18

Sockets

UNE PASSERELLE UNIVERSELLE **1**

CREER UN SERVEUR DE SOCKET XML **6**

LA CLASSE XMLSOCKET 7
CREER UN T'CHAT MULTI-UTILISATEUR 11

LA CLASSE SOCKET **23**

CREER UN SERVEUR DE SOCKET BINAIRE 24
ECHANGER DES DONNÉES 28

Une passerelle universelle

Nous avons vu au cours des chapitres précédents comment le lecteur Flash pouvait communiquer avec l'extérieur. Nous allons aller plus loin en abordant à présent la notion de communication par socket.

Une socket doit être considérée comme une passerelle de communication entre deux applications fonctionnant sur un réseau. Dans la vie courante, une socket pourrait être assimilée à une connexion téléphonique entre deux personnes.

Pour entreprendre une communication par socket, deux acteurs au minimum sont nécessaires :

- Le serveur : le serveur de socket écoute les connexions entrantes, il se charge de gérer les clients connectés et communique avec eux.
- Le client : le client se connecte au serveur par la socket et communique avec le serveur.

Notons que l'avantage principal d'une connexion par socket réside dans le caractère persistent de la communication entre les deux acteurs. Contrairement à une connexion HTTP traditionnelle, la socket

Thibault Imbert
pratiqueactionscript3.bytearray.org

maintient la connexion ouverte entre les deux acteurs et permet au serveur de transmettre des informations aux clients sans que ces derniers n'en fassent la demande.

Ce comportement offre ainsi de nombreuses possibilités en termes d'applications dynamiques temps réel.

Afin de bien comprendre la notion de socket, prenons l'exemple suivant :

Lorsque vous naviguez sur Internet à l'aide votre navigateur favori, ce dernier demande au serveur distant de lui transmettre la page que vous souhaitez afficher. Ce langage commun entre les deux acteurs est appelé *protocole d'application* et fait partie de ce que l'on appelle la *suite de protocoles Internet*.

> Internet est basé sur cet ensemble de protocole généralement appelé modèle TCP/IP.

Afin d'afficher une page spécifique, le navigateur envoie au serveur HTTP la requête suivante :

```
GET /index.php HTTP/1.1
Host: www.bytearray.org
```

Nous pouvons remarquer la présence de mots réservés tels `GET` ou `Host` faisant partie du protocole d'application HTTP. Par ces mots clés, le serveur comprend qu'il doit transmettre le contenu de la page `index.php` au client connecté, ici notre navigateur.

Aussitôt la requête reçue, le serveur l'analyse et répond à l'aide du message suivant :

```
HTTP/1.x 200 OK
Date: Sun, 20 Jan 2008 14:30:34 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-22 mod_ssl/2.0.54
OpenSSL/0.9.7e mod_perl/1.999.21 Perl/v5.8.4
X-Powered-By: PHP/4.3.10-22
X-Pingback: http://www.bytearray.org/xmlrpc.php
Content-Encoding: gzip
Vary: Accept-Encoding
Content-Length: 4260
Keep-Alive: timeout=15, max=97
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

En réalité, l'application serveur et le client communiquent par socket à l'aide d'un protocole commun. Un nombre illimité de protocoles d'applications peuvent ainsi être implémentés ou crées puis utilisés à l'aide d'une connexion par socket.

Nous sommes donc en mesure d'utiliser au sein de Flash n'importe quel protocole d'application tel HTTP, FTP, SMTP ou autres.

Pour différencier les applications d'un ordinateur associées à chaque protocole, chaque application est associée à un port spécifique appelé couramment *port logiciel*. Différentes applications sont donc accessibles à partir d'une même adresse IP mais un port unique.

Parmi les protocoles d'applications les plus connus, nous pouvons établir le tableau suivant :

Port	**Application**
21	FTP (transfert de fichiers)
25	SMTP (envoi de courrier électronique)
80	HTTP (transfert hypertexte)
110	POP (stockage de courrier électronique)

Tableau 1. Liste de protocoles communs.

Afin de connaître les ports logiciels associés à chaque protocole. Vous pouvez consulter la liste disponible aux adresses suivantes :

- http://fr.wikipedia.org/wiki/Liste_des_ports_logiciels
- http://www.iana.org/assignments/port-numbers

Ainsi, nous pouvons placer les différents ports dans trois catégories :

- Les ports bien connus (0 à 1023) : Ces ports sont couramment utilisés par des processus systèmes ayant un accès privilégié. Il est donc déconseillé d'y avoir recours pour un serveur de socket personnalisé, un risque de collision serait encouru.
- Les ports enregistrés (1024 à 49151) : Ils sont dédiés aux développements d'applications courantes. La plupart de ces ports demeurent libres et peuvent être utilisés.
- Les ports dynamiques privés (49152 à 65535) : Ces ports demeurent généralement libres.

Il est important de préciser que les classes de socket ActionScript 3 s'appuient sur un modèle de transmission TCP et sont considérées comme des passerelles d'échanges sûres, s'opposant au modèle de transmission UDP considéré comme peu fiable.

Mais qu'entendons-nous par passerelle d'échange sûre ?

En réalité, deux types de protocoles peuvent être utilisés sur Internet, le premier de type TCP puis le second de type UDP :

- TCP (Transmission Control Protocol) : Dans un contexte de connexion TCP, les données échangées sont contrôlées. Au cas où la transmission est trop rapide ou si des données sont perdues, le serveur ralentit le débit, et réexpédie les données non correctement transmises.
- UDP (User Datagram Packet) : Dans un contexte de connexion UDP, afin de gagner en vitesse de transfert aucun contrôle n'est effectué lors

du transfert des données. En revanche certains paquets peuvent ne jamais parvenir. Ce type de socket est généralement utilisé dans le cas de jeux en temps réel où la perte de quelques paquets n'influence pas le bon fonctionnement de l'application.

Les classes de socket ActionScript 3 s'appuient sur le protocole TCP, une compatibilité future avec le protocole UDP pourrait être intéressante pour certains types d'applications.

Une communication par socket est généralement divisée en plusieurs phases distinctes :

- 1. Le client se connecte au serveur de socket en envoyant une commande spécifique afin d'initier la conversation ou de s'authentifier.
- 2. Le serveur décide d'accepter ou non le client. Cette étape peut être vérifiée de par la provenance du client ou les informations soumises par ce dernier lors de la procédure d'authentification.
- 3. La communication est établie.

A tout moment, un des acteurs de la communication peut décider de clore la connexion. Le serveur peut décider d'interrompre la communication au cas où un client ne serait plus autorisé ou simplement lorsqu'un client souhaite se déconnecter.

Voici les deux classes permettant de se connecter à un serveur de `Socket` en ActionScript 3 :

- `flash.net.XMLSocket` : la classe `XMLSocket` permet l'échange de données au format XML ou texte.
- `flash.net.Socket` : la classe `Socket` permet l'échange de données brutes au format binaire.

Il est important de noter qu'ActionScript 3 n'intègre pas de classes permettant la création de serveurs de sockets, seules des applications clients pourront être développées.

Il est donc impossible de connecter directement deux applications Flash au travers d'une connexion socket. Pour cela, l'une des applications devrait se comporter comme hôte ce qui est impossible en ActionScript 3.

Si tel était le cas, la création d'un réseau *peer 2 peer* entre différentes applications Flash serait envisageable.

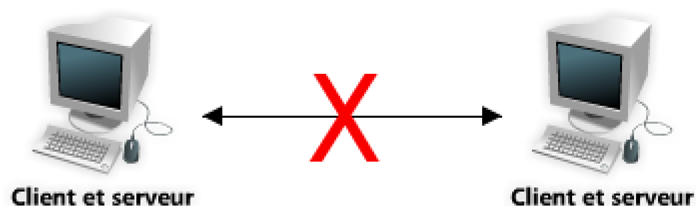

Figure 18-1. Connexion peer 2 peer non supportée.

Le choix du langage utilisé pour la création du serveur dépend de vos besoins. Des langages tels le C, C++, C#, Java ou encore PHP offrent tous la possibilité de créer des serveurs de socket.

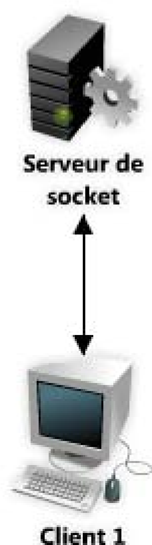

Figure 18-2. Connexion à un serveur de socket.

Nous allons nous intéresser dans la partie suivante aux différents types de sockets disponibles en ActionScript 3.

A retenir

- Une socket est une passerelle de communication entre deux applications évoluant sur un réseau.
- Les classes de Socket ActionScript 3 sont basées sur le modèle TCP/IP.
- Le protocole UDP n'est pas pris en charge.
- Les classes de socket ActionScript 3 ne permettent pas la création de serveur de socket.
- Le nombre de messages échangés à travers une connexion socket est illimité.
- ActionScript 3 intègre deux classes liées aux sockets : `XMLSocket` et `Socket`.

Créer un serveur de socket XML

Avant d'entamer le développement du serveur de socket, il convient de définir son rôle ainsi que son fonctionnement.

Le rôle d'un serveur de socket est d'écouter les connexions entrantes puis de gérer par la suite la connexion ainsi que les échanges entre les acteurs impliqués dans la communication.

Nous avons vu précédemment qu'il était impossible de connecter directement plusieurs applications Flash entre elles. A l'inverse, le serveur de socket peut agir comme relais afin de faire transiter les messages entre les différents clients.

Vous pensiez peut être que cela était réservé à des technologies telles *Flash Media Server* ou *Red 5*, nous allons voir que quelques lignes de PHP vont nous permettre de connecter plusieurs applications Flash entre elles. Nous allons développer au cours de cette partie un serveur de socket XML afin de créer un t'chat multi utilisateurs.

Comme nous l'avons vu précédemment, de nombreux langages permettent la création de serveur de socket de manière simplifiée. Parmi ceux là nous pouvons citer Java, C#, C++ ou PHP qui s'avère être le moyen le langage le plus simple pour déployer votre serveur de socket.

Une explication approfondie de l'API de socket PHP serait hors sujet, c'est la raison pour laquelle nous ne rentrerons pas dans les détails du code du serveur. Nous allons donc commencer par un simple serveur renvoyant un message de bienvenue lorsque nous nous connectons.

Le code PHP suivant est placé au sein d'un fichier nommé `ServeurXMLSocket.php` :

```
#!/usr/local/bin/php -q
```

```
<?php

set_time_limit(0);

$adresse = "localhost";
$port = "10000";

$connexion = socket_create (AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind ($connexion, $adresse, $port);

socket_listen ($connexion, 1);

echo "Le serveur de socket est en route !";

$client = socket_accept ($connexion);

socket_close ($client);

socket_close ($connexion);

?>
```

Afin de démarrer convenablement le serveur, il convient de le lancer par ligne de commande.

Pour cela, nous créons un nouveau fichier texte contenant le code suivant :

```
C:/wamp/bin/php/php5.2.5/php.exe -q c:/wamp/www/serveur/ServeurXMLSocket.php
```

Puis, nous sauvons le fichier texte sous le nom `serveur.bat`.

Depuis la ligne de commande nous accédons au fichier `.bat` puis nous l'exécutons, le message illustré par la figure 18-3 doit s'afficher :

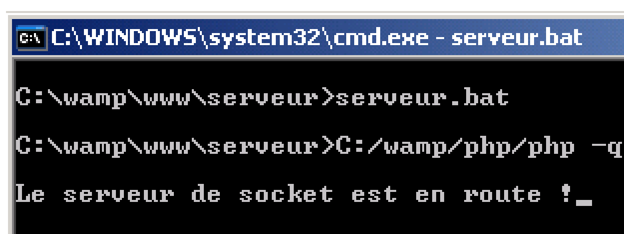

Figure 18-3. Démarrage du serveur depuis la ligne de commande.

Une fois le serveur démarré, nous devons nous y connecter, c'est ce que nous allons découvrir dans la partie suivante.

La classe XMLSocket

Afin de pouvoir se connecter à notre serveur de socket, nous pouvons utiliser une instance de la classe `XMLSocket` dont voici le constructeur :

```
public function XMLSocket(host:String = null, port:int = 0)
```

Voici le détail de chacun des paramètres :

- host : il s'agit de l'adresse du serveur de socket. Attention, si le serveur évolue au sein d'un autre domaine que le SWF, la connexion doit être autorisée par un fichier de régulation.
- port : le numéro de port TCP utilisé lors de la connexion. Par défaut il est impossible de se connecter à un port inférieur à 1024 pour des raisons de précautions. Si vous souhaitez tout de même utiliser un port inférieur vous devez utiliser un fichier de régulation.

Dans le code suivant, nous établissons une connexion avec notre serveur de socket XML :

```
// création de la connexion
var connexion:XMLSocket = new XMLSocket("localhost", 10000);
```

Au cas où l'adresse et le port utilisé ne seraient pas précisés lors de l'instanciation, nous pouvons utiliser la méthode connect :

```
// création de l'objet XMLSocket
var connexion:XMLSocket = new XMLSocket();

// connexion au serveur de socket
connexion.connect("localhost", 10000);
```

Attention, les classes XMLSocket et Socket n'ont pas la possibilité de se connecter à des ports supérieurs à 65535 et inférieurs à 1024. L'utilisation de ports inférieurs à 1024 pourrait entraîner une collision avec des serveur tels HTTP, FTP ou autres.

> Si vous souhaitez utiliser un port inférieur à 1024, l'utilisation d'un fichier de régulation est nécessaire.

Afin d'écouter les différentes phases liées à la connexion nous utilisons les événements diffusés par l'objet XMLSocket dont voici la liste :

- Event.CLOSE : diffusé lorsque la connexion socket est interrompue.
- Event.CONNECT: diffusé lorsque la connexion au serveur a pu être réalisée.
- DataEvent.DATA : diffusé lors de l'envoi ou réception de données.
- IOErrorEvent.IO_ERROR : diffusé lorsque la connexion au serveur de socket n'a pas aboutie.
- SecurityError.SECURITY : diffusé lorsque la connexion socket tente de se connecter auprès d'un serveur no autorisé ou à port inférieur à 1024.

Nous écoutons les différents événements de connexions ainsi que l'événement DataEvent.DATA :

```
// création d'une instance de XMLSocket
var connexion:XMLSocket = new XMLSocket("localhost", 10000);

// écoute des événements
connexion.addEventListener ( Event.CONNECT, connexionReussie );
connexion.addEventListener ( Event.CLOSE, fermetureConnexion );
connexion.addEventListener ( DataEvent.DATA, receptionDonnees );

function connexionReussie ( pEvt:Event ):void

{

  trace("connexion réussie");

}

function fermetureConnexion ( pEvt:Event ):void

{

  trace("fermeture de la connexion");

}

function receptionDonnees ( pEvt:DataEvent ):void

{

  trace("réception des données");

}
```

En testant le code précédent, nous voyons que la connexion aboutit, puis nous sommes immédiatement déconnectés.

Les messages suivants sont affichés par la fenêtre de sortie :

```
connexion réussie
fermeture de la connexion
```

Il serait intéressant de pouvoir parler au serveur de socket. Pour cela nous modifions les lignes PHP suivantes :

```
#!/usr/local/bin/php -q
<?php

set_time_limit(0);

$adresse = "localhost";
$port = "10000";

$connexion = socket_create (AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind ($connexion, $adresse, $port);

socket_listen ($connexion, 1);

echo "Le serveur de socket est en route !";

$client = socket_accept($connexion);
```

```
$messageEntrant = socket_read ($client, 1024);

$messageSortie = "Vous avez dit : ".$messageEntrant."\r\n";

socket_write ($client, $messageSortie);

socket_close ($client);

socket_close ($connexion);

?>
```

Afin de prendre en considération les modifications, nous redémarrons le serveur de socket. Désormais, celui-ci est en attente d'un message du client.

Nous allons envoyer une simple chaîne de caractères à l'aide de la méthode `send` de l'objet `XMLSocket` :

```
// envoie d'une chaîne de caractère au serveur
connexion.send ("il y'a quelqu'un ?");
```

En observant la fenêtre de sortie nous pouvons voir les messages suivants :

```
connexion réussie
réception des données
fermeture de la connexion
```

Aussitôt la méthode `send` exécutée, le lecteur Flash transmet au serveur de socket la chaîne de caractère spécifiée en terminant le message par un octet nul.

> Les données transmises par le serveur sont aussi terminées par un octet nul, ce marqueur permet au lecteur Flash de connaître en interne la fin du paquet transmis.

Comme son nom l'indique, la classe `XMLSocket` est prévue pour échanger des données au format XML. Pourtant, nous échangeons ici une chaîne de caractère traditionnelle.

Le format XML peut être utilisé si vous souhaitez échanger des données structurées, dans d'autres cas, des simples chaînes de caractères peuvent être utilisées.

Afin de lire les données provenant du serveur, nous utilisons la l'événement `DataEvent.DATA`. L'objet événementiel diffusé possède une propriété `data` contenant les données transmises par le serveur :

```
// création d'une instance de XMLSocket
var connexion:XMLSocket = new XMLSocket("localhost", 10000);

// envoie d'une chaîne de caractère au serveur
connexion.send ("il y'a quelqu'un ?");
```

```
// écoute des événements
connexion.addEventListener ( Event.CONNECT, connexionReussie );
connexion.addEventListener ( Event.CLOSE, fermetureConnexion );
connexion.addEventListener ( DataEvent.DATA, receptionDonnees );

function connexionReussie ( pEvt:Event ):void

{

  trace("connexion réussie");

}

function fermetureConnexion ( pEvt:Event ):void

{

  trace("fermeture de la connexion");

}

function receptionDonnees ( pEvt:DataEvent ):void

{

  // affiche : Vous avez dit : il y'a quelqu'un ?
  trace ( pEvt.data );

}
```

Attention, la méthode `send` n'est pas bloquante, nous devons garder à l'esprit le caractère asynchrone des échanges réalisés par une connexion socket. Seul l'événement `DataEvent.DATA` nous permet de récupérer les données transmises par le serveur et de déterminer à quel moment celles-ci sont disponibles.

Nous allons aller plus loin en développant un t'chat multi-utilisateur. Le serveur de socket XML que nous allons développer servira de relais afin de faire transiter les messages des clients connectés.

A retenir

- La classe `XMLSocket` permet la connexion auprès d'un serveur de socket.
- Les données échangées entre les deux acteurs doivent obligatoirement être au format XML ou texte.
- L'objet `XMLSocket` diffuse différents événements permettant d'indiquer l'état de la connexion et du transfert des données.

Créer un t'chat multi-utilisateur

Afin de développer notre t'chat multi utilisateur, nous allons réutiliser le moteur d'émoticones développé au cours du chapitre 16 intitulé *Le texte*.

Nous retrouvons à nouveau l'intérêt de la programmation orientée objet en facilitant la réutilisation d'objets prédéfinis.

La figure 18-3 illustre le fonctionnement d'un chat multi utilisateur, comme nous l'avons vu précédemment, le serveur agit comme relais entre les différents clients connectés :

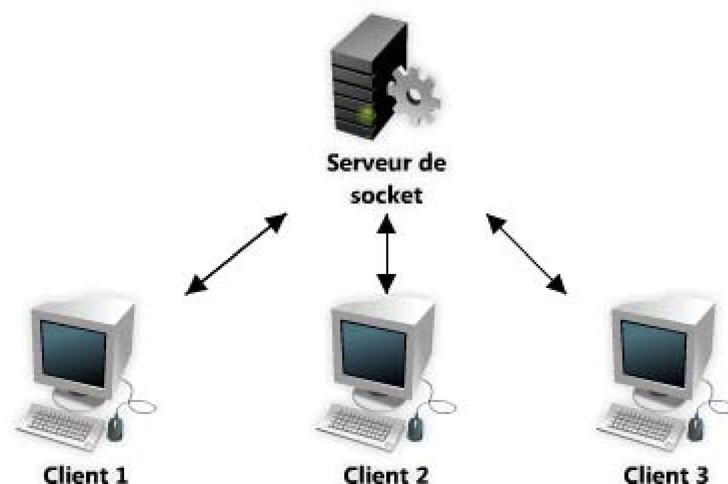

Figure 18-3. T'chat multi-utilisateurs.

Au sein d'un fichier nommé `ServeurMessagerieXML.php` nous définissons le code suivant :

```
#!/usr/bin/php -q
<?php

set_time_limit(0);

$adresse = 'localhost';
$port = 10000;

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_set_option($connexion, SOL_SOCKET,SO_REUSEADDR, 1);

$ret = socket_bind($connexion, $adresse, $port);

$ret = socket_listen($connexion, 5);

$tabSockets = array($connexion);

while (true)
{

  $sockets = $tabSockets;

  socket_select($sockets, $write = NULL, $except = NULL, NULL);

  foreach($sockets as $socket)
  {
```

```
            if ($socket == $connexion)
            {

                if (($client = socket_accept($connexion)) < 0) continue;

                else array_push($tabSockets, $client);

            } else
            {

                $flux = socket_recv($socket, $buffer, 2048, 0);

                if ($flux == 0)
                {

                    $index = array_search($socket, $tabSockets);
                    unset($tabSockets[$index]);
                    socket_close($socket);

                }else
                {

                    $allclients = $tabSockets;
                    array_shift($allclients);
                    send_Message($allclients, $socket, $buffer);

                }

            }

        }

    }

    function send_Message($clients, $socket, $donnees)
    {

      foreach($clients as $client)
      {

      socket_write($client, $donnees);

      }

    }
    ?>
```

Nous ne démarrons pas le serveur pour le moment, nous allons développer à présent la partie client de l'application de messagerie instantanée.

Lors du chapitre 16, nous avions développé une classe `MoteurEmoticone`, celle-ci va nous permettre d'afficher le texte de la conversation. Grâce à ses fonctionnalités, les utilisateurs auront la possibilité d'utiliser un ensemble d'émoticones prédéfinis.

Nous ajoutons les lignes suivantes à la classe `MoteurEmoticone` :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.KeyboardEvent;
  import flash.geom.Rectangle;
  import flash.text.TextField;
  import flash.ui.Keyboard;
  import org.bytearray.evenements.EvenementSaisie;
  import org.bytearray.evenements.EvenementMessage;

  public class MoteurEmoticone extends Sprite

  {

        public var contenuHistorique:TextField;
        public var chaineHistorique:String;
        public var contenuSaisie:TextField;
        public var codesTouches:Array;
        public var tableauSmileys:Array;
        public var lngTouches:int;
        public var coordonnees:Rectangle;
        public var emoticone:Emoticone;

        public function MoteurEmoticone ()

        {

              chaineHistorique = new String();

              codesTouches = new Array (":)",":D",":(",";)",":p");

              tableauSmileys = new Array ();

              lngTouches = codesTouches.length;

              addEventListener ( KeyboardEvent.KEY_DOWN, envoiMessage );

              contenuHistorique.addEventListener ( Event.SCROLL,
saisieUtilisateur );

        }

        private function envoiMessage ( pEvt:KeyboardEvent ):void

        {

              if ( pEvt.keyCode == Keyboard.ENTER )

              {

                    dispatchEvent ( new EvenementSaisie (
EvenementSaisie.ENVOI_MESSAGE, contenuSaisie.text ) );

                    contenuSaisie.htmlText = "";

              }

        }
```

```
		public function afficheMessage ( pEvt:EvenementMessage ):void

		{

			chaineHistorique += pEvt.message;

			contenuHistorique.text = chaineHistorique;

			contenuHistorique.dispatchEvent ( new Event ( Event.SCROLL ) );

		}

		private function saisieUtilisateur ( pEvt:Event ):void

		{

			var i:int;
			var j:int;

			var nombreSmileys:int = tableauSmileys.length;

			for ( i = 0; i< nombreSmileys; i++ ) removeChild (
tableauSmileys[i] );

			tableauSmileys = new Array();

			for ( i = 0; i<lngTouches; i++ )
			{

				j = pEvt.target.text.indexOf ( codesTouches[i] );

				while ( j!= -1 )
				{

					coordonnees = pEvt.target.getCharBoundaries ( j );

					if ( coordonnees != null ) tableauSmileys.push (
ajouteSmiley ( coordonnees, i ) );

					j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

				}

			}

		}

		private function ajouteSmiley ( pRectangle:Rectangle, pIndex:int
):Emoticone

		{

			emoticone = new Emoticone();
			emoticone.gotoAndStop ( pIndex + 1 );

			emoticone.x = pRectangle.x + 1;
			emoticone.y = pRectangle.y - ((contenuHistorique.scrollV -
1)*(contenuHistorique.textHeight/contenuHistorique.numLines))+1;

			addChild ( emoticone );

			return emoticone;
```

```
        }
    }
}
```

La classe MoteurEmoticone est liée à un clip contenant deux champs texte contenuHistorique et contenuSaisie.

Afin de pouvoir informer le reste de l'application de la saisie utilisateur, la classe MoteurEmoticone diffuse un événement EvenementSaisie.ENVOI_MESSAGE.

Au sein du paquetage org.bytearray.evenements nous définissons la classe EvenementSaisie suivante :

```
package org.bytearray.evenements
{
  import flash.events.Event;
  public class EvenementSaisie extends Event
  {
        public static const ENVOI_MESSAGE:String = "envoiMessage";
        public var saisie:String;
        public function EvenementSaisie ( pType:String, pSaisie:String )
        {
              super( pType, false, false );
              saisie = pSaisie;
        }
        public override function clone ():Event
        {
              return new EvenementSaisie ( type, saisie );
        }
        public override function toString ():String
        {
              return '[EvenementSaisie type="'+ type +'" bubbles=' + bubbles +
' eventPhase='+ eventPhase + ' cancelable=' + cancelable +' saisie=' + saisie
+']';
        }
  }
```

```
}
```

La classe EvenementMessage permet d'informer l'application de l'arrivée de nouveaux messages.

Au sein du même paquetage que la classe EvenementSaisie nous définissons la classe EvenementMessage suivante :

```
package org.bytearray.evenements

{

  import flash.events.Event;

  public class EvenementMessage extends Event

  {

       public static const RECEPTION_MESSAGE:String = "receptionMessage";
       public var message:String;

       public function EvenementMessage ( pType:String, pMessage:String )

       {

            super( pType, false, false );

            message = pMessage;

       }

       public override function clone ():Event

       {

            return new EvenementMessage ( type, message );

       }

       public override function toString ():String

       {

            return '[EvenementMessage type="'+ type +'" bubbles=' + bubbles +
' eventPhase='+ eventPhase + ' cancelable=' + cancelable +' message=' +
message + ']';

       }

  }

}
```

Afin de tester la classe MoteurEmoticone nous associons la classe de document suivante à un nouveau document Flash CS3 :

```
package org.bytearray.document

{

  import org.bytearray.abstrait.ApplicationDefaut;
```

```
import org.bytearray.emoticones.MoteurEmoticone;

public class Document extends ApplicationDefaut
{

     public var client:MoteurEmoticone;

     public function Document ()

     {

          client = new MoteurEmoticone();

          client.x = (stage.stageWidth - client.width) >> 1;
          client.y = (stage.stageHeight - client.height) >> 1;

          addChild ( client );

     }

}

}
```

La figure 18-4 illustre l'interface de t'chat :

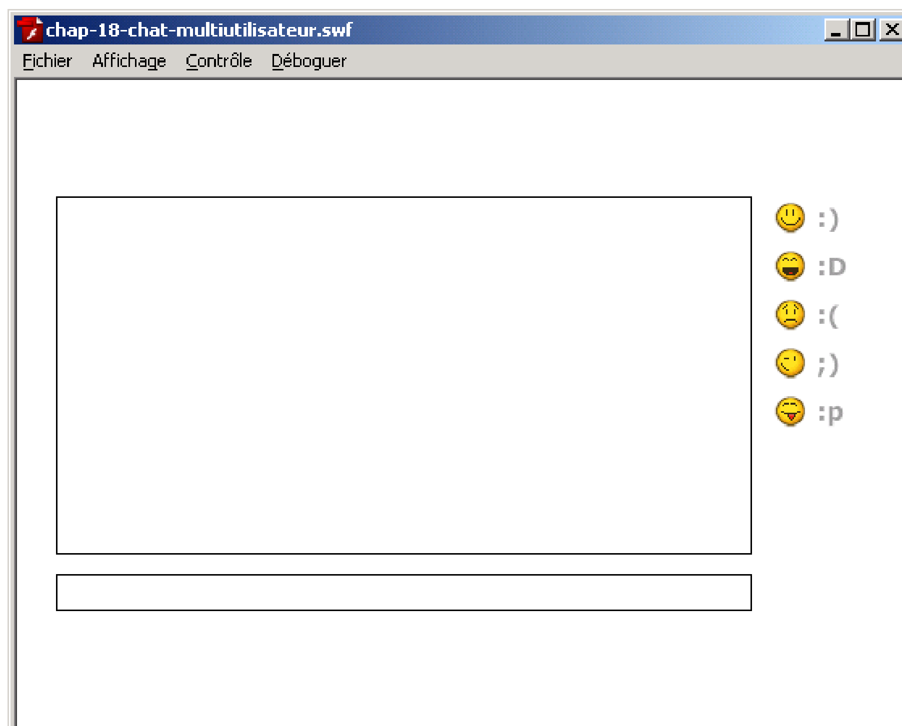

Figure 18-4. Client chat multi-utilisateur.

L'objet `MoteurEmoticone` diffuse un événement `EvenementSaisie.ENVOI_MESSAGE` que nous écoutons afin de récupérer le texte saisi :

```
package org.bytearray.document
```

```
{

  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.emoticones.MoteurEmoticone;
  import org.bytearray.evenements.EvenementSaisie;

  public class Document extends ApplicationDefaut
  {

       public var client:MoteurEmoticone;

       public function Document ()

       {

            client = new MoteurEmoticone();
            client.x = (stage.stageWidth - client.width) >> 1;
            client.y = (stage.stageHeight - client.height) >> 1;

            addChild ( client );

            client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

       }

       private function envoiMessage ( pEvt:EvenementSaisie ):void

       {

            // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
            trace( pEvt );

       }

  }

}
```

Puis nous envoyons les données au serveur de socket grâce à la méthode `send` de l'objet `XMLSocket` :

```
package org.bytearray.document

{

  import flash.events.DataEvent;
  import flash.net.XMLSocket;
  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.emoticones.MoteurEmoticone;
  import org.bytearray.evenements.EvenementSaisie;

  public class Document extends ApplicationDefaut
  {

       private static const ADRESSE:String = "localhost";
       private static const PORT:int = 10000;

       public var client:MoteurEmoticone;
       public var connexionSocket:XMLSocket;
```

```
        public function Document ()

        {

             client = new MoteurEmoticone();
             client.x = (stage.stageWidth - client.width) >> 1;
             client.y = (stage.stageHeight - client.height) >> 1;

             addChild ( client );

             client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

             connexionSocket = new XMLSocket ( Document.ADRESSE, Document.PORT
);

             connexionSocket.addEventListener ( DataEvent.DATA,
receptionDonnees );

        }

        private function envoiMessage ( pEvt:EvenementSaisie ):void

        {

             // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
             trace( pEvt );

             // envoi des données auprès du serveur de socket
             connexionSocket.send ( pEvt.saisie );

        }

        private function receptionDonnees ( pEvt:DataEvent ):void

        {

             // affiche : salut !
             trace( pEvt.data );

        }

  }

}
```

Le principe est très simple, aussitôt les données envoyées, le serveur de socket transmet le texte saisi par un des participants à tous les clients connectés.

Pour que l'objet `MoteurEmoticone` soit averti des messages provenant du serveur et affiche la conversation, nous diffusons un événement `EvenementMessage.RECEPTION_MESSAGE` :

```
package org.bytearray.document

{

  import flash.events.DataEvent;
```

```
import flash.net.XMLSocket;
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.emoticones.MoteurEmoticone;
import org.bytearray.evenements.EvenementMessage;
import org.bytearray.evenements.EvenementSaisie;

public class Document extends ApplicationDefaut
{

      private static const ADRESSE:String = "localhost";
      private static const PORT:int = 10000;

      public var client:MoteurEmoticone;
      public var connexionSocket:XMLSocket;

      public function Document ()

      {

            client = new MoteurEmoticone();
            client.x = (stage.stageWidth - client.width) >> 1;
            client.y = (stage.stageHeight - client.height) >> 1;

            addChild ( client );

            client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

            // écoute de l'événement EvenementMessage.RECEPTION_MESSAGE afin
d'afficher le texte
            addEventListener ( EvenementMessage.RECEPTION_MESSAGE,
client.afficheMessage );

            connexionSocket = new XMLSocket ( Document.ADRESSE, Document.PORT
);

            connexionSocket.addEventListener ( DataEvent.DATA,
receptionDonnees );

      }

      private function envoiMessage ( pEvt:EvenementSaisie ):void

      {

            // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
            trace( pEvt );

            // envoi des données auprès du serveur de socket
            connexionSocket.send ( pEvt.saisie );

      }

      private function receptionDonnees ( pEvt:DataEvent ):void

      {

            // diffusion d'un événement EvenementMessage.RECEPTION_MESSAGE
afin d'afficher les données reçues du serveur
            dispatchEvent ( new EvenementMessage (
EvenementMessage.RECEPTION_MESSAGE, pEvt.data ) );
```

```
        }
    }
}
```

En testant notre client nous remarquons que le texte est bien affiché lorsque nous l'envoyons, comme l'illustre la figure 18-5 :

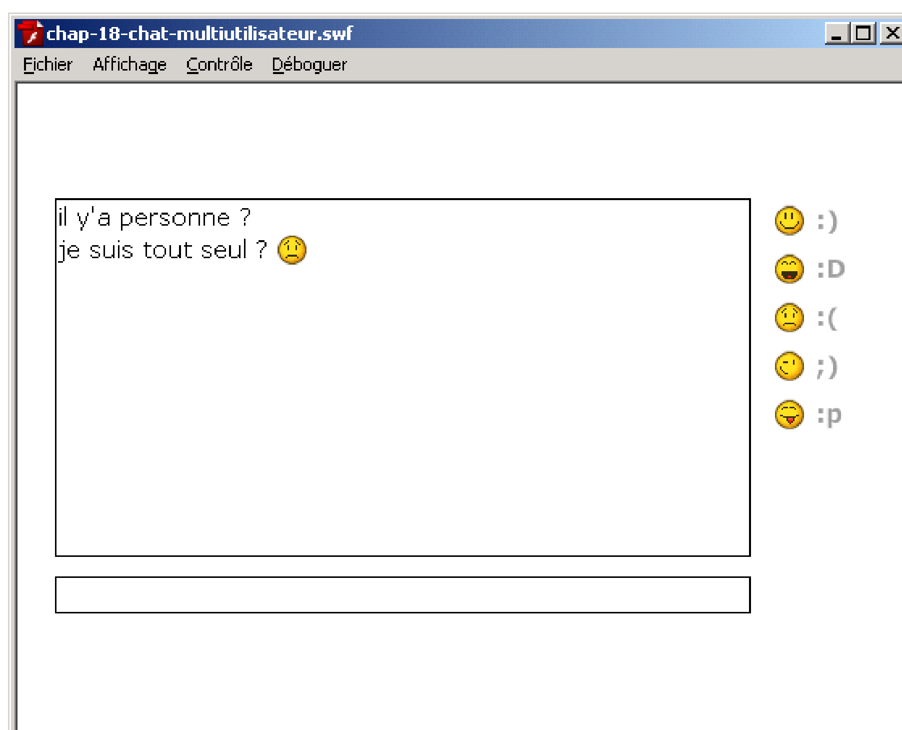

Figure 18-5. Client flash connecté.

Si nous connectons un deuxième client, notre chat multi utilisateur fonctionne. Vous pouvez à présent déployer le serveur de socket ainsi que le client Flash sur votre serveur et diffuser l'adresse aux personnes avec qui vous souhaitez discuter.

La classe `XMLSocket` permet le développement d'applications temps réel de manière très souple. Bien entendu, de nombreux serveurs existent déjà et sont conçus pour fonctionner avec Flash à l'aide de la classe `XMLSocket`.

Parmi ceux là nous pouvons citer les serveurs suivants :

- Unity : serveur de socket XML payant de Colin Moock - http://www.moock.org/unity
- SmartFox Server : serveur de socket XML payant - http://www.smartfoxserver.com

- Jabber : serveur de socket XML gratuit destiné à la création d'applications de messageries instantanées - http://www.jabber.org

Une des limitations de la classe XMLSocket est liée à limitation du format des données échangées.

Nous allons voir dans cette nouvelle partie dans quelle mesure la classe Socket peut s'avérer utile.

A retenir

- La classe XMLSocket permet d'échanger des données au format XML et texte entre un client et un serveur de socket.
- En tant que relais, le serveur de socket permet de faire transiter des informations entre plusieurs clients connectés.
- Chaque message est terminé par un octet nul.

La classe Socket

ActionScript 3 intègre une nouvelle classe Socket se différenciant quelque peu de la classe XMLSocket.

Comme nous l'avons vu précédemment, la classe XMLSocket transmet chaque message en le terminant par un octet nul. Ce comportement provoque un troncage des données binaire.

La classe Socket ne souffre pas de cette limitation et offre la possibilité de transférer un flux binaire brut sans altérations.

Si nous devions schématiser le fonctionnement d'un serveur de socket binaire nous pourrions l'illustrer de la manière suivante :

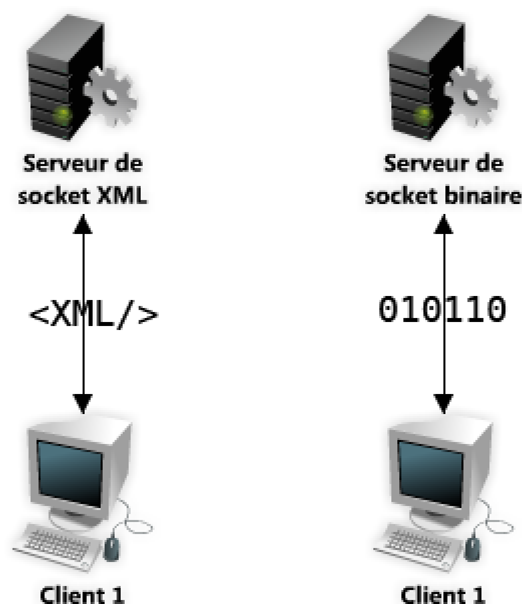

Figure 18-6. `XMLSocket` et `Socket`.

Nous allons ensemble développer un serveur de socket binaire afin de transférer par une connexion socket un élément graphique telle une image ou un SWF.

Outre l'expérimentation technique, ce procédé permet d'éviter la mise en cache des éléments graphiques au sein du navigateur. L'élément graphique est chargé directement en mémoire par le lecteur Flash puis affiché, cette approche évite la mise en cache de fichiers SWF et rend leur décompilation difficile.

A retenir

- La classe `Socket` permet d'échanger des données au format binaire brutes.
- Grâce au caractère bas niveau de la classe `Socket`, il est possible d'implémenter n'importe quel protocole d'application.

Créer un serveur de socket binaire

Nous allons à nouveau utiliser PHP afin de créer notre serveur de socket binaire. L'intérêt de ce dernier sera de pouvoir recevoir des requêtes et d'y répondre. Nous allons ainsi définir notre propre protocole afin de normaliser les échanges entre les deux acteurs.

Dans un fichier PHP nommé `ServeurSocket.php` nous définissons le code suivant :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

socket_close($client);
socket_close($connexion);

?>
```

Une fois le serveur démarré nous pouvons nous y connecter de la manière suivante :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );

function clientConnecte ( pEvt:Event ):void

{

  trace("client connecté");

}

function clientDeconnecte ( pEvt:Event ):void

{

  trace("client déconnecté");

}
```

Si nous testons le code précédent, nous remarquons que la connexion est aussitôt fermée.

Le panneau de sortie affiche les messages suivants :

```
client connecté
client déconnecté
```


Thibault Imbert
pratiqueactionscript3.bytearray.org

Nous modifions le code du serveur de socket binaire en ajoutant un appel à la méthode `socket_write` pour envoyer les données au client en cours :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

$messageSortant = "Bienvenue sur le serveur de socket, mais désolé vous êtes déconnecté !";

socket_write ($client, $messageSortant );

socket_close($client);
socket_close($connexion);

?>
```

Nous envoyons désormais des données depuis le serveur avant de clore la connexion.

Lorsque le serveur transmet les données, l'événement `ProgressEvent.SOCKET_DATA` est diffusé :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );

function clientConnecte ( pEvt:Event ):void

{

  trace("client connecté");

}

function donneesRecues ( pEvt:ProgressEvent ):void

{
```


Thibault Imbert
pratiqueactionscript3.bytearray.org

```
  trace("données reçues");

}

function clientDeconnecte ( pEvt:Event ):void

{

  trace("client déconnecté");

}
```

Si nous testons le code précédent nous voyons que le client parvient à se connecter, puis les données sont reçues, enfin le client est automatiquement déconnecté.

Le panneau de sortie affiche les messages suivants :

```
client connecté
données reçues
client déconnecté
```

Contrairement à la classe `XMLSocket`, les données reçues par la classe `Socket` doivent être décodées.

Afin de lire les données transmises par le serveur nous utilisons ici la méthode `readUTFBytes` de l'objet `Socket` dont voici la signature :

```
public function readUTFBytes(length:uint):String
```

Celle-ci accepte un paramètre `length` correspondant au nombre d'octets à lire et à renvoyer sous forme de chaîne de caractères UTF-8.

Afin de lire les données disponibles, nous utilisons la propriété `bytesAvailable` :

```
function donneesRecues ( pEvt:ProgressEvent ):void

{

  // affiche : Bienvenue sur le serveur de socket, mais désolé vous êtes
déconnecté !
  trace( pEvt.target.readUTFBytes ( pEvt.target.bytesAvailable ) );

}
```

N'oubliez pas que dans un contexte de transfert de données TCP/IP, les données arrivent par paquet d'octets. Il est donc nécessaire de lire le flux de données au fur et à mesure que les données arrivent au sein du lecteur. Celles-ci s'accumulent au sein de l'objet `Socket` et peuvent être lues à l'aide des différentes méthodes définies par la classe `Socket`.

La propriété `bytesAvailable` nous permet de récupérer le nombre d'octets disponibles actuellement au sein du flux téléchargé et évite de sortir du flux de données.

Si nous tentons de lire des octets non disponibles, l'erreur suivante est levée à l'exécution :

```
Error: Error #2030: Fin de fichier détectée.
```

Nous reviendrons en détail sur les différentes méthodes de lecture et d'écriture de flux binaire au cours du chapitre 20 intitulé *ByteArray*.

A retenir

- Contrairement à la classe `XMLSocket`, la classe `Socket` reçoit un flux binaire brut de la part du serveur.
- Ces données doivent être décodées à l'aide des différentes méthodes de la classe `Socket`.

Echanger des données

Nous avons évoqué en début de chapitre la notion de protocole d'application afin d'échanger des messages entre un client un serveur.

Les protocoles d'applications tels FTP, SMTP, ou POP s'appuient sur des messages prédéfinis afin d'appeler certaines fonctionnalités auprès du serveur de socket. Nous allons reproduire ce même comportement en demandant à notre serveur de socket de transférer un fichier graphique présent à ses côtés.

En passant la chaîne de caractère `"ENVOIE"` suivie du nom du fichier, le serveur renverra le flux du fichier demandé.

Afin d'intégrer ce protocole d'application nous modifions les lignes suivantes au sein du serveur de socket binaire :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

$actif = true;

$COMMANDE_ENVOIE = "ENVOIE";

do
```

```
{

  $actif = socket_read($client, 1024, PHP_BINARY_READ);

  $array = split (" ", trim($actif));
  $commande = $array[0];
  $fichier = $array[1];

  $pointeur = fopen ($fichier, "rb");
  $donnees = fread ($pointeur, filesize ($fichier));
  fclose ($pointeur);

  if ( $commande == $COMMANDE_ENVOIE )
  {

       $longueur = pack('N', strlen($donnees));
       socket_write($client, $longueur);
       socket_write($client, $donnees);

  } else socket_write ($client, "commande non reconnue");

} while ( $actif );

socket_close($client);
socket_close($connexion);

?>
```

Rappelez vous d'un point essentiel, le protocole TCP/IP transfère les données par paquets, c'est à nous de nous assurer que la totalité des données ont été transférées.

Il est donc impératif de récupérer l'ensemble des paquets cotés client avant de tenter d'afficher le fichier transféré :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion );

// demande du chargement du fichier
connexion.writeUTFBytes ("ENVOIE:DSC02602.JPG\r\n");
connexion.flush();

function clientConnecte ( pEvt:Event ):void

{

  trace("client connecté");

}

function clientDeconnecte ( pEvt:Event ):void
```

```
{
  trace("client déconnecté");
}
function erreurConnexion ( pEvt:Event ):void
{
  trace("erreur de connexion au serveur");
}
// création d'un tableau binaire pour contenir les données entrantes
var donnees:ByteArray = new ByteArray();

// nombre d'octets totaux à transférer
var longueur:Number;

function donneesRecues ( pEvt:ProgressEvent ):void
{
  // nous récupérons le nombre d'octets totaux
  if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

  // les données sont progressivement stockées au sein du tableau donnees
  pEvt.target.readBytes ( donnees, donnees.length, pEvt.target.bytesAvailable
);

  // données chargées
  if ( donnees.length ==  longueur ) trace ("transfert des données
terminée");
}
```

Grâce à la méthode `writeUTFBytes`, nous encodons la chaîne de caractère passée en paramètre en flux binaire. Afin de transmettre celle-ci au serveur nous poussons les données à l'aide de la méthode `flush`.

Aussitôt la commande reçue, le serveur renvoie la longueur totale des données à recevoir, puis le flux de l'élément graphique est transmis.

Afin de sauver les données entrantes, nous créons un tableau de sauvegarde dans lequel nous plaçons les données téléchargées lors de la diffusion de l'événement `ProgressEvent.SOCKET_DATA`.

Souvenez-vous, nous avions découvert l'objet `Loader` lors du chapitre 13, ce dernier permettait le chargement de contenu externe. La classe `Loader` définit une méthode `loadBytes` permettant de rendre graphiquement un flux binaire passé en paramètre.

Voici la signature de la méthode `loadBytes` :

```
public function loadBytes(bytes:ByteArray, context:LoaderContext = null):void
```

Détail de chacun des paramètres :

- bytes : le flux binaire à rendre graphiquement.
- context : le contexte de chargement, lié au modèle de sécurité.

Attention, la méthode loadBytes n'accepte que des flux binaires compatibles avec l'objet Loader.

En d'autres termes, seuls les flux d'images ou de SWF pourront être affichés. Si un flux non compatible est passé à la méthode loadBytes, une erreur de type IOErrorEvent.IO_ERROR est levée.

Dans le code suivant, un objet Loader est crée. Une fois les données totalement téléchargées, nous les injectons au sein de ce dernier :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion );

// demande du chargement du fichier
connexion.writeUTFBytes ("ENVOIE:DSC02602.JPG\r\n");
connexion.flush();

function clientConnecte ( pEvt:Event ):void

{

  trace("client connecté");

}

function clientDeconnecte ( pEvt:Event ):void

{

  trace("client déconnecté");

}

function erreurConnexion ( pEvt:Event ):void

{

  trace("erreur de connexion au serveur");

}

var chargeur:Loader = new Loader();
```

```
addChild ( chargeur );

// création d'un tableau binaire pour contenir les données entrantes
var donnees:ByteArray = new ByteArray();

// nombre d'octets totaux à transférer
var longueur:Number;

function donneesRecues ( pEvt:ProgressEvent ):void

{

  // nous récupérons le nombre d'octets totaux
  if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

  // les données sont progressivement stockées au sein du tableau donnees
  pEvt.target.readBytes ( donnees, donnees.length, pEvt.target.bytesAvailable
);

  // lorsque le flux binaire est totalement chargé nous l'affichons grâce à
l'objet Loader
  if ( donnees.length ==  longueur ) chargeur.loadBytes( donnees );

}
```

En testant le code précédent nous remarquons que le fichier est téléchargé par la connexion socket puis affiché au sein du lecteur.

Une fois les données injectées, l'objet `Loader` diffuse un événement `Event.COMPLETE`.

Dans le code suivant, nous redimensionnons l'image et la centrons une fois le flux affiché :

```
chargeur.contentLoaderInfo.addEventListener( Event.COMPLETE,
injectionTerminee );

function injectionTerminee ( pEvt:Event ):void

{

  var contenu:DisplayObject = pEvt.target.content;
  var objetChargeur:Loader = pEvt.target.loader;

  if ( contenu is Bitmap ) Bitmap ( contenu ).smoothing = true;

  var ratio:Number = Math.min ( 350 / pEvt.target.content.width, 350 /
pEvt.target.content.height );

  objetChargeur.scaleX = objetChargeur.scaleY = ratio;

  objetChargeur.x = (stage.stageWidth - objetChargeur.width) / 2;
  objetChargeur.y = (stage.stageHeight - objetChargeur.height) / 2;

}
```

Le résultat est illustré par la figure 18-7 :

Figure 18-7. Image chargée par connexion socket.

Il serait intéressant de rendre cette communication transparente au sein d'un objet spécifique. Nous allons créer une classe ChargeurFlux qui procèdera en interne aux différentes commandes du protocole.

Au sein d'un paquetage org.bytearray.chargeur nous définissons la classe ChargeurFlux suivante :

```
package org.bytearray.chargeur

{

  import flash.events.EventDispatcher;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.Socket;

  public class ChargeurFlux extends EventDispatcher

  {

       private var connexion:Socket;

       public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )

       {

            connexion = new Socket(pAdresse, pPort);

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
```

```
            connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
            connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

        }

        private function redirigeEvenement ( pEvt:Event ):void

        {

            dispatchEvent ( pEvt );

        }

        private function transfertDonnees ( pEvt:ProgressEvent ):void

        {


        }

  }

}
```

Puis nous ajoutons un objet `ByteArray` interne afin d'accueillir le flux téléchargé :

```
package org.bytearray.chargeur

{

  import flash.events.EventDispatcher;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.Socket;
  import flash.utils.ByteArray;

  public class ChargeurFlux extends EventDispatcher

  {

        private var connexion:Socket;
        private var donnees:ByteArray;

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )

        {

            connexion = new Socket(pAdresse, pPort);

            donnees = new ByteArray();

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
            connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
            connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

        }
```

```
        private function redirigeEvenement ( pEvt:Event ):void

        {

             dispatchEvent ( pEvt );

        }

        private function transfertDonnees ( pEvt:ProgressEvent ):void

        {


        }

  }

}
```

Puis nous implémentons les méthodes `charge` et `connecte` :

```
package org.bytearray.chargeur

{

  import flash.events.EventDispatcher;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.Socket;
  import flash.utils.ByteArray;

  public class ChargeurFlux extends EventDispatcher

  {

        private var connexion:Socket;
        private var donnees:ByteArray;

        private static const ENVOIE:String = "ENVOIE";

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )

        {

             connexion = new Socket(pAdresse, pPort);

             donnees = new ByteArray();

             connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
             connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
             connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

        }

        public function charge ( pFichier:String ):void

        {
```

```
            connexion.writeUTFBytes ( ChargeurFlux.ENVOIE + " " + pFichier +
"\r\n" );
            connexion.flush();

        }

        public function connecte ( pAdresse:String, pPort:int ):void

        {

            connexion.connect ( pAdresse, pPort );

        }

        private function redirigeEvenement ( pEvt:Event ):void

        {

            dispatchEvent ( pEvt );

        }

        private function transfertDonnees ( pEvt:ProgressEvent ):void

        {


        }

  }

}
```

Enfin, nous ajoutons la logique associée afin de déterminer la fin du chargement du flux :

```
package org.bytearray.chargeur

{

  import flash.events.EventDispatcher;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.Socket;
  import flash.utils.ByteArray;
  import org.bytearray.chargeur.evenements.EvenementChargeurFlux;

  public class ChargeurFlux extends EventDispatcher

  {

        private var connexion:Socket;
        private var donnees:ByteArray;
        private var longueur:Number;

        private static const ENVOIE:String = "ENVOIE";

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )

        {
```

```
            connexion = new Socket(pAdresse, pPort);

            donnees = new ByteArray();

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
            connexion.addEventListener ( ProgressEvent.SOCKET_DATA, 
transfertDonnees );
            connexion.addEventListener ( IOErrorEvent.IO_ERROR, 
redirigeEvenement );

       }

       public function charge ( pFichier:String ):void

       {

            connexion.writeUTFBytes ( ChargeurFlux.ENVOIE + " " + pFichier + 
"\r\n" );
            connexion.flush();

       }

       public function connecte ( pAdresse:String, pPort:int ):void

       {

            connexion.connect ( pAdresse, pPort );

       }

       private function redirigeEvenement ( pEvt:Event ):void

       {

            dispatchEvent ( pEvt );

       }

       private function transfertDonnees ( pEvt:ProgressEvent ):void

       {

            if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

            pEvt.target.readBytes ( donnees, donnees.length, 
pEvt.target.bytesAvailable );

            if ( donnees.length == longueur )

            {

                 dispatchEvent ( new EvenementChargeurFlux ( 
EvenementChargeurFlux.TERMINE, donnees ) );
                 donnees = new ByteArray();

            }

       }

  }

}
```

Afin de charger un élément graphique stocké sur le serveur, nous instancions simplement l'objet ChargeurFlux puis nous appelons sa méthode charge :

```
import org.bytearray.chargeur.ChargeurFlux;
import org.bytearray.chargeur.evenements.EvenementChargeurFlux;

var monChargeur:ChargeurFlux = new ChargeurFlux("localhost", 10000);

monChargeur.charge ("blason.png");

var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener( Event.COMPLETE,
injectionTerminee );

addChild ( chargeur );

monChargeur.addEventListener ( EvenementChargeurFlux.TERMINE, affiche );

function affiche ( pEvt:EvenementChargeurFlux ):void

{

  chargeur.loadBytes( pEvt.donnees );

}

function injectionTerminee ( pEvt:Event ):void

{

  var contenu:DisplayObject = pEvt.target.content;
  var objetChargeur:Loader = pEvt.target.loader;

  if ( contenu is Bitmap ) Bitmap ( contenu ).smoothing = true;

  var ratio:Number = Math.min ( 350 / pEvt.target.width, 350 /
pEvt.target.height );

  objetChargeur.scaleX = objetChargeur.scaleY = ratio;

  objetChargeur.x = (stage.stageWidth - objetChargeur.width) / 2;
  objetChargeur.y = (stage.stageHeight - objetChargeur.height) / 2;

}
```

La classe ChargeurFlux peut ainsi être utilisée pour le chargement de fichiers SWF afin d'éviter leur mise en cache au sein du navigateur et rendre leur décompilation moins facile :

```
monChargeur.charge ("monsite.swf");
```

A vous d'imaginer de nouvelles fonctionnalités !

A retenir

- La méthode `writeUTFBytes` permet d'écrire une chaîne de caractères au format binaire.
- La méthode `flush` permet d'envoyer les données binaires au serveur de socket.

Dans le prochain chapitre nous découvrirons la technologie Flash Remoting afin de dialoguer de manière optimisée avec différents langages serveurs et bases de données.