Concevoir des applications graphiques interactives distribuées avec INDIGO

Renaud Blanch, Michel Beaudouin-Lafon, Stéphane Conversy, Yannick Jestin, Thomas Baudel, Yun Peng Zhao

Concevoir des applications graphiques interactives distribuées avec INDIGO

Designing Interactive Distributed Graphical Applications with INDIGO

Renaud BLANCH (1, 2), Michel BEAUDOUIN-LAFON (1), Stéphane CONVERSY (3), Yannick JESTIN (3), Thomas BAUDEL (4), Yun Peng ZHAO (4)

(1) LRI (Université Paris-Sud et CNRS) & INRIA Futurs, Orsay
mbl@lri.fr

(2) LIG, Université Joseph Fourier, Grenoble
blanch@imag.fr

(3) ENAC & DSNA/SDER, Toulouse
conversy@enac.fr, jestin@cena.fr

(4) ILOG, Paris
baudel@ilog.fr, zhao@ilog.fr

Résumé. INDIGO est une architecture logicielle destinée au développement d'applications graphiques interactives distribuées. L'architecture est constituée de composants de deux types : des serveurs d'objets applicatifs et des serveurs d'interaction et de rendu graphique. Le rendu graphique et l'interaction peuvent ainsi être optimisés en fonction des périphériques disponibles et du contexte tandis que les objets applicatifs peuvent être accédés par plusieurs clients, permettant ainsi le partage et la collaboration. L'approche a été validée sur des exemples illustrant des techniques variées d'interaction et de présentation.
Mots-clés. Architecture logicielle, boîte à outils, interaction avancée, système interactif réparti.

Abstract. INDIGO is a software architecture designed for the development of interactive applications. The architecture is composed of (i) object servers that manage the application data and (ii) interaction and rendering servers that manage display and interaction. Such separation of the core application logic from the interaction makes it possible to optimize graphical rendering and interaction according to the current setup and context. Application objects can be shared among multiple clients, therefore supporting sharing and collaboration. This approach was validated through examples that illustrate various types of presentation and interaction devices.
Keywords. Software architecture, toolkits, advanced interaction, distributed interactive system.

1 Introduction

Depuis une vingtaine d'années, les interfaces graphiques se sont développées de façon considérable et touchent aujourd'hui un large public. Elles sont cependant restées confinées, pour l'essentiel, à une interaction de type "desktop", où un utilisateur est assis à son bureau face à un ordinateur individuel. Cependant, de nombreuses évolutions remettent ce modèle en cause et montrent les limites des outils et modèles actuels pour le développement d'applications interactives graphiques.

Tout d'abord, de nouvelles techniques d'interaction, parfois appelées post-WIMP, ont été développées afin d'améliorer la performance et l'utilisabilité des applications. Cependant, ces techniques ne sont que trop rarement utilisées dans des produits commerciaux. La raison principale en est que les boîtes à outils et constructeurs d'interface actuels sont pour la plupart fondés sur la notion de *widget*. Le problème de cette approche est que les *widgets* sont très stéréotypés et limitent l'interaction aux formes les plus simples : menus, palettes, boîtes de dialogues, etc. Des techniques aussi courantes que le *drag-and-drop* (glisser-déposer) ne sont pas prises en compte et imposent de lourds développements *ad hoc*, pour chaque application. La situation est pire pour des techniques avancées telles que l'interaction bi-manuelle.

Par ailleurs, les plates-formes actuelles offrent des capacités d'interaction très diverses en termes de dispositifs d'entrée et d'affichage : de l'ordinateur portable au PDA et au téléphone portable, de l'ordinateur de bureau à la table interactive et au mur d'images ou au système immersif. Les différences de taille d'écran, l'usage de l'interaction au stylo, la prise en compte de modalités haptique ou sonore, les exigences nouvelles de l'interaction en situation de mobilité imposent que différentes versions d'une même application soient développées pour exploiter au mieux ces dispositifs et prendre en compte la diversité des usages.

Enfin, cette évolution fait que les usages sont de moins en moins individuels et la nécessité de partager des documents et de collaborer à distance, y compris avec soi-même lorsque, par exemple, on veut accéder un même document de son bureau, d'un chantier et de chez soi, est de plus en plus impérieuse. Ainsi, il est souhaitable que les aspects collecticiels fassent partie intégrante des applications interactives, au même titre que le copier-coller.

Face à ces trois défis que sont l'interaction post-WIMP, la diversité des plates-formes et l'intégration du collecticiel, nous avons développé et validé une architecture logicielle appelée INDIGO *(Interactive Distributed Graphical Object)* (Blanch *et al.*, 2005) fondée sur les principes suivants :

- Une architecture répartie formée de serveurs spécialisés dans la gestion d'objets de l'application d'une part, dans l'interaction et le rendu graphique d'autre part ;
- La transformation des objets de l'application en un graphe de scène dont le rendu est contrôlé en fonction de la plate-forme ;
- L'interprétation des actions de l'utilisateur en commandes de haut niveau sur les objets du domaine ;
- Le contrôle de la cohérence permettant à plusieurs serveurs d'interaction et de rendu de représenter les mêmes objets.

Cet article présente l'architecture INDIGO, ses composants et son fonctionnement, puis illustre son utilisation avec quelques exemples qui nous ont permis de la valider, enfin compare INDIGO à l'état de l'art. Nous concluons avec quelques directions pour la suite de ces travaux.

2 Architecture

Les travaux en architecture logicielle des interfaces ont depuis longtemps insisté sur la nécessité de séparer l'interface de l'application de son noyau fonctionnel (gestion des objets du domaine). L'architecture répartie d'INDIGO[1] (Figure 1) distingue deux types de serveurs : les *serveurs d'objets* (SERVO) et les *serveurs d'interaction et de rendu* (SERVIR). Le SERVO réalise le noyau fonctionnel de l'application, en utilisant un vocabulaire propre aux objets manipulés et à leur domaine d'utilisation. Il expose ces données à un ou plusieurs SERVIR. Ceux-ci, en fonction des capacités de leur plate-forme d'exécution, mettent en œuvre les représentations et les techniques d'interactions qui permettent aux utilisateurs d'interagir avec les objets du SERVO. Ce dernier maintient la cohérence de ses données (il peut refuser des modifications demandées par un SERVIR), et notifie l'ensemble des SERVIR qui utilisent ses données des modifications de leur état.

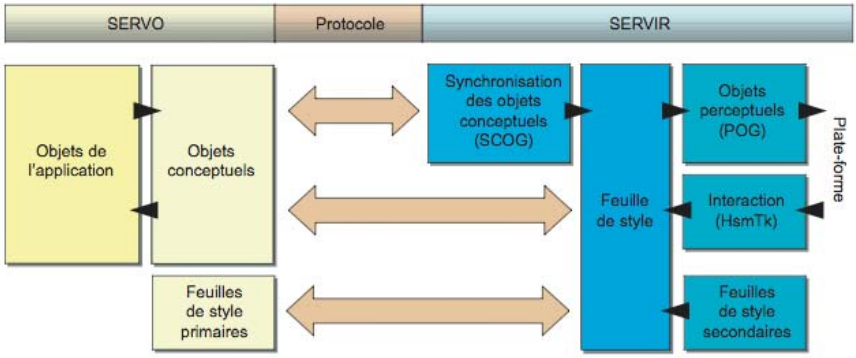

Figure 1. *Architecture INDIGO.*
Les flèches épaisses représentent les échanges gérés par le protocole SERVO-SERVIR ; Les flèche réduites à un triangle représentent les flux d'information

Dans le modèle architectural de référence ARCH (Arch, 1992), l'architecture INDIGO consiste à localiser les composants du domaine (une des branches de l'arche) dans le SERVO, et les composants de présentation et d'interaction (l'autre branche) dans le SERVIR. Le contrôleur de dialogue (la clé de voûte de l'arche), quant à lui, est réparti entre SERVO et SERVIR : le contrôle du dialogue de bas niveau est localisé dans le SERVIR tandis que le contrôle de dialogue de haut niveau est localisé dans le SERVO. Le contrôle de bas niveau est celui qui est mis en œuvre au niveau des interactions élémentaires, comme par exemple une commande d'un menu qui fait apparaître une boîte de dialogue ou la navigation dans un ensemble d'objets par défilement ou de façon hiérarchique. Le contrôle de haut niveau concerne plutôt la logique globale du dialogue, comme la disponibilité de certaines commandes en fonction du mode. La distinction entre ces deux niveaux de dialogue n'est pas stricte, ce qui laisse une certaine liberté au concepteur pour décider de la localisation exacte du contrôleur de dialogue.

Ce choix de répartition architecturale repose sur la constatation que de nombreux processus applicatifs présentent une façade abstraite, en grande partie indépendante de l'interface utilisateur concrète. Cette façade peut s'assimiler au *modèle conceptuel* de l'application, c'est-à-dire la collection des objets et opérations

[1] Malgré l'homonymie, le projet INDIGO n'a rien à voir avec la technologie du même nom annoncée par Microsoft en février 2005.

ayant un sens pour l'utilisateur. Dans INDIGO, le modèle conceptuel est un graphe d'objets comprenant des attributs et des opérations possibles sur ces objets, dénommé le *graphe d'objets conceptuels* (COG pour *Conceptual Object Graph*) et représenté par un arbre XML. Chaque SERVO gère un graphe d'objets conceptuels et maintient sa cohérence en réponse aux demandes de mise à jour qu'il reçoit. Pour être manipulable par l'utilisateur, ce modèle conceptuel doit être transformé en une forme perceptible appelée *graphe d'objets perceptuels* (POG pour *Perceptual Object Graph*). Pour une représentation graphique, ce graphe utilise par exemple SVG *(Scalable Vector Graphics)*, le standard de graphique vectoriel du Web (W3C, 2003). L'architecture n'est cependant pas restreinte aux représentations graphiques, et peut intégrer d'autres graphes encodant des percepts, comme des dispositifs tactiles ou sonores.

La transformation d'un COG en POG est appelée *concrétisation*. Elle est contrôlée par une ou plusieurs *feuilles de style*. Les feuilles de style dites primaires sont stockées dans le SERVO, qui doit choisir la feuille la plus adaptée à la plate-forme et au contexte de l'interaction ; elles peuvent être complétées par des feuilles de styles secondaires stockées dans le SERVIR. Ces feuilles sont propres à la plate-forme ou à l'utilisateur et permettant une adaptation de la présentation à leurs besoins propres. La concrétisation a lieu dans le SERVIR : celui-ci gère une copie du COG appelée SCOG (*Synchronized* COG) et applique la transformation localement afin d'obtenir le POG. Cela permet de réduire le trafic entre SERVO et SERVIR en échangeant uniquement des informations de haut niveau, mais aussi de gérer au sein du SERVIR un unique POG contenant les concrétisations des COGs de plusieurs SERVO gérant des objets de natures différentes. Ainsi, un SERVIR peut donner accès à des objets de différents SERVO, de même que les objets d'un même SERVO peuvent être présentés (de façons différentes) par différents SERVIR. Cette dernière situation permet de mettre en œuvre des applications collecticielles, le SERVO gérant le partage d'objets et notifiant chaque SERVIR client lorsqu'un objet est modifié. Si l'on utilise un seul SERVO, l'architecture est centralisée, avec les problèmes qu'on lui connaît. Rien n'empêche cependant le SERVO de se répliquer à chaque nouvelle connexion d'un SERVIR, mais nous n'avons pas encore testé cette approche.

Les sections suivantes décrivent l'approche générale de conception d'une application avec INDIGO, puis l'architecture elle-même et sa mise en œuvre : SERVO, SERVIR, concrétisation, synthèse générale du fonctionnement.

3 Conception d'applications avec INDIGO

Concevoir une application avec INDIGO consiste à définir le modèle conceptuel des objets de cette application et ses feuilles de style primaires pour les plates-formes (et donc les SERVIR) visés. Le reste de cette section détaille ces deux étapes.

3.1 Définir le modèle conceptuel

Le modèle conceptuel d'une application INDIGO est un ensemble de classes d'objets et de méthodes qui définissent les objets et les commandes qui seront offertes à l'utilisateur à travers l'interface. Par exemple, pour l'interface d'un système de fichiers, les objets sont les fichiers et les répertoires, les méthodes sont la création, le changement de nom, le changement de répertoire, la copie et la destruction. Les objets conceptuels pourront être désignés par l'utilisateur, par exemple par manipulation directe dans une interface graphique, ou par reconnaissance vocale si l'on utilise une interface de ce type. Les méthodes du

modèle conceptuel seront déclenchées par des commandes de l'interface, telles que la sélection dans des menus ou des palettes ou la reconnaissance de gestes. Aussi, même si le modèle conceptuel se veut indépendant du type d'interface qui sera mis en œuvre, il ne peut être correctement défini que si l'on a une idée du ou des interfaces concrètes qui l'utiliseront. L'objectif est que les objets du modèle conceptuel coïncident avec les objets du modèle mental de l'utilisateur. L'identification de ces objets nécessite donc des méthodes de conception centrées sur l'utilisateur, classiques en IHM.

Une situation fréquente lors de la conception d'une application interactive est celle où un noyau fonctionnel existe déjà, par exemple le système de fichiers d'un système d'exploitation ou une base de données. Dans ce cas, le SERVO se comporte comme un adaptateur du domaine dans le modèle Arch, en fournissant une façade aux objets du noyau fonctionnel préexistant. Les objets de l'application et les objets conceptuels de la Figure 1 peuvent alors ne pas être localisés dans le même processus, voire s'exécuter sur des machines différentes. Cette situation possède un avantage, à savoir la possibilité de créer plusieurs modèles conceptuels, chacun implémenté par un SERVO différent, pour interagir avec les mêmes données. Cela peut s'avérer utile si l'on envisage des interfaces tellement différentes qu'elles ne peuvent utiliser le même modèle conceptuel.

3.2 Définir des feuilles de style

Les feuilles de styles contiennent deux types d'information : la description des objets perceptuels correspondant à chaque objet conceptuel, et la description des interactions disponibles sur ces objets. Le formalisme des objets perceptuels est imposé par le SERVIR utilisé. Par exemple, le SERVIR décrit à la section 5 utilise le standard de description d'objets graphiques SVG (W3C, 2003) tandis que celui décrit à la section 8.1 utilise HTML.

La traduction des objets conceptuels vers les objets perceptuels est spécifiée par des règles de transformation dont le fonctionnement est détaillé dans la section 6 sur la concrétisation. Ces règles sont d'autant plus faciles à définir que la correspondance entre modèle conceptuel et modèle perceptuel est simple. En fait notre moteur de concrétisation impose des limitations sur ces transformations, et c'est la raison principale pour laquelle des interfaces trop différentes peuvent nécessiter des modèles conceptuels (et donc des SERVO) différents, même si les données sous-jacentes de l'application sont les mêmes. De telles limitations sur la concrétisation nous semblent, en fait, saines. La complexité des feuilles de styles traduit, d'une certaine façon, la distance entre le modèle conceptuel et le modèle perceptuel. Or on peut difficilement imaginer une interface utilisable qui nécessiterait une trop grande distance. Aussi, de façon générale, on cherchera à ce qu'un objet du graphe conceptuel se transforme en un ensemble d'objets "proches" dans la structure de données cible (le POG).

Dans l'exemple du système de fichiers, un fichier est représenté par un icone dont l'image dépend du type de fichier. Cela nécessite que le type de fichier soit un attribut de celui-ci. La feuille de style peut alors spécifier que l'image de l'icone est le nom du type du fichier. Pour un répertoire, on a deux représentations différentes selon que celui-ci est ouvert ou fermé : une fenêtre s'il est ouvert, avec l'ensemble des fichiers qu'il contient, et un icone s'il est fermé. Si l'état du répertoire est l'un de ses attributs, comme le type pour les fichiers, on utilise le même mécanisme pour paramétrer la transformation par l'état de l'objet. Dans ce cas cependant, cela signifierait que si deux utilisateurs partageaient le même répertoire du SERVO, ils auraient le même état, ouvert ou fermé, pour les deux utilisateurs. Comme cela n'est

a priori pas souhaitable, il faut spécifier deux transformations différentes, et c'est le SERVIR qui appliquera celle correspondant à l'état du répertoire dans l'interface.

Quant aux interactions disponibles sur chaque objet, elles sont également spécifiées dans les règles de transformation de la feuille de style, sous forme d'annotations de la structure cible. Chaque interaction du SERVIR définit un vocabulaire d'actions. Si un objet perceptuel est annoté par l'une de ces actions, l'interaction correspondante sera disponible sur cet objet. Par exemple, une représentation iconique des fichiers spécifiera que l'action de déplacement est disponible si l'on fait un cliquer-glisser de l'image de l'icone, alors que si l'on clique sur le nom de l'icone, c'est l'opération de renommage qui sera activée. Les détails de ce mécanisme sont donnés aux sections 5.2 et 6.

En résumé, le modèle conceptuel comme les feuilles de style sont des formalismes déclaratifs pour spécifier les objets conceptuels, les objets perceptuels correspondant aux objets conceptuels, et les interactions conduisant à l'appel des méthodes du modèle conceptuel.

4 Serveur d'objets

Le serveur d'objets conceptuels (SERVO) présente à un ou plusieurs serveurs d'interaction (SERVIR) l'ensemble des objets de l'application à travers un système transactionnel permettant de gérer les accès multiples. Il repose en général sur un modèle de données persistant, qu'il transforme éventuellement de façon à présenter aux SERVIR des objets pertinents pour l'interaction.

4.1 Description du modèle conceptuel

Le créateur d'une application INDIGO décrit son modèle conceptuel dans le langage de son choix (Java, C#, ou C++), ou au moyen d'un schéma XML. Afin d'assurer l'indépendance vis-à-vis du langage utilisé, plusieurs solutions ont été envisagées, de l'approche maximaliste utilisant un protocole méta-objet pour décrire le modèle du langage, à une approche réductrice reposant sur la sémantique d'un langage unique. La solution maximaliste s'est vite révélée impraticable : elle aurait requis la conception d'un méta-langage capable de décrire toutes les sémantiques des langages cibles. Nous avons opté pour un langage reposant sur l'héritage simple correspondant en pratique au protocole SOAP *(Simple Object Access Protocol)* (Box *et al.*, 2000). Cette approche nous permet d'utiliser les techniques d'introspection des langages actuels (tels que les JavaBeans pour les composants Java), ou une phase d'analyse de code, pour construire un modèle permettant l'échange et la synchronisation de structures de données entre processus.

L'utilisateur décrit son modèle conceptuel avec le langage de programmation, en respectant des conventions d'écriture fournissant un modèle de données exploitable. Par exemple, pour Java, le guide de style des *cleanbeans* (Kaplan, 1999) spécifie des propriétés des composants Java comme l'absence d'état caché ou la commutativité des accesseurs qui sont importantes pour l'observabilité du système. La Figure 2 montre la définition en C++ d'un système de fichiers rudimentaire modélisé par des répertoires pouvant contenir des fichiers, avec un nom pour unique attribut. Cette définition est utilisée pour le langage C++ par la bibliothèque gSOAP (van Engelen et Gallivan, 2002) pour générer toute la mécanique de communication de l'application en utilisant SOAP.

Les classes peuvent correspondre directement à des classes de l'application, ou bien jouer le rôle de simples façades destinées à la communication avec l'interface, derrière lesquelles se cachent les vrais objets de l'application. Dans tous les cas, le créateur de l'application n'a pas à se préoccuper du transport et de la

synchronisation du modèle conceptuel. Il déclare et implante la sémantique de son application dans le SERVO, et lui adjoint une correspondance avec les feuilles de style primaires décrivant le processus de concrétisation pour les SERVIR visés.

```
//////////////////////////////////////////////////////////////////////
// File    : fsCog.h
// Content : file system COG
//////////////////////////////////////////////////////////////////////


/* imports ********************************************************/

#import "indigo.h"


/* classes ********************************************************/

class cog__FileName : public cog__Element {
public :
    xsd__string value;

    cog__FileName(const std::string &name);            // constructor

    cog__FileName &operator=(const std::string &name); // setter
    operator const std::string &() const;              // getter
};


class cog__File : public cog__Element {
public :
    cog__FileName fileName; // name attribute

    cog__File(std::string name); // constructor
};


class cog__Folder : public cog__File {
public :
    std::vector< cog__File * > files; // files in the folder

    cog__Folder(std::string name); // constructor
    ~cog__Folder();                // destructor
};
```

Figure 2. *Modèle conceptuel en C++ pour un système de fichier*

4.2 Utilisation de services Web

Pour créer un SERVO qui publie un modèle conceptuel, il suffit d'instancier et de déclarer un objet racine. Au moyen des mécanismes d'introspection, le module de publication explore l'ensemble de la structure de données et présente un service Web permettant d'y accéder. Ce service Web est publié par un serveur HTTP (Internet Society, 1997) et utilise le protocole SOAP (*Simple Object Access Protocol*) (W3C, 2003) qui permet aux SERVIR d'appeler des méthodes des objets du SERVO, L'utilisation de ces standards permet au protocole INDIGO de fonctionner sur le réseau Internet global, par-delà les passerelles coupe-feu. Outre

les appels d'introspection permettant au SERVIR de connaître les classes d'objets conceptuels, leurs propriétés et leurs méthodes, ce service expose trois fonctions :

- ***Get*** retourne la structure de données (le COG) gérée par le SERVO. Si un filtre est passé en paramètre, seule une partie de la structure est retournée, ce qui permet le chargement paresseux (Cf. explication sur ce type de chargement en 4.3) de la structure. Si un numéro d'historique (voir ci-après) est passé en paramètre, alors c'est une différence entre l'état antérieur correspondant à ce numéro et l'état actuel qui est retourné.
- ***Post*** demande l'exécution d'une requête par le SERVO. Cette requête consiste en une combinaison de créations et de suppressions d'objets, de modifications d'attributs et d'appels de méthodes. Le retour de cet appel est immédiat, mais le résultat de l'opération n'est pas connu : c'est une demande de transaction.
- ***Listen*** rapporte au SERVIR la dernière modification effectuée sur le COG. Appelé à la suite d'un *post,* il permet de savoir si la transaction s'est bien déroulée, et donc de mettre à jour le modèle répliqué du COG. Sinon, il permet au SERVO de transmettre à tous les SERVIR les modifications qui se déroulent dans le serveur d'objets, de manière asynchrone.

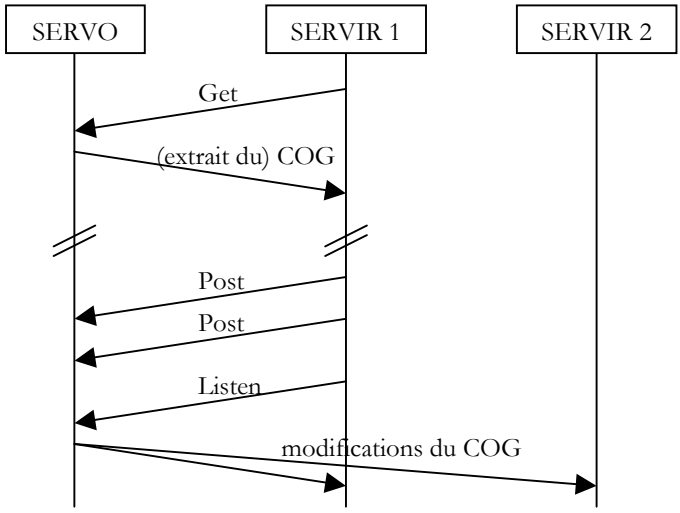

Figure 3. *Communication SERVO-SERVIR*

La fonction *listen* est nécessaire pour mettre en œuvre avec SOAP un mécanisme de notification à l'initiative du SERVO (Figure 3). En effet, seul le client SOAP (ici le SERVIR) peut appeler des fonctions du serveur (ici le SERVO). La notification fonctionne en "*pull-wait-push"* : si aucune modification n'est à signaler, le SERVO ne répond pas et le SERVIR est alors en attente passive sur la connexion, ce qui ne consomme pas de ressource. Dès qu'une modification du COG intervient, le SERVO répond aux SERVIR en attente qui sont ainsi notifiés immédiatement. Au-delà d'un certain délai (1 à 5 secondes) sans modification, le SERVO répond tout de même pour éviter que les connexions ne soient considérées comme perdues par les couches plus basses du protocole réseau.

4.3 Modèle transactionnel de partage des données

Le SERVO doit fournir une présentation fiable et robuste d'un modèle conceptuel. Dans la perspective où ces données peuvent être partagées entre plusieurs utilisateurs ou processus de traitement, il est important que le protocole de partage du modèle incorpore des moyens de notifier l'échec ou le succès des opérations et permette de revenir à un état stable antérieur à une opération.

C'est pourquoi le protocole de partage repose sur un modèle transactionnel (Gray & Reuter, 1993) : un historique de taille réglable conserve toutes les modifications effectuées sur le modèle et permet de revenir à toute version antérieure de la structure de données. Tous les échanges entre un SERVO et ses SERVIR clients sont accompagnés d'un numéro d'ordre qui permet aux SERVIR de se resynchroniser en cas d'échec de transaction ou de perte temporaire du lien physique. Chaque modification d'un ou plusieurs attributs du modèle conceptuel est l'objet d'une transaction. Si une transaction échoue, le SERVIR ne doit pas mettre à jour la structure de données répliquée (SCOG). Le choix de la méthode de restauration en cas d'échec (optimiste ou conservatrice) est laissé au SERVIR.

Afin de limiter l'utilisation de la bande passante et la duplication d'information, le SERVO ne publie pas nécessairement l'intégralité du modèle de données : la méthode *get* accepte un paramètre spécifiant un filtre sur les données, défini sous une forme proche d'une requête XQUERY (W3C, 2006). Cela permet au SERVIR de charger un modèle de façon paresseuse. Par exemple, une arborescence de fichiers n'a pas besoin d'être chargée complètement, mais seulement en fonction de ce que l'utilisateur souhaite afficher. Les parties du COG qui ne sont pas répliquées dans le SCOG sont représentées par la requête nécessaire à leur chargement, selon une approche comparable à ActiveXML (Abiteboul *et al.*, 2004). Ces parties seront chargées à la demande, lors de la concrétisation. Par exemple, lorsque l'utilisateur ouvre l'icone d'un dossier pour voir son contenu, si celui-ci n'est pas déjà chargé dans le SERVIR, le dossier contient la requête permettant de charger son contenu, mais pas celui de ses sous-dossiers. Le SERVIR envoie cette requête et récupère le niveau suivant de l'arborescence des fichiers.

5 Serveur d'interaction et de rendu

Un serveur d'interaction et de rendu (SERVIR) est chargé de fournir une représentation des données de l'application aux utilisateurs, et de leur donner le moyen d'interagir avec ces données. Il est adapté à la plate-forme sur laquelle il s'exécute, et il communique avec un ou plusieurs SERVO qui lui fournissent les objets conceptuels qu'il doit représenter. Ces objets conceptuels (COG) sont transformés dans le SERVIR en objets perceptuels (POG)[2].

Nous avons réalisé un premier SERVIR utilisant HTML comme modèle de rendu, ce qui a eu l'avantage de tester l'architecture aisément mais l'inconvénient de limiter les techniques d'interaction. Dans un second temps, nous avons développé un SERVIR graphique générique plus avancé. Celui-ci repose sur le standard SVG pour le rendu graphique et les machines à états hiérarchiques pour l'interaction. Nous présentons ici ce SERVIR avancé.

[2] Il est bien entendu possible que les objets conceptuels soient identiques aux objets perceptuels : cela peut être le cas dans un éditeur graphique pour lequel le modèle de l'application et celui du rendu coïncident.

5.1 Modèle graphique

L'un des objectifs d'INDIGO est de permettre de tirer parti de modèles de rendu et d'interaction riches : du côté du rendu, il s'agit d'offrir la richesse d'expression à laquelle ont accès les designers graphiques avec des outils tels que Adobe Illustrator ; du côté de l'interaction, il s'agit de tirer parti des nombreuses techniques d'interaction post-WIMP produites par la recherche, comme les *toolglasses* (Bier *et al.*, 1993) qui permettent, par une interaction bimanuelle, de sélectionner en un seul clic une commande dans une palette semi-transparente et l'objet auquel cette commande est appliquée. Le moteur de rendu graphique est un composant critique pour la performance du SERVIR. Il requiert des compromis judicieux entre occupation mémoire et temps de calcul, entre simplicité et efficacité des algorithmes. Nous détaillons ici les aspects les plus saillants de son implémentation qui nous ont permis d'atteindre ces objectifs.

Graphe de scène

Le rendu graphique de ce SERVIR est basé sur *Scalable Vector Graphics* (SVG), un format du W3C permettant de décrire des scènes graphiques structurées en deux dimensions (W3C, 2003). Nous avons choisi SVG d'abord pour ses primitives graphiques de haut niveau, comme les courbes de Bezier, les gradients, la transparence, le *clipping* et le *masking*, ou l'application de filtres. Ces primitives sont nécessaires pour décrire des scènes interactives, comme par exemple le *clipping* d'une fenêtre, une ombre portée simulant une profondeur, ou une interface zoomable grâce aux transformations géométriques s'appliquant sur des objets graphiques vectoriels. Ensuite, le format SVG permet d'organiser un flux de travail avec des graphistes utilisant des outils tels que Adobe Illustrator (Chatty *et al.*, 2004). Enfin, SVG est une norme respectant le format XML, ce qui permet l'utilisation d'outils standards (comme un moteur XSLT) pour transformer le COG en POG (voir section 6 sur la concrétisation).

SVG définit un graphe de scène, c'est-à-dire une structure de données qui permet de décrire une scène graphique en termes d'organisation spatiale, de formes géométriques et d'attributs graphiques. Les problèmes posés par ce type de structure sont de plusieurs ordres : pour les développeurs de systèmes interactifs, il s'agit d'avoir à disposition des services qui facilitent ou rendent possible la description d'interactions. Pour les développeurs de SERVIR, il faut s'assurer que les primitives utilisées sont traitées suffisamment rapidement pour que le système soit réactifs, c'est-à-dire offre un taux de réaffichage de 20 Hz au moins. À cet effet, nous avons développé une librairie de rendu de scène SVG s'appuyant sur la librairie graphique OpenGL (Woo *et al.*, 2003). Les algorithmes de base pour la transformation des primitives de SVG vers OpenGL sont décrits dans Conversy et Fekete (2002). Nous décrivons ici les structures de données spécifiquement développés pour le SERVIR.

Une scène SVG est un graphe orienté sans cycle. Il est composé d'un nœud racine, de nœuds intermédiaires et de feuilles. L'organisation hiérarchique permet de partager des transformations géométriques et des attributs graphiques. Par exemple, un groupe SVG permet de rassembler dans un sous-graphe un ensemble de primitives représentant un icone, et d'appliquer une transformation géométrique à l'ensemble du sous-graphe afin de le positionner à l'écran. De même, il est possible de changer le style de police utilisé par un ensemble de textes en groupant ces textes et en ne changeant l'attribut de style qu'au niveau du groupe. La représentation à l'écran d'une feuille dépend donc de ses parents.

La sémantique de l'affichage d'une scène SVG utilise un parcours en profondeur, en cumulant les transformations de style et de géométrie jusqu'à la traversée d'une feuille qui sera finalement affichée. Comme une scène SVG est un

graphe, une feuille ou un nœud peuvent être référencé par plusieurs nœuds parents et donc peuvent être affichés plusieurs fois, potentiellement avec des attributs de style et de géométrie différents. Par exemple, la partie du graphe qui décrit l'icone représentant un fichier peut être référencé plusieurs fois par une vue d'un répertoire comprenant plusieurs fichiers ayant le même icone.

Une scène SVG peut être affichée en effectuant un parcours en profondeur et en appelant les primitives OpenGL correspondantes : il s'agit de l'algorithme décrit dans Conversy et Fekete (2002). Cependant, cet algorithme ne permet pas d'obtenir des performances optimales dans le cas de scènes interactives : les calculs intermédiaires nécessaires à la transformation de primitives SVG en primitives OpenGL sont coûteux, et il est souvent inutile de les répéter à chaque image. Une solution consiste à conserver le résultat d'un calcul précédent dans un cache, et à le réutiliser pour le calcul de l'image suivante. Cependant, comme le résultat du calcul d'un nœud dépend de la traversée des parents, il n'est pas possible de le cacher au niveau du nœud. Nous avons donc développé une structure plus adaptée, que nous décrivons maintenant.

Le moteur de rendu Sauvage

Notre moteur de rendu SVG, appelé Sauvage, est constitué de quatre composants spécialisés dans un aspect particulier du processus de rendu et d'interaction (Figure 4) :

- le graphe de scène SVG proprement dit, c'est-à-dire une description de haut-niveau de la scène graphique ;
- un **graphe d'affichage**, une structure de données optimisée pour l'affichage de la scène graphique ;
- une structure de données pour l'**élagage** des formes non visibles ; et
- une structure de données pour la gestion de la **taille apparente** des objets.

Comme dans l'application CPN2000 (Beaudouin-Lafon et Lassen, 2000), le graphe d'affichage sert exclusivement à l'affichage et au *picking*, c'est-à-dire au procédé qui permet de déterminer la forme graphique se trouvant sous le curseur. Il est conçu pour être proche de la librairie graphique de bas niveau, dans notre cas OpenGL, afin de bénéficier des accélérations matérielles et d'employer des techniques d'optimisation. Par exemple, le rendu d'un élément SVG `<image>` correspondant à une image stockée dans un fichier peut être partagé par ses références multiples car il ne dépend pas de ses parents. De même, il est possible d'utiliser des listes d'affichage, une fonction d'OpenGL qui permet de mémoriser une suite de primitives lorsque celles-ci sont utilisées plusieurs fois et qui accélère grandement le rendu des primitives concernées.

Le graphe d'élagage permet d'éviter d'afficher des formes graphiques qui sont en dehors de la vue sur la scène. Cette structure est optimisée pour les transformations géométriques typiquement appliquées à une forme graphique. Ainsi, le déplacement par glisser-déposer est rapide car il engendre très peu de mises à jour dans la structure interne. De même le *pan-and-zoom* d'une interface zoomable (Furnas et Bederson, 1995), une transformation qui combine translation et homothétie, n'engendre que des modifications incrémentales qui permettent un calcul rapide des formes à élaguer ou à rendre visibles.

La gestion de la taille apparente permet de choisir une représentation différente selon la taille effective d'un objet sur l'écran. Par exemple, le contour d'un caractère textuel dépend de sa taille à l'écran. Si l'on utilise l'homothétie d'OpenGL pour afficher un caractère à une taille inférieure à celle pour laquelle il a été conçu, sa lisibilité sera dégradée. Afin d'éviter ce problème, il faut calculer la taille

apparente du caractère, qui est la taille spécifiée dans l'objet SVG transformée par l'ensemble des opérations géométriques appliquées à l'objet par ses ancêtres. La structure de données que nous avons développée permet d'optimiser ces calculs. Elle permet également de mettre en œuvre une technique d'optimisation qui consiste à stocker l'image du résultat d'un rendu partiel dans une texture pour la réutiliser.

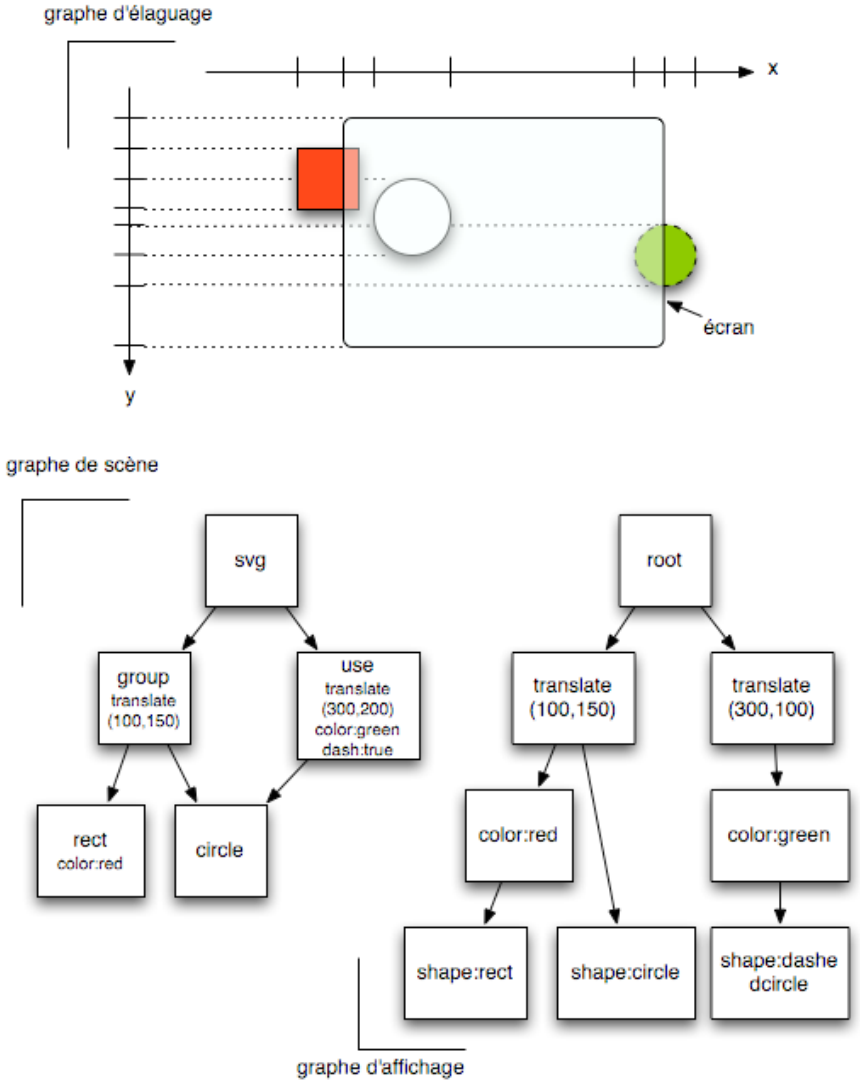

Figure 4. *Les différents graphes de Sauvage*

Grâce à ces structures de données, les performances du moteur de rendu Sauvage sont compatibles avec l'interaction : même sur des scènes complexes, le taux de rafraîchissement observé est presque toujours au-dessus de 20 Hz sur une machine actuelle de puissance moyenne. Grâce à l'utilisation efficace d'OpenGL, le moteur de rendu bénéficie des progrès constants des cartes graphiques, tandis que

nous continuons à développer de nouvelles optimisations pour améliorer encore les performances.

Modèle de rendu et compilation de scènes graphiques

Le modèle de programmation de Sauvage est le modèle de l'implémentation ouverte, par opposition au modèle de la boîte noire (Dourish, 1995). L'interface de programmation principale est celle du graphe de scène SVG. Elle permet de décrire des scènes graphiques et des interactions (grâce notamment au *picking*). Cependant, pour profiter des optimisations qui permettent d'améliorer les performances de rendu et d'interaction, Sauvage donne aussi accès à ses structures de données internes. Le programmeur peut ainsi utiliser des algorithmes qu'il sait efficaces pour le type de rendu et d'interaction qu'il décrit.

Le processus de transformation du graphe de scène en graphe d'affichage est analogue à la compilation d'un programme d'un langage source vers un langage cible. Dans le cas de Sauvage, le langage source est SVG, l'arbre de syntaxe abstraite est le graphe de scène, et le langage cible est le graphe d'affichage. Les différences entre la compilation de scènes graphiques et la compilation de langages sont de plusieurs ordres. Premièrement, il s'agit d'une compilation dynamique puisque la scène est amenée à changer par l'intermédiaire d'interactions. Comme en général seule une partie de la scène change à chaque affichage, la compilation peut ne concerner qu'une partie de l'arbre. Sauvage propose des services de compilation partielle qui permettent d'éviter une compilation entière de la scène à chaque modification. Deuxièmement, la compilation dans Sauvage permet de passer d'une représentation source (SVG) à plusieurs représentations cibles (affichage, élagage...). Enfin, les représentations cibles sont directement accessibles par le programmeur, afin de limiter le recours aux compilations partielles et permettre une rapidité de réaction optimale lors des interactions. Par exemple, il est possible d'avoir accès au nœud du graphe d'affichage qui correspond à un parcours de l'arbre SVG. Ce nœud peut être conservé pendant la durée d'une interaction et modifié directement. Ce mécanisme est employé par notre implémentation de l'interaction de glisser-déposer : le programmeur obtient une référence sur la transformation de translation qui correspond à l'objet manipulé, et la met à jour à chaque mouvement de la souris.

5.2 Interaction

La gestion de l'interaction dans le SERVIR comporte trois aspects. D'abord, il faut prendre en compte les particularités de la plate-forme et de ses dispositifs physiques pour fournir à l'utilisateur des outils adaptés. Ensuite, il faut définir les interactions accessibles à l'utilisateur. Enfin, il faut faire le lien entre les objets graphiques du POG et les objets conceptuels du COG afin de traduire le vocabulaire de l'interaction propre au serveur d'interaction et de rendu —qui possède une sémantique générique— dans celui de l'application —qui possède une sémantique spécifique à son domaine. Ces trois aspects sont mis en œuvre grâce à une boîte à outils d'interaction appelée HsmTk (Blanch, 2002, Blanch et Beaudouin-Lafon, 2006) développée antérieurement et que nous avons modifiée pour l'intégrer au SERVIR.

Des périphériques physiques aux périphériques logiques

HsmTk découvre automatiquement les périphériques physiques disponibles et en fournit une représentation arborescente. Par exemple, la souris est décomposée en une position, un ensemble de boutons, et une molette. La position est elle-même décomposée en deux valeurs unidimensionnelles qu'un périphérique logique peut alors utiliser indépendamment. Il en est de même pour les autres périphériques

comme le clavier, les tablettes graphiques ou, selon la plate-forme, les périphériques USB conformes à la norme Human Interface Devices (USB-HID) (USB Implementers' Forum, 2001). HsmTk permet de créer des périphériques logiques indépendants des périphériques physiques, ce qui fournit un premier niveau d'adaptabilité. Ainsi, on peut définir un périphérique logique pour zoomer un objet, et le réaliser soit par le déplacement horizontal de la souris, soit par la molette de la souris, soit par des touches du clavier.

Programmation des comportements interactifs

HsmTk fournit au programmeur une structure de contrôle proche des machines à états et des *StateCharts* (Harel, 1987) : les machines à états hiérarchiques. Les comportements dynamiques pilotés par des évènements deviennent ainsi des objets à part entière du langage de programmation (Blanch, 2002, Blanch et Beaudouin-Lafon, 2006). Cette structure de contrôle est utilisée pour associer les périphériques logiques aux périphériques physiques et pour réaliser les comportements associés aux objets graphiques. La Figure 5 montre l'exemple simple d'un bouton qui gère correctement l'entrée et la sortie du curseur lorsque le bouton de la souris est enfoncé, et la Figure 6 le code HsmTk correspondant.

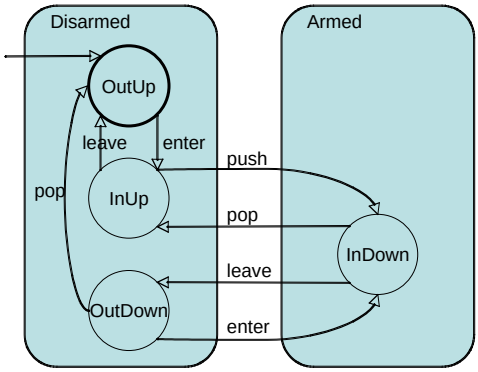

Figure 5. *Le comportement d'un bouton*

Le chargement dynamique de comportements par HsmTk permet de factoriser et de réutiliser des comportements. Un comportement inconnu est d'abord recherché dans l'entrepôt local du SERVIR. S'il n'est pas trouvé, il est réclamé au SERVO et ajouté à l'entrepôt du SERVIR. Ce mécanisme permet de fournir une bibliothèque de comportements allant des *widgets* classiques comme le bouton ci-dessus aux interactions post-WIMP, comme l'interaction bi-manuelle sur une tablette graphique ou les outils transparents (Bier *et al.*, 1993). Il permet aussi d'étendre facilement l'ensemble des techniques d'interaction à disposition du SERVIR.

Ces mécanismes mettent en œuvre une certaine plasticité des interfaces (Thévenin *et al.*, 2003) : l'application ou l'utilisateur peuvent adapter les techniques d'interaction utilisées à la plate-forme ou à leurs besoins. Cette adaptation n'est cependant pas automatique. Du côté de l'application, elle doit avoir été prévue, tandis que du côté de l'utilisateur, elle doit être réalisée explicitement.

Liens entre représentation et comportements

La spécification du comportement des objets graphiques est réalisée par un mécanisme d'annotation du POG : lors de la phase de concrétisation (voir ci-dessous), des comportements paramétrés sont associés aux nœuds du POG et

instanciés par le SERVIR. Afin d'assurer la cohérence entre le modèle du comportement et la représentation graphique, chaque comportement peut exprimer des contraintes structurelles sur le fragment SVG auquel il est associé. Par exemple, le comportement d'un bouton peut spécifier qu'il doit être associé à un groupe ayant deux fils, l'un représentant son état enfoncé, l'autre son état relevé. Le bouton est ainsi un *widget* abstrait spécifié par son comportement et ses contraintes structurelles minimales. De la même façon, une *toolglass* est un ensemble d'outils semi-transparents qui sont déplacés par la main non-dominante et qui répondent au clic de la main dominante "à travers" la palette.

```
hsm Button {
   hsm Disarmed {              // état
      hsm OutUp {              // sous-état
         - enter() > InUp      // transition par "enter"
      }
      hsm InUp {
         - leave() > OutUp
         - push()  > Armed::InDown
      }
      hsm OutDown {
         - enter() > Armed::InDown
         - pop()   > OutUp
      }
   }
   hsm Armed {
      // méthodes appelées à l'entrée / sortie de l'état
      enter { /* armed feedback */ }
      leave { /* disarmed feedback */ }
      hsm InDown {
         - leave() > Disarmed::OutDown
         - pop() broadcast(DO_IT) > Disarmed::InUp
      }
   }
}
```

Figure 6. *Code HsmTk du comportement d'un bouton (la gestion du feed-back est omise pour des raisons de place)*

Les comportements associés aux objets graphiques spécifient les manipulations qu'ils acceptent de la part des périphériques logiques. Le bouton de la Figure 6 répond par exemple aux protocoles *enter/leave* et *push/pop* qui sont utilisés par défaut par le (ou les) curseur(s), le SERVIR se chargeant de trouver, par une fonction de *picking* typée, la cible compatible avec la manipulation en cours parmi les objets présents sous le curseur.

6 Concrétisation

La transformation du SCOG (c'est-à-dire la copie du COG locale au SERVIR) en POG s'appelle la *concrétisation* et vérifie les propriétés suivantes :

- la **réversibilité**, c'est-à-dire le maintien d'un lien bidirectionnel entre un objet du POG et l'objet dont il provient dans le COG ; et

- l'**incrémentalité**, c'est-à-dire le recalcul partiel du POG lors d'une modification du COG (insertion, suppression d'un élément, modification d'un attribut).

Ces propriétés sont nécessaires pour permettre l'interaction graphique. La réversibilité est indispensable à la manipulation directe, car c'est à travers la désignation de leur représentation graphique que l'on manipule les objets de l'application. Il faut donc pouvoir faire correspondre un objet du COG à tout objet du POG. L'incrémentalité est souhaitable pour des questions de performance : si l'ensemble du POG est recalculé à chaque modification du COG, il est peu probable que l'on puisse obtenir un temps de rafraîchissement inférieur au vingtième de seconde, garant d'une bonne qualité de l'interaction, sur des scènes un peu complexes.

La concrétisation a lieu dans le SERVIR et est définie par une *feuille de style*. Nous avons utilisé XSLT (*Extensible Stylesheet Language Transformations)*, un langage de transformation d'arbres XML standardisé par le W3C (W3C, 1999). La feuille de style est donc une transformation XSLT qui est appliquée à des fragments du COG et qui produit des fragments du POG, dans notre cas du SVG annoté par les comportements des objets graphiques reconnus par HsmTk. Un programme XSLT ne satisfait pas en tant que tel la propriété d'incrémentalité : il transforme un arbre XML en un autre. Nous avons donc imposé des restrictions afin de pouvoir recouvrer cette propriété : nous imposons que chaque élément du COG soit transformé en un groupe SVG, et que les images par la transformation des descendants d'un élément soient incluses dans le sous-arbre SVG qui est l'image de cet élément. Cette propriété de monotonie nous a permis de réaliser simplement le moteur de transformation XSLT incrémental nécessaire à notre prototype, sans limitation majeure sur le type de transformation COG-POG. Les travaux récents d'Onizuka et al. (2005) montrent cependant que ces restrictions ne sont pas nécessaires pour y parvenir et offrent donc des perspectives intéressantes pour INDIGO.

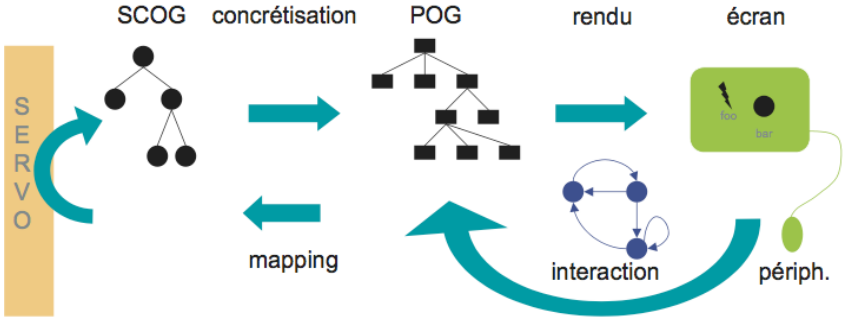

Figure 7. *Fonctionnement général du SERVIR*

La transformation XSLT produit également des annotations du graphe SVG qui permettent d'établir le lien bidirectionnel entre les objets conceptuels et leurs représentations. Pour cela, à chaque groupe SVG issu de la transformation d'un objet conceptuel est associé un identifiant. Le moteur de concrétisation maintient une table qui permet de retrouver toutes les images d'un objet particulier du COG et ainsi de gérer incrémentalement les notifications du SERVO (changement de valeur d'un attribut, ajout ou suppression d'un élément) en remplaçant, insérant ou supprimant des éléments de l'arbre SVG. À l'inverse, chaque groupe SVG contient

un attribut identifiant l'objet qu'il représente sous la forme d'un chemin dans le COG, représenté dans le formalisme XPATH de description de chemins dans les documents XML. Cet attribut permet au comportement associé au groupe de traduire les manipulations interactives en appels de méthodes de l'objet conceptuel. Ces appels de méthode sont relayés au SERVO de manière transactionnée, comme expliqué dans la section 4. Les modifications éventuelles des objets conceptuels qui en résultent sont notifiées à l'ensemble des SERVIR connectés, provoquant en bout de chaîne la mise à jour des parties du POG correspondant aux objets affectés et le réaffichage de l'écran. La Figure 7 illustre ce fonctionnement général.

graphe conceptuel *table des identifiants*

```
<Folder type='Folder'>                                    d0e07
   <Name type='string'>~/test</Name>                      d0e12
   <Entry type='File'>                                    d0e15
      <Name type='string'>README</Name>                   d0e17
   </Entry>
   <Entry type='File'>                                    d0e24
      <Name type='string'>INSTALL</Name>                  d0e26
   </Entry>
   ...
```

graphe perceptuel

```
<!-- arbre -->
<g hsm:behaviour='tree' indigo:path='/' id='d0e07'>
   <rect width='1000' height='16' style='fill:url(#bgd)'/>
   <use xlink:href='#closed'/>
   <text indigo:path='/Name' id='d0e12'
         x='22' y='12.5'>~/test</text>

   <!-- contenu -->
   <g transform='translate(16,16)'>
      <g hsm:behaviour='node' indigo:path='/Entry[0]'
         id='d0e15'>
         <rect width='1000' height='16'
               style='fill:white; opacity:0'/>
         <text indigo:path='/Entry[0]/Name' id='d0e17'
               x='22' y='12'>README</text>
      </g>
      <g hsm:behaviour='node' indigo:path='/Entry[1]'
         id='d0e24'>
         <rect width='1000' height='16'
               style='fill:white; opacity:0'/>
         <text indigo:path='/Entry[1]/Name' id='d0e26'
               x='22' y='12'>INSTALL</text>
      </g>
      ...
```

Figure 8. *COG (XML en haut) et POG (SVG en bas) simplifiés représentant un système de fichiers*

La Figure 8 montre en haut un graphe conceptuel simplifié représentant une arborescence de fichiers, tel qu'il est publié par un SERVO gestionnaire de fichiers, et la table des identifiants donnant les images de chaque élément dans le POG. Cette table est calculée lors de la génération du POG montré en bas de la Figure 8. Celui-ci a la structure d'arbre que nous connaissons enrichie pour chaque groupe correspondant à un élément du COG des annotations permettant de remonter à celui-ci (attribut `indigo:path`) et d'être retrouvé à partir de la table des identifiants (attribut `id`). L'arbre est par ailleurs annoté pour spécifier les comportements interactifs des divers éléments (attributs `hsm:behaviour`).

7 Synthèse du fonctionnement d'une application

Maintenant que nous avons détaillé l'ensemble des composants de l'architecture, nous pouvons résumer le fonctionnement général d'une application. On considère que le SERVIR de l'utilisateur tourne, ainsi que le ou les SERVO auxquels il souhaite pouvoir se connecter. Le SERVIR doit fournir une interface minimale permettant à l'utilisateur de se connecter à un SERVO, par exemple en spécifiant son adresse Internet. La séquence d'initialisation est alors la suivante :

1. Le SERVIR envoie un message de connexion au SERVO contenant sa description, en particulier le type de POG qu'il sait gérer, par exemple un POG SVG ;
2. Le SERVO décide s'il accepte la connexion et si oui confirme l'ouverture de la session ;
3. Le SERVIR demande au SERVO son modèle conceptuel, afin de connaître les classes d'objets et les méthodes qu'il va manipuler ;
4. Le SERVIR demande au SERVO la feuille de style qui correspond à son type de POG, et associe cette feuille de style aux objets de ce SERVO ; il la complète éventuellement par une feuille de style locale, de façon à ce que les définitions locales aient précédence sur celles du SERVO ;
5. Le SERVIR demande au SERVO son objet racine ;
6. Le SERVO retourne cet objet, ce qui provoque dans le SERVIR la création du SCOG, sa concrétisation en POG et son affichage ;
7. Le SERVO et le SERVIR sont désormais dans le mode normal de fonctionnement.

Une fois lancé, le SERVIR doit être réactif à la fois aux événements venant de l'utilisateur et aux messages en provenance des SERVO auxquels il est connecté. Lorsqu'il reçoit un message d'un SERVO, il s'agit d'une modification du COG qui doit être appliquée au SCOG. Cette modification peut résulter soit d'une action de l'utilisateur, soit parce que le SERVO a été modifié par ailleurs et se synchronise avec ses SERVIR. Dans les deux cas, le SERVIR déclenche la concrétisation incrémentale de la partie du SCOG qui a été modifiée en utilisant la feuille de style associée au SERVO qui a provoqué la modification, ce qui met à jour le POG et sa présentation (l'affichage dans le cas d'une interface graphique).

Lorsqu'il reçoit un événement de l'utilisateur, le SERVIR le transmet à sa machine à états. Selon les transitions de l'état courant de la machine à états, le SERVIR peut avoir besoin de lancer une opération de *picking* afin de déterminer l'objet sous le curseur. Ce *picking* peut être typé, c'est-à-dire spécifier que l'on cherche uniquement des objets compatibles avec une ou plusieurs opérations données. Par exemple, l'outil permettant de déplacer les objets par glisser-déplacer ne fonctionne qu'avec des objets ayant déclaré le comportement “move”. En cas de

succès, le *picking* retourne un objet du POG. Cet objet peut être utilisé par la machine à états de diverses manières. Il peut servir par exemple à produire un retour d'information. Ainsi, si l'on veut déplacer l'ombre de l'icone d'un fichier lors d'un glisser-déposer, la machine à états va créer dans le POG une copie de l'objet en cours de déplacement, lui attribuer un niveau de transparence, et le translater en fonction des déplacements de la souris. A la fin de l'interaction, cet objet de feed-back sera détruit par la machine à états.

Un objet du POG permet aussi de modifier le COG, puisqu'il contient le chemin de l'objet du COG qui l'a créé. Le but d'une interaction est typiquement d'activer une méthode d'un objet du modèle conceptuel, comme par exemple changer un fichier de répertoire à la suite d'une opération de glisser-déposer de l'icone du fichier sur celui du répertoire. La machine à états peut récupérer l'identité de l'objet conceptuel associé à l'objet perceptuel qui a été désigné et invoquer une méthode de cet objet. Cette invocation provoquera probablement la modification du COG, qui à son tour déclenchera la notification du changement par le SERVO, qui sera traitée par le SERVIR comme indiqué ci-dessus.

8 Exemples d'applications

Afin de valider notre approche et de tester l'architecture développée, nous avons réalisé quelques applications simples que nous présentons ici.

8.1 Explorateur de Fichiers

Le premier exemple est un explorateur de fichiers illustré en Figure 9. Celui-ci permet les actions élémentaires comme le déplacement, la suppression ou le renommage de fichiers. Le SERVO correspondant assure la cohérence avec une sous-partie de son système de fichiers local. Cette application utilise le mécanisme de population paresseuse du SCOG : le SERVIR ne charge l'arborescence qu'au fur et à mesure de l'ouverture des branches, ce qui permet une utilisation interactive y compris au travers d'un réseau non local.

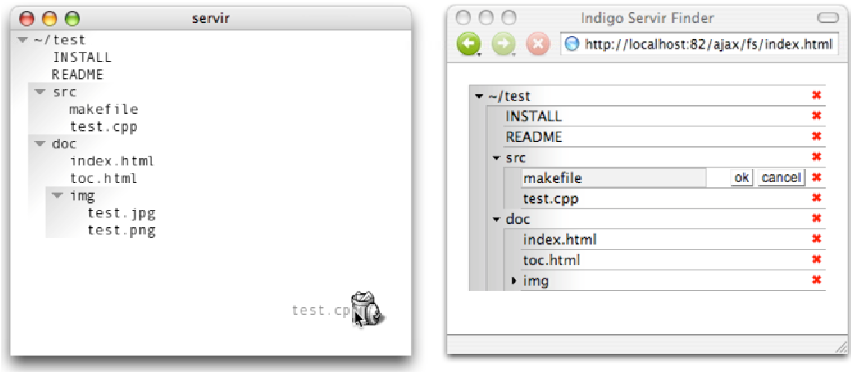

Figure 9. *Explorateur de fichier :*
à gauche, SERVIR basé sur SVG offrant des interactions avancées ;
à droite, SERVIR basé sur HTML et utilisant un navigateur Web

Une première version du SERVIR (Figure 9, à gauche) utilise le rendu SVG et l'interaction par machines à états décrits précédemment. Elle permet des interactions avancées comme le déplacement de fichier par "glisser-déposer", ou la

suppression de fichier de manière iconique par un déplacement dans la poubelle représentée dans le coin inférieur droit de l'espace de travail.

Une seconde version de l'explorateur (Figure 9, à droite) a également été développée sur une plate-forme de rendu et d'interaction plus simple : un navigateur Web standard. La communication avec le SERVO utilise le même protocole que le SERVIR avancé puisque le protocole SOAP est transporté par HTTP, le protocole des serveurs Web. De plus la technologie AJAX (*Asynchronous Javascript And XML*), qui est intégrée à la majorité des navigateurs Web modernes, permet d'interroger un serveur HTTP, d'effectuer des transformations XSLT, et de manipuler le contenu de la page HTML affichée dans la fenêtre du navigateur (son arbre DOM) avec le langage Javascript (Garrett, 2005). Le SERVIR est donc programmé en Javascript, exécuté dans le navigateur Web, et fonctionne sur le même principe que le SERVIR SVG. Le SCOG est transformé localement par une transformation XSLT en document HTML qui est affiché par le navigateur. Les mises à jour sont reçues en réponse à des requêtes SOAP adressées au SERVO selon le même principe de *pull-wait-push*. Le document HTML est mis à jour depuis le Javascript en fonction de ces modifications. Enfin, l'interaction est gérée par du code Javascript qui est téléchargé, tout comme la transformation XSLT qui définit la correspondance entre SCOG et POG, avec le document lors de la connexion initiale au SERVIR.

Outre la validation de l'architecture générale et du partage d'objets d'un même SERVO entre plusieurs SERVIR, cette application démontre surtout la possibilité d'offrir deux interfaces distinctes permettant des niveaux d'interaction différents avec un même SERVO.

8.2 Jeu Multi-Joueurs : Puissance 4

La Figure 10 montre Puissance 4, un jeu qui peut être joué à deux joueurs. Deux SERVIR partagent ainsi les objets d'un même SERVO. Cliquer sur une colonne fait tomber un pion dans celle-ci. Cette interaction réutilise le *widget* abstrait du bouton dont la représentation graphique est un rectangle transparent superposé à chaque colonne, accompagné d'un halo lorsque le bouton est activé.

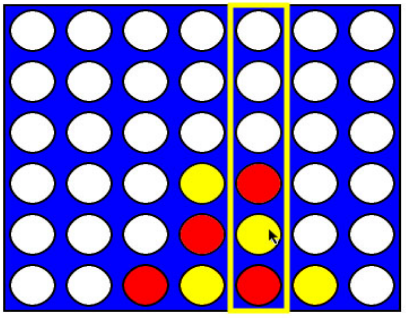

Figure 10. *Jeu multi-joueur "Puissance 4"*

La gestion du dialogue, qui impose ici que chacun joue à tour de rôle, est prise en charge par le SERVO qui refuse l'ajout d'un pion lorsque ce n'est pas le tour du joueur. Afin de fournir des retours d'information, l'échec de cette interaction pourrait produire un feed-back visuel, de même que le modèle conceptuel pourrait être enrichi d'un objet indiquant le joueur dont c'est le tour. Dans un cas aussi simple, le contrôle de dialogue correspondant au tour de rôle pourrait bien sûr avoir

lieu dans le SERVIR. Nous avons voulu montrer qu'il peut aussi avoir lieu dans le SERVO, ce qui est plus approprié si, par exemple, les règles du jeu sont complexes.

Cette application illustre, de façon certes simple, les possibilités de travail coopératif avec une architecture qui est ici centralisée. Le SERVO garantit la synchronisation des différents SERVIR et le transactionnement, qui sont indispensables à un fonctionnement collecticiel robuste. Cependant, le concepteur d'une application collecticielle doit intégrer au modèle conceptuel les objets et méthodes nécessaires à la mise en œuvre d'un collecticiel utilisable, comme par exemple la gestion des rôles qui donnent des droits d'accès différents aux données.

8.3 Vue Radar

Afin de tester l'architecture sur une application plus réaliste, nous avons porté Glance, une application de type "image radar" développée au CENA. Glance permet aux contrôleurs aériens de surveiller un espace aérien en visualisant les informations relatives à un vol : position, identificateur, vitesse au sol, tendance de montée, etc. (La Figure 11 est tirée de la version INDIGO). Glance représente notamment la trajectoire de chaque avion en utilisant cinq cercles situés aux positions récentes de l'avion, le rayon de chaque cercle étant inversement proportionnel à l'âge de la position (cf. en figure 11 le cône, attaché à chaque avion, formé par ces 5 cercles). Le SERVIR n'a pas la puissance d'expression nécessaire pour conserver les positions passées et représenter la trajectoire. En effet, tout nœud du POG doit provenir d'un nœud du COG, c'est donc au COG de s'adapter aux besoins de l'interaction en stockant explicitement les positions récentes. Il ne s'agit pas d'une limitation due à l'architecture, mais bien d'une transformation d'un objet de l'interface en concept de l'application utile à l'interaction.

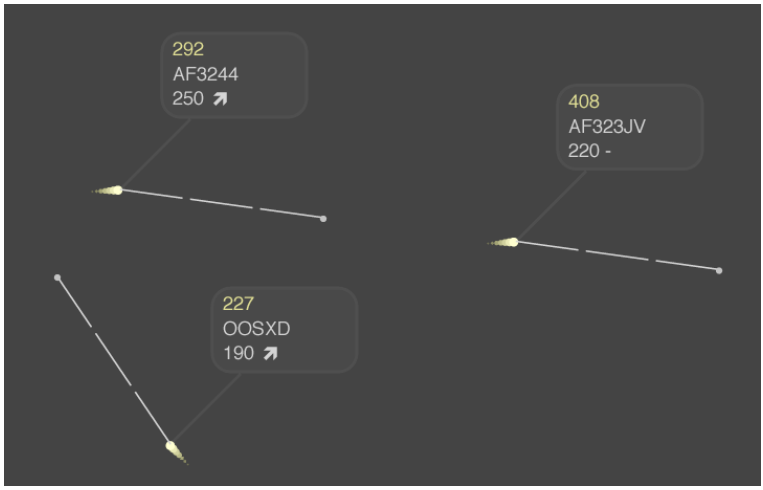

Figure 11. *Vue radar avec 3 avions, leurs positions récentes (cônes), leur direction (pointillés) et leur étiquette (cadre) indiquant l'identifiant, l'altitude et une flèche en cas de changement d'altitude*

L'application originale était monolithique, optimisée pour des types d'affichage et d'interaction spécialisés et adaptés au contexte de l'application : utilisation du détourage ("*clipping*") et de la transparence pour afficher les informations de vol, et interactions post-WIMP de type *pan-and-zoom* (défilement et changement d'échelle continus). Le portage de Glance sur INDIGO offre des performances d'affichage et d'interaction équivalentes à l'application originale, et permet notamment d'effectuer des interactions de type *pan-and-zoom* de façon fluide. La transformation du COG en

POG par l'intermédiaire des feuilles de style est en revanche un peu lente, et représente le principal goulot d'étranglement de l'application. Nous estimons que le moteur de transformation peut être très largement optimisé, notamment parce qu'il ne nécessite pas toute la puissance des transformations XSLT.

Afin de tester d'autres interactions post-WIMP, nous avons ajouté une interaction basée sur une palette transparente bimanuelle ou *toolglass* (Bier et al., 1993). Le principe est le suivant : l'utilisateur peut donner aux avions qu'il contrôle des ordres de changement de cap, de niveau de vol, ou de vitesse. Afin de déterminer les objets graphiques qui sont susceptibles d'interpréter ces interactions, les objets graphiques sont annotés avec une étiquette indiquant par exemple la compatibilité de l'objet "avion" avec l'interaction "changement de cap". Lors d'un clic à travers un outil de la palette transparente, celui-ci parcourt la pile d'objets graphiques se situant sous le clic, trouve le premier objet graphique compatible avec l'interaction, et lance l'interaction effective (par exemple la spécification d'un cap à l'aide d'une ligne élastique). À la fin de cette interaction, la méthode associée au changement de cap est appelée sur l'objet conceptuel lié à l'objet graphique.

Le processus d'interaction se déroule donc entièrement au niveau du SERVIR : aucun événement nécessaire à l'interaction n'est transmis au SERVO, seuls les appels de méthodes qui, à la fin des interactions, déclenchent les actions du noyau fonctionnel lui sont envoyés. Cependant, le SERVO a une connaissance des interactions possibles sur ses objets, puisque c'est lui qui spécifie la compatibilité des objets avec les interactions. C'est ainsi que le contrôle du dialogue est spécifié à un niveau abstrait dans le SERVO, et à un niveau concret dans le SERVIR.

9 Comparaison avec l'état de l'art

De nombreux modèles, architectures logicielles et boîtes à outils ont été proposés pour séparer le code fonctionnel de la description des interactions et du rendu dans des composants répartis. Ils diffèrent en particulier par le niveau d'abstraction du protocole de communication entre les composants, celui-ci résultant notamment de la répartition des rôles entre composants. Ils diffèrent également par la qualité et la quantité des fonctionnalités de rendu et d'interaction. Dans cette section, afin de faciliter les comparaisons, nous appelons "serveur" le composant local de gestion du rendu et des interactions, et "client" ou "application" le composant fonctionnel du logiciel. Le client se connecte au serveur pour proposer des interactions à l'utilisateur. Selon cette terminologie, le SERVO est donc le client ou l'application et le SERVIR est le serveur.

Le modèle VNC (Richardson *et al.*, 1998) correspond au niveau d'abstraction le plus bas du protocole client-serveur : le client gère lui-même le rendu et l'interaction, et envoie au système distant les images générées sous forme de rectangles de pixels. Dans l'autre sens, les informations sur l'interaction sont envoyées sous forme d'événements de bas niveau, comme le déplacement du curseur. Ce modèle est conceptuellement simple à mettre en œuvre mais il a plusieurs inconvénients majeurs. En premier lieu, il n'autorise que des applications qui cohabitent plutôt qu'elles ne collaborent, car le rendu final de chaque application est indépendant. Ensuite, les performances du rendu dépendent fortement de la bande passante du réseau, notamment en cas de changement important de la scène graphique. Par exemple, les interfaces zoomables nécessitent la transmission continue des images complètes de la scène lors de la navigation *pan-and-zoom*. Pour un écran haute définition de 3200×2400 pixels et un taux de rafraîchissement de 20 images/s, il faut une bande passante proche de 5 Gbit/s, ce qui est hors de

portée des réseaux actuels. Enfin, l'interaction étant gérée de façon distante, tous les évènements doivent être transmis : un "glisser-déposer" par exemple doit traiter tous les événements de changement de position du curseur. La boucle interactive est donc limitée par les performances du réseau dans les deux sens, ce qui détériore la fluidité de l'interaction et se traduit par un décalage entre l'action de l'utilisateur et la réaction du système au travers de l'affichage. De plus, le fait que l'interaction soit gérée dans l'application cliente n'encourage pas les programmeurs à séparer le code fonctionnel de l'interaction et limite les possibilités de réutilisation.

X11 (Nye, 1988) met en œuvre un modèle de niveau d'abstraction légèrement plus élevé en sortie, fondé sur des primitives graphiques simples. Ainsi, pour afficher un rectangle, il n'est pas nécessaire d'envoyer des blocs de pixels ; il suffit de lancer une commande de type "TracerRectangle" accompagnée des coordonnées du rectangle. Les performances du rendu ne dépendent donc pas directement de la bande passante, par contre la richesse du rendu graphique est limitée par le modèle graphique, assez pauvre, de X11. De plus, la gestion des interactions reste du ressort des applications, puisque les événements sont toujours de bas niveau. Les interactions, même de type WIMP, sont gérées par des librairies liées au client. Enfin, les possibilités de composition des documents sont limitées à l'imbrication de fenêtres rectangulaires.

Fresco/Berlin (Linton et Price, 1993) est fondé sur un graphe de scène géré au niveau du serveur : l'application cliente construit une scène pour le rendu, stockée dans le serveur, et celui-ci dispose de librairies pour gérer l'interaction à son niveau. Le protocole est donc de niveau d'abstraction plus élevé que dans X11. Cette architecture se rapproche de celle d'INDIGO, cependant elle n'offre pas de notions comparables au modèle conceptuel (COG), au modèle perceptuel (POG) et au modèle d'interaction (machines à états) d'INDIGO et reste donc d'un niveau d'abstraction moins élevé. De plus, le choix de CORBA comme intergiciel est contestable car il pénalise les temps de réponse interactifs. Enfin ce projet, qui devait remplacer la boîte à outils XToolkit de X11, semble actuellement arrêté.

Les applications Web dites "Web 2.0", comme par exemple Gmail, utilisent la technologie AJAX (Garrett, 2005) des navigateurs Web récents qui permet d'invoquer une requête HTTP en réponse à une action de l'utilisateur sans remplacer l'ensemble de la page mais en modifiant une partie de son contenu en fonction du résultat de la requête. La différence avec INDIGO est que cette possibilité est purement technique et que l'application, du côté client comme serveur, doit gérer de façon *ad hoc* cette répartition des rôles. Même si l'on commence à voir quelques outils d'aide au développement de ces applications, comme Dojo (Dojo Foundation, 2006), l'interaction reste limitée et se réduit le plus souvent aux interactions classiques d'un navigateur Web : traversée de liens et remplissage de formulaires. Comme nous l'avons montré plus haut avec le navigateur de fichiers, INDIGO peut utiliser la technologie AJAX pour implémenter un SERVIR dans un navigateur Web.

Display PostScript (DPS) (Adobe Systems Inc., 1993), originellement développé pour le NeXT, s'appuie sur le modèle graphique du langage PostScript et permet d'utiliser des commandes d'affichage de haut niveau pour générer des graphismes 2D de grande qualité. Le protocole d'affichage consiste pour le client à télécharger du code PostScript sur le serveur qui l'exécute pour le rendu. Ce modèle ne prend pas en compte l'interaction, il propose seulement le calcul de la forme graphique sous le curseur. En réalité, DPS n'est pas autonome : il nécessite un système de fenêtrage et se repose sur ses mécanismes pour l'interaction.

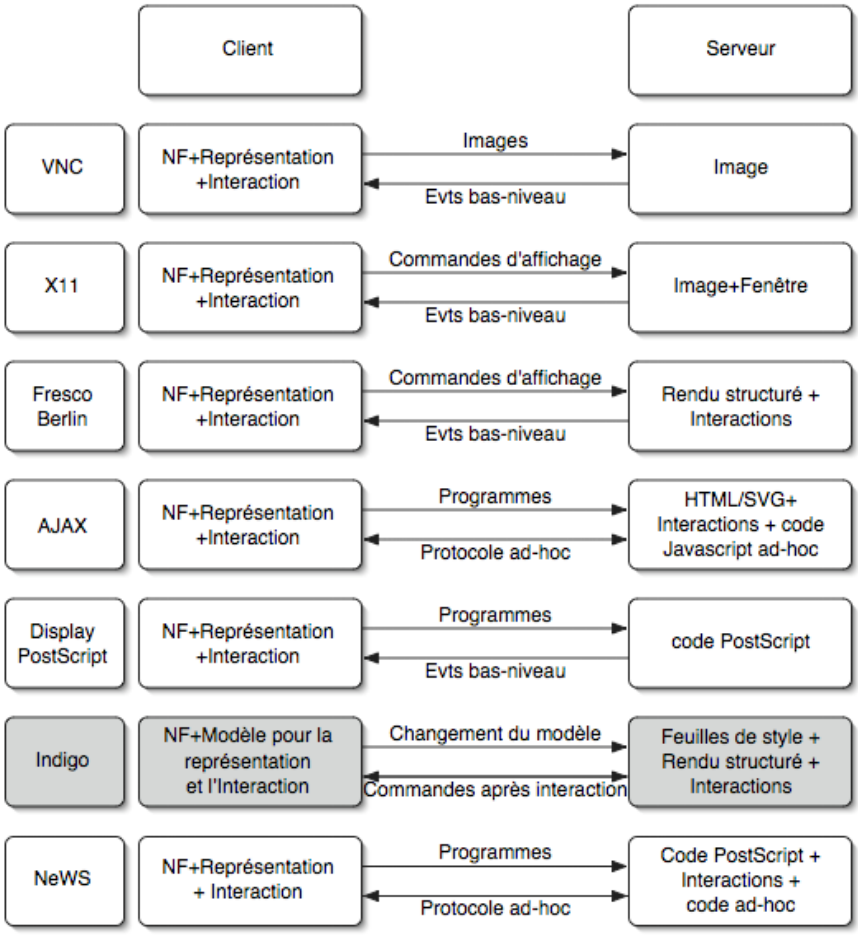

Figure 12. *Comparaison des modèles client-serveur (NF = Noyau Fonctionnel)*

NeWS (Gosling *et al.*, 1989), développé par Sun et contemporain de DPS, est un système d'affichage et d'interaction également basé sur le langage PostScript. Le serveur est capable d'interpréter du code PostScript fourni par l'application cliente. Le langage PostScript est étendu pour permettre la création de fenêtre et la gestion de l'interaction. Le code téléchargé permet donc de définir l'affichage et l'interaction, mais aussi le protocole de communication entre le serveur et l'application cliente. Si les possibilités d'expression de l'interaction, de rendu, et de communication sont maximales, elles engendrent aussi des pratiques qui nuisent à la séparation du code fonctionnel et de l'interaction, et donc à sa réutilisabilité : progressivement, l'ensemble de l'application est développé en PostScript et téléchargé dans le serveur.

La Figure 12 compare ces différents systèmes en fonction du processus (client ou serveur) qui gère les différents aspects d'une application interactive. Nous avons positionné INDIGO entre DPS et NeWS. En effet, bien que la puissance

d'expression des langages permettant de décrire les objets conceptuels, des objets perceptuels et de l'interaction est moins élevée que dans DPS et NeWS, nous considérons que le niveau d'abstraction du protocole de communication est comparable à ce que ces systèmes cherchaient à réaliser, avec en particulier la gestion des interactions au niveau du serveur (le SERVIR d'INDIGO).

Si l'on se réfère au modèle Arch (Arch, 1992), les architectures présentées ci-dessus séparent leurs éléments de part et d'autre du réseau à différents niveaux. Les premiers (VNC, X11, Fresco, DPS) les placent tous du côté du client, alors que NeWS permet de les placer tous du côté du serveur. Nous avons opté, avec INDIGO, pour une solution plus nuancée qui place le noyau fonctionnel de l'application dans le SERVO, en utilisant un vocabulaire propre aux objets manipulés et à leur domaine d'application, et les représentations et les techniques d'interaction associées dans le SERVIR, en les adaptant aux capacités de la plate-forme. La gestion des objets du domaine et du dialogue de haut niveau font donc partie du SERVO, alors que la présentation et l'interaction (dialogue de bas niveau) sont confiées au SERVIR.

10 Conclusion et perspectives

Nous avons présenté une architecture répartie destinée au développement d'applications interactives caractérisées par les propriétés suivantes : prise en compte de plates-formes différentes, utilisation de techniques d'interaction avancées, applications collecticielles. L'architecture INDIGO est fondée sur deux types de composants, les serveurs d'objets et les serveurs d'interaction et de rendu, et sur un protocole de haut niveau. Nous avons réalisé une première implémentation de cette architecture et l'avons validée par le développement de quelques applications.

Ces travaux nous ont convaincu du bien-fondé de notre approche tout en soulevant un certain nombre de points que nous souhaitons aborder dans nos travaux futurs. En premier lieu, la définition d'un formalisme plus adéquat pour décrire la concrétisation est nécessaire. Les langages tels que XSLT sont puissants mais difficiles à mettre en œuvre, aussi des alternatives doivent être étudiées. En second lieu, la validité de notre approche pour des formes non-graphiques de rendu et pour la multimodalité doit être testée. Cela permettra également de tester la possibilité de travail coopératif sur les mêmes objets conceptuels avec des modalités différentes ainsi que la mise en œuvre d'interactions distribuées comme le *pick-and-drop* (Rekimoto, 1997) qui étant l'interaction classique de "glisser-déposer"aux environnements multi-surfaces. En troisième lieu, il nous semble intéressant d'étudier les possibilités de programmation par l'utilisateur qu'offre ce type d'architecture, particulièrement celles qui peuvent être réalisées sans modification du SERVO. Cela peut concerner la définition de nouvelles techniques d'interaction, la modification de la présentation des objets conceptuels, l'ajout de fonctions de collaboration, etc. En tout état de cause, nous sommes convaincus que la migration vers une approche plus distribuée, plus modulaire et plus collaborative de l'interaction, telle que le propose INDIGO, est indispensable si l'on veut s'affranchir des limitations des environnements actuels.

11 Remerciements

Le projet INDIGO a été financé par le RNTL (Réseau National des Technologies Logicielles). Nous remercions les membres du projet In Situ qui ont participé à ce projet, notamment Jean-René Courtois.

12 Références

Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., Preda, N. (2004). Lazy query evaluation for Active XML. In *Proceedings of ACM SIGMOD'04*, 227–238, ACM Press.

Adobe Systems Inc. (1993). *Programming the Display PostScript System with X*. Addison-Wesley Longman Publishing Co., Inc.

Arch (1992). A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop. *ACM SIGCHI Bulletin*, 24(1), 32–37.

Beaudouin-Lafon, M., Lassen, H.M. (2000). The architecture and implementation of CPN2000, a post-WIMP graphical application. In *Proceedings of ACM Symposium on User Interface Software and Technology (UIST '00)*, 181–190, ACM Press.

Bier, E.A., Stone, M.C., Pier, K., Buxton, W., DeRose, T.D. (1993). Toolglass and magic lenses: the see-through interface. In *Proceedings of ACM SIGGRAPH'93*, 73–80, ACM Press.

Blanch, R. (2002). Programmer l'interaction avec des machines à états hiérarchiques. In *Actes des journées francophones sur l'Interaction Homme-Machine (IHM '02)*, 129–136, ACM Press.

Blanch, R., Beaudouin-Lafon, M., Conversy, S., Jestin, Y., Baudel, T., Zhao, Y.P. (2005). INDIGO : une architecture pour la conception d'applications graphiques interactives distribuées. In *Actes des dix-septièmes journées francophones sur l'Interaction Homme-Machine (IHM '05)*, 139–146, ACM Press.

Blanch, R., Beaudouin-Lafon, M. (2006). Programming rich interactions using the hierarchical state machine toolkit. In *Proceedings of the working conference on Advanced visual interfaces (AVI '06)*, 51–58, ACM Press.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D. (2000). *Simple object access protocol (SOAP) 1.1. Technical report*, W3C.

Chatty, S., Sire, S., Vinot, J.-L., Lecoanet, P., Lemort, A., Mertz, C. (2004). Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of ACM Symposium on User Interface Software and Technology (UIST '04)*, 267–276, ACM Press.

Dojo Foundation (2006). *The Dojo Javascript Toolkit*. http://www.dojotoolkit.org

Dourish, P. (1995). Accounting for System Behaviour: Representation, Reflection and Resourceful Action. In *Proceedings of the third decennial conference Computers in Context: Joining Forces in Design (CIC '95)*. Aarhus, Denmark, August 14-18, Department of Computer Science, Aarhus University, 147–156.

Conversy, S., Fekete, J.-D. (2002). The SVGL toolkit: enabling fast rendering of rich 2D graphics. *Technical Report* 02/1/INFO, École des Mines de Nantes.

van Engelen, R.A., Gallivan K.A. (2002). The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *Proceedings of 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGRID '02),* p. 128, IEEE Computer Society.

Furnas, G.W., Bederson, B.B. (1995). Space-scale diagrams: understanding multiscale interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*, 234–241, ACM Press.

Garrett, J.J. (2005). *Ajax: A New Approach to Web Applications*. Adaptive Path, LLC. http://www.adaptivepath.com/publications/essays/archives/000385.php.

Gosling, J., Rosenthal, D.S. H., Arden, M. J. (1989). *The NeWS book: an introduction to the network/extensible window system*. Springer-Verlag New York, Inc.

Gray, J., Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 231–274.

Internet Society (1997). *Hypertext Transfert Protocol (HTTP), version 1.1*. http://www.ietf.org/rfc/rfc2068.txt

Kaplan, P. (1999). Make a sweep with clean beans. *JavaWorld*, 11, 1–4.

Linton, M., Price, C. (1993). Building distributed user interfaces with Fresco. *X Resource*, 5, 77–87.

Nye, A. (1988). *XLIB Programming Manual and Reference Manual*. O'Reilly & Associates, Inc.

Onizuka, M., Fong Y.C., Michigami, R., Honishi, T. (2005). Incremental maintenance for materialized XPath/XSLT views. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)*, 671–681, ACM Press.

Rekimoto, J. (1997). Pick-and-drop: a direct manipulation technique for multiple computer environments. In *Proceedings of the 10th annual ACM Symposium on User Interface Software and Technology (UIST '97)*, 31–39, ACM Press.

Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A. (1998). Virtual network computing. *IEEE Internet Computing*, 2(1), 33–38.

Thevenin, D., Calvary, G., Coutaz, J. (2003). A Reference Framework for the Development of Plastic User Interfaces. In A. Seffah, H. Javahery (eds.), *Multiple User Interfaces: Cross-Platform Applications and Context-Aware Interfaces*, 29–51, John Wiley & Son, Ltd.

USB Implementers' Forum (2001). *Device Class Definition for Human Interface Devices (HID)*. USB Implementers' Forum.

W3C (1999). *XSL Transformations (XSLT), version 1.0*. W3C Recommendation. http://www.w3.org/TR/xslt.

W3C (2003). *Scalable Vector Graphics (SVG) 1.1 Specification*. W3C Recommendation. http://www.w3c.org/Graphics/SVG.

W3C (2003). *Simple Object Access Protocol (SOAP), version 1.2*. W3C Recommendation. http://www.w3.org/TR/soap.

W3C (2006). *XML Query Language (XQuery), version 1.0*. W3C Recommendation. http://www.w3.org/TR/xquery.

Woo, M., Neider, J., Davis, T. (2003) *OpenGL Programming Guide*. Addison-Wesley, 4th ed, ISBN: 0321173481