Threads

Chantal Keller

Threads ?

Qu'est-ce ? But ? Intérêt ? Difficultés ?

Threads ?

Qu'est-ce ? But ? Intérêt ? Difficultés ?

↪ démo

Importance des threads sous Android

L'interface doit toujours être (ré)active !

- pas de calculs longs, bloquants ou infinis
- pas d'accès aux ressources coûteuses

Exemples :

- calcul long : application demandant des calculs coûteux
- calcul bloquant : attente d'une entrée de l'utilisateur, attente de connexion
- calcul infini : écoute sur un réseau
- ressources coûteuses : réseau, internet

Threads sous Android

Thread principal, ou *UI thread* :

- le thread dans lequel s'exécute l'application au démarrage
- gère tout ce qui est interface
- **seul thread à avoir accès à l'interface**
- possibilité de faire des calculs peu coûteux (ex : addition, . . .)

Autres threads, ou *worker threads* :

- au programmeur de les gérer
- effectuent les calculs coûteux, les accès aux ressources coûteuses (dont le réseau)
- **aucun accès à l'interface** $\Rightarrow$ communication avec le UI thread

But atteint

Fluidité de l'interface :

- un thread pour l'interface, n'attendant pas de résultats ou de ressources coûteux
- d'autres threads en tâches de fond pour les résultats et ressources coûteuses

↪ répartition des tâches **obligatoire**

Autres applications :

- naturellement, on peut utiliser les threads pour le parallélisme
- efficacité, mais pas d'obligation

Plan

La classe `Thread`

Principe :

- faire une classe héritant de `Thread` (ex : `AckermannThread`)
- redéfinir la méthode `run`, qui sera exécutée en tâche de fond
- lancer cette tâche
  (ex : `new AckermannThread(...).start()`
  (**pas** d'appel direct à `run`))

Pour mettre à jour l'interface :

- "poster" des messages au thread principal

Démo : lancement d'un thread

```
public class Ackermann extends AppCompatActivity
                        implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }

    @Override
    public void onClick(View view) {
        ...
        new AckermannThread(g0, d0).start();
        ...
    }


    private class CallServerThread extends Thread {
        private final int arg1, arg2;

        public AckermannThread(int a1, int a2) {
            arg1 = a1; arg2 = a2;
        }

        @Override
        public void run () {
            ...
        }
    }
}
```

Mise à jour de l'interface

Rappel : seul le thread graphique peut modifier l'interface

Solution : communication via un `Handler`

Démo : mise à jour de l'interface

⋮

```
private class AckermannThread extends Thread {
    private final int arg1, arg2;


    public AckermannThread(int a1, int a2) {
        arg1 = a1; arg2 = a2;
    }

    @Override
    public void run () {
        ...




        ...
    }
}
```

Démo : mise à jour de l'interface

⋮

```
private class AckermannThread extends Thread {
    private final int arg1, arg2;

    public AckermannThread(int a1, int a2) {
        arg1 = a1; arg2 = a2;
    }

    @Override
    public void run () {
        ...

                resDevice.setText(String.valueOf(r));

        ...
    }
}
```

Démo : mise à jour de l'interface

⋮

```
private class AckermannThread extends Thread {
    private final int arg1, arg2;
    final Handler handler = new Handler() ;

    public AckermannThread(int a1, int a2) {
        arg1 = a1; arg2 = a2;
    }

    @Override
    public void run () {
        ...


                resDevice.setText(String.valueOf(r));


        ...
    }
}
```

Démo : mise à jour de l'interface

⋮

```
private class AckermannThread extends Thread {
    private final int arg1, arg2;
    final Handler handler = new Handler() ;

    public AckermannThread(int a1, int a2) {
        arg1 = a1; arg2 = a2;
    }

    @Override
    public void run () {
        ...
        handler.post(new Runnable() {
            @Override
            public void run() {
                resDevice.setText(String.valueOf(r));
            }
        });
        ...
    }
}
```

Interruption d'un thread

Principe :

- on signale au thread de s'interrompre :
  `ackermannThread.interrupt()`
- à lui de gérer cela pour s'arrêter :
  `if (interrupted()) ...`
  `while (!interrupted()) ...`

Cas d'utilisation :

- calcul trop long
- serveur tournant en continu
- possibilité d'échec
- ...

Conclusion

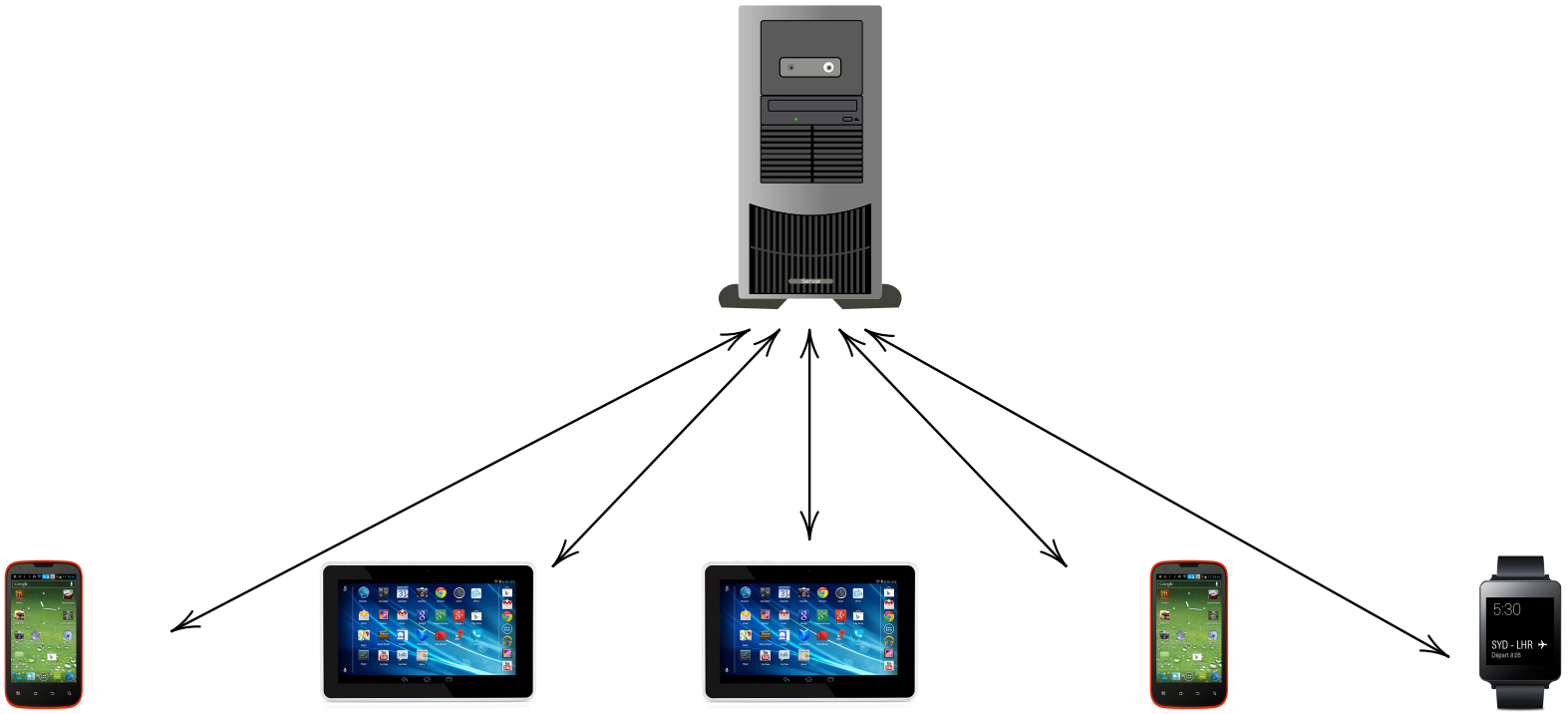
Code complet sur Moodle

Utilisation

- obligatoire : tâches longues ou faisant appel à des ressources coûteuses
- pour les performances : parallélisme (attention à la synchronisation)

Plan

Principe



Protocole

Pour pouvoir communiquer :

- se mettre d'accord sur le *protocole de communication*
- généralement, imposé par le serveur
- un client ne respectant pas le protocole ne pourra pas accéder au service
- ce qui n'est pas prévu dans le protocole ne pourra être fait

Exemple : protocole de chat

```
Welcome John
Welcome Mary
John: Salut !
Mary: Comment ça va ?
John: Super !
Welcome Bob
Mary: Salut Bob !
Bob: Hey !
John: Je dois y aller ++
Bye bye John
```

Exemple : côté serveur

Protocole :

- le serveur attend **indéfiniment** des connexions à un endroit prédéfini (dépendant de l'infrastructure)
- pour chaque connexion, il attend **indéfiniment** les messages suivants :

Message reçu	Action	Message envoyé à tous clients
LOGIN [login]	Authentification	Welcome [login]
SEND [message]	Discussion	[login] : [message]
LOGOUT	Déconnexion	Bye bye [login]

Exemple : côté client

Le client doit :

1. se connecter à l'endroit prédéfini
2. envoyer un message "LOGIN [login]"
3. envoyer autant de messages "SEND [message]" que voulu
4. envoyer un message "LOGOUT"

et **en permanence** afficher les messages venant du serveur

Ils sont partout !

Threads pour :

1. client : se connecter au serveur
2. client : gérer l'affichage des messages du serveur
3. serveur : attendre des connexions
4. serveur : pour chaque client en parallèle, gérer la réception et l'envoi de messages

En TP : application client/serveur via des *sockets*

Le serveur :

- possède une adresse
- écoute sur un port donné (choisi par le protocole)

Les clients :

- établissent une connexion sur ce port
- échangent des données avec le serveur *via* cette connexion

↪ analogie : raccordement de tuyaux

En Java et Android

Connexion :

- *sockets* et *sockets* serveur
- avec un flux sortant et un flux entrant

Permissions :

- nécessite la permission d'accéder au réseau
- déclaration des permissions dans le manifeste : en dehors des balises <application>

```
<uses-permission
  android:name="android.permission.INTERNET" />
<uses-permission
  android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Les *sockets*

La classe `Socket` :

- connexion à une *socket* serveur : constructeur
  ```
  Socket socket = new Socket("android.com", 4444);
  ```
- flux sortant :
  ```
  PrintWriter writer = new
  PrintWriter(socket.getOutputStream(), true);
  ```
- flux entrant :
  ```
  BufferedReader reader = new BufferedReader(new
  InputStreamReader(socket.getInputStream()));
  ```
- fermeture : `socket.close()`

Les *sockets* serveur

La classe `ServerSocket` :

- ouverture : constructeur
  ```
  ServerSocket serverSocket = new ServerSocket(4444);
  ```
- attente d'une connexion client (rend une `Socket`) :
  ```
  Socket socket = serverSocket.accept();
  ```
  Attention : **action bloquante**
- fermeture : `serverSocket.close()`

Conclusion sur les applications client/serveur

Le client :

- doit respecter le protocole
- utilise des threads :
  - pour se connecter au serveur
  - pour attendre, potentiellement indéfiniment, les informations du serveur

Le serveur :

- dual du client
- utilise des threads :
  - pour attendre que les clients se connectent/envoient des requêtes
  - pour attendre les données des clients
- différence : doit pouvoir gérer plusieurs clients en même temps (parallélisme)

Plan

De multiples possibilités

Pour faire des threads :

- héritage de la classe `Thread`
- implantation de l'interface `Runnable`
- utilisation de la classe `Executors`
- ...

Pour faire des applications clientes :

- *sockets*
- serveurs web et API REST
- ...