Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto

Maverick Chardet (IMT Atlantique), Christian Pérez (Inria)
Hélène Coullon

Associate professor at IMT Atlantique, France

Inria researcher, France

Adjunct professor at UiT, Tromsø, Norway

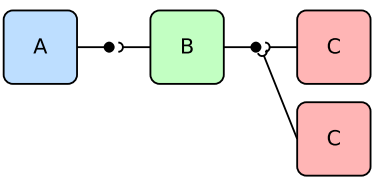
Outline

Table of Contents

Distributed software systems

General definition

- Non monolithic code,
- modular units of code - **components**,
- software system = **architectural assembly** of component instances,
- interactions between components through **communications**.



- Master/workers,
- microservices,
- service-oriented,
- layered,
- etc.

Deployment, management, reconfiguration

Ever-running and long-running distributed systems

What is a reconfiguration?

- Reconfiguration through time
  - need to add/remove components and/or connections
  - need to change internal configurations
- A set of instructions to move from one state of the system to another.

Examples of reconfiguration reasons

- Faults or errors on services or hardware (e.g., re-deploy),
- dynamic energy or security constraints (e.g., change the set of components),
- dynamic improvement of performance (e.g., scaling),
- dynamic upgrade of some modules.

Our goals

1. Efficiency of the reconfiguration

- reach quickly a targeted configuration,
  - dynamic security requirements, more frequent reconfigurations (e.g., Fog Edge) etc.
- reduce disruption time
  - frequent faults or disconnections etc.

2. Execution time prediction

- better decisions on "when to perform the reconfiguration?"
- better decisions on "how to schedule concurrent reconfigurations?"

Table of Contents

Lifecycle

Contributions with a lifecycle abstraction

- fixed lifecycle: TOSCA, DEPLOYWARE, SMARTFROG, ENGAGE
  - easier to use, less flexible
- programmable lifecycle: AEOLUS, ANSIBLE (DevOps configuration tool)
  - more difficult to use, more flexible

Machine 1 **[WHERE]**

Database (DB) [WHAT]

[HOW] [LIFECYCLE]

1. Install
2. Configure
3. Start the service
4. Prepare the service

Machine 2 **[WHERE]**

Web-server (WS) [WHAT]

[HOW] [LIFECYCLE]

1. Install
2. Configure firewall
3. Download
4. Configure parameters
5. Start the service

Dependencies

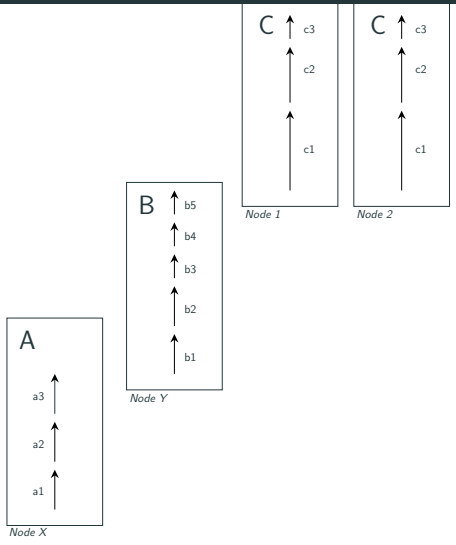
4 levels of dependencies

1. same component level: Ansible
2. component level: Tosca, Deployware
3. lifecycle level: Tosca, Engage, Aeolus
4. intra-lifecycle level: Concerto

Performance through parallelism and dependencies

level1: multiple nodes, same action

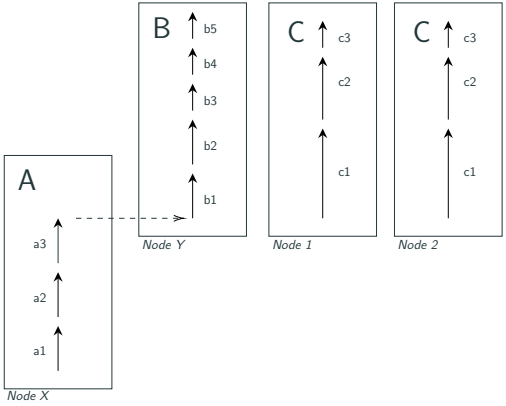
- no dependencies declared
- procedural execution order
- parallelism for the same component
- Ansible



Performance through parallelism and dependencies

level2: level1+non-dependent components

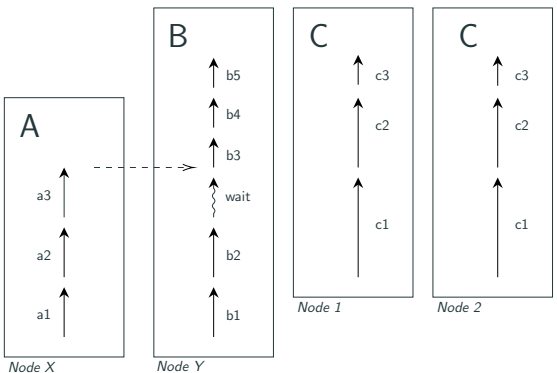
- dependencies at the component level
- DEPLOYWARE, (basic) TOSCA, ENGAGE



Performance through parallelism and dependencies

level3: level1 + level2 + inter-component

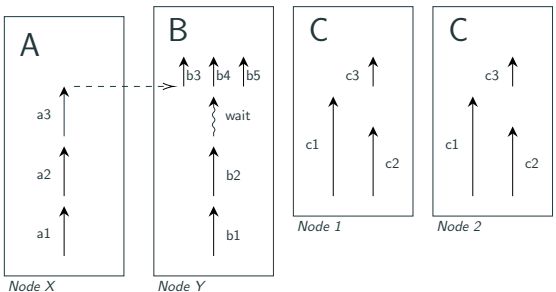
- dependencies at the lifecycle level between components
- (advanced) TOSCA, AEOLUS



Performance through parallelism and dependencies

level 4: level1 + level2 + level3 + intra-component

- parallelism within the lifecycle of one component
- CONCERTO

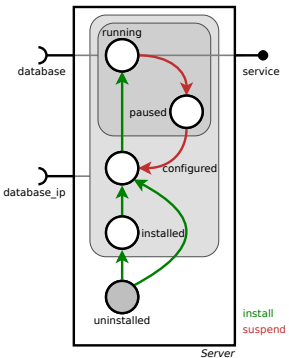


The finer the dependencies granularity is, the better is the efficiency

Table of Contents

Control components

Written by the component developers

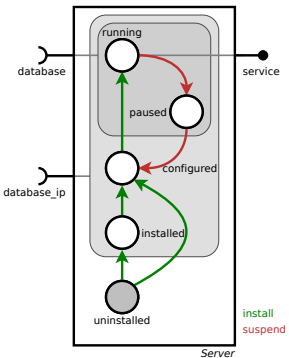


Internal net [LIFECYCLE]

- places = milestones
- transitions = actions to perform
  - concretely: scripts are attached to transitions
  - in the model: exact nature/effects of actions not represented, only coordination

Control components

Written by the component developers



Interfaces [DEPENDENCIES]

- data or service ports
  - use ports = requirements
  - provide ports = provisions
  - during execution: active/inactive
- behaviors
  - subset of transitions
  - during execution: active/inactive

Control components in practice

Written by the component developers

```
class Server(Component):
    def create(self):
        self.places = ['uninstalled','installed','configured','running','paused']

        self.initial_place = 'uninstalled'

        self.behaviors = ['b_install','b_suspend']

        self.transitions = {
            'install1': ('uninstalled','installed','b_install',self.install1),
            'install2': ('uninstalled','configured','b_install',self.install2),
            'configure': ('installed','configured','b_install',self.configure),
            'start': ('configured','running','b_install',self.start),
            'suspend1': ('running','paused','b_suspend',self.suspend1),
            'suspend2': ('paused','configured','b_suspend', self.suspend2)
        }
```

Control components in practice

Written by the component developers

```python
class Server(Component):
    def create(self):
        ...

        self.dependencies = {
            'database_ip': (DepType.USE, ['installed','configured','running','paused']),
            'database': (DepType.USE, ['running','paused']),
            'service': (DepType.PROVIDE, ['running'])
        }

    # Definition of the actions
    def install1(self):
        remote = SSHClient()
        remote.connect(host, user, pwd)
        remote.exec_command(cmd)
        ...
```

Reconfiguration language

Add/remove
Add/remove a component instance to the current assembly

Connect/disconnect
Connect/disconnect two component instances with compatible ports

Push behavior
Push a behavior to the behavior queue on a component instance

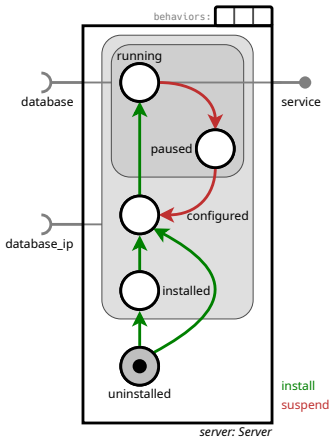
Wait
Wait for a given behavior of a component instance

Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
1 add(server: Server)
2 add(db: Database)
3 con(server.database_ip,db.ip)
4 con(server.database,db.service)
5 pushB(server, install)
6 pushB(db, deploy)
7 wait(server)
```

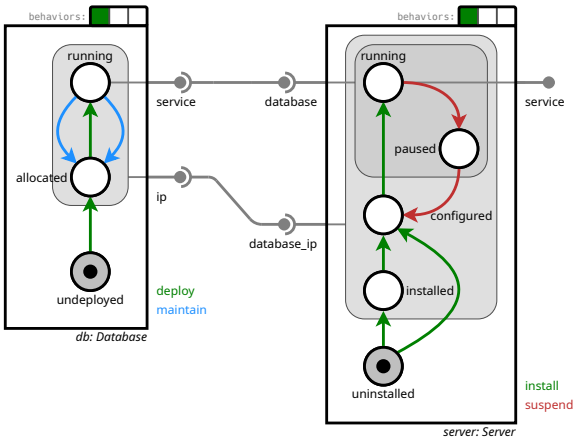


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
1 add(server: Server)
2 add(db: Database)
3 con(server.database_ip,db.ip)
4 con(server.database,db.service)
5 pushB(server, install)
6 pushB(db, deploy)
7 wait(server)
```

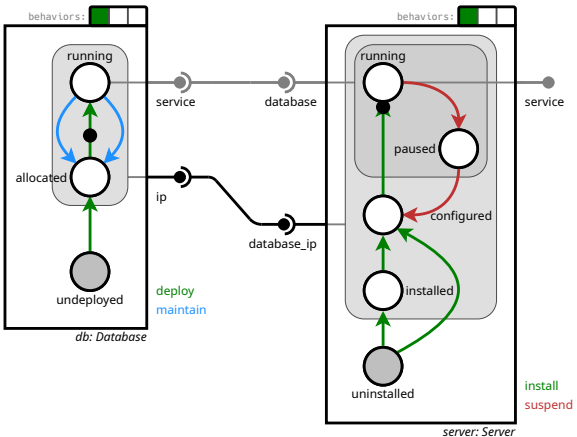


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
1 add(server: Server)
2 add(db: Database)
3 con(server.database_ip,db.ip)
4 con(server.database,db.service)
5 pushB(server, install)
6 pushB(db, deploy)
7 wait(server)
```

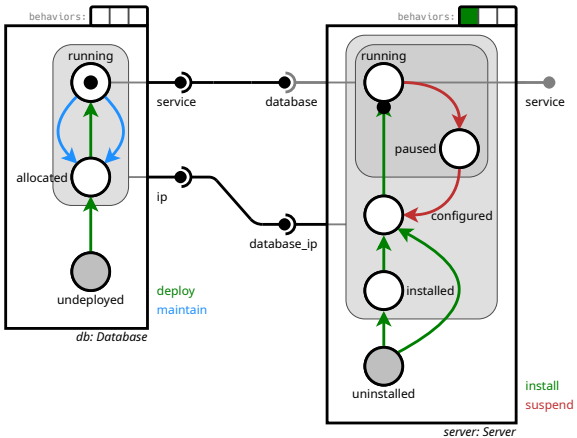


Reconfiguration example - deployment

Written by the reconfiguration developer

Deployment program:

```
1 add(server: Server)
2 add(db: Database)
3 con(server.database_ip,db.ip)
4 con(server.database,db.service)
5 pushB(server, install)
6 pushB(db, deploy)
7 wait(server)
```

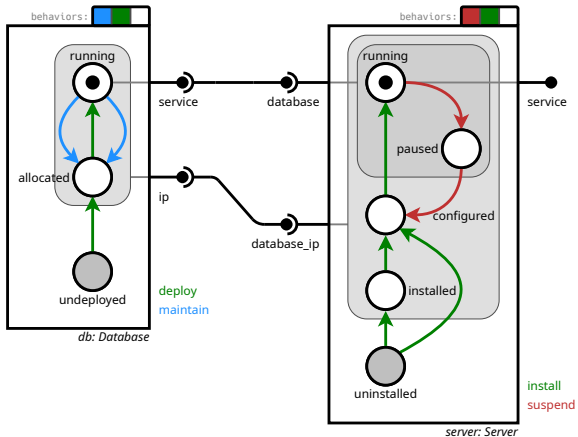


Reconfiguration example - maintenance

Written by the reconfiguration developer

Maintenance program:

```
1 pushB(db, maintain)
2 pushB(db, deploy)
3 pushB(server, suspend)
4 pushB(server, install)
5 wait(server)
```

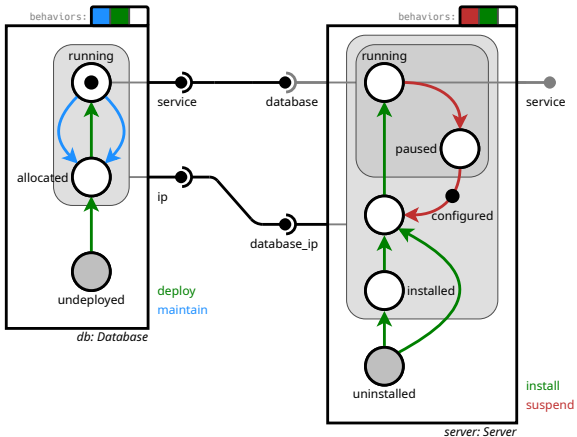


Reconfiguration example - maintenance

Written by the reconfiguration developer

Maintenance program:

```
1 pushB(db, maintain)
2 pushB(db, deploy)
3 pushB(server, suspend)
4 pushB(server, install)
5 wait(server)
```

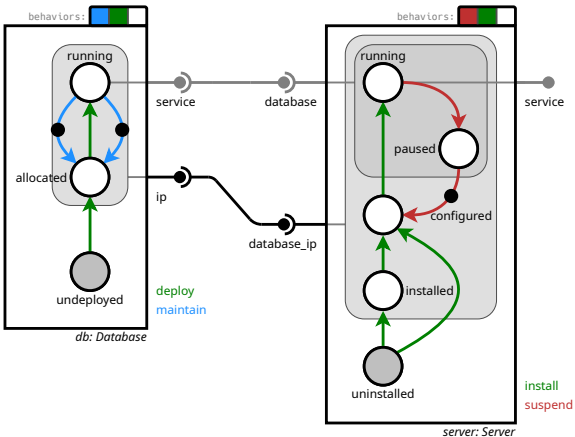


Reconfiguration example - maintenance

Written by the reconfiguration developer

Maintenance program:

```
1 pushB(db, maintain)
2 pushB(db, deploy)
3 pushB(server, suspend)
4 pushB(server, install)
5 wait(server)
```



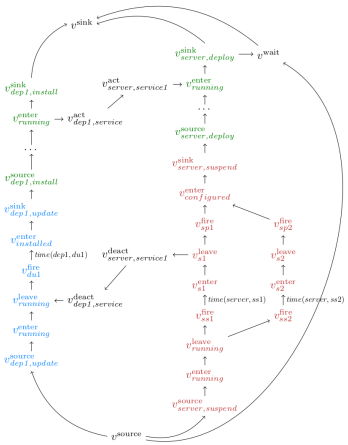
Performance prediction

Inputs:

- reconfiguration program
- time estimations for transitions

Dependency graph generation

- nodes for events such as reaching/leaving place, firing transition
- transition arcs are weighted to reflect execution time
- other arcs are 0-weighted



Performance prediction

Critical path

- length = reconfiguration time (assuming hardware can execute as many concurrent threads as needed)
- highlights the transitions that should be optimized

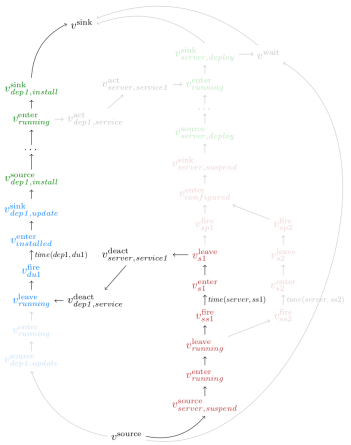


Table of Contents

Evaluation on the reconfiguration of MariaDB

Real use-case extracted from the OpenStack Summit 2018 on a multi-region deployment of OpenSatck

Initial state

- centralized MariaDB running
- additional nodes running some generic components (docker, pip...)

decentralization

- replaces centralized DB with a distributed version with instances on n nodes
- requires a backup of the data, and a restart of the master node

scaling

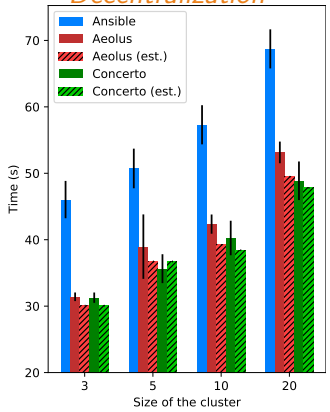
- deploys m new nodes with an instance of the distributed DB

Reproducible experiments

Evaluation on the reconfiguration of MariaDB

Results on nodes of UvB (Sophia) of Grid'5000 (2×6-core Intel Xeon X5670 CPUs, 96 GB RAM, 250 GB HDD, internal 40 Gbps InfiniBand, external 1 Gbps).

Decentralization



Scaling

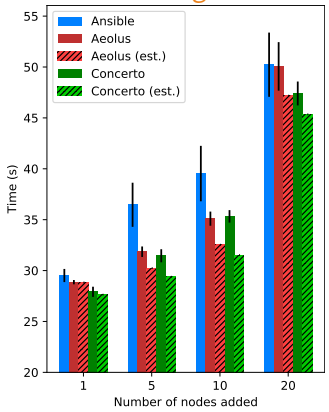


Table of Contents

Conclusion

- Need for software engineering practices in reconfiguration (DevOps community)
- Need for efficiency and execution time prediction
  - reach quickly the new state
  - reduce disruption time
- Presentation of CONCERTO and its performance prediction model
- Evaluation on synthetic use-cases (see the paper) and a real use-case

Perspectives

- Automatic generation of a CONCERTO program from a goal
- Using CONCERTO to reconfigure cyber-physical systems and edge devices
  - PhD oppotunity! Contact me helene.coullon[at]imt-atlantique.fr