Utilisation d'AJAX et SVG pour la visualisation d'un flux de données

Christine Potier

2008D010

Juillet 2008

Département Informatique et Réseaux
Groupe IC2 : Interaction, Cognition et Complexité

Utilisation d'AJAX et SVG pour la visualisation d'un flux de données

Christine Potier

Département Informatique et Réseaux
Institut TELECOM, TELECOM Paristech

1. Introduction

L'objectif de cette étude était de tester une technologie permettant de visualiser graphiquement et en temps réel un flux de données à travers une interface web. Le but de cette visualisation est par exemple de faire de la supervision, d'offrir un service à des clients d'une entreprise et ceci à travers des outils qui leur sont familiers.

Une des contraintes de l'application est donc de permettre l'affichage des données sans obligation d'installer un programme sur chaque poste. C'est pourquoi nous avons fait le choix de la visualisation dans un navigateur web et sans téléchargement de plug-in.

Une autre contrainte est de réaliser un affichage *« au vol »* à partir de données envoyées par un *DSMS* (*Data Stream Management System*). Le DSMS reçoit en entrée des flux de données, effectue éventuellement des calculs et renvoie en sortie *des tuples au format XML* (para. 7).

Pour satisfaire ces contraintes deux choix étaient possibles :

La technologie *Flash* avec le langage ActionScript et éventuellement en utilisant Flex (bibliothèque d'objets graphiques). Le Framework AS3 (ActionScript CS3) contient des classes qui permettent d'appeler des services *via le protocole http*. Flash est un logiciel propriétaire. Le player est gratuit et disponible sur toutes les plate-formes.

Le langage *SVG* (Scalable Vector Graphics) avec interaction en javascript. Dans ce cas, on utilise *AJAX (Asynchronous JavaScript and XML)* pour la connexion à un service *via le protocole http*. Le modèle géométrique de SVG est plus complet que celui de Flash. SVG est une approche portée par le W3C – organisme de normalisation du Web – centré sur XML. C'est une solution ouverte, basée sur des standards partagés et libres de droit.

C'est cette dernière solution, SVG + AJAX, que nous avons choisi afin d'en évaluer les performances.

Après avoir abordé les différentes technologies utilisées pour réaliser cette application : SVG puis Ajax et l'utilisation de l'interface de programmation DOM (Document Object Model) de SVG, nous présenterons l'architecture globale de l'application. Ensuite nous décrirons l'application complète de visualisation d'un flux de données issues de capteurs.

2. SVG Scalable Vector Graphics

SVG est une recommandation du W3C [4] , [5] qui spécifie un langage de description des graphiques 2D en XML. Il supporte trois types d'objets graphiques :

- des formes vectorielles (rectangle, cercle, ellipse, polygone,..),
- des images bitmaps (GIF, JPEG, PNG),
- du texte.

Le ***DOM*** *(Document Objet Model)* de XML définit une méthode standard pour accéder et manipuler des documents XML. Le DOM permet de présenter un document XML sous forme d'arbre avec des éléments, des attributs et du texte. La **description XML, donc textuelle** des graphiques SVG permet entre autres d'accéder par mot-clés aux balises et aux attributs des graphiques. Les balises SVG sont donc considérées comme des objets et peuvent recevoir des propriétés ou des attributs de style. De plus ils peuvent être indexés par des moteurs de recherche.

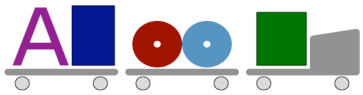
SVG offre la possibilité de définir des couches d'objets graphiques c'est-à-dire de regrouper sous une même entité des éléments auxquels on souhaite appliquer le même traitement (de style, de visiblité, etc). On obtient ainsi une représentation arborescente des graphiques (l'arbre DOM) où chaque élément peut être identifié grâce à son nom ou à son type.

Puisqu'un document SVG est une description textuelle du graphique, il peut être facilement généré par programme.

Les graphiques SVG sont interactifs et dynamiques. L'animation ou les transformations peuvent être définies à l'intérieur des fichiers SVG (balise <animate>) ou à l'aide de scripts externes. La conformité à XML permet d'accéder à l'interface de programmation **DOM de SVG** à partir de nombreux langages de script dont *Javascript*.

SVG reconnaît trois types d'événements : les événements souris, les événements du clavier et les changements d'état comme l'état du chargement et de l'affichage du fichier SVG. Dans cette application, nous utilisons principalement l'événement « `onload` » (chargement de la page).

Sur la fig.1, on peut voir un graphique SVG avec sur la partie droite de la figure l'arbre DOM et sur la partie gauche un exemple des fonctions de l'API du DOM en javascript qui nous permettent d'accéder aux éléments du graphique et d'interagir en fonction des événements reçus.



• la racine du document

```
var SVGdoc = svgDocument;
```

• accéder à l'élément « train »

```
var train =
SVGdoc.getElementById("train");
```

ou si on part de wagon1

```
var train = wagon1.parentNode;
```

• atteindre les composants de wagon1 (tableau de composants)

```
var composants =
        wagon1.childNodes;
```

```
<svg>
   <defs>
      |    <g id="essieux"> …..
   <g id="train">
         <g id="loco">
            <use href="essieux">
            <polyline>  // cabine
            <rect>      // caisse verte
         <g id="wagon1">
            <use href="essieux">
            <circle>    // bobine rouge
            <circle>    // bobine bleue
         <g id="wagon2">
            <use href="essieux">
            <text>      // grande lettre
            <rect>      // caisse bleue
```

Figure 1 : Arbre DOM d'un graphique SVG

3 . AJAX (Asynchronous JavaScript and XML)

Pour faire des pages dynamiques, il existe deux méthodes :

1) La création de pages dynamiques se fait côté serveur. On envoie une requête au serveur qui génère une page html complète en exécutant par exemple un script php et renvoie cette page au client. Dans ce cas, toute la charge de traitement est sur le serveur.

2) L'utilisation d'*Ajax*, nom donné à un ensemble de classes *javaScript* apparues en 2005 [1] dont la principale est la classe `XMLHTTPRequest`. Le but d'Ajax est de répartir les ressources entre le client et le serveur sans surcharger le réseau. Ajax permet d'effectuer en JavaScript des *traitements sur le poste client* à partir *d'informations prises sur le serveur*. Ajax conjointement à l'utilisation du DOM de XML permet de *modifier partiellement* une page affichée dans un navigateur en fonction des données transmises par le serveur sans avoir besoin de recharger complètement cette page. Les informations qui transitent sur le réseau sont moindres et il y a moins de traitement fait sur le serveur (fig. 2).

Suite à un événement intervenu sur la page affichée sur le client (survol de souris, clic, chargement, etc) une *requête Ajax* est faite au *serveur via le protocole http*. Le serveur envoie alors *la réponse* au format texte ou au format XML et cette

réponse est *traitée par le client*. La zone de la page html dans laquelle il doit y avoir des modifications en fonction de la réponse du client est repérée par son identifiant. C'est cette zone et non pas toute la page qui sera modifiée après une requête Ajax.

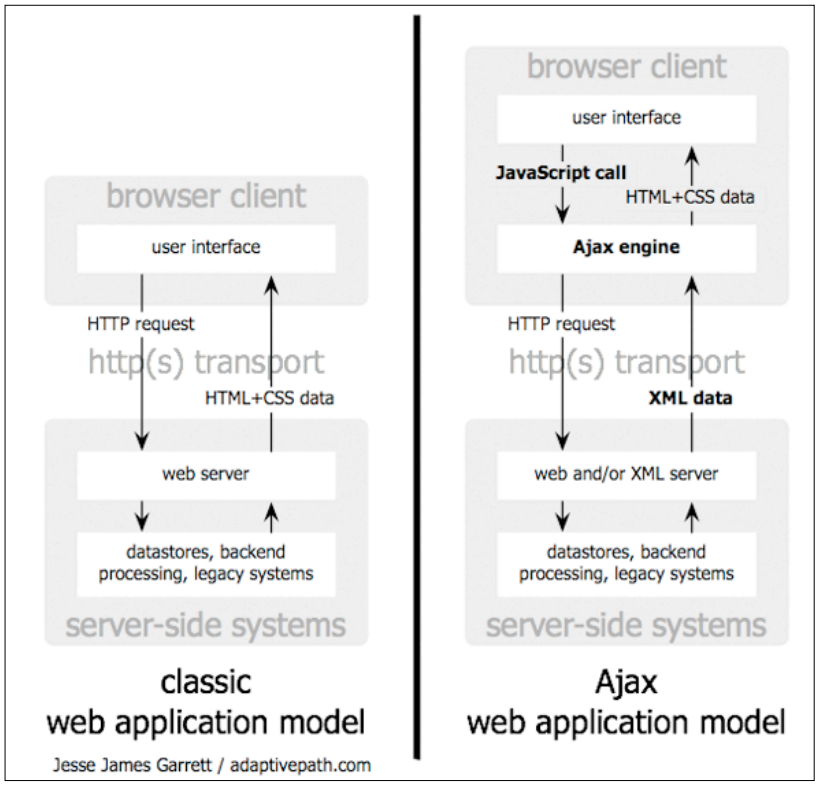


Figure 2 : Schéma extrait de l'article de Jesse James Garrett [1].

Côté client, Ajax utilise JavaScript pour la connexion au serveur et les traitements locaux. Les fonctions JavaScript communiquent avec le serveur par l'objet `XMLHTTPRequest`. Cet objet permet de *lire* des données sur le serveur de façon *asynchrone*. Le terme asynchrone signifie que l'exécution du JavaScript sur le client continue sans attendre la réponse complète du serveur. Celle-ci pourra être traitée au fur et à mesure de son arrivée (cf. Annexe l'API `XMLHTTPRequest`).

JavaScript utilise un modèle de ***programmation qui traite les événements***. Un événement est une action de l'utilisateur (chargement de la page `onload`, clic sur la souris `onclick`, survol `onMouseOver`, , …) qui provoque la création d'un objet `XMLHTTPRequest` pour établir la connexion avec le serveur et un appel de la

fonction « *écouteur* » associée à l'*élément* de la page *source de l'événement*. Cette fonction permet de mettre en forme le graphique en fonction de la réponse du serveur.

L'élément source de l'événement est identifié grâce à l'interface de programmation du *DOM*. De même, on utilisera cette interface pour accéder aux noeuds de l'arbre du fichier XML venant du serveur ainsi qu'aux éléments graphiques de l'arbre SVG (voir Annexe B).

4. Remarques sur le fonctionnement d'Ajax

Notre but étant d'afficher un flux de données donc de recevoir des données sous forme de tuple « en permanence », la question se pose de savoir si l'on reste connecté au serveur en permanence ou non.

Pendant l'exécution de la commande, le résultat de la requête faite au serveur est mise dans un chaîne de caractères (*buffer),* cette chaîne de caractères étant au format texte ou au format XML (voir Annexe A):

a) réponse au format texte: il est alors possible de traiter la réponse dès que l'on a commencé à la recevoir. La connexion peut être ouverte une fois pour toutes et on traite les données au fur et à mesure. Dans notre problématique, ce serait la solution évidente même si c'est contraire à la philosophie du web .

Nous avons testé cette solution pour en connaître les limites. Or elle s'avère techniquement risquée. En effet, dès que la connexion est établie, le serveur met les données dans le *buffer de réponse*. Le nombre de caractères de la réponse peut devenir très grand. La taille limite du buffer n'est pas normalisée et il est difficile de prévoir d'éventuels débordements. D'autre part, le traitement des données côté client peut devenir très compliqué et très long: Comment repérer les dernières données fournies par le serveur dans une chaîne de caractères de grande dimension?

Des tests ont été faits en laissant tourner un programme afin de voir la limite de la taille du buffer. Le buffer a atteint plusieurs **millions de caractères** sans problème apparent. Est-ce que la taille du buffer dépend de la mémoire de l'ordinateur sur lequel tourne le client ?

Par ailleurs, il n'est pas possible de forcer la remise à zéro du buffer contenant la réponse sans couper la connexion.

b) réponse au format XML : il faut dans ce cas attendre que la réponse soit intégralement reçue avant de pouvoir la traiter. En utilisant ce format de réponse, on ne peut donc pas faire du *streaming*, c'est-à-dire commencer à afficher quelque chose avant d'avoir reçu l'ensemble des données. On doit établir la connexion puis la refermer pour obtenir la réponse par morceaux. Il faut alors régler la fréquence de connexion avec la fréquence d'arrivée des tuples.

Si la réponse du serveur est au format XML, on peut utiliser les fonctions du *DOM pour analyser la réponse*. Ce qui représente un avantage certain !

Nous avons retenu cette solution pour l'application de visualisation d'un flux de données issues d'un DSMS. La connexion est ouverte et fermée régulièrement. La réponse à la *requête* est envoyée *au format XML* afin de bénéficier des facilités offertes par le DOM. Le problème est donc de régler la fréquence de connexion par rapport à la fréquence d'apparition des données sur le flux, afin de ne pas perdre de valeurs ni d'encombrer le réseau en faisant des requêtes trop fréquentes.

5. Intégration de SVG et Ajax dans une page html

a) Intégration de SVG dans une page html

Un navigateur web standard (FireFox ou Opera par exemple) permet d'afficher un document purement SVG mais il est aussi possible d'intégrer du code SVG dans une page html, soit directement soit en utilisant un fichier externe. Pour utiliser un fichier externe, cela se fait avec la balise `embed` :

```
<embed src="xxx.svg" width=".." height=".." type="image/svg+xml" />
```

ou la balise `object` (recommandée par le W3C) :

```
<object data="xxx.svg" style="width:..;height:.. ;" type="image/svg+xml" />
```

La figure 3 montre un exemple de page HTML dans laquelle les tracés sont faits en SVG après interrogation d'une base de données.

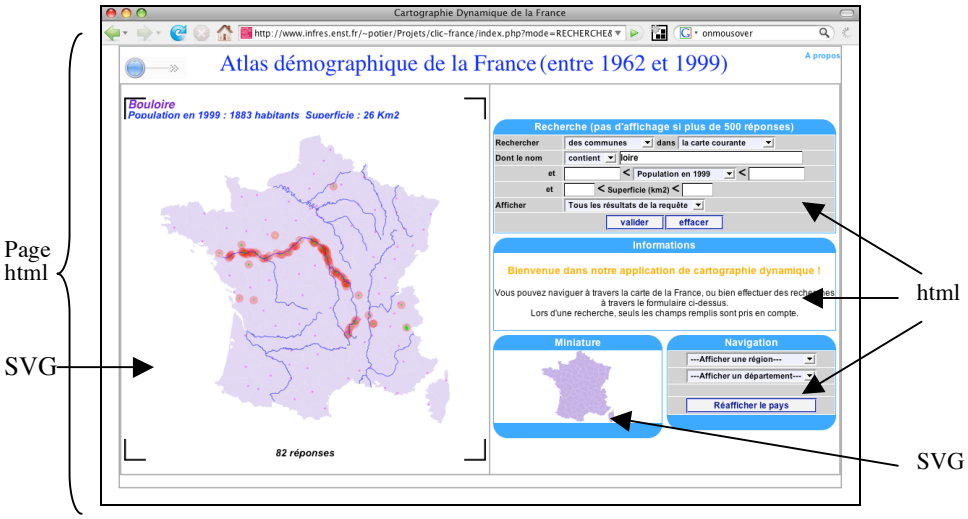


Figure 3 : exemple d'intégration de SVG dans une page HTML

b) Intégration d'AJAX dans du code SVG

Pour interagir avec les éléments graphiques SVG à partir de javascript, il faut intégrer le code javascript dans le fichier SVG, soit en mettant du code javascript dans le code SVG, soit en faisant référence à un fichier javascript externe. Cette solution est recommandée car elle permet de réutiliser facilement du code développé précédemment. Dans le fichier SVG, on indique alors le ou les fichiers contenant les scripts ainsi que les fonctions à invoquer en cas d'événement. Dans notre cas, le fichier SVG commence par :

```
<svg width="100%" height="100%"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
            onload="init(evt)">

<script type="text/ecmascript" xlink:href="ajaxRequest.js"/>

<script type="text/ecmascript"
            xlink:href="realTimeChart.js"/>

. . .
```

Au chargement du graphique SVG, la fonction `init(evt)` sera exécutée. Elle doit donc être définie dans un des deux fichiers javascript. Le premier programme `ajaxRequest.js` permet d'établir la connexion avec le serveur. Le second `realTimeChart.js` contient la fonction d'initialisation et les fonctions de traitement des données renvoyées par le serveur pour affichage.

6. Architecture logicielle de l'application

L'architecture complète de l'application est décrite dans le schéma de la figure 4.

Le DSMS reçoit en entrée des flux de données, effectue éventuellement des calculs (agrégats, moyennes, etc) et renvoie en sortie sur un serveur http *des tuples sous forme de texte balisé*. Ces tuples sont interceptés par un script php, `recoitFlux.php` qui les mémorise sur le serveur dans un *fichier temporaire au format XML*. Ce fichier temporaire ne contient pas l'ensemble des tuples envoyés par le DSMS depuis la mise en service. Les données sont écrasées au fur et à mesure et on verra au paragraphe suivant les tests qui ont été faits pour savoir si le fichier XML devait contenir un seul tuple ou plusieurs tuples et dans ce cas combien, le but étant de ne pas perdre de tuples mais aussi de ne pas en mémoriser inutilement.

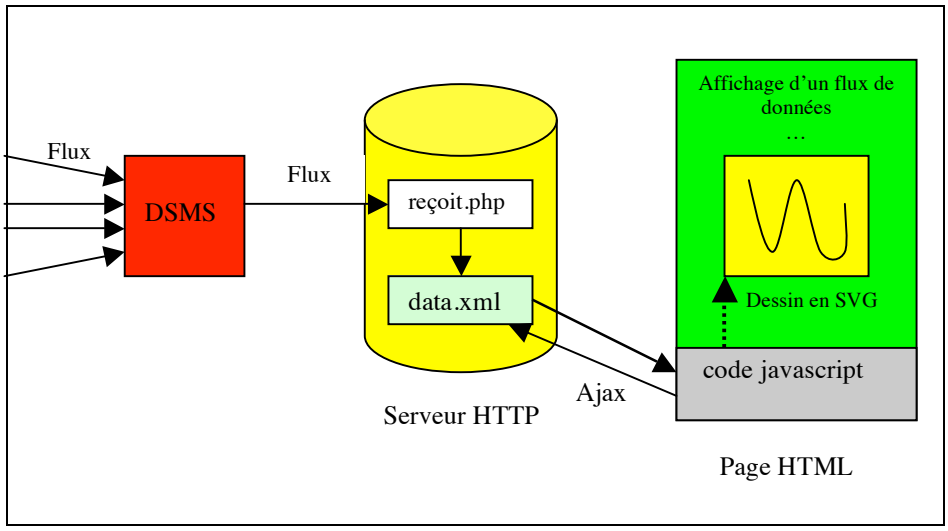


Figure 4 : architecture de l'application

A partir du poste client, Ajax permet de faire des requêtes asynchrones sur ce *serveur http* afin d'obtenir les données (fichier XML) et ainsi de ne modifier que certains éléments graphiques de la page html pour visualiser ces valeurs en temps réel.

Il est nécessaire de passer par l'intermédiaire d'un fichier temporaire car il n'est pas possible de synchroniser l'envoi des tuples par le DSMS (*Data Stream Management System*) et la requête ajax envoyée depuis le poste client. La requête Ajax aura donc comme *url* ce fichier au *format XML*.

Dans cette architecture logicielle, il aurait été possible d'utiliser un fichier au format texte quelconque. Mais nous avons fait le choix du format XML car d'une part les tuples envoyés par le DSMS sont déjà au format XML et d'autre part cela nous permet d'utiliser les fonctions de l'API javascript pour parser un document XML.

7. Une application : simulation de données provenant de compteurs

Afin d'illustrer cette étude, nous avons simulé un flux de données. Les *tuples* envoyés par le DSMS sont de la forme suivante:

```
<tuple>
    <Count>52</Count>
    <StartTime>2007-11-14 11:39:30.130+0100</StartTime>
    <EndTime>2007-11-14 11:39:31.145+0100</EndTime>
</tuple>
```

Chaque tuple contient une valeur numérique ainsi que l'heure de début et d'heure de fin de l'intervalle de validité de cette valeur. Cette valeur pourrait correspondre à une consommation sur une fenêtre temporelle. Les deux autres balises jouent le rôle d'estampille temporelle (*timestamp*) dont il faudra tenir compte pour remettre les tuples dans l'ordre chronologique.

La page html chargée sur le client contient une zone SVG, vide à l'origine et qui inclut les fichiers javascript. Au chargement de cette page, les axes et les légendes sont tracés et les éléments du graphique sont créés dynamiquement. Par la suite, la connexion au flux de données sera établie afin de lire le fichier contenant les tuples. Les requêtes ajax seront faites à intervalle de temps régulier fixé en fonction de la fréquence d'apparition des tuples.

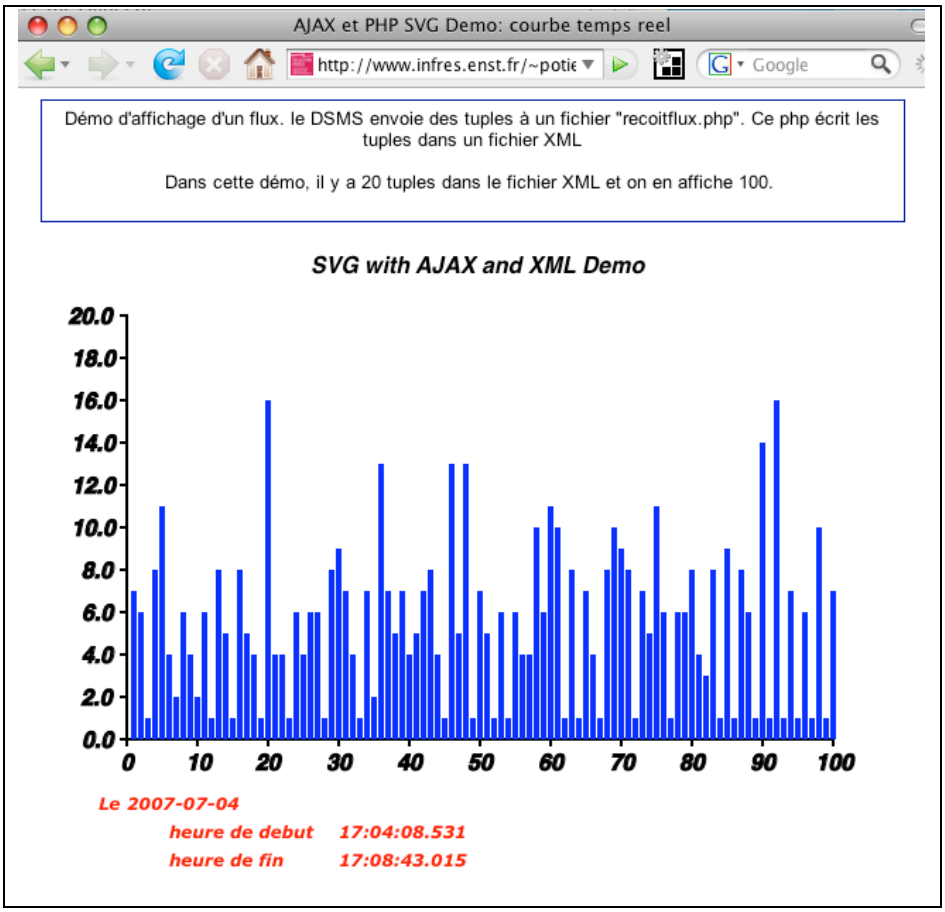


Figure 5 : affichage du flux de données dans un navigateur

Les données reçues sont visualisées sous forme de barres verticales (*histogramme*). Pour chaque nouvelle valeur reçue, les barres figurant déjà sur le graphique sont décalées vers la gauche et une nouvelle barre est insérée à droite. Dans la pratique pour simuler ce décalage, on recalcule toutes les hauteurs des barres et on modifie les attributs « hauteur » des éléments graphiques correspondants (voir Annexe B). La figure 5 montre le graphique obtenu à un instant t.

Connexion fréquente ou moins fréquente

Afin d'évaluer le comportement de cette architecture logicielle, et en particulier de savoir si nous ne perdions pas de tuples, nous avons testé plusieurs solutions :

a) le *fichier* lu ne contient *qu'un seul tuple*. Ce tuple sera alors inclus dans le graphique en décalant les autres valeurs. Il faut dans ce cas avoir une fréquence d'interrogation supérieure à la fréquence d'arrivée des tuples.

b) le fichier lu contient *plusieurs tuples* (une fraction des tuples affichés). Dans ce cas, certains tuples pourront avoir été déjà reçus la fois précédente. Il n'est pas nécessaire de faire des requêtes aussi fréquentes que dans le cas précédent.

Cependant il faut éviter de recevoir trop de tuples nouveaux d'un seul coup car cela conduirait à un décalage brutal dans le graphique d'où une impression d'à-coups.

c) le fichier lu contient *autant de valeurs que le nombre* de valeurs que l'on veut afficher. Il suffit de les ordonner en utilisant les timestamps. Mais cette solution peut aussi donner une impression de sauts.

Pour les deux premières solutions, il faut mémoriser les tuples reçus les fois précédentes et insérer les nouveaux dans le graphique en gérant les décalages.

8. Conclusion

Nous avons implémenté ces solutions pour tester le temps de connexion et de déconnexion, afin de voir si l'on ne perdait pas de tuples. Il s'avère que les trois solutions sont équivalentes mis à part les éventuels sauts dans l'affichage. Les essais ont été faits sur un réseau « local », le serveur http n'était pas au bout du monde! Le temps de connexion, de lecture du fichier et de transmission des données, se mesure en millième de secondes. Il est négligeable par rapport à la fréquence d'arrivée des tuples depuis le DSMS. On imagine assez mal une application de ce type où un être humain regarderait s'afficher des données arrivant toutes les millisecondes !

Puisque ces solutions sont toutes acceptables, il est donc judicieux de choisir entre les trois celle qui offre le moins d'inconvénient. Or dans la solution c) et dans une moindre mesure dans la solution b), le problème reste la gestion des sauts sur le graphique. Pour éviter de gérer une pseudo-synchronisation complexe, une bonne solution est de choisir une fréquence de requête Ajax proche de la fréquence d'apparition des tuples et de lire deux ou trois tuples à chaque fois. Cela évite d'avoir à gérer les sauts dans le graphique et nous garantit contre la perte de tuple.

Notre but était aussi de tester la connexion Ajax avec SVG ainsi que la rapidité de l'affichage SVG. A condition de bien respecter la norme du W3C en cours de développement, cette application a montré que l'utilisation de SVG pour l'affichage d'un flux de données est une très bonne solution qui utilise uniquement des standards ouverts.

Bibliographie

Sur AJAX

[1] Jesse James Garrett, AJAX : A new approach to web application, article paru le 18 février 2005 sur le site de Adaptive Path, http://www.adaptivepath.com/ideas/essays/archives/000385.php

[2] The XMLHTTPRequest Object du W3C :

http://www.w3.org/TR/XMLHttpRequest/

[3] la traduction française http://www.xul.fr/XMLHttpRequest.html

Sur SVG

[4] les spécifications sur le site du W3C : http://www.w3.org/Graphics/SVG/

[5] SVG : Production orientée XML de graphiques vectoriels, J. David Eisenberg, O'Reilly, 2004

Sur Javascript

[6] Javascript : la référence , David Flanagan, O'Reilly, 2002

Pour des recommandations sur l'écriture de SVG et du javascript :

[7] SVG:Scripting, http://developer.mozilla.org/en/docs/SVG:Scripting

[8] SVG : Authoring Guidelines , http://jwatt.org/svg/authoring/

[9] SVG-wiki, http://wiki.svg.org/Main_Page#Scripting_and_Programming

Annexe A : AJAX

A - 1) L'API de la classe XMLHTTPRequest

Voici une description non exhaustive de la classe `XMLHTTPRequest` qui permet de faire une requête Ajax en établlissant une connexion avec un serveur via le protocole http. Pour une description plus complète on se réfèrera à [2] ou [3]. Cependant elle présente les attributs et les méthodes qui sont le plus souvent utilisés.

Attributs :

```
eventListener onreadystatechange;
readonly unsigned short readyState;
unsigned short status;          // 200 si c'est OK, 404 page non trouvée
DOMString responseText;         // réponse sous forme de chaîne de caractères
Document responseXML;           // réponse au format XML
DOMString statusText;
```

Méthodes :

- Constructeur `XMLHTTPRequest();` // dépend du navigateur (cf. A-3)

- `open(`*`méthode, url, [ async, usr, password ]`*`);`

  sert à établir la connexion en donnant l'url du programme ou du fichier et la méthode de connexion (`GET, POST`,…). L'url est celle de la ressource sur le serveur. Les autres paramètres sont optionnels. On l'utilise souvent sous la forme

  ```
  open(méthode, url, true);
  ```

  Si on choisit la méthode "`GET`", il faut éventuellement transmettre les valeurs des paramètres sous un format identique à la tranmission des paramètres à un fichier php. S'il y a plusieurs données, on les sépare par des &. On construit une chaîne de caractères par concaténation:

  ```
  url?parametre1=valeur&parametre2=valeur….
  ```

  Par exemple :

  ```
  open("GET","zone.php?batiment=C200&X=10&Y=234",true);
  ```

  Si on choisit la méthode "`POST`", les données seront envoyées au serveur comme argument de la méthode `send()`.

- `send(data)`

  pour envoyer la requête avec

  `data = null` si on a choisit la méthode `GET`

  `data ="param1=valeur&param2=valeur.."` avec la méthode `POST`

  Par exemple:

  ```
  send("batiment=C200&X=10&Y=234");
  ```

- `setRequestHeader("Content-Type",type);` pour donner un header à la réponse avec par exemple:

  ```
  type= "application/x-www-form-urlencoded"
  ```

Fonctionnement

L'attribut `readyState` permet de connaître le niveau de la transmission. Cet attribut peut prendre 5 valeurs, qui sont atteintes successivement si tout va bien :

0 ou `UNSENT` : à la création de l'objet

1 ou `OPENED` : connexion établie, quand la méthode `open` a été invoquée avec succès

2 ou `HEADERS_RECEIVED` : quand les entêtes de la réponse ont été reçus

3 ou `LOADING` : pendant la réception du corps de la réponse

4 ou `DONE` : quand les données ont été transférées.

Les données fournies par le serveur seront récupérées soit dans l'attribut `responseXML`, soit dans l'attribut `responseText`. Si la réponse est un document XML, il faut attendre que `readyState` prenne la valeur 4 pour traiter les données et il sera lisible par les méthodes du DOM. Si la réponse est du texte, on peut commencer à traiter le résultat de la requête dès que `readyState` ≥ 3.

Dès que le premier octet de la réponse a été reçu, un événement `readystatechange` est envoyé par l'objet « client » ce qui active la méthode « écouteur » correspondante.

A - 2) Exemple d'utilisation

Nous décrivons ici la façon dont nous avons utilisé l'API dans un navigateur **Firefox sur Mac**. Pour une utilisation quel que soit le navigateur et quelle que soit la plateforme, on se reportera au § A-3 « utilisation générale de l'API » .

Pour obtenir un fichier XML sur un *serveur via http*, on procède de la manière suivante :

```
client = new XMLHttpRequest();                                    (1)

if (client && ((client.readyState == 4) ||
               (client.readyState == 0) ) )
  {
    client.open("GET", "data.xml", true);                         (2)
    client.send(null);                                            (3)
    client.onreadystatechange = traceCourbe;                      (4)
  }
```

- *1) création d'un objet `XMLHTTPRequest`*

Dans le cas d'un navigateur Firefox ou Mozilla, en javascript cela s'écrit:

```
client = new XMLHTTPRequest() ;
```

Attention : Pour la création de cet objet quel que soit le navigateur et quelle que soit la plateforme on se réfèrera au § A-3.

- *2) établir la connexion*

```
client.open("GET", "data.xml", true ) ;
```

La méthode `open` peut prendre les valeurs GET, POST, etc. Le fichier `data.xml` est le fichier sur lequel le DSMS envoie les données. Il est situé sur le serveur au même niveau d'arborescence que le fichier html que l'on affiche sur le client, sinon il faudrait donner l'url complète de ce fichier. Cpendant, il s'agit toujours d'un fichier accessible sur le même serveur http.

- *3) envoyer la requête au serveur*

```
client.send ( null ) ;
```

L'attribut `readyState` permet de connaître le niveau de transmission de la requête. Si la réponse est un document au format XML (avce le bon type MIME), il faut attendre que les données soient intégralement reçues pour les traiter en utilisant les méthodes du DOM. Si la réponse est du texte simple, on peut commencer à

traiter le résultat de la requête dès que la transmission du corps de la réponse est débutée et dans ce cas, il faut parser les données.

- *4) Traitement du résultat*

Dès que l'attribut `readyState` prend la valeur 3 (début de transmission du corps de la réponse), un événement `readystatechange` est envoyé par l'objet `client` ce qui active la méthode « callabck » correspondante. Il faut donc avoir un écouteur pour traiter l'événement. Pour cela il faut définir la méthode qui sera invoquée pour gérer l'événement :

```
client.readystatechange = traceCourbe ;

function traceCourbe()
{
   if ((client.readyState==4) && (client.status == 200))
     {
            // traitement de la réponse du serveur
            // pour modifier le graphique
            . . .
   }
}
```

Cette fonction permet de mettre en forme le graphique en fonction de la réponse du serveur. Pour cela on utilisera les possibilités offertes par SVG et l'interface du DOM (voir annexe B).

Remarque :

Si la réponse est construite dynamiquement par un script PHP, il faut donner le bon type MIME à la réponse.

- Si l'on veut renvoyer du texte simple, on utilisera:

```
header("Content-Type: text/plain");
```

- Si l'on veut renvoyer un fichier XML, on utilisera:

```
header("Content-Type: text/xml");
echo "<?xml version='1.0' encoding='iso-8859-1'?>";
```

A – 3) Utilisation générale de l'API

Pour la création d'un objet XMHttpRequest, il est nécessaire de tester les paramètres du navigateur. La fonction `createXMLHttpRequest()` permet de créer cet objet **quelle que soit la plateforme et quel que soit le navigateur**:

```
function createXMLHttpRequest()
{
  var xmlReq;
  // this should work for all browsers except IE6 and older
  try
  {
      xmlReq = new XMLHttpRequest();
  }
  catch(e)
  {
    // assume IE6 or older
    var XmlHttpVersions = new Array("MSXML2.XMLHTTP.6.0",
                                    "MSXML2.XMLHTTP.5.0",
                                    "MSXML2.XMLHTTP.4.0",
                                    "MSXML2.XMLHTTP.3.0",
                                    "MSXML2.XMLHTTP",
                                    "Microsoft.XMLHTTP");
     for (var i=0; i<XmlHttpVersions.length && !xmlReq; i++)
    {
      try
      {
        xmlReq = new ActiveXObject(XmlHttpVersions[i]);
      }
      catch (e) {}
    }
  }

  if (!xmlReq)
    alert("Error creating the XMLHttpRequest object.");
  else
    return xmlReq;
}
```

En reprenant l'exemple d'utilisation du paragraphe précédent A-2 dans le cas général, on remplacera l'appel au constructeur `XMLHttpRequest()` (1) par :

```
client = createXMLHttpRequest() ;
```

A - 4) Ajax et Flux de données

Pour ouvrir la connexion automatiquement à intervalle régulier avec le serveur on peut utiliser la fonction javascript `setTimeout` qui permet de rappeler une fonction à intervalle régulier. On rappelera ainsi à intervalle régulier la fonction qui établit la connexion.

En résumé on procède de la façon suivante:

Au chargement de la page on appelle une fonction d'initialisation. On doit donc avoir dans cette page une balise contenant l'attribut:

```
onload="init(evt)"
```

où la fonction `init(evt)` fait les initialisations nécessaires et se termine par un appel à la fonction `setTimeout`.

Dans le code javascript, on devra donc trouver les fonctions suivantes:

```
function init(evt)
{
    ....  // diverses initialisations

   setTimeout("updateChart()", updateInterval);
  // updateInterval:  nombre de millisecondes
}


/*------------------------------------------------------------*/
/* initialise la requête asynchrone pour obtenir              */
/* des nouvelles données                                      */
/*------------------------------------------------------------*/

function updateChart()
{
      ajaxRequest(nom_fichier_XML, trace_courbe);
}

 function ajaxRequest(url, callback)
{
      // code donné au paragraphe A-2 et A-3
}

/*------------------------------------------------------------*/
// fonction callback qui traite les données reçus du serveur
/*------------------------------------------------------------*/

function trace_courbe(data)
{ .....
}
```

Annexe B : Utilisation de l'interface DOM en Javascript dans un document SVG

L'implémentation de SVG dans les navigateurs web n'est pas toujours identique et la syntaxe peut être légèrement différente d'un navigateur à l'autre. Nous donnons ici la syntaxe qui fonctionne sur le plus grand nombre de navigateurs, c'est-à-dire la syntaxe qui suit les spécifications de SVG et du DOM2.

La programmation javascript permet de gérer l'interaction avec l'utilisateur et les événements. A chaque action de l'utilisateur (chargement de la page, clic de souris, survol, …) un objet `evt` est créé par le programme, et cet objet peut être transmis lors de l'appel à la fonction « callback » qui prend en charge d'événement. Par exemple au chargement du document SVG, on appelle une fonction d'initialisation:

```
onload =  init(evt).
```

On pourra ainsi avoir des fonctions pour chaque événement que l'on décide de traiter.

Fonctions de l'interface pour gérer le graphique :

Pour pouvoir interagir sur les éléments du graphique, il faut pouvoir accéder à la **racine** de **l'arbre** DOM du graphique **SVG**. Quand un événement se produit dans la fenêtre SVG, on récupère la racine du document par:

```
documentSVG = evt.target.ownerDocument;
```

A partir de cette racine, il est ensuite possible d'accéder aux différents éléments du graphique en utilisant par exemple les fonctions qui permettent de naviguer dans l'arbre, de modifier l'arbre en ajoutant ou supprimant des noeuds, d'accéder aux atrributs, etc.
Nous présentons ici quelques proporiétés des classes d'objets de l'API qui nous sommes les plus utiles. Pour la description complète de l'API du DOM1 et du DOM2, on se réfèrera à [4].

Propriétés d'un objet `Node`
Attributs:

- `nodeName` (String)
- `nodeValue` (String)
- `nodeType` (Number)
- `parentNode` (Node)
- `childNodes` (liste de noeuds)
- `firstChild` (Node)

`lastChild` (Node)
`previousSibling` (Node)
`nextSibling` (Node)
`attributes` (NamedNodeMap)
`ownerDocument` (Document)
`namespaceURI` (String)
`prefix` (String)
`localName` (String)

Méthodes les plus utiles:

méthodes	Objet ou valeur renvoyé
`replaceChild(`*`newChild, oldChild`*`)`	`Node`
`removeChild(`*`oldChild`*`)`	`Node`
`appendChild(`*`newChild`*`)`	`Node`
`hasChildNodes()`	`boolean`

Propriétés d'un objet Liste de noeuds (`NodeList`)

`length`

Propriétés d'un objet `Attr`

`name`
`value` (String)

Propriétés d'un objet `document`

Un objet `document` a toutes les propriétés (attributs, méthodes) d'un objet `Node` plus des propriétés spécifiques.

Méthodes les plus utiles qui s'appliquent à un objet `document`

fonctions du DOM1 fonctions du DOM2 (à utiliser de préférence)	Objet ou valeur renvoyé
`createElement(`*`name`*`)` `createElementNS(`*`namespaceURI, localName`*`)`	`Element`
`appendChild(`*`nouveau_noeud`*`)`	`Node`
`getElementsByTagName(`*`name`*`)` `getElementsByTagNameNS(`*`namespaceURI,localName`*`)`	`NodeList`
`getElementsById(`*`nom`*`)`	`Element`

Propriétés d'un objet `Element`

Un objet `Element` a toutes les propriétés d'un objet `Node` plus des propriétés spécifiques parmi lesquelles:

Attribut : `tagName`

Méthodes:

fonctions du DOM1 fonctions du DOM2 (à utiliser de préférence)	Objet ou valeur renvoyé
`getAttribute (`*`name`*`)` `getAttributeNS(`*`namespaceURI`*`,`*`localName`*`)`	`String`
`setAttribute(`*`name, value`*`)` `setAttributeNS(`*`namespaceURI`*`,`*`localName, value`*`)`	
`removeAttribute(`*`name`*`)` `removeAttributeNS(`*`namespaceURI`*`,`*`localName`*`)`	
`getElementsByTagName(`*`name`*`)` `getElementsByTagNameNS(`*`namespaceURI`*`,`*`localName`*`)`	`NodeList`
`getElementsById(`*`name`*`)`	`Element`
`hasAttribute(`*`name`*`)` `hasAttribute(`*`namespaceURI`*`,`*`localName`*`)`	`boolean`

Exemple d'utilisation de ces méthodes

```
chartGroup = documentSVG.createElementNS(svgNS, "g");
dataLine = documentSVG.createElementNS(svgNS, "line");
```

où `svgNS = "http://www.w3.org/2000/svg";`

Après avoir créer un noeud, il faut **l'ajouter** dans l'arbre SVG:

```
documentSVG.documentElement.appendChild(dataLine);
```

On peut modifier les valeurs des attributs:

```
dataLine.setAttributeNS(null,"id",i);
dataLine.setAttributeNS(null,"onmouseover", "ma_fonction");
```

pour associer une interaction à l'objet graphique.

```
dataLine.getAttributeNS(null,"fill");
```

Méthodes permettant d'accéder aux éléments d'un document SVG:

```
documentSVG.getElementById("rectangle1");
documentSVG.getElementsByTagName("circle")
```

Si l'on souhaite connaître le nombre d'élements de ce type:

```
documentSVG.getElementsByTagName("circle").length;
```

et pour accéder à un de ces éléments:

`noeud = documentSVG.getElementsByTagName("circle")[i]` où `i` varie de `0` à `length-1`

Ensuite pour accéder à la valeur de cette balise:

```
y = noeud.firstChild.nodeValue;
```

Remarque: quelle valeur donner à l'argument "`namespaceURI`"

Pour les méthodes du DOM2 qui nécessitent un espace de noms, le premier argument doit être l'uri d'un espace de noms correspondant à l'élement ou l'attribut en question.
Pour les **éléments** SVG, on met comme espace de noms "`http://www.w3.org/2000/svg`".
Pour les attributs sans préfixe, les espaces de noms dans la recommandation XML1.1 n'ont pas de valeur. En d'autres termes, on utilise "`null`" comme espace de noms pour les **attributs** d'éléments SVG.

Ce qui donne pour créer un élément "rect"

```
createElementNS("http://www.w3.org/2000/svg", "rect");
```

Alors que pour obtenir la valeur de l'attribut 'x' d'un élément SVG 'rect' on écrit:

```
getAttributeNS(null, 'x');
```

Il faut noter que ce n'est pas le cas pour les attributs d'un autre espace de noms (non-SVG) comme pour l'attribut xlink:href qui utilise le préfixe xlink. Son espace de noms est la valeur qui est assignée à ce préfixe, `'http://www.w3.org/1999/xlink'`.
Dans ce cas, pour obtenir par exemple, la valeur de l'attribut 'href' d'un élément 'a', on écrit:

```
getAttributeNS('http://www.w3.org/1999/xlink', 'href');
```

En résumé, la règle est simple:

Pour les éléments avec ou sans préfixe et pour les attributs avec un espace de noms pour préfixe, l'espace de noms est l'URI de l'espace de noms de l'élément/attribut en question.

Pour les attributs sans espace de noms comme préfixe, le nom de l'espace de noms est 'null'.

Annexe C: Recommandations de syntaxe SVG

- Le **fichier doit commencer** par

```
<svg version="1.1"
     baseProfile="full"
     xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:ev="http://www.w3.org/2001/xml-events">
```

- **Eviter l'attribut "style".** Il est préférable d'écrire :

```
<circle fill="red" stroke="blue" ... />
```

au lieu de :

```
<circle style="fill:red; stroke:blue;" ... />
```

L'attribut style sépare le contenu de la présentation et à moins d'avoir besoin d'utiliser les CSS, il est préférable de donner des valeurs à ces attributs. De plus cette écriture est supportée par SVG Tiny.

- Spécifier les **unités** de longueur, même si elles peuvent être omises. Exemple, écrire :

```
<text stroke-width="2px" font-size="20px;" ...>
```

- Ne pas utiliser les extensions 'get' et 'set' d'Adobe:
Avec le plugin d'Adobe, on écrivait un code du type:

```
evt.getTarget().getOwnerDocument().getDocumentElement();
```

Les spécifications du DOM et de SVG définissent précisemment ces propriétés, des attributs et non des méthodes, donc la façon correct d'écrire le code est:

```
evt.target.ownerDocument.documentElement;
```

- Utiliser de préférence les méthodes du DOM2

Les scripts doivent être conformes aux standards et marcheront sur toutes les implémentations de SVG en utilisant ces propriétés.

Dépôt légal : 2008 – 3ème trimestre
Imprimé à l'Ecole Nationale Supérieure des Télécommunications – Paris
ISSN 0751-1345 ENST D (Paris) (France 1983-9999)

TELECOM ParisTech

Institut TELECOM - membre de ParisTech

46, rue Barrault - 75634 Paris Cedex 13 - Tél. + 33 (0)1 45 81 77 77 - www.telecom-paristech.frfr

Département INFRES