# Le bus ivy

(Ph. Truillet)

Septembre 2021 – v. 3.4

# 1. introduction

Ivy est un bus logiciel [middleware] (`http://www.eei.cena.fr/products/ivy`) conçu à la DTI R&D (ex CENA) dans le but de connecter d'une manière **extrêmement simple** des applications **interactives** ou pas en elles écrites avec différents langages et fonctionnant sur différentes machines ou plates-formes.

Il s'agit d'un modèle de communication **compatible avec la programmation événementielle** classique des interfaces graphiques. En un sens, ce bus logiciel implémente une approche multi-agents : les agents apparaissent, émettent des messages et en reçoivent, les traitent puis quittent le bus sans bloquer les autres agents présents. Ivy vise principalement à faciliter le développement rapide de nouveaux agents, et à en contrôler une collection dynamique.

Par opposition à certains autres bus logiciels, ivy ne se fonde pas sur un serveur central ou un annuaire qui permet de « router » les demandes d'un agent. Au lancement, tous les agents se présentent à un point de rendez-vous, le reste est transparent pour le programmeur !

En fait, le rôle d'ivy est de définir **principalement une convention de communication entre processus,** mise en application grâce à une collection de bibliothèques. Les messages sont échangés **sous une forme textuelle**, et la sélection des messages récupérés ou non par les agents est basée sur les expressions rationnelles (***regex*** en anglais ; famille de notations compactes et puissantes pour décrire certains ensembles de chaînes de caractères).

La Figure 1 résume le fonctionnement d'ivy : le travail essentiel à effectuer est à faire au **niveau du protocole d'échanges de données** entre agents : *qui communique avec qui et envoie quoi comme flot de données ?* Le protocole représente ici les échanges formatés envoyés par chacun des agents aux autres présents dans le système.

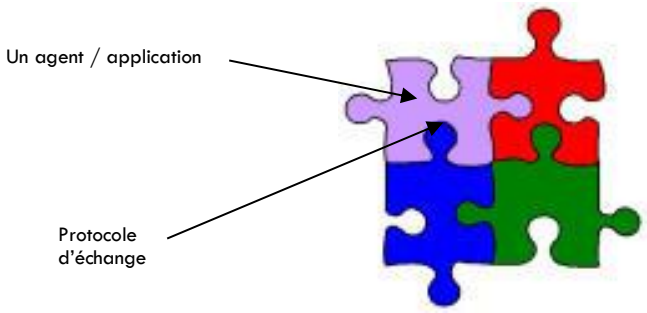


**Figure 1 :** connecter des applications avec le bus ivy

**A quoi cela peut-il bien servir ?**
Bien que limité *à des échanges de* ***messages textuels***, ce principe permet de prototyper très rapidement (en ré-utilisant des agents déjà développés) voire de concevoir des systèmes adaptés spécifiquement aux besoins d'une personne, tout cela indépendamment du système.

Par exemple, pour un système qui aurait à afficher du texte dans une fenêtre graphique, on peut très facilement remplacer cet affichage par un synthétiseur vocal sans faire aucun changement au cœur du système, qui se contenterait d'envoyer sur le bus de l'information à donner aux utilisateurs.

## En résumé :

1. **ivy n'est qu'une extension de la programmation événementielle** utilisant TCP/IP comme vecteur de transport des événements
2. ce **protocole peut être bien évidemment utilisé en complément d'autres protocoles** (exemple : protocole http ; rtp pour la vidéo, CORBA pour l'échange d'objets, …)
3. sa **puissance** (et sa faiblesse) réside(nt) dans la simplicité de la mise en œuvre. Attention néanmoins de ne pas négliger l'aspect conception du protocole d'échange des messages !

Les bibliothèques sont disponibles sur différents systèmes d'exploitation (Sun Solaris, Linux, Win*, Mac OS, Android, …) pour plusieurs langages de programmation (C, C++, C#, Java, Perl, Perl-Tk, Processing, Tcl, Tcl-Tk, Ada, Python, O-Caml, COM, VBA, Adobe Flash, …).

# 2. fonctionnement

Quel que soit le langage utilisé, les principes de fonctionnement restent les mêmes :

1. **Création d'un nouvel acteur du bus**
2. **Connexion de l'agent au bus**
3. **Envoi / réception des messages (par un mécanisme d'abonnement)**
4. **Fermeture de la connexion et destruction de l'agent**

***Nota*** : *L'utilisation d'ivy implique l'utilisation de la pile de protocoles tcp/ip.*

## *2.1 création d'un nouvel acteur du bus*

Cela revient ici à allouer de la mémoire pour l'objet qui va se connecter au "bus".

Quelques exemples de création de cet acteur :

- ```
  Private Ivy bus;
  bus = new Ivy("mon_appli", "mon_apply Ready", null);
  ```
  (***java***)

- ```
  Ivy->init(-loopMode => 'LOCAL', -appName => "mon_appli", -ivyBus =>
  "127.0.0.1:2010");
  ```
  (***perl***)

## *2.2 connexion de l'agent sur le bus*

Le bus ivy se connecte sur un **port** d'une **adresse IP** (ou de broadcast) du réseau local : c'est le point de rendez-vous de tous les agents qui veulent interagir entre eux.

"L'adresse ivy" en elle-même se divise en 2 parties : **adresse_IP (ou adresse de broadcast ou multicast)** et **adresse_port**

L'adresse IP à utiliser est au choix l'adresse de *loopback* (127.0.0.1) propre à chaque machine, l'adresse d'une machine du réseau auquel on appartient (ex : 192.168.0.1) ou une adresse de *broadcast* (solution à préférer, par exemple 127.255.255.255)

***Nota :***
*Une adresse IP se divise elle-même en deux sections, l'adresse du réseau et l'adresse de la machine. L'adresse de broadcast est la dernière adresse adressable sur un réseau. Typiquement pour un réseau de classe C, cette adresse est de la forme* `xxx.xxx.xxx.255` *et pour un réseau de classe B* `xxx.xxx.255.255` *avec "xxx… ", l'adresse du réseau.*
*L'adresse* `127.0.0.1` *a la particularité d'être l'adresse locale de la machine appelée* `localhost`*.*

Il est à noter que l'utilisation d'ivy reste limité à de votre machine ou votre réseau local (*il existe cependant des solutions permettant de passer d'un sous-réseau à un autre*)

Le choix du port n'a pas d'importance particulière à partir du moment où le port sélectionné est libre (i.e. non utilisé par un autre service[1]). Typiquement (et par défaut dans les implémentations d'ivy), le port '2010' est utilisé mais on peut utiliser plusieurs bus à divers ports en même temps.

Voici des exemples de lancement de bus :

- ```
  try {
        bus.start("127.255.255.255:2010");
  }
  catch (IvyException ie) {
        System.err.println("Error : "+ ie.getMessage());
  }
  ```
  (***java***)

- ```
  my $ivy = Ivy->new(-ivyBus => "127.255.255.255:2010", -appName => "mon_appli");

  $ivy->start;
  ```
  (***perl***)

[1] Les ports 1 à 1024 sont réservés par le système pour les services « bien connus » (well known) comme http (port 80), ftp (port 21), …

## *2.3 envoi et réception de messages*

### 2.3.1 envoi d'un message

L'envoi de messages est extrêmement simple : il suffit de préparer la chaîne alphanumérique de données que l'on souhaite diffuser puis on active la fonction *send* (`SendMsg` en java) associée à l'objet bus créé précédemment.

Des exemples :

- ```
  try {
        bus.SendMsg("mon_appli Say=salut");
  }
  catch (IvyException ie) {
        System.err.println("Error: "+ ie.getMessage());
  }
  ```
  (***java***)

- `$ivy->SendMsgs("mon_appli Say=salut");` (***perl***)

### 2.3.2 abonnement et réception d'un message

Pour recevoir des messages du bus, il est nécessaire de s'abonner (fonction **`bindMsg`** en java) à des « patrons » de messages qui permettront l'activation d'une fonction dite de callback (fonction **`receive`** en java) qui traitera le message reçu.

Des exemples :

- ```
  bus.bindMsg("^mon_appli Send=(.*)", new IvyMessageListener() {
     // callback
     public void receive(IvyClient client,  String[] args) {
     ...}
     });
  ```
  (***java***)
- `$ivy->bindRegexp("^mon_appli Send =(.*)",[\&CB_Todo]);` (***perl***)

Ici, le programmeur s'est abonné à tous les messages commençant par « `mon_appli Send=` » et récupérera dans le premier argument du tableau d'argument le message envoyé. Afin de modéliser l'ensemble de « types » de messages, ivy utilise le mécanisme d'expressions rationnelles dites aussi « régulières »

### 2.3.3 regexp - expressions rationnelles

Les expressions rationnelles dites aussi expressions régulières (*regexp* en anglais) permettent de modéliser les prototypes des messages filtrés par un agent ivy. Pour obtenir une information complète sur la syntaxe des expressions régulières, vous pouvez consulter la page : `http://www.regular-expressions.info/gnu.html`
Celles d'ivy utilisent les regexp PCRE (**P**erl **C**ompatible **R**egular **E**xpression)

Quelques opérateurs sont fréquemment utilisés :

- **^** décrit le début d'une expression. Par exemple, "`^MonAppli`" récupère tous les messages qui commencent par `MonAppli`
- **$** décrit la fin d'une ligne. Par exemple, "`^Forme Rectangle$`" récupère les messages de la forme stricte "`Forme Rectangle`"
- **.** décrit n'importe quel caractère
- décrit une chaîne vide ou un nombre de répétition de l'expression précédente. Par exemple, "`^Killer Cmd=(.*)`" filtre et récupère tous les messages qui commencent par la forme "`Killer Cmd=`"

Supposons un agent abonné à l'événement suivant :

- `^CreerRectangle=(.*):(.*):(.*):(.*):(.*):(.*)`

L'arrivée du message : `CreerRectangle=10:10:50:20:plein:jaune` va provoquer en langage Java l'appel de la méthode `public void receive(IvyClient client,String[] args)` avec `args[]` ayant pour valeur `args[0]=10, args[1]=10, args[2]=50,args[3]=20, args[4]=plein, args[5]=yellow`

L'expression régulière est la « *pierre angulaire d'ivy* » : grâce à ce mécanisme, on peut construire des protocoles d'échanges de données très complexes sémantiquement parlant mais très simple au nivaux syntaxique.

### *2.4 fermeture de la connexion au bus*

Enfin la fermeture du bus permet de clore proprement l'échange de données. En voici encore un exemple :

- `bus.stop();` (***java***)

### *2.5 fonctions avancées*

Certaines implémentations d'ivy permettent l'utilisation de fonctions avancées. Parmi ces dernières, vous trouverez comment :

- **s'envoyer des messages à soi-même** (désactivé par défaut) `public void sendToSelf(boolean b);` et `public boolean isSendToSelf();`
- **insérer des retour chariot** (non permis par défaut) dans les messages `protectNewLine(boolean b)`
- **attendre un message ou un client spécifique** pour effectuer un traitement `IvyClient waitForClient(String name, int timeout)` et `IvyClient waitForMsg(String regexp, int timeout)`
- etc.

Référez-vous à la documentation accompagnant votre librairie ivy ! (*javadoc* téléchargeable à l'adresse : `https://github.com/truillet/ivy/blob/master/lib/javadoc-ivy-1.2.18.zip`)

# 3. ivy et java

La librairie ivy est livrée sous forme d'un fichier archive (jar = **j**ava **ar**chive) à inclure dans le *classpath* au moment de la compilation et de l'exécution.
Pour utiliser ivy dans votre code java, utilisez la directive **`import fr.dgac.ivy.* ;`**

Dans la plupart des cas, vous n'aurez à utiliser que la classe `Ivy` et l'interface `IvyMessageListener` (implémentation de la fonction `receive`). N'hésitez pas à consulter la javadoc d'ivy pour l'utiliser de manière optimale !
Voici la liste des fonctions les plus usuelles de la classe Ivy :

- `Ivy(java.lang.String name, java.lang.String message, IvyApplicationListener appcb)` : constructeur de la classe
- `int bindMsg(java.lang.String sregexp, IvyMessageListener callback)` : s'abonne à une expression régulière
- `int bindMsgOnce(java.lang.String sregexp, IvyMessageListener callback)` : s'abonne à une expression régulière pour une réception et une seule
- `java.util.Vector getIvyClients()` : renvoie la liste des clients connectés
- `java.util.Vector getIvyClients(java.lang.String name)` : renvoie la liste des clients connectés dont le nom est donné en paramètre
- `void protectNewlines(boolean b)` : permet l'encodage et le décodage des « \n »
- `int sendMsg(java.lang.String message)` : envoie le message donné en paramètre sur le bus
- `void sendToSelf(boolean b)` : permet l'envoi « à soi-même des messages sur le bus
- `void start(java.lang.String domainbus)` : connecte votre agent sur le bus
- `void stop` : déconnecte l'agent du bus

# 4. un premier exemple pour comprendre

Bien que l'OS Windows soit « préféré » pour effectuer ce TP, vous pouvez tout aussi bien travailler sur une plateforme linux ou macOS : pensez néanmoins à gérer l'ouverture des ports de communication, **source de bien de problèmes** !

- Tout d'abord, téléchargez le fichier `killer_1.4.zip` à l'adresse : `https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/killer_1_4.zip`

- Dézippez le fichier `killer_1.4.zip` : un répertoire killer est créé
  2 fichiers `java` sont présents : `Killer.java`, `Target.java`. Les deux premiers constituent l'exemple à observer.
  - 2 fichiers `bat` et 2 fichiers `sh` (***shell MS-DOS ou shell UNIX***) : pour lancer l'exemple (`exec.bat` et `exec.sh`) et le « *couteau-suisse* » d'ivy (`Probe.bat` et `Probe.bat`)
    **Attention** : le chemin vers java (JRE ou JDK) devra être configuré au préalable !
  - 1 librairie `java` (java archive `.jar`), `ivy-java-1.2.18.jar` pour une JVM (relativement) récente (1.5.x ou supérieure)

- Observez maintenant le fonctionnement du programme `Killer.java` et `Target.java`. Pour cela, lancez les programmes en cliquant sur `exec.bat` (cf. Figure 2). Cliquez sur le bouton "`Kill`". Les instances du programme "`Target`" seront alors détruites.

Quelques outils livrés avec les bibliothèques ivy facilitent certaines opérations notamment le suivi des messages sur le bus et ainsi la compréhension des phénomènes qui le régissent. Par exemple, l'outil "**Probe**" (*API java*) permet de s'abonner à une expression régulière particulière ou à tous les messages échangés à une adresse et un port donné (ici *2010*). Cet outil, **véritable « couteau suisse d'ivy »** permet notamment de tester vos expressions régulières avant d'implémenter votre agent (simulation d'un agent).

Les différentes options de Probe sont accessibles en tapant :

```
java -jar ivy-java-1.2.18.jar -h
```

Pour lancer une instance de « probe » en java, utilisez la ligne de commande suivante :
`java -jar ivy-java-1.2.18.jar "^(.*)" -b 127.255.255.255:2010` (*windows, linux et MacOS*)

***Nota*** :

- Probe se lance par défaut sur l'adresse de broadcast locale (*127.255.255.255*) en utilisant le port 2010
- En java, ces outils qui se trouvaient dans `fr.dgac.ivy` se trouvent depuis la version 1.2.9 dans `fr.dgac.ivy.tools` (ouvrez le fichier `jar` avec *7*zip, Winzip, WinRar par exemple pour accéder à ces différents outils)
- D'autres outils (cf. le point ***6. adresses utiles***) permettent d'utiliser un outil similaire à Probe sous forme graphique (`ivyGUIMonitor` dans le paquetage ivy ou `Visionneur` par exemple)

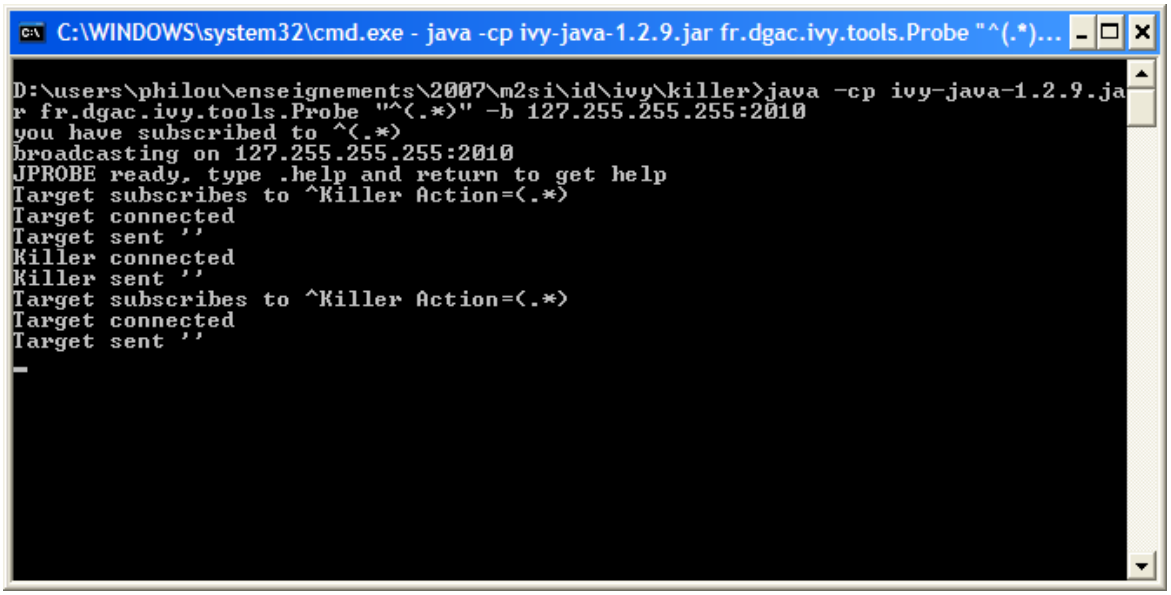


**Figure 2 :** suivi des messages à l'aide de l'utilitaire Probe

# 5. exercices

## *5.1. Maîtriser les expressions régulières et en faire un messagerie*

L'objectif du premier exercice est d'écrire des expressions régulières qui pourront être utilisées dans vos développements ivy.
Vous testerez vos essais avec l'outil « **Probe** » (par exemple en téléchargeant la librairie ivy-java ici : `https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/ivy-java-1.2.18.jar`)
**Vous lancerez deux instances de Probe (cf. plus haut) et testerez vos solutions en ajoutant des regex.**

Soit une application `X` qui émet des messages du style : `X time=1200 receiver=Y msg=blablabla`

- Quelle serait l'expression régulière si :
    - Vous voulez récupérer seulement le nom du récepteur (champ `receiver` du message)
    - Vous voulez récupérer le nom du récepteur et le message (champ `msg`)
    - Tous les champs ?
- Supposons maintenant que des champs optionnels peuvent s'ajouter en fin de message : comment faire pour les récupérer et les traiter ?
- Quels seraient les avantages et inconvénients si on s'abonnait à `"^(.*)"` (tous les messages) ?
- Quel est, selon vous les avantages/inconvénients de structurer un message ivy ?

Développez maintenant un agent ivy permettant :

- D'envoyer un message à l'ensemble des personnes connectées sur la messagerie
- De recevoir et d'afficher les messages envoyés par les autres clients de la messagerie instantanée
- (*optionnel*) D'envoyer un message à une personne particulière connectée (en message privé)

***Nota :*** *pensez que chaque client doit avoir un nom particulier pour pouvoir recevoir des messages privés !*

## *5.2 interagir avec d'autres langages*

- Tout d'abord, téléchargez le fichier `ivy_P5.zip` à l'adresse : `https://github.com/truillet/upssitech/blob/master/SRI/1A/Code/ivyP5.zip`

- Dézippez le fichier, démarrez les deux sketchs Processing et observez le comportement quand vous cliquez sur la fenêtre de « `sender_ivy` ».
- Réécrivez votre agent ivy de messagerie (cf. 5.1) afin de proposer une interface « innovante » de votre messagerie.

## *5.4. réutiliser des agents précédemment développés*

Sur `https://github.com/truillet/icar`

- Récupérez l'agent ICAR (reconnaissance de gestes) `Icar-1.2.zip` et la documentation `icar.pdf`.
- Lisez-la ☺.
- Créez maintenant un petit dictionnaire de reconnaissance de gestes permettant de reconnaître un rectangle et un cercle.
- Utilisez cet agent (en lançant `Icarivy.bat`)
- A l'aide de l'outil de votre choix, étudiez la forme des messages envoyés par ICAR.
- Déduisez-en une expression régulière qui vous permettra de développer une application (`Traceur`) permettant d'afficher dans un `JPanel` ou une `PApplet` Processing la forme reconnue.
- Développez l'application en java ou Processing ☺ (cf. Figure 3)

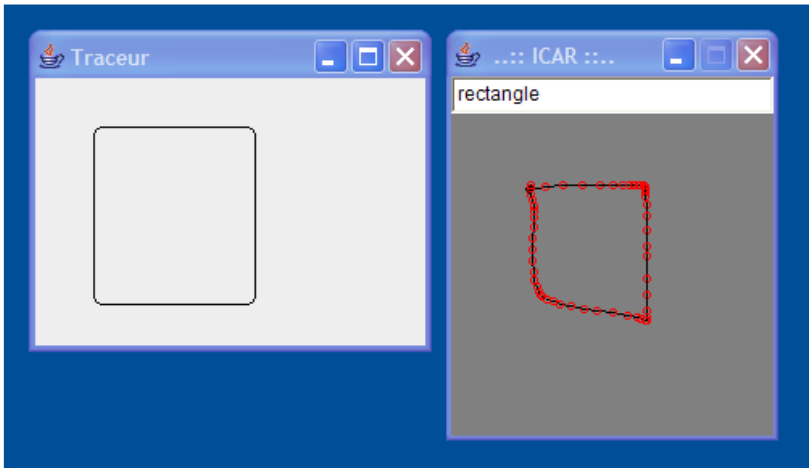


**Figure 3 :** application Traceur avec ICAR

### *5.4 vos propres applications ...*

Maintenant, il ne reste qu'à concevoir et prototyper vos propres applications !

Le principe est d'utiliser le langage le plus approprié pour développer des prototypes haute-fidélité le plus rapidement possible.
Réutilisez, mixez, développez des agents réutilisables et le tour est joué !

# 6. adresses utiles

- `https://github.com/truillet/ivy`

- **Site officiel ivy :** `http://www.eei.cena.fr/products/ivy`
- **SVN ivy :** `https://svn.tls.cena.fr`
- **D'autres librairies**
  - `https://gitlab.com/ivybus/ivy-python`
  - `https://pypi.python.org/pypi/ivy-python`
  - `https://github.com/esden/ivy-c`