Reliable Self-Deployment of Cloud Applications

Xavier Etchevers[1], Gwen Salaün[2], Fabienne Boyer[3],
Thierry Coupaye[1], Noel De Palma[3]

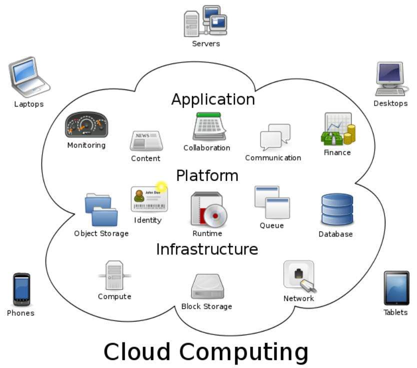
[1]Orange Labs, France

[2]Grenoble INP, Inria, France

[3]UJF-Grenoble 1, INRIA, France

Introduction

- Cloud computing aims at delivering resources and applications as a service over a network (*e.g.*, the Internet)



- Cloud applications are complex distributed applications composed of interconnected software components running on separate virtual machines

- Setting up, (re)configuring, and monitoring these applications are complicated tasks, and involve complex management protocols

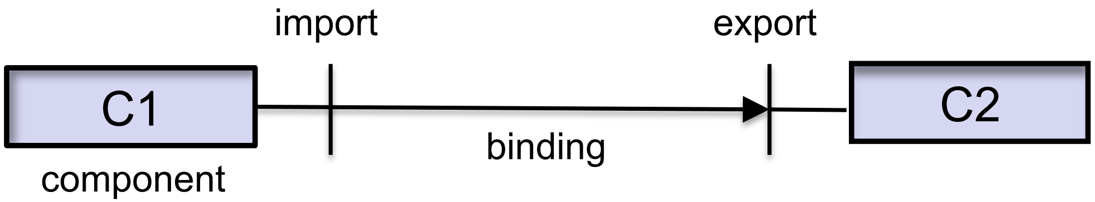
- In this talk, we present a reliable self-deployment protocol automating the configuration and start-up of distributed applications in the cloud

Outline

1. Self-Deployment Protocol
2. Verification
3. Implementation
4. Concluding Remarks

Application Model

- An application model consists of a set of components and a set of bindings connecting these components together
- A component is composed of input and output ports, namely imports and exports
- An import can be either optional or mandatory
- A binding connects an import of one component to an export of another component

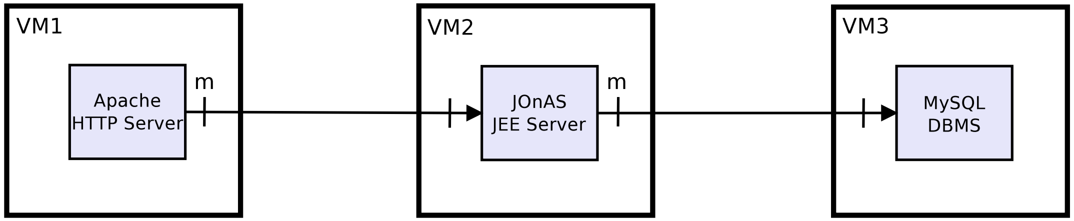


- Components are distributed over virtual machines, which are in charge of their administration (local and remote bindings + start-up process)

Self-Deployment Protocol

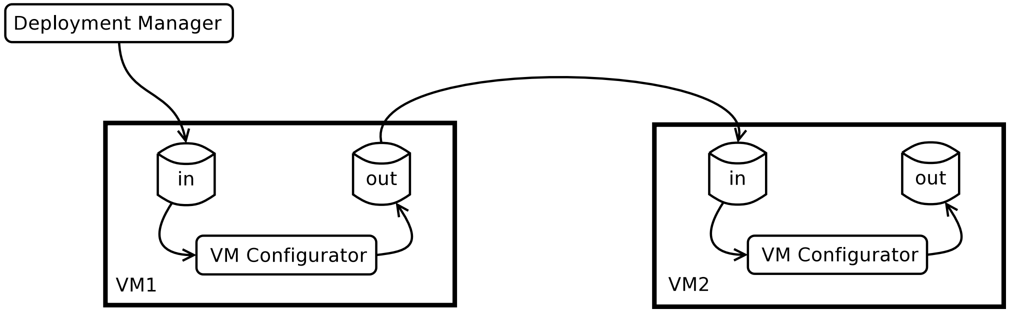
- This protocol (developed at Orange labs) for configuring distributed applications is decentralized and loosely-coupled

- Each virtual machine (VM) embeds the application model and a configurator in charge of the component binding and application start-up

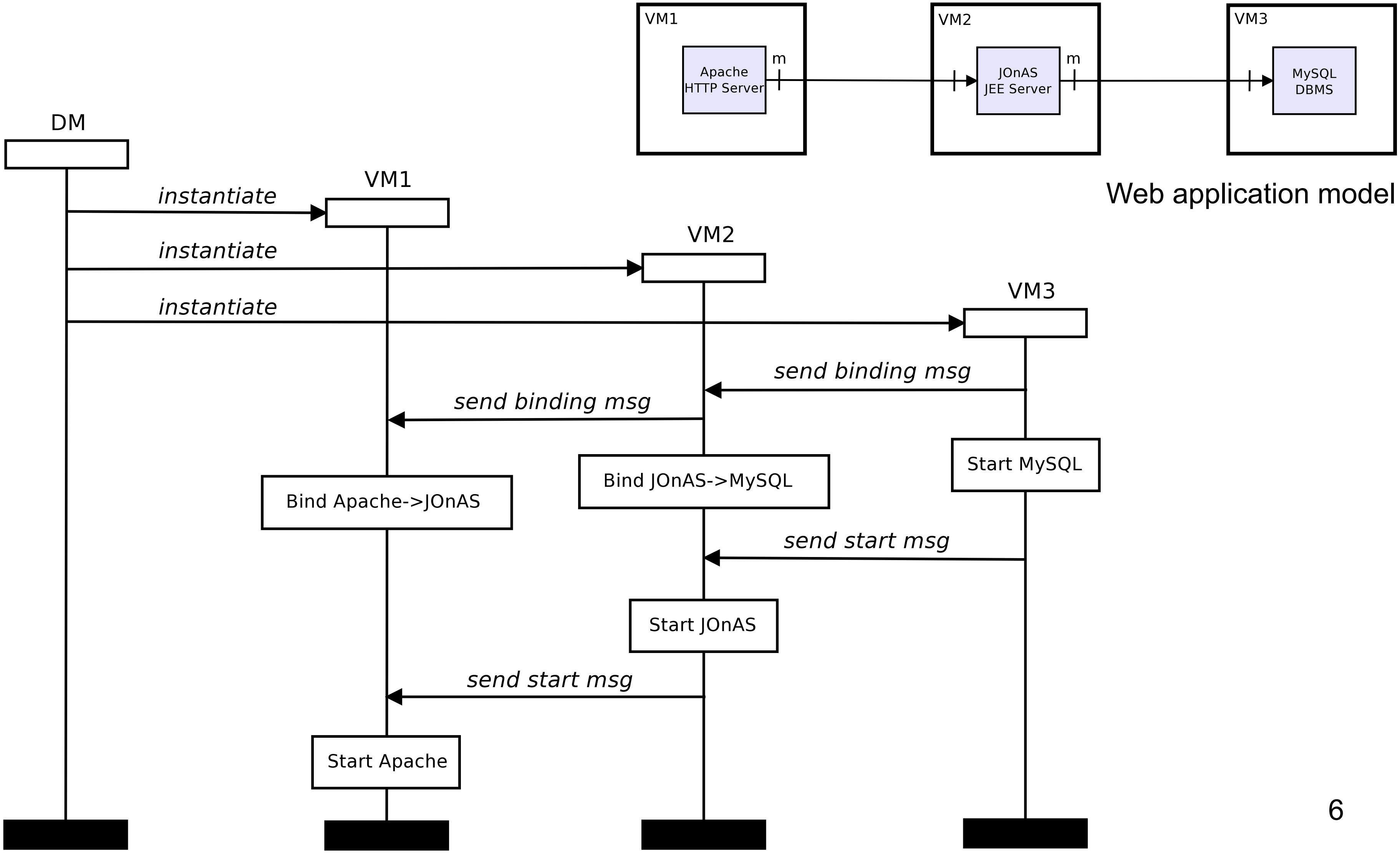


Web application model

- Configurators interact together through a Message Oriented Middleware (MOM), which relies on message buffers



Web Application Start-Up Scenario



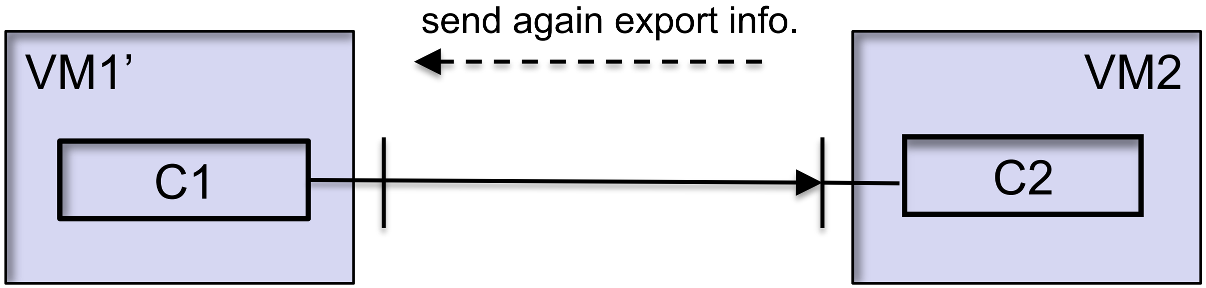
Web application model

Reliable Self-Deployment

- The self-deployment protocol also supports VM / configurator / network failures, detected using a heartbeat mechanism

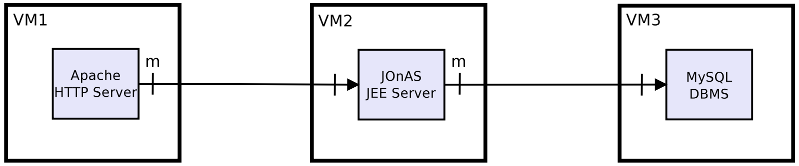
- The deployment manager re-instantiates the failed VM and sends messages to the other VMs to let them know of this failure / instantiation

- Those VMs send a specific message to the new VM and may repeat parts of the configuration protocol

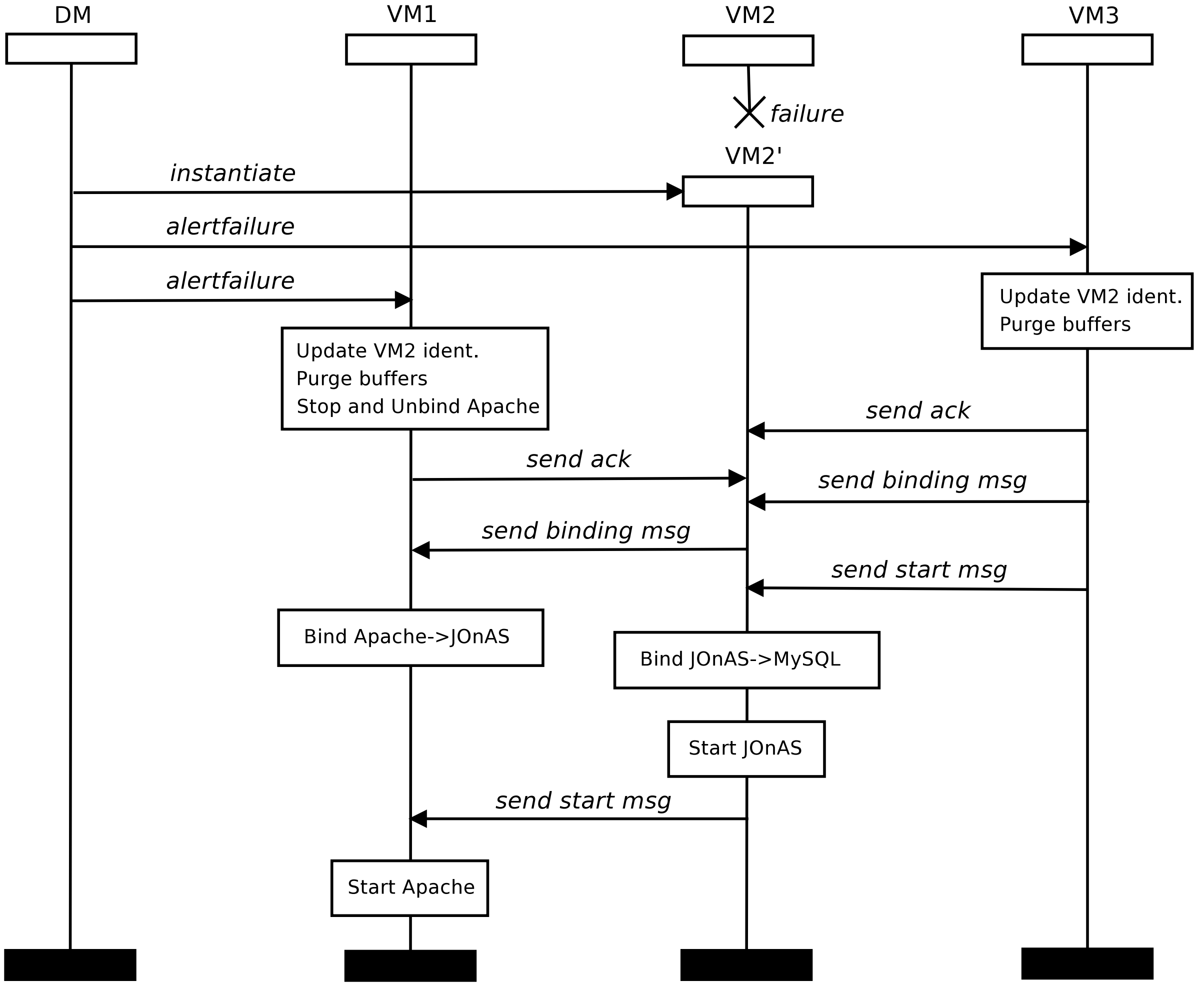


- Several failures may occur, either failures of different instances of a same VM or failures of different VMs

Web Application Failure Scenario



Web application model



Outline

1. Self-Deployment Protocol
2. Verification
3. Implementation
4. Concluding Remarks

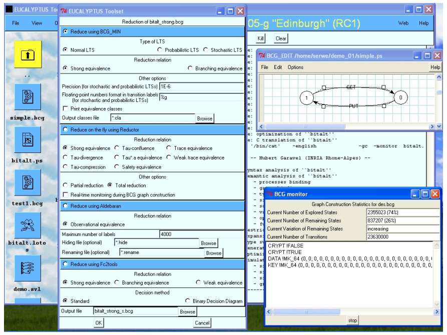
LNT and CADP

- LOTOS NT (LNT) is a value-passing process algebra with user-friendly syntax and operational semantics

- LNT is an imperative-like language where you can specify data types, functions (pattern matching and recursion), and processes

- LNT is one of the input languages of the CADP toolbox, which provides a large variety of verification techniques and tools

- We particularly used branching temporal logics and model checking techniques



Reduction of bitalt_strong.bcg

● Reduce using BCG_MIN

Type of LTS: ● Normal LTS ○ Probabilistic LTS ○ Stochastic LTS

Reduction relation: ● Strong equivalence ○ Branching equivalence

Other options

Precision (for stochastic and probabilistic LTSs): 1E-6

Floating-point numbers format in transition labels (for stochastic and probabilistic LTSs): %g

□ Print equivalence classes

Output classes file: *.cla

○ Reduce on the fly using Reductor

Reduction relation: ● Strong equivalence ○ Tau-confluence ○ Trace equivalence ○ Tau-divergence ○ Tau*.a equivalence ○ Weak trace equivalence ○ Tau-compression ○ Safety equivalence

Other options: ○ Partial reduction ● Total reduction

□ Real-time monitoring during BCG graph construction

○ Reduce using Aldebaran

Reduction relation: ● Observational equivalence

Maximum number of labels: 4000

Hiding file (optional): *.hide

Renaming file (optional): *.rename

○ Reduce using Fc2tools

Reduction relation: ● Strong equivalence ○ Branching equivalence ○ Weak equivalence

Decision method: ● Standard ○ Binary Decision Diagram

Output file: bitalt_strong_s.bcg

```
e: optimization of ``bitalt''
e: C translation of ``bitalt''
 '/bin/cat'    -english              -gc  -monitor  bitalt.

 -- Hubert Garavel (INRIA Rhone-Alpes) --

yntax analysis of ``bitalt''
emantic analysis of ``bitalt''
  - processes binding
```

Graph Construction Statistics for des.bcg

Current Number of Explored States	2355023 (74%)
Current Number of Remaining States	837207 (26%)
Current Variation of Remaining States	increasing
Current Number of Transitions	23630000

```
CRYPT !FALSE
CRYPT !TRUE
DATA !MK_64 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
KEY !MK_64 (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Specification in LNT

- The specification consists of at least 2,500 lines of code (data types, functions, processes)

- Data types describe the application model (components, ports, bindings, buffers, etc.)

- Functions are necessary for:
  - extracting information from the application model
  - describing buffers and basic operations on them
  - keeping track of the started components to know when components can be started

- Processes specify VMs (configurator, input and output buffer), the communication layer (MOM), and the system architecture consisting of VMs interacting through the MOM

Model Checking with CADP

- We identified and checked 15 safety and liveness properties that must be preserved by the protocol

- These properties specify final objectives to be fulfilled (1), architectural invariants (2), or ordering constraints (3, 4, 5)

  1. All components are eventually started

  2. A component cannot be started before the components it depends on through mandatory imports

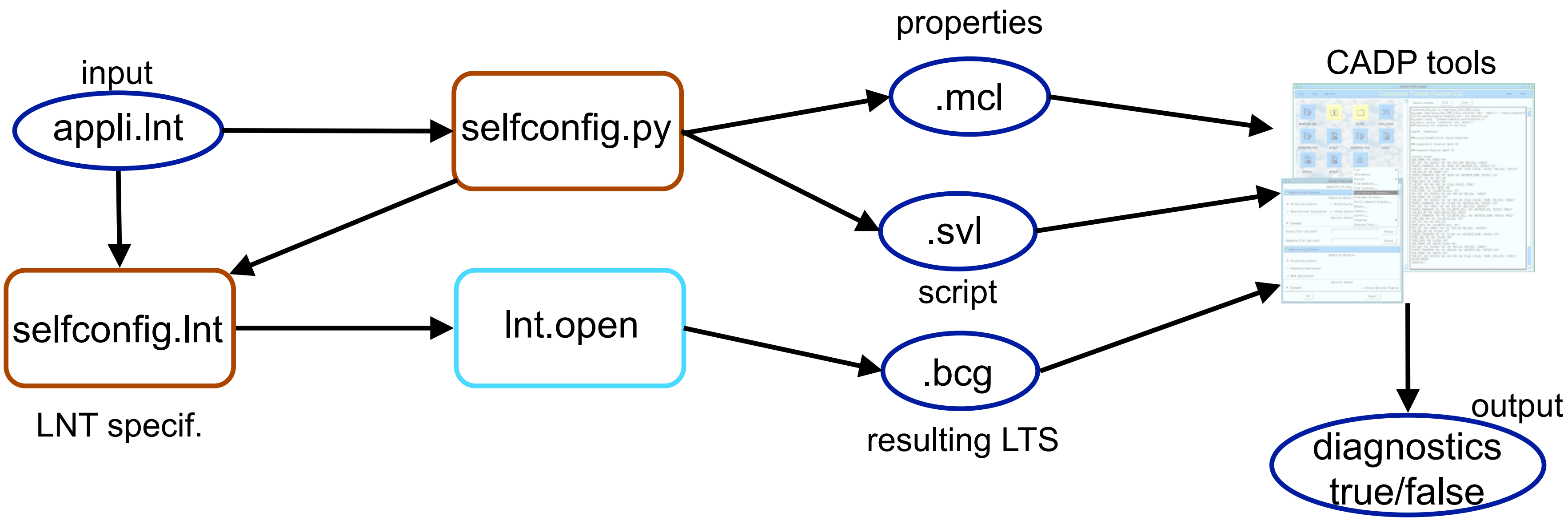
  3. After a VM fails, all other VMs are informed of that failure

  4. Each VM failure is followed by a new creation of that VM

  5. There is no sequence with two failures (same VM) without a VM creation between them

- They were specified in the MCL language and verified with the Evaluator 4.0 model checker

Tool Support



- Experiments were conducted on about 170 application models
- We were able to analyze up to 4 VMs with up to 5 failures in a few hours
- A bug was found in the configurator start-up part of the protocol ⇒ it was corrected in the Java implementation (Orange Labs)

Outline

1. Self-Deployment Protocol
2. Verification
3. Implementation
4. Concluding Remarks

VAMP Principles

- VAMP: Virtual Applications Management Platform

- VAMP first creates a new VM in which a deployment manager is instantiated

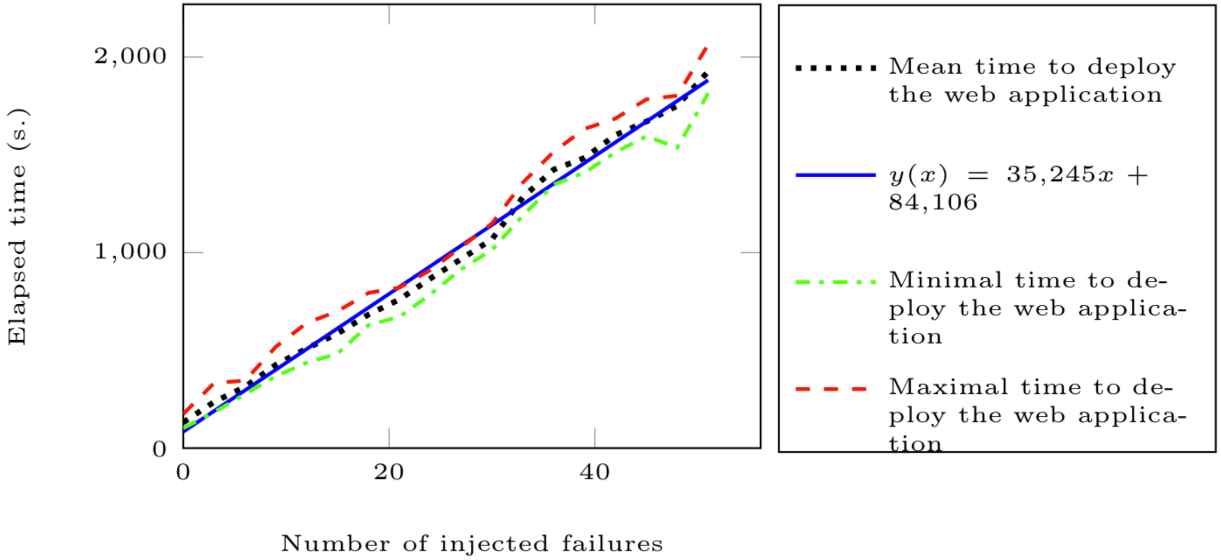
- The DM generates virtual images and instantiates them as VMs in one or several Infrastructure-as-a-Service platforms

- Each virtual image embeds the configurator (written in Java), which encodes most of the self-deployment protocol

- All the participants (DM and configurators) communicate through the AAA asynchronous message-oriented middleware

Evaluation

- The evaluation process aims at measuring the time to deploy the 3-tier Web application (running example) while randomly injecting failures



- The time to deploy the Web application increases linearly with the number of failures

Outline

1. Self-Deployment Protocol
2. Verification
3. Implementation
4. Concluding Remarks

Concluding Remarks

- We propose and design an innovative, decentralized protocol to automatically deploy cloud applications consisting of interconnected software components hosted on several VMs

- The deployment process is able to detect and handle VM and network failures, and always succeeds in configuring the application

- We verified that the protocol respects some key properties using formal specification languages and model checking techniques

- We implemented the protocol in Java and applied it to real-world applications for evaluation purposes

- Main perspective: dynamic reconfiguration of cloud applications