Techniques d'adaptation
de l'intergiciel et des applications

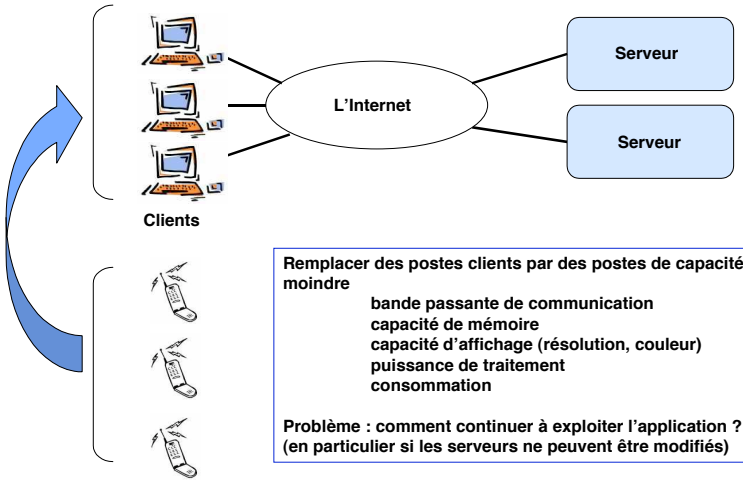
Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/people/krakowia

Besoins des applications : quelques exemples

- **Objectif commun**
  - Préserver différents aspects de la qualité de service …
    - Performances
    - Sécurité
    - Tolérance aux fautes
  - … dans un environnement variable
    - Capacités des supports
    - Conditions de communication
    - Spécifications du service
- **Principes de solution**
  - Utlisation du *middleware* pour l'adaptation

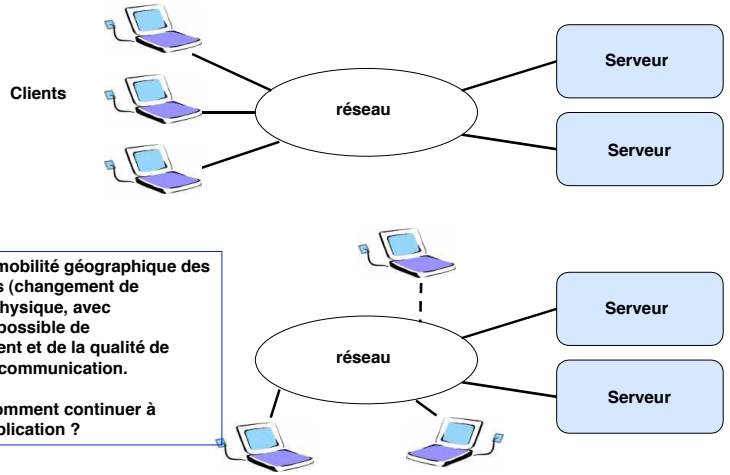
Exemple 1
Adaptation de services en fonction des capacités du poste client



Remplacer des postes clients par des postes de capacité moindre
- bande passante de communication
- capacité de mémoire
- capacité d'affichage (résolution, couleur)
- puissance de traitement
- consommation

Problème : comment continuer à exploiter l'application ? (en particulier si les serveurs ne peuvent être modifiés)

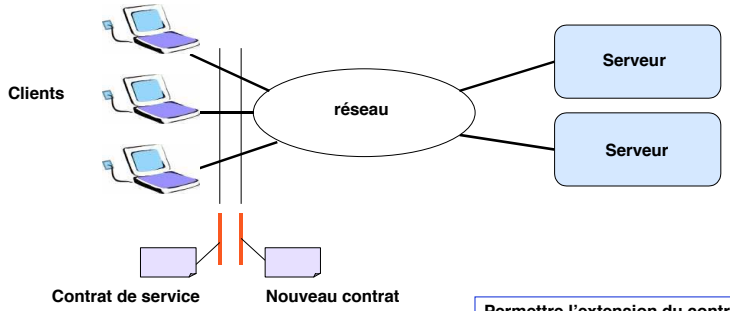
Exemple 2
Adaptation de services du fait de la mobilité



Permettre la mobilité géographique des postes clients (changement de localisation physique, avec modification possible de l'environnement et de la qualité de service de la communication.

Problème : comment continuer à exploiter l'application ?

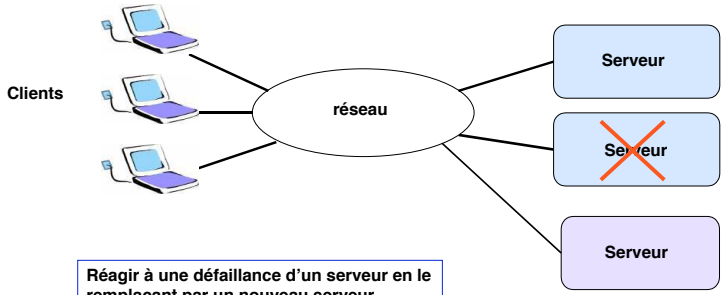
Exemple 3
Extension de services, modification de leur mise en œuvre



Permettre l'extension du contrat de service (exemple : fonctions étendues, ajout de propriétés non fonctionnelles)

Problème : comment réaliser la transition (en particulier sans arrêter l'application) ?

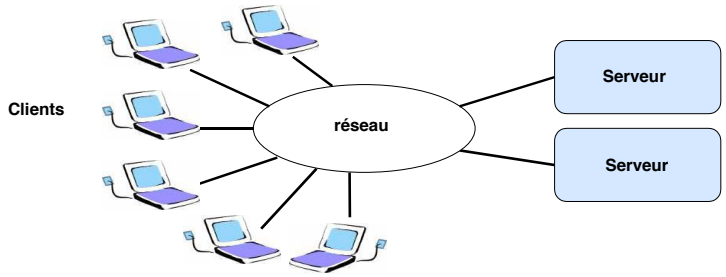
Exemple 4
Adaptation de services pour la tolérance aux fautes



Réagir à une défaillance d'un serveur en le remplaçant par un nouveau serveur équivalent

Problème : comment réaliser ce remplacement (détecter la défaillance, effectuer la transition), sans arrêter l'application ?

Exemple 5
Adaptation de services en fonction de la charge



Réagir à une augmentation de la charge (par ex. nombre de clients connectés)

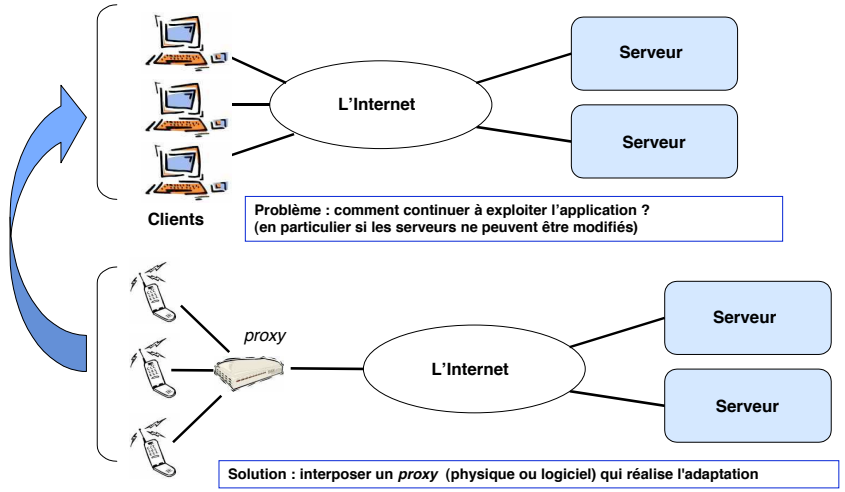
Problème : comment maintenir un niveau acceptable de qualité de service (par ex. temps de réponse moyen) ?

Adaptation des systèmes et applications

- Qu'est ce que l'adaptation ?
  - Changement de la structure et/ou des fonctions d'une application
  - Adaptation dynamique : réalisée sans arrêt de l'application
- Pourquoi l'adaptation ?
  - Pour répondre à l'évolution
    - Des besoins : nouvelles fonctions, nouvelles qualités
    - De l'environnement d'exécution (capacités du matériel, mobilité, conditions de communications, perturbations et défaillances, etc.)
- Comment ?
  - Principe : système réflexif (fournit une représentation de son propre fonctionnement, pour l'inspecter ou le modifier)
- Techniques
  - Techniques ad hoc (intercepteurs)
  - Protocoles à méta-objets (MOP)
  - Programmation par aspects (AOP)

Ces 3 techniques sont examinées successivement

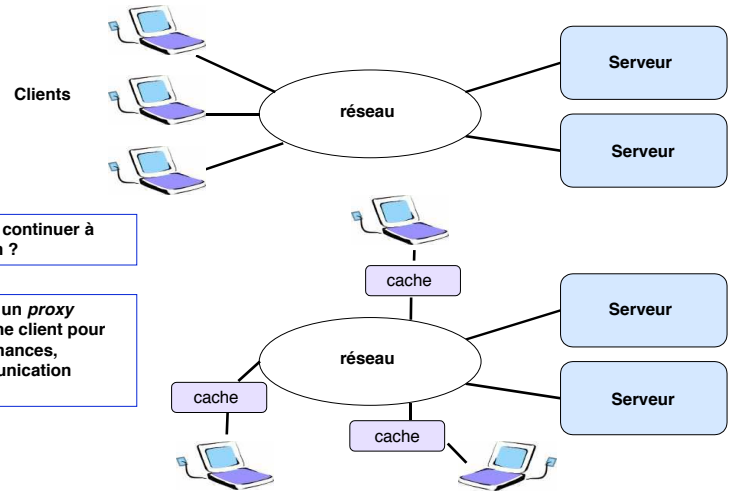
Exemple 1
Adaptation de services en fonction des capacités du poste client



Problème : comment continuer à exploiter l'application ?
(en particulier si les serveurs ne peuvent être modifiés)

Solution : interposer un *proxy* (physique ou logiciel) qui réalise l'adaptation

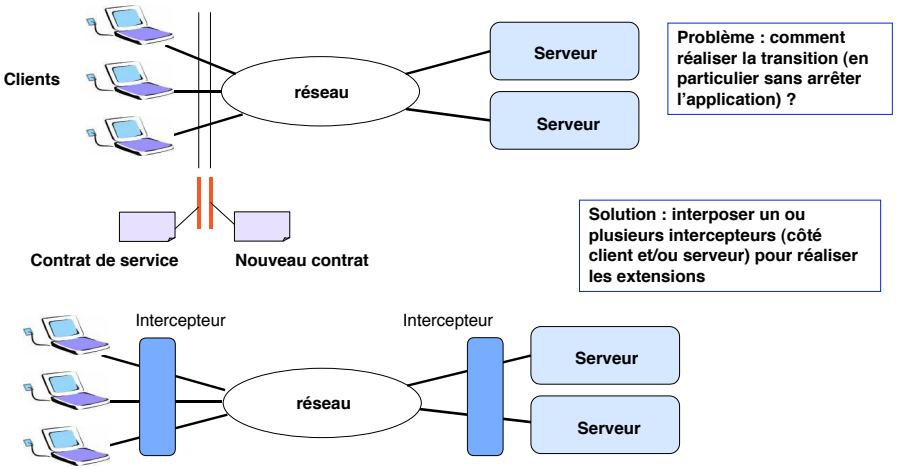
Exemple 2
Adaptation de services du fait de la mobilité



Problème : comment continuer à exploiter l'application ?

Solution : interposer un *proxy* jouant le rôle de cache client pour améliorer les performances, notamment si communication défectueuse

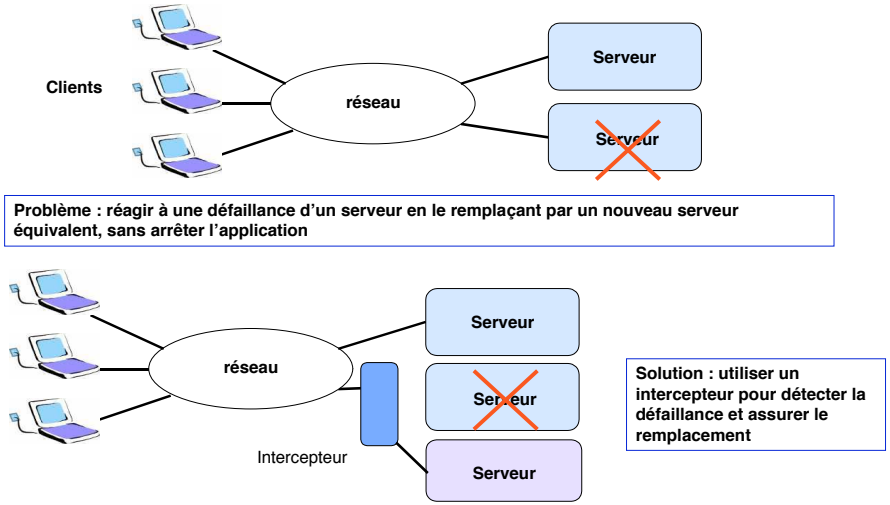
Exemple 3
Extension de services, modification de leur mise en œuvre



Problème : comment réaliser la transition (en particulier sans arrêter l'application) ?

Solution : interposer un ou plusieurs intercepteurs (côté client et/ou serveur) pour réaliser les extensions

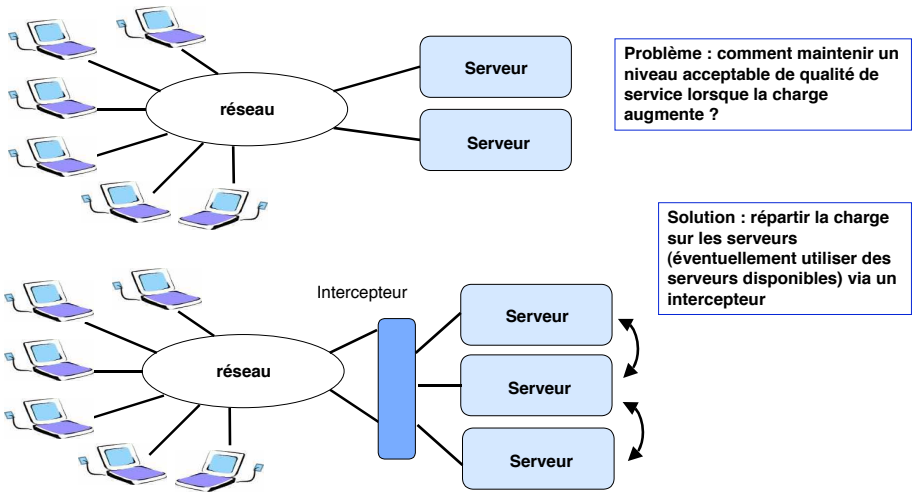
Exemple 4
Adaptation de services pour la tolérance aux fautes



Problème : réagir à une défaillance d'un serveur en le remplaçant par un nouveau serveur équivalent, sans arrêter l'application

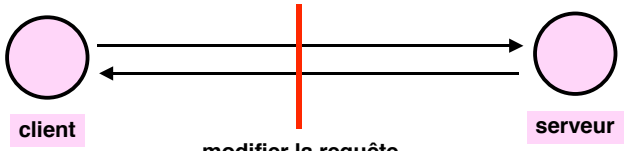
Solution : utiliser un intercepteur pour détecter la défaillance et assurer le remplacement

Exemple 5
Adaptation de services en fonction de la charge



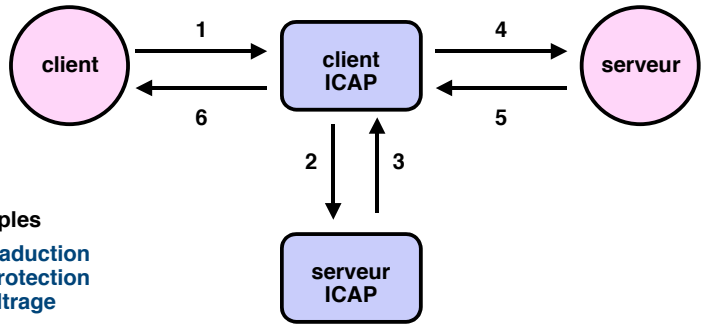
Exemple 6 : protocole ICAP (*Internet Content Adaptation Protocol*)

- **Fonction : fournir des services à valeur ajoutée**
  - utilisation de ressources (processeurs, appareils spécialisés)
  - réalisation de fonctions (filtrage, transformation, traduction, protection)
- **Mode d'action**
  - interposition dans un système client-serveur utilisant HTTP



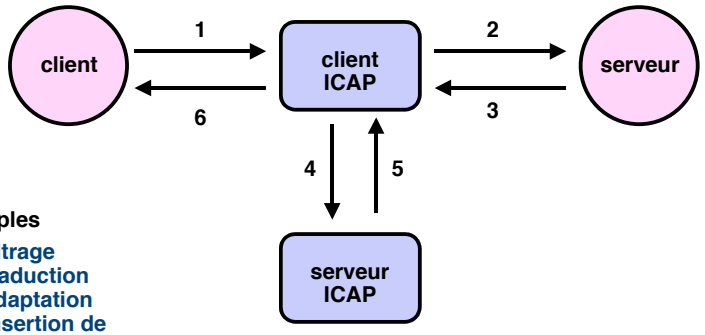
modifier la requête
modifier la réponse
ajouter une fonction

Protocole ICAP : modifier la requête



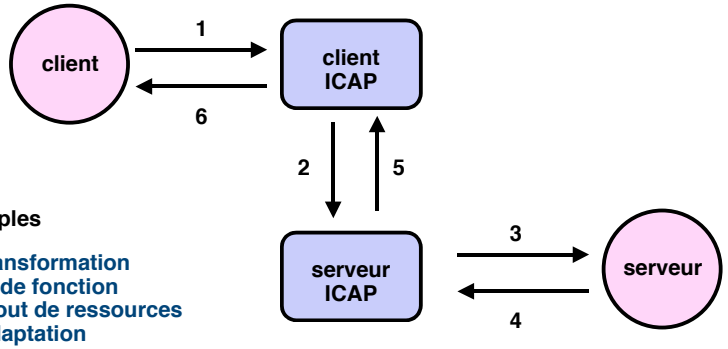
Exemples
- traduction
- protection
- filtrage

Protocole ICAP : modifier la réponse



Exemples
- filtrage
- traduction
- adaptation
- insertion de commentaires (ou de publicité)

Protocole ICAP : interposer une fonction



Exemples

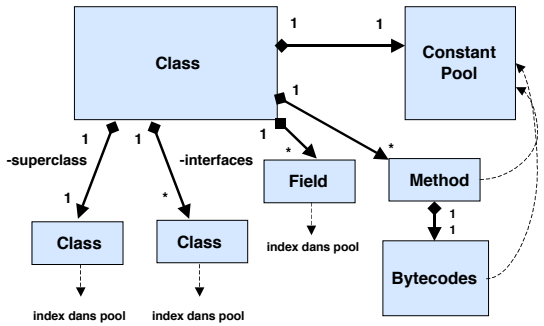
transformation
de fonction
ajout de ressources
adaptation

Une autre technique d'adaptation : la réflexivité

- **Définition**
  - Un système est réflexif s'il comporte une représentation de son propre état, lui permettant d'inspecter et de modifier cet état
- **Revue rapide de la réflexivité**
  - Un exemple simple : la réflexivité en Java
  - Une méthode générale pour réaliser la réflexivité : protocoles à méta-objets
    - Exemples simples d'application

Exemple de réflexivité : API Java Reflection

Les méta-données d'une classe Java contiennent une description des différents constituants de cette classe.



Contient toutes les constantes liées àl la classe et en particulier les chaînes de caractères qui sont les noms de méthodes, de classes, etc.

Les autres éléments contiennent des index vers des entrées de ce pool

L'API Java Reflection fournit une interface Java pour l'accès à ces méta-données et pour leur manipulation

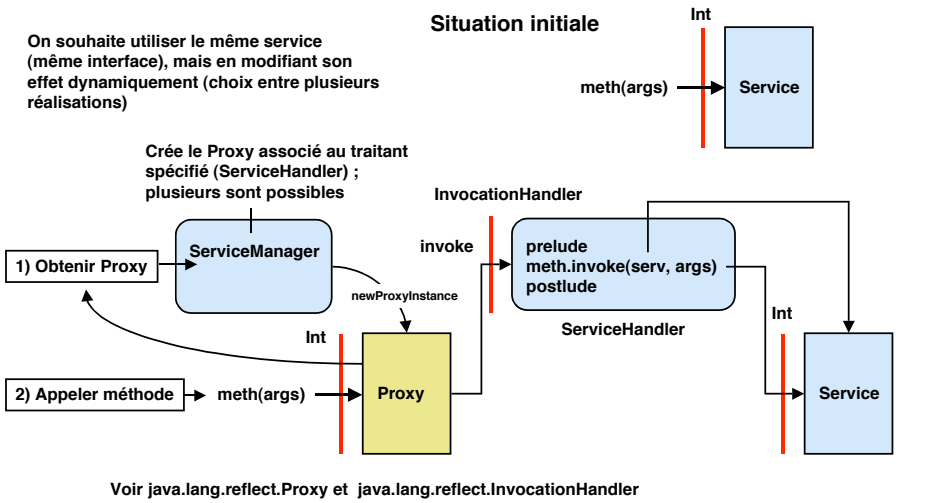
Utilisation de la réflexivité en Java

Permet de réaliser des opérations génériques (indépendantes d'une classe particulière)
Très utile si la classe n'est connue qu'au moment de l'exécution.

Exemples

```
…
String className = "le nom d'une classe" ;
Class c = Class.forName(className) ;        // la classe n'est connue qu'à l'exécution

…
Object obj = c.newInstance()                // constructeur standard pour c
…
methodName = "le nom d'une méthode";
Method m = c.getMethod(methodName, new Class[ ] {…}) ;
…
m.invoke(c, new Object  [ ] {…}) ;          // appel dynamique de méthode  (générique)
…
```

Application de la réflexivité en Java : *Dynamic Proxy*



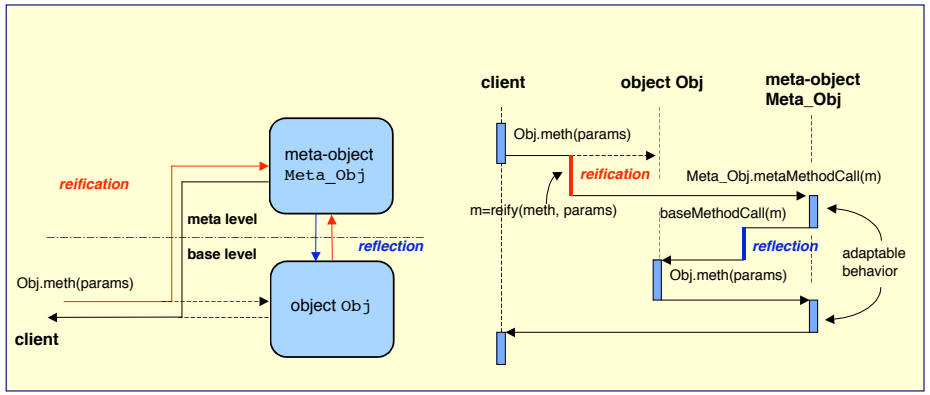
Voir java.lang.reflect.Proxy et java.lang.reflect.InvocationHandler

Protocoles à méta-objets

- Un service adaptable est organisé en deux niveaux
  - Niveau de base : réalise les fonctions définies par la spécification
  - Méta-niveau : utilise une représentation du niveau de base pour observer ou modifier le comportement de celui-ci
    - La représentation doit être causalement connectée (i.e. toujours fidèle : tout changement du niveau de base est reflété dans la représentation et vice-versa)
  - Relations entre les niveaux
    - Création d'une représentation d'une entité : réification
    - Action du méta-niveau sur le niveau de base : réflexion
- L'organisation peut être étendue récursivement
  - "Tour réflexive" : méta-méta-niveau, etc.
  - En pratique, on se limite à 2 ou 3

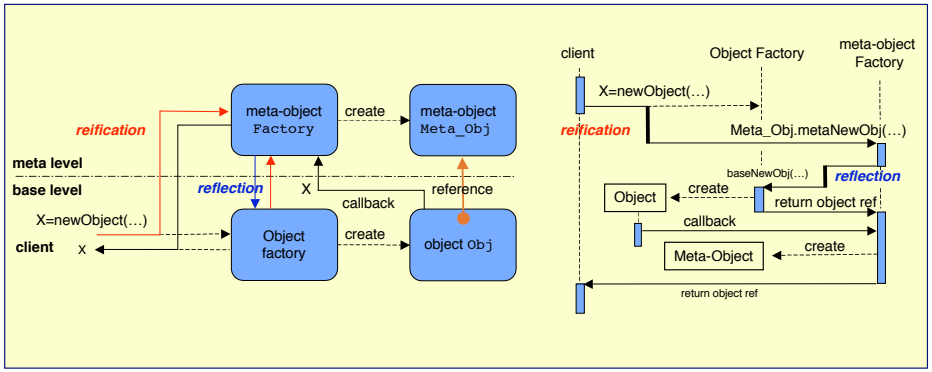
Protocoles à méta-objets : exemple 2

- Réification d'un appel de méthode



Protocoles à méta-objets : exemple 2

- Création d'un objet



Programmation par aspects

- **Principe** (application de la séparation des préoccupations)
  - Identifier un comportement de base et des "aspects" supplémentaires aussi indépendants que possible
  - Décrire séparément le comportement de base et les différents aspects
  - Intégrer l'ensemble dans un programme unique
- **Méthode**
  - Description séparée des aspects (éventuellement format spécifique)
  - Intégration ("tissage"), statique ou dynamique
- **Notions de base**
  - Point de jointure (*join point*) : point d'insertion de code d'aspects
  - Coupure (*cut point*) : ensemble de points de jointure logiquement corrélés
  - Indications (*advice*) : définition des relations entre le code inséré et le code de base (exemple : avant, après, …)
- **Problème : réalisation efficace**

Programmation par aspects : utilisation

- **Réalisation d'un *Wrapper* en AspectJ**

```
public aspect MethodWrapping  {

/* point cut definition */
    pointcut Wrappable(): call(public * MyClass.*(..));

 /* advice definition */
    around(): Wrappable() {
            <prelude>     /* a sequence of code to be inserted before the call */
            proceed();    /* performs the call to the original method  */
            <postlude>   /* a sequence of code to be inserted after the call */
    }
}
```

Effet : encadre tout appel à une méthode publique de la classe MyClass par <prelude> et <postlude> (application possible :journalisation, test de conditions, etc.)

Exemple de programmation par aspects : journal (*log*)

- **Définition**
  - Journal (*log*) = séquence d'enregistrements dont chacun contient des informations relatives à un événement spécifié dans un système. La séquence est ordonnée par le temps
    - Les informations peuvent être élémentaires (au minimum un booléen : l'événement est arrivé) ou complexes (état du système)
  - Utilisation : mise au point, statistiques, audit, etc.
- **Solutions**
  - Implantation directe dans le code de l'application
    - Complexe, peu adaptable, mélange code de l'application et code de l'outil de journalisation
  - Définition séparée de la journalisation
    - Bibliothèque spécialisée - exemple en Java : log4j (Apache)
      - … mais les appels doivent être introduits "à la main"
    - Programmation par aspects
      - La journalisation est un aspect particulier

Journalisation "à la main"

```
public class A {
    …
    public void meth1 (…) {
    … }
    public void meth2 (…) {
    … }
    public static void main (…) {
    … }
}
```

```
public class B {
    …
    public int meth3 (…) {
    … }
    public String meth4 (…) {
    … }
}
```

Insertion à la main des appels aux méthodes de journalisation dans le code des classes à observer

Fastidieux et peu sûr (risque d'erreur
Peu pratique pour de gros volumes de code

```
import java.util.logging
public class A {
    static Logger _logger = Logger.getlogger("trace");
    …
    public void meth1 (…) {
   _logger.logp(Level.INFO, "A", "meth1", "entrée");
    … }
    public void meth2 (…) {
    _logger.logp(Level.INFO, "A", "meth2", "entrée");
    … }
    public static void main (…) {
    _logger.logp(Level.INFO, "A", "main", "entrée");
    … }
}
```

```
import java.util.logging
public class B {
    static Logger _logger = Logger.getlogger("trace");
    …
    public int meth3 (…) {
    _logger.logp(Level.INFO, "B", "meth3", "entrée");
    … }
    public String meth4 (…) {
    _logger.logp(Level.INFO, "B", "meth4", "entrée")
    … }
… }
}
```

Source : R. Laddad, *AspectJ in Action*
Manning, 2003

Résultats

Effet de l'exécution du programme journalisé

nom de la classe

```
Jan 28, 2004  12:30:45 AM   A   main
INFO:  entrée
Jan 28, 2004  12:30:45 AM   A   meth1
INFO:  entrée
Jan 28, 2004  12:30:45 AM   B   meth4
INFO:  entrée
Jan 28, 2004  12:30:45 AM   A   meth2
INFO:  entrée
Jan 28, 2004  12:30:45 AM   B   meth3
INFO:  entrée
Jan 28, 2004  12:30:46 AM   A   meth1
INFO:  entrée
Jan 28, 2004  12:30:46 AM   A   meth2
INFO:  entrée
…
```

message. spécifié

nom de la méthode

Journalisation avec AspectJ

```
import java.util.logging.*;
import org.aspectj.lang.*;

public aspect TraceAspect {
   private Logger _logger = Logger.getLogger("trace");

   pointcut traceMethods()
         : execution(* *.*(..)) && !within(TraceAspect);

   before() : traceMethods() {
         Signature sig = thisJoinPointStaticPart.getSignature();
         _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
                  sig.getName(), "entrée");
   }
```

En compilant ce programme avec les deux classes "tracées", on obtient le même résultat qu'avec l'insertion à la main

spécifie un appel de méthode quelconque

exclut les méthodes de traçage (sinon récursion infinie)

avant l'exécution du code au point de coupure

créé à la compilation

extrait la signature du point de coupure

nom de la méthode appelée

nom de la classe de la méthode appelée

Utilisation de l'introspection

Source : R. Laddad, *AspectJ in Action*, Manning, 2003

Commentaires sur l'exemple

- **Avantages**
  - Économie : plus besoin d'insérer à la main tous les appels au code de journalisation
  - Séparation de l'expression de la journalisation
  - Sûreté : réduit le risque d'erreurs mécaniques (recopie)
  - Facilité d'évolution : modification en un seul point (l'aspect)
    - Changer l'outil de journalisation (exemple : passer à log4j)
    - Redéfinir les points d'enregistrement (coupure)
    - Modifier les paramètres enregistrés
    - Changer la présentation des résultats (exemple : indentation)

Adaptation et composants

- **Composants logiciels**
  - Unité de décomposition d'un système ou d'une application
    - Pour les fonctions (interfaces)
    - Pour le déploiement (installation physique)
- **Lien entre adaptation et composants**
  - Un composant peut être aussi une unité d'adaptation
    - Le mécanisme qui gère les composants (conteneur ou autre) peut aussi servir à l'adaptation
  - L'interception s'applique bien aux interactions entre composants
    - Une interface est un point de passage obligé et bien défini
  - La configuration physique est un autre outil d'adaptation
    - Redéploiement
    - Reconfiguration dynamique

Administration d'applications : plan de présentation

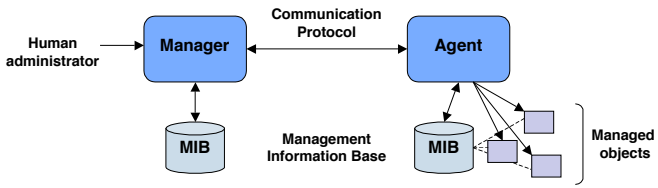
- Introduction à l'administration
  - Objectifs, moyens, notions de base, terminologie
  - Administration d'applications
- Configuration et déploiement
  - Principes de base
  - Exemples
- Reconfiguration
  - Principes de base
  - Exemples
- Introduction à l'administration autonome

Introduction à l'administration

- Rappels
  - Un système est un assemblage de parties (matériel, logiciel, …) qui fournit des services
  - Un système n'est jamais isolé, mais interagit avec un environnement (milieu physique, personnes, autres systèmes, …)
  - L'environnement fournit des ressources et des services (par ex. énergie, …) qui permettent au système de fournir ses propres services
  - Ces définitions s'appliquent au système et à chacune de ses parties (le reste du système fait alors partie de l'environnement)
  - Le système interagit avec son environnement
    - Interactions prévues et souhaitées (fourniture de services)
    - Interactions imprévues ou indésirables (charge inattendue, attaque, défaillance)
- Définition et propriétés
  - L'administration est l'ensemble des fonctions qui permettent à un système de maintenir sa capacité à fournir ses services, dans un environnement donné
    - Prévention : éliminer les interactions indésirables, ou réduire la probabilité de leur occurrence
    - Détection et traitement : détecter une interaction indésirable et éliminer ou réduire son effet (boucle de rétroaction)
  - La distinction entre "système d'administration" et système administré" n'est pas toujours évidente (le systèma administré peut avoir des fonctions d'administration)

Situation de l'administration

- Un modèle de base…
  - Le modèle *Manager-Agent*
    - Utilisé dans les standards : SNMP (réseaux), CMIS/CMIP (télécom)



- …qui atteint ses limites
  - Architecture centralisée
    - Passe mal à l'échelle, ne résiste pas à la charge et aux défaillances
  - Structure statique
    - Manque de réactivité, alors que les systèmes actuels sont très évolutifs
  - Mal adapté à la complexité
    - Tendance vers une administration plus autonome (self-*)

Configuration et déploiement

- Définitions
  - Configuration : choix, génération, paramétrages des constituants d'une application ; [liaison des constituants]
  - Déploiement : [liaison des constituants] ; placement des constituants sur les sites ; lancement de l'application
- Principes de mise en œuvre
  - Ces opérations sont exécutées par des *scripts* (commandes interprétées par un système d'exploitation ou par un utilitaire spécialisé)
  - Application du principe de séparation des préoccupations
    - Description de la configuration (ADL par ex.) et du déploiement (carte d'implantation) : description déclarative et séparée du code
    - Génération automatique des *scripts* de configuration et déploiement à partir de la description déclarative
    - [Prospective] la description déclarative elle-même pourrait être engendrée automatiquement à partir de meta-règles (politique)
  - Configuration et déploiement sont orientés par l'architecture
  - Configuration et déploiement prennent une place croissante car les systèmes sont de plus en plus grands et complexes

Un exemple de canevas de gestion de configuration

- **Contexte**
  - Initialement proposé dans le système Jonathan (un canevas pour la construction d'ORB).
    - Voir http://jonathan.objectweb.org/doc/tutorial/
      - chapitre Configuration Framework
  - Convergence en cours avec ADL de Fractal
- **Objectif**
  - Automatiser la production de configurations construites à partir de "composants" Java
    - Avec ou sans modèle de composants associé

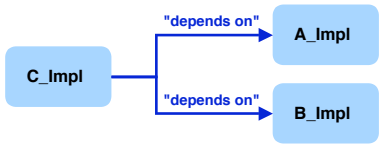
The Static Configuration Problem

```
public class C_Impl implements C {
        A field_A ;
        B field_B ;
        …

        field_A = new A_Impl() ;
        field_B = new B_Impl() ;
}
```

```
public interface  A {
        …
}
public interface B {
        …
}
public interface C {
        …
}
```

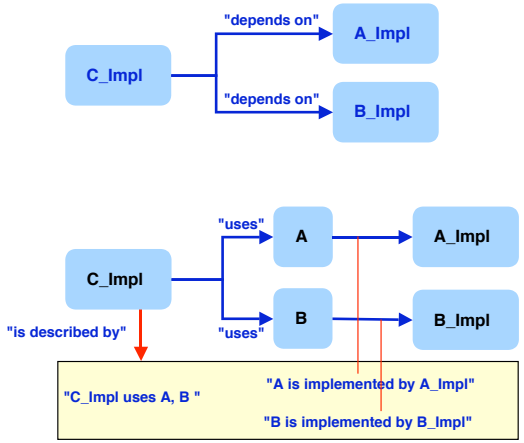
```
public class A_Impl implements A {
        …
}
public class B_Impl implements B {
        …
}
```

What if we change the implementation of A to A_Impl1 ?
We have to modify the code of class C_Impl



We need a description of the dependency between C_Impl, A_Impl and B_Impl, separate from the code

Describing a Configuration



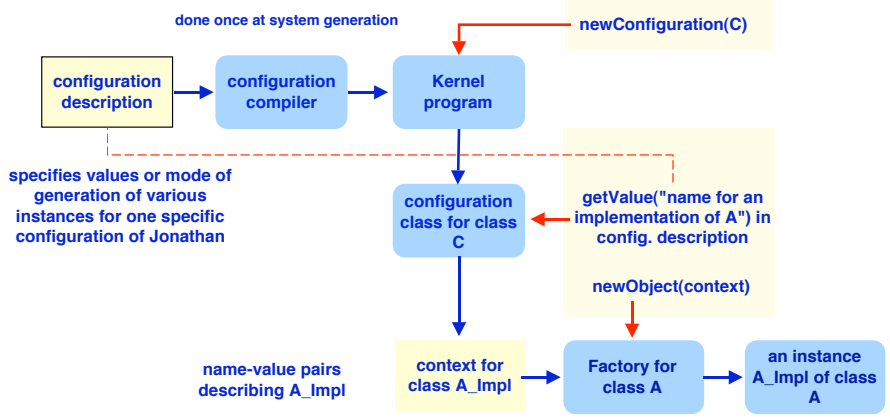

```
public class C_Impl implements C
{
        A field_A ;
        B field_B ;
        …

        field_A = new A_Impl() ;
        field_B = new B_Impl() ;
}
```

Changing the implementation of A or B means changing the configuration description, not the code of C_Impl

Configuration in Jonathan

C_Impl implements C, uses A implemented by A_Impl

Initialization sequence in the program of C_Impl



Configuration Description in Jonathan

```
[/]
 [jonathan]
   ...
  [IPv4ConnectionFactory]        -- a configuration
       factory =>
          org.objectweb.jonathan.libs.resources.tcpip.
              IPv4ConnectionFactoryFactory     --  an atom
              instance => {factory, .}         -- an assemblage
              verbose -> /jonathan/tcpip/verbose -- an alias
          ...
  [JDomain]
     [binders]
        0 -> /david/orbs/iiop/instance
        org.objectweb.david.libs.binding.orbs.iiop.IIOPORB
           =>(int.class, 0)                  -- a property
     ...
  [tcpip]
      verbose => (Boolean.class, false)   -- a property
     ...
 [jeremie]
     ...
   [jiop]
      factory =>
         org.objectweb.jeremie.libs.binding.jiop.JIOPFactory
      instance => {factory, .}
     Chunkfactory -> /jonathan/JChunkFactory/instance
     ...
```

The actual description is in XML

A configuration [conf] is described by four types of elements

Atom: description of a class
name => class

Property: associates a type and a value
name => (type, value)

Assemblage: associate a factory and a configuration
name => {fact, conf}

Alias: an alternative name for an existing element
name -> target

getValue() uses these descriptions to generate actual values for attributes

Contexts in Jonathan

A context is a set of elements (typed objects), each identified by a name.
Example: a tree context (structured as a tree)

The getValue() method builds a context, starting from a configuration description (another context). Its effect depends on the type of the element in the description:

A property element (e.g. (Integer.class, 123))
return its value

An atom element (e.g. class name)
load the class, then create an instance of it

An assemblage element (e.g. (factory, context))
find factory and context, then create an instance using factory.newObject(context)

An alias element (e.g. name -> target)
apply getValue() to the target element of the alias

An Example

Class LocateRegistry uses a specific instance of class JIOP

In the static initializer of LocateRegistry, we find the following code sequence:

```
$initial_context =
      (Context) Kernel.newConfiguration(LocateRegistry.class).getValue("/jeremie/jiop", '/');
binder = (JIOP) (new JIOPFactory()).newObject($initial_context);
```

Creating a specific instance is a two-step process:

Create a context describing the instance
- create a configuration using the configuration description
- get the parameters corresponding to the created class

Use a factory to create the instance
- create an instance of the class-specific factory
- create an instance of the class using the new context

Using ConfigurableFactories

```
public abstract class GenericFactory implements Factory {
        public Object newObject (Context _c) {…
            $comps = getUsedComponents (_c) …
            return newInstance (_c, $comps) }
        abstract protected Object[] getUsedComponents (Context _c);
        abstract protected Object newInstance (Context _c, Object[ ]used_components);
}
public class JIOPFactory extends GenericFactory {     // an example of specific factory
        final protected Object[] getUsedComponents(Context _c) {
            used_components[0] = _c.getValue ("ChunkFactory", (char) 0);
            used_components[1] = …}
        final protected Object newInstance (Context _c, Object[ ] used_components) {
            return new JIOP(_c, used_components); }
}
public class JIOP … {
        JIOP(Context c, Object[] used_components) … {
            super();
             initialize(c, used_components); }
        protected void initialize (Context c, Object[ ] used_components) {
            ChunkFactory chunk_factory= (ChunkFactory) used_components[0];
            … }
}
```

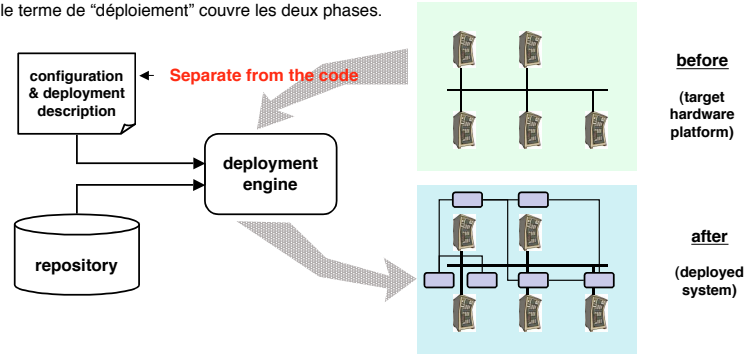
Un modèle simple pour le déploiement

Rappel de la définition

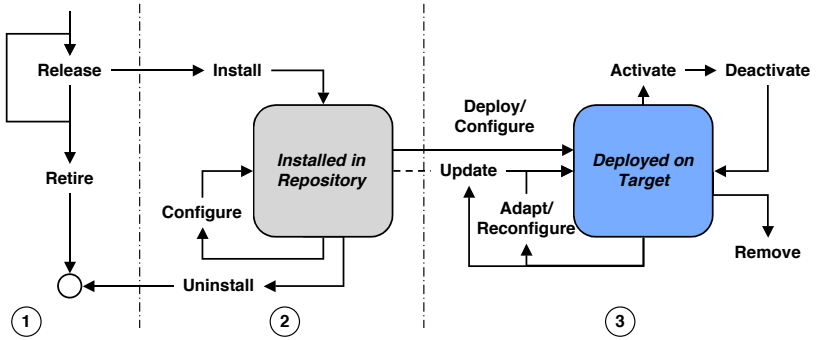
Configuration : choix, génération, paramétrages des constituants d'une application ; [liaison des constituants]

Déploiement : [liaison des constituants] ; placement des constituants sur les sites ; lancement de l'application

Parfois le terme de "déploiement" couvre les deux phases.



Cycle de vie du déploiement



Problème du déploiement : préserver la variabilité (capacité de faire des modifications en tout point du cycle de vie, et de maintenir plusieurs versions de chaque partie du système)

- **Garder trace des multiples configurations possibles**
- **Préserver la cohérence en cas de changement : nécessite d'expliciter les dépendances, ce qui n'est pas toujours fait.**

Éléments du déploiement (orienté par l'architecture) (1)

Référentiel (*Repository*)

Contient les éléments à déployer (unités de déploiement). Propriétés requises :

- identification unique
- définition explicite des dépendances (listes export-import)
- composabilité (de préférence hiérarchique)

Exemples : fichiers .jar, .ear (J2EE), *bundles* OSGI, composants Fractal, fichiers RPM

Langage de description de déploiement

Définit

- le produit à déployer (en termes de ses composants)
- les contraintes d'administration (par ex. tel composant doit aller sur telle machine physique)
- les outils à utiliser pour le déploiement

Pour le moment, pas de standard car problème encore mal compris (langages ad hoc). Utilisation possible d'une extension de l'ADL (cf Fractal, travail en cours)

Éléments du déploiement (orienté par l'architecture) (2)

Machine de déploiement

Vue actuelle : un moteur de *workflow* (flot de tâches) : ensemble de tâches à réaliser T_i + liste de contraintes de précédence, (ex : T_i après T_j et T_k)
Les tâches sont des actions élémentaires de déploiement, par exemple :

- Configurer les composants de l'application
- Charger les composants sur les sites de la plate-forme cible, et terminer (si nécessaire) la configuration
- Exécuter les liaisons nécessaires (entre composants, entre un composdant et son environnement local sur son site)
- Activer les composants

Déploiement : état de l'art

2 références conseillées :

Pour l'organisation du référentiel et l'analyse des dépendances : Nix

[Dolstra 2005]Dolstra, E. (2005). Efficient upgrading in a purely functional component deployment model. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005)*, volume 3489 of *Lecture Notes in Computer Science*, pages 219-234, St. Louis, Missouri, USA. Springer-Verlag.

[Dolstra et al. 2004a]Dolstra, E., de Jonge, M., and Visser, E. (2004a). Nix: A safe and policy-free system for software deployment. In Damon, L., editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79-92, Atlanta, Georgia, USA. USENIX.

Pour les aspects d'ingénierie : SmartFrog

[SmartFrog 2003]SmartFrog (2003). SmartFrog: Smart Framework for Object Groups. HP Labs. http://www.hpl.hp.com/research/smartfrog/.

[Goldsack et al. 2003]Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., and Toft, P. (2003). SmartFrog: Configuration and Automatic Ignition of Distributed Applications. HP OVUA 2003 - HP OpenView University Association.

Reconfiguration (1)

- **Définition**
  - Modification d'une configuration de système (ou application) existante (déployée)
  - Reconfiguration dynamique : sans arrêt du système
  - Hypothèse : système (ou application) = ensemble de composants
- **Modalités**
  - Modification d'attributs d'un composant sur un site
  - Modification des liaisons entre composants existants
  - Déplacement d'un composant d'un site à un autre, sans changer ses liaisons
  - Ajout, suppression ou remplacement d'un composant sur un site (ce qui entraîne une modification des liaisons)
- **Exemples**
  - Remplacer d'un composant (ajout de fonctions, correction d'erreurs)
  - Dupliquer un composant sur plusieurs sites pour répondre à un pic de la demande de services
  - Déplacer un serveur d'un site à un autre (défaillance, maintenance)
  - Reconfigurer un réseau en chargeant de nouveaux programmes dans les routeurs
  - Reconfigurer un réseau (physique ou virtuel) en changeant sa topologie

Reconfiguration (2)

- **Propriétés souhaitables de la reconfiguration**
  - Les modifications doivent être exprimées en termes de la structure du système (i.e. au niveau des composants et de leur interconnexion)
    - Exclut des spécifications de grain trop fin
  - La spécification des modifications doit être déclarative
    - Séparation entre la spécification et sa mise en œuvre
  - La spécification des modifications doit être indépendante des algorithmes et protocoles propres à l'application
    - Permet de développer des méthodes et outils génériques
  - La cohérence de l'état doit être maintenue
    - Il faut donc identifier des invariants caractéristiques du système
  - La perturbation pendant le régime transitoire doit être réduite ou éliminée
    - Il faut donc identifier le sous-ensemble minimal du système mis en cause par l'opération de reconfiguration

Source : J. Kramer, J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. On Software Engineering*, 16 (11), Nov. 1990

Reconfiguration (3)

- **Réalisation de la reconfiguration dynamique**
  - "Mécanique" de la reconfiguration (outils disponibles dans un environnement donné)
    - Aspect encore encore peu développé
    - Exemple dans Fractal : interface pour la création et la liaison dynamique de composants inclus
    - L'interposition est le mécanisme le plus répandu actuellement (mais les fonctions réalisables sont limitées)
      - Exemples en Java: utilisation de *Dynamic Proxy* ou d'AspectJ
- **Exemples récents représentatifs de l'état de l'art**
  - Changement dynamique de topologie d'un système *publish-subscribe* (suppression et replacement d'une liaison)
    - G. Cugola, D. Frey, A.L. Murphy, G. P. Picco. Minimizing the Reconfiguration Ovrhead in Content-Based Publish-Subscribe, *SAC'04*, March 2004
  - Reconfiguration dynamique utilisant l'interposition ou le remplacement d'un composant : mécanismes de base
    - C.N. Soules et al. System Support for Online Reconfiguration, *Proc. Usenix Annual Technical Conference*, San Antonio, TX, June 2003

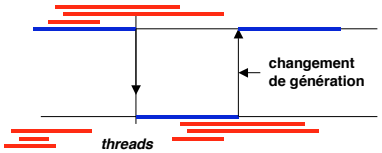
Reconfiguration : techniques de mise en œuvre (1)

- **État quiescent**
  - Un composant peut être dans 2 états d'exécution
    - Actif : il existe des *threads* en cours d'exécution dans le composant
    - Passif : pas de *threads* en cours d'exécution dans le composant
  - La condition "état passif" ne suffit pas pour permettre le changement du composant
    - En effet il peut redevenir actif par suite d'un nouvel appel
  - Une propriété plus forte est nécessaire : état quiescent
    - Quiescent : passif + garantie qu'il ne sera pas réactivé

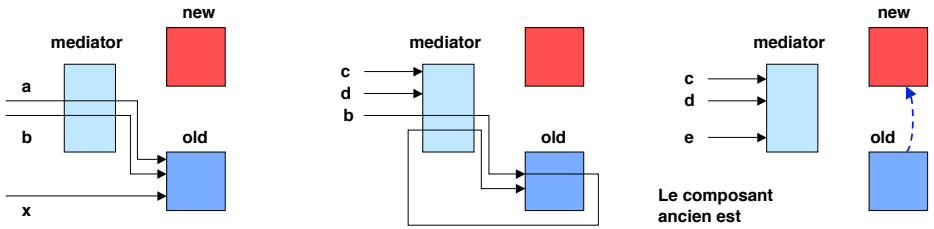
Reconfiguration : techniques de mise en œuvre (2)

- **Comment garantir la quiescence d'un composant ?**
  - Notion de multiples (époques)
  - Hypothèse (à garantir par l'application) : toute exécution de *thread* se termine en un temps fini
  - Soit un changement de structure réalisé au temps t ; on définit 2 générations de *threads*
    - Ceux lancés avant t : ils accèdent à l'ancienne structure
    - Ceux lancés après t : on les fait accèder à la nouvelle structure
    - Donc la quiescence de l'ancienne structure est garantie au bout d'un temps fini

```
i: = 0                                          [Soules et al, 2003]
while (i <2)
    if (non-current generation's thread count = 0
        make it the current generation
    else
        wait until it is zero and
        make it the current generation
        i := i+1
```



Remplacement d'un composant (K42)



Attendre fin *threads* de la génération courante et changer de génération

Les nouveaux *threads* sont bloqués par le médiateur ; on laisse se terminer ceux en cours

Le composant ancien est quiescent ; on peut faire la substitution ; les *threads* bloqués seront relancés sur la nouvelle version.

Source : C.N. Soules et al. System Support for Online Reconfiguration, *Proc. Usenix Annual Technical Conference*, San Antonio, TX, June 2003

Administration autonome *(Autonomic Computing)*

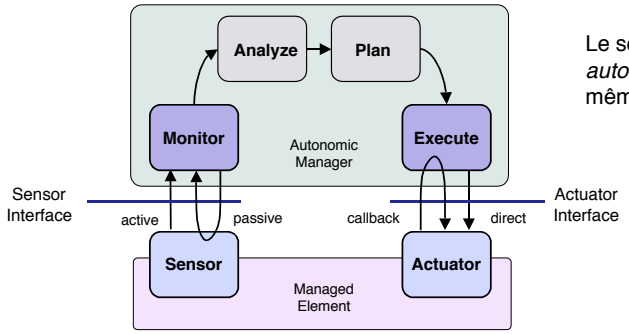
- **Motivations**
  - L'administration des grands systèmes devient trop complexe pour rester confiée à des humains
    - Il est difficile de corréler les actions d'administration (de bas niveau) avec les objectifs stratégiques (à haut niveau)
    - L'adaptation demande souvent une réponse très rapide
- **Principes**
  - Commande par boucle de rétroaction : cycle observation, analyse et décision, action
- **État**
  - Domaine de recherche très actif, notamment dans l'industrie (IBM, HP, etc.)
  - Domaine multidisciplinaire (Systèmes, Commande, IA, …)

Kephart, J. O. Research challenges of autonomic computing, *ICSE'05: Proc. 27th International Conf. on Software Engineering*, pp. 15-22, ACM Press, 2005

Domaines de l'administration autonome

- **Auto-optimisation (seul aspect traité ici)**
  - **Objectif : maintenir un niveau spécifié de perfomances malgré des variations de la charge**
- **Auto-configuration**
  - **Objectif : réaliser la configuration automatique ; servir d'outils pour les autres aspects**
- **Auto-réparation**
  - **Objectif : maintenir un niveau spécifié de disponibilité malgré des défaillances logicielles ou matérielles**
- **Auto-protection**
  - **Objectif : maintenir un niveau spécifié de sécurité (confidentialité, intégrité, contrôle d'accès, libre disposition des services) malgré des actions malveillantes**

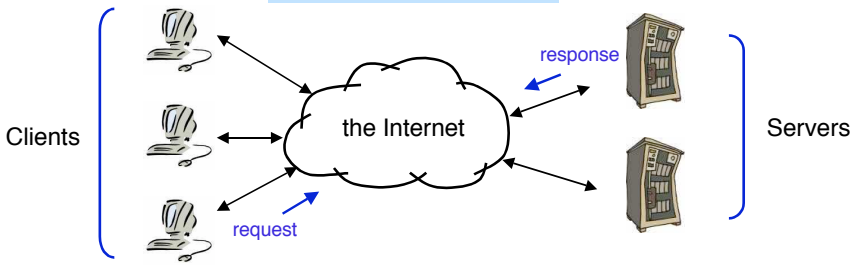
Modèle général de l'administration aotonome



Le schéma est récursif : un *autonomic manager* est lui-même un *managed element*

Internet Services

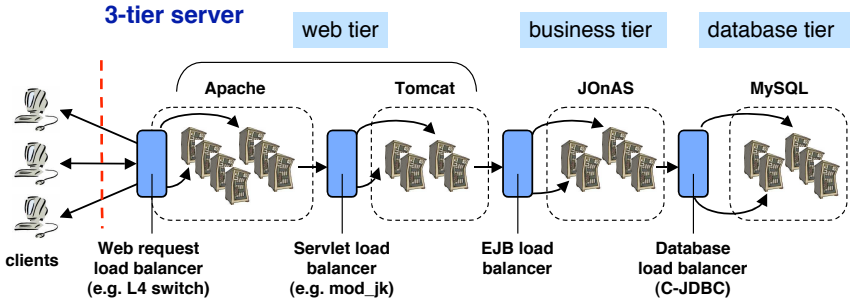
The client-server scheme



Application domains
- Electronic commerce, auction sites
- Content distribution
- Administration (taxes, etc.)
- …

Server standards
- J2EE, .NET, web services
- open source and proprietary cluster platforms

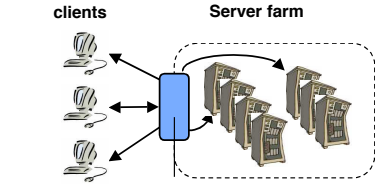
N-tier application platform



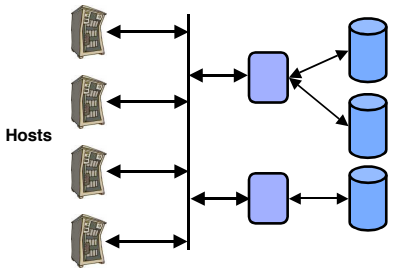
Hardware platforms
- clusters

Software components
- Web server
- Applet server
- Component)based framework
- Database server

Other examples of services (1)

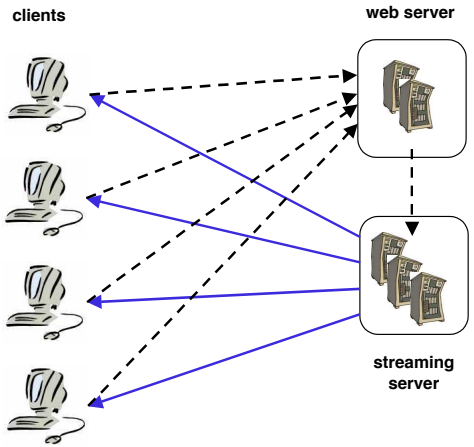


Using a cluster (or a grid) for CPU intensive computing



Managing a large scale storage utility for a set of customers

Other examples of services (2)



Delivering multimedia content to a large set of clients

A (possible) catalog of objectives

Service Level Agreement (SLA):
a contract between provider and clients
a set of Service Level Objectives (SLO)

Service availability
e.g., up 99.9% of the time

Guaranteed response time
e.g., < 5 seconds 95% of the time
for "typical" request

Differentiated service, e.g.
Gold: < 3 seconds
Silver: < 4 seconds
Others: best effort

Stability (for streaming media)
Limits on jitter, etc.
Guaranteed delivery rate

most of our current work

Adverse conditions

Unexpected demand peak
wide variations are possible

Failures
Hardware failures
(e.g. failure of one or several node(s))
Software failures
(e.g. failure of a component or subsystem)
Administration failures
(e.g. misconfiguration)

Performability
Specifies degraded service
(needs failure hypotheses, metrics)

Objectives of resource management

Satisfy Service Level Objectives (i.e. QoS specs)
- Various quality factors linked to performance (throughput, latency), availability, security
- Various forms of contract: guaranteed, probabilistic, conditional, best effort
- Constraint: ensure fairness
  - Equal treatment for equal rights
  - Differentiated treatment for unequal rights
  - Must guarantee progress (no starvation)

Optimize resource usage
- Avoid overprovision (capacity planning)
- No idle resources if unsatisfied demand
- Maximize utilization, sharing
- Constraints
  - Limited resources
  - Locality, dedicated resources, etc.

A metric is needed in all cases
The objectives may be contradictory; resolved by negotiation or by higher level policy (e.g. weighted mix, etc.)

Challenges of resource management

Some known problems …

Congestion (thrashing)
- Exploding overhead ; no useful work can be done
- Due to overcommitment

Deadlock
- Mutual blocking
- Due to circular dependencies

Violation of fairness
- Starvation
- Inversion of priority
- Due to improper policy design

A general heuristic: admission control
restrict allowed requests, so the admitted requests get acceptable level of service

Well-known solutions
ordered resource classes
global allocation
variable priority ("aging", etc.)

A general principle: separate policies from mechanisms
Mechanisms should be neutral wrt policies
Policies should not impose mechanisms

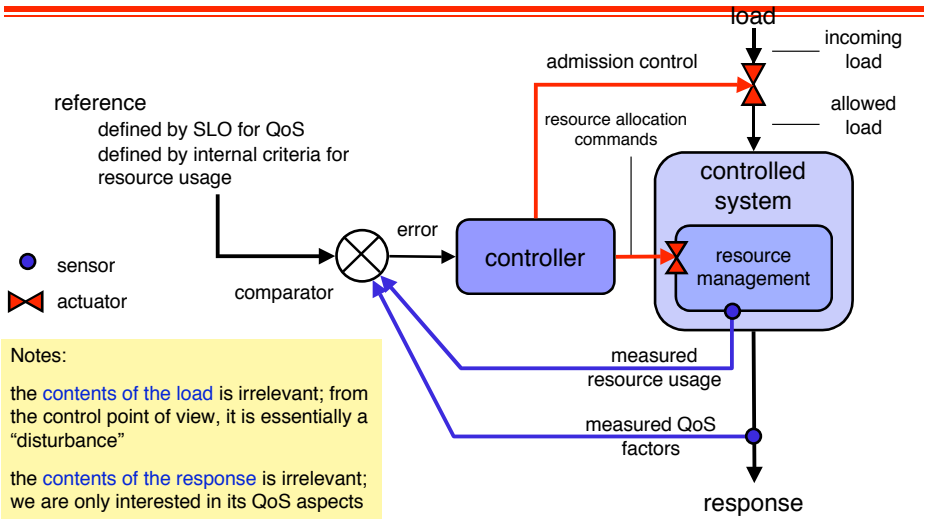
Resource management: typical situations

Requests known in advance, globally sufficient resources
A constrained combinatory optimization problem - solutions known in many usual cases, solvers efficient in real time

Unpredictable requests, globally sufficient resources
(quasi-) optimal policies known in some cases, e.g. *Earliest Deadline First* (EDF) for real-time scheduling

Unpredictable requests, unsufficient resources (the usual case)
The case for control
- Open loop (feedforward, i.e. prediction)
  needs accurate model of the system, usually not available…
  … but may be feasible for a specific subsystem
  assumes disturbances can be observed before they have an effect
- Closed loop (feedback) - the preferred method
  "black box" approach (no model; only need to know the effect of knob turning)
  well adapted to unpredictable inputs and disturbances
  risk of unstability

Dynamic resource management: the big picture



Notes:

the contents of the load is irrelevant; from the control point of view, it is essentially a "disturbance"

the contents of the response is irrelevant; we are only interested in its QoS aspects

Dynamic resource management: observed variables

Two classes of variables are observed

Related to resource usage (on the server)
- Availability (detecting partial failures)
- Actual utilization (e.g. % occupation): CPUs, storage, channels,…
- Local load (e.g. queue length, requests submitted per unit time)
- Throughput (global, per resource): requests processed per unit time

Related to user-perceived QoS
- Latency (time between request send and response receive)
- Availability (% time the system is up for service)
- Performability (a measure of performance in degraded mode)

Resource usage is much simpler to observe, but less representative of QoS

Attempts towards better correlation
- Decomposing response time and relating its components to completion of partial requests
- Using elaborate statistical methods (inferring response time from observations on the server)

Dynamic resource management: controlled variables

The system is controlled through three channels

Controlling the load: admission control

For the whole system, or for local steps

Controlling resource usage: request scheduling, resource allocation and sharing

Changing the system's structure: reconfiguration

A special case of resource allocation: changing the bindings between resources

Technical problems

Implementing the above methods may be a nontrivial task

Global admission control is usually easy (done externally)

However, controlling resource usage may imply access to the inner structure of the system

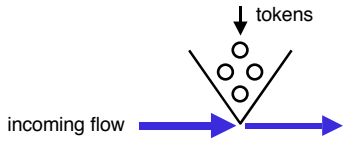
Some actuators for resource management

Actuators for admission control

Accept / Reject (a request, a group of requests)

Flow throttling

a common algorithm: token bucket



A token gives a right to a unit of flow (T bit/s).
Controlling variables:
token issue rate
size of bucket

Actuators for reconfiguration

No accepted standard yet

Dynamic reconfiguration is in itself a whole area of research

A system structured into components with well-identified interfaces for binding control helps implementing reconfiguration actuators (cf Fractal)

Dynamic resource management: control policies

The control policy determines the action to be taken, based on objectives and current observations

Needs at least an approximate model of the system's operation, i.e. the effect of the actuators on the observed variables. This model may be:

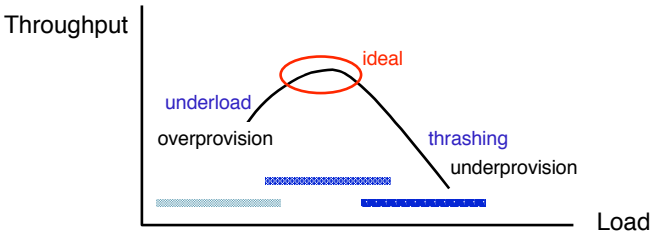
- known by empirical observation

  early load levelers: removing a job improves the performance of the remaining ones

  repair management: restore state as before failure

- determined by identification: fitting model parameters by input-output correlation, assuming the form of the model is known (e.g. piecewise linear)
- determined by analysis; may be tractable for simple subsystem
- in all cases, the model may be improved by learning (a subject of active current research)

Example: Admission control



Admission control: admit additional load only if it does not overload the system

assumes

reasonable estimates of current and additional load

reasonable capacity planning and provisioning

what of rejected load?

dismiss with "try later" message

reschedule

Applying control to computing systems: a bit of history

In the 70s (time-sharing operating systems, first networks):
- Early attempts to apply control to resource management
  - No real success in using theory…
  - … but good empirical results of admission control (e.g. thrashing avoidance; Ethernet)

In the 80s (rise of the Internet)
- Control applied to networks (adaptive routing, flow control, etc.)

In the 90s (networks of workstations, local servers, transaction processing):
- QoS for networks
- Open loop methods (proportional scheduling, fair queuing)

In the 2000s (Internet services):
- Revival of application of control theory to resource management, stimulated by:
  - Growing needs for end user QoS
  - Better structuring of software (open middleware, design patterns)
  - Better understanding of performance issues
- Feedback control, initially for CPU scheduling, but fast extending to other resources
- "Autonomic" computing; also exploring model-based approaches

First book on subject: Hellerstein et al., *Feedback Control of Computing Systems*, Wiley 2004

Case studies

1. An approach to managing a large scale storage utility:
   Chameleon [Uttamchandani et al., 2005]
   self-refining model

S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, G. Agha. *Chameleon: a Self-Evolving, Fully-adaptive Resource Arbitrator for Storage Systems*, Usenix Technical Conference, April 10-15 2005, Anaheim, CA, USA.

2. A proxy-based approach to managing a 3-tier web server:
   Gatekeeper [Elnikety et al., 2004]
   admission control based on service estimates, request scheduling

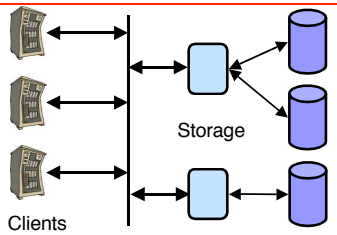
Elnikety, Sarneh and Nahum, Erich and Tracey, John and Zwaenepoel, Willy (2004) A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In Proceedings International WWW Conference, New York, USA.

Other examples:

For 1: M.Karlsson, C. Karamanolis, X. Zhu. *Triage: Performance Isolation and Differentiation for Storage Systems*, 12th IEEE International Workshop on QoS, June 2004, Montreal, Canada.

For 2: Abhinav Kamra, Vishal Misra, and Erich Nahum. *Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites*. In International Workshop on Quality of Service (IWQoS), Montreal, June 2004.

Resource management for a storage system (1)



Service: a large scale storage utility, shared by a number of independent workloads (clients)

Problem: arbitrate resource provision between the clients; currently solved by static provisioning and rule-based reactions, with no guarantees (best effort)

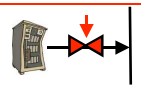
Performance objectives

Performance isolation: guarantee a portion of the shared resource to each client, regardless of other client's behavior

Service differentiation

Performance metrics
- throughput (IO/s): shared, aggregated resource provided by the service
- latency (s): goal derived from application requirements

Resource management for a storage system (2)



Constraints

the requests are often unpredictable: no available model
the storage system is a "black box": control should use flow throttling (rate limiting), a form of admission control

Service Level Agreement (SLA)

Conditional: guarantee upper bound on average I/O latency to a client, provided its throughput is below a specified limit.

The system attempts to meet SLAs, while respecting service differentiation, if present (specifies an order of priority for SLA satisfaction)

In addition, the system attempts to maximize bandwidth usage (no unused bandwidth if requests are pending)

Approaches to storage management

Rules (current commercial systems)

Event-condition-action, defined by administrator. Events are usually crossing of thresholds. Complex to manage, does not scale well

Scheduling

Specifies relative priorities between workloads and individual I/O requests. Works well in common cases, does not handle exception scenarios well (e.g. load surge, failures, changes of policy).

Model-based

Decisions based on model of storage system, solved by constrained optimization. Model may be built beforehand, or created/refined on line. Example: Chameleon.

Feedback-based

Reaction based on difference between prescribed and observed state, assuming a specific form of the control law. Example: Triage.

A model-based approach to storage management

Goal

Guarantee (conditional) SLA for a shared storage service

Approach

build a model of the system
use the model for action ("observe-analyze-act")
evolve the model by feedback

Input

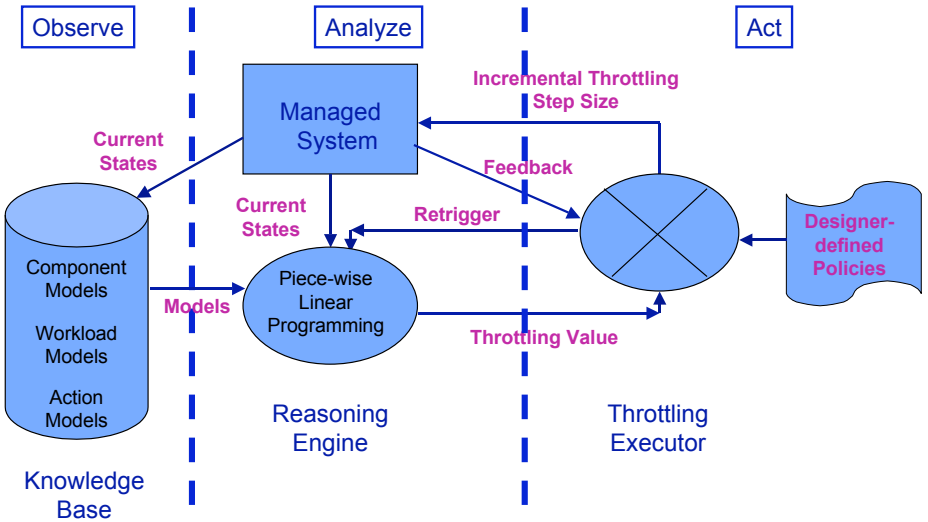
SLAs for workloads (clients)
Current system status (performance)

Output

Resource allocation action (throttling decisions)

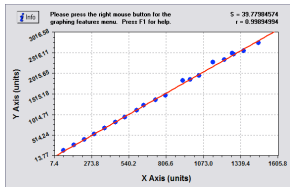
S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, G. Agha. *Chameleon: a Self-Evolving, Fully-adaptive Resource Arbitrator for Storage Systems*, Usenix Technical Conference, April 10-15 2005, Anaheim, CA, USA.

Chameleon: overview of operation



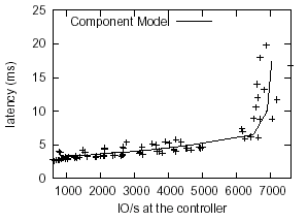
From a presentation by Li Yin (Berkeley), Oasis retreat, Jan. 2005

Chameleon: models



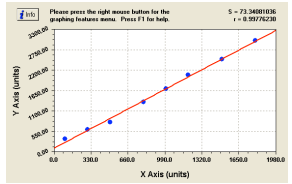
Workload model: load on component (e.g. disk unit) for given request rate

Component_load$_{i,j}$ = $W_{i,j}$(workload j request rate)



Component model: service time for given load at component

Latency$_{ij}$ = Li(I/O rate at controller)



Action model: effect of corrective action on workload

Workload J req. Rate = A_j(Token Issue Rate for Workload J)

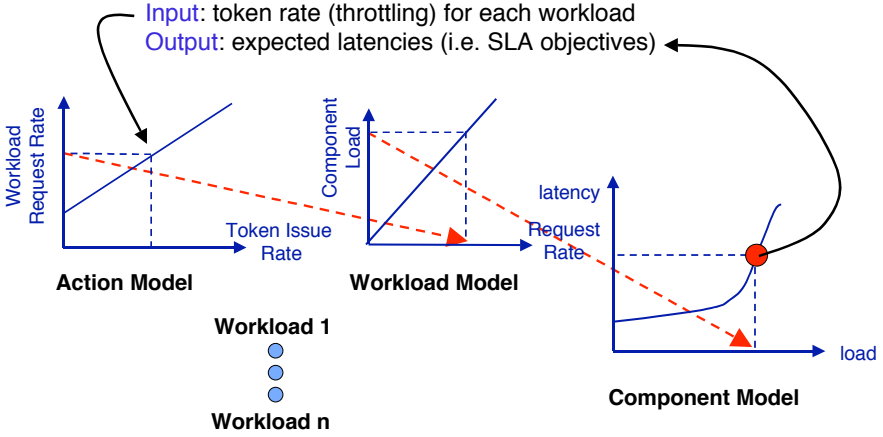
(Piecewise) linear models give a good approximation

Chameleon: reasoning engine

Goal: predict action results by chaining the models together (a constrained optimization problem)

Input: token rate (throttling) for each workload

Output: expected latencies (i.e. SLA objectives)

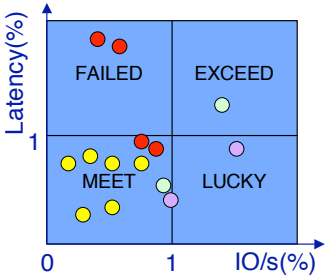


Chameleon: constraint solving

Formulated using Linear Programming

Formulation:

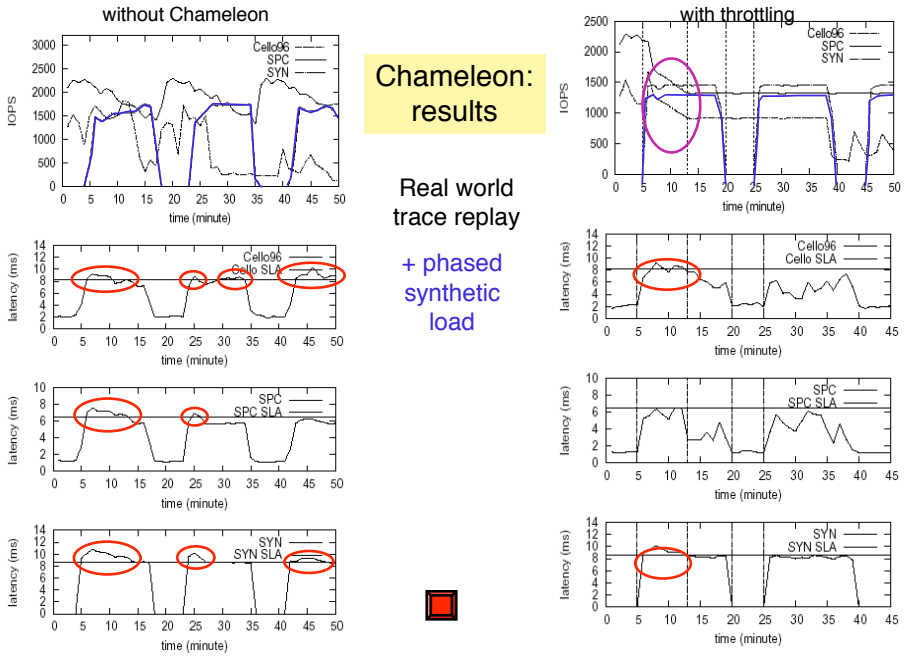
- Variable: Token issue rate for each workload
- Objective Function:
  - Minimize number of workloads violating their SLA goals
  - Workloads are as close to their SLA IO rate as possible
- Constraints:
  - Workloads should meet their SLA latency goals
- Example



$$\text{Minimize} \sum_{i \notin \text{failed}} p_{ai} p_{bi} \frac{[\,SLA_i - T(current\_throughput_i, t_i)]}{SLA_i}$$

where p_{ai}= Workload priority p_{bi} = Quadrant priority

Chameleon: results



Real world trace replay

+ phased synthetic load

Conclusion on storage management

Storage systems are good candidates for autonomic management

Current administration tools are complex to use and far from optimal

The system may be administered as a black box, with unintrusive sensors and actuators

Performance isolation and differentiation is feasible

However, be careful of "coupling" effects, which reduce selectivity

Adaptive methods look promising. They allow the system to react to varying workloads, evolving system conditions, and changing objective function.

Few available results on workload characterization

The impact of failures does not seem to have been explored yet

Managing a 3-tier Web server

Goal

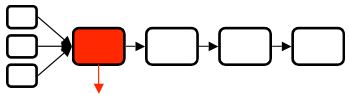
Avoid thrashing in presence of overload: maintain high throughput, low response time

Context & significance

First experiment on MIMO feedback control of the Apache web server [Diao et al., 2002] (using identification techniques to determine the parameters)

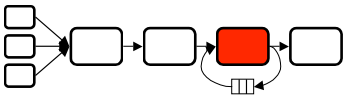
The work presented is an improvement over that pioneering experiment:

- Applies to dynamic content services
- Uses non-invasive techniques (server is unmodified)
- Controls user-related (instead of server-related) variables



Yaksha: admission control with adaptive PI feedback

Not examined here



Gatekeeper: measurement-based admission control at DB tier, with request scheduling

Gatekeeper: admission control and request scheduling

Goal

Avoid thrashing under overload (same as above), keep low latency
(does not consider differentiated service)

Approach

1. Use admission control at DB tier (assuming this is the bottleneck)
2. Reschedule dropped requests
   SJF, with aging to prevent starvation

Technique

1. Preliminary step: determine system capacity by external measurement under known load
2. Identify different request classes (servlets) and dynamically determine load estimate for each class
3. Admission control decision: if new estimated load exceeds capacity, drop the request (reschedule); readmit when estimated load acceptable

Gatekeeper: experimental results (1)

Workload: TPC-W benchmark
Client workload generator: exponentially distributed session duration

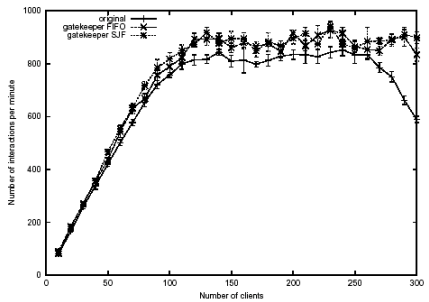


Figure 5: Throughput (MySQL, locking in application server).

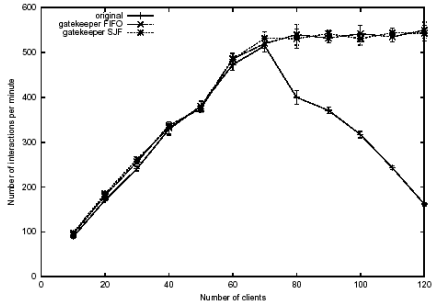


Figure 7: Throughput (MySQL, locking in database).

Gatekeeper: experimental results (2)

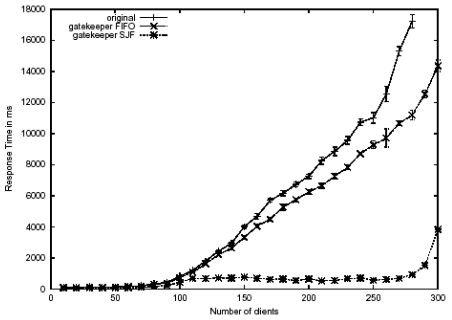


Figure 9: Response time (MySQL, locking in appl. server).

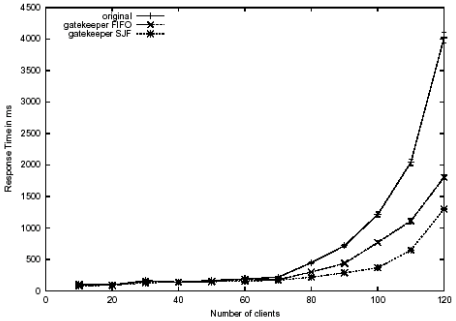


Figure 10: Response time (MySQL, locking in database).

Conclusion on 3-tier Web site control

Comparing the approaches

- Both use user-related controlled variables
- Both use proxies (at different places)
- Gatekeeper needs previous calibration step; maintains and refines workload characteristics; does not cater for configuration update
- Yaksha benefits from dynamic adaptation

Limitations

- Still small scale experiments (single-machine servers)
- Artificial load generation
  - Work needed on parameter selection (may require identification, real traces)
- Requests processed individually
  - Previous work [Cherkassova & Phaal] has shown benefits of session-based admission control
- Model may need refinement
  - Interaction between tiers, etc.

Conclusion on autonomic computing

- **A new, emerging area**
- **Interaction of several domains**
  - **Systems and software**
  - **Control theory**
  - **Artificial intelligence**
  - **Human-systems interfaces**
- **The importance of architecture**
  - **Both observation and action apply to local elements**
  - **Therefore component-based approaches are useful**
  - **New abstractions need to be developed (managers, managed resources)**
  - **Hierarchical structure needed for arbitering policy conflicts**
- **The importance of a quantitative approach**
  - **Criteria needed for QoS**
  - **Correlating resource usage with user-related perfomance figures**
  - **Statistical analysis (to discover patterns of usage)**