

Factorisons la gestion des événements des applications interactives !

Stéphane Conversy, Paul Janecek et Nicolas Roussel

Laboratoire de Recherche en Informatique
UMR 8623 CNRS - Université Paris-Sud
LRI - Bât. 490 - Université Paris-Sud
91405 Orsay Cedex, France

{*conversy, pjanecek, roussel*}@lri.fr

INTRODUCTION

Notre travail de recherche en Interaction Homme Machine nous conduit à imaginer sans cesse différentes formes d'interaction entre l'homme et la machine ou entre les hommes via les machines. Les boîtes à outils disponibles pour la construction d'interfaces sont généralement basées sur les primitives graphiques des systèmes de fenêtrage sous-jacents (X, Macintosh ou Windows). Ces primitives, bien qu'elles suffisent à la grande majorité des applications que nous utilisons tous les jours, freinent le développement de certaines techniques: peu ou pas de support pour le son, la transparence, des vues 3D ou zoomables, des périphériques d'entrée/sortie autre que le clavier ou la souris.

Afin de pouvoir implémenter et évaluer les formes d'interaction qui nous intéresse, nous sommes donc contraints à faire cohabiter plusieurs composants: un premier pour les éléments classiques d'interface graphique (e.g. Motif), un deuxième pour les éléments plus spécifiques (e.g. OpenGL), un troisième pour la communication avec des périphériques (e.g. MIDI), un autre pour la communication avec d'autres applications par le réseau (e.g. TCP).

Dans la section suivante, nous montrons que le concept d'événement est à la base des composants actuels et nous dégageons les éléments essentiels à un système unifié de gestion d'événements. Nous examinons ensuite plusieurs politiques d'utilisation d'un de ces éléments puis nous montrons l'intérêt d'une conception de l'application basée sur des événements de différents niveaux.

FACTORISATION DE LA GESTION DES EVENEMENTS

Les composants actuels imposent un style de programmation articulé autour d'une boucle de traitement d'*événements*. Cette boucle de traitement ainsi que les événements peuvent apparaître explicitement dans le programme (e.g. X, Mac ToolBox), ou implicitement au moyen de callbacks (e.g. Xt, Motif, Tk).

Les concepteurs de ces composants ont généralement prévu de pouvoir les étendre: on peut ainsi, par exemple, définir de nouvelles sources d'événements au sein d'une application utilisant Tk. Cependant, cette approche impose le choix d'un des composants comme base de l'application. Dans certains cas, ce choix peut être difficile. Un éditeur partagé est-il d'abord une application graphique, ou une application partagée ? Dans tous les cas, le changement du composant de base impose une refonte totale de l'application.

La minimisation du coût des changements de composants passe par l'utilisation d'un noyau stable et spécifique de gestion d'événements. Ce noyau doit permettre d'intégrer les composants existants et de décrire de manière homogène les événements et mécanismes de traitement associés de chacun d'entre eux. Ainsi, une interaction pourra être définie comme une composition d'événements provenant de différentes sources: le tracé d'un rectangle peut se faire en positionnant un coin avec la souris et en changeant sa taille avec un contrôleur MIDI.

Nous définissons quatre éléments de base pour la conception d'un noyau de gestion d'événements:

- l'événement, caractérisé par un type et des attributs;
- l'émetteur (ou *source*), qui émet des événements;
- le récepteur, qui s'abonne à des événements et en reçoit;
- l'aiguilleur (ou *dispatcher*), qui transmet les événements d'un émetteur à plusieurs récepteurs.

Si les trois premiers éléments semblent inévitables, nous allons voir dans la section suivante que les possibilités offertes par le noyau varient suivant le nombre d'aiguilleurs que l'on autorise.

DE L'IMPORTANCE DU NOMBRE D'AIGUILLEURS

Nous allons successivement envisager trois cas: aucun, un ou un nombre quelconque d'aiguilleurs. Pour chaque cas, nous donnerons un exemple de fonctionnalité ou de limite.

Aucun aiguilleur

Dans cette situation, un événement ne sera transmis que si le développeur a explicitement connecté l'émetteur à un récepteur. Cette approche est utilisée par des boîtes à outils comme Qt, AWT 1.1.5 ou par les callbacks de Tk ou Motif:

```
# Crée deux émetteurs
button Ok ;
button Cancel ;

# Crée deux récepteurs
fonction deleteProc(event e) { delete e.filename }
fonction destroyWindow(event e) { ... }

# Connecte chaque émetteur à un récepteur
Ok.bind(deleteProc) ;
Cancel.bind(destroyWindow) ;
```

L'inconvénient de cette approche est qu'on ne peut pas ajouter de façon transparente un récepteur ou un émetteur à une application existante: pour ajouter une notification sonore

dans le cas de la suppression de fichier de l'exemple précédent, il faut pouvoir modifier le récepteur *deleteProc*. Ce type de modification n'est pas toujours possible, certains récepteurs pouvant faire partie de librairie externes à l'application.

Un aiguilleur unique (et implicite)

L'intérêt d'un aiguilleur unique réside dans le fait que les émetteurs et les récepteurs ne sont pas explicitement connectés: tous les événements sont émis vers l'aiguilleur. Les récepteurs doivent s'abonner aux types d'événement qui les intéressent auprès de l'aiguilleur:

```
# Abonne le récepteur deleteProc à l'événement 'Delete'
bind('Delete', deleteProc)
...
# Dans un émetteur : génère l'événement Delete
generate('Delete', {filename:'core'})
```

Dans ce cas, on peut aisément ajouter un émetteur ou un récepteur : il suffit de connaître le type d'événement à émettre ou à traiter. L'inconvénient de cette approche est qu'elle va conduire à mélanger des événements abstraits comme 'Delete', qui peut être émis par n'importe quelle source, et des événements beaucoup plus pratiques comme 'option 3 sélectionnée dans la boîte de dialogue 4', qui ne concerne qu'une toute petite partie de l'application.

Plusieurs aiguilleurs

Dans le cas d'une application à vues multiples, un événement stipulant un changement va provoquer le rafraîchissement de toutes les vues. Une politique spéciale de distribution pourrait être mise en place pour assurer un temps de réponse optimal pour la vue où a lieu l'interaction: les autres vues peuvent se contenter d'un résumé généré par l'aiguilleur. Une telle politique peut être appliquée par un dispatcher local, spécifique aux événements générés par les vues multiples. Cette approche permet d'éviter la pollution de l'espace des événements évoquée précédemment.

Nous pensons qu'aucune des trois solutions présentées ne convient à elle seule. Nous nous orientons vers une solution hybride associant un aiguilleur implicite avec les possibilités d'ajout d'aiguilleurs locaux et de liens explicites entre émetteur et récepteur.

OBSERVABILITE

Afin de rendre l'application totalement indépendante des composants, les interactions doivent être décrites en termes d'événements et de services de haut niveau: lors d'une sélection multiple de fichiers par un rectangle élastique dans le finder, les événements pertinents pour l'application sont l'ajout ou la suppression d'un fichier dans la sélection, et non le déplacement de la souris. Une application peut donc être considérée comme un ensemble de modules liés par des mécanismes de traduction et de composition d'événements. Une exécution de cette application correspond au flux des événements générés.

L'utilisation d'événements dans toutes les couches de l'application permet de rendre celle-ci *observable* avec différentes granularités. Nous définissons l'observabilité comme la capacité pour un module de l'application de connaître les activités en cours: un module qui génère des événements est observable par ceux qui les reçoivent. Voici quelques exemples du bénéfice

apporté par cette propriété :

- la synchronisation de vues multiples grâce à l'observabilité de chacune d'elles ;
- l'ajout ou le retrait de modalités de notification : par exemple, utiliser la modalité sonore dans une application graphique, sans toucher au code existant ;
- la réplication de l'application: en choisissant la granularité des événements transmis sur le réseau, on peut implémenter différents niveaux allant de la réplication stricte à l'écho sémantique [1] ;
- l'enregistrement et le rejeu de l'application.

La notion d'observabilité est fortement liée au nombre d'aiguilleurs : elle est pratiquement nulle dans le cas d'un système sans aiguilleur, maximale dans le cas d'un aiguilleur unique et choisie dans le cas d'aiguilleurs multiples.

CONCLUSION

La notion d'*événement* est une notion fondamentale des applications interactives et se retrouve naturellement dans chaque type de composant évoqué. Malheureusement, chaque composant propose sa propre vision de cette notion, qui est difficilement combinable avec d'autres. Il nous paraît intéressant de "factoriser" la notion commune d'événement afin de pouvoir articuler les applications autour d'une gestion unifiée. Nous pensons qu'un tel système peut être utilisé pour décrire des interactions en termes de traduction et de composition d'événements de différents niveaux, allant du niveau physique lié aux périphériques jusqu'à des événements abstraits liés à l'application. Nous pensons enfin que l'utilisation de ces événements abstraits offre quelque éléments de réponse concernant des problèmes comme la synchronisation de vues multiples sur des données, la notification, la réplication d'exécution ou l'enregistrement et le rejeu.

Ces premières réflexions nous conduisent à plusieurs questions : est-il toujours possible de décrire une application en termes d'événements ? à quel point cette approche est-elle coûteuse ? le noyau doit-il permettre l'émission et la réception synchrone d'événements ? Nous avons implémenté plusieurs noyaux de gestion d'événements décrits dans cet article en encapsulant chaque composant dans un thread. Partant d'une telle approche, on peut également se demander à quel niveau doivent se placer les threads.

En résumé, notre position est la suivante : factorisons la gestion des événements des applications interactives !

REFERENCES

- [1] M. Beaudouin-Lafon and A. Karsenty. **Transparency and awareness in a real-time groupware system**. In *Proc. ACM Symposium on User Interface Software and Technology, UIST'92, Monterey (USA)*, pages 171-180. ACM Press, November 1992. Également Rapport de Recherche LRI 704, octobre 1991. □
- [2] M. Green. **A survey of three dialogue models**. *ACM Trans. on Graphics*, 5(3):244-275, July 1986. □
- [3] R. D. Hill. **Event-Response Systems: A Technique for Specifying Multi-Threaded Dialogues** . In *Proc. ACM CHI+GI*, pages 241-248, 1987. □