Perl::IRC - 1/3

Ecrire un bot IRC en Perl

Cet article est le premier d'une série de trois dont le but est de vous présenter une solution permettant, en Perl, d'écrire un Bot IRC.
Ici, nous verrons comment créer un petit programme se connectant à IRC. Le prochain article nous permettra de rajouter quelques fonctionnalités à celui-ci. Et enfin, au cours du troisième, nous verrons quelques astuces permettant d'améliorer son fonctionnement.

Avant que nous ne nous lancions dans le vif du sujet, j'aimerais insister sur le fait que ces articles demandent de votre part quelques pré-requis. Notamment, je suppose que vous connaissez déjà assez bien le Perl, et ne ferai pas de rappel concernant les bases de ce langage de programmation.
Si vous ne connaissez pas le Perl, je ne peux que vous encourager à consulter un tutorial, tel celui de Sylvain Lhullier, intitulé Introduction à la programmation en Perl, qui est relativement complet, et somme toute assez facile à lire, ou alors, carrément, à trouver un livre traitant de Perl, ce qui n'est pas non plus une mauvaise idée.

Encore une chose : comme tout programmeur Perl le sait (ou devrait le savoir ?), TMTOWTDI - *"There's More Than One Way To Do It"*.
Ce que je veux dire par là, c'est qu'il existe 36 façons différentes de coder quelque chose... et 36 façons différentes d'écrire chacune de ces façons de coder...
Au cours des exemples que je donnerai, j'ai choisi à chaque fois une implémentation, en essayant de trouver un bon compromis entre efficacité et lisibilité : après tout, le but de mes exemples est qu'ils soient compréhensibles, même si ça ne les rend pas forcément agréables à regarder !

Perl ?

IRC étant un protocole en mode texte, du moins pour tout ce qui est discussions, comme nous l'indique la RFC le décrivant, la logique veut qu'un langage adapté à la manipulation de données textes soit plutôt bien adapté à l'écriture d'un programme en rapport avec IRC.
Voila pourquoi j'ai choisi, pour ces quelques articles, d'utiliser le langage Perl, dont le nom est l'acronyme de "Practical Extraction and Report Language".

D'autre part, il existe déjà, en Perl, une librairie permettant de se connecter à IRC ; autrement dit, une librairie qui fait une bonne partie de ce que nous voulons faire ici !
Il s'agit de la librairie Net::IRC, que nous utiliserons au cours de cette série d'articles.
Bien évidemment, nous aurions pu tout recoder en partant de la RFC et des sockets... Mais est-ce que cela n'aurait pas été une perte de temps inutile... et quelque peu ridicule ? Après tout, une des premières qualités d'un programmeur est la (bonne) paresse, n'est-ce pas ?

Si vous êtes sous un sytème Linux, installer Perl est simple : "apt-get install perl", pour les amateurs de Debian, ou équivalent selon votre distribution.
Sous un système windows, il vous faudra installer une version de Perl telle ActivePerl, distribuée gratuitement par ActiveState.
Cela dit, je ne rentrerai pas plus dans les détails, considérant, comme je l'ai déjà dit plus haut, que vous avez déjà un certain niveau en Perl.

Pas à pas

Tout d'abord, vous noterez que ce script a écrit écrit sous une version d'UNIX ; un Linux, plus précisément ; et UNIX offre la possibilité de préciser, en première ligne d'un script, quel interpréteur doit l'exécuter (On appelle cette ligne le Shebang (en)).
Notez que le chemin vers votre exécutable Perl peut varier selon la distribution que vous utilisez, ou les choix effectués par votre administrateur (Il arrive notamment qu'il ait choisi de le placer dans un sous-répertoire de /usr/).

Vous ne manquerez pas de remarquer que ce script est programmé en mode strict (de la sorte, l'emploi d'une variable non déclarée entraine l'affichage d'un message d'erreur, ce qui évite toute erreur - entre autres), et en mode warnings.
Même si cela implique d'être plus attentif lorsqu'on programme, cela force à devoir programmer plus proprement. Vous verrez donc toujours ces deux pragmas activés dans mes scripts, du moins pour ceux de plus de quelques lignes... Et je ne peux que vous encourager **fortement** à faire de même.

```
#!/usr/bin/perl

use strict;
use warnings;
```

Ensuite, puisque, comme on l'a dit plus haut, on utilise la librairie Net::IRC pour se connecter à IRC, il nous faut l'inclure :

```
# On utilise la librairie Net::IRC pour se connecter à IRC
use Net::IRC;
```

Maintenant, nous allons configurer notre Bot.
Tout d'abord, nous devons indiquer à quel serveur IRC nous souhaitons nous connecter. Pour cet exemple, j'ai choisi irc.worldnet.net, que j'ai l'habitude d'utiliser ; mais libre à vous d'en choisir un autre, éventuellement sur un autre réseau - en théorie, le bot devrait aussi fonctionner, même si je n'ai pas testé sur tous les serveurs existant.
Ensuite, il faut bien que notre Bot ait un nom d'utilisateur, visible pour les autres personnes connectées ; dans le cadre de cet exemple, j'ai choisi de le nommer "perlBot_v1" (très original, je l'admet). Notons que chaque nickname (pseudonyme) ne peut être utilisé que par un seul utilisateur à la fois sur un réseau ; il est donc possible qu'il vous faille en choisir un autre.
Après quoi, nous configurons quelques informations supplémentaires, telles le nom d'utilisateur IRC (généralement visible seulement pour les utilisateurs le demandant), son numéro de version, ...

```
# Configuration des options de connexion (serveur, login) :
my $server = 'irc.worldnet.net';
my $nick = 'perlBot_v1';


# Informations concernant le Bot :
my $ircname = 'Bot IRC Perl';
my $username = 'perlBot';
my $version = '1.0';
```

Pour simplifier les choses, dans cet article, notre bot ne sera capable de joindre qu'un seul channel à la fois, contrairement à ce qu'un utilisateur humain aurait tendance à faire (j'ai moi-même généralement plus d'une demi-douzaine de channels ouverts simultanément).
Ici, j'ai choisi de lui faire joindre le channel #krypton. Encore une fois, libre à vous de modifier ceci.

Cela dit, tant que vous testez, je vous recommande de ne pas aller sur un channel fréquenté : le fait de sans cesse joindre puis quitter puis revenir, ... ferait probablement que votre Bot (et vous-même) serait vite détesté par les autres utilisateurs !

```
# Channel sur lequel on veut que le Bot aille :
my $channel = '#krypton';
```

C'est mainetant que tout commence : la librairie Net::IRC est orientée objet... Et nous allons, maintenant que nous avons défini la configuration de notre Bot, créer l'objet permettant de lui donner vie.
Rien de bien particulier à dire à ce niveau :

```
# On crée l'objet qui nous permet de nous connecter à IRC :
my $irc = new Net::IRC;
```

Nous avons à notre disposition une instance de la classe Net::IRC, nommée $irc (how convenient !)... Utilisons là !
IRC est un protocole fonctionnant en mode connecté ; nous allons donc créer une nouvelle connexion vers le serveur IRC définit plus haut, avec les informations de configuration que nous avons choisi : le serveur, le nickname, ...
Notez que nous nous connecterons sur le port 6667 ; il s'agit du port par défaut, sur lequel la plupart des serveurs IRC écoutent. Il se peut que le serveur qui vous intéresse soit en attente de connexion sur un autre port, mais c'est fort peu probable... Et si c'est le cas, il est quasiment certain que vous serez déjà au courant.
Attention : nous obtenons ici un objet correspondant à une connexion, mais celle-ci n'est pas encore activée : nous ne nous sommes pas encore connectés - nous avons juste les moyens de le faire.

```
# On crée l'objet de connexion à IRC :
my $conn = $irc->newconn(
    'Server'      => $server,
    'Port'        => 6667,
    'Nick'        => $nick,
    'Ircname'     => $ircname,
    'Username'    => $username
);
```

La librairie Net::IRC fonctionne sur le principe des événements : à chaque fois qu'elle reçoit une information utile, elle appelle, si vous le lui avez demandé, une fonction que vous avez défini.
Ces événements peuvent être le fait que vous soyez connecté, le fait que quelqu'un ait joint le channel où vous vous trouvez, le fait que quelqu'un a demandé votre numéro de version, ...
Des alias nommés sont définis par la librairie pour certains de ces événements ; pour les autres, vous n'aurez à leur disposition que leur numéro ; pour plus d'information, je vous invite à consulter la documentation, et la RFC.

Pour cette première version de notre Bot, nous resterons simples, et nous ne gérerons que deux événements :
Le fait que nous soyons connectés à IRC, qui se détecte grâce à l'événement levé par la fin de ce qui est appelé le MOTD (Message Of The Day), que les serveurs utilisent pour donner quelques informations sur leurs services, ou des avertissements aux utilisateurs. Cet événement a pour numéro 376.
Et l'événement 'public', qui est levé à chaque fois qu'un utilisateur parle sur le channel où vous vous trouvez.

Pour assigner une fonction à un événement, on utilise la fonction add_handler, appelée sur l'objet retourné plus haut, au moment où nous avons lancé le processus de connexion au serveur.

Cette méthode prend en paramètre l'événement qui doit déclencher son appel, et une référence vers la fonction, généralement définie par nous-même, qui doit être appelée au moment où l'événement se produit.
Par exemple, nous avons choisi de demander à la librairie que la fonction on_connect soit appelée à la fin du MOTD, et que la fonction on_public soit appelée à chaque fois que quelqu'un parle sur le channel où nous nous trouvons.

```
# On installe les fonctions de Hook :
$conn->add_handler('376', \&on_connect);          # Fin du MOTD => on est connecté
$conn->add_handler('public', \&on_public);        # Sur le chan
```

Deux choses restent à faire ; la première est de réellement se connecter. Pour cela, nous appelons la méthode start() de l'instance de classe Net::IRC.
Cette méthode lance la connexion, et gère toute la boucle de notre client : elle se charge d'absolument tout ce qui est communication entre le client et le serveur, y compris la gestion des événements.
Nous l'utilisons ici car elle permet de grandement faciliter les choses ; cela dit, nous verrons dans l'un des articles suivant celui-ci que sa simplicité est aussi une faiblesse : elle ne nous permet pas de personnaliser certaines parties de la communication client/serveur, ni de faire "autre chose" que gérer les événements. Nous verrons donc une autre solution, qui peut être avantageuse dans certaines situations.

```
# On lance la connexion et la boucle de gestion des événements :
$irc->start();
```

Pour finir, il nous faut définir les deux fonctions qui sont appelée automatiquement à la réception de deux événements que nous avons choisi de gérer.
Les fonctions de gestion d'événements reçoivent deux paramètres : le premier correspond à la connexion (l'objet retourné par le newconn vu plus haut), et le second contient toutes les informations correspondant à l'événement.
Ces informations varient d'un événement à l'autre... et, dans les cas où la documentation n'est pas suffisament complète, il est parfois un peu difficile de savoir quelles sont ces informations... Il peut donc être utile, au départ, d'utiliser le module Data::Dumper, pour faire un écho à l'écran de la structure de données de l'événement, afin de déterminer quelles sont les données qu'elle contient, et quelles sont celles que nous pouvons expoiter. Nous verrons un exemple un peu plus bas ; je pense que cela ne peut faire de mal.

Tout d'abord, voici la fonction que j'ai choisi d'implémenter pour gérer l'événement de fin du MOTD.
Cette fonction demande au Bot de joindre le channel que nous avons défini plus haut, et d'y dire "Salutations !". Ce même message est aussi affiché sur la console, afin que nous-même le voyons.

```
sub on_connect
{
    my ($conn, $event) = @_;

    $conn->join($channel);
    $conn->privmsg($channel, 'Salutations !');
    print "<$nick>\t| Salutations !\n";

    $conn->{'connected'} = 1;
} # Fin on_connect
```

Et maintenant, voici le code de la fonction qui sera appelée lorsque quelqu'un prononcera un message sur le channel où notre Bot se trouve :
Elle ne fait rien de plus qu'un écho sur la sortie standard de ce qui a été dit sur le channel.

```
sub on_public
{
    my ($conn, $event) = @_;
    my $text = $event->{'args'}[0];
    print "<" . $event->{'nick'} . ">\t| $text\n";
} # Fin on_public
```

Et voila comment, en moins de 70 lignes de Perl, on est à même de développer un Bot IRC ! Certes, il ne fait pas grand chose d'autre qu'un écho vers la console de ce qui se dit sur le channel... Il n'est à l'écoute d'aucune commande, n'effectue aucune action, ne dit rien, ne gère pas les messages privés ni les notices... Mais tout ça, ce ne sont que d'autres événements à intercepter : vous avez déjà toutes les bases nécessaires !

La suite de ce chapître va nous permettre de voir comment se comporte la première version de notre Bot lorsqu'on l'exécute, et comment comprendre quelles sont les données correspondant à un événement, afin de pouvoir en utiliser d'autres que ceux que nous avons vu ici.

Exécution

Nous avons vu la théorie... Passons maintenant à un peu de pratique !
Lancez notre bot (après avoir adapté, au besoin, sa configuration), et voyons ce qui se passe...

(Pour cet exemple, j'ai conservé la configuration donnée dans le code source que je vous ai proposé ; un utilisateur nommé "squale92" est déjà présent sur le channel rejoint par le bot ; les captures commençant par un timestamp sont faites en utilisant le client IRC nommé kvirc).
Nous pouvons voir ci-dessous que le bot joint le channel #krypton, et dit "Salutations !".
En réponse, l'utilisateur "squale92" lui dit "hello you !" (nous verrons par la suite les conséquences de ce message).

```
[21:17:47] perlBot_v1 [~perlBot@Wnet-4wepmk.wanadoo.fr] a rejoint #krypton
[21:17:47] <perlBot_v1> Salutations !
[21:18:03] <squale92> hello you !
```

Voici le résultat d'un whois sur notre bot :
(Notez que la présentation du résultat dépend généralement du client IRC employé)
Nous pouvons notamment remarquer une partie des options de configuration choisies plus haut...

```
[21:17:56] perlBot_v1 est perlBot_v1!~perlBot@Wnet-4wepmk.wanadoo.fr
[21:17:56] perlBot_v1 au nom réel: Bot IRC Perl
[21:17:56] perlBot_v1 est sur: #krypton
[21:17:56] perlBot_v1 est sur le serveur: Club.FR.Worldnet.Net - Club-Internet /
T-Online France
[21:17:56] perlBot_v1 au temps d'inactivité: 0d 0h 0m 9s
[21:17:56] perlBot_v1 s'est connecté le: lun déc 19 21:17:46 2005
[21:17:56] perlBot_v1 au WHOIS venant de Club.FR.Worldnet.Net
```

Nous n'avons pas implémenté, au sein de notre bot, de méthode propre pour quitter...
Il faut donc le terminer par un Ctrl+C (ou équivalent sous votre système) ; et le message de quit par défaut est un "Client closed connection".
(Nous verrons probablement plus tard comment personnaliser ceci pour afficher un message plus fantaisiste et moins déplaisant pour les utilisateurs du channel).

```
[21:18:05] perlBot_v1 [~perlBot@Wnet-4wepmk.wanadoo.fr] a quitté IRC: Client
closed connection
```

Et voici ce qui affiché sous la console depuis laquelle nous avons lancé notre mini-bot :

```
$ ./bot.pl
<perlBot_v1>    | Salutations !
<squale92>      | hello you !
```

Vivement quelque chose de plus utile, me direz-vous ? Vous avez raison !
Mais avant ça, il fallait voir les bases.

Données d'un événement

On a vu plus haut que les fonctions que l'on indiquait comme devant être exécutées lorsqu'un événement se produit, en utilisant la méthode add_handler, recevaient en paramètres une référence sur l'objet de connexion, et une autre sur les données correspondant à l'événement en cours.
Nous allons maintenant voir comment nous pouvons déterminer quelles sont ces données, de manière à pouvoir écrire des fonctions adaptées à chaque type d'événement nous intéressant.

Pour cela, nous ferons simple : nous utiliserons le module Data::Dumper, fourni d'office avec Perl. Ce module permet, entre autre, d'obtenir sous forme d'une chaîne de caractères (que l'on peut imprimer à l'écran, donc) le contenu d'une ou plusieurs variables, y compris si celle-ci correspond à une structure de données complexe, contenant par exemple des hash ou des listes imbriquées.
De plus, l'affichage peut se faire - et, par défaut, se fait - de manière indentée, et donc, extrêmement lisible.
Le but de cet article n'étant pas de présenter Data::Dumper, je vous renvoie à sa documentation pour plus de détails si vous n'avez pas déjà l'habitude de l'utiliser.

Par exemple, admettons que l'on veuille savoir quelles données sont à notre disposition lorsque la fonction branchée sur l'événement 'public' est appelée.
La solution que j'ai retenu est de faire appel à Data::Dumper dans celle-ci, et d'afficher le résultat sur la sortie standard.
Par exemple, nous pourrions remplacer notre fonction 'on_public' par celle-ci, qui va provoquer l'affichage des données de la connexion, et de l'événement :

```
sub on_public
{
    my ($conn, $event) = @_;
    use Data::Dumper;
    print Data::Dumper->Dump([\$conn, \$event], [qw(conn event)]);
} # Fin on_public
```

Et dans le cas où quelqu'un dit quelque chose sur le channel - par exemple, ceci :

```
[01:46:19] <squale92> coucou !
```

Nous aurons à l'écran affichage des données correspondant à la connexion :

```
$conn = \bless( {
                    '_handler' => {
                                    'public' => [
                                                  sub { "DUMMY" },
                                                  0
                                                ],
                                    'endofmotd' => [
                                                     sub { "DUMMY" },
                                                     0
                                                   ]
                                  },
                    '_ignore' => {},
                    '_debug' => 0,
                    '_ssl' => 0,
                    '_port' => 6667,
                    '_connected' => 1,
                    '_maxlinelen' => 510,
                    '_ircname' => 'Bot IRC Perl',
                    '_format' => {
                                   'default' => '[%f:%t]  %m  <%d>'
                                 },
```

```
                    '_verbose' => 0,
                    '_parent' => bless( {
                                          '_outputqueue' => bless( {
                                                                     'queue' => {}
                                                                   },
'Net::IRC::EventQueue' ),
                                          '_conn' => [],
                                          '_connhash' => {
                                                           'IO::Socket::INET=GLOB(0x841054
c)' => [

sub { "DUMMY" },

${$conn}

]
                                                         },
                                          '_write' => bless( [
                                                               undef,
                                                               0
                                                             ], 'IO::Select' ),
                                          '_read' => bless( [
                                                              ',
                                                              1,
                                                              undef,
                                                              undef,
                                                              undef,
                                                              bless( \*Symbol::GEN0,
'IO::Socket::INET' )
                                                            ], 'IO::Select' ),
                                          '_timeout' => 1,
                                          '_debug' => 0,
                                          '_error' => bless( [
                                                               undef,
                                                               0
                                                             ], 'IO::Select' ),
                                          '_schedulequeue' => bless( {
                                                                       'queue' => {}
                                                                     },
'Net::IRC::EventQueue' )
                                        }, 'Net::IRC' ),
                    'connected' => 1,
                    '_pacing' => 0,
                    '_username' => 'perlBot',
                    '_nick' => 'perlBot_v1',
                    '_frag' => '',
                    '_lastsl' => 0,
                    '_server' => 'irc.worldnet.net',
                    '_socket' => ${$conn}->{'_parent'}->{'_read'}->[5]
                  }, 'Net::IRC::Connection' );
```

Et celles correspondant à l'événement en lui-même :

```
$event = \bless( {
                    'to' => [
                              '#krypton'
                            ],
                    'format' => 'public',
                    'from' => 'squale92!~squale@Wnet-4wepmk.wanadoo.fr',
                    'user' => '~squale',
                    'args' => [
                                'coucou !'
                              ],
                    'nick' => 'squale92',
                    'type' => 'public',
                    'userhost' => '~squale@Wnet-4wepmk.wanadoo.fr',
                    'host' => 'Wnet-4wepmk.wanadoo.fr'
                  }, 'Net::IRC::Event' );
```

Vous pouvez constater que les données de la connexion ne vous apportent pas énormément d'informations utiles ; au point que je ne les affiche généralement pas...
Par contre, les données de l'événement contiennent toutes les informations dont vous avez besoin pour réagir : nickname de la personne qui a parlé, contenu du message, identifiant (username et hostname) de l'utilisateur, channel de destination, ...

Notez que les informations de l'événement changent selon le type d'événement (on aurait pu s'en douter ^^) ; à vous de déterminer quelles sont celles qui vous intéressent, pour les événements que vous avez choisi d'écouter.
Pour cela, il vous suffira d'utiliser une fonction telle celle-ci que j'ai reproduite ci-dessus... Ou comme celle qui suit, qui n'affiche pas les données de la connexion, du fait que celles-ci ne sont pas vraiment indispensables, du moins dans un certain nombre de cas simples.

```
sub on_public
{
    my ($conn, $event) = @_;
    use Data::Dumper;
    print Data::Dumper->Dump([\$event], [qw(event)]);
} # Fin on_public
```

Les suites de cet article nous permettront probablement de gérer quelques autres événements...

Code source complet

Pour terminer, voici, en un seul morceau, le code source du mini-bot que nous venons de créer ; il vous sera plus facile de l'exécuter ainsi, sans avoir à rassembler les différents morceaux présentés au sein de cet article.

```
#!/usr/bin/perl

use strict;
use warnings;

# On utilise la librairie Net::IRC pour se connecter à IRC
use Net::IRC;


# Configuration des options de connexion (serveur, login) :
my $server = 'irc.worldnet.net';
my $nick = 'perlBot_v1';


# Informations concernant le Bot :
my $ircname = 'Bot IRC Perl';
my $username = 'perlBot';
my $version = '1.0';


# Channel sur lequel on veut que le Bot aille :
my $channel = '#krypton';


# On crée l'objet qui nous permet de nous connecter à IRC :
my $irc = new Net::IRC;


# On crée l'objet de connexion à IRC :
my $conn = $irc->newconn(
    'Server'      => $server,
    'Port'        => 6667,
    'Nick'        => $nick,
    'Ircname'     => $ircname,
    'Username'    => $username
);


# On installe les fonctions de Hook :
$conn->add_handler('376', \&on_connect);          # Fin du MOTD => on est connecté
$conn->add_handler('public', \&on_public);        # Sur le chan


# On lance la connexion et la boucle de gestion des événements :
$irc->start();
```

```
## Les fonctions de gestion des événements :

sub on_connect
{
    my ($conn, $event) = @_;

    $conn->join($channel);
    $conn->privmsg($channel, 'Salutations !');
    print "<$nick>\t| Salutations !\n";

    $conn->{'connected'} = 1;
} # Fin on_connect


sub on_public
{
    my ($conn, $event) = @_;
    my $text = $event->{'args'}[0];
    print "<" . $event->{'nick'} . ">\t| $text\n";
} # Fin on_public
```

Exemple Complet

Maintenant que vous savez comment créer un bot simple, je ne peux que vous encourager à lire la seconde partie de cet article, destinée à vous montrer comment aller plus loin !