RFC - Persistance

Équipe de développement de Metacosm v0.2, 28 novembre 2000

Le but de ce document est de définir comment la persistance du monde est obtenue dans le projet Metacosm.

Contents

1 Licence

2 Metacosm/JAMBON

Pour plus d'informations sur Metacosm/JAMBON, consultez le site *Metacosm* `<http://metacosm.org>`.

3 Historique

- 28/11/2000 Interface pour la persistence (0.2)
- 06/13/2000 Première version de ce document (0.1)
- 05/04/2000 Premier brouillon
- Premières discussions sur la persistance˜: été 1999

4 Type de stockage

Nous passons ici en revue les différentes méthodes de stockage.

4.1 Format binaire

Par format binaire, nous entendons ici le format binaire du JDK utilisé pour la sérialisation des objets. Il n'y a aucun intérêt à en utiliser un autre, nous préférons utiliser ce qui existe déjà et qui marche (à l'exception du bogue sur la sérialisation des chaînes de caractères de plus de 65535 caractères dans le JDK 1.2).

La sérialisation permet de sauvegarder un objet dans un flux ou de charger un objet à partir d'un flux. Seuls les attributs sont concernés. Le format décrit quels sont les champs, quel est leur type et quelle est leur valeur. Par défaut, la sérialisation opère en profondeur récursivement, donc sur les attributs objets. En sauvegardant puis en restaurant, un objet identique au précédent est obtenu (seuls les emplacements en mémoire varient). Il est néanmoins possible de définir ses propres méthodes de chargement/sauvegarde. Ex.˜: limitation de la profondeur de descente, contrôle des données chargées

Les champs 'static' ou 'transient' ne sont pas sauvegardés par la sérialisation.

Avantages˜:

- déjà écrit et testé
- indépendant de la plateforme ou du système de fichiers
- standard
- transfert des données via le réseau extrêment facile
- ne nécessite rien de supplémentaire
- format concis

Inconvénients˜:

- binaire difficile à lire/éditer à la main
- changement de version du code à gérer manuellement

4.2 Format texte

Par format texte, nous entendons ici format XML. Il n'y a aucun intérêt à utiliser à l'heure actuelle un format ASCII simple. De plus en plus d'outils permettent de manipuler de l'XML et les avantages de l'XML par rapport à l'ASCII simple sont nombreux.

Avantages˜:

- lisibilité, facilité d'édition
- peu sensible aux changements du code
- indépendant de la plateforme ou du système de fichiers
- outils existants pour lire/écrire l'XML
- validation

Inconvénients˜:

- entièrement à définir et à mettre en place
- format verbeux, donc long à manipuler

4.3 Base de données (BDD)

Il faut séparer les deux types de SGBD que nous sommes susceptibles d'utiliser˜: les SGDB relationnels (SGBDR) et les SGBD objets (SGBDO).

Les SGBDR ne sont pas adaptés pour sauvegarder directement des objets. Il faut recourir à des identifiants pour lire/écrire les objets. Le schéma de la base n'est pas évident à déduire de la hiérarchie des objets.

Les SGBDO s'adaptent plus facilement à la POO. La persistence des objets est beaucoup plus facile à gérer.

Avantages˜:

- requêtes sur de grosses quantités de données
- outils existants
- cohérence assurée à l'intérieur de la BDD
- adapté aux très grandes quantités de données
- scalable

Inconvénients˜:

- peu portable
- lenteur/lourdeur
- nécessite une BDD
- administration de la BDD
- indépendance limitée
- cohérence mémoire/BDD difficile à gérer, à écrire
- transfert via le réseau pénible (nécessite une forme intermédiaire)

4.4 Conclusion

Le nombre de sauvegardes est très supérieur au nombre de chargements. Typiquement il y a un chargement au démarrage et des sauvegardes régulières jusqu'à l'arrêt ou au plantage.

L'utilisation d'une BDD est un gros désavantage pour une implantation chez les joueurs. Autant lors de l'utilisation d'un serveur sur Internet, l'utilisateur n'a rien à faire, autant s'il souhaite avoir son propre serveur sur sa machine cela l'oblige à installer et à administrer une BDD, ce qui n'est pas à la portée de tout le monde. Néanmoins la BDD est le seul système 'scalable'. Dans le cas d'une grande quantité de données, seule la BDD (qui est fait pour ça...) est capable de gérer le volume et d'offrir des performances acceptables. Le fait de cumuler le stockage tout en permettant l'accès est clairement un avantage. La séparation des deux fonctions coûte trop cher pour les gros volumes.

Le format XML n'est pas adapté à des sauvegardes fréquentes et rapides en raison de sa verbosité. Par contre il est pratique pour le stockage ou le transfert.

Le format binaire est adapté à des sauvegardes pour de petites quantités de données. En particulier, il convient à nos premiers essais et au prototypage.

5 Je ne lis pas le binaire dans le texte

Pour˜:

- pallier au manque de lisibilité du binaire,
- permettre un interfaçage facile avec l'éditeur de monde et
- permettre la récupération des données des autres muds (manuelles ou automatisées)

nous disposerons de programmes de conversion du binaire sérialisé vers le XML et réciproquement, baptisé Bin2XML et XML2Bin.

Le principe est le suivant˜:

1. Bin2XML produit à partir d'un binaire sérialisé (la sauvegarde du jeu) à la fois la DTD et le fichier XML, le tout sans nécessiter la présence du source des classes et de façon indépendante du serveur.
2. Édition de la DTD et de l'XML (attention à la compatibilité avec le code)
3. XML2Bin produit à partir de la DTD et du fichier XML un binaire sérialisé compatible avec le serveur. Ce binaire peut être utilisé pour le chargement.

6 Évaluation et implantation des média de sauvegarde

Les mondes étant indépendants, il est logique de les séparer lors de la sauvegarde. Il y aura donc au moins un fichier par monde.

Pour le moment, la durée de la sauvegarde et la taille des fichiers sont des inconnues. Il est possible qu'il faille revoir la stratégie présentée ici pour l'adapter à l'expansion en taille des mondes par exemple.

Il ne doit pas y avoir d'accès aux données en cours de sauvegarde, pour garantir la validité de la sauvegarde. Cela implique qu'une partie du jeu est bloquée en attente et a donc des conséquences sur la durée des sauvegardes.

Pour optimiser les sauvegardes, nous avons plusieurs voies˜:

- compter sur Sun pour optimiser la sérialisation (comme lors du passage JDK˜1.2 vers JDK˜1.3)
- utiliser correctement les mots-clés transient et static
- le multi-threading, avec les problèmes que cela peut poser
- modifier les fonctions de sérialisation

La dernière idée peut utiliser un système de flags (indicateurs) pour ne pas re-sauvegarder des données non modifiées. Avantages˜:

- sauvegarde plus rapide
- sauvegarde plus petite

Inconvénients˜:

- nécessité de garder un certain nombre de sauvegardes antérieures pour disposer de toutes les données
- nécessité d'alterner X sauvegardes partielles et une sauvegarde totale
- plus complexe à gérer
- ne pas alourdir la sauvegarde en croyant l'alléger (multiplication des tests)

Seules les premières expérimentations pourront nous éclaircir les idées sur les points précédents˜: elles dépendent fortement des temps de sauvegarde et de la fréquence des sauvegardes (donc de la stabilité du serveur).

7 Mécanisme de reprise sur incident

En principe le serveur est parfait du premier coup, fonctionne ad vitam eternam et la sauvegarde est juste une précaution... Mais dans le cas très improbable où un problème surgirait, nous serons prêts :)

Pour avoir une sécurité de haut niveau, nous pouvons sauvegarder régulièrement les données du jeu (partiellement ou totalement) et sauvegarder en permanence dans un journal des informations susceptibles de permettre la re-simulation en cas d'incident.

Ces informations sont typiquement˜:

- l'état du générateur aléatoire
- les commandes de chaque joueur (a priori seulement les commandes valides, les bogues du parseur sont censés ne pas exister :)
- les Actions entreprises (au moins pour les non-joueurs)
- les changements de tour
- ...

Ces informations sont stockées séquentiellement, avec leur date d'arrivée.

Cela implique que nous devons disposer d'un mécanisme de re-simulation qui puisse recréer de fausses connexions et faire transister par celles-ci les copies des commandes des joueurs, et court-circuiter l'intelligence artificielle des bots en les obligeant à effectuer des Actions pré-déterminées.

Ce système permet de prévenir la majorité des problèmes, ce qui devrait être largement suffisant pour nos besoins. Néanmoins gardons à l'esprit qu'il ne s'agit pas d'une forteresse imprenable...

8 Interfacage avec l'éditeur

Il faut différencier l'éditeur en ligne et l'éditeur hors ligne XML, même s'ils peuvent être en grande partie composés des mêmes éléments.

L'éditeur en ligne agit directement sur les objets (au sens POO), alors que l'éditeur hors ligne n'agit que sur une copie statique des données.

L'éditeur en ligne pose donc plus de problèmes˜:

1. faut-il stopper le jeu pour utiliser l'éditeur˜?
2. comment gérer les modifications du monde (rupture de la cohérence)˜?
3. comment éviter les modifications fatales (données incorrectes)˜?

Le dernier point est censé être géré par l'éditeur qui doit veiller à ne pas autoriser les modifications dangereuses (ou alors avertir du danger, demander un mot de passe ultra haute sécurité, une carte d'accès, une identification rétinienne, un délai de réflexion de 5 min et la saisie en 8 exemplaires de "oui je sais que je vais faire quelque chose de dangereux" (cf Debian)).

Pour l'édition de nouvelles 'zones', il est possible de travailler sur un coin isolé du jeu (un autre monde pour chaque développeur par exemple).

La consultation des données pendant le jeu doit être possible. Cela sera notamment utile pour le débogage. L'édition des données est bien sûr plus problématique et pourrait être envisagée dans des phases inter-tours de jeu (sous réserve que les modifications ne seront pas fatales au tour suivant...). En cas de modification sur le code (rechargement nécessaire), il est nécessaire de gérer les dépendances et de retirer toutes les entités affectés du jeu. Si des objets critiques sont concernés (gestion du temps par exemple), exiger l'arret du serveur.

9 Implantation en Java

Chaque objet (au sens Java) qui doit être sauvegardé (tout ou partiellement) doit 'implémenter' l'interface suivante (Persistent)˜:

- public void load(java.io.InputStream is) throws java.io.IOException;
- public void save(java.io.OutputStream os) throws java.io.IOException;

La méthode load() est utilisée pour restaurer les données. La méthode save() est utilisée pour sauver les données courantes. Les flux sont utilisés pour obtenir du code générique (ils peuvent signifier des fichiers, la mémoire, une base de données, etc). Une IOException peut être lancée si quelque chose d'inattendu arrive.

Le chef d'orchestre de la sauvegarde et du chargement est un objet de la classe Persistence qui propose l'interface suivante˜:

- public void load(long time) throws java.lang.SecurityException, java.io.IOException
- public void save(long time) throws java.lang.SecurityException, java.io.IOException
- public void save() throws java.lang.SecurityException, java.io.IOException

La méthode load(long time) permet de recharger les données qui étaient valides à l'instant 'time'. Les méthodes save(long time) et save() permettent respectivement de sauver les données en utilisant comme marqueur temporel l'instant courant ou un instant 'time' spécifié. Les trois méthodes sont susceptibles de rencontrer des erreurs de lecture/écriture ou de sécurité (lecture impossible, écriture impossible, etc).

Le marqueur temporel permet de différencier les sauvegardes˜:

- pour des fichiers, nom_du_fichier.marqueur (ex.˜: random.123, world.123, passwd.123, etc),
- pour une base de données, le marqueur peut servir de clé primaire,
- pour une zone mémoire, le marqueur permet de calculer l'emplacement des données,
- etc.