Correction in distributed systems, from usage to reconfiguration

Progress and insights from my research journey

Jolan Philippe

January 24, 2025 - Seminaire STR

DiverSE team, IRISA (Univ. Rennes)

Hello world

Jolan Philippe, Postdoc

Université de Rennes, DiverSE Team

$(\lambda x.\lambda y.x@y)$ Jolan.Philippe inria.fr

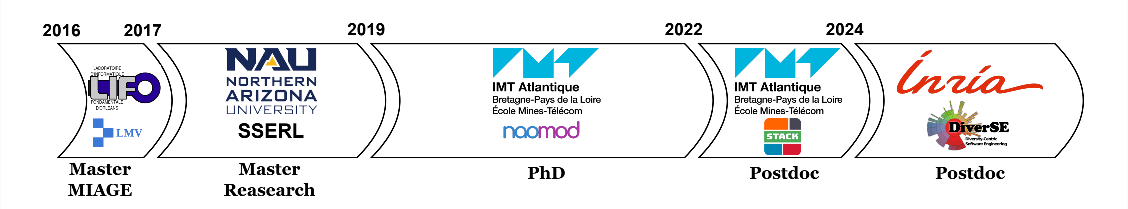
Topic of interest:

Correctness in

- Distributed computing
- Model driven engineering
- Reconfiguration

More details on: `https://jolanphilippe.github.io/`

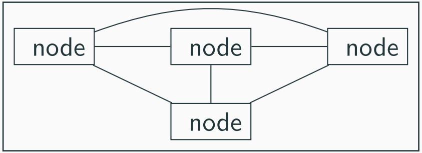
Disclaimer



All the work presented in this talk is from different projects, contexts, and people.

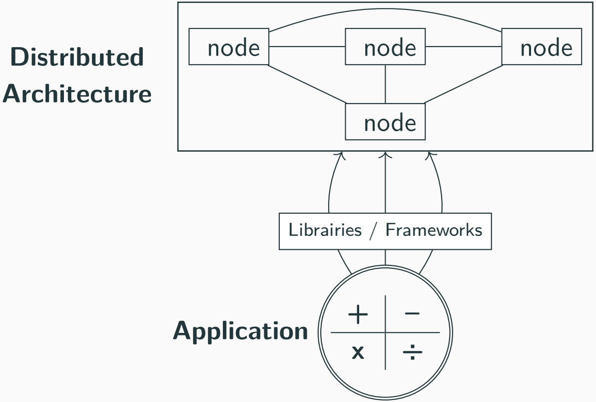
Context: Using distributed infrastructure

Distributed Architecture



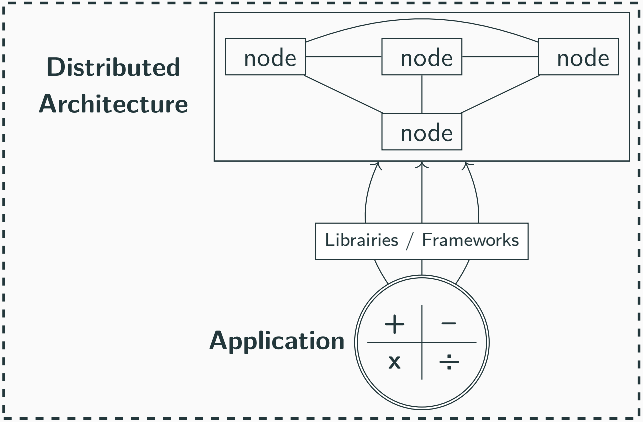
- Different topologies
  - Ring
  - Star
  - Bus
  - etc.

Context: Using distributed infrastructure



- Different topologies
  - Ring
  - Star
  - Bus
  - etc.
- Different architecture
  - Single Instr. Single Data
  - Single Instr. Multiple Data
  - Multiple Instr. Single Data
  - Multiple Instr. Multiple Data

Context: Using distributed infrastructure



Configurable

- Configuring architecture
  - Parametrized ressources
  - Services
- Configuring application
  - Allocated ressources
  - Features

⇒ DevOps perspective
  - Continuous Integration
  - Continuous Deployement

Challenges

Developing and managing application on distributed architecture is error-prone

Ensuring correctness

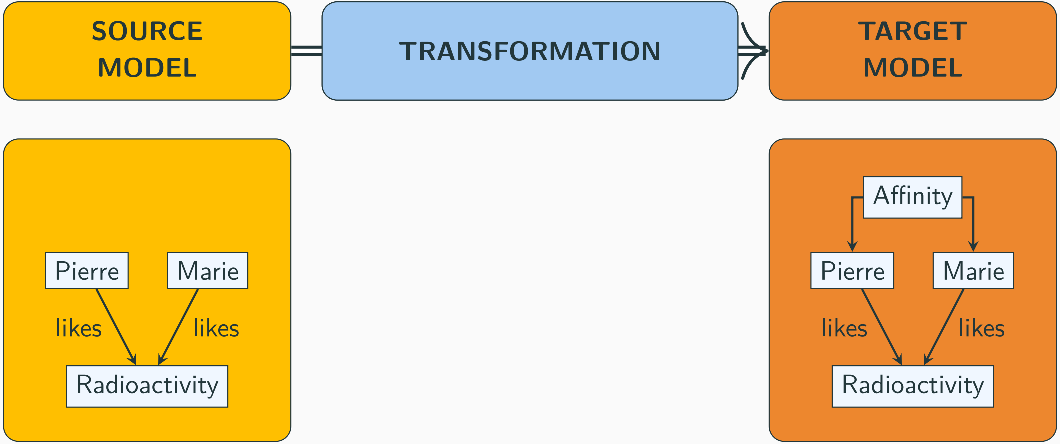
Goal: Concrete application meets expectations, at different level:

- The application itself
- The used libraries / frameworks
- The reconfiguration

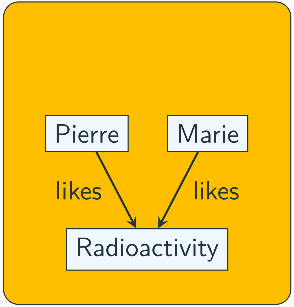
Hint : Using formal approaches

Running example - A distributed model transformation engine

Running example - A distributed model transformation engine

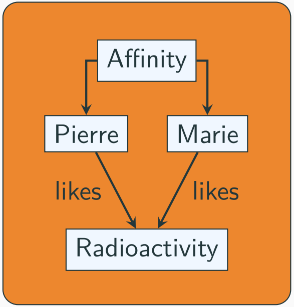


Running example - A distributed model transformation engine

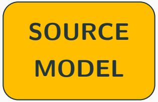


rule copy (e:Element)
output:
new Element(content $\leftarrow$ e.content)

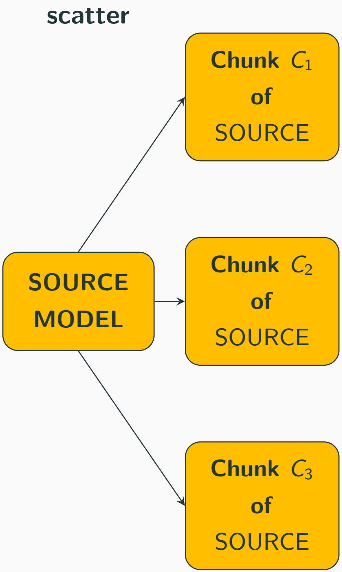
rule affinity (e_1:Element, e_2:Element)
guard:
query(e_1, likes) $\cap$ query(e_2, likes) $\neq \varnothing$
output:
new Affinity(from $\leftarrow e_1$, to $\leftarrow e_2$)



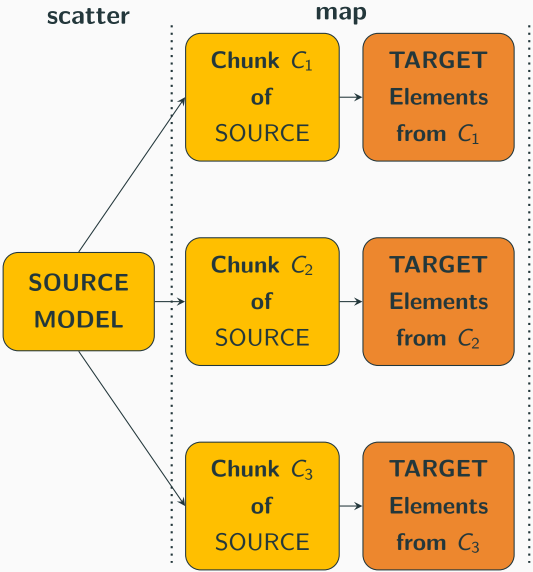
Running example - A distributed model transformation engine



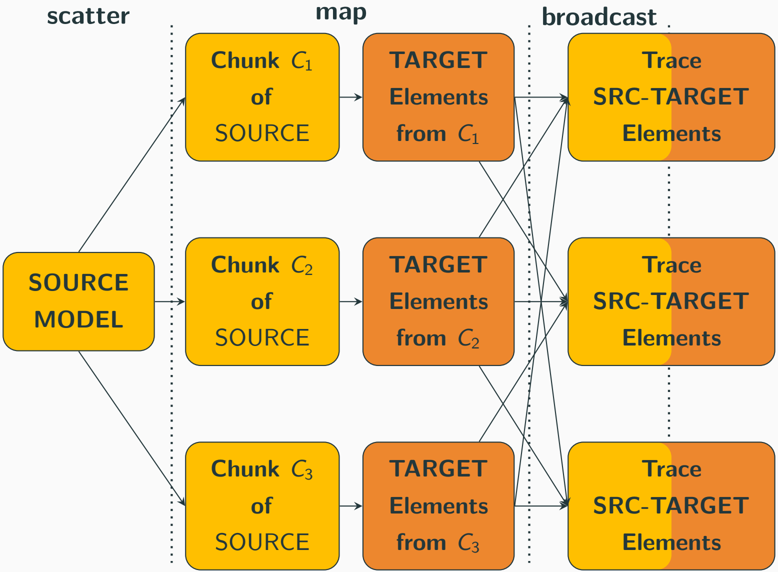
Running example - A distributed model transformation engine



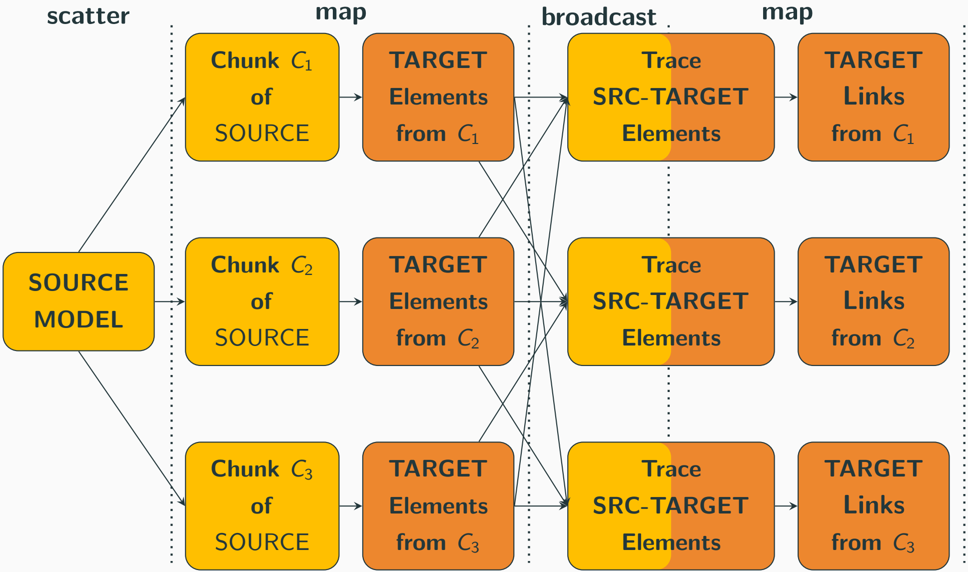
Running example - A distributed model transformation engine



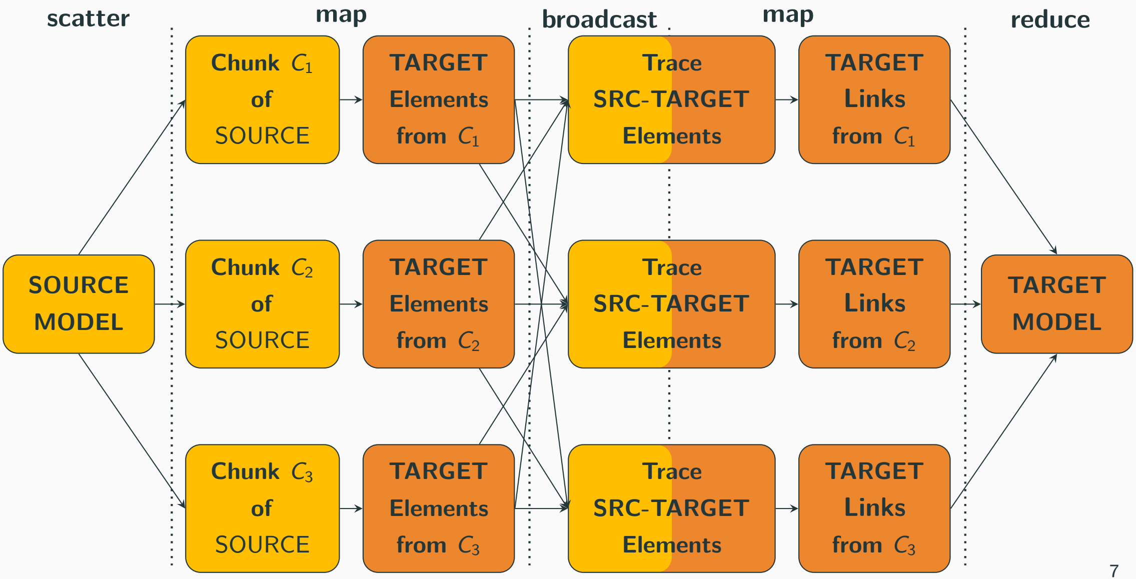
Running example - A distributed model transformation engine



Running example - A distributed model transformation engine



Running example - A distributed model transformation engine



Outline

1. **SparkTE, a correct-by-construction model transformation engine**
    - A configurable engine
    - Running correct-by-construction transformations
2. **Verifying frameworks for distributed calculation**
    - Skeletons and correctness
    - A Coq library: SyDPaCC
    - SyDPaCC for Spark
3. **Coordinated reconfiguration**
    - Complex architecture for SparkTE
    - Decentralized reconfiguration
    - Ballet for reconfiguring
    - Model-checking on Ballet

SparkTE - A configurable engine

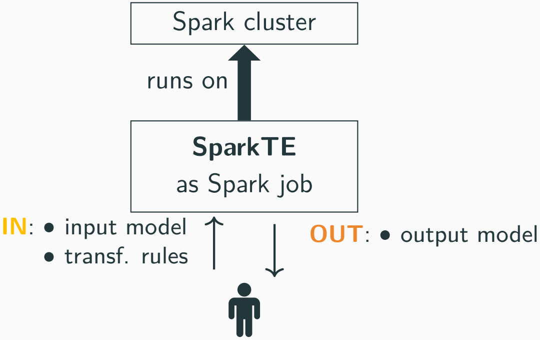
Distributed model transformation engine

- Offers scalability
- Based on Apache Spark
  - Popular framework for large-scale data processing
  - Support for many paradigms
  - Open-source

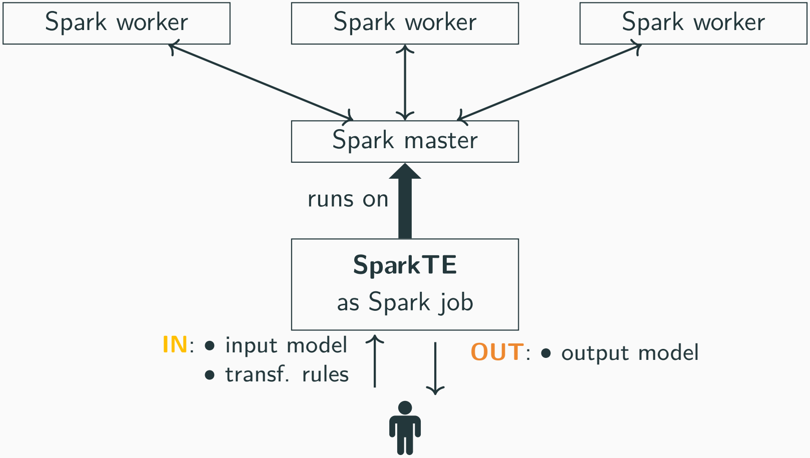
Highly configurable transformation

- Several possible execution semantics
- Multi-paradigm approach for querying input model
- Engineering design choices configuration

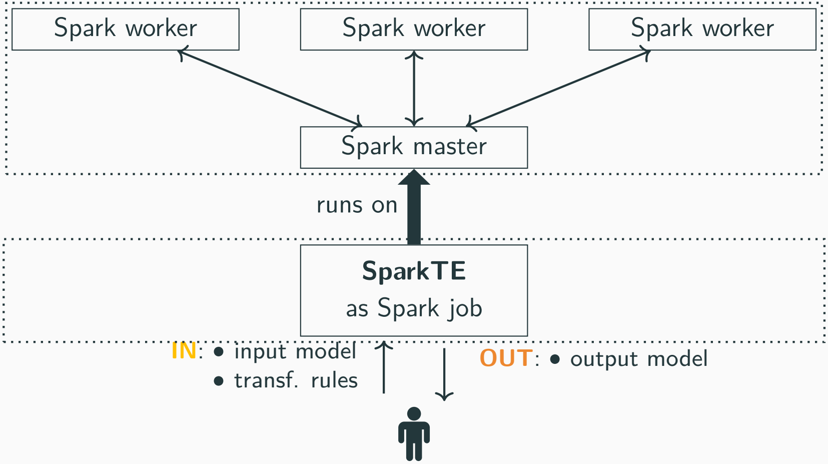
SparkTE - A configurable engine



SparkTE - A configurable engine



SparkTE - A configurable engine



Configure

- memory
- num workers
- num threads
- etc.

Configure

- queries on input model
- semantics execution
- engineering choices

SparkTE - Correct-by-construction transformations

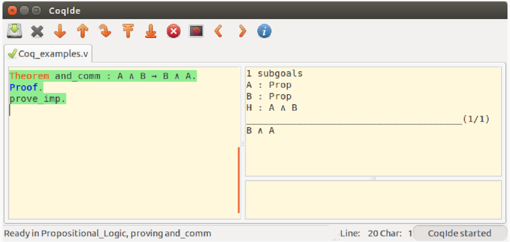
General approach for reasoning on model transformation

- Formalize transformation in the proof assistant Coq
- Refine the formalization for performances
- Extract a running engine

  ⇒ Extract spec. into Scala code

  ⇒ Run Scala code on Spark Cluster

Use of CoqTL for reasoning

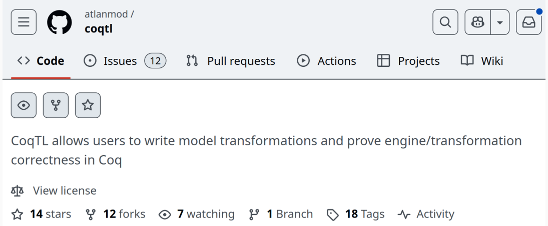


The Coq proof assistant

- Designed for specifying semantics
- A proof assistant based **calculus of constructions** and **Hoare logic**
- **Extraction** mechanism (to ML langs)

CoqTL

- **DSL** for rule-based model transformation
- Made for **reasoning on model transformations**
- Can **reason on the semantic** of the transformation



Correct-by-construction: Parallelizable CoqTL

Increase parallelization

1. Two distinct phases : **instantiate** and **apply**
   - Defined as map-reduce phases
2. Iterate on rules instead of source patterns
   - Avoid unnecessary computations
3. Iterate on trace for apply instead of source patterns
   - Reuse intermediate results while everything is redefined in CoqTL

	spec (loc)	cert (loc)	effort (man-days)
1.	69	484	10
2.	42	487	7
3.	69	520	4

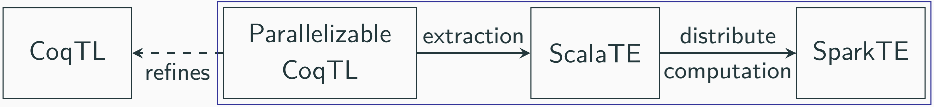
Correct-by-construction: Build SparkTE

Scala

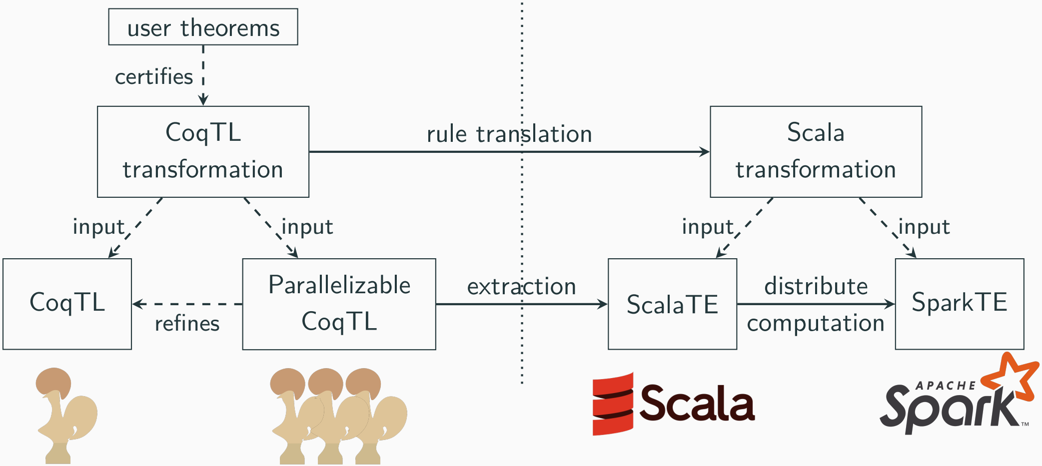
APACHE Spark

CoqTL to SparkTE

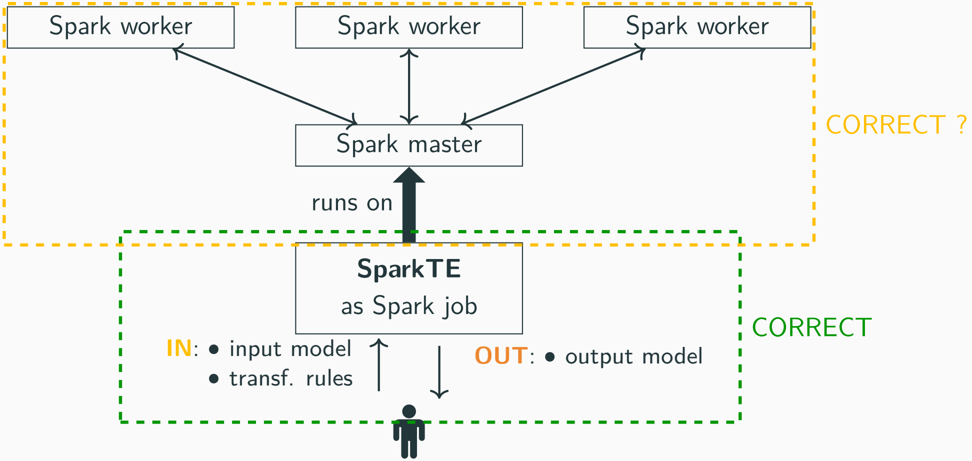
1. Produce executable and maintainable code
   - By hand: Object-oriented approach, with pure Scala functions
   - With Scallina: Not maintainable, but certified
2. Distribute the computation
   - Distribute data-structure
   - Explicit communication operations (scatter, broadcast and reduce)



Correct-by-construction



Correctness... really ?



Spark

Certify Spark

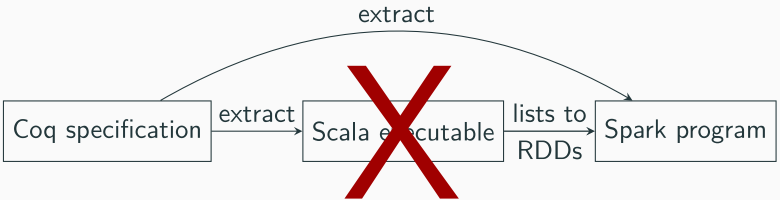
Spark defines program as

- Usage of high-order functions (e.g., *map*, *reduce*)
- Using a distributed implementation (i.e., skeletons)
- Considering the sequential implementation equivalent to distributed one

```
val instantiatedElements =
  transformation.rules.map { rule => instantiate(model.elements, rule) }
```

No guarantee that the parallel implementation behaves the same as the sequential implementation.

Coq to Spark



Extract Coq code into Spark program

- Formalize Spark's distributed structure (i.e., RDD) in Coq
- Formalize computation on RDDs
- Prove the equivalence between function on lists and on RDDs

SyDPaCC for formalizing skeletons

SydPaCC

- Coq library for writing data-parallel program specification
- Code can be extracted into BSML (BSP for Ocaml)
- Ensure the correctness of the extracted parallel program
- Based on type equivalences (with composition)

$$
\begin{array}{ccc}
A_p & \xrightarrow{f_p} & B_p \\
{\scriptstyle join_A}\downarrow & & \downarrow{\scriptstyle join_B} \\
A & \xrightarrow{f} & B
\end{array}
$$

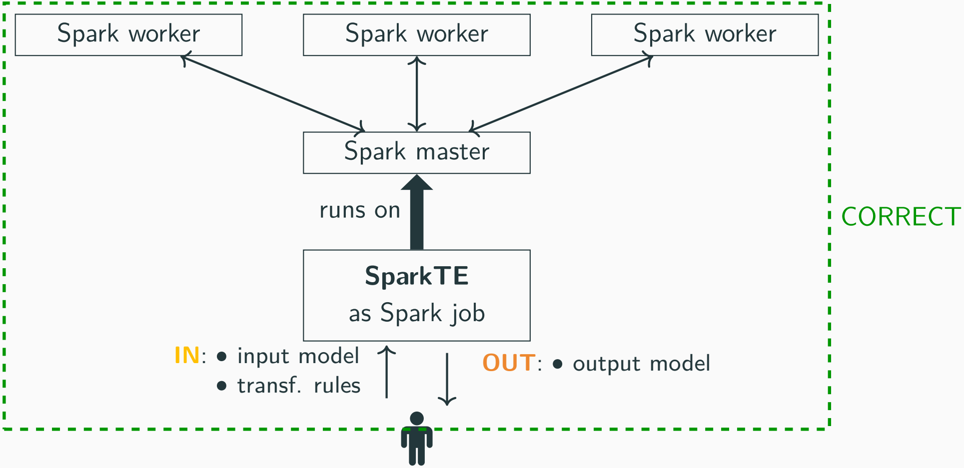
Example - Equivalence of map

$$\begin{array}{ccc} A_p & \xrightarrow{f_p} & B_p \\ join_A \downarrow & & \downarrow join_B \\ A & \xrightarrow{f} & B \end{array} \qquad \begin{array}{ccc} RDD[A] & \xrightarrow{RDD.map} & RDD[B] \\ collect \downarrow & & \downarrow collect \\ List[A] & \xrightarrow{List.map} & List[B] \end{array}$$

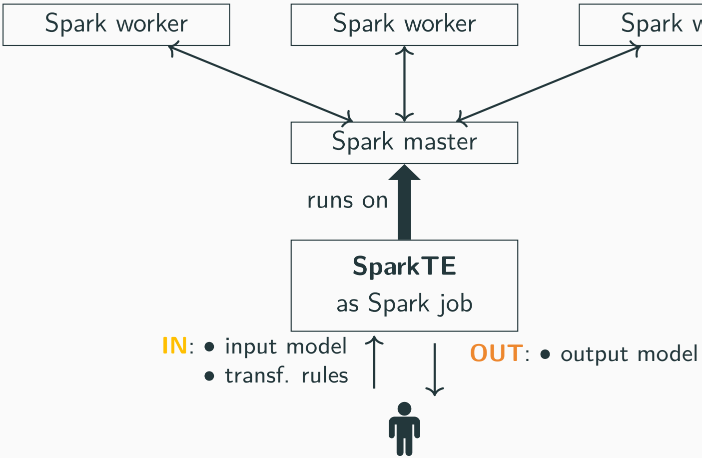
Extending SyDPaCC

- Formalize RDDs
- Additional proofs
  - $RDD.map \circ collect = collect \circ List.map$
  - Surjectivity of *collect*

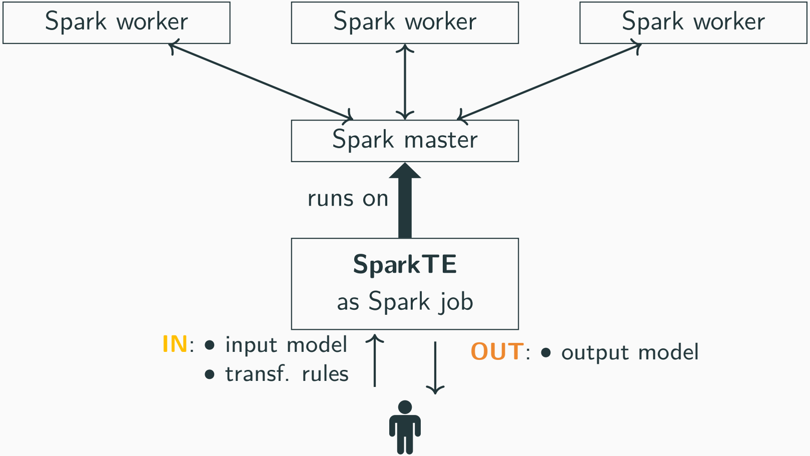
Towards verified parallel computing with Coq and Spark



SparkTE, as a "prototype", architecture



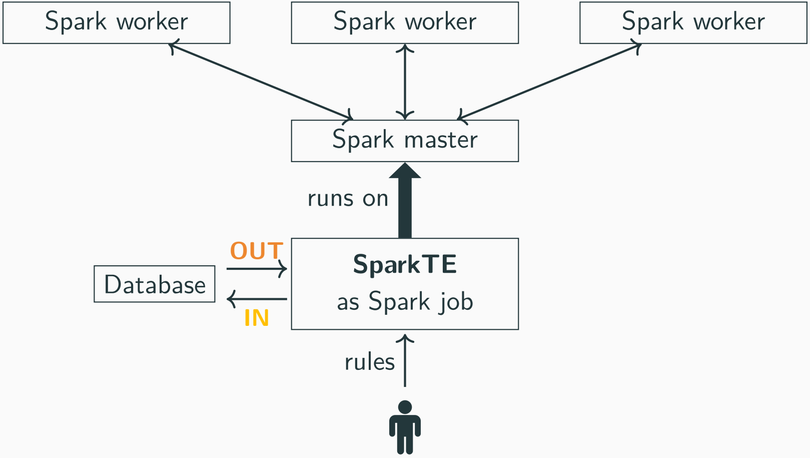
SparkTE, as a “production tool”, architecture



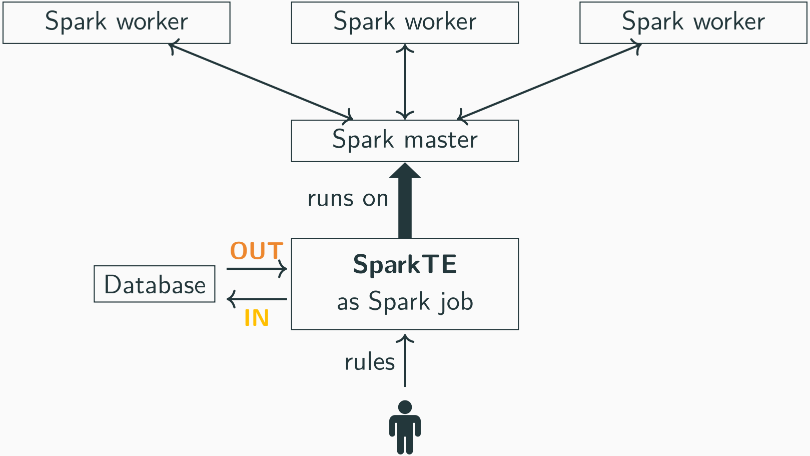
We want to store the model within a database

NB: We could use HDFS files alongside Apache Hive

SparkTE, as a “production tool”, architecture



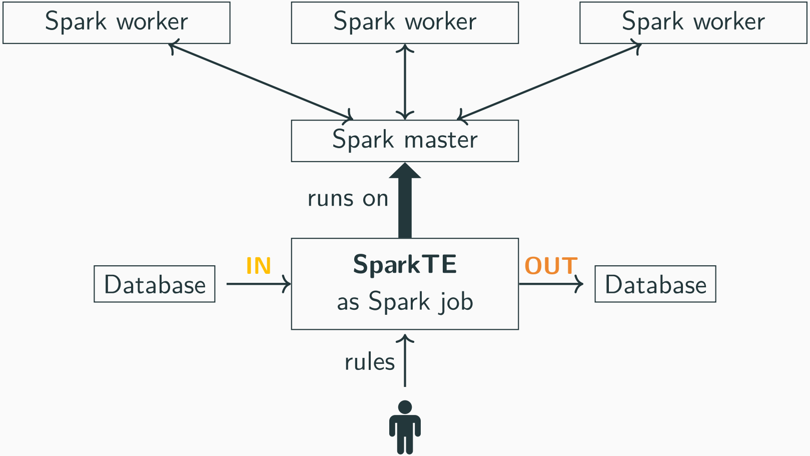
SparkTE, as a "production tool", architecture



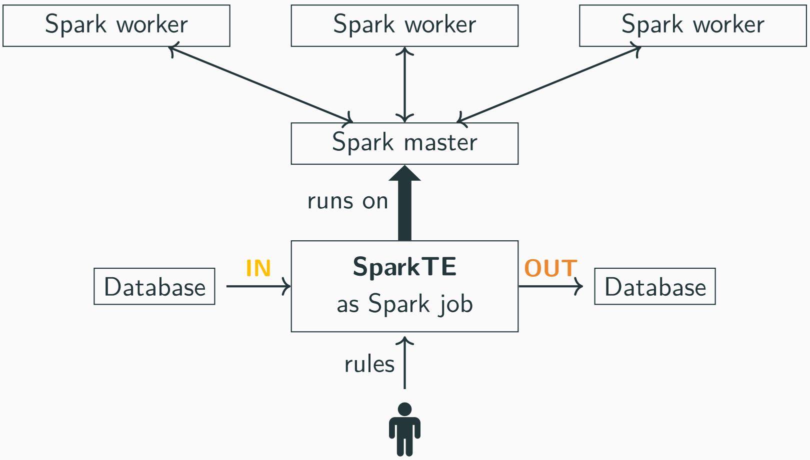
Since the transformation is not necessarily "in-place"

we might want to store the input model and output model in different databases

SparkTE, as a "production tool", architecture

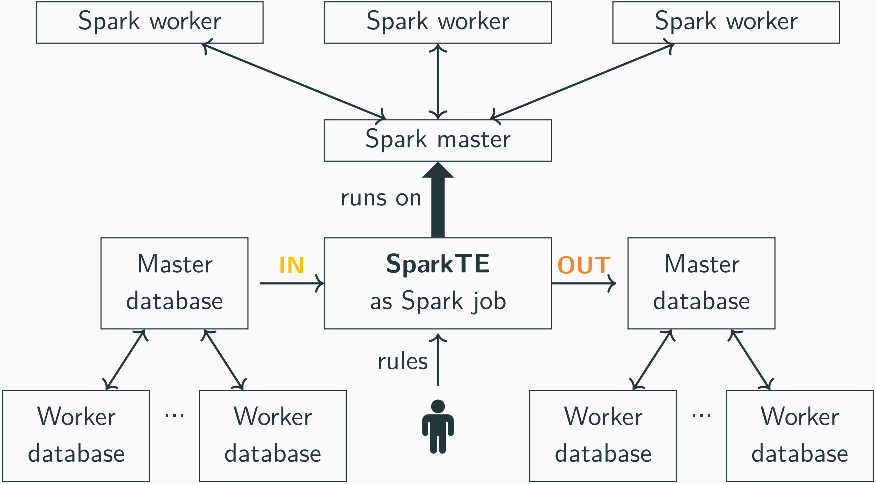


SparkTE, as a “production tool”, architecture

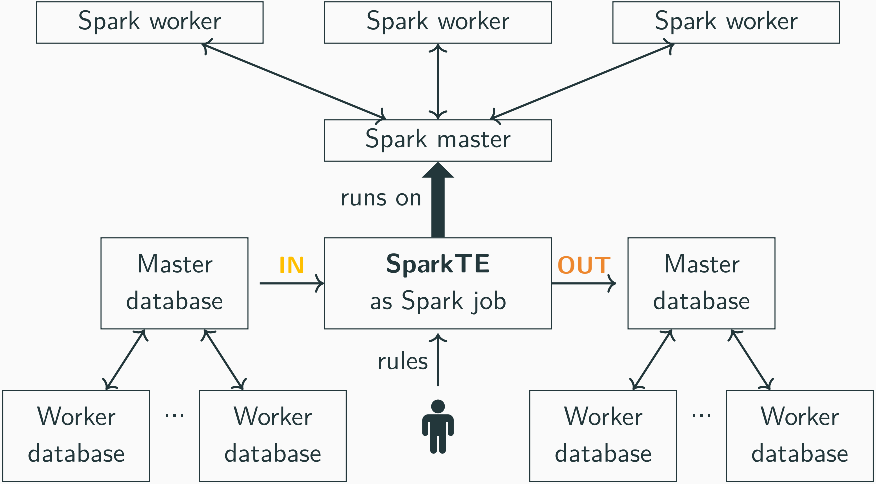


Because we handle very large model, we want distributed databases

SparkTE, as a "production tool", architecture

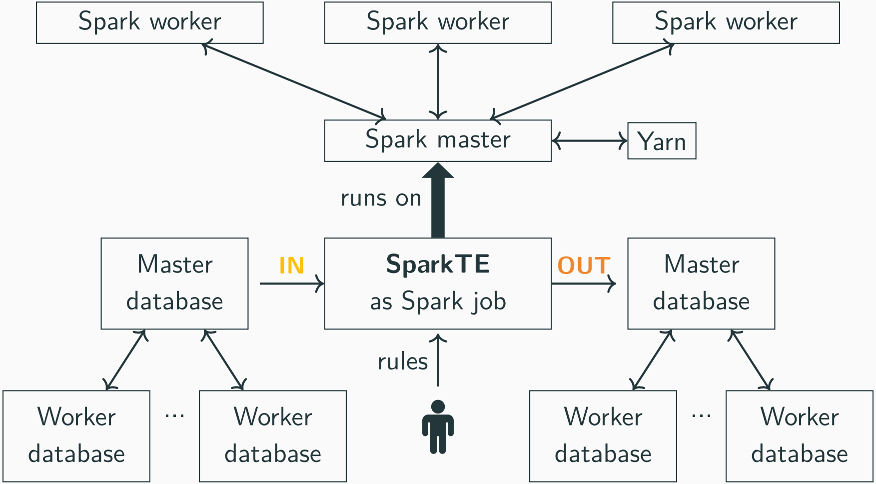


SparkTE, as a "production tool", architecture



We want more fine-grained management of Spark resources using Yarn

SparkTE, as a "production tool", architecture

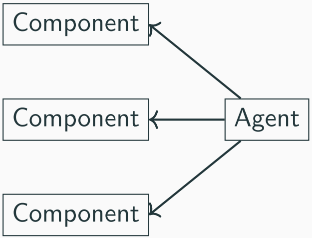


Deploy and reconfigure SparkTE

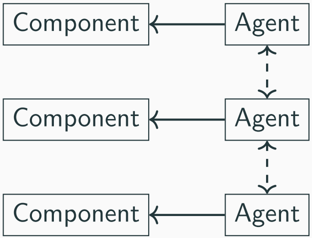
Two approaches for reconfiguring distributed systems

Reconfiguration : Change of the state of entities by applying operations (e.g., deploy, update, destroy)

- Centralized: single agent manages the reconfiguration with control components
- Decentralized: several agents manage the reconfiguration with control components



Centralized approach



(Fully) Decentralized approach

Decentralized approach

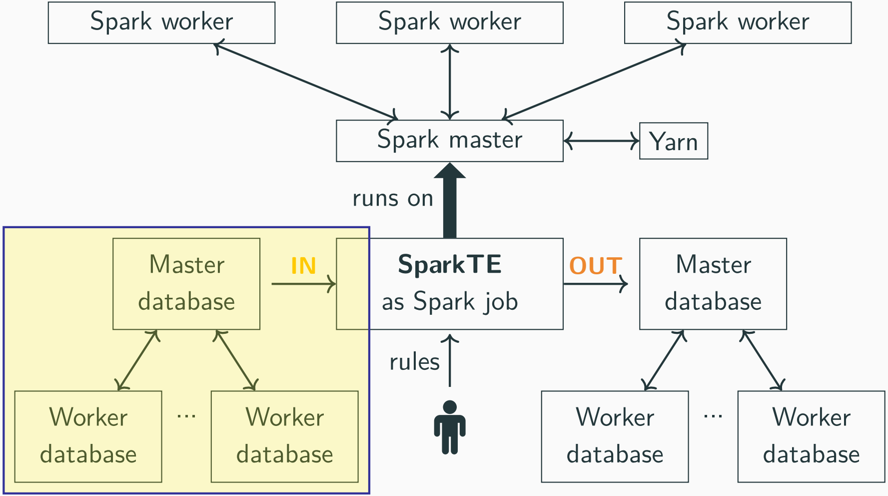
Strength

- Not a single point of failure
- Separation of information
- Scalability
- Allow geo-distribution

Challenges

- All agents must coordinate
- Operate communications

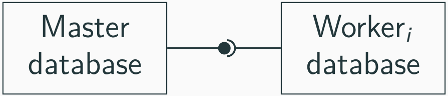
Reconfiguring distributed databases



How to reconfigure ?

Let's have a look

Example: Deploy distributed databases



Database Master (DM) plan

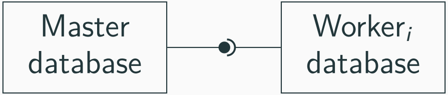
1. Configure the service
2. Bootstrap the database
3. Start the service
4. Expose API

Database Worker$_i$ (DW$_i$) plan

1. Configure the service
2. Register to master
3. Bootstrap the database
4. Start the service
5. Expose API

- **Component granularity:** DM $<<$ DW$_i$
- **Lifecycle granularity:** DM(4) $<<$ DW$_i$(2) (partial order)

Example: Update distributed databases



Database Master (DM) plan

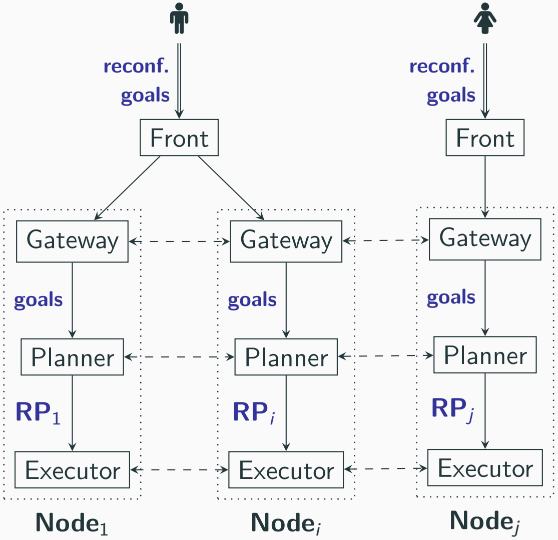
1. Interrupt the service
2. Make the update
3. Start the service
4. Expose API

Database Worker_i (DW_i) plan

1. Interrupt the service
2. Make the update
3. Register to master
4. Start the service
5. Expose API

- **Component granularity?** Destroy DW_i << Update DM << Deploy DW_i
- **Lifecycle granularity:** $\text{DW}_i(1)$ << DM(1) & DM(4) << $\text{DW}_i(3)$

Ballet for decentralized reconfiguration



- Decentralized tool (one instance of Ballet on each node)
- Declarative input as goals
- Reconfiguration with automatic planning and efficient execution

Gateway

Global knowledge building of reconfiguration goals

Planner

Decentralized inference of reconfiguration plans (RPs)

Executor

Coordinated execution of RP

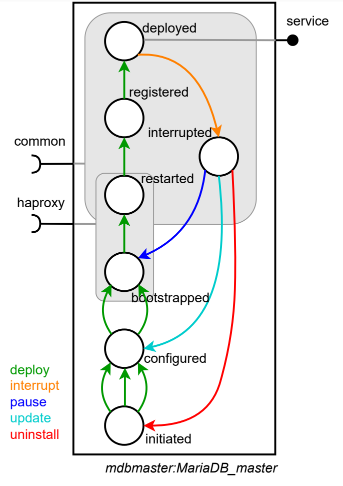
Specify lifecycle and dependencies for control components

Lifecycle and dependencies

- **Places**: milestones of the reconfiguration
- **Behaviors**: interface for executable actions
- **Transitions**: concrete actions between places, associated to behaviors
- **Ports**: Provide (resp. use) information to (resp. from) external components
  - Ports are bounded to places and transitions

Exemple: Database for SparkTE

Lifecycle representation of a MariaDB database with 5 executable behaviors: deploy, interrupt, pause, update, and uninstall



Usage of Ballet : goals

Reconfiguration goals

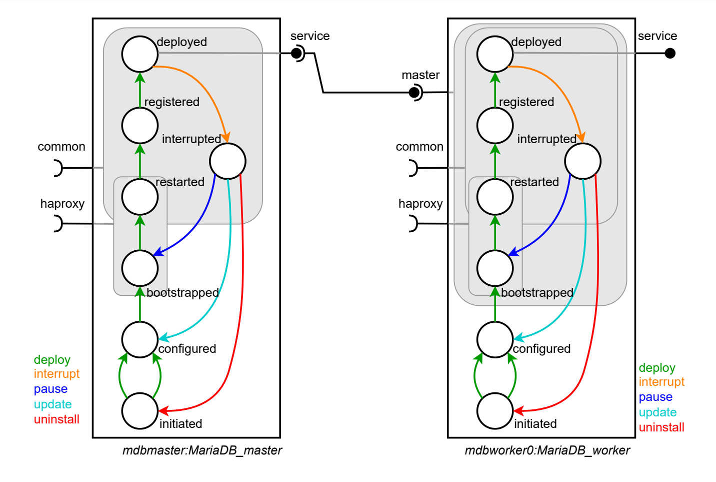
Declarative language for defining reconfiguration goals

- **Behavior goal**: Specify a behavior that must be executed
- **Port goal**: Specify a port status (active, inactive)
- **State goal**: Specify a component state (specific, running, initial)

Listing 1: Language to define reconfiguration goals for DevOps usage

```
<goals> ::= behaviors : <bhvr_list>
            ports : <port_list>
            components : <comp_list>
<bhvr_list> ::= ...
<bhvr_item> ::= - forall : <bhvr_name>
              | - component : <comp_name>
                  behavior : <bhvr_name>
<port_list> ::= ...
<port_item> ::= - forall : <port_status>
              | - component : <comp_name>
                  port : <port_name>
                  status : <port_status>
<comp_list> ::= ...
<comp_item> ::= - forall : <comp_status>
              | - component : <comp_name>
                  status : <comp_status>
```

Assembly of components



Assembly of MariaDB master and worker components

Similar to synchronous Petri nets

Concerto-D language

A simple language to interact with components - i.e., write a reconfiguration plan

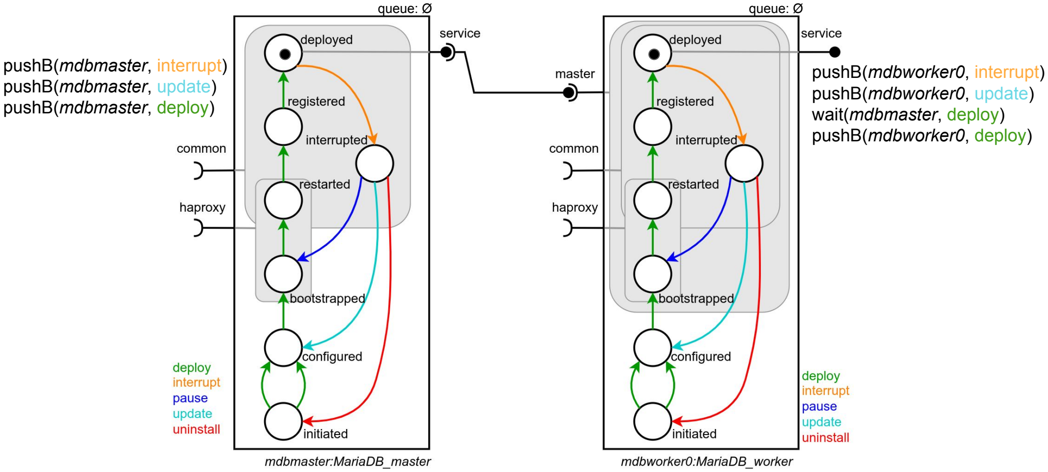
- **Add/remove** a component instance to the current assembly
- **Connect/disconnect** two component instances with compatible ports
- **Push behavior** to the behavior queue on a component instance

$$\text{pushB}(id_C, bhv)$$

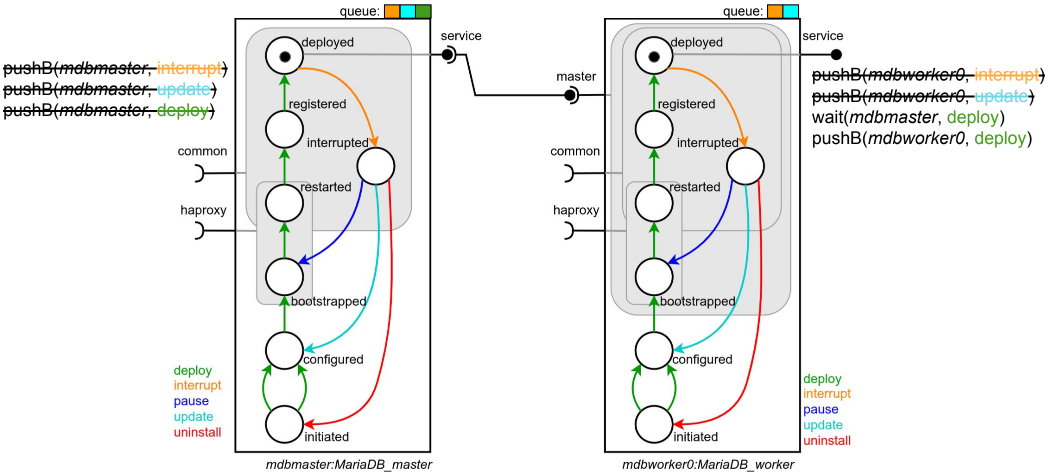
- **Wait** for a given component instance to execute a behavior

$$\text{wait}(id_C, bhv)$$

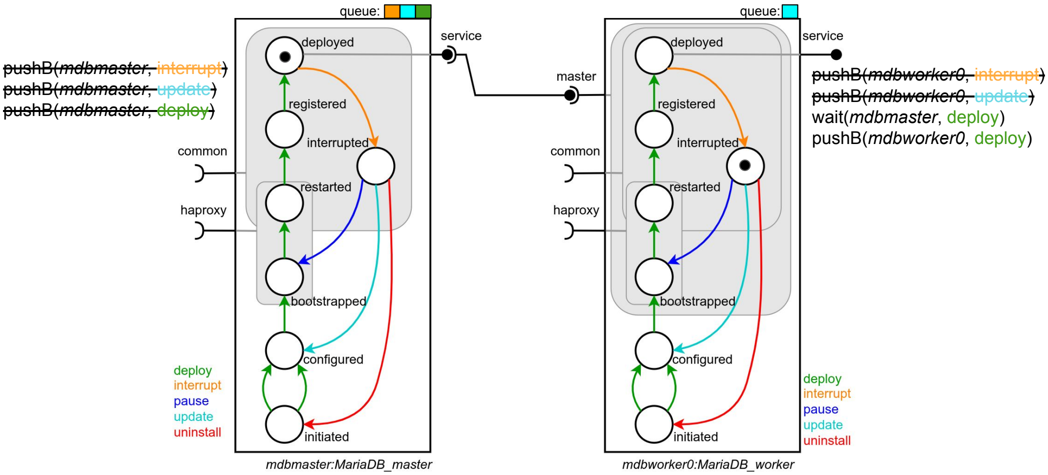
Execution example



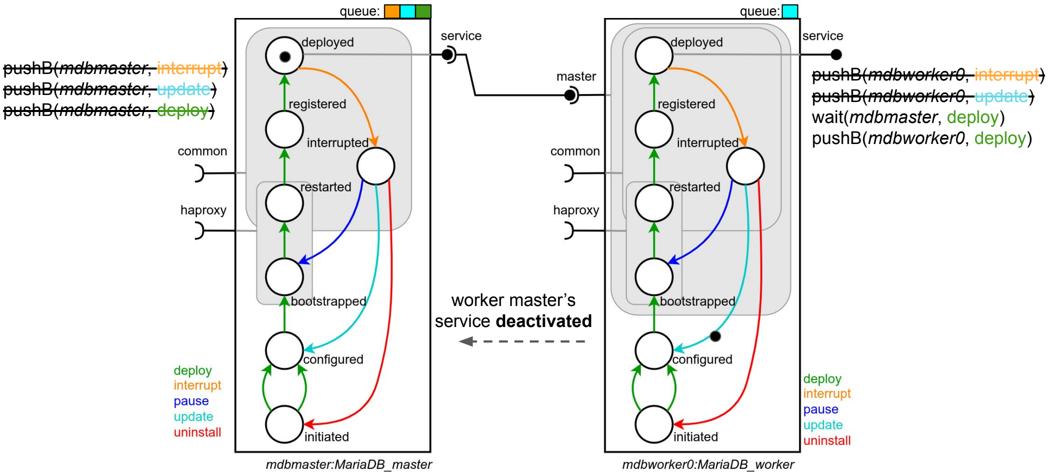
Execution example



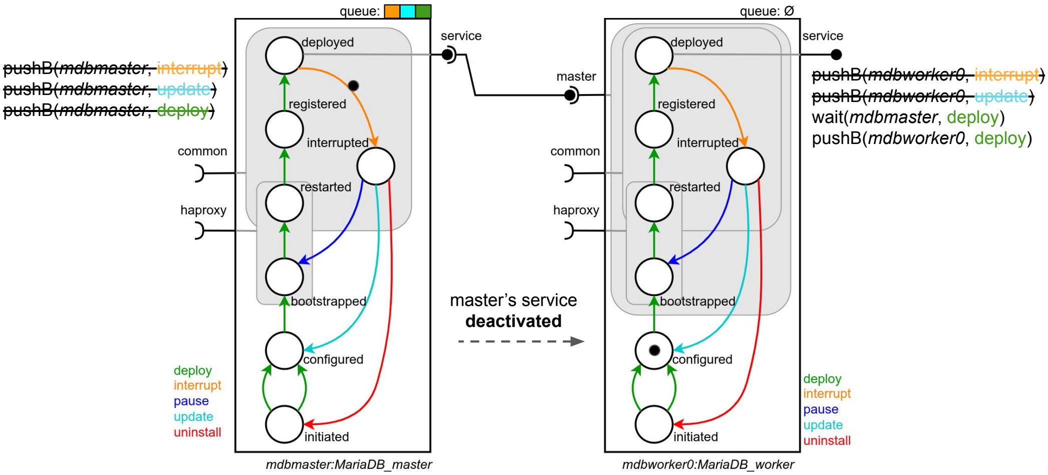
Execution example



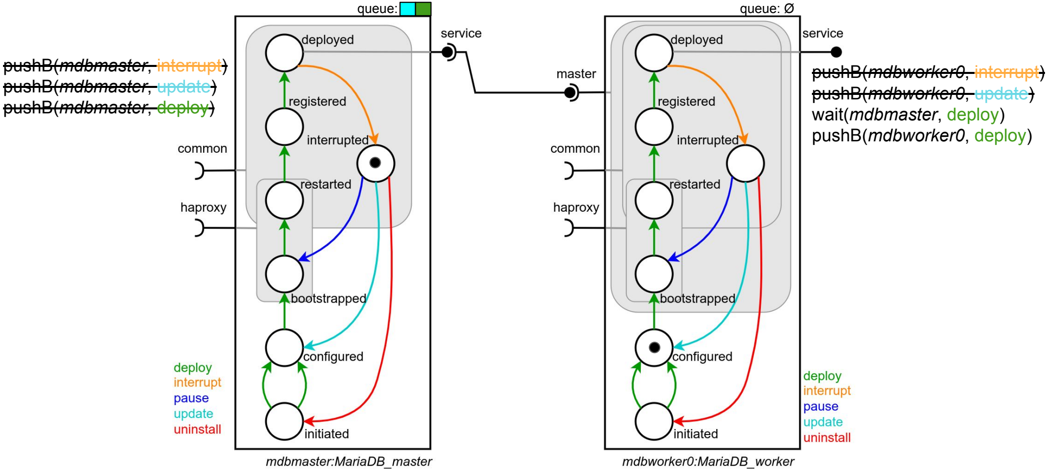
Execution example



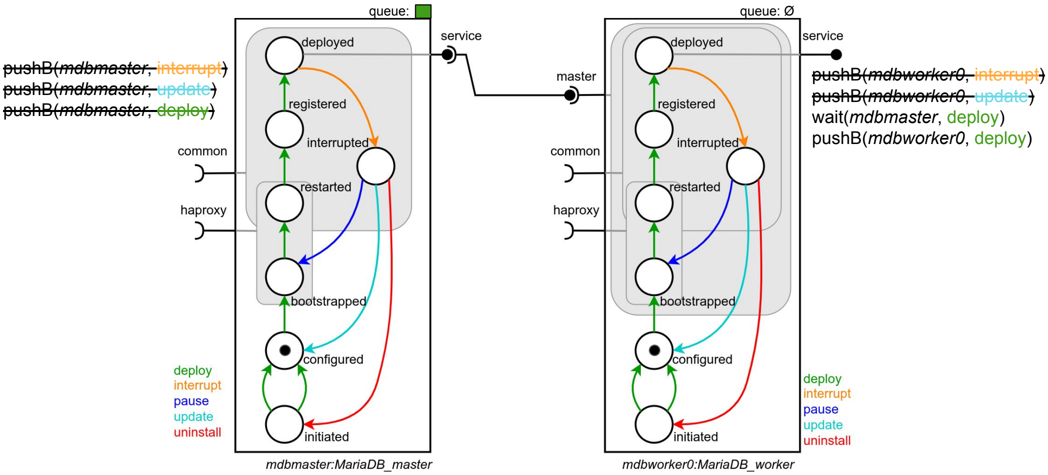
Execution example



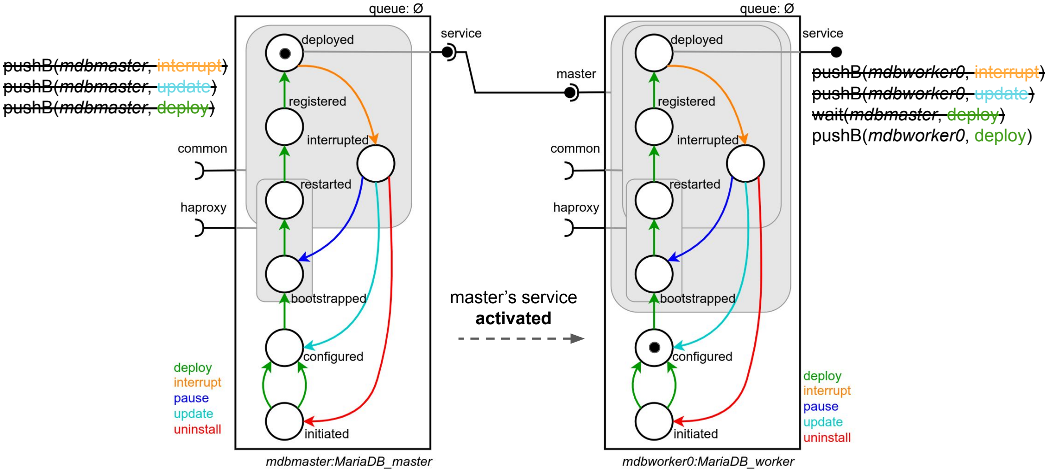
Execution example



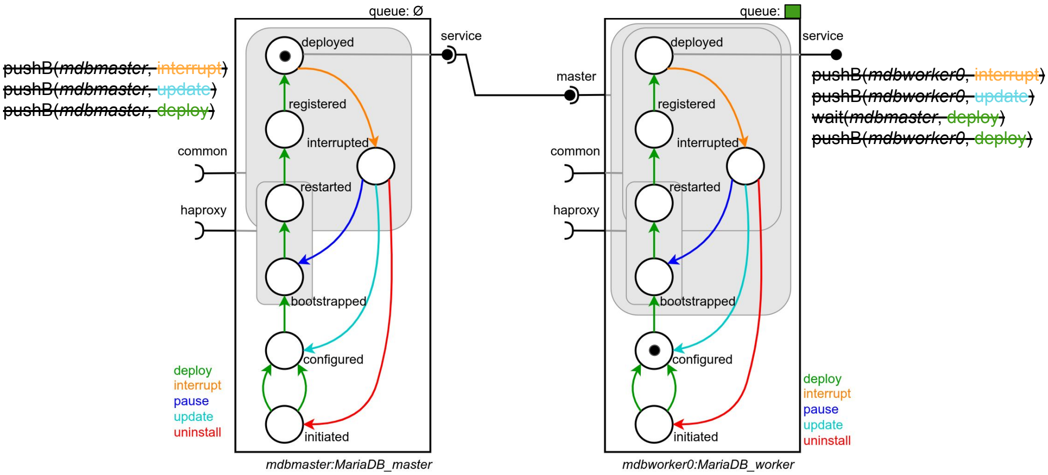
Execution example



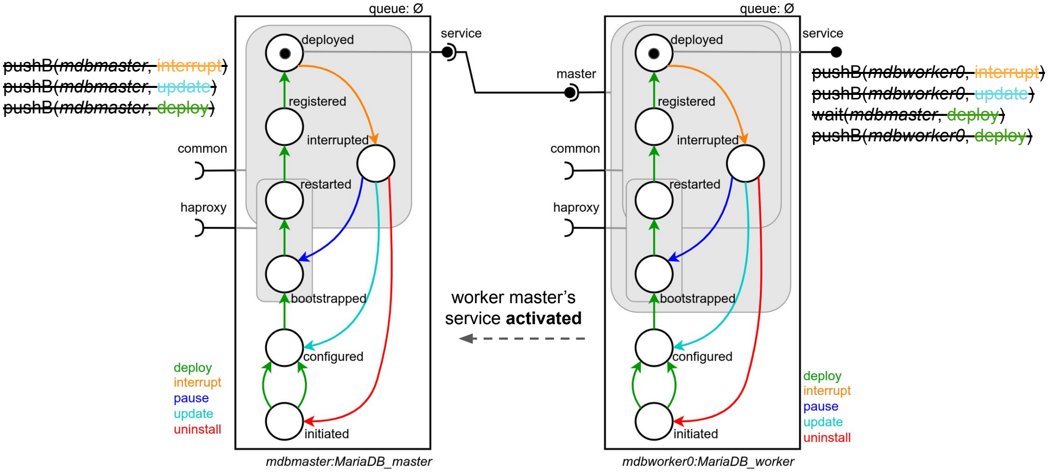
Execution example



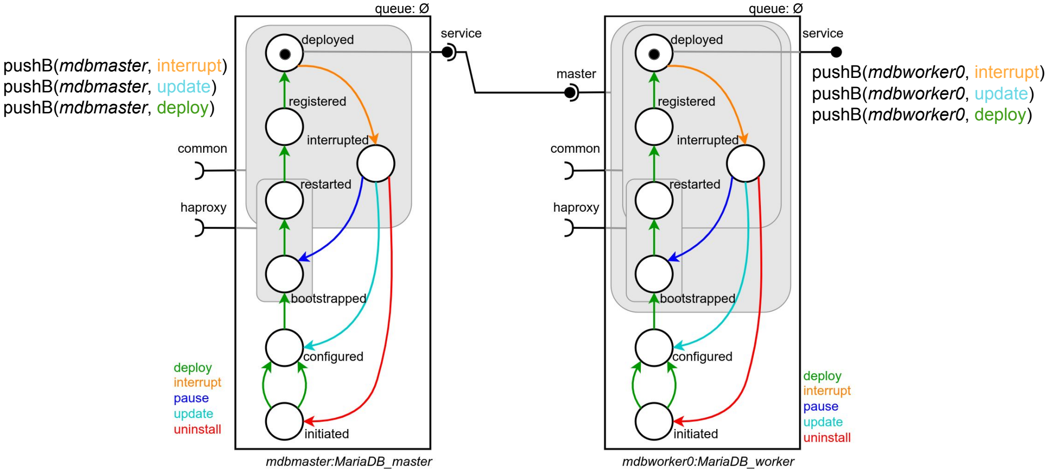
Execution example



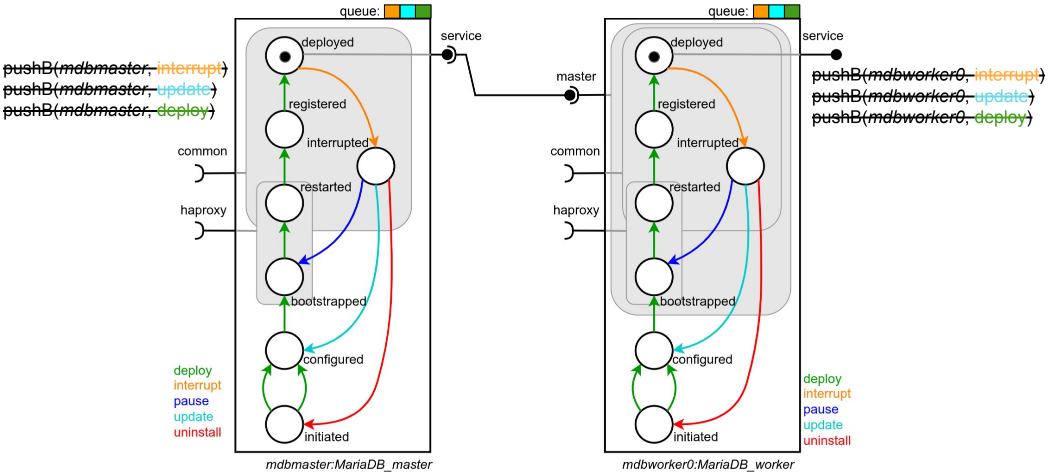
Execution example



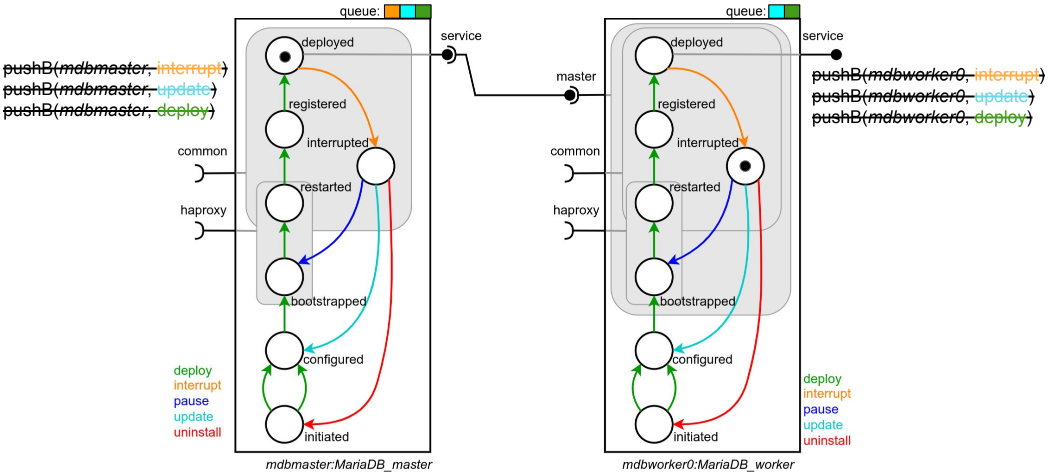
Without the wait ? Failing execution



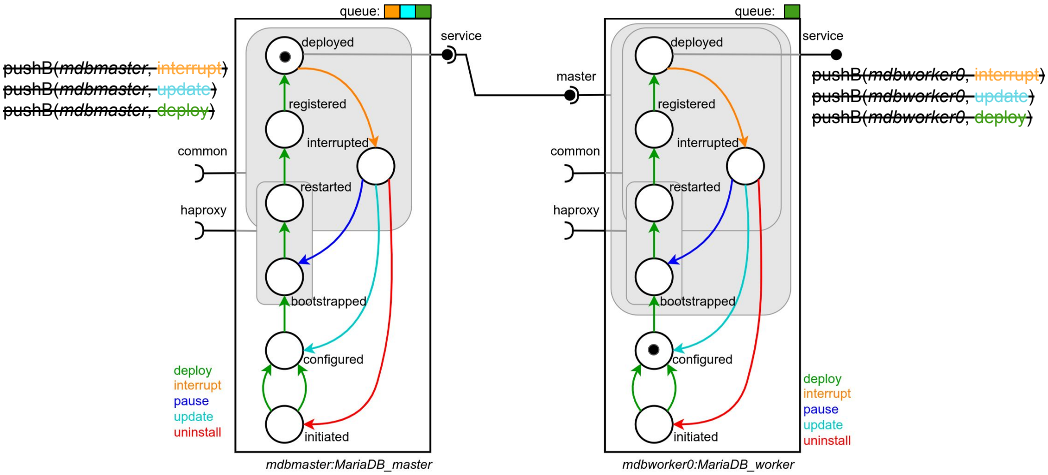
Without the wait ? Failing execution



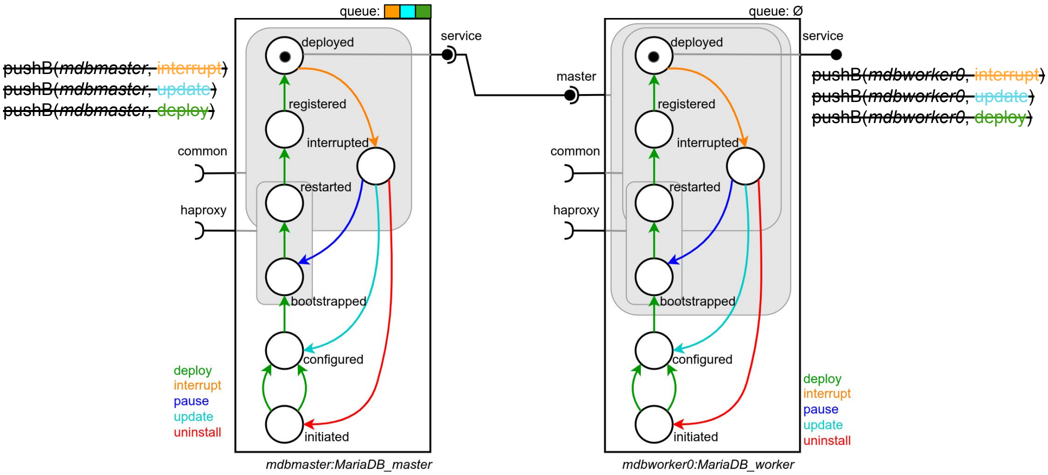
Without the wait ? Failing execution



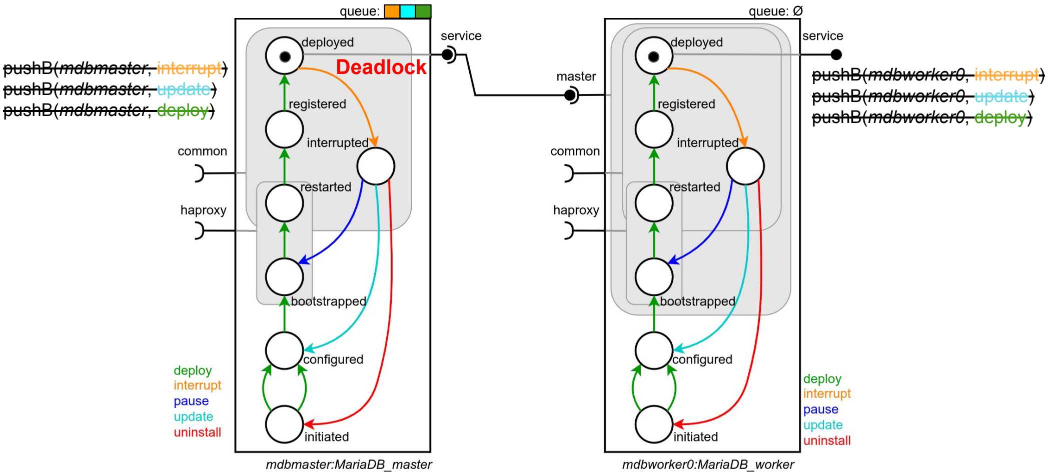
Without the wait ? Failing execution



Without the wait ? Failing execution



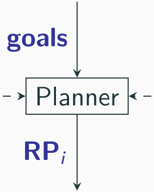
Without the wait ? Failing execution



Planning Concerto-D programs

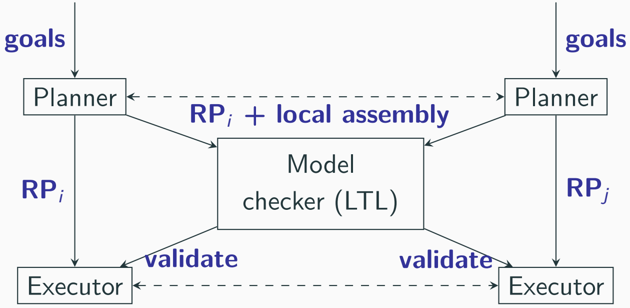
Decentralized planner

- **Input**: goals and lifecycle
- **Output**: a reconfiguration plan
- On each node, iterative resolution :
  - Using SAT solver for intermediate plans
  - Diffusing port constraints, to enrich neighborhood constraint models



- Sat solver ensure validity of the Reconfiguration Plan (RP)
- If the model is unsat, we find the MUS (Minimum Unsat Satisfiability) for explainability, and return error to user

Verify Ballet's execution ?



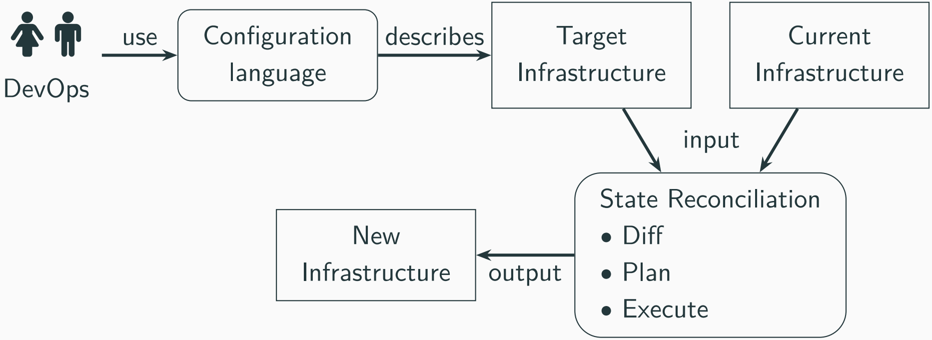
Model checker

- Formalized Ballet's executor within Maude
- Model checking with linear temporal logic (LTL)
- **Pros**: A first step for verifying properties
- **Cons**: Works with all plans and full assembly
- **Cons**: Current formalization does not scale for realistic applications

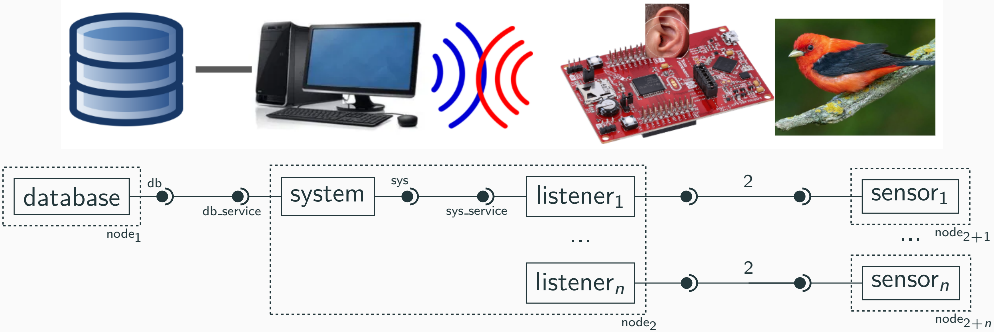
Integrate Ballet - Infrastructure as code

Infrastructure as code (IaC)

Infrastructure as Code (IaC) is the practice of defining and managing infrastructure using code (written in configuration languages). Tools then take this code and automatically deploy the infrastructure as specified.



Integrate Ballet - Control fleet of CPS



Conclusion

SparkTE

- A configurable model transformation engine
- A correct by construction engine on top Spark

Verifying parallel implementation with SyDPaCC

- Extended a Coq library for verifying skeletons
- Formalized a subpart of Spark

Reconfiguration with Ballet

- Declarative tool for decentralized reconfiguration
- Decentralized planning with SAT solver
- Decentralized execution of plans
- Premises of model checking

Perspectives

Verifying Spark

- Formalize distribution process of Spark RDDs calculation
- More support for Spark functions
- Implements additional skeletons in Spark

Model checking for Ballet

- Define additional properties (e.g., safety)
- Use partial order reduction techniques for reducing state-space exploration
- Decentralized checking ? by composition + distribution

Distributed systems

- Energetic optimization (e.g., placement problem)
- Energetic model for configuration space

References

[1] Jolan Philippe. Contribution to the Analysis of the Design-Space of a Distributed Transformation Engine. Ph.D thesis. IMT Atlantique, 2022.

[2] Jolan Philippe, Massimo Tisi, Hélène Coullon, and Gerson Sunyé. Executing Certified Model Transformations on Apache Spark. In 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE), pages 36–48. ACM, 2021.

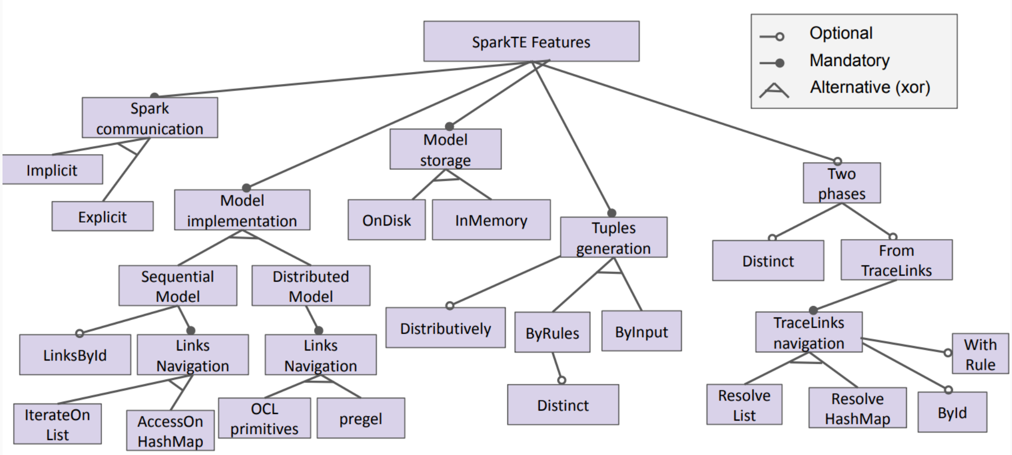
[3] Frédéric Loulergue, and Jolan Philippe. Towards Verified Scalable Parallel Computing with Coq and Spark. In 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP), pages 11–17. ACM, 2023.

[4] Jolan Philippe, Antoine Omond, Hélène Coullon, Charles Prud'Homme, and Issam Raïs. Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study. In IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2024.

[5] Farid Arfi, Hélène Coullon, Frédéric Loulergue, Jolan Philippe, and Simon Robillard. A Maude Formalization of the Distributed Reconfiguration Language Concerto-D. In 17th Interaction and Concurrency Experience (ICE). EPTCS 2024.

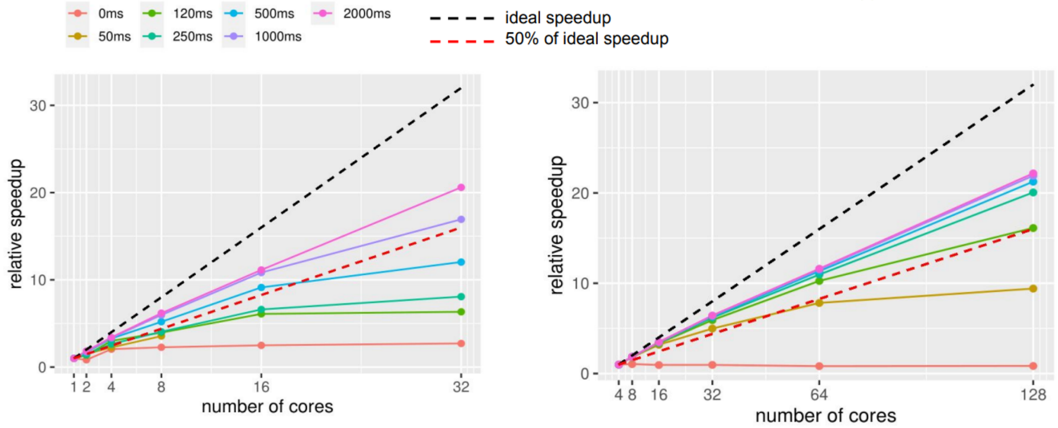
Backup

SparkTE - Configuration space overview



SparkTE Performances

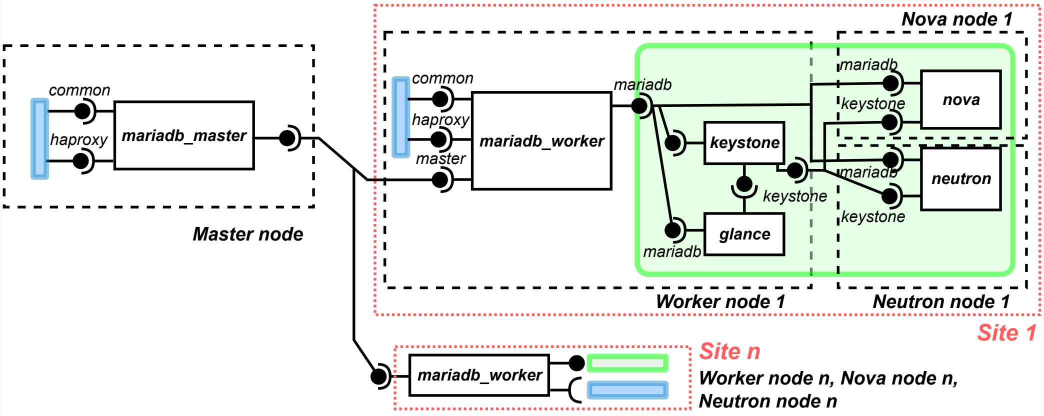
- Simulate a uniform amount of computation on nodes
  - fixed time for each task



Model of 150 elements and 290 links, on 4 machines

Model of 600 elements and 1060 links, 8 machines

Ballet performances on real use-case



Ballet performances on real use-case

Sc.	# Sites	Ballet Planning	Ballet Execution	Ballet Total	Muse	Gain
Deploy	1	1.69s	306.02s	307.71s	536.57s	42.7%
	2	1.78s	306.09s	307.86s	536.69s	42.6%
	5	1.77s	306.19s	307.97s	537.09s	42.7%
	10	2.02s	306.14s	308.19s	538.13s	42.7%
Update	1	3.36s	416.84s	420.20s	555.56s	24.4%
	2	4.39s	416.92s	421.31s	555.70s	24.2%
	5	6.05s	417.17s	423.22s	556.08s	24.0%
	10	5.97s	417.46s	423.43s	556.77s	24.0%

Table 1: Comparison of time for planning and executing a deployment and an update of the `MariaDB_master` instance with Ballet and Muse.

CP Model

- $(B, \Pi, \mathcal{C}, s_{init}, S_{goal})$
- $s_{i+1} = inc_{\Pi}[s_i][b_i], \ \forall i \in 1..m$
- $(b, B, >, 0)$
- $status(p, s_{m+1}) = \Gamma_p$

where

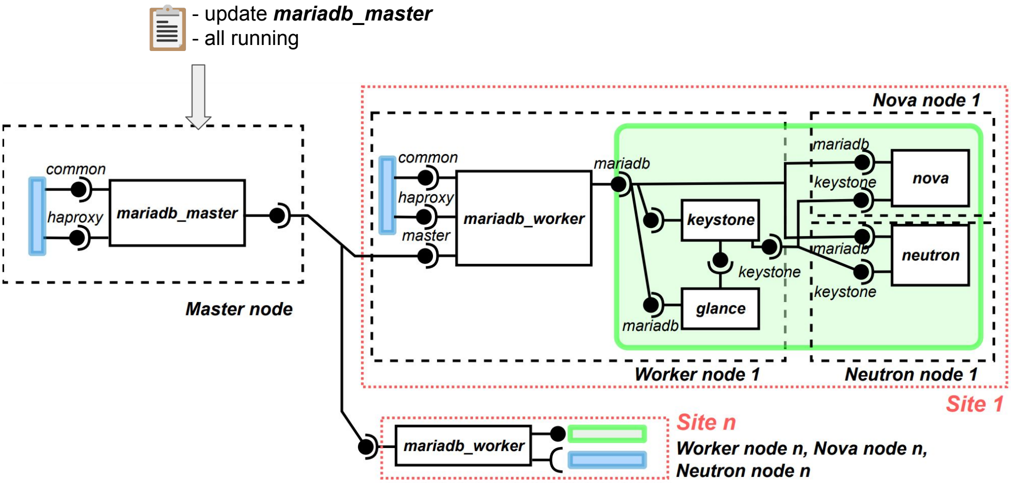
Π an automaton with $\mathcal{C}$ costs

B a sequence of m behaviors

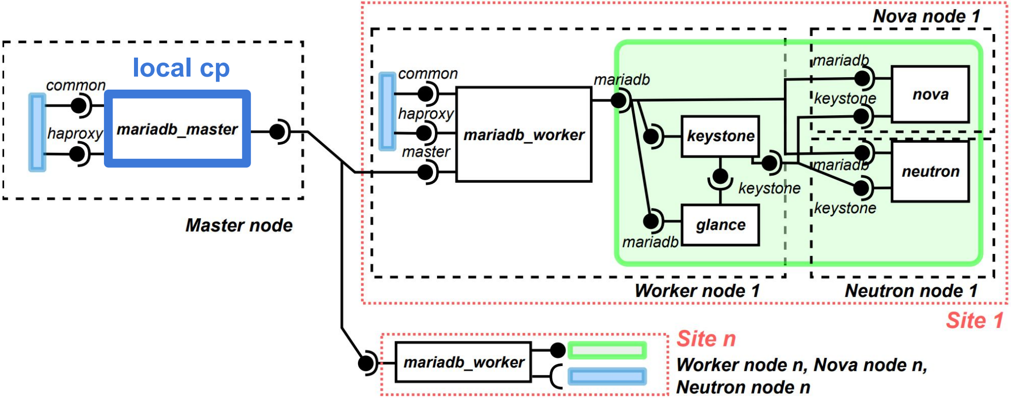
$\Gamma_p \in \{\texttt{active}, \texttt{inactive}\}$ i.e. $\{ \checkmark, \times \}$

$b \in \{$ interrupt, deploy, pause, update, uninstall$\}$

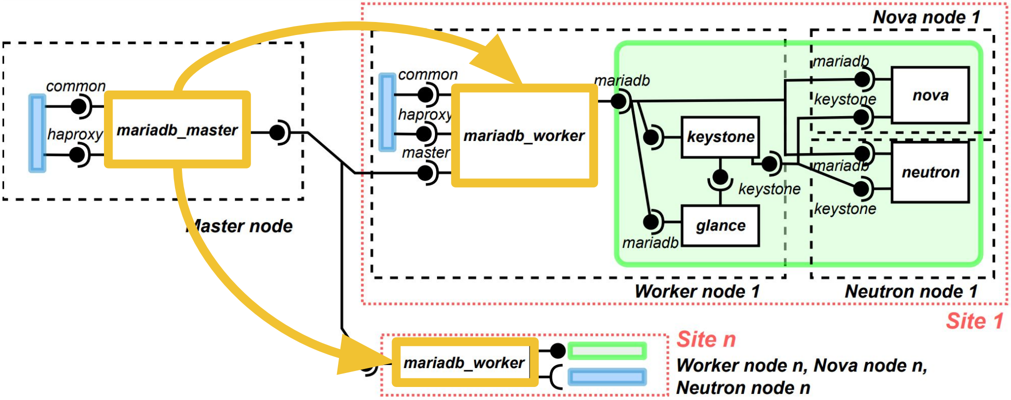
Communication protocol



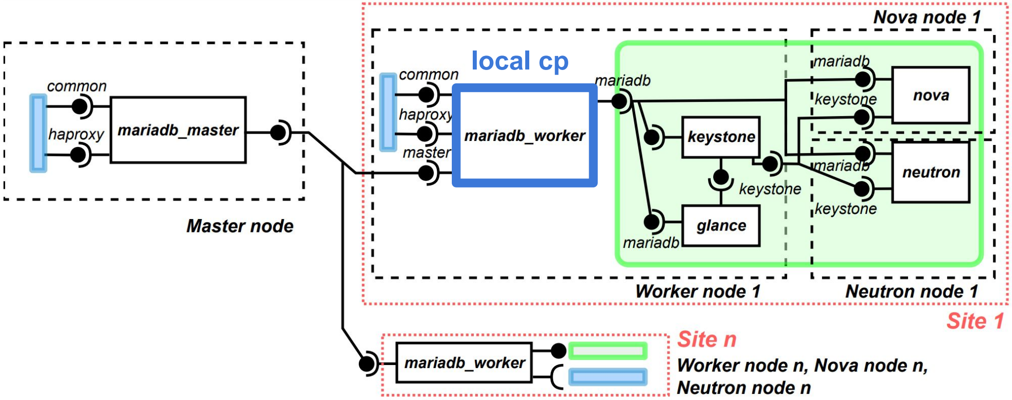
Communication protocol



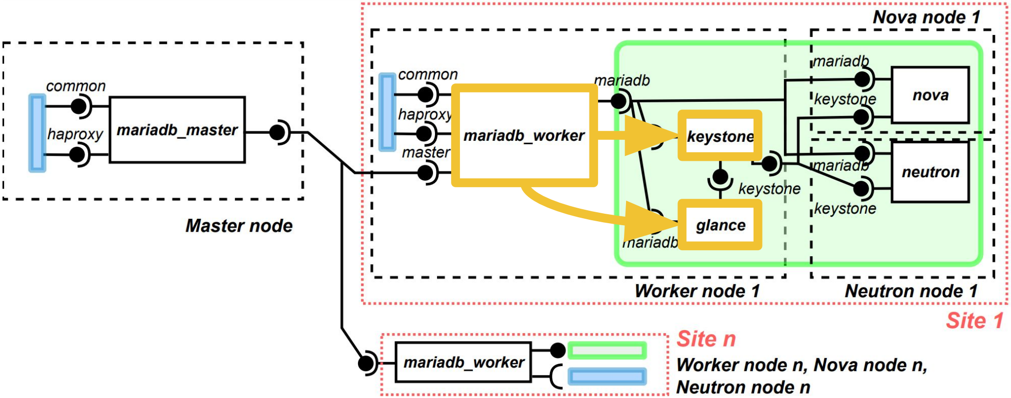
Communication protocol



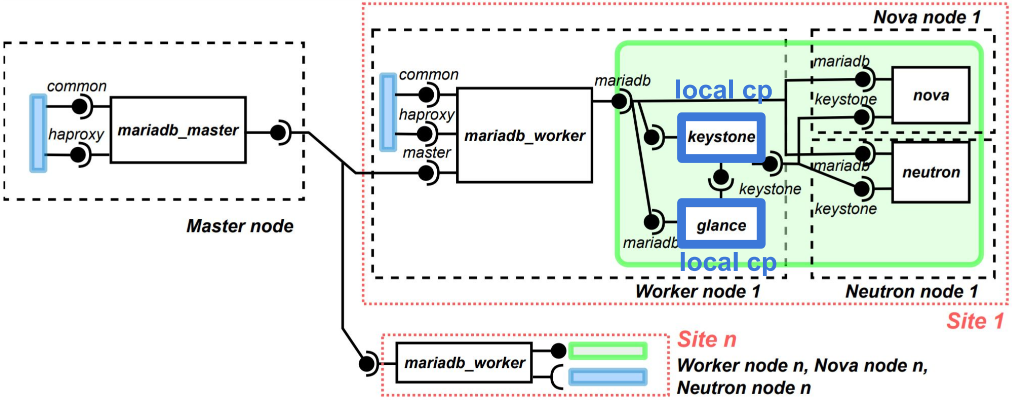
Communication protocol



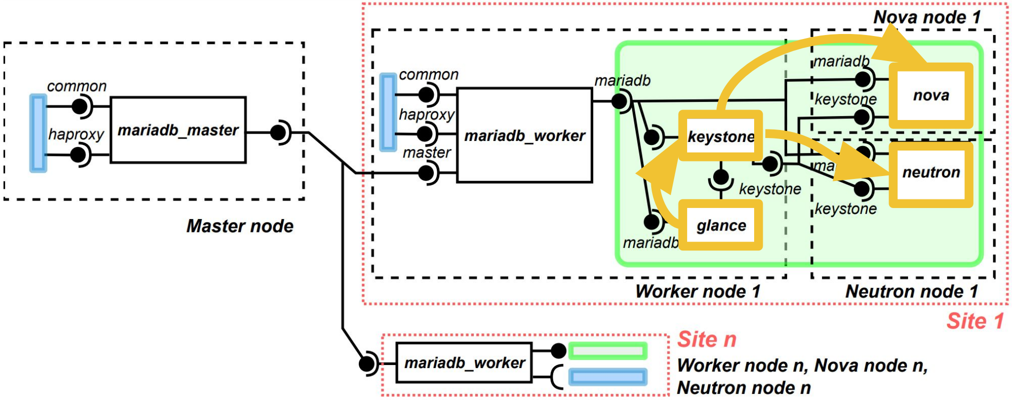
Communication protocol



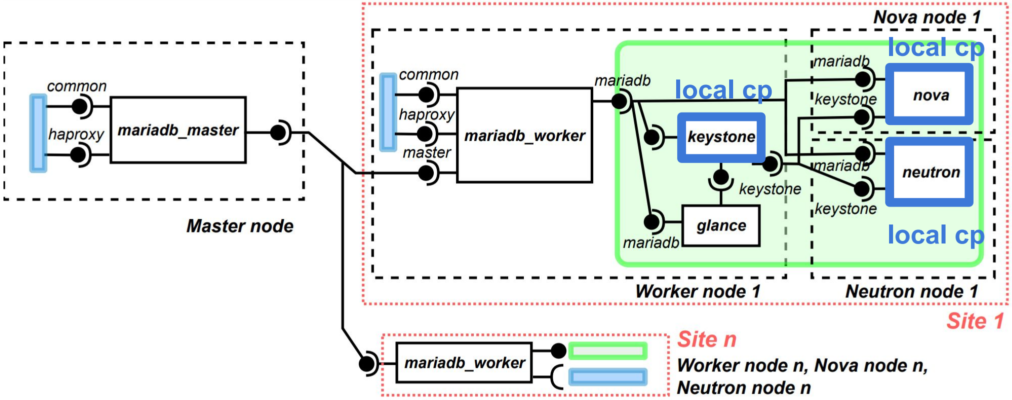
Communication protocol



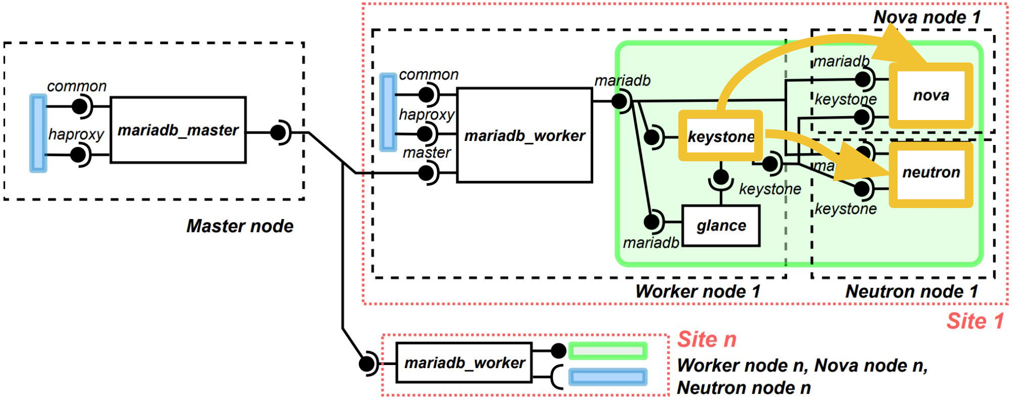
Communication protocol



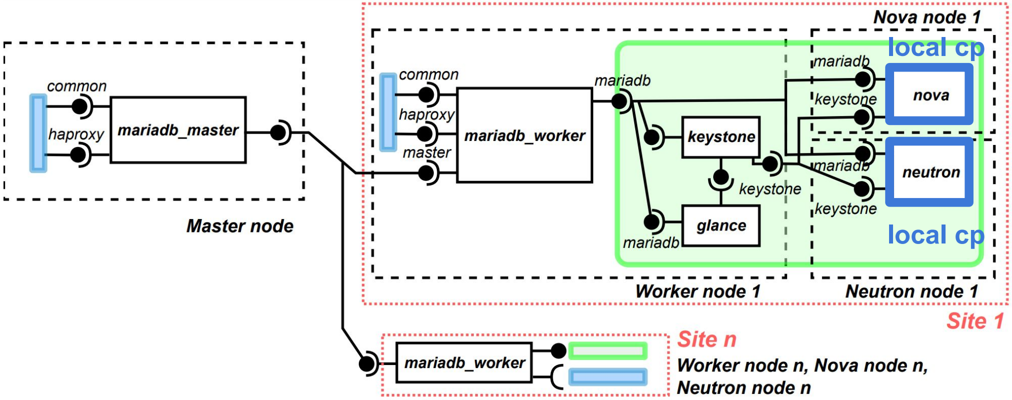
Communication protocol



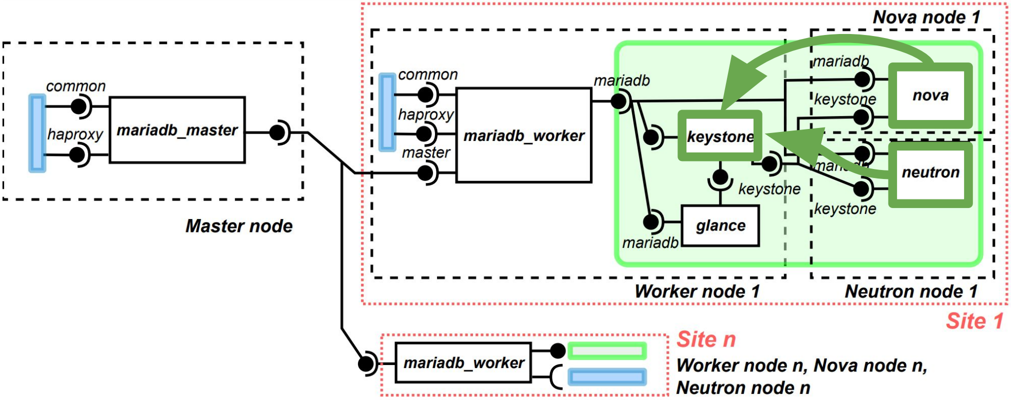
Communication protocol



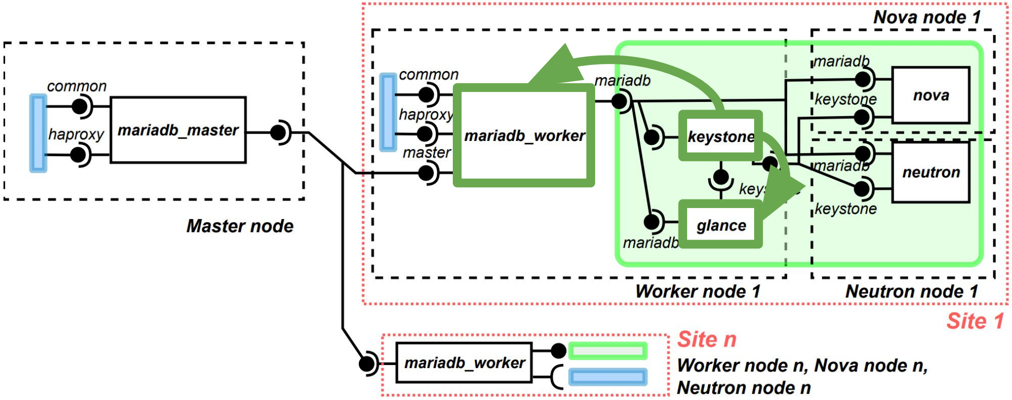
Communication protocol



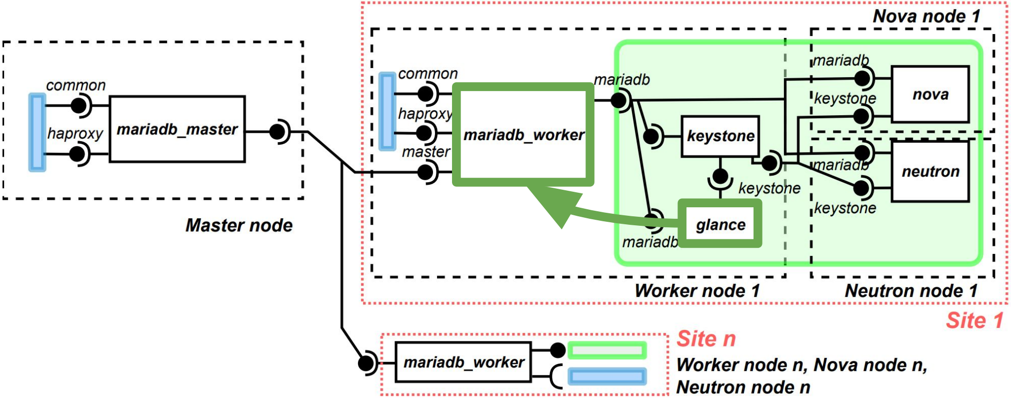
Communication protocol



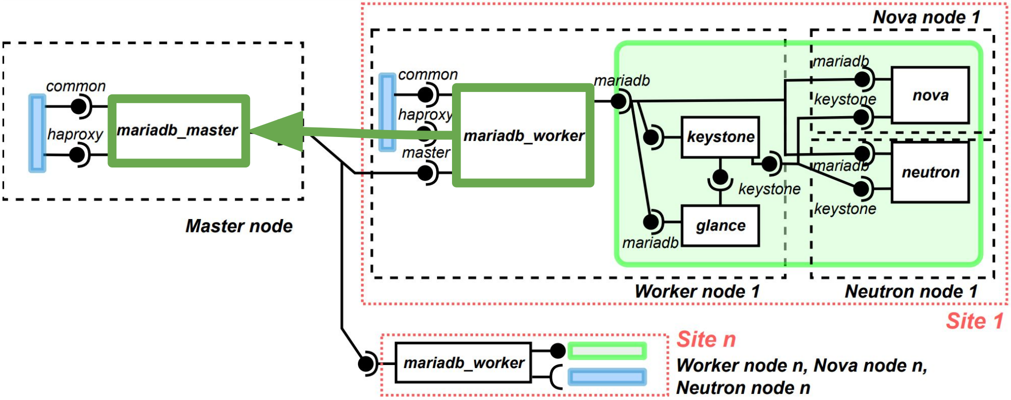
Communication protocol



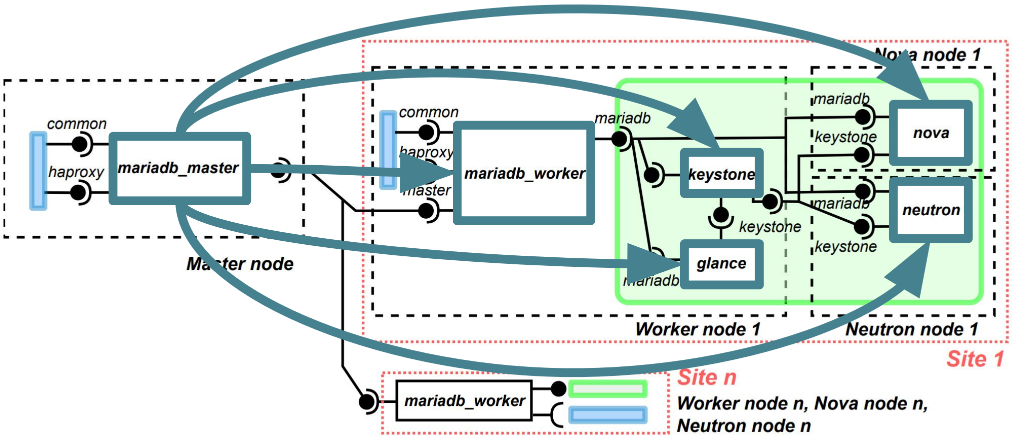
Communication protocol



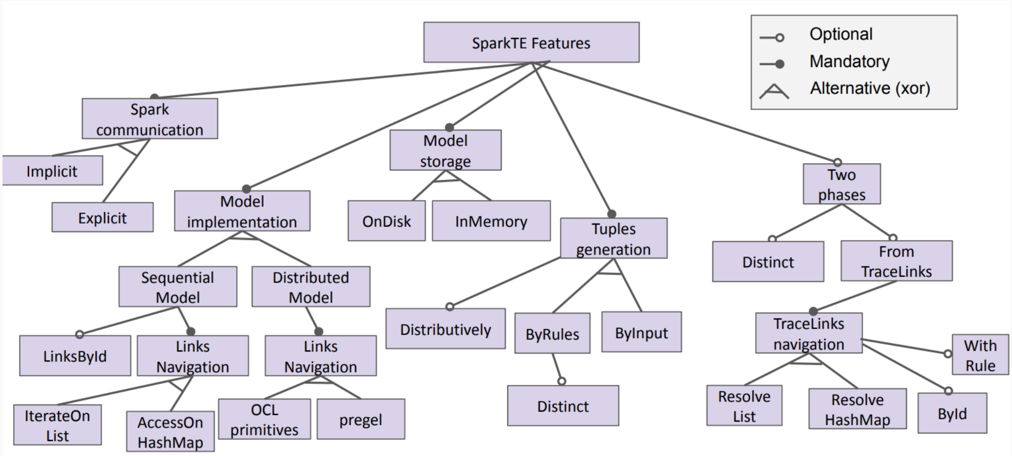
Communication protocol



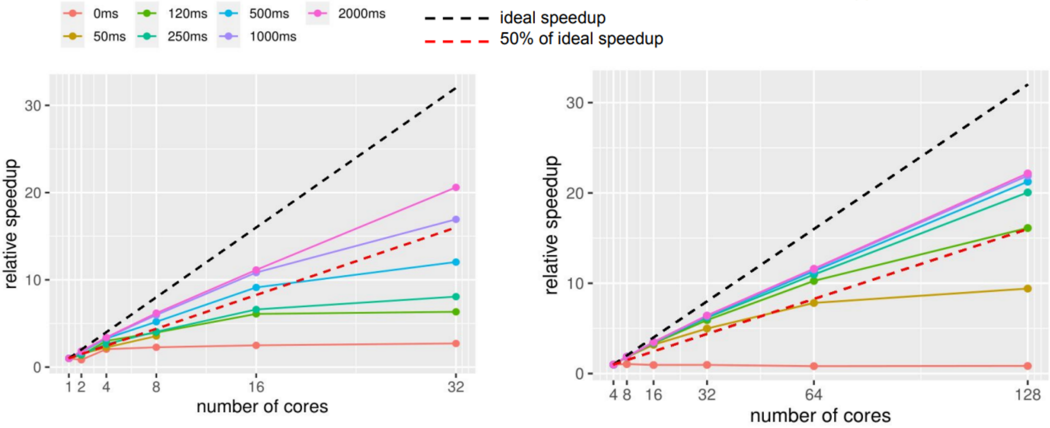
Communication protocol



SparkTE - Configuration space overview



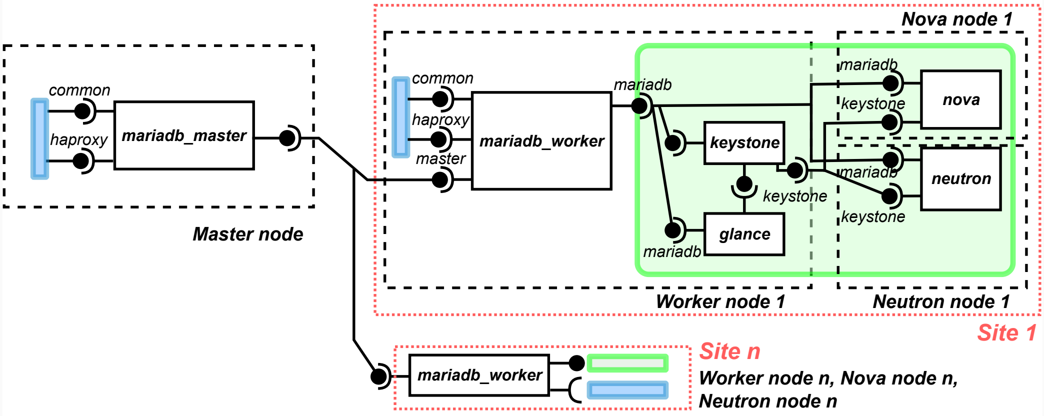
- Simulate a uniform amount of computation on nodes
  - fixed time for each task



Model of 150 elements and 290 links, on 4 machines

Model of 600 elements and 1060 links, 8 machines

Ballet performances on real use-case

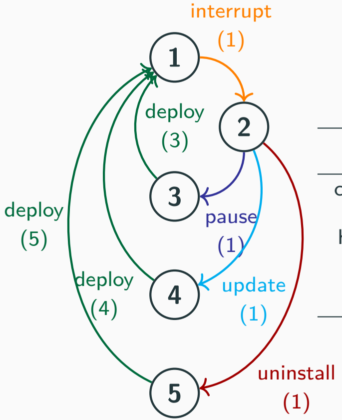


Ballet performances on real use-case

Sc.	# Sites	Ballet			Muse	Gain
		Planning	Execution	Total		
Deploy	1	1.69s	306.02s	307.71s	536.57s	42.7%
	2	1.78s	306.09s	307.86s	536.69s	42.6%
	5	1.77s	306.19s	307.97s	537.09s	42.7%
	10	2.02s	306.14s	308.19s	538.13s	42.7%
Update	1	3.36s	416.84s	420.20s	555.56s	24.4%
	2	4.39s	416.92s	421.31s	555.70s	24.2%
	5	6.05s	417.17s	423.22s	556.08s	24.0%
	10	5.97s	417.46s	423.43s	556.77s	24.0%

Table 2: Comparison of time for planning and executing a deployment and an update of the `MariaDB_master` instance with Ballet and Muse.

CP Model



state	1	2	3	4	5
common	✓	✓	×	×	×
haproxy	×	×	✓	×	×
service	✓	×	×	×	×

Figure 2: Automaton representation of *Mariadb_master* component's life cycle with its matrix for ports statuses.

- $(B, \Pi, \mathcal{C}, s_{init}, S_{goal})$
- $s_{i+1} = inc_{\Pi}[s_i][b_i], \ \forall i \in 1..m$
- $(b, B, >, 0)$
- $status(p, s_{m+1}) = \Gamma_p$

where

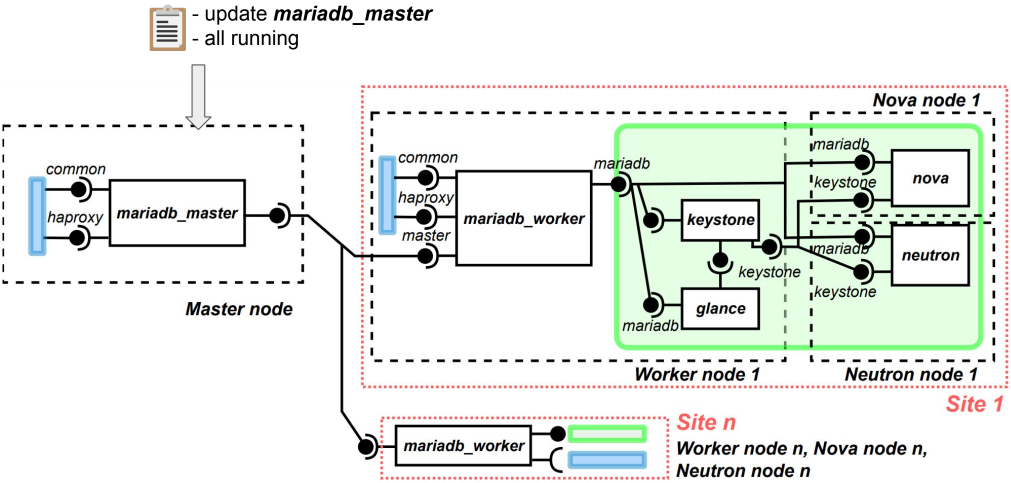
Π an automaton with $\mathcal{C}$ costs

B a sequence of m behaviors

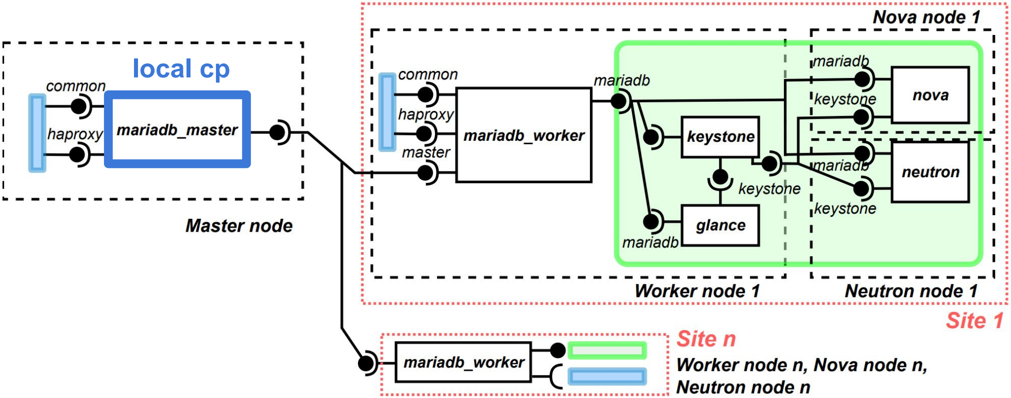
$\Gamma_p \in \{\texttt{active}, \texttt{inactive}\}$ i.e. { ✓, × }

$b \in$ { interrupt, deploy, pause, update, uninstall}

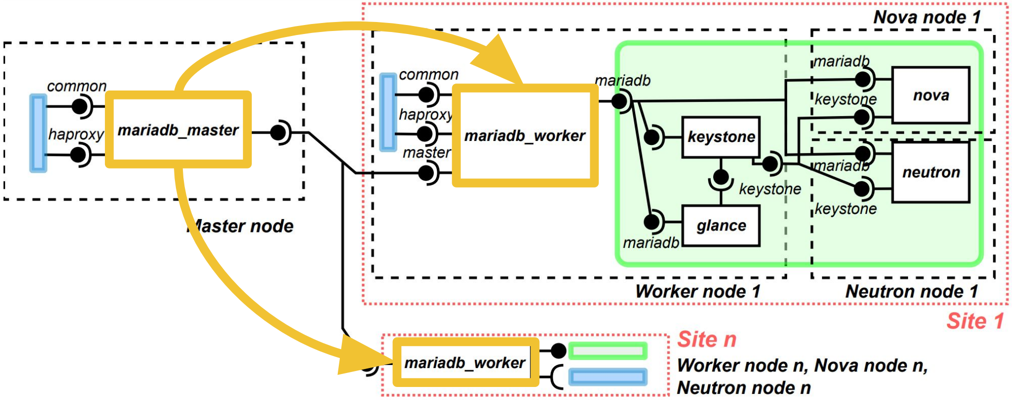
Communication protocol



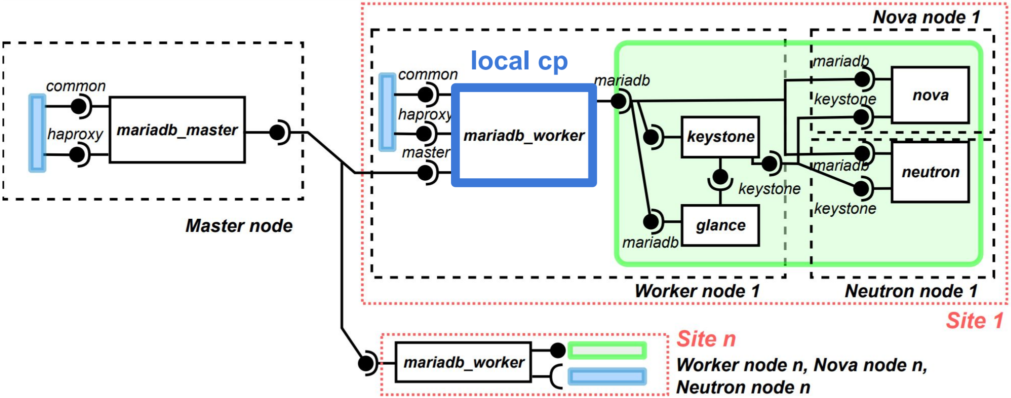
Communication protocol



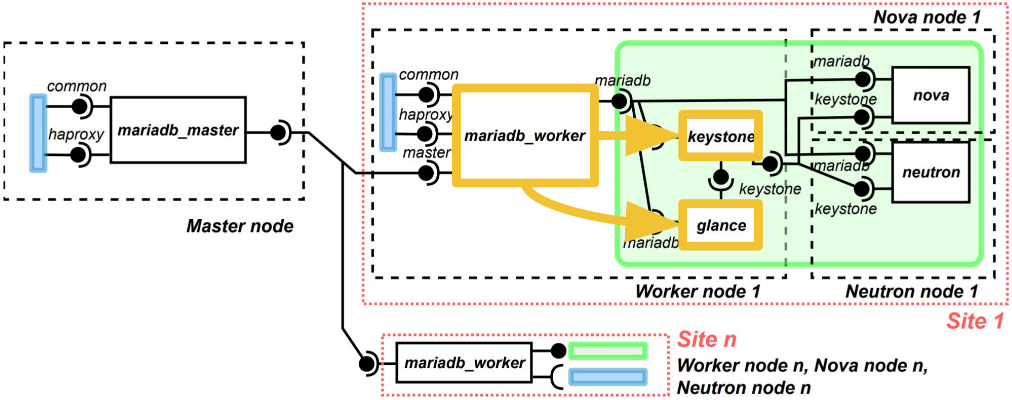
Communication protocol



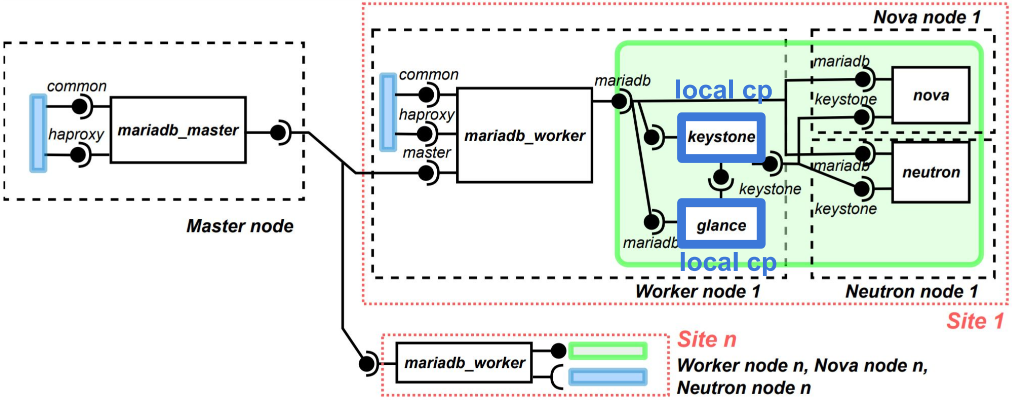
Communication protocol



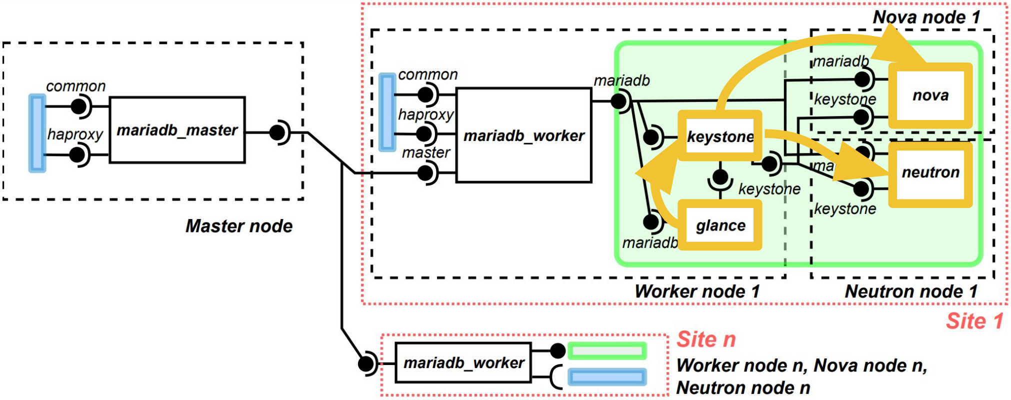
Communication protocol



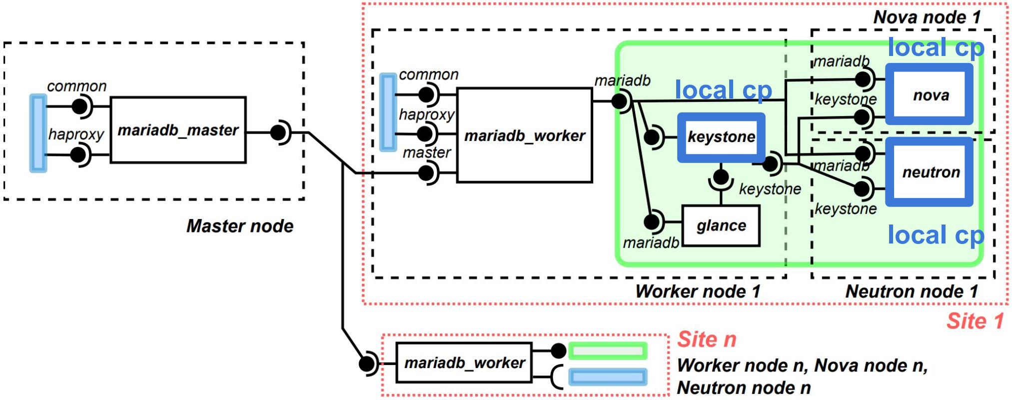
Communication protocol



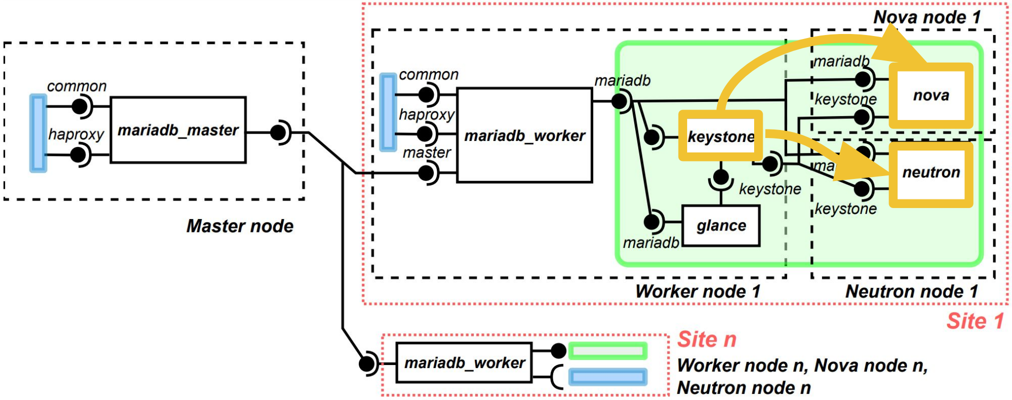
Communication protocol



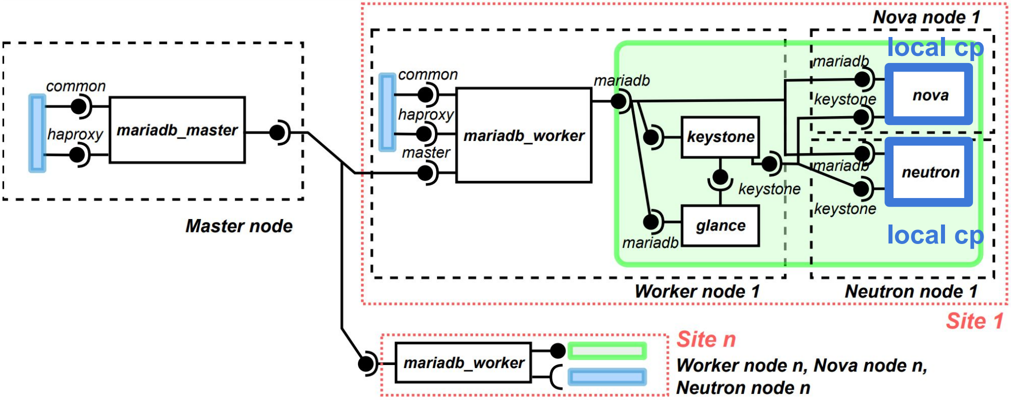
Communication protocol



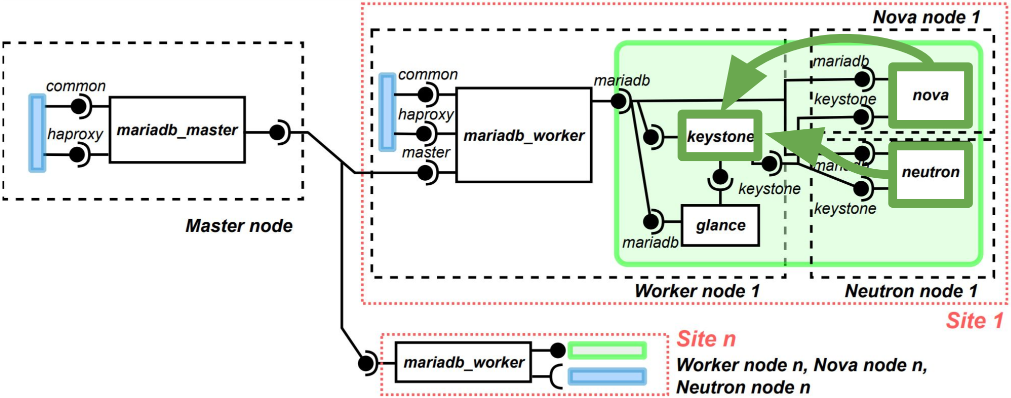
Communication protocol



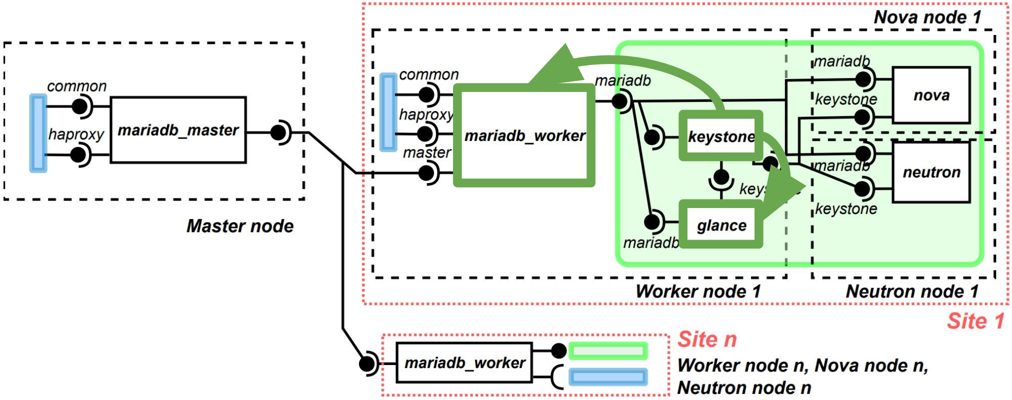
Communication protocol



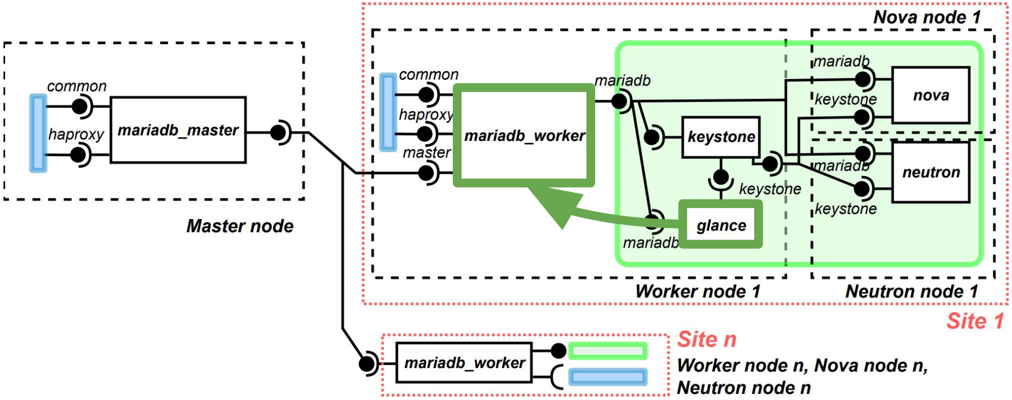
Communication protocol



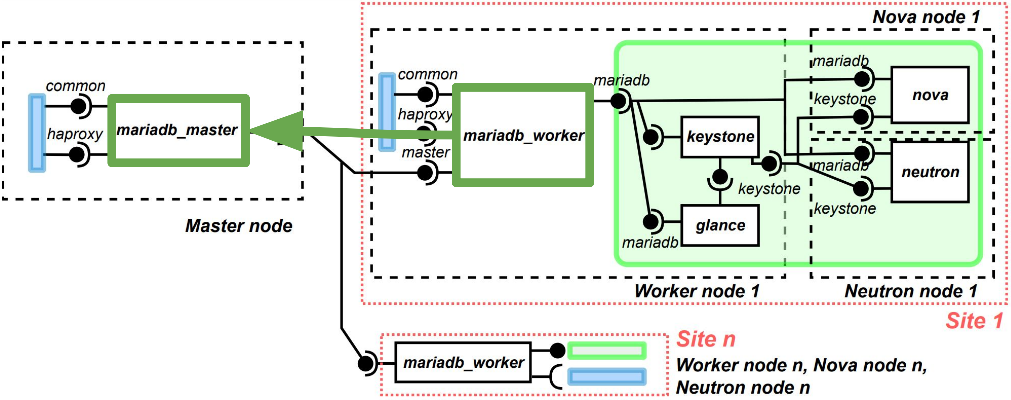
Communication protocol



Communication protocol



Communication protocol



Communication protocol