CONTROL AND RESOURCE ALLOCATION IN A DATA CENTER

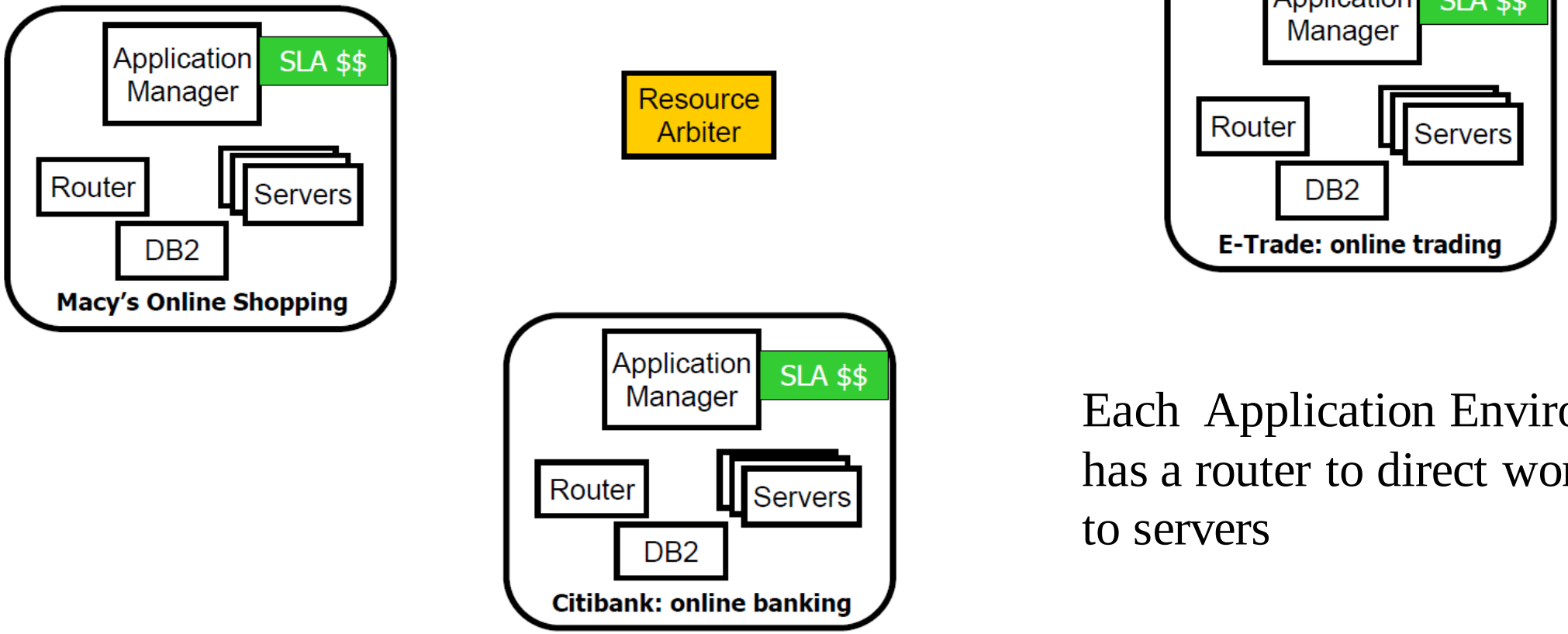
The data center manages numerous resources, including compute servers, database servers, storage devices, etc., and serves many different customers using multiple large-scale applications.

The focus here is on the dynamic allocation and management of the compute servers within the data center, although the general methodology applies to multiple, arbitrary resources.

The high-level architecture of the data center model and
the architecture of the Application Managers, which manage individual applications,
are described, and details are provided on how they enable
the use of utility functions to manage the data center resources.

Data Center Architecture

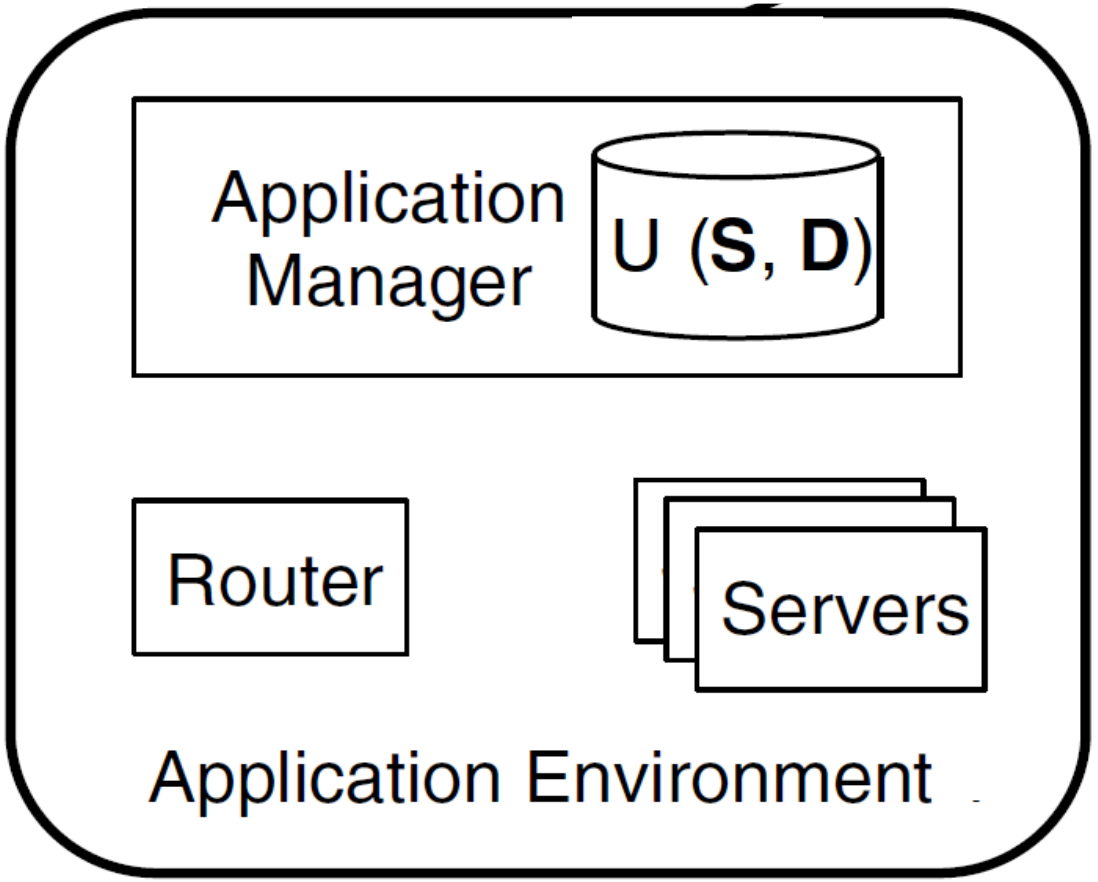
The data center contains a number of logically separated ***Application Environments***, each providing a distinct application service using a dedicated, but dynamically allocated, pool of resources of various types, such as application servers, databases, or even virtual resources such as logical partitions.



Each Application Environment has a router to direct workload to servers

Each Application Environment has a ***service-level utility function*** $U(\boldsymbol{S}, \boldsymbol{D})$ specifying the business value of providing a given level of service to users of the Application Environment.

The utility function may reflect the payment/penalty terms of service-level agreements with customers, and may also incorporate considerations such as the value of maintaining the data center's reputation for providing good service.

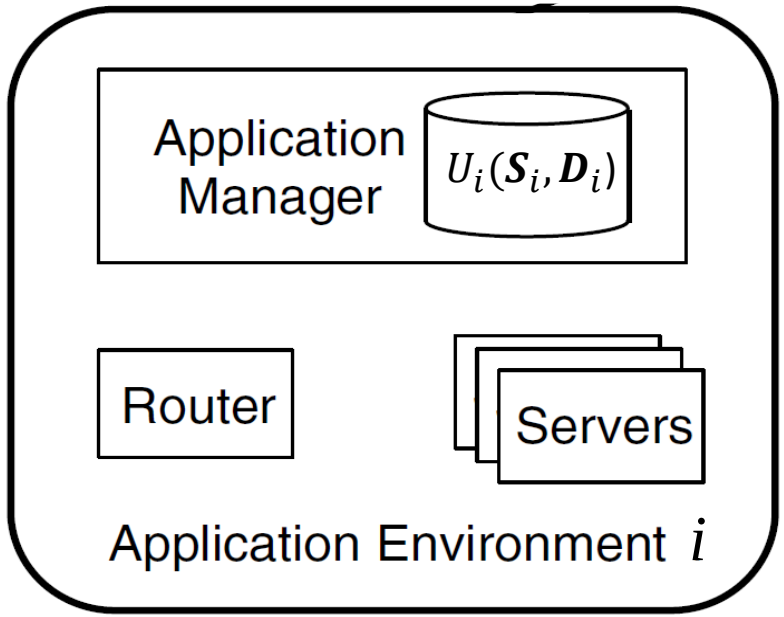


The utility function is independent of that of other Application Environments.
All utility functions share a common scale of valuation, such as a common currency.

The utility function for environment i is of the form

$$U_i(\mathbf{S}_i, \mathbf{D}_i),$$

where $\mathbf{S}_i$ is the ***service level space*** in i and $\mathbf{D}_i$ is the ***demand space*** in i.



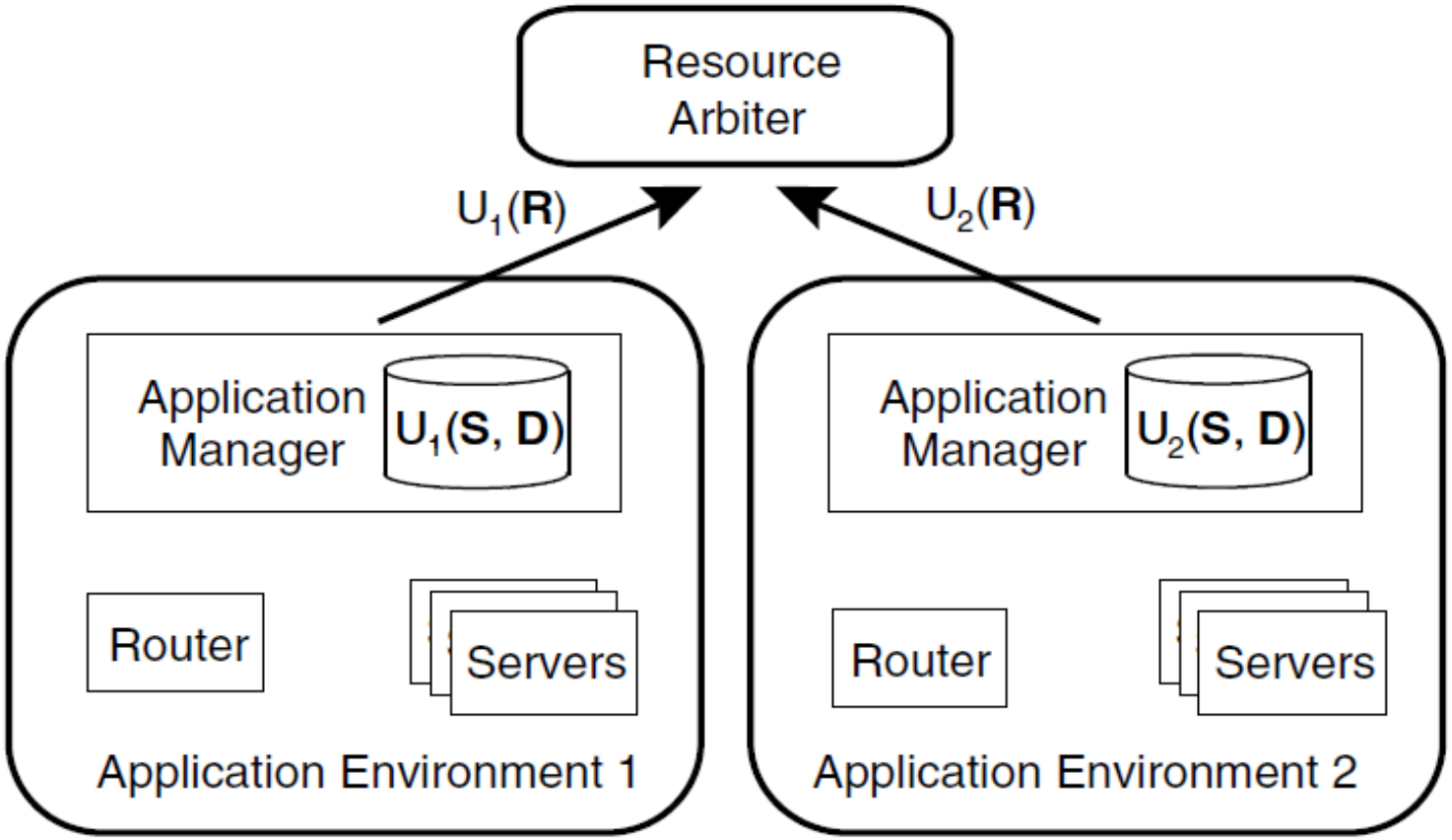
$\mathbf{S}_i$ and $\mathbf{D}_i$ are vectors that specify ***values for multiple user classes*.**

$\mathbf{S}_i$ is particular to i, and can contain any viable service metrics (e.g., response time, throughput, etc.).

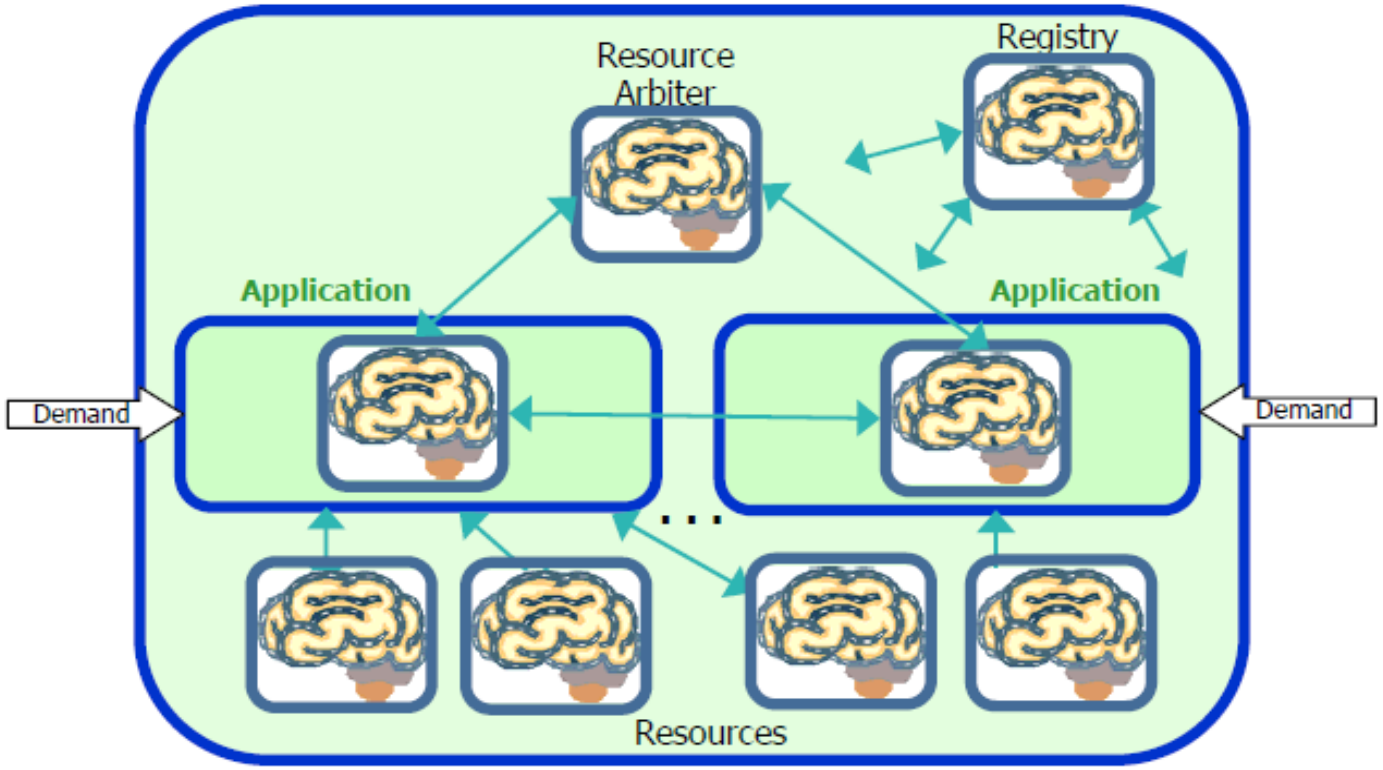
Although such service-level specification of utility will often be most useful, $\mathbf{S}_i$ could possibly directly measure resources assigned to the classes in i.

The system goal is to optimize $\sum_i U_i(\mathbf{S}_i, \mathbf{D}_i)$ on a continual basis to accommodate fluctuations in demand.

A distributed two-level architecture is employed to achieve this end.



The distributed architecture is built out of multiple interacting autonomic elements.



Autonomic elements, analogous to software agents, are the basic self-managing building blocks of autonomic computing systems.
They manage their own behavior and their relationships with other autonomic elements, through which they provide or consume computational services.

The global optimization task is distributed among autonomic elements in the two-level structure.

At the lower level, the detailed control and optimization of a fixed amount of resources within an Application Environment is handled by a resident ***Application Manager***.

As demand shifts, Application Manager *i* may find it necessary to adjust certain control parameters or divert resources from one transaction class to another in order to keep $U_i(\mathbf{S}_i, \mathbf{D}_i)$ as optimal as possible, given a fixed amount $\mathbf{R}_i$ of resources.

$\mathbf{R}_i$ is a vector, each ***component*** of which indicates the amount of a specific ***type of resource*** that is allocated to Application Manager *i*.

At the higher level, allocation of resources across different Application Environments is performed by a global ***Resource Arbiter***.

The Resource Arbiter does not know details of how the individual Application Managers optimize their utility, nor details of the services provided by the individual Application Environments.

Instead, an Application Manager, when prompted by its own perceived need for more resource, or by a query from the Resource Arbiter, sends to the Arbiter a *resource-level utility function* $\widehat{U}(\mathbf{R})$ that specifies the value to the Application Environment of obtaining each possible level $\mathbf{R}$ of resources.

Given the current functions $\widehat{U}_i(\mathbf{R}_i)$ from the Application Managers, the Resource Arbiter periodically recomputes the resource allocation $\mathbf{R}^*$ that maximizes the global utility $\sum_i U_i(\mathbf{S}_i, \mathbf{D}_i) = \sum_i \widehat{U}_i(\mathbf{R}_i)$:

$$\mathbf{R}^* = \arg\max_{\boldsymbol{R}} \sum_i \widehat{U}_i(\mathbf{R}_i) \text{ such that } \sum_i \mathbf{R}_i = \overline{\mathbf{R}},$$

where $\overline{\mathbf{R}}$ indicates the total quantities of resources available.

This distributed two-level architecture is preferable to the centralized approach to global system optimization.

As each application environment is responsible for optimizing its own resource usage and for expressing its resource needs in a common, comparable form, it naturally supports the coexistence of multiple application environments that offer heterogeneous and arbitrarily complex services.

The internal complexities of individual Application Environments, including representing and modeling a potentially infinite variety of services and systems, are compressed by the Application Manager into a uniform resource-level utility function that relates value to resources, all in common units.

It is easy to add, change or remove Application Environments—even different *types* of Application Environments—because the Resource Arbiter requires no information about their internal workings. Any reconfiguration required of other elements is handled automatically by the system. In contrast, a centralized approach would require constant updates to the Resource Arbiter.

The two-level architecture also neatly handles the different time scales that are appropriate to different types of optimization, by treating them independently.

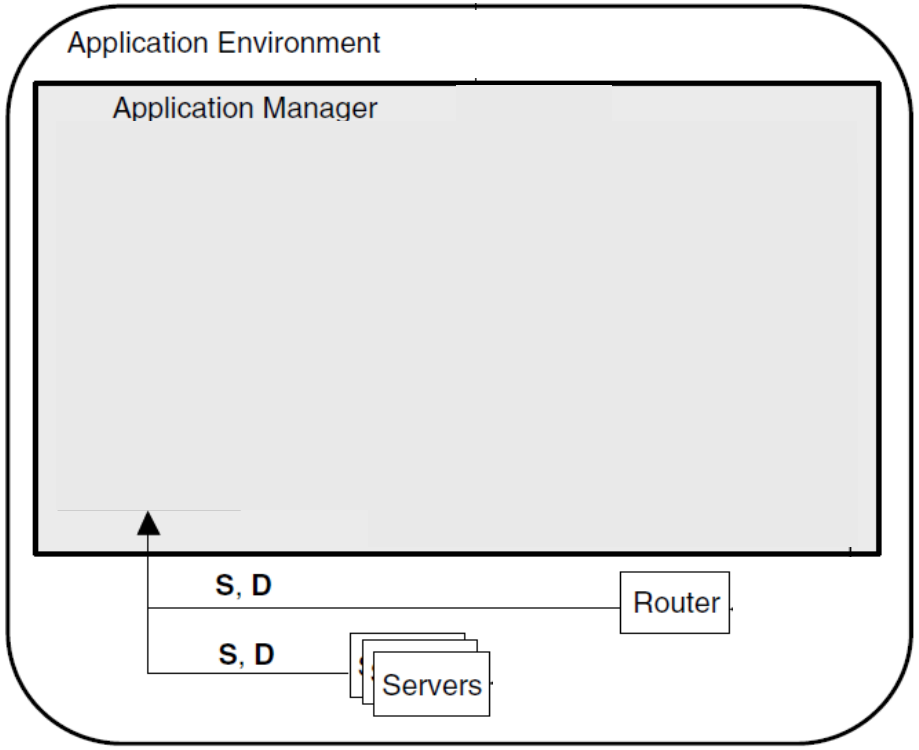
Application Managers adjust control parameters on a time scale of seconds to respond to changes in demand, while the Resource Arbiter typically operates on a time scale of minutes, more commensurate with switching delays necessitated by flushing out the current workload, changing connections, and installing or uninstalling applications.

There is time to recompute the resource allocation $\mathbf{R}^*$ that maximizes the global utility, which is an NP-hard discrete optimization problem that can be solved by mixed-integer programming.

Application Manager Architecture

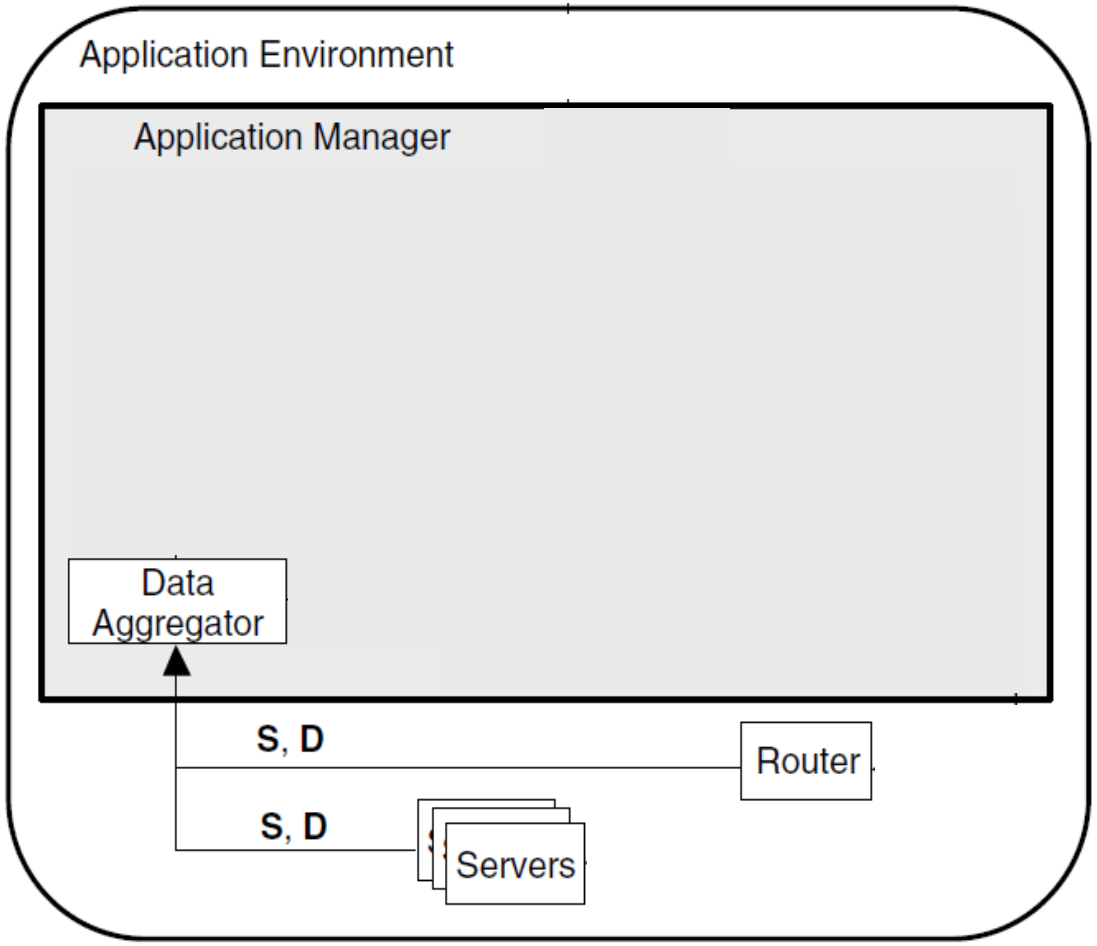
To understand how an Application Manager optimizes its utility $U_i(\mathbf{S}_i, \mathbf{D}_i)$ subject to fixed resource constraints and computes $\widehat{U}_i(\mathbf{R}_i)$ from $U_i(\mathbf{S}_i, \mathbf{D}_i)$, a close look at its internal architecture is needed.

Since a single Application Manager is considered here, the *i* subscripts are dropped.

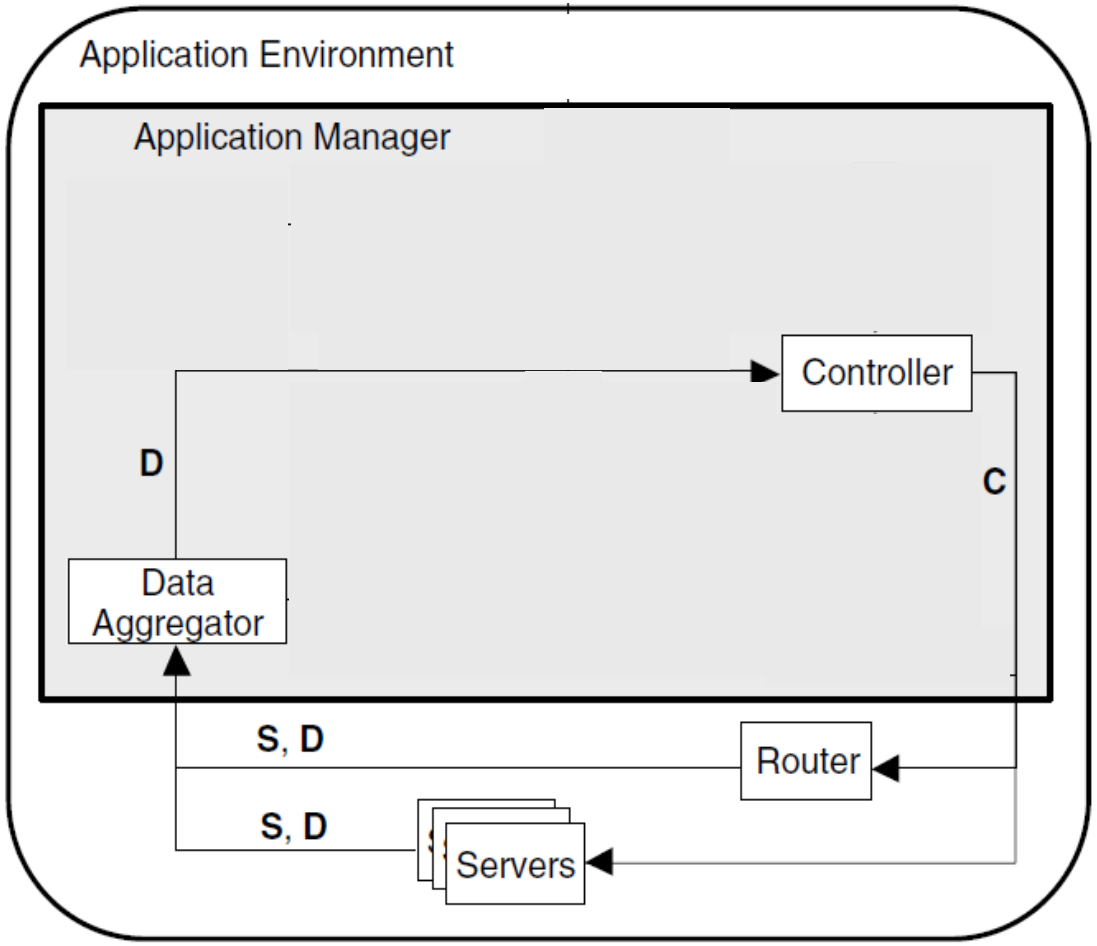


S = service level
D = demand

The Application Manager receives a continual stream of measured service **S** and demand **D** data from the router and servers.



The ***Data Aggregator*** aggregates these raw measurements, e.g. by averaging them over a suitable time window.



C = control parameters

The ***Controller*** continually adjusts the router and server control parameters **C** in an effort to optimize the utility in the face of fluctuating demand.

These parameters may specify how workloads from different customer classes are routed to the servers, as well as any other tunable parameters on the servers (e.g. buffer sizes, operating system settings, etc.).

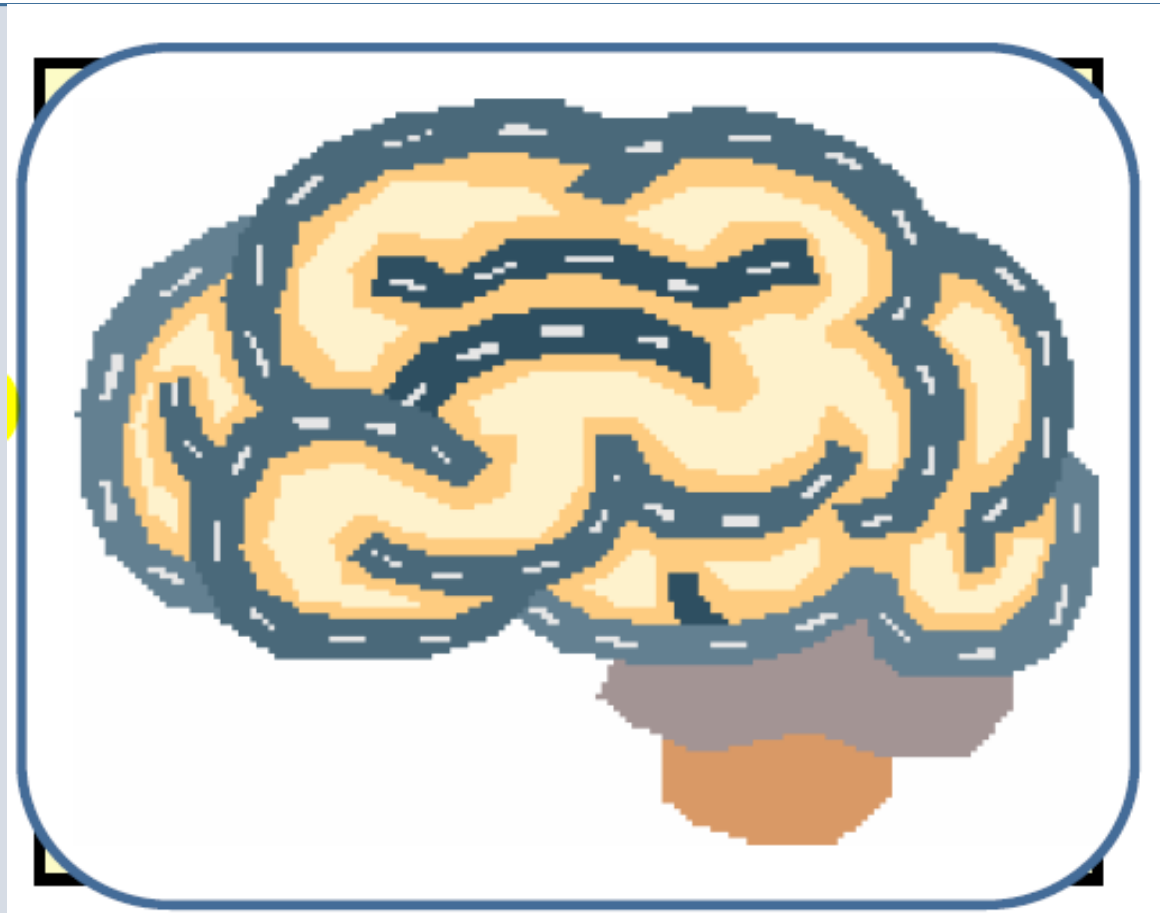
Composants autogérés

• Surveillance : réception des données via les capteurs.

• Analyse : obtention d'un diagnostic.

• Planification : détermination des actions à prendre.

• Exécution : mise en œuvre du plan.



An Autonomic Element

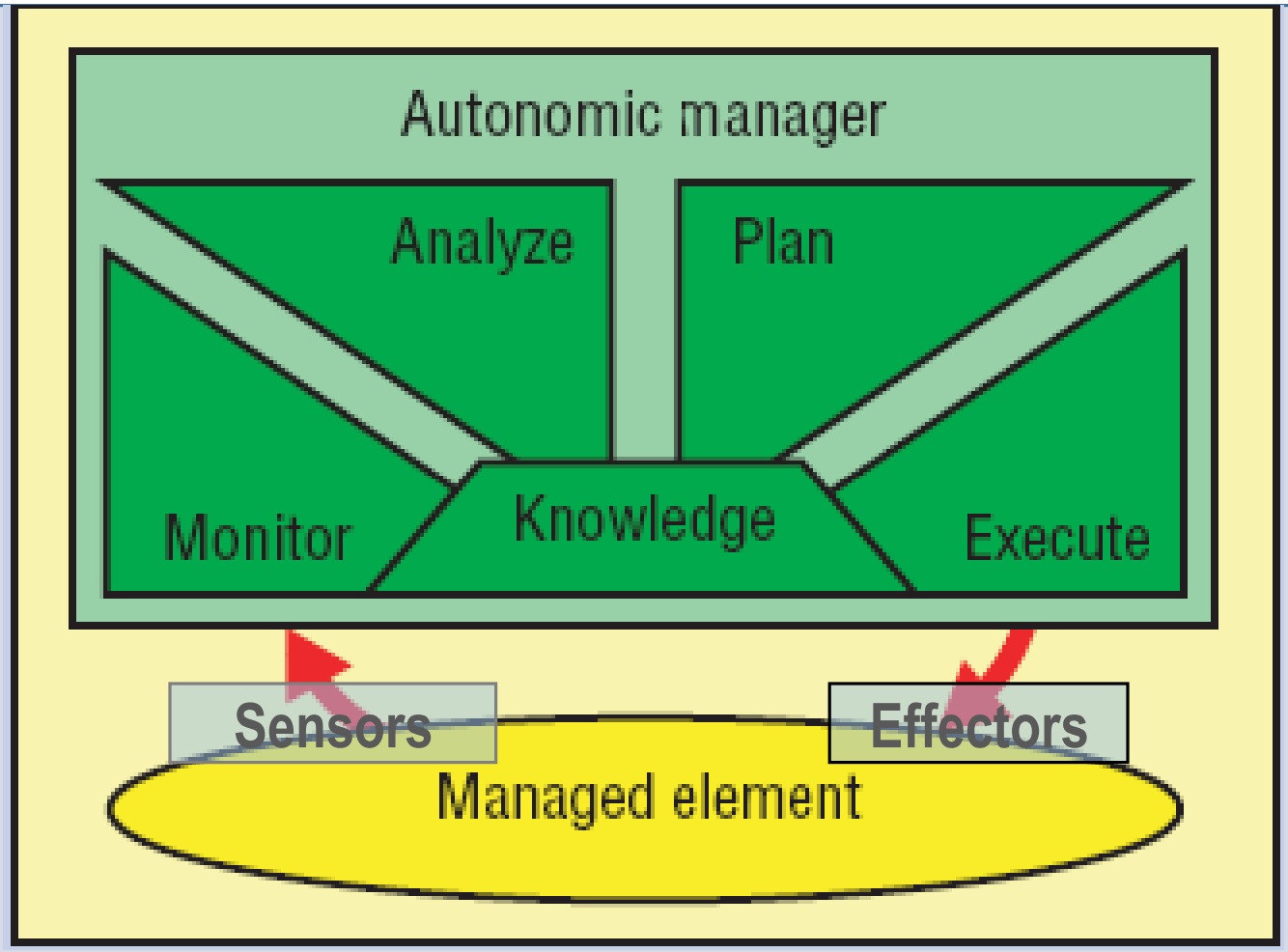
Composants autogérés

• Surveillance : réception des données via les capteurs.

• Analyse : obtention d'un diagnostic.

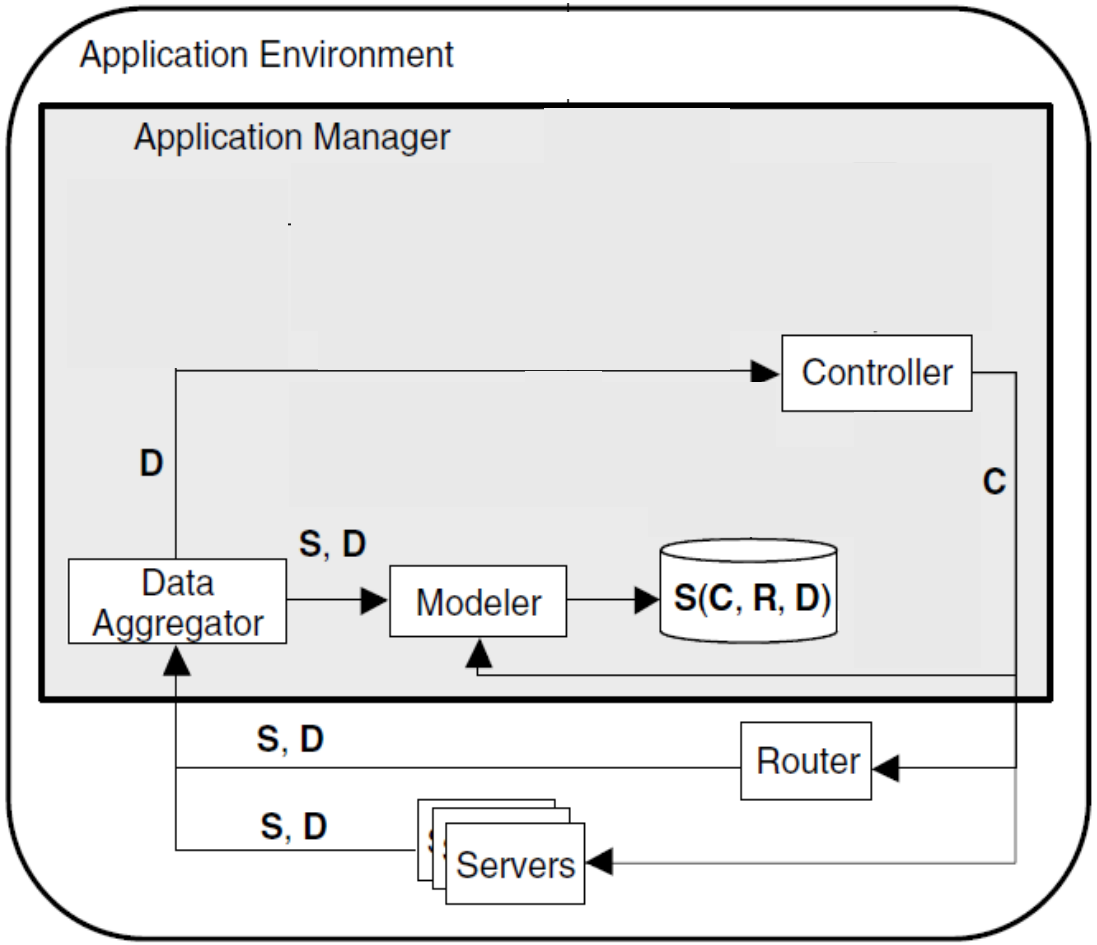
• Planification : détermination des actions à prendre.

• Exécution : mise en œuvre du plan.

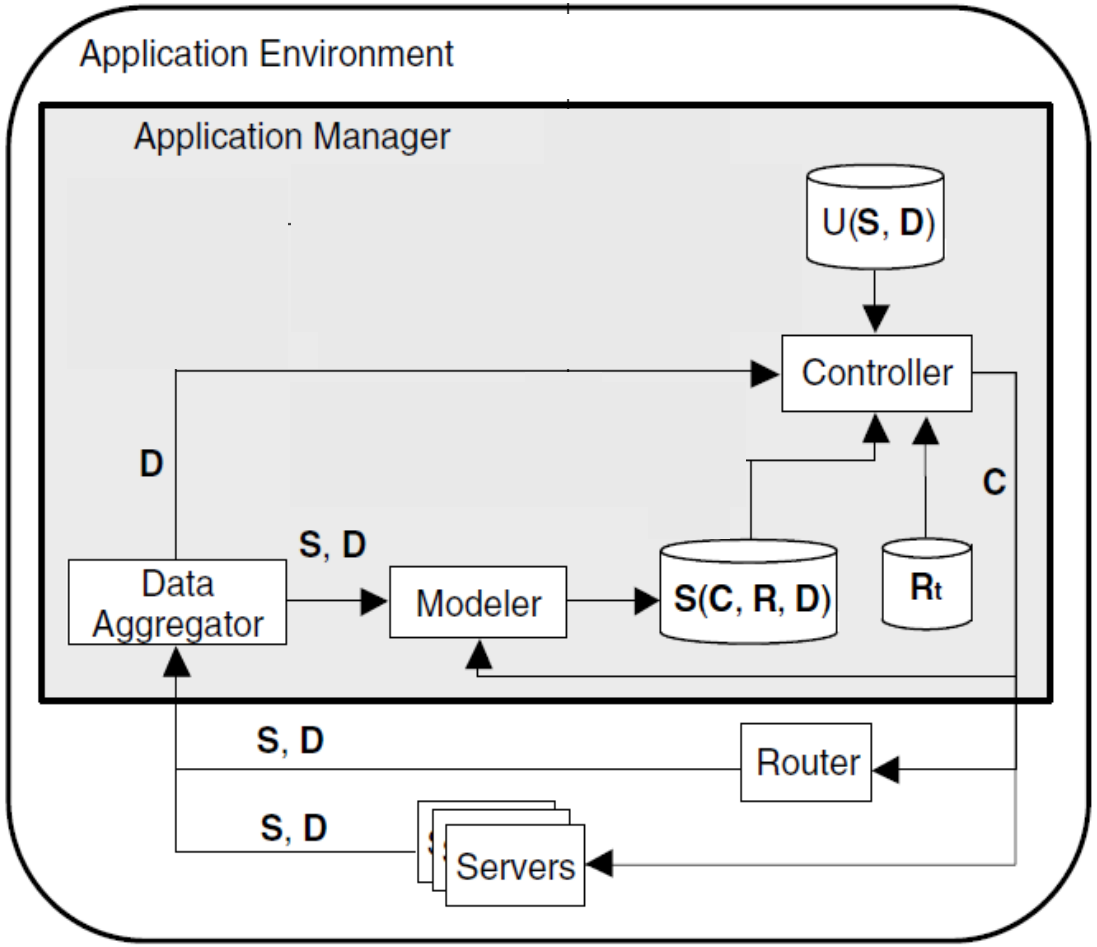


The Application Manager maintains at least three kinds of ***knowledge***:

- the service-level utility function $U(\mathbf{S}, \mathbf{D})$,
- the current resource level $\mathbf{R}_t$, and
- a ***model*** $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$ ***of system performance***.

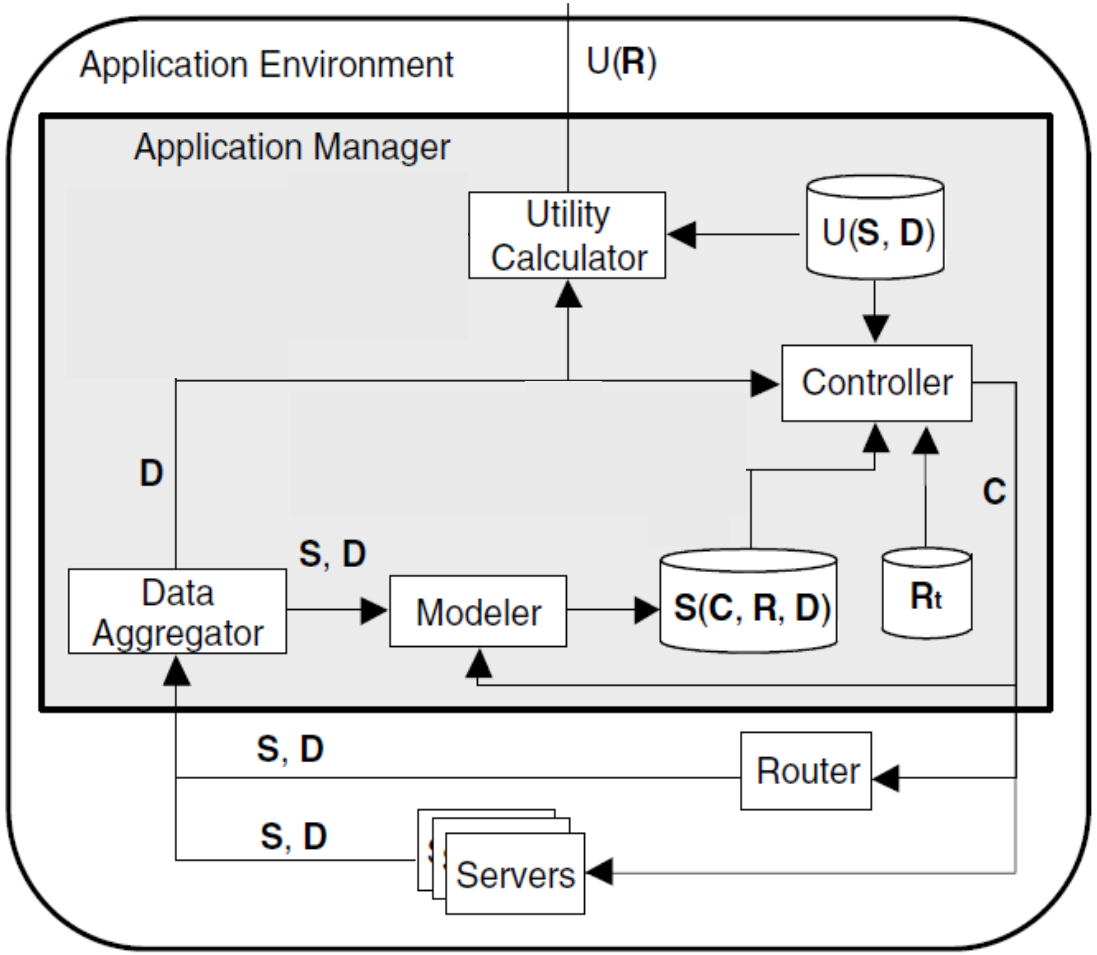


The ***Model*** specifies the vector of service levels that is obtained if the control parameters are set to **C**, the resources allocated to the Application Environment is **R**, and the demand is **D**. The model yields a ***vector of expected service attribute measurements***, which could, for example, represent one or more performance values for each customer class.

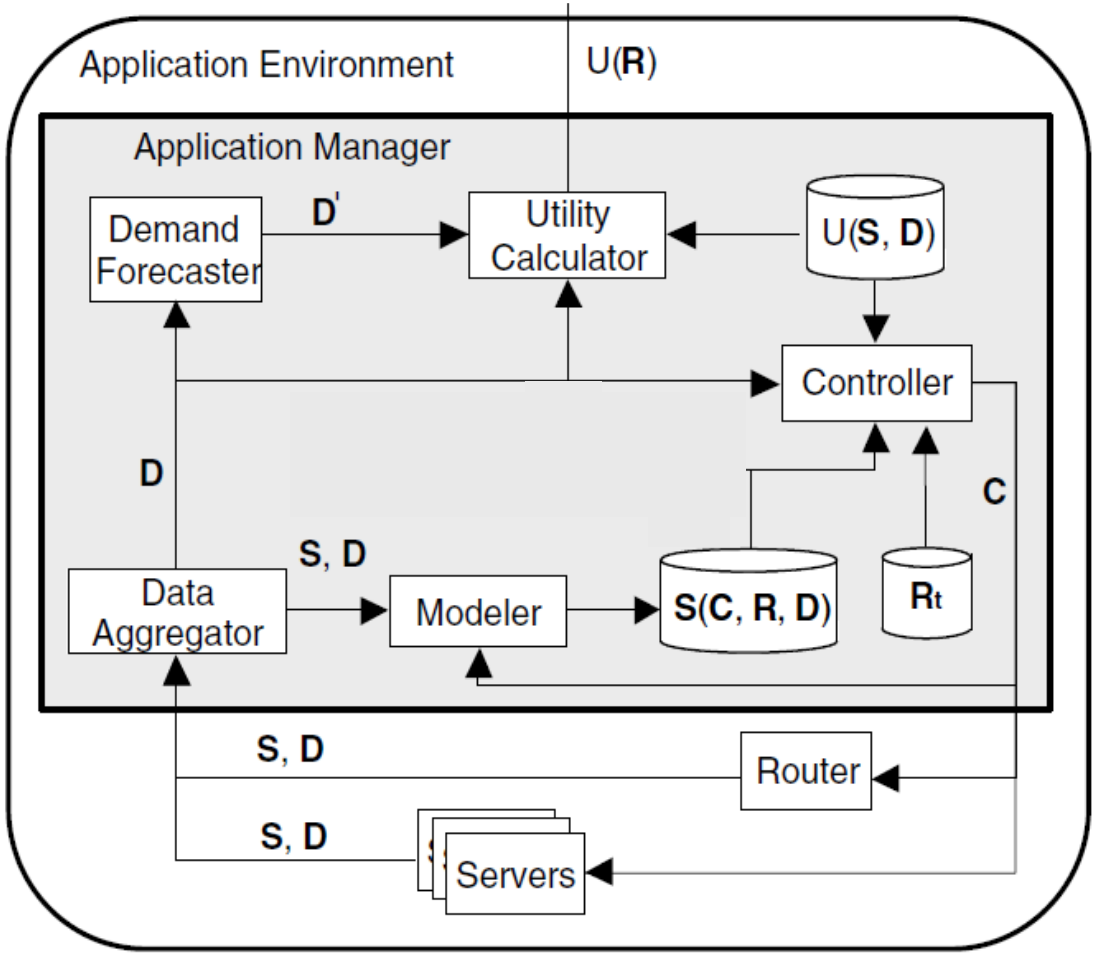


The ***Controller*** optimizes the utility $U(\mathbf{S},\mathbf{D})$ subject to fixed resource constraints.
It receives the aggregated demand $\mathbf{D}$ from the Data Aggregator. When this quantity changes sufficiently, the Controller recomputes the control parameters $\mathbf{C}^*$ that optimize $U(\mathbf{S},\mathbf{D})$ based on the performance model and current resource level:

$$\mathbf{C}^* = \underset{\mathbf{C}}{\operatorname{argmax}}\, U(\mathbf{S}(\mathbf{C},\mathbf{R}_t,\mathbf{D}),\mathbf{D})$$ and resets the control parameters to $\mathbf{C}^*$

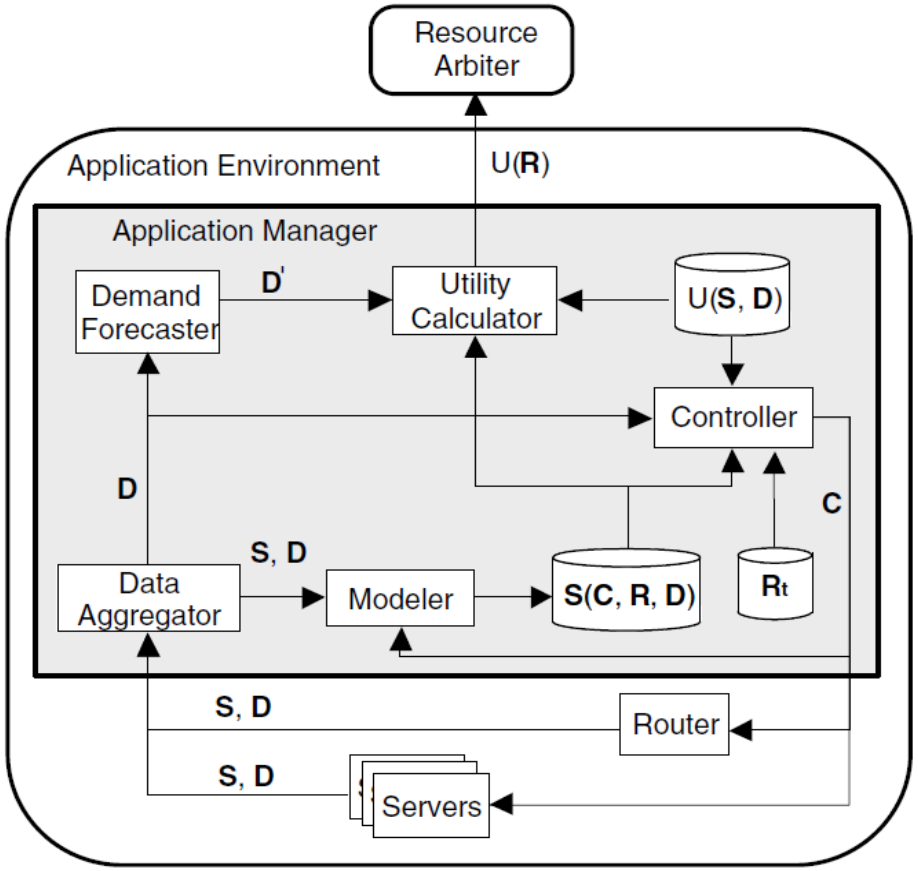


The ***Utility Calculator*** is responsible for computing the resource-level utility function $\widehat{U}(\mathbf{R})$ from the service-level utility function $U(\mathbf{S},\mathbf{D})$.



Since shifting resources among different Application Environments may entail substantial delays, the Application Manager uses a ***Demand Forecaster*** to estimate the average future demand $\mathbf{D}'$ over an appropriate time window (e.g., up until the next reallocation), based on the historical observed demand **D** received from the Data Aggregator.

The Demand Forecaster may use time series analysis methods, supplemented by special knowledge of the typical usage patterns of the application.



The ***Utility Calculator*** computes the optimal resource-level utility $\widehat{U}(\mathbf{R})$ that could be obtained based on the forecasted demand $\mathbf{D}'$.

Given the performance model $\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D})$, and the service-level utility function $U(\mathbf{S},\mathbf{D})$, the Utility Calculator computes

$$\widehat{U}(\mathbf{R}) = \max_{\mathbf{C}} U(\mathbf{S}(\mathbf{C}, \mathbf{R}, \mathbf{D}'), \mathbf{D}')$$

for all possible resource levels $\mathbf{R}$.

The ***Controller*** computes the control parameters $\mathbf{C}^*$ that optimize $U(\mathbf{S},\mathbf{D})$ based on the performance model and ***current*** resource level:

$$\mathbf{C}^* = \underset{\mathbf{C}}{\operatorname{argmax}}\, U(\mathbf{S}(\mathbf{C},\mathbf{R}_t,\mathbf{D}),\mathbf{D}) \qquad (1)$$

while the ***Utility Calculator*** computes

$$\widehat{U}(\mathbf{R}) = \max_{\mathbf{C}} U(\mathbf{S}(\mathbf{C},\mathbf{R},\mathbf{D}'),\mathbf{D}') \qquad (2)$$

for ***all*** possible resource levels **R**.

To compute $\widehat{U}(\mathbf{R})$ requires repeated computation of (2) using each possible resource level **R**, rather than just the current resource level $\mathbf{R}_t$, and with the predicted demand $\mathbf{D}'$, rather than the current demand **D**.

With complex applications, it may be difficult for human developers to determine an accurate ***performance model*** *a priori*. To address this problem, the Application Manager can have a ***Modeler*** module that employs ***inference*** and ***learning algorithms*** to create, update, and revise the performance model based upon joint observations of $(\mathbf{S},\mathbf{C},\mathbf{R}_t,\mathbf{D})$.

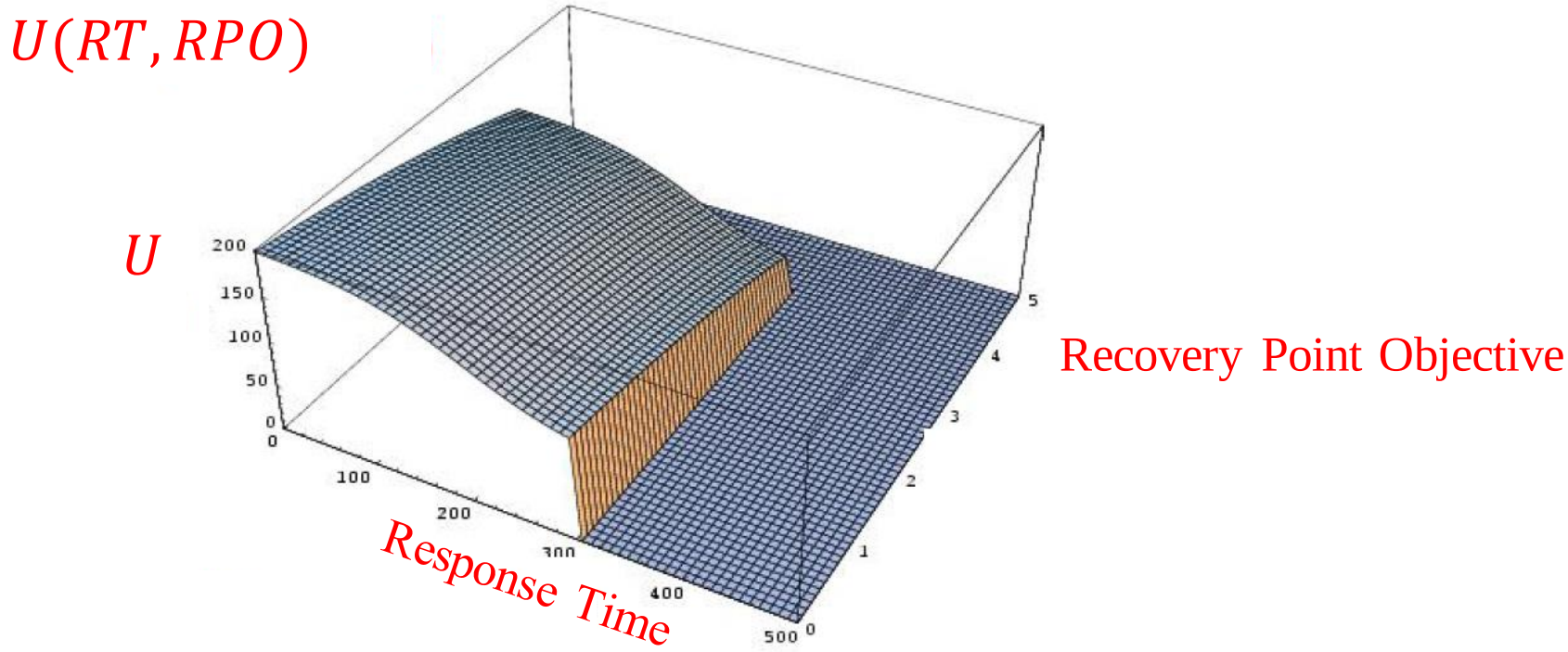
Example of management with high level policies by one Application Manager

1. Elicit utility function $U(\mathbf{S})$ expressed in terms of service attributes **S**.

$$\mathbf{S} = [S_1, S_2] = [RT, RPO]$$

where RT = ***Response Time*** (in ms)

and RPO = ***Recovery Point Objective***, time interval (in hours) specified by the Business Continuity team to be the longest time the business can allow for without incurring significant risks or significant loss.



2. Model how each attribute S_i depends on controls **C** and observables **D**

- Models expressed as $\mathbf{S}(\mathbf{C}, \mathbf{D})$

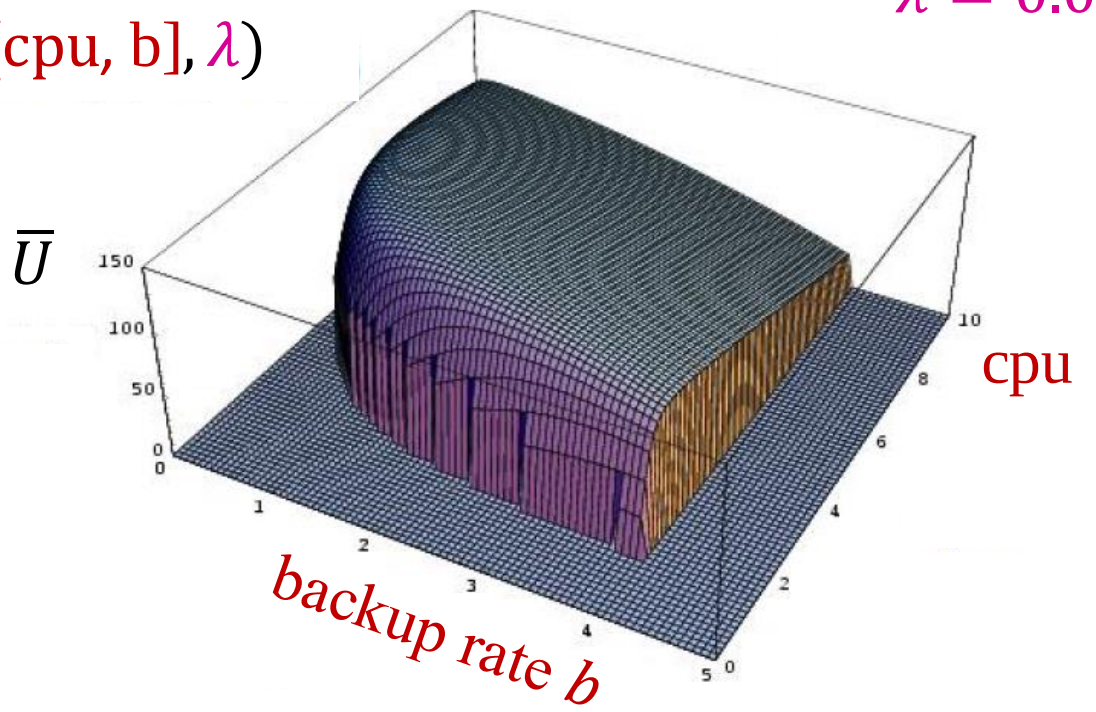
  e.g. $S_1 = RT$(routing weights, request rate)

- Models from experiments, learning, theory

3. Transform from service utility U to resource utility $\bar{U}$ by substitution

$$U(\mathbf{S}) = U\big(\mathbf{S}(\mathbf{C}, \mathbf{D})\big) = \bar{U}(\mathbf{C}, \mathbf{D})$$

$\bar{U}([\text{cpu, b}], \lambda)$ $\quad \lambda = 0.01$



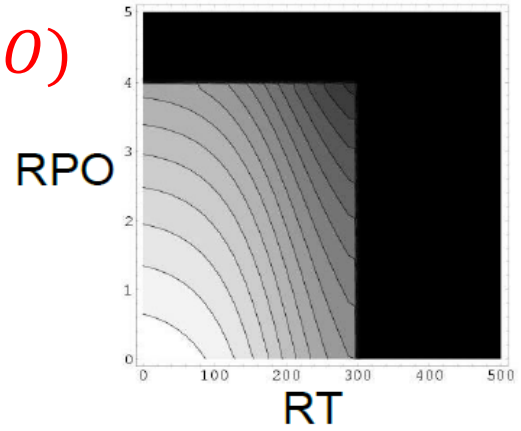
4. Optimize resource utility.

As observable **D** changes, set **C** to values that maximize $\bar{U}(\mathbf{C}, \mathbf{D})$

$$\mathbf{C}^*(\mathbf{D}) = \operatorname{argmax}_{\mathbf{C}} \bar{U}(\mathbf{C}, \mathbf{D})$$

$$\hat{U}(\mathbf{D}) = \max_{\mathbf{C}} \bar{U}(\mathbf{C}^*(\mathbf{D}), \mathbf{D})$$

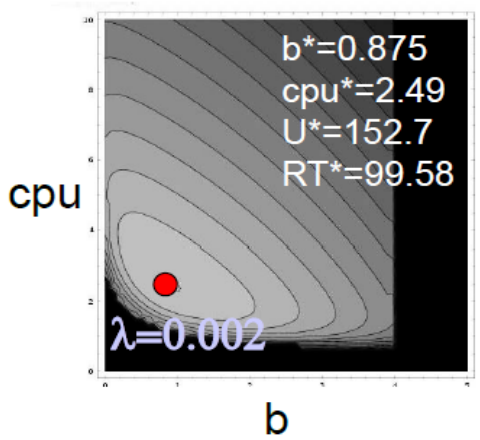
$U(RT, RPO)$

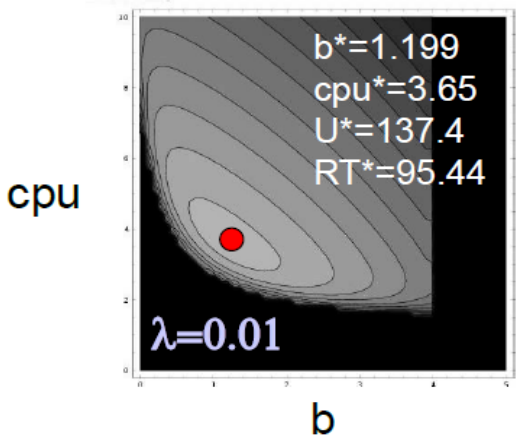


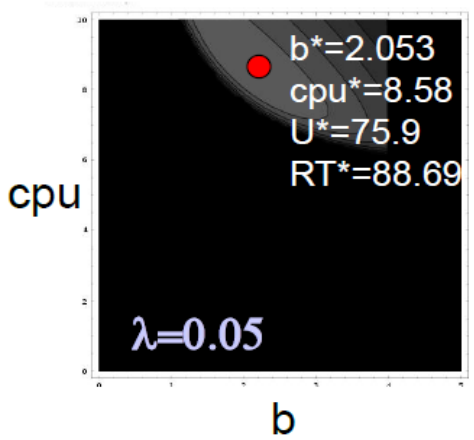
Even if service-level utility remains fixed, resource-level utility depends upon environment.

Thus system responds to environmental changes.

$\bar{U}([\text{cpu, b}], \lambda)$





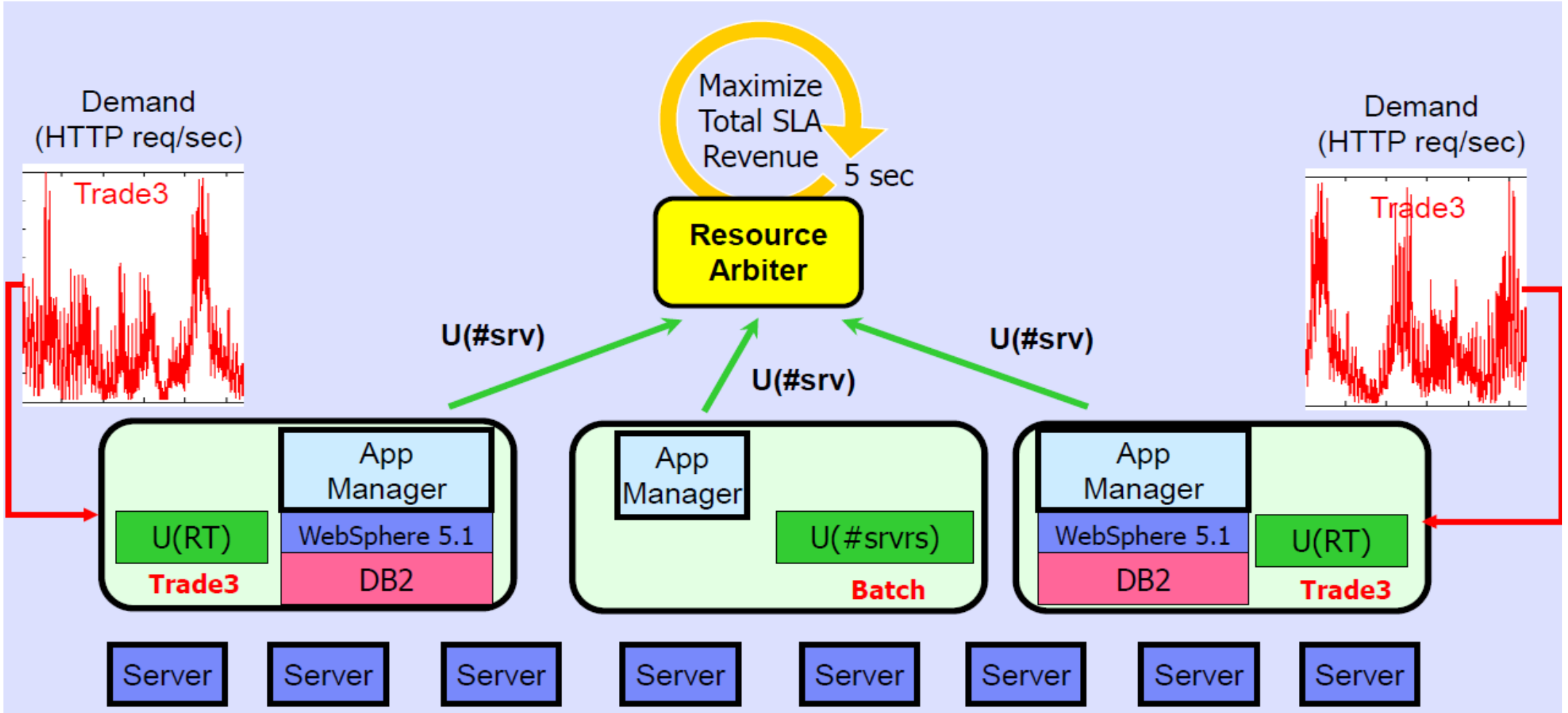


Example of resource allocation for several Application Environments

With ***several Application Environments***, need to make resources explicit.

R vector of resource levels.

R_i = number of servers for application environment i



Example of resource allocation for several Application Environments

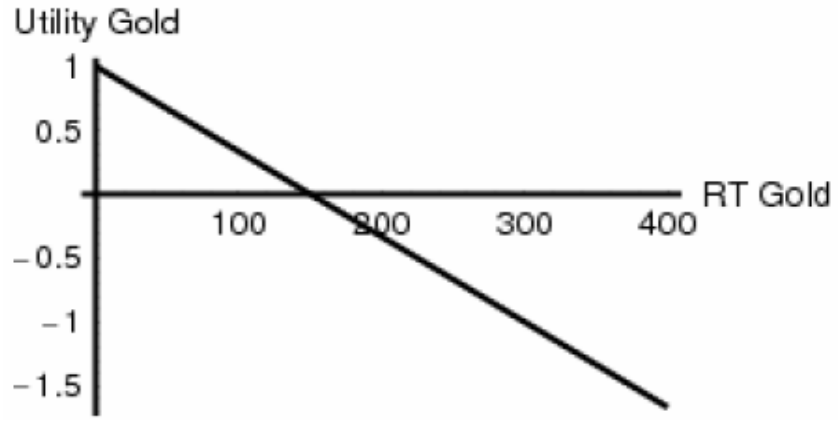
WAS XD Configuration by Administrator

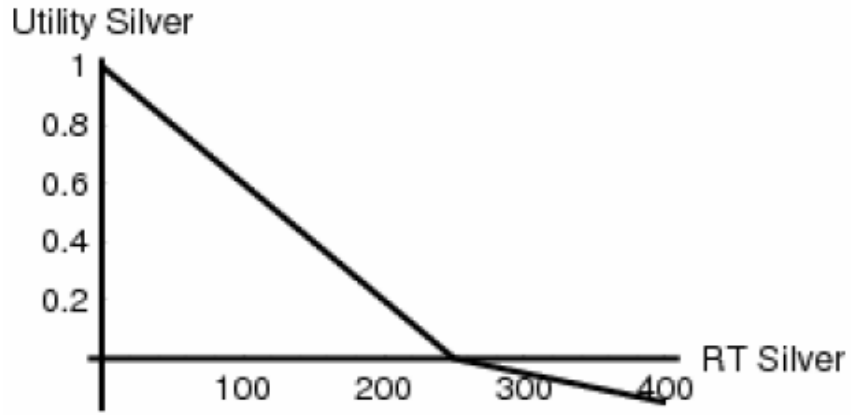
WAS : Websphere Application Server

Utility Function Specification

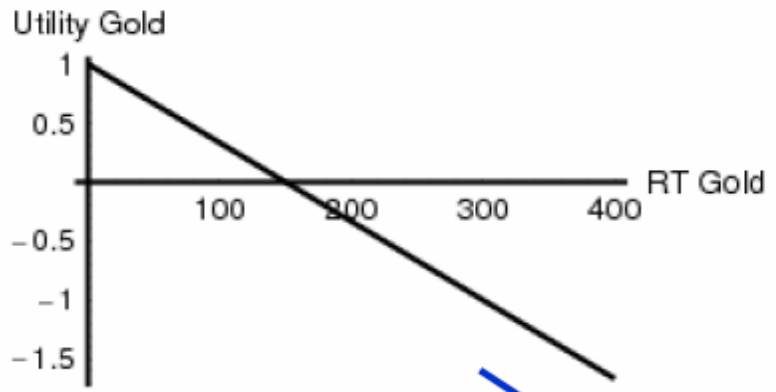
XD Gold
- Target RT = 150 ms
- Importance = 1

XD Silver
- Target RT = 250 ms
- Importance = 50

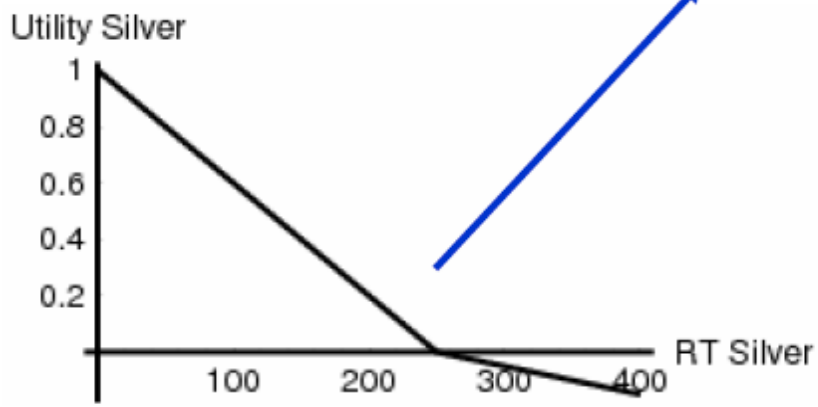


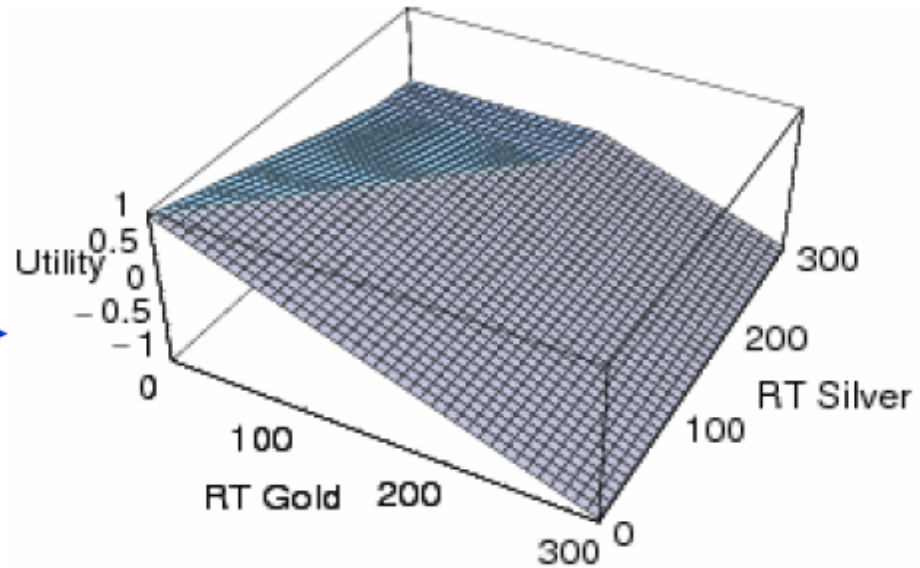


WAS XD Utility Function Combination



$$\min(U_G, U_S)$$

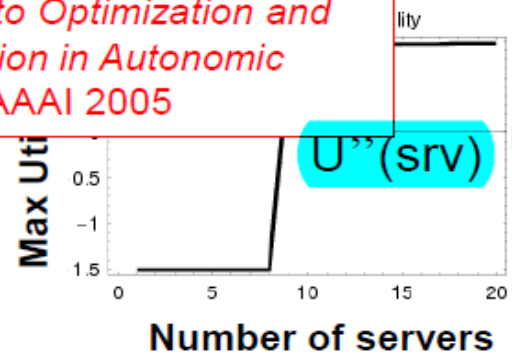


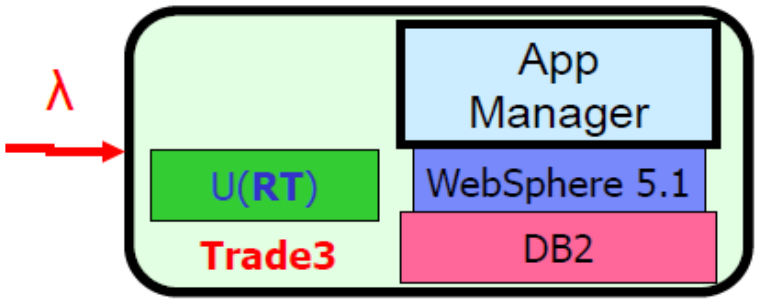


How App Mgr computes its external resource utility

Alternative to generating full curve: utility elicitation

Patrascu, Boutilier et al. *New Approaches to Optimization and Utility Elicitation in Autonomic Computing*, AAAI 2005





Resource Arbiter

Elicit: $U(RT)$ — Service-level utility

Model: $U(RT(C; srv, \lambda))$ — My controls: C; Arbiter's controls: srv; Observable: λ

Transform: $U'(C; srv, \lambda) = U(RT(C; srv, \lambda))$ — Internal resource-level utility

Optimize:

Optimal internal control settings: $C^*(srv, \lambda) = \text{argmax}_C U'(C; srv, \lambda)$

External resource-level utility: $U''(srv, \lambda) = U'(C^*(srv, \lambda); srv, \lambda)$

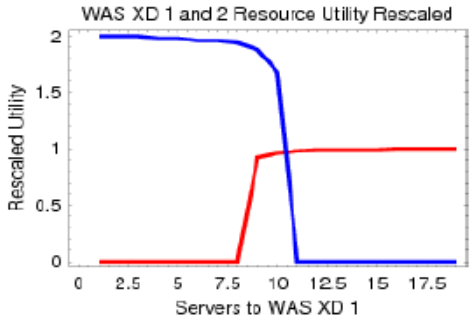
How the Arbiter determines optimal resource allocation

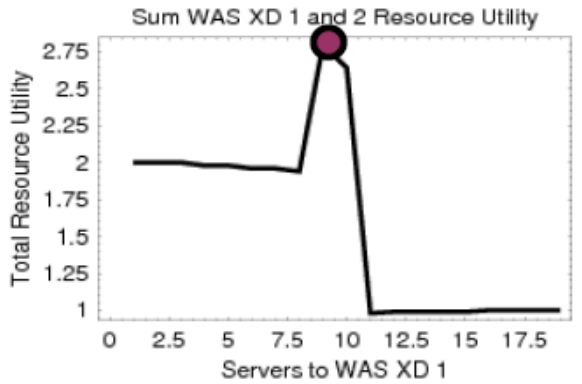
Decision problem:

Allocate resources

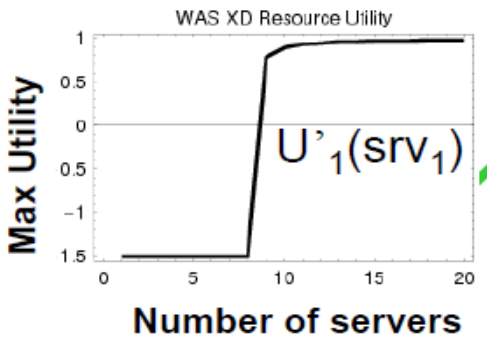
$\mathbf{srv}^* = \text{argmax}_{\mathbf{srv}} \Sigma U''_i(srv_i)$

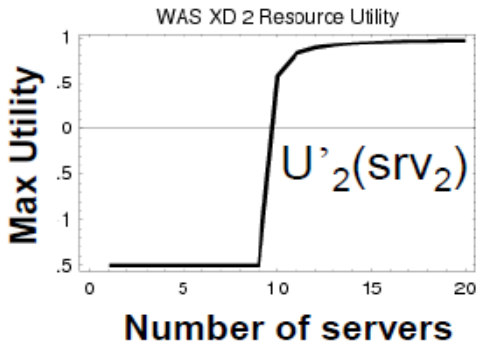
Effectively maximizes $\Sigma U_i(\mathbf{S}_i)$

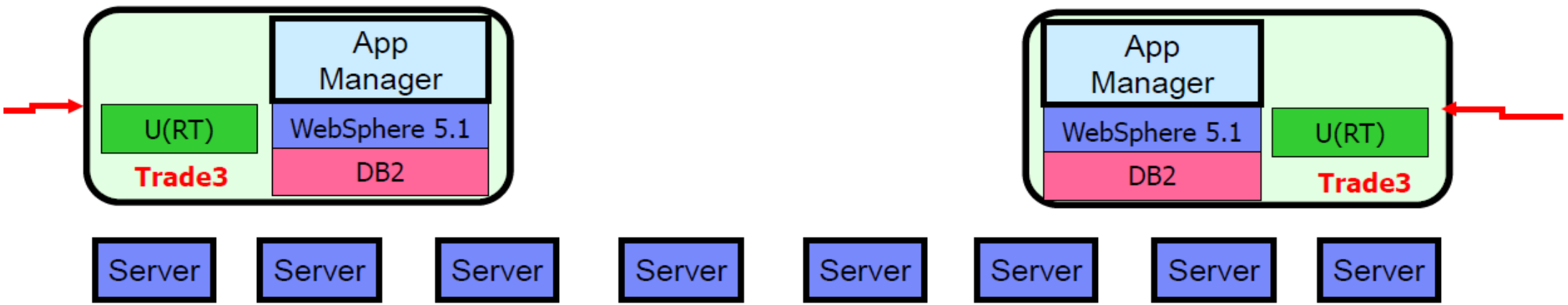




Resource Arbiter

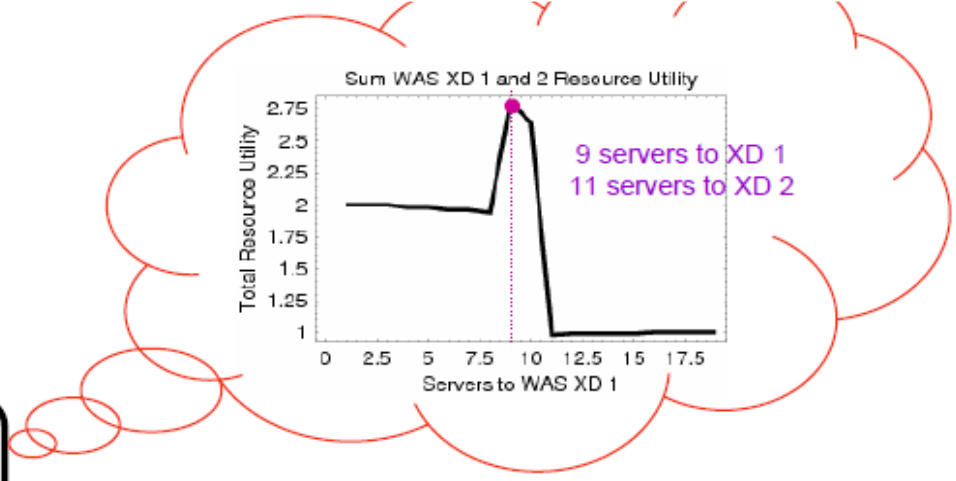




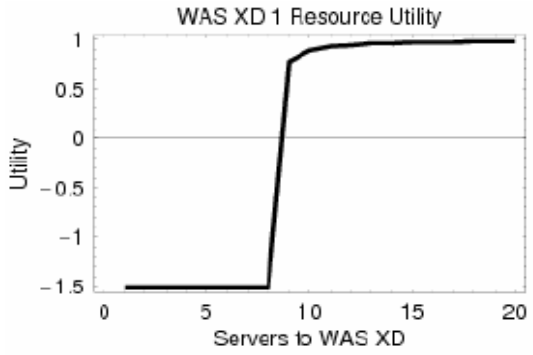


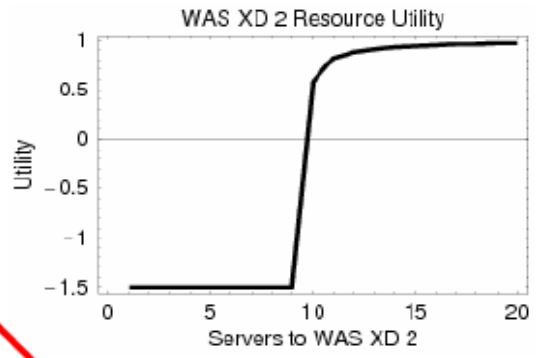
Resource Utility Functions

TIO : Tivoli Intelligent Orchestrator



TIO Resource Arbiter

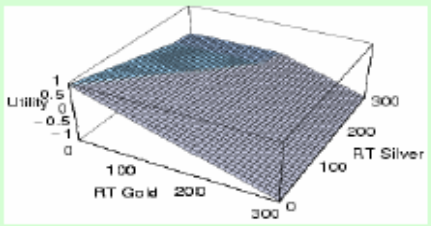




WAS XD Installation 1

XD 1 Gold
- Target RT = 150 ms
- Importance = 1

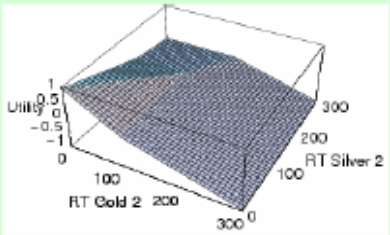
XD 1 Silver
- Target RT = 250 ms
- Importance = 50



WAS XD Installation 2

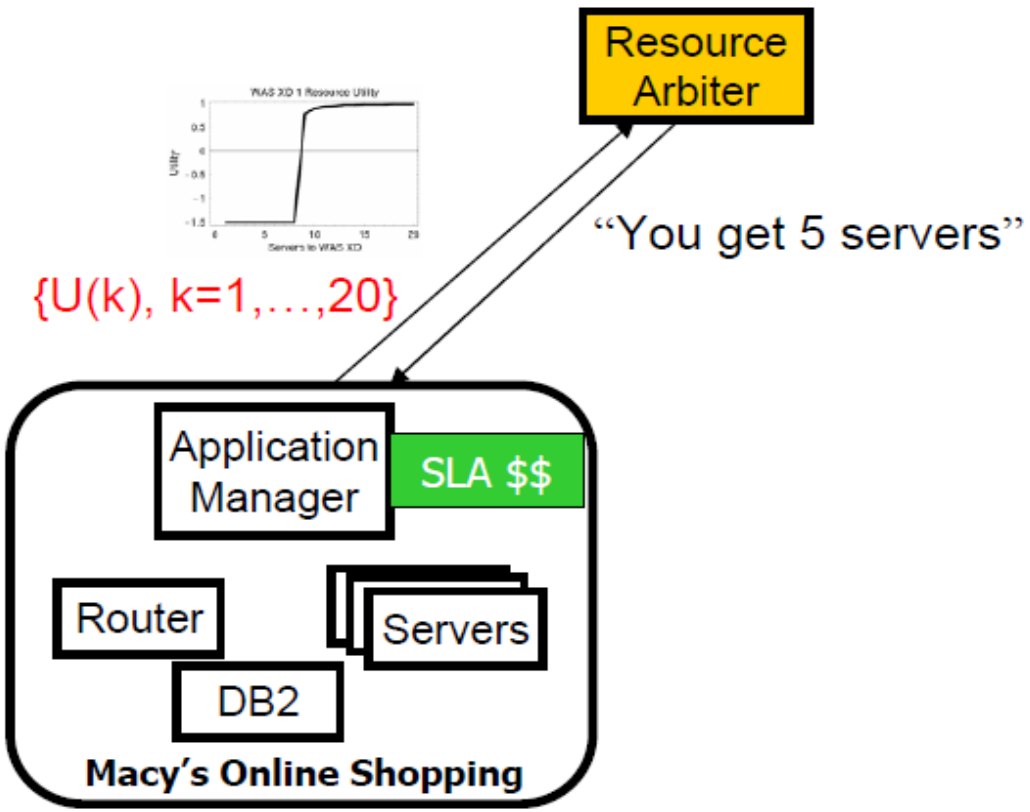
XD 2 Gold
- Target RT = 100 ms
- Importance = 5

XD 2 Silver
- Target RT = 200 ms
- Importance = 25

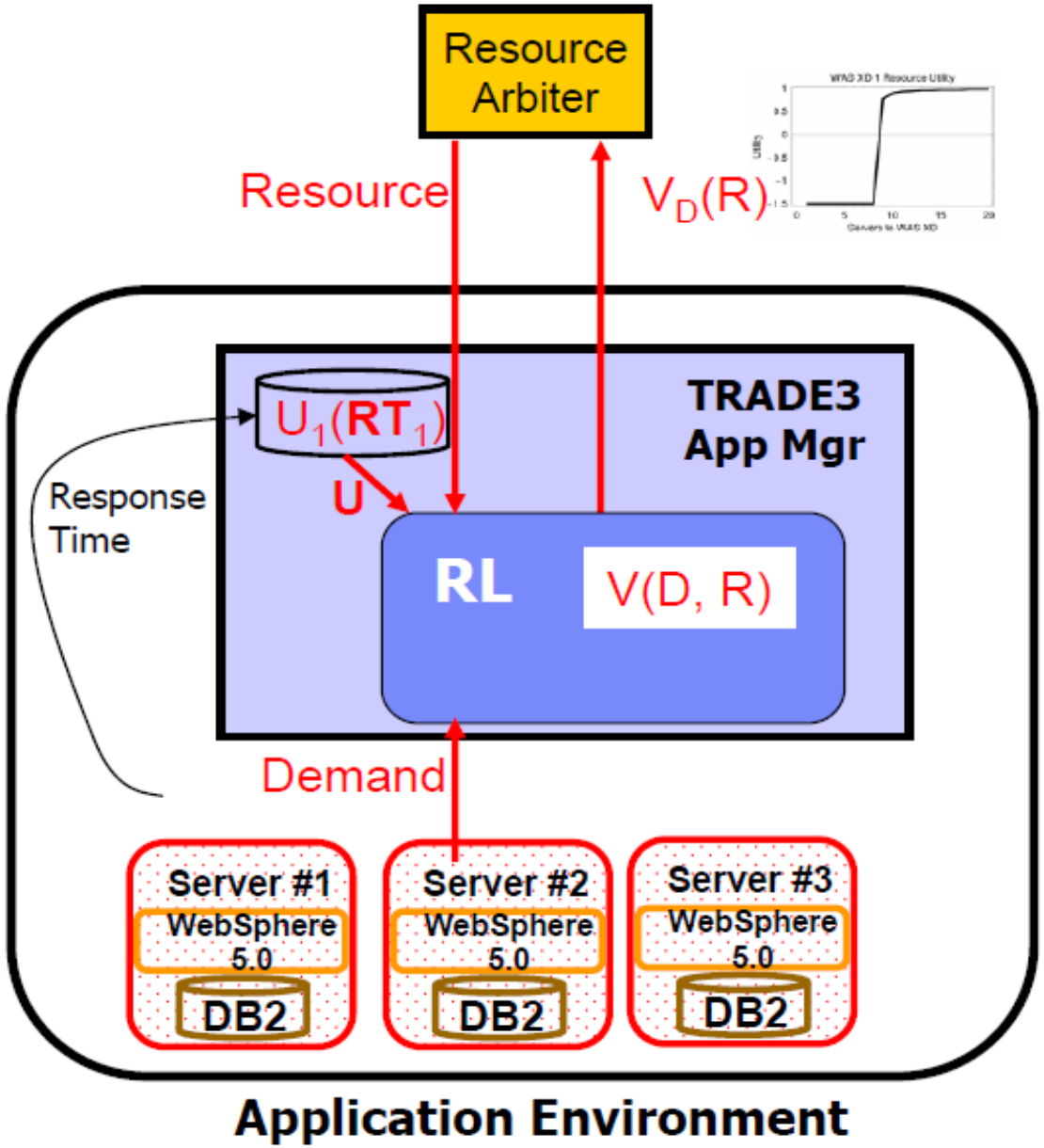


Approach 1: Performance Modeling using Queuing Theory

- Application estimates how extra/less resource would affect performance
  - Apply an appropriate queuing model (e.g. M/M/k); estimate its parameters
  - Use model to predict new steady-state if amount of resource changes

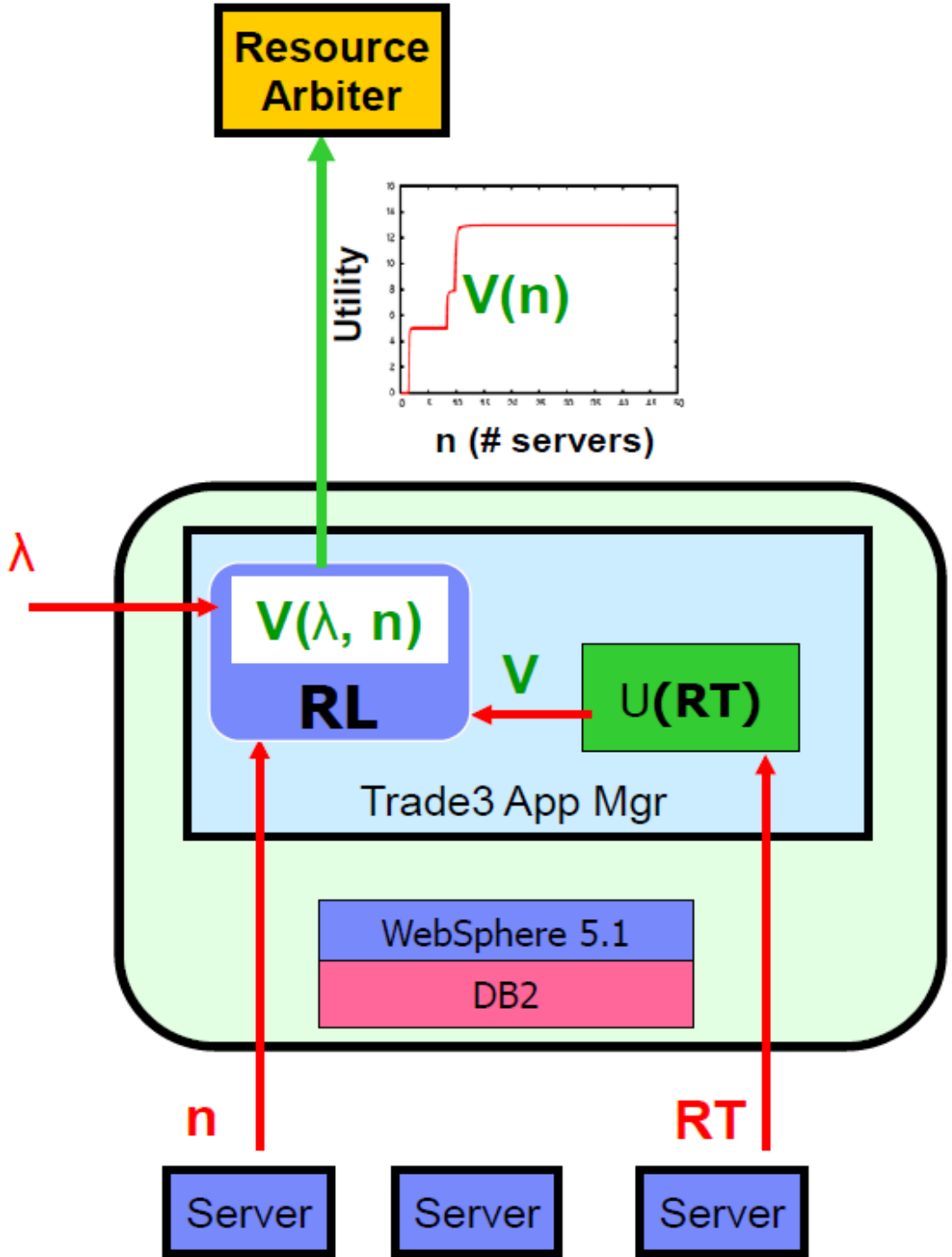


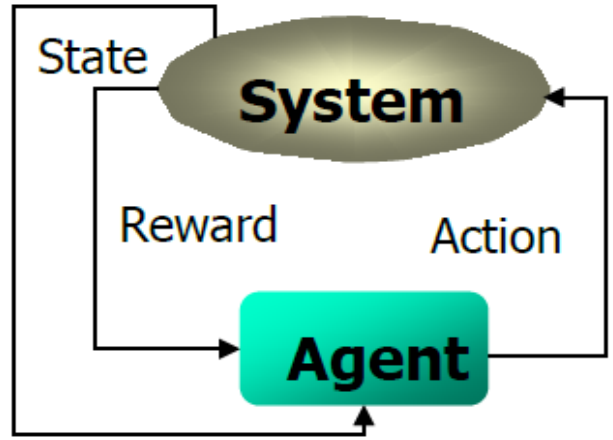
Approach 2: Local Reinforcement Learner in each Application Manager



- RL learns by observation how Value depends on Demand and Resource (# servers)
- Learns *long-range* expected value function **V(state, action) = V(D, R)**

- Several theoretical and practical issues
  - Will learning converge?
    - Multiple learners
    - Non-Markov
  - Is learning fast enough?
  - Exploration penalties

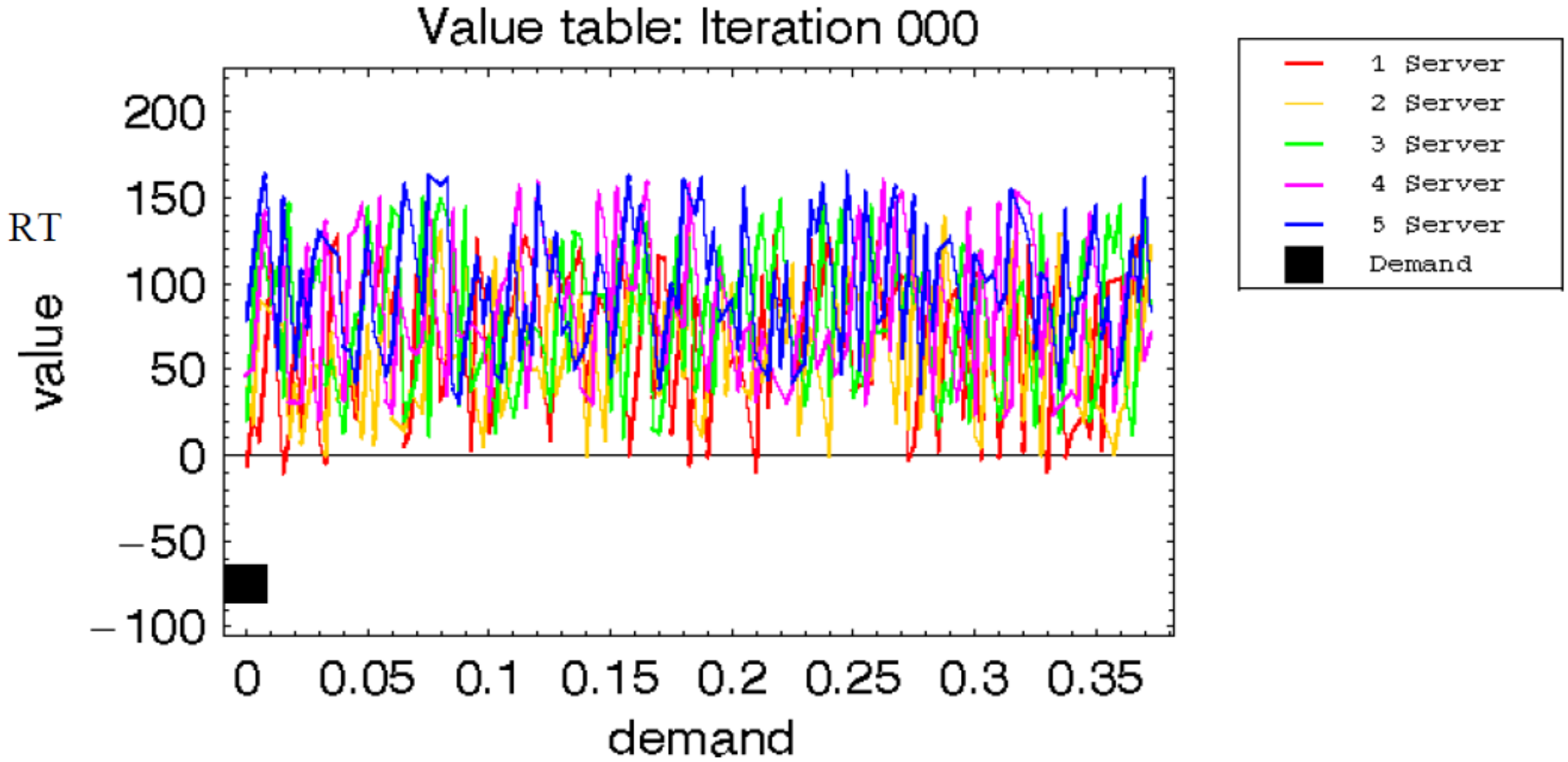




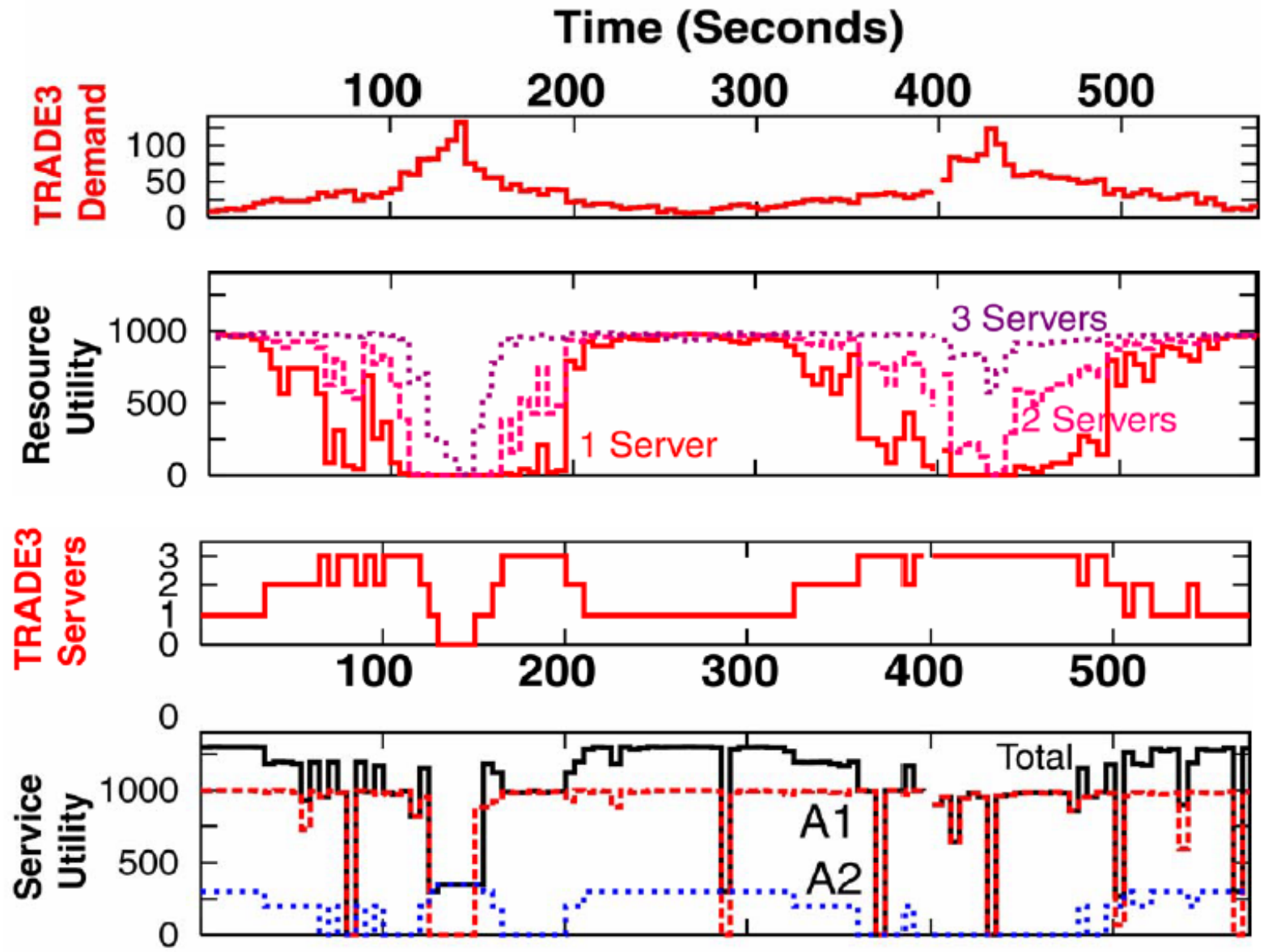
- App Mgr can use reinforcement learning (**RL**) to compute external resource utility
  - State = λ *demand*
  - Action = n *# servers*
  - Reward = V(RT) *SLA payment*
- It learns *long-range* value function V(state, action) = V(λ, n)
- It reports V(n) for current or predicted value of λ

RL Works!

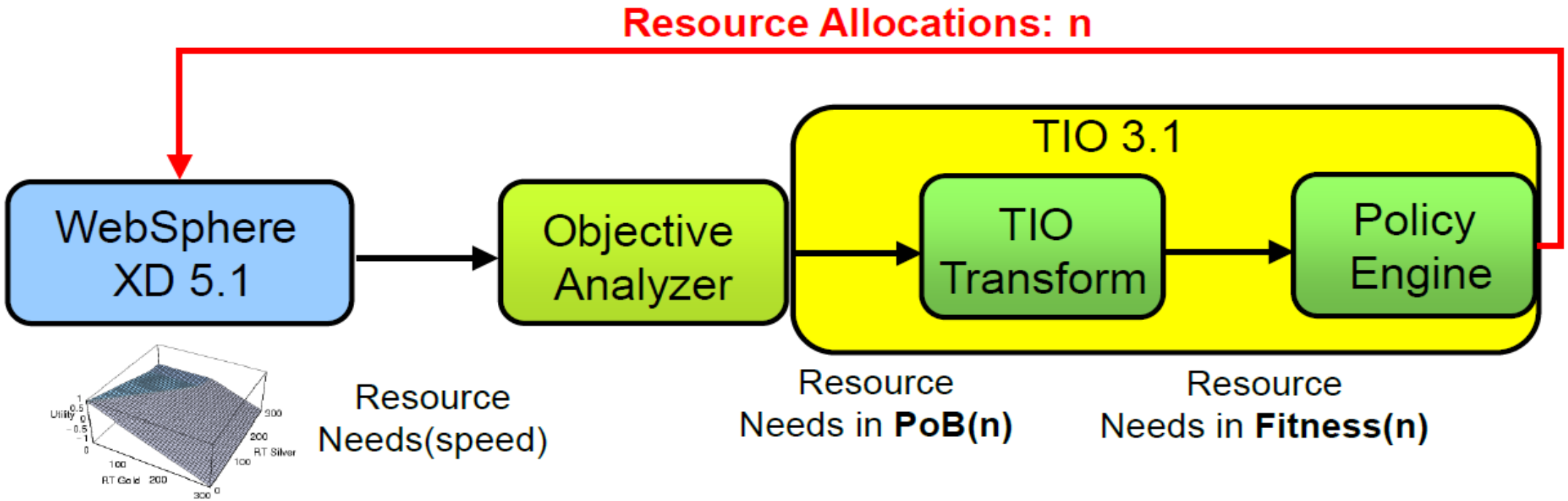
Results of overnight training (~25k RL updates = 16 hours real time) with random initial condition



Resource Allocation Results



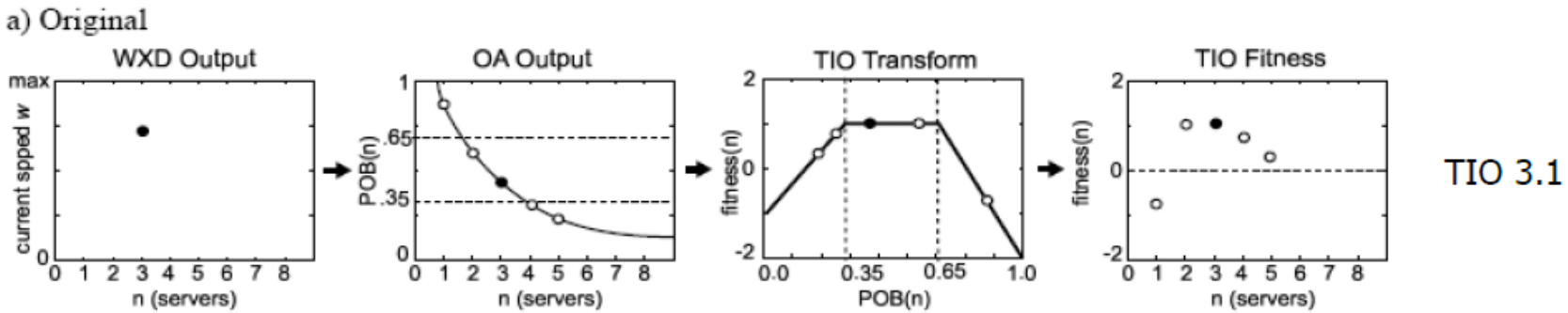
Utility-based Interactions between WXD and TIO: Step 1



Utility(current n)

Original

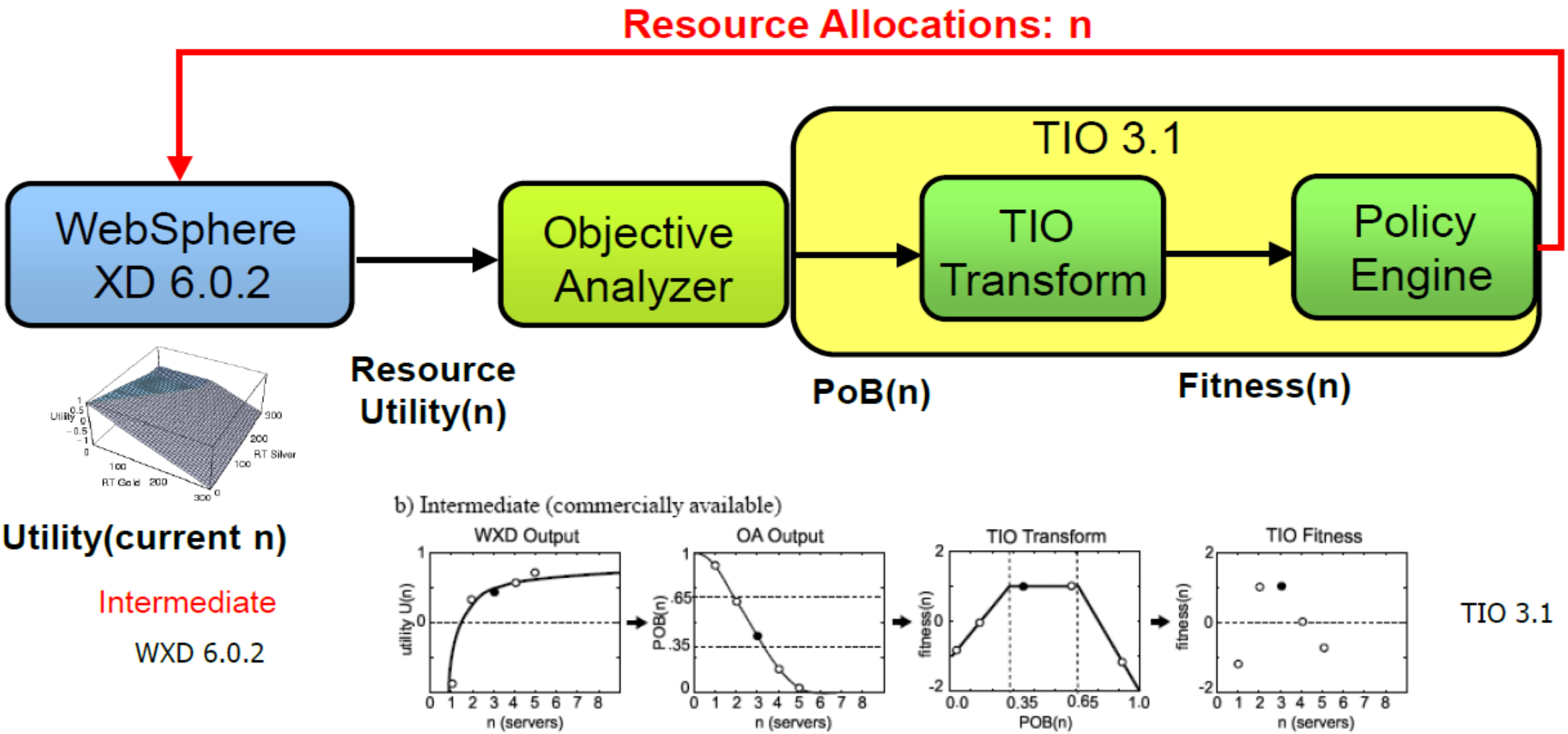
WXD 5.1



TIO 3.1

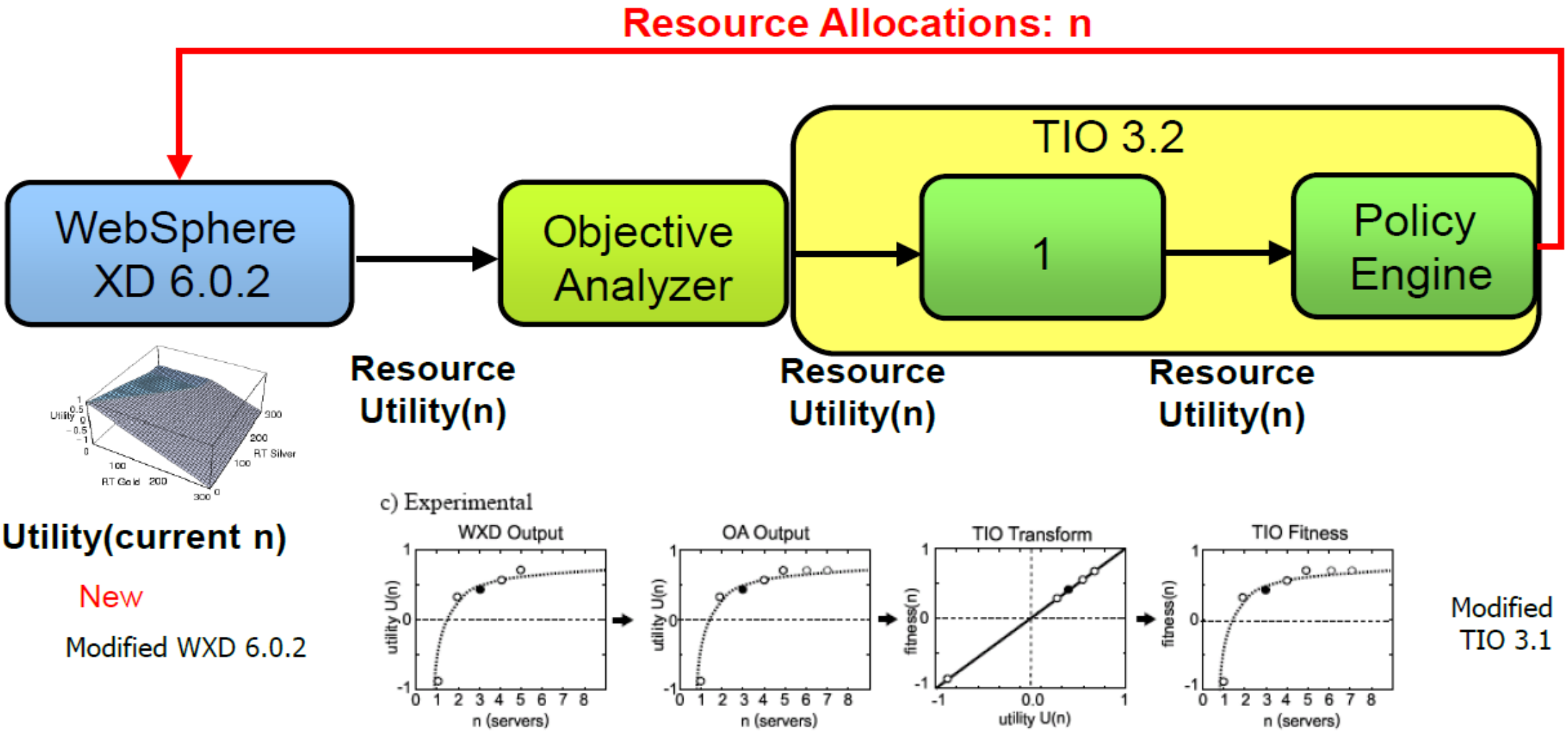
- TIO cannot make well-founded resource allocation decisions
  - WS XD can't articulate its needs to TIO
  - PoB not commensurate with utility

Utility-based Interactions between WXD and TIO: Step 2

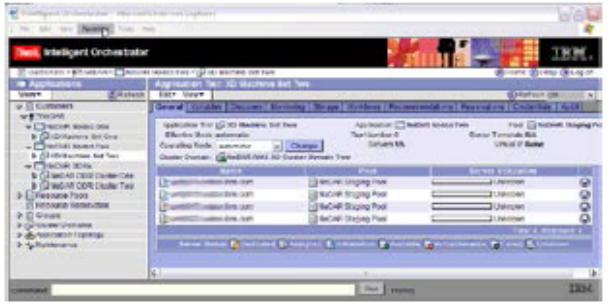


- WS XD research team added ResourceUtil interface of WXD
- We developed a good heuristic for converting ResourceUtil to PoB in Objective Analyzer
  - Interpolate discrete set of ResourceUtil points and map to PoB
  - This PoB better reflects WS XD's needs

Utility-based Interactions between WXD and TIO: Step 3



- We modified TIO to use ResourceUtil(n) directly instead of PoB(n)
- Most mathematically principled basis for TIO allocation decisions
- It enables TIO to be in perfect synch with the goals defined by WS XD
- Basic scheme can work, not just for XD, but for any other entity that may be requesting resource, provided that it can estimate its own utilities



Tivoli Intelligent Orchestrator

If I give you *n* servers, how often will you exceed the response time goal?

I need 300M CPU cycles/sec

If I give you *n* servers, how valuable will that be?

V(n)

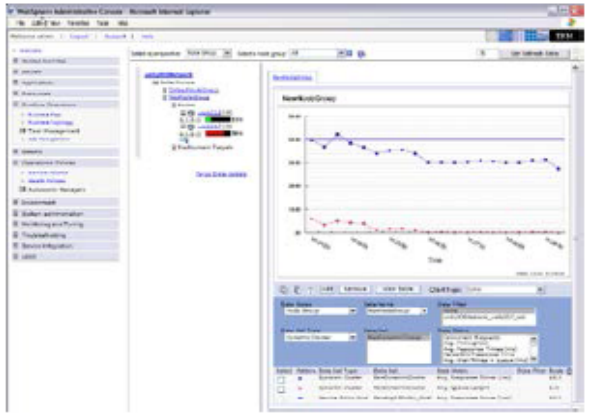
U(**S**)

WebSphere Extended Deployment

The confusing old way

The clean new way

To commercialize this solution, infuse agency/autonomicity gradually into existing products and demonstrate value incrementally at each step



This was not actually that simple – product release cycles didn't mesh, so we needed an evolutionary approach.

Utility Functions in Autonomic Systems - Recapitulation

An ***autonomic*** computing system must optimize its own behavior in accordance *with* ***high-level guidance*** from humans, and hence have the capability of ***self-optimization***.

- What form should this guidance take?
- What mechanisms should the system employ to translate this guidance into low-level actions that achieve the desired optimization objective?

In order to dynamically allocate system resources, the administrators of an autonomic computing system ***no longer*** have to ***ascribe value to low-level resources*** or to use simple standard mappings between resources and quality of service (while in a real data center mappings from resource to QoS can be arbitrarily complex and application specific).

Utility functions are used by the administrators to specify utility in ***high-level business terms***: the ***service-level attributes*** that matter to them or their customers, such as end-to-end response time, latency, throughput, etc.

Utility functions provide the ***objective function*** for self-optimization in autonomic computing systems, by mapping each possible ***state*** of an entity (an autonomic system or component) into a real ***scalar value***:

- the ***state*** can be described as a vector of attributes measured directly by or synthesized from sensor measurements,
- the ***value*** may be expressed in any suitable unit (typically a monetary unit),
- the ***utility function*** might be specified by a human administrator, derived from a contract, or derived from another utility function.

Given a utility function, the system or component must use an ***appropriate optimization technique*** in conjunction with a ***system model*** to determine the most valuable feasible state and the means for achieving it.
Typically, these means may include tuning system parameters or reallocating.
Since conditions are constantly changing, the optimization ought to be performed recurrently.

Utility functions provide a natural and advantageous framework for achieving self-optimization in distributed autonomic computing systems.

The computing system has a distributed architecture which enables, by means of utility functions, a collection of autonomic elements to continually optimize the use of computational resources in a dynamic, heterogeneous environment.

The architecture is a ***two-level*** structure of independent autonomic elements that supports flexibility, modularity, and self-management:

- ***Individual autonomic elements*** manage application resource usage to optimize local *service-level utility functions*, and
- a ***global Arbiter*** allocates resources among application environments based on *resource-level utility functions* obtained from the managers of the applications.

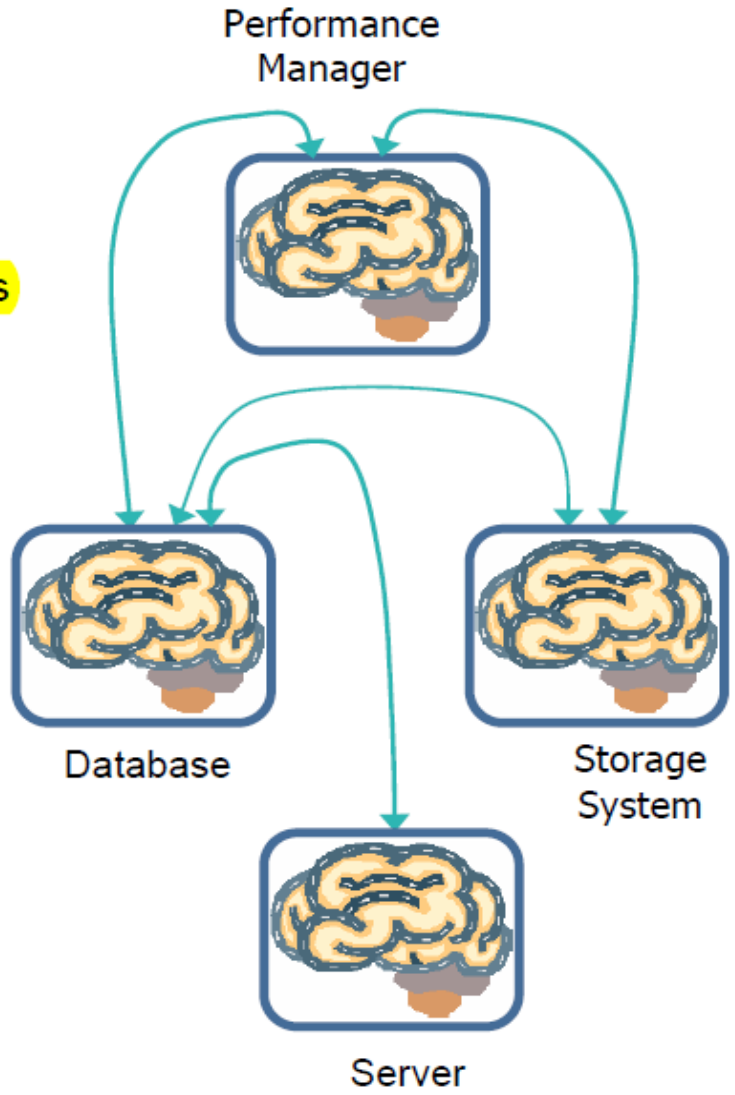
The scheme supports ***multiple heterogeneous services*** by encapsulating their differences at a local level and providing a uniform means of communicating resource needs to a resource arbiter.
The form of communication is a resource-level utility function that is derived locally from the service-level utility function by optimization algorithms coupled with a model.

This scheme has been used to handle Web-based, fluctuating, transactional workloads running on a Linux cluster.

Other Self-* Properties

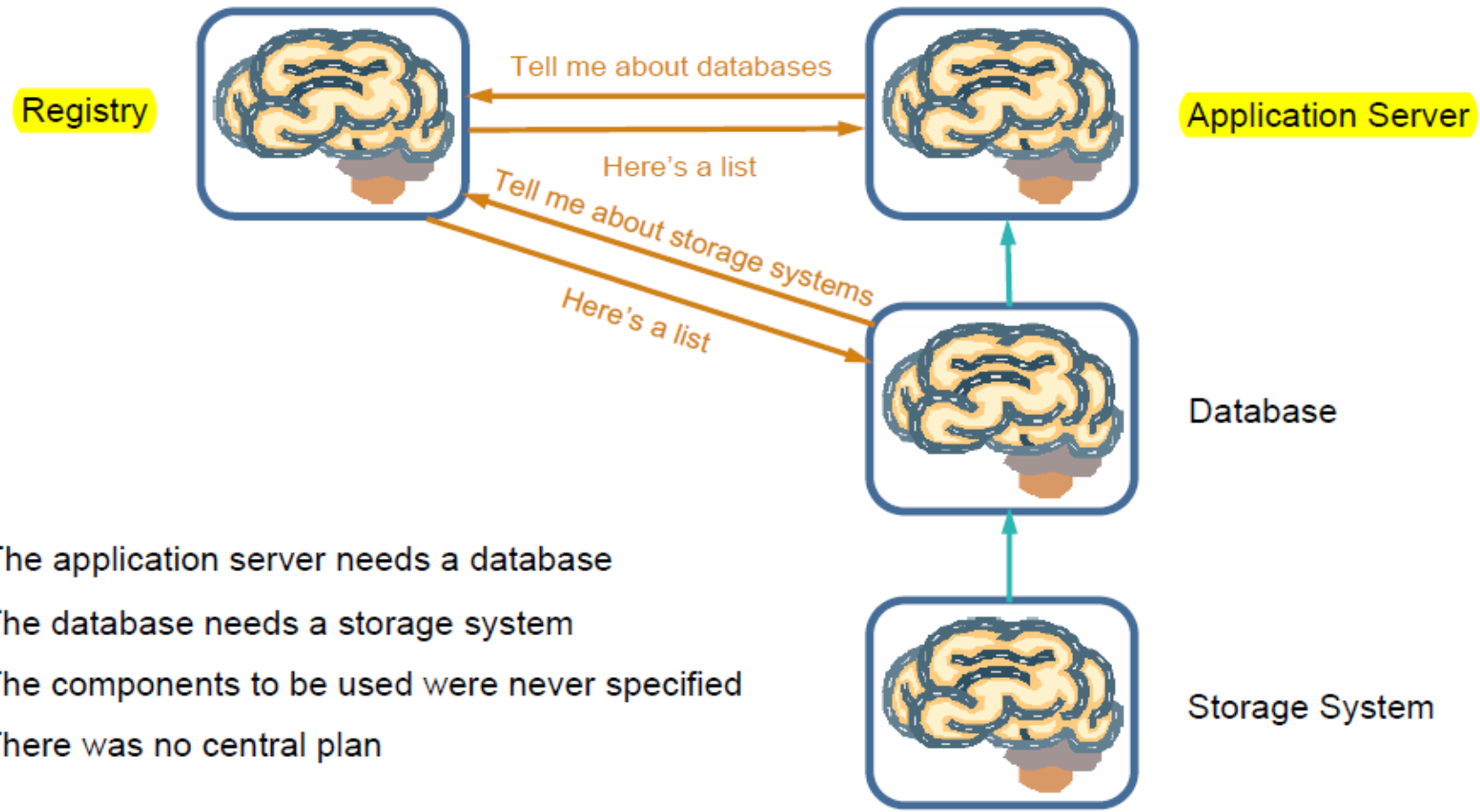
Multi-agent System Architecture

- Autonomic elements are IT components that:
  - Manage their own low-level behavior in accordance with
    - policies, agreements, management relationships
  - Establish and honor service agreements with other elements
- System-level autonomic behavior arises from:
  - Interactions (service-oriented, agent-oriented)
    - Founded on Web Services, Grid Services
  - System integration components (registries, sentinels, …)
  - System design patterns
- Interactions and agreements are, in general:
  - Dynamic, flexible in pattern



Goal-Driven Self-Assembly

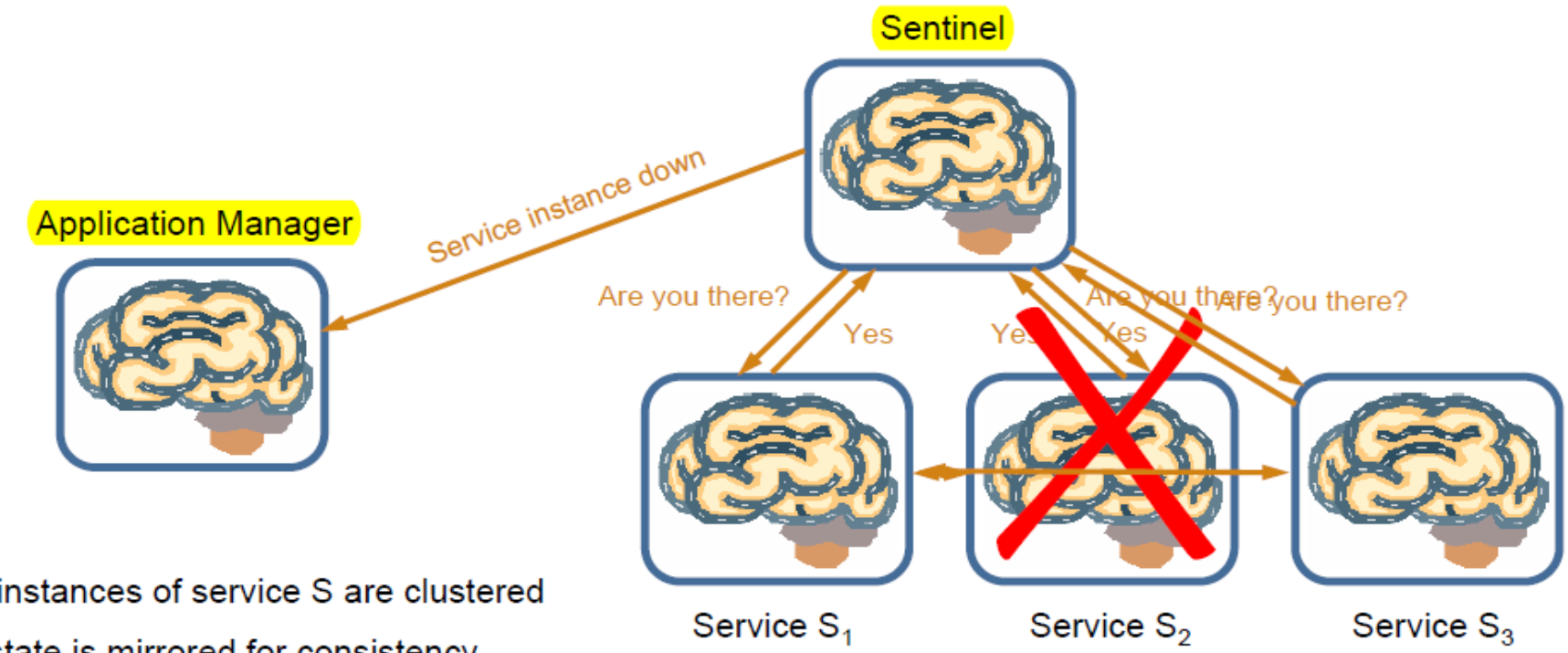
A Design Pattern for Self-Configuration in Autonomic Systems



- The application server needs a database
- The database needs a storage system
- The components to be used were never specified
- There was no central plan

Self-Healing Clusters

A Design Pattern for Self-Healing in Autonomic Systems



- Multiple instances of service S are clustered
  - Their state is mirrored for consistency
  - A sentinel monitors their availability
- If an instance goes down …
  - The sentinel notifies the application manager
  - The application manager arranges for a new instance of S
  - The new instance is integrated into the cluster
  - … and the sentinel begins monitoring it

Perspectives - Challenges

Challenge: Learning

Generic AE+AS technologies

Establish theoretical foundation for understanding and performing learning and optimization in multi-agent systems.

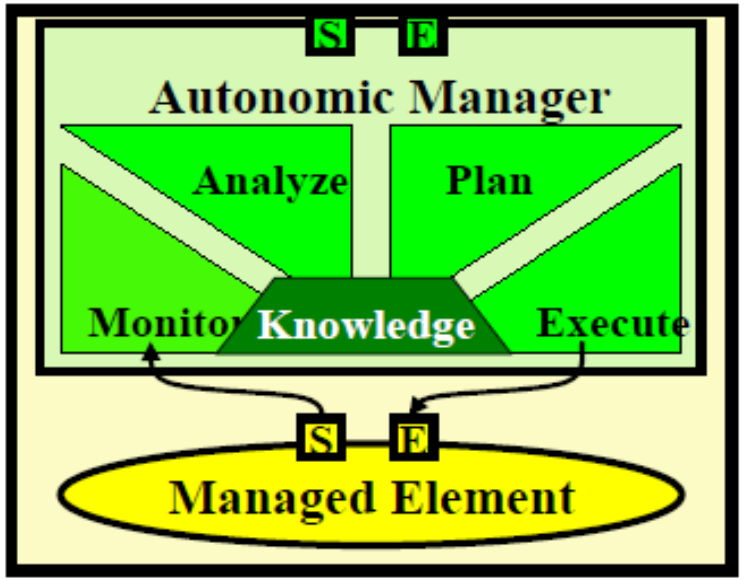
- Single element level
  - AE needs to learn a model of itself and environment quickly
  - Deal with noisy, dynamic environments
  - On-line, so exploration of parameter space can be costly and/or harmful
  - Cope with several dozens to hundreds of tunable parameters
- System level
  - Multi-agent system: several interacting learners
  - What are good learning algorithms for cooperative, competitive systems?
    - What are conditions for stability?
    - What is sensitivity to perturbations?

Challenge: Architecture

AE+AS architectures

Define set of fundamental architectural principles from which self-* emerges

- **AE level**: Coordinate multiple threads of activity
  - AE's live in complex environments
  - Multiple task instances and types
    - Concurrent, asynchronous
  - Multiple interacting expert modules
  - Conflict resolution



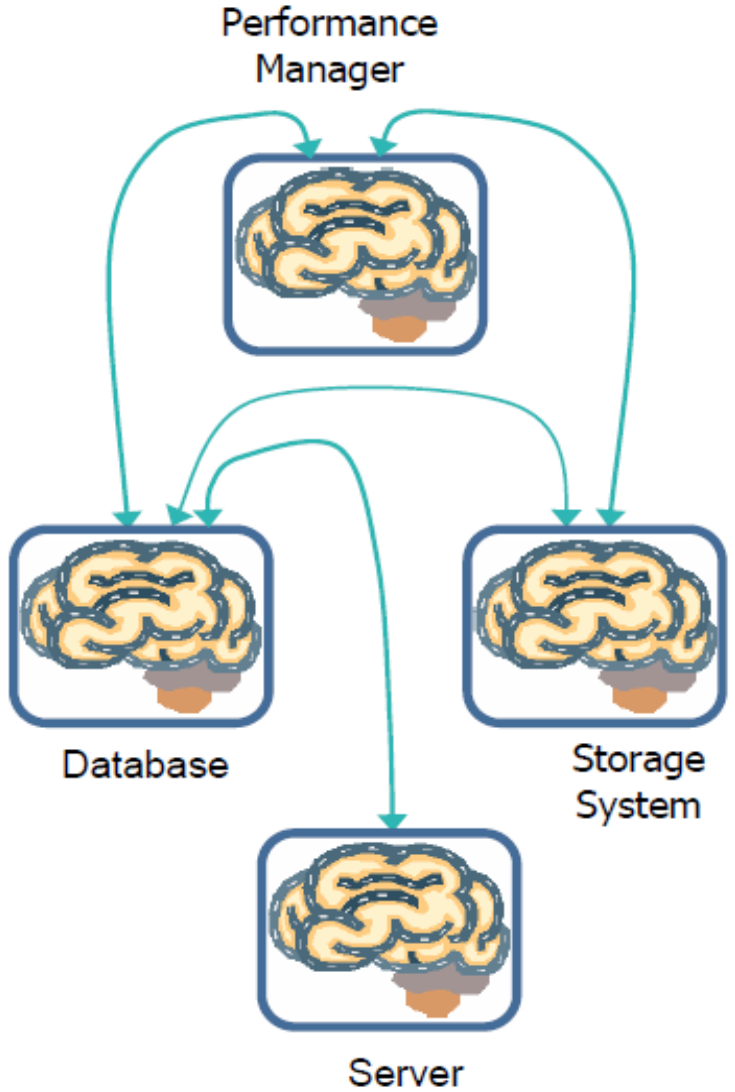
An Autonomic Element

- **System level**: Enable more flexible, service-oriented patterns of interaction
  - How decentralized can/should we make it?
  - Multi-agent architecture
    - Representing and reasoning about needs, capabilities, dependencies

Challenge: Negotiation

Generic AS technologies, AS science

- Develop and analyze
  - Methods for expressing or computing preferences
  - Negotiation protocols
  - Negotiation algorithms
- Establish theoretical foundation for negotiation
  - Explore conditions under which to apply
    - Bilateral
    - Multi-lateral (mediated, or not)
    - Supply-chain
  - Study how system behavior depends on mixture of negotiation algorithms in AE population



Challenge: Control and Harness Emergent Behavior

AS science

- Understand, control, exploit emergent behavior in autonomic systems
  - How do self-*, stability, etc. depend on
    - Behaviors and goals of the autonomic elements
    - Pattern and type of interactions among AEs
    - External influences and demands on system
  - Invert relationship to attain desired global behavior
    - How?
    - Are there fundamental limits?

- Develop theory of interacting feedback loops
  - Hierarchical
  - Distributed

Challenge: Policy and Human-System Studies

Human interface

- Human interface
  - How do/could sysadmins work; what do they need
  - Authoring and understanding policies
  - "What-if" analyses
  - Avoiding or ameliorating specification errors
  - Iterative elicitation of preferences, tradeoffs
- Universal representation and grammar
  - Many different application domains, disciplines
  - Connections among rules, goals, utility functions?
- Algorithms that operate upon policies
  - Derive lower-level policies from high-level policies
  - Derive actions from goals (e.g. planning, optimization)
- Conflict detection, resolution
  - Both design time and run time
  - Protocols, interfaces, algorithms

"IF (workload > 10/sec) THEN (Add CPU)"

"Avg RT < 200 msec"