Distributed Configuration & Service Change Planning

Paul Anderson & Herry
<dcspaul@ed.ac.uk>
<h.herry@sms.ed.ac.uk>

http://homepages.inf.ed.ac.uk/dcspaul/publications/hp-2012.pdf

THE UNIVERSITY *of* EDINBURGH
informatics

cisa Centre for Intelligent Systems and their Applications

Overview

❶ Configuration Research (paul)
- overview of area & current work

❷ Agent-based configuration with lcc (paul)
- centralised policy and distributed execution
- an example using the "lightweight coordination calculus"

❸ Planning for configuration change (herry)
- centralised planning and workflow execution
- distributed workflow execution using "behavioural signatures"
- distributed workflow execution using lcc

Some Current Projects

- Constraint-based specification (John Hewson)
- Planning for configuration change (Herry)
- Agents and interaction models for VM Migration
- Student projects
  - distributed planning for service changes
  - planning deployments on the HP public cloud
  - machine learning for VM migration
- Other interests
  - Semantics, provenance and security of configuration specifications

Implementing Virtual Machine Migration Policies With LCC

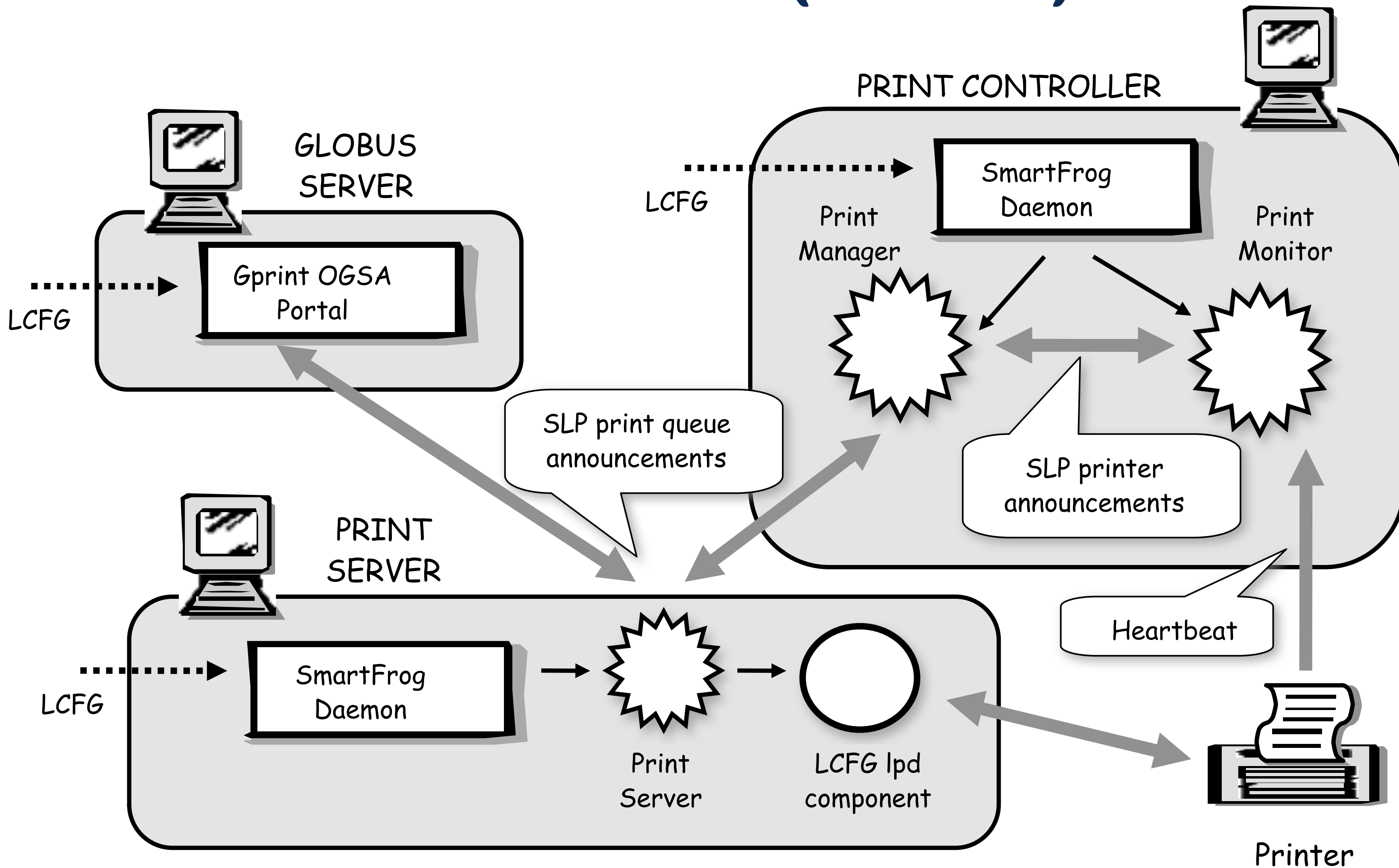
Work with Shahriar Bijani
<S.Bijani@sms.ed.ac.uk>

http://homepages.inf.ed.ac.uk/s0880557

Centralised Configuration?

- Centralised configuration
  - allows a global view with complete knowledge
- But ...
  - it is not scalable
  - it is not robust against communication failures
  - federated environments have no obvious centre
  - different security policies may apply to different subsystems
- The challenge ...
  - devolve control to an appropriately low level
  - but allow high-level policies to determine the behaviour

GPrint (2003)



- Distributed configuration with centralised policy
- Subsystem-specific mechanisms

“OpenKnowledge” & LCC

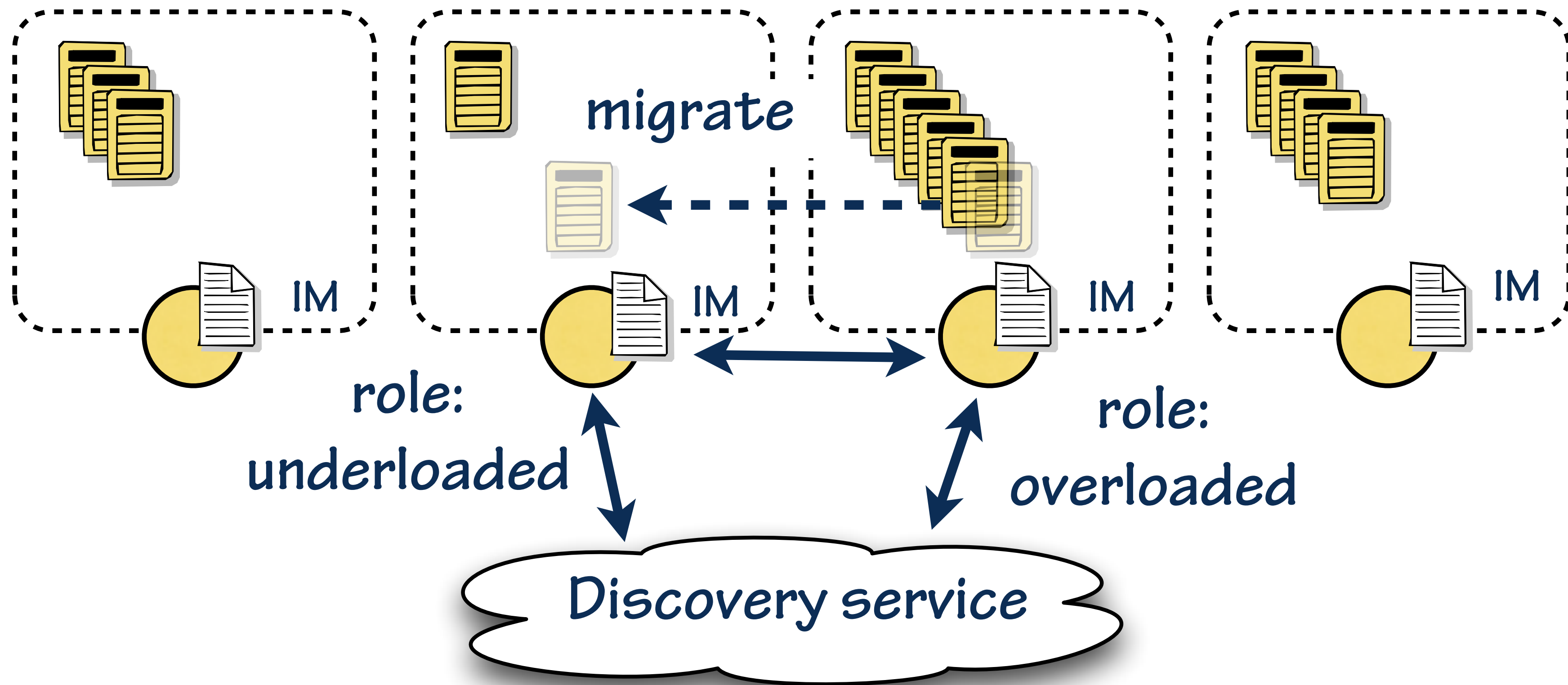
- Agents execute “interaction models”
- Written in a “lightweight coordination calculus” (LCC)
- This provides a very general mechanism for doing distributed configuration
- Policy is determined by the interaction models themselves which can be managed and distributed from a central point of control
- The choice of interaction model and the decision to participate in a particular “role” remains with the individual peer
  - and hence, the management authority

A Simple LCC Example

```
a(buyer, B) ::
 ask(X) => a(shopkeeper, S) then
 price(X,P) <= a(shopkeeper, S) then
 buy(X,P) => a(shopkeeper, S)
             ← afford(X, P) then
 sold(X,P) <= a(shopkeeper, S)

a(shopkeeper, S) ::
  ask(X) <= a(buyer, B) then
  price(X, P) => a(buyer, B)
                 ← in_stock(X, P)then
  buy(X,P) <= a(buyer, B) then
  sold(X, P) => a(buyer, B)
```

An Example: VM Allocation



- Policy 1 - power saving
  - pack VMs onto the minimum number of physical machines
- Policy 2 - agility
  - maintain an even loading across the physical machines

An Idle Host

```
a(idle, ID1) ::
      null
      ← overloaded(Status)
    then
      a(overload(Status), ID1)
  ) or (
      null
      ← underloaded(Status)
    then
      a(underload(Status), ID1)
  ) or (
    a(idle, ID1)
  )
```

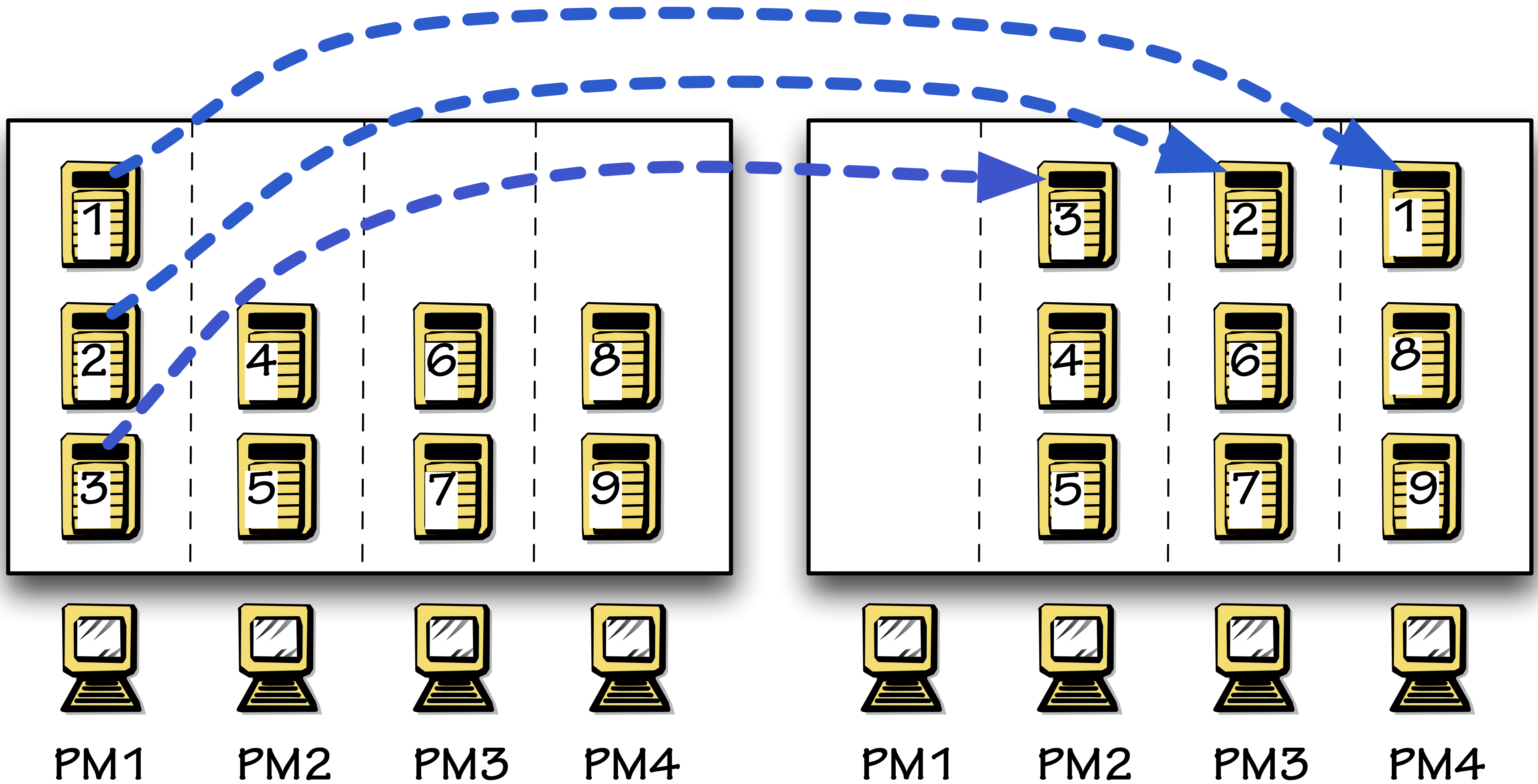
An Overloaded Host

```
a(overloaded(Need), ID2) ::
    readyToMigrate(Need)
    => a(underloaded, ID3)
  then
    migration(OK)
    <= a(underloaded, ID3)
  then
    null
    ← migration(ID2, ID3)
  then
    a(idle, ID2)
```

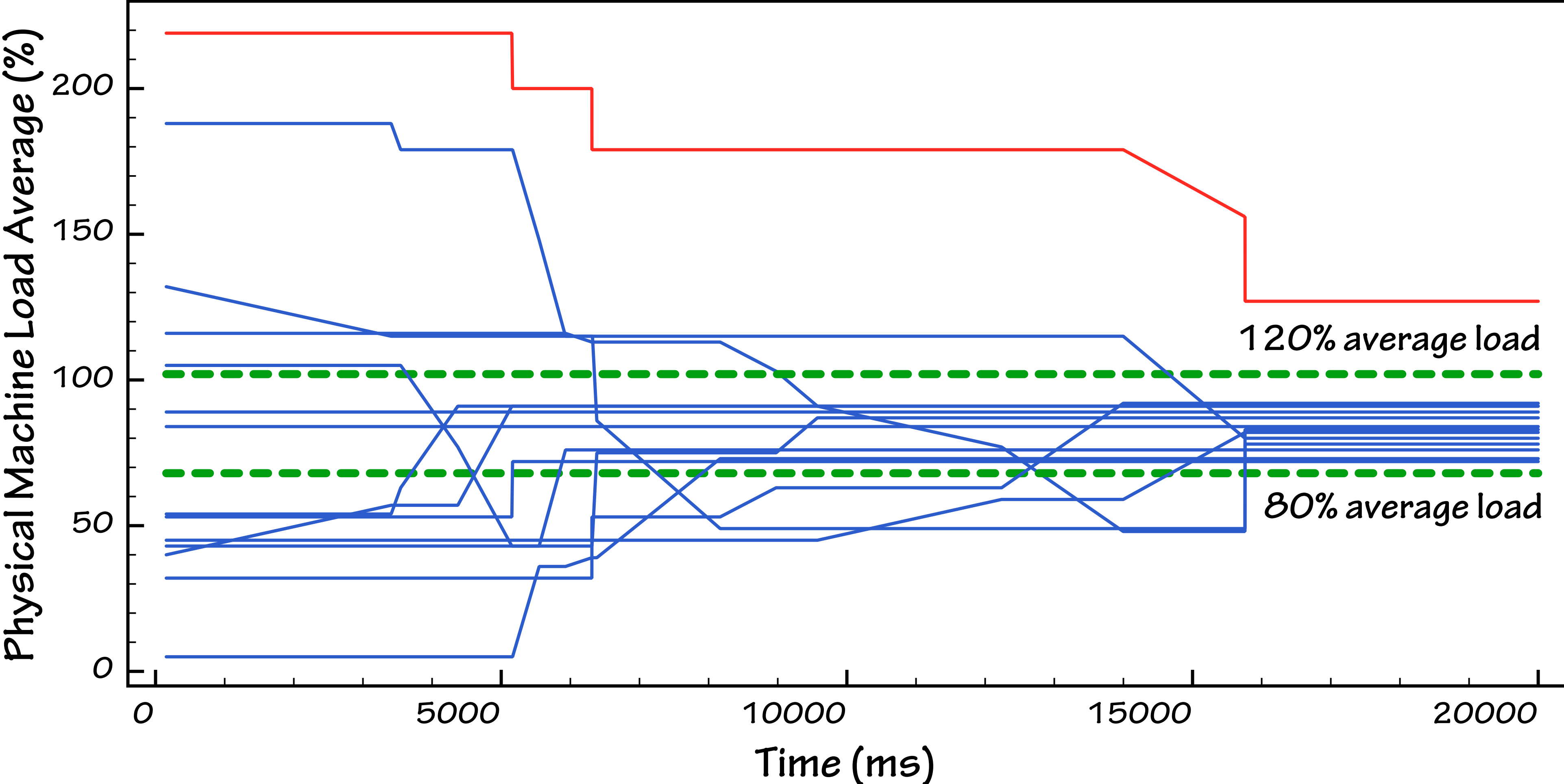
An Underloaded Host

```
a(underloaded(Capacity), ID3) ::
    readyToMigrate(Need)
    <= a(overloaded, ID2)
  then
    migration(OK)
    => a(overloaded, ID2)
    ← canMigrate(Capacity, Need)
  then
    null ← waitForMigration()
  then
    a(idle, ID3)
```

Migration Example



A Simulation



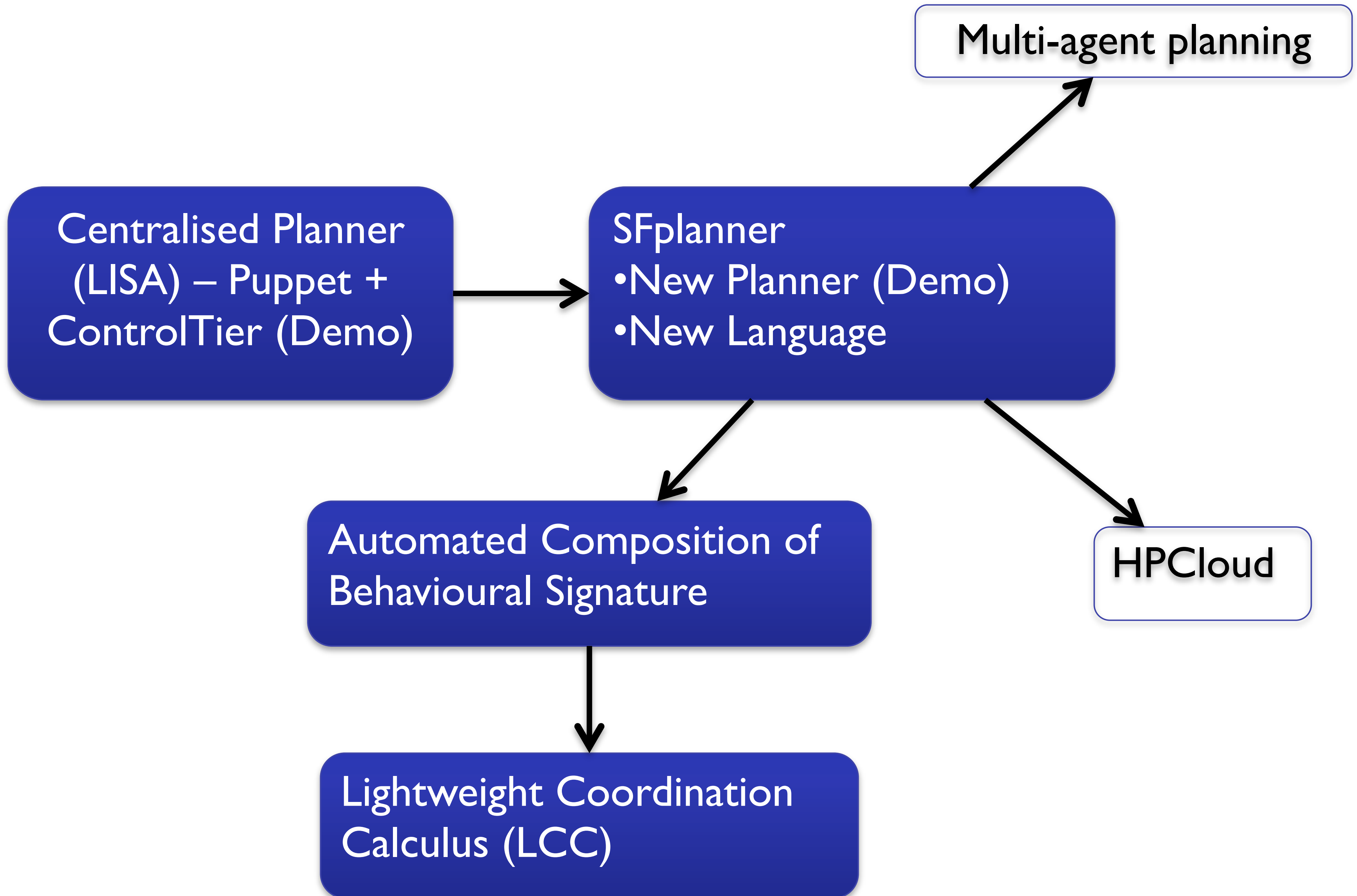
Some Issues

- LCC can be used to implement more sophisticated protocols - such as "auctions" which are ideal for many configuration scenarios
- But some things are hard to do without global knowledge
  - balance the system so that all the machines have exactly the same load?
- Handling errors and timeouts in an unreliable distributed system is hard

Planning for Configuration Changes

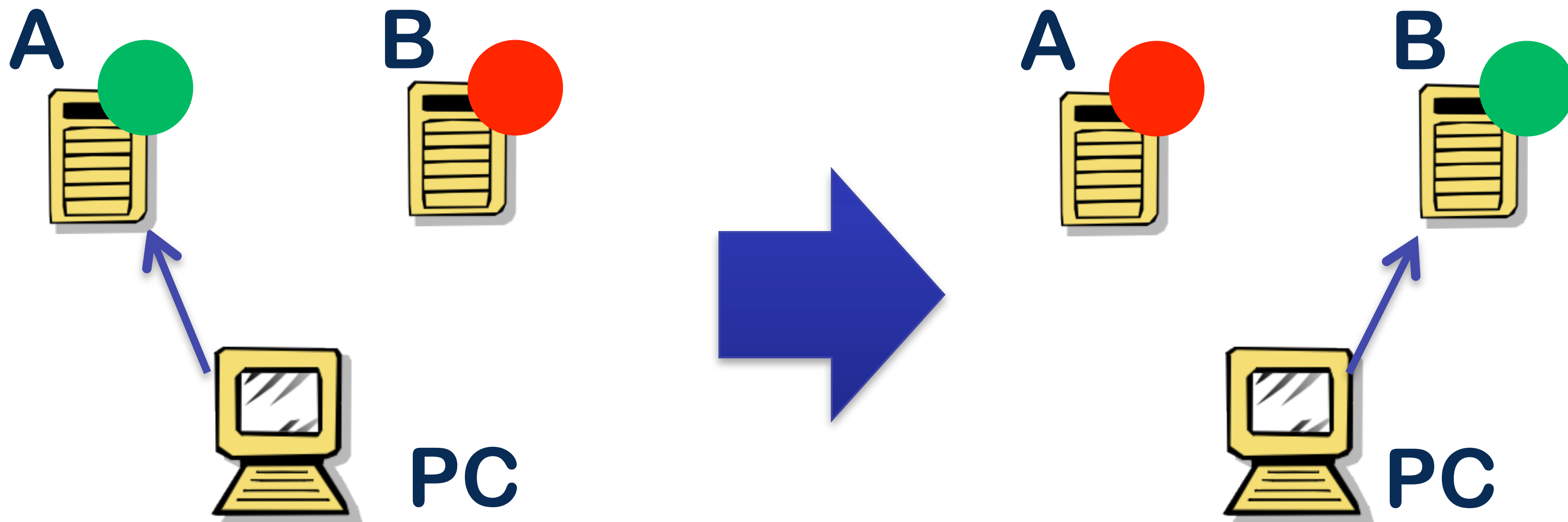
HP Innovation Research Project

Overview



Declarative approach

- Most commonly used today
- Popular tools: Puppet, Chef, LCFG
- Critical shortcomings
  - Indeterminate order execution of actions
  - Could violates the system's constraints



Client must always refer to a running server

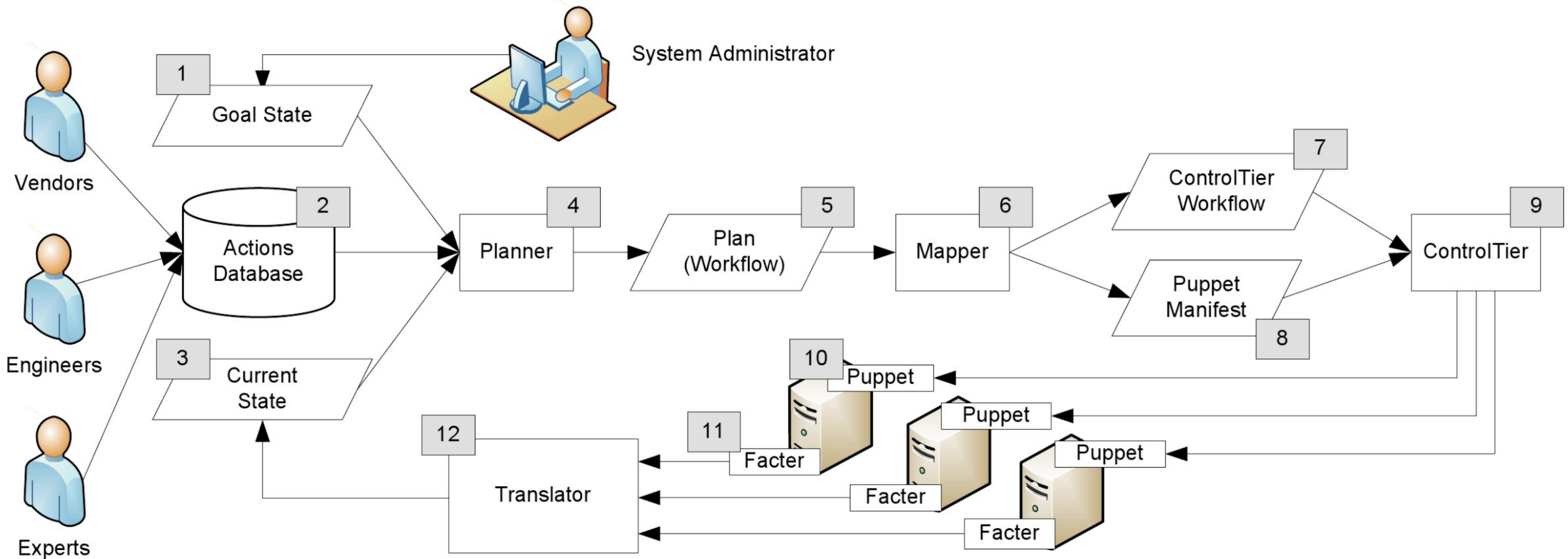
Solutions

- Declarative tools
  - Possible sequences of states

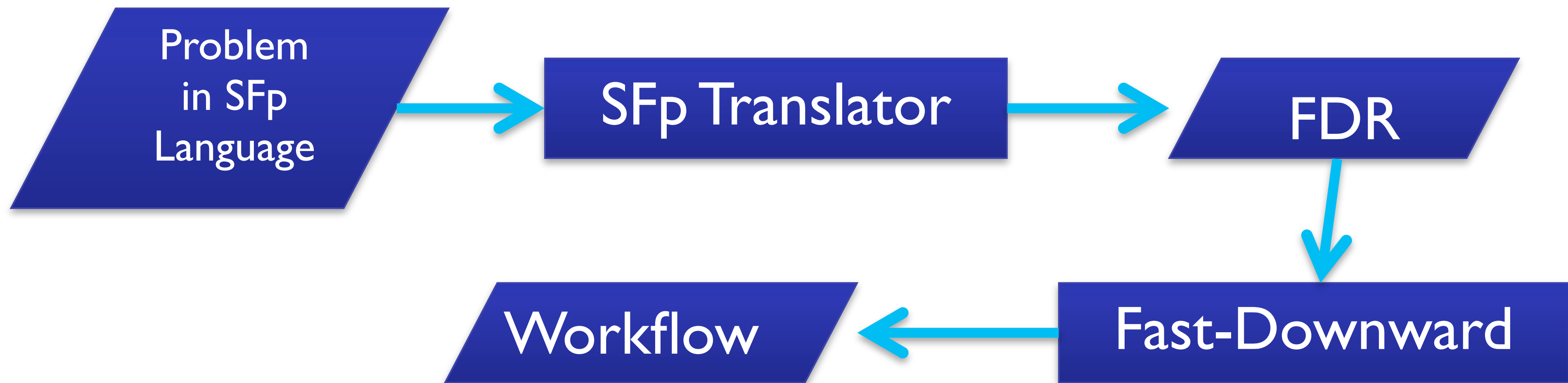
1)	A.running = false	PC.refer = B	B.running = true	X
2)	PC.refer = B	A.running = false	B.running = true	X
3)	B.running = true	A.running = false	PC.refer = B	X
4)	A.running = false	B.running = true	PC.refer = B	X
5)	PC.refer = B	B.running = true	A.running = false	X
6)	B.running = true	PC.refer = B	A.running = false	√

  - Highly likely producing the wrong sequence!
- Our prototype
  - Automated planning technique to generate the workflow
  - Each action has pre- and post-conditions

System Architecture (LISA '11)

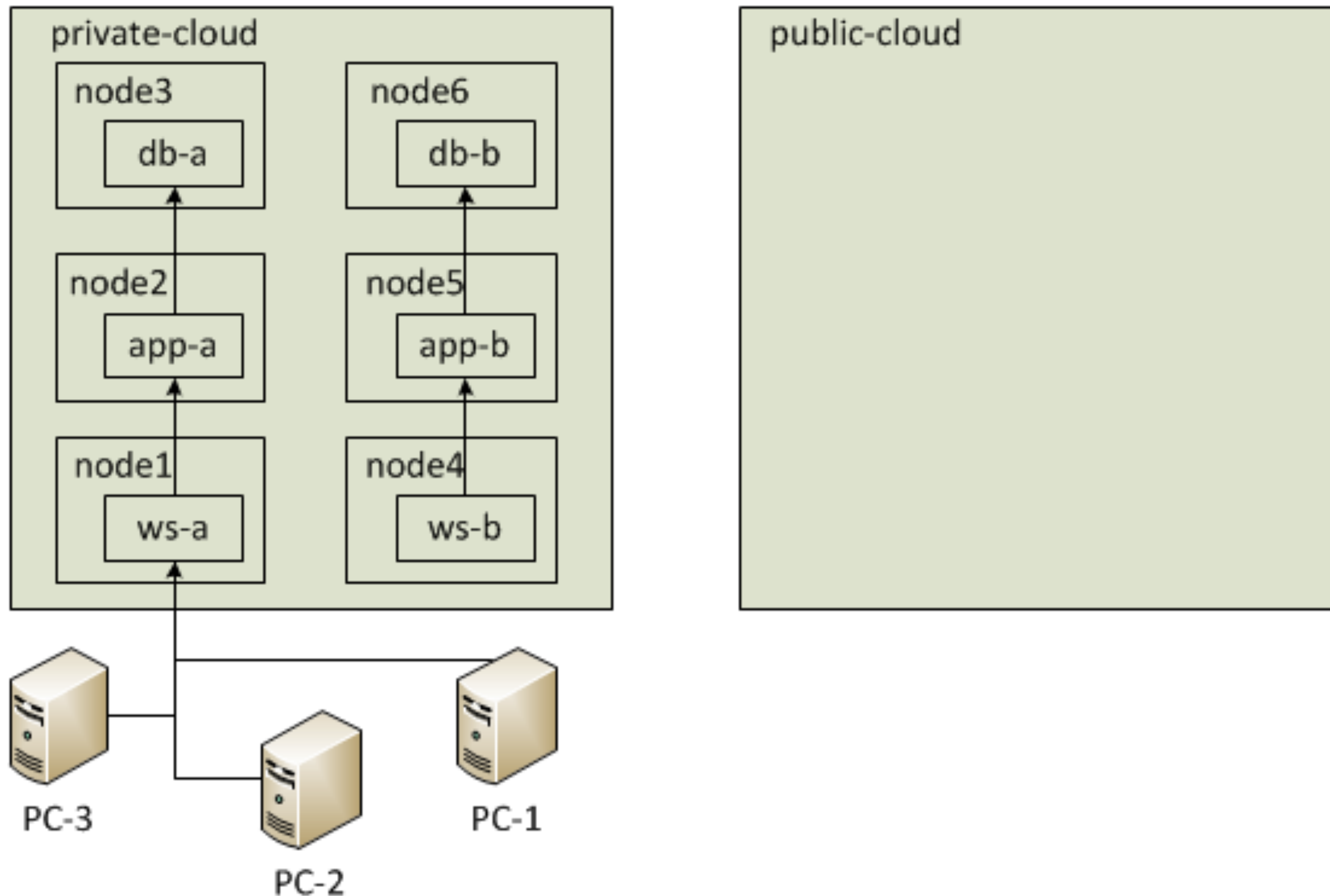


SFp planning system



- SFp language – object-oriented planning language
- Web service interface
  - Submit planning problem using HTTP POST
- Implemented as an OSGi Bundle
  - OSGi platform: Equinox or Felix
  - Linux OS
- http://homepages.inf.ed.ac.uk/s0978621/sfp.html

SFp planning system



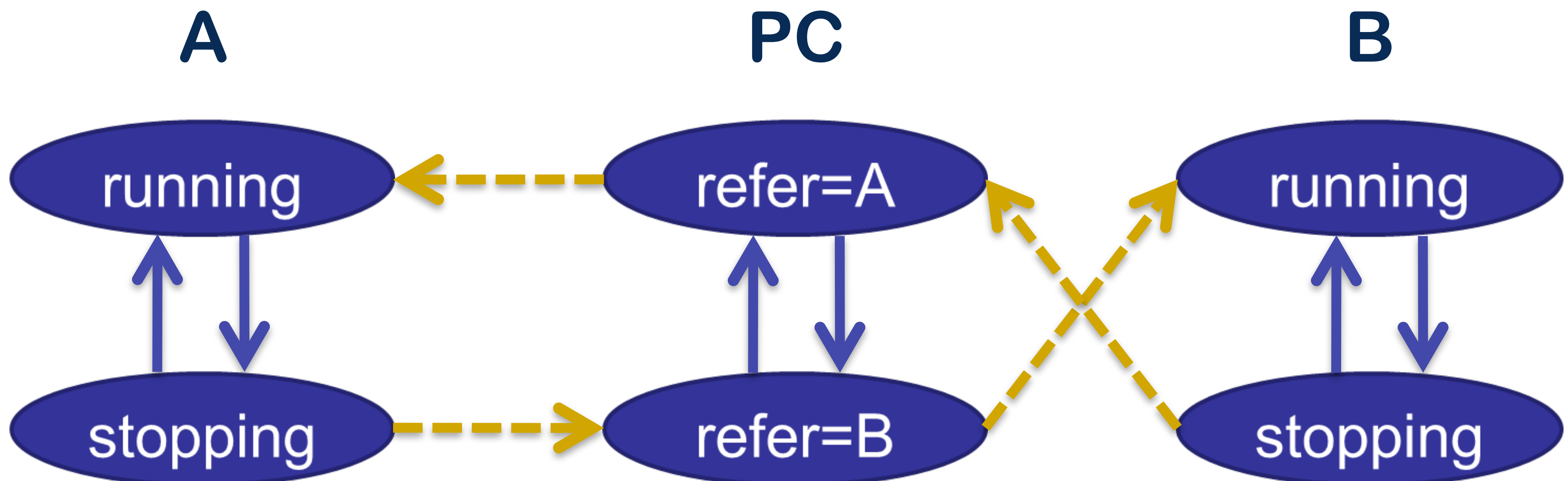
- Demo: http://hpvm2.diy.inf.ed.ac.uk

Centralised Architecture

- Central Controller – generates and orchestrates the execute of the workflow
- Problems – failure on the central controller
  - The managed system is out of control
  - Must compute the workflow for every changes
- Proposed solution
  - Executing the workflow in distributed way
  - Implant the pre-compiled workflow onto the components
  - Employ Behavioural Signature model

Behavioural Signature (BSig)

- Component can have state-dependencies
- If a change occurs, each component determines
  - What action
  - When to be executed
- Cascading effects



BSig: Manual Composition

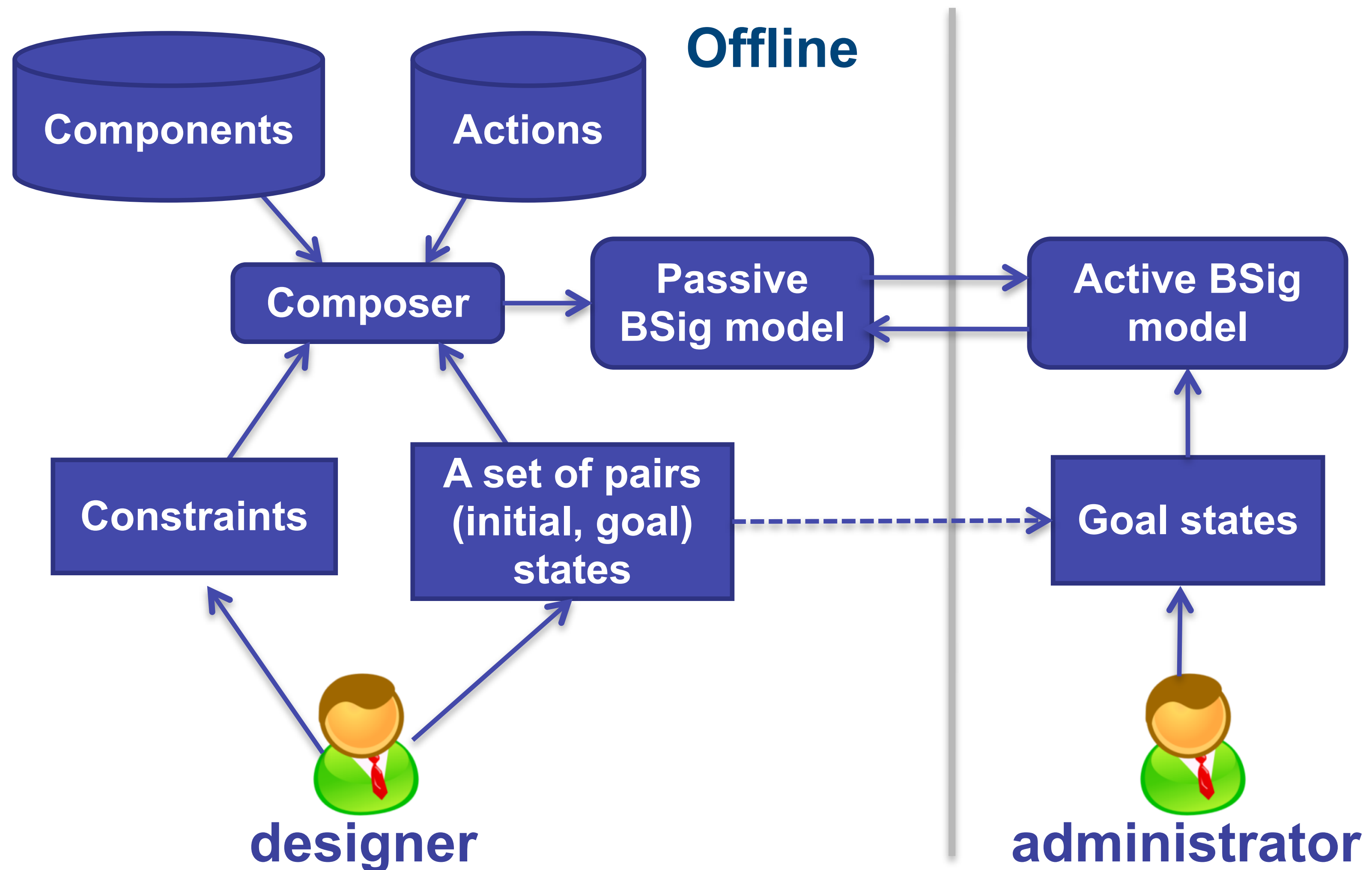
- Error prone, time consuming, hard to prove that the result is correct
- Complex task
- M components, N states per component

  $(M^2 - M)(N^2)$ possible state dependencies
- Difficult to solve deadlock situation
- Proposed solution: automated composition

BSig: Automated Composition

- Fact – define the state-dependencies, define the workflow
- User works in planning domain
  - Defines a set of pairs (initial, goal) states
  - Defines the global constraints
- Experts or engineers define the actions
- Use the planner to generate the workflow
- The generated workflow is translated into state-dependencies

BSig: Automated Composition



Inputs for Composer

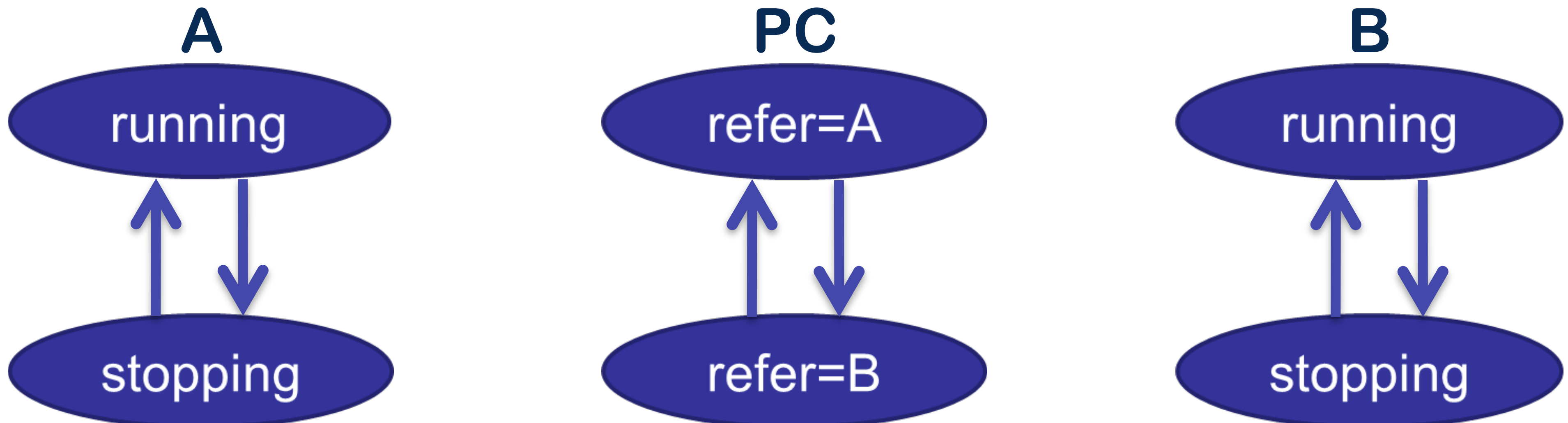
- Actions

```
startServer {
    s    *Server
    precondition { }
    postcondition {
        $s.running true
    }
}
```

```
stopServer {
    s    *Server
    precondition { }
    postcondition {
        $s.running false
    }
}
```

```
changeReference {
    c    *Client
    s    *Server
    precondition {
        $s.running true
    }
    postcondition {
        $c.refer $s
    }
}
```

- Components



Inputs for Composer

- Constraints

[None]

- Pairs (Initial, Goal)

Pair #1
Initial: A.running, B.stopping, PC.refer=A
Goal: A.stopping, B.running, PC.refer=B

Pair #2
Initial: A.stopping, B.running, PC.refer=B
Goal: A.running, B.stopping, PC.refer=A

designer

Composition Process

- Pair #1
- Generated Workflow

  startService(B) → changeReference(PC, B) → stopService (A)
- State-Transition

  <none> → B.running → PC.refer=B → A.stopping
- State-dependencies

  -<none> → B.running

  -B.running → PC.refer=B

  -PC.refer=B → A.stopping

Composition Process

- Pair #2
- Generated Workflow

  startService(A) → changeReference(PC, A) → stopService (B)
- State Transition

  <none> → A.running → PC.refer=A → B.stopping
- State-dependencies

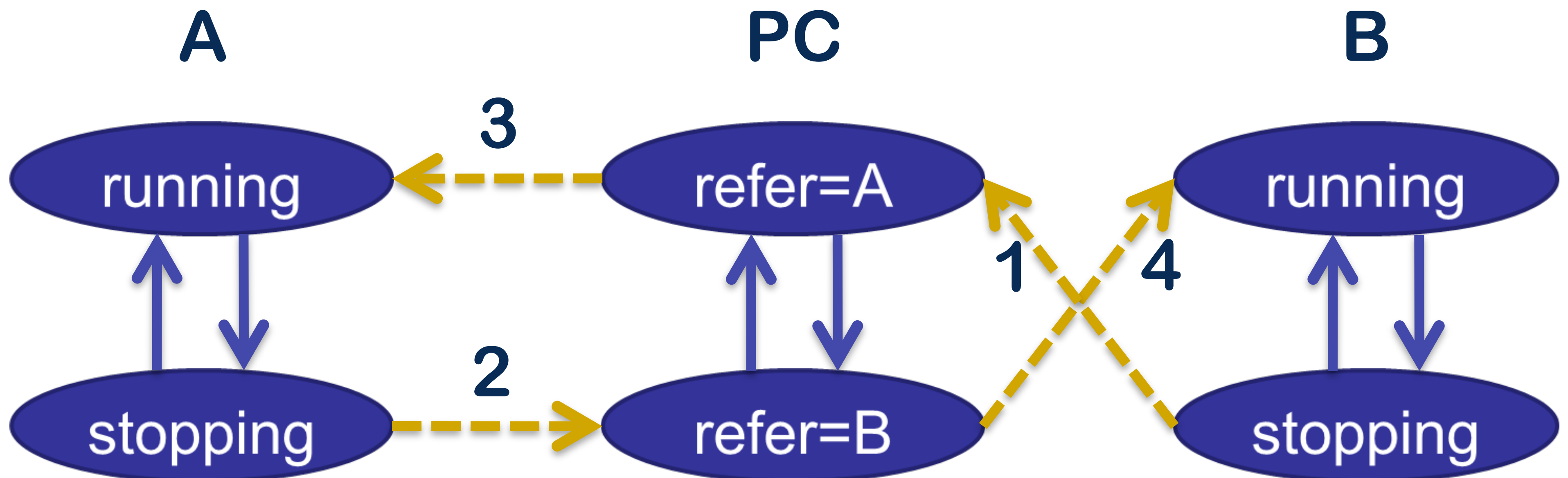
  -<none> → A.running

  -A.running → PC.refer=A

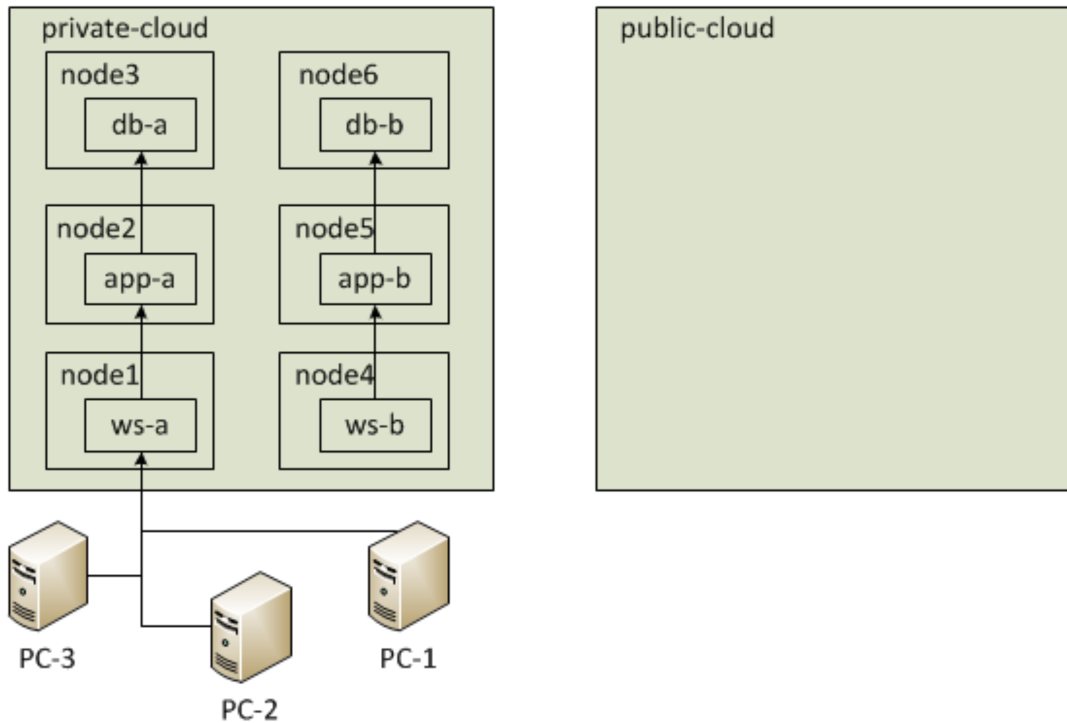
  -PC.refer=A → B.stopping

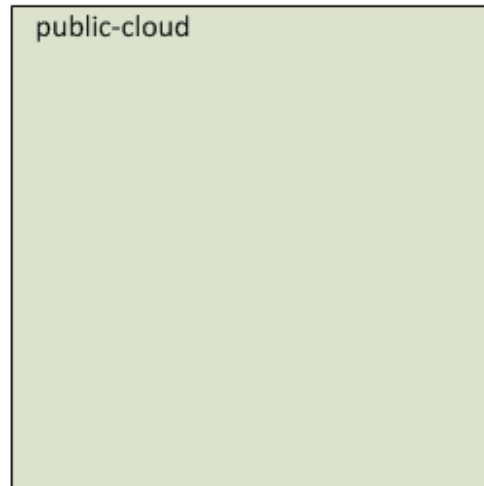
Composition Process

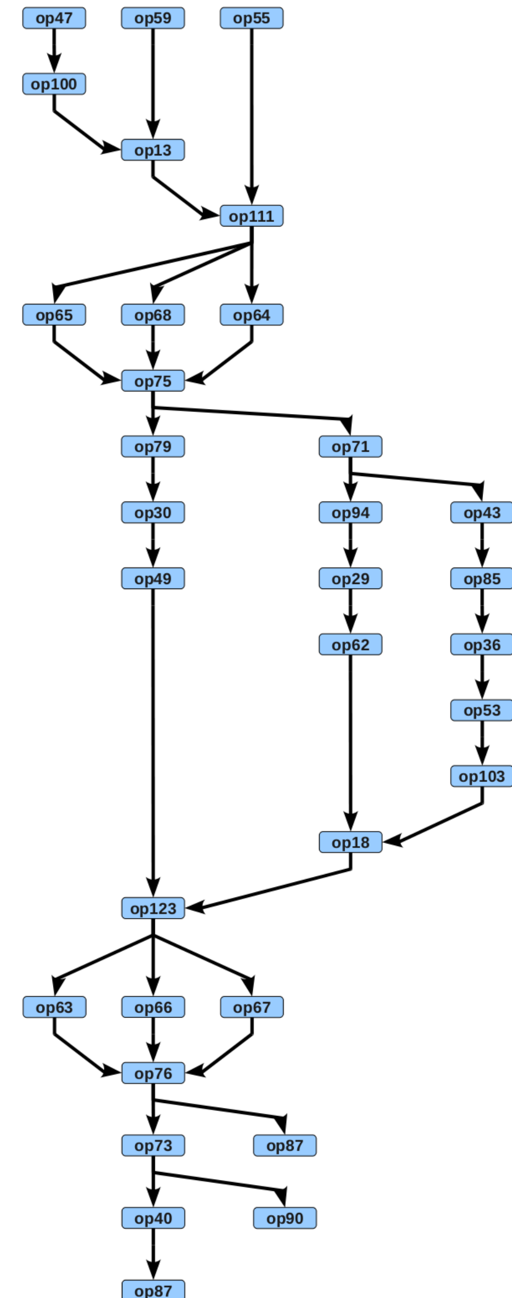
- Result State-Dependencies
  1. B.running → PC.refer=B
  2. PC.refer=B → A.stopping
  3. A.running → PC.refer=A
  4. PC.refer=A → B.stopping



Cloud Burst of 3-Tier WebApps



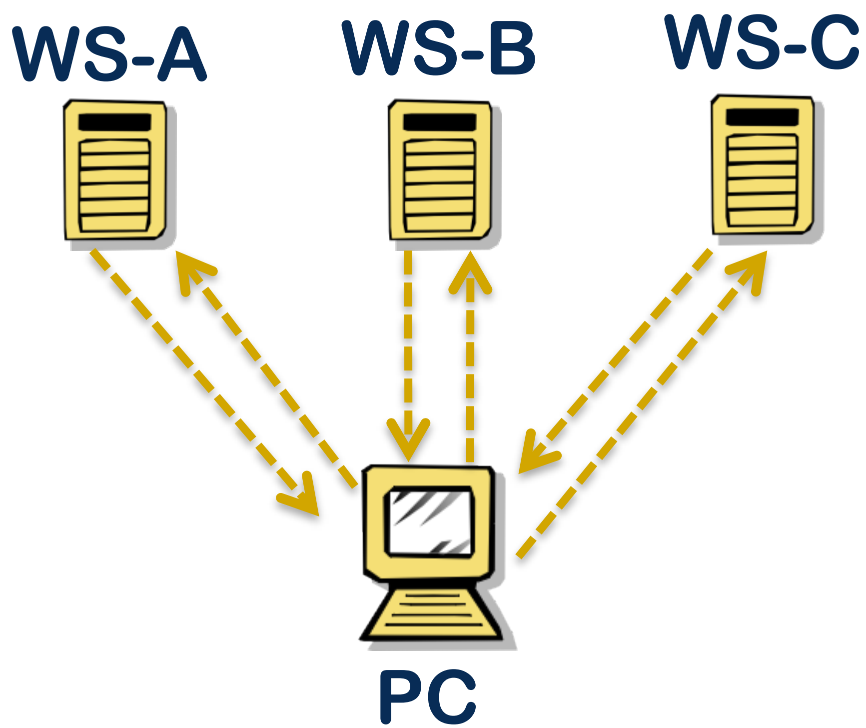


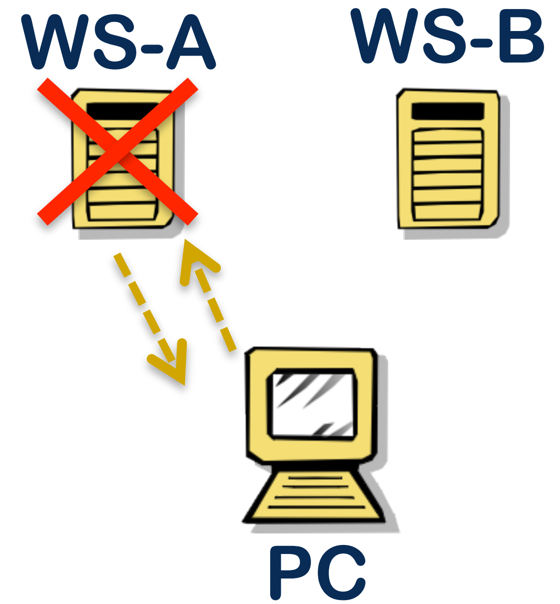


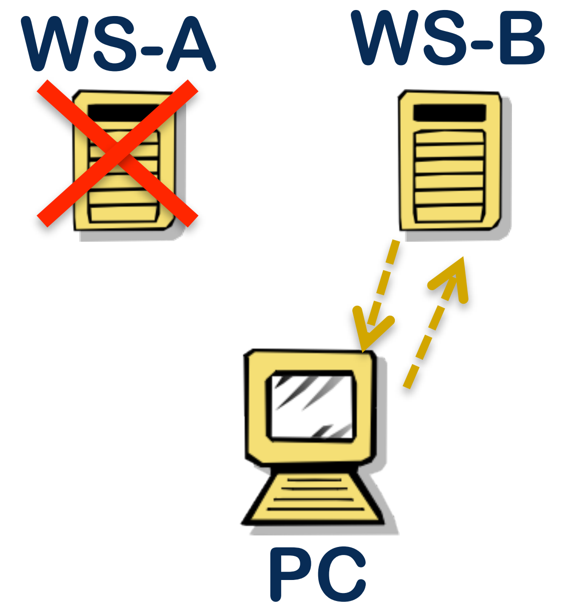
Problem in BSig

- BSig's state dependency – defines the dependency between two instances
- Resource pool problem
  - Using one of multiple resources requires multiple state-dependencies
- Repairing problem
  - Replacing a failure component
  - Require repairing the state-dependency

Problem in BSig





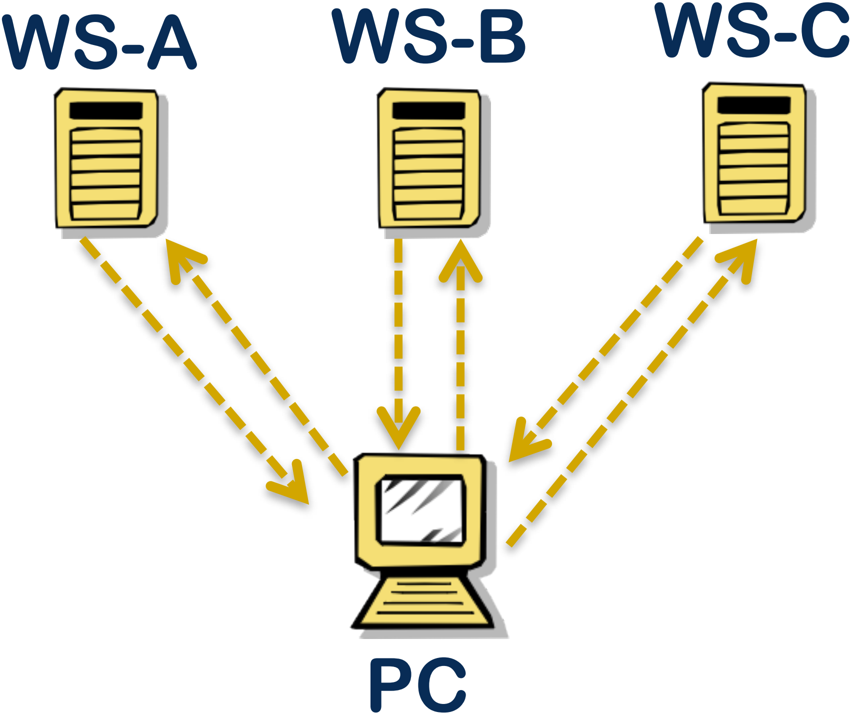


LCC

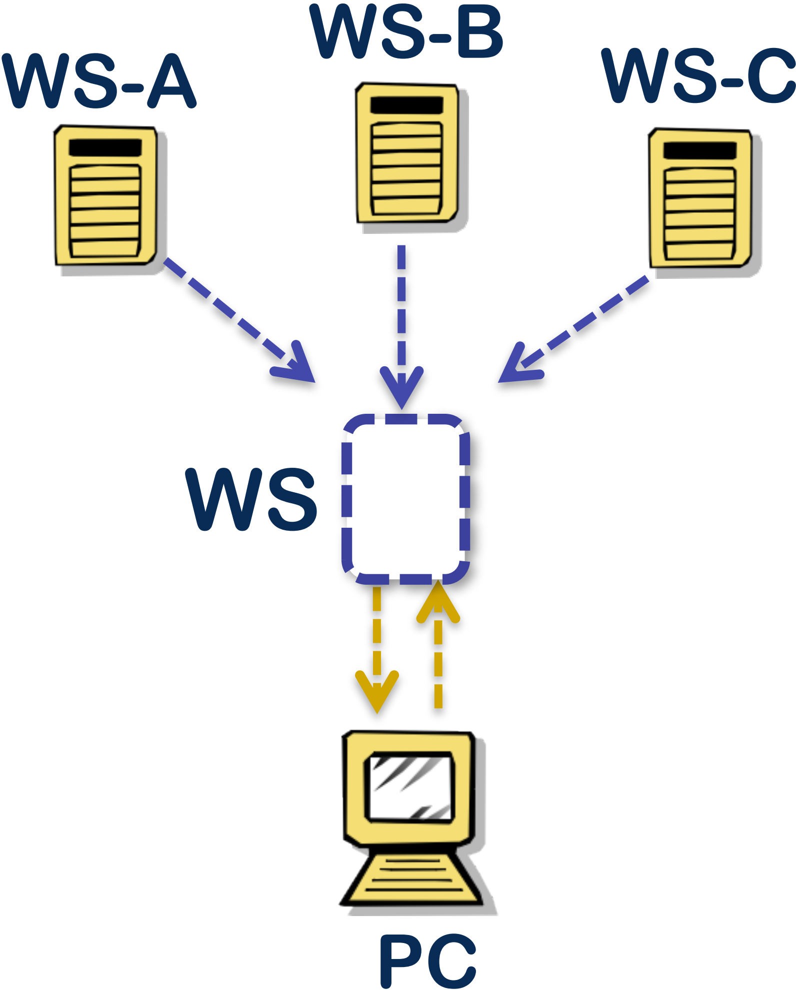
- Lightweight Coordination Calculus
- Define relation between roles, not instances
- Notation for clearly defining the interaction between components in BSig
- Some interpreters
  - OpenKnowledge, LiJ (Java)
  - Okeileidh (Javascript, Node.js)

BSig vs LCC

- BSig



- LCC
  - WS is a "Role"

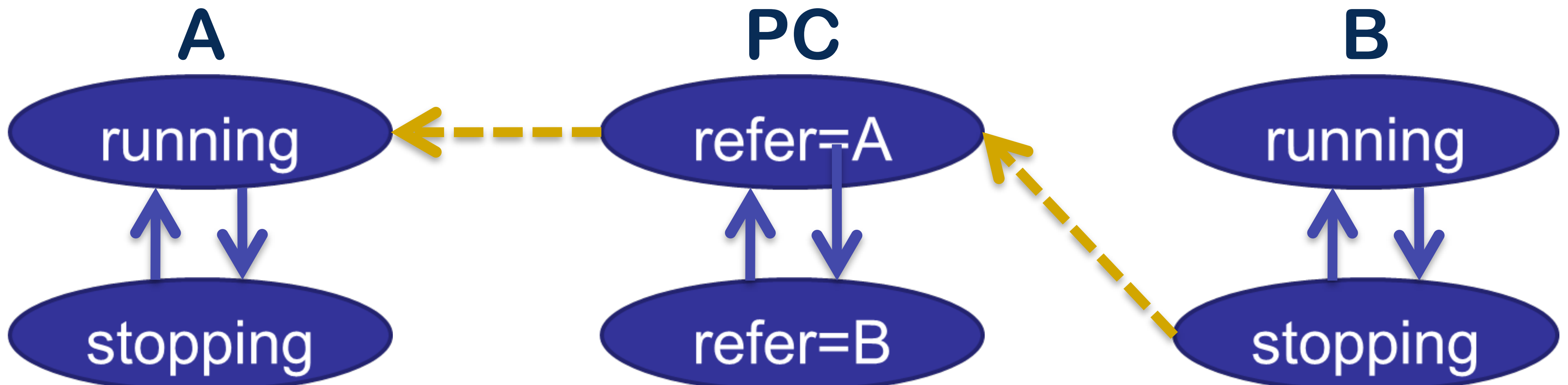


BSig in LCC Notation

```
a(ws-b, B) ::
   set(refer,ws-a) => a(pc, PC) then
   achieve(running,false) <- at(refer,ws-a) <= a(pc, PC).

a(pc, PC) ::
   set(refer,ws-a) <= a(ws-b, B) then
   set(running,true) => a(ws-a, A) then
   at(running,true) <= a(ws-a, A) then
   at(refer,ws-a) => a(ws-b, B) <- achieve(refer,ws-a).

a(ws-a, A) ::
   set(running,true) <= (pc, PC) then
   at(running,true) => a(pc, PC) <- achieve(running,true).
```



LCC Design Pattern for BSig

```
a(webService, WS) ::
    set(Variable, Value) <= a(pc, PC)        then
    a(webServiceResponder(Variable, Value), WS) then
    at(Variable, Value) => a(pc, PC).

a(webServiceResponder(Variable, Value), R) ::
    null <-- current(Variable, Value)        or
    (
       null <- getPrecondition(Variable, Value, Comps, Vars, Vals) then
       a(webServicerequester(Comps, Vars, Vals), R) then
       null <- achieve(Variable, Value)
    ) or
    null <- achieve(Variable, Value).

a(webServiceRequester(C, Vars, Vals), R) ::
    null <- C = [] or
    (
       set(Var1, Val1) => a(C1, CX) <- list(C,C1,Cr) && list(Vars,Var1,Varr)
          && list(Vals,Val1,Valr) then
       at(Var1, Val1) <= a(C1, CX) then
       a(webServiceRequester(Cr, Varr, Valr), R)
    ).
```

Future Works

- Automated composition of BSig for real use cases
- Adopt LCC relation on BSig
- Hierarchical composition for large scale system

Thank you!

Q & A