Un modèle génératif pour le développement de serveurs Internet

Soutenance de thèse de l'Université Paris-Est

Gautier LOYAUTÉ

Gautier.Loyaute@univ-mlv.fr

5 Septembre 2008

Laboratoire d'Informatique
Institut Gaspard-Monge, UMR8049

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

UNIVERSITÉ
PARIS-EST

Les défis des serveurs sur Internet (1)

De plus en plus de clients. . .

- Google : ∼250 millions de requêtes / jour (2000). . .
- ∼1 milliard de requêtes / jour (actuellement)

dont le nombre peut varier très brusquement.

- facteur 10 en ∼10 secondes pour les sites d'informations en ligne !

Les défis des serveurs sur Internet (2)

Pour des applications de plus en plus diverses...

- information en ligne (CNN, site d'une université)
- payement en ligne (Amazon, Paypal)
- calcul haute performance (SETI)
- liste de diffusion

et sur des matériels de plus en plus hétérogènes.

- serveur dédié (multi-processeur)
- routeur / box
- assistant personnel

Les besoins exprimés

1- Besoins d'efficacité

2- Besoins de portabilité

3- Besoins de sûreté

Besoins d'efficacité

Les serveurs ont besoin de :

- monter en charge
- minimiser la latence
- maximiser la bande passante

Entrelacer le traitement de plusieurs clients

Entrelacement du traitement de plusieurs clients

Deux grandes philosophies :

- compétition
- coopération

Les différentes techniques d'entrelacement sont appelées modèles de concurrence.

Pas de consensus sur le meilleur modèle de concurrence !

- *Why Events are a Bad Idea (for High Concurrency Servers) ?*
  In Workshop on Hot Topics in Operating Systems, 2003.
- *Why Threads Are a Bad Idea (for most purposes) ?*
  In USENIX Annual Technical Conference, 1996.

Besoins de portabilité

Pas de consensus sur le meilleur modèle de concurrence !

S'adapter aux matériels / ressources disponibles...

et aux besoins de l'utilisateur.

Abstraction vis-à-vis du modèle de concurrence !

Besoins de sûreté

Plus vite une erreur est détectée, moins chère est sa correction !

- éviter les comportements non désirés. . .
- et les interblocages

Utiliser des méthodes de vérification de logiciels

- complexité

Abstraction vis-à-vis de l'application à vérifier !

Objectif

Outil pour assurer la conception et la construction de serveurs Internet performants, portables et sûrs !

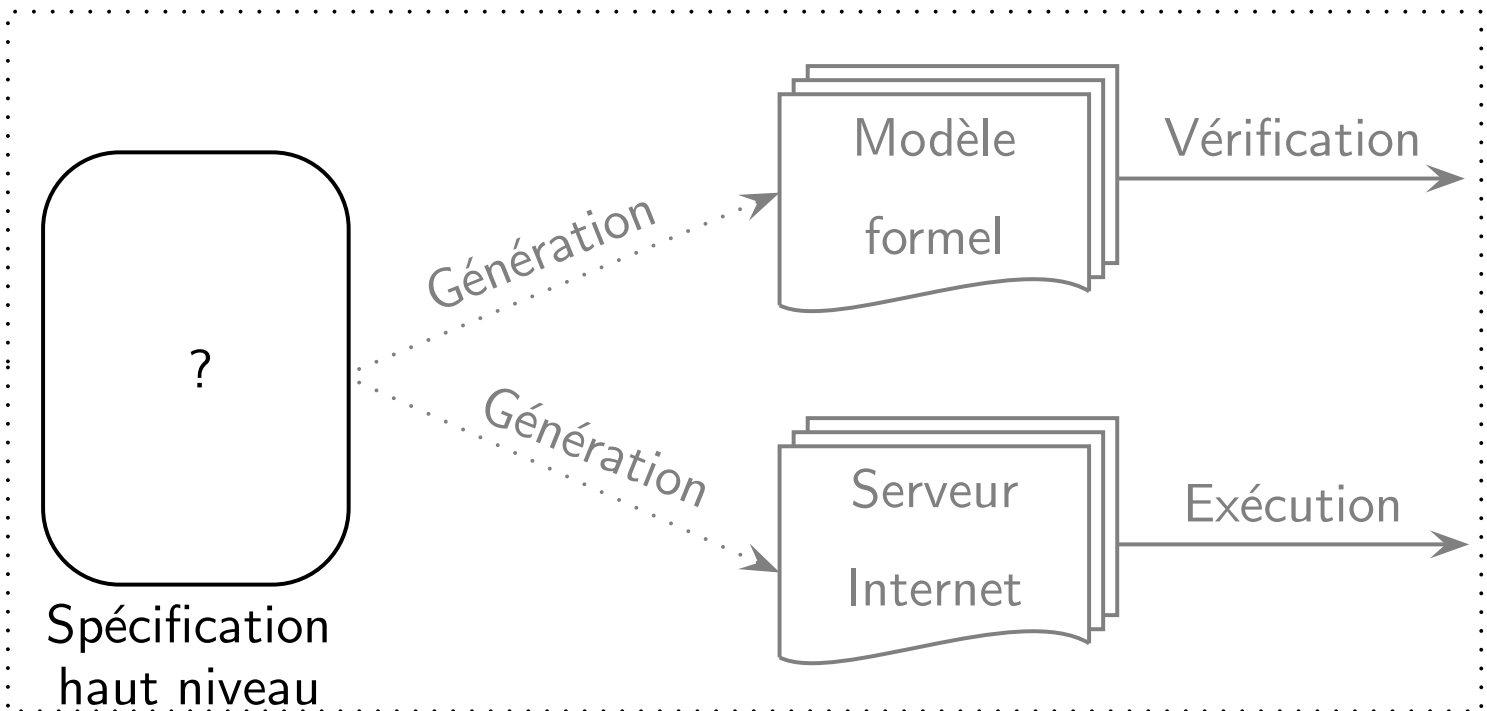
Idée

Utiliser une abstraction unique qui est dédiée à un domaine spécifique !

Saburo

http ://igm.univ-mlv.fr/∼loyaute/project/saburo/

Saburo : Abstraction



Spécification haut niveau de Saburo (1)

Les besoins :

- indépendence vis-à-vis du modèle de concurrence
- transformation aisée vers un modèle formel
- facilité d'apprentissage et d'utilisation pour le développeur

Les choix :

- utiliser une machine à états finie
- décomposer la spécification selon :
  - la concurrence
  - les E/S
  - la logique
- offrir une API dédiée en Java

Spécification haut niveau de Saburo (2)

Concurrence :

- six modèles de concurrence prédéfinis

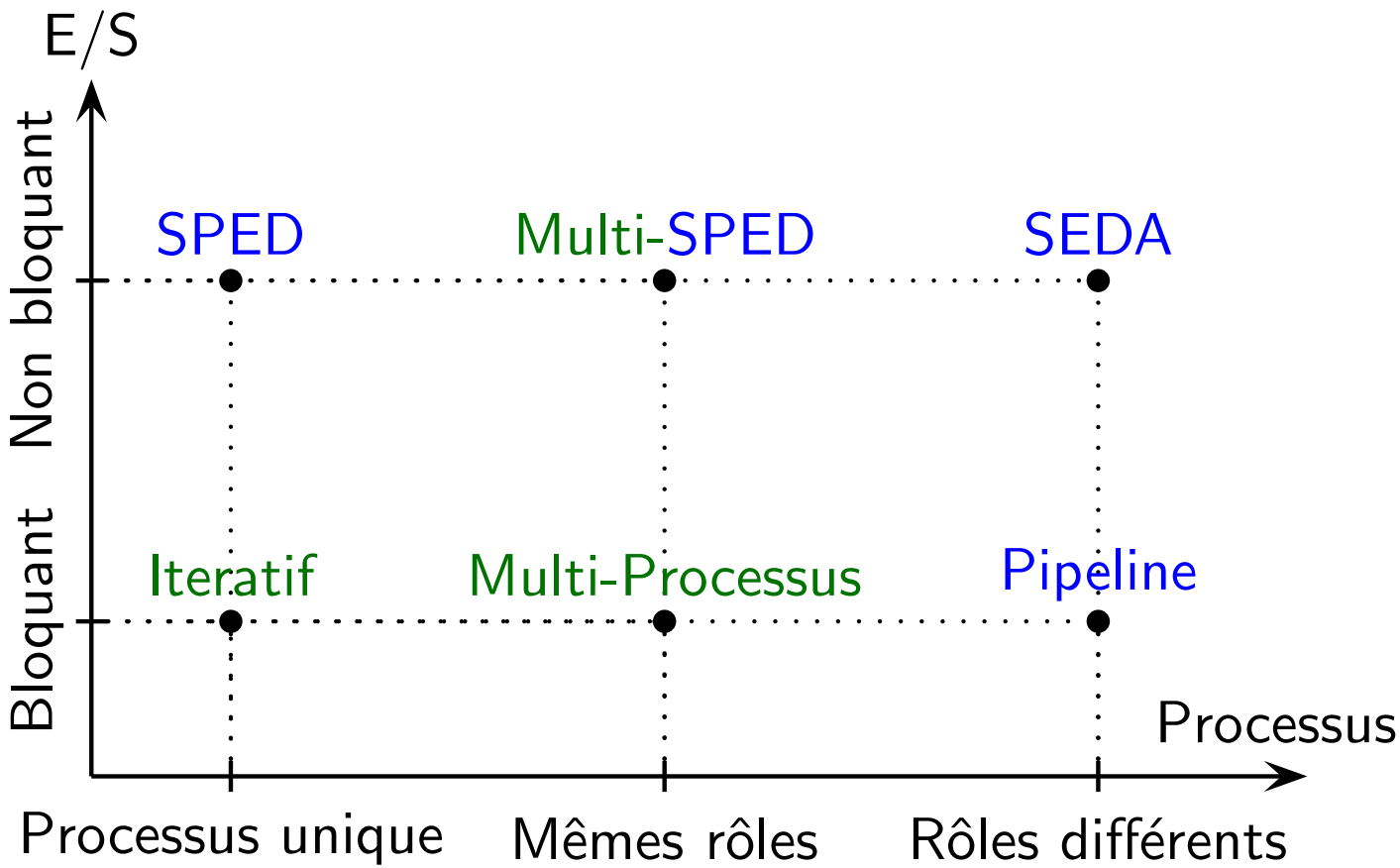
Graphe de communication :

- les synchronisations, les communications et la logique.

Code métier : les stages

- suite d'instructions en Java
- deux contraintes fortes !
  - aucune synchronisation
  - au plus une opération d'E/S

Six modèles de concurrence prédéfinis

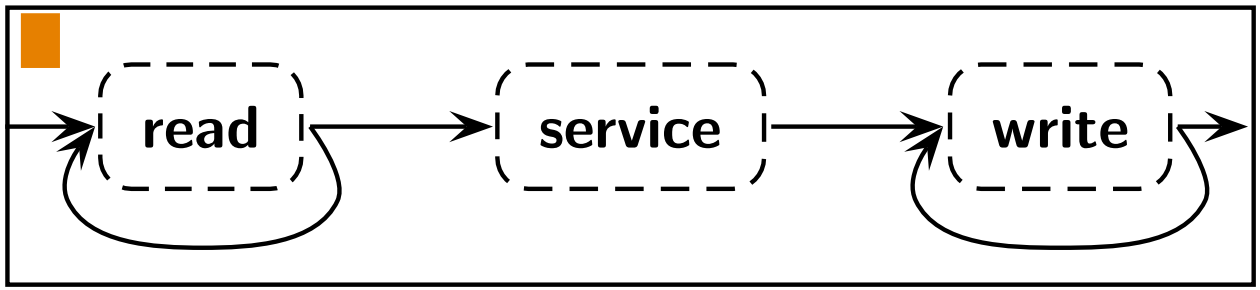


Compétition

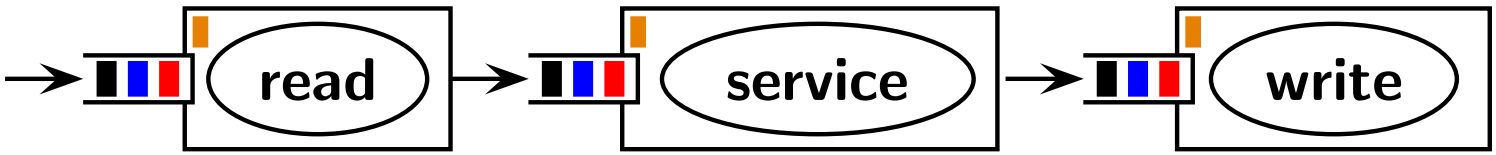
Coopération

Trois exemples de modèles de concurrence

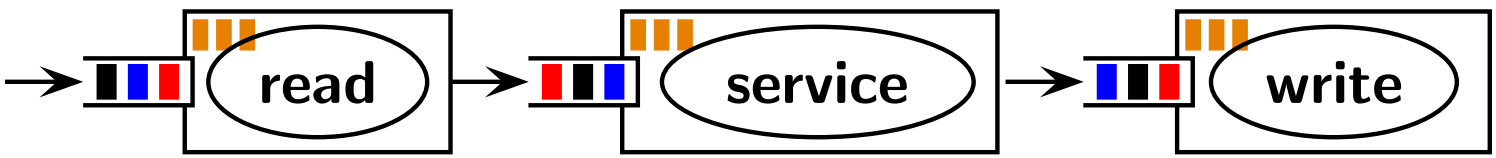
Modèle Itératif (1 seul processus, E/S bloquante)



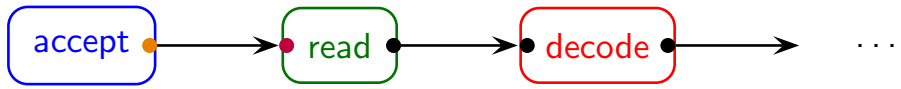
Modèle Pipeline (Rôle différent, E/S bloquante)



Modèle SEDA (Rôle différent, E/S non bloquante)



Exemple de graphe de communication



Connexion des éléments du graphe (Ant) :

```
<connect source="fr.umlv.saburo.core.stage.AcceptStage"
         sink="fr.umlv.saburo.core.stage.ReadStage" />
<connect source="fr.umlv.saburo.core.stage.ReadStage"
         sink="fr.umlv.saburo.samples.http.stage.DecodeStage" />
...
```

Evénements de communication :

```
public interface OutputAcceptEvent {
  public void setSaburoSocket(SaburoSocket s);
}
public interface InputReadEvent {
  public SaburoSocket getSaburoSocket();
}
```

Le code métier : les stages

Spécifiés par le développeur

- traitement fondamental du serveur (méthode `handle()`)
- langage Java

Deux contraintes fortes

- aucune synchronisation
- au plus une opération d'E/S

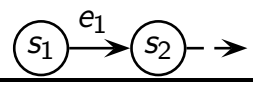
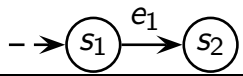
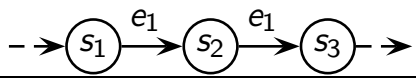
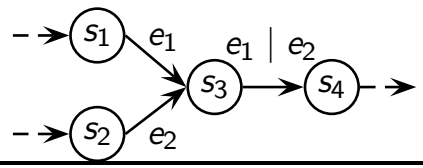
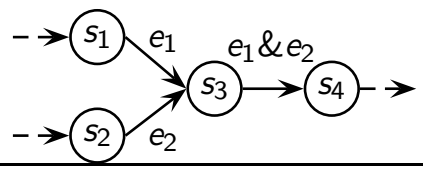
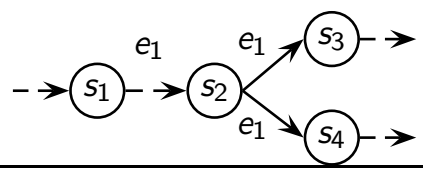
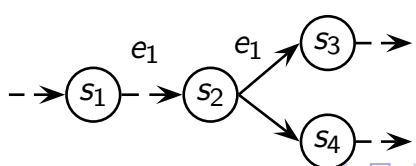
Quelques stages fournis « sur étagères »

- acceptation d'une nouvelle connexion réseau
- lecture sur une connexion réseau
- écriture sur une connexion réseau

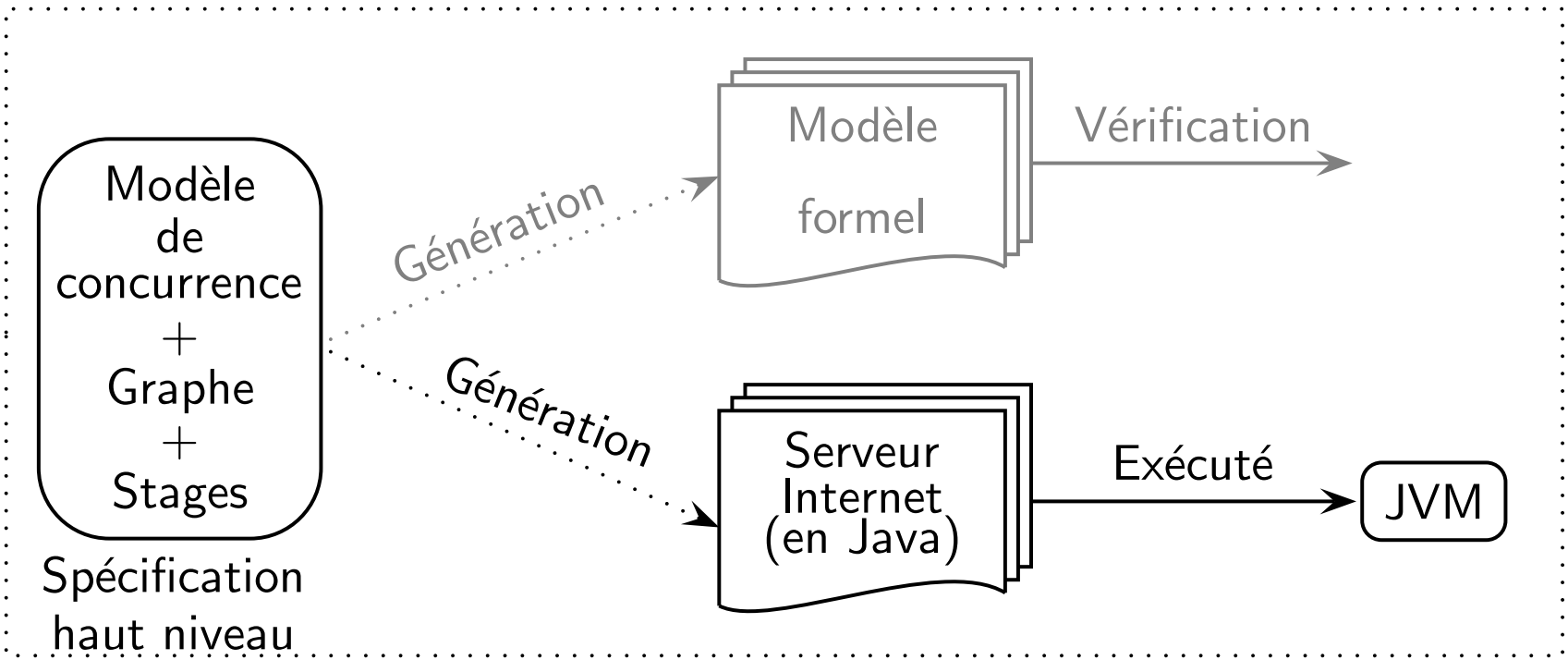
Exemple d'un stage fourni par Saburo

```
@Initial
public class AcceptStage {
  private SaburoServerSocket server;
  public AcceptStage(Configuration conf) {
    server = conf.getSaburoServerSocket("acceptStage.SaburoServerSocket");
  }
  public void handle(StageContext<OutputAcceptEvent> ctx,
                     OutputAcceptEvent out) {
    SaburoSocket client = server.accept();
    out.setSaburoSocket(client);
    ctx.dispatchToSuccessor(out);
  }
}
```

Les différents types de stage

@Initial	
@Final	
@Stage	
@Collector	
@Combiner(E1 & E2)	
@Multicaster	
@Router(E1->S3)	

Saburo : le processus de génération



La phase de génération (1)

Contexte selon le modèle de concurrence

- utilisé pour envoyer les événements de sortie
  - compétition : appels de fonction
  - coopération : files locales
- implanté comme un *décorateur*

Boucle principale selon le modèle de concurrence

- mécanisme de sélection dans le cas d'E/S non bloquante

```
while(true)
  selector.doSelect() ;
```

Génération du code des événements

- selon les interfaces des événements

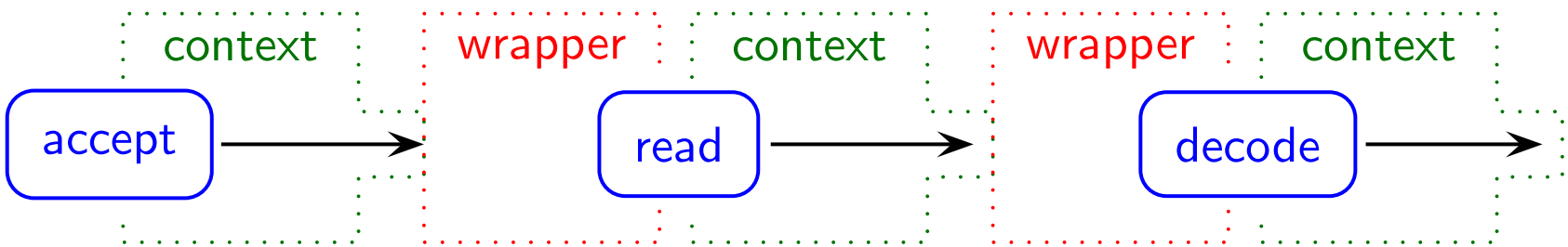
La phase de génération (2)

Wrapper :

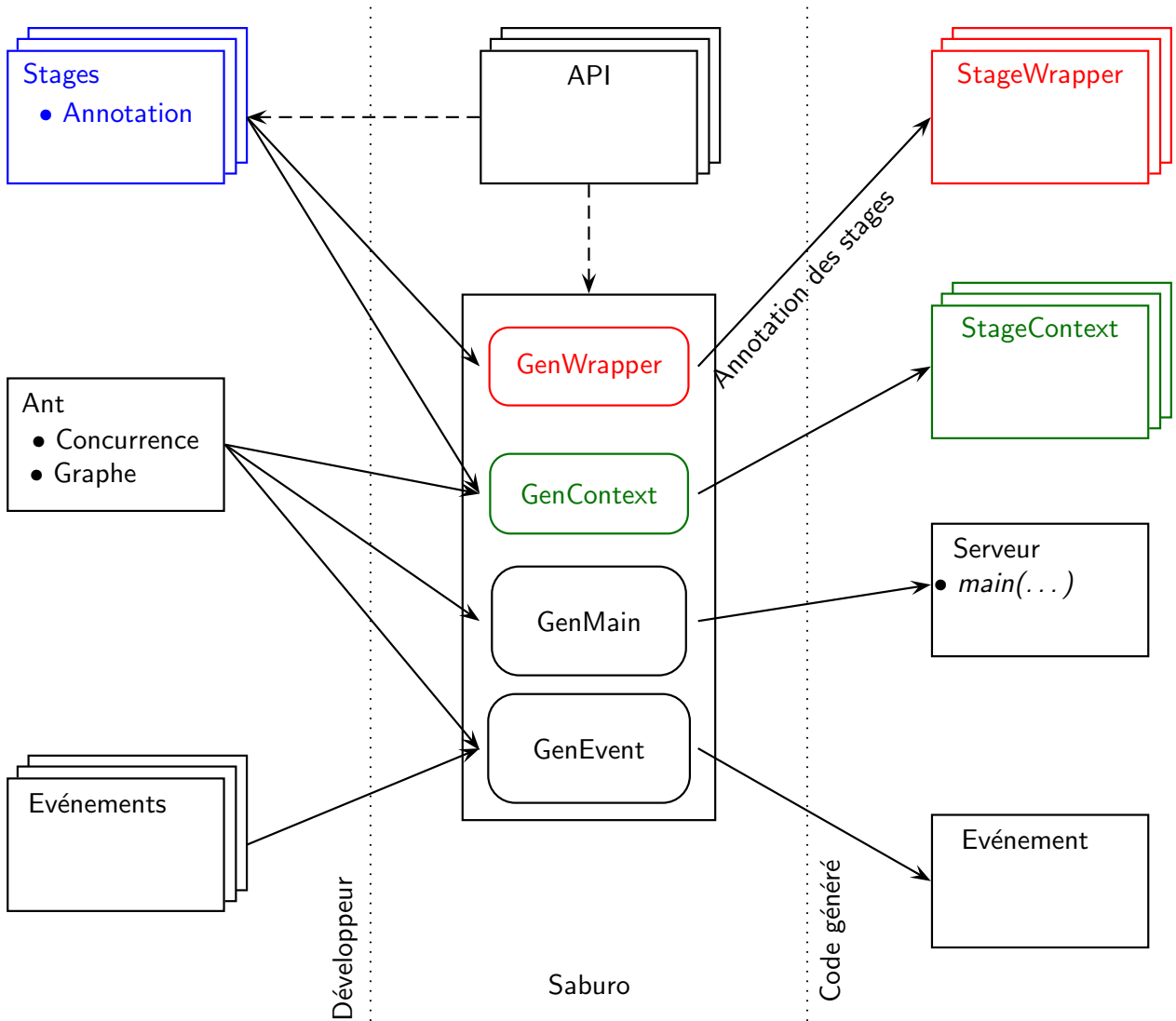
- code exécuté avant le stage

Context :

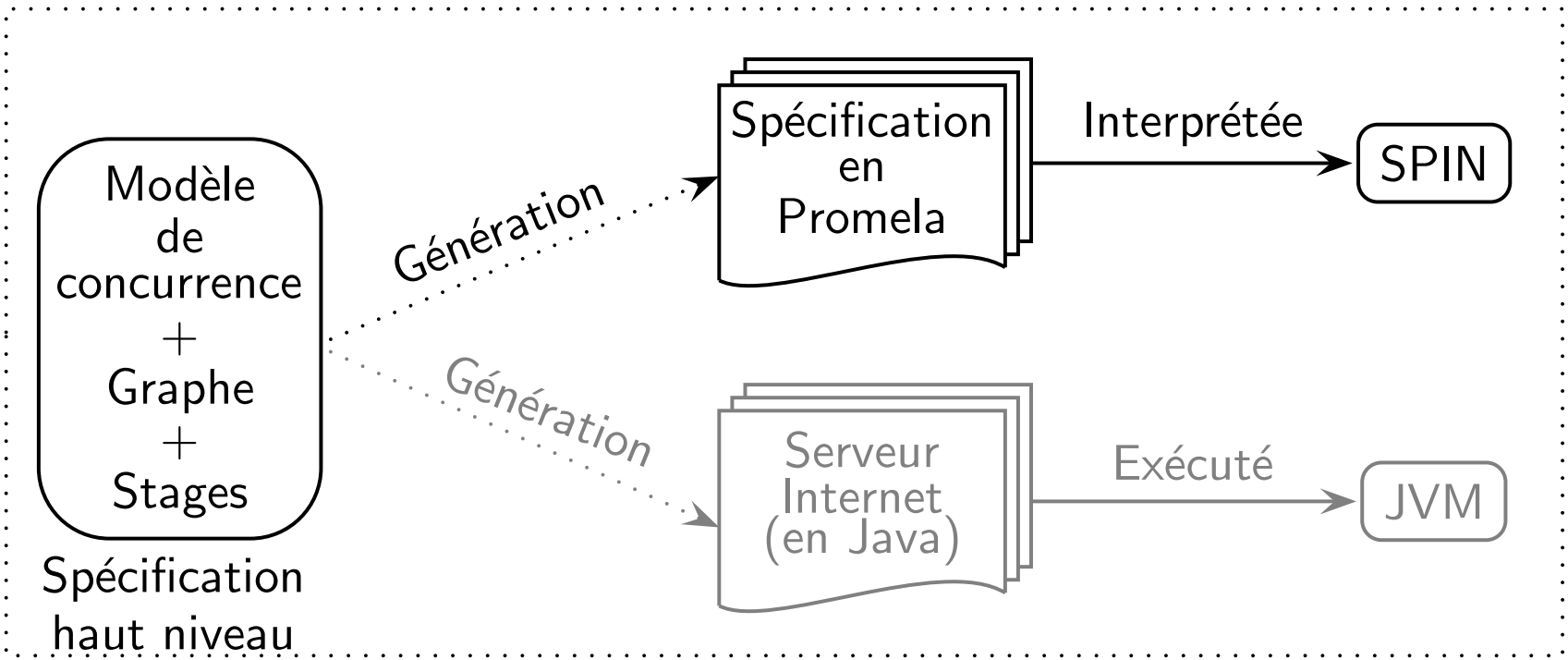
- code exécuté après le stage



La phase de génération (3)



Saburo : le processus de vérification



SPIN : un vérificateur de modèle

http ://spinroot.com/spin/whatispin.html

Langage de spécification : **Promela**

Modèle sémantique basé sur un automate fini

Vérification d'un ensemble de propriétés basiques :

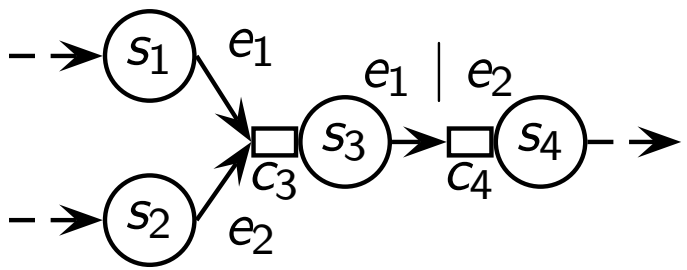
- interblocages
- états inatteignables

Utilisation de formules de logiques temporelles

Génération du modèle formel (1)

Spécification du graphe en Promela

- patrons prédéfinis pour chaque type de stage
- exemple : un stage « collecteur »

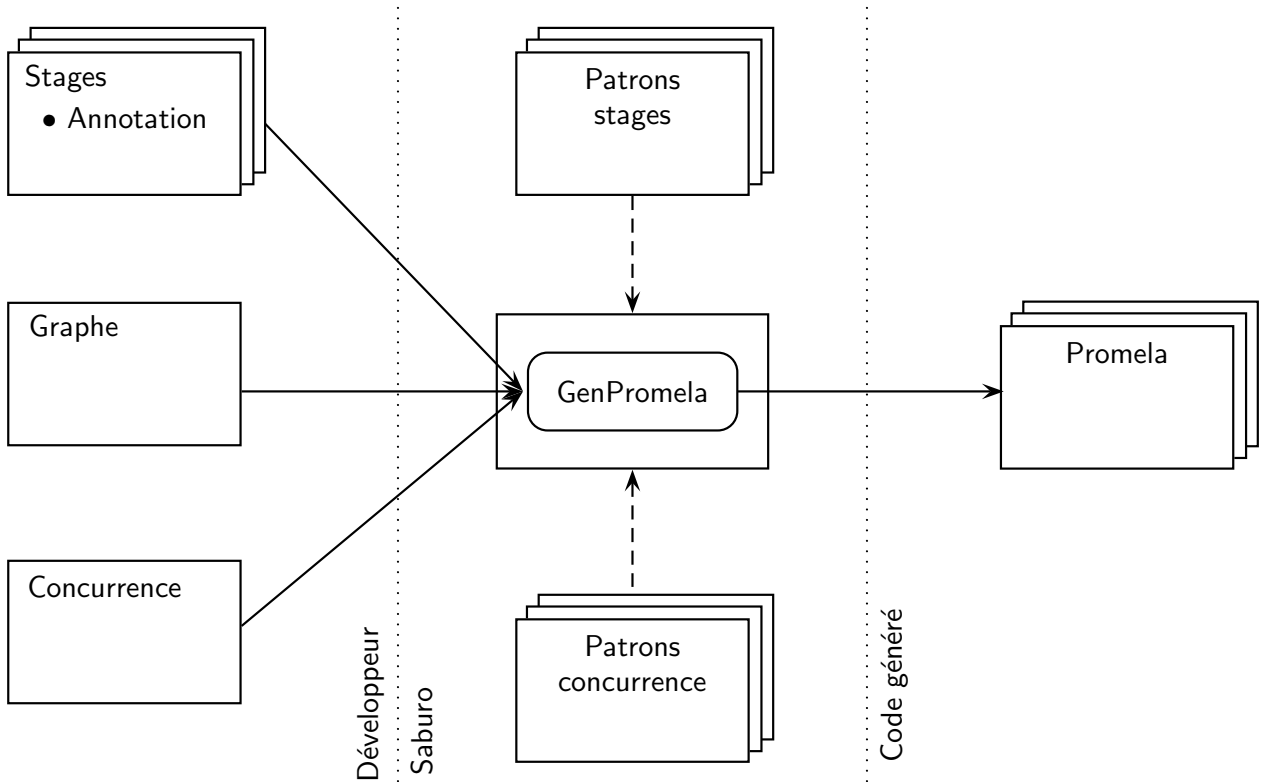



```
s1 : c3 !e1 ;
s2 : c3 !e2 ;
s3 : c3 ?e ;
     c4 !e ;
s4 : c4 ?e ;
```

Le modèle de concurrence

- un patron pour chaque modèle prédéfini

Génération du modèle formel (2)



Vérification avec SPIN

Propriétés vérifiées par défaut

- les états inateignables
- l'absence d'interblocages

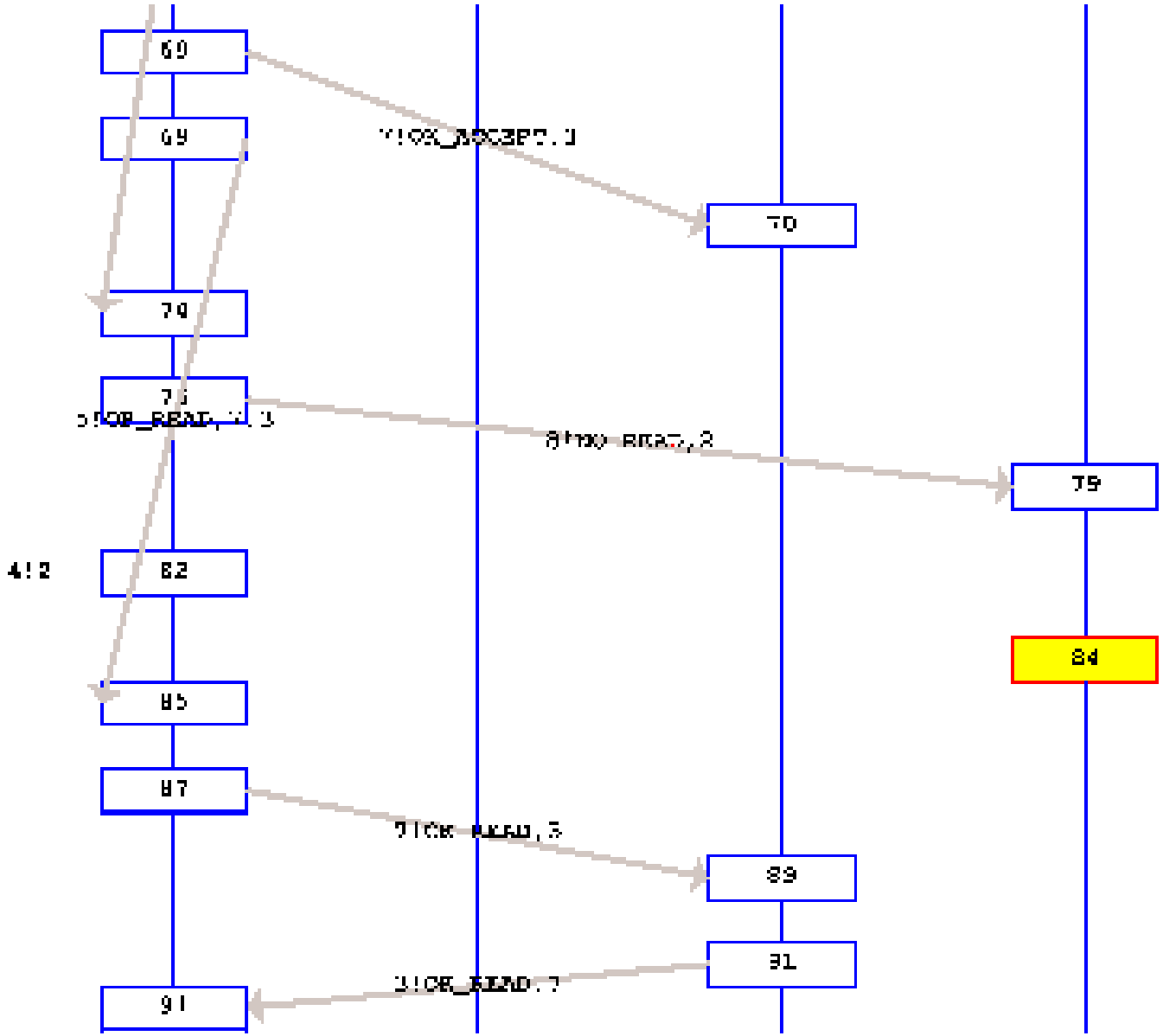
Vérification de formules de logique temporelle

$$\diamond(requete == (reponse + erreur))$$

Si un client se connecte au serveur, il reçoit une réponse ou une erreur.

Certaines propriétés sont vraies pour un modèle mais peuvent être fausses pour un autre !

Simulation avec SPIN



Limitations lors de la vérification

L'abstraction que l'on vérifie doit être bornée !

- explosion du nombre d'états du système

On vérifie un « sous-système » !

Le code des stages n'est pas sûre !

- le développeur doit respecter le modèle de développement
- risque de contenir des erreurs non vérifiées

On peut améliorer la sûreté de certains stages !

Sûreté et portabilité du décodage de requêtes

Décodage des requêtes

Les protocoles de communication sont définis via des RFCs

Le décodage d'une requête peut être formalisée sous forme de grammaire formelle :

- partie lexicale : les « mots »
- partie syntaxique : les « phrases »
- partie sémantique : le « sens »

Intêrets des grammaires :

- facilite la portabilité
- augmente la sûreté

Problématique (rappel)

Besoins d'efficacité

- embarquement dans un serveur !

Besoins de portabilité

- indépendant du modèle de concurrence

Besoins de sûreté

- respect du protocole

Génération automatique d'analyseur syntaxique embarqué

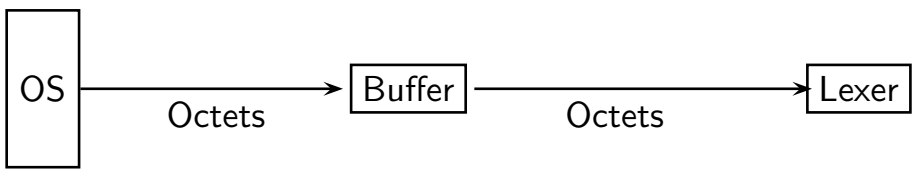
Analyseur embarqué

Les besoins :

- éviter les fuites mémoires
- minimiser l'empreinte mémoire
- s'abstraire vis-à-vis du type des E/S

Les choix :

- utilisation de viviers d'objets
- table d'analyse partagée
- Méthode « push » d'analyse de texte :



A l'époque, aucun générateur d'analyseur syntaxique ne répondait à ces besoins !

Tatoo : un nouveau générateur d'analyseur syntaxique

http ://tatoo.univ-mlv.fr/

Besoins exprimés dans les spécifications de développement de Tatoo

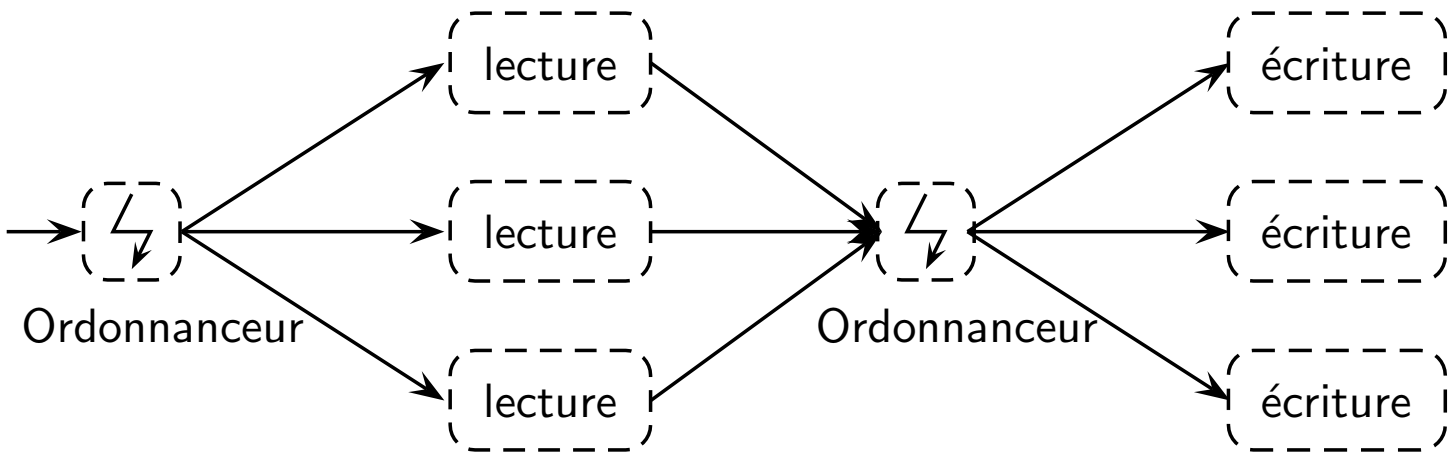
Utilisation de Tatoo :

- spécification du langage à analyser (sous forme EBNF)
- génération des analyseurs lexical et syntaxique
- implantation de la sémantique

Performance de Saburo et Tatoo : Banzaï

Banzaï (1)

Une extension du modèle SEDA :



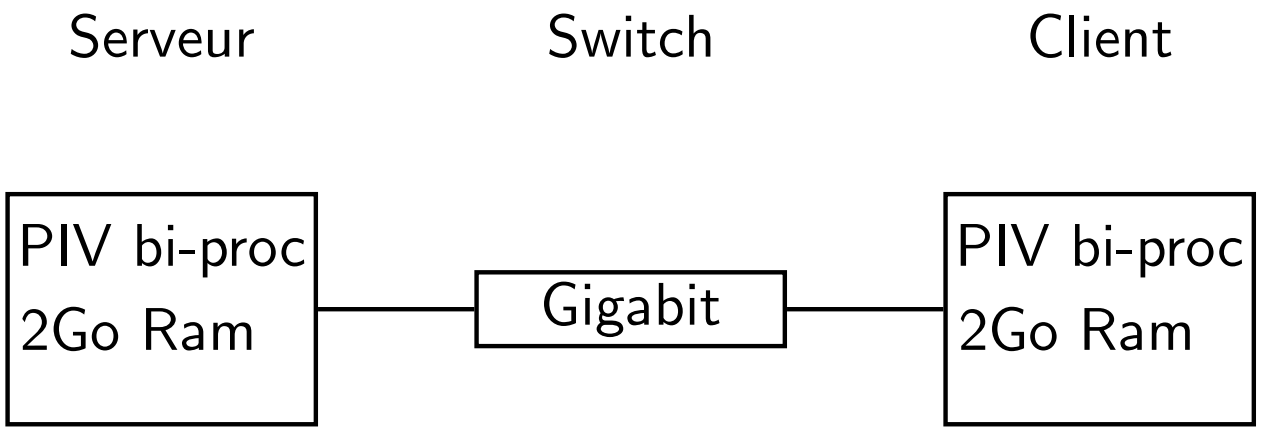
Banzaï (2)

Un principe simple :

Favorisé l'itératif !

1. Essayer de faire les lectures/écritures dans le processus courant.
2. Si ce n'est pas possible, délégation à un processus dédié.
3. Si ce n'est toujours pas possible, enregistrement auprès d'un sélecteur.

Environnement matériel des tests



Pages Internet statiques (taille variant de 4Ko à 1Mo)

Environnement logiciel des tests

Serveurs en comparaison :

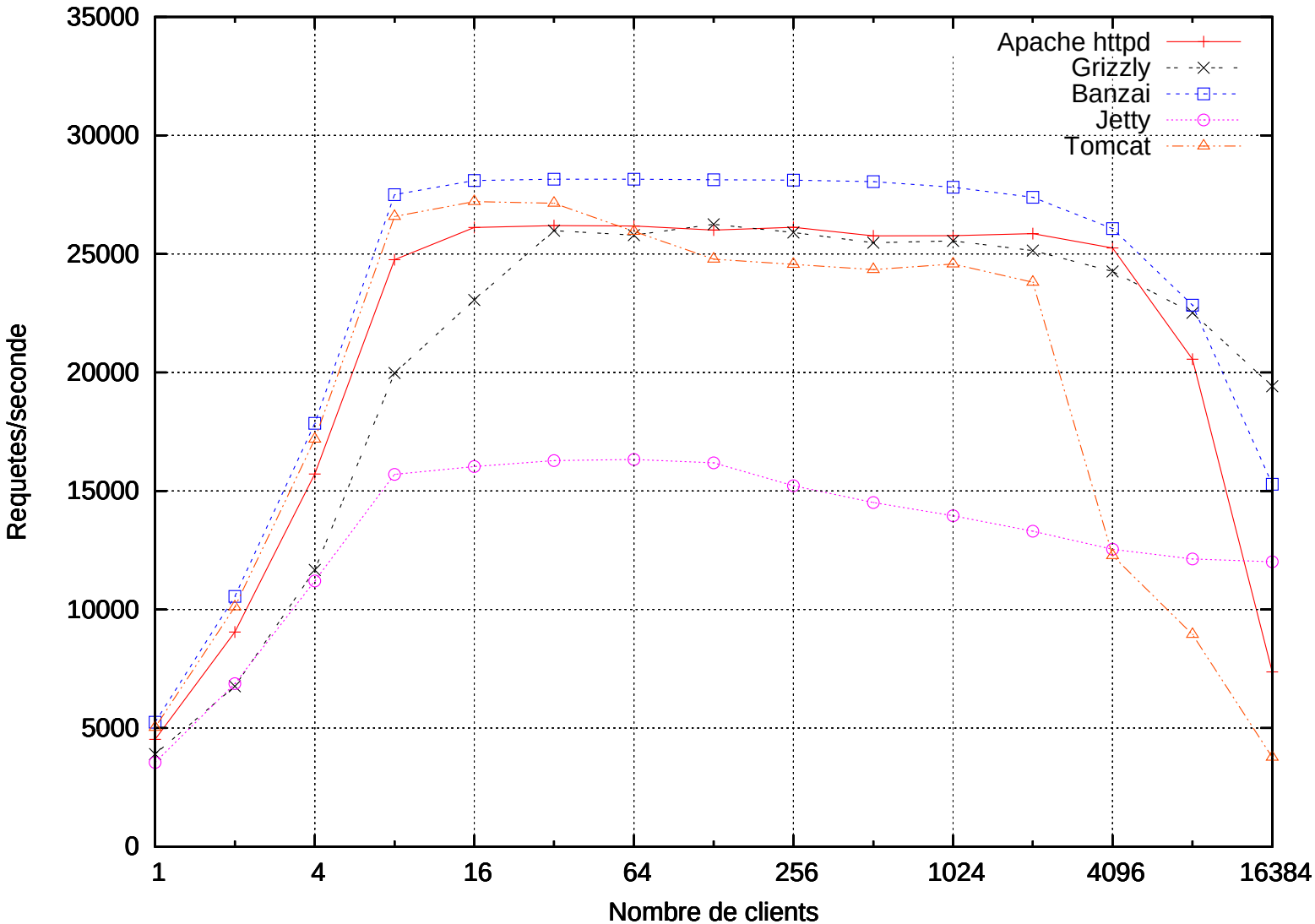
- Apache httpd
- Grizzly
- Tomcat
- Jetty
- Banzaï

Beaucoup de fonctionnalités. . .

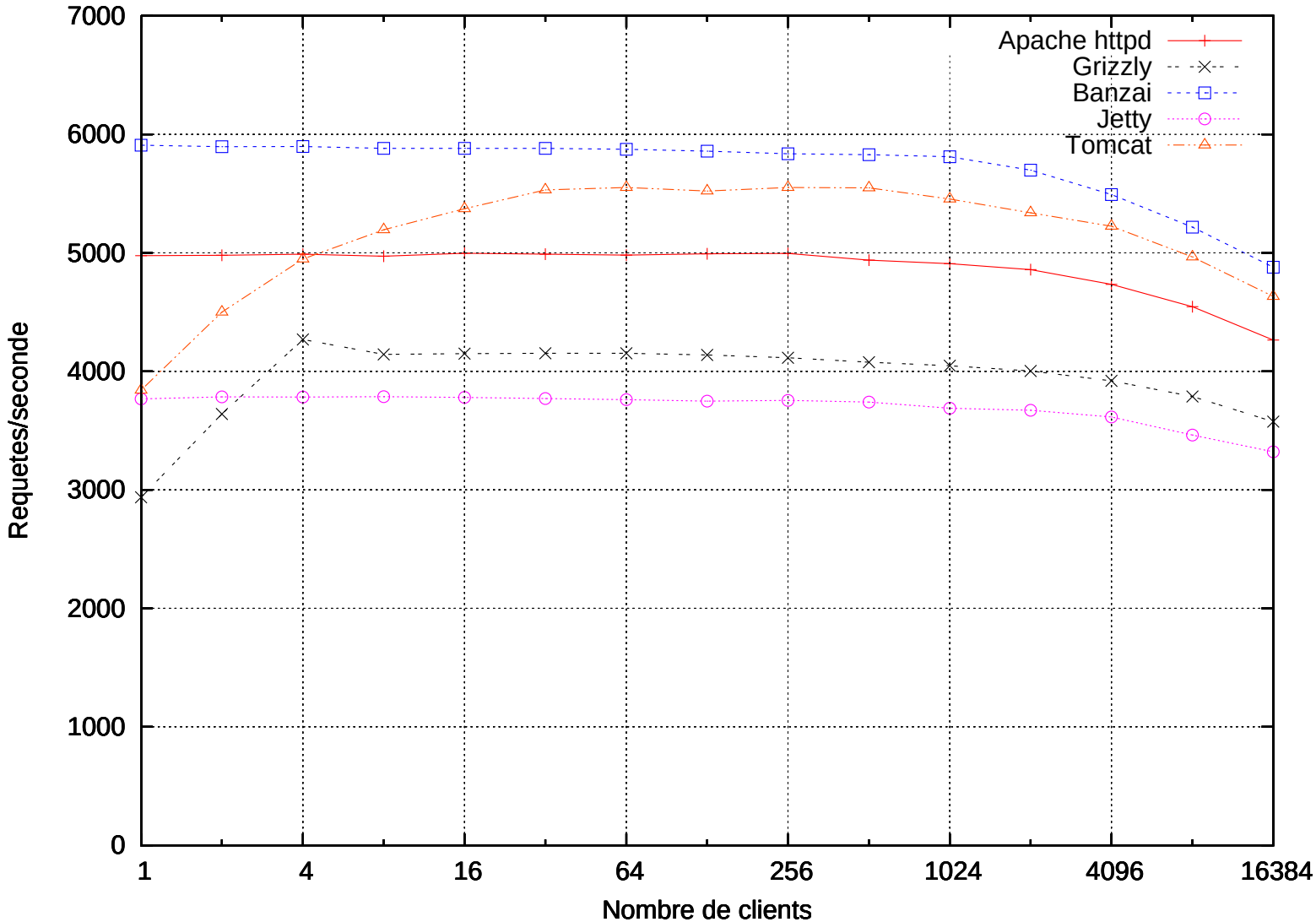
mais le contexte des tests est homogène

Les pages sont situées dans le cache du serveur !

Variation du nombre de requêtes par seconde en fonction du nombre de clients avec ApacheBench (HTTP/1.0)



Variation du nombre de requêtes par seconde en fonction du nombre de clients avec Httperf (HTTP/1.1)



Analyses des résultats obtenus

Banzaï à des performances similaires aux autres serveurs !

Mais Banzaï est généré automatiquement...

et les autres sont écrits à la main.

Le code généré automatiquement peut être aussi performant que le code manuel !

Conclusion : Une fabrique de serveurs Internet (Saburo)

La génération automatique de la concurrence et des E/S

- *A framework for development of concurrency and I/O in servers*, EuroSys 2006.
- *Saburo : A Tool for I/O and Concurrency Management in Servers*, JPDC 2006.
- *Multi-SPED : A new Internet server architecture*, EuroSys 2007.

La vérification automatique des serveurs produits

- *A Java Method for the Design and the Automatic Checking of Server Architectures*, PPPJ 2007.

La génération automatique du décodage des requêtes

- *Banzaï : A Java Framework for the Implementation of High-Performance Servers*, Soumis à SAC 2009.

Perspectives

Compléter l'implantation

Générer automatiquement l'encodage des réponses

Transformer pour d'autres outils de vérification

Transformer les générateurs en compilateurs

Adapter automatiquement le modèle de concurrence