Fast Choreography of Cross-DevOps Reconfiguration with Ballet

Multi-Site OpenStack Case Study

Jolan Philippe, Hélène Coullon, Antoine Omond, Charles Prud'Homme, Issam Raïs

March 13th, 2024

STACK, IMT Atlantique
SeMaFoR project

DevOps deployment and reconfiguration

Responsible for **development**

DevOps team

Run tests | Package app | Build image Docker | Push artifact to repo | Deploy to server

Continuous integration + Continuous deployment

Responsible for **operations**

⇒ Continuous deployment then **reconfiguration**

Cross-DevOps reconfiguration

Responsible for **development** of A

DevOps team

Run tests on A → Package A → Build image for A → Push A to repo → Deploy A

Responsible for **operations** on A

Responsible for **development** of B which uses A

DevOps team

Run tests on B → Package B → Build image for B → Push B to repo → Deploy B (after A?)

Responsible for **operations** on B (using A)

Approach

Naive solution

Using a centralized tool on top of all DevOps teams is not suitable for scale and fault tolerance reasons.

Decentralized solution

Make a plan for each DevOps team, and execute them conccurently.

Muse (Sokolowski et. al.) covers cross-DevOps decentralized reconfiguration with planning, but inefficient because of the fixed life cycles (i.e., on-off mode for resources).

Ballet overview

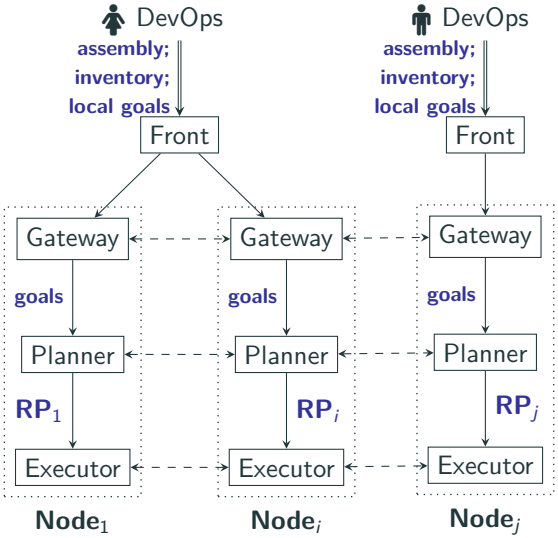


Figure 1: Ballet overview

- Decentralized tool (one instance of Ballet on each node)
- Declarative input
- Automatic planning
- Efficient reconfiguration

Gateway

Global knowledge building of reconfiguration goals

Planner

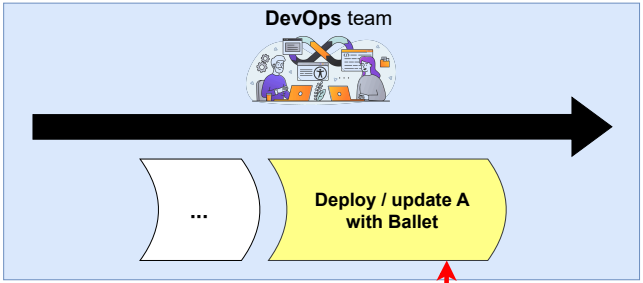
Decentralized inference of reconfiguration plans (RPs)

Executor

Coordinated execution of RP

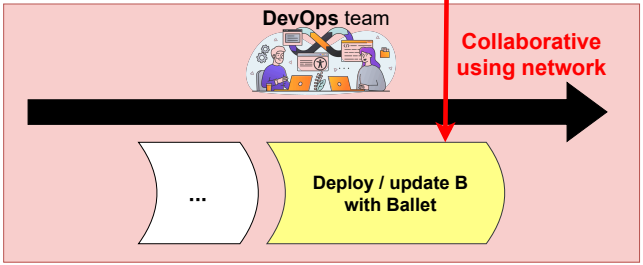
Ballet's usage

Responsible for **development** of A and A's life cycle



Responsible for **operations** on A

Responsible for **development** of B and B's life cycle



Responsible for **operations** on B (using A)

Developers' concern

Life-cycle and ports

Simple language to define component

- **Places**: milestones in reconfiguration
- **Transitions**: reconfiguration actions (can be concurrent) associated to a general behavior
- **Ports**: dependencies in the reconfiguration process between components

⇒ Python definition

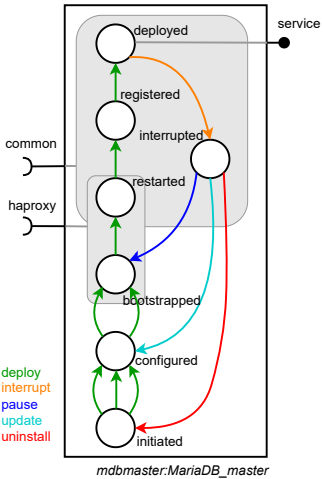


Figure 2: Visual representation of a component for MariaDB

DevOps' concern

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>
```

Ballet execution

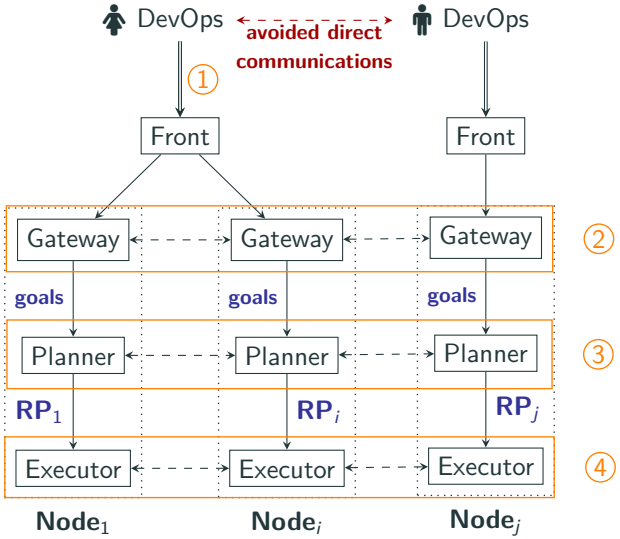


Figure 3: Ballet execution

Ballet execution: Planner

Local resolution

- **Purpose**: Find a sequence of actions to execute

⇒ Constraint programming approach:

1. Component's life-cycle as an automaton where transitions are actions of reconfiguration
2. Find a word in this automaton
3. Constraint the word with reconfiguration goals

Constraint propagation

- **Purpose**: Inferring additional actions and synchro. bareers

⇒ Propagation of constraint based on Gossip algorithm

1. Send message about what would be port statuses with found sequence
2. Enrich local constraint model with received message
3. Consensus using Paxos-like approach to end the propagation

Ballet execution: Execution

Concurrency thanks to fine-grained life-cycles

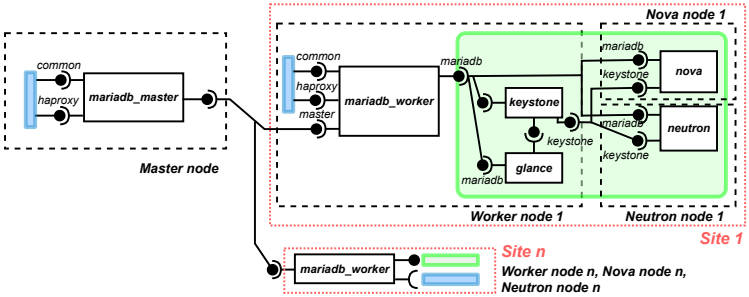
- Execute actions on components concurrently (inter-component parallelism)
- Execute component's actions concurrently when allowed (intra-component parallelism)
- Synchronize with external components with barrier inferred by the planner

Communication between components

- Message when a port is turned *active*
- Message when a port is turned *inactive*
- Messages exchanged for synchronization barriers

Evaluation with multi-site OpenStack deployment and update

Deployment and update of OpenStack with Galera cluster of MariaDB with $n \in [1, 2, 5, 10]$ sites, that is a total of $7 + 11 * n$ components.



Metric of interest

- Execution time
  - Planner
  - Executor
- Inference (see paper)
  - Num. constraints
  - Num. actions

Experimental results

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

Conclusion

Contributions

- Ballet as a DevOps reconfiguration tool
- Infer reconfiguration actions
- Efficient execution of actions

Target applications

- OpenStack, and CPS
- Fog areas, smart cities, IoT devices, etc.

Perspectives

- Extend our constraint propagation to other problems (e.g., placement or reconfiguration)
- Formalization and reasoning for correctness
- Manage asynchronous communications for intermittent systems