Open Telephony Platform

Brique ROSE

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

Erlang et OTP

- La plupart des programmes Erlang utilisent des processus
- La plupart des processus remplissent une fonction telle que :
  - Serveurs de fonctionnalité
  - Machine à états finis
  - Gestionnaire d'événements
  - Supervision des autres processus
  - Démarrage, fin et mise à jour de l'application
- OTP (Open Telephony Platform) fournit ces services sous formes de **comportements** pour Erlang

État et enregistrement

- Chaque module possède un état
- L'état est en général un **enregistrement**

```
% Enregistrement nomme state
-record (state, {free_blocks = 300, color = red,
                 temperature}).

% Acces ^^e0 un champ
f (S) -> S#state.free_blocks.

% Changement d'un champ
inc (S) -> S#state{free_blocks =
                 S#state.free_blocks + 1}.
```

Comportements

- Un comportement correspond à un module, et est déclaré en tête de fichier :

```
-module (disk_alloc).
-behaviour (gen_server).
```

- Ce module fournit un ensemble de fonctions qui seront appelées par le comportement (*callbacks*) et qui renvoient un statut, un nouvel état et éventuellement des informations pour l'appelant :

```
handle_call ({allocate, Blocks}, From, State)
  when Blocks <= State#state.free_blocks ->
    {reply, ok,
     State#state{free_blocks =
       State#state.free_blocks - Blocks}};
```

Réponses

- Plusieurs types de réponses peuvent être renvoyées par un *callback*. Exemple de `handle_call` dans le comportement `gen_server` :
  - `{reply, Reply, State}`
  - `{reply, Reply, State, Timeout}`
  - `{noreply, State}`
  - `{noreply, State, Timeout}`
  - `{stop, Reason, Reply, State}`
  - `{stop, Reason, State}`

Comportement `gen_server`

- But : faire un serveur générique (question/réponse) appelable à distance, avec gestion des *timeouts*.
- Fonctions :
  - Initialisation
  - Gestion d'un appel synchrone entrant
  - Gestion d'un appel asynchrone entrant
  - Gestion d'une information entrante
  - Gestion de la terminaison
  - Gestion du changement de code

Autres fonctions

- `gen_server` exporte des fonctions :
  - Lancement (avec nom local ou nom global)
  - Appel synchrone, sur un ou plusieurs serveurs
  - Appel asynchrone, sur un ou plusieurs serveurs
  - Envoi d'une information
  - Demande de terminaison
  - Demande de changement de code

Comportement gen_event

- But : faire un gestionnaire d'événements génériques
- Utilisations :
  - enregistrement des erreurs
  - gestion d'alarmes
  - enregistrement d'appels de fonctions
  - debugging
  - gestion d'équipement

Définitions

- **Événement :** quelque chose qui a lieu
- **Catégorie d'événements :** type ou classe d'un événement
- **Gestionnaire d'événements :** processus chargé de la coordination des événements de la même catégorie
- **Notification :** acte d'informer un gestionnaire qu'un événement s'est produit
- **Traiteur d'événements :** (*event handler*) module qui exporte des fonctions permettant de traiter des événements d'une certaine catégorie

Principe

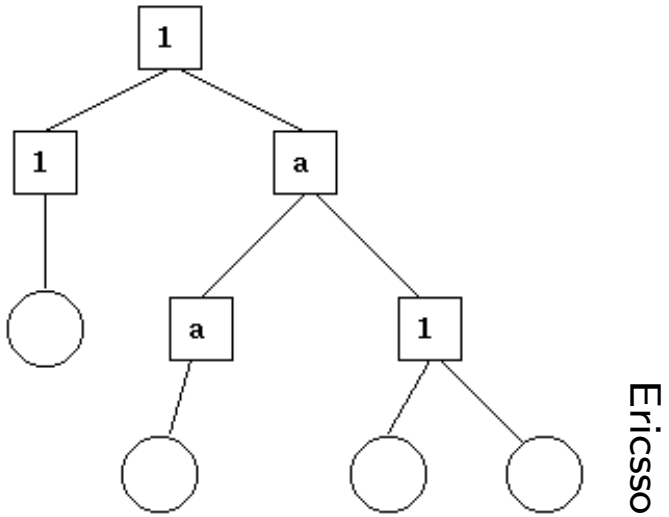
- Un gestionnaire d'événements est lancé
- Un ou plusieurs traiteurs d'événéments sont lancés
- Les événements sont notifiés au gestionnaire d'événements
- Le gestionnaire renvoie les événements aux traiteurs
- Les traiteurs peuvent prendre les actions nécessaires, modifier leur état interne, demander à être supprimés ou demander à être remplacé par un autre traiteur

Comportement `gen_fsm`

- But : faire un gestionnaire de machine à états finis générique
- Principe :
  - si la machine est dans l'état S
  - si l'événement E a lieu
  - la machine effectue les actions $A(S, E)$
  - la machine passe dans l'état $S'(S, E)$

Supervision

- Problèmes :
  - un programme informatique peut comporter des erreurs
  - dans un système réparti, des machines ou des réseaux peuvent tomber en panne
- Erlang/OTP implémente un arbre de supervision



Ericsson

Superviseurs

Ils supervisent des travailleurs ou des superviseurs, et ont plusieurs stratégies :

- **Un pour tous :** (*one for all*) si un processus supervisé meure, ils sont tous tués (récursivement) puis éventuellement relancés
- **Chacun pour soi :** (*one for one*) si un processus supervisé meure, il est éventuellement relancé

Travailleurs

Les travailleurs (ou superviseurs fils) sont de plusieurs types :

- **permanent :** il est toujours relancé s'il meure (processus indispensable)
- **éphémère :** (*transient*) il est relancé s'il meure de façon anormale (processus temporaire dont le résultat est indispensable)
- **temporaire :** (*temporary*) il n'est jamais relancé

Avalanche

- Si à cause d'une erreur de programmation un processus meure, il se peut que cela recommence à chaque fois qu'il est relancé
- Les processus ont une fréquence maximum de relance
- Si un processus ne peut pas être relancé, son superviseur meure
- Le superviseur du superviseur prend le relai

OTP et pont de supervision

- Tous les comportements OTP supportent la supervision :
  - ils implémentent les bons *callbacks* pour terminer et être redémarrés
- Il n'est pas toujours possible d'utiliser un comportement OTP :
  - code Erlang existant
  - interface avec l'extérieur
  - performances
- Un **pont de supervision** est possible, permettant d'insérer un processus Erlang de supervision dans la chaîne

Autres points d'OTP

OTP traite aussi :

- les applications
  - Regroupement des modules en application
  - Inclusion d'applications
  - Dépendances entre applications
- la mise à jour du code pendant l'exécution
  - Gestion des dépendances entre applications et modules
  - Migration des données de l'ancien au nouveau format
  - Gestion des fautes lors de la mise à jour

Conclusions

- Il est important d'utiliser OTP lorsque c'est possible
  - factorisation du code dans les `gen_xxx`
  - machines à état claires et facile à comprendre
  - insertion dans l'arbre de supervision
- Il faut superviser ses processus lorsqu'ils sont critiques