# *Prise en compte des ressources dans les composants logiciels parallèles*

Aperçus de l'action RASC et du projet Concerto



F. Guidec

Frederic.Guidec@univ-ubs.fr

# *Plan de cet exposé*

- Action RASC
  - Contexte
  - Motivations
  - Problématique
  - Domaines d'application possibles
- Projet Concerto
  - Axes du projet
  - Liens avec l'action RASC
  - Travaux connexes

# *Contexte de l'action RASC*

- Laboratoire VALORIA : trois équipes de recherche
  - Aglae
    - Génie logiciel (dévpt et maintenance de systèmes logiciels à objets)
  - Orcade
    - Objets répartis pour la conception d'applications distribuées
  - Équipage
    - Interaction homme-machine (via geste et langage)
- RASC = *Resource-Aware Software Components*
  - Action transversale fédérant certaines activités des équipes Aglae et Orcade autour du concept de « composant logiciel conscient de ses besoins vis à vis des ressources »

# *Resource-Aware Software Components*

- Composant logiciel ?
  - Pas encore de consensus sur ce qu'est un C.L.
    - Programme exécutable / Application / Bibliothèque / Objet (Java, C++…) / EJB...
    - Code source / Module / Classe (Java, C++…) / *bytecode* (Java)…
    - Paquetage (Linux RPM, Java .jar)…
    - « Objet » disposant de plusieurs interfaces (LMO'2000, CORBA)
- Une tendance : le développement à base de COTS
  - COTS = *Commercial Off-The-Shelf* (ou *Component Off-The-Shelf*)
  - Composants « sur étagères »
  - ☞ Construction de nouveaux composants ou applications par composition (assemblage) de composants pré-existants

# *Resource-Aware Software Components*

- Constats
  - Un C.L. a besoin de ressources spécifiques pour fonctionner
    - Quelles ressources ? Combien ? Dans quelles conditions ?
    - Informations rarement explicites dans les C.L. actuels
  - Connaître les ressources nécessaires à chaque C.L. serait un « plus »…
    - pour aider à choisir entre plusieurs C.L. sur étagères (COTS)
    - pour déduire (calculer) les besoins spécifiques d'un C.L. produit par assemblage de COTS
    - pour qu'un C.L. puisse « négocier » l'accès aux ressources offertes par la plate-forme sur laquelle il s'exécute (contractualisation, et QoS en retour)
    - pour qu'une plate-forme puisse « surveiller » l'utilisation des ressources par les C.L. qu'elle héberge (prévention du déni de service, facturation...)
    - etc.

# *Quelles ressources ?*

- Ressources « physiques » (proches du matériel)
  - CPU → type, vitesse, …
  - Mémoire → quantité, temps d'accès, niveaux de cache, …
  - Disque(s) → nombre, capacité, temps d'accès, débits d'accès, …
  - Réseau → type (Ethernet, *Token Ring*, ATM…), débits, QoS, …
  - Autres → IHM (clavier, écran, souris, *touchpad*), imprimante, scanner, WebCam, GPS, …
- Ressources « logiques » (services offerts)
  - Bibliothèques dynamiques, paquetages de classes Java
  - Répertoires et fichiers spécifiques avec droits d'accès appropriés
  - Service de *multithreading*
  - Protocoles (TCP, UDP, RTP, RSVP, IP, ATM, …)
  - Services divers (*DNS*, répertoires LDAP ou X.500, …)

☞ *Liste non exhaustive !*

# *Problèmes*

- Comment décrire les besoins d'un C.L. ?
  - Définition d'un formalisme générique utilisable si possible tout au long du cycle de vie des C.L.
    - Langage ad hoc ? DTD XML ? QML ? Objets (réflexifs) ? …
- Comment décrire les ressources disponibles ?
  - A priori avec le même formalisme que ci-dessus
- Comment exploiter ces informations ?
  - À l'aide d'une « arithmétique » associée
    - Combinaison et/ou comparaison des besoins exprimés, vérification de compatibilité entre besoins exprimés et ressources disponibles, etc.
  - À l'aide d'outils d'observation et de contrôle fin des ressources
    - Observation → vérif. ressource utilisée conforme à ressource demandée
    - Contrôle → possibilité de mise en œuvre de politiques de *scheduling*

# *Exemple : besoins spécifiques d'un composant « serveur HTTP » (1/3)*

- Type de CPU, type et version d'OS, bibliothèques
  - Cas d'un C.L. compilé
- Requiert accès réseau
  - Protocole TCP
  - Capacité ouverture port(s) TCP en mode d'écoute (LISTEN)
- Requiert accès système de fichiers
  - Accès en lecture seule, ou...
  - Accès en lecture et exécution (si exécution de programmes CGI possible)
- Requiert capacité *multithreading* (éventuellement)

⇒ Besoins « statiques »

  - Caractérisent toute instance du composant « serveur HTTP »
  - Informations pouvant être associées au code source (ou à l'exécutable)

# *Exemple : besoins spécifiques d'un composant « serveur HTTP » (2/3)*

- Besoins vis à vis du réseau
  - Numéro du port (ou des ports) d'écoute
    - Souvent port 80, mais parfois un autre
  - Clients acceptés ou refusés
    - Filtrage des requêtes en fonction de l'adresse IP source ou du domaine source
- Besoins vis à vis du système de fichiers
  - Chemin(s) du (des) répertoire(s) accessible(s) en lecture
  - Chemin(s) du (des) répertoire(s) accessible(s) en exécution (programmes CGI)
- Nombre de clients pouvant être servis simultanément
  - Nombre de ports TCP pour répondre aux clients
  - Nombre de *threads* pour traiter les requêtes des clients

↳ Besoins « dynamiques »
  - Connus lors de l'installation du serveur HTTP, et éventuellement modifiés ensuite en cours d'exécution

# *Exemple : besoins spécifiques d'un composant « serveur HTTP » (3/3)*

- Autres « besoins » possibles...
  - vis à vis du CPU
    - Niveau de priorité
    - Pourcentage d'activité CPU (globalement / pour chaque *thread*)
  - vis à vis du réseau
    - Niveau de priorité
    - Fraction de la bande passante globale
    - Débits min/max par client desservi
  - vis à vis de la mémoire
    - Quota de mémoire « réservé » pour mettre en œuvre un cache rapide
  - Autres idées ?


# *Exploitation de l'information relative aux besoins « statiques » du serveur HTTP*

- Informations pouvant être exploitées avant son déploiement
  - lors d'une sélection « sur catalogue »
  - dans un atelier de génie logiciel (e.g., choix de C.L. en fonction d'une description de la plate-forme cible)
- Exemple : il est inutile d'essayer d'installer notre serveur HTTP sur une plate-forme…
  - n'ayant pas la bonne combinaison CPU/OS
  - n'offrant pas d'accès réseau via TCP, ou interdisant le mode LISTEN
  - n'offrant pas d'accès à un système de fichiers en lecture (ou lecture/exécution)

# *Exploitation de l'information relative aux besoins « dynamiques » du serveur HTTP*

- Lors de son déploiement sur une plate-forme d'accueil
  - Contrôle d'admission selon disponibilité des ressources demandées
    - e.g., port(s) demandé(s) accessibles ?
    - e.g., répertoire(s) demandé(s) accessible(s) ?
  - Prise en compte des besoins exprimés dans la gestion des ressources
    - e.g., « réservation » bande passante, fraction CPU, q$^{té}$ mémoire, etc.
- En cours d'exécution
  - Supervision des ressources utilisées par le serveur, et vérification de la conformité entre ressources utilisées et ressources demandées
    - e.g., création nouveau *thread*, ouverture nouveau port, etc. → en conformité avec besoins exprimés ?

# *Constats*

- Les besoins d'un C.L. ne sont pas tous connus statiquement
  - L'information se « raffine » tout au long du cycle de vie du C.L.
  - On ne peut identifier une fois pour toutes l'ensemble des ressources à prendre en compte, ni la façon de les caractériser
  - ⇘ Le formalisme développé doit « accompagner » le C.L. pendant son cycle de vie

- Pendant l'exécution, on doit pouvoir maîtriser les ressources
  - Observer l'utilisation qui en est faite par chaque C.L.
  - Contrôler l'accès à ces ressources (droits d'accès, réservation, etc.)
  - ⇘ On doit disposer de mécanismes *ad hoc* (au niveau O.S. ou *middleware*)

# *Plan d'action*

- Environnement de développement / expérimentation
  - Java + Linux
- Description des ressources, expression des besoins
  - Au niveau modélisation → UML ou dérivé (e.g., QML)
  - En cours d'utilisation → objets Java
  - Format intermédiaire « portable » → dialecte XML
- Observation et contrôle des ressources
  - Développement d'outils Java de type intersticiel *(middleware)*
    - Code natif pour interaction avec l'OS (Linux)
    - Interception des accès aux ressources depuis Java (*sockets*, fichiers, etc.)
    - Supervision et gestion des ressources (contrôle d'admission, gestion dynamique de priorité, etc.)

# *Domaines d'application*

- Génie logiciel
  - Assemblage de C.L. guidé par description plate-forme cible
  - « Validation » de C.L. via vérification de comportement à l'exécution
- Code mobile
  - Plates-formes d'accueil « sécurisées » avec supervision des C.L.
  - Migration conditionnée par une négociation préalable des ressources
  - C.L. adaptables (adaptabilité aux variations de l'environnement)
- Qualité de service
  - Renforcement QoS via garanties sur disponibilité ressources (contrats)
- etc.

↳ *Appel à collaborations !*

# *Projet Concerto*

- Objectifs
  - Définition et mise en œuvre d'un modèle de composants parallèles adaptables
    - Modélisation par composants (≈ objets)
    - Exécution sur grilles de calculateurs
    - Réactivité aux variations de l'environnement (adaptation et reconfiguration)

# *Axes du projet Concerto*

- Composant parallèle (C.P.) = collection
  - Collection = données + activités distribuées
  - Ajout, retrait, modifications d'éléments
  - Opérateurs agissant sur l'ensemble des éléments d'une collection
- Prise en compte des ressources
  - Thèmes de l'action RASC considérés dans le cadre particulier des systèmes distribués (déploiement de C.P. sur des grilles)
- Support d'exécution
  - Exécutif adapté au support de C.P. sur grilles
    - Services objets performants (sérialisation, appel à distance…)
    - Mécanismes de supervision et de contrôle des ressources


# *Prise en compte des ressources nécessaires aux composants parallèles*

- Identification de quelques classes de « besoins » types
  - Architecture (CPU, mémoire, disques, protocoles et QoS réseau, topologie, etc.)
  - Services (disponibilité paquetages Java ou code natif, possibilité réservation de ressources, localisation autres composants, etc.)
- Définition d'un formalisme pour l'expression de ces besoins
  - DTD XML + passerelle vers le monde Java ?
  - Langage ad hoc ?
- Définition d'une API d'interaction avec le support d'exécution
  - Expression des besoins
  - Notification d'événements si l'environnement change (variations de la charge CPU, des performances du réseau, etc.)
  - Négociation/renégociation de l'accès aux ressources

# *Supervision et contrôle des ressources*

- Supervision *(monitoring)* de l'utilisation des ressources
  - CPU
  - Mémoire
  - Réseau
  - etc.
- Mécanismes de gestion *(scheduling)* de ces ressources
  - Contrôle d'admission
  - Réservation avec « garantie de service »
  - etc.
- Mécanisme de notification
  - Utilisation de ressources non conforme
  - Ressources indisponibles
  - etc.

# *Travaux connexes*

- *Global Grid Forum*
  - Groupe de travail *Grid Information Service*
    - Dialectes XML pour description « objets » relatifs aux grilles
  - Groupe de travail *Scheduling and Resource Management*
    - Méthodes et outils pour ordonnancement et gestion des ressources (contrôle d'admission, etc.)
- Projet Globus
  - Réservation de ressources et applications « adaptatives »
  - Langage RSL *(Resource Reservation Language)* + API associée
  - Enregistrement et localisation des ressources et services via répertoires LDAP

⇘ Approches focalisées sur l'administration de grilles

- Niveau de granularité = application (et besoins associés)

✋ *Dans Concerto, granularité plus fine (grain = composant)*

# *Résumé*

- Action RASC
  - Développement et support de composants logiciels
  - Prise en compte des ressources
    - Expression des besoins (statiques/dynamiques)
    - Supervision/contrôle des ressources utilisées
- Projet Concerto
  - Composants parallèles adaptables
    - Collections distribuées
    - Prise en compte des ressources spécifiques au contexte de la distribution sur grille
    - Support exécutif pour supervision/contrôle des ressources

*Questions ?*

*Remarques ?*

*Points de vue ?*