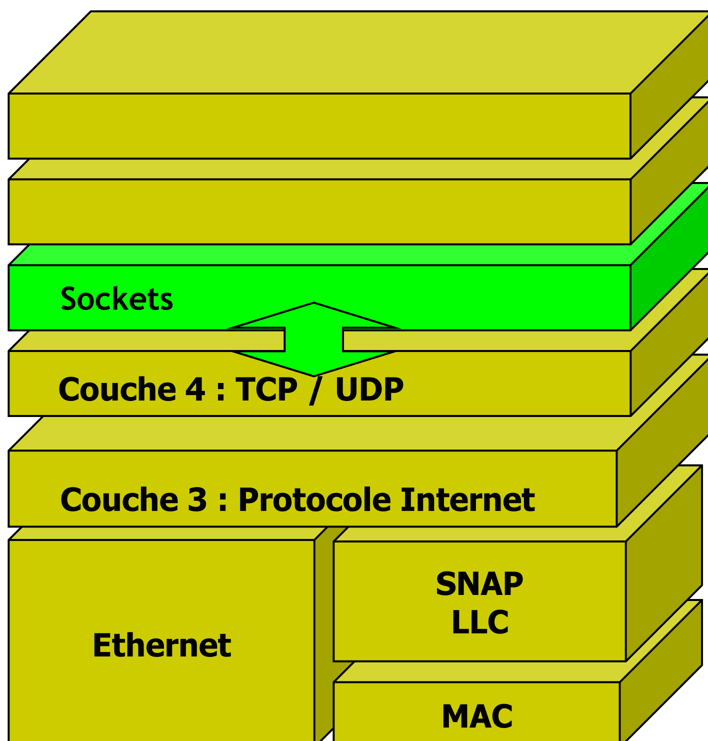
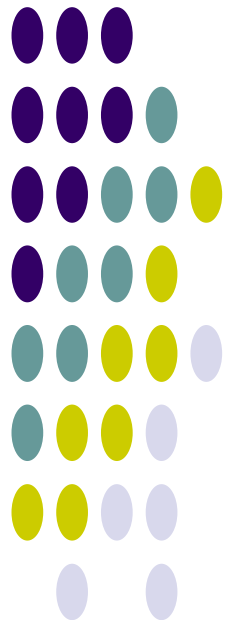
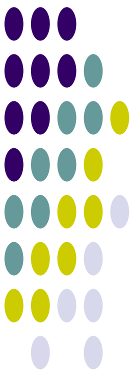
Programmation Réseau : Les sockets

Jean-Yves Tigli
tigli@polytech.unice.fr

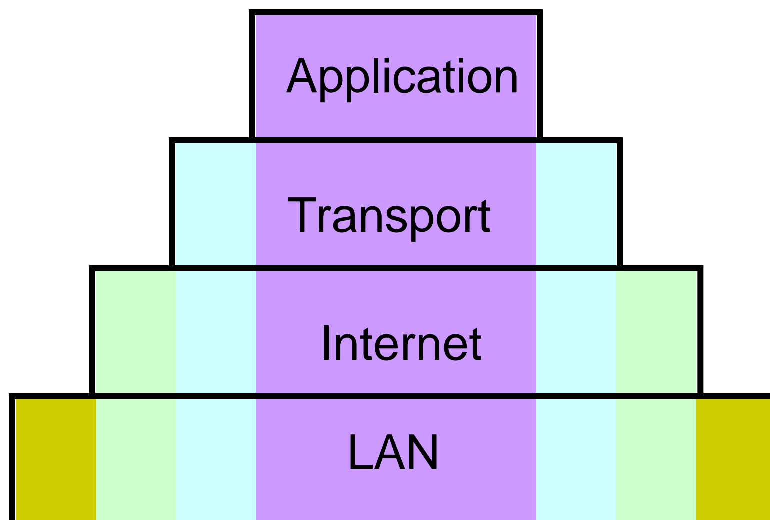


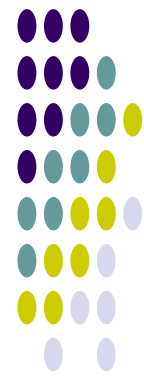


Plan du cours : Modèle simplifié 4 couches



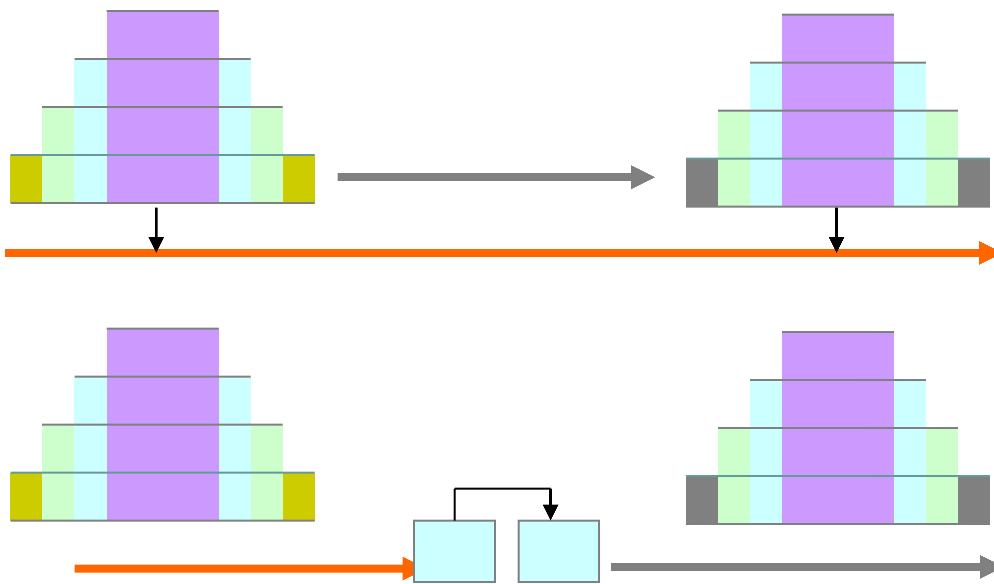
- Niveau Application
- Niveau Protocole de Transport
- Niveau Internet
- Niveau Réseau Local (LAN)



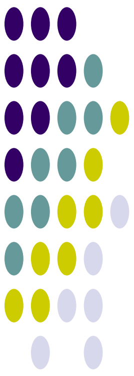


Modèle simplifié LAN

- Niveau Réseau Local (LAN)



Modèle simplifié IP/LAN

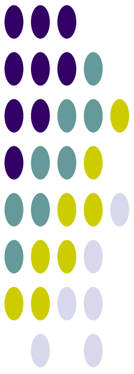


- Niveau Application
- Niveau Protocole de Transport

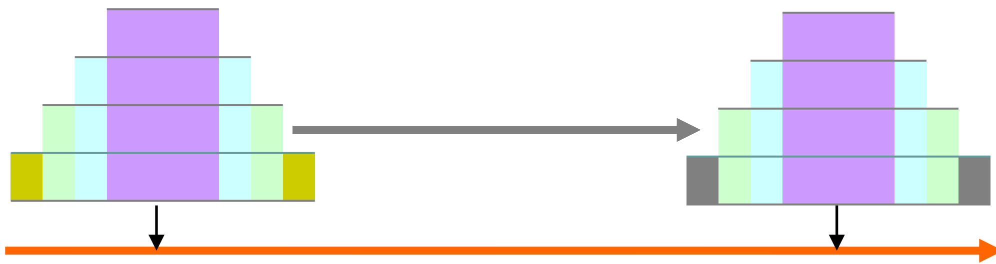
- Internet cours : Socket BSD
- TD : Programmation Socket en C#

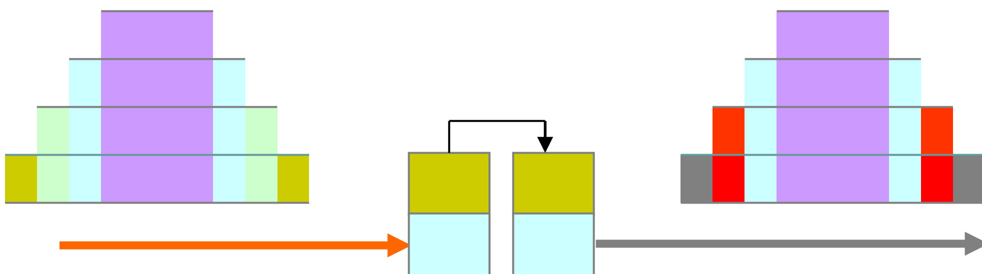
- Internet cours : IP et TCP/UDP

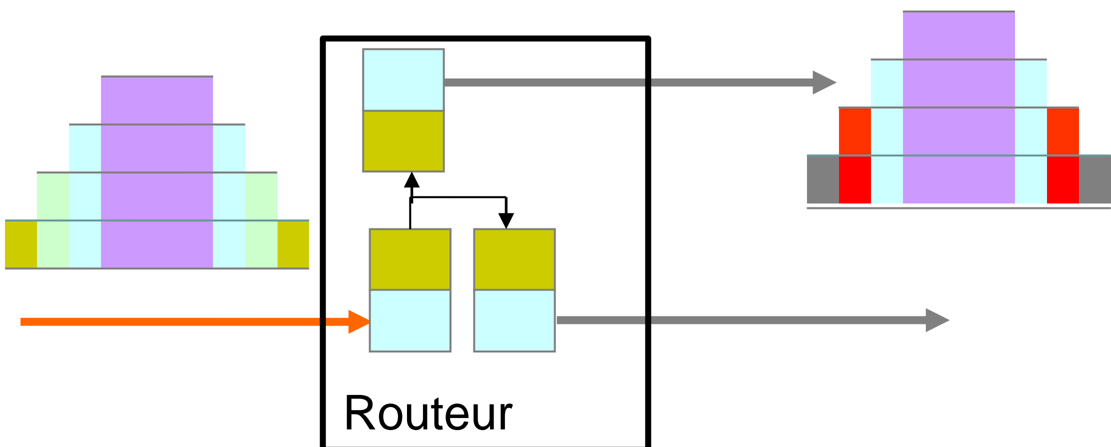
Modèle simplifié IP/routage



- Niveau Application
- Niveau Protocole de Transport
- Niveau Internet

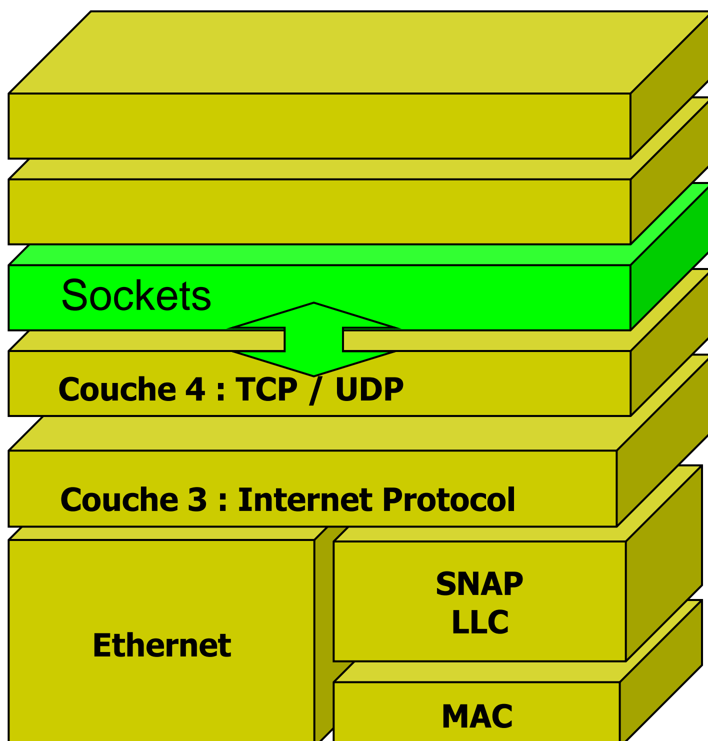


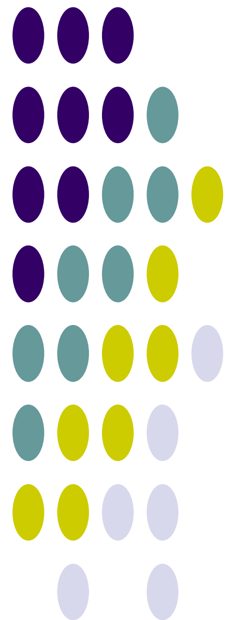


- 

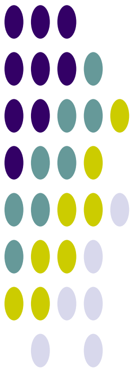
Les Socket

Jean-Yves Tigli
tigli@polytech.unice.fr





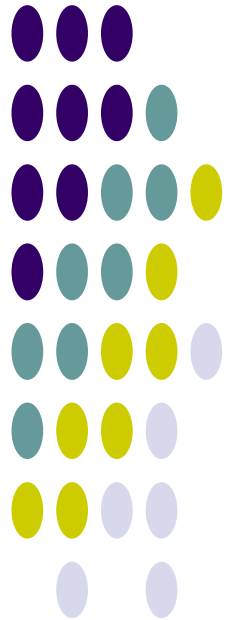
Applications niveau Socket

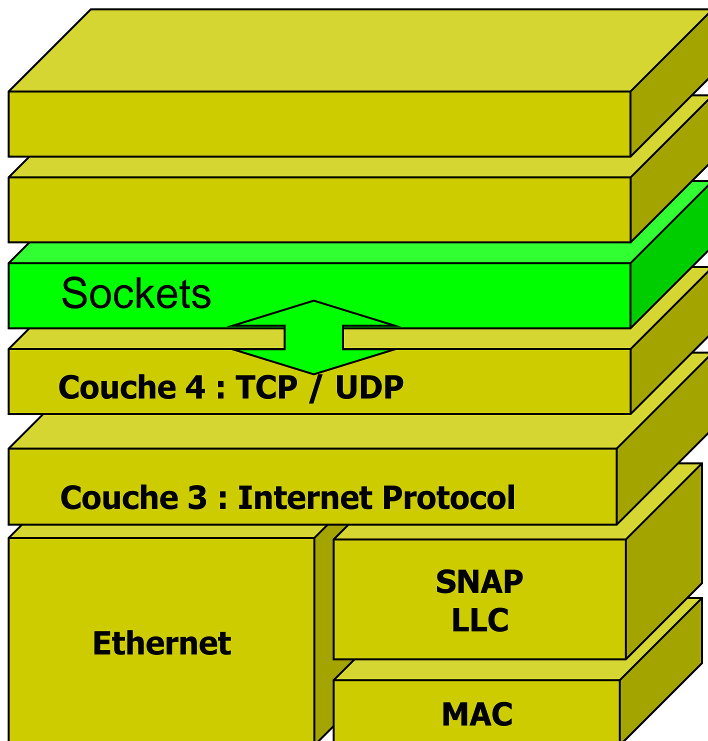


- Un grand nombre de services programmés au niveau sockets, avec des protocoles simples (voir RFCs) :
  - HTTP
  - SMTP
  - …
- Des serveurs :
  - Des démon (sendmail –qd)
  - xinetd le démon des démons et xinetd.conf
  - Les services sous Windows
- Une relation service/port
  - /etc/services

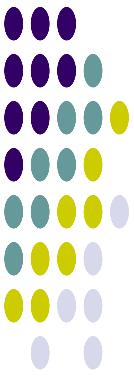
Les Socket C#

Jean-Yves Tigli
tigli@polytech.unice.fr



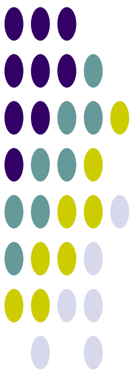


Les sockets en C#



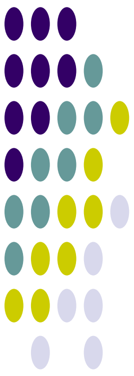
- Les sockets sont une API de communication basée sur TCP/IP qui a été développée pour le langage C en 1983.
- Depuis tout ce temps, les sockets sont portés d'une plateforme et d'un langage à l'autre et ce n'est pas étonnant qu'ils existent encore en C#.
- En fait, un socket est totalement indépendant du langage.
- Un programme qui joue le rôle de serveur peut communiquer avec un autre programme jouant le rôle de client si ceux-ci utilisent des sockets, peu importe les langages impliqués.

Les notions de base sont les suivantes:



- L'espace de nom System.Net.Sockets contient tout ce qu'il faut pour manipuler des sockets.
- La classe Socket (de ce même espace de nom) est évidemment la principale.
- Un socket doit d'abord être créé pour pouvoir être utilisé.

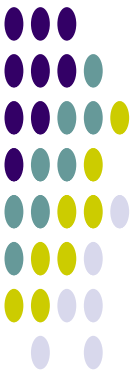
Le Socket



- Les 3 paramètres que nous devrons fournir à son constructeur sont les suivants:
  - une famille d'adresses (qui représente en fait un type d'adressage),
  - un type de socket (qui détermine la façon qu'il utilisera pour communiquer) et
  - un type de protocole.
- Il existe justement un enum pour chacun de ces paramètres; il nous suffira simplement de sélectionner les bonnes valeurs dans chacun d'entre eux:

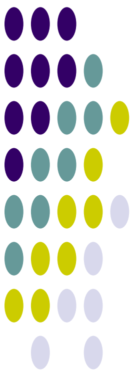
```
Socket sock = new Socket(
AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
```

Les paramètres d’un Socket



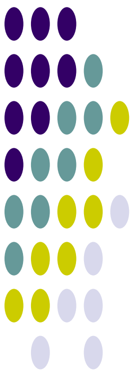
- Address.Family.InterNetwork correspond à IPv4.
  - Notez qu'il existe un InterNetworkV6 et tout un tas d'autres modes d'adressage.
- SocketType.Stream permet des communications 1 à 1 et exige qu'une connexion soit établie avant que la communication puisse débuter.
  - La communication pourra alors se faire dans les deux sens, en utilisant un flot d'octets.
  - C'est la méthode à utiliser en TCP.
- Évidemment, ProtocolType.Tcp correspond au protocole TCP!

Sockets Client/Serveur



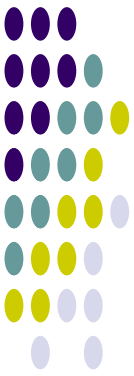
- Dans une communication client/serveur, chacune des deux parties devra se créer un socket.
- Les deux sockets formeront ultimement les deux bouts d'un tuyau de communication.

Le serveur



- Du côté du serveur, le socket, une fois créé, doit être attaché à un point de communication (endpoint).
- Un point de communication correspond à une adresse et un port TCP sur lequel écouter.

IPEndPoint

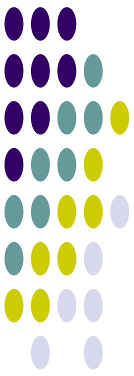


- Il existe un type IPEndPoint défini pour cet usage.
- On peut le créer en lui passant une adresse et un port au constructeur (l'adresse sera une instance de IPAddress, un autre type spécifiquement créé pour la communication TCP/IP et le port sera simplement un entier).
- On utilisera généralement l'adresse IP du serveur:

```
IPEndPoint iep = new
IPEndPoint(IPAddress.Parse("192.168.0.4"),1212);
sock.Bind(iep);
```

- Notez la méthode statique Parse de IPAddress qui accepte un string et qui retourne un objet IPAddress si elle est valide (sinon, une exception de type FormatException sera lancée).

Listen

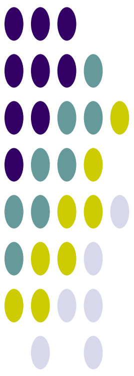


- Une fois le socket attaché à un point de communication, on le place en mode "écoute":

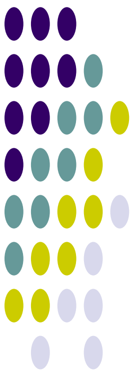
  ```
  sock.Listen(1);
  ```

- Le nombre passé en paramètre représente le nombre de connexions simultanées maximal.
- Lorsqu'un client tentera de se connecter, si le nombre maximal est déjà atteint, le client recevra une exception de type SocketException (le champ ErrorCode de l'exception donnera les détails appropriés).
- Lorsque le socket est en mode "écoute", il pourra recevoir autant de demandes que l'indiquera le paramètre.

Plus simple : la classe TcpListener



- La classe TcpListener fournit des méthodes simples qui écoutent et acceptent les demandes de connexion entrante en mode blocage synchrone.
- Créez TcpListener en utilisant une adresse IP locale et un numéro de port, ou seulement un numéro de port.
- // TcpListener server = new TcpListener(port); server = new TcpListener(localAddr, port);



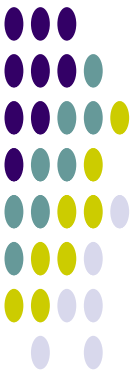
Accept

- Ces demandes seront placées en file et devront être traitées en les acceptant de façon synchrone ou asynchrone. La façon synchrone est la plus simple:

  Socket actif = sock.Accept();

- Remarquez que la méthode Accept() retourne un socket "connecté".
- C'est dans celui-là qu'on pourra écrire et lire.

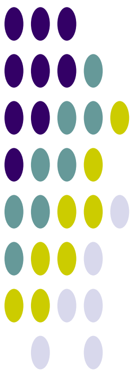
Le client



- Du côté du client, c'est encore plus simple.
- On doit créer un socket de la même façon que pour le serveur.
- Le socket tentera ensuite une connexion au point de communication correspondant au serveur:

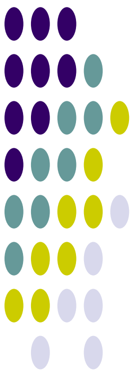
  IPEndPoint iep = new IPEndPoint(IPAddress.Parse("192.168.0.4"),1212); sock.Connect(iep);
- Si la connexion n'a pas pu être établie, une SocketException sera levée.

Plus simple : la classe TcpClient



- La classe TcpClient fournit des méthodes simples de connexion, d'envoi et de réception de flux de données sur un réseau en mode blocage synchrone.
- Afin que TcpClient puisse se connecter et échanger des données, un TcpListener.

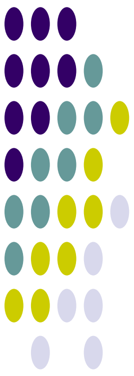
Plus simple : la classe TcpClient



- Vous pouvez vous connecter à cet écouteur des deux manières suivantes :
- Créer TcpClient et appeler une des trois méthodes Connect disponibles.
- Ce constructeur tentera automatiquement d'établir une connexion.

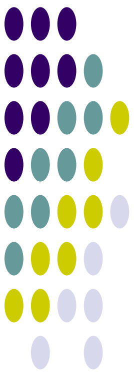
```
System.Net.Sockets.TcpClient clientSocket
= new System.Net.Sockets.TcpClient();
…
clientSocket.Connect("127.0.0.1", 8888);
```

Send



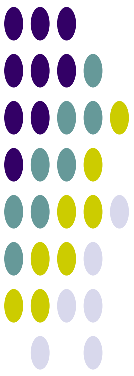
- On peut écrire dans le socket en envoyant un tableau d'octets (byte[]) dans le flot.
- Par exemple:
  - byte[] buffer = new byte[512]; //
- Supposons pour l'instant qu'il y a quelque chose dans le tableau...
  - sock.Send(buffer, 0, buffer.length, SocketFlags.None);

Send

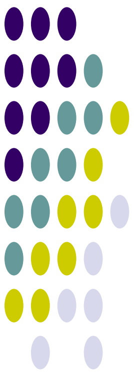


- Il s'agit de la surcharge la plus complexe de Send:
- on envoie les données contenues dans le buffer, en partant de 0 et en lisant toute la longueur (on pourrait spécifier des bornes différentes),
- puis on définit des options pour l'envoi en donnant une valeur de l'enum SocketFlags (ici, None, donc rien de spécial).
- Une combinaison d'options peut être faite en combinant les valeurs au niveau binaire...

Send



- Si on veut simplifier, on peut enlever un ou plusieurs de ces paramètres additionnels et ne passer que le buffer, ce qui reviendrait au même dans notre exemple.
- On peut convertir un string en byte[] en utilisant la fonction System.Text.Encoding.ASCII.GetBytes().
- L'inverse peut être fait grâce à sa fonction miroir System.Text.ASCIIEncoding.ASCII.GetString().



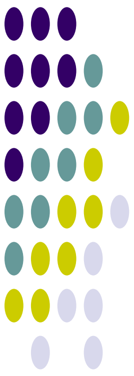
Receive

- On peut lire dans le socket en remplissant un byte[]:

  sock.Receive(buffer, 0, buffer.length, SocketFlags.None);

- Encore une fois, des versions surchargées différentes peuvent être utilisées.

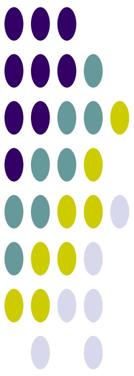
Shutdown



- Le shutdown interdit les opérations spécifiées sur le socket, ce qui permet à tout le monde de récupérer les données restantes avant de fermer. Par exemple:

  ```
  sock.Shutdown(SocketShutdown.Send);
  ```

- Les valeurs de paramètres possibles sont Send, Receive et Both, tous dans l'enum SocketShutdown.
- Une fois l'envoi interdit, on peut lire en étant certain que c'est tout ce qui reste. Ensuite on pourra fermer.



Close

- On pourra fermer le socket en faisant simplement:

  sock.Close();

- La propriété *Connected* du socket passe à False lorsqu'il est déconnecté.
- Notez que c'est une bonne pratique de faire un *shutdown* du socket avant de le fermer.