Solutions pour la représentation des concepts LfP en UML2.0

Type : **SPEC**	Projet RNTL **MORSE**	Réf : **SPEC-REPR-UML2**
Date : **27 octobre 2005**		Sous-projet n° : **1**
Version : **3.0**		Tâche n° : **1.2**
Auteurs : **Maurice Assouline**		
Relecteurs :		

Liste des révisions

Révisions	Planche	Commentaire
Version : 1.0 *Date* : Janvier 2005 *Auteur* : Maurice Assouline		Version initiale, correspondant au document Word SPEC-UML-LfP, version 1.0 du 21/06/2004.
Version : 2.1 *Date* : Juillet 2005 *Auteur* : Maurice Assouline	1 à 60	Version fournissant des solutions de représentation pour tous les concepts de base de LfP utilisés dans l'exemple Client-Serveur.
Version : 3.0 *Date* : Octobre 2005 *Auteur* : Maurice Assouline	1 à 104	Cette version est a priori complète. Elle fournit une représentation pour tous les éléments du langage LfP. Les solutions de représentation proposées tiennent compte des limitations d'Ameos 10.0. Elles ont été testées sur Ameos pour les exemples Client-Serveur et Messagerie.

Sommaire

- 1. Diagrammes d'Architecture
  - Exemple : DA de l'application Client-Serveur
- 2. Déclarations
  - types de données, variables, constantes
- 3. Diagrammes de Comportement
  - Exemple : DC de l'application Client-Serveur
    - Diagramme principal de la classe Server
    - Diagramme de méthode : Server.handle_request
    - Diagramme principal de la classe Client
    - Diagramme principal du média RPC

Sommaire

Diagramme d'architecture LfP

- Les diagrammes d'architecture LfP seront représentés principalement par des diagrammes de composants UML2.0
  - *$ « Component diagrams »* : ce terme a été redéfini en UML2.0, il n'a plus la même signification qu'en UML 1.x
  - C'est une sorte de diagramme de classes, où sont définies les notions de composants, ports, interfaces, connecteurs, parties d'un composant ou d'une classe.

Notions UML utilisées dans les diagrammes de composants

- Composant
- Interface offerte, interface requise
- Connecteur
  - d'assemblage,
  - de délégation.
- Port
- Parties
- Diagrammes de composants

(Pour les détails, voir définition d' UML2.0)

Diagrammes UML complémentaires

- Certains détails des aspects statiques du modèle LfP ne peuvent pas être représentés sur un diagramme de composants UML2.0,
  - Soit pour des raisons inhérentes au langage UML2.0,
  - Soit à cause de certaines limitations de l'outil de modélisation UML2.0.
- Ces aspects statiques complémentaires seront alors représentés
  - sur des diagrammes de classes - tout composant UML2.0 est aussi une classe, et peut être représenté comme tel,
  - ou sur des diagrammes de structure interne ($ *composite structure diagram*) – un composant UML2.0 est une « *classe structurée* ».
- exemples : opérations d'un composant LfP, parties internes à un composant LfP.

Diagramme d'architecture LfP

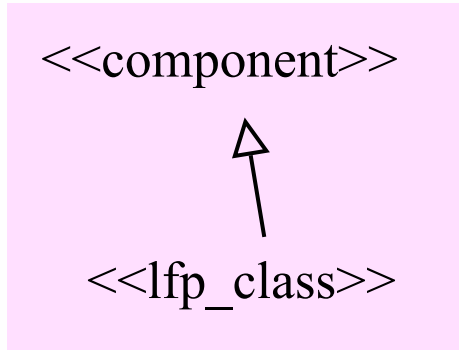
Principales entités LfP à représenter :

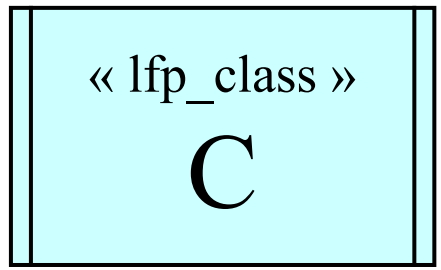
- Classes
- Médias
- Binders
- Ports
- Arcs de liaison

Note : En LfP, le terme « composants » désigne les classes et les médias

Représentation d'une Classe LfP

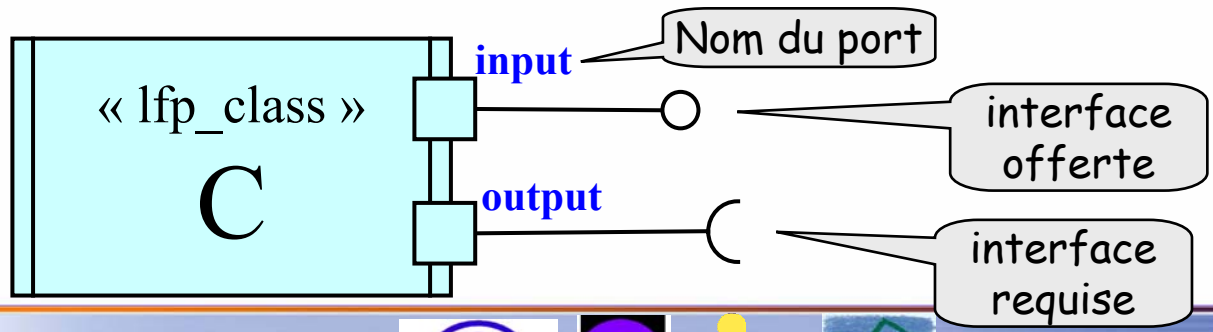
- Spécialisation du « *composant* » UML ($ *component*), muni du stéréotype « lfp_class »
- particularités sémantiques :
  - Exécution active
  - Exécution séquentielle (sans parallélisme interne)





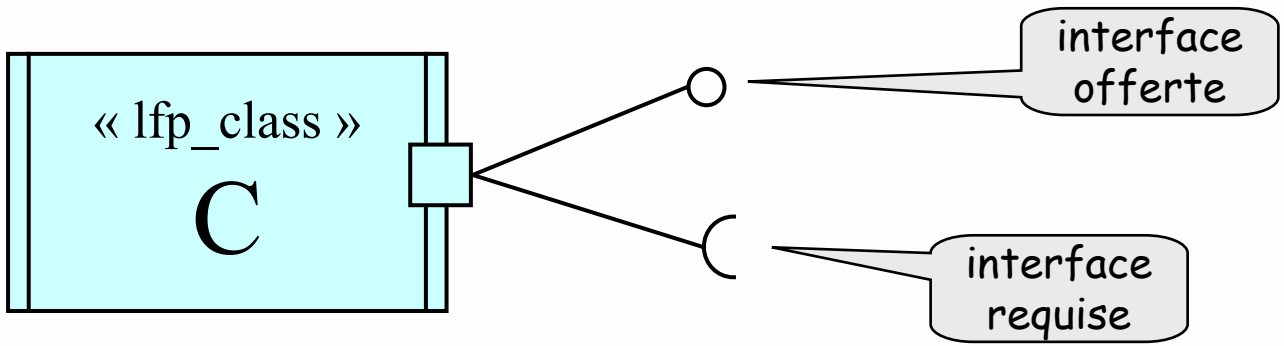
Représentation des Ports et Interfaces d'un composant LfP

- Un composant UML peut, comme un composant LfP
  - exporter des services - décrits dans une *interface offerte*,
  - importer des services - décrits dans une *interface requise*,
- Les messages de demande de service et leurs réponses sont communiqués à travers des points d'interaction appelés *ports* du composant UML.
  - En LfP, les ports sont des références vers des tampons de messages appelés *binders*. Un composant LfP émet ou reçoit des messages via des *binders*, qu'il accède au moyen de ses ports : on peut donc aussi voir les ports d'un composant LfP comme ses points d'interaction avec l'extérieur.
- un port LfP correspondant à des messages entrants est représenté par un port UML lié à une interface offerte (symbole *boule*)
- un port LfP correspondant à des messages sortants est représenté par un port UML lié à une interface requise (symbole *socket* = *demi-cercle*)



Représentation des Ports et Interfaces d'un composant LfP

- un port LfP correspondant à la fois à des messages entrants et sortants est représenté par un port UML lié d'une part à une interface offerte (symbole *boule*), d'autre part à une interface requise (symbole *socket* = *demi-cercle*)



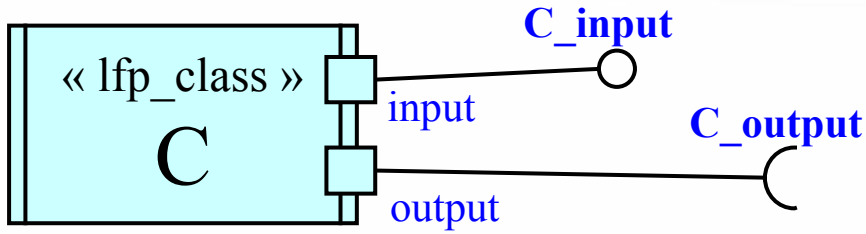
Interface offerte d'une classe LfP

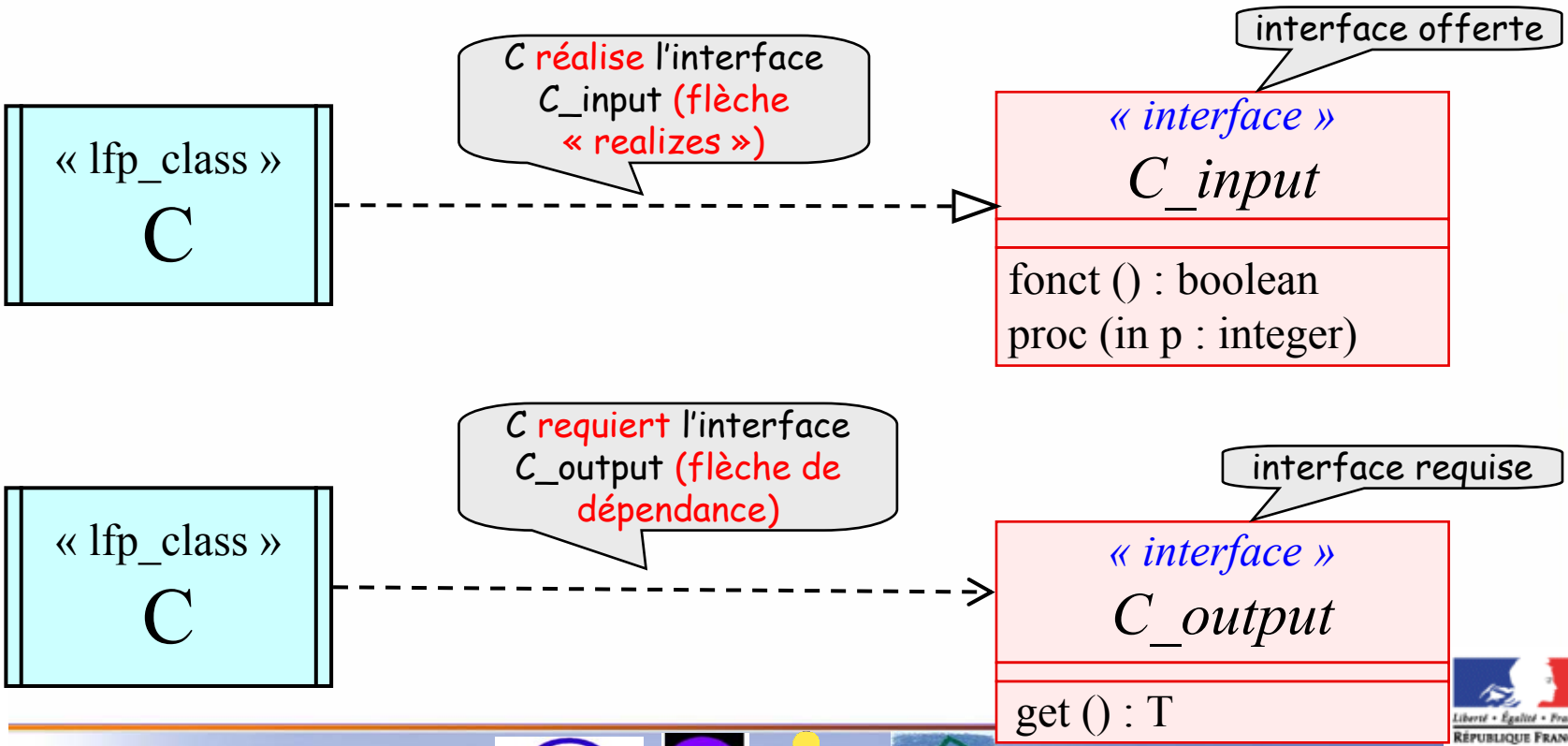
- Pour un composant UML,
  - on peut décrire globalement son interface offerte, qui définit l'ensemble des services qu'il offre,
  - mais on peut aussi diviser cette interface globale en plusieurs interfaces offertes partielles dont chacune correspond à un des ports du composant, et définit donc les services offerts à travers ce port.
- Pour représenter l'interface offerte d'une classe LfP,
  - on décrira pour le composant UML correspondant, autant d'interfaces offertes partielles que la classe a de ports.
  - Chaque interface offerte partielle définit l'ensemble des opérations exportées par la classe via le port associé, ou (plus largement) l'ensemble des messages reçus par la classe via ce port. (ne contient pas d'attributs)

Interface requise d'une classe LfP

- De même,
  - l'interface requise d'un composant UML peut être fractionnée en plusieurs interfaces requises, associées chacune à un port.
- Pour représenter l'interface requise d'une classe LfP,
  - on décrira pour le composant UML correspondant autant d'interfaces requises partielles que la classe a de ports.
  - Chaque interface requise partielle définit l'ensemble des opérations externes invoquées par la classe via le port associé, ou (plus largement) l'ensemble des messages émis par la classe via ce port. (ne contient pas d'attributs)
- On peut dériver le nom de chaque interface du nom du port associé. (ex. : port C.input =>interface C_input ou C_input_interface)

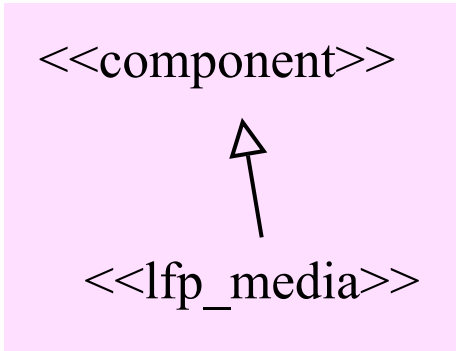
Interface d'une classe LfP : exemple

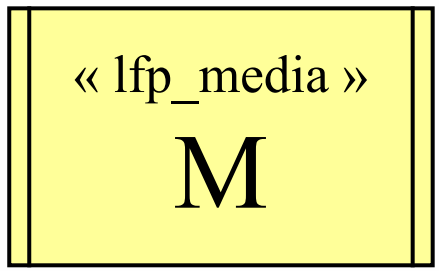




Représentation d'un média LfP

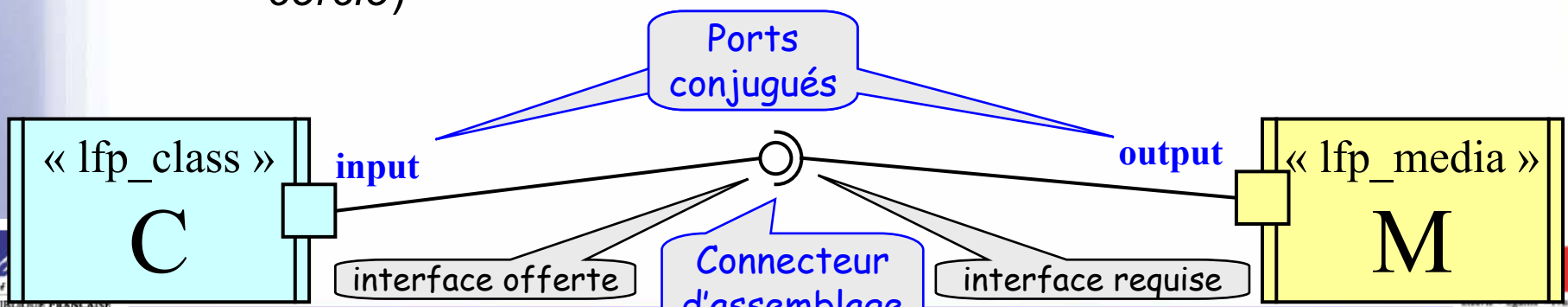
- Spécialisation du *composant* UML ($ *component*), muni du stéréotype « lfp_media »
- particularités sémantiques :
  - Exécution active
  - Pas de méthodes
  - Avec ou sans attributs (état)
  - Ports conjugués de ceux des classes LfP auxquelles le média est connecté
  - Un protocole lui est associé
    - En LfP, protocole décrit par un diagramme de comportement. En UML : voir plus loin





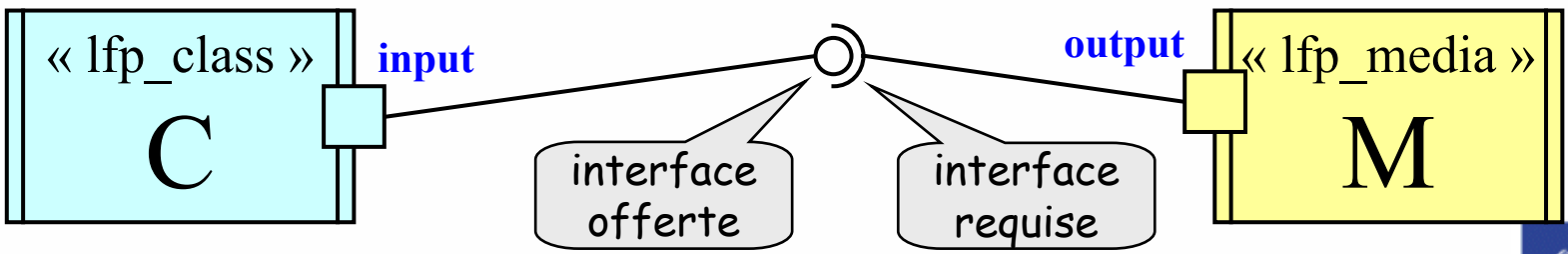
Représentation d'un Binder LfP

- Le binder est représenté par un Connecteur d'assemblage et 2 ports conjugués, dont l'un appartient à une «lfp_class», l'autre à un «lfp_media».
- Les ports conjugués partagent une même interface
  - Celui des 2 ports qui correspond à des messages entrants (ex. *C.input*) est représenté par un port UML lié à la partie interface offerte du connecteur (symbole *boule*),
  - L'autre port correspond à des messages sortants (ex. *M.output*), il est représenté par un port UML lié à la partie interface requise du connecteur (symbole *socket* = *demi-cercle*)



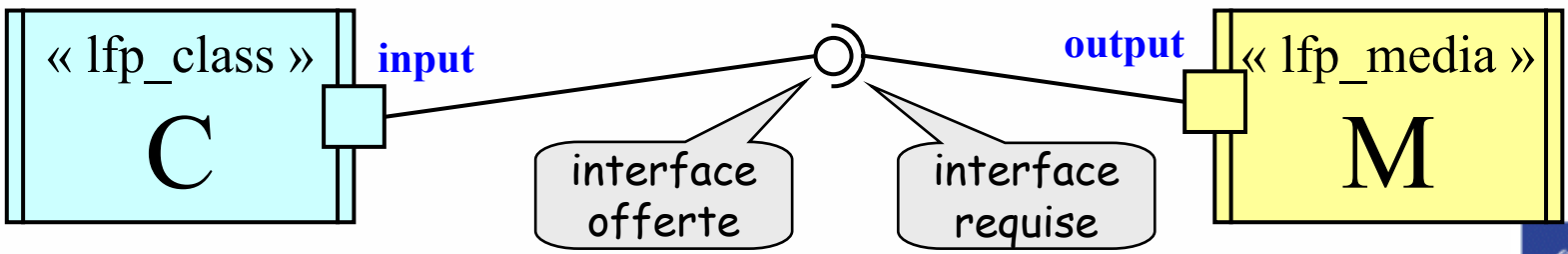
Pas de symbole *binder*

- Attention, le symbole *boule-socket* ne remplace pas le symbole binder de LfP, il ne symbolise pas un tampon de messages. Il représente les interfaces offerte et requise (entités non instanciables).
- Sous la forme UML, les binders ne sont pas représentés directement. Pas de symbole *binder* : les tampons de messages restent implicites.
  - Le concepteur UML-Morse doit être conscient qu'un port est une référence vers un tampon de messages (tampon double si la communication est bidirectionnelle).



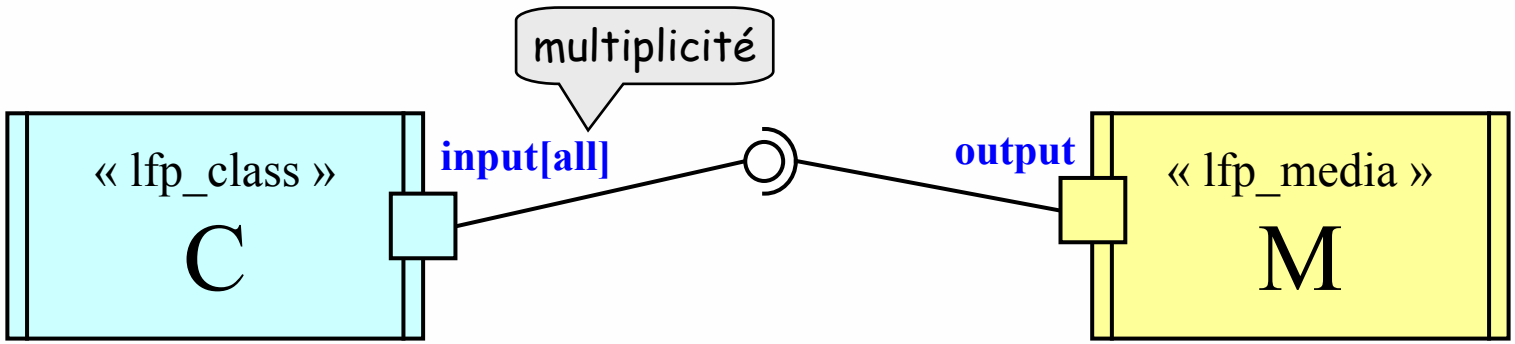
Interfaces conjuguées

- L'interface offerte par une classe LfP à travers un port contient l'ensemble des méthodes de la classe qui sont appelables via ce port. En LfP sa description est omise, son contenu découle du DC de la classe. En UML on peut choisir de la représenter.
- Pour le port conjugué, côté média, l'interface requise est identique, sa description n'a donc pas lieu d'être fournie.
- Contrainte : un connecteur d'assemblage ne peut pas relier 2 composants « lfp_class », ni 2 composants « lfp_média »



Multiplicité d'un binder

- *Multiplicité* d'un binder : représentée par la multiplicité UML attachée au port correspondant de la classe connectée au binder.
  - Valeurs possibles [1] ou [all]. Si l'outil UML n'autorise pas la valeur [all], on convient de remplacer cette valeur par [*]
  - La sémantique attachée à ces deux valeurs est celle de LfP.
  - Multiplicité par défaut = 1
  - On n'indique pas de multiplicité pour les ports des médias



Attribut “binding” d’un binder : port “attaché” au binder

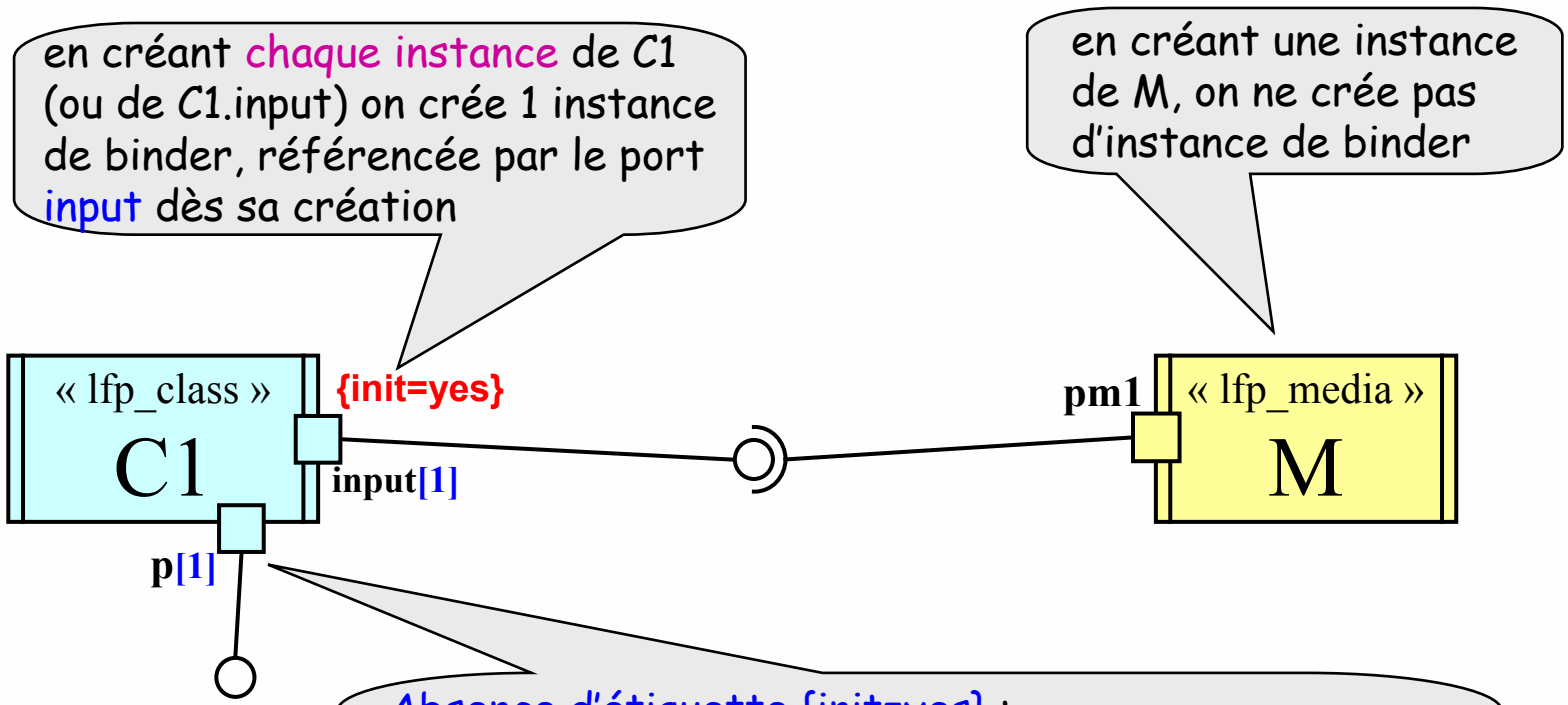
- Un port est une référence vers un binder (tampon de messages - tampon double si la communication est bidirectionnelle)
- Chaque binder est référencé par 2 ports (ou plus), appartenant à 2 composants différents (ou plus). Mais
  - les instances d’un de ces ports et d’un seul sont initialisées automatiquement, de façon à référencer une instance du binder dès leur création.
  - Ce port est dit « attaché » au binder. Il est désigné par un attribut du binder, l’attribut *binding*.
  - le composant correspondant (celui qui contient ce port) est toujours une classe, appelée ici *classe attachée au binder*.
  - et dans certains cas la création d’instances de cette classe entraîne la création d’instances du binder : voir ci-après.

Les 2 modes de création des instances de binder

- La multiplicité d'un binder influe sur le mode de création de ses instances :
  - Si la multiplicité est 1, une instance du binder est créée lors de la création de chaque instance de la classe *attachée* au binder. L'instance correspondante de ce port est alors initialisée pour référencer l'instance de binder créée.
  - Si la multiplicité est all, une instance unique du binder, partagée par toutes les instances de la classe attachée au binder, est créée lors de l'initialisation de l'application. A la création de chaque instance de la classe, l'instance correspondante du port *attaché* est initialisée pour référencer cette instance partagée du binder.
- Les ports des médias ne sont jamais désignés par un *binding* (jamais *attachés* à un binder).
- Si un port d'une classe n'apparaît dans aucun binding, il pourra être connecté à un binder existant, soit à sa création s'il est déclaré avec une initialisation explicite, soit au moyen d'une affectation ultérieure.
  - Il en est toujours ainsi pour les ports des médias.

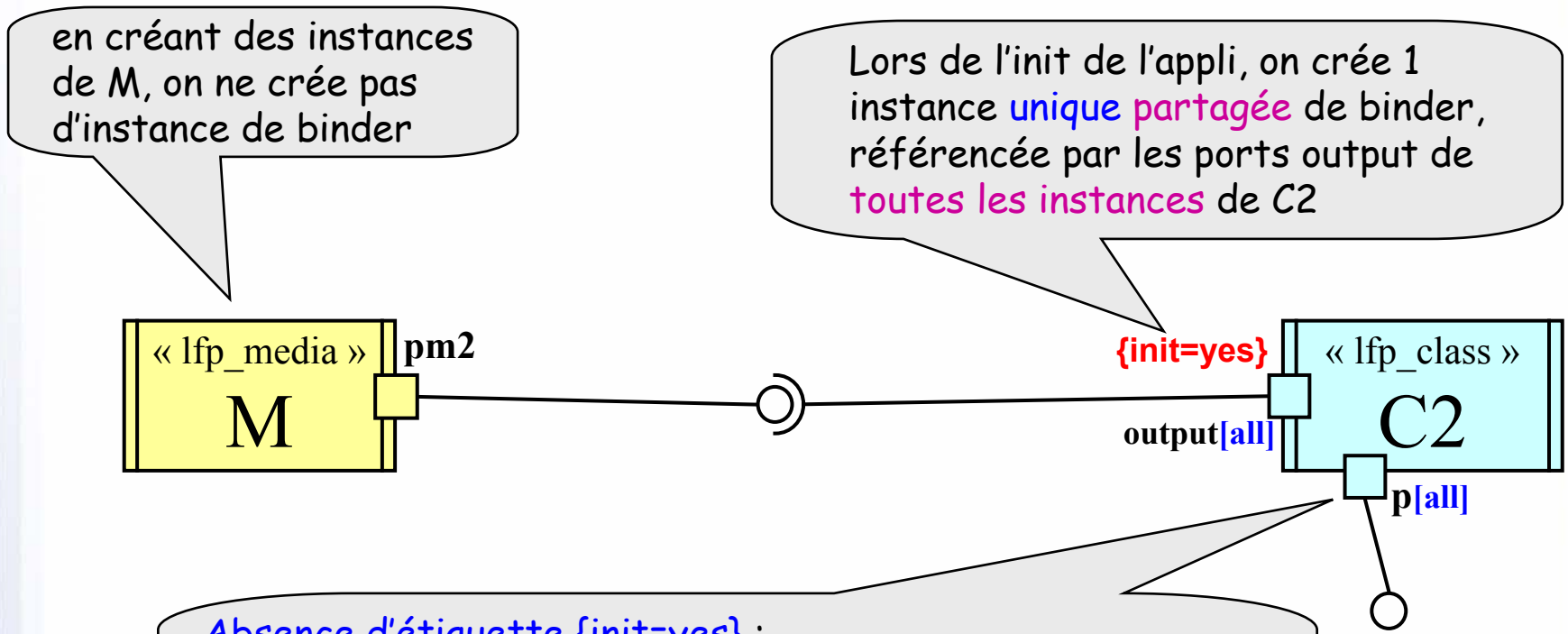
"*binding*" d'un binder de multiplicité 1 : représentation UML

- Une étiquette {init=yes} qualifie le port « *attaché* » au binder



"*binding*" d'un binder de multiplicité all : représentation UML

- Une étiquette {init=yes} qualifie le port « *attaché* » au binder

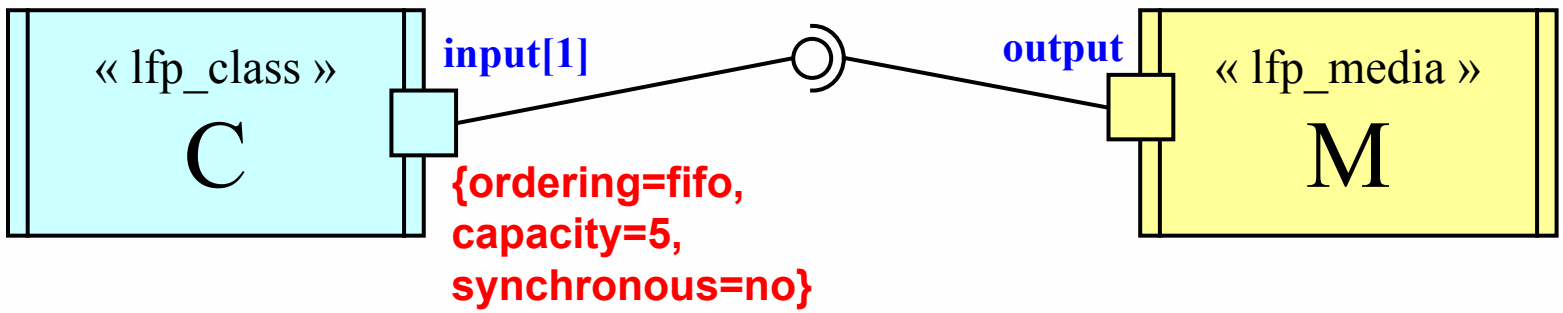


Déclaration des ports : UML ≠ LfP

- En LfP, les ports n'apparaissent dans le diagramme d'architecture que par leur éventuel binding.
- Ceux qui n'ont pas de binding ne sont vus que par leur déclaration dans un diagramme de comportement.
  - En particulier les ports des médias sont absents du diagramme d'architecture.
- En UML, tous les ports sont modélisés et nommés (« déclarés ») dans le diagramme de composants
  - y compris ceux qui correspondent à un port LfP n'ayant pas de binding.
  - Seuls ceux qui correspondent à un port LfP ayant un binding sont munis de l'étiquette {init=yes}.

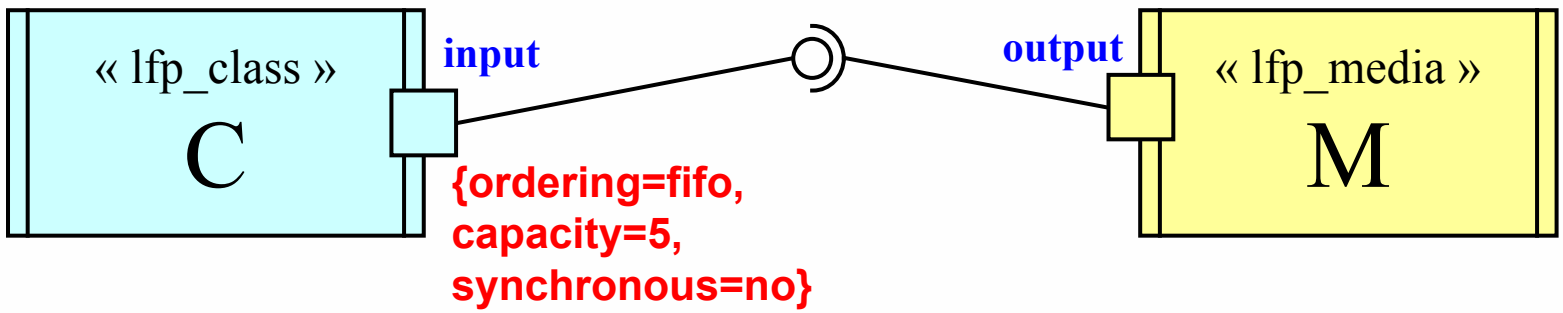
Autres attributs des binders

- Représentés par des étiquettes (*tagged values*) *ordering*, *capacity*, et *synchronous* attachées au port de la classe (les mêmes valeurs s'appliquent au port conjugué du média, sans qu'il soit nécessaire de dupliquer les étiquettes).



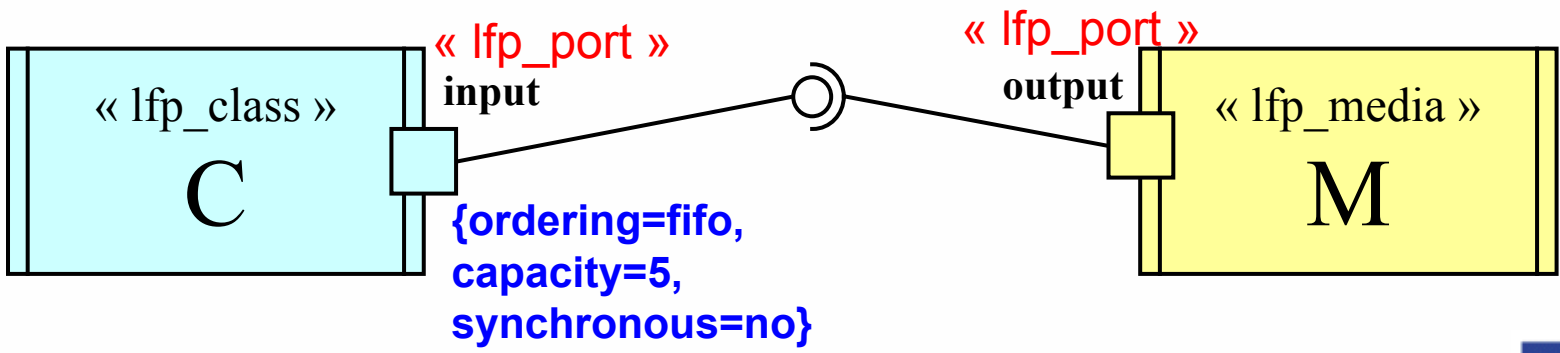
Attributs des binders : valeurs

- *ordering* peut valoir fifo ou bag (par défaut:fifo),
- *capacity* a une valeur entière (par défaut:1),
- *synchronous* peut valoir yes ou no (par défaut yes).



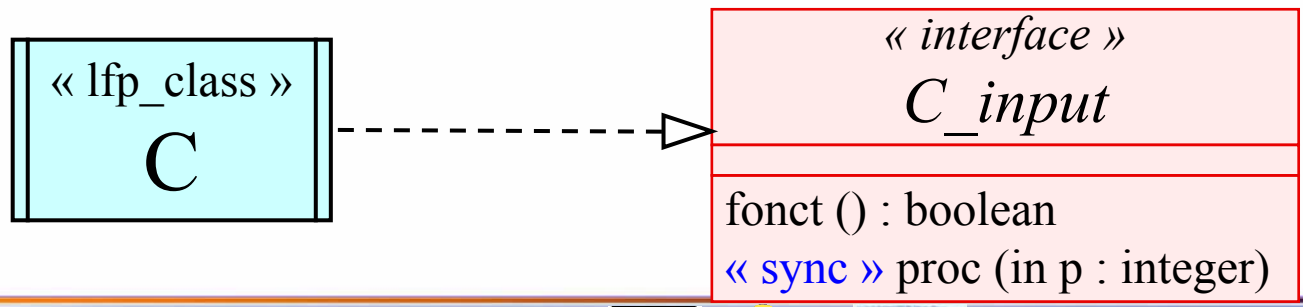
Stéréotype « lfp_port »

- Les outils de modélisation appliquent en général la règle UML2.0 qui oblige à rattacher chaque étiquette d'un profil à un stéréotype;
- les étiquettes (*init*, *ordering*, *capacity*, *synchronous*) affectées aux ports nous amènent donc à définir un stéréotype qui devra être systématiquement appliqué aux ports : le stéréotype « lfp_port »
  - Pour alléger les diagrammes ce stéréotype aurait pu rester implicite car la sémantique qu'il représente est associée à tous les ports sans exception, mais son omission n'est pas autorisée par les outils.



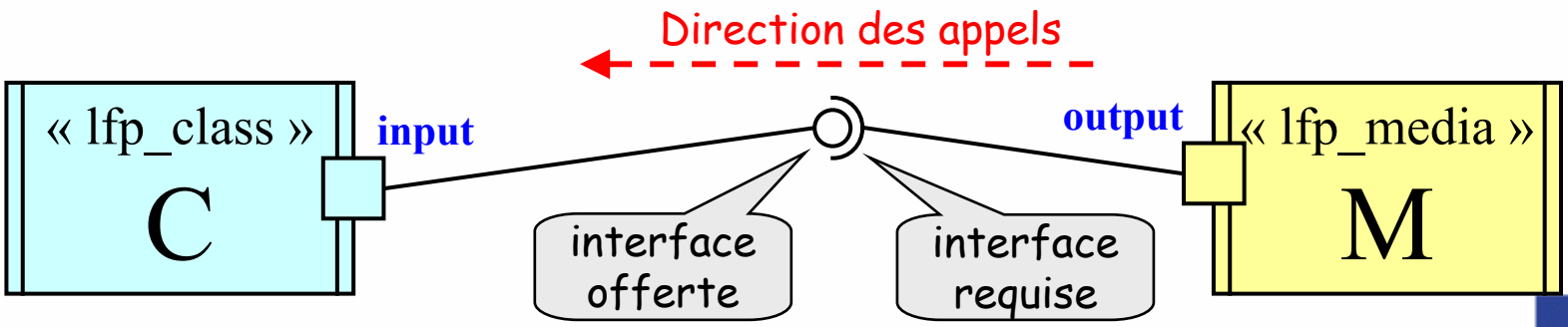
Méthodes synchrones / asynchrones

- Méthode synchrone : pendant son exécution, l'appelant doit se suspendre (attendre le message de retour).
- Toute fonction LfP, et toute procédure LfP dont au moins un paramètre est en mode *out* ou *inout*, est obligatoirement synchrone.
  - Il est inutile d'indiquer un stéréotype « sync »
- Si tous les paramètres d'une procédure sont en mode in
  - Un stéréotype « sync » peut être indiqué pour spécifier que la procédure est synchrone;
  - si le stéréotype « sync » n'est pas indiqué, la procédure est asynchrone.
- Le stéréotype précède la déclaration de la méthode. Celle-ci peut figurer dans le composant ou dans son interface.



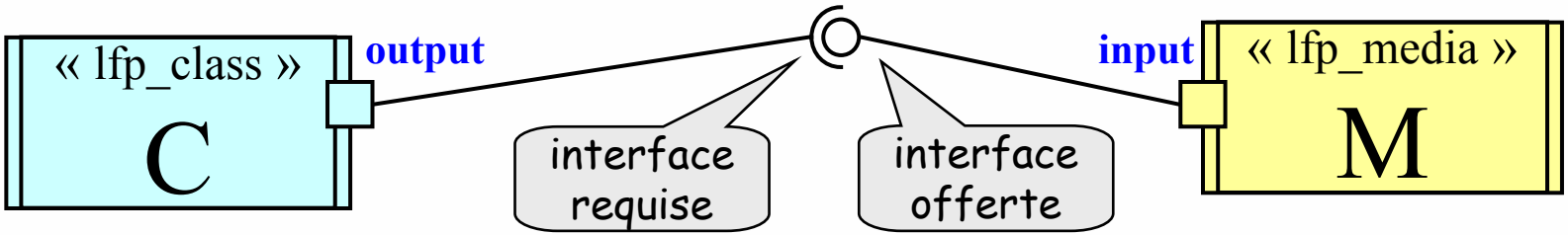
Orientation des arcs LfP, sens des messages

- L'orientation des arcs LfP est représentée par la socket et la boule : les messages transitent de la socket vers la boule. A noter :
  - Si l'interface comporte au moins une méthode synchrone, la communication est implicitement bidirectionnelle, car un message de retour d'appel répond à un message d'appel;
  - Si l'interface ne comporte que des méthodes asynchrones, la communication est implicitement unidirectionnelle.

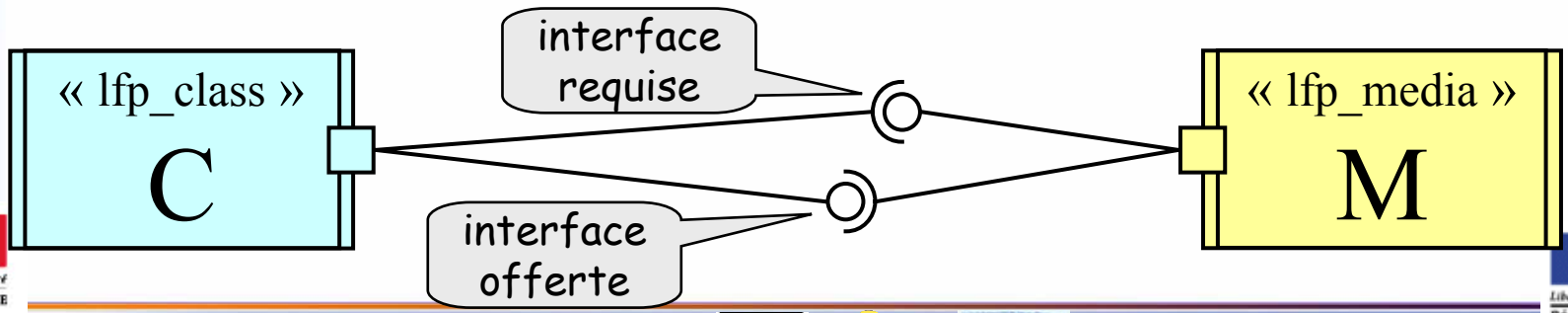


Echanges croisés via un même binder

- Classe LfP du genre « Client » (avec interface requise)

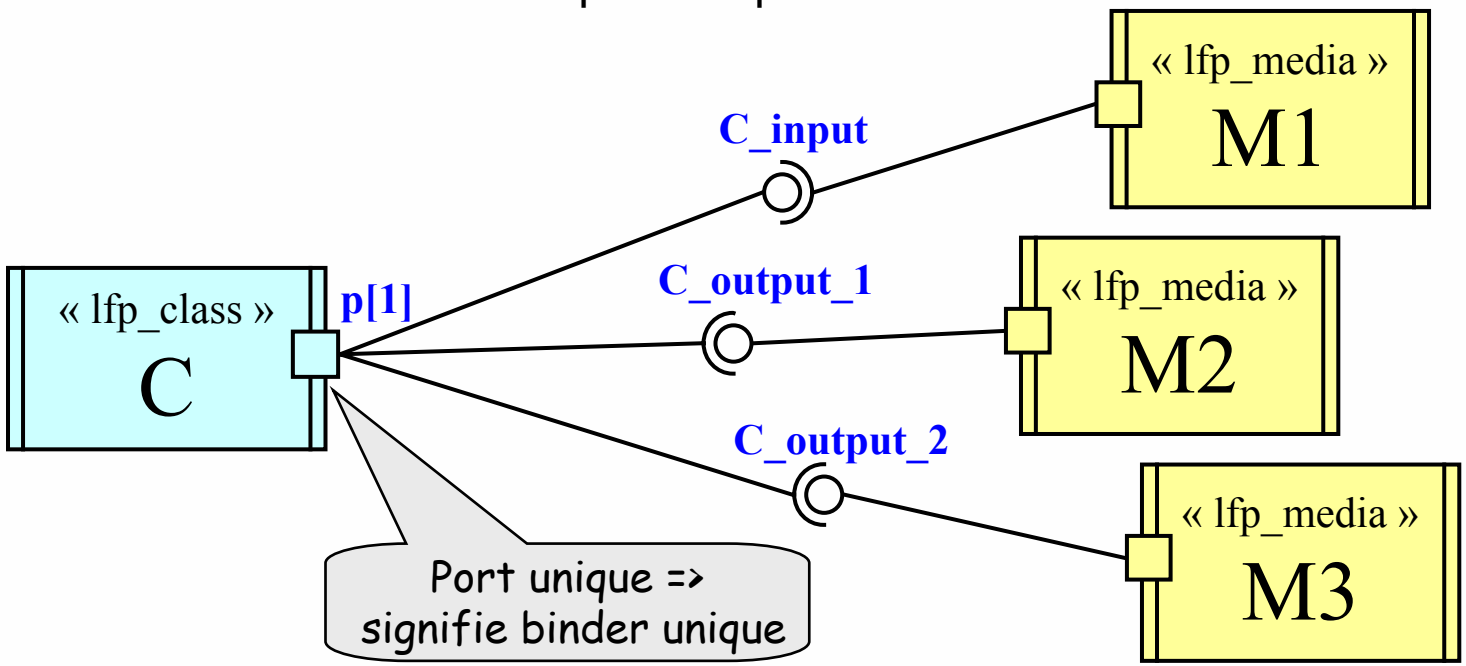


- Classe LfP à la fois Serveur et Client, avec interfaces offerte et requise via un binder unique :
  Le binder est représenté par un couple unique de ports conjugués, reliés par 2 connecteurs d'assemblage de directions opposées.



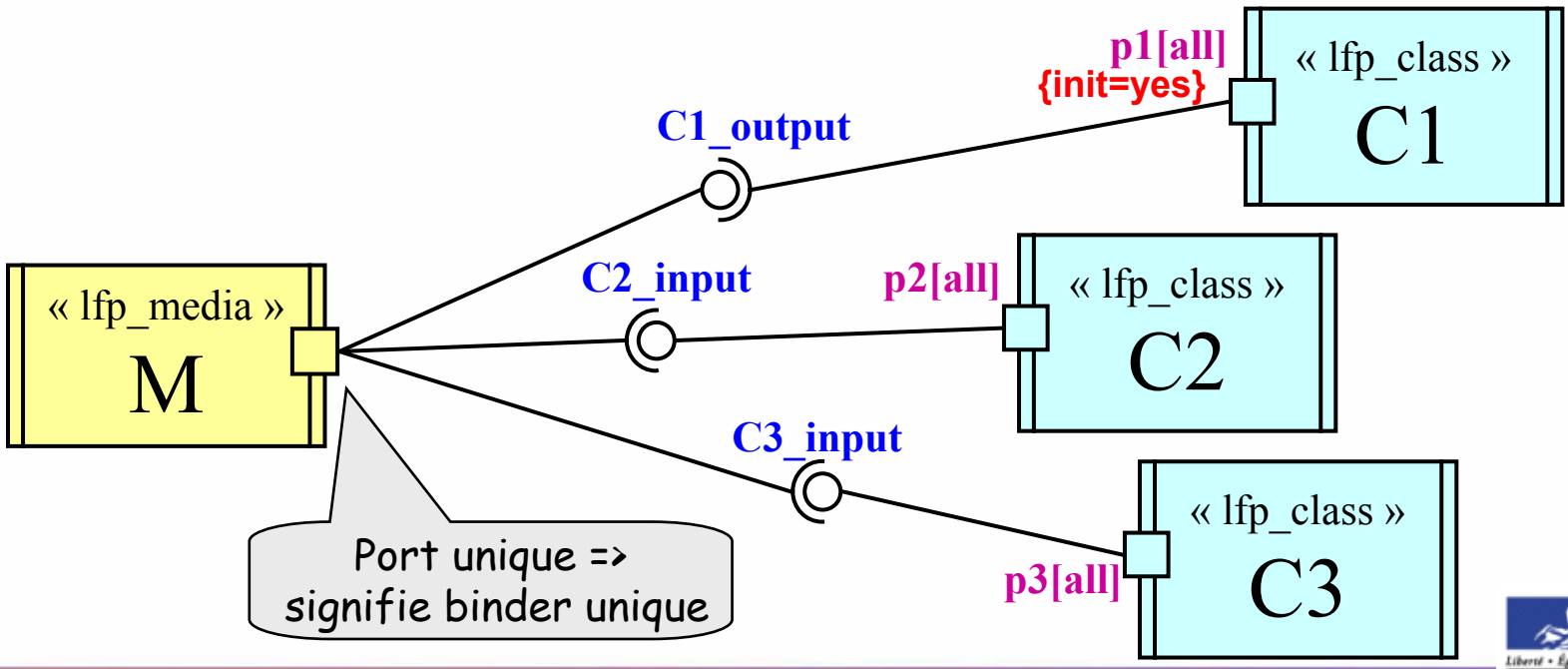
Communication entre une classe et plusieurs médias via un même binder

- Une classe peut ajouter ou retirer dans un seul et même binder des messages ayant comme destinataires ou émetteurs plusieurs médias différents.
  - Côté médias : un port par média
  - Côté classe : un port unique



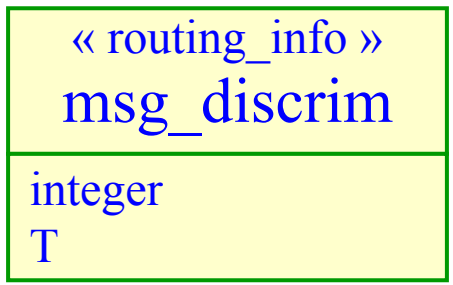
Communication entre un média et plusieurs classes via un même binder

- Un média peut insérer ou retirer dans un seul et même binder des messages ayant comme destinataires ou émetteurs plusieurs classes différentes.
  - Côté classes : un port par classe
    - NB: Un seul de ces ports, indiqué par {init=yes}, est *attaché* au binder (que la multiplicité soit 1 ou all)
  - Côté média : un port unique



Type de discriminant et binders, Type de discriminant et type port

- Tous les messages transitant par un binder donné ont des discriminants de même composition. Un type de discriminant définit la composition et la structure commune à tous ces discriminants.
- Chaque binder est donc lié à un type de discriminant. Un type de discriminant sera représenté par une classe UML, stéréotypée « routing_info ».
- Le type de chaque composant du type discriminant sera représenté par un attribut de la classe « routing_info », dont le nom seul est indiqué : ce nom d'attribut est le nom du type du composant.



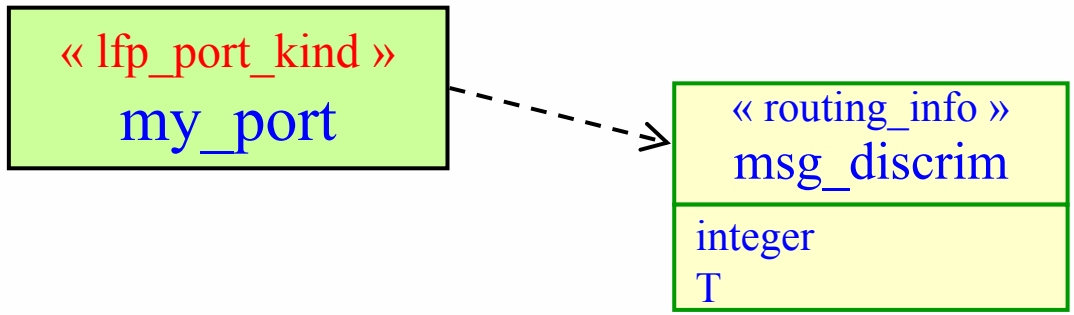
- Un type port définit un ensemble de ports qui référencent tous des binders de même type, c'est-à-dire des binders liés à un même type de discriminant. Chaque type port est donc lui aussi lié à un type de discriminant.

Représentation d'un type port

- Un type port LfP (dont les instances sont des références vers des binders) est représenté par une classe UML, stéréotypée « lfp_port_kind ».
- Chaque type port est caractérisé par un type de discriminant correspondant. Ceci est modélisé par une relation de dépendance entre le type port et le type discriminant.
- Exemple : le type port défini en LfP comme suit :

```
type my_port is port (integer, T);
```

est associé à un type discriminant ayant un composant de type *integer* et un autre de type *T*. Il est représenté par la classe UML ci-dessous :

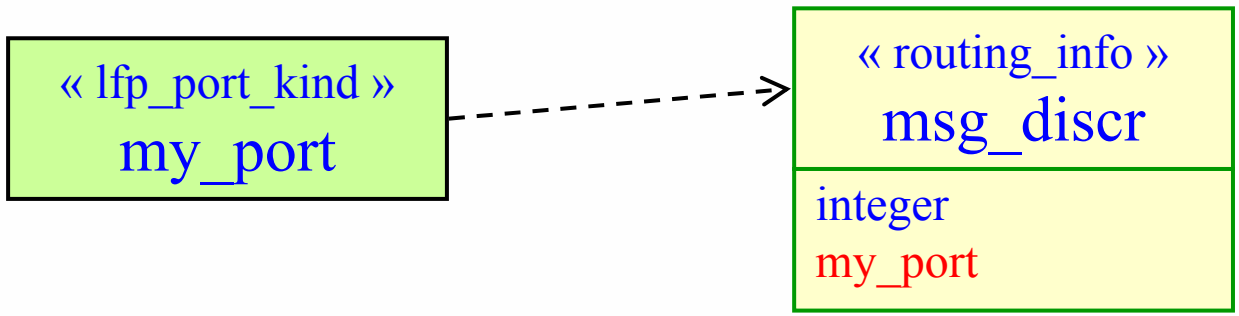


Composant d'un type port inclus dans le discriminant associé

- Un type de discriminant peut inclure un composant de type port, y compris un composant d'un type port auquel il est associé.
- Exemple : le type port défini en LfP comme suit :

```
type my_port is port (integer,
  my_port);
```

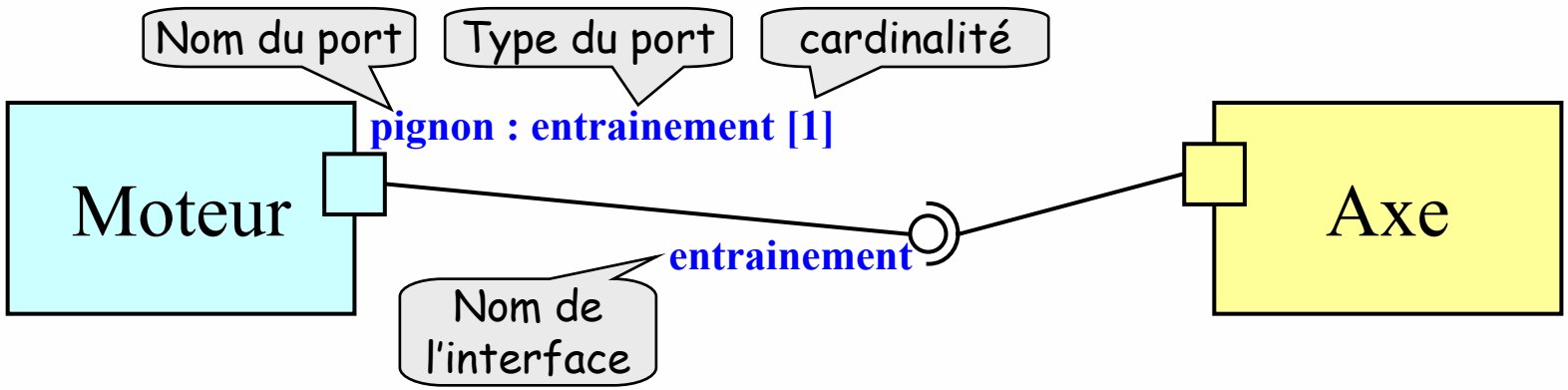
est représenté par la classe UML ci-dessous :



- Cette possibilité permet à un composant d'envoyer à un autre composant un message incluant la référence d'un binder. L'autre composant pourra ainsi affecter cette référence à l'un de ses ports, afin de pouvoir utiliser ce binder pour communiquer.

Typage des ports : UML ≠ LfP

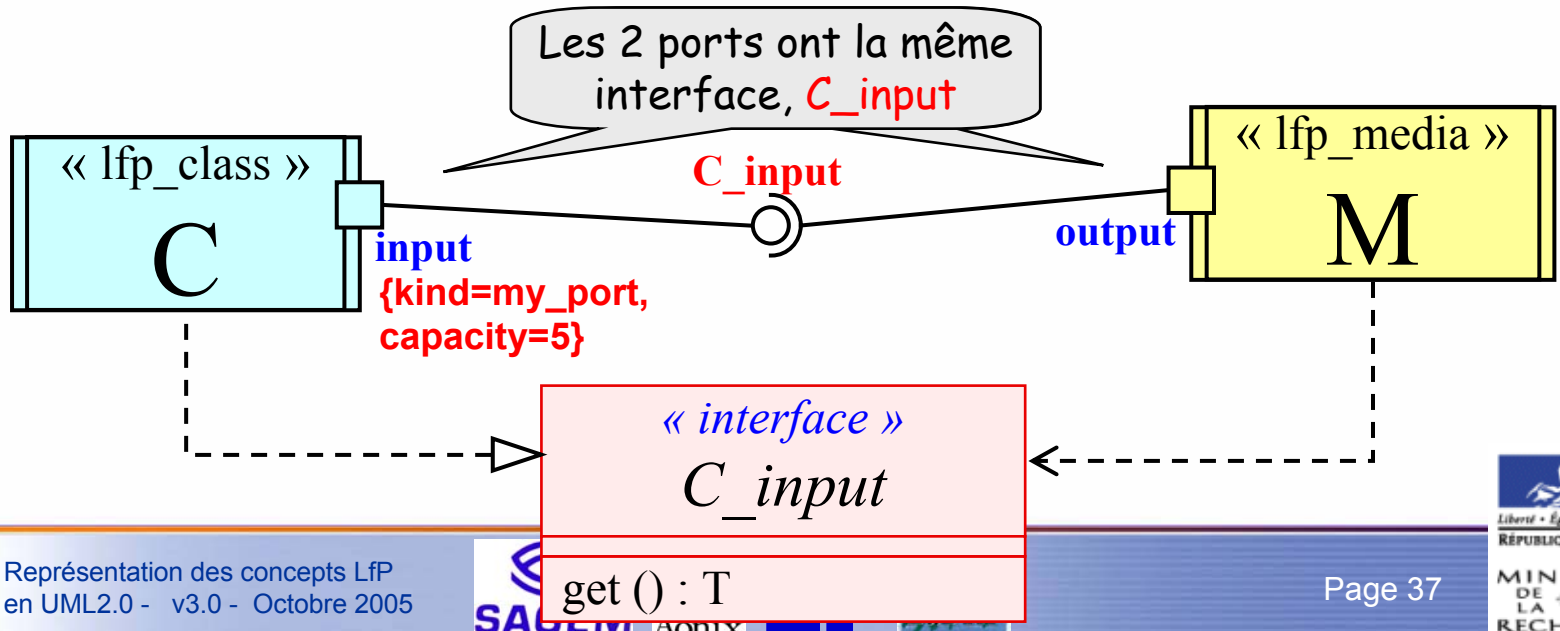
- **En UML**, le nom d'un port sur le diagramme peut être suivi de son type, et de sa cardinalité :



- En UML, le type d'un port représente l'interface associée au port (et porte le même nom), alors qu'en LfP, un type port n'est pas défini en termes d'interface, mais en tant que référence à un type de binder. Le type d'un binder est lui-même caractérisé en particulier par un format de discriminants.

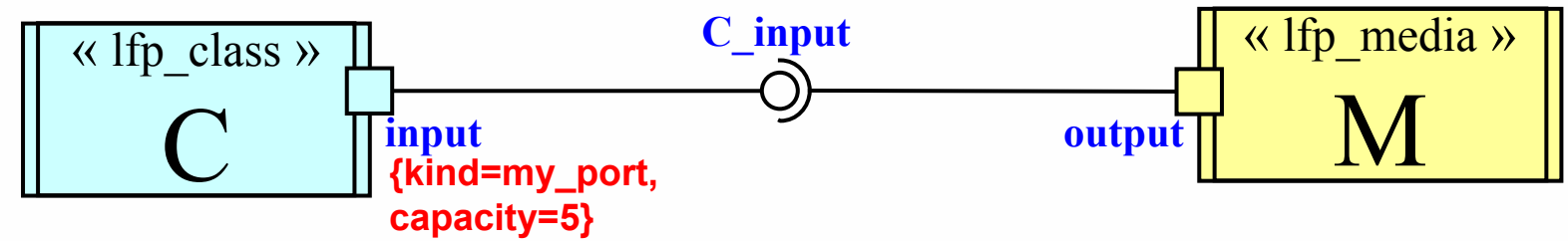
Typage des ports : UML ≠ LfP

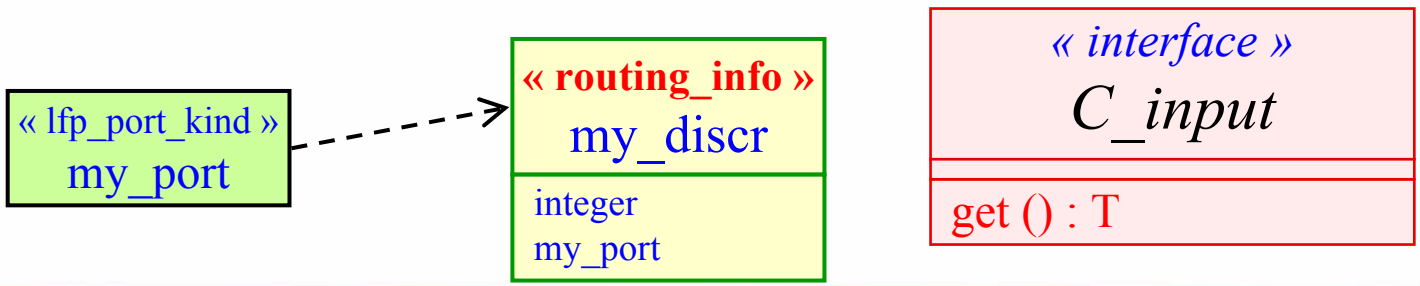
- Dans ce profil, pour un symbole port figurant dans un diagramme de composants, on n'indiquera aucun nom de type après le nom du port. Le type du port au sens LfP sera indiqué par une étiquette $\{\text{kind} = nom\}$.
  - On pourra omettre cette indication pour le port conjugué, qui est forcément de même type, car il référence le même binder
- Le nom de l'interface associée, qui représente le type du port au sens UML, sera indiqué près du symbole d'interface (boule, ou socket, ou connecteur d'assemblage) relié au port.
  - les détails de cette interface seront décrits séparément, dans une représentation par symbole *classifieur* (rectangle).



Typage des ports : UML ≠ LfP

- L'information relative au type de discriminant de ces ports conjugués (c'est-à-dire l'information spécifiant la composition des discriminants des messages échangés par ces 2 ports) est fournie par la classe stéréotypée « routing_info » qui est liée au type port par une relation de dépendance.
- La liste des messages comportant un discriminant de ce type est donnée par la liste des opérations figurant dans la forme détaillée de l'interface des ports conjugués.

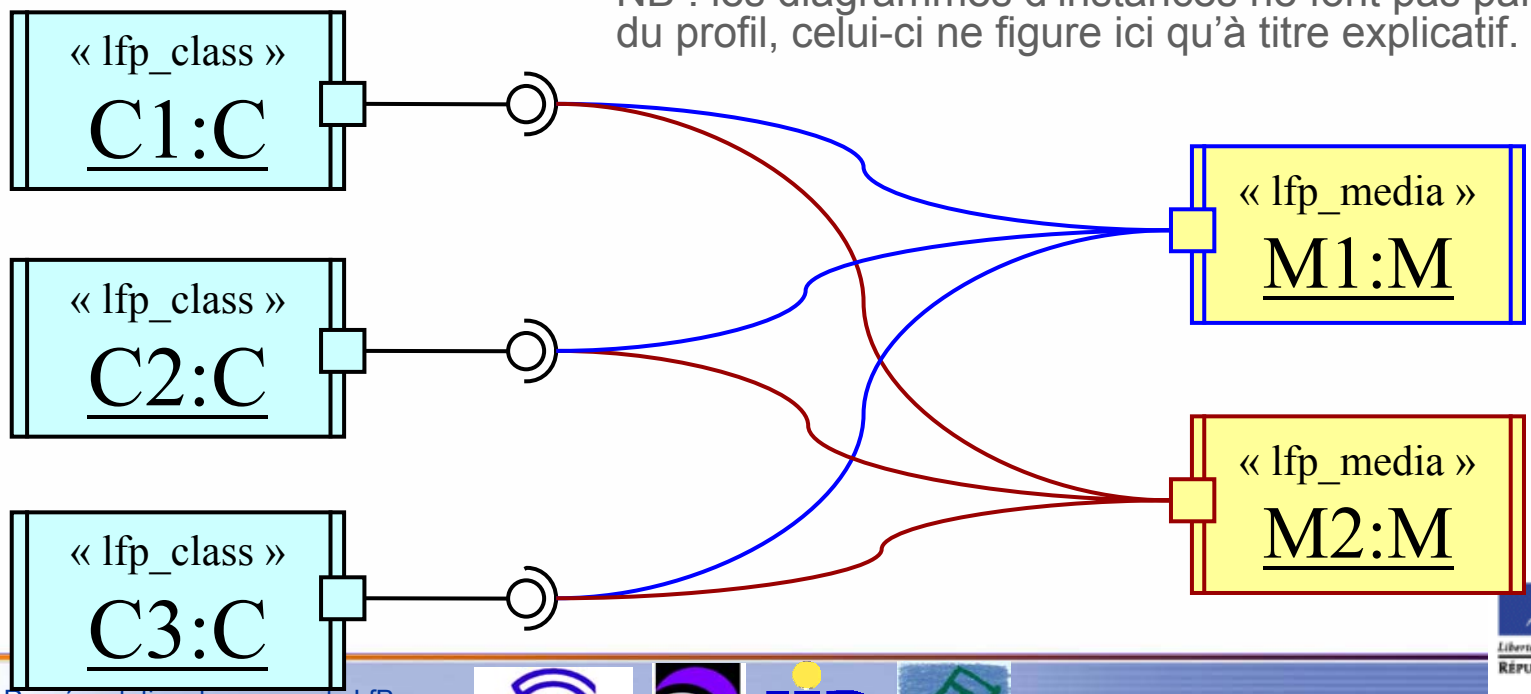




Déploiement d'une liaison Classe-Média avec binder de multiplicité 1

- Dans le diagramme de composants : un port de multiplicité 1 signifie qu'il existe une instance du binder pour chaque instance de la classe.
- Dans ce diagramme d'instances, on convient de « mettre en facteur » le symbole *socket* entre toutes les instances du média
- On ne fait pas de même pour le symbole boule entre les instances de la classe (même si en UML cela n'aurait pas d'implication sémantique)

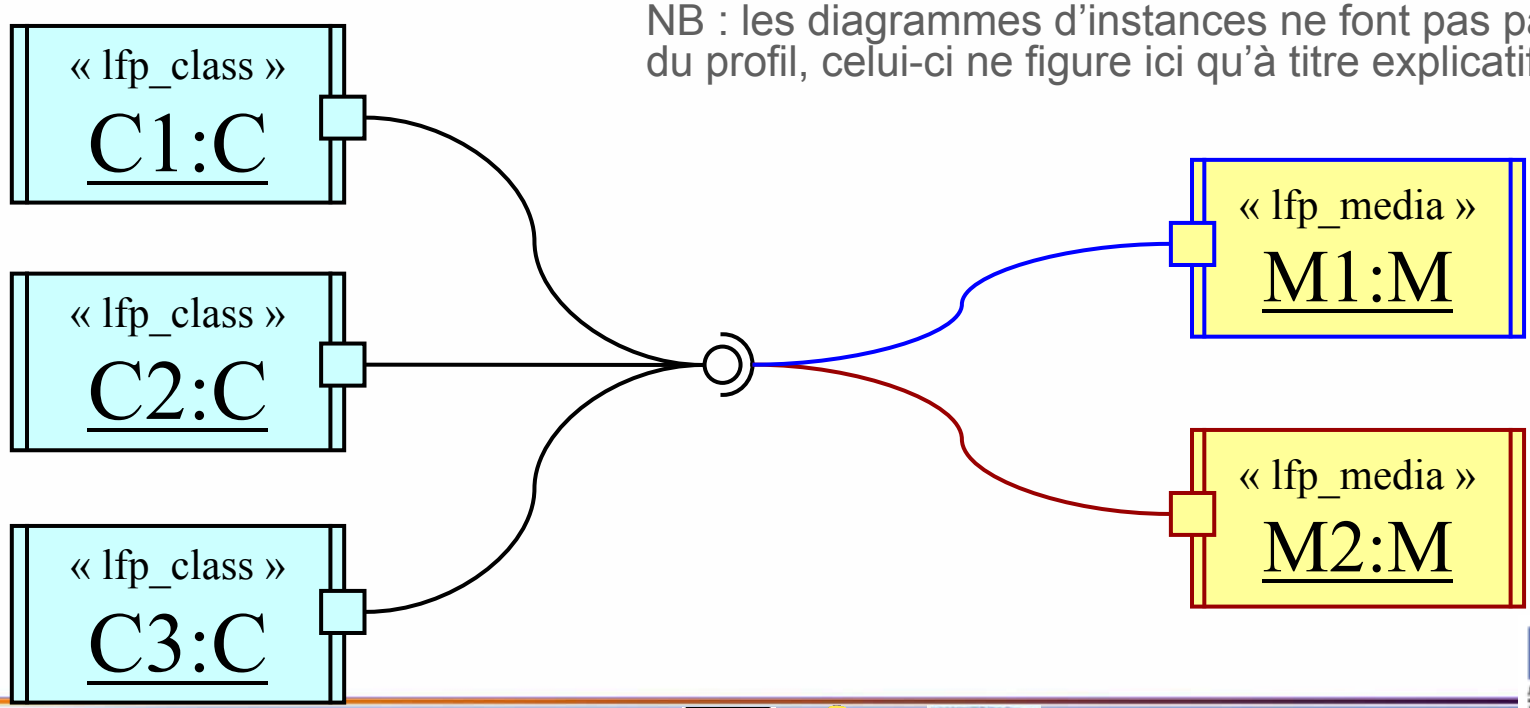
NB : les diagrammes d'instances ne font pas partie du profil, celui-ci ne figure ici qu'à titre explicatif.



Déploiement d'une liaison Classe-Média avec binder de multiplicité all

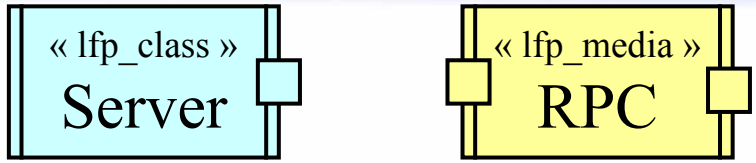
- un port de multiplicité all signifie qu'il n'existe qu'une seule instance de binder, partagée entre toutes les instances de la classe C
- Dans ce diagramme d'instances, on convient de « mettre en facteur » le symbole *socket* entre toutes les instances du média, et aussi le symbole boule entre les instances de la classe (NB: en UML cela n'a pas d'implication sémantique)

NB : les diagrammes d'instances ne font pas partie du profil, celui-ci ne figure ici qu'à titre explicatif.



Représentation d'instances statiques

composants



- les instances statiques de composants sont celles déclarées dans le diagramme d'architecture, sous la forme
  `x : Server with (...);` -- entre parenthèses : valeurs initiales d'attributs
- Ne pas confondre avec les déclarations de *références* vers composants, comme : `s_ref : Server;` -- décl. ne créant aucune instance de composant

instances

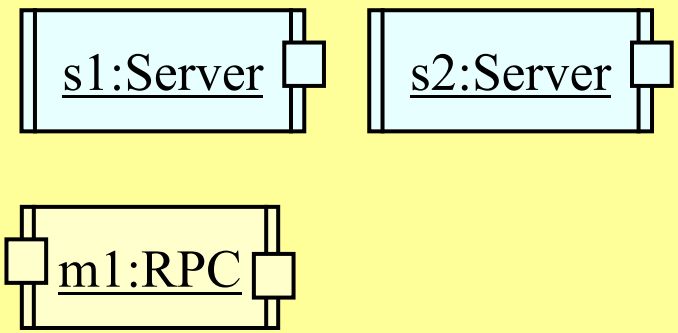
diagramme d'architecture LfP

s1 : Server with () ;

s2 : Server with () ;

m1 : RPC with () ;

représentation UML



Représentation de valeurs d'attributs d'instances statiques

composant

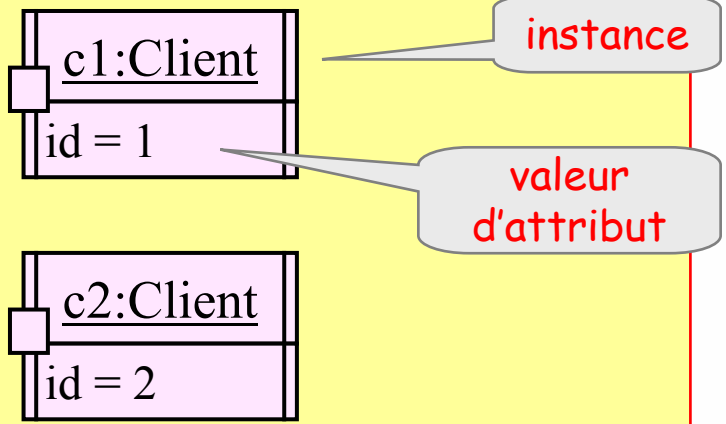
« lfp_class »
Client

- Les valeurs initiales d'attributs d'instances statiques de composants sont représentées par des valeurs d'attributs UML, dans un compartiment du symbole d'objet.

diagramme d'architecture LfP

```
c1 : client with (id => 1) ;
c2 : client with (id => 2) ;
```

représentation UML

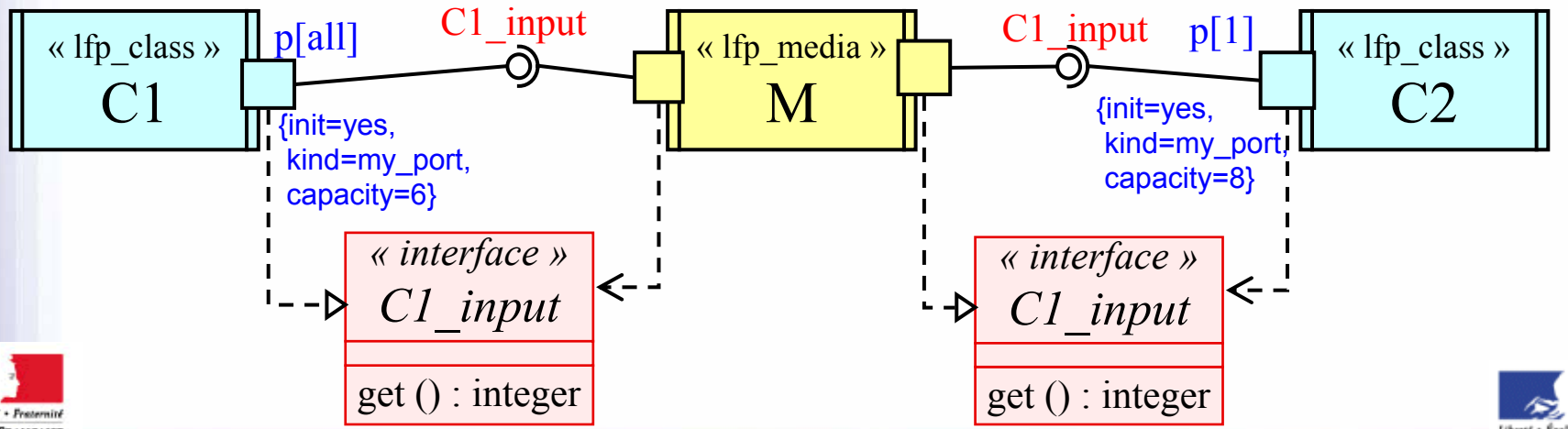


Variantes (en cas de limitation des outils)

- L'interface détaillée (symbole rectangle des *classifieurs*) peut ne pas apparaître dans le diagramme de composants : on peut la fournir dans un diagramme de classes.
- Idem pour les classes « lfp_port_kind » et « routing_info »
- Un composant LfP peut apparaître à la fois comme composant UML dans le diagramme de composants, et comme classe UML dans un diagramme de classes ou dans un diagramme de structure composée.
  - Ces vues additionnelles permettent le cas échéant de compléter la description du composant, en y apportant des attributs, opérations, relations, structure interne, …
- Une instance de composant LfP peut être aussi décrite dans un diagramme de classes, sous forme d'instance d'une telle classe UML (classe représentant une vue d'un composant en tant que classe).
  - cette représentation permet de fournir des valeurs d'attributs de l'instance.

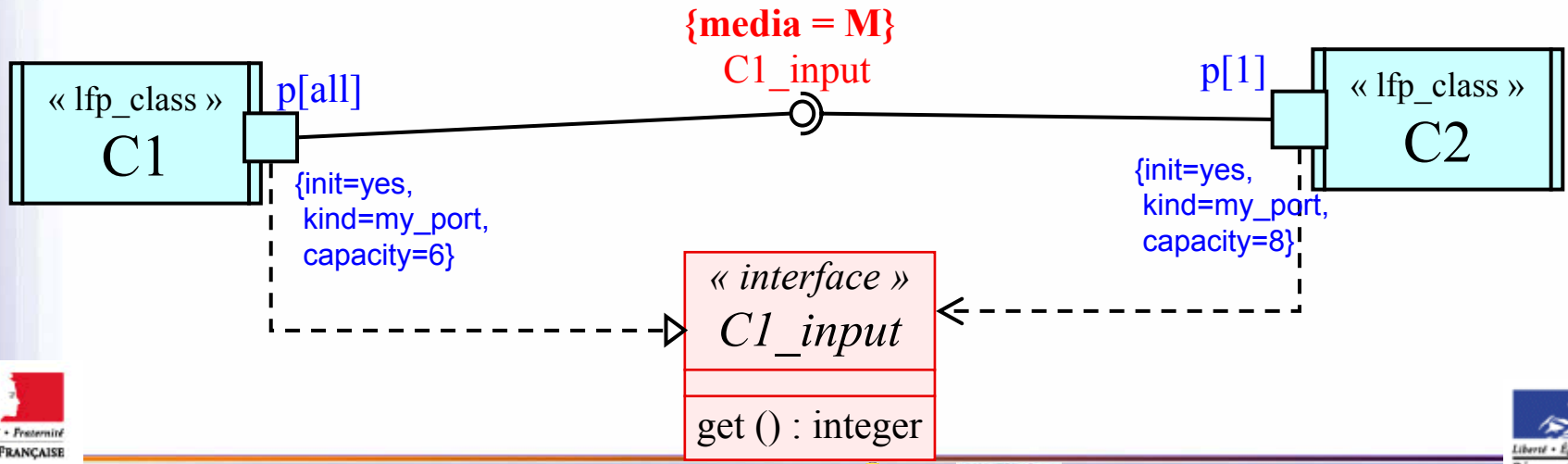
Communication entre 2 classes : Possibilité de simplification du diagramme (1)

- Lorsque le schéma de communication entre deux classes est simple, on peut simplifier le diagramme de composants en faisant abstraction du média qui assure la transmission des messages entre les deux classes.
- Conditions dans lesquelles cela peut s'appliquer :
  - Les 2 classes communiquent par un média et un seul;
  - Elles utilisent 2 binders (distincts) de même type, un binder chacune, pour échanger des messages avec le média;
  - Le média possède 2 ports de même type, désignant respectivement ces 2 binders.
  - Le média n'assure pas d'autre communication que celle entre ces 2 classes;
- Dans ce cas les 2 interfaces entre le média et chacune des 2 classes sont identiques. Exemple :



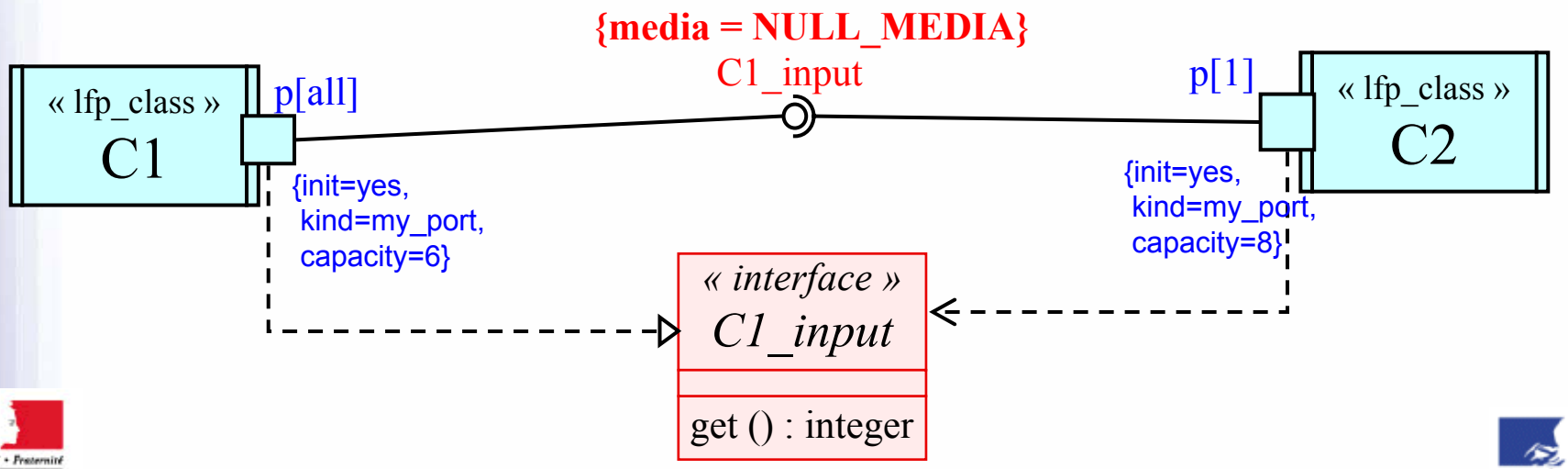
Communication entre 2 classes : Possibilité de simplification du diagramme (2)

- On peut ne pas représenter le média M, ni les 2 connecteurs d'assemblage qui lui sont connectés,
- et les remplacer par un connecteur d'assemblage unique, muni d'une étiquette {media = M}.
  - On décrit le média M isolément par ailleurs;
  - L'étiquette {media = M} indique la présence implicite d'un média M et de 2 binders, attachés aux 2 ports des 2 classes. Ces 2 ports ne désignent pas un même binder. Ils ont donc chacun une étiquette {init=yes}, une multiplicité, et des étiquettes représentant les attributs du binder qui leur est attaché.
- L' interface entre les 2 classes est la même que celles entre le média et chacune d'elles.



Communication entre 2 classes : Possibilité de simplification du diagramme (3)

- Le média assure la transmission des messages selon un certain protocole,
  - si ce protocole se réduit à une simple copie de message d'un binder vers l'autre, on convient de qualifier le média comme étant de type NULL_MEDIA.
  - La représentation simplifiée reste la même, mais l'étiquette devient {media = NULL_MEDIA}

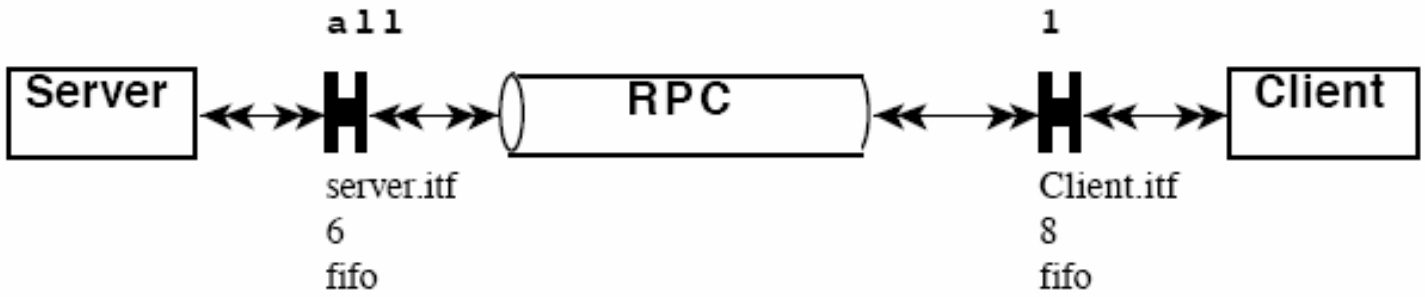


Sommaire

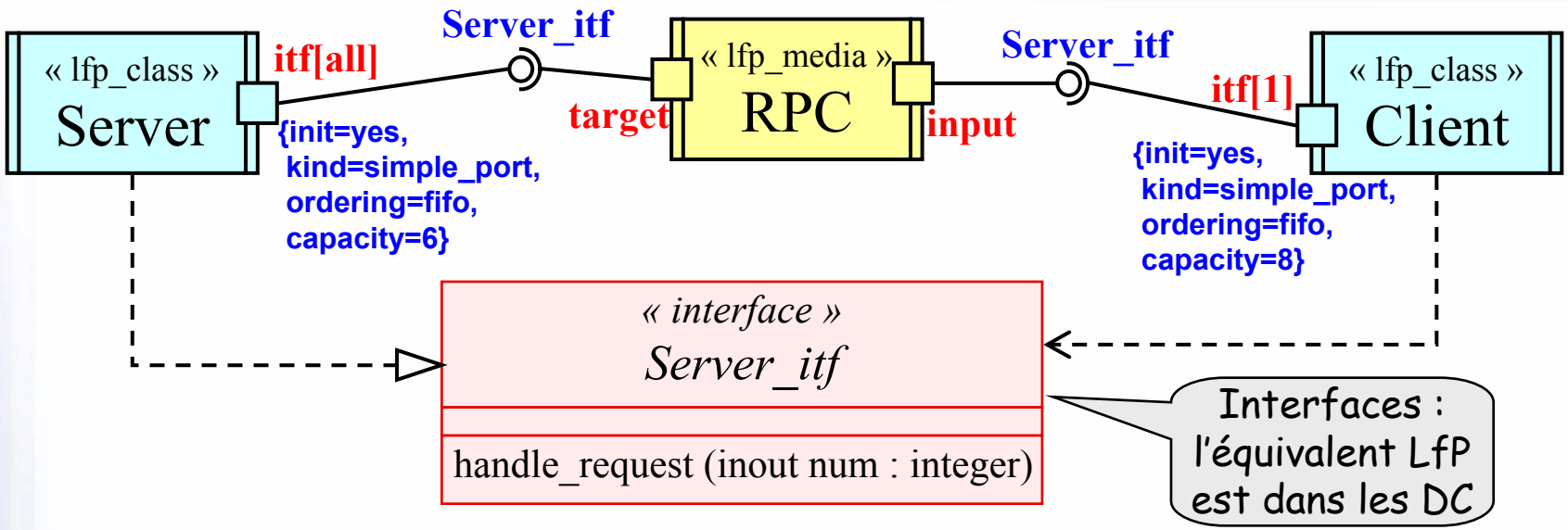
Exemple : l'application Client-Serveur Diagramme d'architecture, en LfP

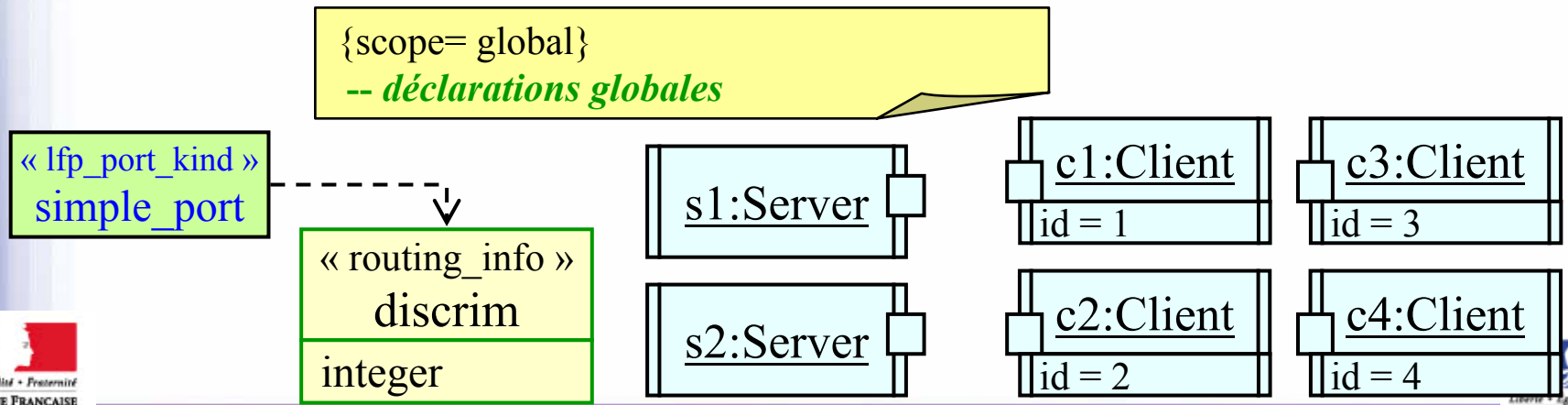
CLIENT_SERVER

```
type simple_port is port (integer);

s1 : server with () ;
s2 : server with () ;

c1 : client with(id => 1) ;
c2 : client with (id => 2) ;
c3 : client with (id => 3) ;
c4 : client with(id => 4) ;
```



Exemple : l'application Client-Serveur
Diagramme d'architecture, équivalent UML



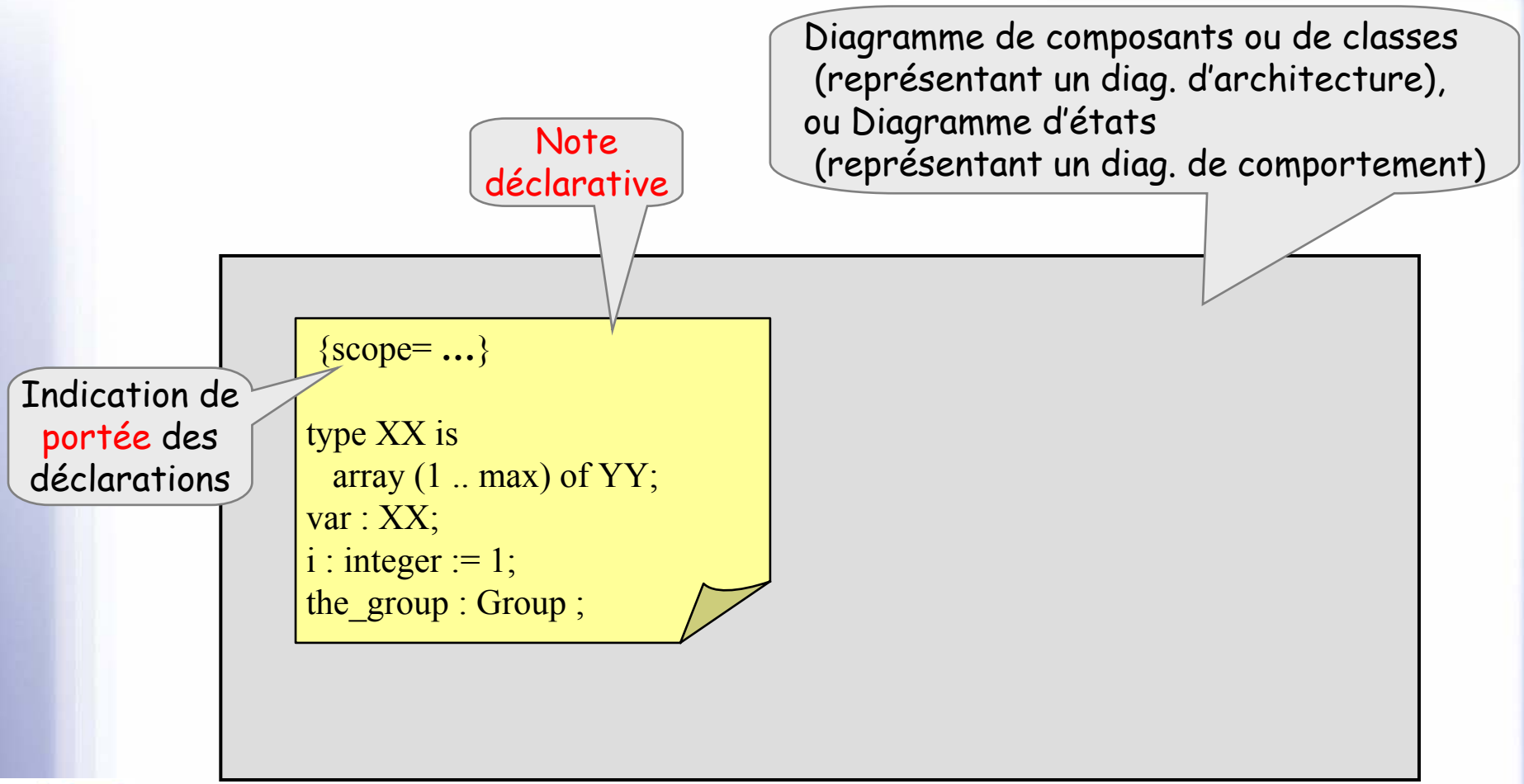


Sommaire

Représentation des parties déclaratives

- La partie déclarative du diagramme d'architecture peut comporter, outre les éléments vus antérieurement, des déclarations de types de données, de variables, de constantes.
- De même chaque diagramme de comportement peut comporter une partie déclarative, incluant aussi de telles déclarations.
- La représentation UML de la plupart de ces déclarations de types de données, variables, constantes, sera textuelle et non graphique. Une *note* UML sera utilisée pour chaque partie déclarative.
  - Cela permet d'alléger la représentation de ces « petites » entités, qui peuvent être nombreuses.
  - Autre avantage : chaque DC étant représenté par un diagramme d'états UML, sa partie déclarative, composée de déclarations à portée locale, pourra être représentée par une note textuelle incluse dans le diagramme d'états, donc visualisable en même temps que le diagramme : cela améliore la lisibilité du modèle et la facilité de compréhension.

Exemple de représentation de partie déclarative



Indication de portée d'une note déclarative

- La note déclarative débute par l'étiquette

  { *scope* = *indication_de_portée* }

  indication_de_portée : *global*
  | nom de composant
  | nom d'état composé
  | spécification de méthode ou de trigger

- L'étiquette *indication_de_portée* peut avoir pour valeur :
  - *global* : pour les déclarations globales. La note portant ces déclarations se trouve dans le diagramme de composants représentant le DA ;
  - Nom d'un composant : pour les déclarations locales à ce composant. La note portant ces déclarations se trouve dans le diagramme d'états représentant le DC principal du composant ;
  - Nom d'un état composé : pour les déclarations locales au sous-diagramme correspondant à cet état. La note portant ces déclarations se trouve dans le diagramme d'états détaillant l'intérieur de l'état ;
  - spécification de méthode ou de trigger : pour les déclarations locales à une méthode ou un trigger. La note portant ces déclarations se trouve dans le diagramme d'états de la sous-machine décrivant le corps de la méthode ou du trigger.

Cas des parties déclaratives représentées en partie graphiquement

- Le choix a été fait de représenter graphiquement certaines déclarations qui s'y prêtent (exemple: décl. d'un type port, ou d'une instance statique de composant). Ces déclarations ne peuvent donc pas être intégrées dans les notes textuelles.
- Si parmi les déclarations ayant une même portée, certaines sont représentées graphiquement, d'autres par du texte, on regroupe dans un même diagramme de classes ces déclarations. Les déclarations graphiques sont représentées par des symboles UML de classes et d'objets, les déclarations textuelles figurent dans une note unique commençant par une étiquette

  { *scope* = *indication_de_portée* }

  - Pourquoi diag. de classes plutôt que de composants ? => limitation Ameos (classes/objets représentables, mais pas leurs attributs)
- S'il n'y a que des déclarations graphiques pour une portée donnée, on insére néanmoins dans le diagramme de classes correspondant une note, ne contenant que l'étiquette d'indication de portée.
- S'il n'y a que des déclarations textuelles pour une portée donnée, la note déclarative pour cette portée sera insérée
  - dans le diagramme de composants, si la note a une portée globale;
  - sinon, dans le diagramme d'états correspondant à sa portée.

Représentation des déclarations de variables et constantes

- La syntaxe des déclarations de constantes est identique à celle du langage LfP textuel :

const *IDENTIFIER* : *IDENTIFIER* := *expression* ;

Nom de la constante — Type — valeur

- La syntaxe des déclarations de variables est similaire, sans le mot *const* :

IDENTIFIER [, *IDENTIFIER*] : *IDENTIFIER* [:= *expression*] ;

Noms des variables — Type — valeur initiale

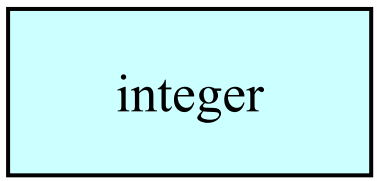
  - on peut déclarer et initialiser plusieurs variables de même type par une seule déclaration.

Représentation des déclarations de types

- Types non représentés dans les modèles :
  - les types prédéfinis du langage : integer, boolean, message, semaphore ; ces types font partie du profil.
- Types représentés de manière graphique :
  - les types ports (et les types des discriminants) ;
  - Les types opaques
- Types représentés de manière textuelle :
  - les types tableaux ;
  - les types énumératifs ;
  - les types définis par un intervalle de valeurs ;
  - les types structure (record) ;
  - Les références vers composants ;

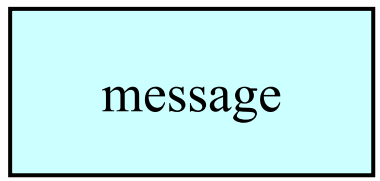
Types non représentés dans les modèles : integer

- Le type integer est prédéfini, il fait (implicitement ou explicitement) partie du profil.
- On pourrait *éventuellement* le représenter dans le profil par une classe du même nom,
  - sans attributs,
  - Inutile de modéliser les opérations.



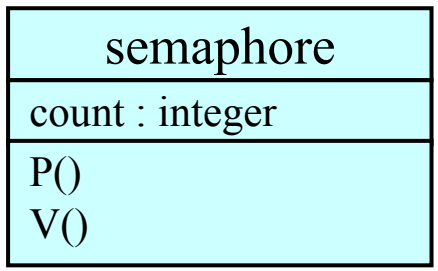
Types non représentés dans les modèles : message

- Une valeur message représente une vue encapsulée, privée, des données transportées dans un message.
- C'est la seule vue des messages que les médias peuvent manipuler : ils n'ont pas accès au contenu interne des messages.
- Le type message est prédéfini, il fait (implicitement ou explicitement) partie du profil.
- On pourrait *éventuellement* le représenter dans le profil par une classe du même nom.
  - sans attributs,
  - Opérations (lecture, envoi) non modélisées.
- Type utilisable par les médias seulement.



Types non représentés dans les modèles : semaphore

- Le type semaphore est prédéfini, il fait (implicitement ou explicitement) partie du profil.
- On pourrait *éventuellement* le représenter dans le profil par une classe du même nom.
  - Attribut : `count : integer`
  - Opérations : `P()` et `V()`.

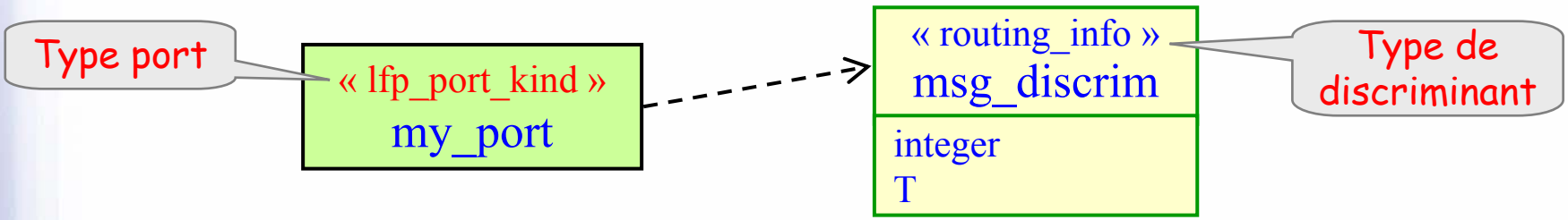


Types représentés de manière graphique : types ports, types discriminants (rappels)

- le type port défini en LfP comme suit :

```
type my_port is port (integer, T);
```

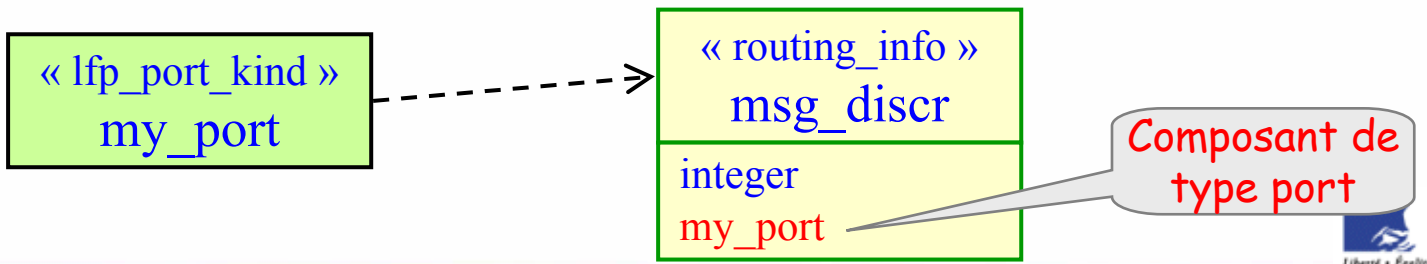
est associé à un type discriminant ayant un composant de type *integer* et un autre de type *T*. Il est représenté par la classe UML ci-dessous :



- Un type de discriminant peut inclure un composant de type port, y compris un composant d'un type port auquel il est associé.
- Exemple : le type port défini en LfP comme suit :

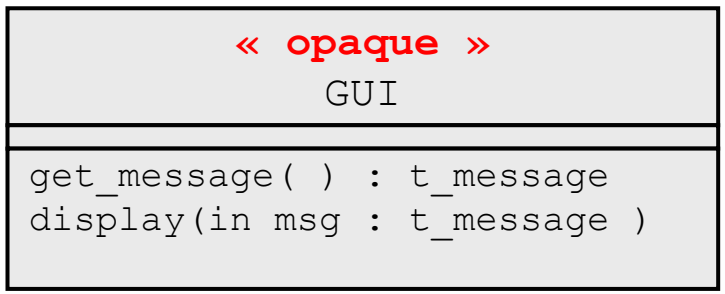
```
type my_port is port (integer, my_port);
```

est représenté par la classe UML ci-dessous :



Types représentés de manière graphique : types opaques

- Un type opaque est représenté par une classe, munie du stéréotype « opaque ».



Types représentés de manière textuelle : types tableaux

- La déclaration d'un type tableau est identique à celle du langage LfP textuel :

  type IDENTIFIER is array (range_list) of IDENTIFIER ;

  Type des éléments

  avec

  range_list : *range* [, *range*]*

  range : *expression* .. *expression*

- Exemple :

```
type USER_TABLE is array (1 .. max_users) of USER;
```

Types représentés de manière textuelle : types énumératifs

- La déclaration d'un type énumératif est identique à celle du langage LfP textuel :

  type IDENTIFIER is enum (identifier_list) ;

  Ou bien, pour un type circulaire :

  type IDENTIFIER is circular enum (identifier_list) ;

  avec

  range : *expression* .. *expression*

- Exemple :

```
type request_kind is enum (join, send, quit, get_list);
```

Types représentés de manière textuelle : types définis par un intervalle de valeurs

- La déclaration d'un type défini par un intervalle de valeurs est identique à celle du langage LfP textuel :

  type IDENTIFIER is range (range) of IDENTIFIER ;

  Type entier, intervalle, ou énumératif

  Ou bien, pour un type circulaire :

  type IDENTIFIER is circular range (range) of IDENTIFIER ;

  avec

  identifier_list : *IDENTIFIER* [, *IDENTIFIER*]*

- Exemple :

```
type sub_kind is range (join..quit) of request_kind;
```

Types représentés de manière textuelle : types structure (record)

- La déclaration d'un type structure (record) est identique à celle du langage LfP textuel :

  type IDENTIFIER is record [*champs*]+ *end* ;

  avec

  champs : *IDENTIFIER* [, *IDENTIFIER*] : *IDENTIFIER* ;

  Noms des champs

  Type des champs

- Exemple :

```
type t_group is record
    name : t_message ;
    reference : group ;
end ;
```

Types représentés de manière textuelle : références vers composants

- Comme en LfP, les types « références vers composants » sont anonymes, ils sont utilisés de manière implicite dans les déclarations de variables "référence" vers composant.
- La déclaration d'une variable "référence" vers composant a la même forme que celle d'une variable ordinaire, sauf que le type est le nom d'une classe ou d'un média :

IDENTIFIER [, *IDENTIFIER*] : *IDENTIFIER* [:= *expression*] ;

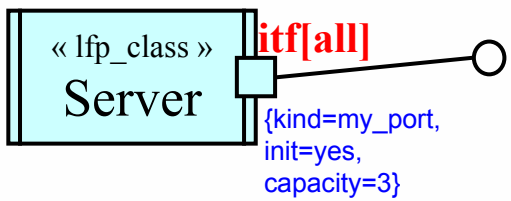
Noms des variables

Type = nom de composant

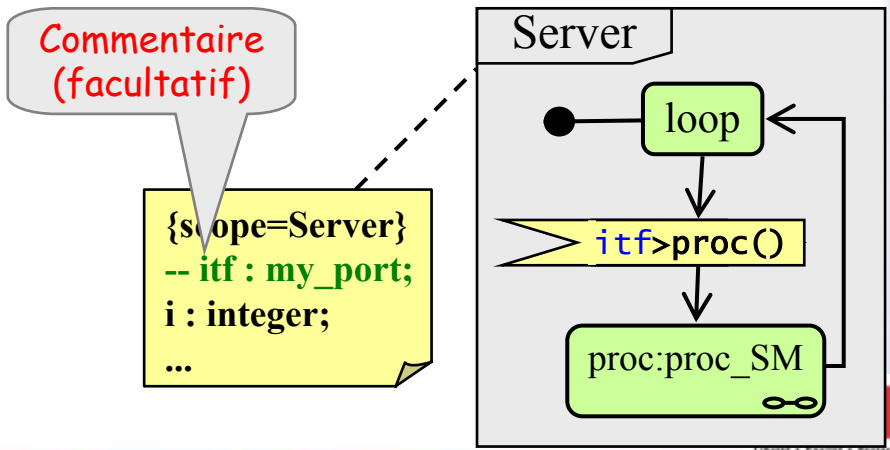
- NB : Une variable ainsi déclarée n'est pas une instance de composant, c'est une référence vers une instance.
- Cette déclaration, contrairement à celle d'une variable ordinaire, ne crée aucune instance de composant.
- Rappel: Les déclarations d'instances de composants statiques sont graphiques (symboles d'objets).

Déclaration des attributs ports d'un composant

- Les attributs ports d'un composant sont représentés graphiquement sur le diagramme de composants représentant le DA.
  - Alors qu'en LfP, ils sont déclarés de façon textuelle dans le DC du composant.
- Exemple, extrait du diag. de composants



- Pour la lisibilité du diagramme d'états représentant le DC d'un composant, on peut, si on le désire, ajouter une *pseudo-déclaration textuelle* des attributs ports, en commentaire, dans la note déclarative associée.
- Exemple, extrait du diag. d'états



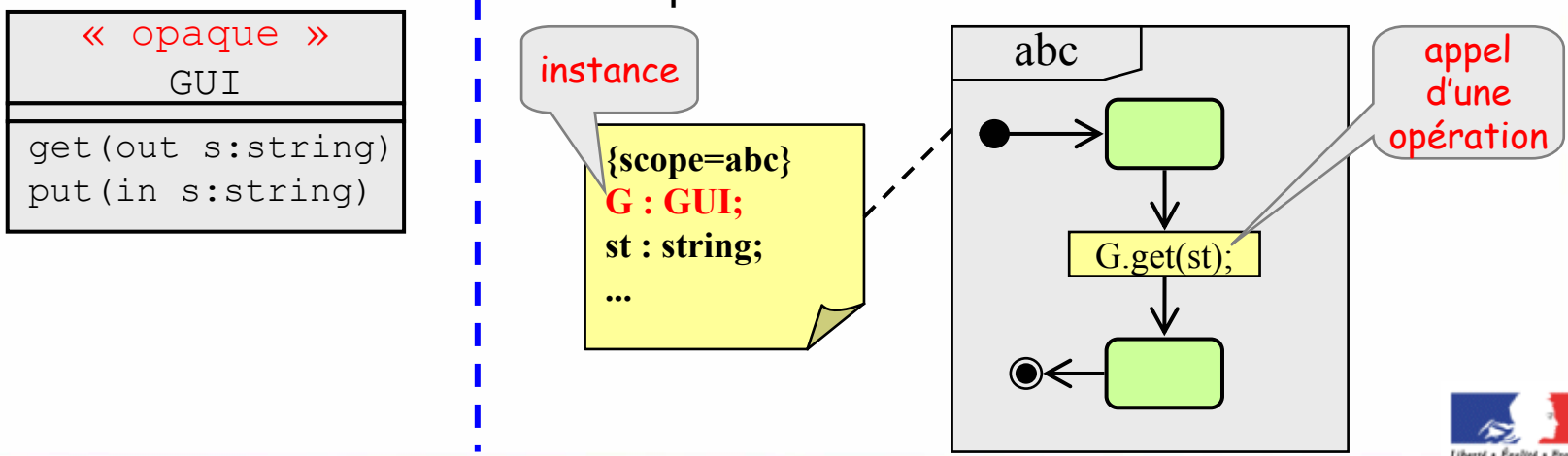
Déclaration d'instances de types opaques

- Les types opaques sont représentés graphiquement, mais leurs instances sont représentées par des déclarations de variables (textuelles)
- La déclaration d'une instance de type opaque est située dans la note déclarative correspondant à sa portée.
- L'appel d'une opération d'un type opaque s'écrit comme tout appel d'opération en UML :

IDENTIFIER . *IDENTIFIER* ([*expression* [, *expression*]*])

instance — opération — Param. effectifs

exemple :



Sommaire

Diagramme de Comportement LfP

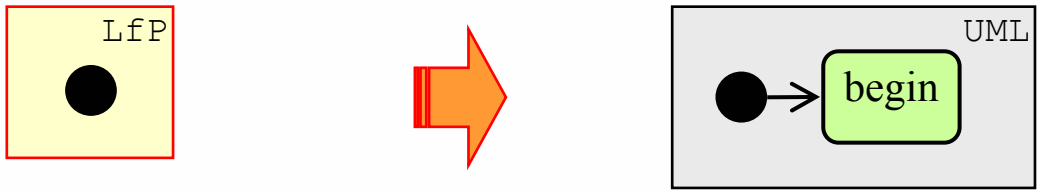
- Les diagrammes de comportement LfP seront représentés par des diagrammes d'états UML2.0.

Diagramme de Comportement LfP : Entités à représenter

- État (initial, final, d'attente d'activation de méthode)
- Transition (terminale, hiérarchique, de communication, d'instanciation, opération sur sémaphore)
  - Priorité d'une transition
- Garde
- Sous-diagramme
- Trigger
- Méthode
  - fonction, procédure synchrone, procédure asynchrone
- Attributs d'un composant
- Déclaration de type, variable, constante

Représentation des états

- État initial LfP : représenté par un pseudo-état initial UML, + un état UML
  - Car un état initial LfP peut avoir une transition entrante, ce qui est interdit pour un pseudo-état initial UML.



- État final LfP : représenté par un pseudo-état final UML.

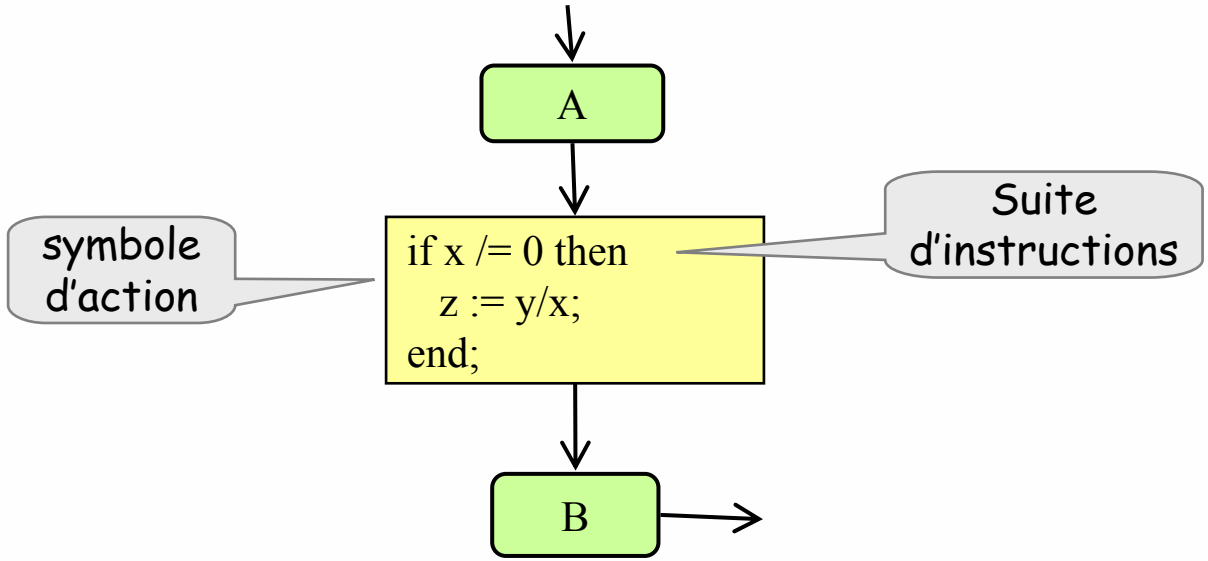
- État ordinaire LfP : représenté par un état ordinaire UML,
  - sans actions en entrée ni en sortie, sans activité, sans sous-états ni sous-régions.

Représentation des transitions

- La représentation UML doit prendre en compte :
  - Les transitions simples
  - Les transitions d'instanciation dynamique de composants
  - Les transitions hiérarchiques
  - Les transitions de communication
  - Les priorités des transitions

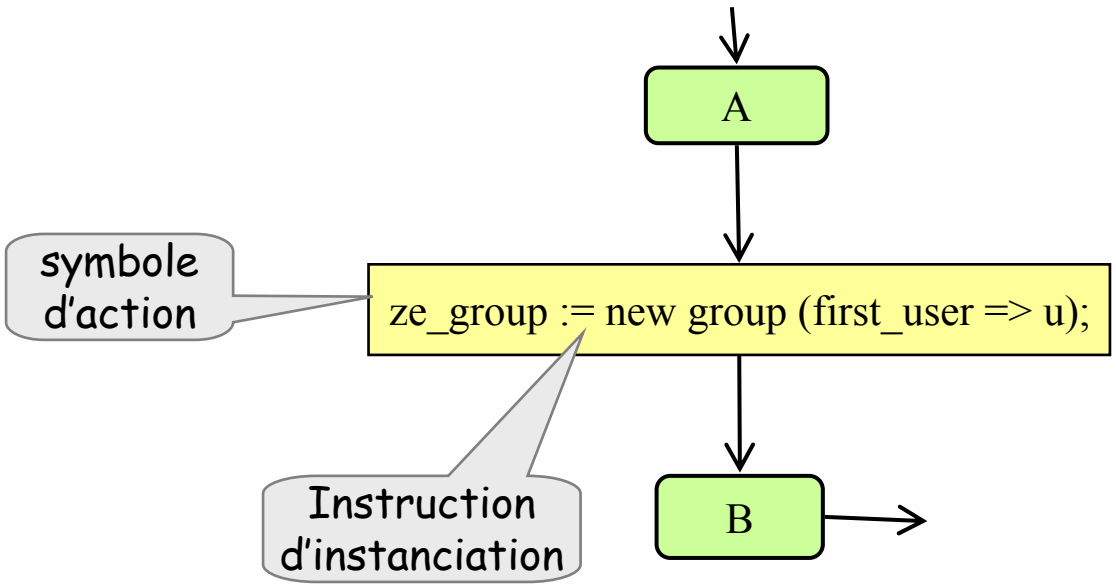
Représentation des transitions simples

- Les transitions simples sont représentées par un symbole d'action (nouveauté UML2.0) situé sur le chemin d'une transition UML.
- Ce symbole porte un label ayant la syntaxe d'une suite d'instructions en LfP textuel.
  Exemple :



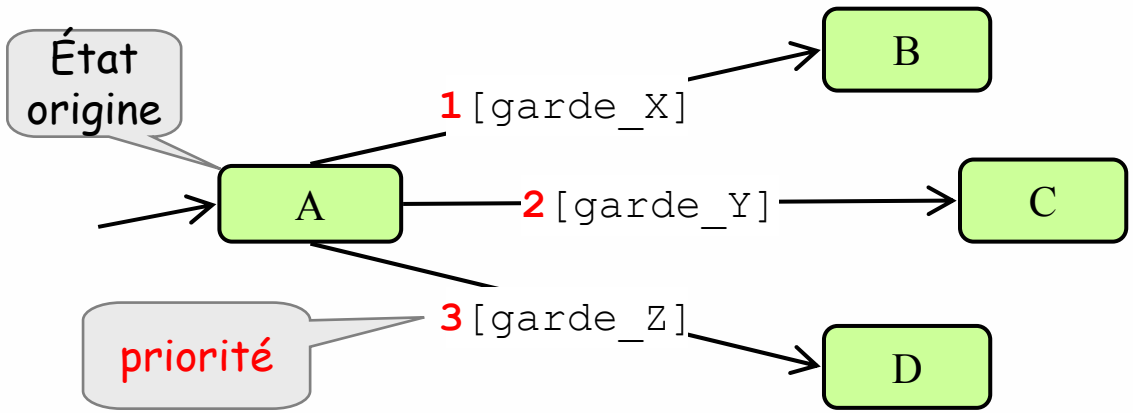
Représentation des transitions d'instanciation dynamique de composants

- Les transitions d'instanciation dynamique de composants sont représentées par un symbole d'action situé sur le chemin d'une transition UML.
- Ce symbole porte un label ayant la syntaxe d'une instruction d'instanciation en LfP textuel.
  Exemple :



Représentation de la priorité des transitions

- Rappels:
  - en LfP, une « transition » (changement d'état) est franchissable dès que sa garde est vraie, sauf si elle attend un évènement de réception de message (pas d'autre sorte d'évènement).
  - Plusieurs transitions ayant un même état origine peuvent être franchissables du fait de gardes vraies : en LfP cela ne constitue pas une erreur. Les priorités peuvent être utilisées dans ce cas pour déterminer la transition choisie.
- La priorité d'une transition est représentée par un nombre placé sur l'arc correspondant sortant de l'état origine. Ce nombre prend l'emplacement syntaxique du nom de l'évènement déclencheur (trigger) en UML : s'il y a une garde, il est placé avant elle.



Représentation des transitions de communication

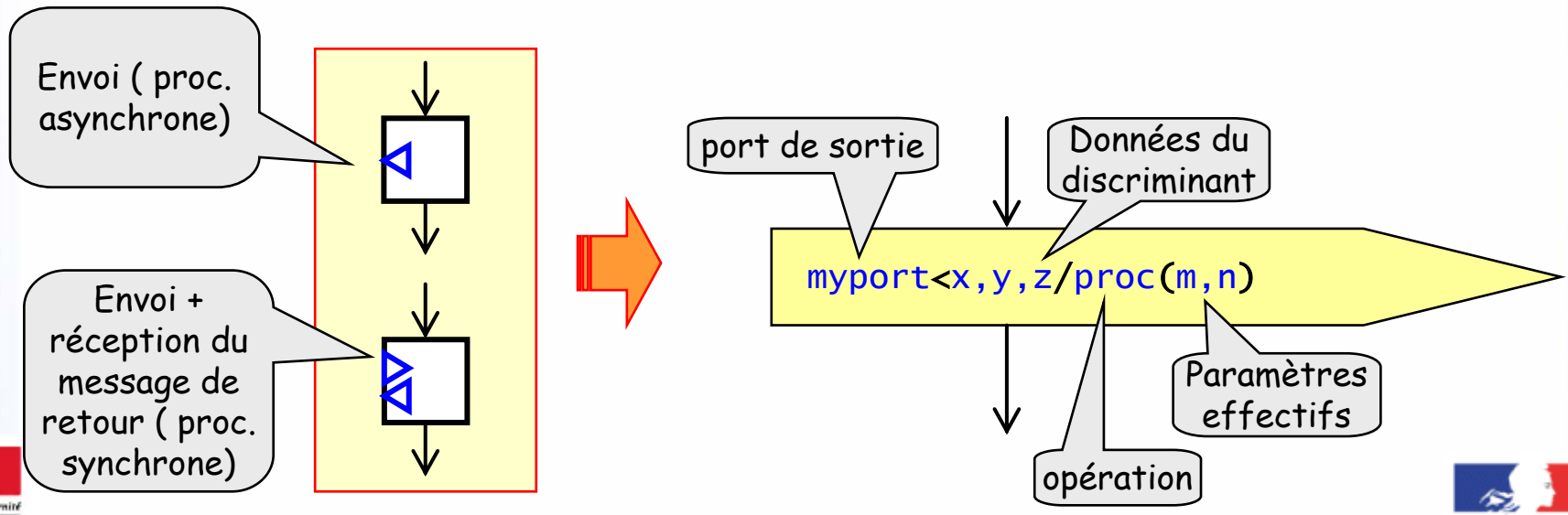
- Transitions d'envoi :
  - appel de procédure asynchrone
  - appel de procédure synchrone
  - appel de fonction
  - envoi du résultat d'un appel (*return*)
  - envoi de message
    - par une classe,
    - par un média
- Transitions de réception :
  - réception d'appel de méthode
  - réception de message
    - par une classe,
    - par un média

Représentation d'un appel de procédure synchrone ou asynchrone

- L'appel est représenté dans les diagrammes d'états par un symbole d'émission de signal, situé sur un arc de transition. Le label de ce symbole doit avoir la syntaxe suivante :

```
port_de_sortie < discriminant/nom_d_opération(paramètres)
```

- Le caractère synchrone ou asynchrone de l'appel découle de la déclaration de l'opération appelée (voir diag. de composants)

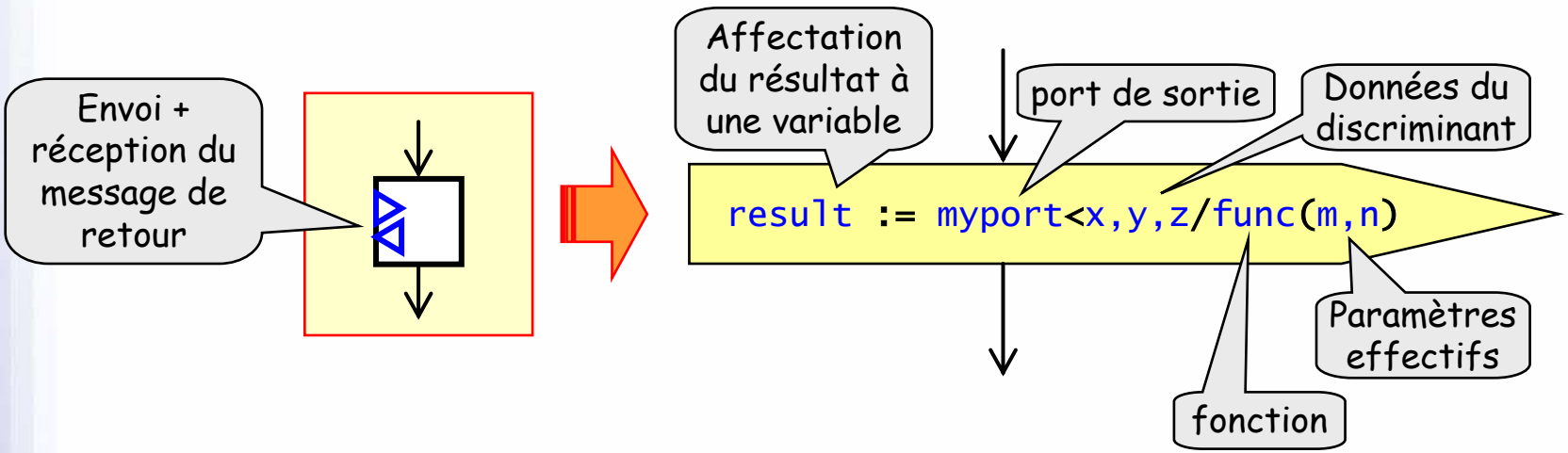


Représentation d'un appel de fonction

- L'appel est représenté dans les diagrammes d'états par un symbole d'émission de signal, situé sur un arc de transition. Le label de ce symbole doit avoir la syntaxe suivante :

```
variable := port_de_sortie < discriminant/nom_d_opération(paramètres)
```

- L'appel est toujours synchrone

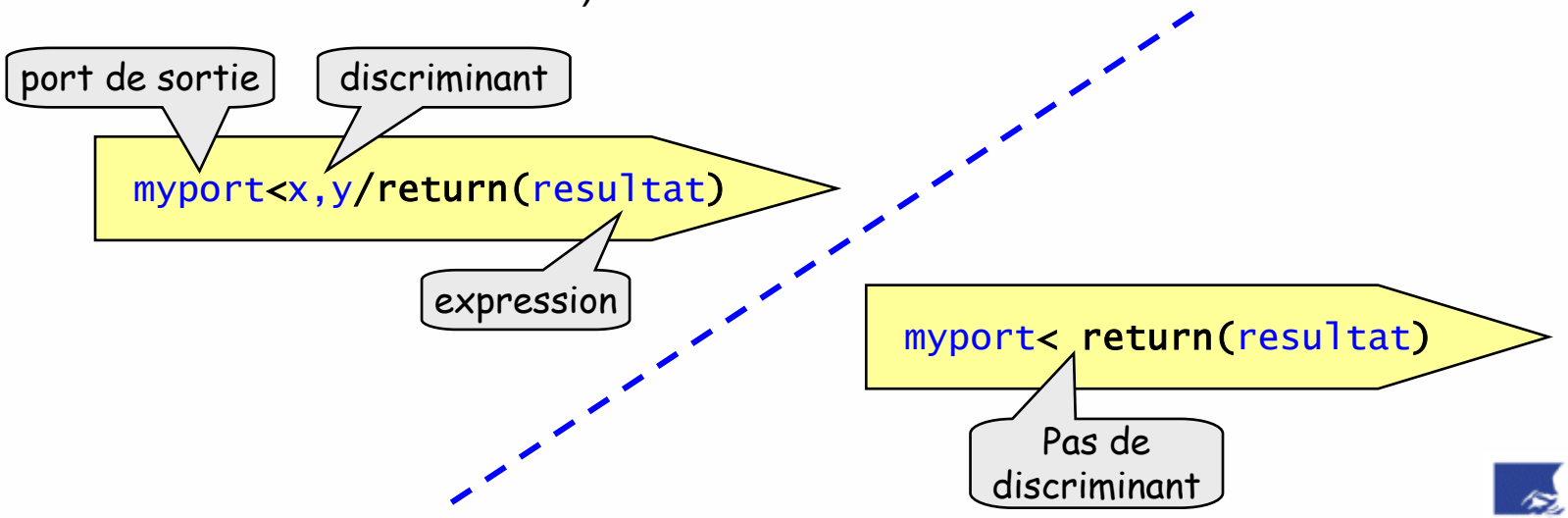


Transition « return » dans un diagramme de méthode

- La Transition « return » dans un diagramme de méthode est représentée dans le diagramme d'états correspondant par un symbole d'émission de signal, situé sur un arc de transition.
- Le label de ce symbole suit la syntaxe suivante :

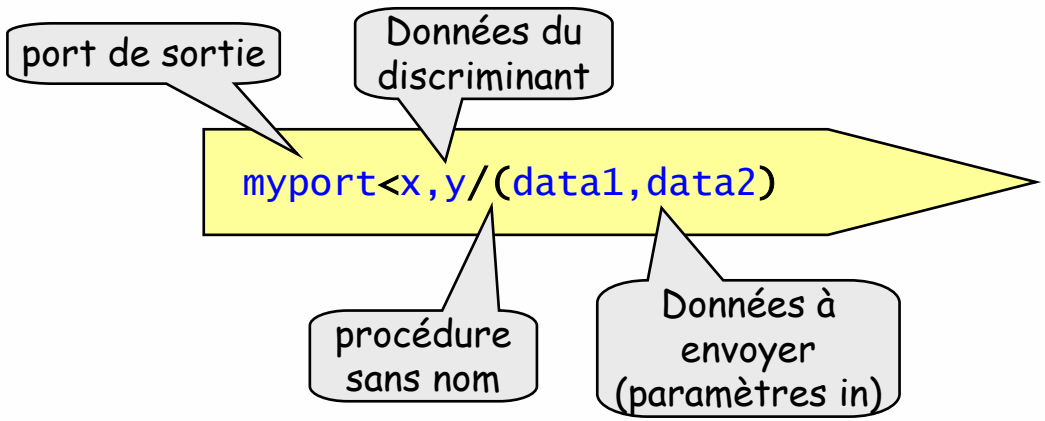
  `port_de_sortie < discriminant / return(expression)`

- Le discriminant est optionnel. En fait la classe est supposée ne pas avoir accès au discriminant : elle renvoie dans le message de retour le même discriminant qu'elle a reçu dans le message d'appel. (ce message d'appel peut avoir été transmis par un média sans son discriminant)



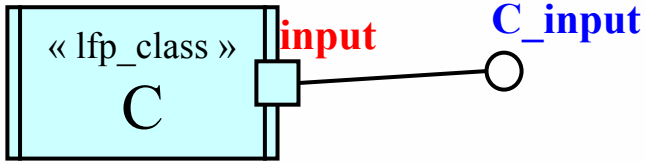
Envoi de message par une classe

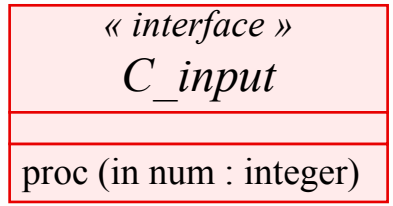
- Chaque classe peut envoyer ou recevoir un message, composé d'un nombre arbitraire de données, et accompagné d'un discriminant. Ce type de communication se réduit à une transmission de données. L'envoi de message est autorisé dans tout diagramme d'état, y compris dans les sous-diagrammes.
- Pour sa représentation UML, l'envoi d'un message par une classe est assimilé à l'appel d'une opération spéciale asynchrone, n'ayant pas de nom.
- Cette opération est implicitement déclarée (voir commentaire). Elle admet un nombre arbitraire de paramètres, de types quelconques, de mode in.



Représentation d'une transition de type réception d'appel de méthode

- Le port sur lequel est attendu l'appel est connu d'après le diagramme de composants

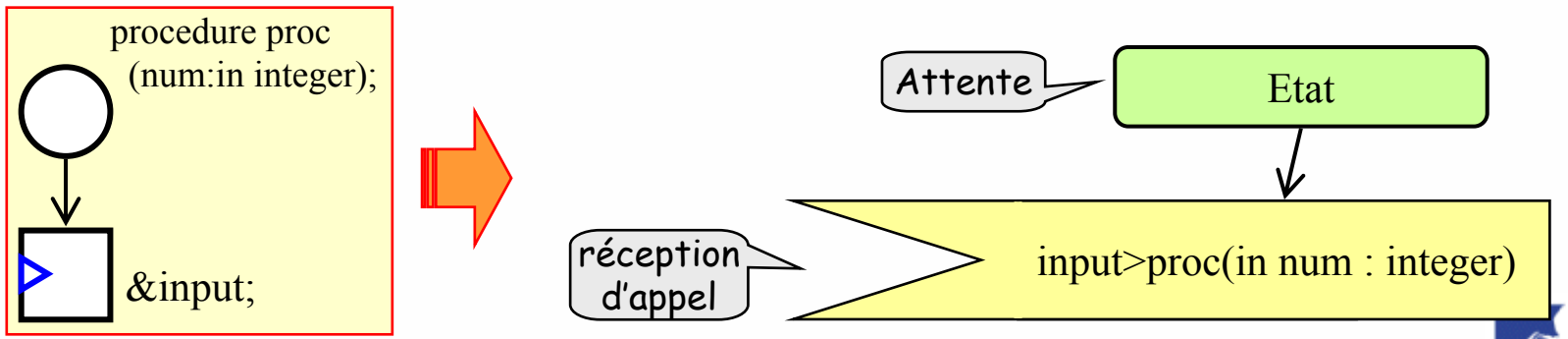




- L'appel est représenté dans les diagrammes d'états par un symbole de réception de signal, situé sur un arc de transition.
  Le label de ce symbole doit avoir la syntaxe suivante :

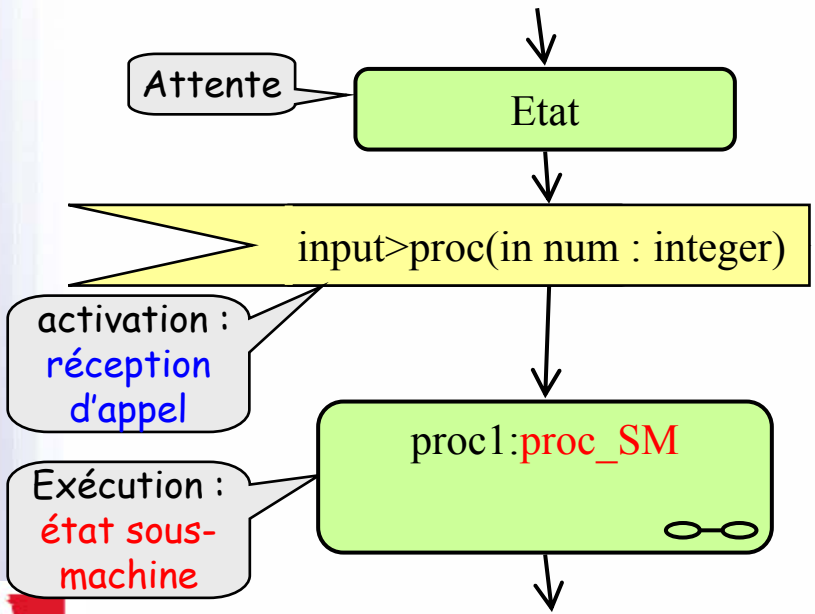
  `port_d_entrée > spécification_d_opération`

  (la spécification d'opération doit être conforme à la syntaxe UML)

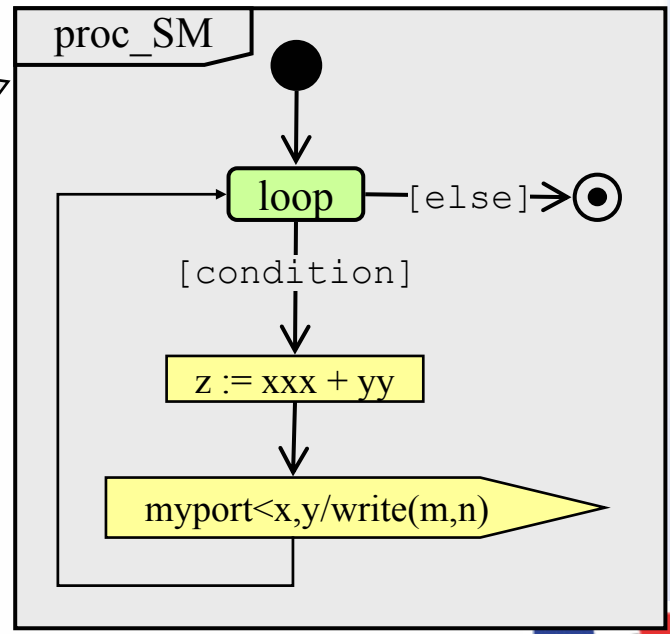


Représentation d'une méthode, et de son activation

- Le diagramme de méthode est représenté par une sous-machine à états. Cette sous-machine *n'inclut pas la réception* de l'appel.
- L'exécution de la méthode est représentée par un état sous-machine, référençant la sous-machine.
- Une même sous-machine peut être référencée plus d'une fois dans le diagramme d'états principal de la classe



Diag. de méthode : sous-machine

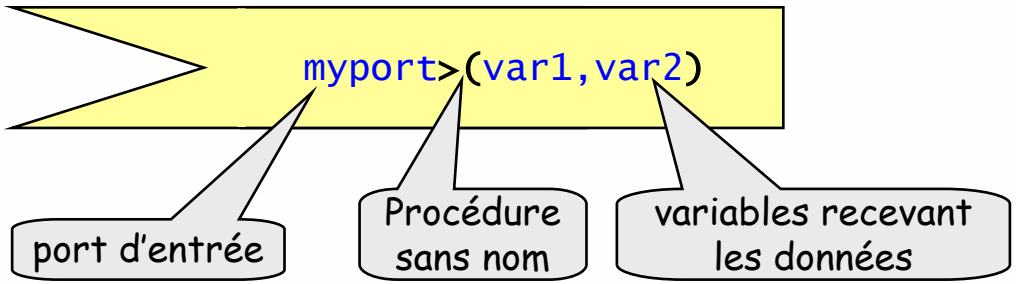


Réception de message par une classe

- Comme l'envoi, la réception ou lecture de message est autorisée dans tout diagramme d'état, y compris dans les sous-diagrammes.
- La lecture d'un message par une classe est vue en UML comme l'appel d'une opération spéciale, n'ayant pas de nom. Attention: cet appel est porté par un symbole de réception de signal (voir commentaire). Syntaxe :

  `port_d_entrée > (paramètres_effectifs)`

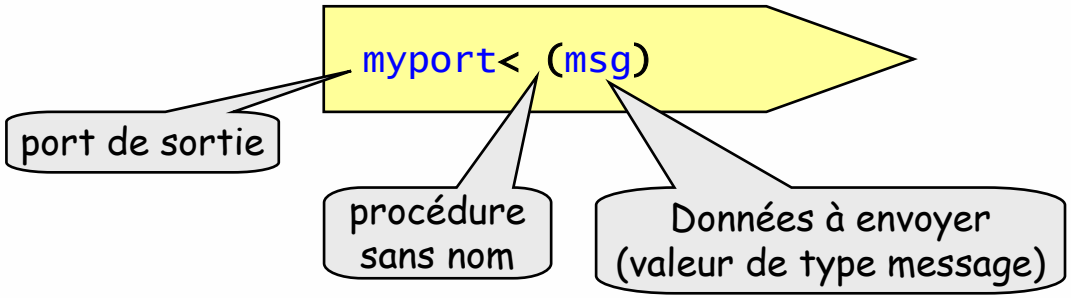
- L'opération sans nom est implicitement déclarée (voir commentaire). Elle admet un nombre arbitraire de paramètres, de mode out. Les paramètres doivent être des variables. La lecture consiste à retirer un message du binder et à affecter aux paramètres le contenu du message.
- Les données du message doivent être correctement typées par rapport aux types des paramètres effectifs. L'opération de lecture est bloquante si le binder dans lequel le message est lu est synchrone et ne délivre pas un message correctement typé. (voir doc sémantique LfP).



Envoi de message par un média

- L'envoi d'un message par un média est semblable à l'envoi de message par une classe sauf sur les points suivants :
  - L'envoi d'un message par un média ne comporte pas de partie discriminant.
  - L'ensemble des données envoyées est vu par le média comme encapsulé dans une valeur unique, du type prédéfini message. Ce type n'est utilisable que par les médias. Le média ne sait pas si une valeur de type message véhicule un appel d'opération, ou un retour d'appel, ou un simple message.
- L'envoi est représenté dans le diagramme d'états par un symbole d'émission de signal, situé sur le chemin d'une transition. Le label de ce symbole doit avoir la syntaxe suivante :

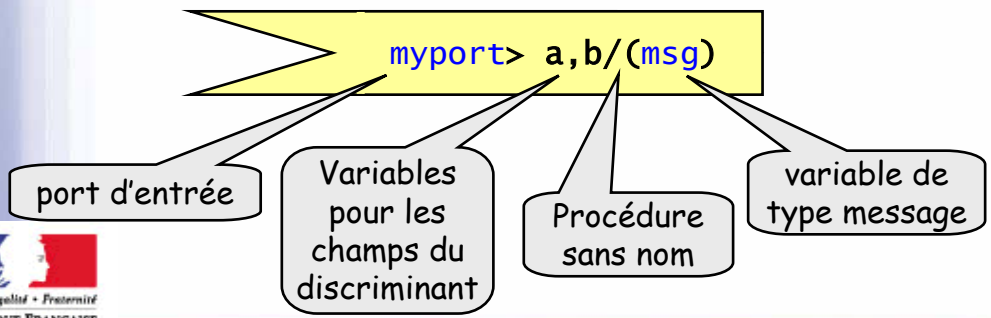
```
port_de_sortie < ( valeur_de_type_message )
```

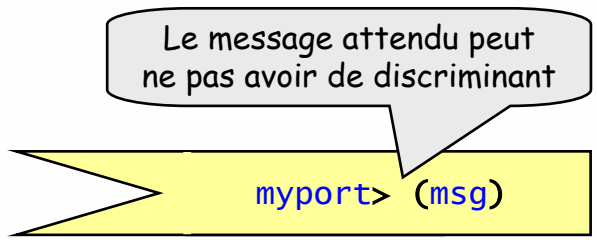


Réception de message par un média (1)

- La réception d'un message par un média est semblable à la réception de message par une classe sauf sur les points suivants :
  - La réception d'un message par un média peut inclure la réception d'un discriminant.
  - L'ensemble des données reçues est vu par le média comme encapsulé dans une valeur unique, du type prédéfini message. Le média fournit une variable de type message, paramètre en mode out dans lequel le message reçu est stocké.
- La réception est représentée dans le diagramme d'états par un symbole de réception de signal, situé sur le chemin d'une transition. Le label de ce symbole doit avoir la syntaxe suivante :

  `port_d_entrée > variables/( variable_de_type_message )`
- Les `variables` sont destinées à recevoir les valeurs des composants du discriminant. La partie `variables/` est omise si le message attendu n'a pas de discriminant.





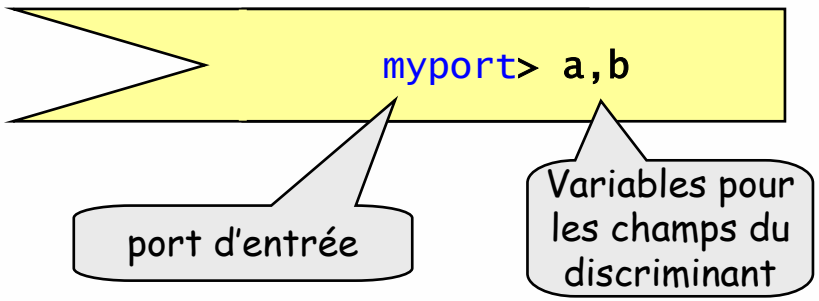
Réception de message par un média (2) message de contrôle

NB: si la transition de réception est munie d'une garde, cette garde peut dépendre de la valeur du discriminant (voir doc sémantique LfP). Le discriminant est donc lu avant l'évaluation de la garde.
Le message, lui, n'est lu et retiré du binder que si la garde vaut True.

- Réception d'un message de contrôle :
  - Un message de contrôle ne contient qu'un discriminant.
  - sa réception est représentée dans le diagramme d'états par un symbole de réception de signal. Le label de ce symbole doit avoir la syntaxe suivante :

    `port_d_entrée > variables`

- Les `variables` sont destinées à recevoir les valeurs des composants du discriminant.

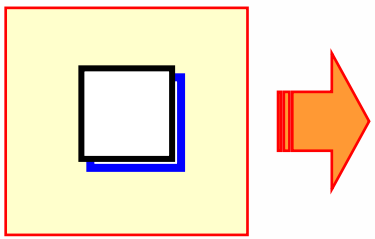


Représentation des transitions hiérarchiques

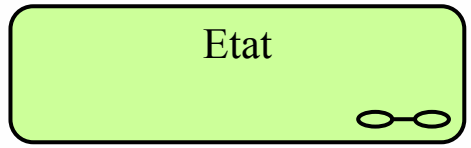
- Une transition hiérarchique LfP peut représenter
  - un sous-diagramme,
  - une méthode,
  - un trigger.
- Sa représentation UML sera fournie par
  - Un état composé dans le cas d'un sous-diagramme,
    - => simple mécanisme de décomposition d'un diagramme trop chargé.
  - Une sous-machine à états dans le cas d'une méthode ou d'un trigger.
    - => référençable plusieurs fois dans un même diagramme d'états.

Représentation d'une transition hiérarchique LfP représentant un sous-diagramme

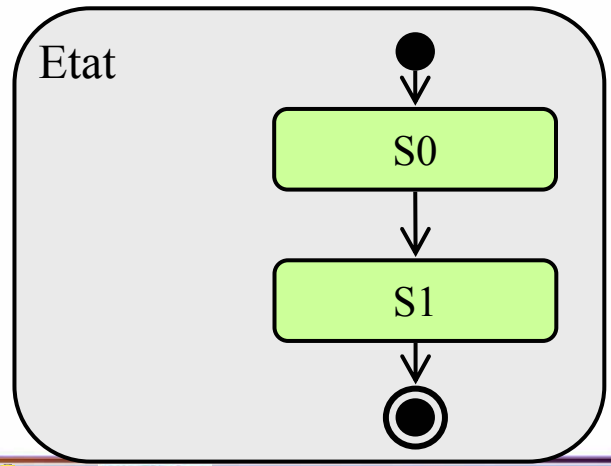
- Transition hiérarchique, (référence à un sous-diagramme)



- état composé (vue synthétique)

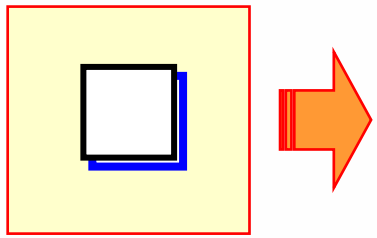


- état composé (vue interne)



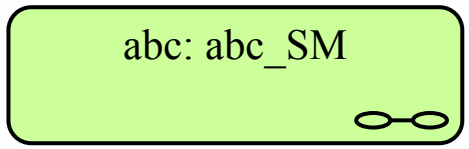
Transition hiérarchique représentant l'exécution d'une opération ou d'un trigger

- Transition hiérarchique, (référence à un diagramme de trigger ou de méthode)

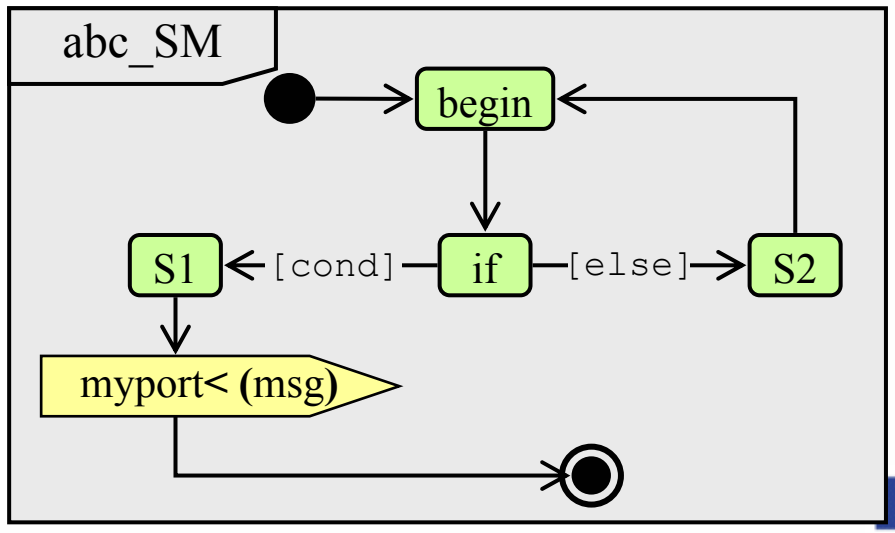


- Rappel : contrairement aux sous-diagrammes, les diagramme de trigger ou de méthode peuvent être invoqués (référencés) plusieurs fois.

- état référençant une sous-machine (plusieurs états peuvent référencer une même sous-machine)



- sous-machine



Sommaire

- 1. Diagrammes d'Architecture
  - Exemple : DA de l'application Client-Serveur
- 2. Déclarations
  - types de données, variables, constantes
- ➡ 3. Diagrammes de Comportement
  - ➡ Exemple : DC de l'application Client-Serveur

Exemple : DC de l'application Client-Serveur

- Cette application comporte 4 diagrammes ou sous-diagrammes de comportement :
  - Diagramme principal de la classe Server
  - Diagramme de méthode : Server.handle_request
  - Diagramme principal de la classe Client
  - Diagramme principal du média RPC

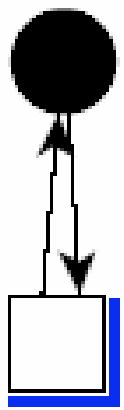
Sommaire

Diagramme principal de la classe Server

server

client_server

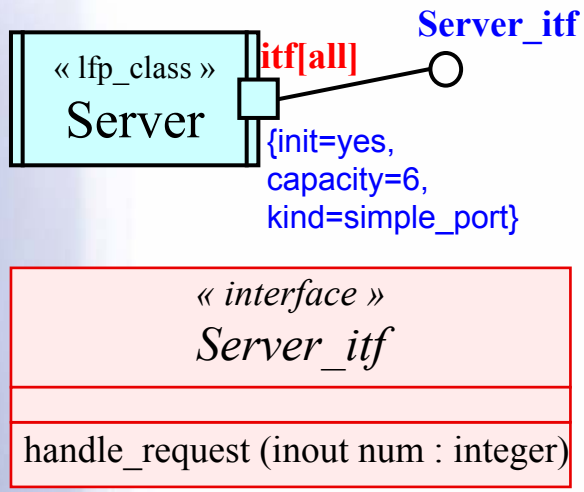
```
itf : simple_port ;
procedure handle_request (num : inout integer) ;
```



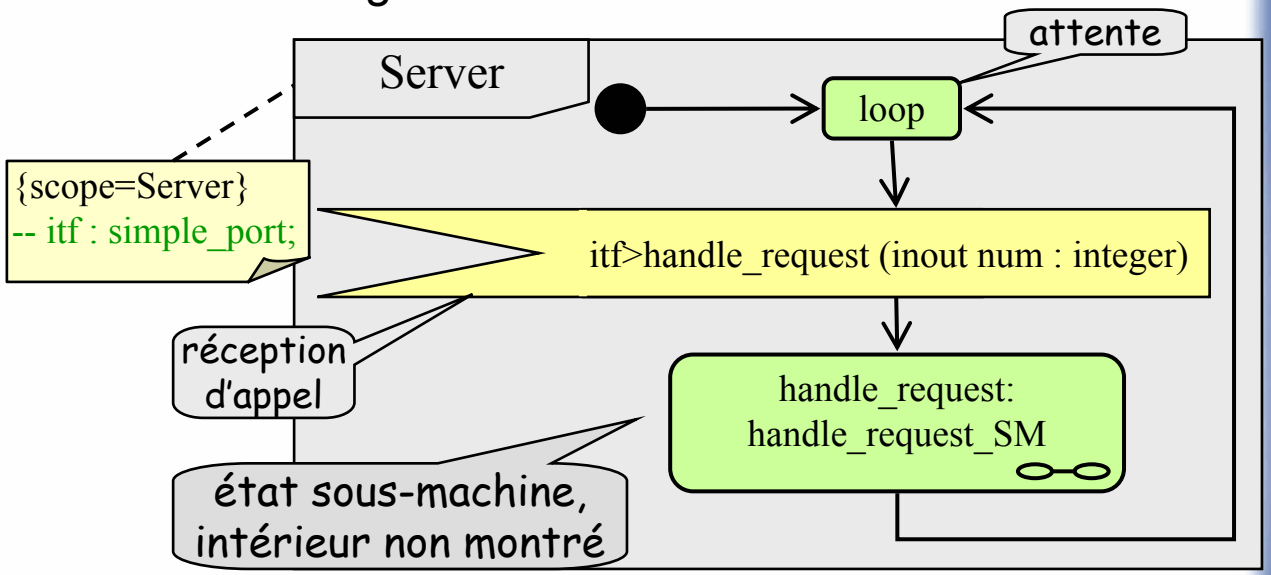
handle_request

Diagramme principal de la classe Server; Equivalent UML

- Extrait du diag. de composants



- Diagramme d'états de Server



- La transition hiérarchique (méthode) se traduit par un état sous-machine
- L'attente/réception d'appel sont modélisés à l'extérieur de la sous-machine référencée. Seule l'exécution de la méthode est modélisée à l'intérieur.
  - La sous-machine est nommée en ajoutant au nom de la méthode un suffixe _SM.
  - Si elle est référencée plusieurs fois dans le diag ppal. => donner des noms différents aux états qui la référencent (ex. : handle_request_1, handle_request_2)
- Le port sur lequel est attendu l'appel est connu d'après le diagramme de composants
- La transition vers l'état initial LfP est traduite par une transition vers l'état désigné par le pseudo-état initial UML

Sommaire

- 1. Diagrammes d'Architecture
  - Exemple : DA de l'application Client-Serveur
- 2. Déclarations
  - types de données, variables, constantes
- ➡ 3. Diagrammes de Comportement
  - ➡ Exemple : DC de l'application Client-Serveur
    - Diagramme principal de la classe Server
    - ➡ Diagramme de méthode : Server.handle_request
    - Diagramme principal de la classe Client
    - Diagramme principal du média RPC

Diagramme de méthode : Server.handle_request

server/handle_request
client_server

```
procedure handle_request
    (num : inout integer) is end;
```

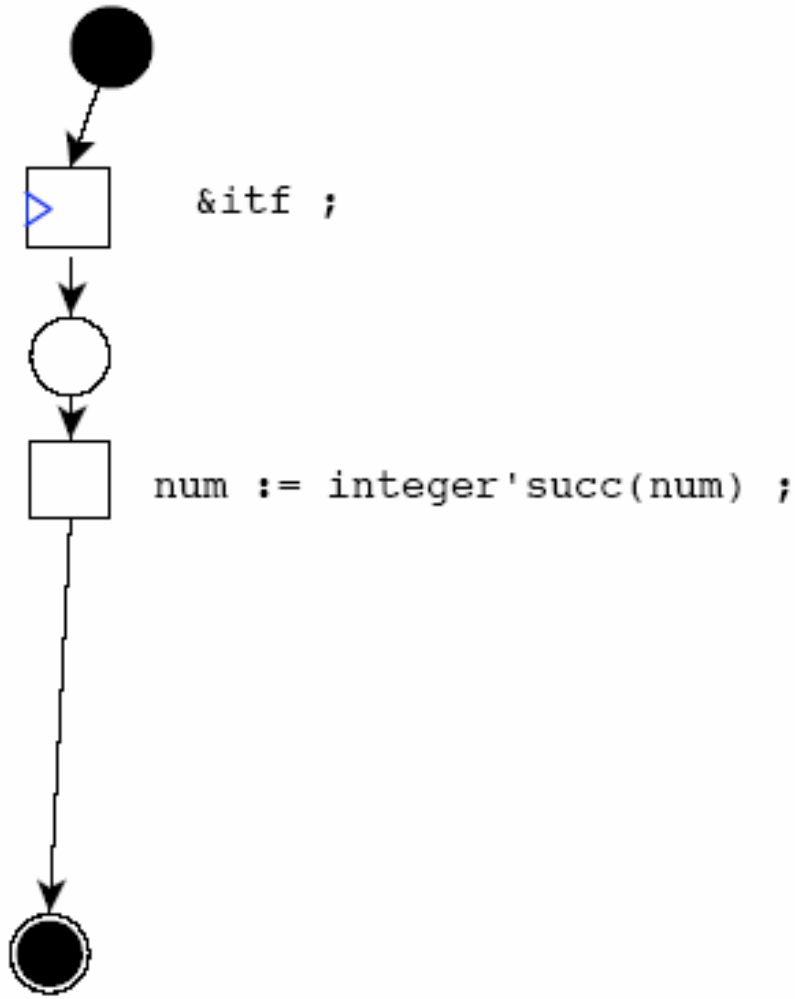
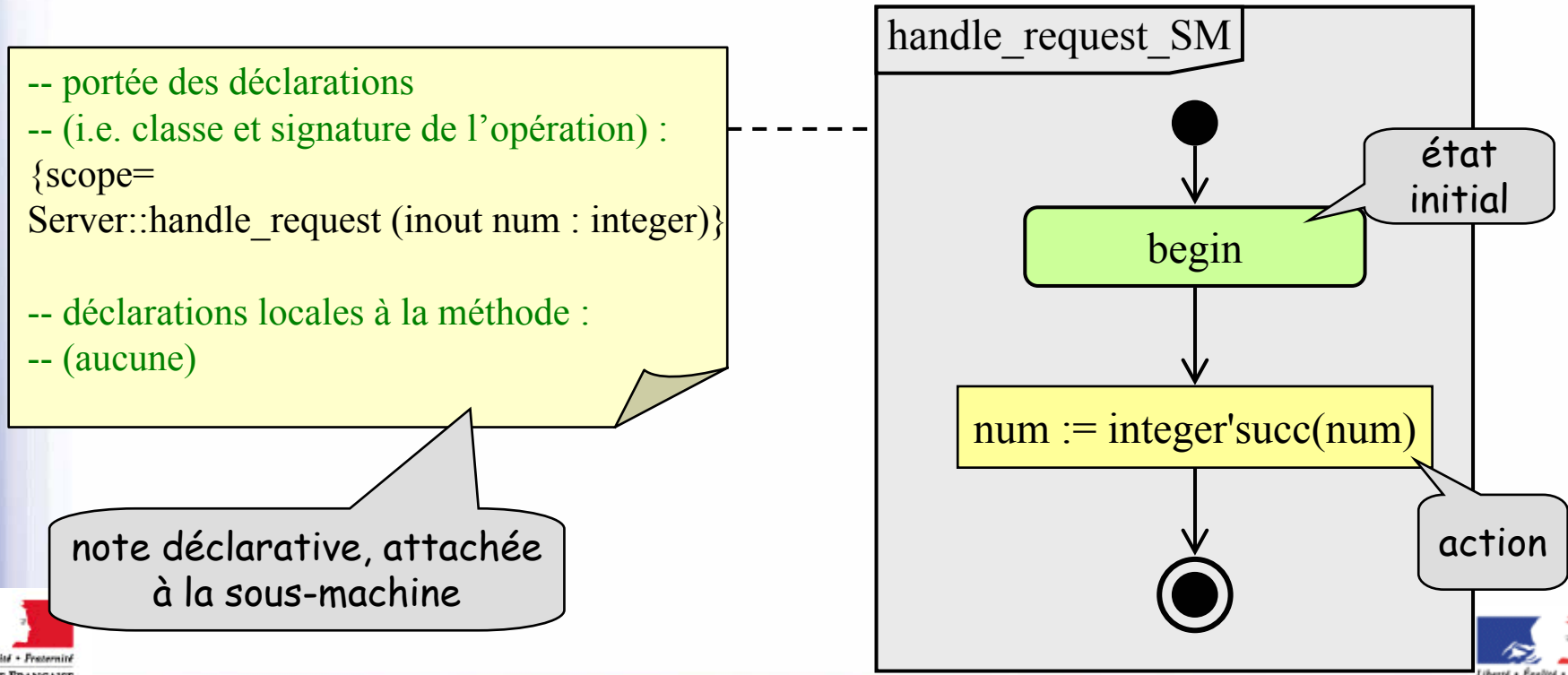


Diagramme de méthode : Server.handle_request; Equivalent UML

- Vue interne de la sous-machine handle_request_SM
  - N'inclut pas la réception d'appel
  - Plusieurs instances (invocations) de cette sous-machine peuvent exister dans le diagramme principal du composant Server.



Sommaire

- 1. Diagrammes d'Architecture
  - Exemple : DA de l'application Client-Serveur
- 2. Déclarations
  - types de données, variables, constantes
- ➡ 3. Diagrammes de Comportement
  - ➡ Exemple : DC de l'application Client-Serveur
    - Diagramme principal de la classe Server
    - Diagramme de méthode : Server.handle_request
    - ➡ Diagramme principal de la classe Client
    - Diagramme principal du média RPC

Diagramme principal de la classe Client

i, req_response : integer := 0 ;
id : integer ;
itf : simple_port ;
link : rpc ;

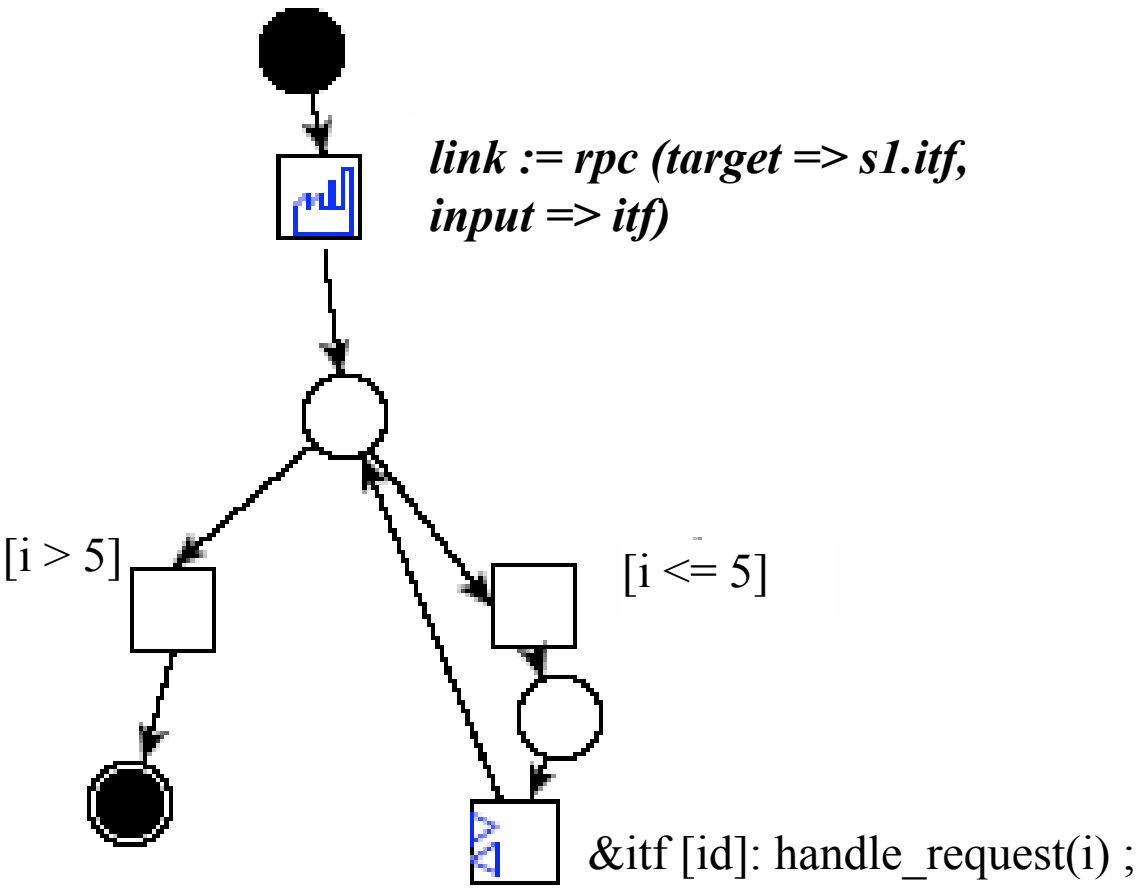
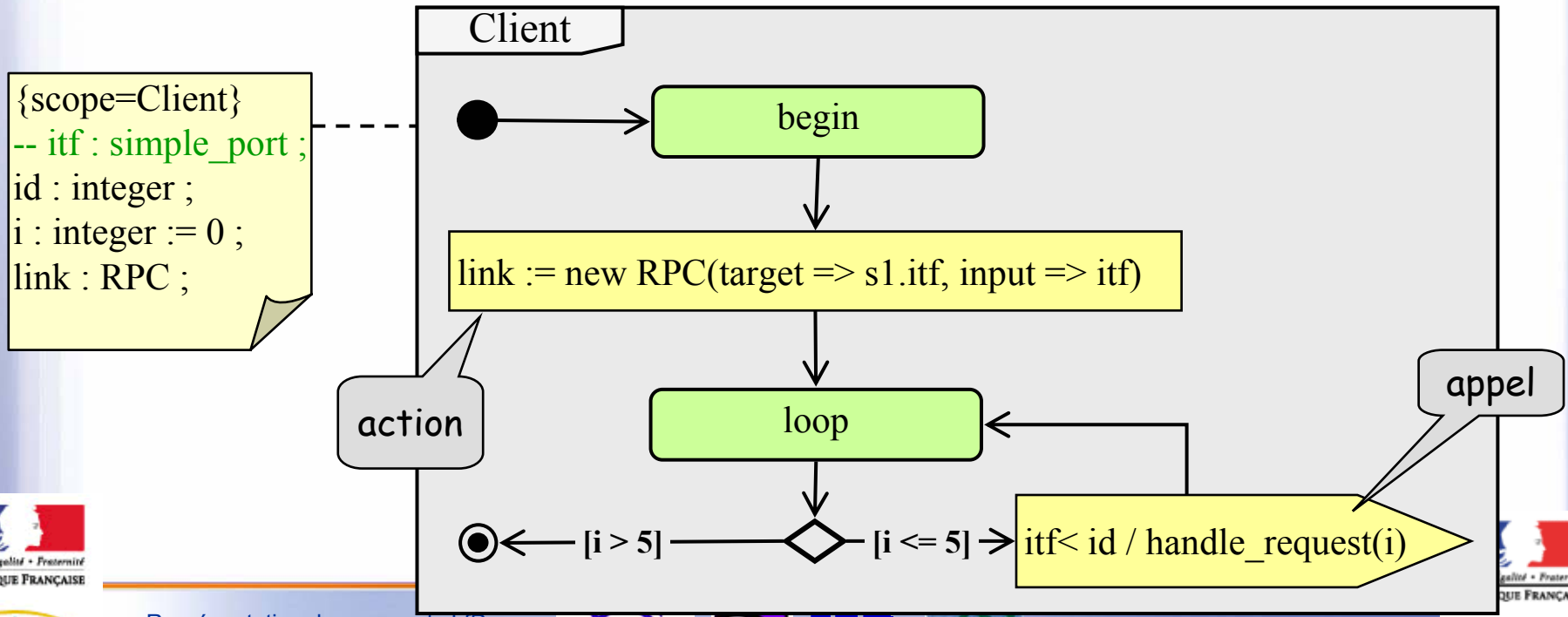


Diagramme principal de la classe Client; Equivalent UML

- Extraits du diagramme de composants et des déclarations globales

- Diag. d'états



Sommaire

Diagramme principal du média RPC

RPC
client_server

```
msg : message ;
input : simple_port ;
target : simple_port ;
id1, id2 : integer ;
```

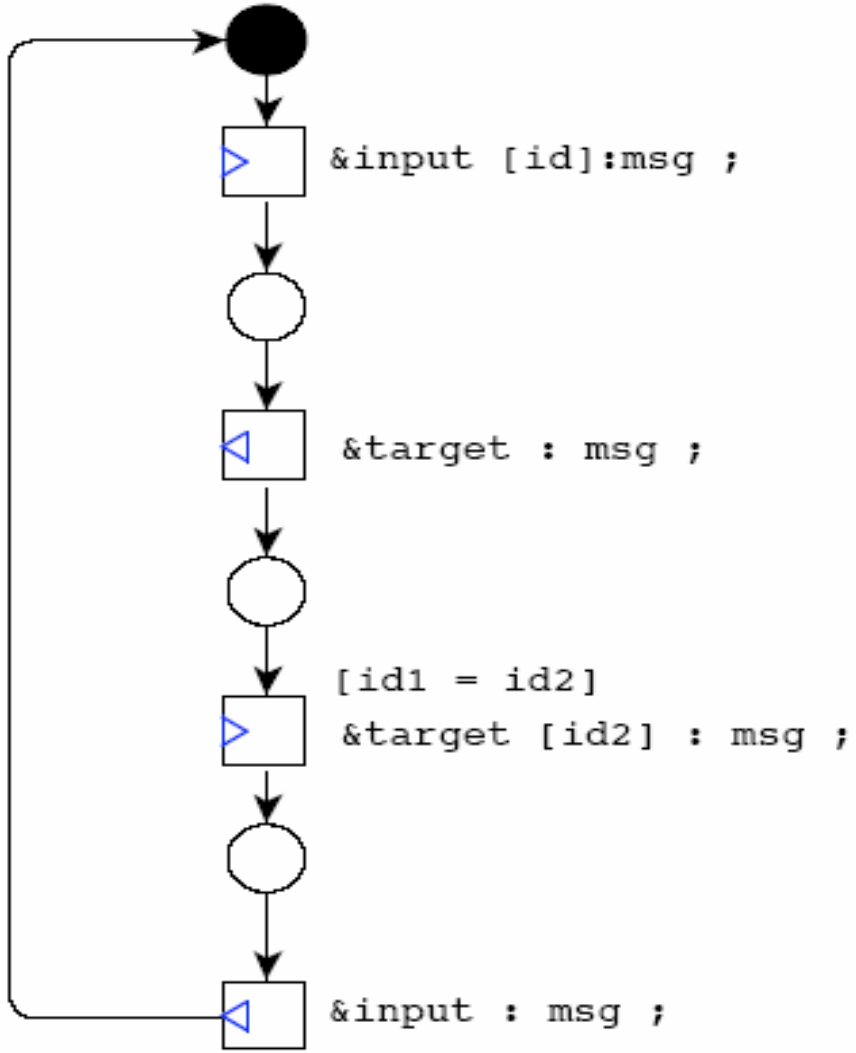
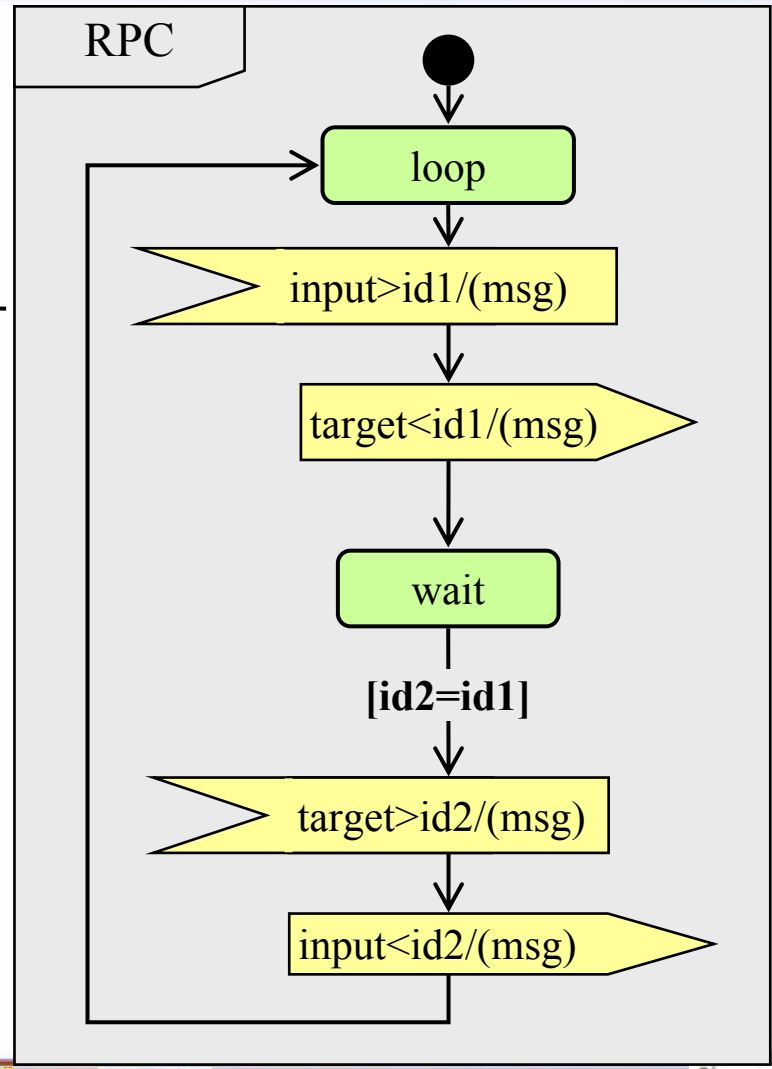


Diagramme principal du média RPC , équivalent UML

```
-- portée des déclarations
{scope=RPC}

-- déclarations locales au média :
-- input : simple_port ;
-- target : simple_port ;
msg : message ;
id1, id2 : integer ;
```



- Extrait du diag. de composants