IA04 – Systèmes Multi-Agents

Chapitre 3 : Communication Multi-Agent

Sylvain Lagrue

sylvain.lagrue@hds.utc.fr

I- Introduction

Plan de l'intervention

1. Rappels multi-agent
2. Communication via channel
3. Communication à travers le réseau
4. Sérialisation et JSON
5. Les contextes

I- Introduction

Qu'est-ce qu'un agent ?

> *Entité autonome pouvant interagir sur son environnement dont il a une vision potentiellement partielle et agissant de manière rationnelle pour atteindre des buts qui lui sont propres.*

Quelques remarques et concepts clés

- Les agents peuvent être humains, économiques, logiciels, etc.
- *Autonomie* et *rationalité* relatives (de l'agent *réactif* à l'agent *cognitif*)
- Vision potentiellement partielle de son environnement ⇒ modélisation des *croyances* et de leur évolution en fonction des *perceptions* de l'agent
- *Buts* qui leur sont propres ⇒ modélisation de *préférences* et leur évolution en fonction des croyances
- D'autres agents peuvent faire partie de son environnement, ils composent alors un système *multi-agent*

I- Introduction

Et un système multi-agent ?

> *Système composé de plusieurs agents qui interagissent entre eux selon différentes règles définies.*

Quelques remarques et concepts clés

- *Plusieurs* = au moins 3 et jusqu'à plusieurs millions
- Agents *collaboratifs*/*compétitifs*
- Problème potentiel de *partage* et d'*accès* aux *ressources*
- Problème de *synchronisation*
- Idéalement sans point centralisateur

⇒ Nécessité de protocoles de *communication*

I- Introduction

Systèmes répartis/systèmes centralisés

- **Système centralisé** : tout est disponible sur une même machine et accessible par le programme (code, mémoire, processeur, périphériques, etc.)
- **Système réparti** : ensemble d'entités qui ne partagent pas toutes les ressources, mais qui peuvent communiquer entre elles ; d'un point de vue utilisateur, il s'agit d'une entité unique exécution concurrente

Avantage des systèmes répartis

- Partage de ressources (données, processeur, etc.)
- Concurrence
- Mise à l'échelle et montée en charge (*scalability*)
- Tolérance aux fautes (réseau, hardware, processus, etc.)
- Transparence (localisation, d'accès aux ressources, de l'hétérogénéité, de l'extension, de la migration de la réplication, etc.)

I- Introduction

Désavantages...

- Pannes et problèmes réseau
- Temps non commun (besoin de synchronisation)
- Hétérogénéité (du matériel, des OS, des langages, des plateformes, etc.)
- Accès concurrentiel aux ressources

II- Communication en Go via channel

Rappels sur les *channels*

- Les channels permettent la communication entre les différents processus
- Communication *asynchrone* et *bidirectionnelle*
  - Les envois et les réceptions sont bloquants
  - On peut *bufferiser* les communications (lors de la création avec `make`)
- On peut passer n'importe quelle structure via un channel
- Les messages sont sérialisés
- On peut gérer plusieurs channels avec le mot-clef `select`

II- Communication en Go via channel

Exemple (1)

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c := make(chan int)
    go func() {
        fmt.Println("je commence l'envoi")
        c <- 12
        fmt.Println("j'ai fini l'envoi !")
    }()
    time.Sleep(3 * time.Second)
    fmt.Println("je commence la reception")
    <-c
    fmt.Println("j'ai fini la réception !")
    fmt.Scanln()
}
```

II- Communication en Go via channel

Exemple (2)

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c := make(chan string, 1)
    go func() {
        c <- "IA04, c'est bon, mangez-en !"
        // non bloquant
        // la goroutine finit dès l'envoi fait
    }()

    time.Sleep(time.Second)
    msg := <-c
    // bloquant tant qu'on n'a pas reçu le message de la goroutine

    fmt.Println(msg)
}
```

Résultat

```
IA04, c'est bon, mangez-en !
```

II- Communication en Go via channel

Un cas d'étude : ping-pong

- un agent ping envoie un message `ping id` à un agent pong qui renvoie `pong id`
- l'agent pong est capable de gérer plusieurs messages simultanément

Exemple d'échanges

```
agent "ponger" has received "ping" from "pinger n°2"
agent "ponger" has received "ping" from "pinger n°4"
agent "ponger" has received "ping" from "pinger n°1"
agent "pinger n°2" has received: "pong"
agent "pinger n°4" has received: "pong"
agent "pinger n°1" has received: "pong"
agent "ponger" has received "ping" from "pinger n°3"
agent "ponger" has received "ping" from "pinger n°0"
agent "pinger n°3" has received: "pong"
agent "pinger n°0" has received: "pong"
agent "ponger" has received "ping" from "pinger n°1"
agent "ponger" has received "ping" from "pinger n°0"
agent "ponger" has received "ping" from "pinger n°3"
agent "pinger n°1" has received: "pong"
agent "ponger" has received "ping" from "pinger n°4"
agent "ponger" has received "ping" from "pinger n°2"
agent "pinger n°3" has received: "pong"
agent "pinger n°4" has received: "pong"
```

II- Communication en Go via channel

Interface **Agent**

```
type Agent interface {
    Start()
}
```

Constantes

```
const PingString = "ping"
const PongString = "pong"
```

Type **Request**

```
type Request struct {
    senderID string
    req      string
    c        chan string
}
```

II- Communication en Go via channel

Type **Pinger**

```
type PingAgent struct {
    ID   string
    cin  chan string
    cout chan Request
}


func NewPingAgent(id string, cout chan Request) *PingAgent {
    cin := make(chan string)
    return &PingAgent{id, cin, cout}
}


func (ag *PingAgent) Start() {
    go func() {
        for {
            ag.cout <- Request{ag.ID, PingString, ag.cin}
            answer := <-ag.cin
            fmt.Printf("agent %q has received: %q\n", ag.ID, answer)
            time.Sleep(time.Second)
        }
    }()
}
```

II- Communication en Go via channel

Type **Ponger**

```
type PongAgent struct {
    ID string
    c  chan Request
}


func NewPongAgent(id string, c chan Request) *PongAgent {
    return &PongAgent{id, c}
}


func (ag *PongAgent) Start() {
    go func() {
        for {
            req := <-ag.c
            fmt.Printf("agent %q has received %q from %q\n", ag.ID,
                       req.req, req.senderID)
            go ag.handlePing(req) // et si on enlève go ?
        }
    }()
}


func (ag PongAgent) handlePing(req Request) {
    req.c <- PongString
}
```

II- Communication en Go via channel

Programme principal

```
func main() {
    c := make(chan Request)

    ponger := NewPongAgent("ponger", c)
    ponger.Start()

    for i := 0; i < 5; i++ {
        id := fmt.Sprintf("pinger n°%d", i)
        pinger := NewPingAgent(id, c)
        pinger.Start()
    }

    time.Sleep(time.Minute)
}
```

II- Communication en Go via channel

Résultat

```
$ go run .
agent "ponger" has received "ping" from "pinger n°4"
agent "ponger" has received "ping" from "pinger n°0"
agent "pinger n°0" has received: "pong"
agent "ponger" has received "ping" from "pinger n°1"
agent "pinger n°1" has received: "pong"
agent "pinger n°4" has received: "pong"
agent "ponger" has received "ping" from "pinger n°3"
agent "ponger" has received "ping" from "pinger n°2"
agent "pinger n°2" has received: "pong"
agent "pinger n°3" has received: "pong"
agent "ponger" has received "ping" from "pinger n°4"
agent "ponger" has received "ping" from "pinger n°3"
agent "ponger" has received "ping" from "pinger n°0"
agent "ponger" has received "ping" from "pinger n°1"
agent "ponger" has received "ping" from "pinger n°2"
agent "pinger n°0" has received: "pong"
agent "pinger n°4" has received: "pong"
agent "pinger n°3" has received: "pong"
agent "pinger n°1" has received: "pong"
agent "pinger n°2" has received: "pong"
agent "ponger" has received "ping" from "pinger n°1"
agent "pinger n°1" has received: "pong"
agent "ponger" has received "ping" from "pinger n°0"
agent "pinger n°0" has received: "pong"
...
```

III- Communication en Go à travers le réseau

Quelques modes de communication usuels pour la communication réseau

- Mode client-serveur
  - **sockets TCP**/sockets UDP/sockets Unix
  - Web sockets
- Remote procedure calls (RPC)
  - sun-RPC
  - XML-RPC/JSON-RPC
  - gRPC
- Objets répartis
  - Java RMI
  - CORBA
- Services Web/ Architectures orientées service SOA
  - SOAP
  - **REST**
  - Web Workers

III- Communication en Go à travers le réseau

Le package `net` et les sockets

Interface générique pour les connexions réseaux : Conn

```
type Conn interface {
    Read(b []byte) (n int, err error)
    Close() error
    LocalAddr() Addr
    RemoteAddr() Addr
    SetDeadline(t time.Time) error
    SetReadDeadline(t time.Time) error
    SetWriteDeadline(t time.Time) error
}
```

```
func Dial(network, address string) (Conn, error)
```

avec network à choisir parmi `tcp`, `tcp4` (IPv4-only), `tcp6` (IPv6-only), `udp`, `udp4` (IPv4-only), `udp6` (IPv6-only), `ip`, `ip4` (IPv4-only), `ip6` (IPv6-only), `unix`, `unixgram` et `unixpacket`

III- Communication en Go à travers le réseau

Exemple

```
Dial("tcp", "12.12.12.12:1212")
Dial("tcp", "syl.lagrue.ninja:http")
Dial("tcp", "[2001:db8::1]:80")
Dial("tcp", ":80")
```

Autres fonctions

```
func DialTimeout(network, address string, timeout time.Duration) (Conn, error)
```

```
func Pipe() (Conn, Conn)
```

III- Communication en Go à travers le réseau

Exemple : clients ping

```
func ping(i int) {
    var raddr = net.TCPAddr{
        IP:   net.IPv4(127, 0, 0, 1),
        Port: 8888,
    }

    // conn, err := net.Dial("tcp", "localhost:8888")
    // conn, err := net.DialTimeout("tcp", "localhost:8888", 5*time.Second)
    conn, err := net.DialTCP("tcp", nil, &raddr)

    if err != nil {
        fmt.Println("connect error:", err)
    }
    defer conn.Close()
    buffer := make([]byte, 1024)
    conn.Write([]byte("ping " + strconv.Itoa(i)))

    s, err := conn.Read(buffer)
    if err != nil {
        fmt.Println("read error:", err)
    }

    fmt.Printf("message reçu: '%s' (%d bytes)\n", string(buffer), s)
}
```

III- Communication en Go à travers le réseau

```
func main() {
    for i := 0; i < 25; i++ {
        go ping(i)
    }
    time.Sleep(time.Second)
}
```

III- Communication en Go à travers le réseau

Exemple : serveur pong

```
func HandleRequest(conn net.Conn) {
    buf := make([]byte, 1024)

    len, err := conn.Read(buf)
    if err != nil {
        fmt.Println("Error reading:", err.Error())
    }
    fmt.Print("message reçu: '", string(buf), "' (", len, " bytes)\n")

    conn.Write([]byte("pong"))
    conn.Close()
}
```

III- Communication en Go à travers le réseau

```
func main() {
    var addr net.TCPAddr

    addr.IP = net.IPv4(127, 0, 0, 1)
    addr.Port = 8888

    //l, err := net.Listen("tcp", "localhost:8888")
    l, err := net.ListenTCP("tcp", &addr)
    if err != nil {
        fmt.Println("Error Listen: ", err.Error())
        os.Exit(1)
    }
    defer l.Close()

    for {
        conn, err := l.Accept() // bloquant !
        if err != nil {
            fmt.Println("Error accepting: ", err.Error())
            os.Exit(1)
        }
        go HandleRequest(conn)
    }
}
```

III- Communication en Go à travers le réseau

Services web

Créer un serveur web

```go
package main

import (
    "log"
    "net/http"
)

func main() {
    // create file server handler
    fs := http.FileServer(http.Dir("./"))

    // start HTTP server with `fs` as the default handler
    log.Fatal(http.ListenAndServe(":8080", fs))
}
```

III- Communication en Go à travers le réseau

Un Web service/serveur REST en Go

Principe

- *Representational state transfer*
- On utilise `http(s)` (1, 1.1, voire 2.0) pour fournir un service (en mode client-serveur)
- On utilise la sémantique des verbes `http` : `GET`, `POST`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`, `CONNECT`, `TRACE`, `PATCH`, etc.
- On utilise les codes d'erreur `http` pour donner le résultat : `2xx` success, `4xx` client error, `5xx` server error, etc.
- Les URLs permettent d'accéder aux services, voire de donner certains arguments (ex. : `http://www.exemple.fr/list/produit/12`)
- Sans état (le service n'est pas censé garder trace des échanges)
- Les services sont plus ou moins *RestFul*...

III- Communication en Go à travers le réseau

Exemple simplissime de création d'un serveur REST en Go

```
package main

import (
    "fmt"
    "html"
    "log"
    "net/http"
    "time"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("/hello", hello)

    s := &http.Server{
        Addr:           ":12000",
        Handler:        mux,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20}

    log.Println("Listening on localhost:8080")
    log.Fatal(s.ListenAndServe())
}
```

III- Communication en Go à travers le réseau

Remarques

- On peut faire passer une méthode à `HandleFunc` et ainsi récupérer différentes informations du serveur...

```
package main

import (
    "fmt"
    "html"
    "log"
    "net/http"
    "time"
)

type MyServer struct {
    RequestCount int
}

func (s *MyServer) hello(w http.ResponseWriter, r *http.Request) {
    s.RequestCount++
    fmt.Fprintf(w, "Hello, %q (%d)",
                html.EscapeString(r.URL.Path),
                s.RequestCount)
}
```

III- Communication en Go à travers le réseau

```
func main() {
    ms := new(MyServer)

    mux := http.NewServeMux()
    mux.HandleFunc("/hello", ms.hello)

    s := &http.Server{
        Addr:           ":12000",
        Handler:        mux,
        ReadTimeout:    10 * time.Second,
        WriteTimeout:   10 * time.Second,
        MaxHeaderBytes: 1 << 20}

    log.Println("Listening on localhost:8080")
    log.Fatal(s.ListenAndServe())
}
```

- Il existe des packages tiers permettant de faire des choses plus fines sur les arguments dans les URL (ex. https://github.com/gorilla/mux)

III- Communication en Go à travers le réseau

Et côté client ?

Version simplissime

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form",
                           url.Values{"key": {"Value"}, "id": {"123"}})
```

Attention à la fermeture de `body` avant de le lire...

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := io.ReadAll(resp.Body)
// ...
```

Pour aller plus loin...

Le type `Request` : https://pkg.go.dev/net/http#Request

IV- Sérialisation via JSON

Présentation

- *JavaScript Object Notation*
- Format de sérialisation issu de JavaScript (mais pas tout à fait une extension)
- Créé en 2002 par Douglas Crockford
- RFC 72159
- Type MIME : application/json
- Mode texte (différent par exemple de `pkg/encoding/gob`)
- Description exhaustive : http://json.org/

Avantages

- un peu plus lisible "humainement" et plus compact que XML
- "déssérialisable" quasi directement pour JavaScript, Python et Perl

D'autres formats en vogue

- BSON, TOML, YAML, ...

IV- Sérialisation via JSON

Format

- utf-8 (sauf contre-indication)

Types

- Objet : `{"clé1": val1, "clé2" : val2}`
- Chaîne : `"ceci est une chaîne"`
- Numériques (réels) : `12 -12.4 1e-12`
- Booléens : `true false`
- Tableaux : `[12, "toto", {x:1 , y : 2}, [1, 2]]`
- Valeur nulle : `null`

IV- Sérialisation via JSON

package `encoding/json`

- Encoding : `func Marshal(v interface{}) ([]byte, error)`
- Decoding : `func Unmarshal(data []byte, v interface{}) error`

Exemple

```
type Person struct {
   Name string
   Age  int
   Job  string
   id   string // minuscule : attribut non-accessible !
}
```

IV- Sérialisation via JSON

Codage

```
p := Person{"Goodman", 46, "lawyer",
            "(505) 503-445"}
fmt.Println(p)  // {Goodman 46 lawyer (505) 503-445}

buffer, err := json.Marshal(p) //
if err != nil {
    log.Fatal(err)
}

fmt.Println("msg:", string(buffer)) // {"Name":"Goodman","Age":46,"Job":"lawyer"}
```

Décodage

```
msg := `{"Name": "Heisenberg", "Age": 51, "Job": "chemist"}`
msgb := []byte(msg)

fmt.Println("msg:", msg)

var p Person
err := json.Unmarshal(msgb, &p)
if err != nil {
    log.Fatal(err)
}

fmt.Println("Person:", p)  // Person: {Heisenberg 51 chemist }
```

IV- Sérialisation via JSON

Conversion de types

- `bool` ↔ JSON booleans
- `float64` ↔ JSON numbers
- `string` ↔ JSON strings
- `[]interface{}` ↔ JSON arrays
- `map[string]interface{}` ↔ JSON objects
- `nil` ↔ JSON null

Remarque : Certaines correspondances ne sont pas possibles (ex. : nom de clefs impossible pour une structure).

IV- Sérialisation via JSON

Renommage des attributs

Les attributs des objets JSON ne correspondent pas forcément aux attributs des structures

Exemple

```
{
   "nom": "Fring",
   "Age": 58,
   "actual-job": "Fast food restaurant manager"
}
```

Problème

```
{ 58 }
```

Solution

```
type Person struct {
   Name string `json:"nom"`
   Age  int
   Job  string `json:"actual-job"`
   id   string
}
```

IV- Sérialisation via JSON

Autres labels possibles

```
// Field is ignored by this package.
`json:"-"`

// Field appears in JSON as key "myName".
`json:"myName"`

// idem + the field is omitted from the object if its value is empty,
`json:"myName,omitempty"`

// Field appears in JSON as key "Field" (the default), but
// the field is skipped if empty.
Field int `json:",omitempty"`

// int64 to string
Int64String int64 `json:",string"`
```

IV- Sérialisation via JSON

Cas des slices et des tableaux vides

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
)

func main() {
    var sl []int
    fmt.Println("initial:", sl)

    buffer, err := json.Marshal(sl)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("msg:", string(buffer))
}
```

```
initial: []
msg: null
```

IV- Sérialisation via JSON

Cas des slices et des tableaux vides (code corrigé)

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"
)

func main() {
    var sl []int
    sl = make([]int, 0)
    fmt.Println("initial:", sl)

    buffer, err := json.Marshal(sl)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Println("msg:", string(buffer))
}
```

```
initial: []
msg: []
```

V- Les contextes

Objectif

- Gérer les *deadlines*, les signaux d'annulation et d'autres valeurs liées aux demandes entre les différents traitements concurrents
- La chaîne d'appels de fonction doit propager le contexte, en le remplaçant éventuellement par un contexte dérivé créé à l'aide de `WithCancel`, `WithDeadline`, `WithTimeout` ou `WithValue`
- Lorsqu'un contexte est annulé, tous les contextes qui en sont dérivés sont également annulés
- De nombreuses fonctions de l'API `go` ont été étendues aux contextes
- On n'utilise pas de Context `nil`, on préférera donne `context.TODO`

Fonctions de création de contextes

```
func Background() Context
func TODO() Context
func WithValue(parent Context, key, val interface{}) Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

V- Les contextes

Exemple recherche parallèle dans un tableau

```
func find(tab []int, val int) int {
    for i, v := range tab {
        if val == v {
            return i
        }
    }
    return -1
}
```

V- Les contextes

Version parallèle avec `context`

```
func findCtx(ctx context.Context, tab []int, res chan int, val int, start int) {
    for i, v := range tab {
        select {
        case <-ctx.Done():
            fmt.Println("Cancel...")
            return // pour ne pas avoir de fuite de mémoire
        default:
            if val == v {
                fmt.Println("trouvé:", start+i, v)
                res <- start + i
                return
            }
        }
    }
    return -1
}
```

V- Les contextes

Fonction principale

```
func main() {
    const size = 512 * (1 << 20)
    var tab [size]int
    val := 384 * (1 << 20)
    initTab(tab[:])

    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    res := make(chan int)

    index := -1
    n := 4
    step := size / n

    for i := 0; i < n; i++ {
        go findCtx(ctx, tab[i*step:(i+1)*step], res, val, i*step)
    }

    for i := 0; i < n; i++ {
        index = <-res
        if index >= 0 {
            close(res)
            cancel()
            break
        }
    }
    fmt.Println("index:", index)
    fmt.Println("val:", tab[index])
}
```

VI. Communication et gestion des dates

Remarques liminaires

- Besoin de dates dans les échanges pour la synchronisation, pour poser des jalons, etc.
- La communication se faisant par Internet, on a besoin de gérer les fuseaux horaires, l'heure d'été, etc.
- Il existe 2 types sur un OS, les horloges monotones (dont le temps augmente toujours) et les horloges murales (*wall clock*) qui peuvent bouger lors de synchronisation (par exemple via un serveur NTP - *Network Time Protocol*)
- De nombreuses méthodes ont été proposées *timestamp* Unix (temps en s depuis le 01/01/1970 00:00:00 UTC), RFC 5322 (mail), ISO 8601, **RFC 3339**

VI. Communication et gestion des dates

La RFC 3339 (https://datatracker.ietf.org/doc/html/rfc3339)

- Plus d'info sur https://www.bortzmeyer.org/3339.html
- Sous-ensemble de la norme ISO 8601 (non accessible)
- Format compréhensible par un être humain
- Calendrier grégorien de notre ère (pas d'année négative)
- Suppose que le décalage entre l'horloge locale et UTC est connu
- Gère la localisation et l'heure d'été via des décalages (+00, pas "UTC")
- Les dates sont en format "numériques" comparables lexicographiquement
- Z étant l'équivalent de +00:00

Exemples

- `2023-10-09T23:05:08+02:00`
- `1985-04-12T23:20:50.52Z`
- `1990-12-31T23:59:60Z`
- `1937-01-01T12:00:27.87+00:20`

VI. Communication et gestion des dates

En Go

- Le package `time`
- Documentation : https://pkg.go.dev/time
- Le type `time.duration`
- Le type `time.time`

Remarque : Go gère les 2 types d'horloge, utilisant sur celle utile pour la fonction considérée (par exemple monotone sur les opérations d'ajout, les 2 simultanément sur un constructeur, etc.)

VI. Communication et gestion des dates

Le type `time.Time`

Constructeurs

```
func Date(year int, month Month, day, hour, min, sec, nsec int
                                                    , loc *Location) Time
func Now() Time

func Parse(layout, value string) (Time, error)
func ParseInLocation(layout, value string, loc *Location) (Time, error)

func Unix(sec int64, nsec int64) Time
func UnixMicro(usec int64) Time
func UnixMilli(msec int64) Time
```

VI. Communication et gestion des dates

Opérations sur les dates

```
// Ajout/supression
func (t Time) Add(d Duration) Time
func (t Time) AddDate(years int, months int, days int) Time

// Comparaison de dates
func (t Time) After(u Time) bool
func (t Time) Before(u Time) bool
func (t Time) Compare(u Time) int
func (t Time) Equal(u Time) bool

// Accesseurs
func (t Time) Date() (year int, month Month, day int)
func (t Time) Day() int
func (t Time) Hour() int
func (t Time) Minute() int
func (t Time) Month() Month
func (t Time) Nanosecond() int
func (t Time) Second() int
func (t Time) Year() int
...

// Localisation
func (t Time) Local() Time
func (t Time) Location() *Location
func (t Time) UTC() Time
...
```

VI. Communication et gestion des dates

Formatage de dates

```
func (t Time) Format(layout string) string
```

Les différents formats

```
const (
    Layout      = "01/02 03:04:05PM '06 -0700" // The reference time,
                                               // in numerical order.
    ANSIC       = "Mon Jan _2 15:04:05 2006"
    UnixDate    = "Mon Jan _2 15:04:05 MST 2006"
    RubyDate    = "Mon Jan 02 15:04:05 -0700 2006"
    RFC822      = "02 Jan 06 15:04 MST"
    RFC822Z     = "02 Jan 06 15:04 -0700" // RFC822 with numeric zone
    RFC850      = "Monday, 02-Jan-06 15:04:05 MST"
    RFC1123     = "Mon, 02 Jan 2006 15:04:05 MST"
    RFC1123Z    = "Mon, 02 Jan 2006 15:04:05 -0700" // RFC1123 with numeric zone
    RFC3339     = "2006-01-02T15:04:05Z07:00"
    RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
    Kitchen     = "3:04PM"

    // Handy time stamps.
    Stamp      = "Jan _2 15:04:05"
    DateTime   = "2006-01-02 15:04:05"
    DateOnly   = "2006-01-02"
    TimeOnly   = "15:04:05"
    ...
)
```

VI. Communication et gestion des dates

Exemple

```go
package main

import (
    "fmt"
    "time"
)

func printTime(t time.Time) {
    fmt.Println("Format Go:", t)
    fmt.Println("Format Layout:", t.Format(time.Layout))
    fmt.Println("Format Unix:", t.Format(time.UnixDate))
    fmt.Println("Format UTC:", t.UTC().Format(time.UnixDate))
    fmt.Println("Format RFC339:", t.Format(time.RFC3339))
}

func main() {
    fmt.Println("Initial Time")
    t := time.Now()
    printTime(t)
    fmt.Println("\nDeadline")
    deadline := t.Add(2 * time.Second)
    printTime(deadline)
}
```

VI. Communication et gestion des dates

Exemple (sortie)

```
Initial Time
Format Go: 2023-10-09 23:05:08.255936 +0200 CEST m=+0.000411251
Format Layout: 10/09 11:05:08PM '23 +0200
Format Unix: Mon Oct  9 23:05:08 CEST 2023
Format UTC: Mon Oct  9 21:05:08 UTC 2023
Format RFC339: 2023-10-09T23:05:08+02:00

Deadline
Format Go: 2023-10-09 23:05:10.255936 +0200 CEST m=+2.000411251
Format Layout: 10/09 11:05:10PM '23 +0200
Format Unix: Mon Oct  9 23:05:10 CEST 2023
Format UTC: Mon Oct  9 21:05:10 UTC 2023
Format RFC339: 2023-10-09T23:05:10+02:00
```

VI. Communication et gestion des dates

Date et JSON

Convertir en JSON vers et depuis le format RFC3339

```
func (t Time) MarshalJSON() ([]byte, error)
func (t *Time) UnmarshalJSON(data []byte) error
```

⇒ À utiliser dans les web services !

Annexe (pour aller plus loin)

Communication FIPA

Il existe une normalisation pour la communication entre agents :

- http://www.fipa.org/specs/fipa00061/SC00061G.html
- https://en.wikipedia.org/wiki/Agent_Communications_Language

Annexe (pour aller plus loin)

À propos...

Information	Valeur
Auteur	Sylvain Lagrue (sylvain.lagrue@utc.fr)
Licence	Creative Common CC BY-SA 4.0
Version document	1.2.0