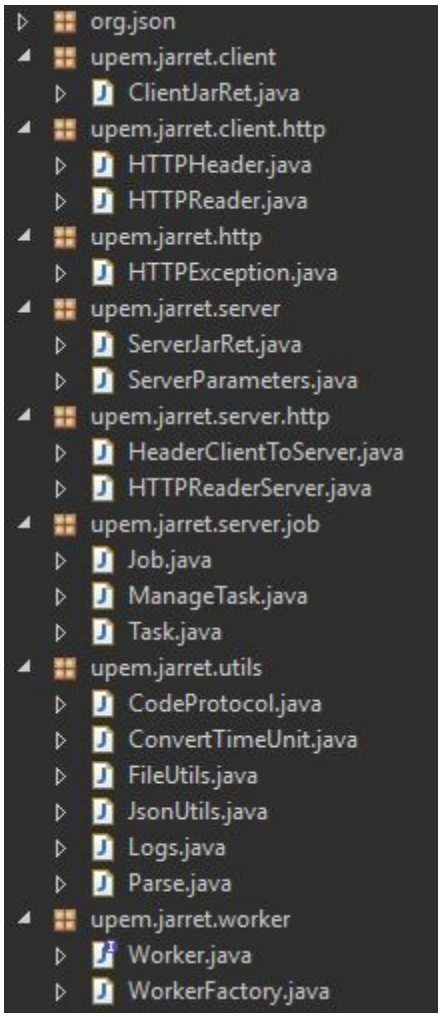
Duchemin Kévin
Le Gourrierec Maugan

JarRet
Manuel Développeur

Choix d’architecture :



Client :

Notre client est un client implémenter suivant le principe TCP en mode bloquant, car nous ré-utilisons pour la partie header du protocole http du projet, les HTTPReader, HTTPException, HTTPHeader développer en TP. Notamment la récupération des headers, ainsi que la méthode readBytes() pour toute la partie suivant le header de réception(c’est-à-dire la partie JSON) grâce au content-length du header. Nous avons également choisis d’utiliser la structure d’énumération pour la partie des codes du protocole. Pour la partie JSON, nous utilisons l’api org.json disponible depuis le lien suivant : https://github.com/stleary/JSON-java Nous avons pris cette api pour sa simplicité d’utilisation quant à la création d’objet JSON à partir de JSON au format de String.

Serveur :

Le serveur que nous avons développé pour ce projet est un serveur TCP en mode non-bloquant. Pour ce serveur nous avons séparé le serveur en plusieurs sous paquets, notamment pour la partie de gestion, création et suppression de job et des tâches dans des classes distinctes. ManageTask qui gère :

- une liste de jobs actifs
- une hashSet de jobs finis
- une hashSet de chemins des différents répertoires de à parcourir pour récupérer les fichiers des différents jobs à importer
- une hashSet des fichiers dans lesquels on vérifie et récupères les informations pour la créations des jobs
- et un fil d'exécution(plus couramment appelé thread) pour pouvoir recharger les fichiers depuis les répertoires enregistrer dans le ManageTask et/ou les fichiers données à cette objects, et après avoir recharger tout cela, il recharge les jobs non déjà récupéré depuis les fichiers trouvées et stockés .
  Pour l'ajout de nouveaux jobs dans la liste des jobs actifs, nous passons par une factory method depuis la classe job qui selon le jobPriority renvoie une liste possèdent jobPriority-copies de la même instance du job créer avec les paramètres données à la méthode. Grâce à cela nous pouvons récupérer de façon pseudo-aléatoire un nombre entre 0 et la taille de la liste des Jobs Actifs(par exemple la liste des jobs actifs ressemblant à : job6000, job6000, job7000, job7000, job7000. tiré du fichier de configuration des jobs :
  {jobId: 6000,jobPriority : 3} {jobId: 7000,jobPriority : 2}.)
  un index qui nous permet de choisir un job aléatoirement tout en conservant les probabilités des priorités données pour les jobs. Pour récupérer une tâche non finis pour envoyer au client, une fois le job sélectionner le job regarde dans son bitSet de tâche finis de façon pseudo-aléatoire une tâche non finis et créer cette tâche avant de la renvoyer ce qui nous permet de garder un ManageTask avec un taille en mémoire raisonnable même avec beaucoup de jobs. Lorsque le serveur réceptionne un paquet réponse depuis un client, nous vérifions que la tâche récupéré est bien valide et non finis, et que ce json réponse contient bel et bien un réponse("Answer") ou une erreur("Error"), une fois tout cela vérifié et validé le manageTask copie la partie json de la réponse dans un fichier nommé comme suit : jobId(numéroDeTâche)nomDuClient.json (numéro de création du fichier depuis le lancement du serveur)
- Réutilisation des contexts pour éviter d'avoir à réallouer des contexts.

Ce qui fonctionne :

- Le client
- Le serveur
- La gestion des jobs/tâches
- Les logs
- Le fichier de configuration du serveur
- Le protocol
- L’écriture des réponses recues
- Le chargement des jobs/tâches depuis des fichiers

Ce qui ne fonctionne pas :

- A priori rien

Difficultés rencontrées :

- La purge du code, l’écriture du rapport, readme et de la javadoc en 24h

Soutenance bêta :

Amélioration supplémentaire :

- Réutilisation des Contexts

Points soulevés :

1 - Reprise de la reconnection côté client sous forme de lambda :

```
// In Class : upem.jarret.client.ClientJarRet.java
	@FunctionalInterface
	private interface RunnableIOException{
		public boolean run() throws IOException;
	}

	private boolean reconnectionIfFail(RunnableIOException r){
		long timeSleep = ConvertTimeUnit.secondsToMillis(2);
		int i = 0;
		while(i<3){
			i++;
			try{
				if(r.run())
					return true;
			}
			catch (IOException io){
				logErrorAndDisplayIfVerbose("Reconnection to server ...");
				logErrorAndDisplayIfVerbose(io.toString());
				resetConnection();
			}
			try {
				System.out.println("Reconnection tentative in " + timeSleep + "
seconds");
				Thread.sleep(timeSleep);
			} catch (InterruptedException ie) {}
			timeSleep *= 2;
			resetConnection();
		}
		return false;
	}
```

2 - Faire des états plus explicites côté serveur :

```
// In Class : upem.jarret.server.ServerjarRet
// Utilisation d'une énumération pour les états
private static enum State {
		READ_MORE_FOR_HEADER_NEEDED,
		READ_HEADER,
		TASK,
		RESULT,
		WRITE_FULL,
		READ_MORE_FOR_RESULT_NEEDED,
		NEED_CLOSED;
}

	class Context {
		private		SocketChannel sc;
		private		SelectionKey key;
		private		State state;
		private	boolean byteBufferOutIsFull = false;
		private final	int MAX_HEADER_CODE_SIZE =
Integer.max(createHeaderAnswer(AnswerKind.ERROR).remaining(),
createHeaderAnswer(AnswerKind.ANSWER).remaining());

		public void setNeedClosed(){ state = State.NEED_CLOSED; }

		private Context(SocketChannel sc, SelectionKey key) {
			readerServer = new HTTPReaderServer(bbIn);
			resetTimeRemainingBeforeClose();
			state = State.READ_HEADER;
			this.sc = sc;
			this.key = key;
		}

		private Context setContext(SocketChannel sc, SelectionKey key) {
			readerServer = new HTTPReaderServer(bbIn);
			resetTimeRemainingBeforeClose();
			state = State.READ_HEADER;
			this.sc = sc;
			this.key = key;
			return this;
		}

		private boolean needClosed(){ return state == State.NEED_CLOSED; }

		private void doRead() throws IOException{
			if(ByteBufferOutIsFull()){
```

```
                updateInterestOps();
                return;
        }
        switch(state){
        case READ_MORE_FOR_HEADER_NEEDED:
                state = State.READ_HEADER;
                int n = sc.read(bbIn);
                process();
                if(-1 == n)
                        throw new HTTPException();
                break;
        case READ_MORE_FOR_RESULT_NEEDED:
                state = State.RESULT;
                int ret = sc.read(bbIn);
                process();
                if(-1 == ret)
                        throw new HTTPException();
                break;
        default:
                process();
                break;
        }
        updateInterestOps();
}
private void doWrite() throws IOException {
        bbOut.flip();
        sc.write(bbOut);
        bbOut.compact();
        byteBufferOutIsFull = false;
        process();
}
private void updateInterestOps(){
        int ops = 0;
        if(!ByteBufferOutIsFull() && bbIn.hasRemaining() && !needClosed()){
                ops |=  SelectionKey.OP_READ;
        }
        if(bbOut.position() != 0 ){
                ops |= SelectionKey.OP_WRITE;
        }
        if(ops == 0){
                Context context = (Context) key.attachment();
                silentlyClose(key.channel(), log);
                return;
        }
        key.interestOps(ops);
}
```

```
private void process() throws IOException{
	switch(state){
	case READ_HEADER:
		if(ByteBufferOutIsFull()){
			return;
		}
		processHeader();
		break;
	case TASK:
		if(ByteBufferOutIsFull()){
			return;
		}
		processTask();
		break;
	case RESULT:
		if(ByteBufferOutIsFull()){
			return;
		}
		processResult();
		break;
	default:
		updateInterestOps();
		break;
	}
}
private void processHeader() throws IOException{
	if(header == null){
		Optional<HeaderClientToServer> headerOptional =
readerServer.readHeader();
		if(!headerOptional.isPresent()){
			log = new StringBuilder();
			state = State.READ_MORE_FOR_HEADER_NEEDED;
			return;
		}
		header = headerOptional.get();
	}
	switch(header.getTypeProtocol()){
	case "GET":
		state = State.TASK;
		process();
		return;
	case "POST":
		state = State.RESULT;
		process();
		return;
	default:
```

```
                        break;
                    }
                }
                private void processTask() throws IOException{
                    String addressServer = header.getHost();
                    Optional<Task> job = jobs.getOneTaskUnfinished();
                    ByteBuffer bbJson;
                    ByteBuffer head;
                    if(job.isPresent()){
                        bbJson = createJsonContent(job.get());
                    } else {
                        bbJson = CHARSET_UTF8.encode("{ \"ComeBackInSeconds\"
: \"" + serverParam.getComeBackTime() +"\" }");
                    }
                    head = createHeaderGiveTask(bbJson.remaining());
                    if(bbOut.remaining() < head.remaining() + bbJson.remaining()){
                        byteBufferOutIsFull = true;
                        return;
                    }
                    bbOut.put(head);
                    bbOut.put(bbJson);
                    header = null;
                    state = State.READ_HEADER;
                    process();
                }
                private ByteBuffer createHeaderGiveTask(int size){
                    return CHARSET_ASCII.encode(
                                "HTTP/1.1 200 OK\r\n"
                                + "Content-type: application/json; charset=utf-8\r\n"
                                + "Content-Length: " + size + "\r\n"
                                + "\r\n");
                }
                private void processResult() throws IOException{
                    if(bbOut.remaining() < MAX_HEADER_CODE_SIZE){
                        byteBufferOutIsFull = true;
                        return;
                    }
                    Optional<ByteBuffer> bb =
readerServer.readBytes(header.getContentLength());
                    if(!bb.isPresent()){
                        state = State.READ_MORE_FOR_RESULT_NEEDED;
                        return;
                    }
                    bb.get().flip();
                    long jobId = bb.get().getLong();
                    int taskValue = bb.get().getInt();
```

```
                        String json = CHARSET_UTF8.decode(bb.get()).toString();
                        if(JsonUtils.isJSONValid(json)){
                                if(jobs.endTask(jobId, taskValue, json))
                                        System.out.println("Task: " + taskValue +" from job:
"+jobId+" now is done !");
                                bbOut.put(createHeaderAnswer(AnswerKind.ANSWER));
                        } else {
                                bbOut.put(createHeaderAnswer(AnswerKind.ERROR));
                        }
                        header = null;
                        state = State.READ_HEADER;
                        process();
                }
                private ByteBuffer createHeaderAnswer(AnswerKind answer){
                        if(answer == AnswerKind.ANSWER)
                                return (CHARSET_ASCII.encode("HTTP/1.1 " +
HTTPProtocolCode.OK_CODE + " OK\r\n\r\n"));
                        else
                                return (CHARSET_ASCII.encode("HTTP/1.1 " +
HTTPProtocolCode.BAD_REQUEST_CODE + " Bad Request\r\n\r\n"));
                }
        }
```

3 - Ajout de l’état ByteBuffer de sortie est plein du côté serveur bloquer sa lecture
//

```
	private boolean ByteBufferOutIsFull(){ return byteBufferOutIsFull; }
	private void doRead() throws IOException{
			if(ByteBufferOutIsFull()){
				updateInterestOps();
				return;
			}
			switch(state){
			case READ_MORE_FOR_HEADER_NEEDED:
				state = State.READ_HEADER;
				int n = sc.read(bbIn);
				process();
				if(-1 == n)
					throw new HTTPException();
				break;
			case READ_MORE_FOR_RESULT_NEEDED:
				state = State.RESULT;
				int ret = sc.read(bbIn);
				process();
				if(-1 == ret)
					throw new HTTPException();
				break;
			default:
				process();
				break;
			}
			updateInterestOps();
		}
	private void doWrite() throws IOException {
			bbOut.flip();
			sc.write(bbOut);
			bbOut.compact();
			byteBufferOutIsFull = false;
			process();
		}
	private void updateInterestOps(){
			int ops = 0;
			if(!ByteBufferOutIsFull() && bbIn.hasRemaining() && !needClosed()){
				ops |=  SelectionKey.OP_READ;
			}
			if(bbOut.position() != 0 ){
				ops |= SelectionKey.OP_WRITE;
			}
			if(ops == 0){
				silentlyClose(key.channel(), log);
```

```
                return;
            }
            key.interestOps(ops);
        }
        private void process() throws IOException{

            switch(state){
            case READ_HEADER:
                if(ByteBufferOutIsFull()){
                    return;
                }
                processHeader();
                break;
            case TASK:
                if(ByteBufferOutIsFull()){
                    return;
                }
                processTask();
                break;
            case RESULT:
                if(ByteBufferOutIsFull()){
                    return;
                }
                processResult();
                break;
            default:
                updateInterestOps();
                break;
            }
        }
        private void processTask() throws IOException{
            String addressServer = header.getHost();
            Optional<Task> job = jobs.getOneTaskUnfinished();
            ByteBuffer bbJson;
            ByteBuffer head;
            if(job.isPresent()){
                bbJson = createJsonContent(job.get());
            } else {
                bbJson = CHARSET_UTF8.encode("{ \"ComeBackInSeconds\"
: \"" + serverParam.getComeBackTime() +"\" }");
            }
            head = createHeaderGiveTask(bbJson.remaining());
            if(bbOut.remaining() < head.remaining() + bbJson.remaining()){
                byteBufferOutIsFull = true;
                return;
            }
```

```
                bbOut.put(head);
                bbOut.put(bbJson);
                header = null;
                state = State.READ_HEADER;
                process();
        }

    private void processResult() throws IOException{
                if(bbOut.remaining() < MAX_HEADER_CODE_SIZE){
                        byteBufferOutIsFull = true;
                        return;
                }
                Optional<ByteBuffer> bb =
readerServer.readBytes(header.getContentLength());
                if(!bb.isPresent()){
                        state = State.READ_MORE_FOR_RESULT_NEEDED;
                        return;
                }
                bb.get().flip();
                long jobId = bb.get().getLong();
                int taskValue = bb.get().getInt();
                String json = CHARSET_UTF8.decode(bb.get()).toString();
                if(JsonUtils.isJSONValid(json)){
                        if(jobs.endTask(jobId, taskValue, json))
                                System.out.println("Task: " + taskValue +" from job:
"+jobId+" now is done !");
                        bbOut.put(createHeaderAnswer(AnswerKind.ANSWER));
                } else {
                        bbOut.put(createHeaderAnswer(AnswerKind.ERROR));
                }
                header = null;
                state = State.READ_HEADER;
                process();
        }
```