Threads

Chantal Keller

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 :

- ressources coûteuses : réseau, internet
- calcul bloquant : attente d'une entrée de l'utilisateur, attente de connexion
- calcul infini : écoute sur un réseau

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

$\hookrightarrow$ répartition des tâches **obligatoire**

Autres applications :

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

Difficultés des threads

Complexité de l'exécution parallèle :

- non déterminisme
- mémoire partagée
- points de synchronisation

Implantations en Android

Threads de Java :

+ toute la généralité des threads (parallélisme)
- toute la complexité des threads (ressources partagées)
- communication avec le thread difficile

La classe `AsyncTask` :

+ haut niveau
+ garanties de synchronisation
- parallélisme uniquement avec le thread principal (pas entre `AsyncTasks`)

Plan

La classe `AsyncTask`

Encore de la programmation événementielle :

- méthode qui sera exécutée par la tâche de fond
- méthodes qui seront exécutées par le thread principal pour mettre à jour l'interface avant, pendant et après l'exécution de la tâche de fond

Principe :

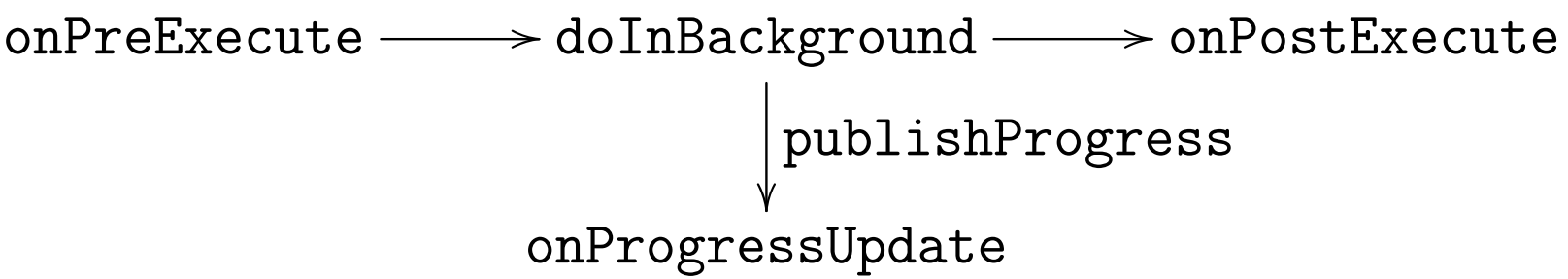
- faire une classe `Tache` héritant de `AsyncTask`
- redéfinir quelques méthodes (transparent suivant)
- lancer la tâche (ex : `new Tache().execute()`)

Détails de la classe `AsyncTask`

Méthodes à redéfinir :

- code à exécuter par la tâche : `doInBackground`
- code à exécuter pour mettre à jour l'interface :
  - avant l'exécution de la tâche : `onPreExecute`
  - pendant l'exécution de la tâche : `onProgressUpdate`
  - après l'exécution de la tâche : `onPostExecute`

Garanties de synchronisation :



Un peu de généricité

`class AsyncTask<Params, Progress, Result>`

- `Params` : type des paramètres envoyés à la tâche (entrée)
- `Progress` : type des résultats fournis au fur et à mesure de l'exécution de la tâche (sortie)
- `Result` : type du résultat de l'exécution de la tâche (sortie)

Répercussion sur la signature des méthodes

```
class AsyncTask<Params, Progress, Result>
```

Méthodes à redéfinir :

- `Result doInBackground (Params... params)`
- `void onPreExecute ()`
- `void onProgressUpdate (Progress... values)`
- `void onPostExecute (Result result)`

Méthode qu'on peut appeler :

- `void publishProgress (Progress... values)`

Exemple

```
private class WaitingThread extends AsyncTask<Void, Integer, Integer> {
    private final int wait = 5000;
    private final int number = 6;

    @Override
    protected void onPreExecute() {
        affichage.setText("Lancement du thread...");
    }

    @Override
    protected Integer doInBackground(Void... voids) {
        for (int count = 0; count < number; count++) {
                Thread.sleep(wait);
                publishProgress(count+1);
        }
        return number;
    }

    @Override
    protected void onProgressUpdate(Integer... counts) {
        int time = counts[0] * wait / 1000;
        affichage.setText("Le thread s'execute depuis " + time + " secondes");
    }

    @Override
    protected void onPostExecute(Integer res) {
        int time = res * wait / 1000;
        affichage.setText("Le thread a fini ; il s'est execute pendant " + time + " secondes");
    }
}
```

Interruption d'une tâche (1/2)

On peut vouloir interrompre une tâche en cours d'exécution :

- quand l'utilisateur quitte l'activité
- quand l'utilisateur appuie sur un bouton
- au bout d'une trop longue durée (*timeout*)
- ...

On signale à la tâche qu'on souhaite l'interrompre :

- méthode : `tache.cancel(true)`
- à elle ensuite de véritablement s'interrompre

Interruption d'une tâche (2/2)

Une tâche peut déterminer si elle doit s'interrompre :

- la méthode `isCancelled()` renvoie `true`

Lorsque c'est le cas :

- la tâche doit arrêter son long calcul
- elle peut ensuite encore effectuer quelques opérations
- ex dans le corps d'une boucle infinie :
  `if (isCancelled()) break;`

Pas de parallélisme entre AsyncTasks

Exemple :

```
tache1.execute();
tache2.execute();
```

`tache2.doInBackground` n'est appelée que lorsque `tache1.doInBackground` a fini

Plan

La classe `Thread`

Une seule méthode à redéfinir :

- méthode qui sera exécutée par la tâche de fond
- pour mettre à jour l'interface : "poster" des messages au thread principal

Principe :

- faire une classe `Fil` héritant de `Thread` (ou implantant `Runnable`, au choix)
- redéfinir la méthode `run`
- lancer la tâche (ex : `new Fil().start()`)

Détails de la classe Thread

Un seule méthode à redéfinir :

- `public void run() { ... }`
- “correspond” à `doInBackground`

Utilisation et mise à jour de l'interface

```
public class MainActivity extends Activity {


    private class Fil extends Thread {

        public void run() {
                    ...




                    ...
        }

    }
}
```

Utilisation et mise à jour de l'interface

```
public class MainActivity extends Activity {


    private class Fil extends Thread {

        public void run() {
                    ...

                    affichage.setText(text);


                    ...
        }

    }
}
```

Utilisation et mise à jour de l'interface

```
public class MainActivity extends Activity {



    private class Fil extends Thread {

        public void run() {
                    ...
            Runnable posterAffichage = new Runnable() {
                public void run() {
                    affichage.setText(text);
                }
            };
            handler.post(posterAffichage);
                    ...
        }

    }
}
```

Utilisation et mise à jour de l'interface

```
public class MainActivity extends Activity {

    private final Handler handler = new Handler(); // classe android.os.Handler

    private class Fil extends Thread {

        public void run() {
                    ...
            Runnable posterAffichage = new Runnable() {
                public void run() {
                    affichage.setText(text);
                }
            };
            handler.post(posterAffichage);
                    ...
        }

    }
}
```

Simulation de AsyncTask avec Thread

```
private class WaitingThread extends Thread {
    private void publish(final String text) {
        Runnable affichagePre = new Runnable() {
            public void run() { affichage.setText(text); }
        };
        handler.post(affichagePre);
    }

    private void preExecute() { publish("Lancement du fil..."); }

    private void progressUpdate(Integer... counts) {
        int time = counts[0] * wait / 1000;
        publish("Le fil s'exécute depuis " + time + " secondes");
    }

    private void postExecute(Integer res) {
        int time = res * wait / 1000;
        publish("Le fil a fini ; il s'est exécuté pendant " + time + " secondes");
    }

    private Integer inBackground(Void... voids) {
        for (int count = 0; count < number; count++) {
            try { Thread.sleep(wait);
                  progressUpdate(count + 1); }
            catch (InterruptedException e) { break; }
        }
        return number;
    }

    public void run() { preExecute(); int res = inBackground(); postExecute(res); }
}
```

Interruption d'un fil

Même principe :

- on signale au fil de s'interrompre
- à lui de gérer cela pour s'arrêter

Autre syntaxe :

- signalement : appel à `interrupt()`
- savoir si on est interrompu : test `interrupted()`

Parallélisme entre Threads

Exemple :

```
fil1.start();
fil2.start();
```

Les deux fils sont lancés et s'exécutent "simultanément"
(ordre d'exécution non déterministe)

Plan

Conclusion

Utilisation de worker threads :

- dès qu'on a une tâche longue ou utilisant des ressources coûteuses
- attention à la mise à jour de l'interface

Choix de l'implantation :

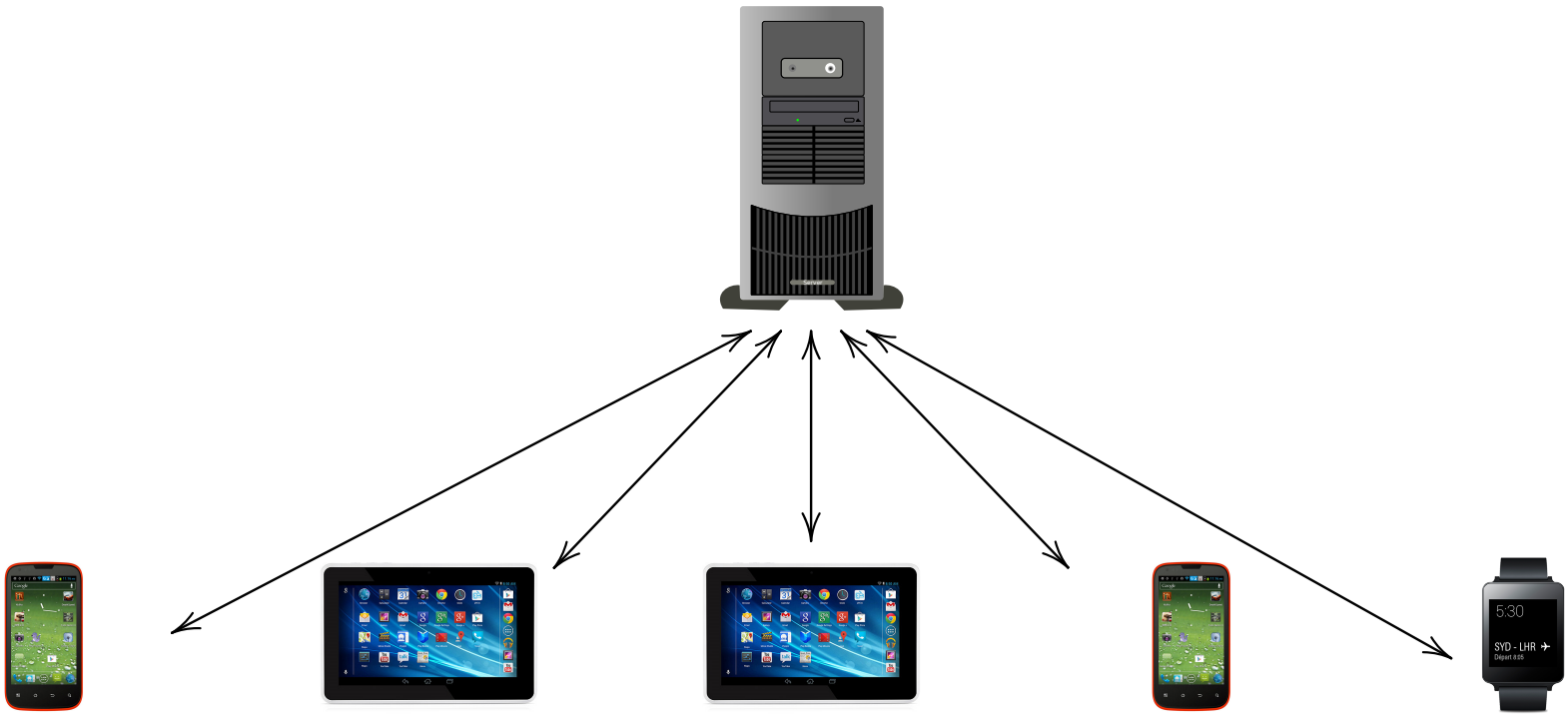
- `AsyncTask` : simple à utiliser, facile de modifier l'interface en cours de route
- `Thread` : si on a besoin de parallélisme entre plusieurs worker threads

AsyncTask vs Thread

	AsyncTask	Thread
Code du thread	redéfinir `doInBackground`	redéfinir `run`
Affichage	utiliser `publishProgress` et redéfinir `onProgressUpdate`	`handler.post` (arg : `Runnable`)
Parallélisme	non	oui

Plan

Principe



Serveur

Rôle :

- proposer des services
- stocker des données
- réaliser des actions
- ...

Infrastructure :

- “cloud”
- appareil
- autre appareil Android
- ...

Clients

Principe :

- tous appareils Android
- éventuellement, autres OS
- multiples, en parallèle

Infrastructure de communication :

- réseau (local ou internet)
- bluetooth
- ...

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. gérer l'infrastructure de communication
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

Ils sont partout !

Threads pour :

1. gérer l'infrastructure de communication : `Asynctask`
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

Ils sont partout !

Threads pour :

1. gérer l'infrastructure de communication : `Asynctask`
2. client : gérer l'affichage des messages du serveur : `Asynctask`
3. serveur : attendre des connexions
4. serveur : pour chaque client en parallèle, gérer la réception et l'envoi de messages

Ils sont partout !

Threads pour :

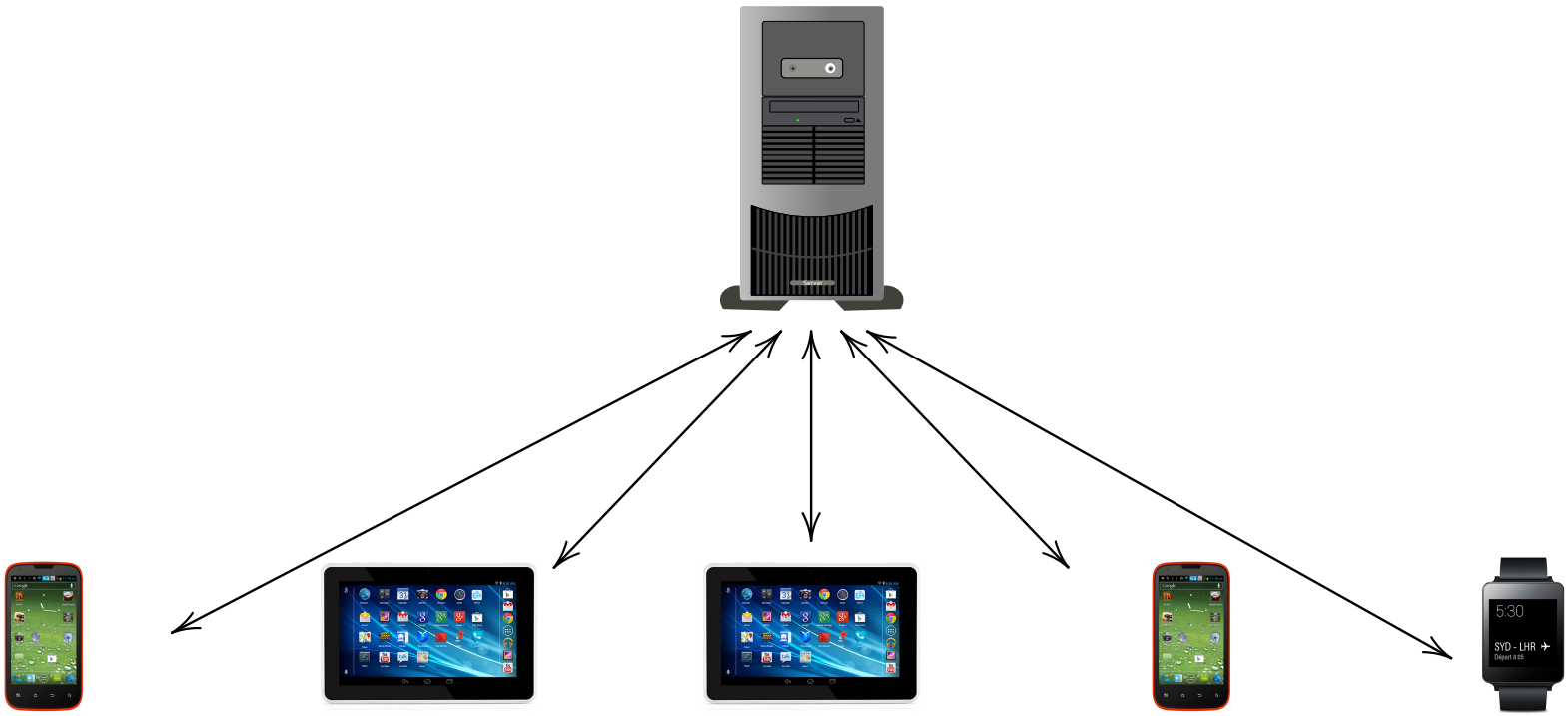
1. gérer l'infrastructure de communication : `Asynctask`
2. client : gérer l'affichage des messages du serveur : `Asynctask`
3. serveur : attendre des connexions : `Thread`
4. serveur : pour chaque client en parallèle, gérer la réception et l'envoi de messages

Ils sont partout !

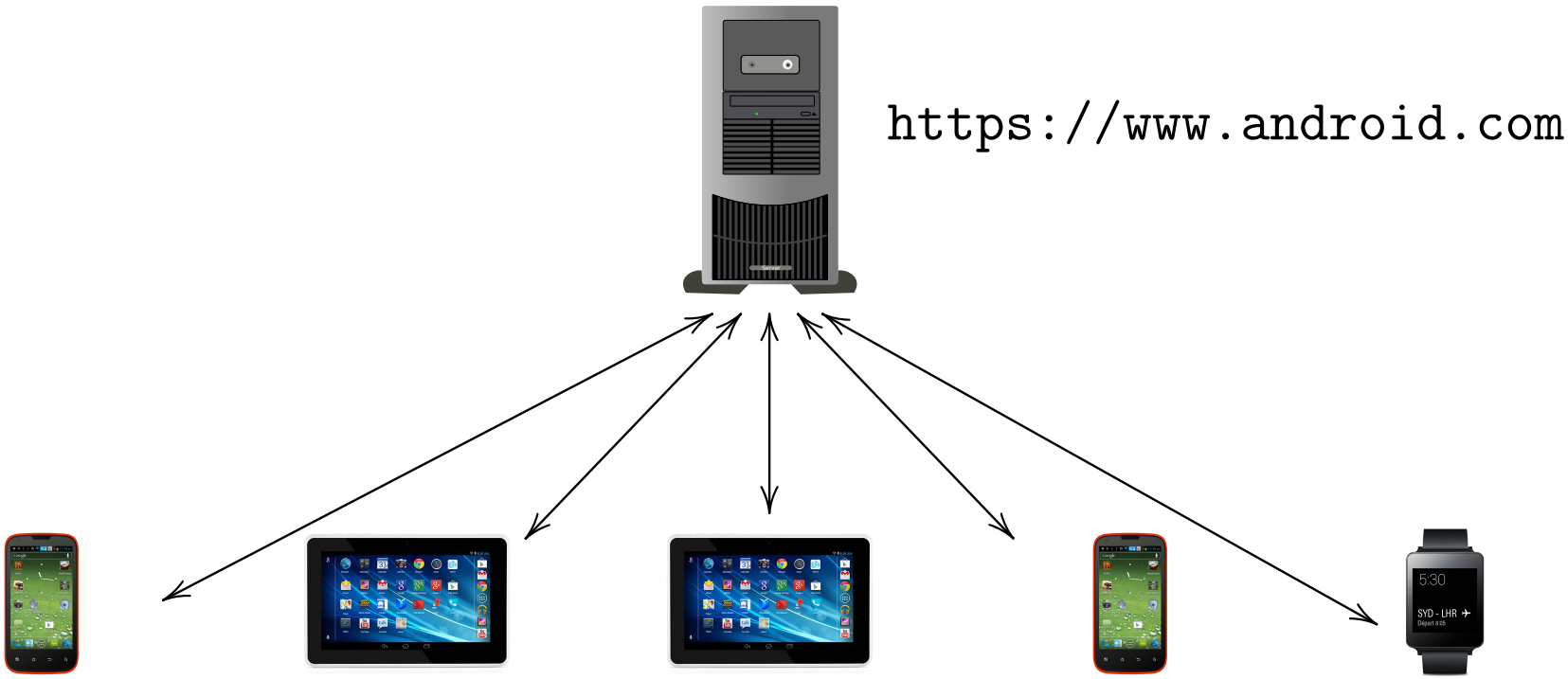
Threads pour :

1. gérer l'infrastructure de communication : `Asynctask`
2. client : gérer l'affichage des messages du serveur : `Asynctask`
3. serveur : attendre des connexions : `Thread`
4. serveur : pour chaque client en parallèle, gérer la réception et l'envoi de messages : `Thread`

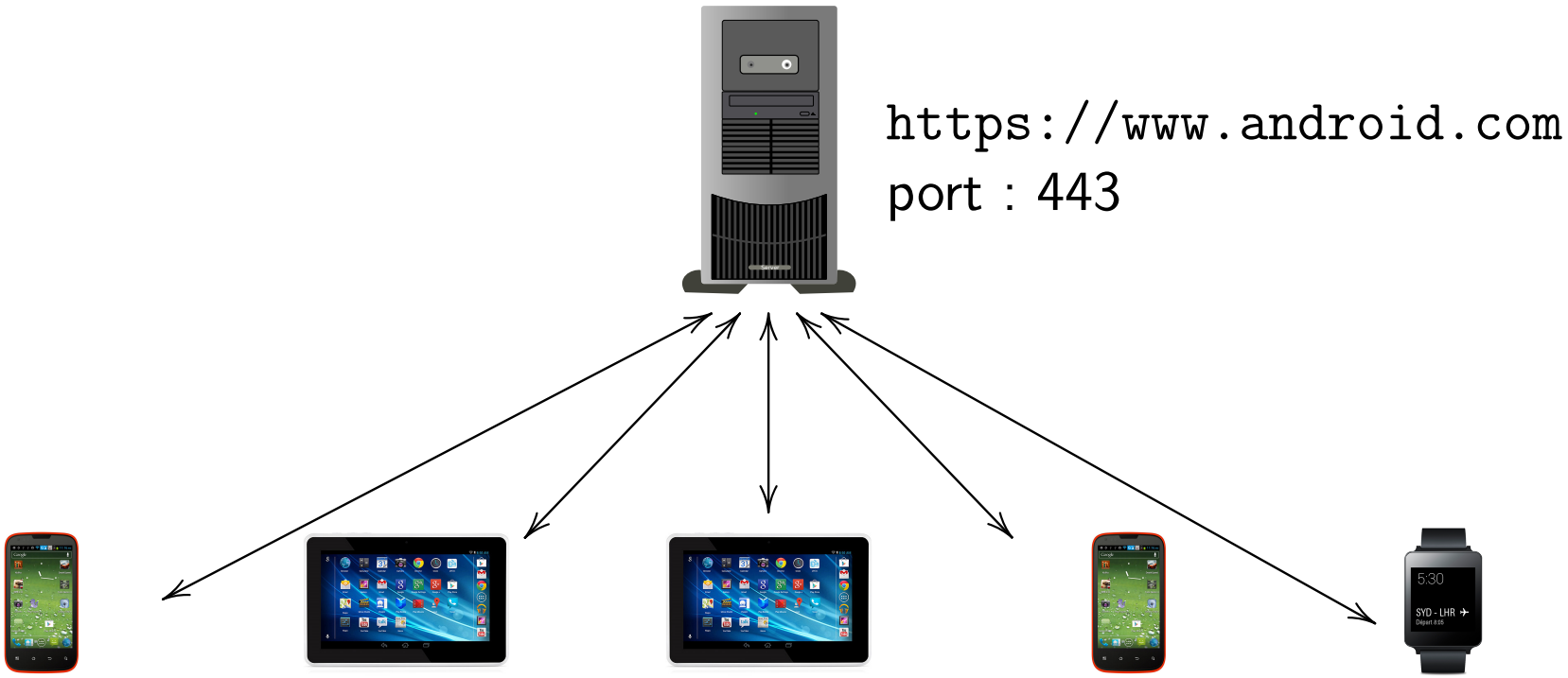
Communication par des sockets



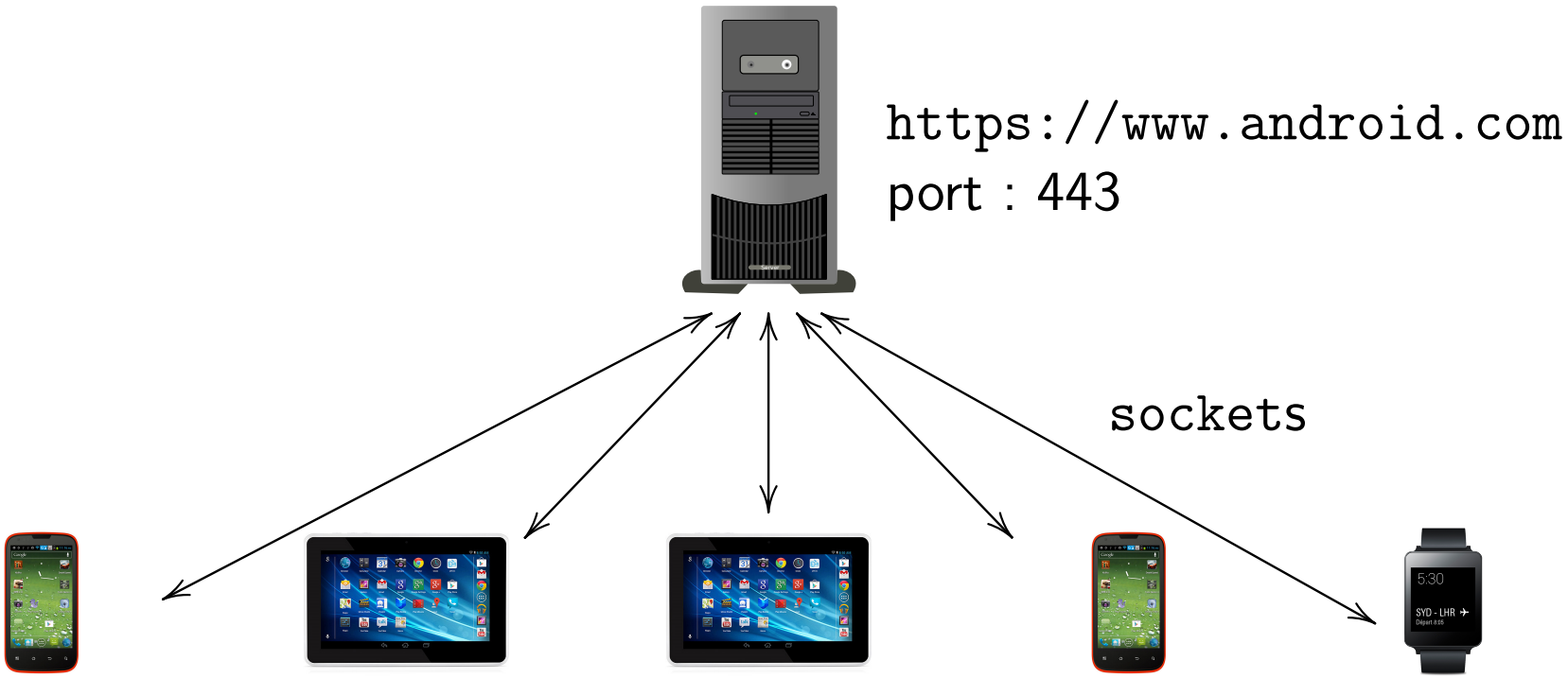
Communication par des sockets



Communication par des sockets



Communication par des sockets



Application client/serveur via le réseau

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, 443);
  ```
- 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(443);`
- attente d'une connexion client (rend une `Socket`) :
  `Socket socket = serverSocket.accept();`
  Attention : **action bloquante**
- fermeture : `serverSocket.close()`

Applications client/serveur

Le client :

- doit respecter le protocole
- utilise des threads :
  - pour se connecter à la socket serveur
  - pour attendre sur cette socket les informations du serveur

Le serveur :

- dual du client
- différence : doit pouvoir gérer plusieurs clients en même temps (parallélisme, donc utilisation de la classe `Thread`)