Université Joseph Fourier
DESS Génie Informatique - année 2000-2001

Examen de Systèmes Répartis
18 décembre 2000

Durée : 3 heures ; documents autorisés

N.B. *L'examen comporte 3 problèmes indépendants. Prière de lire attentivement l'ensemble de l'énoncé avant de commencer à répondre, et de respecter les notations du texte. La longueur de l'énoncé n'est pas un signe de difficulté, mais est nécessaire pour bien spécifier les problèmes. La* ***clarté****, la* ***précision*** *et la* ***concision*** *des réponses, ainsi que leur* ***présentation matérielle****, seront des éléments importants d'appréciation.*

Problème 1. (7 points).

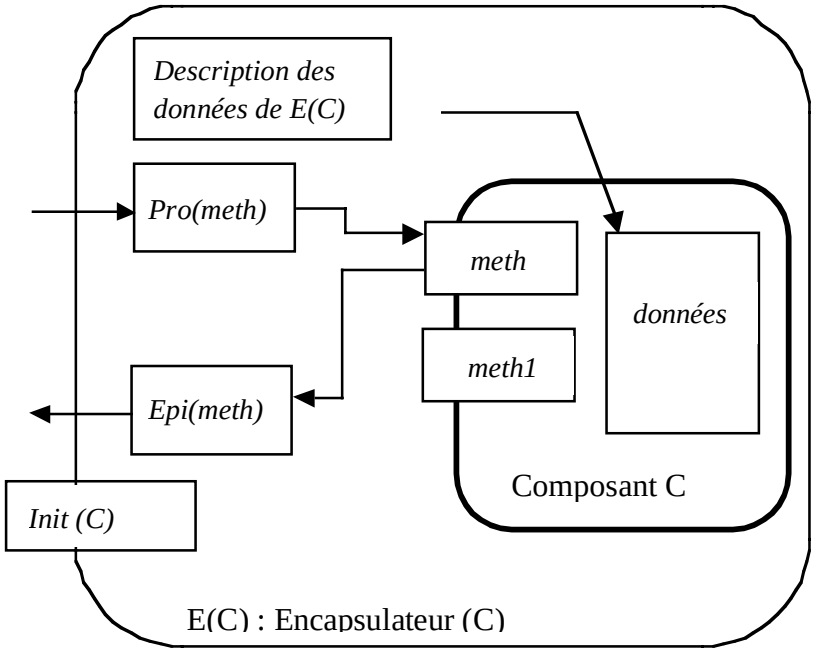
Le thème de ce problème est l'étude d'un environnement simple de programmation permettant de construire des applications à base de composants logiciels.

Dans le cadre de ce problème, un composant est un module comportant des données communes et un ensemble de procédures (ou « méthodes » agissant sur ces données. Le composant n'est utilisable que par l'appel de ces méthodes, dont l'ensemble constitue son interface.

L'environnement considéré fonctionne de la manière suivante. À tout composant est associé un module appelé encapsulateur, fourni par l'environnement. Pour chaque méthode *meth* du composant, cet encapsulateur comporte un prologue *Pro* (*meth*) et un épilogue *Epi* (*meth*). Lorsque la méthode *meth* est appelée (par exemple depuis le corps d'une méthode d'un autre composant), la séquence exécutée est *Pro* (*meth*) ; *meth* ; *Epi* (*meth*). Autrement dit, l'appel et le retour de chaque méthode sont interceptés par le module encapsulateur, qui exécute le prologue avant l'appel et l'épilogue après le retour, avant de revenir à la méthode appelante (figure ci-après). Le module encapsulateur possède lui-même des données propres, comprenant, entre autres, une description de la zone de données du composant encapsulé (nom, type, taille et adresse des différents champs).

L'interface de l'encapsulateur comporte entre autres une opération d'initialisation *Init(C)*, qui est appelée une fois et une seule avant la première utilisation de *C*.

Le but du module encapsulateur est d'ajouter au composant des propriétés spécifiques. Dans ce problème, nous examinons comment ces propriétés peuvent être obtenues, dans deux cas particuliers : persistance et sécurité.



Question 1. Cet environnement a quelques ressemblances avec un de ceux vus en cours. Lequel ? quelles sont les similitudes ? quelles sont les différences ?

Question 2. On souhaite ajouter au composant encapsulé la propriété de persistance. Rappeler la définition de cette propriété. Indiquer dans les grandes lignes comment la persistance peut être réalisée : que faut-il mettre dans *Init (C)* ? que faut-il mettre dans *Pro*(*meth*) et dans *Epi*(*meth*) ? quelles données est-il utile d'avoir dans la zone de données de l'encapsulateur ?

Question 3. On souhaite contrôler l'utilisation du composant C au moyen de listes d'accès. Rappeler la définition d'une liste d'accès. Montrer que l'on peut définir des listes d'accès différentes pour chaque méthode de *C*. Indiquer dans les grandes lignes comment cela peut être réalisé : que faut-il mettre dans *Init (C)* ? que faut-il mettre dans *Pro*(*meth*) et dans *Epi*(*meth*) ?

Question 4. Jusqu'ici, rien n'a été dit sur la façon dont sont produites les séquences *Init (C)* , *Pro(…)* et *Epi*(…). À votre avis, comment ces séquences pourraient-elles êtres produites automatiquement ? On ne vous demande pas de fournir un algorithme, mais d'indiquer à partir de quelles informations, et à quel moment, ces informations pourraient être construites.

Question 5. Quels avantages voyez-vous à l'utilisation d'un environnement tel que celui décrit ci-dessus ? quels inconvénients ou limitations ?

Problème 2. (7 points)

On propose de réaliser un service d'accès à des données dupliquées, pour assurer la tolérance aux fautes. Les données sont reproduites à *N* exemplaires, sur *N* serveurs différents. Un client peut faire deux opérations sur des données : des lectures ou des écritures, notées respectivement *lire*(*x*, *v*) et *écrire*(*x*, *v*) où *x* désigne la donnée spécifique lue ou écrite et *v* la valeur lue ou écrite. Dans ce problème, nous n'avons pas besoin de donner plus de détail sur le mode de désignation ou de représentation des données (on suppose qu'un serveur sait retrouver une donnée connaissant sa désignation *x*).

L'hypothèse de défaillance des serveurs est l'arrêt sur défaillance (*fail stop*).

On dispose (chez le client) d'une primitive d'accès multiple distant *exec*(*liste*(*S*), *x*, *v*, *liste*(*res*)) qui fonctionne comme suit :

- le paramètre *liste*(*S*) contient la liste des serveurs sur lesquels on veut exécuter la lecture ou l'écriture (cette liste peut comprendre un ou plusieurs serveurs),
- le paramètre *x* est la désignation de la donnée lue ou écrite.
- Le paramètre *v* est la valeur écrite dans le cas d'une écriture, et une valeur spéciale *nil* dans le cas d'une lecture.
- le paramètre *liste*(*res*) est une liste des résultats, de même taille que *liste*(*S*).

Un résultat peut avoir les valeurs suivantes : pour une lecture, la valeur lue ; pour une écriture, la valeur écrite ; dans les deux cas, si le serveur est défaillant, une valeur spéciale *erreur*. La valeur *erreur* est rendue si le serveur ne répond pas au bout d'un délai de garde.

Question 1. On souhaite préserver l'invariant suivant, supposé réalisé à l'initialisation du système : tous les serveurs en état de marche (non défaillants) contiennent des versions identiques des données, au sens où le résultat d'une même opération de lecture est identique quel que soit le serveur (valide) sur lequel a été lue la donnée. Expliquer en détail (donner des algorithmes) comment se déroulent une opération de lecture *lire*(*x*, *v*) et une opération d'écriture *écrire*(*x*, *v*) à partir d'un client (les opérations doivent préserver l'invariant ci-dessus). Quelles propriétés doit avoir l'opération *exec*, si on veut que plusieurs clients puissent exécuter concurremment des opérations sur les données ? (noter qu'*exec* est réalisée par un protocole réparti, faisant intervenir plusieurs sites ; on ne demande pas ici d'indiquer comment ce protocole est réalisé, mais de donner les propriétés qu'il doit avoir).

Question 2. On veut maintenant modifier le protocole de lecture et d'écriture de la manière suivante : une lecture n'est possible que si r serveurs (au moins) sont disponibles, et une écriture n'est possible que si w serveurs au moins sont disponibles. Chaque serveur maintient en outre un numéro de version, qui est incrémenté de 1 à chaque écriture.

a) Lors d'une lecture, les valeurs lues sur différents serveurs peuvent-ils avoir des numéros de version différents ? illustrer si nécessaire par un exemple.

b) L'invariant indiqué à la question 1 est remplacé par le suivant : une opération *lire* délivre toujours une valeur à jour de la dernière opération *écrire* (c'est-à-dire issue du serveur qui a le numéro de version le plus élevé existant dans le système). Montrer que la condition $r + w > N$ garantit que cet invariant est bien respecté.

c) Indiquer comment doivent être modifiés les algorithmes des opérations de lecture et d'écriture donnés en réponse à la question 1.

d) Quel est d'après vous l'intérêt de ce nouveau protocole ?

e) À quelles valeurs situations correspondent respectivement les deux couples de valeurs suivants : $r = 1$, $w = N$, et $r = N$, $w = 1$?

Question 3. Que doit faire un serveur qui est remis en service après une panne ? la réponse est-elle la même dans les hypothèses de la question 1 et dans celles de la question 2 ?

Question 4. On modifie maintenant de la manière suivante le protocole de la question 2 : le nombre N ne représente plus le nombre de serveurs, mais un nombre total de "jetons" qui sont répartis entre les serveurs. La condition $r + w > N$ reste imposée.

Un exemple est la configuration suivante : 3 serveurs S1, S2, S3 ; $N = 4$: 2 jetons pour S1, 1 pour S2, 1 pour S3 ; $nr = 2$, $nw = 3$. Quel est d'après vous l'intérêt de cette organisation ? Quel est son inconvénient (indication : examiner les scénarions de panne) ?

Problème 3. (6 points)

Question 1. Les propriétés de confidentialité et d'intégrité d'un message sont-elles indépendantes ? en d'autres termes, peut-on assurer chacune d'elles sans assurer l'autre ? donner des exemples.

Question 2. Dans le protocole d'authentification de Needham – Schroeder utilisant les clés secrètes, le dernier échange est :

A -> B : $\{I_B - 1\}_{KAB}$

Peut-on le remplacer par :

A -> B : $\{I_B\}_{KAB}$

Indiquer avec précision quel est le risque.

Question 3. Dans le protocole d'authentification de Needham – Schroeder utilisant les clés publiques, le dernier échange est :

A -> B : $\{I_B\}_{KPB}$

Pourquoi, dans ce cas, et contrairement au cas du protocole à clé secrètes, n'a-t-on pas besoin de mettre $I_B - 1$?

Question 4. Quels sont les avantages et les inconvénients du système d'authentification par mot de passe non réutilisable ?

Question 5. On considère un système de cryptographie à clés publiques qui n'utilise pas de certification. Chaque correspondant communique sa clé publique sur demande.

Soit deux correspondants A et B, où A contacte B et s'authentifie auprès de lui :

A -> B : je suis A

B -> A : quelle est ta clé publique ?

A -> B : ma clé publique est KPA

B -> A : I_B

A -> B : $\{I_B\}_{KSA}$

Montrer qu'un intrus E capable d'intercepter et d'injecter des messages sur le réseau pourrait se faire authentifier par B en tant que A.