La conception des logiciels de très grande taille

J.P. Martin-Flatin, CERN, Switzerland
jp.martin-flatin@ieee.org
http://cern.ch/jpmf/

Université de Neuchâtel, Suisse -- 7 mai 2004

Plan

- Le développement logiciel en général
- Le cas particulier des logiciels de très grande taille (LTGT)
- Etude de cas: distribution de la corrélation d'évènements en présence de systèmes autogéres
- Q&R

Le développement logiciel en général

Processus itératif de développement logiciel

- Analyse des besoins (quoi)
- Conception de haut niveau (comment)
- Conception détaillée (comment en détail)
- Codage
- Débogage (proc. d'assurance qualité)
- Déploiement
- Tests/Utilisation
- *…et on recommence…*

Conception: deux types d'aspects

- Aspects fonctionnels:
  - Résoudre un problème donné
  - Modèles métiers
- Aspects non fonctionnels:
  - Architecture
  - Communication
  - Sécurité
  - *Middleware*:
    - Couplage fort (ex. OO distribués)
    - Couplage faible (ex. Web Services)
  - Technologie de composants
  - Stockage des données (RDBMS, OODBMS, KBMS)
  - …

Problèmes classiques en développement logiciel (1/3)

- On passe son temps à réinventer la roue:
  - Perte de temps, perte d'argent
- Génération après génération, les informaticiens refont les mêmes erreurs:
  - Mauvaise transmission du savoir-faire
- La maintenance de la plupart des logiciels est difficile à faire
- Presque tous les développements logiciels coûtent cher

Problèmes classiques en développement logiciel (2/3)

- Les logiciels requièrent plus de flexibilité que jamais:
  - Les technologies évoluent rapidement:
    - Langages
    - *Middleware*
    - Composants
    - Protocoles
    - …
  - La plupart des applications écrites aujourd'hui devront survivre à des changements technologiques

Problèmes classiques en développement logiciel (3/3)

- De nombreux projets logiciels échouent:
  - Dépassement de budget
  - Echéances manquées
  - Projets arrêtés avant que le logiciel ne soit livré à un seul client (ou utilisateur)

Le cas particulier des logiciels de très grande taille

Tentative de définition

- Beaucoup de définitions floues…
- Quantitativement:
  - C: > 100’000 lignes de code
  - Java: > 1’000 classes
- Qualitativement:
  - Pas possible d’avoir tout le code en tête
  - Nombreux développeurs
  - Longue durée de vie
  - Investissement élevé (argent, années-hommes)
  - Coût du *reengineering* prohibitif

Plusieurs types de LTGT

- Scientifique:
  - Ex.: modèle météorologique, collision de galaxies
  - Beaucoup de calculs, peu d'utilisateurs
  - Centralisé ou fortement distribué (MPP, Grid)
  - Fortran, C
- E-business:
  - Ex.: serveur d'application e-business
  - Systèmes distribués
  - Très grand nombre de clients
  - Java, C#, XML
- …

Problèmes propres aux LTGT

- Retour sur investissement à long terme
- Le TCO est souvent impossible à calculer
- Le débogage est difficile
- Les tests sont difficiles à réaliser

Problèmes de conception propres aux LTGT

- Aucun

Problèmes de conception critiques pour les LTGT

- Passage à l'échelle (*scalability*):
  - Serveur Web: 10'000, 100'000 ou 1'000'000 requêtes par jour?
- Robustesse
- Hétérogénéité:
  - A l'instant t
- Fréquents changements de technologie:
  - Au fil du temps
- Changements de cahier des charges:
  - Le *reengineering* complet n'est pas possible

Questions que se pose un concepteur (1/2)

- Où sont les goulets d'étranglement de mon applications? Comment les supprimer?
- Quels patterns de conception favorisent le passage à l'échelle?
- Pour un cas d'application donné, quelles sont les erreurs de conception typiques (*antipatterns*) à ne pas faire?
- Comment rendre mes classes métiers indépendantes du *middleware* et de la technologie de composants de mon application?
- Comment utiliser des données auto-descriptives pour se prémunir contre l'hétérogénéité des données que reçoit (et recevra) mon application?

Questions que se pose un concepteur (2/2)

- Comment rendre mon application indépendante des protocoles de communication pour l'immuniser contre les changements de protocoles à venir?
- Comment exploiter au mieux MDA, les modèles UML multicouches et le raffinement des modèles objets pour concevoir mon application de façon à ce qu'elle puisse tourner dans un environnement hétérogène?
- Comment concilier les contraintes de flexibilité, d'évolutivité et de sécurité? Comment garantir la sécurité d'une application dont l'architecture devra évoluer avec le temps?

Comment résoudre ces problèmes?

- Architecture logicielle
- *Reengineering*
- Modélisation
- …
- Méthodes non-techniques

Architecture logicielle

- Patterns et *antipatterns*:
  - Réutilisation conceptuelle
- Définir les dépendances entre sous-systèmes:
  - Couplage faible ou fort
  - ex.: SOA ou OO distribué
- Séparation des aspects fonctionnels et non-fonctionnels
- Langages de définition d'architectures logicielles

Reengineering

- Maintenir la flexibilité architecturale à tout prix
- Pas de duplication de code
- Pas de « on le fait comme ça pour des raisons historiques »
- Certains prônent un processus ultra-itératif de développement logiciel:
  - Beaucoup de cycles courts
  - Découverte et raffinement des besoins au fur et à mesure
  - XP, méthodes agiles

Modélisation

- Modèles UML multicouches
- *Model-Driven Architecture* (OMG)
- UML 2.0

Autres méthodes

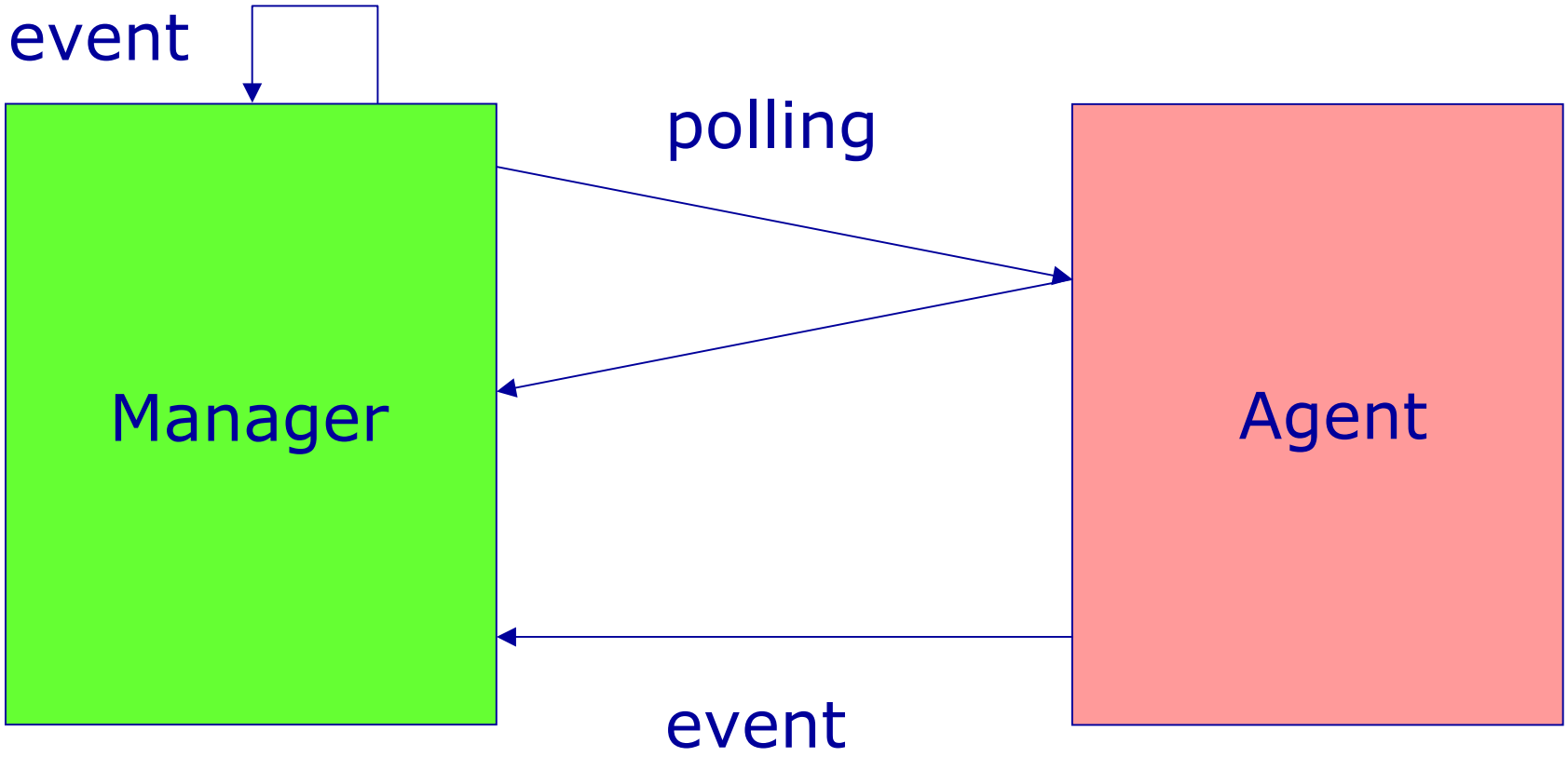
- Données auto-descriptives:
  - XML
  - Ontologies:
    - Traduction automatique
- Ingénierie des exigences (*requirements*)
- Ingénierie de la performance logicielle (SPE)
- Changer le design pour faciliter le débogage et les tests

Méthodes non techniques

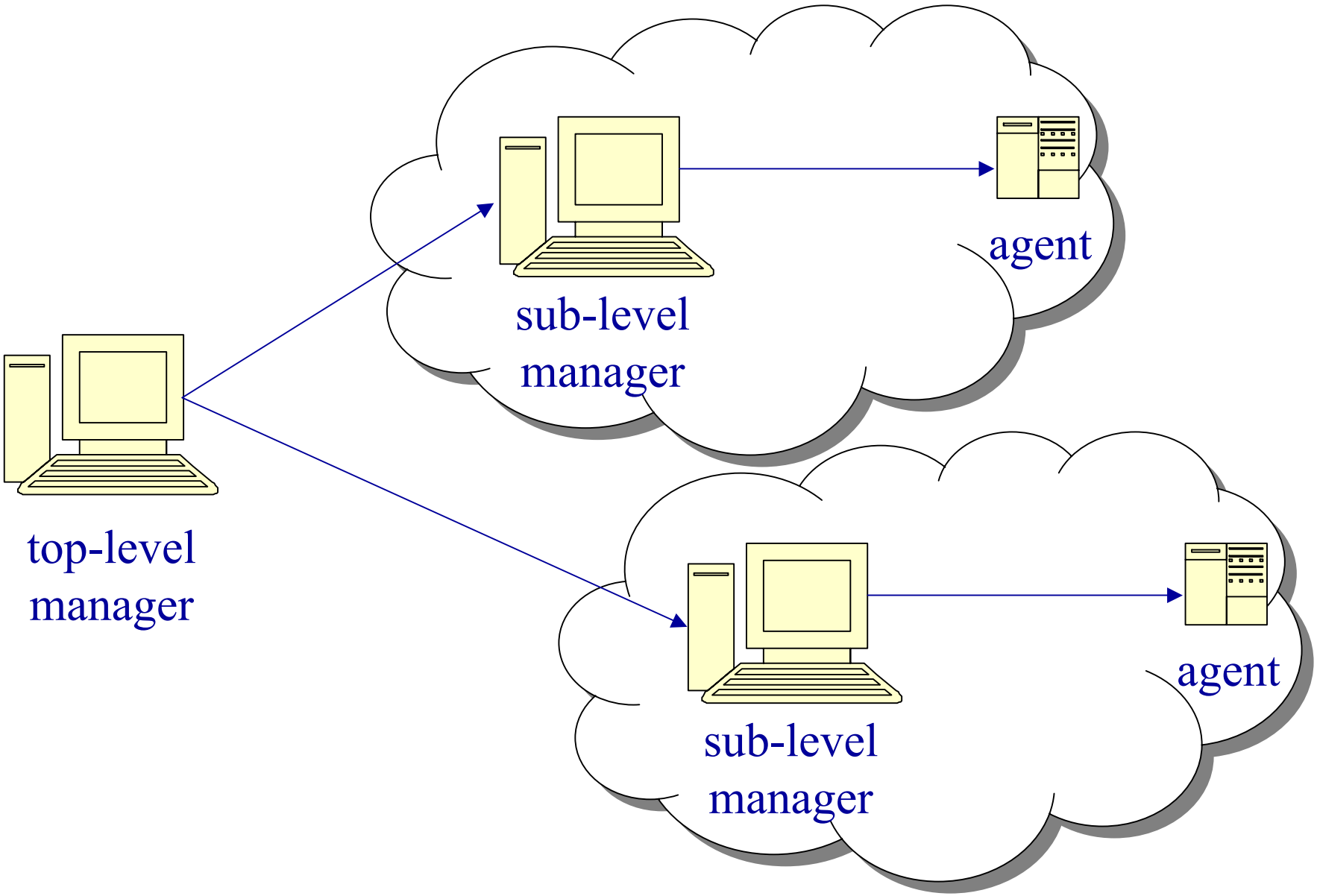
- Formation, éducation
- Favoriser financièrement la réutilisation de code et la réutilisation conceptuelle
- Acheter de meilleurs ateliers de production logicielle
- Acheter de meilleurs outils de gestion de projet
- …

Etude de cas :
La corrélation d'évènements distribuée

Le paradigme agent-gestionnaire



Modèle hiérarchique de gestion



Corrélation d'évènements

- Objectif: analyse de cause principale
- Localisation: un corrélateur par gestionnaire
- Techniques utilisées:
  - Machines à états finis
  - Raisonnement à base de règles
  - Raisonnement à base de cas
  - Raisonnement à base de modèles
  - Codages binaires (*codebooks*)
  - Graphes de dépendances probabilistes (réseaux bayesiens)
  - Réseaux de neurones
  - …

Deux règles empiriques

- Un gestionnaire gère au plus quelques centaines d'agents
- Pour gérer chaque agent dans son domaine, un gestionnaire doit:
  - Monitorer entre 1 et 10 « objets gérés » toutes les 10-15 minutes
  - Recevoir des évènements/notifications de temps en temps (très variable)

Erreur architecturale actuelle

- Supposition:
  - un serveur d'e-business est un agent comme un autre
- Réalité:
  - un serveur d'e-business un peu complexe (distribué sur 5-10 machines) nécessite de monitorer autant d'objets gérés (plusieurs milliers) que tous les agents d'un domaine
- Conséquence:
  - un gestionnaire ne peut pas gérer un serveur d'e-business en plus de tous les agents de son domaine

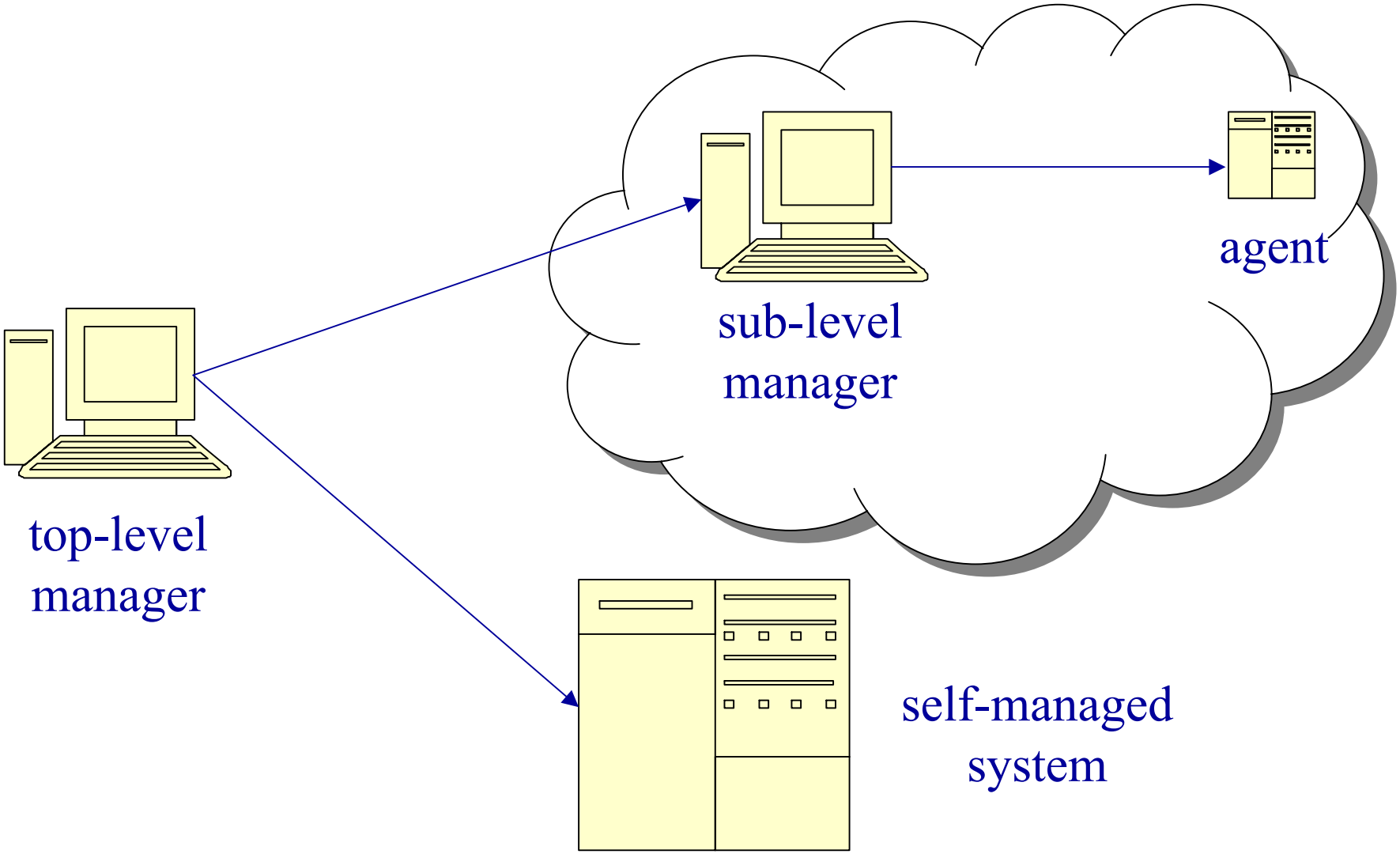
Changement architectural proposé

- Les serveurs d'e-business doivent être transformés en systèmes autogérés:
  - Chacun a son propre corrélateur d'évènements
- Comme ces serveurs n'ont pas la vision de bout en bout d'un SLM ou TLM, ils ont besoin de moniteurs extérieurs (SLA monitoring)
- Les bases de règles, de cas, de modèles… utilisées par ces systèmes autogérés doivent pouvoir être mises à jour par un SLM ou TLM

Un système autogéré, c'est quoi?

- Autonome (cf. robot ou système intelligent en IA distribuée)
- Capable de détecter les problèmes qui surgissent dans ses sous-systèmes
- Capable de trouver les causes de ces problèmes
- Capable de corriger ces problèmes sans l'intervention d'une tierce partie
- Capable de recevoir des ordres, de nouvelles configurations…
- Capable d'exploiter des données venant d'une source extérieure

Modèle hiérarchique de gestion



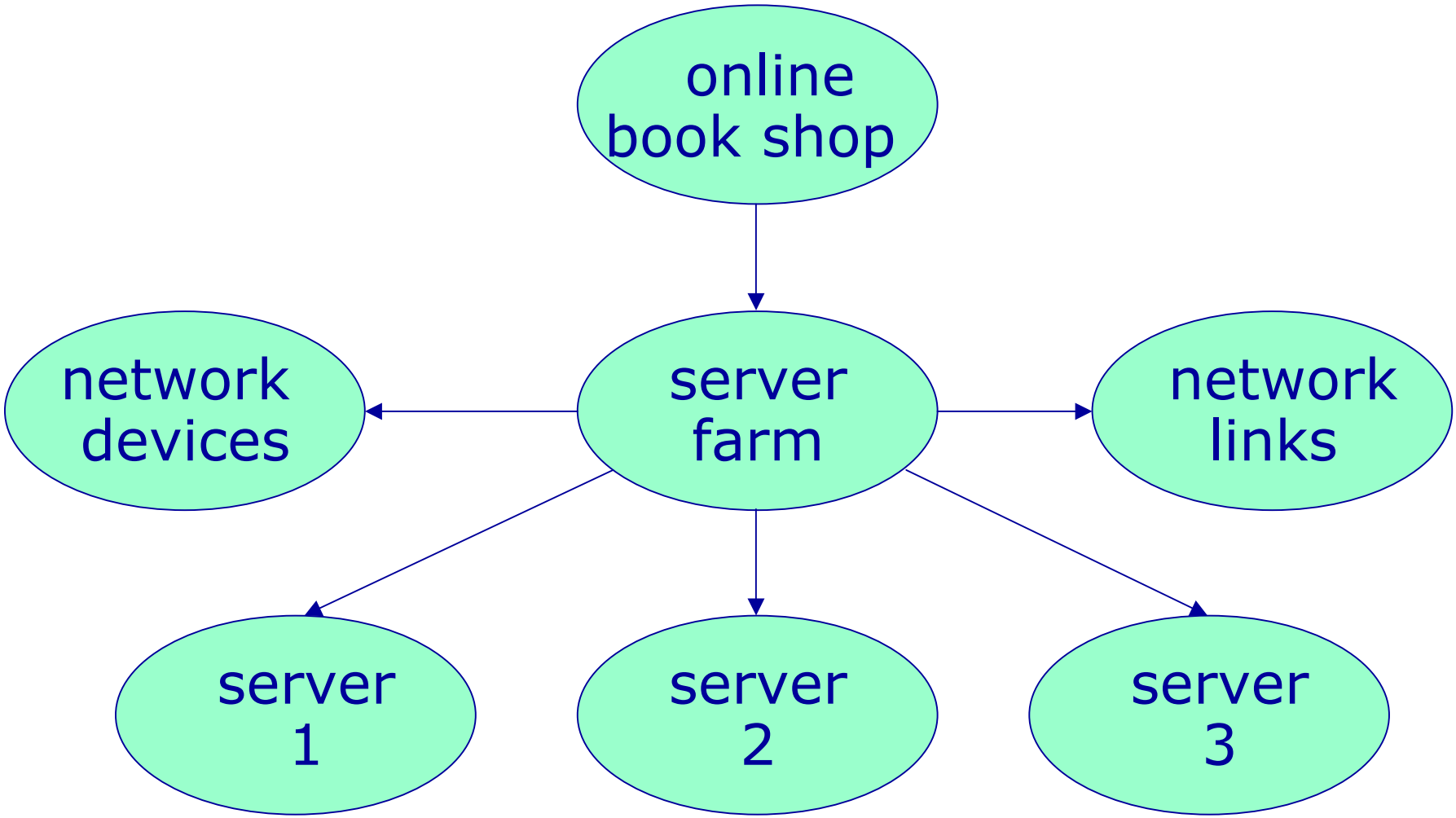
Graphes de dépendances dynamiques (1/2)

- Avantage 1: le polling est effectué localement, donc l'*overhead* est plus faible, donc on peut se permettre d'en faire plus, notamment pour de courtes durées
- Avantage 2: les évènements étant tous envoyés localement, il est plus facile d'avaler une rafale
- Le système autogéré n'a donc plus besoin de maintenir à jour en permanence un graphe de dépendances complet et un « modèle du monde » complet

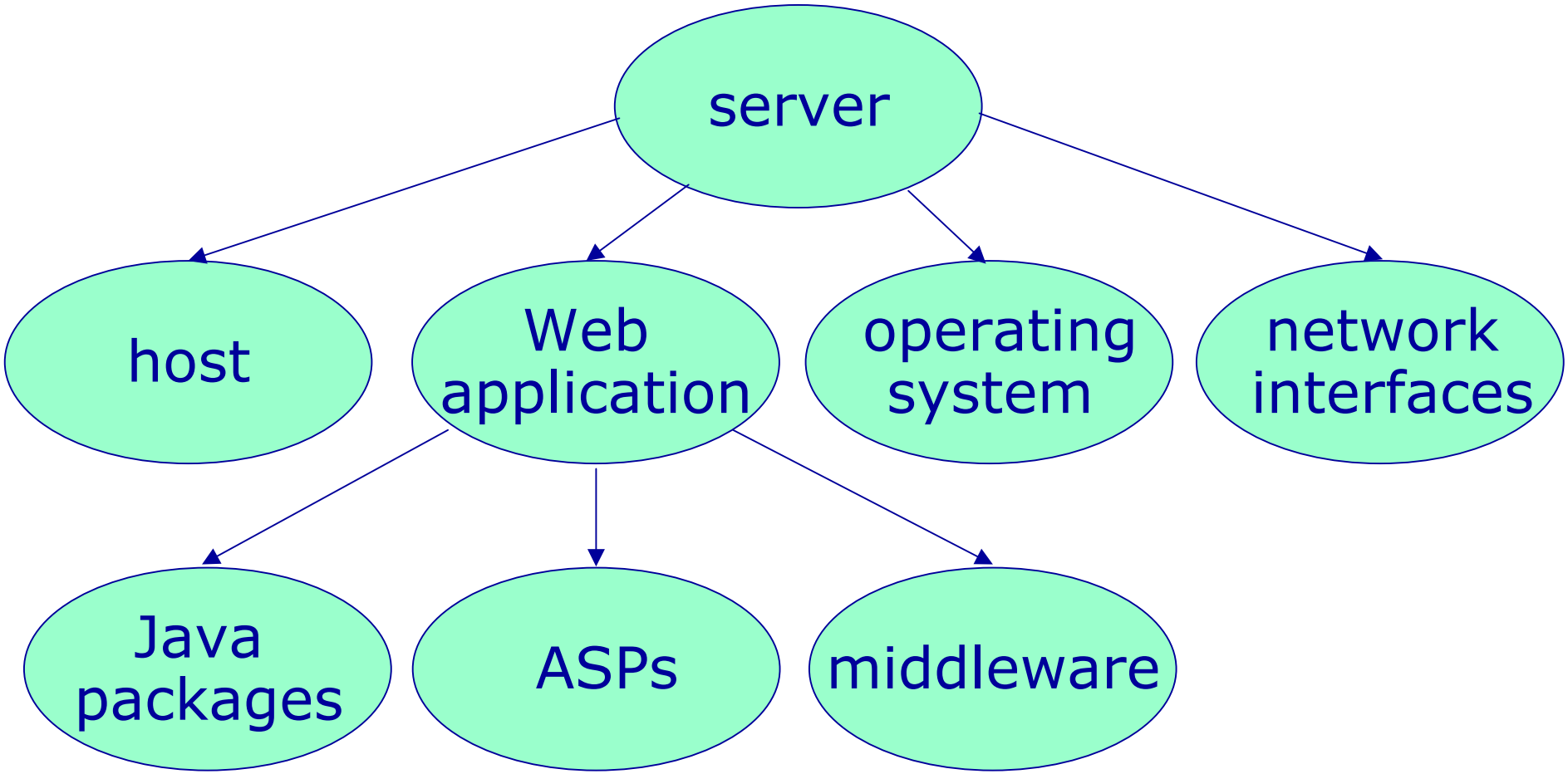
Graphes de dépendances dynamiques (2/2)

- Le système autogéré peut « poller » ses sous-systèmes et composants à la demande, et raffiner ainsi son « modèle du monde » pendant qu'il progresse dans l'investigation d'un problème
- Il peut également raffiner son arbre de dépendances à la demande et découvrir dynamiquement de nouvelles dépendances
- Si ses constituants matériels ou logiciels le permettent, il peut temporairement basculer un composant en mode débogage pour acquérir plus d'informations:
  - irréaliste avec un SLM

Graphe de dépendances partiel



Raffinement du graphe de dépendances



Conclusion

- Les LTGT posent beaucoup de problèmes intéressants
- Etudes de cas:
  - Systèmes autogérés
  - Corrélation d'évènements distribuée
  - Grilles de traitement de données
  - Gestion des services offerts sur réseaux ad hoc sans fils
  - …