# PDR S8 : Un service d'objets partagés, répartis, dupliqués... et robustes

2SN

11 mars 2023

## 1 Contexte, objectif et terminologie

On se place dans la perspective d'un service réparti, où les applications sont constituées d'un ensemble de processus coordonnés et déployés sur plusieurs machines distinctes (sites), reliées par un réseau asynchrone. Un réseau asynchrone est un réseau où le temps d'acheminement d'un message est fini, mais non borné.

Dans ce contexte, on se propose d'étudier des protocoles de cohérence pour des objets partagés, implantés par un ensemble de copies réparties. Le principe est que les différents sites disposent chacun d'une copie locale de l'objet partagé. Le protocole de cohérence doit assurer que les copies sont « équivalentes » : chaque copie reflète la même valeur, qui est celle de l'objet (virtuellement) partagé. Maintenir en permanence une égalité stricte entre copies s'avère extrêmement coûteux : il faudrait diffuser et synchroniser chaque mise à jour. Différents protocoles de cohérence existent, qui permettent de réduire le coût en relâchant cette contrainte d'égalité stricte, au prix d'une dégradation (en temps ou en contenu) des accès. Par exemple, le protocole développé au semestre précédent n'impose pas que toutes les copies aient en permanence la même valeur : il suit une politique « paresseuse » et ne met une copie à jour que lorsqu'un accès à cette copie est demandé. Le gain en termes de messages est appréciable, mais les accès sont ralentis dans le cas où la mise à jour est nécessaire.

Plus précisément le présent projet devra

- compléter le protocole développé au semestre précédent (appelé PODP par la suite), pour permettre de synchroniser la mise à jour d'un ensemble de copies.
- concevoir un (nouveau) protocole assurant une gestion robuste (c'est-à dire résistante aux pannes) des objets dupliqués.

**Terminologie** Par la suite, les termes de site, de processus lourd, ou de JVM seront employés comme des synonymes. La nuance se situe uniquement selon le point de vue (site est orienté vers la modélisation, processus vers l'exécution, et JVM vers l'implémentation)

## 2 Échauffement : de l'IRC à la cohérence continue

### 2.1 Etape 1 : rendre l'IRC actif

L'application IRC n'est pas pratique pour gérer des conversations en temps réel. Telle quelle, un utilisateur (ou l'application) doit scruter (lire) fréquemment et régulièrement l'objet partagé contenant les messages échangés. On souhaite donc compléter la gestion des objets

dupliqués en offrant aux applications la possibilité de suivre les changements d'état d'un objet partagé. Pour cela, il faudra implanter un schéma publier/s'abonner au niveau du serveur, c'est-à-dire qu'il faudra

- étendre l'interface au niveau du serveur en offrant les méthodes permettant à un client
  - de s'abonner et de se désabonner aux changements d'état (externes) d'un objet partagé,
  - et de notifier au serveur ses changements d'états locaux
- que le client fournisse un objet de rappel pour la méthode d'abonnement, qui lui permettra de traiter le changement d'état.
- que le serveur gère une liste d'abonnés qu'il rappelle à chaque nouvelle écriture

**Notes**

- L'extension apportée au service de gestion d'objets dupliqués devra être minimale. En particulier, le traitement effectué par un abonné suite à la notification d'un changement d'état est laissé à la charge du client. Le traitement naturel sera sans doute de demander une lecture de la nouvelle valeur, mais la décision de transférer la valeur revient au client et ne doit pas être automatiquement prise par le serveur.
- La conception devrait vous amener à réfléchir et à faire des choix sur un certain nombre de questions. En particulier
  - L'opération unlock() doit-elle être bloquante ?
  - La méthode de rappel (chez le client) peut être bloquante. Est ce un inconvénient ? Si oui, comment le traiter, si non pourquoi ?
  - Les choix que vous ferez devront explicitement être argumentés et documentés.

**Livrables** (10 jours) Vous devrez déposer une archive **au format tar**, contenant

- Un bref rapport, (1 à 2 pages) précisant
  - Les algorithmes et protocoles de principe pour les modifications et ajouts apportés.
  - Les choix de conception **motivés et argumentés**, en particulier par rapport aux points soulevés dans le paragraphe précédent
- Le code source de l'implémentation du protocole étendu avec la mise en place du schéma publier-s'abonner. Il est recommandé de s'appuyer sur la version de PODP obtenue à l'issue de l'étape 1, afin de limiter les risques d'erreur.
- Le code source de la version adaptée de l'application IRC
- Une démonstration : un bref compte rendu écrit et éventuellement illustré voire une petite vidéo au format mp4 (attention au poids).

## 2.2 Etape 2 : gestion active des copies

Le PODP vise à réduire le plus possible le nombre d'interactions et le volume de données échangées. Il est particulièrement adapté lorsque les écritures et les lectures fréquentes et **répétées** sur un **même** site. En effet :

- la valeur à jour reste locale au(x) dernier(s) site(s) l'ayant utilisée
  → en l'absence d'autres demandes, les accès à une même copie locale ne demandent aucune interaction avec le serveur ou les autres copies
- la copie à jour n'est échangée, et/ou une synchronisation effectuée uniquement lorsque cela est nécessaire, c'est à dire lorsqu'un site ne disposant pas de la copie à jour demande un accès à l'objet partagé. Dans ce cas, la valeur à jour transite par le serveur, puis est

transmise au site demandeur. Les échanges sont donc limités au site détenant la copie à jour, au serveur et ou site demandeur.

Ce protocole très économe en termes d'échanges est moins intéressant dans le cas d'écritures et de lectures **dispersées**, sur des objets **volumineux** :

- la mise à jour des copies est asynchrone, à l'initiative des clients
  - il n'est donc pas adapté lorsqu'il faut coordonner des lectures et des écritures concurrentes sur plusieurs sites. L'IRC illustre bien ce cas.
  - les opérations de lecture demandent un délai lorsque le cache est invalide, du fait d'écritures sur d'autres sites.
- le stockage des copies sur le serveur en fait un goulot d'étranglement et limite la capacité de stockage pour les objets partagés

On se propose donc de compléter le PODP pour répondre simplement aux situations où les copies doivent être mises à jour de manière coordonnée et réactive (cohérence continue) : ils s'agit de permettre une mise à jour systématique des copies locales après chaque écriture, pour les sites ayant besoin d'une vue synchrone de l'objet partagé.

Les principales modifications à apporter à l'interface/au protocole seront une adaptation simple de l'extension au PODP développée pour l'IRC :

- les sites ont la possibilité de s'abonner/se désabonner à une vue synchrone de l'objet partagé (opérations track()/leave_track())
- pour les sites abonnés, read() retourne **instantanément** la **copie locale**, sans qu'il soit besoin d'échanger avec le serveur **au moment du read()**
- write() transmet systématiquement la copie à jour aux sites abonnés : lors du unlock(), le serveur diffuse cette copie à l'ensemble des sites abonnés (rappel des clients abonnés, provoquant la mise à jour de leur verrou et de leur copie locale)

**Notes**

- Le schéma lecteur rédacteurs est maintenu par cette extension de PODP.
- Le serveur doit nécessairement attendre la fin de la diffusion aux abonnés avant de libérer l'écrivain, afin de
  - maintenir la cohérence des copies (il faut que les écritures soient faites en séquence, pour éviter que
  - garantir que les abonnés disposent de la dernière valeur à jour (ou de la valeur en cours de mise à jour)
- Il n'est pas demandé de traiter la question de la limitation de capacité induite par le stockage des objets sur le serveur. Cette question pourrait être traitée facilement en conservant seulement sur le serveur l'identifiant du site disposant de la copie à jour, afin de pouvoir récupérer et transmettre cette copie à la demande (ou mieux : afin de demander au site disposant de la copie de la transmettre directement au site qui la demande). Ce traitement améliorerait clairement les performances, mais il alourdirait (un peu) le protocole et surtout est en dehors du sujet central du projet, qui est la gestion de la cohérence d'objets dupliqués.

**Livrables (10 jours)** Vous devrez déposer une archive **au format tar,** contenant

- Un petit rapport, (2 à 4 pages) comportant
  - les structures de données introduites ou modifiées
  - le protocole et l'algorithmique de principe mis en jeu pour réaliser cette extension

  - une justification claire et synthétique des choix de conception opérés
  - une évaluation **qualitative** du surcoût induit par l'utilisation de la cohérence continue. Il n'est pas demandé de réaliser de réaliser une étude comparative basée sur des mesures de performances effective. Il est simplement demandé d'évaluer et de comparer de manière synthétique la complexité des algorithmes mis en place, en termes de :
    - nombre de messages
    - volumes de données échangées
    - surcoût d'exécution (opérations et délais de synchronisation)

    Cette évaluation doit aboutir à un bilan, comparaison qualitative entre le mode paresseux (cohérence ponctuelle) et le mode actif (cohérence continue)
- Le code source de l'implémentation du protocole étendu avec la mise en place de la cohérence continue pour les sites abonnés.
- Le code source de la version adaptée de l'application IRC
- Une démonstration : un bref compte rendu écrit et éventuellement illustré voire une petite vidéo au format mp4 (attention au poids).

# 3 Tolérance aux pannes

Le protocole PODP (ainsi que les extensions réalisées pour la partie précédente) présente un certain nombre de caractéristiques essentielles, qui vont être revues ou au contraire conservées pour la suite :

- Le contrôle de la cohérence s'appuie sur l'utilisation d'objets de synchronisation bloquants au niveau des applications : les accès aux objets partagés doivent être encadrés par la prise et la libération de verrous associés à l'objet. Ce parenthésage est explicite pour l'étape 1 du PODP ; les étapes suivantes permettent de le masquer pour l'application, mais le code généré pour les stubs conserve l'encadrement des accès parenthésage explicite par lock_read (ou lock_write) et unlock().
- Les verrous implémentés par le PODP réalisent un schéma lecteurs-rédacteurs (exclusion mutuelle entre tout rédacteur et tout autre processus)
- Ce schéma lecteurs-rédacteurs procure une forme de cohérence naturelle en copies : les écritures se font en séquence, et une lecture fournit toujours la dernière valeur écrite.
- Enfin, les pannes ne sont pas gérées par le protocole PODP (et ses extensions). Or lsa robustesse est plutôt limitée.
- pour la cohérence ponctuelle (PODP) : le protocole bloque sans issue en cas de panne du serveur ou (possiblement) du site ayant effectué la dernière mise à jour
- pour la cohérence continue : le protocole bloque en cas de panne du serveur ou d'un abonné

Le but essentiel de cette dernière partie du projet est de définir et implémenter un protocole de cohérence robuste, qui puisse fonctionner correctement, même si une partie des sites est défaillante.

- Le point clé est d'éviter que des processus fonctionnant correctement (sites corrects) ne se trouvent bloqués car en attente de réponses de sites tombés en panne (sites défaillants).
- Une conséquence (a priori bénéfique pour les performances) est que le schéma de synchronisation lecteurs-rédacteurs va devoir être abandonné : il ne faut pas qu'un rédacteur ayant l'accès exclusif bloque tous les autres processus en tombant en panne. Autrement dit, le protocole va devoir gérer le fait que les accès en lecture et en écriture peuvent être

concurrents, tout en garantissant le même niveau de cohérence que celui qui est obtenu avec le schéma lecteurs-rédacteurs.
- Une conséquence de cette conséquence est que le protocole tolérant aux pannes sera radicalement différent du protocole précédent

Cependant, si le protocole est entièrement différent, l'architecture générale du service, ainsi que les structures de données nécessaires resteront sensiblement les mêmes :
- l'architecture de base reste une architecture client-serveur
- le serveur gère un ServerObject par objet partagé et assure un service de nommage identique à celui du protocole non robuste.
- les clients gèrent un SharedObject par copie locale d'objet partagé
- les requêtes d'accès transiteront de l'application à la couche Client, puis (posiblement) à la couche Serveur, puis au ServerObject (et retour éventuel)
- les mises à jour des copies seront réalisées par rappel des SharedObject depuis le serveur

La prochaine section définit les objets nécessaires à la gestion de la cohérence en autorisant des accès concurrents en lecture et écriture. La section suivante précise la notion de panne, ainsi que les critères de correction attendus pour le protocole. La dernière section fournit des précisions, indications et suggestions utiles pour l'implémentation du protocole tolérant aux pannes.

### 3.1 Accès concurrents et cohérents à un objet partagé

Dans le contexte d'un service d'accès à un objet partagé implanté par un ensemble de copies dupliquées, la notion de registre permet de caractériser les propriétés attendues pour les protocoles gérant la cohérence entre les différentes copies. Un registre est un objet partagé R qui fournit 2 opérations élémentaires :
- écriture : R.write(v), qui affecte la valeur v au registre
- lecture : R.read(), qui retourne la valeur courante du registre

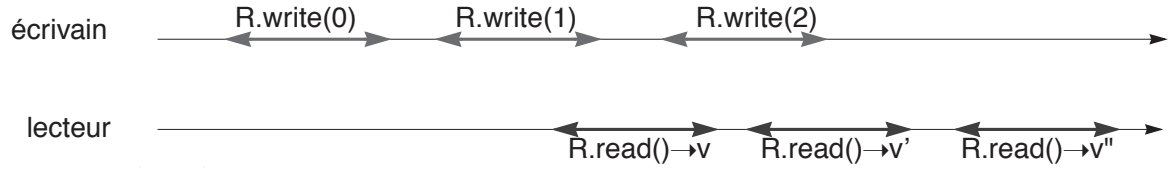
**Remarques**
- Il faut bien noter que les opérations de lecture et d'écriture ont une durée non négligeable : la réalisation d'un accès peut entraîner des échanges entre sites pour récupérer ou mettre à jour les copies dupliquées du registre. Par conséquent, il est possible qu'il y ait un chevauchement entre les périodes d'exécutions de 2 accès demandés par 2 sites distincts. Dans ce cas les accès sont dits concurrents.
- Il faut d'autre part conserver à l'esprit que les opérations de lecture et d'écriture devraient rester non bloquantes, dans tous les cas : un site ne doit pas pouvoir monopoliser indéfiniment l'accès au registre (ou, dit autrement, tout site qui demande un accès doit obtenir une réponse au bout d'un temps fini).
- Différentes sortes de registres ont été définis, correspondant à différentes formes de cohérence. Les registres utiles pour la suite du projet sont les registres réguliers et les registres atomiques.

### 3.1.1 Registre régulier

Un registre régulier gère les accès concurrents entre un unique écrivain et un ensemble de lecteurs. Dans ce cadre, le résultat d'une lecture est

- la dernière valeur écrite ou la valeur de l'une des écritures en cours, si la lecture est concurrente avec une ou plusieurs écritures.
- sinon, la dernière valeur écrite

**Remarque** : un registre régulier tolère des exécutions « contre-intuitives », où il est possible d'observer des « inversions de valeur ». Ainsi dans l'exemple suivant, le lecteur pourrait observer successivement que le registre R prend les valeurs 2, puis 1, puis 2, ce qui ne correspond pas à l'ordre des écritures réalisées au niveau de l'écrivain (0, puis 1, puis 2).




$v \in \{0,1,2\}$ / $v' \in \{1,2\}$ / $v'' = 2$

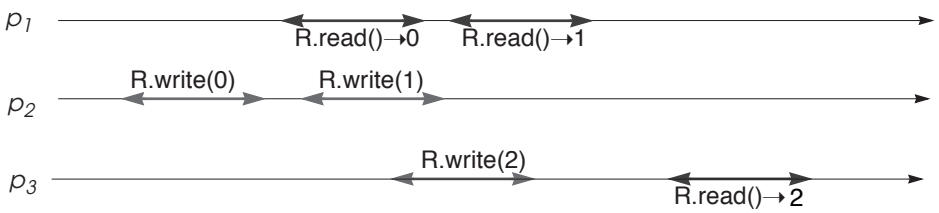
### 3.1.2 Registre atomique

Un registre atomique gère les accès concurrents entre un ensemble d'écrivains et un ensemble de lecteurs. (Tout site peut être alternativement lecteur et écrivain). Dans ce cadre, le protocole assure que les accès sont linéarisables, c'est-à-dire que pour tout ensemble d'accès concurrents, il existe un entrelacement S de ces accès

(i) qui donne le même résultat final, pour ce qui est de la valeur du registre

(ii) qui respecte la chronologie des opérations non concurrentes, c'est-à-dire que si un accès a1 se termine avant le début d'un accès a2, alors a1 précède a2 dans S.

(iii) tel que tout lecture de S renvoie la dernière valeur écrite dans S (légalité)




**Exemple**

S = R.write(0) ;R.read()→0 ; R.write(2) ; R.write(1) ;R.read$_1$()→1 ; R.read$_3$()→2

**Remarques**

- La condition (i) est analogue au critère de sérialisabilité pour les transactions, la différence étant que la condition (i) considère les accès élémentaires, alors que la sérialisabilité considère des blocs d'instructions vus comme atomiques.
- C'est ainsi le niveau de cohérence assuré par le schéma lecteurs-rédacteurs.

### 3.2 Tolérance aux pannes : définitions et hypothèses

La principale motivation du protocole recherché est de permettre aux sites corrects de fonctionner normalement, même s'il existe un certain nombre de sites défaillants. Selon la complexité des pannes considérées, le protocole mis en œuvre sera lui même plus ou moins complexe et les hypothèses nécessaires à son bon déroulement seront plus ou moins fortes. Pour ce projet, nous considèrerons uniquement des pannes franches, qui sont les pannes les plus simples à traiter, mais qui couvrent une large classe de situations.

Dans le modèle des pannes franches,

- Un site en panne est un site qui n'émet plus de message et n'en émettra plus. Ce site est et reste définitivement isolé des autres sites. Tout se passe donc comme s'il était éteint pour ne jamais repartir.

  Un site en panne peut être terminé (définitivement arrêté) ou encore bloqué indéfiniment
- Les serveurs/sites qui répondent envoient des réponses correctes (non altérées et conformes au comportement attendu par l'application)

**Remarque** Cela implique tout particulièrement qu'un site indisponible ne doit pas empêcher les opérations de lecture ou d'écriture de se terminer. Leslie Lamport formule cette contrainte de manière plaisante : "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"

On supposera en outre (et enfin) que le réseau de communication est fiable : il ne perd pas de messages, ne crée pas de messages qui n'aient pas été envoyés et ne modifie jamais le contenu des messages acheminés. Par contre, le réseau est asynchrone, c'est-à-dire qu'il n'existe pas de borne sur les temps d'acheminement des messages : le fait de ne pas avoir reçu de message d'un site à un moment donné ne signifie pas forcément que le site est défaillant : le message peut être en transit, mais prendre « beaucoup de temps » pour arriver. Il n'est donc pas possible (ou permis) de détecter la défaillance d'un site

- en recourant à des horloges de garde : quel que soit le délai fixé, le temps de transit peut être supérieur
- en s'appuyant sur des protocoles de communication en mode connecté comme TCP, qui eux mêmes recourent à des horloges de garde

### 3.3 Mise en œuvre

Le protocole sera conçu à partir de la même architecture client-serveur que PODP :

- le serveur gère et conserve une version « à jour » de l'objet dupliqué (copie maître) ;
- les clients gèrent une copie locale, possiblement en retard par rapport à la copie maître ;
- les écritures se font toujours et uniquement sur (via) le serveur, qui propage chaque mise à jour vers les clients ;

  Il est donc parfaitement acceptable qu'une écriture demandée (et obtenue) par un client ne soit pas répercutée instantanément sur la copie locale de ce client ;
- les lectures se font en collectant une ou plusieurs copies **locales** et doivent au final fournir la dernière valeur écrite (ou len cours d'écriture)sur le serveur, sur le serveur.

**La copie maître dont le serveur dispose ne devra pas être utilisée par les clients pour déterminer la valeur courante de l'objet partagé.**

La conservation de la copie maître est destinée à faciliter la phase de test (il sera ainsi possible de connaitre exactement quelle est la dernière valeur écrite), mais c'est son seul rôle : le protocole fonctionne et doit pouvoir fonctionner uniquement à partir de la gestion des valeurs des **copies locales**.

La section suivante précise le rôle exact du serveur.

### 3.4 Rôle et fiabilité du serveur

Le protocole robuste doit être conçu pour fonctionner uniquement à partir des différentes copies réparties sur l'ensemble des sites. Cependant, un serveur reste conservé, afin de

- **faciliter la mise au point** : comme indiqué dans le paragraphe précédent, la gestion d'une copie maître pourra permettre de connaître à tout moment la valeur et la version les plus récentes des objets partagés, à des fins de supervision et de mise au point.
- **fournir un service de désignation** global pour les objets partagés. Sur ce point, les structures de données peuvent être conservées. En revanche les méthodes create(−) et register(−) sont remplacées par une méthode publish(−). En effet, la création (paresseuse) d'un objet sur **un** client et sur le serveur n'a plus vraiment de sens : la création des copies et la diffusion des valeurs sont actives, c'est à dire qu'elles doivent être réalisées sur l'ensemble des copies, et non au fur et à mesure des demandes d'accès. Pour simplifier, enfin, la création est combinée avec l'affection d'un identifiant externe.
  L'opération lookup(−) est conservée, mais pourrait être éliminée simplement, car chaque client pourrait disposer du nom des objets partagés au moment de la création des objets.
- **gérer l'initialisation des applications**. En effet, Les opérations de lecture et d'écriture supposent (au moins au départ) que l'ensemble des sites est disponible. Le serveur fournit donc une méthode addClient(Client_itf client) qui devra être appelée par le client dès qu'il a obtenu la référence du serveur. Cette méthode fonctionne comme une barrière : elle doit bloquer le client, jusqu'à ce tous les clients soient prêts. Elle renvoie alors à chaque client la liste complète des sites participants.
- **simplifier la construction d'un ordre global** total sur les versions d'un objet. Les numéros de version sont uniques et croissants. Le passage par le serveur va permettre d'ordonner les écritures (et, au passage, de mettre à jour la copie maître). Le protocole sera le suivant : un site demandant une écriture appellera la méthode write(−) du serveur, et reprendra la main une fois que le serveur aura attribué une numéro de version, puis diffusé la valeur écrite vers l'ensemble des copies.

Ces services sont annexes au protocole lui même (à l'exception de la méthode write(−) qui pourrait être remplacée par un protocole intégralement réalisé par les clients, au prix d'une plus grande complexité).

En conséquence, il n'est pas demandé de prendre en compte la robustesse du serveur : **vous pouvez faire l'hypothèse que le serveur est fiable et reste toujours disponible**. Une version de l'interface correspondant à ces nouvelles orientations est disponible sur Moodle.

### 3.5 Indications

Les caractéristiques du protocole recherché amènent quelques premières remarques.
- Afin de déterminer dans un ensemble de copies quelle est celle qui correspond à la dernière écrite, il semble naturel d'attacher un numéro de version à chaque valeur prise par l'objet partagé.
- De même, étant donné que les accès devraient être non bloquants, et que le réseau est asynchrone, il est tout à fait possible (par exemple) que plusieurs requêtes successives émises par un même site pour obtenir la dernière version écrite d'un objet donné soient en transit vers différents sites. Il sera donc sans doute nécessaire de pouvoir désigner chacune des requêtes de manière unique, afin que le lecteur puisse déterminer à quelle requête correspond une réponse qu'il reçoit.
- Le protocole recherché ne s'appuie en aucune façon sur la gestion d'un service de verrous. C'est une différence (et une simplification) radicale avec le POPD : les objets côté serveur n'ont pas d'état de verrou à gérer ou à considérer pour déterminer si un accès est permis (ou doit être mis en attente).
- Ce point a aussi pour conséquence que les opérations de base fournies par le service ne seront plus lock_read()/lock_write()/unlock() mais simplement read() et write().

Pour construire la solution, il peut être utile de séparer les problèmes en procédant en 2 temps
- implémenter un service de registre régulier
- étendre ce service pour implanter un registre atomique pour des écritures séquentielles, (ce qui est assuré par le passage via le serveur) et des lectures concurrentes, ce qui consiste essentiellement à gérer le problème de l'inversion de valeurs.

### 3.6 Ébauche d'interfaces

Ces considérations amènent à ébaucher les interfaces suivantes :
- **ObjetPartagé**, (encapsulant un Object)
  - *read()* : retourne un Object
  - *write(valeur : Object)*
  - *getVersion()* : renvoie un entier (numéro de version unique, (compteur croissant, attribué par le serveur à l'écriture). Le numéro de version est un attribut de l'objet)
- **Client** (Remote)
  - *initialiser_un_objet(idObjet : entier, valeur : Object)* : crée la copie locale au Client d'un objet partagé
  - *enquête(idObjet : entier, cbl : Rappel_lec)*
  - *mise_à_jour(idObjet : entier, version : entier, valeur : Object, cbr : Rappel_ecr)*

  Les objets de rappel sont fournis par l'émetteur de la requête d'enquête ou de mise à jour, afin de lui permettre de vérifier la bonne prise en compte de ses requêtes par le Client.
  - **Rappel_lec** (Remote)
    - *réponse(version : entier, valeur : Object)* : permet au client de transmettre sa version courante de l'objet partagé, en réponse à l'enquête.
  - **Rappel_ecr** (Remote)
    - *réponse()*
- **Serveur** (Remote) : voir les fournitures sur Moodle

**Notes**

- Cette ébauche pourra être complétée ou remaniée selon le cours de vos réflexions. Dans ce cas, les idées de remaniement (important) devraient être soumises à validation aussi en amont que possible.
- Ces interfaces ne sont pas normalisées, car le travail demandé est avant tout un travail de réflexion. L'implémentation demandée peut être simple à programmer (de l'ordre de 200 LDC) et permettra essentiellement de formaliser et de démontrer (illustrer) votre travail.
- Outre Server_itf, les fournitures sur Moodle comporte une version de l'application Irc utilisant les opérations read/write proposées pour SharedObject.
- Afin de faciliter la mise au point, un protocole et un outil élémentaires de supervision devraient être fournis un peu avant la prochaine séance de suivi.

## 3.7 Supervision

(rédaction réservée)

## 3.8 Livrables

Vous devrez déposer une archive **au format tar,** contenant

- Un rapport, (4 à 8 pages) comportant
  - les structures de données introduites ou modifiées
  - le protocole et l'algorithmique de principe mis en jeu pour réaliser
    - les registres réguliers (2 semaines)
    - les registres atomiques pour des écritures séquentielles et des lectures concurrentes (1 semaine)
  - ces protocoles devront être clairement justifiés : faudra expliquer
    - pourquoi les registres ont les propriétés souhaitées, si vous pensez vos protocoles corrects
    - ou sinon, pourquoi vos protocoles ne fonctionnent pas, en exhibant les situations problématiques et en présentant les pistes explorées.
- Le code source de l'implémentation du protocole tolérant aux pannes
- Il n'est donc pas utile, ni pertinent de chercher à maintenir une cohérence continue : les lectures/écritures seront demandées explicitement. Par contre, il faudra montrer que l'application continue à fonctionner malgré la perte de sites. La perte d'un site pourra être provoquée a minima en tuant la JVM correspondant à ce site, ou mieux en instrumentant le protocole et en gérant/simulant les pannes au niveau d'un superviseur.
- Une démonstration : un bref compte rendu écrit et éventuellement illustré voire une petite vidéo au format mp4 (attention au poids).
- Bonus :
  - version physiquement répartie (sur plusieurs machines)
  - (plutôt difficile : ) étendre le protocole de registres atomiques pour gérer des écritures concurrentes.

## 3.9 Défi

**Concevoir un protocole garantissant la cohérence et permettant le plus grand nombre possible de défaillances de sites.**

Il existe en effet un nombre maximum de défaillances possibles : clairement, le protocole ne peut fonctionner si tous les sites sont en panne.

# 4 Modalités pratiques

**Le calendrier et les modalités sont détaillées sur la page Moodle du projet :**
**https ://moodle-n7.inp-toulouse.fr/course/view.php ?id=1974**

**Vous devez vous référer à cette page**, qui contient les informations précises et à jour sur le contenu et le déroulement du projet.
Les changements éventuels seront annoncés et affichés en début de page.

**Points essentiels (résumé)**

- Projet réalisé *en trinôme, ou binôme à défaut.*
- **Déroulement**
  - Des séances de suivi jalonnent le déroulement du projet.
  - Les livrables correspondant aux différents protocoles seront à déposer sur Moodle : fin février pour les 2 premiers protocoles, et début mai pour le protocole robuste.
  - Une restitution finale aura lieu mi-mai.
- **Evaluation** L'évaluation tiendra compte à parts sensiblement égales
  - des livrables portant sur les 2 premiers protocoles (extensions de PODP)
  - du rapport de conception relatif au protocole robuste
  - de l'implémentation et de la démonstration du protocole robuste.