Projet de Programmation Système – CS207-a – 2018

Description général du projet principal :

Système de hashage distribué

Plan de ce document

1. Contexte du projet
2. Buts du projet
3. Description générale des structures de données impliquées
4. Glossaire

Contexte

Le but premier de ce projet est de vous faire développer un large programme en C sur une thématique « système ». Le cadre choisi cette année est celui des réseaux « pair à pair » et plus particulièrement celui de la représentation distribuée sur un tel réseau d'un système d'associations clés-valeurs (système de hashage, appelé aussi « *key-value store* » ou « *distributed hash table* (DHT) »).
Le but de ce projet **n'**est **pas** du tout de se substituer au cours de *Computer Networks* que vous avez eu le semestre passé (ni même de le compléter), mais simplement de vous faire développer, en suivant une spécification pas à pas, une version simplifiée de concepts proches de ceux qui vous ont été présentés.
Tous les concepts de base requis pour ce projet sont introduits ici de façon simple en ne supposant qu'une connaissance « utilisateur » standard d'un système informatique. En cas de doutes sur la terminologie, le glossaire en fin de document peut se révéler pratique.

Vous allez développer une version simplifiée de DHT inspirée de Dynamo d'Amazon (vous n'avez pas à lire ce papier, c'est juste pour référence, si ça vous intéresse). Un tel système peut se voir comme une version des Map de Java qui serait rendue robuste par sa distribution redondante sur plusieurs ordinateurs (au lieu de rester localement dans la mémoire d'un seul). Un tel système offre, bien sûr, une interface d'utilisation simple, très similaire à une table associative (ou une Map en Java), mais derrière distribue le stockage effectif des données sur plusieurs serveurs. Cela augmente d'une part disponibilité des données par le fait que plusieurs machines ont des copies de celles-ci (au cas où l'une ou l'autre machine serait indisponible), et d'autre part la capacité totale de stockage (en additionnant les capacités de toutes les machines impliquées).

L'interface (API) d'utilisation minimale peut se résumer à :

```
put(key, value, N, W);
value = get(key, N, R);
```

L'opération `put` ajoute (ou met à jour) des données dans le système en associant la valeur `value` à la clé `key`. Une telle clé peut ensuite être utilisée pour retrouver la valeur associée au moyen d'une opération `get`. Ces opérations sont exécutées par une machine « client », laquelle envoie les requêtes correspondantes aux machines « serveurs » et coordonne leur réponses (détails plus bas). Des paramètres supplémentaires (N, W, et R, expliqués en détails plus bas) permettent de configurer le niveau de redondance désiré et de l'équilibrer par rapport au niveau de performances (robustesse, temps d'accès) attendu : N indique combien de serveurs doivent, en principe, stocker la valeur (c'est donc le nombre *maximum* de serveurs à contacter), W (comme « **w**rite ») et R (comme « **r**ead ») déterminent le nombre *minimum* de serveurs qui doivent répondre positivement à une opération respectivement `put` et `get`. Un choix adéquat de ces valeurs permet au système globale de résister à certaines défaillances des serveurs ou du réseau tout en permettant des performances raisonnables (en terme de temps) pour les opérations `put` et `get`.

Au niveau global, en réponse à une commande de `put` ou de `get`, la machine « client » va contacter entre W ou R et N machines « serveurs » et, en fonction de leur réponse (ce sera expliqué plus tard comment en détails), répond elle-même par un échec ou un succès (réponses cohérentes ; on dira que le « *quorum* » W ou R est atteint).

Donnons quelques exemples (où l'on note de façon compacte « `a:b` » l'association entre la valeur `b` et la clé `a`) :

Dans l'exemple illustré dans l'image ci-dessus, le client exécute une opération `put` demandée pour N=3 serveurs, avec un minimum de W=2 (quorum simple) en accord. Dans le cas illustré, les 3 serveurs reçoivent la requête, stockent l'association « `a:b` » et répondent positivement ; donc le client répond par un succès.

Supposons maintenant (image ci-dessus) que le client exécute une nouvelle opération `put` pour mettre à jour la valeur précédente (remplacer `b` par `c`). Mais cette fois le message envoyé au « Server 3 » est perdu. Les deux premiers serveurs font la mise à jour de leur propre association et stockent « `a:c` » puis répondent positivement ; mais le « Server 3 » n'étant pas au courant ne fait rien du tout (garde l'association « `a:b` » et ne répond rien du tout). Au niveau du client cependant, l'opération est un succès puisque le quorum de W=2 serveurs répondant positivement a été atteint.

Continuons l'exemple en imaginant maintenant que le client exécute une opération `get` pour rechercher la valeur associées à la clé `a` ; et supposons que les N=3 serveurs reçoivent la requête et y répondent. Cependant en l'état, ils ne répondent pas la même chose : deux répondent `b` et un répond `c`. Au niveau du client pourtant c'est un succès et la valeur `b` est retournée puisque celle-ci a obtenue

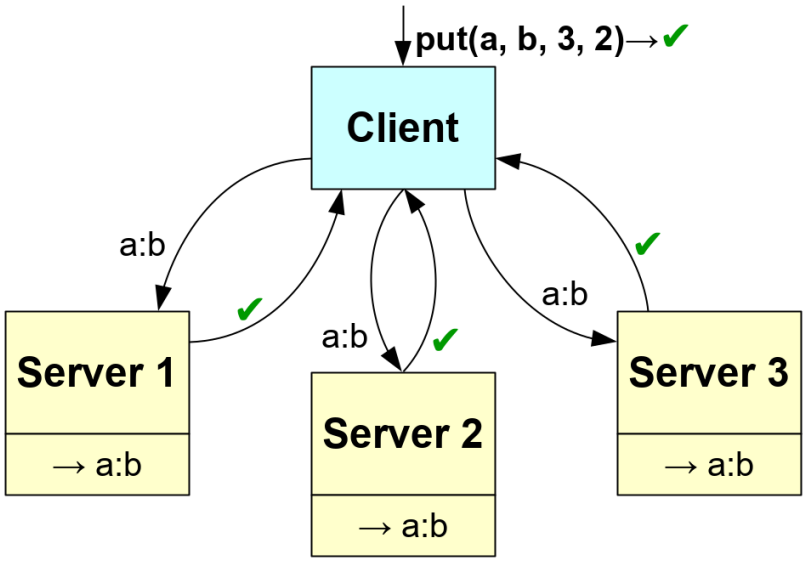

Figure 1: Exemple d'opération `put` réussie.

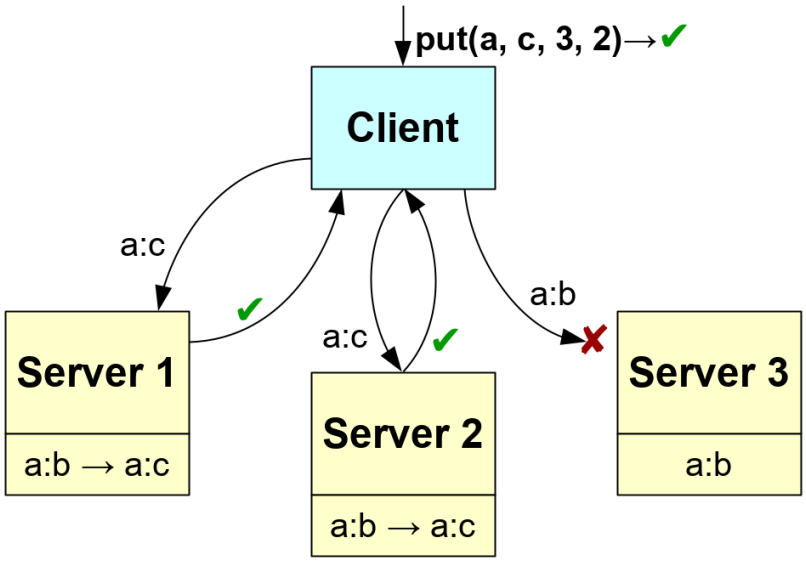

Figure 2: Exemple d'opération `put` avec échec réseau.

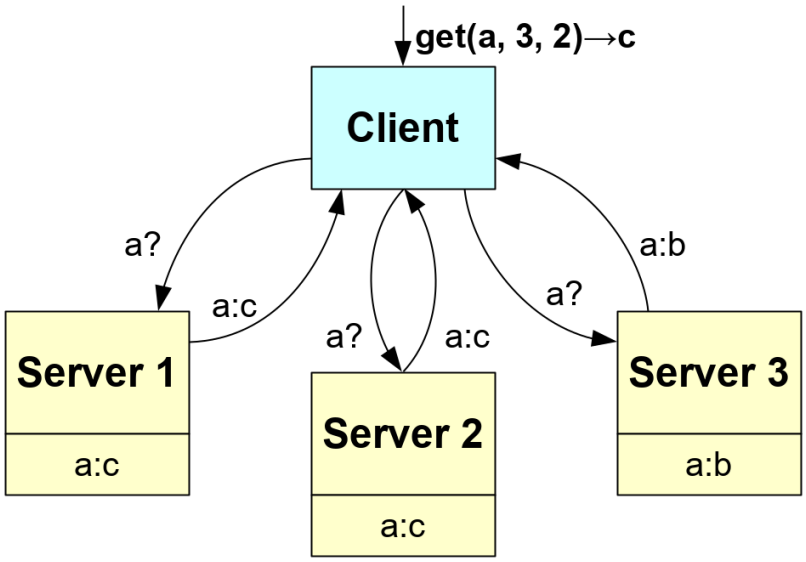

Figure 3: Exemple d'opération `get` réussie.

le quorum de R=2 (le cas où plusieurs valeurs différentes atteignent le quorum sera traité dans la série correspondante).

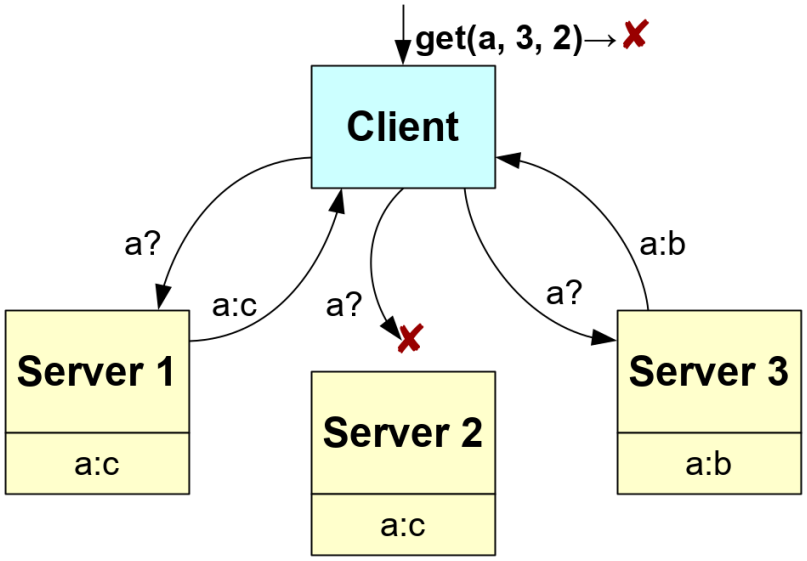

Figure 4: Exemple d'opération `get` échouée.

Pour finir sur cet exemple, supposons que le client exécute à nouveau une opération `get` pour la clé `a`, mais que le « Server 2 » soit injoignable. Le client ne reçoit donc que deux valeurs : `b` et `c`, dont aucune n'atteint le quorum R=2. Le client répond donc par un échec. Les paramètres N=3, W=R=2 ont été choisis pour résister à *une* erreur ; or là nous en avons maintenant deux (une en écriture sur « Server 3 » et une en lecture sur « Server 2 »), d'où l'échec.

L'exemple précédent illustre un scénario simple dans lequel tous les serveurs stockent toutes les données. Cela n'est pas réaliste en pratique pour de grands ensembles de données stockés sur un grand nombre de serveurs. En réalité, l'espace des clés possibles est partitionné en différents segments affectés chacun à un serveur (un même serveur pouvant être affecté à plusieurs segments, voir la notion de « *nœud* » ci-dessous). L'idée est d'ordonner de façon croissante l'espace des clés possibles (on appelle cela « *l'anneau des clé* » ou « *ring* » ; voir illustration ci-dessous) pour y répartir les serveurs, responsables alors de la tranche des clés allant jusqu'au nœud suivant (dans l'ordre imposé sur les clés). Dans ce projet, nous utiliserons l'algorithme SHA-1 pour représenter/projeter les clés (et les serveurs) dans l'anneau des clés.

Je crois qu'un exemple s'impose : supposons que l'on fasse correspondre les clés

à un nombre entier entre 0 et 255 (on travaille modulo 256) et supposons que l'on y répartisse 6 serveurs (A à E) comme ceci :

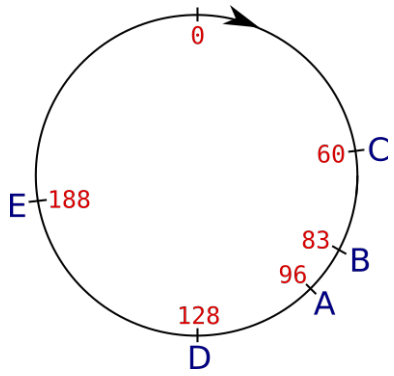

Figure 5: Exemple de 6 serveurs dans l'anneau des clés.

En réalité, les valeurs SHA-1 que nous allons utiliser pour projeter les clés varient entre 0 and 1461501637330902918203684832716283019655932542975, ce qui n'est pas facile a représenter graphiquement...

Pour déterminer quel(s) serveur(s) contacter pour une clé donnée, le client calcule le code SHA-1 de celle-ci (représenté en vert) puis recherche par ordre croissant le premier serveur correspondant :

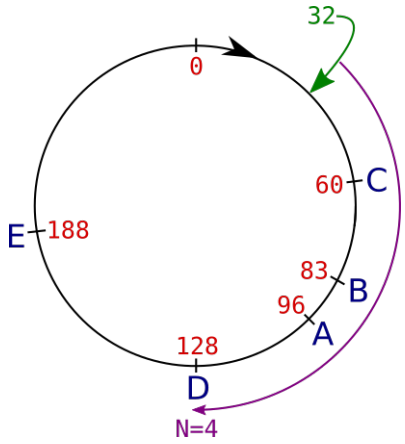

Figure 6: Exemple de projection d'une clé dans l'anneau des clés.

Il continue ensuite, toujours par ordre croissant jusqu'à en avoir trouvé N.

Jusqu'ici nous n'avons pas trop souligné la différence entre nœud et serveur, les 6 nœuds de l'exemple précédent correspondant chacun à une machine, un serveur, différent(e). Mais si l'on se contente de répartir des serveurs *différents*

sur l'anneau des clés, cela peut nuire à l'efficacité du système en pratique ; typiquement lorsqu'un serveur tombe, il laisse alors un gros « trou » dans l'anneau des clés. Pour cette raison, il est préférable qu'un même serveur soit réparti à différents endroits de l'anneau des clés. D'où la notion de « *nœud* » : un même serveur correspond à plusieurs nœuds dans l'anneau. Un même serveur continue à gérer l'ensemble de ses clés (et valeurs) mais peut, grâce a la notion de « *nœud* », apparaître à plusieurs endroits dans l'anneau des clés. Un nœud de l'anneau des clés est donc une pure abstraction représentant simplement **une** projection d'un serveur dans l'anneau des clés.

Pour illustrer cela, considérons que dans l'exemple précédent les lettres (A-E) correspondent aux serveurs et ajoutons un indice pour représenter les nœuds. Décidons par exemple d'avoir 9 nœuds cette fois :

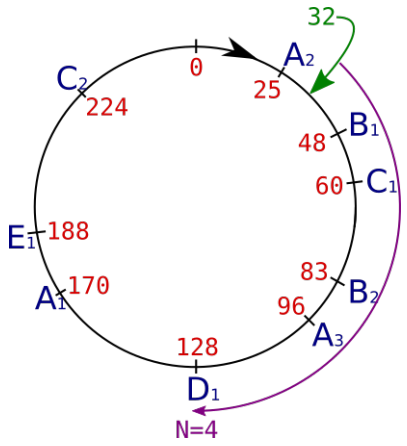

Figure 7: Exemple position ring with 6 serveurs and a key.

Note : remarquez que la position des nœuds/serveurs dans l'anneau peut être tout à fait quelconque.
[**fin de note**]

Du point de vue de l'explication du fonctionnement du client, cela ne change presque rien à ce qui a été décrit précédemment ; simplement, il se peut maintenant au vue de la répartition que pour une même requête, le client contacte effectivement plusieurs fois le même serveur : il ne faut alors par compter ce serveur plusieurs fois : lors de la recherche de N serveurs, on ne compte donc pas les nœuds provenant d'un serveur d'un nœud déjà visité (autrement dit : on compte bien le nombre de **serveurs** différents répondant à la requête et non pas le nombre de nœuds). C'est pour cette raison que dans l'illustration ci-dessus, la recherche N=4 couvre bien 5 nœuds : le nœud B2 n'est pas compté car correspondant à un serveur (B) déjà contacté (via B1).

Pour simplifier dans le cadre de ce projet, on supposera que le client connaît par avance (ce sera un fichier de configuration) la liste de nœuds et des serveurs,

plutôt que de les découvrir par un autre protocole réseau.

Un exemple plus complet est détaillé plus bas.

Buts et organisation du projet

Durant les 10 semaines de ce projet, vous allez devoir implémenter, graduellement morceau par morceau les trois composants clés d'un système de hashage distribué :

- tout le travail local sur un serveur (stockage clés-valeurs, réponses aux requêtes) ;
- l'opération `put` coté client ;
- l'opération `get` coté client.

Vous allez aussi devoir développer des outils complémentaires utiles pour observateur et analyser le fonctionnement du système de hashage distribué. Tous ces outils seront développés sous formes d'exécutables indépendants (ensemble d'outils sur la ligne de commande usuelle).

Enfin, en dernière semaine vous allez également faire une étude expérimentale de l'impact des différents paramètres sur les performances du système.

Afin de faciliter au mieux l'organisation de votre travail (dans le groupe et dans le temps), voici une représentation synthétique des différents modules logiciels impliqués dans ce projet :

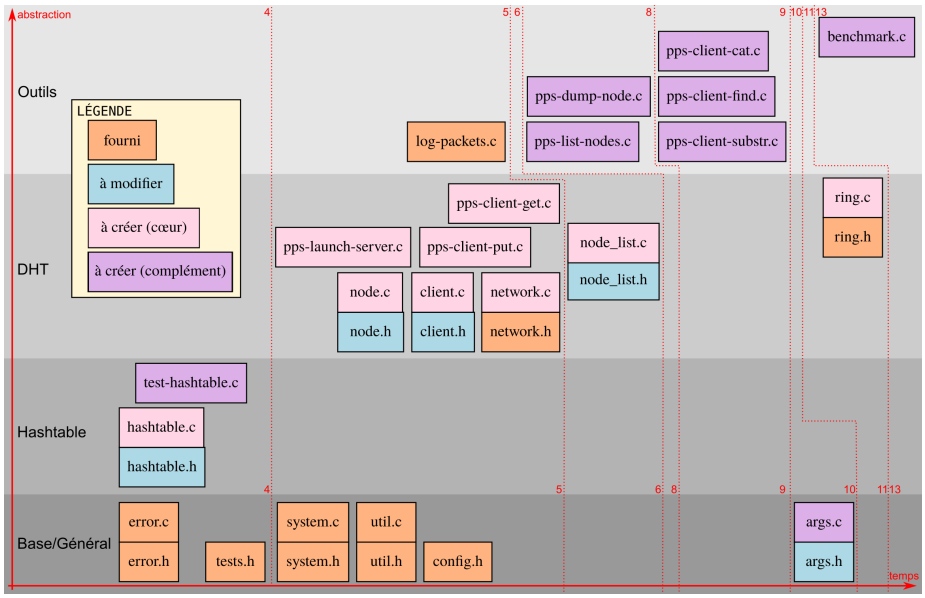

Figure 8: Modules du code du projet

Toujours pour vous organiser au mieux, veuillez également consulter la page de barème du cours (et la lire en entier !!).

Structures de données (description générale)

Nous décrivons ici de façon générale les principales structures de données que ce projet nécessitera. Leur détail d'implémentation sera précisé plus tard lorsque nécessaire dans le sujet précis correspondant.

- `bucket_t` : élément interne (« *alvéole* ») d'une table de hashage locale (aux serveurs) ;
- `client_t` : représentation d'un client ;
- `Htable_t` : pour stocker localement (sur chaque serveur) les associations clé-valeur dont il est responsable ;
- `pps_key_t` : pour représenter les clés ; au début du projet sera simplement un seul caractère, à la fin sera une chaîne de caractères ;
- `ring_t` : anneau des clés : table de correspondance entre position dans l'anneau et serveurs représentée sous la forme d'une liste de nœuds (`node_t`) ;
- `node_t` : représentation d'un serveur (en fait : d'un nœud, mais la distinction ne sera introduite qu'en semaine 11) ;
- `pps_value_t` : pour représenter les valeurs ; au début du projet sera simplement un entier, à la fin sera une chaîne de caractères.

Exemple

Voici un exemple complet de la version finale visée par ce projet. Il y aura tout au cours du semestre diverses versions simplifiées de ce modèle : on passera progressivement d'une version très simple à l'exemple donné ici.

Supposons que l'on veuille stocker des associations chaîne-chaîne dans un réseau de S=5 serveurs, pour faire, par exemple, des opérations telles que `get("ma_clé")` ou `put("ma_clé", "ma_valeur")`.

Supposons de plus que (voir figure ci-dessous) :

- la fonction `Hring()` de hashage des clés au niveau global (anneau des position) retourne des valeurs entre 0 et 255 ;
- la fonction `Hlocal()` de hashage des clés au niveau local (chaque serveur) retourne des valeurs entre 0 et 42 ;
- M = 9 (9 nœuds en tout dans l'anneau des positions) ;
- N = 4 (on cherche à faire les opérations de lecture/écriture sur 4 serveurs) ;
- R = 2 (2 lectures valides sont requises pour récupérer une valeur depuis le réseau) ;
- W = 3 (3 écritures valides sont requises pour insérer une valeur dans le réseau) ;

- la répartition des serveurs et des nœuds dans l'anneau des positions soit la suivante :
- serveur A : 25, 96, 170 ;
- serveur B : 48, 83 ;
- serveur C : 60, 224 ;
- serveur D : 128 ;
- serveur E : 188 .

Voici une figure illustrant cette architecture :

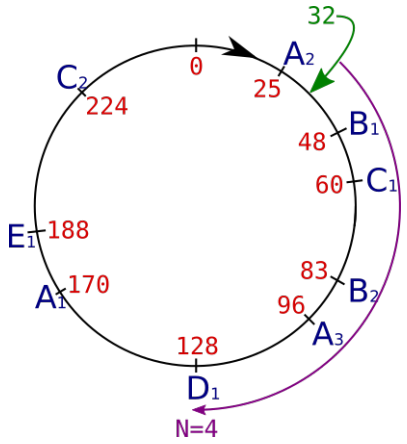

Figure 9: Un exemple de DHT avec S=5, N=4, W=3 et R=2.

Supposons encore que :

- le serveur B soit tombé (il serra détecté comme tombé via un timeout) ;
- `Hring("ma_clé")` = 32 ;
- et `Hlocal("ma_clé")` = 10.

Que se passe-t-il lors d'un `put("ma_clé", "ma_valeur")` ?

1. On calcule `Hring("ma_clé")`, qui donne 32 ;
2. On cherche à écrire sur N serveurs différents dans l'anneau des positions à partir de 32 : cela donne les serveurs B, C, A et D ;
   le nœud 83 (serveur B) est ignoré car correspond à un serveur déjà considéré (B) ;
3. Comme B est tombé, seules les écritures sur C, A et D fonctionnent ;
4. puisque W=3 et que l'on a 3 serveurs qui ont répondu positivement à l'écriture, celle-ci est considérée comme réussie.

De plus, sur chacun des serveurs A, C et D, l'écriture de la valeur `"ma_valeur"` associées à la clé `"ma_clé"` est faite localement. Avec le choix que nous avons fait pour l'implémentation locale, cela se fait à l'aide d'une table de hashage (locale) dont la fonction de hashage est `Hlocal()`.

On calcule donc `Hlocal("ma_clé")`, qui vaut 10, et stocke à la position 10 du tableau local la paire `("ma_clé", "ma_valeur")`.

Que se passe-t-il ensuite lors d'un `get("ma_clé")` (alors que B est toujours tombé) ?

1. On calcule `Hring("ma_clé")`, qui donne 32 ;
2. On cherche à lire sur N serveurs différents dans l'anneau des positions à partir de 32 : cela donne les serveurs B, C, A et D ;
3. Comme B est tombé, seuls C, A et D répondent, chacun `"ma_valeur"` ;
4. Puisque l'on a plus de R=2 réponses concordantes, on retourne `"ma_valeur"`.

Que se passe-t-il avec `get("ma_clé")` si B est opérationnel mais C est tombé ?

Les étapes 1 et 2 se déroulent de même, mais à l'étape 3, B répond « n'importe quoi » (soit inconnu, soit une ancienne valeurs, supposons différente) et A et D répondent toujours chacun `"ma_valeur"`. On a donc toujours au moins R=2 réponses cohérentes, donc on retourne `"ma_valeur"`.

Et si A ou D tombent aussi ?

On aura alors une seule fois la valeur `"ma_valeur"` et une fois « n'importe quoi ». Aucune valeur n'a été lue au moins R fois, la lecture globale est donc un échec.

Glossaire

- **anneau des positions** : abstraction servant à faire la correspondance entre des clés et des serveurs (représentés par des nœuds positionnés sur cet anneau) ;
- **clé** : identifiant unique pour une valeur à stocker ;
  au début du projet sera simplement un caractère, à la fin sera une chaîne de caractères.
  Exemple : `"ma_clé"` dans `get("ma_clé")` ou dans `put("ma_clé", "ma_valeur")`.
- **client** : les machines qui gèrent la table de hashage distribuée ;
- **éléments de la table de hashage** : parfois appelés « *alvéoles* », « *buckets*» en anglais ;
- **fonction de hashage** : fonction qui associe une clé à un index pour accéder à une valeur stockée dans la table de hashage ;
  Exemple : `H()` dans `index = H("ma_clé")` ;
  il y a en fait **deux** fonctions de hashage différentes dans ce projet :
- celle de l'anneau des positions, qui transforme une clé en une position de l'anneau (nombre entier, type `size_t`) ;

- celle des tables de hashage locales à chaque serveur pour y stocker les valeurs ;
- **index** : position dans une table de hashage locale à un serveur ; résultat du hashage de la clé par la fonction de hashage locale ;
  Exemple : `index = Hlocal("ma_clé")`.
- **M** : nombre total de nœuds dans l'anneau des clés ;
  On doit avoir $M \geq N$.
- **N** : nombre maximum de serveurs qui stockent une clé particulière ; c'est aussi le nombre maximum de lectures/écritures effectuées pour une valeur donnée (voir R et W).
- **nœud** : abstraction (= entité conceptuelle) de l'anneau des clés couvrant un sous-ensemble de positions possibles ; c'est une abstraction car un serveur réel peut très bien représenter plusieurs nœuds (cette distinction nœud/serveur permet de répartir un même serveur à plusieurs endroits différents non consécutifs de l'anneau des positions) ;
  Exemple : le serveur 23 couvre(= est représenté par) les nœuds 3, 11 et 17.
- **position** : position dans l'anneau des positions ; résultat du hashage de la clé par la fonction de hashage spécifique à l'anneau ;
  Exemple : `position = Hring("ma_clé")`.
- **R** : (comme « *read* ») nombre de serveurs fonctionnels requi pour récupérer une valeur depuis le réseau ; une opération de lecture tente de lire sur N serveurs et réussit si (au moins) R des ces lectures ont réussi et coïncident .
- **S** : (comme « *serveur* ») nombre de serveurs dans le réseau ;
  On doit avoir $M \geq S \geq N$.
- **serveur** : machine dans le réseau servant à stocker physiquement les données ; est représenté par un ou plusieurs nœud(s) dans l'anneau des positions ;
- **valeur** : une valeur à stocker dans la table de hashage distribuée ;
  au début du projet sera simplement un entier, à la fin sera une chaîne de caractères.
  Exemple : `"ma_valeur"` dans `put("ma_clé", "ma_valeur")`.
- **W** : (comme « *write* ») nombre de serveurs fonctionnels requi pour stocker une valeur dans le réseau ; c'est donc le nombre minimum de réplication de chaque valeur dans le réseau ; une opération d'écriture tente d'écrire sur N serveurs et réussit si (au moins) W des ces écritures ont réussi.