `path` messages. The paths are stored in an `ets` table named `linc_route`. In this table there are elements {`Target, Distance, Via`}, meaning that in order to reach `Target`, the node must send the message to `Via` and `Via` will forward it to the next node etc, etc...

With this protocol, nodes keep a table telling who to send the message to in order to reach each node.

## 1.4 Killing a node

(Data management will be considered in the Data chapter)

When a node is properly killed with the command `shutdown`, it sends to its neighbours a `dead` message. This way, neighbours remove the node from their routing tables.

Then, if a node is killed, or dies, we may have the problem of connectivity. The topology may have been splited in two or more disconnected parts. This situation is detected with Acknowledgment messages. There is a list of "important" commands than necessitate an Ack message. So when a node sends this kind of message, it will wait for an Ack (see next section for more in-depth description). If the node does not receive the Ack, then the message may be lost, or the node dead. In this situation, the node calls the function `shrink`. This function considers all nodes reachable in 2 hops, and builds a direct link to these nodes. This way, if a neighbour died, we add a link to bypass this dead node. Once the new path is established, we send the message one more time.

The advandages of this method is that it limits the number of connections built. But if too many nodes die at the same time, the damages may be irrecuperable and part of the topology may be splited.

## 1.5 Ack and how Agents behave

The first basic principle of Agent behaviour is that Agents must always be listening for messages, and must NEVERblock, waiting for a message.

When receiving a command to execute, the node spawns a process that will execute the command. But the communication protocol adds some challenge.

### 1.5.1 Sending to distant nodes

In previous section I described how the `whereis` message works. But remember that the node must never be waiting for an answer to its message. So, when a node A wants to send a message to distant node B, it sends the `whereis` message, and puts the message for B in a table named `linc_wait`. Messages in this table are waiting for path to their recipient. So, each time a node learns a new path by receiving a `path` message, it checks if it has messages to send to this newly-discovered node. This way, the node is never waiting for the results of its `whereis` message.

### 1.5.2 Ack

Important messages require an Ack message. But we must never wait for the Ack. So, when a message is sent, we add an entry in the `linc_ack` table, noting which Ack needs to be received. When receiving an Ack, we delete it from the table. And regularly, we check if we have unreceived Ack. If so, we `shrink` the node (see previous section) and add the un-confirmed message to the `linc_wait` table, waiting for a new path to its recipient to be discovered. This way, when a node is disconnected, new path are build around it in order to keep the connected properties of the graph.

# Chapter 2

# Data

## 2.1 Store Data

When a user wants to store a file on LINC, it sends it to a node and tells how many parts the file must be split into. Then, the node cuts the file and makes pieces. The different pieces can be stored in different nodes. We do this in order to balance the load on the nodes. This way, we can spread the load over all nodes and not overload one node. A unique UUID is given to the user in order to recover the file later on.

A table containing all files stored localy is built, named `linc_files`. An entry in this table has the pattern {`UUID, Part_number, Total_nb_of_parts`}. Files are written in the hard drive, not stored in the RAM. The files are named `<UUID>.part<part_nb>`

## 2.2 Recover Data

From any node, a user can ask for a given file with its UUID. Then the User node will go into a blocking state, waiting for the full file to come. Remember that the Agents must never be in blocking state, but user nodes can. When asking for a file, the user node broadcast to the whole network the UUID of the file and to whom to send the parts. Each node will then check its stored files in `linc_files` and send all parts that match this UUID.

When the first part is received, the user can deduce the total number of parts it is waiting for and will block until all pieces are recived. There is a timeout to stop if a piece is missing or if the file is not found on the network.

## 2.3 Balance load

When storage operations are done on a node, it automaticaly tries to balance the load. It asks for the load of its direct neighbours. If a neighbour is less loaded than this node, we duplicate a part and send it to the node. When the node successfuly received the part, it sends back a message to the original node to delete the original part. We only delete the file after it is properly stored on

the new node. This is why the receiving node asks the origin node to delete it, and not the origin node deleting it automaticaly.

With this mechanism, parts are spread across the network and the load is even for all nodes.

When copying a part, there is a small chance that the part is not deleted from the original node. This way we have multiple times the same part, so if a node is lost, the data is still in another node. This duplication is automaticaly done with small probability when copied to another node.

## 2.4 Killed node and release data

When a node is properly terminated, before shutting down, it sends all its parts to an Agent neighbour. This way, no parts are lost.

Users and Agents can also send a signal to a node to release a given part or a given file. Then the nodes will delete the parts corresponding.

# Chapter 3

# Monitoring

From a unique node, one can spawn Agents in remote nodes only knowing their name and cookie. All commands to do so are described in the Command chapter.

When using a shell to spawn the topology, distant nodes will display debug informations in this shell.

A user can connect to the topology and ask for statistsics. For now, a user can only ask for the load and the neighbours list of all nodes in the topology.

# Chapter 4

# Commands and Control

See the Readme.md for cleaner explanations

## 4.1 Setup and deploy

### 4.1.1 Prepare the processes

If no distant shells are already running the command `make spawn` will create detached processes ready to receive an Agent. One can also give the `NB_AGENT` variable to specify the number of agents to spawn. For example `make spawn NB_AGENT=15` will spawn 15 processes. Processes spawned this way are named `n<id>@<host>` with `<id>` going from 1 to NB_AGENT and `<host>` the local host name.

All these names are using short names. In order to connect to a truely remote node, change the Makefile and replace -sname with -name. -sname is shorter and easyer to use for local demos.

### 4.1.2 Spawn the Agents

We then need to spawn the Agent code on these nodes. To do so, we use the `setup` module. Enter `make ctrl` to open a control shell. It will be used to spawn the remote code and display the debug informations of the remote nodes.

In the shell,

- `c(setup).` : Compile the module
- `setup:spawn_all(N).` : Auto spawn N nodes on remote nodes that were spawned with the Makefile (using the Makefile notation for nodes name.). For example, after a `make spawn NB_AGENT=15`, use `setup:spawn_all(15).` to bootstrap all nodes.
- `setup:bootstrap(Node)` : Bootstrap a specific node
- Now we must build a connected topology. For now, only one function is available : `setup:build_ring()` automaticaly builds a ring on all spawned nodes. LINC does not need to be a ring but it is an easy topology to start with and then transform it into a more complex one.

Once all this is done, the control node can stay opened in order to display the debug.

### 4.1.3 Spawn the user

We must now spawn the user in a shell in order to interact with the topology. `make user` starts a new shell.

The user node uses module `usr`.

In this shell,

- `c(usr).` : Compile the module
- `usr:setup().` : Prepare the node, spawns listening processes and tables so that this node can interact with the topology.
- `usr:link(Node).` : Connects to the node Node.

Once this is done, the User node is ready to receive commands from humans.

## 4.2 Topology control commands

All commands can be sent to LINCby using `usr:sendCommand()` function. Once a node receives a commad, it spawns `command:<command>(<arguments>)`. So all possible commands are in the `command` module.

It can take several arguments :

- `usr:sendCommand(<command>, [<arguments>]).` Sends the command with given arguments to the first node User is connected to.
- `usr:sendCommand(<target>, <command>, [<arguments>]).` Same but this time the command is sent to and executed on node ¡target¿. If Target is `all`, the command will be spawned on all nodes.
- `usr:shutdown(Node)` cleanly shutdowns an Agent. This does not kill the erl process, only the Agent processes. This way we can spawn back an Agent on the node.
- `usr:sendCommand(<entry>, <target>, <command>, [<arguments>]).` Same but specifies the node to be used as entry point on the topology.

There are a few topology commands that can be sent :

- `usr:sendCommand(Target, link, [Node])` : Add a link between the node executing this command and node Node.
- `usr:neighbours()` : Asks all nodes to send their direct neighbours to this User node. When the answer is received, it will automaticaly be displayed in the user shell

## 4.3 Data Management Commands

Commands to manage data :

- `usr:store_file(<file_name>, <nb_parts>` will store the file and split it into parts. This returns {`ok, UUID`} if success.
- `usr:recover_file(<UUID>, <file_name>` will gather the pieces of the file and, once all are received, save the file under name ¡file_name¿
- `usr:sendCommand(Target, balance, [])` : Will force the balance of parts of node Target. (remember, Target can be `all`)
- `usr:load()` will ask all nodes to display their load on the user shell.

More commands are available, see the `command.erl` file for a complete list.

## 4.4 Cleaning

After the tests, all files are stored in folder `nodes/`. To clean all files, just use `make clean`

Spawned processes are hard to kill because they are detached and have no shell. Command `make kill` will kill all erlang processes.

# Chapter 5

# Conclusions and work left to do

This topology is stable, works perfectly fine, is fully decentralized with a random topology. But some improvements are still possible and may be implemented in the future :

- Load balancing with stored size : For now, nodes balance the number of parts stored, not the stored size. An improvement would be to consider the size.
- Crash resistance : When an node crashes, paths can be rebuilt with the protocol described in previous chapters. But if too many nodes crash, it is not recoverabl. I'm thinking about another protocol to handle this.
- Distributed computations : It seems not that hard to distribute computations on this network because spawned functions are spawned on different nodes, file parts can be replaced by data to treat or programs to run. It is possible and could be fun to try.