The Niche Platfrom

Vladimir Vlassov vladv@kth.se
Ahmad Al-Shishtawy ahmadas@kth.se
Leif Lidbäck leifl@kth.se

Royal institute of Technology (KTH), Stockholm, Sweden

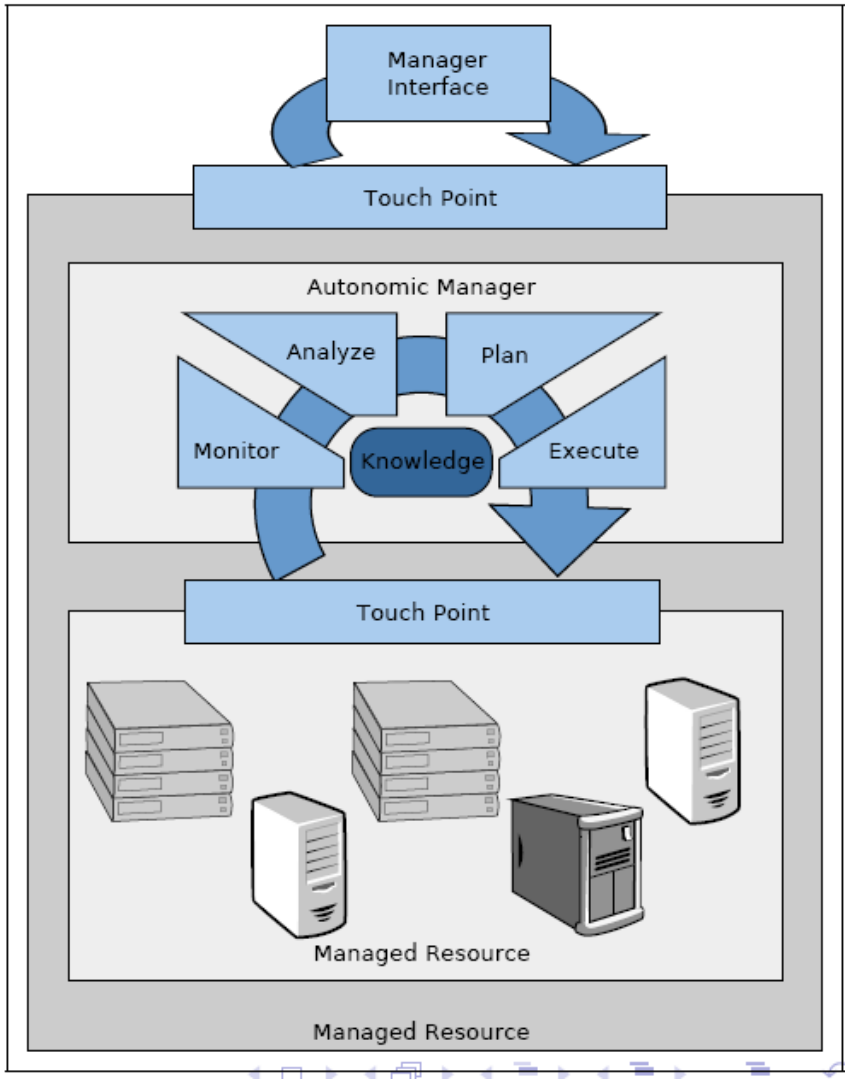
Niche seminar, FTRD, Paris, Oct 9, 2009

Background

- Primary issues in system ownership (including administration, maintenance, etc)
  - Complexity
  - Cost
- How to address these issues?
  - Autonomic computing
    - Building systems which can manage themselves with minimal human intervention
    - Requires a platform: model, API, execution environment (middleware) and a language support
- Grid4All project
  - Self-* grid

The Autonomic Computing Archecture

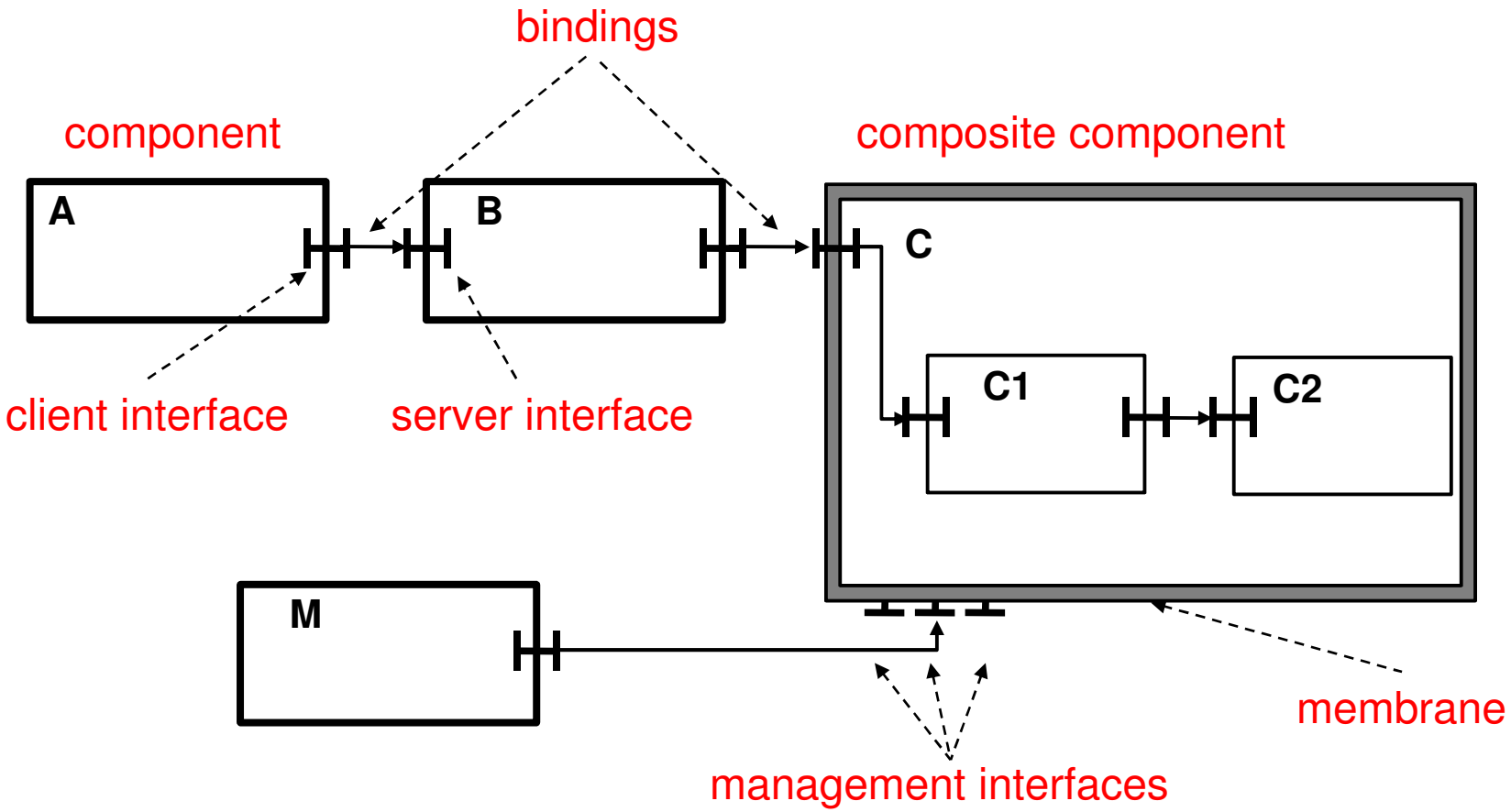
- Managed Resource
- Touchpoint
  - Sensors
  - Actuators
- Autonomic Manager (MAPE loop)
  - Monitor
  - Analize
  - Plan
  - Execute
- Knowledge Source
- Communication
- Manager Interface



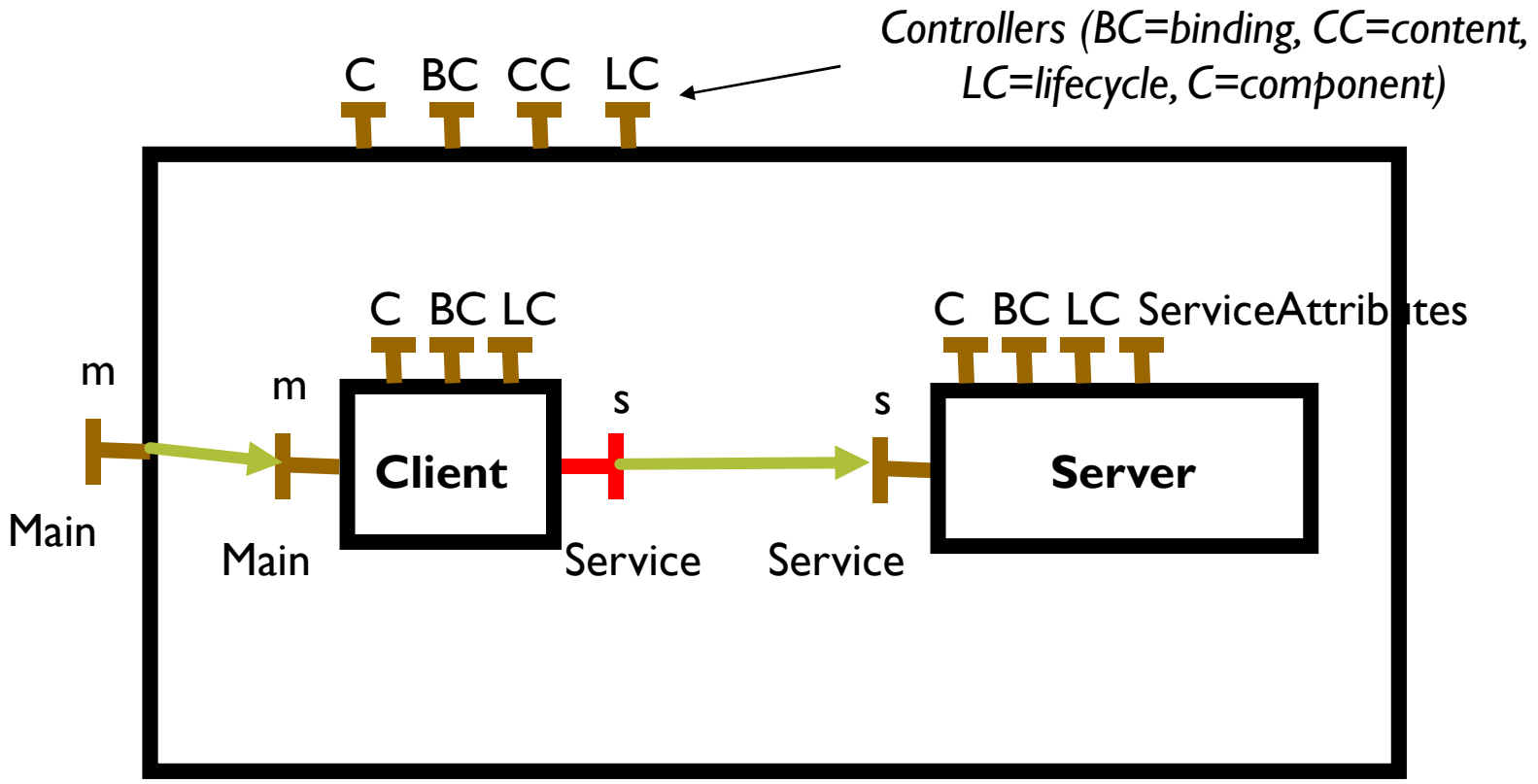
Niche (http://niche.sics.se)

- A platform for developing, deploying and execution of self-managing distributed systems and applications
- Implements the autonomic computing architecture (allows building MAPE loops)
- Includes a component-based programming model (Fractal), API, and an execution environment
  - Supports Component Group abstraction with One-to-All and One-to-Any bindings
  - Separates programming of functional and management parts
  - Management Element abstractions: watchers, aggregators, managers, executors
- Transparent replication of management elements for robustness
- Uses a structured overlay network (SON)

A Component-Based Application

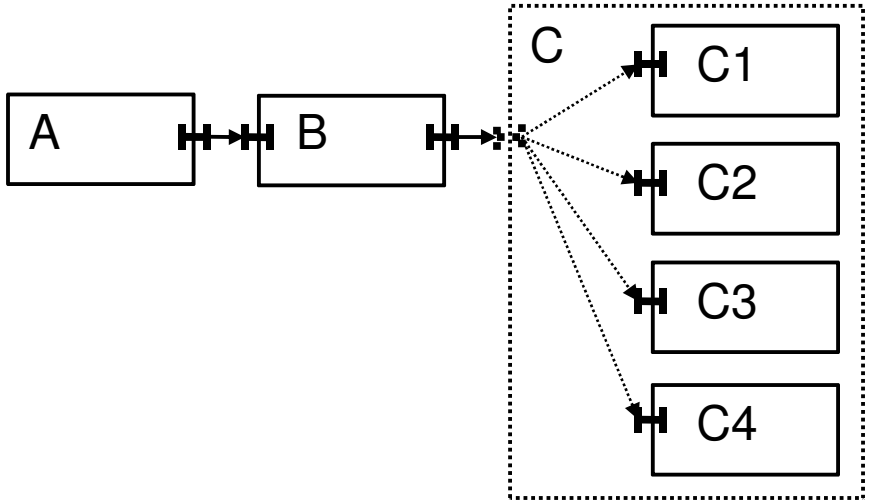


A Client-Server Application

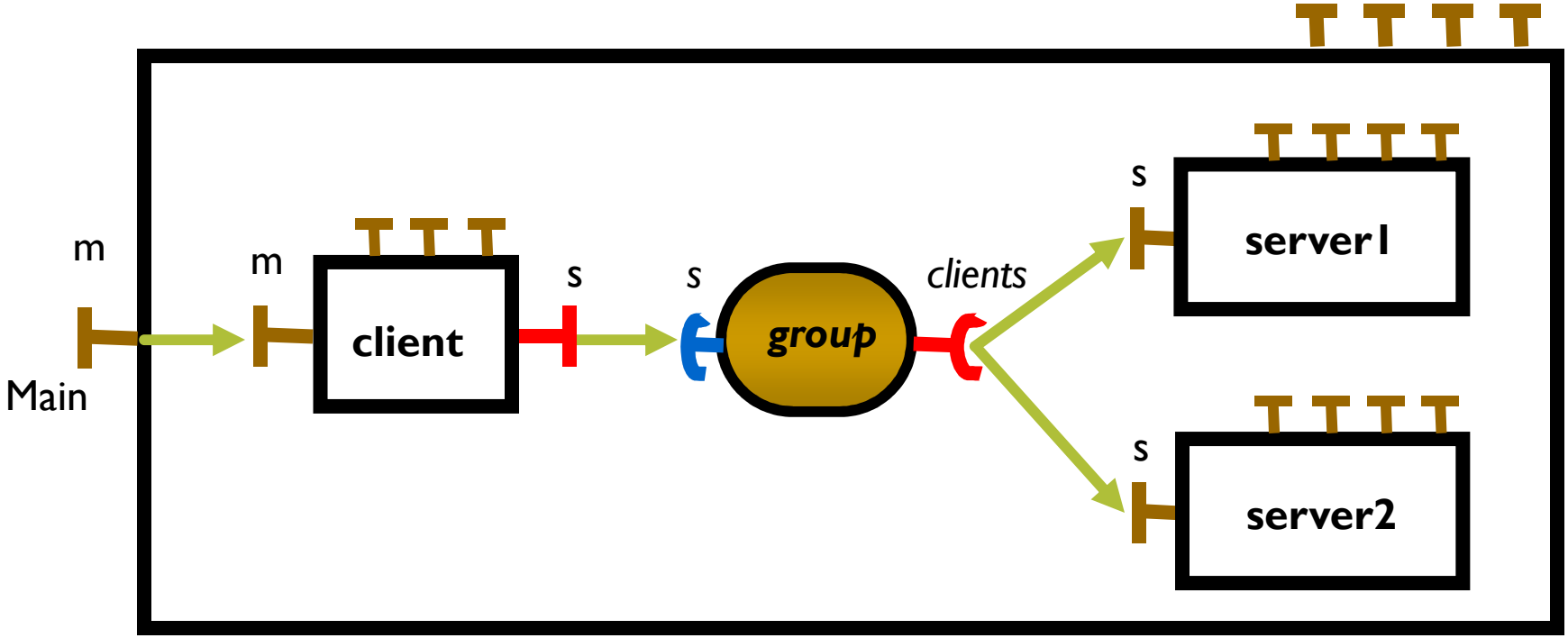


Components Groups

- A group of components with the same (at least one) interfaces
- Group bindings
  - One-to-All
    - All members are called
  - One-to-Any
    - A randomly selected member is invoked

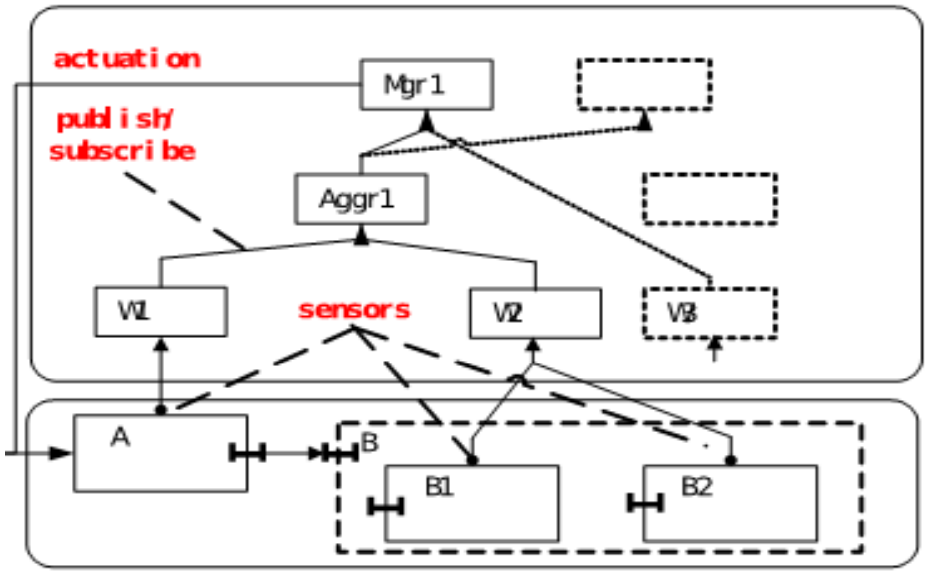


A Client-Server Application using Groups



A Niche Autonomic Manager

- Autonomic Manager = network of Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate using events
  - pub/sub mechanism
- Hard-coded in Java
- Niche Autonomic Manager uses sensors and actuators
  - system and/or user-defined
  - bound to functional components

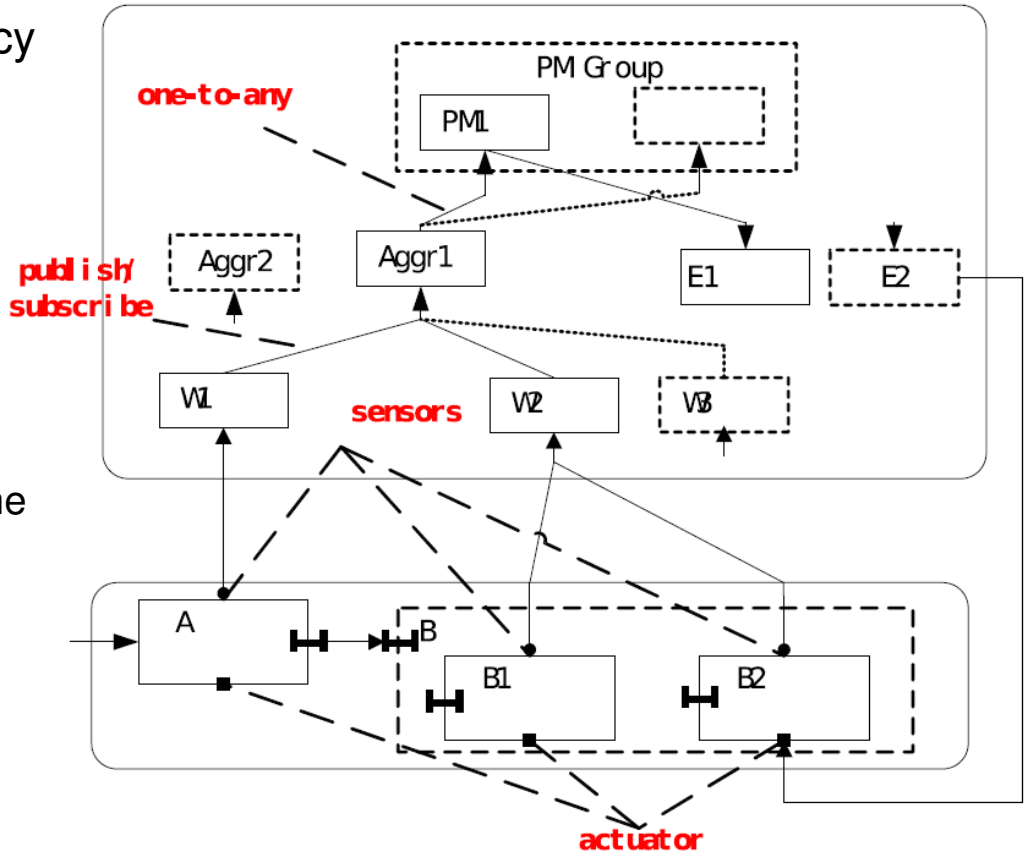


Policy-based Management

- Self-management under guidelines defined by humans in the form of management policies
- Management policy
  - A set of rules that govern the system behaviors
    - Rule combining algorithms
  - Reflects the business goals and/or management objectives

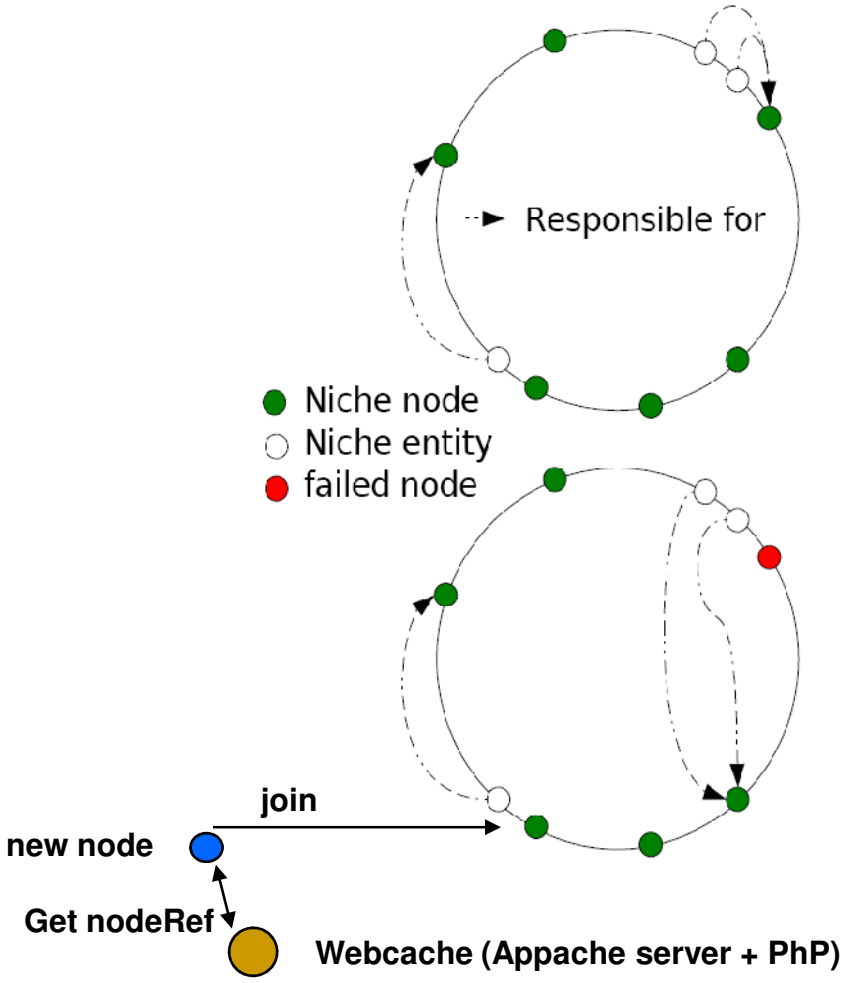
Niche Policy Based Management Framework

- A policy language, API, and a policy engine
  - to describe, evaluate and enforce policies
  - SPL, XACML
- Policies, Policy Reposiroty
- Policy Manager
  - loads policies, makes decisions based on policies and delegates obligations to Executors
  - PM Group = a group of PMs with the same set of policies
- Policy Watchers
  - monitors the policy repositories for policy changes
- Policy Engine
  - evaluates policies and returns obligations
- **Work in progress**

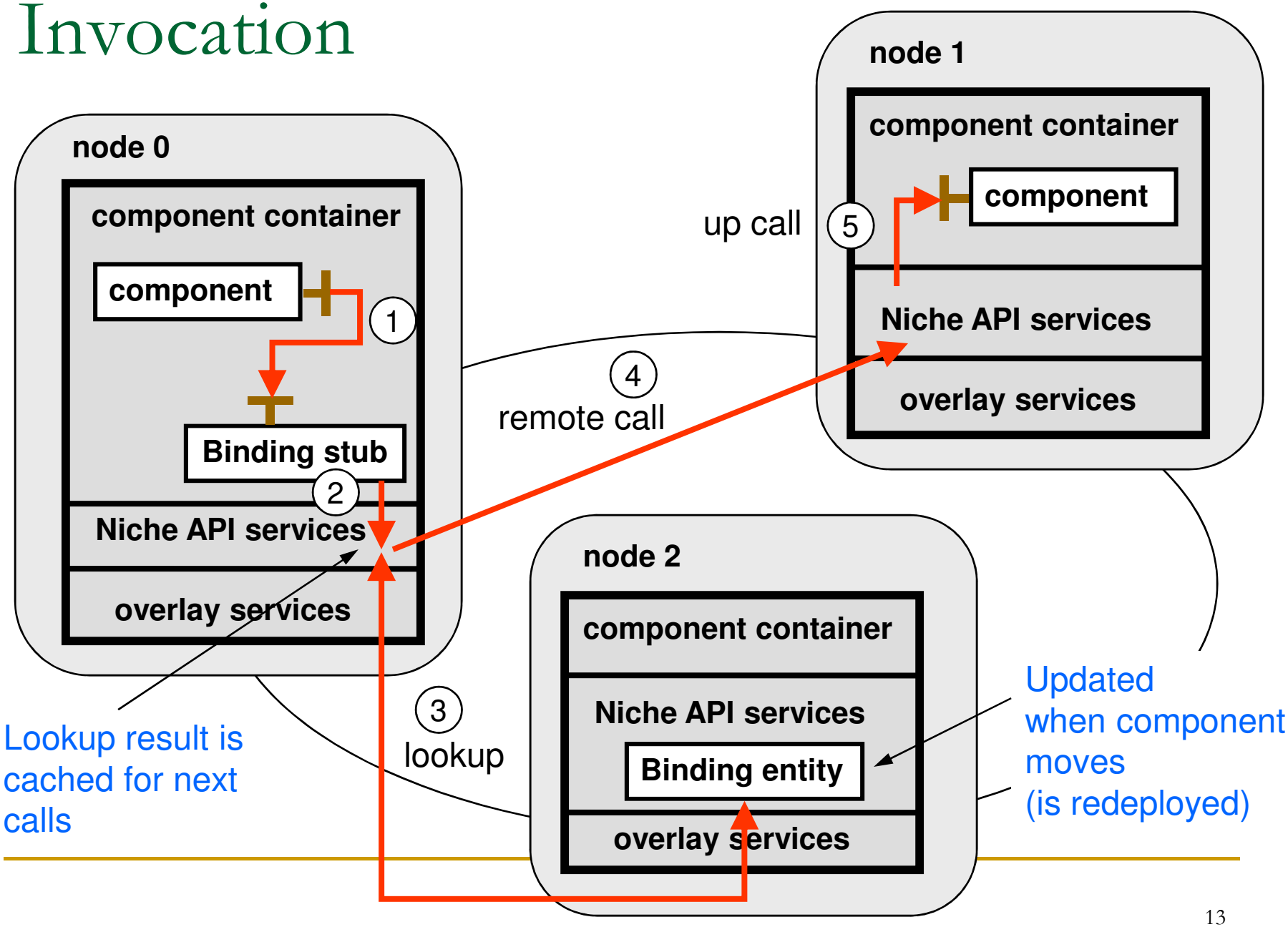


Niche Execution Environment

- A set of containers connected through the Niche/DKS structured P2P overlay network
- Container configurations
  - JadeBoot
    - bootstraps the system
    - interprets ADL (*.fractal) files on deployment
  - JadeNode
- DHT-based registry of components, groups, bindings, etc.
- Broadcast resource discovery

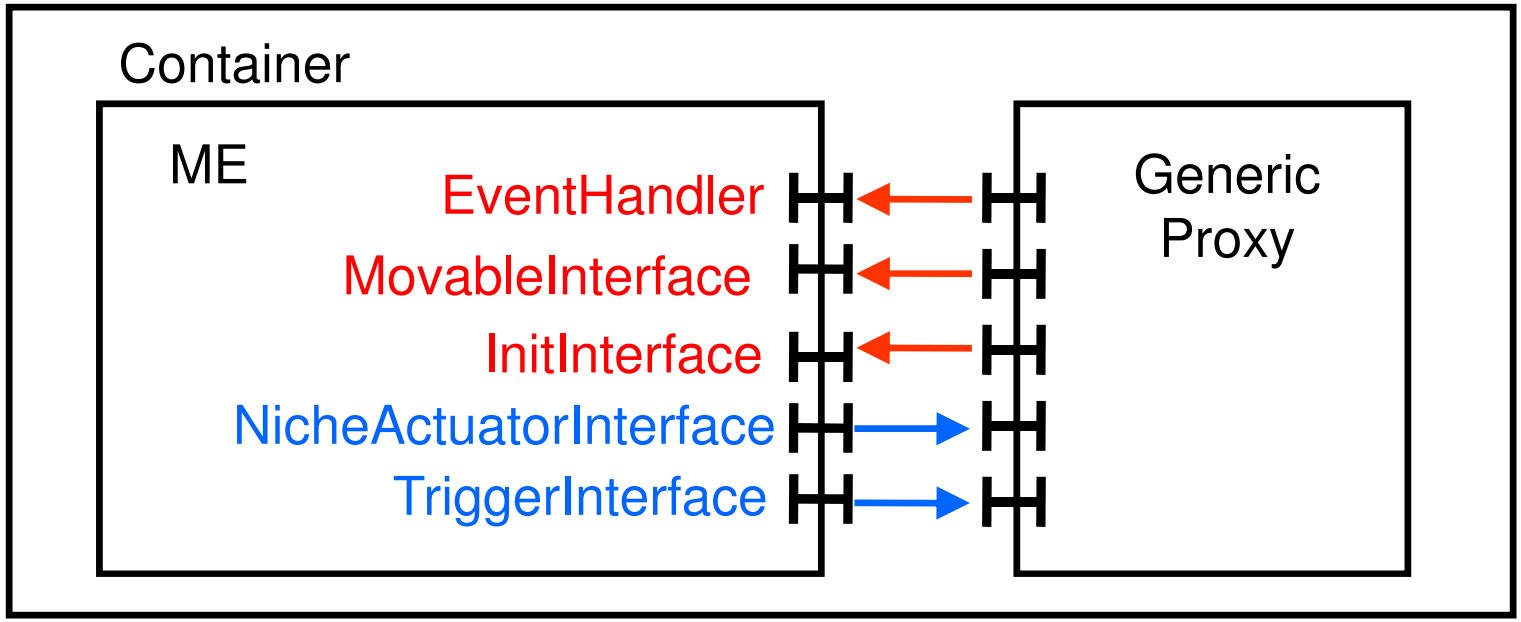


Invocation


node 0
component container
component
1
Binding stub
2
Niche API services
overlay services
node 1
component container
component
up call
5
Niche API services
overlay services
4
remote call
node 2
component container
Niche API services
Binding entity
overlay services
3
lookup
Lookup result is cached for next calls
Updated when component moves (is redeployed)

Management Element Interfaces

- Management Element is bound to a generic proxy that provides connectivity between MEs and access to overlay services



Overlay Services (1/2)

- Resource Discovery
  - Currently based on broadcast over SON
    - Not scalable, however…
  - there are approaches to make broadcast-based discovery more efficient and scalable
    - E.g. incremental controlled broadcast
- Initial deployment and dynamic (runtime) reconfiguration
  - Allocate/deallocate, deploy, (un)bind

Overlay Services (2/2)

- Publish / (un)subscribe
  - Predefined and programmer-defined events
  - Event delivery: push method using EventHandlers (listeners)
  - A source maintains a list of subscribers (in its proxy)
    - The proxy sends events to subscribers when events are issued (triggered)
- DHT-based lookup service
  - To lookup components, groups, bindings
  - SNR: Set of Network References: data structure to hold references (helps to enable transparent component migration)

Niche as a Development Environment

- Java, ADL, BeanShell scripts (to launch deployment and start)
- Separates programming of functional and management parts
- Functional part: components, component groups, bindings (singletons & groups)
  - Identified by names
    - Bindings & references to a component unchanged on component migration.
  - Can be found & controlled (Fractal)
    - Implementation: overlay – lookups with caching
- Management part: management elements (watchers, aggregators, managers)
  - Communicate using events
  - Sensing and actuation API

Two APIs for Developers

- Jade

  includes implementation of the Fractal model and an interface to the Niche execution environment

  - `org.objectweb.fractal.api`
  - `org.objectweb.jasmine.jade`
  - `...`
  - `org.objectweb.jasmine.jade.service.nicheOS`

- Niche/DKS

  includes classes and interfaces for events, actuation, groups, etc.

  - `dks.niche.events`
  - `dks.niche.fractal`
  - `dks.niche.ids`
  - `dks.niche.interfaces`
  - `dks.niche.wrappers`
  - `dks.niche.sensors`

Programming Concepts (functional part)

- Component = membrane + content (sub-components)
  - Encapsulated data and behavior
  - With well identified interfaces
  - With sub-components
- Interface
  - A named access point to a component
  - Client and server interfaces
  - Typed
- Binding
  - Communication path between components
  - Client to server interface binding
- Membrane
  - Includes control interfaces
  - Supports a component's reflective capabilities
- Component group
  - A group of components with the same (at least one) interfaces
  - One-to-all, one-to-any binding

Programming Concepts (mgnt part)

- Sensor
  - monitors a component or a group through bindings, and triggers events to convey monitored information to watchers
- Watcher
  - collects information from sensors and communicates it to Aggregators
- Aggregator
  - aggregates information from watchers, detect and report symptoms to Managers
- Manager
  - analyzes the symptoms, make decisions and request Executors to act
- Executor
  - receives commands from Managers and issues commands to Actuators
- Actuator
  - receive commands from Executors and act on components through bindings

- Management components (W, Agg, Mgr, Extr) interact with each other via events (pub/sub)
- Sensors and actuators interact with functional components and groups through bindings

Events

- Predefined events, e.g.
  - ComponentFailEvent, CreateGroupEvent, MemberAddedEvent, ResourceJoinEvent, ResourceLeaveEvent, ResourceStateChangeEvent
- Programmer-defined events
- Niche guarantees delivery
- Trigger events using the TriggerInterface client interface
- Event triggering (publishing) options
  - To any subscriber (randomly selected)
  - To all subscribers (default)
  - With a specified tag (topic)
    - Subscribe to events with specified tags

Development Steps

1. Design architecture of the functional and management parts
   - Functional components (including server and client interfaces) and component groups
   - Names for all interfaces
   - Bindings (singletons and groups)
   - Management components (including event handlers)
   - Events, subscriptions
2. Describe initial architecture of functional and management parts in ADL
   - Components (interfaces), bindings
   - Groups and group bindings are not yet supported in ADL

Development Steps (cont'd)

3. Program functional and management components
   - Define classes and interfaces;
   - Implement server interfaces (functional), event handlers (management), Fractal and Niche control interfaces
4. Program a (startup) component, which completes initial deployment, e.g.
   - looks up and binds components created using ADL files;
   - creates groups and group bindings;
   - configures and deploys management components;
   - subscribes management components to events

Deploy and Run

- The current prototype uses BeanShell scripts to deploy and to start an application
- A BeanShell deploy script example
  - NicheHelloWorld-Deploy.bsh

```
source("init");
print("Start NicheHelloWorld deployment");
deploy("NicheHelloWorld");
print("End NicheHelloWorld deployment");
```

- A BeanShell start script example
  - NicheHelloWorld-Start.bsh

```
source("init");
print("Starting NicheHelloWorld");
c = lookupcomp("NicheHelloWorld_0"); // base
lc=getitf(c, "lifecycle-controller");
lc.startFc(); // start the application
print("NicheHelloWorld started");
```

Start of an Application

```
c = lookupcomp("NicheHelloWorld_0"); // base
lc=getitf(c, "lifecycle-controller");
lc.startFc(); // start the application
```

- Lookup the composite component that represents the application and contains all components deployed from the ADL description
  - The composite component is named according to the naming convention
- Get its life-cycle interface and call `startFC`
- This causes start of all inner components by calling `startFC` on their life-cycle interfaces
  - No order of starts should be assumed

Example: Service Component

```
public class ServiceComponent implements
  HelloAnyInterface, HelloAllInterface,
  BindingController, LifeCycleController {
```

- Component server interfaces
  - HelloAnyInterface, HelloAllInterface
- Fractal control interfaces
  - BindingController, LifeCycleController

Fractal Control Interfaces

- Server interfaces
  - Should be implemented by any component (functional or management)
- Defined in org.objectweb.fractal.api.control
  - `LifeCycleController`
    - Start, stop, get state of the component
  - `BindingController`
    - Bind, unbind, lookup, list names of client interfaces
  - `AttributeController`
    - Getters, setters
  - `ContentController`
    - Get internal interfaces (by names), get sub-components (by names), add/remove sub-components

BindingController Interface

- **`String[] listFc ()`**
  - Returns names of client interfaces of this component.
- **`Object lookupFc (String clientItfName)`**
  - Returns a server interface bound to a client interface with the given name.
- **`void bindFc (String clientItfName, Object serverItf)`**
  - Binds a client interface with the given name to the given server interface.
- void unbindFc (String clientItfName)
  - unbinds a client interface with the given name.

Implementation of BindingController Example

```
private boolean status;
private String[] clientInterfaceNames = { "component", "helloAny", "helloAll" };
private Map<String, Object> interfaces = new HashMap<String, Object>();

public FrontendComponent() { // constructor
    for (String s: clientInterfaceNames) interfaces.put(s, null);
}
public String[] listFc() {
    return clientInterfaceNames;
}
public Object lookupFc(final String itfName) throws NoSuchInterfaceException {
    if (!interfaces.containsKey(itfName)) throw new NoSuchInterfaceException(itfName);
    return interfaces.get(itfName);
}
public void bindFc(final String itfName, final Object itfValue) throws
    NoSuchInterfaceException {
    if (!interfaces.containsKey(itfName)) throw new NoSuchInterfaceException(itfName);
    interfaces.put(itfName, itfValue);
}
public void unbindFc(final String itfName) throws NoSuchInterfaceException {
    if (!interfaces.containsKey(itfName)) throw new NoSuchInterfaceException(itfName);
    interfaces.put(itfName, null);
}
// use a client interface
 public synchronized void helloAll() {
    ((HelloAllInterface)interfaces.get("helloAll")).helloAll("HelloWorld");
 }
```

LifeCycleController Interface

- `String getFcState ()`
  - Returns the execution state of this component
- `void startFc ()`
  - Starts this component
- `void stopFc ()`
  - Stops this component.
    - The result of a method call on a stopped component is undefined, except on its control interfaces (these calls are executed normally)

Implementation of LifeCycleController Example

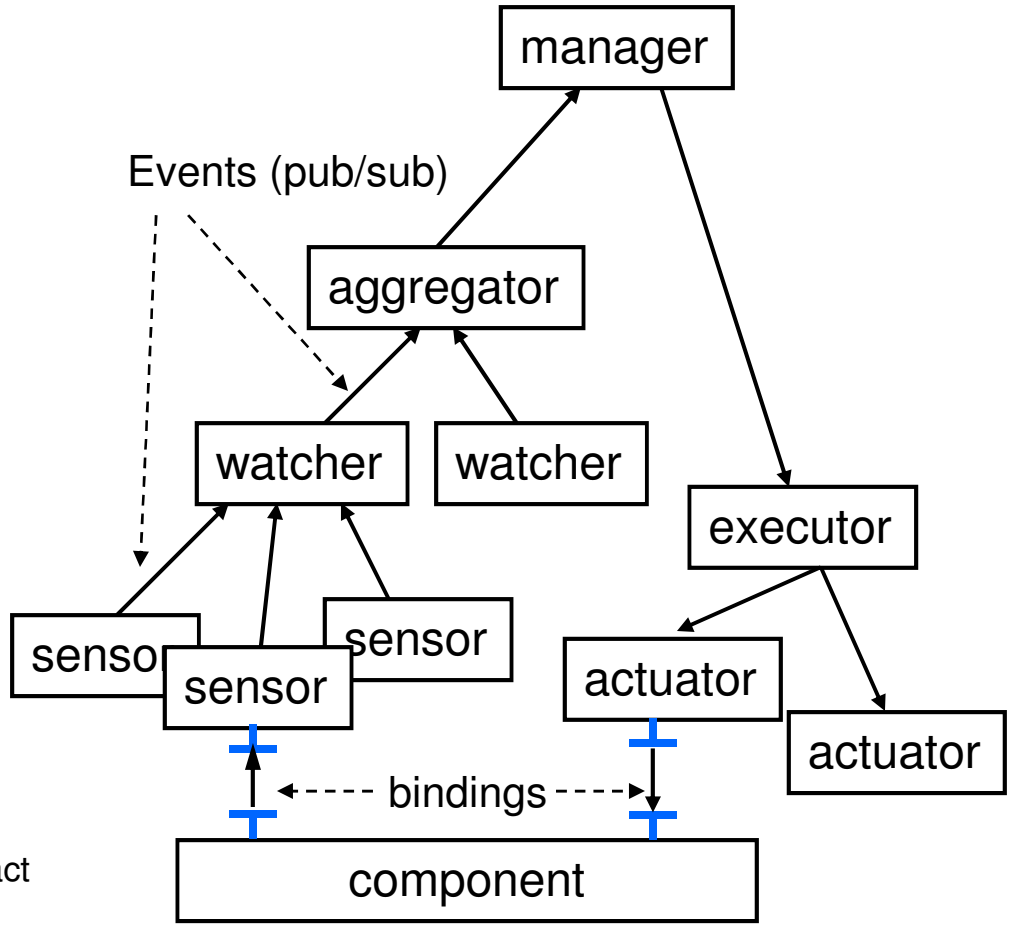
```
public String getFcState() {
        return status ? "STARTED" : "STOPPED";
}

public void startFc() throws IllegalLifeCycleException {
        // Create the GUI.
        new UserInterface(this);
        status = true;
        System.err.println("Frontend component started.");
}

public void stopFc() throws IllegalLifeCycleException {
        status = false;
}
```

Client Interfaces of a Start Manager

- A start manager component completes deployment
  - Serves as an ”extension” to ADL-based deployment
    - To implement unsupported deployment actions, e.g. create groups, subscribe to events, etc.
- **`NicheIdRegistry`**
  - to lookup components (SNRs)
  - Name *ID_REGISTRY* = "nicheIdRegistry”
- **`OverlayAccess`**
  - to get access to overlay
    - Used to get different ”supports” (services), e.g. NicheActuatorInterface
  - Name *OVERLAY_ACCESS* = "overlayAccess”

Management Elements

- Communicate with events (pub/sub)
- Sensors
  - monitor components through interfaces
  - trigger events
  - Predefined sensors
- Watchers (W)
  - receive information from sensors and communicate it to Aggregators
- Aggregators (Aggr)
  - aggregate the information, detect and report symptoms to Managers
- Managers (Mgr)
  - analyze the symptoms, make decisions and request Executors to act
- Executors
  - receive commands from managers and issues commands to actuators
- Actuators
  - receive commands from Executors and act on components through interfaces



Sensors

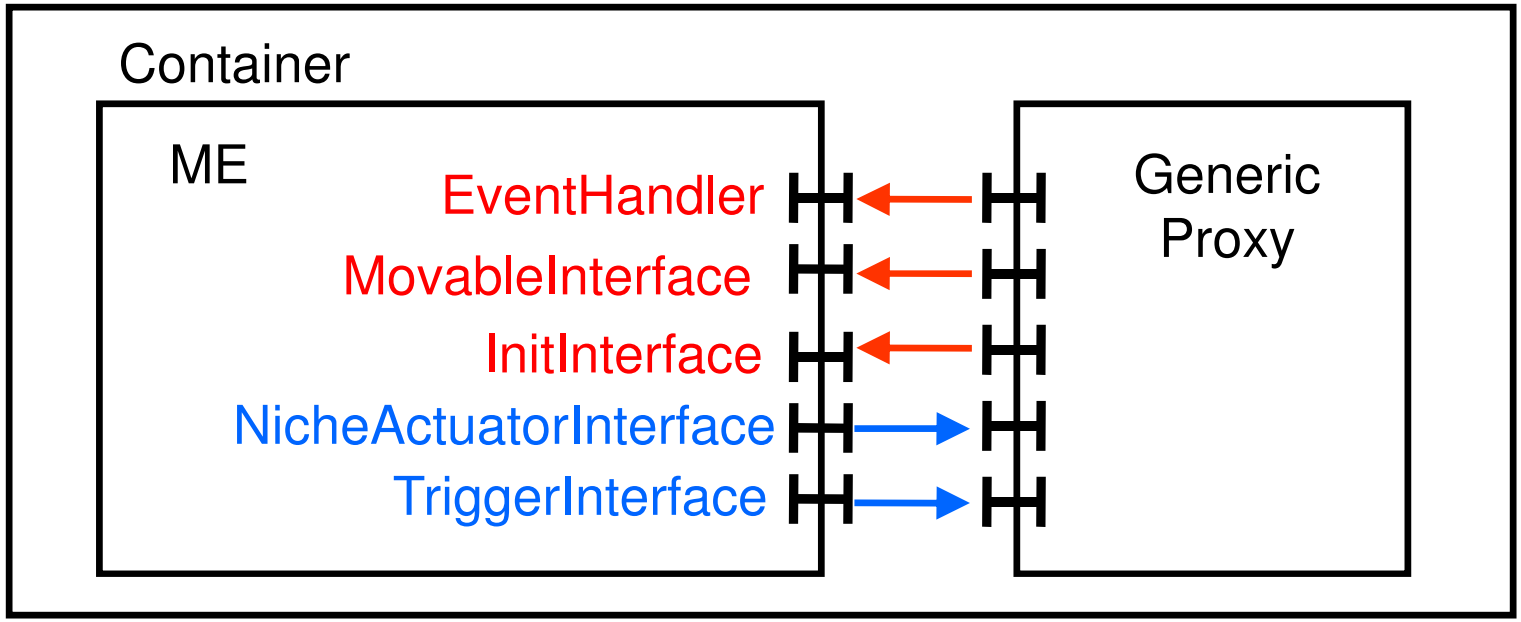
- Interfaces between sensors and components are defined by a programmer (push or/and pull)
  - Push: a component pushes a sensor to issue an event
    - Component's client interface is bound to the sensor's server interface
  - Pull: a sensor pulls a component to get its state
    - Sensor's client interface is bound to the component's server interface
  - A sensor and a component are auto-bound when the sensor is deployed (by a watcher)

Actuation

- Using either actuators (bound to components) or the Actuation API
- Actuators are programmed in a similar way as sensors
  - Deployed by executors
  - Push: an actuator pushes a component through control interfaces
    - Programmer defined interfaces
    - Fractal control interfaces, e.g. **`LifeCycleController`** and **`AttributeController`**
  - Pull: a component checks its actuator for actions to be executed

Management Element Interfaces

- Management Element is bound to a generic proxy that provides connectivity between MEs and access to overlay services



Server Interfaces of MEs

- Should be implemented by management components
- Defined in **`dks.niche.fractal.interfaces`**
  - **`EventHandlerInterface`**
    - To receive events (according to subscription)
  - **`MovableInterface`**
    - To get checkpoint, when moved and redeployed (for replication or migration)
    - The checkpoint is passed to a new instance through its InitInterface
  - **`InitInterface`**
    - To initialize a management component

Client Interfaces of MEs. Actuation and Event API

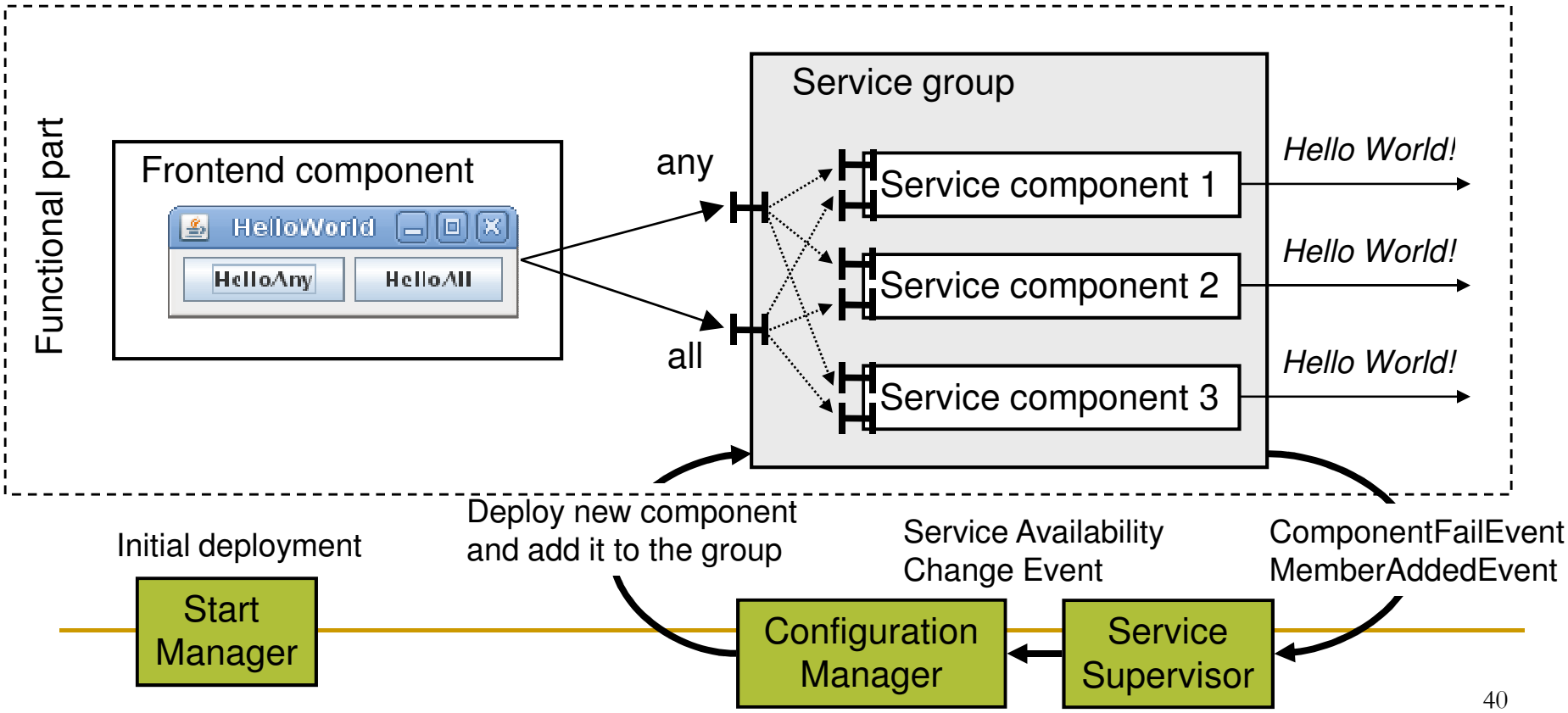
- **`NicheActuatorInterface`** extends **`NicheComponentSupportInterface`**
  - **The Actuation API**: discover, allocate, deallocate, deploy, bind, unbind, subscribe, unsubscribe
    - named *ACTUATOR_CLIENT_INTERFACE* = “actuator”
    - bound on deployment
    - implemented by dks.niche.fractal.ManagementElement
- **`TriggerInterface`**
  - **The Publish API**: trigger events
    - named *TRIGGER_CLIENT_INTERFACE* = "trigger“
    - Bound on deployment
    - implemented by dks.niche.fractal.ManagementElement

Typical Life Cycle of a Component

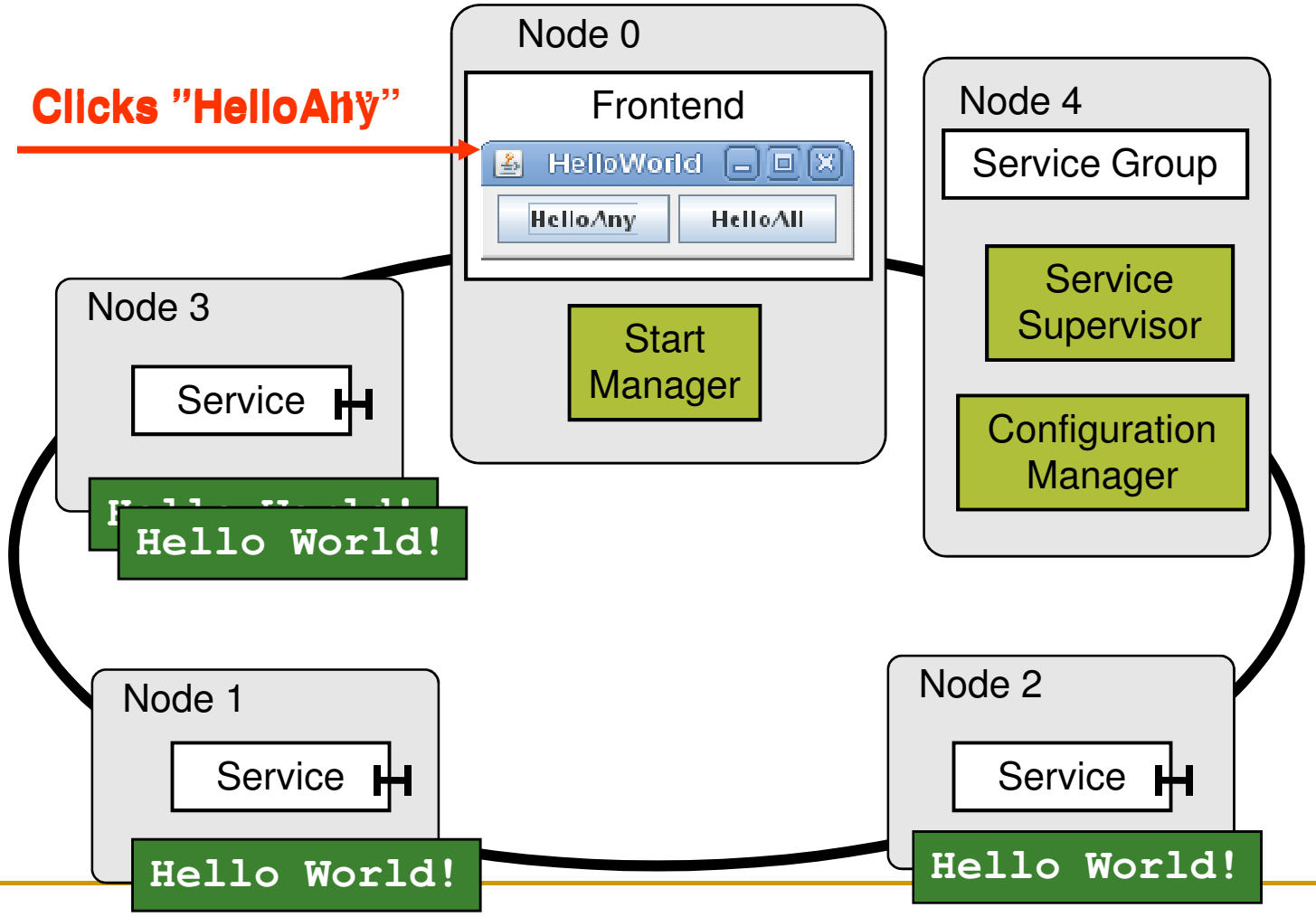
- Instantiated
- Initialized
- Client interfaces are bound
- Started

Hello World Example

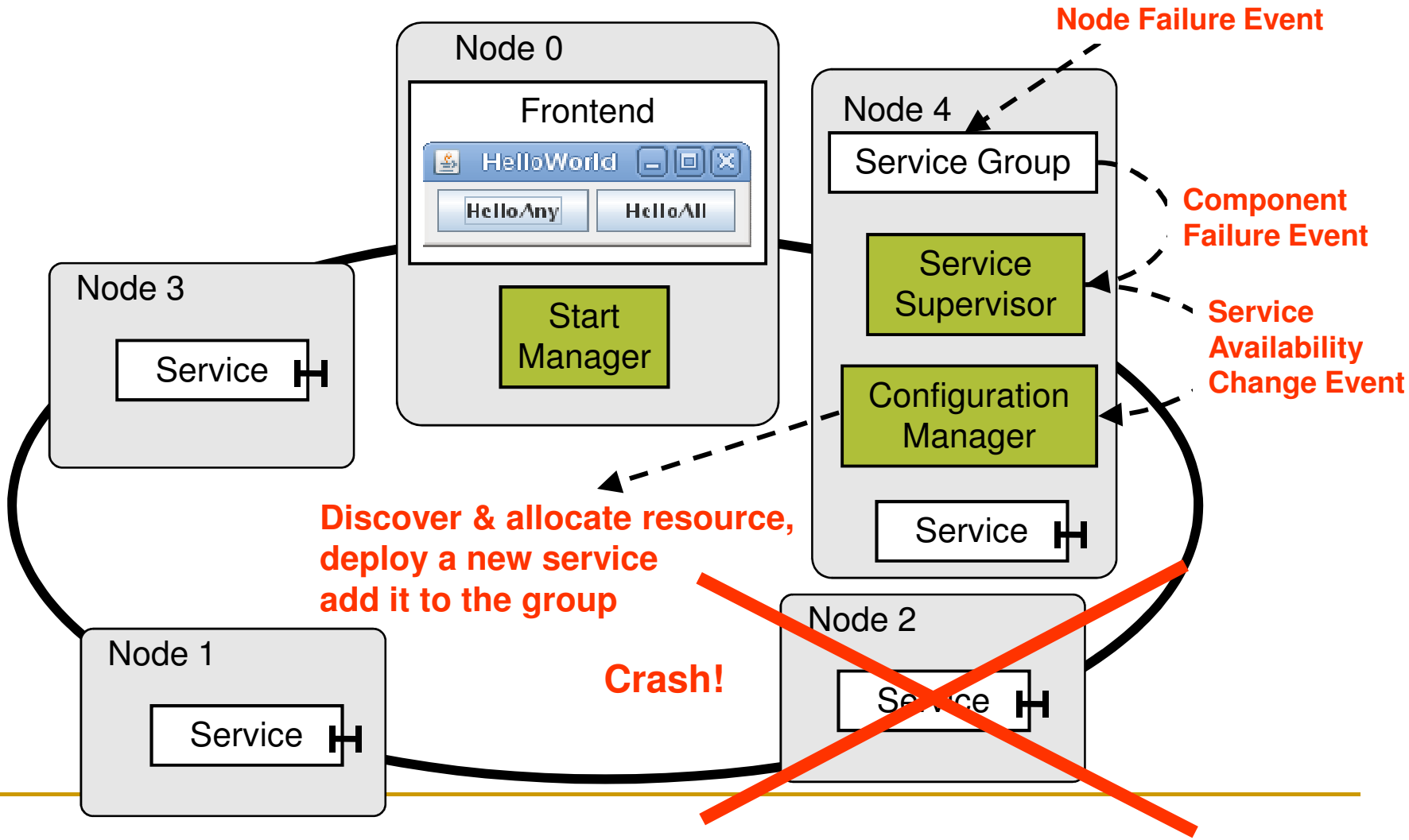
- Self-managing service with self-healing control loop
  - Prints the greeting ”Hello World” (by all or any service component)
  - Maintains specified minimum number of service components



Deployment and Use



Self-Healing Control Loop



ADL File for Initial Deployment

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/jasmine/jade/service/deployer/adl/xml/jadeApplication.dtd">
<definition name="NicheHelloWorld">
  <component name="StartManager" definition="org.objectweb.jasmine.jade.ManagementType">
    <content class="helloworld.managers.StartManager"/>
    <controller desc="primitive"/>
  </component>
  <component name="frontend">
    <interface name="helloAny" role="client" signature="helloworld.interfaces.HelloAnyInterface"
    contingency="optional" />
    <interface name="helloAll" role="client" signature="helloworld.interfaces.HelloAllInterface"
    contingency="optional"/>
    <content class="helloworld.frontend.FrontendComponent"/>
    <virtual-node name="lightweight1" resourceReqs="10"/>
  </component>
  <component name="service1">
    <interface name="helloAny" role="server" signature="helloworld.interfaces.HelloAnyInterface"
    contingency="optional" />
    <interface name="helloAll" role="server" signature="helloworld.interfaces.HelloAllInterface"
    contingency="optional"/>
    <content class="helloworld.service.ServiceComponent"/>
    <virtual-node name="medium1" resourceReqs="950000"/>
  </component>
  ...
</definition>
```

Defenition of the StartManager Type

- Defines client interfaces of the start manager

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
    "classpath://org/objectweb/jasmine/jade/service/deployer/adl/xml/jadeApplication.dtd">

<definition name="org.objectweb.jasmine.jade.ManagementType">
  <interface name="overlayAccess" role="client"
    signature="org.objectweb.jasmine.jade.service.nicheOS.OverlayAccess"
    contingency="optional"/>
  <interface name="nicheIdRegistry" role="client"
  signature="org.objectweb.jasmine.jade.service.componentdeployment.NicheIdRegistry"
  contingency="optional"/>
</definition>
```

Startup Manager

- Creates a group of service components
  - Looks up service components created using the HelloWorld ADL definition
  - Adds the components to the group
- Looks up and Binds the frontend component to the group
- Configures and deploys the ServiceSupervisor aggregator
  - Subscribes it to ComponentFailEvent, MemberAddedEvent events from the serviceGroup
- Configures and deploys the ConfigurationManager manager.
  - Subscribes it to ServiceAvailabilityChangeEventevent events from the serviceSupervisor aggregator

Creating groups in **StartManager** (1/2)

```
private OverlayAccess nicheService;// bound on deployment
private NicheIdRegistry nicheIdRegistry;// bound on deployment
// Get a reference to the Niche API.
NicheActuatorInterface myActuatorInterface =
   nicheService.getOverlay().getJadeSupport();
// Find the front-end component.
ComponentId frontendComponent = (ComponentId) nicheIdRegistry.lookup(
                        APPLICATION_PREFIX + FRONTEND_COMPONENT);
// Find all service components.
ArrayList<ComponentId> serviceComponents = new ArrayList();
int serviceComponentIndex = 1;
ComponentId serviceComponent = (ComponentId) nicheIdRegistry.lookup(
           APPLICATION_PREFIX + SERVICE_COMPONENT + serviceComponentIndex);
while (serviceComponent != null) {
       serviceComponents.add(serviceComponent);
       serviceComponentIndex++;
       serviceComponent = (ComponentId) nicheIdRegistry.lookup(
            APPLICATION_PREFIX + SERVICE_COMPONENT + serviceComponentIndex);
}
```

Creating groups in **StartManager** (2/2)

```
// Create a component group containing all service components.
GroupId serviceGroupTemplate = myActuatorInterface.getGroupTemplate();
serviceGroupTemplate.addServerBinding("helloAny",
                                      JadeBindInterface.ONE_TO_ANY);
serviceGroupTemplate.addServerBinding("helloAll",
                                      JadeBindInterface.ONE_TO_MANY);
GroupId serviceGroup = myActuatorInterface.createGroup(
                    serviceGroupTemplate, serviceComponents);
// Create a one-to-any binding from the front-end to
// the service group. This binding uses the helloAny interface.
String clientInterfaceName = "helloAny";
String serverInterfaceName = "helloAny";
myActuatorInterface.bind(frontendComponent, clientInterfaceName, serviceGroup,
                         serverInterfaceName, JadeBindInterface.ONE_TO_ANY);
// Create a one-to-all binding from the front-end to
// the service group. This binding uses the helloAll interface.
clientInterfaceName = "helloAll";
serverInterfaceName = "helloAll";
myActuatorInterface.bind(frontendComponent, clientInterfaceName, serviceGroup,
                         serverInterfaceName, JadeBindInterface.ONE_TO_MANY);
```

Deploying a Management Element in **StartManager**

```
// Configure the ServiceSupervisor aggregator.
ManagementDeployParameters params = new ManagementDeployParameters();
params.describeAggregator(ServiceSupervisor.class.getName(),
                          "SA", null,
                          new Serializable[] {serviceGroup.getId()});
params.setReliable(true);

// Deploy the ServiceSupervisor aggregator.
NicheId serviceSupervisor =
 myActuatorInterface.deployManagementElement(
                                              params, serviceGroup);

// Make the ServiceSupervisor aggregator subscribe to events.
myActuatorInterface.subscribe(serviceGroup, serviceSupervisor,
                              ComponentFailEvent.class.getName());
myActuatorInterface.subscribe(serviceGroup, serviceSupervisor,
                              MemberAddedEvent.class.getName());
```

Frontend Component

- Creates GUI
- Calls Service Component Group through one-to-all (HelloAllInterface) or one-to-any (HelloAnyInterface)

Implementation of **BindingController** in **FrontendComponent**

```
public String[] listFc() {
        return new String[] { "component", "helloAny", "helloAll" };
}
public void bindFc(final String itfName, final Object itfValue)
                    throws NoSuchInterfaceException {
        if (itfName.equals("helloAny")) {
            helloAny = (HelloAnyInterface) itfValue;
        } else if (itfName.equals("helloAll")) {
            helloAll = (HelloAllInterface) itfValue;
        } else if (itfName.equals("component")) {
            myself = (Component) itfValue;
        } else {
            throw new NoSuchInterfaceException(itfName);
        }
    }
public void unbindFc(final String itfName) throws NoSuchInterfaceException {
        if (itfName.equals("helloAny")) {
            helloAny = null;
        }
        ...
    }
public Object lookupFc(final String itfName) throws NoSuchInterfaceException {
        if (itfName.equals("helloAny")) {
            return helloAny;
        }
        ...
    }
```

Implementation of **LifeCycleController** in **FrontEndComponent**

```
public String getFcState() {
        return status ? "STARTED" : "STOPPED";
    }

    public void startFc() throws IllegalLifeCycleException {
        // Create the GUI.
        new UserInterface(this);
        status = true;
        System.err.println("Frontend component started.");
    }

    public void stopFc() throws IllegalLifeCycleException {
        status = false;
    }
```

Implementation and Use of the Service

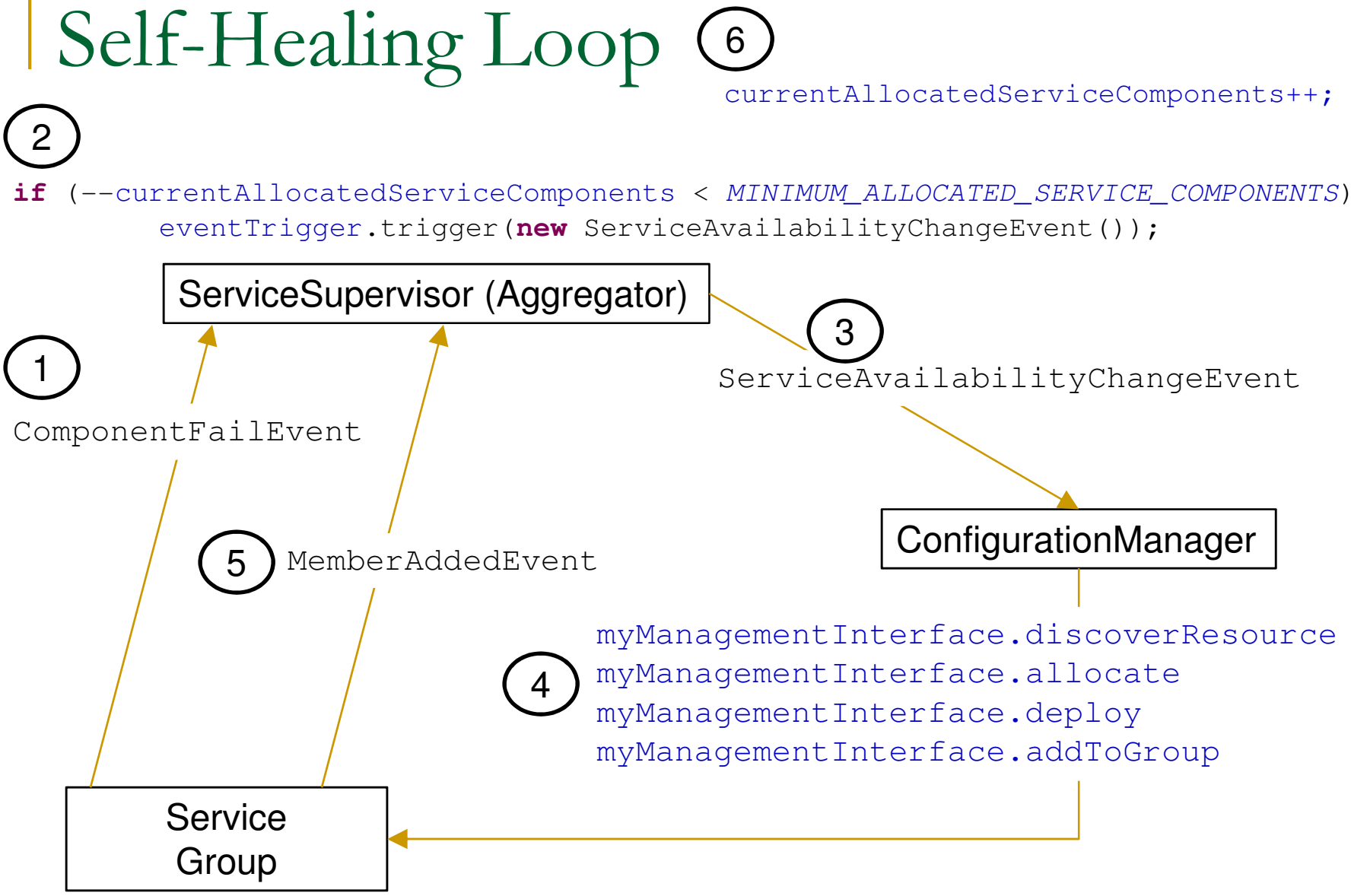
- Server interface implementations in **`ServiceComponent`**

```
public void helloAny(String s) {
        System.out.println(s);
}
public void helloAll(String s) {
        System.out.println(s);
}
```

- Calls to **`ServiceComponent`** in **`FrontendComponent`**

```
// Client Interfaces
private HelloAnyInterface helloAny;
private HelloAllInterface helloAll;
...
public synchronized void helloAny() {
        helloAny.helloAny("HelloWorld");
}
public synchronized void helloAll() {
        helloAll.helloAll("HelloWorld");
}
```

Self-Healing Loop



Self-Healing Loop (1/4)

- Event Handler in **`ServiceSupervisor`**

```
private void handleComponentFailEvent(ComponentFailEvent failedEvent) {
   String idAsString = failedEvent.getFailedComponentId().getId().toString();
   if (!currentComponents.containsKey(idAsString)) {
      // The failed component is not in our list of active service components.
        return;
   }
   // Remove failed component from list of active components.
   currentComponents.remove(idAsString);
   currentAllocatedServiceComponents--;
   ...
  if (currentAllocatedServiceComponents < MINIMUM_ALLOCATED_SERVICE_COMPONENTS) {
       ...
      // Tell ConfigurationManager there are too few service components.
      eventTrigger.trigger(new ServiceAvailabilityChangeEvent());
      ...
  }
  ...
}
```

Self-Healing Loop (2/4)

- Event Handler in **`ConfigurationManager`**

```
// Reference to the Niche Actuation interace.
private NicheActuatorInterface myManagementInterface;
public void eventHandler(Serializable e, int flag) {
  // Find a node that meets the requirements for a service component.
  NodeRef newNode = null;
  try {
    newNode = myManagementInterface.oneShotDiscoverResource( nodeRequirements);
  } catch (OperationTimedOutException err) {
     ... // retry later (the code is removed)
  }
  if (newNode == null) {
    System.out.println("ConfigurationManager could not get resource."); return;
  }
  // Allocate resources for a service component at the found node.
  List allocatedResources = null;
  try {
       allocatedResources = myManagementInterface.allocate(newNode, null);
  } catch (OperationTimedOutException err) {
       ... // retry later (the code is removed)
  }
  ResourceRef allocatedResource = (ResourceRef) allocatedResources.get(0)
```

Self-Healing Loop (3/4)

- Event Handler in **`ConfigurationManager`** (cont'd)

```
// Deploy a new service component instance at the allocated node.
String deploymentParams = null;
try {
    deploymentParams = Serialization.serialize(serviceCompProps);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
List deployedComponents = null;
try {
   deployedComponents = myManagementInterface.deploy(allocatedResource,
                                                      deploymentParams);
} catch (OperationTimedOutException err) {
   ... // Retry later (the code is removed)
}
ComponentId cid = (ComponentId) ((Object[]) deployedComponents.get(0))[1];
// Add the new component to the service component group and start it.
myManagementInterface.update(componentGroup, cid,
   NicheComponentSupportInterface.ADD_TO_GROUP_AND_START);
}
```

Self-Healing Loop (4/4)

- Event Handler in **`ServiceSupervisor`**

```
private void handleMemberAddedEvent( MemberAddedEvent newMemberEvent)
{
      String idAsString = newMemberEvent.getSNR().getId().toString();

      if (currentComponents.containsKey(idAsString)) {
          // We already know about this component.
          return;
      }
      // Add the component to our list
      currentComponents.put(idAsString, true);
      currentAllocatedServiceComponents++;
 }
```

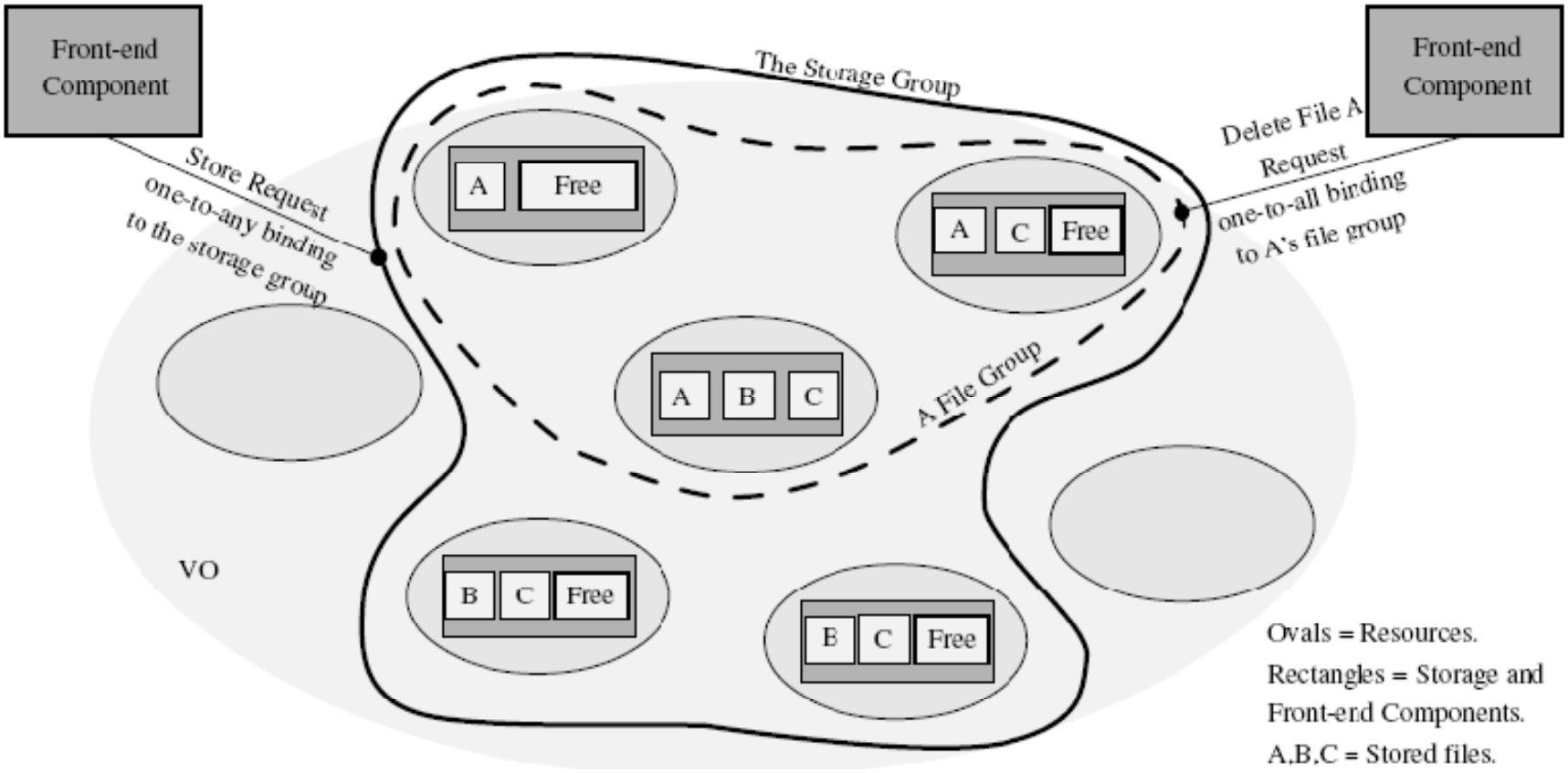
Self-Managing Services Using Niche

- YASS: Yet Another Storage Service
- YACS: Yet Another Computing Service
- Each of the services
  - Can be deployed and provided on computers donated by users or by a service provider.
  - Can operate even if computers join, leave or fail at any time.
  - Has self-healing and self-configuration capabilities and can execute on a dynamic environment.
  - Implements relatively simple self-management algorithms, which can be replaced by more sophisticated, while reusing existing monitoring and actuation code of the services.
  - Self-managing capabilities of services allows the users to minimize the human resources required for the service management.

YASS: Yet Another Storage Service

- A robust self-managing file storage
  - Users can store, read and delete files on a set of distributed resources (storage components)
  - Transparently replicates files for robustness and scalability
  - Can be deployed in a dynamic distributed environment

YASS Functional Part

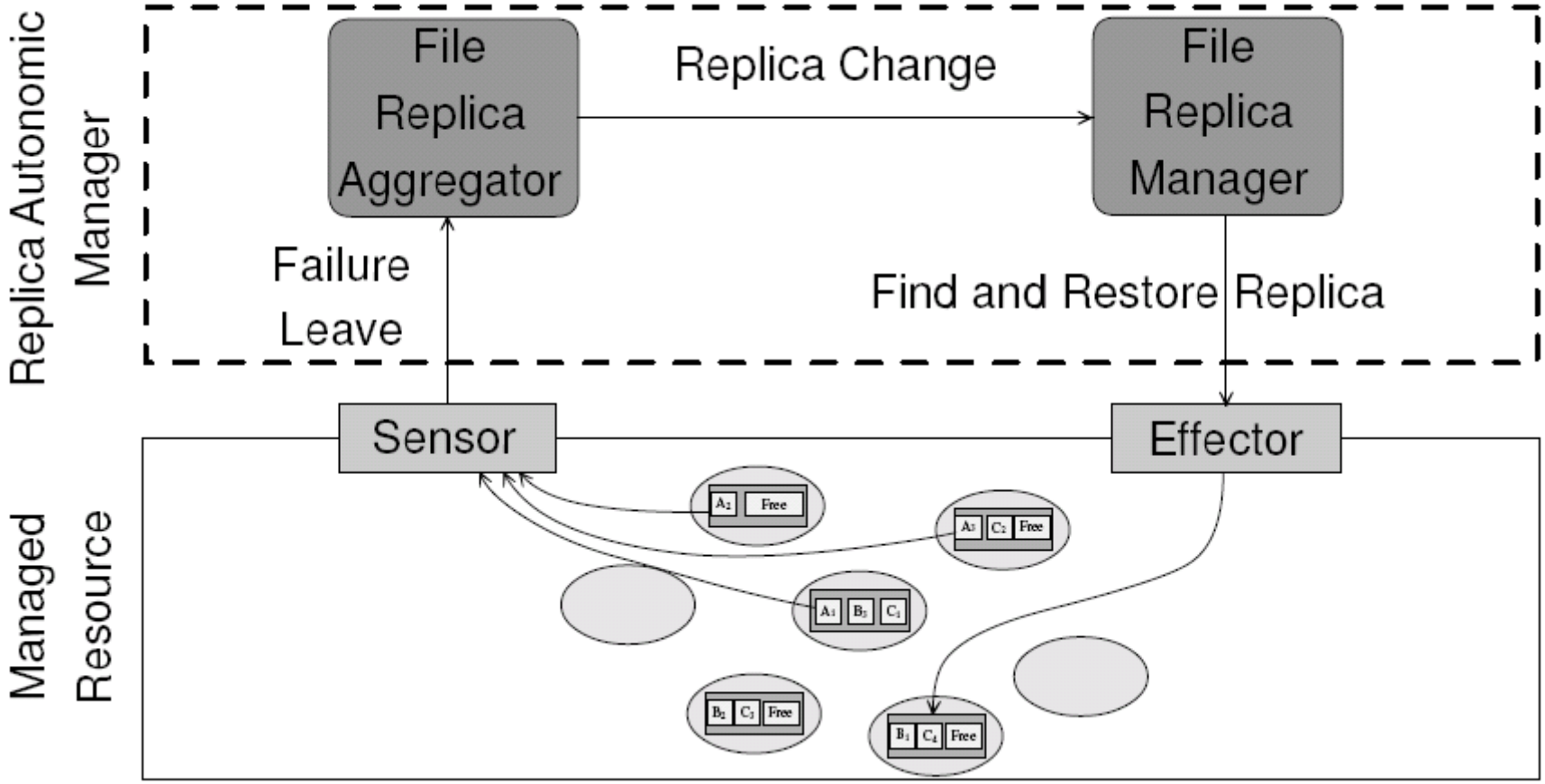


YASS self-management

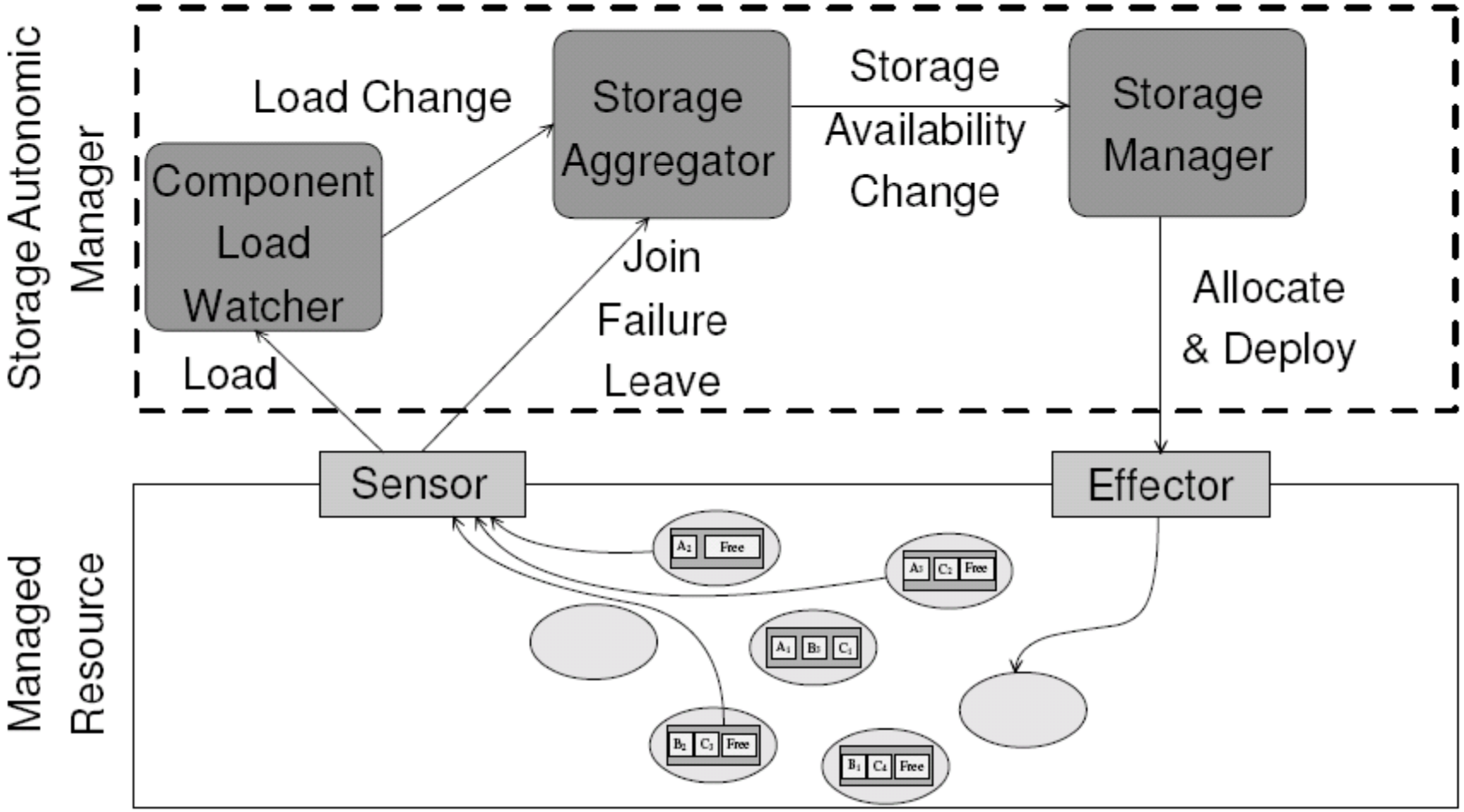
- Objectives
  1. Maintain the file replication degree
  2. Maintain the total storage space and total free space
  3. Increase the availability of popular files
  4. Release extra (unused) allocated storage
  5. Balance the stored files among the allocated resources

- Touchpoints
  - Sensors: Total free space, total storage space, access frequency, etc.
  - Actuators: replicate file, move file

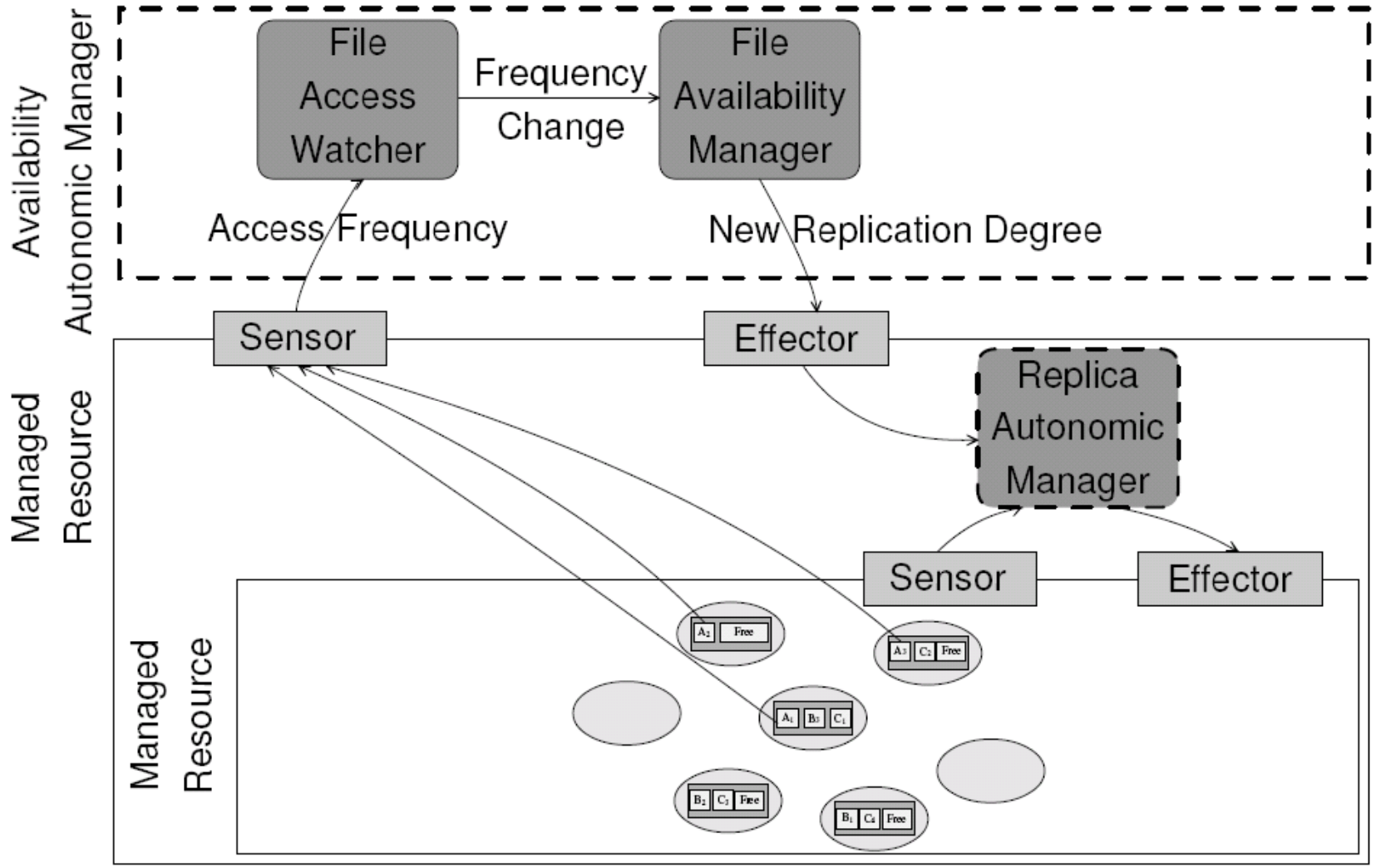
Self-Healing: Maintain the file replication degree



Self-Configuration: Maintain the total storage space and total free space



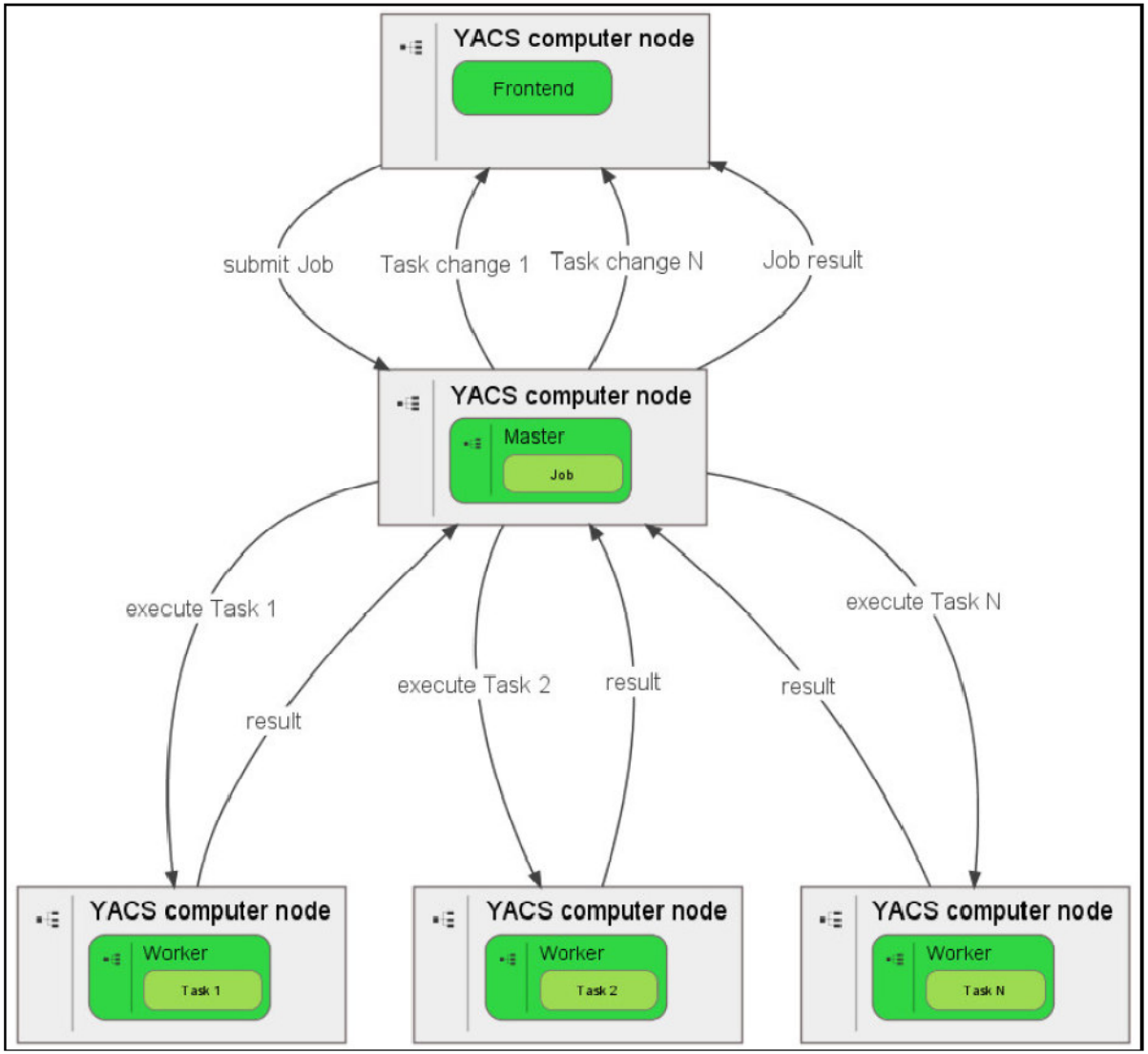
Increasing the availability of popular files



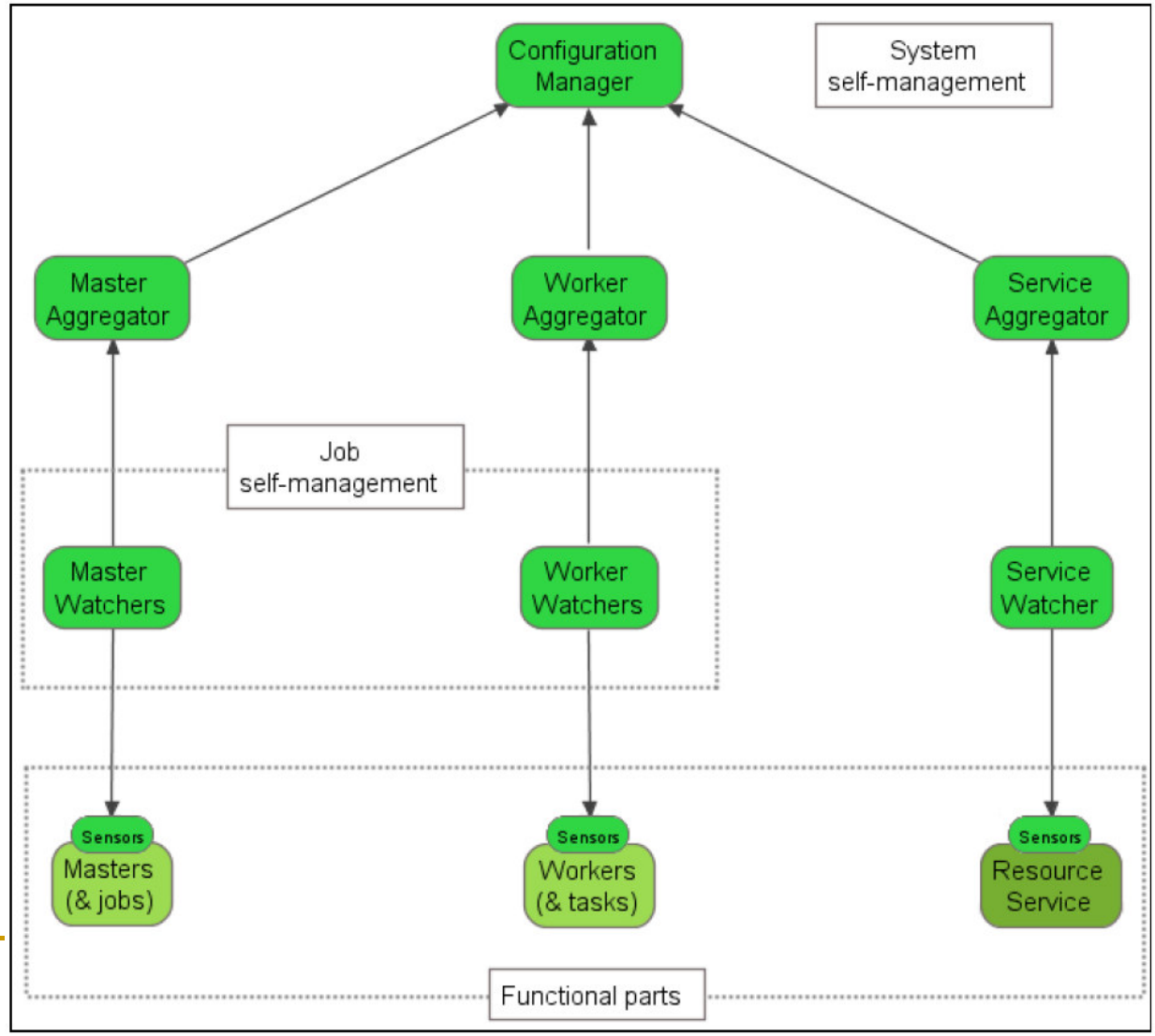
YACS: Yet Another Computing Service

- A robust distributed computing service that allows a client to submit and execute jobs, which are bags of independent tasks, on a network of nodes (computers).
  - Guarantees execution of jobs despite of nodes leaving or failing;
  - Scales, i.e. changes the number of execution components, when the number of jobs/tasks changes.
  - Supports checkpointing that allows restarting execution from the last checkpoint when a worker component fails or leaves.

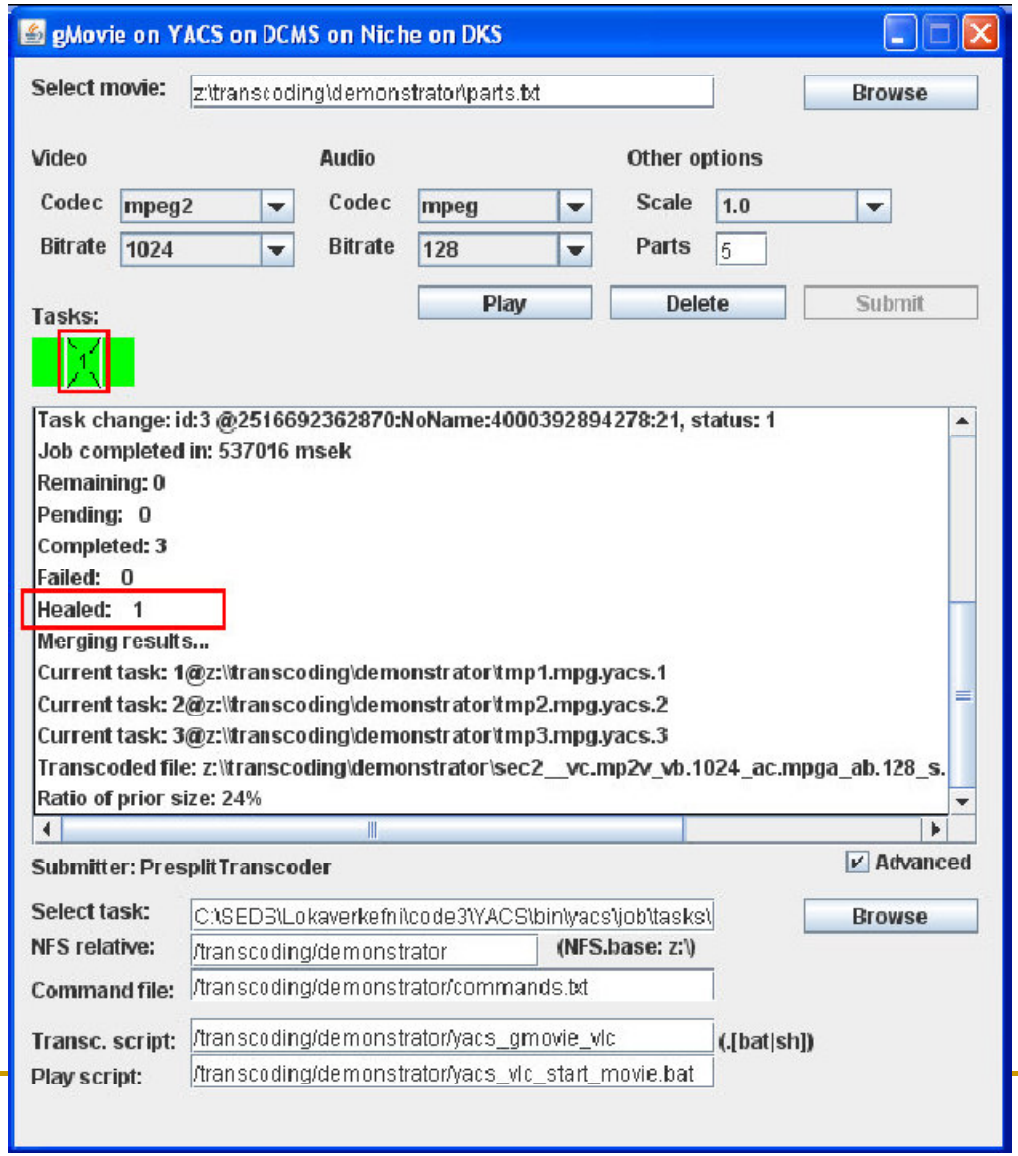
YACS Functional Part: Frontend, Masters and Workers



YACS Management Part



GUI of gMovie Using YACS



Lessons Learnt (1/2)

- A middleware, such as Niche, clearly reduces burden from an application developer because it enables and supports self-management
  - by leveraging self-organizing properties of structured P2P overlays;
  - by providing useful overlay services such as deployment, DHT (can be used for different indexes) and name-based communication
- Comes at a cost of self-management overhead, in particular, the cost of monitoring and replication of management
  - though this cost is necessary in dynamic environments

Lessons Learnt (2/2)

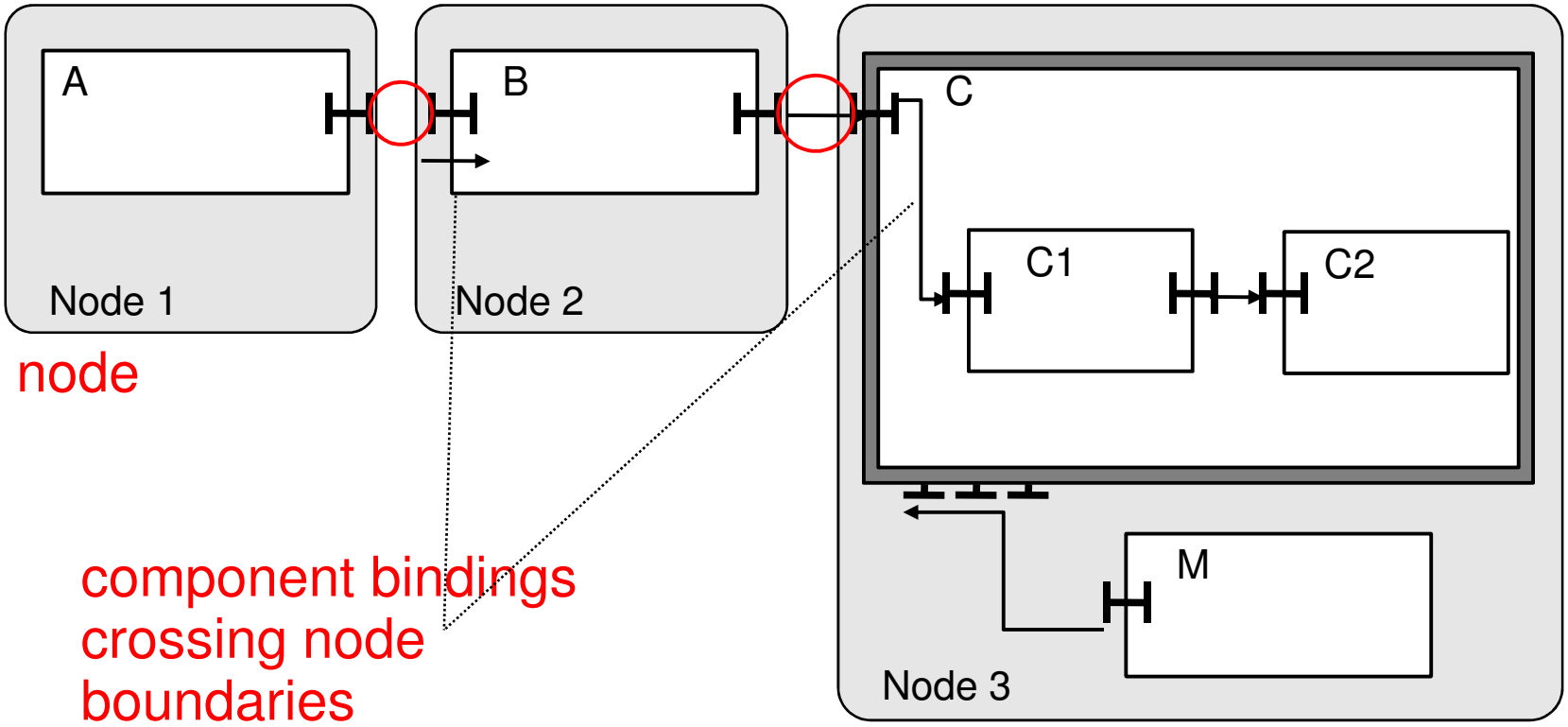
- Management functions should be distributed among several cooperative autonomic managers
  - Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns.
- Design steps include
  - spatial and functional partitioning of management,
  - assignment of management tasks to autonomic managers,
  - orchestration of multiple autonomic managers.
- Design space of manager interaction includes
  - indirect stigmergy-based interaction,
  - hierarchical management,
  - direct interaction,
  - using shared management elements.
- The major way to achieve robust self-management is to replicate management elements.

Future Work

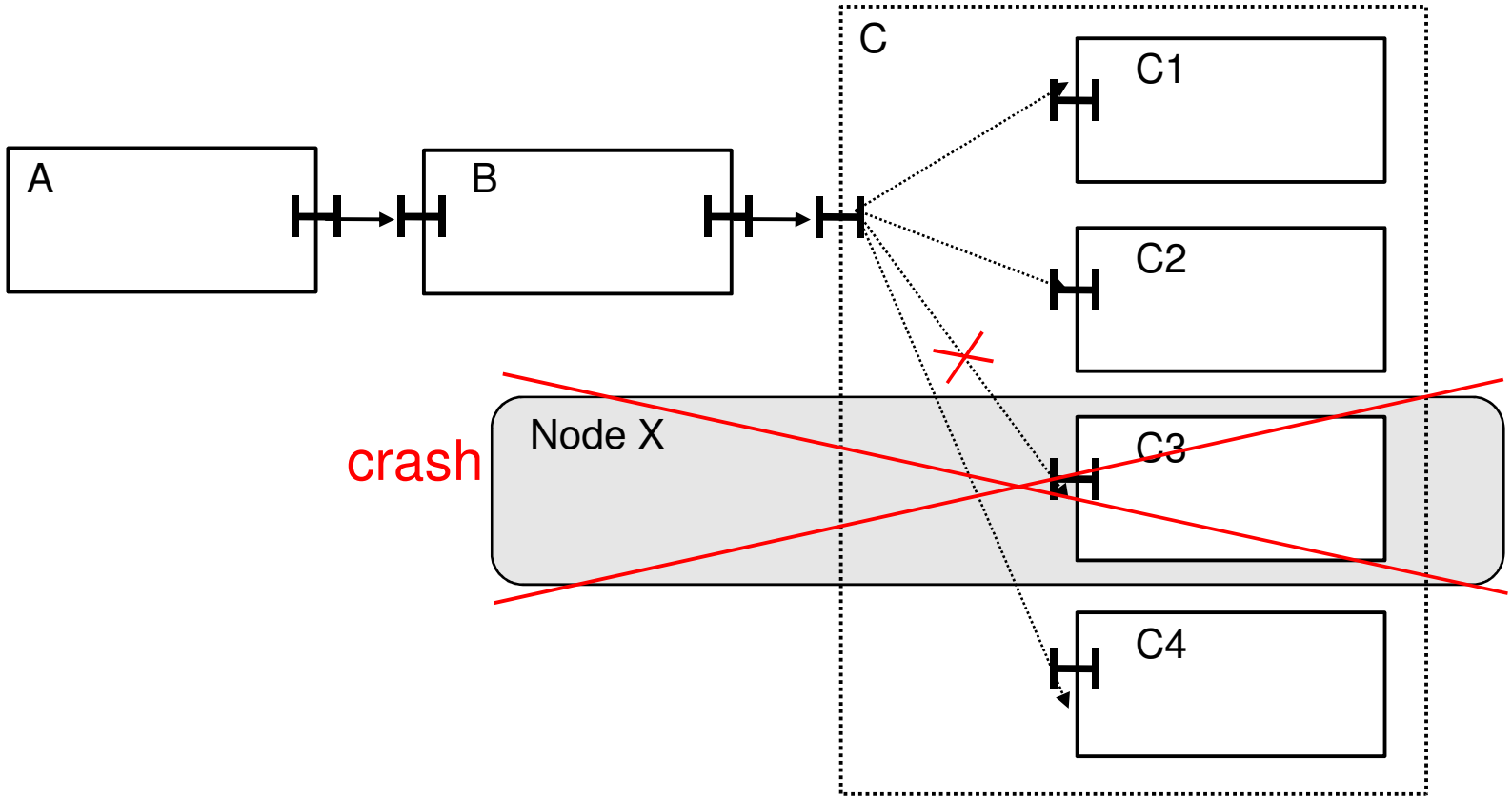
- Research on efficient monitoring and information gathering/aggregating infrastructures to reduce monitoring overhead
- Research on high-level programming abstractions, language support and tools that facilitate development of self-managing applications.
- The issue of coupled control loops (e.g. oscillations)
- Study large-scale systems (performance issues, oscillations)
- Research on efficient management replication

Additional slides

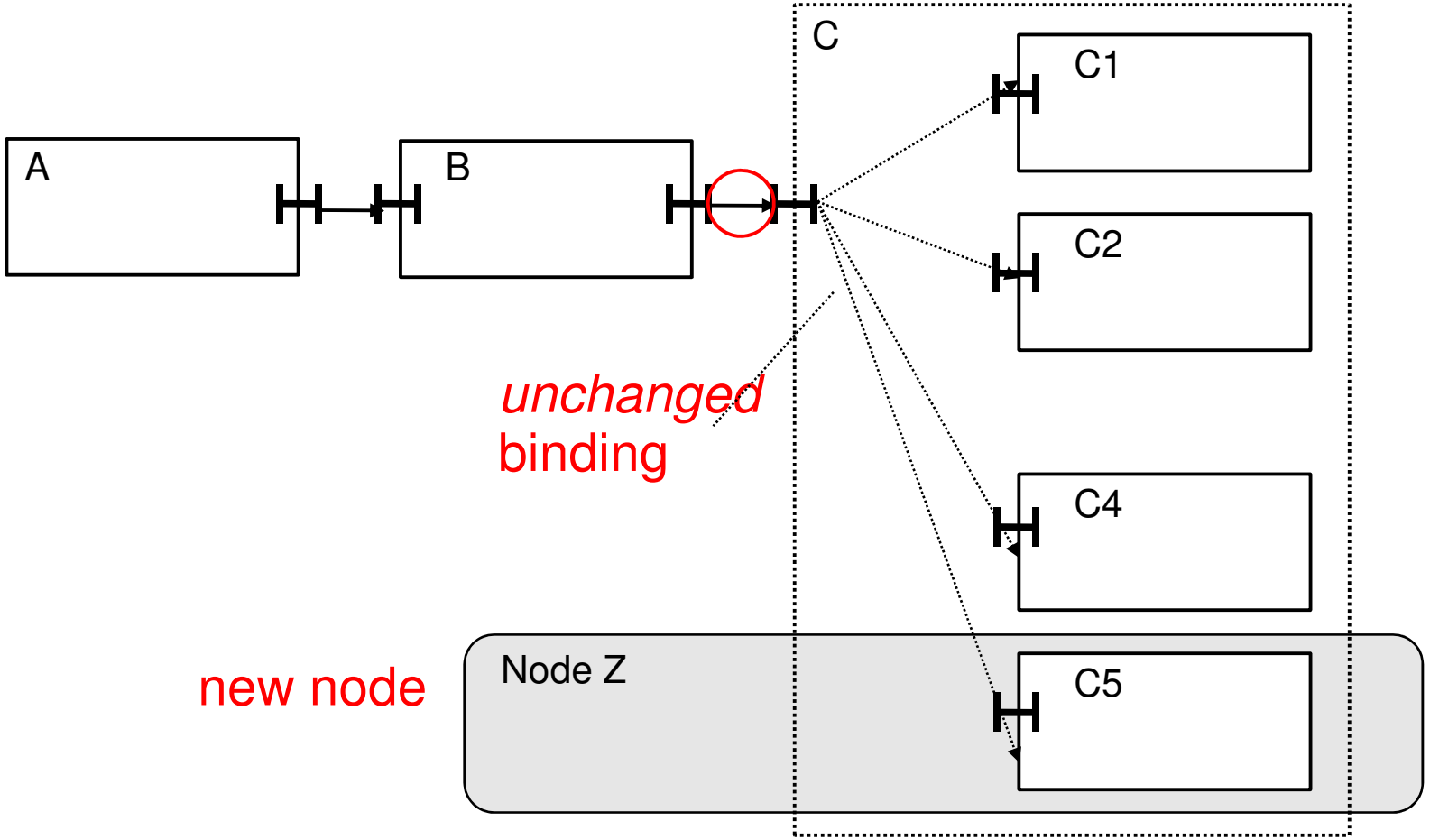
Network-Transparency for Component-Based Applications



Hiding Churn from Applications Using Group Components (I)

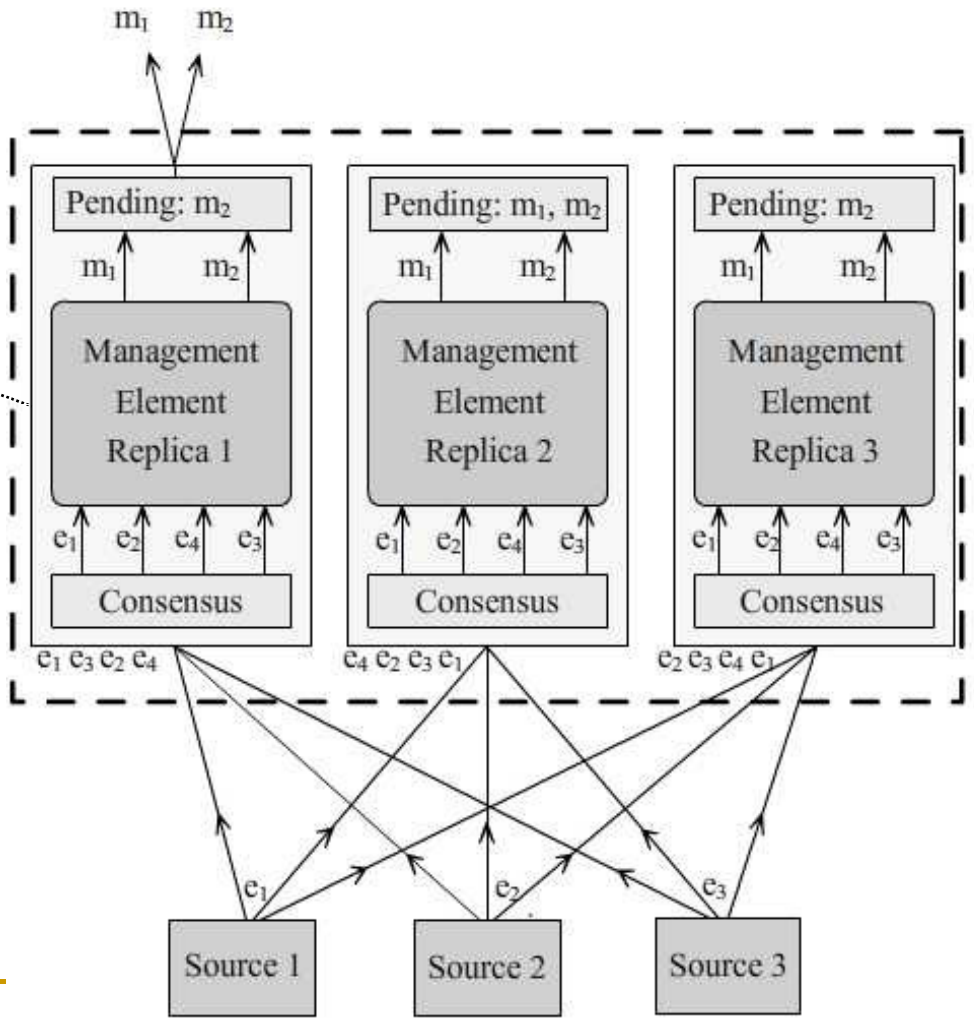


Hiding Churn from Applications Using Group Components (II)

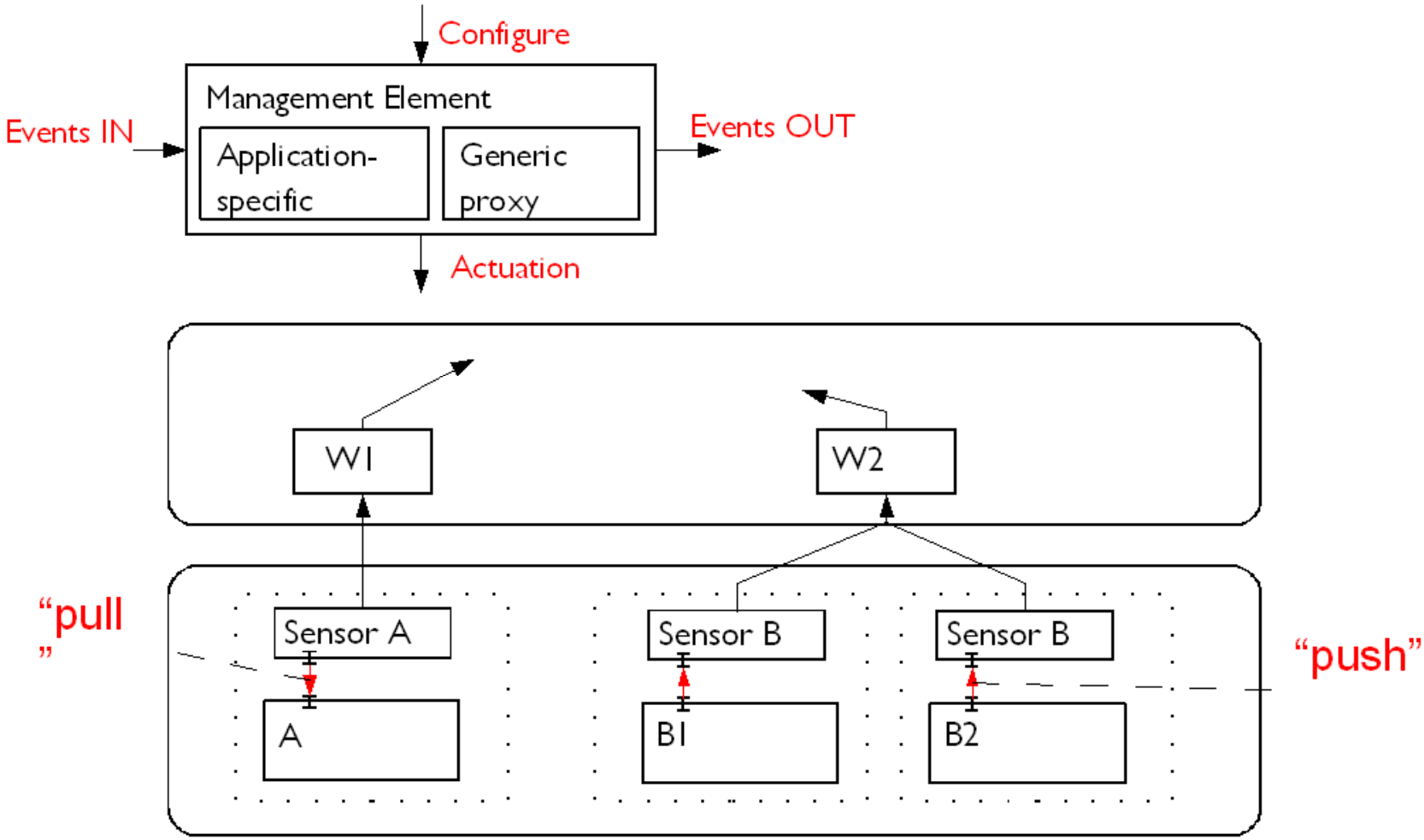


Making The Self-Management Reliable Using Replication

Management element *wrappers*



MEs and Sensors



References

- Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, Seif Haridi, "A Design Methodology for Self-Management in Distributed Environments," CSE, pp.430-436, 2009 International Conference on Computational Science and Engineering, 2009
- L. Bao, A. Al-Shishtawy, and V. Vlassov, “Policy based self-management in distributed environments,” in Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2009), (San Francisco, California), September 2009.
- A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, “Enabling self-management of component based distributed applications,” in From Grids to Service and Pervasive Computing (T. Priol and M. Vanneschi, eds.), pp. 163–174, Springer US, July 2008.
- A. Al-Shishtawy, J. Höglund, K. Popov, N. Parlavantzas, V. Vlassov, and P. Brand, “Distributed control loop patterns for managing distributed applications,” in Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008), (Venice, Italy), pp. 260–265, Oct. 2008.
- P. Brand, J. Höglund, K. Popov, N. de Palma, F. Boyer, N. Parlavantzas, V. Vlassov, and A. Al-Shishtawy, “The role of overlay services in a self-managing framework for dynamic virtual organizations,” in Making Grids Work (M. Danelutto, P. Fragopoulou, and V. Getov, eds.), pp. 153–164, Springer US, 2007.