PC3R
Cours 04 - Passage de Message (II)

Romain Demangeon

PC3R MU1IN507 - STL S2

18/02/2021

Plan du Cours 4

- Canaux et évènements en *OCaml*.
- Futures
- Clients-Serveurs
- Appels Distants

OCaml: Concurrence

- *OCaml* possède des mécanismes élégants (fortement typé, encapsulé dans des modules) de programmation concurrente:
  - threads préemptifs avec mutexes et variables de conditions,
  - concurrence asynchrone `Async`,
  - monade de concurrence coopérative `Lwt`,
  - évènements et canaux synchrones `Event`.
- *OCaml* ne possède pas de modèle concurrent efficace:
  - l'environnement d'exécution utilise un verrou global sur les threads systèmes créés par le programme,
  - un seul thread d'une application s'exécute à la fois (comme en *Python*)
  - en s'interfaçant avec du *C*, il est possible de lancer des threads sur plusieurs coeurs.
- Ca reste un bon langage pédagogique pour la concurrence
- *Multicore OCaml* par *OCaml Labs* pourrait un jour (???) exister.

OCaml: Canaux synchrones

- le module `Event` offre des mécanismes de communication et synchronisation par évènements.
- les canaux sont des objets manipulables, avec un type spécifique,
- les évènements sont des objets manipulables, avec un type spécifique,
- les opérations élémentaires sur les canaux (envoi/réception) renvoient des évènements.
- une primitive de synchronisation permet d'attendre qu'un évènement se produise.
- les canaux sont fortement et statiquement typés
  - pas de polymorphismes douteux avec `chan interface{}`

OCaml: Evénements

- deux types abstraits : `'a channel` et `'a event`
- `new_channel : unit -> 'a channel` : création d'un canal
- `send : 'a channel -> 'a -> unit event` : envoi
  - renvoie un évènement de type `unit`.
- `receive : 'a channel -> 'a event`: réception
- ni `send` ni `receive` ne sont bloquants.
  - ils renvoient un évènement qu'on peut utiliser directement (le passer à une fonction, par exemple)
- `sync : 'a event -> 'a` : fonction de synchronisation
  - bloquante jusqu'à ce que l'évènement se produise.

OCaml: Evénements (exemple)

```
let ch = Event.new_channel () ;;
let v = ref 0;;

let reader () = Event.sync (Event.receive ch);;
let writer () = Event.sync (Event.send ch ("S" ^ (string_of_int !v)));;

let loop_reader s d () =
  for i=1 to 10 do
    let r = reader() in
    print_string (s ^ " " ^ r); print_newline();
    Thread.delay d
  done ;;

let loop_writer d () =
  for i=1 to 10 do incr v; writer(); Thread.delay d
  done ;;

Thread.create (loop_reader "A" 1.1) ();;
Thread.create (loop_reader "B" 1.5) ();;
Thread.create (loop_reader "C" 1.9) ();;
Thread.delay 2.0;;
loop_writer 1. ();;
```

- souvent, on enchaîne la synchronisation aux opérations d'envoi/réception.

OCaml: Enveloppe

- `poll: 'a event -> 'a option` : non bloquant, retourne `Some v` si un événement l'évènement est disponible, `None` sinon,
- `always : 'a -> 'a event` : crée un évènement disponible
  - c'est un *return*
- `wrap : 'a event -> ('a -> 'b) -> 'b event`: enveloppe l'évènement d'un futur.
  - `wrap f e` est disponible quand `e` est disponible de valeur `r` et `f r` est disponible,
  - la valeur de `wrap f e` est la valeur de `f r` si `r` est la valeur de `e`.
  - c'est un *foncteur*

OCaml: Enveloppe

- `poll: 'a event -> 'a option` : non bloquant, retourne Some `v` si un événement l'évènement est disponible, `None` sinon,
- `always : 'a -> 'a event` : crée un évènement disponible
  - c'est un *return*
- `wrap : 'a event -> ('a -> 'b) -> 'b event`: enveloppe l'évènement d'un futur.
  - `wrap f e` est disponible quand `e` est disponible de valeur `r` et `f r` est disponible,
  - la valeur de `wrap f e` est la valeur de `f r` si `r` est la valeur de `e`.
  - c'est un *foncteur*
  - du coup on pourrait écrire:

```
bind : 'a event -> ('a -> 'b event) -> 'b event
let bind e f = wrap (fun x -> Event.Sync (f x)) e
```

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

OCaml: Enveloppe (exemple)

- `wrap` est particulièrement utile avec les primitives de choix:
  - `choose: 'a event list -> 'a event`
    - renvoie un évènement correspondant au premier évènement disponible de la liste.
  - `select: 'a event list -> 'a`
    - c'est choose + sync

```
let rec accum sum =
  print_int sum; print_newline();
  Event.sync (
  Event.choose [
    wrap (receive addCh) (fun x -> accum(sum + x));
    wrap (receive subCh) ( fun x -> accum(sum - x));
    wrap (send readCh sum) ( fun x -> accum(sum))
    ]
  )
```

Encodage de la mémoire partagée

```
type 'a mvar = MV of ('a Event.channel *
                      'a Event.channel *
                       bool Event.channel)

let mVar () =
  let takeCh = Event.new_channel ()
  and putCh = Event.new_channel ()
  and ackCh = Event.new_channel () in
  let rec empty () =
    let x = Event.sync (Event.receive putCh) in
    Event.sync (Event.send ackCh true);
    full x
  and full x = Event.select
    [Event.wrap (Event.send takeCh x) empty ;
     Event.wrap (Event.receive putCh)
       (fun _ -> (Event.sync (Event.send ackCh false); full x))]
  in
  ignore (Thread.create empty ());
  MV (takeCh, putCh, ackCh)
```

- on utilise les canaux synchrones pour simuler une case mémoire.
- les opérations sur la case sont atomiques

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Exercice CS de l'examen PC2R de 2016

- table d'association clefs (chaînes de caractères) / valeurs (entiers) partagée
- implémentation séquentielle:

```
type 'a option = Some of 'a | None
type assoc
cr_assoc : unit -> assoc
setv: assoc * string * int -> assoc
getv: assoc * string -> int option
```

Un thread "serveur" `table_p` qui maintient une `assoc` et est joignable sur deux canaux publics `s` (mise à jour) et `g` (récupération) Les clients ont un canal personnel (différent), et connaissent `s` et `g` du serveur:

- pour mettre à jour, ils envoient sur `s` une paire `(k, v)` à inclure dans la table ;
- pour récupérer une valeur, ils envoient sur `g` une paire `(cp, k)`, canal personnel et clef de la table. Ils attendent ensuite sur cp qu'on leur envoie la valeur entière de l'association avec k. Si la clef n'existe pas dans la table, ils attendent indéfiniment.

Exercice CS de l'examen PC2R de 2016 (II)

```
type assoc = string * int list
type 'a option = Some of 'a | None

let cr_assoc () = []

let setv ass k v = (k,v)::ass

let rec getv ass k = match ass with
    [] -> None
  | (k',v)::q -> if (k = k') then Some(v) else (getv q k)

let g = Event.new_channel ()
let s = Event.new_channel ()
let c2 = Event.new_channel ()
```

- en-tête du système
  - code de la table d'association,
  - création des canaux,

Exercice CS de l'examen PC2R de 2016 (III)

```
let rec table_p ass =
  let conts (k, v) =
    (print_endline "Serveur: Mise a jour.");(table_p (setv ass k v))
  in
  let contg (cp, k) =
    match (getv ass k) with
      None -> (print_endline "Serveur: Pas trouve.");(table_p ass)
    | Some(v) ->
        let _ = (Event.sync (Event.send cp v)) in
        (print_endline "Serveur: Trouve.");(table_p ass)
  in
  Event.select [(Event.wrap (Event.receive s) conts);
                (Event.wrap (Event.receive g) contg)]
```

- code du serveur:
  - il se passe récursivement une table `ass`,
  - il sélectionne sur les réceptions des deux canaux,
  - les continuations (`wrap`) sont définies comme fonctions internes.

Exercice CS de l'examen PC2R de 2016 (IV)

```
let client1 () =
  let _ = Event.sync (Event.send s ("brouette", 28)) in
  let _ = Thread.delay (Random.float 2.0) in
  Event.sync (Event.send s ("brouette", 15))

let client2 () =
  let _ = Thread.delay (Random.float 1.0) in
  let _ = Event.sync (Event.send g (c2, "brouette")) in
  let a = Event.sync (Event.receive c2) in
  print_string "Client: Valeur"; print_int a; print_endline ""

let main =
  let _ = Thread.create table_p (cr_assoc ()) in
  let c1 = Thread.create client1 () in
  let c2 = Thread.create client2 () in
  Thread.join c1;
  Thread.join c2
```

- code des clients:
  - le premier associe `"brouette"`
  - seul le deuxième cherche `"brouette"`
  - en fonction des valeurs des `delay`, le 2ème client peut bloquer indéfiniment ou non.

Futures

- Principe: créer un calcul qui sera exécuté séparément (sur un autre processus) et qui produit un résultat.
- Manipulation:
  - création et lancement du calcul,
  - attente du résultat,
  - sondage de terminaison.
- Utilité:
  - déléguer des calculs lourds sur un autre coeur pendant que le programme principal continue son flot de calcul,
  - reporter le caractère bloquant d'une entrée-sortie jusqu'au moment où l'on a besoin de son résultat.
  - déléguer des calculs spécifiques sur une unité de calcul distante (future à travers un réseau, par exemple).
- deux manières principales d'utiliser les futures:
  - manière bloquante:
  - manière non-bloquante:
- c'est la base de la programmation asynchrone.
  - accessible directement dans certains langages (*Scala*, *Java*, *Rust*, `Lwt` d'*OCaml*)
  - facilement programmable depuis une bibliothèque de threads.

Futures: Utilisation

Pour manipuler des futures il faut:

- un type pour les futures, qui intègre le type du résultat
  - parfois le type existe dans le langage
    - la classe générique `Future<T>` en *Java*
    - la classe générique `Future[T]` en *Scala*
    - le trait `Future<Item=T, Error=Box<Error>>` en *Rust*
  - on l'implémente facilement dans les langages avec généricité:
    - `type 'a future` en *OCaml*
  - c'est moins évident sans:
    - `type intFuture ...` en *Go*
  - c'est théoriquement une monade:
    - return : $A \to \text{Fut } A$
      c'est $\lambda x.\texttt{spawn}(\lambda().x)$
    - bind : $\text{Fut } A \to (A \to \text{Fut } B) \to \text{Fut } B$
      c'est $\lambda x, f.f\ (\text{wait } x)$
    - clairement, bind (return x) f c'est f (wait (spawn($\lambda().x$))) donc $f\ x$
    - bind m return c'est spawn $\lambda().(\text{wait } m)$ équivalent m
    - bind est associatif.
  - concrètement, les futures sont utilisées pour faire des effets de bords, ce qui casse les lois monadiques. (transparence référentielle)

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Futures: Utilisation (II)

Pour des manipuler des futures il faut:

- un créateur, qui prend un calcul, et lance un thread qui l'exécute.
  - le créateur doit retourner une manière de récupérer le calcul quand il est fini (une implémentation du type des futures)
  - `spawn: 'a -> 'a future`
- une attente qui bloque sur une future et récupère le résultat quand il est disponible
  - `wait: 'a future -> 'a`

On peut trouver aussi:

- un sondage:
  - `poll: 'a future -> bool`
  - permet de changer le flot de contrôle d'un programme en fonction de si un calcul est fini ou non
- une composition:
  - prend plusieurs futures et produit une future
  - souvent, les futures sont un foncteur
    - `fmap: ('a -> 'b) -> 'a future -> 'b future` est donné par $\lambda f, x.f$ (wait x)
    - on a bien g (wait (f (wait x)) qui vaut g (f (`wait` x))

Futures en *Go*

```
type futureInt struct {
        fut chan int
}

func spawnInt(f func() int) futureInt {
        c := make(chan int)
        go func() { c <- f() }()
        return futureInt{fut: c}
}

func waitInt(f futureInt) int {
        return <-f.fut
}

type futureString struct {...}
func spawnString(f func() string) futureString {...}
func waitString(f futureString) string {...}
```

- on propose une implémentation utilisateur des futures.
- une future, c'est simplement un canal sur lequel arrivera le résultat.
- à la création, on lance une goroutine.
- on définit une copie de chaque opération pour chaque type.

Futures en *Go* (II)

```
func pollInt(f futureInt) bool {
        select {
        case a := <-f.fut:
                go func() { f.fut <- a }()
                return true
        default:
                return false
        }
}

func composeWithFunIntInt(f1 futureInt, f2 futureInt, f func(int, int) int)
        return spawnInt(func() int { return f(waitInt(f1), waitInt(f2)) })
}

func onCompleteIntUnit(f futureInt, cont func(int)) {
        cont(waitInt(f))
}
```

- le sondage se fait grâce au cas `default` de la sélection.
  - il faut penser à repasser le résultat sur le bon canal, si on l'a récupéré.
- la composition est typée
  - on écrit des composition pour chaque type de fonction.
- le rappel aussi.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Futures en *Scala*

- en *Scala*, les futures sont natives:
  - type générique `Future[T]`,
  - chaque `Future` est exécutée (automatiquement, pas besoin de manipuler des threads) sur un thread séparé
- deux manières de gérer les futures:
  - de manière bloquante: on utilise `Await.result` qui attend la terminaison d'une future.
  - de manière non-bloquante: on passe à la future un (ou plusieurs) fonctions(s) de rappel (*callback*) à qui passer le résultat de la future quand celle ci termine.
  - l'utilisation non-bloquante des futures est encouragées:
    - plus élégant, ne bloque pas l'application,
    - si nécessaire, la synchronisation doit être faite à la main.
    - on peut composer des futures à l'aide des compréhensions.

Futures en *Scala* (II)

```
object Main extends App {

  val gen = scala.util.Random

  def somme (x:Int) : Future[Int] = Future[Int] {
    var acc = 0;
    val att = gen.nextInt(10)
    for (k <- 0 to x) {
      Thread.sleep(att)
      acc = acc + k
    }
    acc
  }

  var futures:Array[Future[Int]] = new Array[Future[Int]](10)
  for (i <- 0 to 9){
    futures(i) = somme(i * 1000)
  }
  println("Valeur:" + Await.result(futures(5), 5 seconds))
}
```

- on crée plusieurs futures d'entier,
- on attend de manière bloquante une de ces futures
  - l'attente prend un délai maximal

Futures en *Scala* (III)

```
object Main extends App {

  val gen = scala.util.Random

  def somme (x:Int) : Future[Int] = Future[Int] {
    var acc = 0;
    val att = gen.nextInt(10)
    for (k <- 0 to x) {
      Thread.sleep(att)
      acc = acc + k
    }
    acc
  }

  for (i <- 0 to 9){
    somme(i * 100).onComplete{
      case Success(k) =>  println("Valeur:" + k)
      case Failure(_) => println("Zut")
    }
  }

  Thread.sleep(10000)
}
```

- la méthode de rappel `onComplete` permet de lancer une future avec des instructions de rappel à calculer sur le résultat.

Futures en *Scala* (IV)

```
def somme (x:Int) : Future[Int] = Future[Int] {
  var acc = 0;
  val att = gen.nextInt(10)
  for (k <- 0 to x) {
    Thread.sleep(att)
    acc = acc + k
  }
  acc
}

val total = for {
  f1 <- somme(3)
  f2 <- somme(10)
  f3 <- somme(f1)
} yield (f2 + f3)

total.onComplete {
  case Success(k) => println("Valeur:" + k)
  case Failure(_) => println("Zut")
}
```

- les compréhensions de Scala permettent d'utiliser la fonctorialité
  - c'est `somme(3).flatmap(f1 => somme(10).flatmap(f2 => ...))`
  - avec `flatmap` de signature, dans `Future[T]`,
    `flatmap[S](f : T => Future[S]) : Future[S]`

Modèle Client-Serveur

Modèle de programmation basé sur:

- une séparation entre deux entités
  - géographiquement: communication distantes,
  - localement: espace de noms différents.
- une asymétrie des composants:
  - serveur: entité unique, centralisant des données ou de la puissance de calcul, point de synchronisation.
  - clients: entités multiples indépendantes entre elles.
- une asymétrie des messages:
  - requêtes du client vers le serveur
    - initiative de la communication,
    - demande d'accès à des données,
    - demande de calcul,
    - synchronisation avec un autre client.
  - réponses du serveur vers le client:
    - réaction du serveur,
    - contenu (souvent) plus lourd que la requête
- concurrence interne au serveur:
  - le serveur est toujours disponible
  - plusieurs requêtes de plusieurs clients sont traitées simultanément

L'Internet électronique

- Réseau organisé en couches de protocoles:
  - tuyaux
  - IP: transport de paquets (*datagrammes*)
  - TCP: protocole fiable de gestions des paquets
  - Application:
    - `HTTP`: protocole web (cf. Cours 06)
    - `SMTP`: protocole mail
    - `FTP`, `Telnet`, ...
- les protocoles sont des ensembles de règles décrivant la composition et la manipulation des messages
- la plupart des langages de programmation donne accès à des primitives systèmes accèdant à l'envoi et la réception de messages internet.

Serveurs internet

- programmation (relativement) bas-niveau
  - modèle des *sockets* Unix.
  - accessible dans la plupart des langages
  - serveur:
    - définition du domaine,
    - création,
    - définition des paramètres (mode, protocole),
    - liaison (*bind*) à une adresse,
    - mise à l'écoute (*listen*),
    - réception d'une demande de connection (*accept*)
  - client:
    - création
    - envoi d'une demande de connection (*connect*)
  - plus la gestion d'erreur
- programmation haut-niveau
  - on passe `adresse:port` à des primitives `Listen`, `Accept`, `Connect`
- dans les deux cas, la connexion (ou *socket*) est bidirectionnelle
  - la plupart des langages permettent de récupérer deux tampons d'écriture, entrée bloquante et sortie bloquante (vis-à-vis du tampon du système).
- les messages contiennent des chaînes de caractères
  - on peut construire des protocoles textes par-dessus

Client internet d'écho *Go*

```
if len(os.Args) < 3 {
  return}

add := os.Args[1] + ":" + os.Args[2]
conn, err := net.Dial("tcp", add)
if err != nil {
  return}

reader := bufio.NewReader(os.Stdin)
continu := true
for continu {
  fmt.Println("Entrer une ligne de texte (vide pour stopper):")
  chaine, _ := reader.ReadString('\n')
  if chaine == "\n" {
    fmt.Fprintf(conn, fmt.Sprint(chaine))
    continu = false
    conn.Close()
  } else {
    fmt.Print("Envoi:", chaine)
    fmt.Fprintf(conn, fmt.Sprint(chaine))
    reponse, _ := bufio.NewReader(conn).ReadString('\n')
    reponse = strings.TrimSuffix(reponse, "\n")
    fmt.Println("Recoit:", reponse)
 }}
```

- `Dial` permet d'obtenir une connexion à un serveur,
- *Go* dispose aussi de primitives de bas-niveau (sockets Unix)

Serveur internet d'écho *Go*

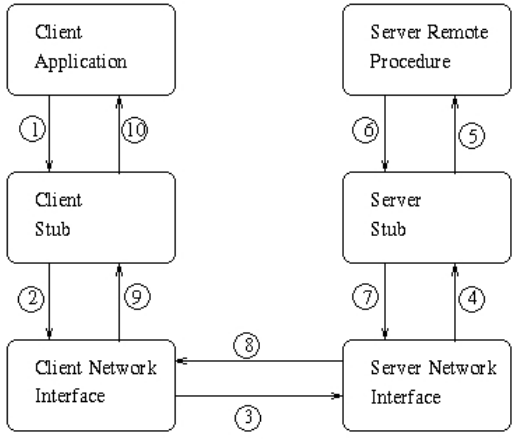
```
func gere_connection(conn net.Conn) {
  continu := true
  for continu {
    message, _ := bufio.NewReader(conn).ReadString('\n')
    if message != "\n" {
      conn.Write([]byte(strings.ToUpper(message)))
    } else {
      conn.Close()
      continu = false
}}}

func main() {
  if len(os.Args) < 2 {
    return}
  addr := ADRESSE + ":" + os.Args[1]
  ln, _ := net.Listen("tcp", addr)
  for {
    conn, _ := ln.Accept()
    go gere_connection(conn)
  }}
```

- `Listen` et `Accept` sont la contrepartie de `Dial`
- une goroutine lancée à chaque connexion (serveur concurrent)

Remote Procedure Call

- Définition: un Appel de Procédure Distante (RPC) est un protocole réseau permettant à un programme informatique de faire appel à une procédure (ou routine, ou méthode) s'exécutant dans un autre espace d'adresse (un ordinateur distant sur un réseau commun) à l'aide d'un serveur d'applications, sans que le développeur du programme initial code explicitement les interactions réticulaires.
- Maître-mot: Transparence.
- Idée datant de 1976.
- Paradigme de concurrence par passage de messages.



SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

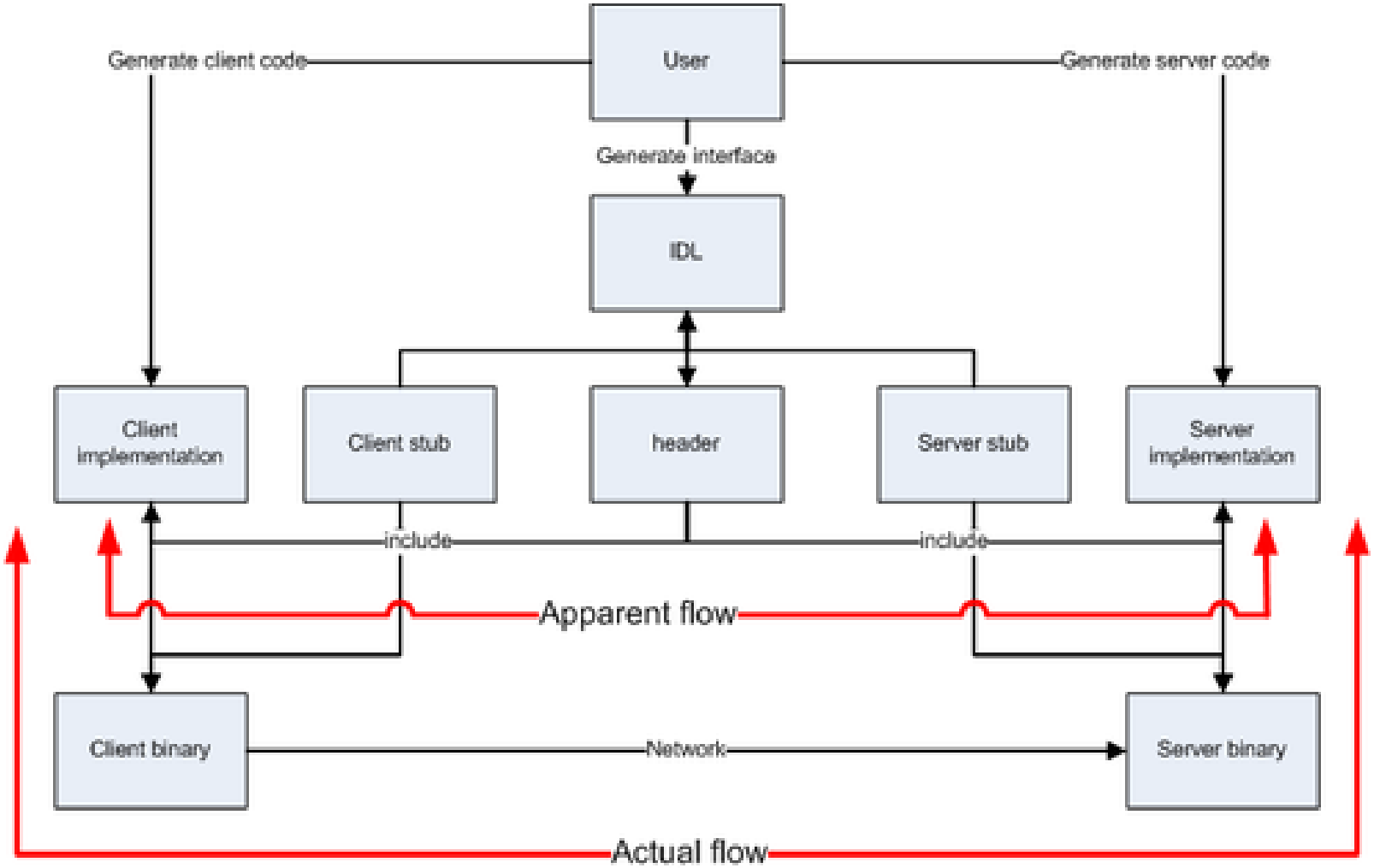
Principes du RPC

- Principe général: un serveur offre un service qui:
  - peut être appelé depuis un client,
  - peut prendre en compte des paramètres,
  - peut renvoyer une valeur.
- on trouve usellement en plus:
  - un répertoire de services,
  - une interface typée pour chaque service (spécifique à un langage de programmation ou non),
- Utilisations:
  - Programmation distribuée,
  - Communication entre processus,
  - Services webs.

Suite d'actions RPC

1. le programme client appelle la souche client
   - appel local standard, les paramètres sont mis sur la pile de manière standard
2. la souche client conditionne (*marshalling*) les paramètres dans un message et fait un appel système pour envoyer le message.
3. le système d'exploitation du client envoie le message de la machine client à la machine serveur
4. le système d'exploitation du serveur reçoit le message et les passe à la souche serveur.
5. la souche serveur déballe (*unmarshalling*) les paramètres du message.
6. la souche serveur appelle la procédure sur le serveur avec les paramètres.
7. la réponse de la procédure suit le chemin inverse.

Schéma de RPC



Avantages et Inconvénients

- Avantages:
  - transparence pour le programme client (et le programme serveur),
  - distribution de la charge de calcul.
  - création de bibliothèques de services disponibles en lignes.
- Inconvénients:
  - client bloqué en attendant la réponse,
  - erreurs possibles dans le transfert d'informations,
  - les données sont copiées,
  - les communications peuvent être coûteuses,
  - sécurité.

Implémentation de RPC

- Description du service
  - avec langage de description (IDL, XML, ...),
  - ou avec des annotations de code source (par exemple RMI en Java),
  - sinon pas de description de service
    (reflexivité: manipuler les objets comme des objets locaux).
- Sérialisation:
  - action d'empaqueter les données pour les faire passer sur le réseau.
    - encodage en binaire ou texte,
  - facile pour les types de base (nombres, texte)
  - faisable pour les structures (tableaux, enregistrements)
  - difficile pour les objets, pointeurs, fonctions
  - problèmes de sécurité (authentification, confidentialité),
  - problèmes deversionnage.
    - différence entre client et serveur.
- Transport:
  - synchrone: TCP
  - synchrone: HTTP
  - asynchrone: SMTP

RPC à travers les âges

1976	Description de RPC dans RFC707
1981	Courier de Xerox
1984	RPC de Sun Network File System (puis Unix/Windows)
1997	Java RMI (API pour l'appel distant en Java)
1998	XML-RPC (appel encodé en XML transmis en HTTP)
1998	SOAP
2001	WSDL (Web Service Description Language)
2001	UDDI (Universal Description Discovery and Integration)
2005	JSON-RPC

- Implémentations: Courier, Sun RPC, RMI,
- Formats d'appel: XML-RPC, JSON-RPC
- Annuaire de Services Webs: UDDI

Exemple de JSON-RPC

- Interaction simple:

```
--> {"method": "echo", "params": ["Hello JSON-RPC"], "id": 1}
<-- {"result": "Hello JSON-RPC", "error": null, "id": 1}
```

- Application de clavardage:

```
...
--> {"method": "postMessage", "params": ["Bonjour tous!"], "id": 99}
<-- {"result": 1, "error": null, "id": 99}
<-- {"method": "handleMessage", "params": ["Jeanne", "asv ?"], "id": null}
<-- {"method": "handleMessage", "params": ["Bob", "allez ++"], "id": null}
--> {"method": "postMessage", "params": ["C bi1 Caramail"], "id": 101}
<-- {"method": "userLeft", "params": ["Bob"], "id": null}
<-- {"result": 1, "error": null, "id": 101}
...
```

Exemple de JSON-RPC

- Avec des paramètres:

```
{"method": "verifierCommande",
 "params": [
     {
         "nom": "Cordy",
         "prénom": "Annie",
         "id": 1234567890
     },
     {
         "quantite": 8,
         "nom": "saucisse",
         "montant": 45.50
     }
 ],
 "id": 1234}
```

Appels Distants

- styles d'appels distants:
  - RPC (historique, ou par le web):
    - programmation impérative
    - on simule une fonction locale
    - le code de la fonction se trouve sur le serveur
    - lors d'un appel: on doit empaqueter le nom de la fonction et les arguments de l'appel.
  - RMI (implémentation Java):
    - programmation objet
    - on simule un objet local
    - l'objet vit sur le serveur
    - on doit empaqueter un identifiant pour l'objet cible, le nom de la méthode et les arguments de l'appel.
- souvent on utilise un annuaire de services sur le serveur.
- on peut gérer des appels dans les deux sens:
  - on appelle `a.m(b)` avec `a` objet distant et `b` objet local
  - on peut sérialiser `b` et l'envoyer au serveur
    - et si une méthode de `b` appelle un objet `c` ?
    - il faut passer tout l'environnement (le contenu des références) de `b`
  - sinon on peut envoyer un moignon de `b`
    - si `a` appelle une méthode de `b`, l'appel sera effectué sur le client.

Conclusion

- Résumé:
  - *OCaml* propose une implémentation inefficace mais élégante (typage fort, évènements de première classe) de processus (*threads*) communiquant avec des canaux synchrones.
  - les futures sont ubiquitaires dans la programmation moderne.
  - les langages modernes proposent des primitives de haut-niveau pour mettre en place un système client-serveur internet.
  - la programmation d'appels distants insiste sur la notion de transparence.
- TD / TME:
  - TD: "Canaux Synchrones (II)" en *Go* et *OCaml*
  - TME: interfaces distantes en *Go*,
    - canaux synchrones,
    - client-serveur internet,
    - RPC.
- Séance prochaine:
  - Vérification de systèmes concurrents.