Projet Données Réparties
Un service de partage d'objets répartis et dupliqués en Java

Thierry Xu
Tom Bonetto

Département Sciences du Numérique - Deuxième année
2022-2023

Contents

List of Figures

1 Introduction

Le but de ce projet est d'illustrer les principes de programmation répartie. Pour ce faire, nous allons réaliser sur Java un service de partage d'objets par duplication, reposant sur la cohérence à l'entrée (entry consistency). Les applications Java utilisant ce service peuvent accéder à des objets répartis et partagés de manière efficace puisque ces accès sont en majorité locaux (ils s'effectuent sur les copies (réplicas) locales des objets). Durant l'exécution, le service est mis en œuvre par un ensemble d'objets Java répartis qui communiquent au moyen de Java/RMI pour implanter le protocole de gestion de la cohérence. Le service est architecturé comme suit :

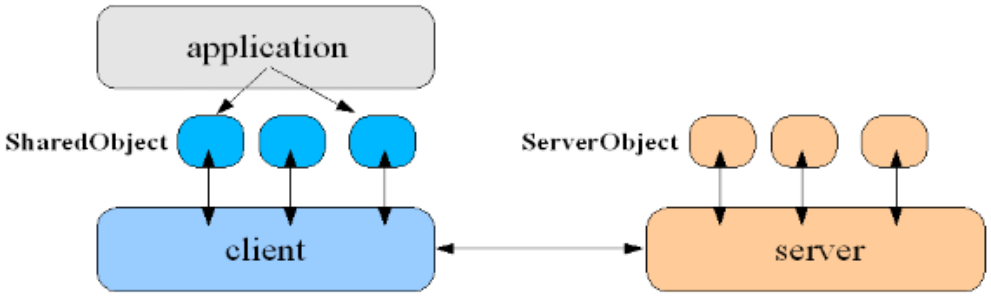

Figure 1: Architecture du service

L'utilisation d'un objet partagé se fait toujours avec une indirection à travers un objet de classe SharedObject. Cette classe fournit notamment des méthodes lock_read(), lock_write() et unlock() qui permettent de mettre en œuvre la cohérence à l'entrée depuis l'application. Notons que dans le cadre de ce service, on ne gère pas les prises de verrou imbriquées ; un lock_read() ou lock_write() sur un objet doit être suivi d'un unlock() pour pouvoir reprendre un verrou sur le même objet. Un SharedObject contient notamment un entier (id) qui est un identifiant unique alloué par le système (ici un serveur centralisé) à la création de l'objet, ainsi qu'une référence à l'objet lorsqu'il est cohérent (obj).
La couche (classe) appelée Client fournit les services pour créer ou retrouver des objets dans un serveur de noms (comme le Registry de RMI) :

- static void init() : initialise la couche cliente, à appeler au début de l'application.
- static SharedObject create (Object o) : permet de créer un objet partagé (en fait un descripteur) initialisé avec l'objet o. Le descripteur de l'objet partagé est retourné. A la création, l'objet n'est pas verrouillé.
- static SharedObject lookup (String n) : consulte le serveur de nom et retourne l'objet partagé enregistré.
- static void register (String n, SharedObject_itf so) : enregistre un objet partagé dans le serveur de noms.

Le fichier Irc.java vous donne un exemple d'application utilisant un objet partagé de classe Sentence (mais elle n'utilise pas des structures complexes d'objets partagés comme des graphes). Cette application sera utilisée pour tester (dans un premier temps) votre projet, mais vous devrez implanter d'autres jeux de test.

2 Implantation du service

Les SharedObject, qui sont utilisés par les programmes pour tous les accès aux objets, contiennent des informations sur l'état des objets du point de vue de la cohérence. On y trouve notamment une variable entière lock qui signifie :

- NL : no local lock
- WLC : write lock cached
- RLT : read lock taken
- WLT : write lock taken
- RLT_WLC : read lock taken and write lock cached

Un verrou est dit taken s'il est pris par l'application. Il sera libéré au prochain unlock() et passera alors à l'état cached. Le verrou est dit cached s'il réside sur le site sans être pris par l'application. Un verrou cached peut alors être pris par l'application sans communication avec le serveur. L'état hybride RLT_WLC correspond à une prise de verrou en lecture par l'application alors que le SharedObject possédait un WLC. En fonction de l'état de l'objet sur le site client, la demande d'un verrou nécessitera (ou pas) de propager un appel au serveur.

Pour mettre en œuvre la cohérence, les méthodes lock_read() et lock_write() de la classe SharedObject ont besoin d'appeler des méthodes du serveur qui gèrent la cohérence. Ces appels passent par la couche cliente (classe Client) qui fournit les méthodes :

- static Object lock_read (int id) : demande d'un verrou en lecture au serveur, en lui passant l'identifiant unique de l'objet (le SharedObject devait être en NL). Retourne l'état de l'objet.
- static Object lock_write (int id) : demande d'un verrou en écriture au serveur, en lui passant l'identifiant unique de l'objet (le SharedObject devait être en NL ou RLC). Retourne l'état de l'objet si le SharedObject était en NL.

Ces méthodes (statiques) de la classe Client ne font que propager ces requêtes au serveur, en ajoutant aux paramètres de la requête une référence au client (instance) lorsque cela est nécessaire. L'interface du serveur (distante) comprend donc les méthodes suivantes (d'autres étant ajoutées, notamment pour implanter le service de nommage évoqué ci-dessus) :

- Object lock_read(int id, Client_itf client)
- Object lock_write(int id, Client_itf client)

Ces méthodes incluent une référence au client afin de pouvoir le rappeler ultérieurement. Sur le site serveur, la gestion de la cohérence d'un objet est attribuée à une instance de la classe ServerObject. La classe ServerObject indique l'état de la cohérence de l'objet du point de vue du serveur, avec notamment le client écrivain si l'objet est en écriture ou la liste des clients lecteurs si l'objet est en lecture (afin d'être en mesure de propager des invalidations). Les appels au serveur sont transférés au ServerObject concerné, ce ServerObject implantant une interface similaire:

- Object lock_read(Client_itf client)
- Object lock_write(Client_itf client)

La référence au client reçue par le serveur lui permet de réclamer un verrou et une copie de l'objet au client qui la possède. L'interface du client (distante) permettant cette réclamation est la suivante :

- Object reduce_lock(int id) : permet au serveur de réclamer le passage d'un verrou de l'écriture à la lecture.
- void invalidate_reader(int id) : permet au serveur de réclamer l'invalidation d'un lecteur.
- Object invalidate_writer(int id) : permet au serveur de réclamer l'invalidation d'un écrivain.

Une telle réclamation est propagée au descripteur d'objet concerné (SharedObject) sur le site client qui implante les mêmes méthodes:

- Object reduce_lock()
- void invalidate_reader()
- Object invalidate_writer()

Les appels de méthodes d'un client au serveur et du serveur à un client sont représentés sur la figure suivante.

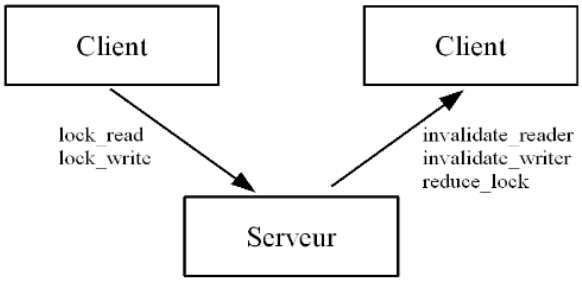

Figure 2: Appels de méthodes d'un client au serveur et du serveur à un client

On peut résumer les méthodes de verrouillage et d'invalidation avec les 2 figures suivantes :

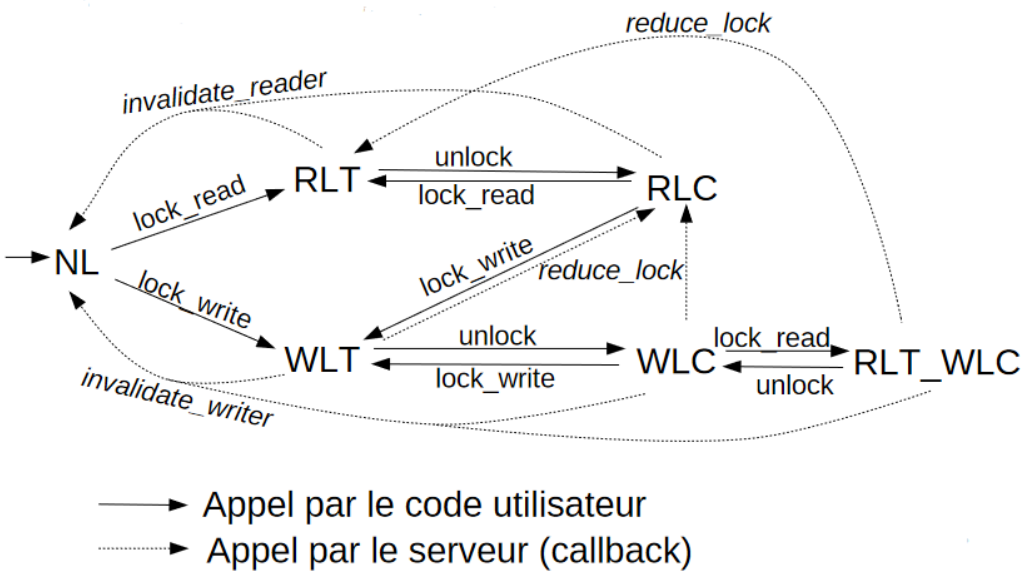

Figure 3: Point de vue client pour un objet partagé

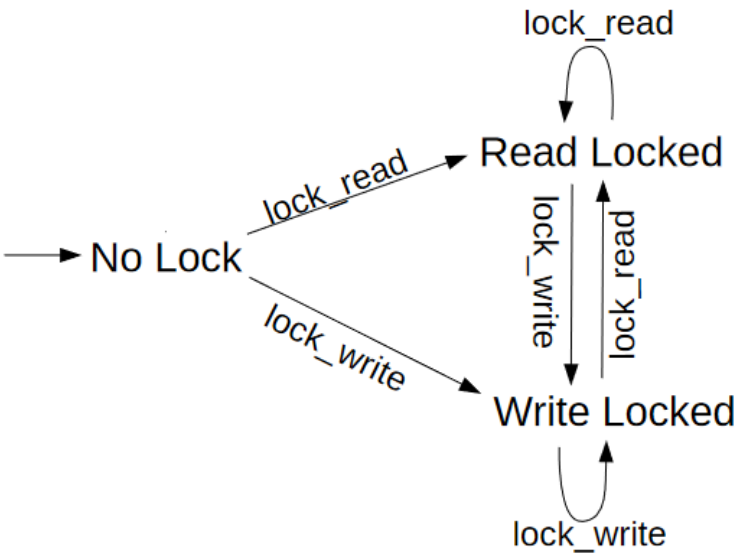

Figure 4: Point de vue serveur pour un objet partagé

3 Travail à effectuer

3.1 Etape 1

On va implanter le service de gestion d'objets partagés répartis. Dans cette première version, les SharedObject sont utilisés explicitement par les applications. Plusieurs applications peuvent accéder de façon concurrente au même objet, ce qui nécessite de mettre en œuvre un schéma de synchronisation globalement cohérent pour le service que nous implantons. On suppose que chaque application voulant utiliser un objet en récupère une référence (à un SharedObject) en utilisant le serveur de nom. On ne gère pas le stockage de référence (à des objets partagés) dans des objets partagés.

3.2 Etape 2

On désire soulager le programmeur de l'utilisation des SharedObject. On doit donc implanter un générateur de stubs. Nous prenons les hypothèses suivantes:

- un objet partagé est une classe sérialisable (par exemple Sentence). Il s'agit de la classe métier de l'objet partagé.
- l'objet partagé est utilisable à partir de variables de type une interface (par convention, l'interface pour utiliser un objet partagé de classe Sentence s'appellera Sentence_itf) qui inclut les méthodes métier de Sentence auquel on ajoute les méthodes de verrouillage en héritant de SharedObject_itf. Mais Sentence n'implémente pas Sentence_itf, car la classe métier ne définit pas les méthodes de verrouillage (c'est le stub qui le fait).
- stub est généré, appelé Sentence_stub. Ce stub hérite de SharedObject (donc des méthodes de verrouillage) et il implémente l'interface Sentence_itf.

3.3 Etape 3

On désire prendre en compte le stockage de référence (à des objets partagés) dans des objets partagés. Le problème qui se pose est la copie d'un objet partagé O1, qui inclut une référence à un objet O2, entre deux machines M1 et M2. O1 inclut une référence au stub de O2 sur la machine M1. Après la copie, il faut que la copie de O1 inclut la référence au stub de O2 sur la machine M2. Ceci nécessite d'adapter les primitives de sérialisation des stubs pour que, lorsqu'un stub est sérialisé sur M1, on ne copie pas l'objet référencé et lorsqu'il est désérialisé sur M2, le stub soit installé de façon cohérente sur M2 (sans installer plusieurs stubs pour un même objet sur la même machine). Nous exploiterons (spécialiser) la méthode Object readResolve().

4 Architecture

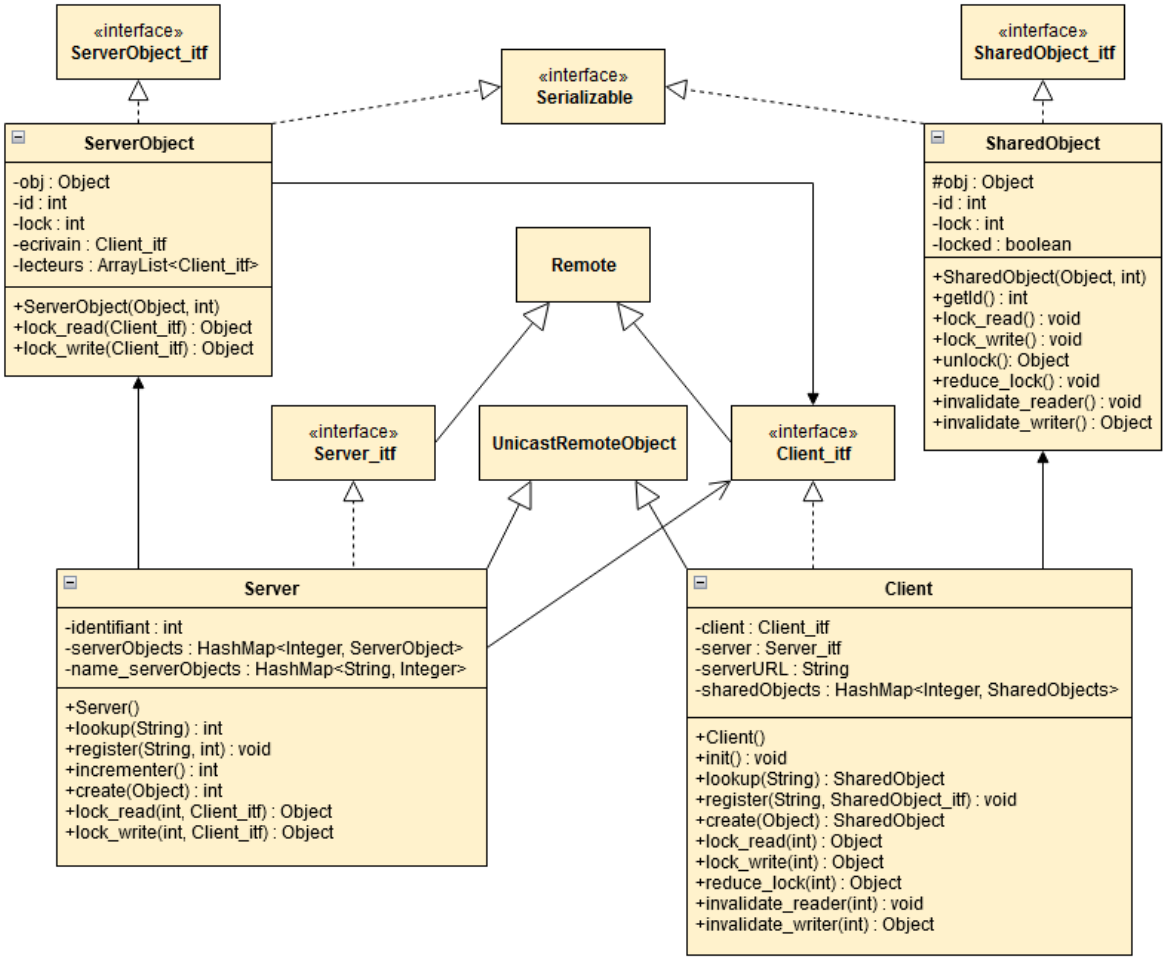

Figure 5: Diagramme de classe - UML

4.1 SharedObject et SharedObject_itf

Pour ce qui est de la classe `SharedObject`, elle implémente l'interface `Serializable` pour que les objets partagés qui hérite de `SharedObject` soit des classe sérialisable (par exemple Sentence). Elle implémente évidemment l'interface `SharedObject_itf` et instancie donc les méthode lock_read(), lock_write(), et unlock(). Ces différentes méthodes ont été implémentées grâce aux différents cas de figure présents sur les slides de présentation du PDR et également grâce à la figure 3. La classe possède un attribut locked qui est un booléen permettant de savoir si le verrou est tenu. Savoir si le verrou est tenu ou non, nous permet de faire appel à wait() lors d'une réclamation. Pour traiter les différents états de verrouillage, nous avons créé des constantes entières (NL = 0, RLC = 1, etc), ce qui permet grâce à une structure switch-case de traiter facilement les différents cas sans avoir du code trop chargé. Des appels à notify() dans unlock(), invalidate_reader(), invalidate_writer(), reduce_lock() permettent de notifier les différentes méthodes qui étaient en attente. Nous avons également rajouté un getter sur l'attribut identifiant.

4.2 ServerObject et ServerObject_itf

La classe `ServerObject` implémente l'interface `ServerObject_itf` et instancie donc les méthodes lock_read(Client_itf) et lock_write(Client_itf), ces deux méthodes ont été notamment réalisées grâce à la figure 4. Comme dit précédemment, la classe `ServerObject` indique l'état de la coherence de l'objet du point de vue serveur, grâce au client écrivain si l'objet est en écriture et grâce à la liste des clients lecteurs si l'objet est en lecture. La classe possède donc un attribut ecrivain de type Client_itf et un attribut lecteurs qui est un ArrayList de Client_itf.

4.3 Server et Server_itf

La classe `Server` implémente l'interface `Server_itf` et instancie donc ses différentes méthodes. Le server propose deux objets accessibles en RMI, une HashMap contenant les objets partagés au niveau du serveur associés à leur identifiant et une deuxième HashMap associant aux identifiants des objets partagés le nom du serveur. Dans le main, le serveur initialise son registre et l'associe à son URL pour que le client puisse y accéder. La méthode incrementer permet de donner des identifiants uniques aux différents objets créés dans la méthode create(Object).

4.4 Client et Client_itf

La classe `Client` implémente l'interface `Client_itf` et instancie donc les méthodes reduce_lock(int), invalidate_reader(int) et invalidate_writer(int). La classe possède également une HashMap pour stocker les objets partagés au niveau Client. Lors de l'initialisation du client, on initialise à la fois la HashMap mais également le serveur en récupérant ses informations via la méthode lookup de Naming.

5 Synchronisation

Les clients effectuent des lectures et des écritures sur des objets partagés, il faut donc mettre en place une concordance pour éviter les conflits d'accès. Une mauvaise gestion de la synchronisation entre blocs peut mener à une situation de blocage total d'une application, appelée deadlock .

Un deadlock intervient lorsqu'un premier thread T1 se trouve dans un bloc synchronisé B1, et est en attente à l'entrée d'un autre bloc synchronisé B2. Malheureusement, dans ce bloc B2, se trouve déjà un thread T2, en attente de pouvoir entrer dans B1. Ces deux threads s'attendent mutuellement, et ne peuvent exécuter le moindre code.

Pour éviter cela, on peut utiliser le mot clé synchronized qui permet de poser un verrou exclusif sur une portion de code : ceci permet de garantir que les accès à une ressource partagée ne

se feront pas en concurrence. La JVM garantit qu'un bloc de code déclaré synchronized ne sera exécuté que par un seul thread à un instant T sous réserve que le moniteur utilisé pour le verrou soit le même pour tous les threads. L'instruction synchronized permet de demander l'obtention exclusive du moniteur pour le thread courant. Les autres threads qui tenteront d'acquérir le moniteur devront attendre leur tour tant qu'un thread le possède déjà. Le thread conserve le moniteur jusqu'à la fin de l'exécution du bloc de code.

Nous avons donc utilisé le mot-clé synchronized pour les différentes méthodes sensibles aux conflits, c'est-à-dire les méthodes des classes `SharedObject` et `ServerObject`. Et nous avons ensuite créé des Irc un peu particulier pour pouvoir tester le bon fonctionnement de notre synchronisation : `Spamming_writer`, `Spamming_reader` et `Spamming_client`. Le premier correspond à un écrivain qui lit constamment. Le second est un lecteur qui peut soit lire en boucle rapidement, soit lire toutes les secondes. Le choix se fait à travers deux boutons sur la fenêtre. Quant à la dernière classe, il s'agit d'une fusion des deux: un client qui écrit et lit constamment selon les boutons.

Pour tester la synchronisation au niveau de l'écrivain, nous avons par exemple lancé six écrivains qui ne s'arrêtent pas d'écrire et un lecteur qui lit toutes les secondes de sorte à pouvoir visualiser correctement sur la fenêtre. On remarque qu'il n'y a pas d'interblocage et que le lecteur reçoit des messages de tous les différentes écrivains.
Au niveau du lecteur, nous avons lancé plusieurs lecteurs et un écrivain qui écrit constamment. Nous n'avons pas remarqué d'anomalie et les lecteurs reçoivent bien tous les messages de l'écrivain. Comme un client est une personne qui peut en réalité à la fois écrire et lire, nous avons également effectué des tests en ajoutant un `Spamming_client`. À nouveau, nous n'avons rencontré aucun interblocage.

6 Transparence d'accès aux objets

Après avoir testé le bon fonctionnement du programme avec Irc et le bon fonctionnement de la synchronisation avec les différents Irc modifiés. Nous pouvons passer au travail de l'étape 2, qui consiste à implanter un générateur de stub et à modifier les méthodes de la classe `Client` pour que celle-ci utilise les objets partagés de la classe `Sentence`. Pour le générateur de stub nous avons créé une classe `StubGenerator` qui utilise l'introspection de java pour récupérer les différentes méthodes de la classe passée en paramètre. Ainsi, un appel à `StubGenerator` avec Setence en paramètre fournira une classe `Setence_stub`. Nous avons validé notre générateur en comparant son résultat avec la classe `Setence_stub` qui nous était fourni dans le dossier dédié à l'étape 2. Maintenant, que nous avons créé le stub, il faut mettre en place la transparence d'accès aux objets. Pour cela il faut modifier les méthodes create() et lookup() de la classe `Client` afin qu'il ne manipule plus directement le constructeur de `SharedObject`. Nous avons donc créé une méthode StubCreator qui prend en paramètre un objet et un entier et retourne l'instance du SharedObject créé, elle sera appelée par lookup() et create(). Nous avons réeffectué les tests de l'étape 1 pour vérifier que le fonctionnement n'avait pas changé.

7 Stockage de référence dans des objets partagés

Cette partie du projet est restée assez floue. Malgré les explications du sujet et nos recherches personnelles sur la question, nous n'avons pas réussi à cerner suffisamment ce qui était demandé pour faire une bonne implémentation. Nous avons tout de même tenté d'implanter les méthodes readResolve() et writeObject() dans la classe `SharedObject`. Au moment de la désérialisation d'un objet, la méthode readObject() vérifie en interne si l'objet en cours de désérialisation possède la méthode readResolve() implémentée. Si la méthode readResolve() existe, elle sera invoquée, de même pour writeObject() lors de la sérialisation d'un objet. À cause du manque de recul sur cette partie, nous n'avons pas réellement pu tester.

8 Conclusion

De manière générale, ce projet nous a permis de mettre en œuvre les différentes notions et aspects que nous avions appris de Système Concurrent et d'Intergiciel. N'étant pas forcément très à l'aise avec ces matières, ce projet représentait un réel challenge pour nous. La difficulté majeure rencontrée a été de comprendre le projet et ce qui était attendu dans chacune des étapes. La synchronisation nous a donné beaucoup de fil à retordre, il est quasi impossible de trouver un test sur lequel on serait sûr et certains qu'il garantit le bon fonctionnement de notre programme, les tests que nous avons élaborer semble fonctionner, mais nous ne sommes jamais à l'abri d'une surprise. D'un point de vue technique, nous avons mis en place un dépôt git pour pouvoir travailler à distance facilement et nous avons dû nous familiariser davantage avec les classes de Java qui permettent l'élaboration de serveurs RMI, de synchronisation, d'écriture et de lecture de données, de systèmes concurrents. Ce sont ses détails qui nous permettent de progresser et d'améliorer la qualité de notre travail.