OUTIL POUR LA CONCEPTION D'INTERFACES

APPROCHE « BUS LOGICIEL D'ÉCHANGES DE MESSAGES »

http://www.irit.fr/~Philippe.Truillet

v.2.3 – septembre 2020

UNE ARCHITECTURE RÉPARTIE ?

les systèmes informatiques deviennent de plus en plus complexes

- en terme de périphériques utilisés divers et variés
- d'informations échangées
- d'interacteurs et de supports utilisés (fixes et/ou mobiles)

→ **il y a nécessité d'une architecture répartie**

le principe : établir des communications interprocessus

ET aller au-delà du niveau d'abstraction de la *socket*

UNE ARCHITECTURE RÉPARTIE ?

Les inconvénients fréquents des approches réparties …

- une **centralisation** à un moment donné (où se trouve l'objet/ la méthode distante ?)
- un **coût d'apprentissage** élevé des différentes approches
- des architectures fréquemment **spécifiques** (ex : RMI, CORBA, OSGi,…) et peu adaptées à du multi-langage et au développement événementiel

Ceci amène une incompatibilité des modèles d'architecture et modèles d'exécution

APPROCHE RÉPARTIE POUR L'IHM ?

la plupart de middlewares ne sont pas orienté **interaction** mais centrés sur les **échanges/appels d'objets/fonctions …**

En fait, de quoi a t'on réellement besoin en interaction ?

1. séparer le Noyau Fonctionnel de l'interface
2. pouvoir émettre et/ou de recevoir des *événements* et non pas d'appeler des méthodes ou des fonctions !

→ **une solution (parmi d'autres)** : utiliser un bus « *événementiel* » (et il en existe beaucoup !) [

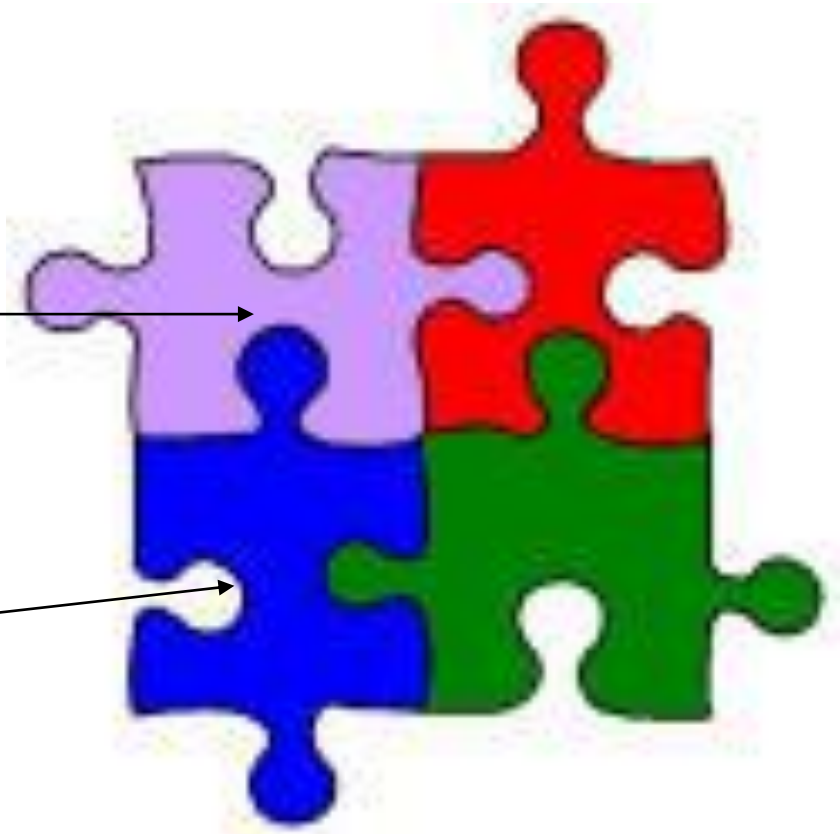
APPROCHE RÉPARTIE POUR L'IHM ?

Le système interactif peut être vu comme « un *assemblage* » d'agents répartis, chaque agent ayant des capacités de calcul et d'interaction avec ses voisins …

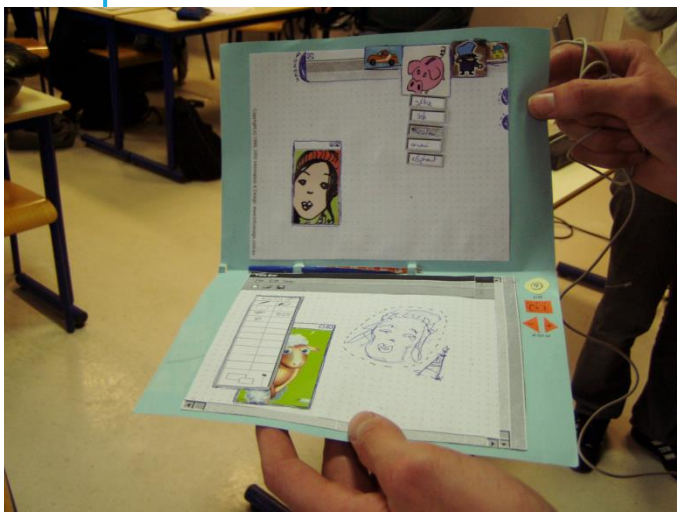
le travail se situe au niveau du protocole d'échange entre agents
(la « ***sémantique*** » de l'événement …)

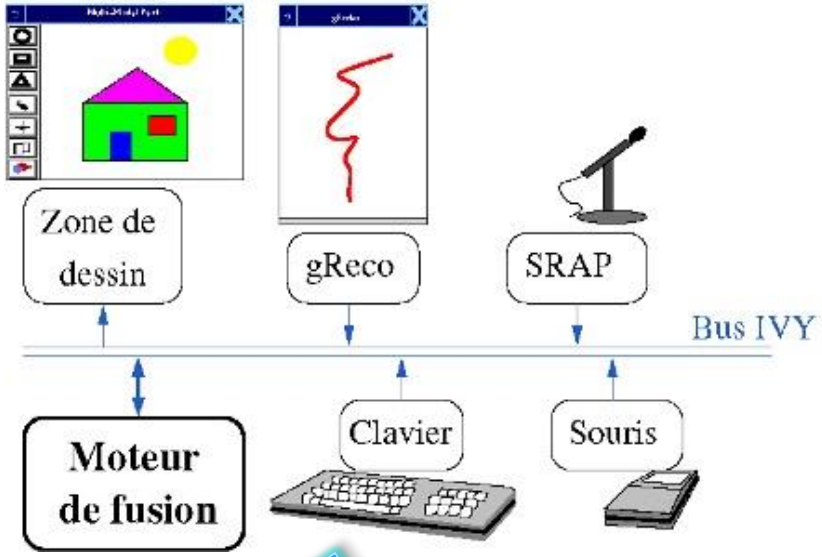
application/agent

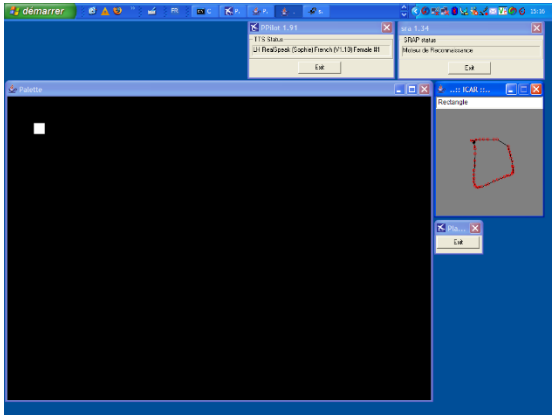
protocole d'échange

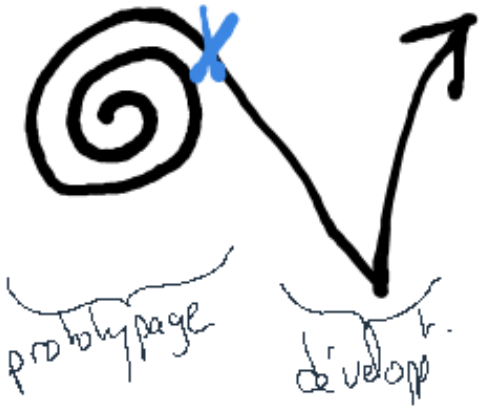


OBJECTIFS DE L'APPROCHE









OBJECTIFS DE L'APPROCHE

Intéressant

pour la **conception…**

1. modularité = réutilisabilité
2. usages de plusieurs plate-formes et langages afin de passer rapidement de la phase « ***papier*** » au(x) ***prototype(s) moyenne/haute fidélité***

et pour **l'évaluation**

- possibilité de tester les différents modules séparément

  → **meilleure visibilité du système**

LE BUS IVY

ivy est un **bus logiciel** qui permet un échange d'informations entre des applications réparties sur différentes machines tournant sous différents OS et écrites avec des langages différents …

créé en 1996 au CENA (DGAC) pour des besoins de prototypage rapide

ivy est simple (**http://www.eei.cena.fr/products/ivy**)

- à comprendre,
- à mettre en œuvre
- et c'est gratuit ;-)

LE BUS IVY

adresse IP
adresse de broadcast
adresse de multicast

port de communication

- ivy n'est pas basé sur un serveur centralisé
  - chaque agent propose un ou des services
  - chaque agent réagit à un ou des événements

- sémantique proche de la programmation événementielle (Java, .NET, X-window, …)

LE BUS IVY

ivy est disponible

- en ada95, C, C++, C#, Flash , java, ocaml , perl, perl/Tk, Processing , python (2 et 3), ruby, Tcl, Tcl/Tk, VBA, …

- sous MacOS, Win32, Win64, Un*x, linux, Android, …

conséquence : la conception est facilitée en profitant des avantages liés à chaque langage de programmation

UTILISATEURS (CONNUS)

Laboratoires

Entreprises

COMMENT PROGRAMMER AVEC IVY ?

ivy

: développé au CENA (DTI R&D)

librairie de « mise en réseau » d'agents

Le bus permet d'ajouter la possibilité de recevoir et d'envoyer des messages avec toutes les APIs nécessaires au développement (ex : SAPI pour la parole [Windows], ARToolkit pour la réalité augmentée, Processing pour de l'animation graphique, …) tout en restant indépendant des OS et des langages !

LE BUS IVY

2 mécanismes « basiques » : la réception (**bindMsg**) et l'émission (**sendMsg**) de messages

```
/* listener ivy */
bus.bindMsg("ivyEcobe! to=(.*) event=(.*)", new IvyMessageListener() {
  public void receive(IvyClient client, String[] args)
  {
    try
    {
      /* envoi vers le robot EcoBe!*/
      if (args[1].compareTo("Stop")==0)
      {
```

Fonction callback

```
// Envoi que sur trames GGA -> les autres ne servent qu'à mettre à jour les champs
bus.sendMsg(name + " type=" + DT + " temps="+time + " lat="+lat + " long="+lon + " alt="+
altitude +" vitesse="+vitesse + " cap="+cap + " mode="+mode + " HDOP="+ HDOP + " Nb_Satellites=" +
Nb_Satellites + " Force_Signal=" + Force_Signal);
```

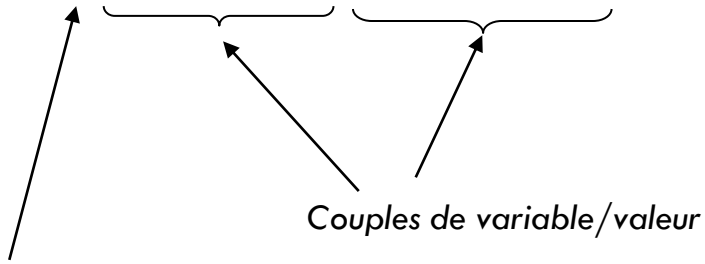
LE BUS IVY

le protocole d'échanges de messages est *purement textuel*
(abonnement par expressions rationelles / **regexp PCRE**)

Vous êtes libres du format d'échange … MAIS il vaut mieux être structuré …

- exemples d'envoi :
  ICAR command=back

  IMM media=SRAP action=previous



Essayer les regex !

https://regex101.com/

LE BUS IVY

exemples d'abonnement :

^ICAR command=(.*)

^IMM (.*) action=(.*)

4 étapes essentielles :

1. Créer un agent
2. Définir les comportements (envoi/réception)
3. « Lancer l'agent sur le bus »
4. Arrêter le l'agent sur le bus avant de quitter

LE BUS IVY

Les éléments de **regexp** les plus utilisés :

^ : « doit débuter par »

.* : n'importe quel caractère répété n fois

(.*) : permet de récupérer un argument (chaîne de caractères)

Ex : **^**Appli timestamp=.* x=(.*) y=(.*)

Le message « **Appli timestamp=123456 x=12 y=56** »
permet de récupérer les arguments 12 et 56

LE BUS IVY

Recevoir un message consiste :

1. À extraire les arguments intéressants
2. A les « caster » dans le type souhaité
3. Les utiliser

Un exemple →

LE BUS IVY

```
void mousePressed()
{
  try
  {
    bus.sendMsg("Demo_Processing Command=Damn ... New message");
  }
  catch (IvyException ie)
  {
  }
}
```

```
try
{
  bus = new Ivy("demo", " demo_processing is ready", null);
  bus.start("127.255.255.255:2010");

  bus.bindMsg("^Demo_Processing Command=(.*)", new IvyMessageListener()
  {
    public void receive(IvyClient client,String[] args)
    {
      message = args[0];
      try
      {
        bus.sendMsg("Demo_Processing Feedback=ok");
      }
      catch (IvyException ie) {}
    }
  });
```

UNIVERSITÉ TOULOUSE III PAUL SABATIER

LE BUS IVY

Chaque argument de la **regex** (que l'on retrouve entre « () ») est accessible au travers d'un tableau de valeurs et directement exploitable

La fonction callback s'exécute quand la regex est vérifiée

Ex :

Regex → Appli X=(.*) Y=(.*)

Messages envoyés

Appli x=12 Y = 14 n'aura aucun effet

Appli X=12 Y=14 permettra de récupérer
2 arguments de type « *string* » dans args[0] et args[1]
(en java)

CONCLUSIONS

l'approche « *bus événementiel* » permet au final :

- de **se focaliser sur les problèmes de conception** et non sur la façon de les implémenter
- et **de prototyper très rapidement** pour « *donner à voir* » et « *donner à tester* »

LIENS

Sites officiels d'ivy

- http://www.eei.cena.fr/products/ivy (maintenance aléatoire)
- https://svn.tls.cena.fr (*Subversion*)

Sites spécifiques « librairies »

- **Python** : https://gitlab.com/ivybus/ivy-python et https://pypi.python.org/pypi/ivy-python
- **C :** https://github.com/esden/ivy-c/
- **Java** : ~~http://lii-enac.fr/~jestin/homepage/software.html~~

Github SRI3

- **https://github.com/truillet/upssitech/wiki/Interaction-Distribuée**