HAL
open science

INDIGO: une architecture pour la conception d'applications graphiques interactives distribuées

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

To cite this version:

Renaud Blanch, Michel Beaudouin-Lafon, Stéphane Conversy, Yannick Jestin, Thomas Baudel, et al.. INDIGO: une architecture pour la conception d'applications graphiques interactives distribuées. IHM'05 17ème Conférence Francophone sur l'Interaction Homme-Machine, ACM, Sep 2005, Toulouse, France. pp.139-146 / ISBN: 978-1-4503-3844-8, 10.1145/1148550.1148568 . hal-01285967

HAL Id: hal-01285967

https://enac.hal.science/hal-01285967

Submitted on 5 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INDIGO : une architecture pour la conception d'applications graphiques interactives distribuées

Renaud Blanch, Michel Beaudouin-Lafon
LRI & INRIA Futurs*
Orsay, France
blanch@lri.fr, mbl@lri.fr

Stéphane Conversy, Yannick Jestin
ENAC & DSNA/SDER †
Toulouse, France
conversy@enac.fr, jestin@cena.fr

Thomas Baudel, Yun Peng Zhao
ILOG
Paris, France
baudel@ilog.fr

RÉSUMÉ
Le projet INDIGO vise à concevoir une nouvelle génération d'outils pour le développement d'applications graphiques interactives distribuées. L'architecture proposée 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. 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
The INDIGO project develops a new generation of tools for distributed interactive applications. The proposed architecture is composed of object servers that manage the applications's data and 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. This approach was validated through examples that illustrate various types of presentation and interaction devices.

CATEGORIES AND SUBJECT DESCRIPTORS: H.3.4 [Information Storage and Retrival]: Systems and Software — Distributed systems; H.5.2 [Information Interfaces and Presentation]: User Interfaces.

GENERAL TERMS: Design, Human factors

KEYWORDS: Advanced interaction, distributed interactive system, software architecture, toolkits.

*projet InSitu – Pôle Commun de Recherche en Informatique du plateau de saclay – CNRS, École Polytechnique, INRIA, Université Paris-Sud.
†Ecole Nationale de l'Aviation Civile - Direction des Services de la Navigation Aérienne / Sous-Direction Etudes et Recherche appliquée

INTRODUCTION

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). Ceci a conduit aux boîtes à outils et aux constructeurs d'interface actuels, qui 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* (cliquer-tirer) ne sont pas prises en compte et imposent de lourds développements ad hoc, pour chaque application.

Si quelques boîtes à outils post-WIMP vont au-delà du modèle du *widget*, d'autres aspects importants ne sont pas pris en compte. Ainsi, les plate-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. L'évolution des usages rend aussi 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 plate-formes et l'intégration du collecticiel, nous avons développé et validé une architecture logicielle appelée INDIGO (Interactive Distributed Graphical Object) fondée sur les principes suivants :

- 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 ;
- transformation des objets de l'application en un graphe de scène dont le rendu est contrôlé en fonction de la plate-forme ;
- interprétation des actions de l'utilisateurs en commandes de haut niveau sur les objets du domaine ;
- 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.

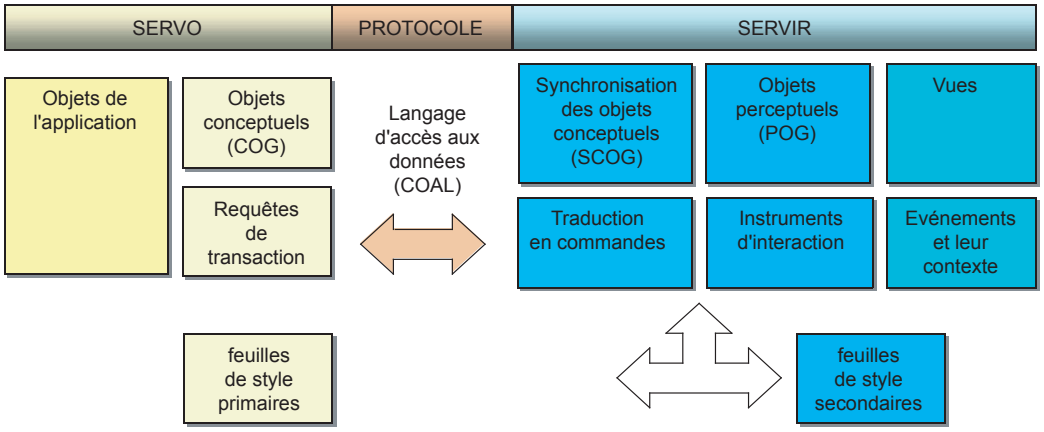

Figure 1 : Architecture INDIGO

ARCHITECTURE

L'architecture répartie d'INDIGO[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.

Dans le modèle architectural de référence ARCH, l'architecture INDIGO consiste à localiser les composants du domaine (une des branches de l'arche) ainsi que le contrôle du dialogue (la clé de voûte de l'arche) dans le SERVO, et les composants de présentation et d'interaction (l'autre branche) dans le SERVIR. Ce choix de répartition architecturale (figure 1) repose sur la constatation que de nombreux processus applicatifs présentent une façade indépendante de l'interface utilisateur. Cette façade peut s'assimiler au *modèle conceptuel* de l'application, c'est-à-dire la collection des objets et opérations ayant un sens pour l'utilisateur.

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) 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 une représentation graphique, ce graphe utilise par exemple une forme standardisée comme SVG (Scalable Vector Graphics). 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 SERVIR, qui doit choisir la feuille la plus adaptée à la plateforme et au contexte de l'interaction ; elles peuvent être complétées par des feuilles de styles secondaires propres à la plateforme ou à l'utilisateur, 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 correspond à un collecticiel. Si l'on utilise un seul SERVO, l'architecture est centralisée, avec les problèmes qu'on lui connait. Rien n'empêche en principe 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 en détail l'architecture et sa mise en œuvre : SERVO, SERVIR, et concrétisation.

SERVEUR D'OBJETS

Le serveur d'objets conceptuels est un transiciel type : il repose en général sur un modèle de données persistant, il prend en charge des opérations de transformation de ces données et les présente à un ou plusieurs serveurs d'interaction à travers un système transactionnel permettant de gérer les accès multiples.

Description du modèle conceptuel

Le créateur d'une application INDIGO décrit son modèle conceptuel dans le langage de son choix (pour l'instant, Java ou C#), ou au moyen d'un schéma XML. Afin d'assu-

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

rer 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) [6]. Cette approche nous permet d'utiliser les techniques d'introspection des langages actuels (tels que les JavaBeans) 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, suivant par exemple le guide de style des "cleanbeans" [11]. Les classes peuvent correspondre directement à des classes de l'application, ou bien jouer le rôle de simples façades adaptées à la communication avec l'interface. 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, et lui adjoint une correspondance avec les feuilles de style primaires décrivant le processus de concrétisation pour les SERVIR visés.

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 et utilise SOAP [6], ce qui permet au protocole INDIGO de fonctionner sur le réseau Internet global, par delà les passerelles coupe-feu. Outre les appels d'introspection, ce service expose trois fonctions :

- **get** retourne la structure de donnée (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 de la structure. Si un numéro d'historique (voir ci-après) est passé en paramètre, alors c'est une différence entre un l'état antérieur correspondant 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 ou 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.

La fonction *listen* est nécessaire pour mettre en œuvre avec SOAP un mécanisme de notification à l'initiative du SERVO. En effet, seul le client SOAP (ici le SERVIR) peut appeler des fonctions du serveur. 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.

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 doivent 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 : 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 donnée répliquée (SCOG). Le choix de la méthode de restauration en cas d'échec (optimiste ou conservatrice) est laissé au SERVIR.

Afin de limiter 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. 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 [1]. Ces parties seront chargées à la demande, si la feuille de style en a besoin lors de la concrétisation.

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 plateforme 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).

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. Nous présentons ici le modèle graphique retenu pour un SERVIR graphique générique, basé sur le standard SVG pour le rendu graphique et les machines à états hiérarchiques pour l'interaction.

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 auxquels 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 développées ces dernières années.

Nous avons développé un SERVIR dont le rendu graphique est basé sur SVG, un format du W3C décrivant des scènes graphiques structurées en deux dimensions. 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 utilisants des outils tels que Adobe Illustrator [7]. 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.

Le moteur de rendu Sauvage

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. Notre moteur, Sauvage, est constitué de quatre composants spécialisés dans un aspect particulier du processus de rendu et d'interaction :

- 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 donnée qui sert à l'affichage de la scène graphique ;
- une structure de données pour l'*élagage* des formes non visibles ;
- une structure de données pour la gestion de la *taille apparente* des objets.

Le *graphe d'affichage* sert exclusivement à l'affichage et au *picking* [3]. 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>` peut être partagé par ses références multiples car il ne dépend pas des parents.

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 cliquer-tirer est rapide car il engendre très peu de mises à jour dans la structure interne. De même le pan-and-zoom n'engendre que des modifications incrémentales qui permettent un calcul rapide des formes à élaguer ou à rendre visible.

La gestion de la *taille apparente* permet de choisir une représentation différente selon la taille affichée. Par exemple, la lisibilité du texte est améliorée en adaptant automatiquement la taille de la police en fonction de sa taille apparente, et les artefacts de rendu d'OpenGL sont évités en adaptant la géométrie à la taille apparente. Cette structure permet également de gérer une technique d'optimisation qui consiste à réutiliser l'image du résultat d'un rendu partiel dans une texture.

Les performances du moteur de rendu sont compatibles avec l'interaction : même sur des scènes complexes, le taux de rafraîchissement dépasse les 20Hz 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.

Interaction

La gestion de l'interaction 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 [5] développée antérieurement et que nous avons modifié 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 les périphériques USB conformes à la norme HID (selon la plate-forme). Les périphériques logiques permettent un premier niveau d'adaptabilité. Ainsi, on peut créer un périphérique logique permettant de 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* [9] : les machines à états hiérarchiques. Les comportements

```
hsm Button {
   hsm Disarmed {
      hsm OutUp {
         - enter() > InUp
      }
      hsm InUp {
         - leave() > OutUp
         - push()  > Armed::InDown
      }
      hsm OutDown {
         - enter() > Armed::InDown
         - pop()   > OutUp
      }
   }
   hsm Armed {
      enter { /* armed feedback */ }
      leave { /* disarmed feedback */ }
      hsm InDown {
         - leave() > Disarmed::OutDown
         - pop() broadcast(DO_IT) > Disarmed::InUp
      }
   }
}
```

Figure 2 : Le comportement d'un bouton (la gestion du *feed-back* est omise pour des raisons de place)

dynamiques pilotés par des évènements deviennent ainsi des objets à part entière du langage de programmation [5]. Cette structure de contrôle est utilisée pour créer des périphériques logiques à partir de périphériques physiques et pour réaliser les comportements associés aux objets graphiques. La figure 2 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.

Le chargement dynamique de comportements par HsmTk permet de factoriser et de réutiliser des comportements entre applications. 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 : interaction bimanuelle sur une tablette graphique, outils transparents [4], etc. Il permet par ailleurs d'étendre facilement l'ensemble des techniques d'interaction à disposition du SERVIR.

Ces mécanismes mettent en œuvre une certaine plasticité des interfaces [15] : l'application ou l'utilisateur peuvent adapter les techniques d'interaction utilisées à la plateforme 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 "à travers" de la main dominante.

Les comportements associés aux objets graphiques spécifient les manipulations qu'ils acceptent de la part des périphériques logiques. Le bouton de la figure 2 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 par une fonction de *picking* typée de trouver la cible compatible avec la manipulation en cours parmi les objets présents sous le curseur.

CONCRETISATION

Le mécanisme de concrétisation transforme l'arbre XML des objets abstraits (le SCOG) en un document concret présenté à l'utilisateur (le POG). Notre implémentation pour le SERVIR graphique utilise une feuille de style XSL qui produit un document SVG annoté, comme on l'a vu ci-dessus, par les comportements des objets graphiques. Ces annotations incluent également les informations nécessaires pour établir un lien bidirectionnel entre les objets conceptuels et leurs représentations (figure 3).

```
<getReturn type="Volume">
   <Version type="int">0</Version>
   <Directory type="string">~/test</Directory>

   <Folder type="Folder">
      <Name type="string">~/test</Name>
      <Entry type="File">
         <Name type="string">README</Name>
      </Entry>
      <Entry type="File">
         <Name type="string">INSTALL</Name>
      </Entry>
      ...
```

```
<g behaviour="node" path="/" id="d0e7">
   <use xlink:href="#closed"/>
   <text path="/Name" id="d0e150" x="22" y="12">~/test</text>
   <g transform="translate(16,16)">
      <g behaviour="leaf" path="/Entry[0]" id="d0e12">
         <text path="/Entry[0]/Name" id="d0e17" x="22" y="12">README</text>
      </g>
      <g behaviour="leaf" path="/Entry[1]" id="d0e24">
         <text path="/Entry[1]/Name" id="d0e29" x="22" y="12">INSTALL</text>
      </g>
      ...
```

Figure 3 : COG (XML en haut) et POG (SVG en bas) simplifiés représentant un système de fichier

Chaque objet conceptuel est transformé en un groupe SVG auquel est associé un identifiant. Le moteur de concrétisation maintient une table qui permet de retrouver toutes les images d'un objet particulier 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. Dans sa version actuelle, ce mécanisme requiert que la transformation de tout sous-arbre du COG produise un sous-arbre unique du POG. Par ailleurs, chaque groupe SVG contient un attribut identifiant l'objet qu'il représente sous la forme d'un chemin

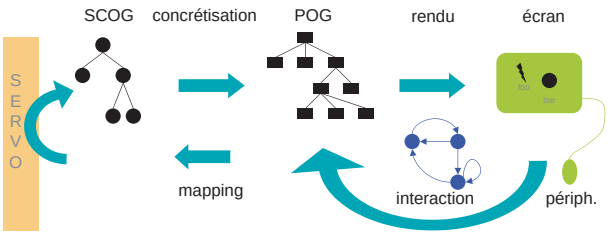

Figure 4 : Fonctionnement général

XPATH dans le COG. 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. 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. La figure 4 illustre ce fonctionnement général.

EXEMPLES D'APPLICATIONS

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

Explorateur de Fichiers

Le premier exemple est un explorateur de fichier dont un fragment est montré figure 5 à gauche. Celui-ci permet quelques 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 chargeant l'arborescence qu'au fur et à mesure de son exploration, ce qui permet une utilisation interactive y compris au travers d'un réseau non local. Une version non graphique de l'explorateur a également été développée : le POG généré est un arbre HTML qui s'affiche dans un navigateur Web standard.

Jeu Multi-Joueur : Puissance 4

La figure 5 à droite 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. 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.

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 6 est tirée de la version INDIGO).

Glance représente notamment la trajectoire de chaque

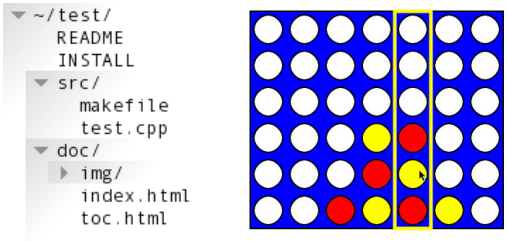

Figure 5 : Explorateur de fichiers et Puissance 4

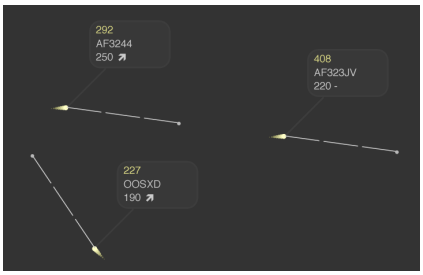

Figure 6 : Vue radar

avion en utilisant cinq cercles situés aux positions passées de l'avion, le rayon de chaque cercle étant inversement proportionnel à l'âge de la position. 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.

L'application originale était monolithique, optimisée pour des type d'affichage et d'interaction spécialisés et adaptés au contexte de l'application : utilisation du clipping et de la transparence pour afficher les informations de vol, et interactions post-WIMP de type pan-and-zoom. Le portage de Glance sur INDIGO offre des performances 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 goulet 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 sommes en train d'ajouter une interaction basée sur une toolglass. 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, ce dernier parcourt la pile d'objets graphiques se situant sous le clic, trouve le premier objet 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 celle-ci, la méthode associée à l'interaction 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 pour être interprété par le SERVO, sinon les appels de méthodes à la fin des interactions. 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.

COMPARAISON AVEC L'ETAT 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, nous appellons "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.

Le modèle VNC [14] 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 lors de la navigation pan-and-zoom. Pour un écran haute définition de 3200x2400 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-lâcher" 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 [12] propose 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.

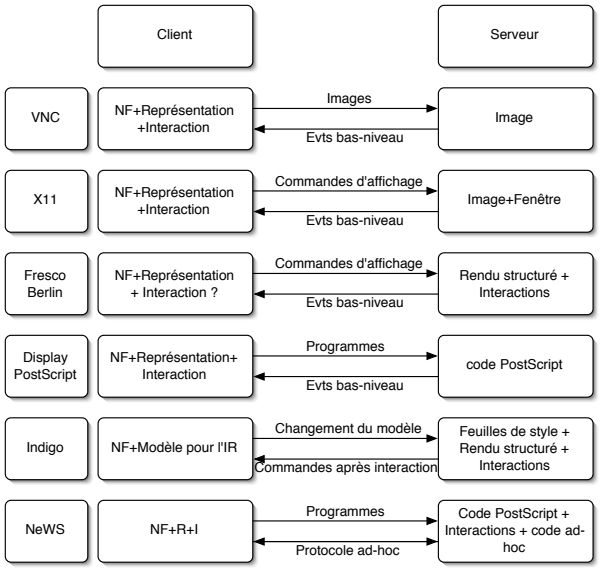

Figure 7 : Comparaison des modèles Client-Serveur

Fresco/Berlin [10] 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 le 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 les notions de modèle d'interaction et de modèle de représentation d'INDIGO. De plus, le choix de CORBA comme intergiciel est contestable car il pénalise les temps de réponse interactifs. Enfin ce projet semble actuellement arrêté.

Display PostScript (DPS) [2], originellement développé pour le NeXT, s'appuie sur le modèle graphique de 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.

NeWS [8], développé par Sun et contemporain de DPS, est un système d'affichage et d'interaction également basé sur 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 communications 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 7 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 qu'INDIGO n'utilise pas un langage Turing-complet comme PostScript, nous considérons que le niveau d'abstraction du protocole de communication est comparable à ce que ces systèmes cherchaient à réaliser.

INDIGO peut également être comparée aux applications Web telles que GMail de Google qui utilisent la possibilité récente des navigateurs Web 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. De plus, l'interaction reste limitée aux modes d'interaction d'un navigateur Web : traversée de lien et remplissage de formulaires.

CONCLUSION ET PERSPECTIVES

Nous avons présenté une architecture répartie destinée au développement d'applications interactives caractérisées par les propriétés suivantes : prise en compte de plateformes 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 difficile 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* [13]. 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 à terme si l'on veut s'affranchir des limitations des environnements actuels.

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.

BIBLIOGRAPHIE

1. S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for Active XML. In *Proceedings of SIGMOD'04*, pages 227–238. ACM Press, 2004.
2. Adobe Systems Inc. *Programming the Display PostScript System with X*. Addison-Wesley Longman Publishing Co., Inc., 1993.
3. M. Beaudouin-Lafon and H. M. Lassen. The architecture and implementation of CPN2000, a post-WIMP graphical application. In *Proceedings of UIST'00*, pages 181–190. ACM Press, 2000.
4. E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. In *Proceedings of SIGGRAPH'93*, pages 73–80. ACM Press, 1993.
5. R. Blanch. Programmer l'interaction avec des machines à états hiérarchiques. In *Actes IHM'02*, pages 129–136, 2002.
6. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. Technical report, W3C, May 2000.
7. S. Chatty, S. Sire, J.-L. Vinot, P. Lecoanet, A. Lemort, and C. Mertz. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of UIST'04*, pages 267–276. ACM Press, 2004.
8. J. Gosling, D. S. H. Rosenthal, and M. J. Arden. *The NeWS book: an introduction to the network/extensible window system*. Springer-Verlag New York, Inc., 1989.
9. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. T. Hunger. Nieder mit X-Mauer! *Linux Magazine*, December 2000. English translation available in Linux Magazine UK, Jan 2001. See also http://www.fresco.org/architecture.html.
11. P. Kaplan. Make a sweep with clean beans. *JavaWorld*, (11), 1999.
12. A. Nye. *XLIB Programming Manual and Reference Manual*. O'Reilly & Associates, Inc., 1988.
13. J. Rekimoto. Pick-and-drop: a direct manipulation technique for multiple computer environments. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 31–39, New York, NY, USA, 1997. ACM Press.
14. T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2.1:33–38, Jan/Feb 1998.
15. D. Thevenin, G. Calvary, and J. Coutaz. *A Reference Framework for the Development of Plastic User Interfaces*. John Wiley & Son, Ltd, 2003.