Enabling and Achieving Self-Management for Large Scale Distributed Systems

Platform and Design Methodology for Self-Management

Ahmad Al-Shishtawy
ahmadas@kth.se

Unit of Software and Computer Systems (SCS)
School of Information and Communication Technology (ICT)
The Royal Institute of Technology (KTH)

Licentiate Seminar
April 9 2010

Outline

Outline

Outline

Outline

Outline

Outline

The Autonomic Computing Initiative

Problem

All computing systems need to be managed

The Autonomic Computing Initiative

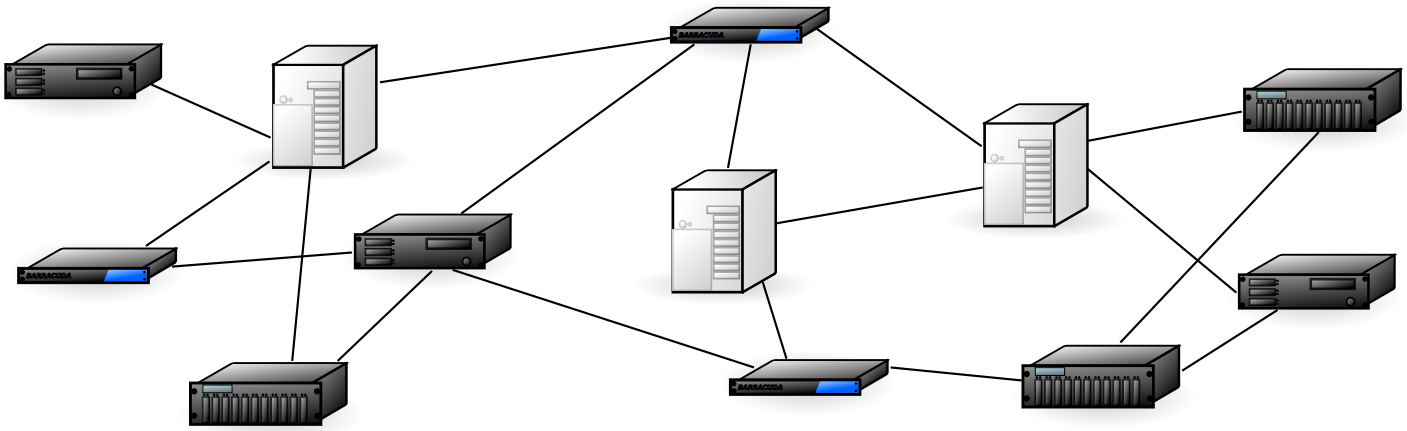
Problem

All computing systems need to be managed

The Autonomic Computing Initiative

Problem

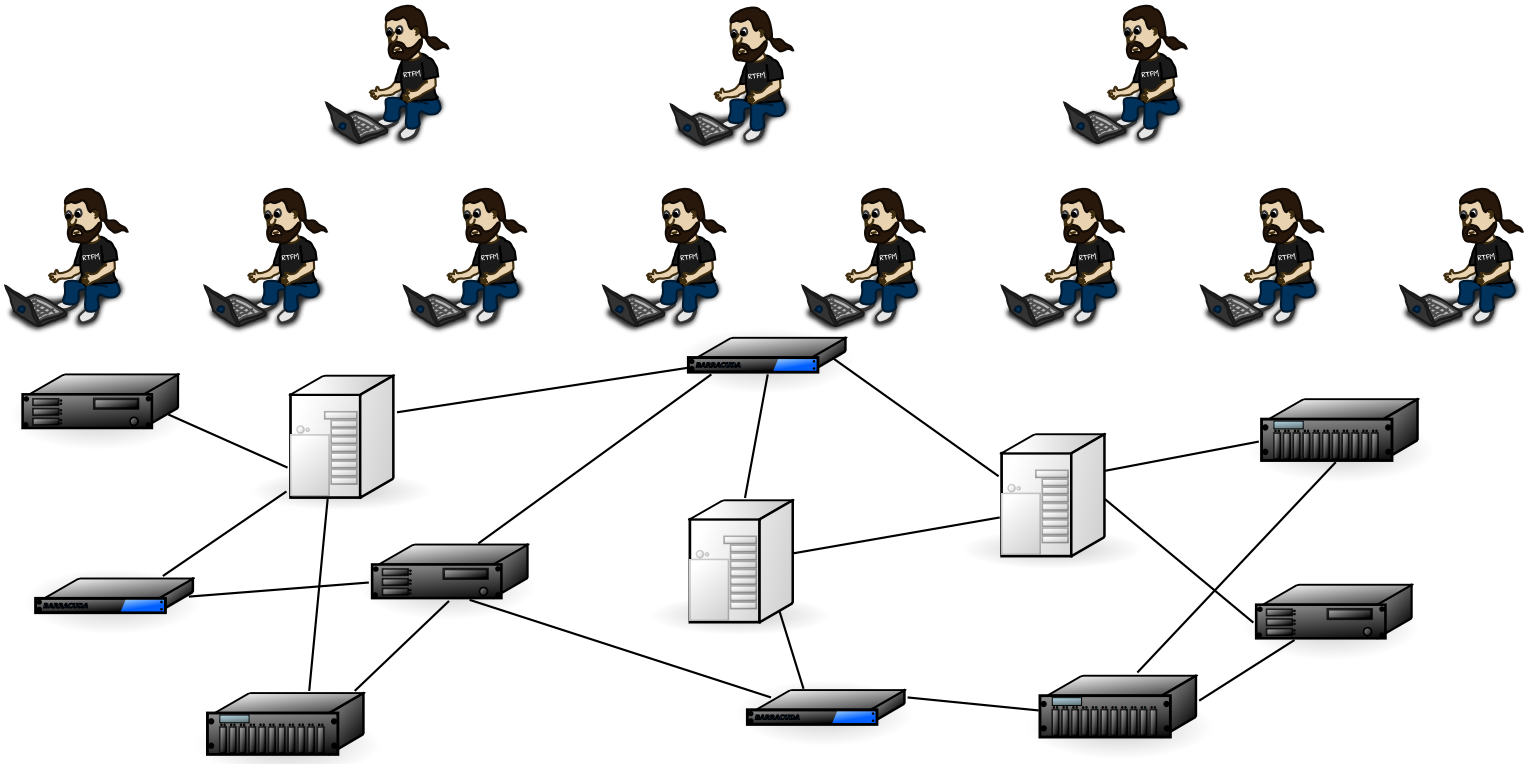
Computing systems are getting more and more complex



The Autonomic Computing Initiative

Problem

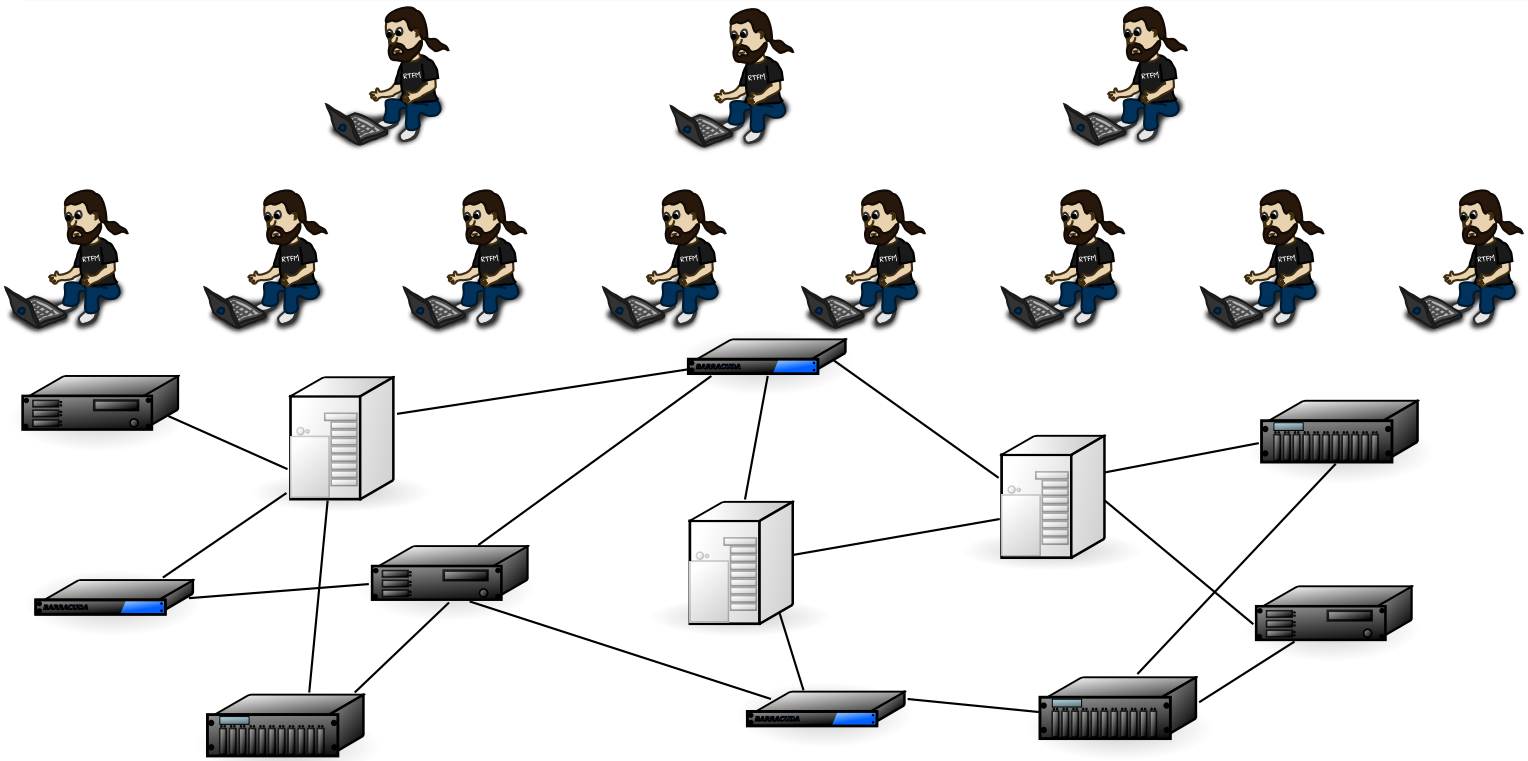
Complexity means higher administration overheads



The Autonomic Computing Initiative

Problem

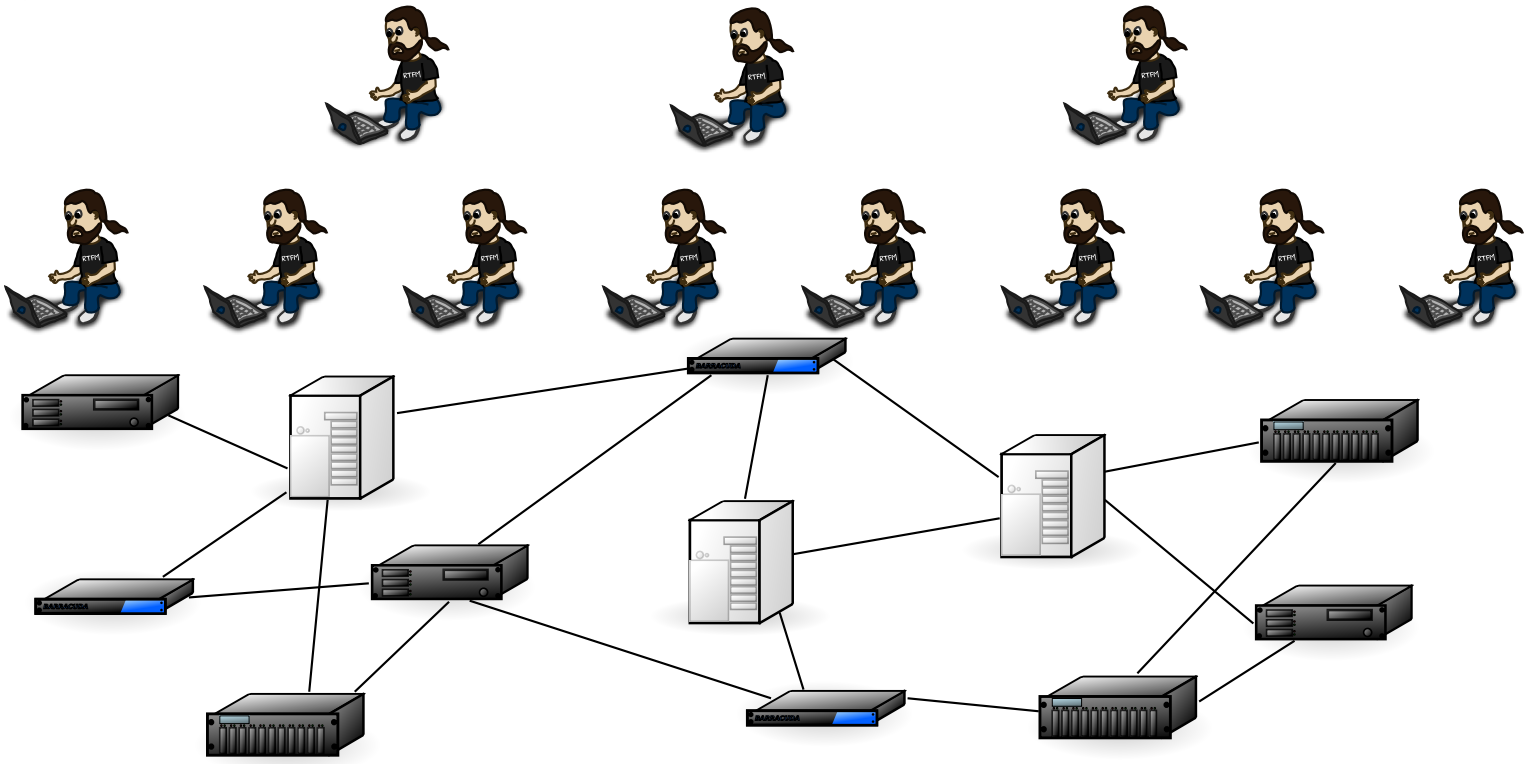
Complexity poses a barrier on further development



The Autonomic Computing Initiative

Solution

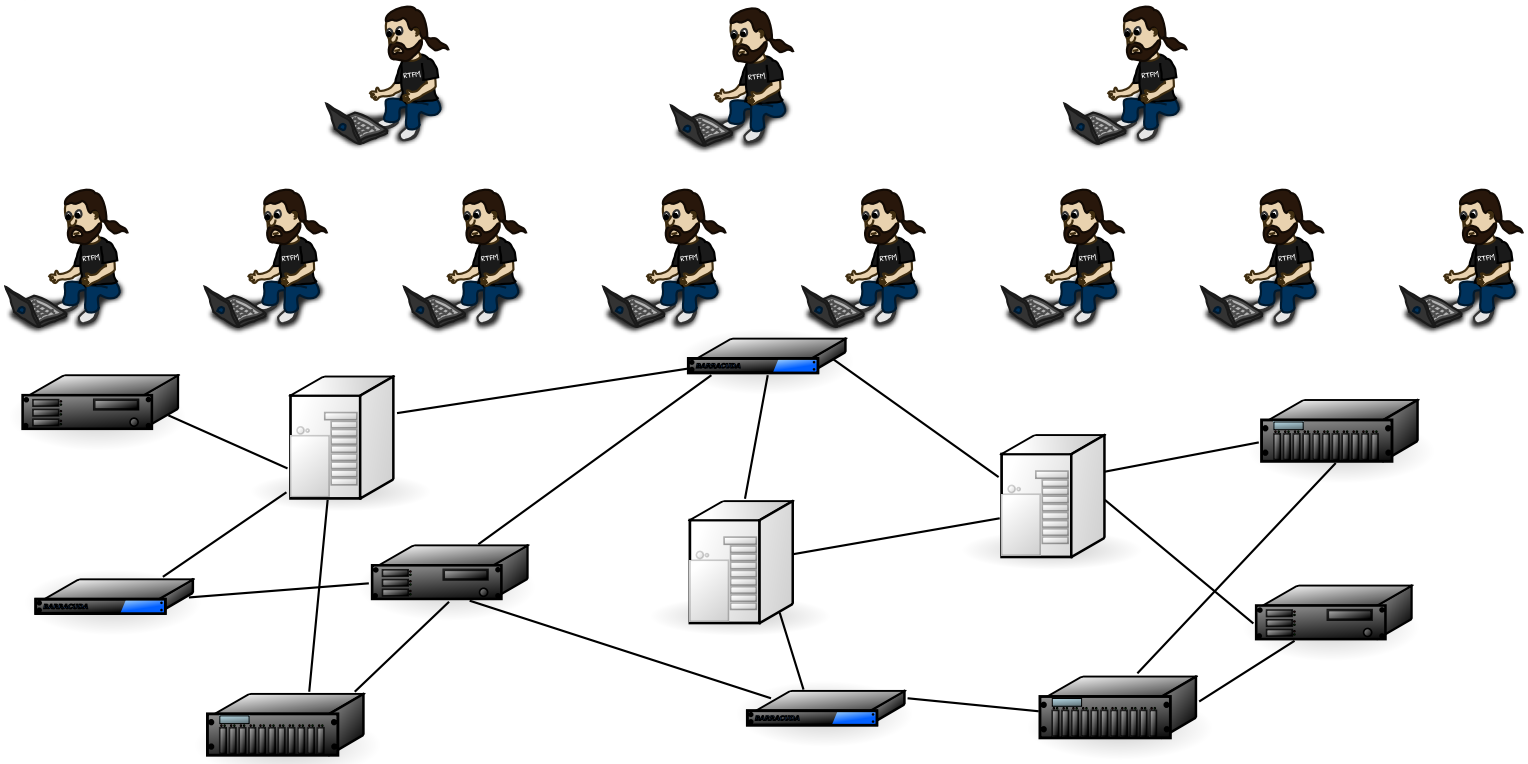
The Autonomic Computing initiative by IBM



The Autonomic Computing Initiative

Solution

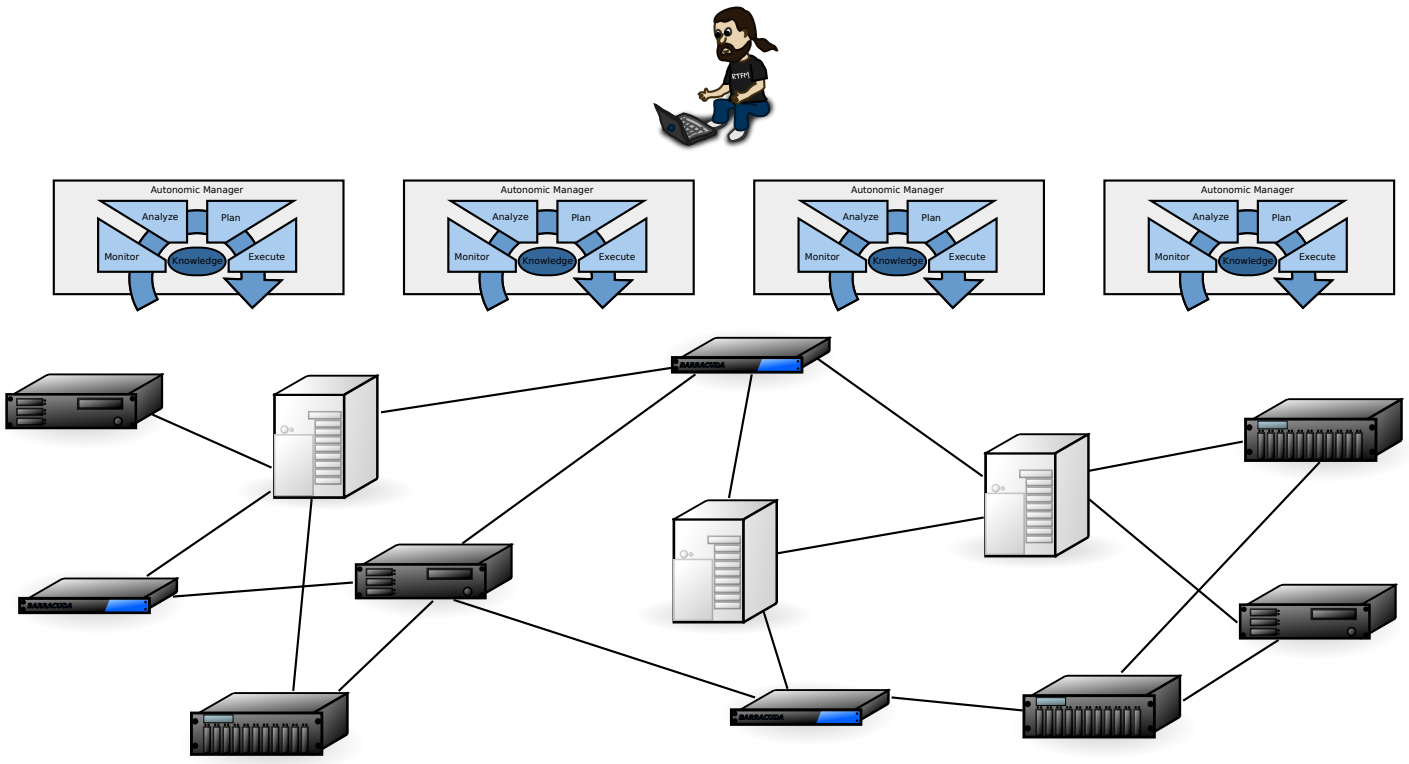
Self-Management: Systems capable of managing themselves



The Autonomic Computing Initiative

Solution

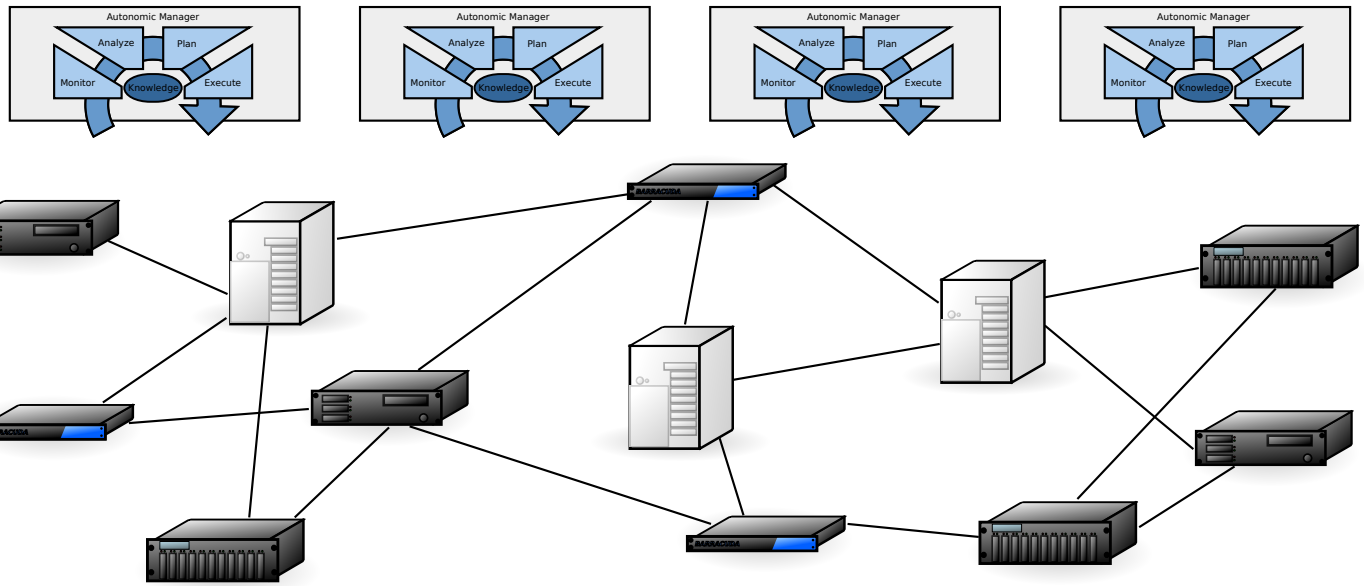
Use Autonomic Managers



The Autonomic Computing Initiative

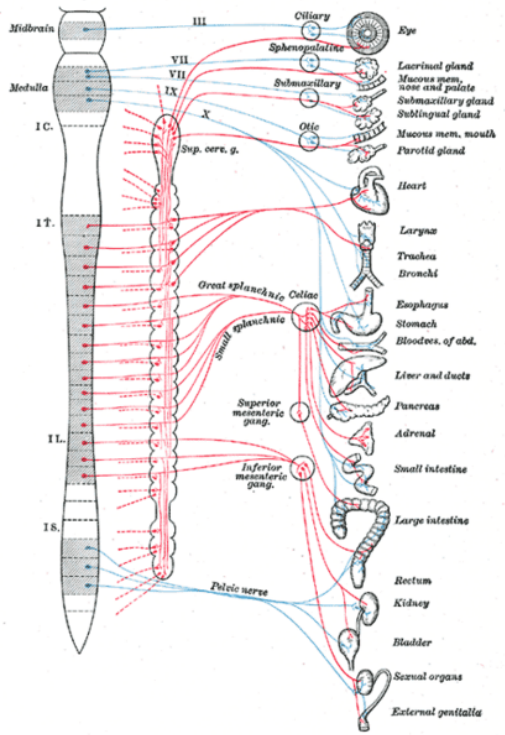
Open Question

How to achieve Self-Management?



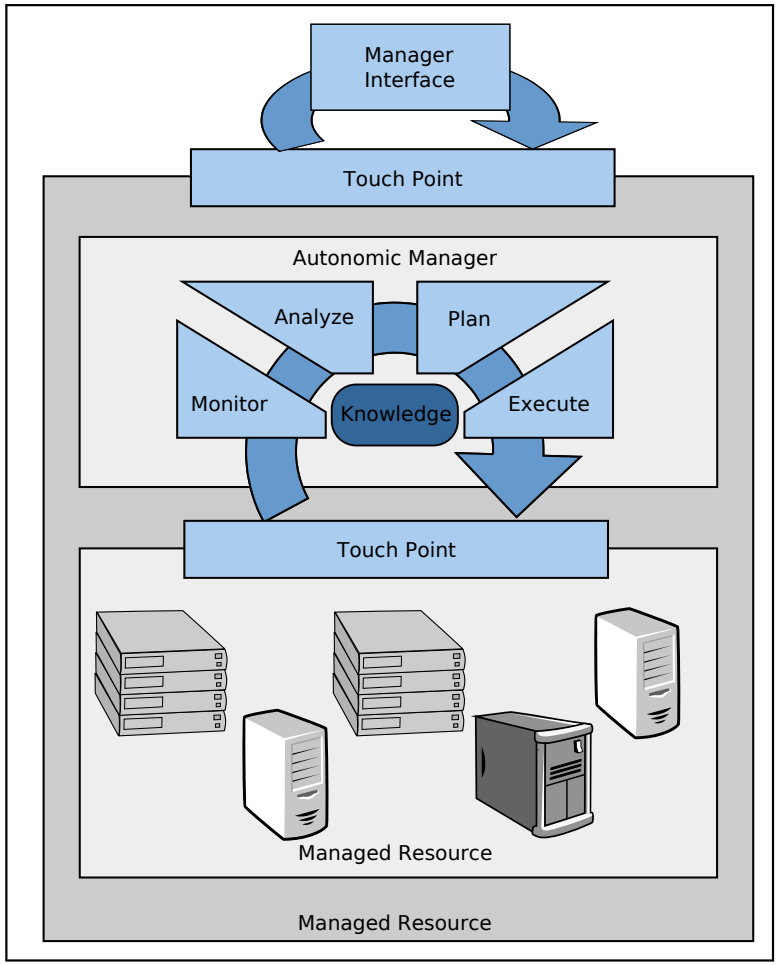
Self-* Properties

- Inspired by the autonomic nervous system of the human body
- Control loops from Control Theory
- Self-management along four main axes (self-* properties):
  - self-configuration
  - self-optimization
  - self-healing
  - self-protection



The Autonomic Computing Architecture

- Managed Resource
- Touchpoint (Sensors & Actuators)
- Autonomic Manager
  - Monitor
  - Analyze
  - Plan
  - Execute
- Knowledge Source
- Communication
- Manager Interface



Problem Statement

Large-scale distributed systems

- Complex and require self-management
- May run on unreliable resources
- Major sources of complexity:
  - Scale (resources, events, users, . . .)
  - Dynamism (resource churn, load changes, . . .)

Goal

- A platform (concepts, abstractions, algorithms. . .) that facilitates development of self-managing applications in large-scale and/or dynamic distributed environment.
- A methodology that help us to achieve self-management.

Problem Statement

Large-scale distributed systems

- Complex and require self-management
- May run on unreliable resources
- Major sources of complexity:
  - Scale (resources, events, users, . . .)
  - Dynamism (resource churn, load changes, . . .)

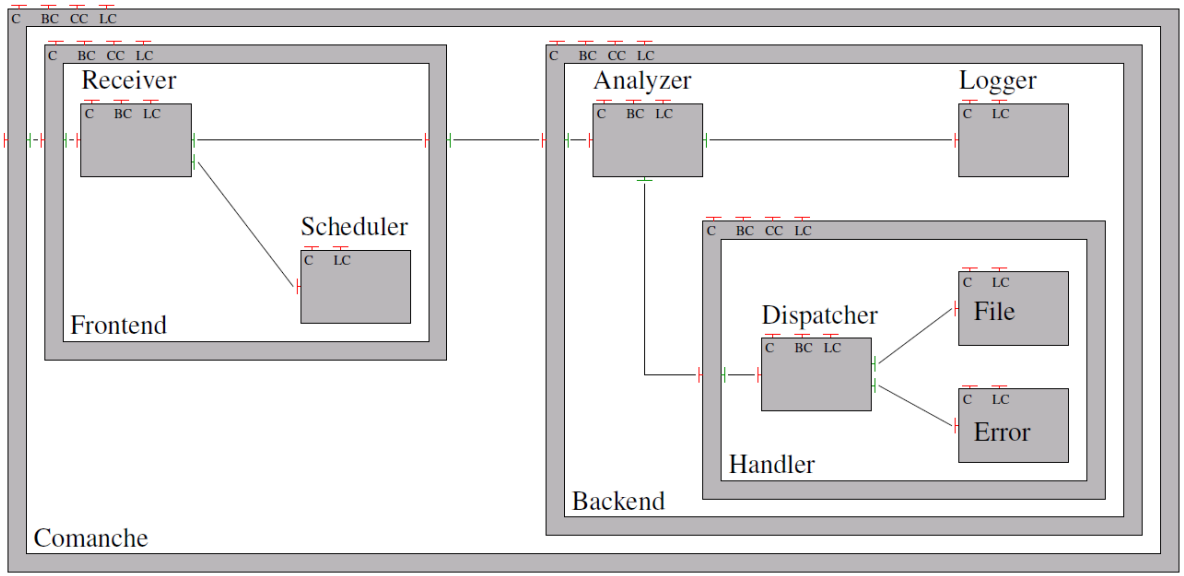
Goal

- A platform (concepts, abstractions, algorithms. . .) that facilitates development of self-managing applications in large-scale and/or dynamic distributed environment.
- A methodology that help us to achieve self-management.

Outline

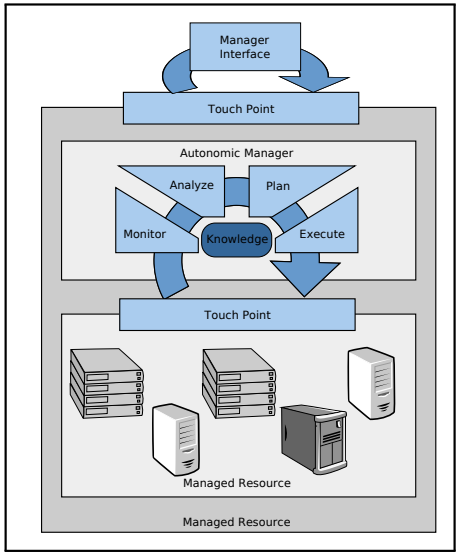
Component Model

- Architectural approach to autonomic computing
- Applications built of components
- Improved manageability through introspection and reconfiguration
- The Fractal component model



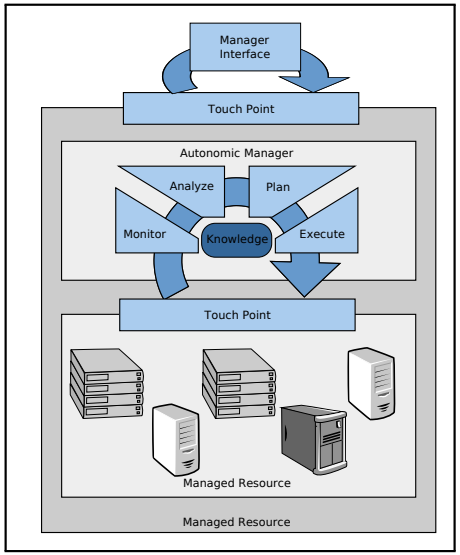
Niche

- Niche is a Distributed Component Management System
- Niche implements the Autonomic Computing Architecture
- Niche targets large-scale and dynamic distributed environment and applications
  - Resources and components are distributed
  - Autonomic managers are distributed network of Management Elements (MEs)
  - Sensors and Actuators are distributed



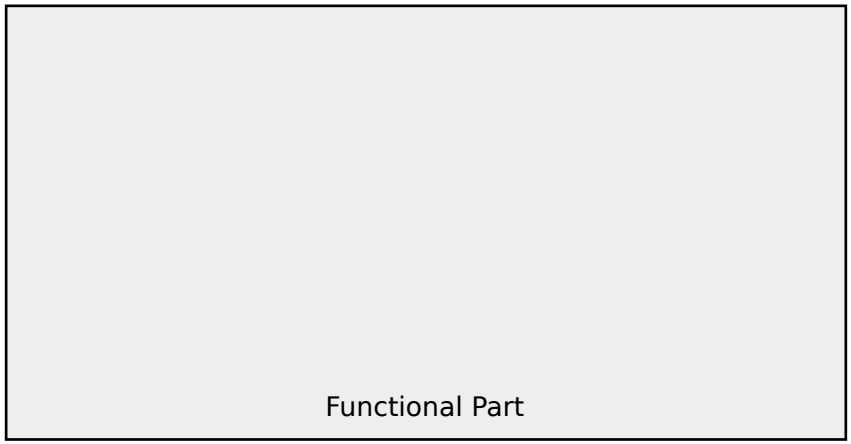
Niche

- Niche leverages Structured Overlay Networks (SONs) for communication and for provisioning of basic services
  - Name based communication and bindings
  - DHT, Publish/Subscribe, Groups, ...
- Niche separates functional part from management part of the application



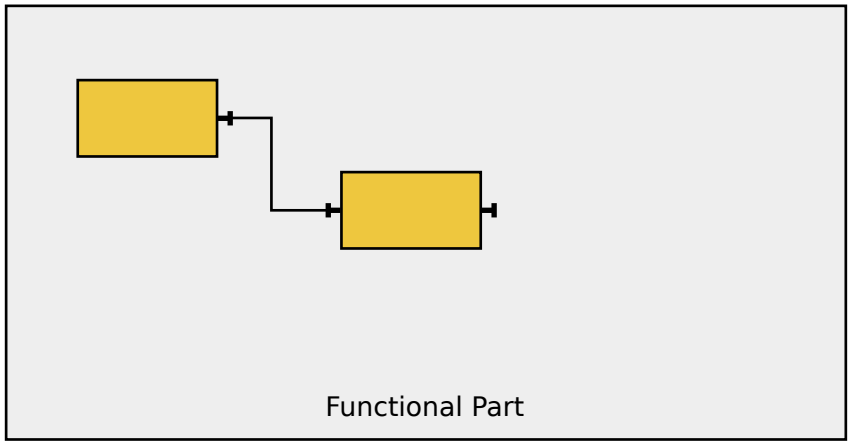
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



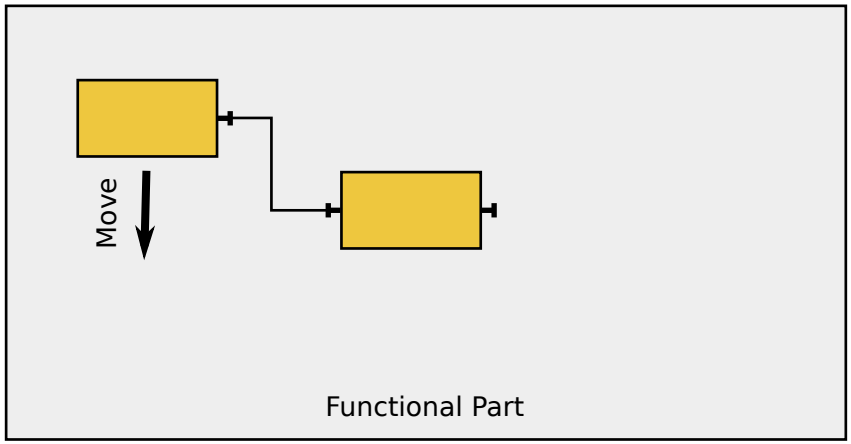
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



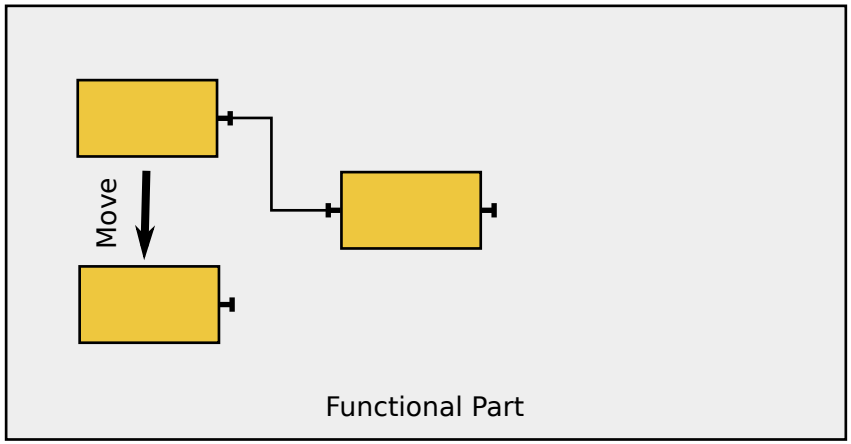
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



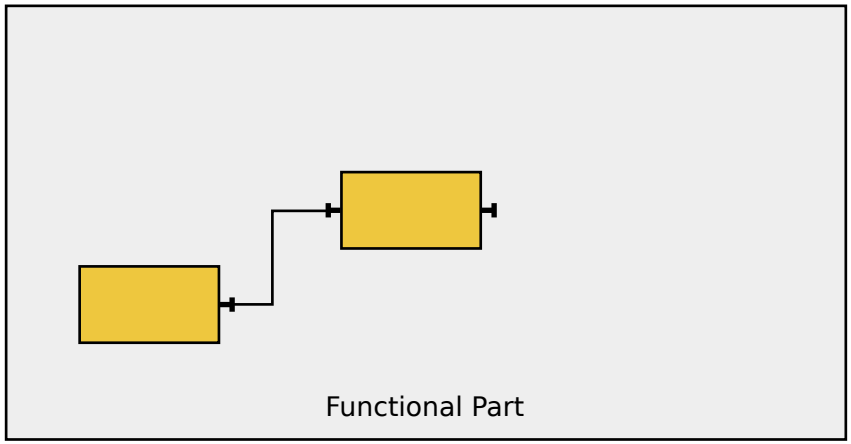
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



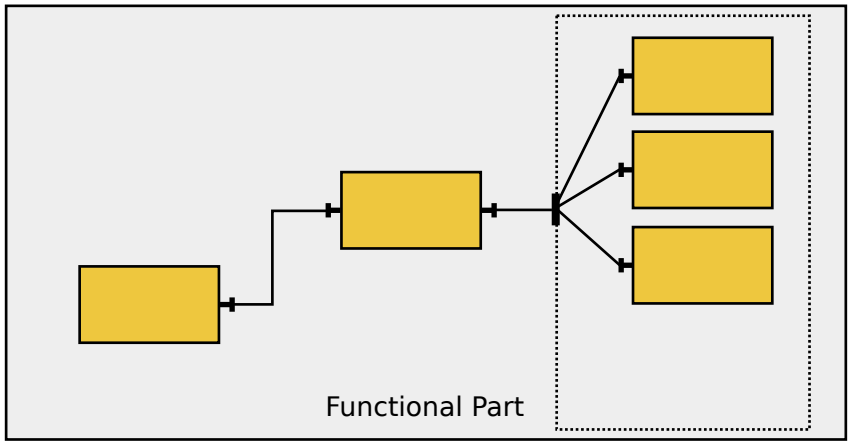
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



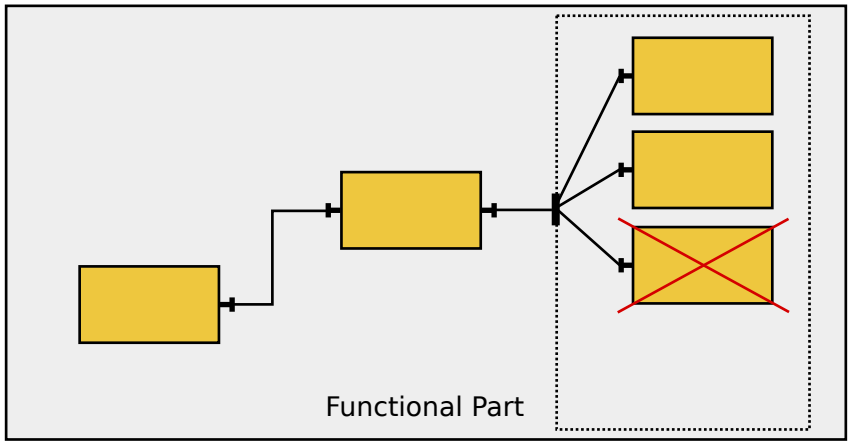
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



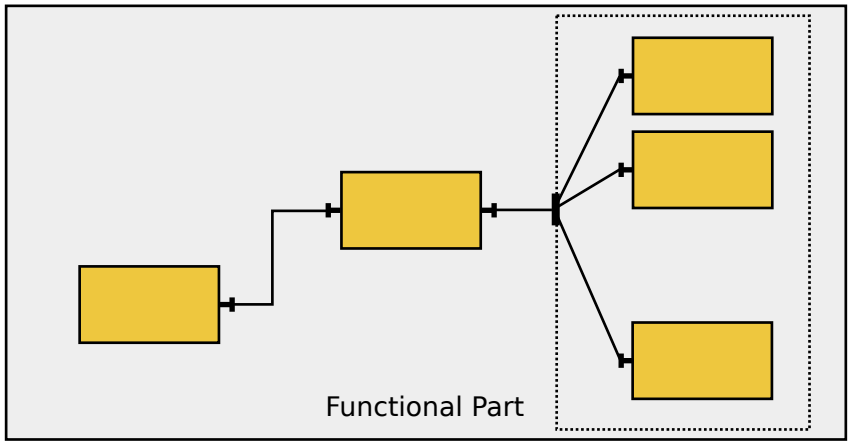
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



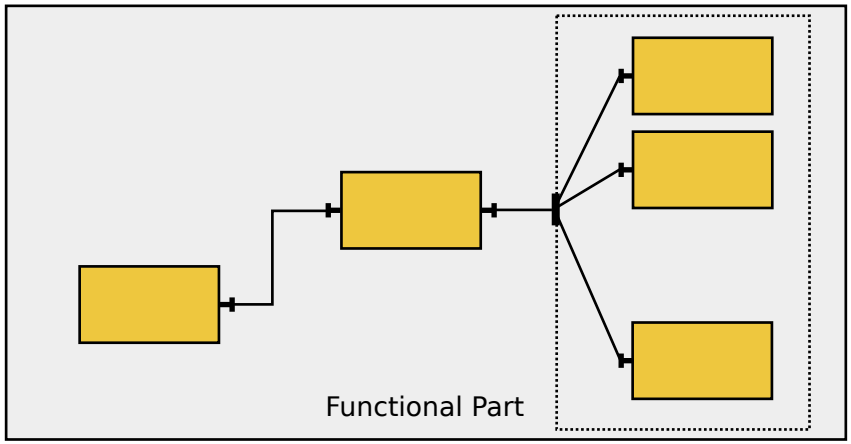
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



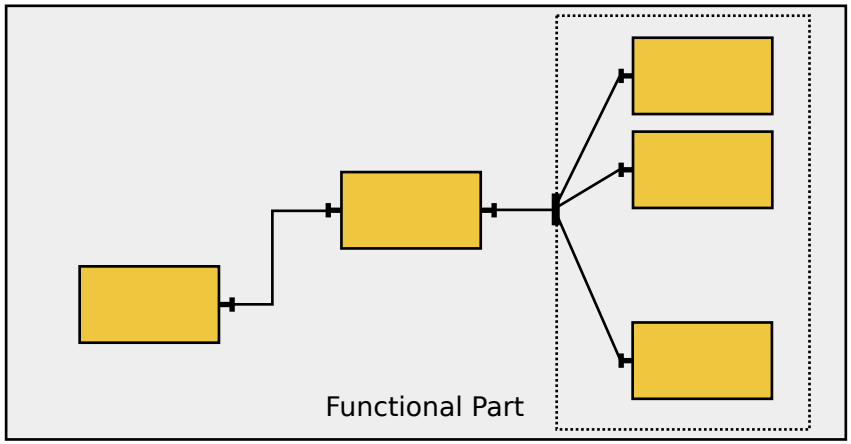
Functional Part

- Components, Interfaces, and Bindings
- System wide identification
- Support for mobility
- Component groups
- One-to-all and one-to-any bindings
- Dynamic group membership
- Deployment using ADL



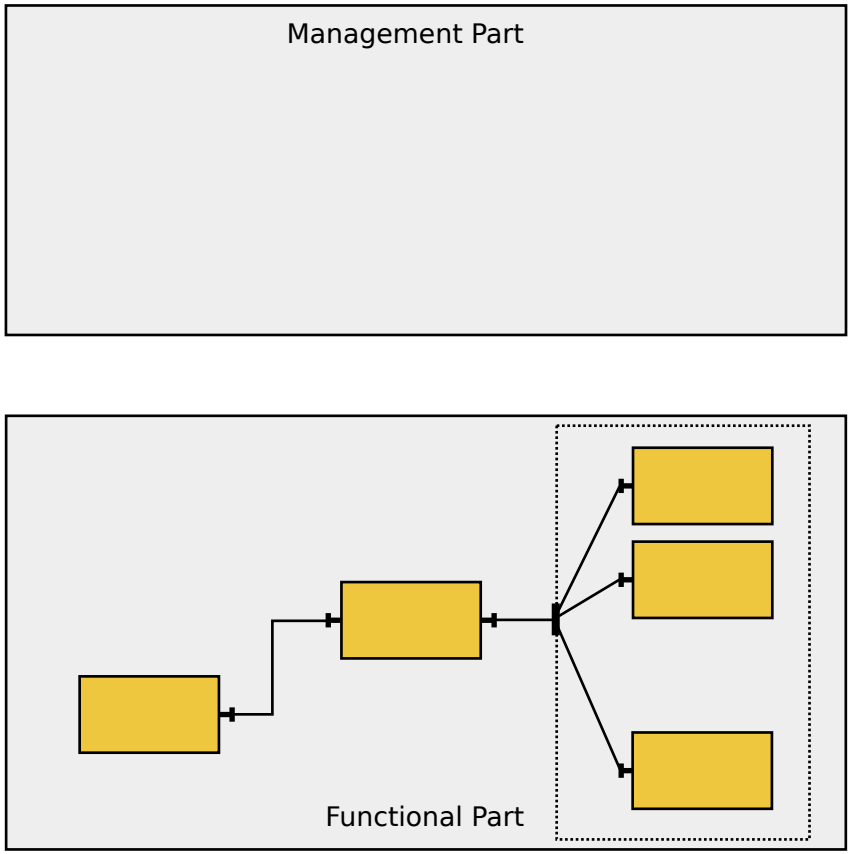
Management Part

- Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate through events
- Publish/Subscribe
- Autonomic Managers (control loops) built as network of MEs



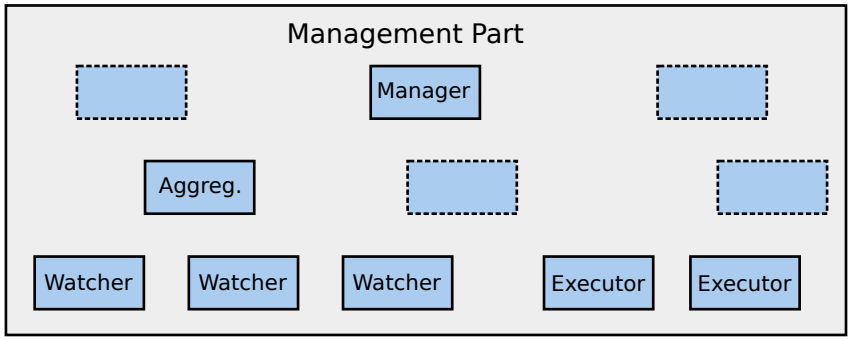
Management Part

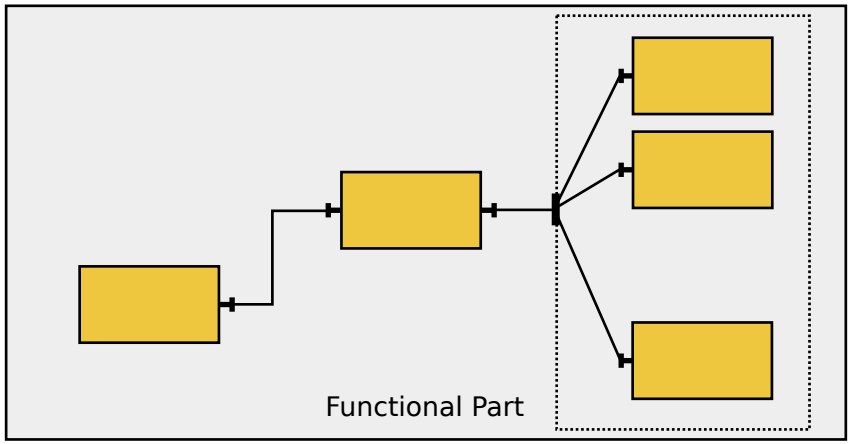
- Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate through events
- Publish/Subscribe
- Autonomic Managers (control loops) built as network of MEs



Management Part

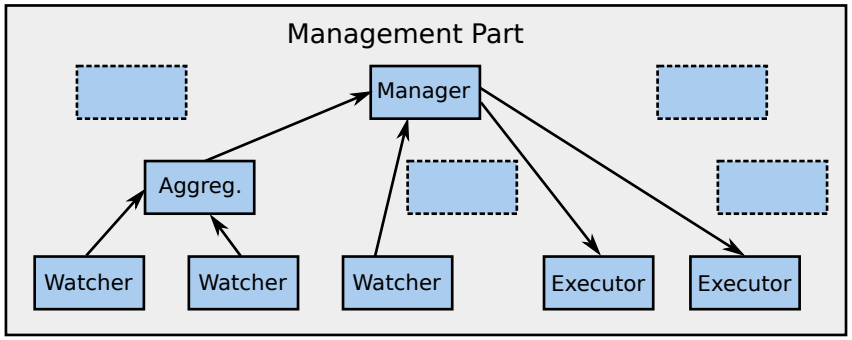
- Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate through events
- Publish/Subscribe
- Autonomic Managers (control loops) built as network of MEs

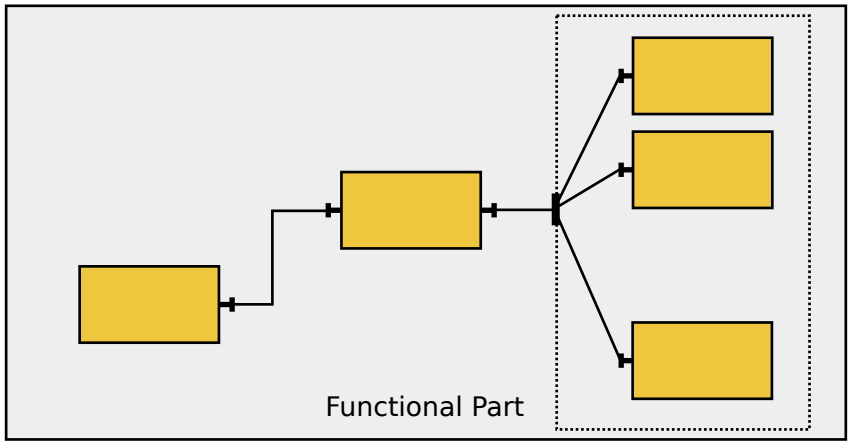




Management Part

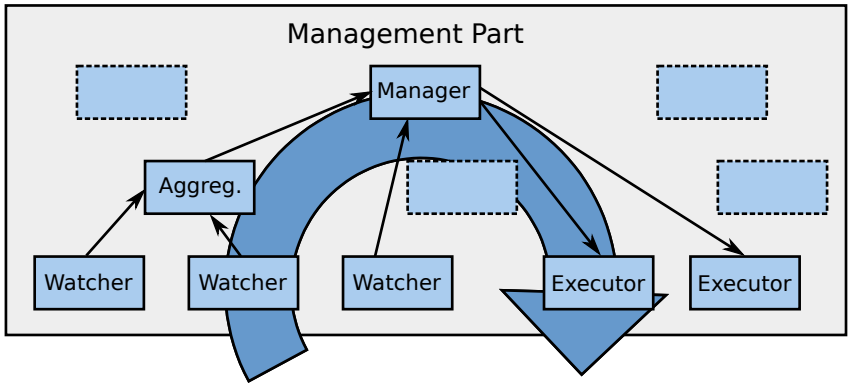
- Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate through events
- Publish/Subscribe
- Autonomic Managers (control loops) built as network of MEs

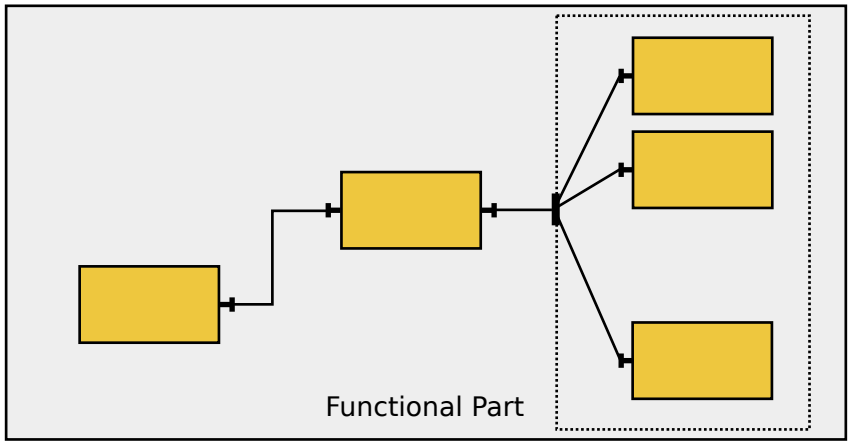




Management Part

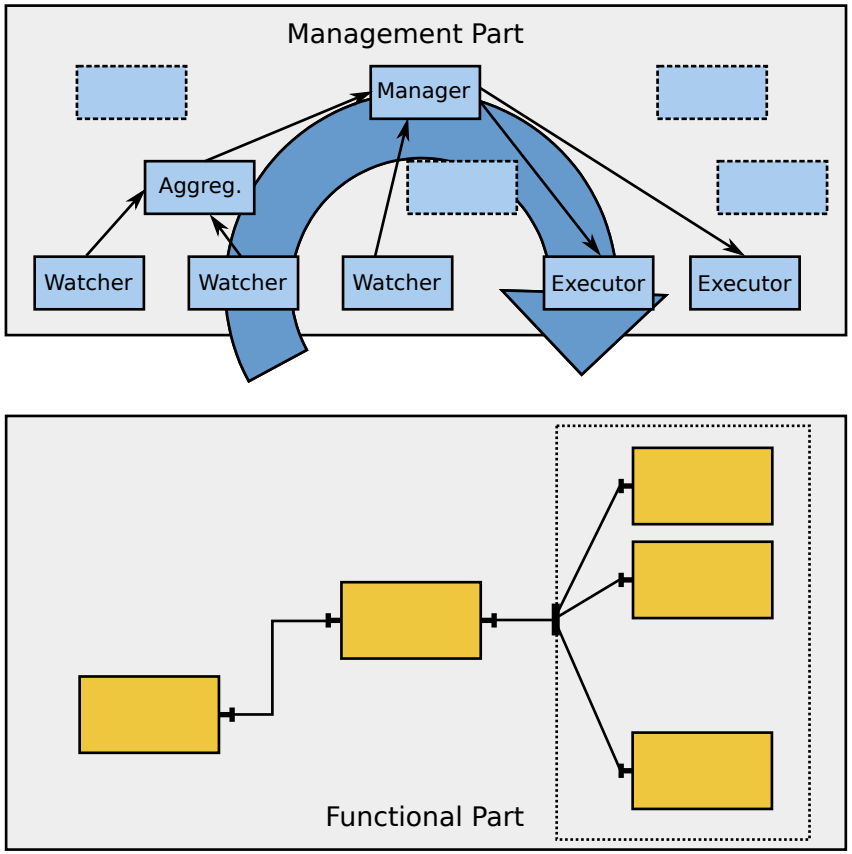
- Management Elements
  - Watchers
  - Aggregators
  - Managers
  - Executors
- Communicate through events
- Publish/Subscribe
- Autonomic Managers (control loops) built as network of MEs





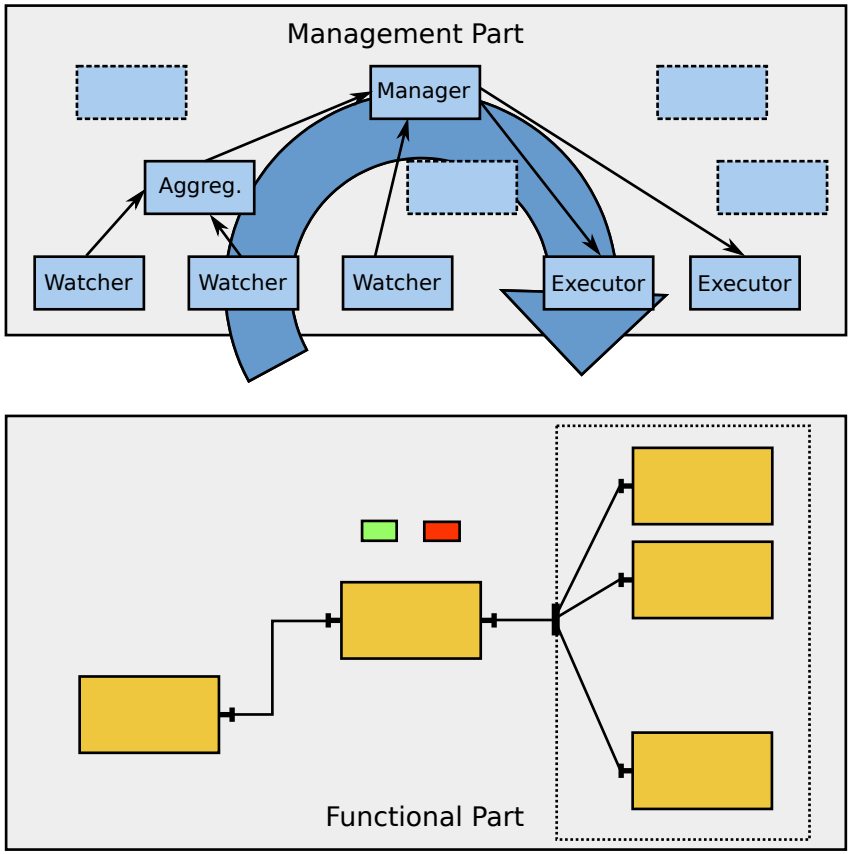
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



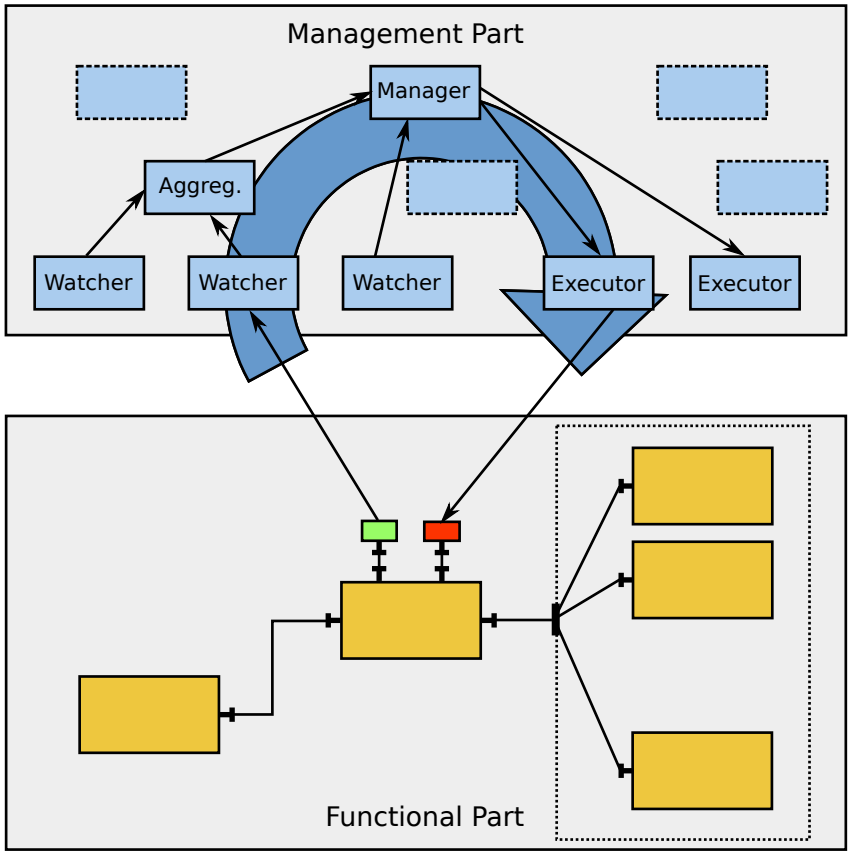
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



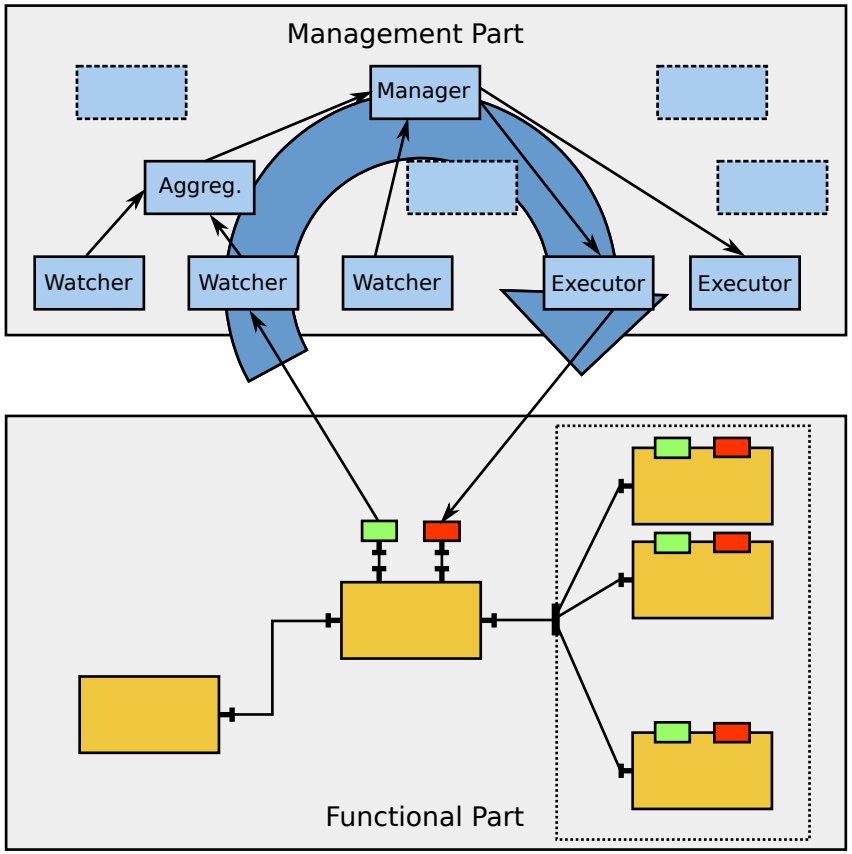
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



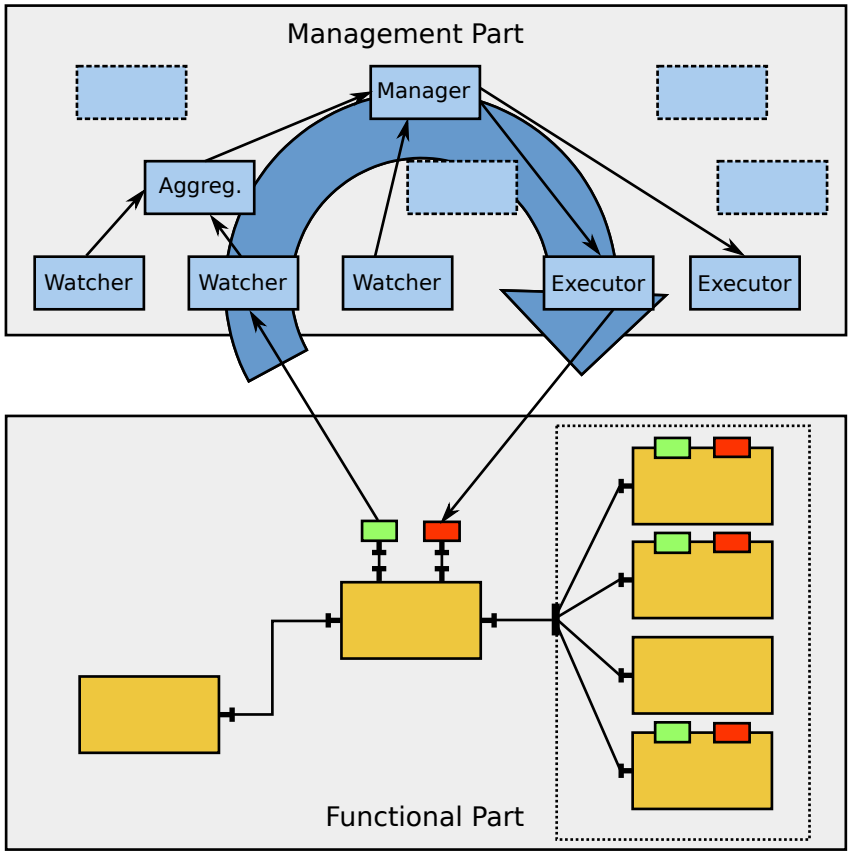
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



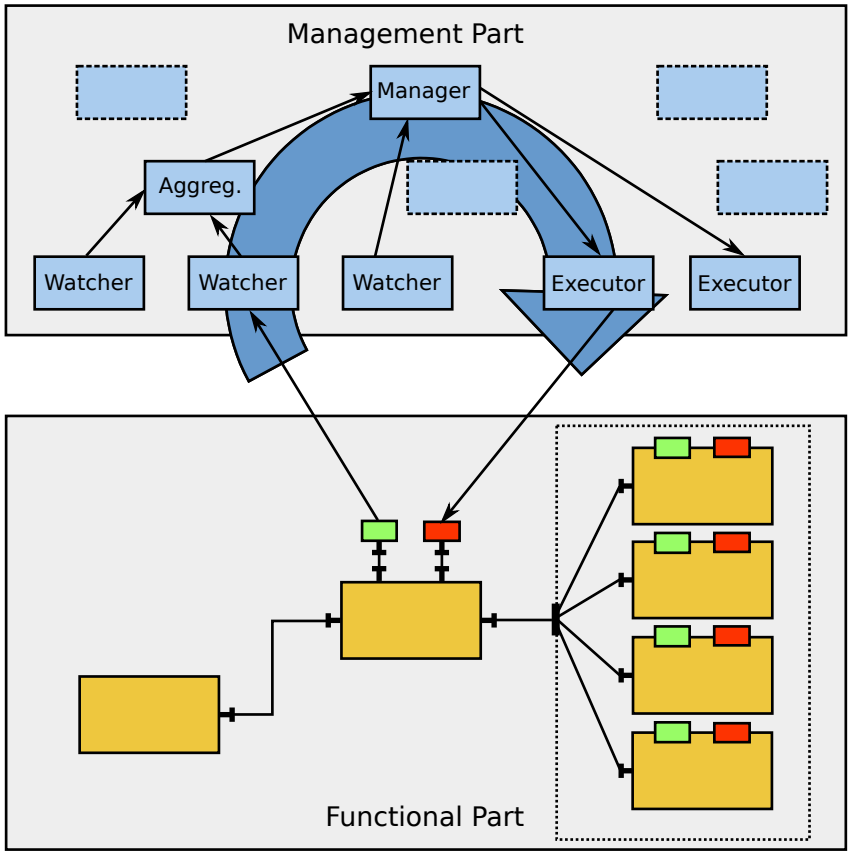
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



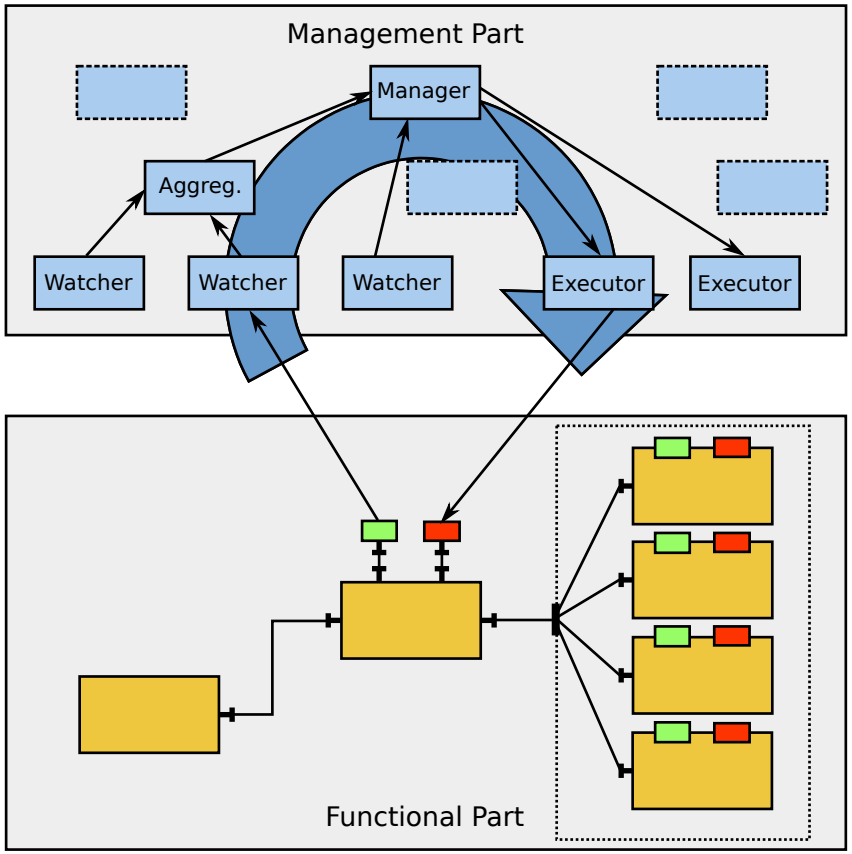
Touchpoints

- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



Touchpoints

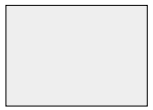
- Sensors and Actuators
- For Components and Groups
- Automatically install sensors/actuators on group members
- Predefined events (failures, group creation, . . .)
- API (bind, start/stop, create group, discover, . . .)



Runtime Environment

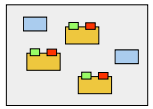
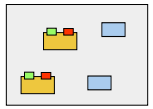
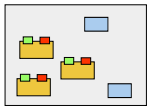
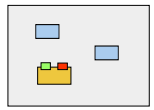
- Containers that host components and MEs
- Use a Structured Overlay Network (SON) for communication
- Provide overlay services
  - Resource Discovery
  - Initial deployment
  - Dynamic runtime reconfiguration
  - Publish/subscribe
  - DHT-based registry of identifiable entities such as components, groups, and bindings



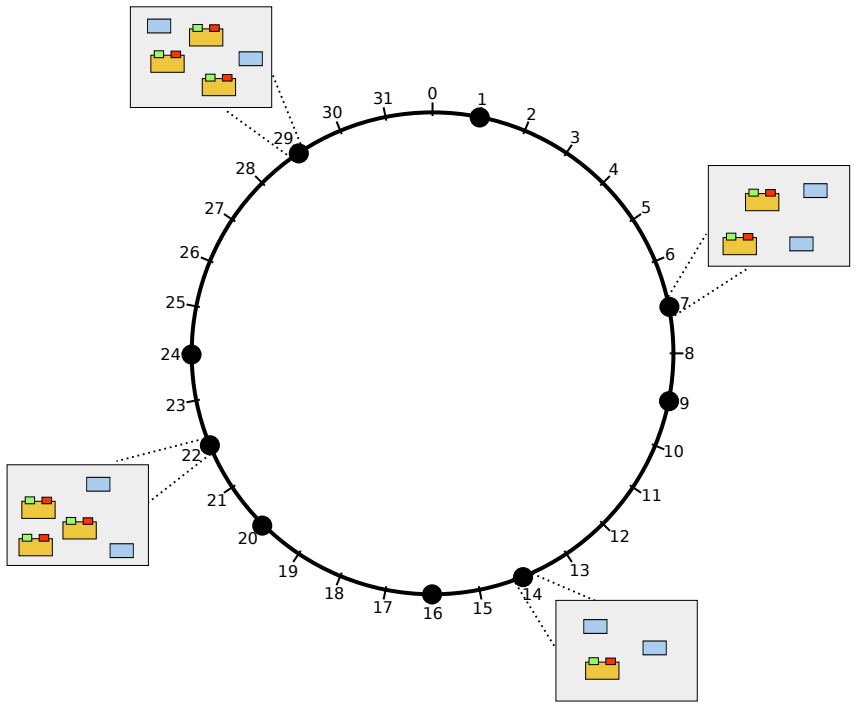
Runtime Environment

- Containers that host components and MEs
- Use a Structured Overlay Network (SON) for communication
- Provide overlay services
  - Resource Discovery
  - Initial deployment
  - Dynamic runtime reconfiguration
  - Publish/subscribe
  - DHT-based registry of identifiable entities such as components, groups, and bindings

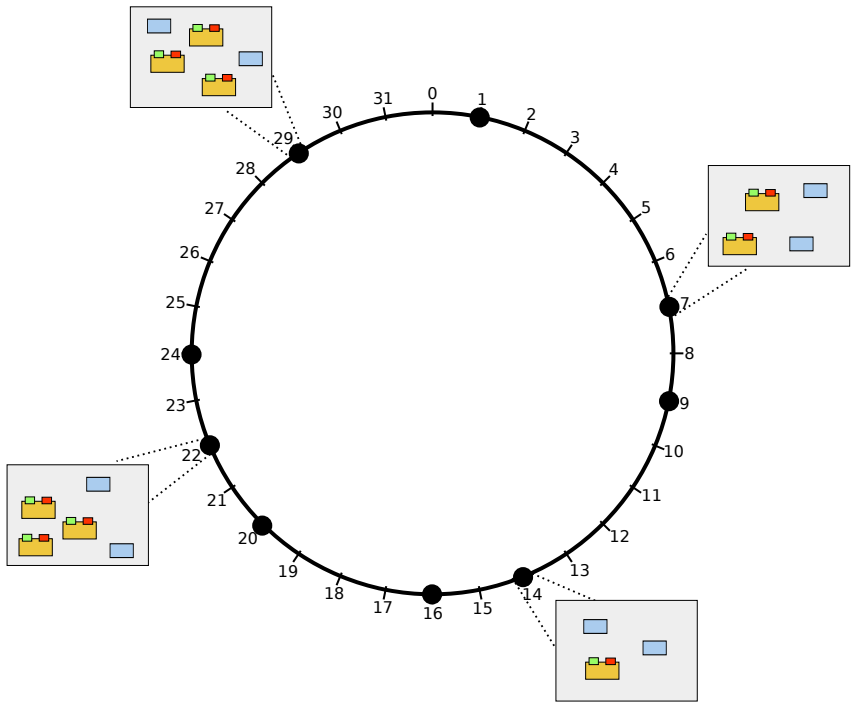
Runtime Environment

- Containers that host components and MEs
- Use a Structured Overlay Network (SON) for communication
- Provide overlay services
  - Resource Discovery
  - Initial deployment
  - Dynamic runtime reconfiguration
  - Publish/subscribe
  - DHT-based registry of identifiable entities such as components, groups, and bindings



Runtime Environment

- Containers that host components and MEs
- Use a Structured Overlay Network (SON) for communication
- Provide overlay services
  - Resource Discovery
  - Initial deployment
  - Dynamic runtime reconfiguration
  - Publish/subscribe
  - DHT-based registry of identifiable entities such as components, groups, and bindings



Outline

Distributed Management

- In distributed environments we advocate for distribution of management functions among several cooperative managers
- Multiple managers are needed for scalability, robustness, and performance and also useful for reflecting separation of concerns
- Need guidance on how to design distributed management

High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping

High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping

High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping

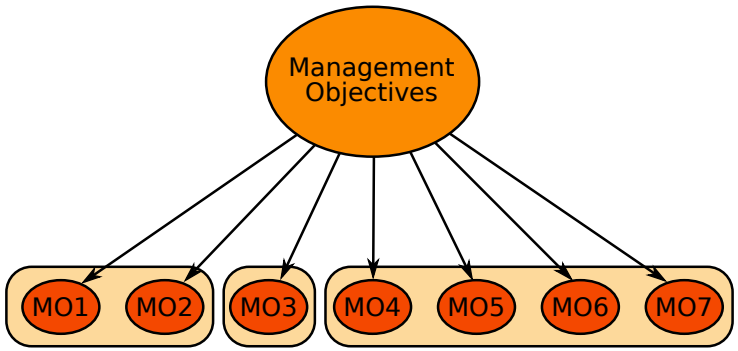
High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping



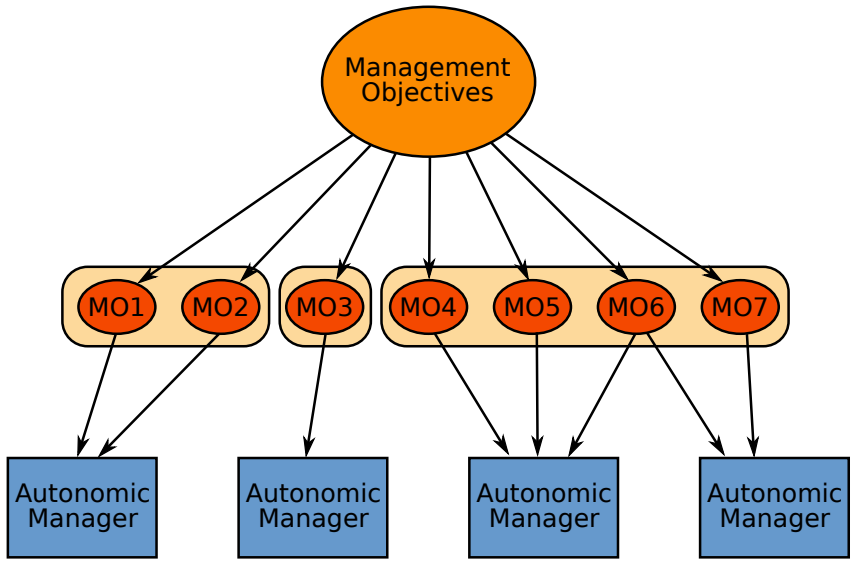
High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping



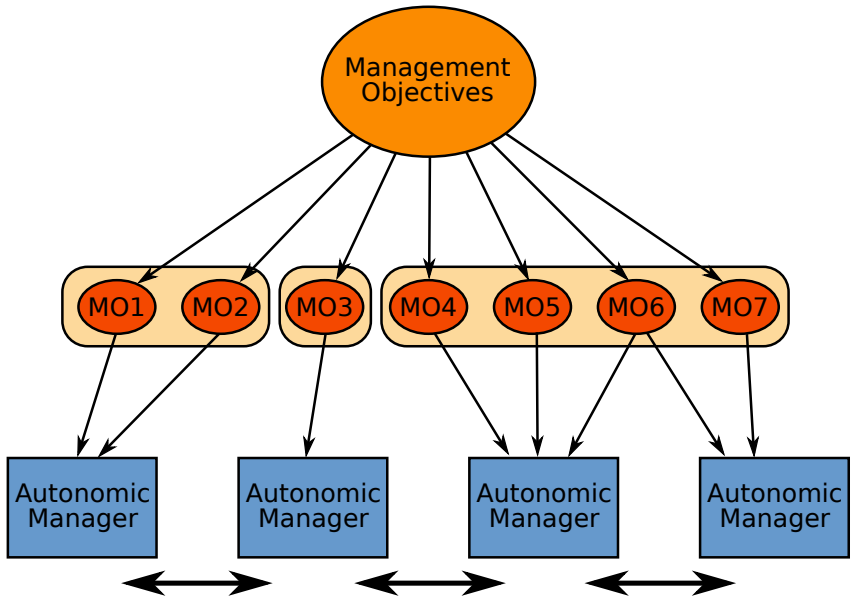
High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping



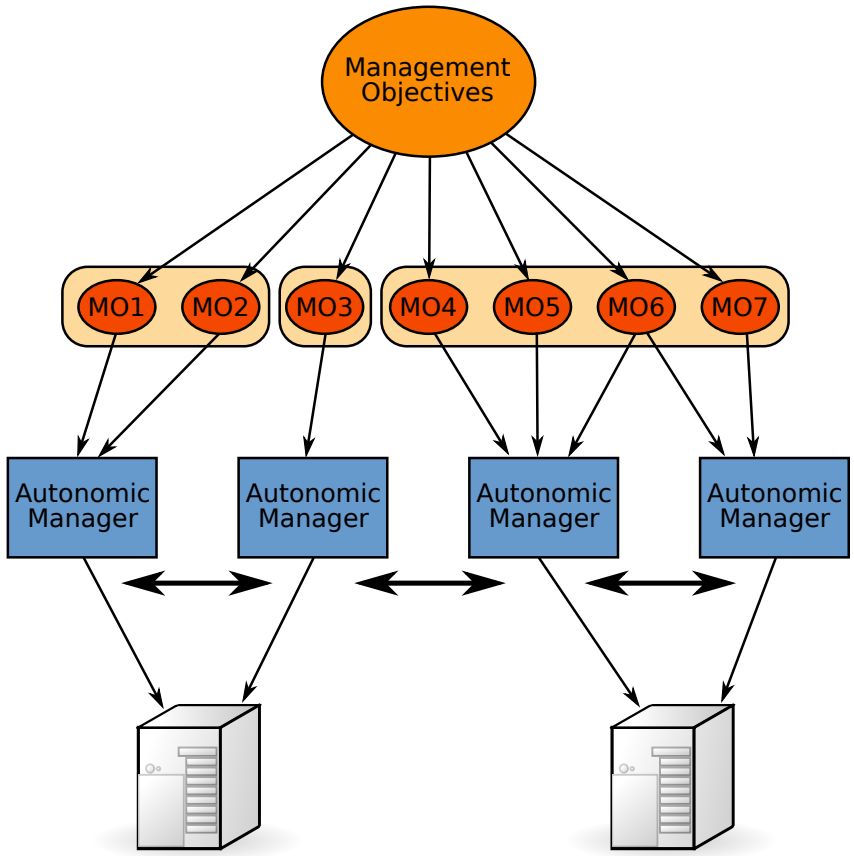
High Level Design Steps

A self-managing application

- Functional part
- Management part
- Touchpoints

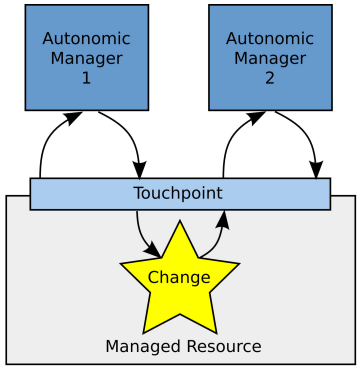
Iterative steps to distribute management

- Management objectives
- Decomposition
- Assignment
- Orchestration
- Mapping

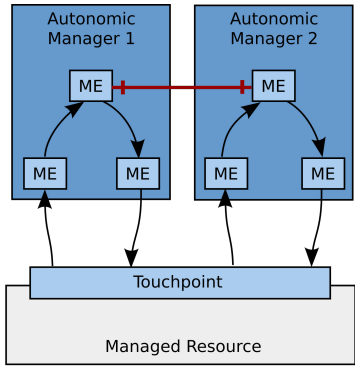


Design Space for Management Interaction

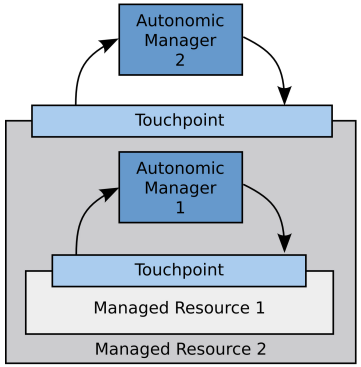
- Stigmergy
- Hierarchical
- Direct Interaction
- Sharing of MEs



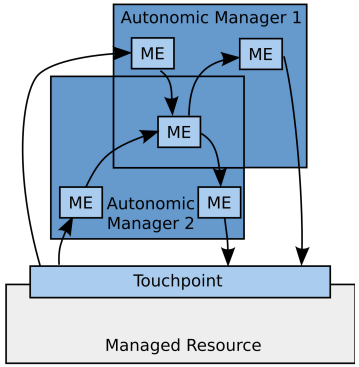
a. The stigmergy effect.



b. Direct interaction.



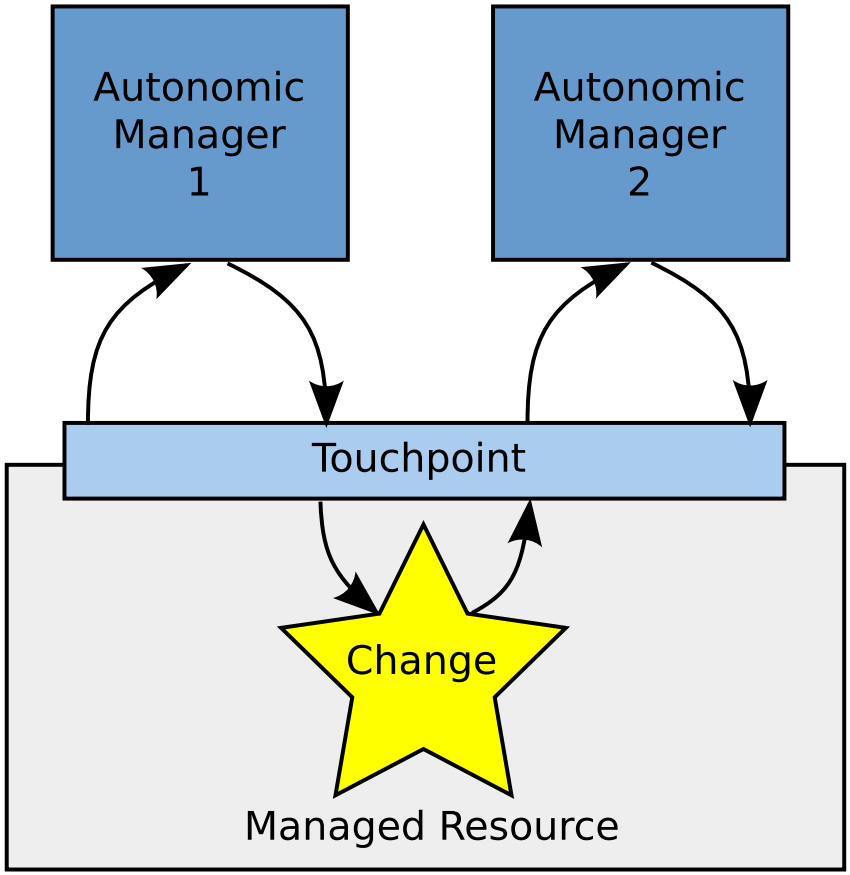
c. Hierarchical management.



d. Shared Management Elements.

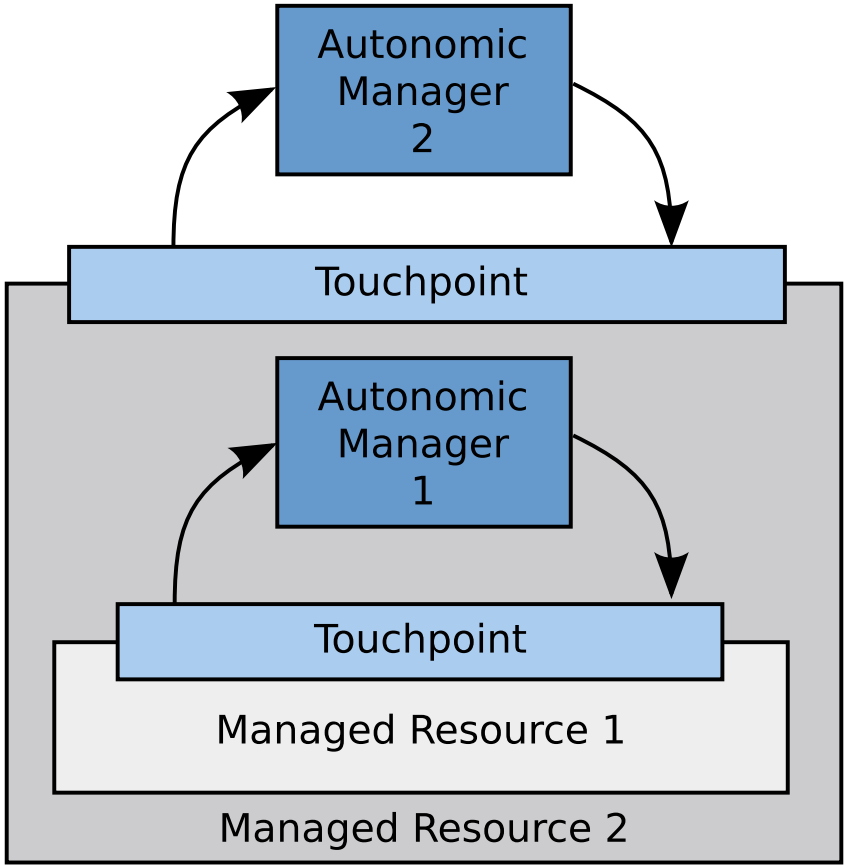
Design Space for Management Interaction

- Stigmergy
- Hierarchical
- Direct Interaction
- Sharing of MEs



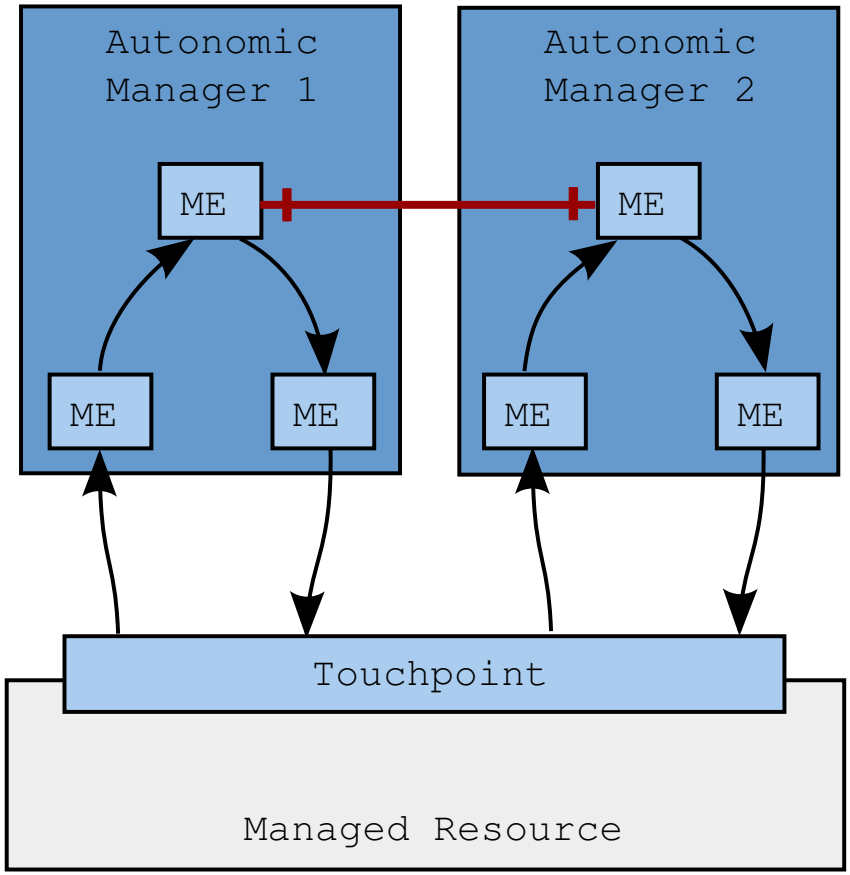
Design Space for Management Interaction

- Stigmergy
- Hierarchical
- Direct Interaction
- Sharing of MEs



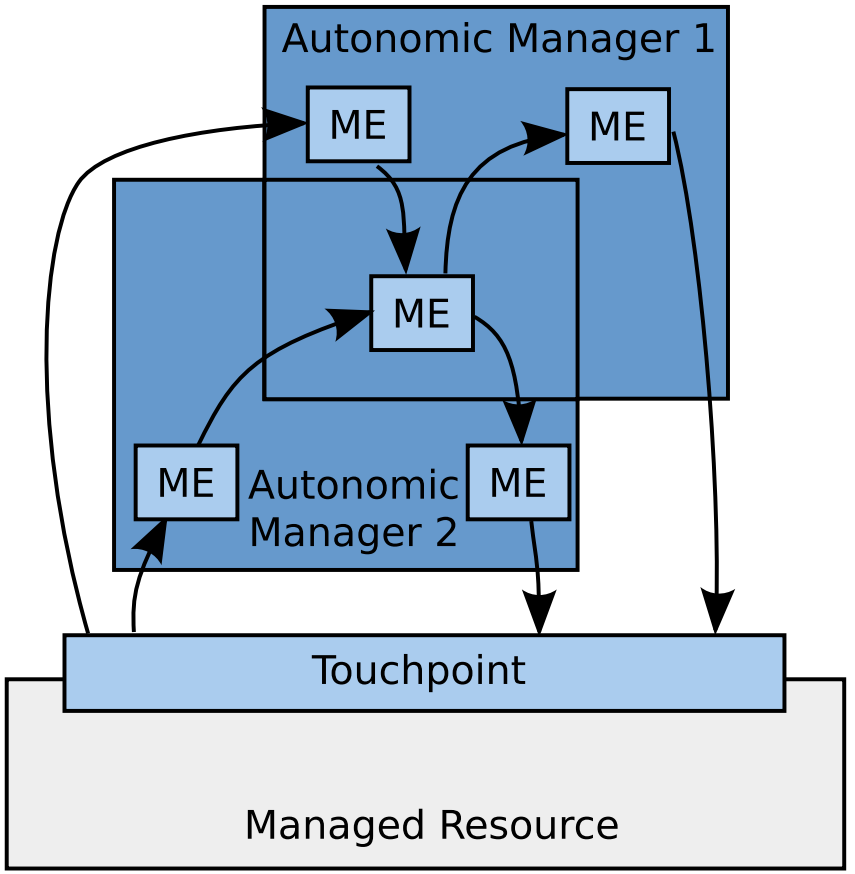
Design Space for Management Interaction

- Stigmergy
- Hierarchical
- Direct Interaction
- Sharing of MEs



Design Space for Management Interaction

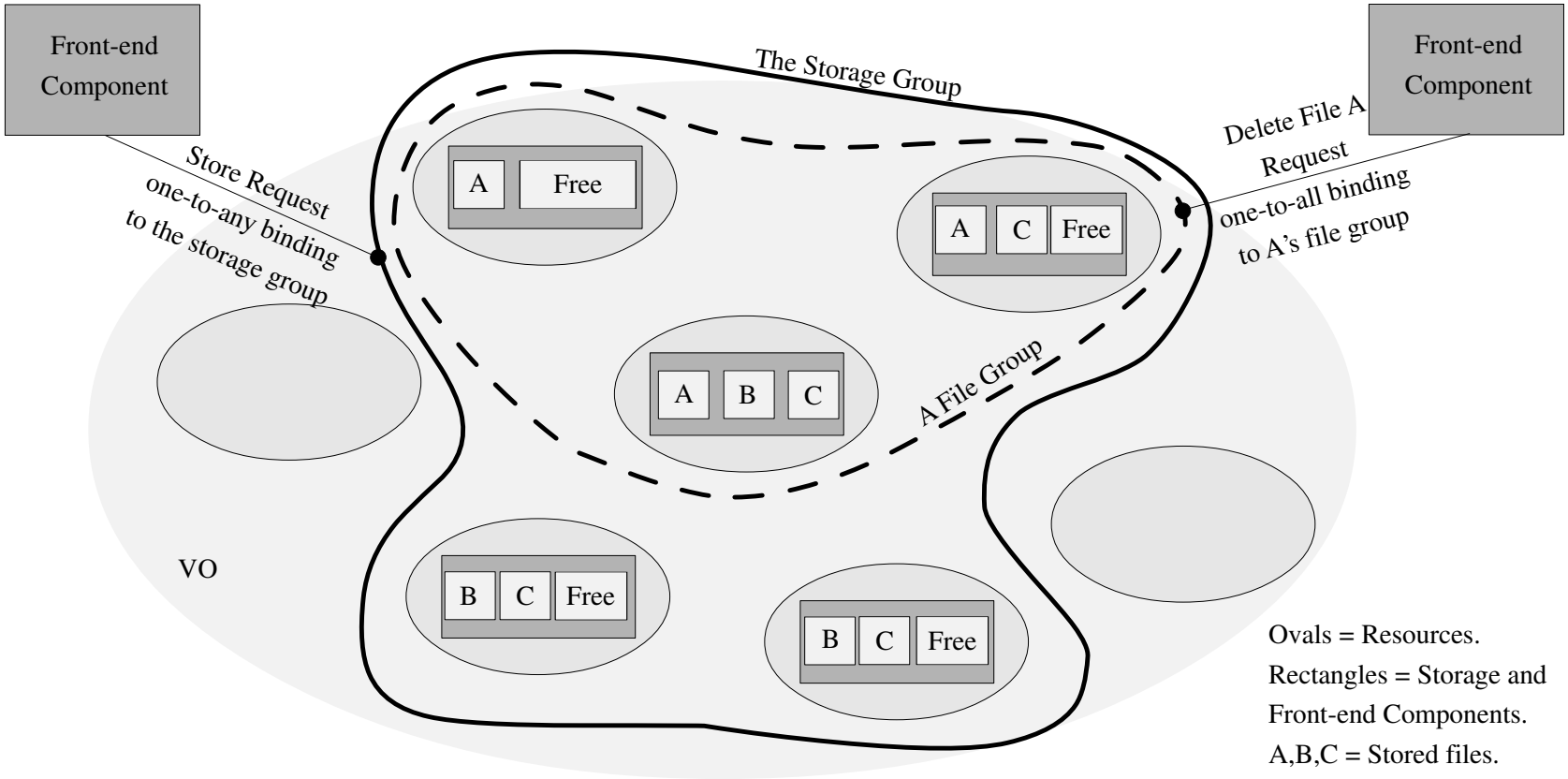
- Stigmergy
- Hierarchical
- Direct Interaction
- Sharing of MEs



Use Case: YASS

- YASS: Yet Another Storage Service
- Users can store, read and delete files on a set of distributed resources.
- Transparently replicates files for robustness and scalability.
- Deployed in a dynamic distributed environment

YASS functional part

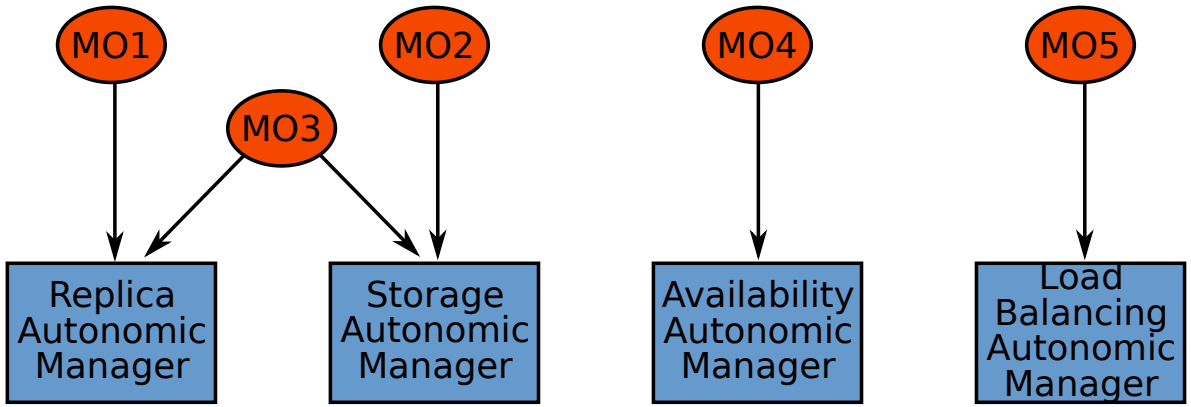


YASS Management Objective

- **MO1:** Maintain file replication degree
- **MO2:** Maintain total storage space and total free space
- **MO3:** Release unused storage
- **MO4:** Increasing availability of popular files
- **MO5:** Balance stored files among allocated resources

YASS Management Objective

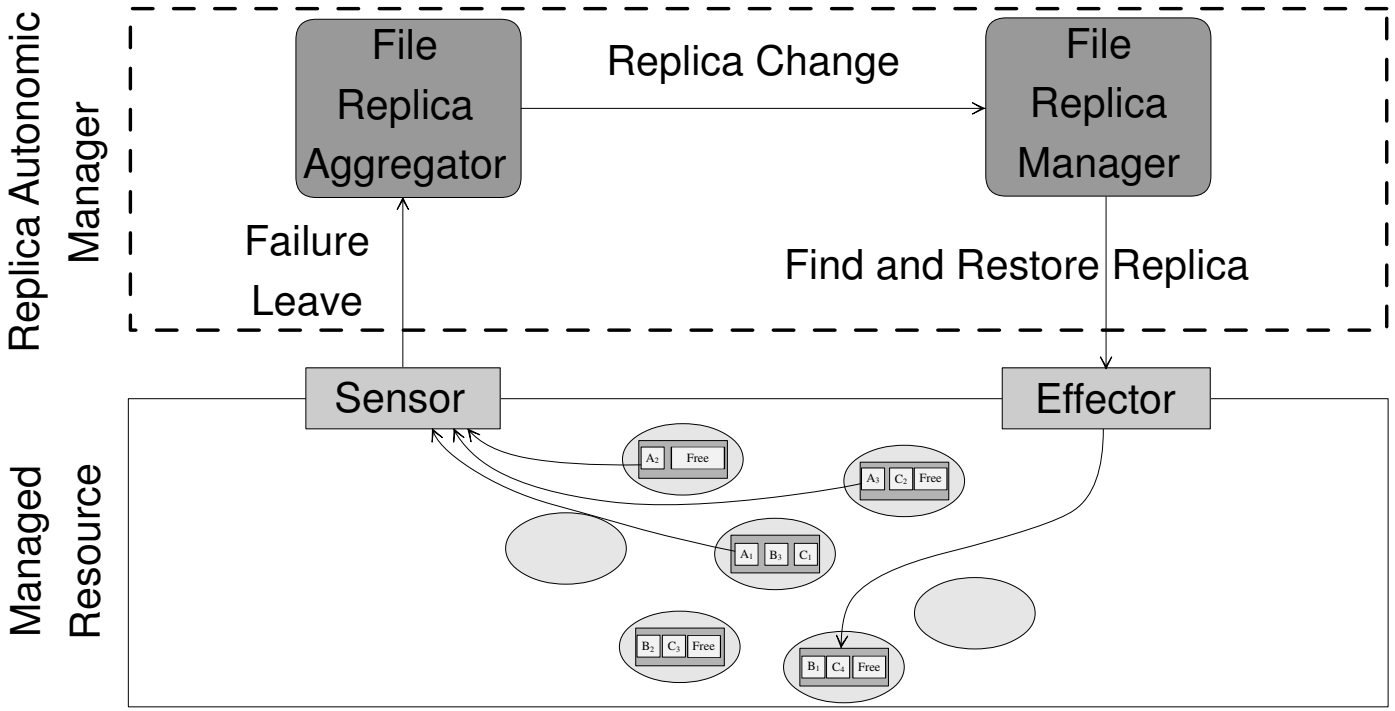
- **MO1:** Maintain file replication degree
- **MO2:** Maintain total storage space and total free space
- **MO3:** Release unused storage
- **MO4:** Increasing availability of popular files
- **MO5:** Balance stored files among allocated resources



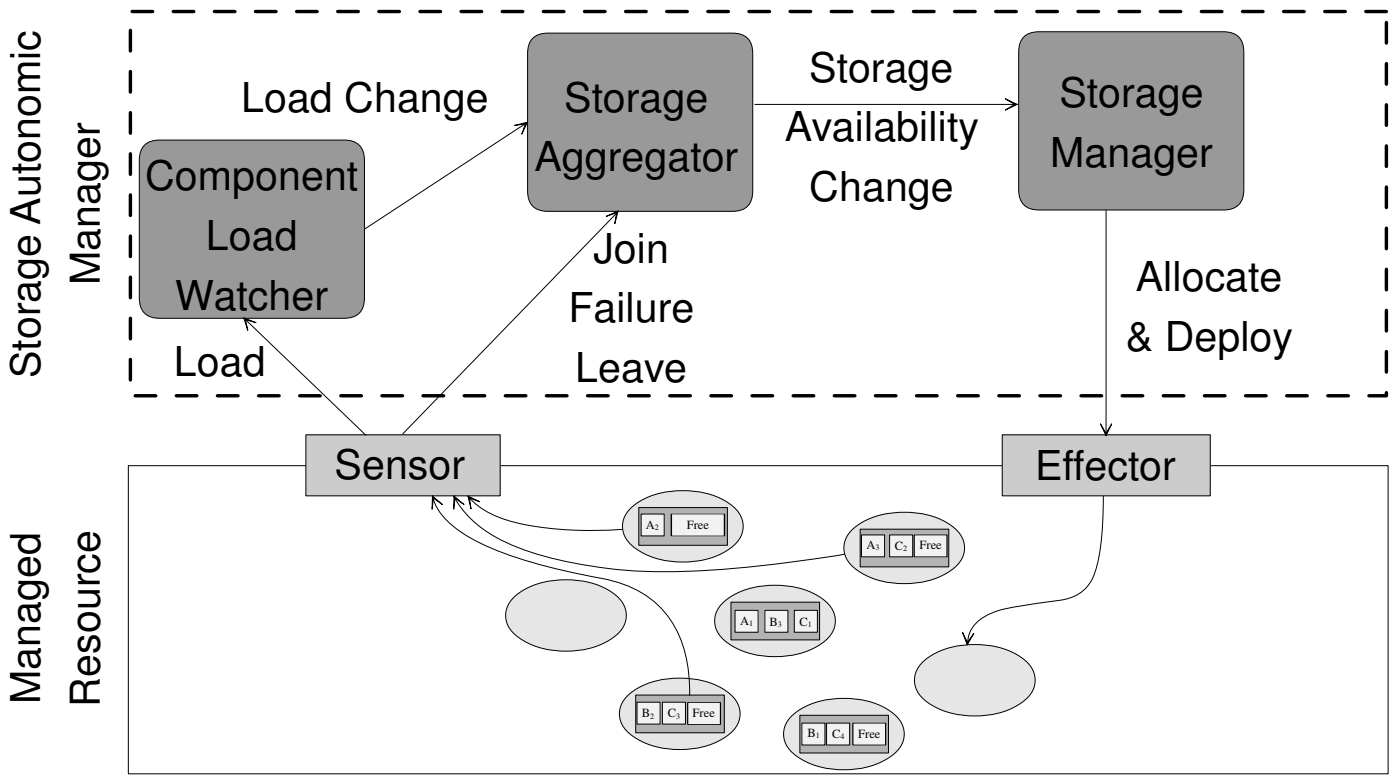
Touchpoints

- Load sensor to measure the current free space
- Access frequency sensor to detect popular files
- Replicate file actuator to add one extra replica of a file
- Move file actuator to move files for load balancing

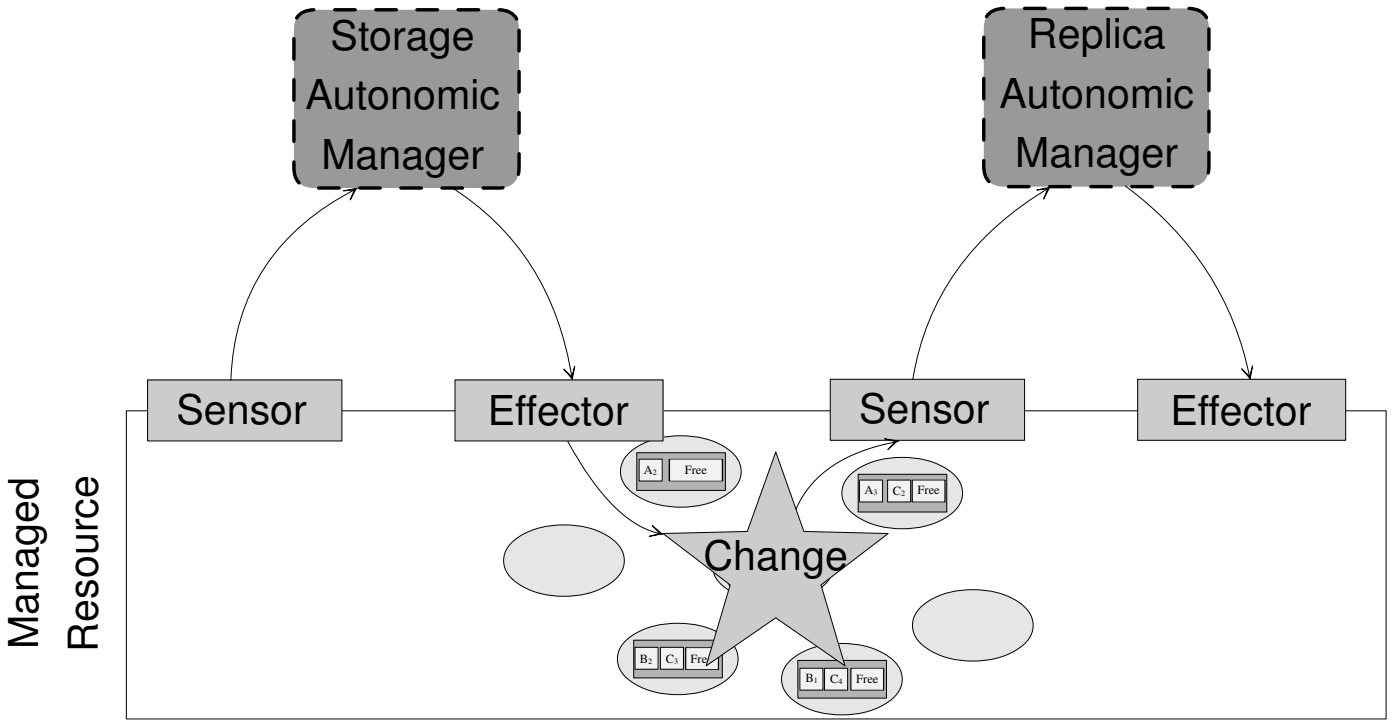
MO1: Maintain the File Replication Degree



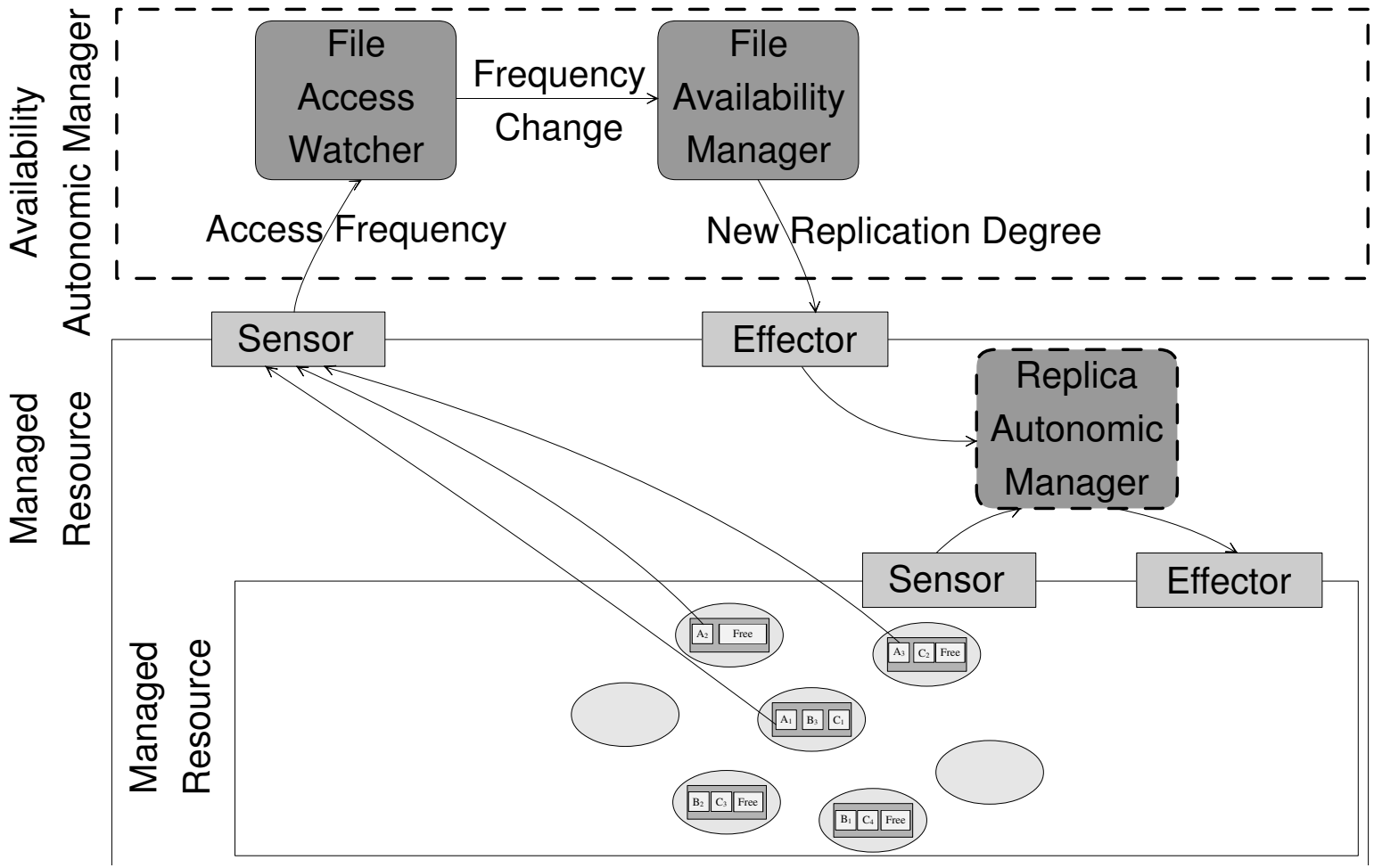
MO2: Maintain the Total Storage Space and Total Free Space



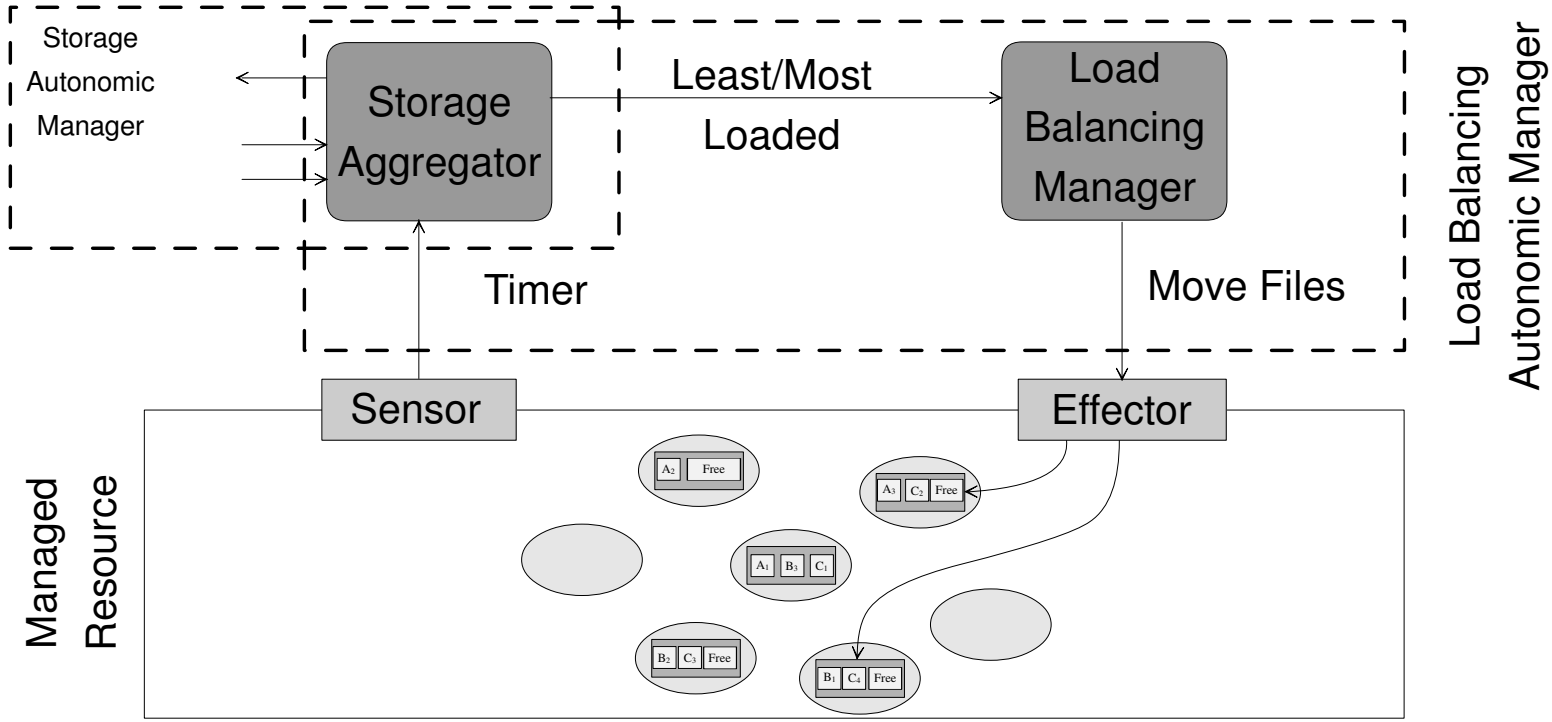
MO3: Release Unused Storage



MO4: Increasing the Availability of Popular Files



MO5: Balance the Stored Files Among the Allocated Resources



Outline

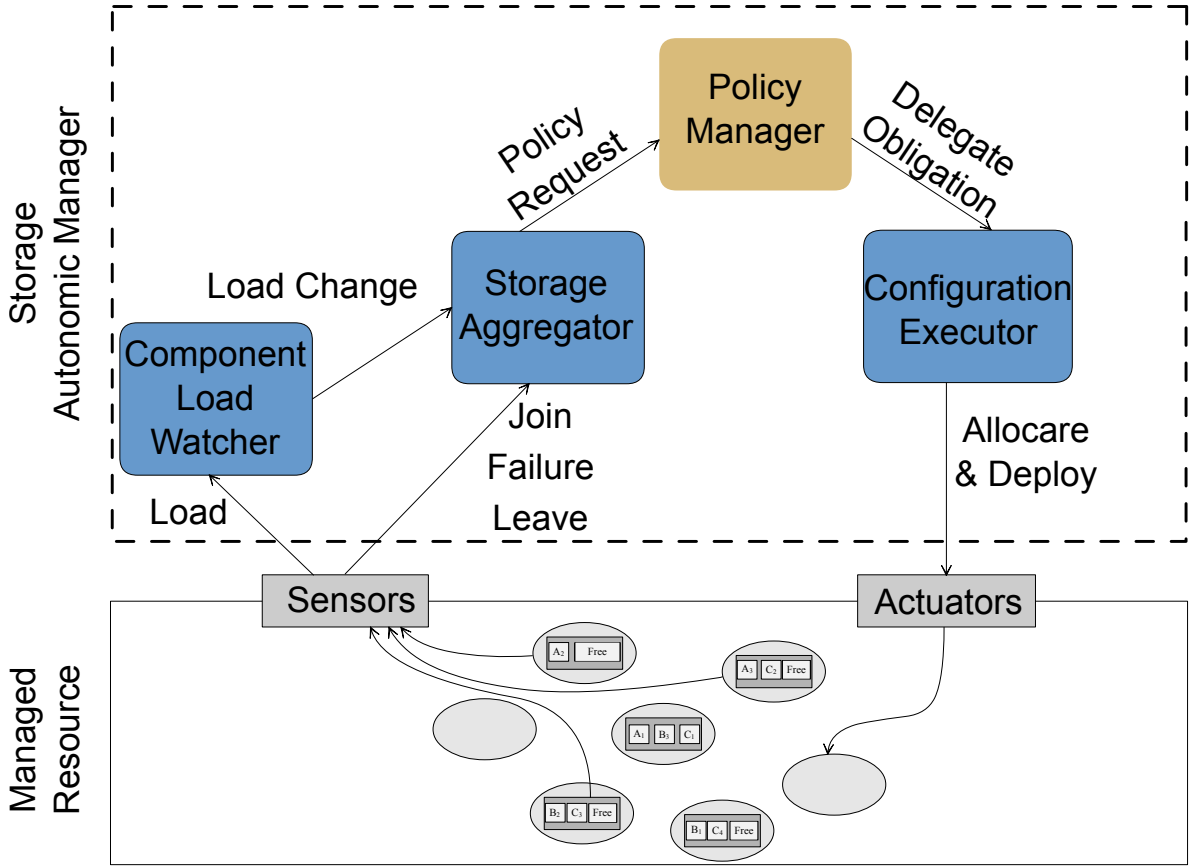
Policy-based Management

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

Drawbacks of "Hard-coded" Policy

- Application developer has to be involved in policy implementation
- Hard to trace policies
  - Policies are "hard-coded" (embedded) in the management code of a distributed system
  - Policy logic is scattered in implementation
- Change of policies may requires rebuilding and redeploying of the application (or at least its management part)

Example: YASS Self-Configuration Using Policies



Policy Languages (used in this work)

- SPL
  - Simplified Policy Language
  - Designed for management
  - ▸ SPL example
- XACML
  - eXtensible Access Control Markup Language
  - Primarily designed for access control
  - ▸ XACML example

Performance Evaluation

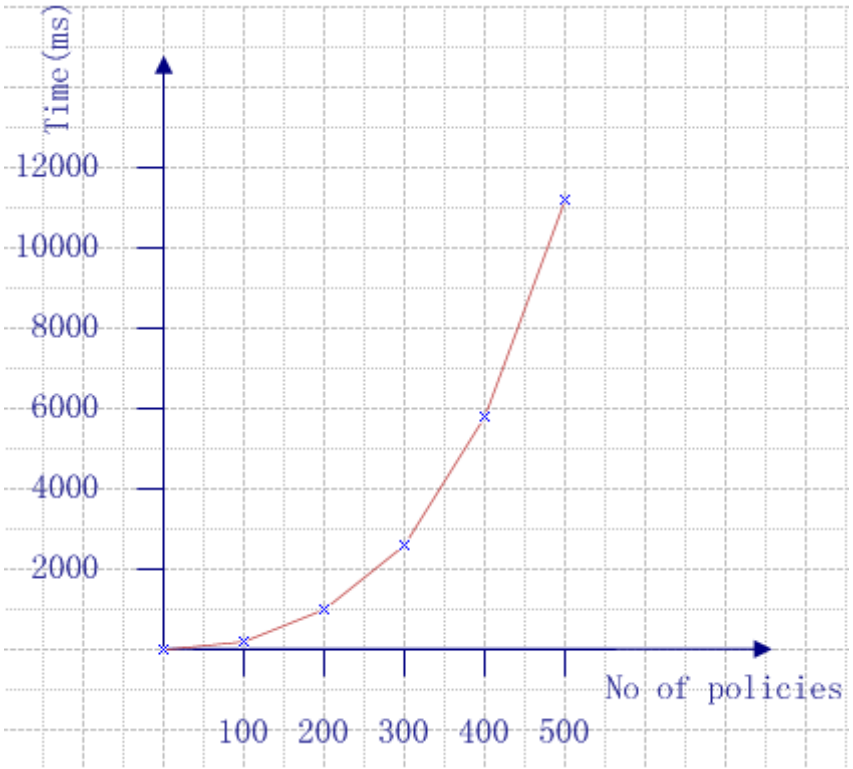


Figure: SPL

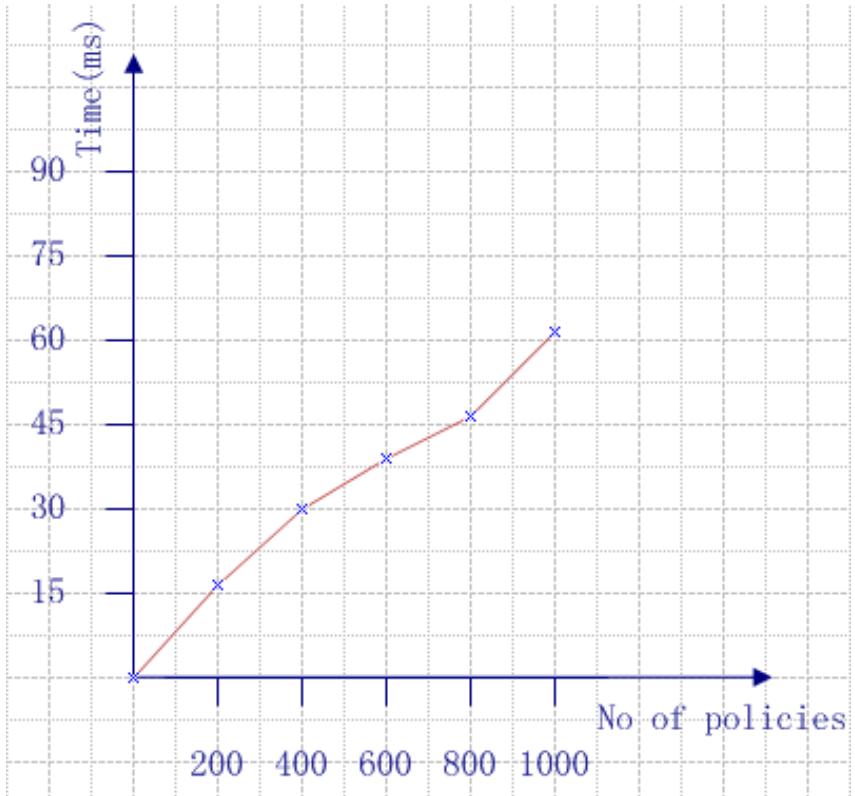
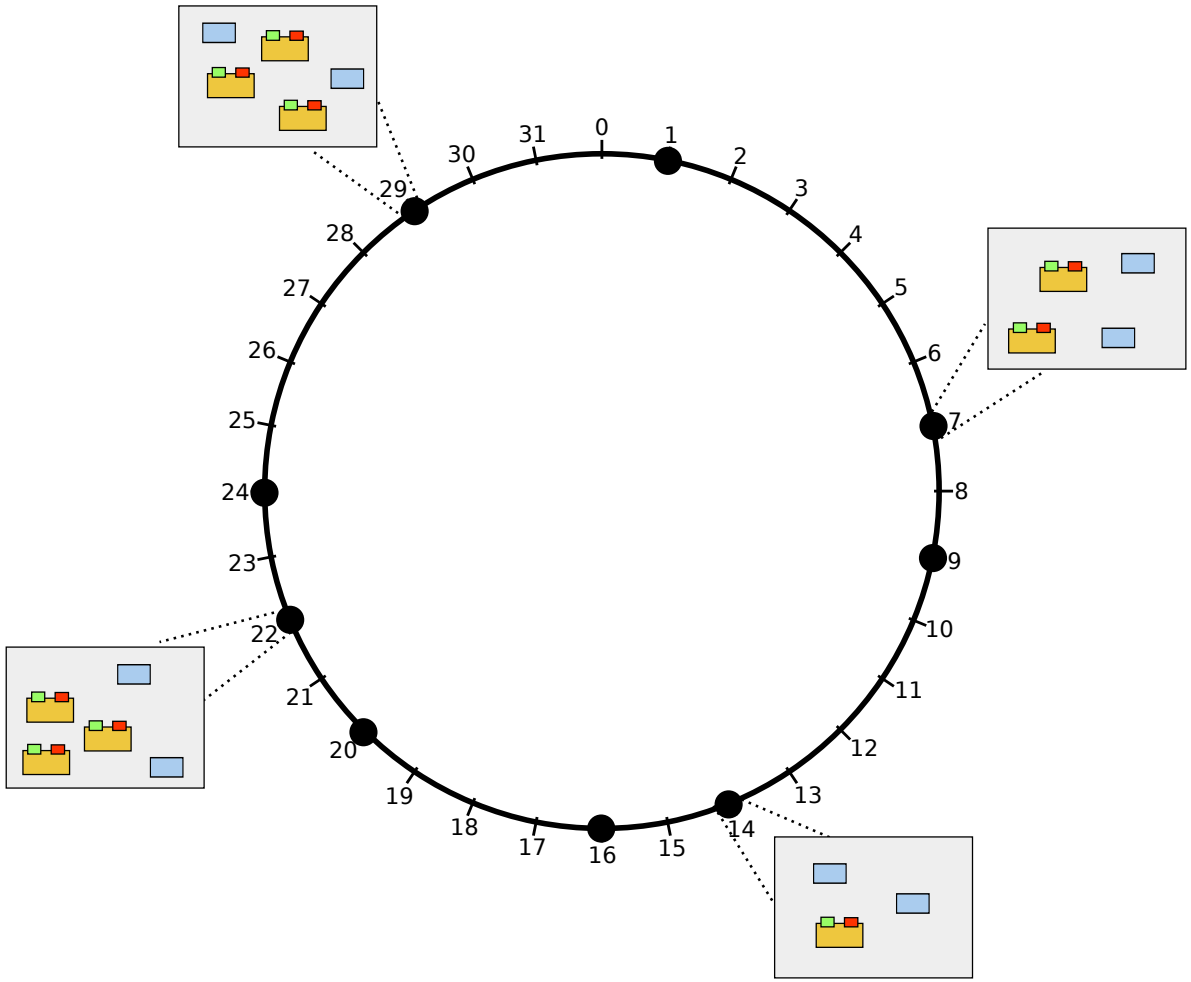


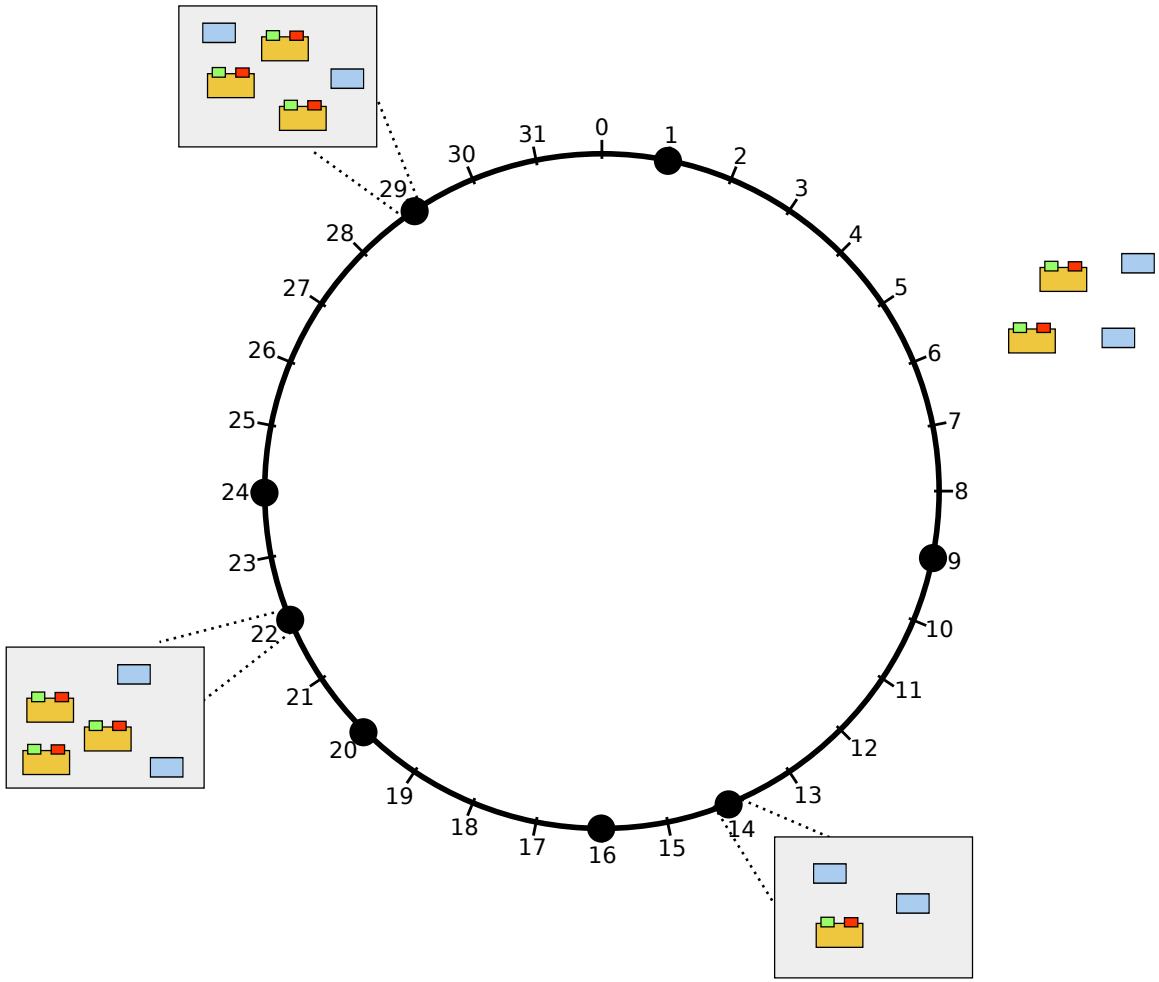
Figure: XACML

Outline

Robust Management Elements



Robust Management Elements



Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Robust Management Elements

A Robust Management Element (RME) should:

- Be replicated to ensure fault-tolerance
- Survive continuous resource failures by automatically restoring failed replicas on other nodes
- Maintain its state consistent among replicas
- Provide its service with minimal disruption in spite of resource join/leave/fail (high availability)
- Be location transparent (i.e. clients of the RME should be able to communicate with it regardless of its current location)

Solution Outline

- Finite state machine replication
- SMART algorithm for changing replica set (migration)
- Our decentralized algorithm to automate the process

End Result

A Robust Management Element (RME) that can be used to build robust management!

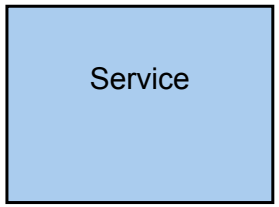
Solution Outline

- Finite state machine replication
- SMART algorithm for changing replica set (migration)
- Our decentralized algorithm to automate the process

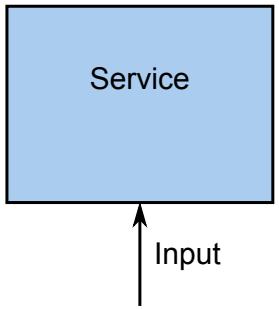
End Result

A Robust Management Element (RME) that can be used to build robust management!

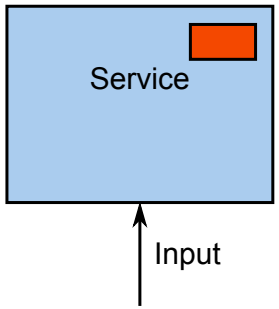
Replicated State Machine



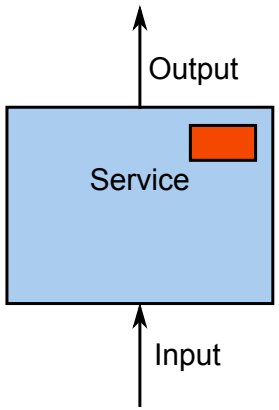
Replicated State Machine



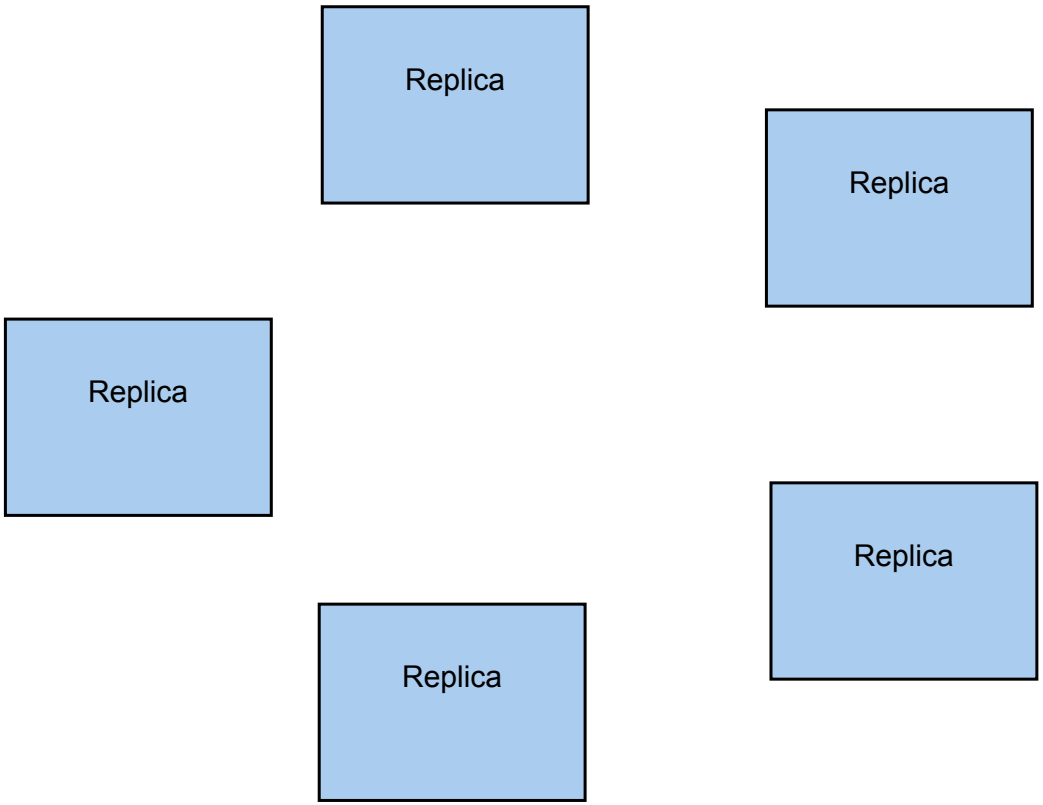
Replicated State Machine



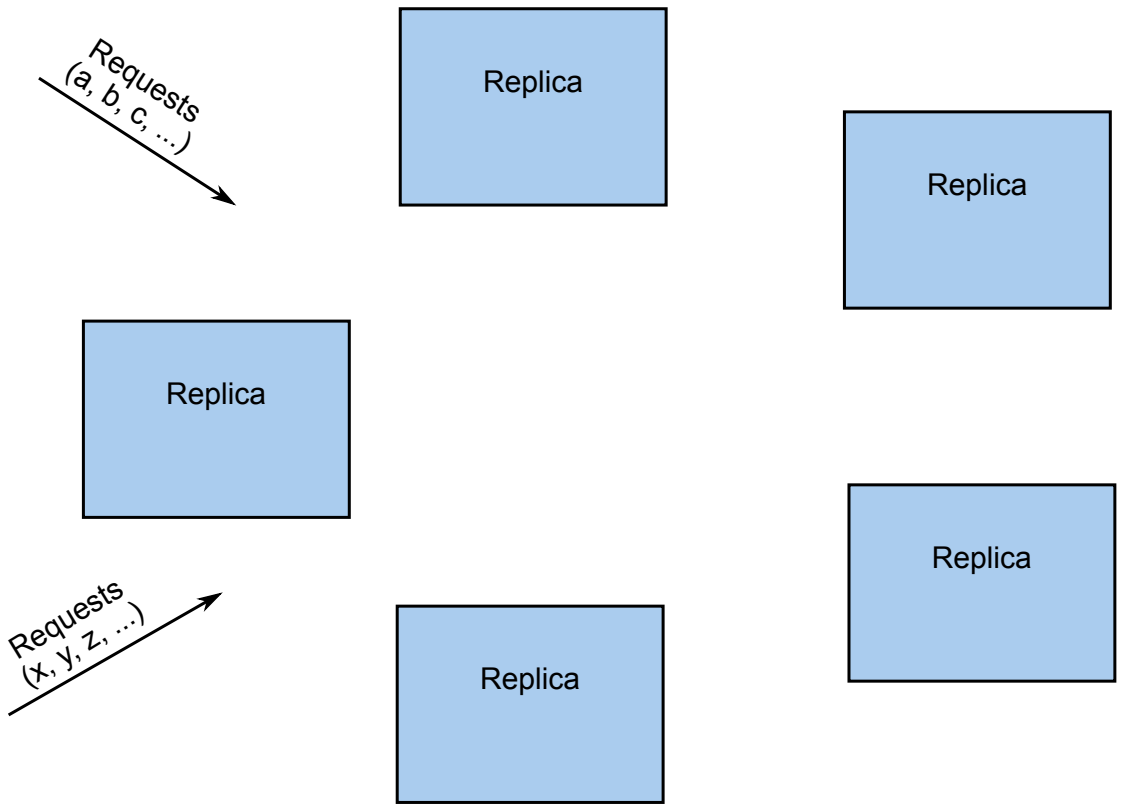
Replicated State Machine



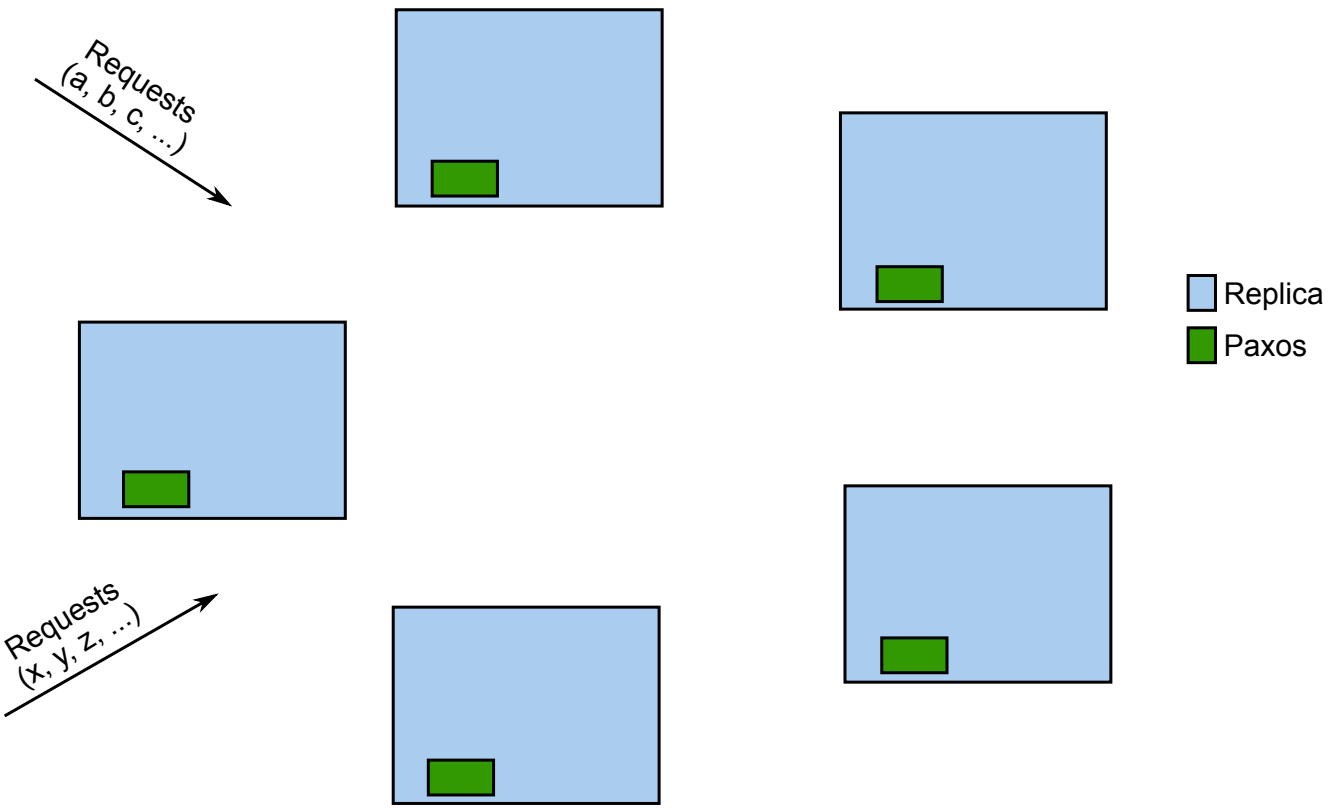
Replicated State Machine



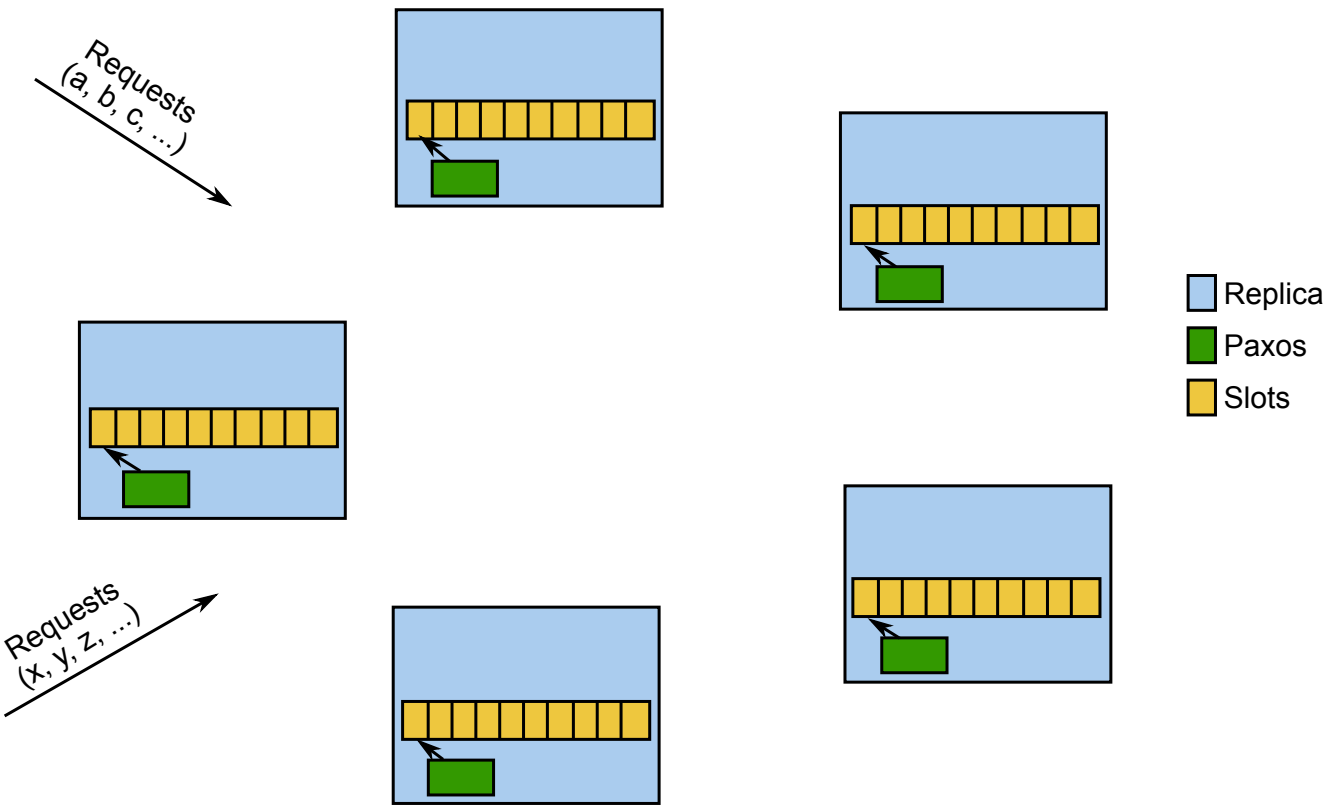
Replicated State Machine



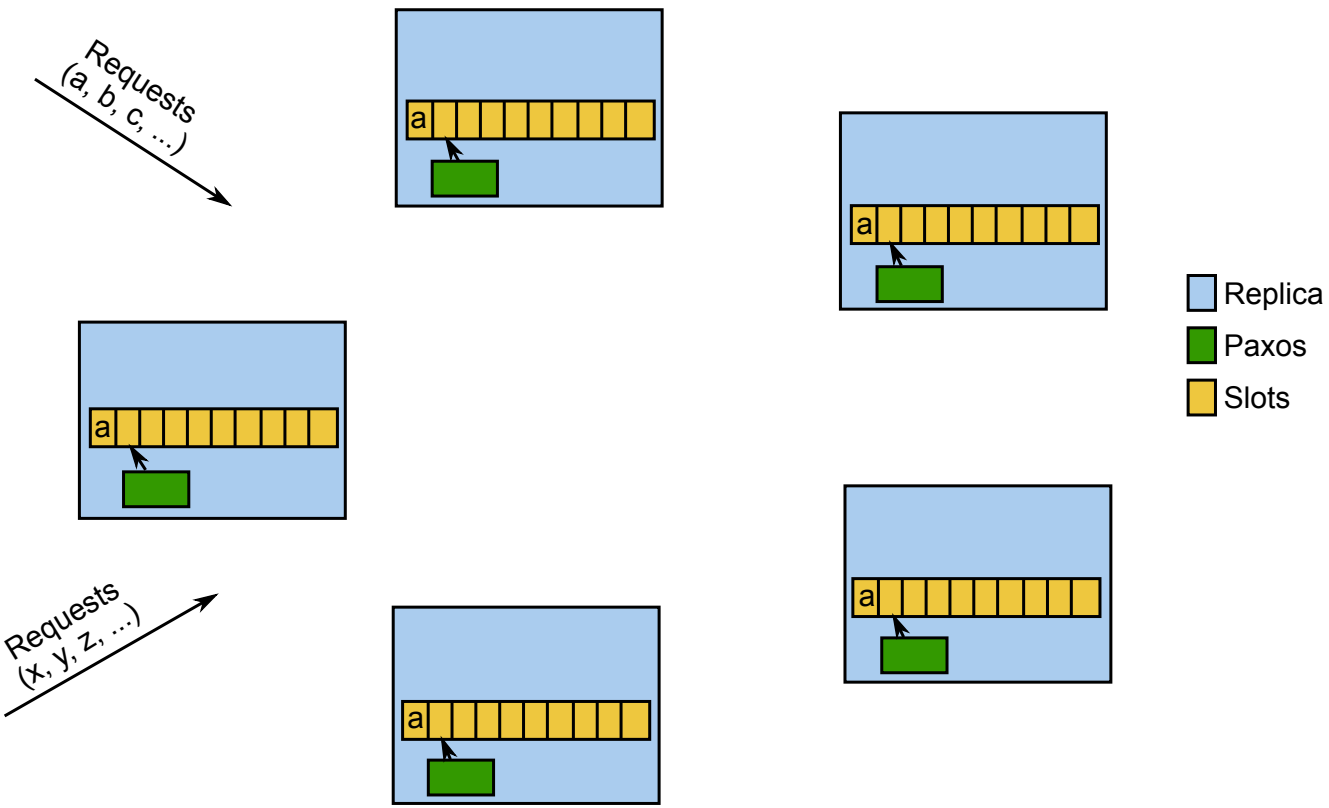
Replicated State Machine



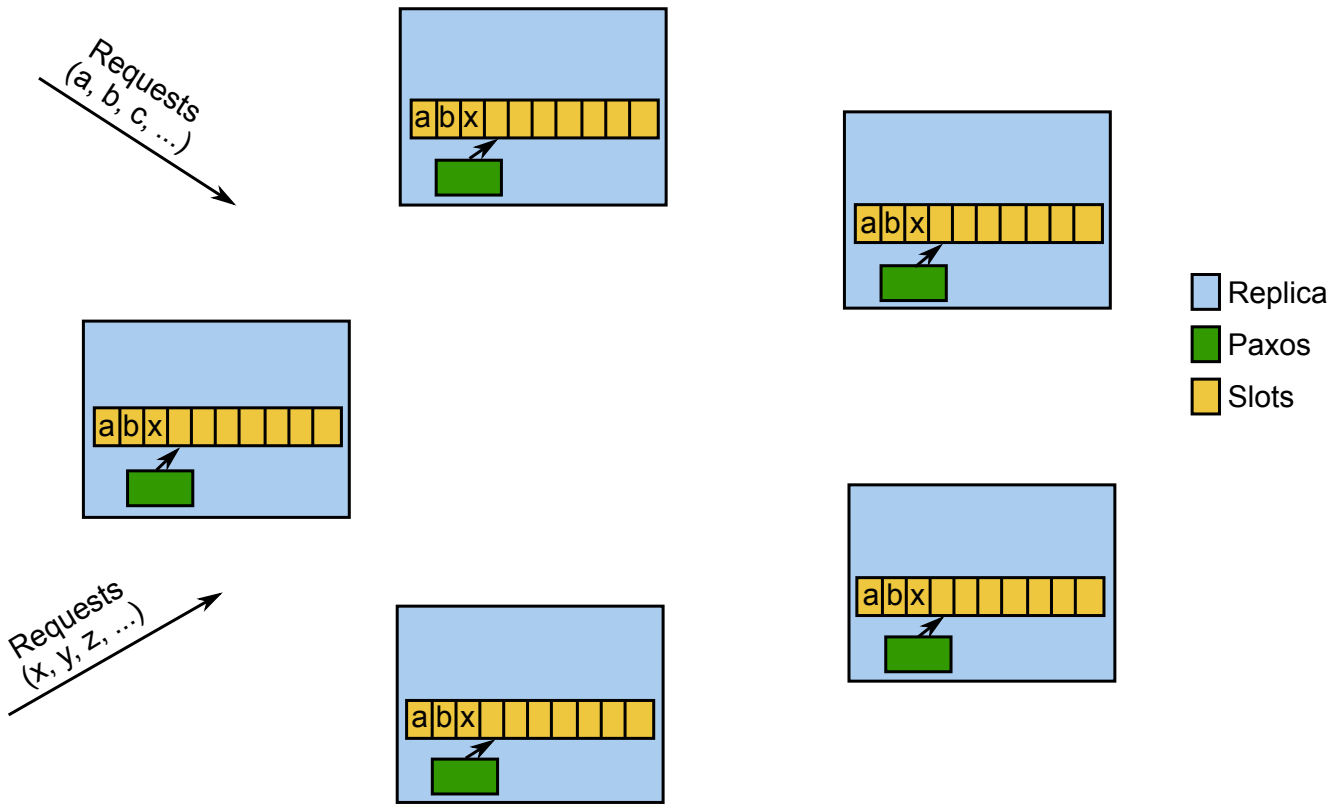
Replicated State Machine



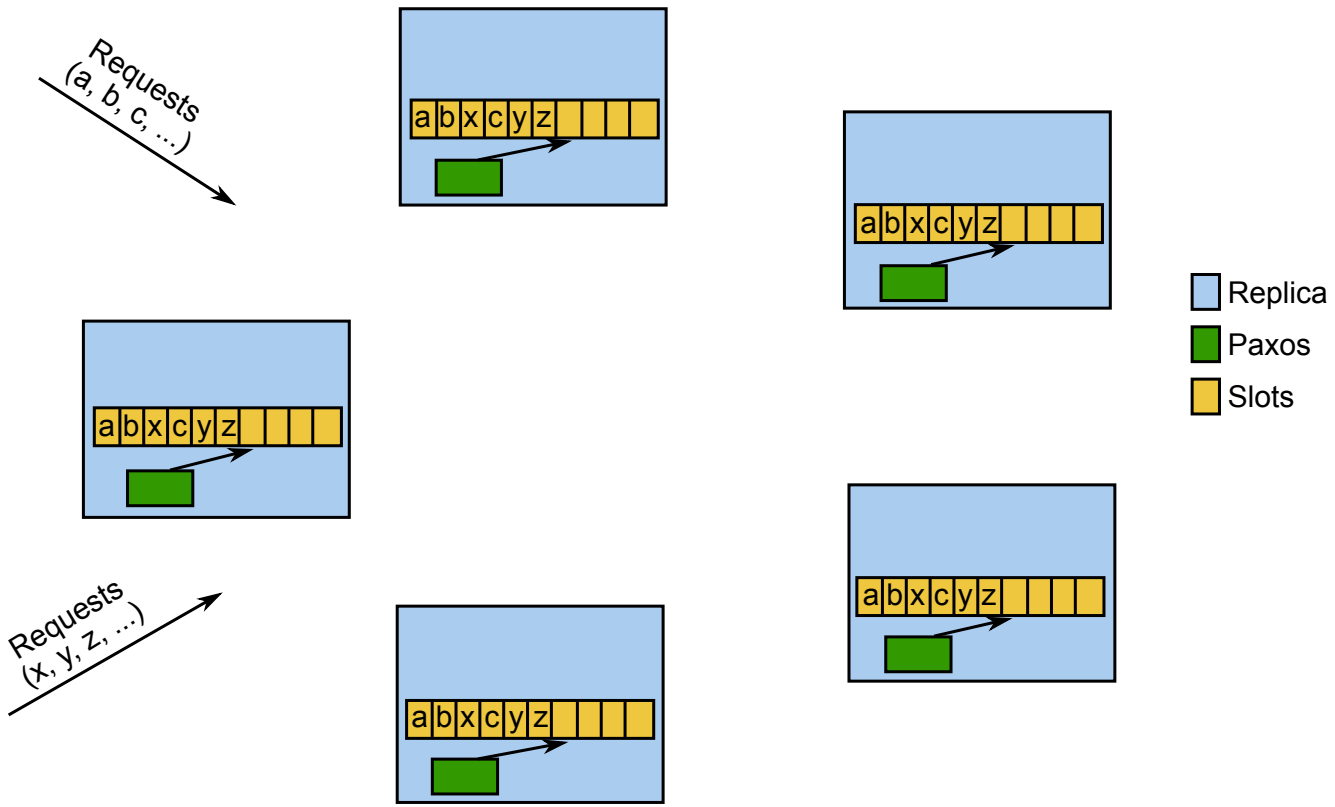
Replicated State Machine



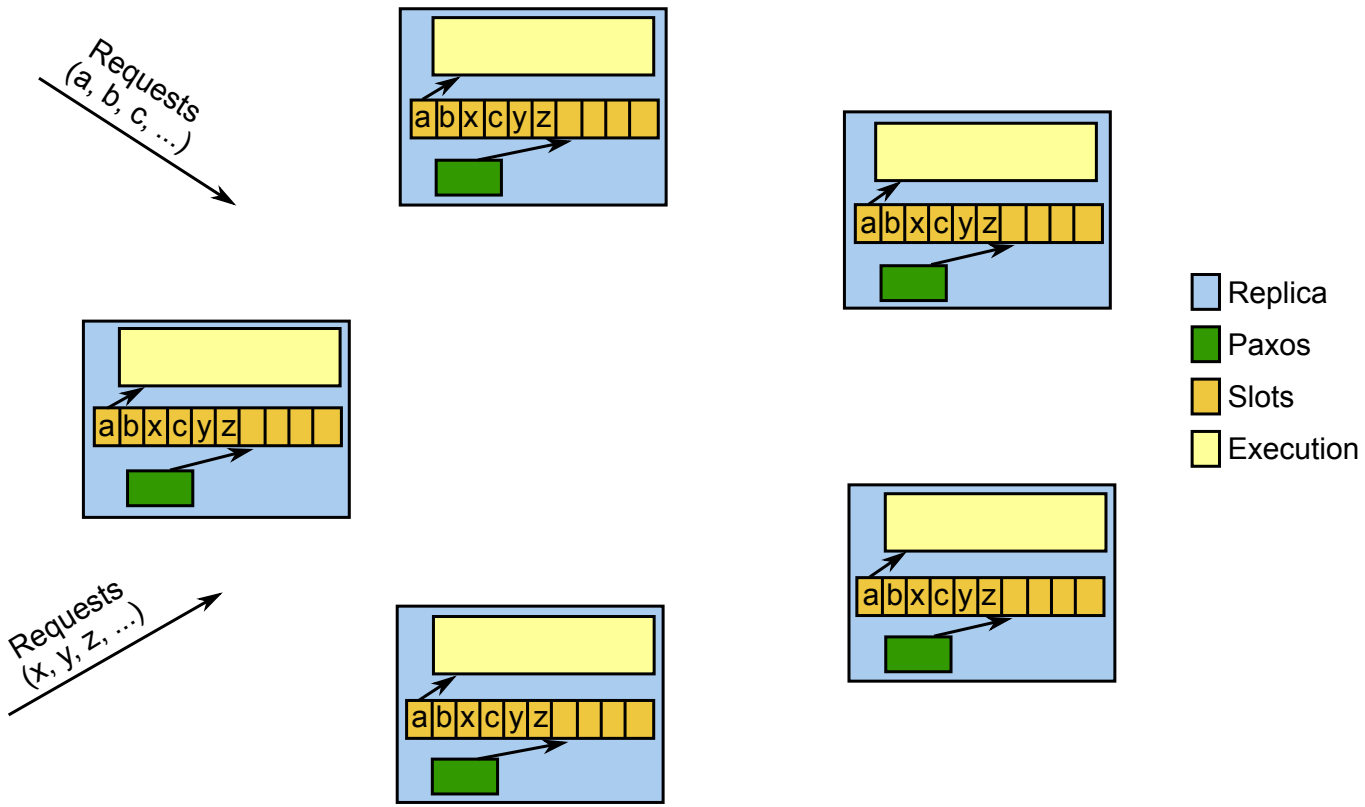
Replicated State Machine



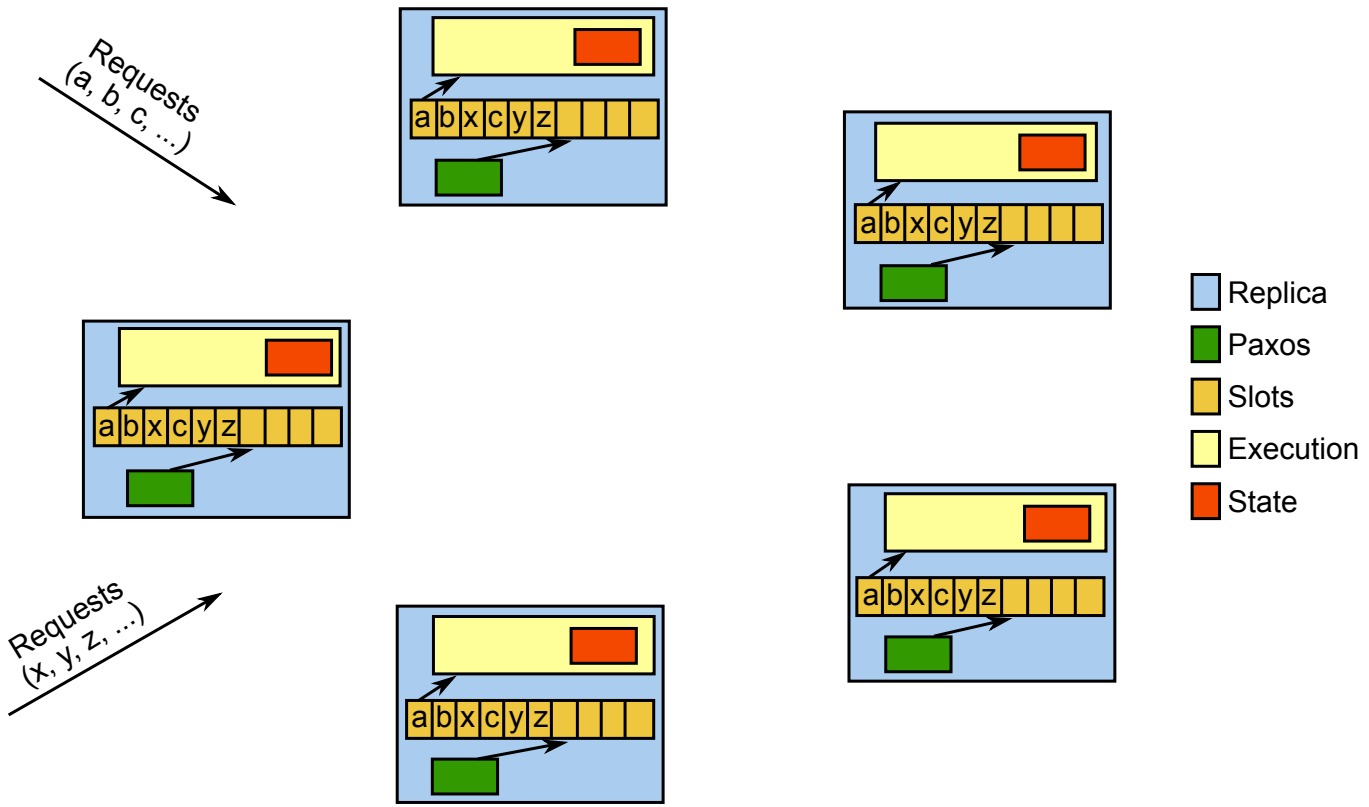
Replicated State Machine



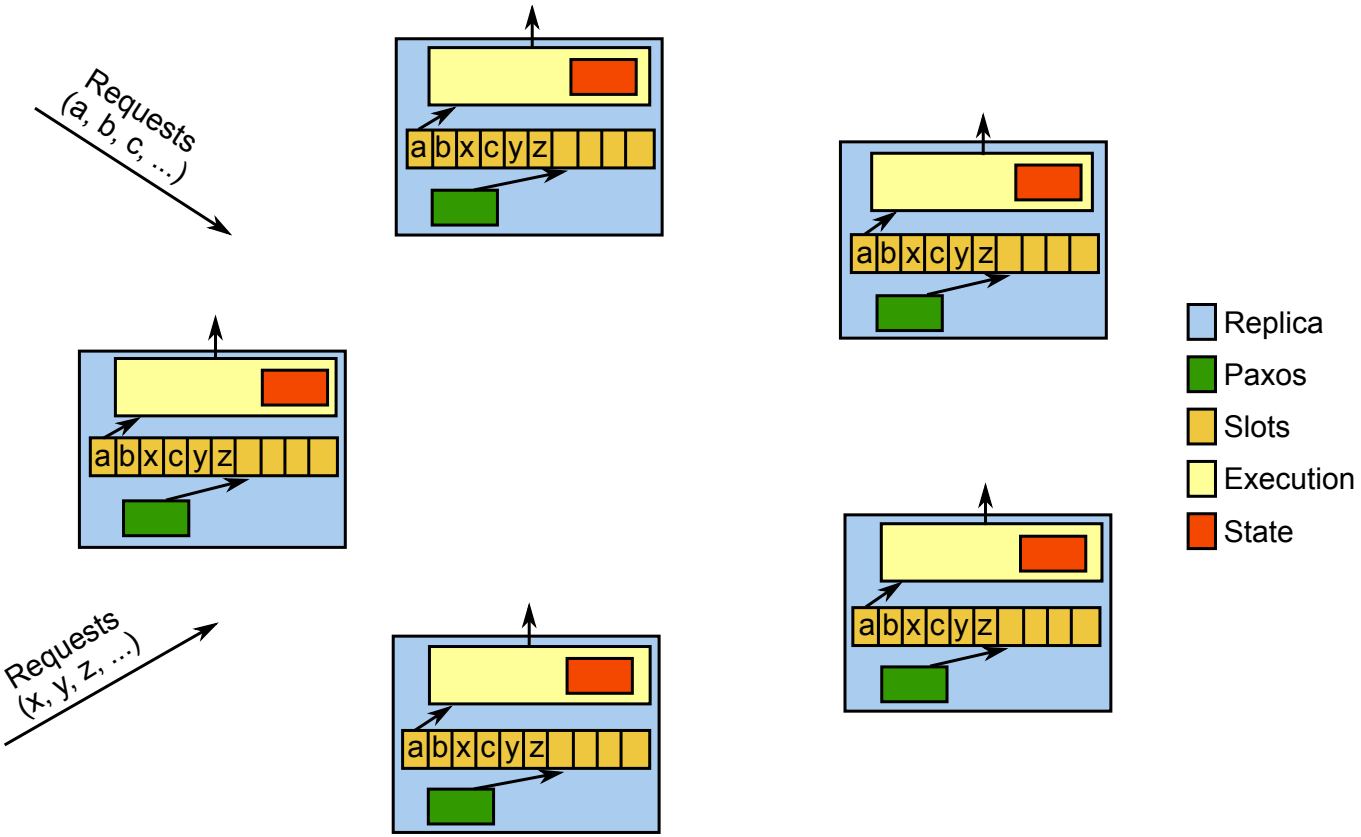
Replicated State Machine



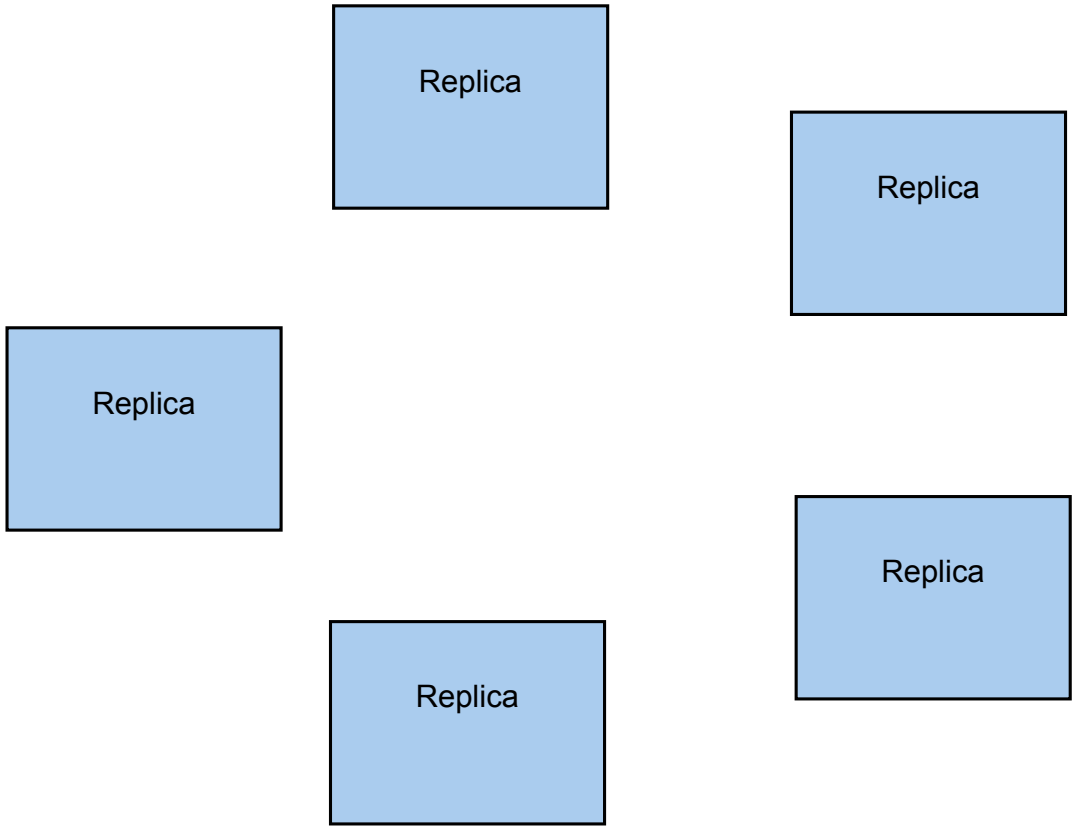
Replicated State Machine



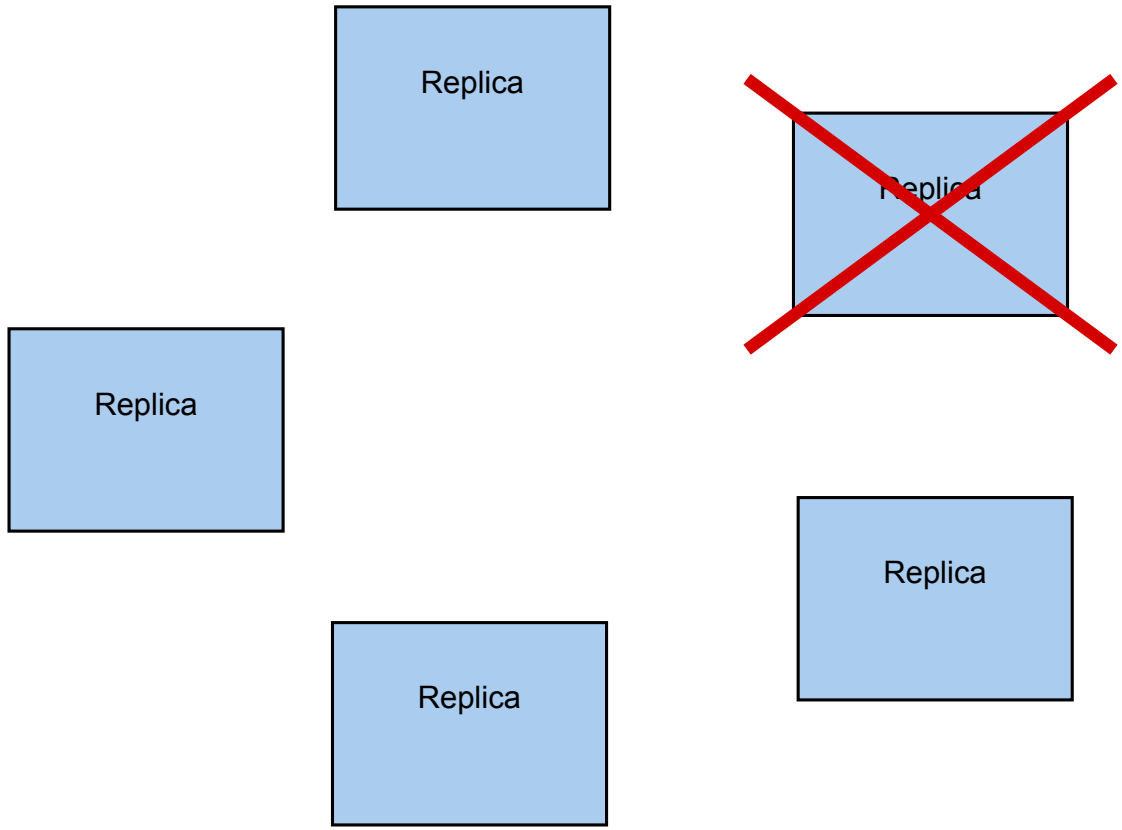
Replicated State Machine



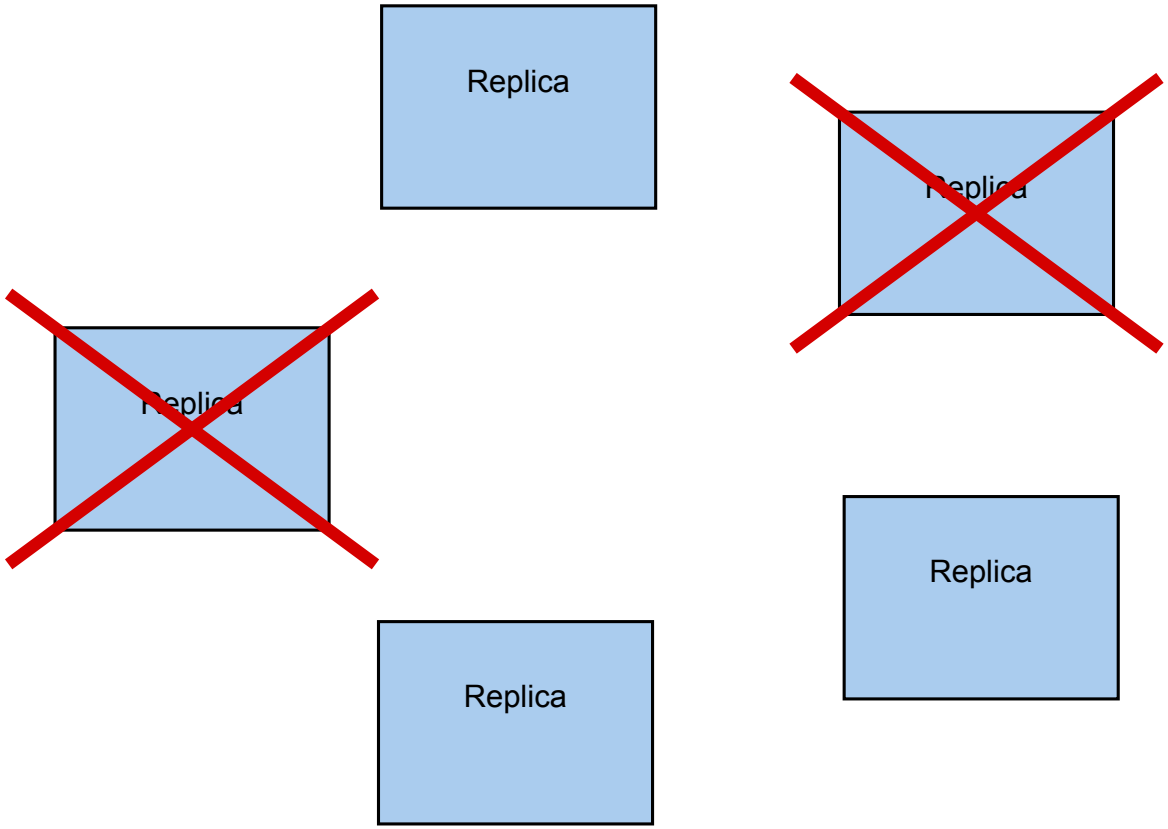
Replicated State Machine is Not Enough



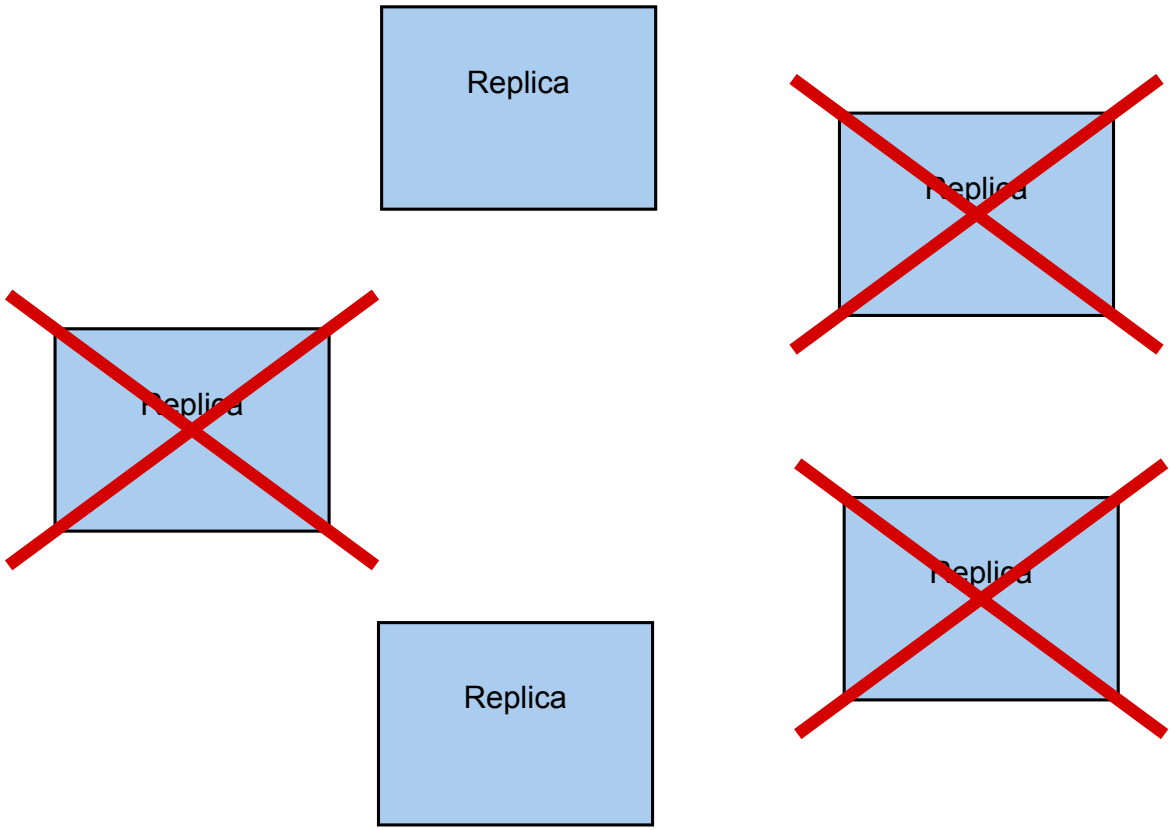
Replicated State Machine is Not Enough



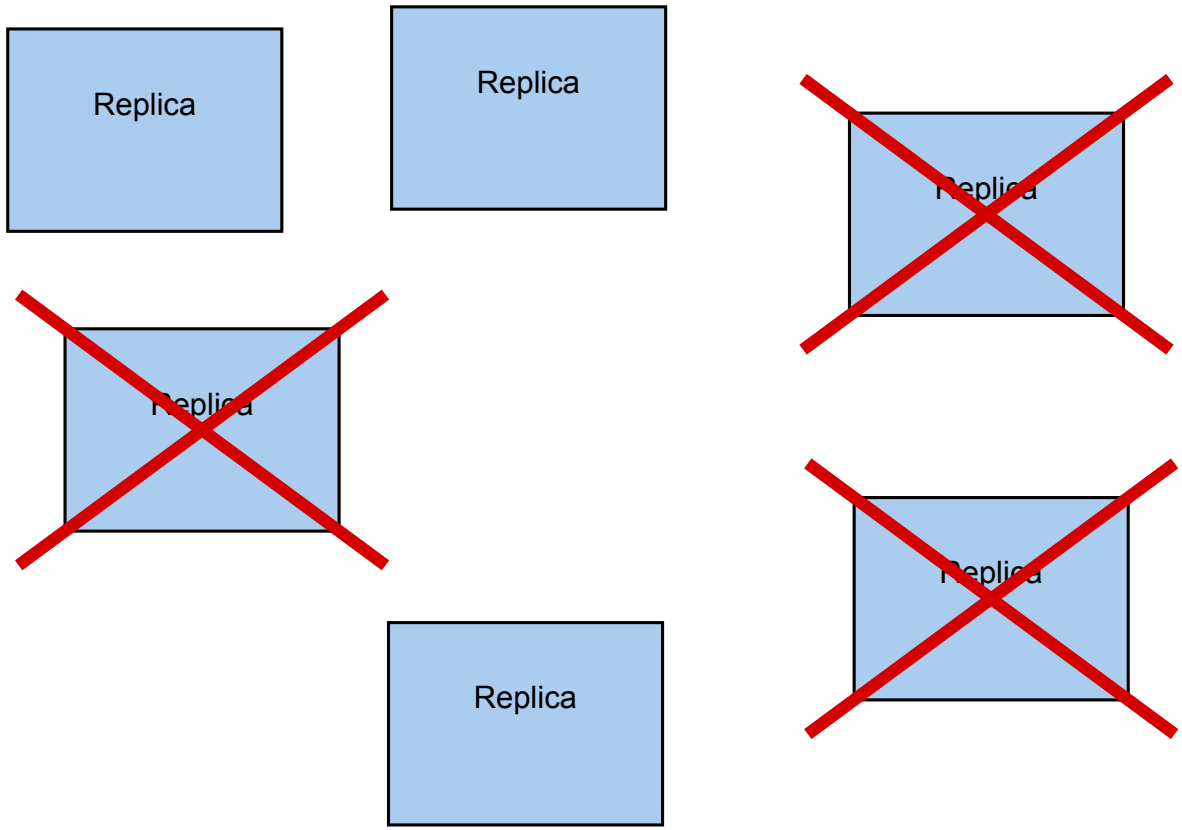
Replicated State Machine is Not Enough



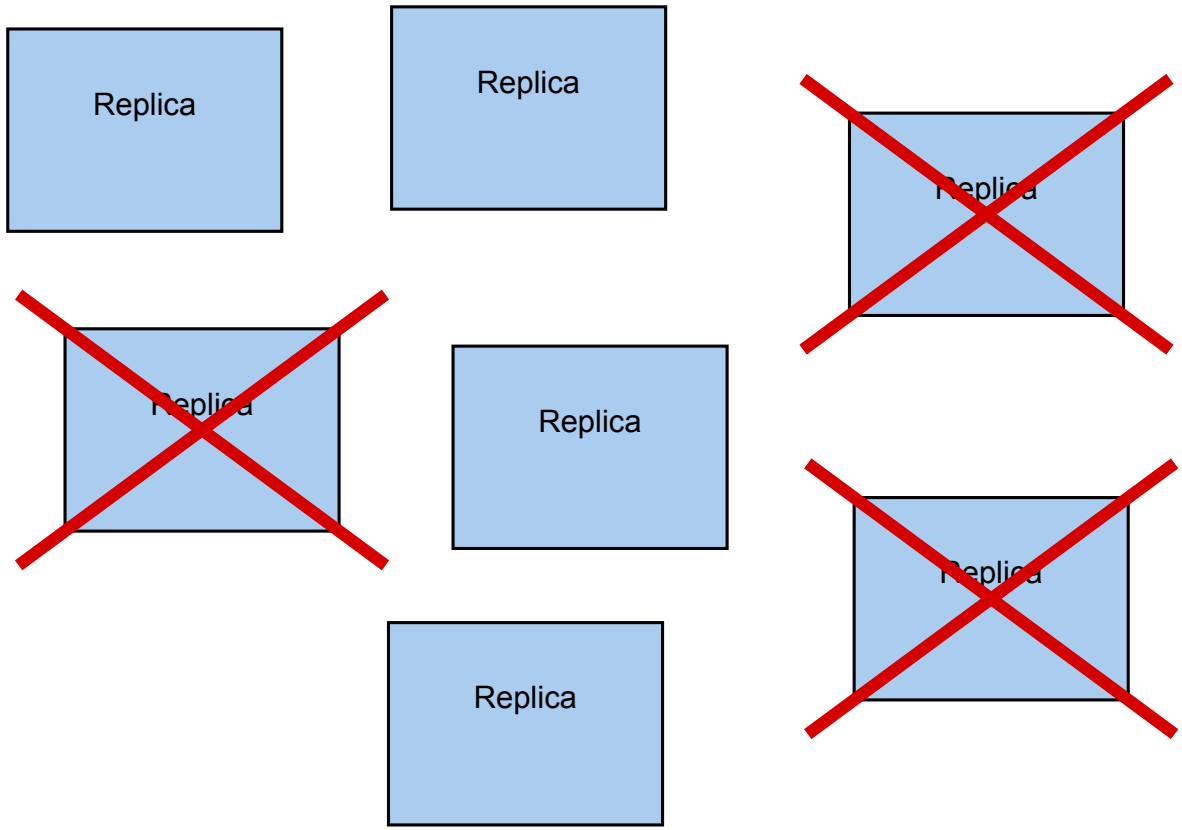
Replicated State Machine is Not Enough



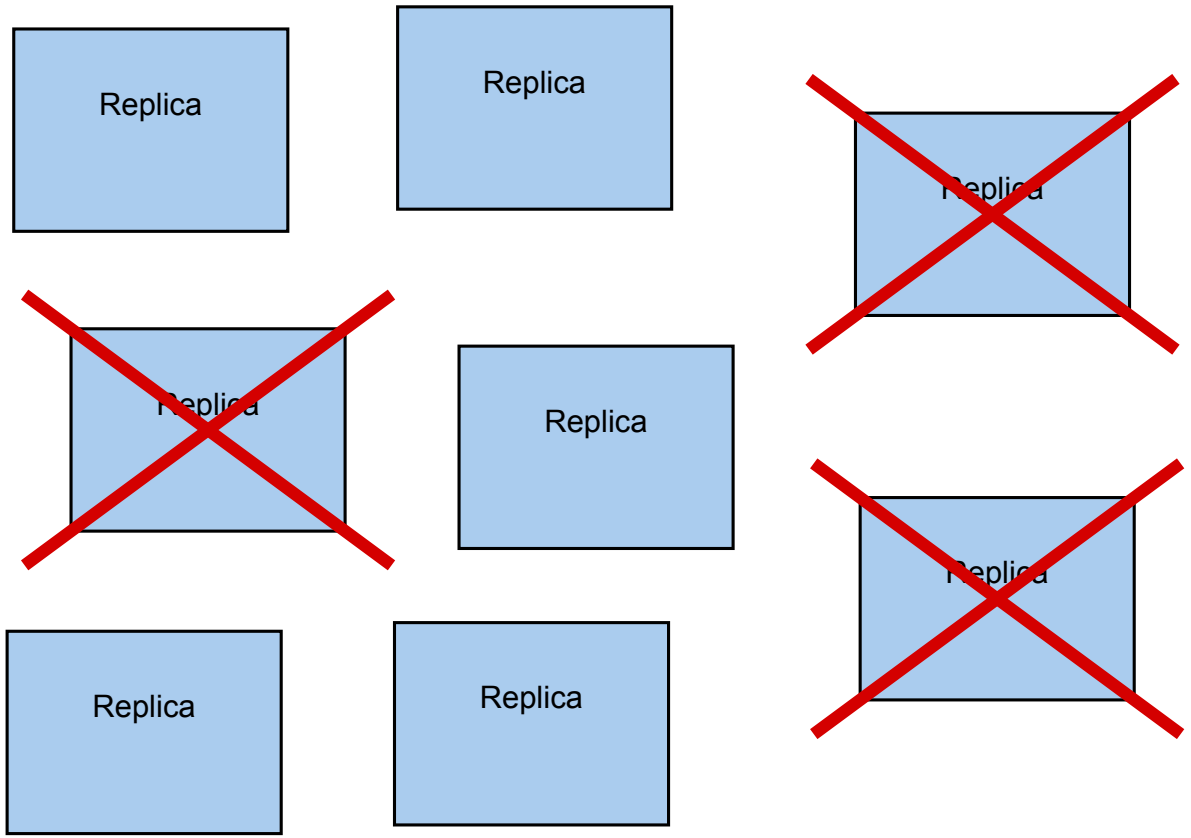
Replicated State Machine is Not Enough



Replicated State Machine is Not Enough



Replicated State Machine is Not Enough



Migration: Basic Idea

- A configuration is the set of replicas
- Replicas include the configuration as part of the state
- A special request that changes the configuration
- Handled like normal requests (assigned a slot then executed)
- The change take effect after α slots
- We used the SMART algorithm Details

Our Algorithm

Goals

- Automatically maintain configuration in a decentralized way
- Select resources, detect failures, and decide to migrate
- Users find service without central repository

Approach

- We use Structure Overlay Networks(SONs)
- We use replica placement schemes (such as symmetric replication) to select nodes that will host replicas
- We use lookups and DHT ideas
- We use failure detection provided by SONs

Our Algorithm

Goals

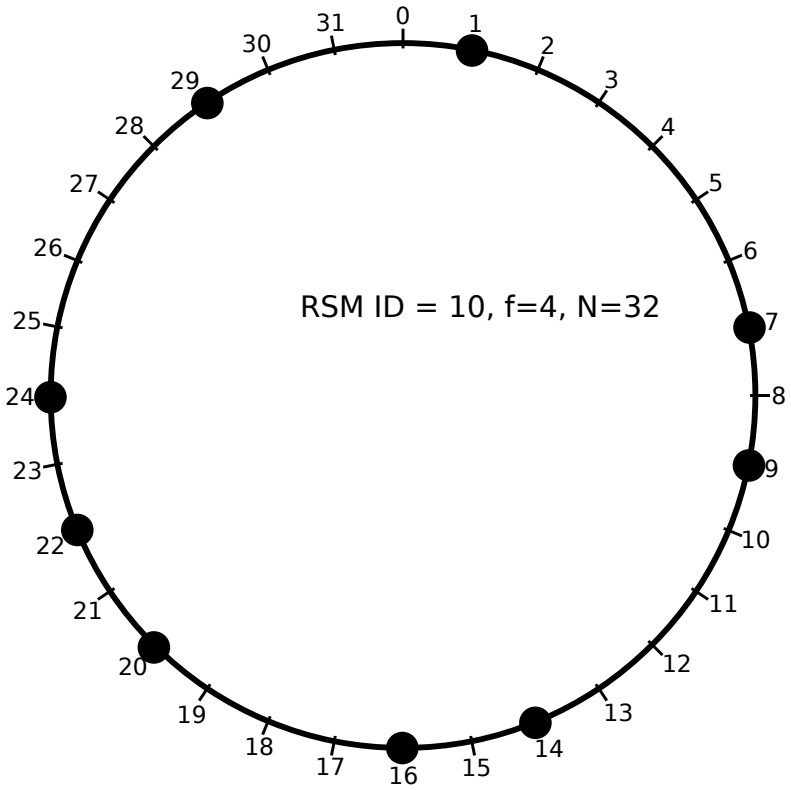
- Automatically maintain configuration in a decentralized way
- Select resources, detect failures, and decide to migrate
- Users find service without central repository

Approach

- We use Structure Overlay Networks(SONs)
- We use replica placement schemes (such as symmetric replication) to select nodes that will host replicas
- We use lookups and DHT ideas
- We use failure detection provided by SONs

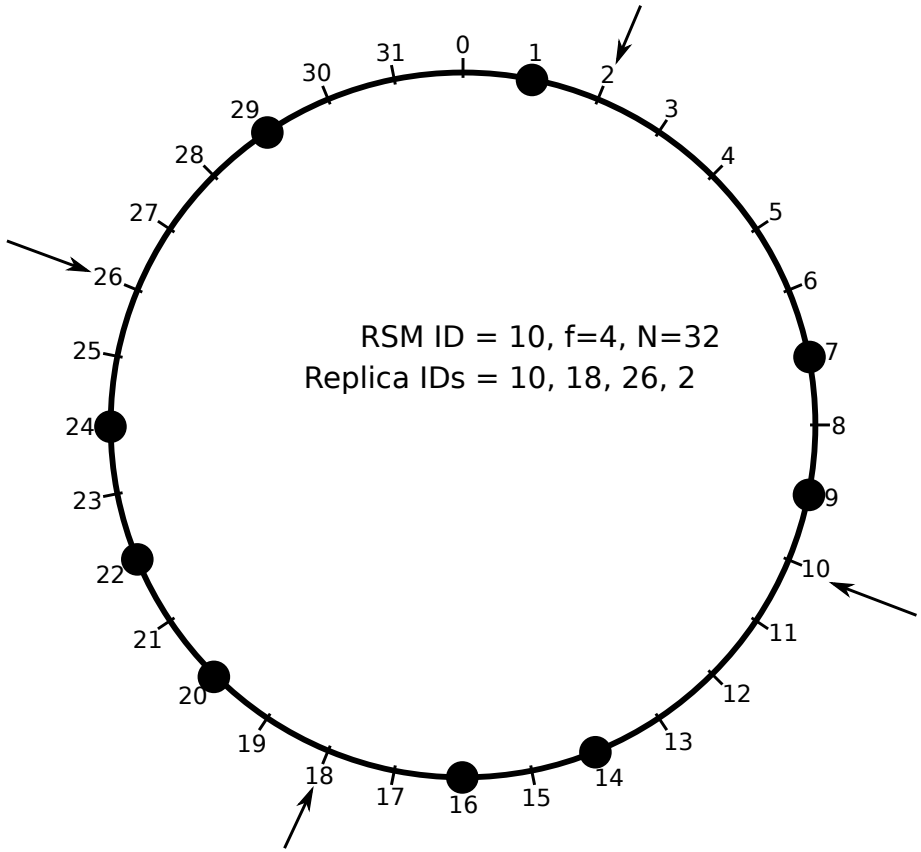
Creating a Replicated State Machine (RSM)

Any node can create a RSM. Select ID and replication degree



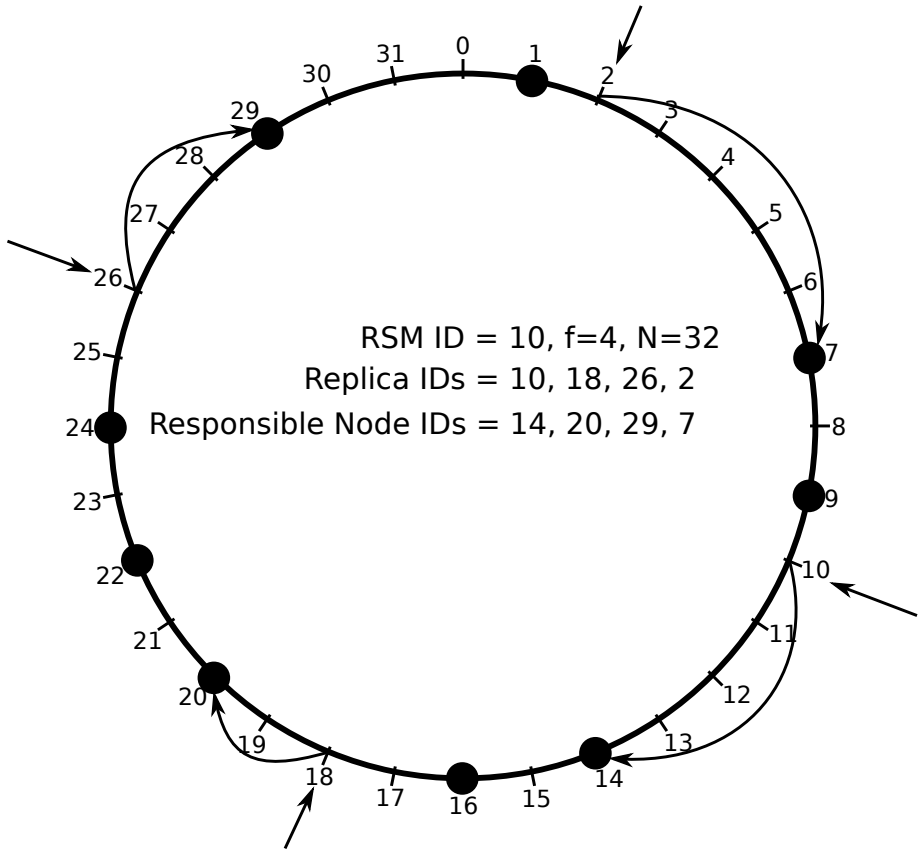
Creating a Replicated State Machine (RSM)

The node uses symmetric replication to calculate replica IDs



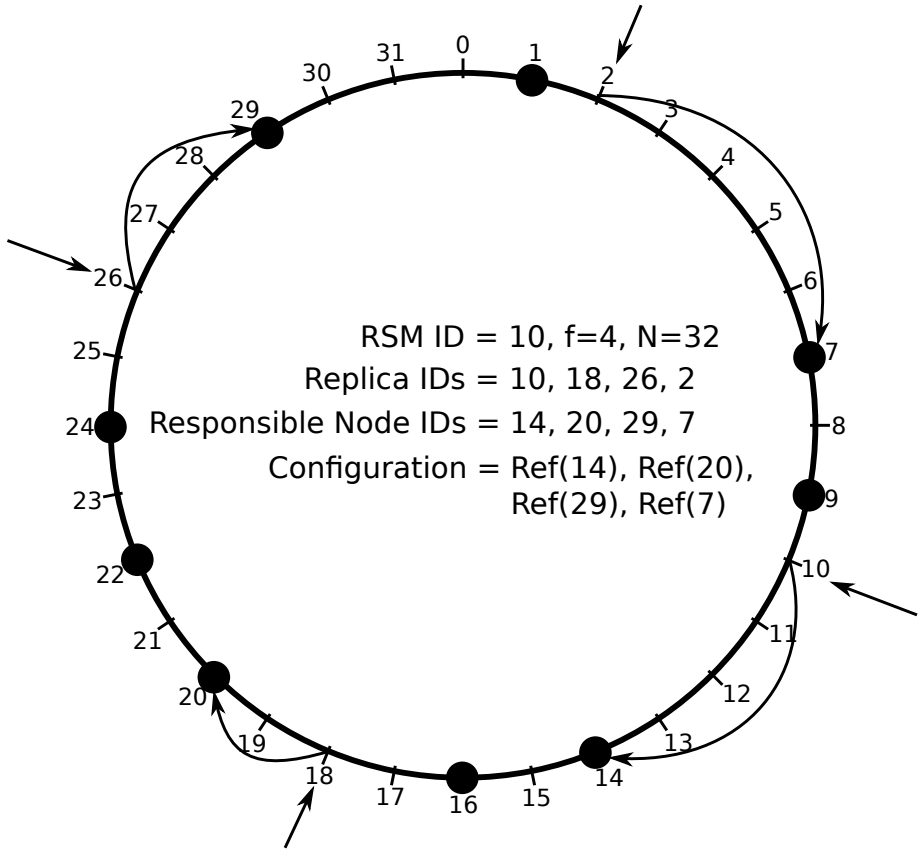
Creating a Replicated State Machine (RSM)

The node use lookups to find responsible nodes . . .



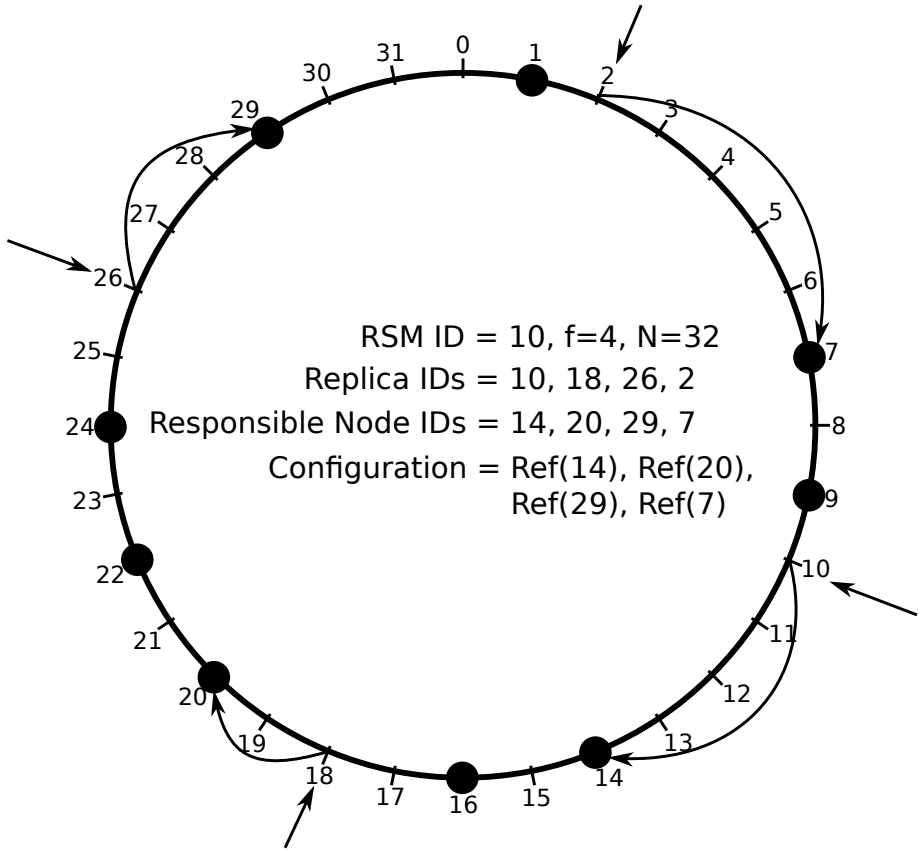
Creating a Replicated State Machine (RSM)

. . . and gets direct references to them



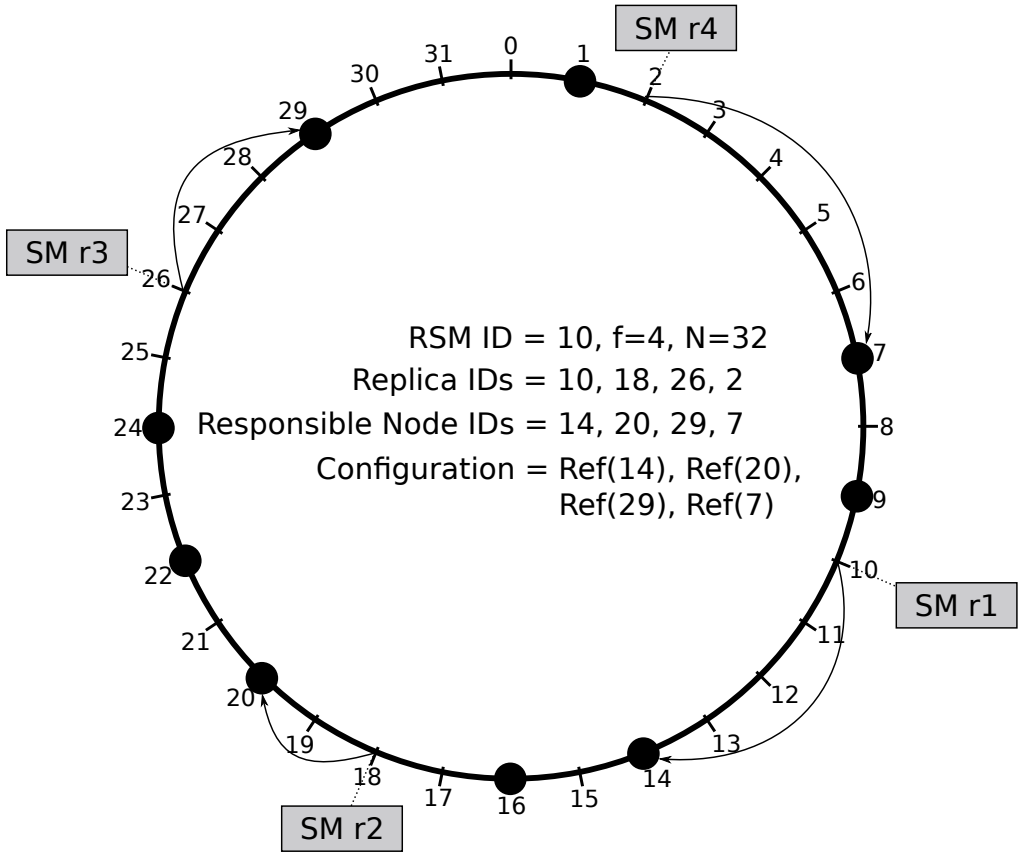
Creating a Replicated State Machine (RSM)

The set of direct references forms the configuration



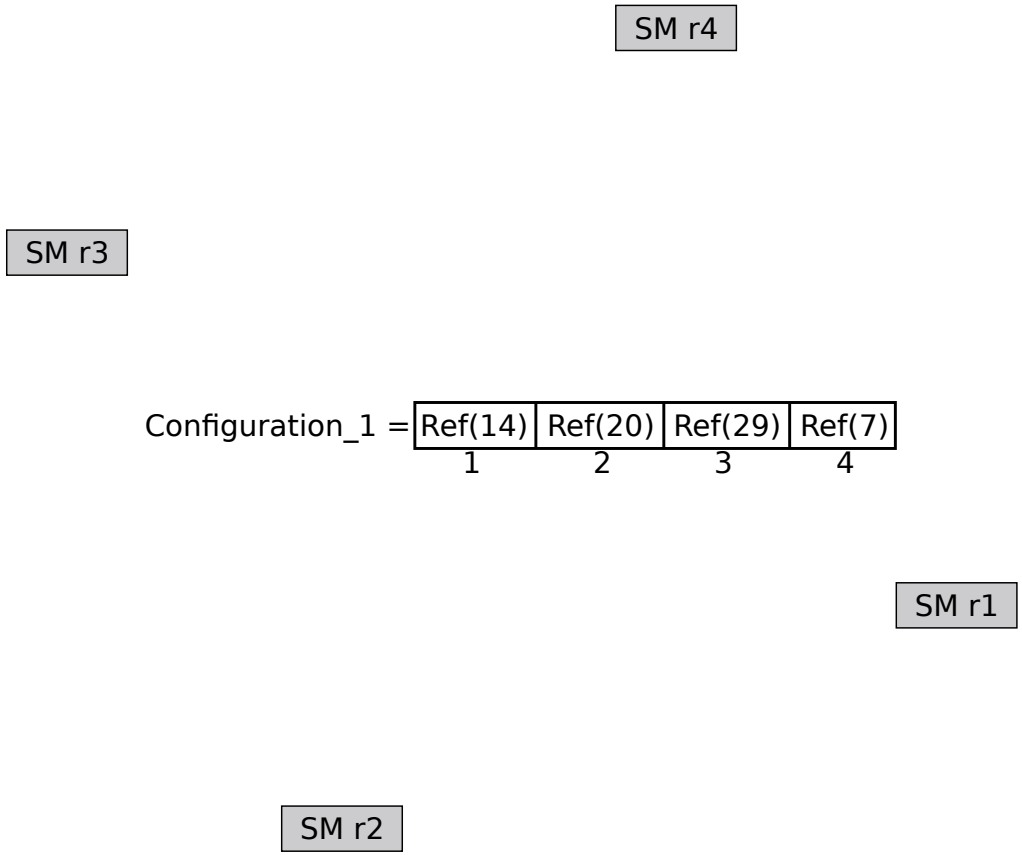
Creating a Replicated State Machine (RSM)

The node sends a *Create* message to the configuration

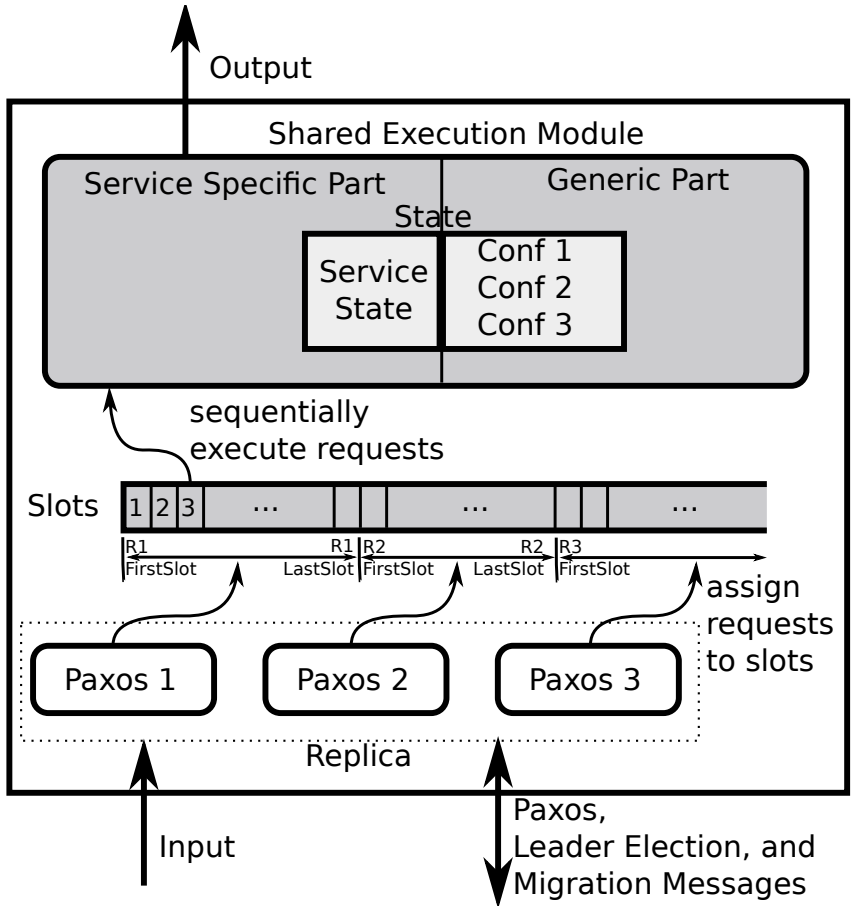


Creating a Replicated State Machine (RSM)

Now replicas communicate directly using the configuration



Replica Architecture



When to Migrate?

- To fix Lookup inconsistencies
- To handle resource churn

Handling Lookup Inconsistency

- Because of lookup inconsistency the configuration may contain incorrect nodes
- The inconsistency is detected when a node receives a request targeted at a replica that the node does not have but should be responsible for
- In this case the node issues a configuration change request asking the current configuration to replace the incorrect node with itself

Handling Churn

- Similar to handling churn in a DHT
  - When a node joins it gets a list of replicas (RSM_ID and rank) it is responsible for form its successor
  - When a node leaves it hand over replicas to its successor
  - When a node fails the successor uses symmetric replication and interval cast to find replicas it should be responsible for
- After getting the list of replicas the node issue a configuration request to each RSM to replace incorrect node with itself

Changing the Configuration (Migration)

- In SMART the admin sends a configuration change request that contains all nodes in the new configuration
- We can not do the same in a decentralized fashion to avoid conflicts

Example

- Assume current configuration is {A, B, C, D}
- Node X detects that C is dead and requests change to {A, B, X, D}
- Node Y detects that D is dead and requests change to {A, B, C, Y}
- Y overrides the change made by X!

Changing the Configuration (Migration)

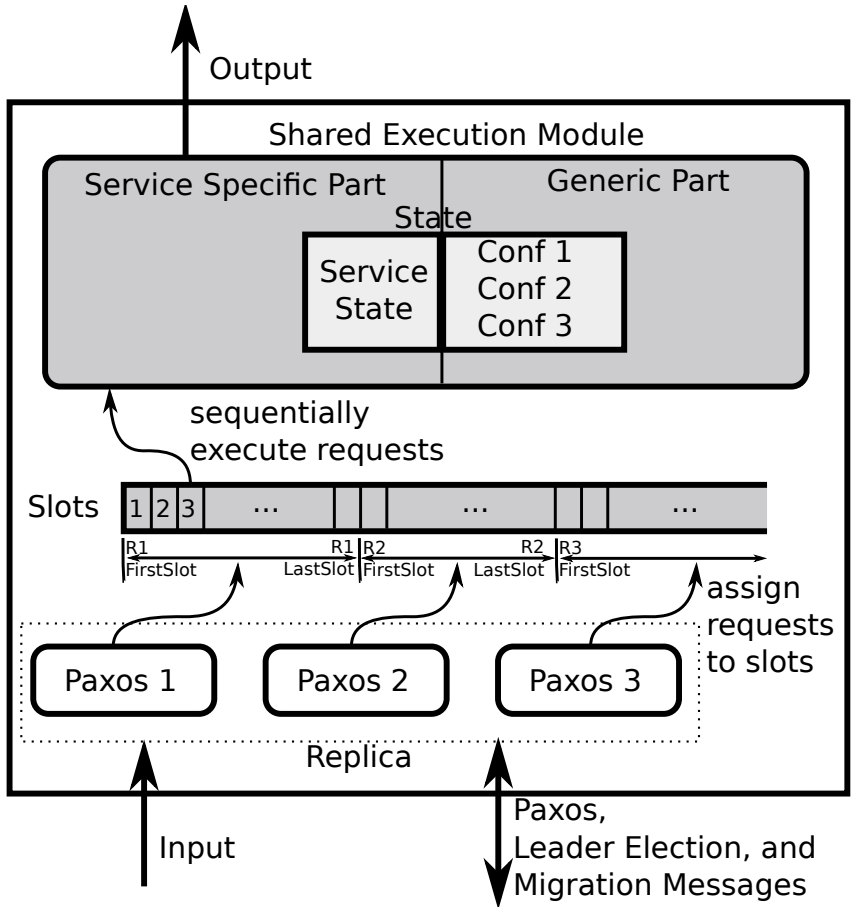
- In our approach the request does not contain the entire configuration. It contain only a request to replace a particular node

Example

- Assume current configuration is {A, B, C, D}
- Node X detects that C is dead and requests replacing replica 3 with itself
- Node Y detects that D is dead and requests replacing replica 4 with itself
- The end result is {A, B, X, Y}

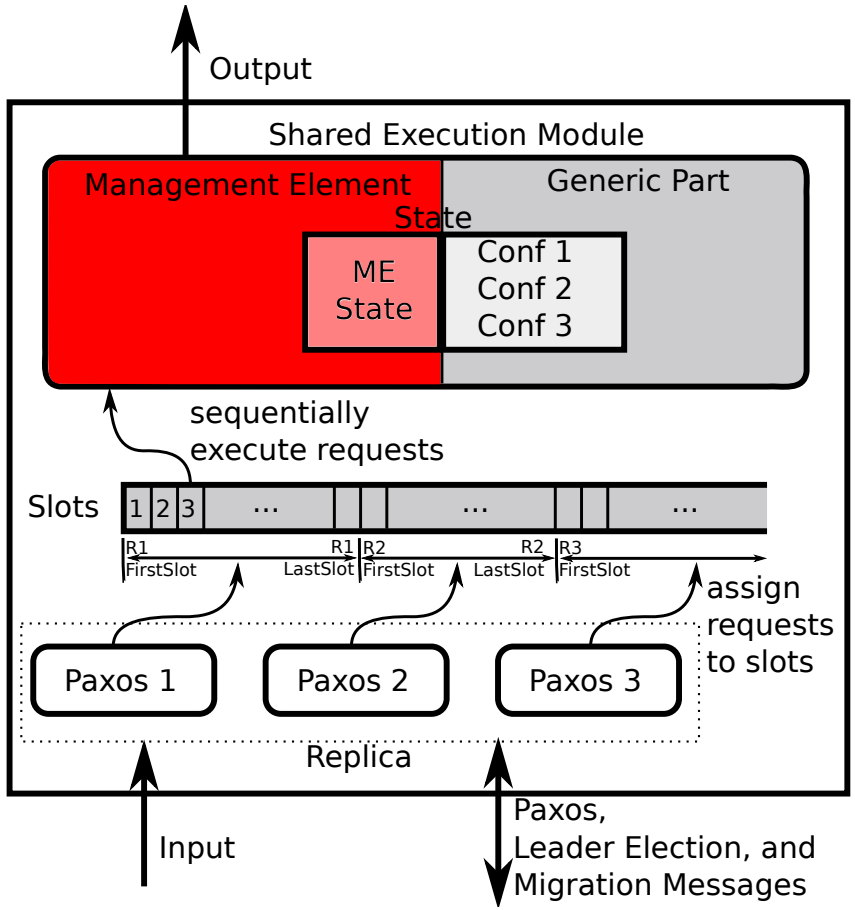
Robust Management Elements

- Our approach is generic and can be useful for many services
- We use it in Niche to implement Robust Management Elements
- Replace the service specific part of the execution module with a management element



Robust Management Elements

- Our approach is generic and can be useful for many services
- We use it in Niche to implement Robust Management Elements
- Replace the service specific part of the execution module with a management element



Outline

Conclusions

- Niche Platform
  - Enable self-management
  - Programming and runtime execution
  - Large-scale and/or dynamic systems
- Methodology
  - Design space and guidelines
  - Interaction patterns
- YASS use case
- Policy based management
- Robust Management Elements

Future Work

- Refine design methodology including steps and interaction patterns
- Consider more use cases focusing on real applications
- Study and investigate management patterns and techniques
  - Distributed control, distributed optimization
  - Model Predictive Control (MPC)
  - Reinforcement learning in (feedback) control
  - Networked Control System (NCS)
- Focus more on self-tuning
- Complete work on Robust Management Elements
- Port Niche to Kompics component model

Thank you for careful listening :-)

Questions?

SPL Policy Example

```
Policy {
  Declaration {
    lowloadthreshold = 500;
  }
  Condition {
    storageInfo.totalLoad <= lowloadthreshold
  }
  Decision {
    manager.setTriggeredHighLoad(false) &&
    manager.delegateObligation("release storage")
  }
}:1;
```

Return

XACML Policy Example

```
<Policy PolicyId="lowLoadPolicy"
        RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Target>
    <Subjects>       <AnySubject />     </Subjects>
    <Resources>       <AnyResource />     </Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
            load
          </AttributeValue>
          <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
                                     DataType="http://www.w3.org/2001/XMLSchema#string" />
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule Effect="Permit" RuleId="lowLoad">
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-less-than-or-equal">
        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:double-one-and-only">
         <EnvironmentAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#double"
                                    AttributeId="totalLoad"/>
       </Apply>
      <AttributeValue> 500 </AttributeValue>
    </Condition>
  </Rule>
  <Obligations>
    <Obligation FulfillOn="Permit" ObligationId="2">
      <AttributeAssignment AttributeId="lowLoad_obligation" DataType="http://www.w3.org/2001/XMLSchema#integer">
        "release storage"
      </AttributeAssignment>
    </Obligation>
  </Obligations>
</Policy>
```

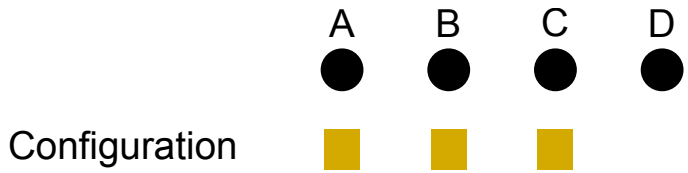
Return

Migration: The SMART Algorithm

- SMART is a new technique for changing the set of nodes (configuration) where a replicated service runs (i.e. migrating the service)
- Advantages over other approaches (as described by SMART authors):
  - Allows migrations that replace non-failed nodes (suitable for automated service)
  - Can pipeline concurrent requests (performance optimization)
  - Provides complete description

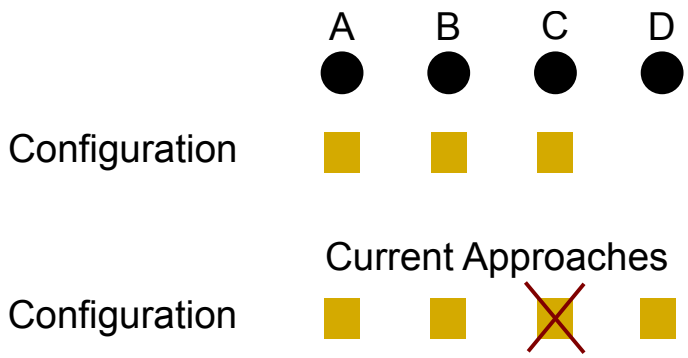
Configuration-Specific Replicas

- Each replica is associated with one and only one configuration
- Migration creates a new set of replicas (configuration)
- Simplifies the migration process
- Each configuration uses its own instance of the Paxos algorithm
- Inefficient implementation (use shared execution module to improve it)



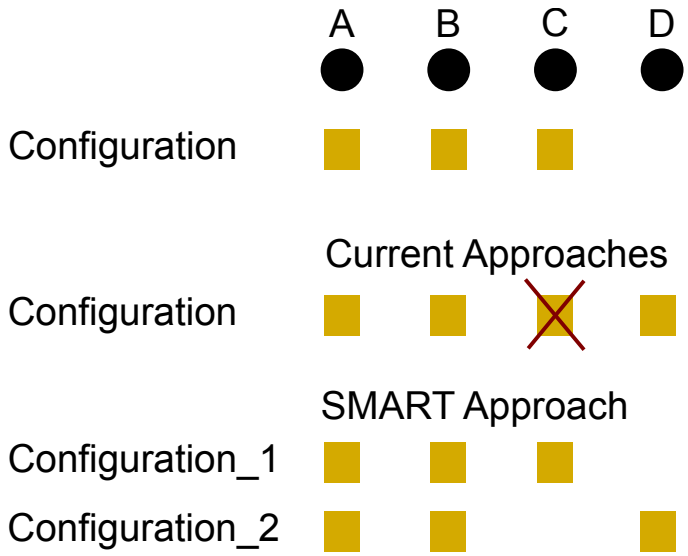
Configuration-Specific Replicas

- Each replica is associated with one and only one configuration
- Migration creates a new set of replicas (configuration)
- Simplifies the migration process
- Each configuration uses its own instance of the Paxos algorithm
- Inefficient implementation (use shared execution module to improve it)



Configuration-Specific Replicas

- Each replica is associated with one and only one configuration
- Migration creates a new set of replicas (configuration)
- Simplifies the migration process
- Each configuration uses its own instance of the Paxos algorithm
- Inefficient implementation (use shared execution module to improve it)



SMART

- Avoids inter-configuration conflicts by assigning none overlapping range of slots [*FirstSlot*, *LastSlot*] to each configuration
- The old configuration sends a Join message to the new configuration
- A replica in a new configuration need to copy state from another replica (up till at least $FirstSlot - 1$)
- Destroying old configurations (Finished and Ready messages)
- Clients use a configuration repository to find the current configuration
- SMART does not deal with how to select a configuration and when to migrate

Challenges Implementing Lamport's Idea

- **Unaware-leader challenge:** A new leader may not know the latest configuration
- **Window-of-vulnerability challenge:** Migrations that remove or replace a machine can create a period of reduced fault tolerance
- **Extended-disconnection challenge:** After a long disconnection, a client may be unable to find the service
- **Consecutive-migration challenge:** If request n changes the configuration, requests $n + 1$ through $n + \alpha - 1$ cannot change the configuration
- **Multiple-poll challenge:** A new leader may have to poll several configurations

Return