IFT604 / IFT717 – Applications Internet et Mobilité

Date : jeudi 8 octobre 2015
Heure : 15 h 30 – 17 h 20

Professeur : Sylvain GIROUX
Local : D3-2038

Note :…/100 points

Toute documentation permise.

Etude de cas : Le combat des chefs

Les débats électoraux sont une occasion privilégiée où les instituts de sondage prennent le pouls des électeurs. S'en suivent les débats sur le débat : qui a gagné ? quel a été le moment de vérité ? etc.. En plus des sondages réalisés avant et après le débat, les médias analysent aussi en direct diverses informations recueillies pendant le débat, par exemple, mesurer les échanges sur Twitter ou recueillir les avis et commentaires d'un panel d'électeurs. On pourra discuter longtemps de la valeur scientifique et statistique de ces nouvelles formes de sondage qui ne respectent plus nécessairement les règles élémentaires de la statistique dans la composition des échantillons. La technologie évolue et ouvre de nouvelles perspectives. Ainsi les nouvelles montres connectées, e.g. Microsoft Band, permettent de recueillir en temps réel, peu importe le lieu, des données physiologiques, comme le rythme cardiaque, la température corporelle, la réponse galvanique de la peau (GSR)... La réponse galvanique de la peau est particulièrement intéressante dans notre contexte. En effet, cette dernière permet de mesurer les émotions de manière fiable.

Les instituts de sondage ont décidé de connaitre et diffuser en direct pendant le débat les émotions et les réactions des membres d'un panel qu'ils ont choisis et qu'ils jugent représentatifs. Vous devez donc construire une application client-serveur pour le prochain débat. Il y aura deux types de clients :

- L'application client des *panellistes*. Le panel sera composé de 1000 membres qui auront deux rôles :
    - Lors du débat, les émotions des panellistes seront mesurées à l'aide d'un flux GSR recueilli à l'aide de leur téléphone et d'une montre connectée. Les panellistes pourront démarrer et arrêter à tout moment l'enregistrement des données GSR, et par conséquent la mesure de leurs émotions en direct.
    - Les panellistes évalueront l'impact de la performance des candidats. Pour ce faire, nous supposerons qu'il y a trois politiciens qui débattront : Harper, Mulcair et Trudeau. Chacun d'eux commencera avec un capital de 1000 points. Si un membre du panel juge que l'un d'eux a marqué des points aux dépens d'un des deux autres, il peut transférer 10 points du gagnant au perdant de l'échange.

  Ainsi, l'application client d'un panelliste devra :
    - transmettre le flux GSR au serveur pour qu'il calcule les émotions du panelliste. Cette information est transmise au serveur.
    - permettre au panelliste de savoir quelle émotion lui est attribuée par le système. Cette information est calculée par le serveur à partir du flux de données GSR. Elle enregistrée sur le serveur qui la rend au panelliste sur demande de l'application client.
    - permettre au panelliste de démarrer et d'arrêter l'enregistrement de données GSR à tout moment. Cette information est transmise au serveur.
    - permettre au panelliste de coter la performance des candidats via le transfert de points entre candidats. Cette information est transmise au serveur.
    - permettre à un panelliste de savoir quel est le pointage qu'il a attribué à chaque politicien. Cette valeur est conservée localement sur le téléphone du panelliste.
- L'application client des *électeurs*. Le nombre d'électeurs qui se brancheront au système est inconnu a priori. L'institut de sondage anticipe toutefois qu'il y aura des pics de 1 million de personnes connectées pendant le débat. Les électeurs peuvent avoir sur demande un portrait global des émotions des membres du panel et de leur cotation de la performance des candidats. Ainsi l'application client d'un électeur quelconque aura les fonctionnalités suivantes :
    - afficher l'émotion dominante chez les panellistes
    - afficher le gagnant du débat
    - afficher la cotation de chacun des candidats
    - mettre à jour sur demande de l'électeur, e.g. via un bouton, les deux informations précédentes.

Donc le serveur reçoit et compile les données des membres du panel : activation/ désactivation du flux GSR, stockage des données du flux GSR, ajout ou retrait des points pour un politicien. Il envoie aux électeurs qui le

demandent le portrait global. Sur demande de l'application client des panellistes, il envoie la dernière émotion qu'il a calculée.

En annexe, vous trouverez un rappel sur les méthodes liées au cycle de vie des activités et des services, le code pour se connecter au lecteur GSR et des classes utilitaires pour transmettre des messages via UDP ou TCP, des classes pour représenter les messages, les requêtes et réponses des panellistes et des électeurs.

Le serveur Java (50 points)

Vous devez définir l'infrastructure client-serveur qui servira à supporter le système pour le prochain débat lors des élections canadiennes. Le code du serveur s'exécutera dans un même processus. Il sera exécuté sur une machine capable de gérer environ 10 000 fils d'exécution en parallèle, au-delà de ce nombre la performance diminue sérieusement.

Question 1 : Le protocole de communication (7 points)

a) Quel protocole de communication (UDP/datagramme ou TCP/flot) choisiriez-vous pour les communications avec les membres du panel? Justifier votre réponse. 10 lignes max.

b) Quel protocole de communication (UDP/datagramme ou TCP/flot) choisiriez-vous pour la communication avec les électeurs? Justifier votre réponse. 10 lignes max.

Question 2 : L'architecture du serveur (8 points)

a) Quelle architecture choisiriez-vous pour le traitement des requêtes des membres du panel (thread-per-request, thread-per-connection, thread-per-objects) ? Justifier votre réponse. 10 lignes max.

b) Quelle architecture choisiriez-vous pour le traitement des requêtes des électeurs (thread-per-request, thread-per-connection, thread-per-objects)? Justifier votre réponse. 10 lignes max.

Question 3 : L'implémentation du serveur (10 points)

À l'aide d'un schéma et/ou de diagrammes de séquence, décrivez une implémentation alternative du serveur si le protocole UDP et l'architecture *thread-per-request* sont utilisés. Sur les figures, vous devez en particulier identifier les objets impliqués, les fils d'exécution lancés, les requêtes et les réponses échangées.

Question 4 : La synchronisation et l'intégrité des données (25 points)

Du côté du serveur, on vous a donné la classe **`DebateStats`** (voir ci-dessous) pour traiter chacune des requêtes des deux types de clients (panellistes et électeurs). Cependant cette implémentation ne fonctionne pas bien dans une architecture *thread-per-request* ou *thread per connection*.

a) Quel est le problème typique de concurrence que l'on rencontre ici (philosophes, producteur-consommateur ou lecteur-écrivain)? Justifier votre réponse.

b) Chez les applications clientes des électeurs, on observe que le résultat du vainqueur et les cotations ne sont pas toujours cohérents : un candidat peut être déclaré vainqueur à un moment donné alors que son total de points est moindre que celui d'un autre candidat. Il en va de même du nombre total de points attribués aux candidats. Ce total devrait être égal à "nombre de candidats" * "nombre de points attribués initialement" * "nombre de panellistes", puisque seules des opérations de transfert sont effectuées par les panellistes. Expliquer d'où proviennent ces incohérences.

c) Identifier un autre cas où le comportement du programme pourrait être incorrect. Expliquer pourquoi à l'aide d'un court exemple.

d) Corriger l'implémentation de la classe **`DebateStats`** pour qu'il n'y ait plus de problème de synchronisation, ni d'intégrité des données. Prenez soin de maximiser la concurrence. Ne pas recopier le code qui ne change pas.

```
// LES CLASSES UTILISEES PAR LE SERVER POUR COLLIGER ET ANALYSER LES DONNEES DU DEBAT

// enums describing the debate

public class Debate {      public enum Debater {Mulcair, Harper, Trudeau};
                           public enum Emotion {angry, sad, happy, neutral};
}// END of class Debate
```

```
// data structure containing a debate current status for transferring to clients

public class DebateStatus {

	private Emotion dominantEmotion; // predominant emotion in the panel members
	private Debate.Debater winner;   // current debater having the best score
	// performance of each debater
	private Hashtable<Debate.Debater, Integer > performance;

	// all getters and setters
	public Emotion getDominantEmotion() {return dominantEmotion;}
	public void setDominantEmotion(Emotion dominantEmotion) {
		this.dominantEmotion = dominantEmotion;
	}

	public Debate.Debater getWinner() { return winner;}
	public void setWinner(Debater winner) { this.winner = winner; }

	public Integer getPerformance(Debater debater) {
		return this.performance.get(debater);
	}
	public void setPerformance(Debater debater, Integer performance) {
		this.performance.put(debater, performance);
	}
} // END of class DebateStatus
```

```
// the object on the server that collects all the stats and data

public class DebateStats {

	// for each panelist, number of points initially allocated to a debater
	final static public int DEBATER_INITIAL_SCORE = 1000;

	// number of points transferred from one debater to another
	// when a panelist judges that the latest has a better performance
	private static final int PERFORMANCE_BONUS = 10;

	// number of members in the panel
	final static public int PANEL_SIZE = 1000;

	// Note : HashMap and ArrayList are not threadsafe

	// String = panelist ID,
	// ArrayList<Integer> = streams of GSR data
	private HashMap<String, ArrayList<Integer>> dataWarehouse;

	// String = panelist ID,
	// Emotion = last computed emotion for this panelist
	private HashMap<String, Emotion> emotionMap;

	// Debater = Harper, Mulcair, or Trudeau
	// Integer = current performance score for this debater
	private HashMap<Debater, Integer> debatersPerformance;
```

```
        public DebateStats(){
        // initialization
        dataWarehouse = new HashMap<String, ArrayList<Integer>>();
        emotionMap = new HashMap<String, Emotion>();
        debatersPerformance = new HashMap<Debater, Integer>();
        int totalInitialPerformance = DEBATER_INITIAL_SCORE * PANEL_SIZE;
        debatersPerformance.put(Debater.Harper, totalInitialPerformance);
        debatersPerformance.put(Debater.Mulcair, totalInitialPerformance);
        debatersPerformance.put(Debater.Trudeau, totalInitialPerformance);
}

// voter REQUEST
public DebateStatus getCurrentDebateStatus(){
        DebateStatus debateStatus = new DebateStatus();

        debateStatus.setDominantEmotion(computeDominantEmotion());
        debateStatus.setWinner(computeWinner());
        debateStatus.setPerformance(Debater.Harper,
                                    debatersPerformance.get(Debater.Harper));
        debateStatus.setPerformance(Debater.Mulcair,
                                    debatersPerformance.get(Debater.Mulcair));
        debateStatus.setPerformance(Debater.Trudeau,
                                    debatersPerformance.get(Debater.Trudeau));
        return debateStatus;
}

// panelist REQUEST
// adding a GSR data to the flow of data for a given panelist
// recomputing and updating her current emotion
public void addData(String panelMember, Integer gsrValue){
        if(isDataTransmissionActivated(panelMember)) {
                ArrayList<Integer> gsrData = dataWarehouse.get(panelMember);
                gsrData.add(gsrValue);
                emotionMap.put(panelMember, inferEmotion(gsrData));
        }
}

// panelist REQUEST
// activating and deactivating GSR data recording
public void dataTransmissionActivated(String panelMember, boolean activated){
        if (activated) {dataWarehouse.put(panelMember, new ArrayList<Integer>());
        }
        else {dataWarehouse.remove(panelMember);
        }
}

// panelist REQUEST
// transferring performance points from one debater to another
public void transferPoints(String panelMember, Debater winner, Debater loser){
        Integer winnerOldScore = debatersPerformance.get(winner);
        Integer winnerNewScore = winnerOldScore + PERFORMANCE_BONUS;
        debatersPerformance.put(winner, winnerNewScore);

        Integer loserOldScore = debatersPerformance.get(loser);
        Integer loserNewScore = loserOldScore - PERFORMANCE_BONUS;
        debatersPerformance.put(loser, loserNewScore);
}

// panelist REQUEST
// current emotion associated to a given panelist
public Emotion getMyEmotion(String panelMember){
        return emotionMap.get(panelMember);
}
```

```
	// Helper methods

	// GSR recording is activated
	// if there is key for the panelist in the datawarehouse
	private boolean isDataTransmissionActivated(String panelMember){
		return dataWarehouse.containsKey(panelMember);
	}

	// infer current emotion from latest GSR data
	private Emotion inferEmotion(ArrayList<Integer> arrayList) {

		// complex and long computation
		return null;
	}

	// the winner is the debater having the best score
	private Debater computeWinner() {
		int harper = debatersPerformance.get(Debater.Harper);
		int mulcair = debatersPerformance.get(Debater.Mulcair);
		int trudeau = debatersPerformance.get(Debater.Trudeau);

		if(harper > mulcair && harper > trudeau ) return Debater.Harper;
		if(mulcair > harper && mulcair > trudeau ) return Debater.Mulcair;
		if(trudeau > mulcair && trudeau > harper ) return Debater.Trudeau;

		return null; // no clear winner
	}

	// the dominant emotion is the most prevalent emotion amongst panelists
	private Emotion computeDominantEmotion() {
		Collection<Emotion> emotions = emotionMap.values();
		Emotion dominantEmotion = Debate.Emotion.neutral;
		int dominantFrequency = 0;
		for (Emotion emotion : emotions) {
			int emotionFrequency = Collections.frequency(emotions, emotion);
			if (emotionFrequency > dominantFrequency) {
				dominantEmotion = emotion;
				dominantFrequency = emotionFrequency;
			}
		}
		return dominantEmotion;
	}
} // END of class DebateStats
```

Le client Android (50 points)

Dans cette question, vous devez implémenter (partiellement) l'application Android d'un panelliste s'exécutant sur son téléphone cellulaire. Cette application compte deux grandes fonctions : la reconnaissance des émotions du panelliste et la cotation des candidats par le panelliste. Chaque fonction aura son propre écran et l'application devra permettre de passer de l'une à l'autre.

Question 5 : L'interface usager (5 points)

Dessiner les deux écrans de l'interface-usager offerte au panelliste, plus précisément :

- Écran pour les opérations liées à la reconnaissance des émotions
- Écran pour les opérations liées à la cotation des candidats

Question 6 : L'architecture du client (10 points)

Faites un schéma général décrivant l'architecture du client du panelliste. Identifier les classes pertinentes et leurs méthodes principales: les activités, les services, les fils d'exécution, les communications avec le serveur, etc.

Question 7 : L'implémentation du client (35 points)

Implémenter sous Android les éléments qui sont liés à la reconnaissance des émotions. Plus précisément, les fonctions à implémenter sont :

- Collecter et transmettre les données GSR au serveur. Cette fonctionnalité doit être active, sauf si l'usager l'a arrêtée. Elle est activée au lancement de l'application.
- Démarrer / arrêter la transmission des données GSR au serveur explicitement par le panelliste
- Afficher l'émotion courante du panelliste telle qu'elle a été déterminée par le serveur
- Transiter entre l'activité de cotation des candidats et l'activité de reconnaissances des émotions

Vous n'avez pas besoin de donner les fichiers XML, seul le code suffit. Portez une attention particulière aux méthodes liées au cycle de vie des activités et des services. N'oubliez pas que l'application sera utilisée aussi bien en orientation *portrait* qu'en orientation *paysage* et que l'usager pourra passer d'un mode à l'autre n'importe quand.

```
// Rappel sur les méthodes liées au cycle de vie d'une activité.

public class MainActivity extends Activity  {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
  protected void onRestart() { super.onRestart();
  }
  protected void onStart() { super.onStart();
  }
  protected void onResume() { super.onResume();
  }
  protected void onPause() { super.onPause();
  }
  protected void onStop() { super.onStop();
  }
  protected void onDestroy() { super.onDestroy();
  }
  public void  onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
  }
}
```

```
// Rappel sur les méthodes liées au cycle de vie d'un service.

public class MyService extends Service {
       public void onCreate() { super.onCreate();
       }
       public int onStartCommand(Intent intent, int flags, int startId) {
              return super.onStartCommand(intent, flags, startId);
       }
       public IBinder onBind(Intent intent) {
       }
       public boolean onUnbind(Intent intent) { return super.onUnbind(intent);
       }
       public void onDestroy() { super.onDestroy();
       }
}
```

```
// Connexion au capteur de GSR
// et réception des notifications lors des changements de valeur. Patron observateur.

// Get an instance of the sensor service, and use it to get an instance of
// a particular sensor.
SensorManager mSensorManager = aContext.getSystemService(Context.SENSOR_SERVICE);
Sensor gsrSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GSR);

SensorEventListener aListener = new SensorEventListener() {
         public final void onSensorChanged(SensorEvent event) {
           int gsrValue = event.values[0];
           // Do something with this sensor data.
         }
};

// (Un)Register the listener with the sensor to receive value updates
gsrSensor.addSensorEventListener(aListener);
gsrSensor.removeSensorEventListener(aListener);
```

```
// Classe utilitaire pour la transmission et la réception de messages via UDP

public class UDPHelper {
	//transmission non bloquante sur UDP
	static public void sendUDP(String ip, int port, Object msg) { … }
	//réception bloquante sur UDP
	static public Object receiveUDP(String ip, int port) { … }
}
```

```
// Classe utilitaire pour la transmission et la réception de messages via TCP

public class TCPHelper {
	//transmission non bloquante sur TCP,
	// l'écriture peut être bloquée par le contrôleur TCP
	static public void sendTCP(String ip, int port, Object msg) { … }
	//réception bloquante sur TCP
	static public Object receiveTCP(String ip, int port) { … }
}
```

```
// superclass for messages

public class Message {

	public String destinationIP;
	public int destinationPort;

	public String senderIP;
	public int senderPort;

	public int messageID; // message unique ID number
}
```

```
// classes related to voters requests and replies

public class CitizenRequest extends Message {

	enum CitizenMethods {getCurrentDebateStatus};

	public CitizenMethods m;

	public CitizenRequest() { this.m = CitizenMethods.getCurrentDebateStatus;
	}
}

// REPLY to "getCurrentDebateStatus" CitizenRequest message

public class CitizenReply extends Message {

	public int requestMessageID;

	// stats includes
	//     predominant emotion amongst panelists
	//     debate current winner according to all panelists
	//     debaters current quotations by all panelists

	public DebateStatus debateStatus;
}
```

```
// classes related to panelists requests and replies

public class PanelRequest extends Message {

	enum PanelMethod {  addData, dataTransmissionActived,
	                    transfertPoints, getMyEmotion};

	public String panelMember; // name of the panel member

	public PanelMethod m;
	public int gsrValue;             // used if m = addData
	public boolean transmittingGSR;  // used if m = dataTransmissionActived
	public boolean agree;            // used if m = iAgree
	public Debater winner;           // used if m = transfertPoints
	public Debater loser;            // used if m = transfertPoints

	public PanelRequest(String panelMember, PanelMethod methodName) {
		this.panelMember = panelMember;
		this.m = methodName;
	}

	// factory methods

	public static PanelRequest newAddDataRequest(String panelMember, int gsrValue) {
		PanelRequest r = new PanelRequest(panelMember, PanelMethod.addData);
		r.gsrValue = gsrValue;
		return r;
	}

	public static PanelRequest newDataTransmissionActivated(String panelMember,
	                                                         boolean transmitting) {
		PanelRequest r = new PanelRequest( panelMember,
		                                   PanelMethod.dataTransmissionActived);
		r.transmittingGSR = transmitting;
		return r;
	}

	public static PanelRequest newTransfertPointsRequest( String panelMember,
	                                                       Debater winner,
	                                                       Debater loser) {
		PanelRequest r = new PanelRequest( panelMember,
		                                   PanelMethod.transfertPoints);
		r.winner = winner;
		r.loser = loser;
		return r;
	}

	public static PanelRequest newGetMyEmotion(String panelMember) {
		PanelRequest r = new PanelRequest(panelMember, PanelMethod.getMyEmotion);
		return r;
	}
}

// REPLY to "getMyEmotion" PanelRequest message

public class PanelReply extends Message {
	public int requestMessageID;
	public Emotion inferredEmotion; // emotion of the panelist that made the request
}
```

FIN DE L'EXAMEN