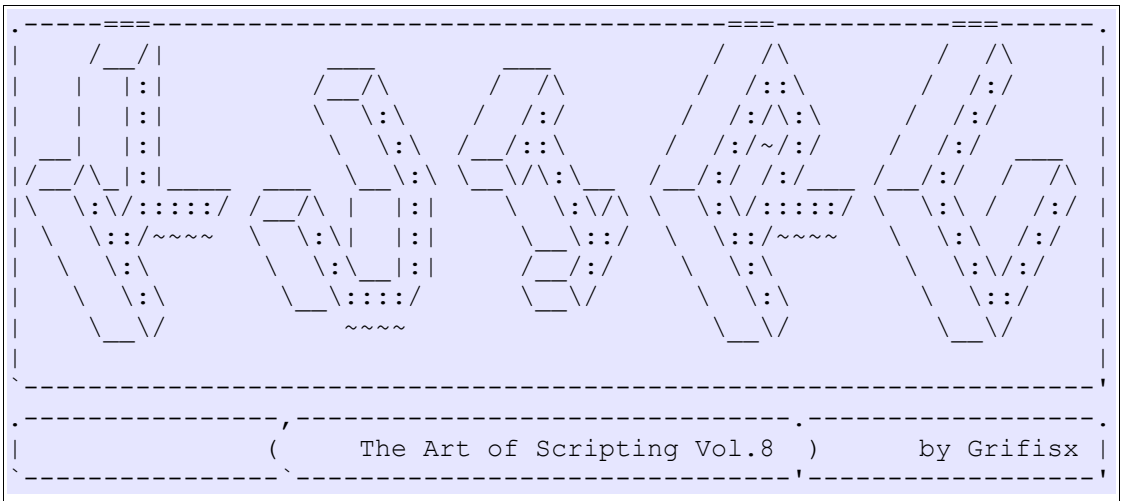

Socket e Perl con KVIrc

Start:

The Server side:
In questo tutorial cercherò di essere molto conciso.
Per prima cosa andremo a costruirci un serverino cioè faremo in modo che una socket si predisponga in ascolto su una certa porta e si dichiari disponibile ad accettare eventuali connessioni verso di questa.
La classe che ci interessa è, ovviamente, la classe socket quindi da bravi studenti andiamo a leggerci un po' le funzioni che il nostro bel client ci mette a disposizione dal famoso file di help =).
Fatto questo cominciamo subito a buttar giù codice allegramente :D.
Per prima cosa, guardando la documentazione, si nota che l'elemento fondamentale di una socket sono gli eventi ed è tramite questi che possiamo capire cosa sta succedendo sulla nostra socket, se qualcuno ci si sta connettendo, se la connessione è terminata e così via.
Questo ci fa anche pensare che la cosa migliore è andare a creare una bella classe tutta nostra che derivi da socket in modo da poter scrivere del codice più leggibile e immediato, senza troppi privateImpl(bla bla bla).
Quindi cominciamo subito a far nascere la nostra classe *myserver:*

```
class(myserver,socket)
{
      constructor(){};
}
```

Ok, ora che è nata possiamo cominciare a pensare a cosa debba fare e gestire la nostra socket; abbiamo detto che sarà un server quindi la prima cosa che dovrà fare è mettersi in ascolto su una porta.
Vediamo la funzione *$listen()* cosa ci permette di fare:
mette in ascolto una socket su una porta, usando una determinata interfaccia (più semplicemente un ip) e permettendoci la possibilità, se volessimo, di usare il protocollo ipv6, ottimo direi...
Quindi quello che dovremmo fare nel nostro caso sarebbe:
(Nostra_socket)*->$listen(80,127.0.0.1,0)*
Specifichiamo la porta 80, interfaccia mettiamo l'ip locale e ipv6 lo settiamo a 0 cioè FALSO.

Quindi riscriviamo il nostro codice:

```
class(myserver,socket)
{
	constructor()
	{
		if(!$$->$listen(80,127.0.0.1,0))
		{
			echo Err: Listen fallito.
		}
		echo Server in ascolto
	}
}
```

Come vedete ho messo un *if* per farci dire se il comando fallisce. Infatti questa funzione ci restituisce 1 se è andata a buon fine e 0 nel caso opposto, quindi posso valutare il valore restituito tramite un semplice *if*.
Mi scuso con gli utenti avanzati per queste precisazioni ma i tutorial devono essere leggibili da tutti ^_^.
Adesso vediamo che la socket che stiamo costruendo deve reagire alle connessioni in arrivo, quindi sarà necessario reimplementare l'evento *incomingConnectionEvent()*, ma non soltanto poiché se andiamo a leggere la documentazione su questo evento ci dice:

'*incomingConnectionEvent()*
This function is called when an incoming connection arrives over a socket in listening state.
***You must call $accept() passing a newly created socket object** to accept and handle the connection.*
If you don't call $accept() the incoming connection will be automatically terminated.'

Quindi, a quanto ci dice la documentazione, noi **dobbiamo** chiamare la funzione $accept() passandogli un nuovo oggetto socket creato sul momento e sarà poi lui a gestirsi quella particolare connessione. Vediamo un po' come dovrà essere il nuovo codice

```
class(myserver,socket)
{
	constructor()
	{
		if(!$$->$listen(80,127.0.0.1,0))
		{
			echo Err: Listen fallito.
		}
		echo Server in ascolto
	}
	incomingConnectionEvent()
	{
		%Tmp = $new(socket)
		$$->$accept(%Tmp)
		echo "Connessione da: %Tmp->$remoteIp : %Tmp->$remotePort"

		%Tmp->$close()
		# Eliminiamo la sock di appoggio
		delete %Tmp;

	}
}
```

```
%Serverside=$new(myserver)
```

Ora possiamo eseguirlo per vedere cosa succede, quindi apriamo il nostro Script Tester, incolliamo ed eseguiamo!
Ci dice subito:
'Server in ascolto.'
Quindi la porta è stata messa in ascolto correttamente.
Adesso vediamo se reagisce ad una connessione, apriamo la shell dei comandi (oppure ms-dos se siamo sotto windows) e diamo il comando **telnet 127.0.0.1 80**
Come possiamo vedere il nostro serverino ci dice:
'Connessione da: 127.0.0.1 : 2161'
adesso però dobbiamo fare in modo da creare una sorta di comunicazione tra server e client, vediamo come possiamo fare.
Dovremmo usare le funzioni *$write()*, *$read()* ed ovviamente l'evento *dataAvailableEvent()* per poter gestire lettura e scrittura sulla nostra socket (la socket slave ovviamente, cioè %Tmp, perché è quella la socket alla quale ci appoggiamo per mantenere la connessione.
Scriviamo il codice:

```
class(myserver,socket)
{
	constructor()
	{
		if(!$$->$listen(80,127.0.0.1,0))
		{
			echo Err: Listen fallito.
		}
		echo Server in ascolto
	}
	incomingConnectionEvent()
	{
		%Tmp = $new(socket)
		$$->$accept(%Tmp)
		echo "Connessione da: %Tmp->$remoteIp : %Tmp->$remotePort"
		%Tmp->$write("Welcome on KVIrc!\r\n");
		privateImpl(%Tmp,dataAvailableEvent)
		{
			%actualData=%Tmp->$read($0)
			if(%actualData!=$cr$lf){
			%Data = %Data%actualData;}
			else{
			echo $b %Tmp->$remoteIp() \-\> $b %Data
			%Data=
			}
		}

	}
	destructor()
	{
		delete %Tmp
		%Data=

	}

}
%Serverside=$new(myserver)
```

Come potete vedere tramite la funzione:
%Tmp->$write("Welcome on KVIrc!\r\n");
faccio scrivere sulla nostra socket un messaggio di benvenuto, poi gestisco l'evento *dataAvailableEvent()* della socket a cui ci stiamo

appoggiando e gli faccio leggere i dati con la funzione:
%actualData=%Tmp->$read($0);
se i dati che ha letto sono uguali ai carattere di **crfl** (guardatevi queste 2 funzioni, $cr e $fl, nella documentazione) allora mi faccio dare l'echo dei caratteri che ha accumulato fino a quel momento, altrimenti glieli faccio accumulare, in questo modo sul KVIrc l'echo apparirà delle stringhe inviate e non del singolo carattere (tutto questo perché telnet invia i dati uno alla volta nel moemnto stesso in cui scriviamo e non tutti insieme quando diamo l'Invio).
Adesso mettiamo il codice nello script tester, eseguiamo e poi dal prompt dei comandi diamo il:
telnet 127.0.0.1 80
e proviamo a scrivere qualche cosa nel prompt dei comandi e poi dare Invio, vedrete che sarà nata la prima forma di comunicazione tra KVIrc ed il nostro telnet tramite le socket =D. Notate che ho implementato anche il distruttore nella classe, cosi quando deleterò l'oggetto %Serverside, automaticamente mi deleterà i dati e la socket che uso come slave.

Client Side:
Adesso proviamo a fare quello che facciamo con il nostro telnet, tramite KVIrc stesso, creandoci una sorta di piccolo telnet sempre tramite script =).

```
class (mySocket,socket)
{
	constructor()
	{
		$$->$connect("127.0.0.1",81)
	}

	connectFailedEvent()
	{
		echo "Connection to localhost failed..."
	}

	connectEvent()
	{
		$$->$write("Connesso: $cr$lf")
	}

	dataAvailableEvent()
	{
		echo $$->$read($0)
	}
	send()
	{
		$$->$write($0$cr$lf)
	}

}
%Connessione = $new(mySocket)
```

Come vedete la costruzione è abbastanza semplice non c'è niente di particolare, adesso che sapete creare un server il codice del client è chiaro.
Da notare che ho fatto la funzione *$send()*, questa ci permetterà di poter comunicare con il server anche dopo la connessione; basterà infatti il comando del tipo:
%Connessione->$send(frase da inviare)
La connessione la si crea tramite la funzione *$connect(<host>,*

<porta>) e tutto il resto lo si gestisce più o meno come abbiamo visto con il server.
Adesso andiamo a modificare il server che abbiamo scritto prima in modo da renderlo adatto anche a questo tipo di connessione e capiremo perché è diversa da quella fatta tramite il telnet.

```
class(myserver,socket)
{
	constructor()
	{
		if(!$$->$listen(81,127.0.0.1,0))
		{
			echo Err: Listen fallito.
		}
		echo Server in ascolto
	}
	incomingConnectionEvent()
	{
		%Tmp = $new(socket)
		$$->$accept(%Tmp)
		echo "Connessione da: %Tmp->$remoteIp : %Tmp->$remotePort"
		privateImpl(%Tmp,dataAvailableEvent)
		{
			%actualData=%Tmp->$read($0)
			echo $b %Tmp->$remoteIp() \-\> $b %actualData
		}
	}
	destructor()
	{
		delete %Tmp
	}
}
%Serverside=$new(myserver)
```

La differenza è che, come vedete, non controllo più se c'è il carattere(in realtà sono 2 caratteri) di invio (ritorno a capo+nuova linea) (crlf)prima di farmi dare l'echo e questo semplicemente perché adesso i dati li inviamo tutti insieme e non più un carattere alla volta come con il telnet.
Facciamo la prova pratica, avviate prima il server e dopo il client e dopo date il comando
/%Connessione->$send(Questa è una prova)
Come vedete, se tutto è andato per il verso giusto, il server vi avrà stampato a schermo i dati che avete inviato.
La funzione $send() che abbiamo fatto infatti invia la frase+crlf tutta insieme mentre, come ho detto prima, con il telnet man mano che scrivevamo un carattere questo veniva inviato ecco perché ci serviva controllare quando arrivavano i caratteri di fine riga e ritorno a capo prima di stampare a schermo.

Perl:

Prima di concludere questo tutorial volevo parlare un po' della possibilità del KVIrc di eseguire codice Perl il che puo' essere molto utile.
Cominciamo col dire che il codice Perl va inserito tra i comandi **perl.begin** e **perl.end**, quindi avremo un codice di questo genere

```
perl.begin
<codice>
<codice>
```

```
perl.end
```

dall'interno del codice Perl possiamo usare alcuni comandi, che ora andremo a vedere, per poter interagire con il KVS (KVIrc Scripting).
Come in questo esempio:

```
perl.begin
KVIrc::echo("Prova del Perl riuscita!");
perl.end
```

Se lo eseguiremo vedremo che stamperà a schermo la frase che abbiamo passato al comando KVIrc::echo, che non è altro che la 'controparte' del normale echo che usiamo in scripting.
La sintassi completa è:
KVIrc::echo(<text>[,<colorset>[,<windowid>]])
<text> è il testo da stampare.
<colorset> è l'equivalente dell'opzione -i dell'echo.
<windowid> è l'equivalente dell'opzione -w dell'echo.
Quindi se volessimo scrivere la frase blu, ad esempio, potremmo fare:
KVIrc::echo("Prova del Perl riuscita!",4);
Ovviamente i parametri colorset e windowid possono essere omessi.
Un'altra possibilità è quella di passare delle variabili all'interno del nostro micro-ambiente Perl.
KVIrc::getLocal(<x>)
Ritorna il valore della variabile locale %x di KVIrc
KVIrc::getGlobal(<Y>)
Ritorna il valore della variabile %Y di KVIrc.
KVIrc::setLocal(<x>,<value>)
Setta il valore della variabile locale, di KVIrc , %x a <value>
KVIrc::setGlobal(<Y>,<value>)
Setta il valore della variabile Globale, di KVIrc ,%Y a <value>
Ecco un esempio direttamente dalla documentazione:

```
%pippo = test
%Pluto = 12345
perl.begin
$mypippo = KVIrc::getLocal("pippo");
$mypluto = KVIrc::getGlobal("Pluto");
KVIrc::setLocal("pippo",$mypluto);
KVIrc::setGlobal("Pluto",$mypippo);
perl.end
echo "\%pippo is" %pippo
echo "\%Pluto is" %Pluto
```

Oppure:

```
%x = 10
perl.begin
$x = KVIrc::getLocal("x");
KVIrc::eval("echo \"The value is ".$x."\"");
perl.end
```

Qui vedete anche il comando **KVIrc::eval(codice)** che serve ad eseguire codice di scripting in modo diretto.
Ultimo comando è **KVI::say**, anche questo è molto semplice nella sua utilità e nella sintassi:
KVIrc::say("Ciao a tutti!");
Fa la stessa cosa di *'/say Ciao a tutti'*

Ricapitolando i comandi a nostra disposizione sono:

KVIrc::echo(<text>[,<colorset>[,<windowid>]])
KVIrc::getLocal(<x>)
KVIrc::getGlobal(<Y>)
KVIrc::setLocal(<x>,<value>)
KVIrc::setGlobal(<Y>,<value>)
KVIrc::eval(<code>)
KVIrc::say(<text>)

Adesso vediamo, come fare a passare dei valori all'interno del nostro codice Perl:
La sintassi di *perl.begin* in realtà è questa:
perl.begin(<perl context>,<arg0>,<arg1>,...)
Cioè possiamo passare al nostro ambiente Perl degli argomenti che poi andremo a leggere tramite la funzione speciale **$_[InDICE]** che non è altro che un array contenente i parametri che abbiamo passato.
Per questo potrete guardavi direttamente gli esempi sulla guida ufficiale inutile che ve li traduca io.
Invece adesso andiamo a fondere insieme quello che abbiamo imparato:

```
class(myserver,socket)
{
	constructor()
	{
		if(!$$->$listen(80,127.0.0.1,0))
		{
			echo Err: Listen fallito.
		}
		echo Server in ascolto
	}
	incomingConnectionEvent()
	{
		%Tmp = $new(socket)
		$$->$accept(%Tmp)
		echo "Connessione da: %Tmp->$remoteIp : %Tmp->$remotePort"
		$$->$makeFileList()
		%Tmp->$write("HTTP/1.0 200 OK\r\n");
		%Tmp->$write("Content-type: text/html\r\n\r\n");
		%Tmp->$write("<html>\n")
		%Tmp->$write(" <head><title>The KVIrc Power</title></head>\n")
		%Tmp->$write(" <body bgcolor=\"#000000\" text=\"#00FF00\">\n")
		%Tmp->$write("  <center><h1><b>.:Kvirc
Power!:.</b></h1></center>\n")
		%Tmp->$write(%MyFiles)
		%Tmp->$write(" </body>\n")
		%Tmp->$write("</html>\n\r\n\r\n")
		# Chiudiamo
		%Tmp->$close()
		# Eliminiamo la sock di appoggio
		delete %Tmp;
	}
	dataAvailableEvent()
	{
		%Data = %Data + $$->$read($0)
	}
	makeFileList()
	{
		%MyFiles=""

		perl.begin
		opendir(DIR, ".");
```

```
            @files = readdir(DIR);
            closedir(DIR);
            foreach $file (@files) {
            KVIrc::eval("%MyFiles << <b>$file</b><br>");
            }
            perl.end
      }
}
%ServerSide=$new(myserver)
```

Come vedrete esaminando il codice, abbiamo creato una specie di piccolo e semplice serverino http, che dà come pagina iniziale un documento html contenente la lista di tutti i file della directory corrente, questa lista l'abbiamo realizzata tramite Perl, nella funzione *$makeFileList()*
Per provare come funziona, dopo averlo eseguito, connettiamoci tramite il nostro Browser preferito all'indirizzo http://127.0.0.1 ..magia =).
Infine, per completare, voglio farvi leggere anche il codice che si trova pubblicato sulla wiki ufficiale del KVIrc, anche qui troviamo l'uso del Perl ma per esaminare, tramite una connessione client, quello che un server ci sta dicendo... ecco a voi il GetIp:

```
class (mySocket,socket) {
      constructor() {
            # trying to connect
            $$->$connect("whatismyip.com",80)
      }
      connectFailedEvent() {
            # connection failed
            echo "Connection to whatismyip.com failed..."
      }
      connectEvent() {
            # Seinding request...
            $$->$write(GET / HTTP/1.1$char(13)$char(10))
            $$->$write(Host: whatismyip.com$char(13)$char(10))
            $$->$write(Connection:
close$char(13)$char(10)$char(13)$char(10))
      }
      dataAvailableEvent() {
               # reading data
            %Data = %Data + $$->$read($0)
      }
      disconnectEvent() {
            # disconnected from server
            %MyIP = ""
            perl.begin (0, %Data)
            my $data = $_[0];
            if ($data =~ /([0-9]+?)\.([0-9]+?)\.([0-9]+?)\.([0-9]+)/)
            {
                  KVIrc::eval("%MyIP = $&");
            }
            perl.end
            perl.destroy(0)
            %blub = "bla"
      }
}
%Data = ""
%myCon = $new(mySocket)
echo %MyIP
```

Il codice di sopra non l'ho scritto io, lo trovate su 'http://we-are-

teh-b.org/~kvirc/index.php/GetIP'.
A questo punto siete in grado di capirlo =) ..digitiamo soddisfatti il nostro
/ECHO STOP.

- - - - - - -- - - - - - - -- - - - - - - - - -- - - - - -- - - - - -
" Tu vedi cose e ne spieghi il perché, io invece immagino cose che non sono mai esistite e mi chiedo perché no." (George Bernad Shaw)
- - - - - - - - -- - - - - - - -- -- - - - - - - - - - - - - - - - -
Grifisx