

Entrepôt Réparti en mémoire

PSAR : Sujet 4



Clients : SENS Pierre – KORDON Fabrice
Auteurs : MARECAR Farzana – HE Chenhui

Table des matières

- Introduction
- Principaux points à coder
- Choix des protocoles réseaux
- Fils de connexions
- Structures et normes définies, & leur utilités
- Démonstration par le codage du fonctionnement
- Démonstration schématique du raisonnement
- Finitions & Améliorations à apporter

Introduction

Principe :

Système client/serveur

Plusieurs serveurs

Mémoire répliquée entre Serveurs

Synchronisation entre serveurs

Cohérence entre serveurs

Répartition des charges entre serveurs

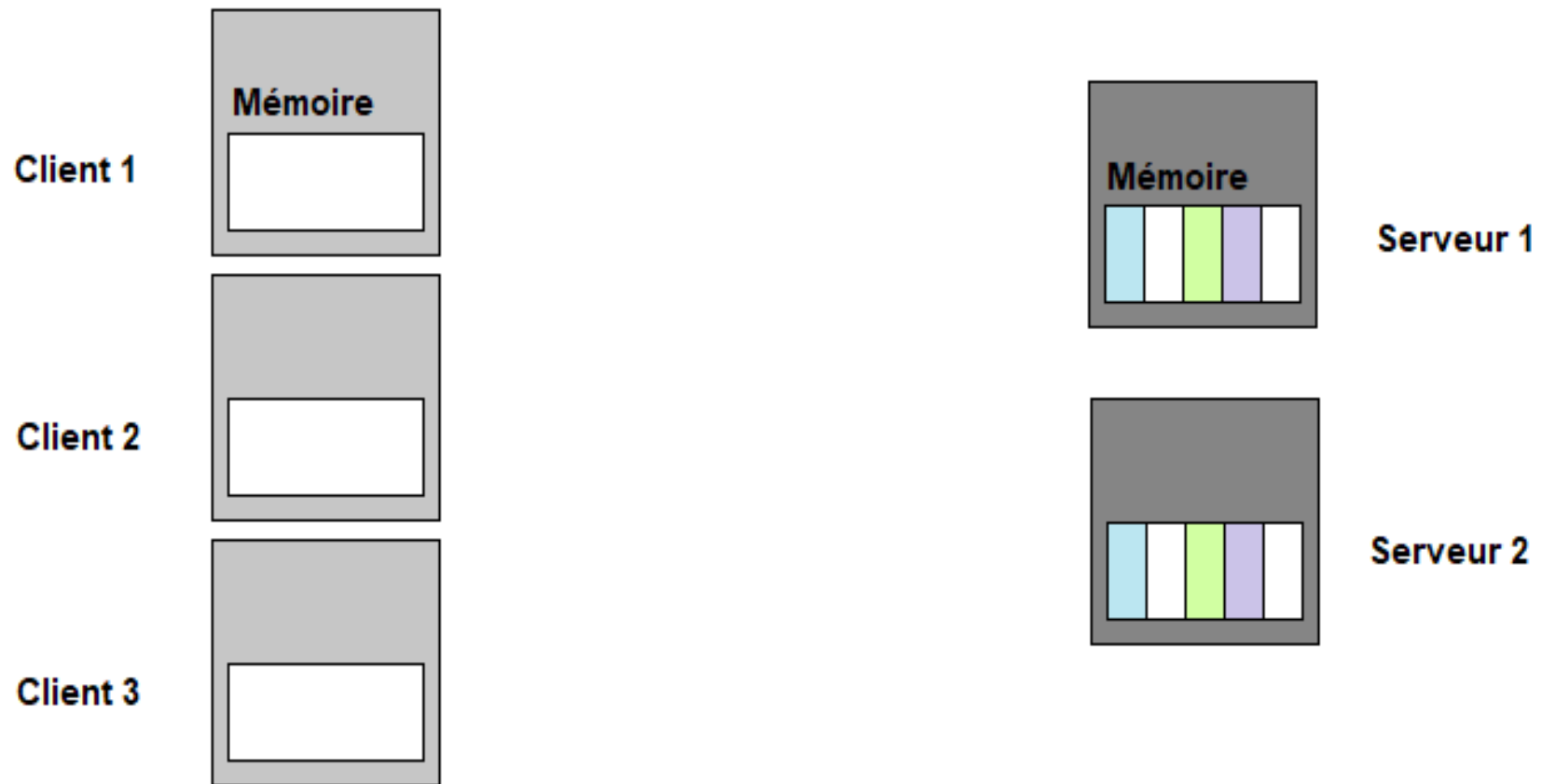
Fournir des services

Création de données

Lecture de données

Modification de données – accès en Lecture/Écriture

Schéma simplifié du système étudié



5 parties à coder

Côté Client :

Méthodes liées à l'appel des services

Main initiant la connexion à un serveur pour l'accès aux données

Côté Serveur :

Méthodes liées aux traitements des services clients

Méthodes liées à la synchronisation entre serveurs

Main

- Initiant la connexion pour les connexions clients
- Initiant la connexion pour la synchronisation entre serveurs
- Appelant la méthode adéquat pour un message client
- Appelant la méthode adéquat pour un message serveur

De plus :

Fichier « log » pour connaître & communiquer avec serveurs

Three black server racks are positioned on a dark, reflective surface. Each rack has several horizontal green light bars at the bottom, suggesting active components or cooling fans. The racks are slightly staggered, with the middle one being the most prominent. The background is a dark, gradient grey.

Partie Réseau

Choix des protocoles réseaux

TCP :

Pour les services clients

→ Services de création, et manipulations de données en temps réel

Pour les messages de synchronisation entre serveurs

→ Besoin de synchronisation en temps réel pour la cohérence des données et informations

UDP :

Pour les *heartbeats* :

Connaître l'état du système (entre serveurs)

Fils de connexion – chez les Serveurs

2 ports :

Un port de service

Un port pour la communication entre serveurs

Socket vers clients :

Socket Internet

Adresse IP & numéro de port spécifique

Nombre de connexions acceptées bornées

Redirection vers handler qui appelle la méthode de traitement du service adéquat

Socket vers serveurs :

Socket Internet, Adresse IP & numéro de port spécifique

Appel des méthodes de traitement plus simplifiées


```

//Connect to remote server
if (connect(sock , (struct sockaddr *)&server , sizeof(server)) < 0)
{
    perror("Connection failed - Error");
    return 1;
}

puts("Connected\n");

//keep communicating with server
while(1)
{
    printf("Entrez une commande (creer, lire, modifier ou fin pour terminer les traitements) en précisant le nom de la donnée :\n");
    scanf("%s" , message);

    if(strcmp(message, "fin")==0){
        goto fin;
    }

    strtok(message, " ");


    if(strcmp(message, "creer")==0){
        char* nom = strtok(NULL, " ");
        printf("Veuillez entrer la valeur d'initialisation :\n");
        char val[4096];
        scanf("%s", val);
        creer(sock, server, nom, val);
        continue;
    }

    else if(strcmp(message, "lire")==0){
        char* nom = strtok(NULL, " ");
        lire(sock, server, nom);
        continue;
    }

    else if(strcmp(message, "modifier")==0){
        char* nom = strtok(NULL, " ");
        modifier(sock, server, nom);
        continue;
    }

    else{
        printf("Commande non reconnue\n");
        break;
    }
}

```



```
#define PORT_SERVERS 4000
#define PORT_SERVICE 2000
#define N 10

char* myIP = "127.0.0.1";

char* listIP[N][N];

void init_list(){
    srand(time(NULL));
    listIP[0][0] = "127.0.0.1";
    listIP[0][1] = "127.0.0.2";
    listIP[0][2] = "127.0.0.3";
    listIP[0][3] = "127.0.0.4";
    listIP[0][4] = "127.0.0.5";
    listIP[0][5] = "127.0.0.6";
    listIP[0][6] = "127.0.0.7";
    listIP[0][7] = "127.0.0.8";
    listIP[0][8] = "127.0.0.9";
    listIP[0][9] = "127.0.0.10";
}

//~ char* SIP_alea(){
    //~ int r = (rand() % (N + 1 - 0));
    //~ return listIP[r];
//~ }
```

Handler de messages clients

```
void *connection_handler(void *socket_desc)
{
    ///Get the socket descriptor
    int sock = *(int*)socket_desc;
    int read_size;
    char *message, client_message[4096];

    ///Receive a message from client
    while( (read_size = recv(sock, client_message, 4096, 0)) > 0 ){
        strtok(client_message, " ");

        /** Format de messages recus : create <nom> <val>
         * Format de messages envoyes : create <nom> OK <details>
         *                               ou      : create <nom> KO <details>
         */
        if(strcmp(client_message, "create")==0){
            char* nom = strtok(NULL, " ");
            char *val = strtok(NULL, " ");
            char* res = p_create(nom, val, sock);
            write(sock, res, strlen(res));
            continue;
        }

        /** 1) Récupération de valeur : (p_access_read)
         * Format de messages envoyes : access_read <nom>
         * Format de messages recus : access_read <nom> OK <val>
         *                               ou      : access_read <nom> KO <details>
         * 2) Fin Lecture :
         * Format de messages envoyes : release_read <nom>
         * Format de messages recus : release_read <nom> OK <details>
         *                               ou      : release_read <nom> KO <details>
         */
        else if(strcmp(client_message, "access_read")==0){
            char* nom = strtok(NULL, " ");
            char* res = p_access_read(nom, sock);
            write(sock, res, strlen(res));
        }
    }
}
```

```
/** 1) Récupération de valeur :
 * Format de messages envoyes : access_readwrite <nom>
 * Format de messages recus : access_readwrite <nom> OK <val>
 *                               ou      : access_readwrite <nom> KO <details>
 * 2) Renvoi de modification :
 * Format de messages envoyes : release_readwrite <nom> <val>
 * Format de messages recus : release_readwrite <nom> OK <details>
 *                               ou      : release_readwrite <nom> KO <details>
 */
else if(strcmp(client_message, "access_readwrite")==0){
    char* nom = strtok(NULL, " ");
    char* res = p_access_readwrite(nom, sock);
    write(sock, res, strlen(res));
    continue;
}

else if(strcmp(client_message, "release_readwrite")==0){
    char* nom = strtok(NULL, " ");
    char *val = strtok(NULL, " ");
    char* res = p_release_readwrite(nom, sock, val);
    write(sock, res, strlen(res));
    continue;
}

else{
    char *res = "Error : request not known";
    write(sock, res, strlen(res));
    continue;
}

if(read_size == 0){
    puts("Client disconnected");
    // libérer ces ressources
    fflush(stdout);
}
```

Messages normalisés

```
/** Format de messages recus : create <nom> <val>  
 *   Format de messages envoyes : create <nom> OK <details>  
 *                                   ou      : create <nom> KO <details>  
 **/
```

```
/**  
 *   Format de messages recus : access_read <nom>  
 *   Format de messages envoyes : access_read <nom> OK <val>  
 *                                   ou      : access_read <nom> KO <details>  
 **/
```

```
/**  
 *   Format de messages recus : access_readwrite <nom>  
 *   Format de messages envoyes : access_readwrite <nom> OK <val>  
 *                                   ou      : access_readwrite <nom> KO <details>  
 **/
```

Three black server racks are positioned on a dark, reflective surface. Each rack has several green lights at the bottom, which are reflected on the surface below. The background is a dark, gradient grey.

Services & Synchronisation

Quelques structures & normes

```
#define NB_PAGES 10//00000// On peut créer jusqu'à pow(2,50) pages dans une server consacré à cette tâche
// mais nous allons tester dans le cadre du projet des serveurs offrant le stockage de 10 /00000 pages/données au plus
```

```
/**
 * Lors de l'allocation de pages pour le stockage de données
 * une structure comme celle-ci sera initialisée pour y contenir les informations
 */
```

```
struct page{
    void* begin;           // Adresse de départ
    char* name;            // Si page vide/non utilisée, name = NULL
    struct client* lects;  // Lecteur(s) en cours de lecture
    int writer;
    struct page *next;     // Liste chaînée avec la page suivante indiquée
    pthread_mutex_t *mutex;
};
```

```
struct unusedPages{
    struct page *list;
    int nbUnused;
    pthread_mutex_t *mutex;
};
struct unusedPages* unused;
```

```
struct usedPages{
    struct page *list;
    int nbUsed;
    pthread_mutex_t *mutex;
};
struct usedPages* used;
```

```
struct client{
    int sock;
    struct client *next;
};
```

```
struct page{
    void* begin;
    char* name;
    struct client* lects;
    int writer;
    struct page *next;
    pthread_mutex_t *mutex;
};
```


Quelques structures & normes

```
int init_persistent(){
    unused=(struct unusedPages*) malloc(sizeof(struct unusedPages*));
    unused->nbUnused=0;
    pthread_mutex_init(&unused->mutex, NULL);

    used=(struct usedPages*) malloc(sizeof(struct usedPages*));
    used->nbUsed=0;
    pthread_mutex_init(&used->mutex, NULL);

    psize = getpagesize();    /// En général 4096 octets
    printf("Page size = %d\n", psize);
    int *addr;

    /** On peut créer 4503599627370495 pages en mémoire (250) mais on va en créer que 1.000.000.000 dans le cadre de ce projet
     * On boucle 1.000.000.000 fois et alloue des pages mémoire : meilleure solution selon moi car on fixe le nombre de pages
     */

    pthread_mutex_lock(&unused->mutex);

    while((unused->nbUnused < NB_PAGES) && (addr=malloc(psize))!=NULL){
        struct page* nouveau;
        nouveau=(struct page*) malloc(sizeof(struct page*));
        nouveau->begin = addr;
        nouveau->lects = NULL;
        pthread_mutex_init(&nouveau->mutex, NULL);
        nouveau->next=unused->list;
        unused->list=nouveau;
        unused->nbUnused++;
        printf("Page n°%d:    @:%p\n", unused->nbUnused, nouveau->begin);
    }
    pthread_mutex_unlock(&unused->mutex);

    // appeler méthode répliquage initiale sur serveur
    printf("\n%d page(s) créées\n", unused->nbUnused);
    return unused->nbUnused;
}
```

Côté Service VS Côté Synchro

Création :

Avoir une référence sur la donnée

Lecture & Écriture :

Avoir une référence sur les lecteurs & l'écrivain

→ Pour assurer une exclusion mutuelle

Verrou :

Garantie la cohérence des données en interne

Création, Lecture & Écriture :

Avoir toutes les informations sur la donnée, par « copie » indépendamment du serveur ayant créé, ou modifié

i.e. :

Alerte des serveurs à chaque modification

Copie chez chacun d'eux de l'information

Poursuite du traitement avant de répondre

```

int creer(int sock, struct sockaddr_in server, char* nom, char* val){
    char buf[4096];
    sprintf(buf, "create %s %s", nom, val);
    printf("%s\n", buf);

    /** Demande de creation **/
    if(sendto(sock, buf, sizeof(buf), 0, (struct sockaddr *)&server, (socklen_t)sizeof(server))==-1){
        printf("Creation - envoi échoué: %s\n", strerror(errno));
        return 0;
    }

    /** Reception de reponse **/
    if(recvfrom(sock, buf, 10, 0, (struct sockaddr *)&server, (socklen_t *)&sizeof(server))<sizeof("create")+sizeof(nom)+sizeof(" xx")){
        printf("Creation - réception échouée: %s\n", strerror(errno));
        return 0;
    }

    char *res = buf;
    strtok(res, " ");
    int i = 0;
    while (res != NULL){
        if(i==0 && strcmp(res, "create")!=0){
            break;
        }
        if(i==1 && strcmp(res, nom)!=0){
            break;
        }
        if(i==2 && strcmp(res, "OK")==0){
            printf("Creation reussie :\n");
            while((res=strtok(NULL, " "))!=NULL){
                printf("%s ", res);
            }
            printf("\n");
            return 1;
        }
        else if(i==2 && strcmp(res, "KO")==0){
            printf("Creation echouee :\n");
            while((res=strtok(NULL, " "))!=NULL){
                printf("%s ", res);
            }
            printf("\n");
            return 0;
        }
        else if(i==2 && strcmp(res, "OK")!=0 && strcmp(res, "KO")!=0){
            break;
        }
        res = strtok (NULL, " ");
        i++;
    }
    printf("Creation Echouee: Erreur message format\n");

    return 0;
}

```

```

char* p_create(char* name, char* val, int sock){
    static char res[4096];
    if(getPage(name)!=NULL){
        sprintf(res, "create %s KO Error - Page already exists", name);
        return res;
    }
    pthread_mutex_lock(used->mutex);
    pthread_mutex_lock(unused->mutex);
    if(used->nbUsed==NB_PAGES || unused->nbUnused==0){
        /// Si déjà 1000000000 de pages allouées OU une page portant ce nom existe OU 0 page libre disponible --> false
        sprintf(res, "create %s KO Error Server - Server Memory full", name);
        pthread_mutex_unlock(used->mutex);
        pthread_mutex_unlock(unused->mutex);
        return res;
    }

    /// Retrait d'une page depuis la liste des pages non utilisees
    struct page *p = unused->list;
    unused->list = unused->list->next;
    pthread_mutex_unlock(unused->mutex);

    if(p!=NULL){
        /// Poser le verrou de la page
        pthread_mutex_lock(p->mutex);

        p->name=name;

        if(mmap(p->begin, psize, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)==MAP_FAILED){ // mmap fonction
            /// On le remet dans la liste des non utilisés
            p->name=NULL;
            p->next=unused->list;
            unused->list=p;
            pthread_mutex_unlock(p->mutex);
            sprintf(res, "create %s KO Error Server - Memory Error", name);
            return res;
        }
        memcpy(p->begin, val, sizeof(*val));
        p->next = used->list;
        used->list = p;
        used->nbUsed++;
        /*printf("New page used: Name=%s Size=%zu octets @=%p\n", p->name, psize, p->begin);*/
        pthread_mutex_unlock(used->mutex);

        // Prevenir les autres
        prevent_create(name, val);

        /// Deverrouiller l'accès à la page
        pthread_mutex_unlock(p->mutex);
        sprintf(res, "create %s OK Page created with success !", name);
    }
    else{
        sprintf(res, "create %s KO Error Server - Could not create page", name);
    }
    return res;
}

```

```

int modifier(int sock, struct sockaddr_in server, char* nom){

    char buf[4096];
    sprintf(buf, "access_readwrite %s", nom);
    printf("%s\n", buf);

    /* Récupération de valeur */
    /** Demande de lecture/écriture **/
    if(sendto(sock, buf, sizeof(buf), 0, (struct sockaddr *)&server, (socklen_t)sizeof(server))==-1){
        printf("Lecture/Ecriture - envoi échoué: %s\n", strerror(errno));
        return 0;
    }

    /** Reception de resultat **/
    if(recvfrom(sock, buf, 10, 0, (struct sockaddr *)&server, (socklen_t *)&server)<sizeof("access_readwrite")){
        printf("Lecture/Ecriture - réception échouée: %s\n", strerror(errno));
        return 0;
    }

    char* res = strtok(buf, " ");
    int i = 0;
    while (res != NULL){
        if(i==0 && strcmp(res, "access_readwrite")!=0){
            break;
        }
        if(i==0 && strcmp(res, nom)!=0){
            break;
        }
        if(i==2 && strcmp(res, "KO")==0){
            printf("Lecture/Ecriture echouee :\n");
            while((res = strtok (NULL, " "))!=NULL){
                printf("%s ", res);
            }
            printf("\n");
            return 0;
        }
        if(i==2 && strcmp(res, "OK")==0){
            res = strtok (NULL, " ");
            printf("Lecture/Ecriture reussie :\n");
            while((res = strtok (NULL, " "))!=NULL){
                printf("%s ", res);
            }
            printf("\n");
            goto writing;
        }
        else if(i==2 && strcmp(res, "OK")!=0 && strcmp(res, "KO")!=0){
            break;
        }
        res = strtok (NULL, " ");
        i++;
    }
    printf("Ecriture Echouee: Erreur message format\n");
    return 0;
}

```

```

/* Renvoi de modification - Fin Lecture/Ecriture */
writing:
    sprintf(buf, "release_readwrite %s ", nom);
    //printf("%s\n", buf);
    printf("Veuillez entrez la nouvelle valeur :\n(Termez votre saisie en saisissant 'fin')\n");
    char tmp[4096];
    while(scanf("%s", tmp) && strcmp(tmp, "fin")!=0){
        strcat(buf, tmp);
    }

    if(sendto(sock, buf, sizeof(buf), 0, (struct sockaddr *)&server, (socklen_t)sizeof(server))==-1){
        printf("Ecriture - envoi échoué: %s\n", strerror(errno));
        goto writing;
    }
    while(recvfrom(sock, buf, 10, 0, (struct sockaddr *)&server, (socklen_t *)&server)<sizeof("release_readwrite")){
        printf("Ecriture - réception échouée: %s\n", strerror(errno));
    }
    res = strtok(buf, " ");
    i = 0;
    while (res != NULL){
        if(i==0 && strcmp(res, "release_readwrite")!=0){
            break;
        }
        if(i==1 && strcmp(res, nom)!=0){
            break;
        }
        if(i==2 && strcmp(res, "OK")==0){
            /** Affichage de resultat -> lecture seule **/
            printf("Fin Ecriture reussie : %s\n", res);
            while((res = strtok (NULL, " "))!=NULL){
                printf("%s ", res);
            }
            printf("\n");
            return 1;
        }
        if(i==2 && strcmp(res, "KO")==0){
            res = strtok (NULL, " ");
            printf("Fin Ecriture echouee : %s\n", res);
            goto writing;
        }
        else if(i==2 && strcmp(res, "OK")!=0 && strcmp(res, "KO")!=0){
            break;
        }
        res = strtok (NULL, " ");
        i++;
    }
    printf("Fin Ecriture Echouee: Erreur message format\n");
    return 0;
}

```

```

char* p_access_readwrite(char *name ,int sock){    // prévoir le cas où un client
    struct page* p = getPage(name);
    static char res[4096];
    if(p==NULL){    /// La page n'existe pas
        sprintf(res, "access_readwrite %s KO Error - Page doesn't exist", name);
        return res;
    }

    while(p->writer!=0 || p->lects!=NULL){    /// Si une écriture en cours OU u
        sleep(2); // attente active, pas le plus optimal
    }

    /// Poser le verrou de la page
    pthread_mutex_lock(p->mutex);

    char* dest=malloc(sizeof(psize));
    memcpy(dest, p->begin, psize);
    sprintf(res, "access_readwrite %s OK %s", name, dest);
    /** printf("p_access_readwrite %s %p '%s'\n", p->name, p->begin, dest);**/

    // Prévenir les autres
    p->writer=sock;
    prevent_access_readwrite(name, sock);

    /// Déverrouiller l'accès à la page
    pthread_mutex_unlock(p->mutex);
    return res;
}

```



```

char* p_release_readwrite(char* name, int sock, char* buf){
    struct page *p = getPage(name);
    static char res[4096];
    if(p==NULL){          /// La page n'existe pas
        sprintf(res, "release_readwrite %s KO Error - Page doesn't exist", name);
        return res;
    }
    if(p->writer==sock){
        /// Poser le verrou de la page
        pthread_mutex_lock(p->mutex);

        char* buf=malloc(psize);
        read(sock, buf, psize);

        if(memcpy(p->begin, buf, psize)==NULL||mprotect(p->begin, psize, PROT_NONE)==-1){
            pthread_mutex_unlock(p->mutex);
            sprintf(res, "release_readwrite %s KO Error Server - Memory error", name);
            return res;
        }
        p->writer=0;

        // prévenir les autres
        prevent_release_readwrite(name, sock, buf);

        /// Déverrouiller l'accès à la page
        pthread_mutex_unlock(p->mutex);
        sprintf(res, "release_readwrite %s OK Released successfully", name);
    }
    else{
        sprintf(res, "release_readwrite %s KO Error - You're not a writer of this page", name);
    }
    return res;
}

```

Schéma modélisant des lectures dans le système

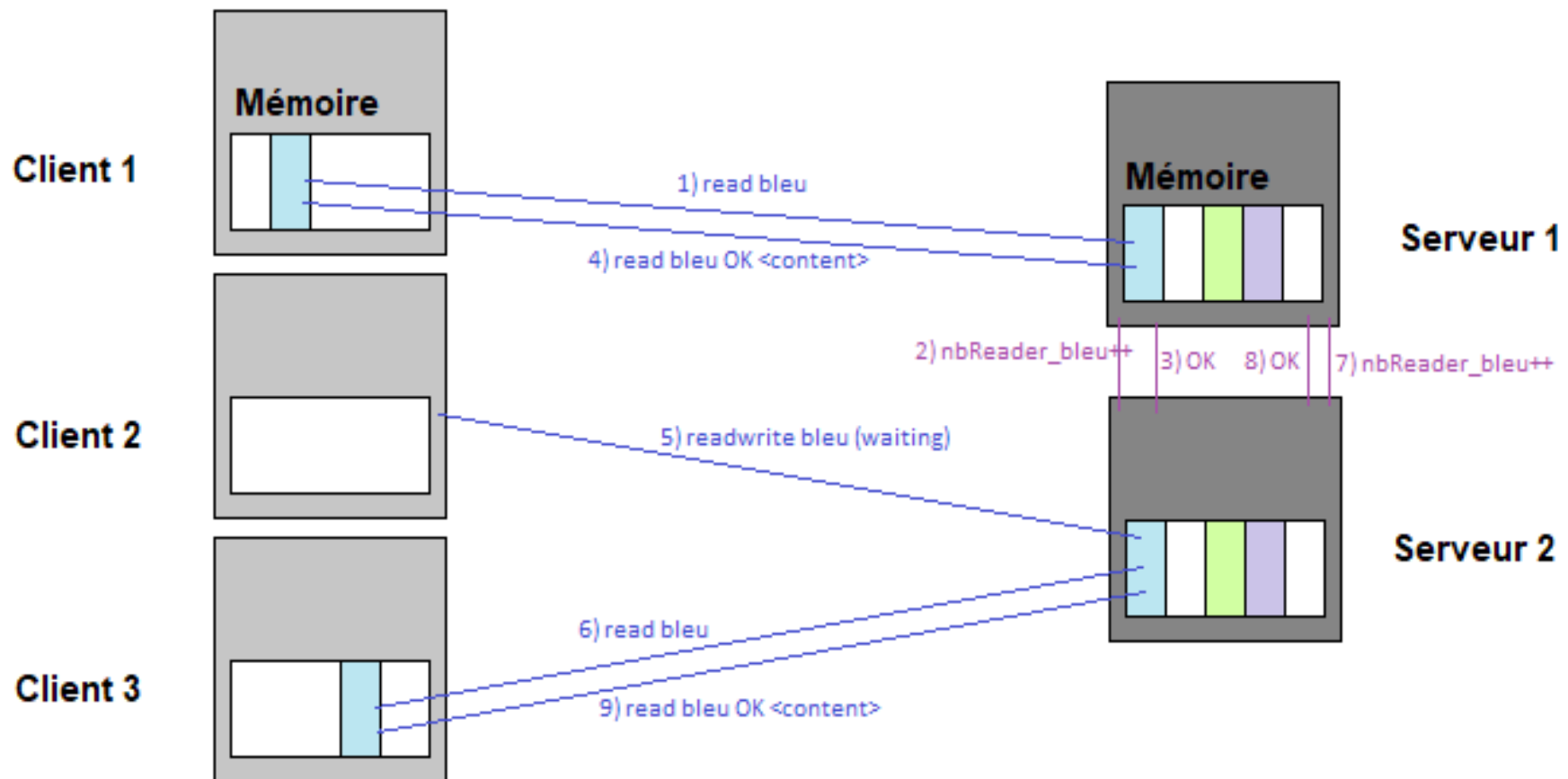


Schéma modélisant une écriture dans le système

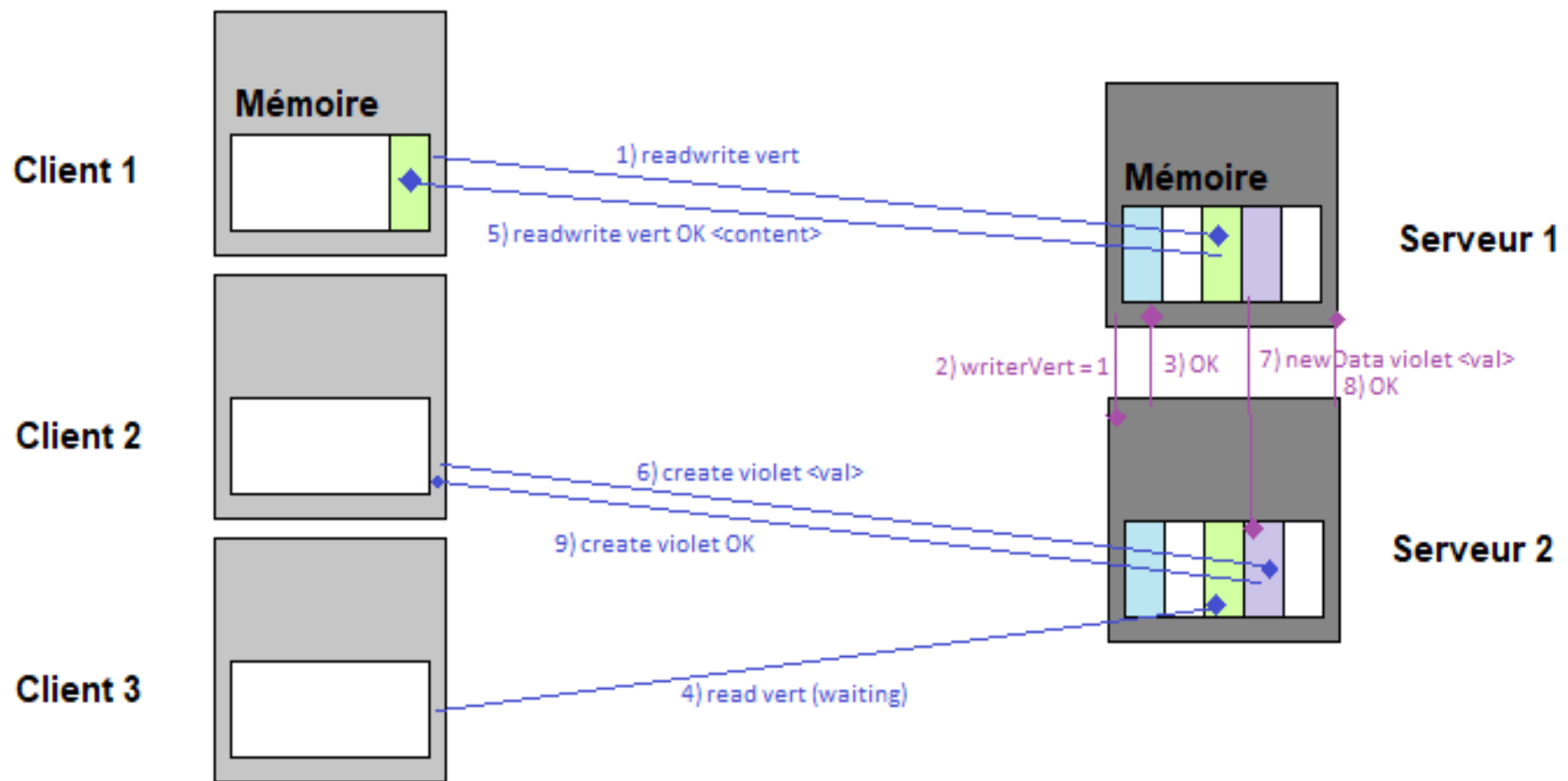
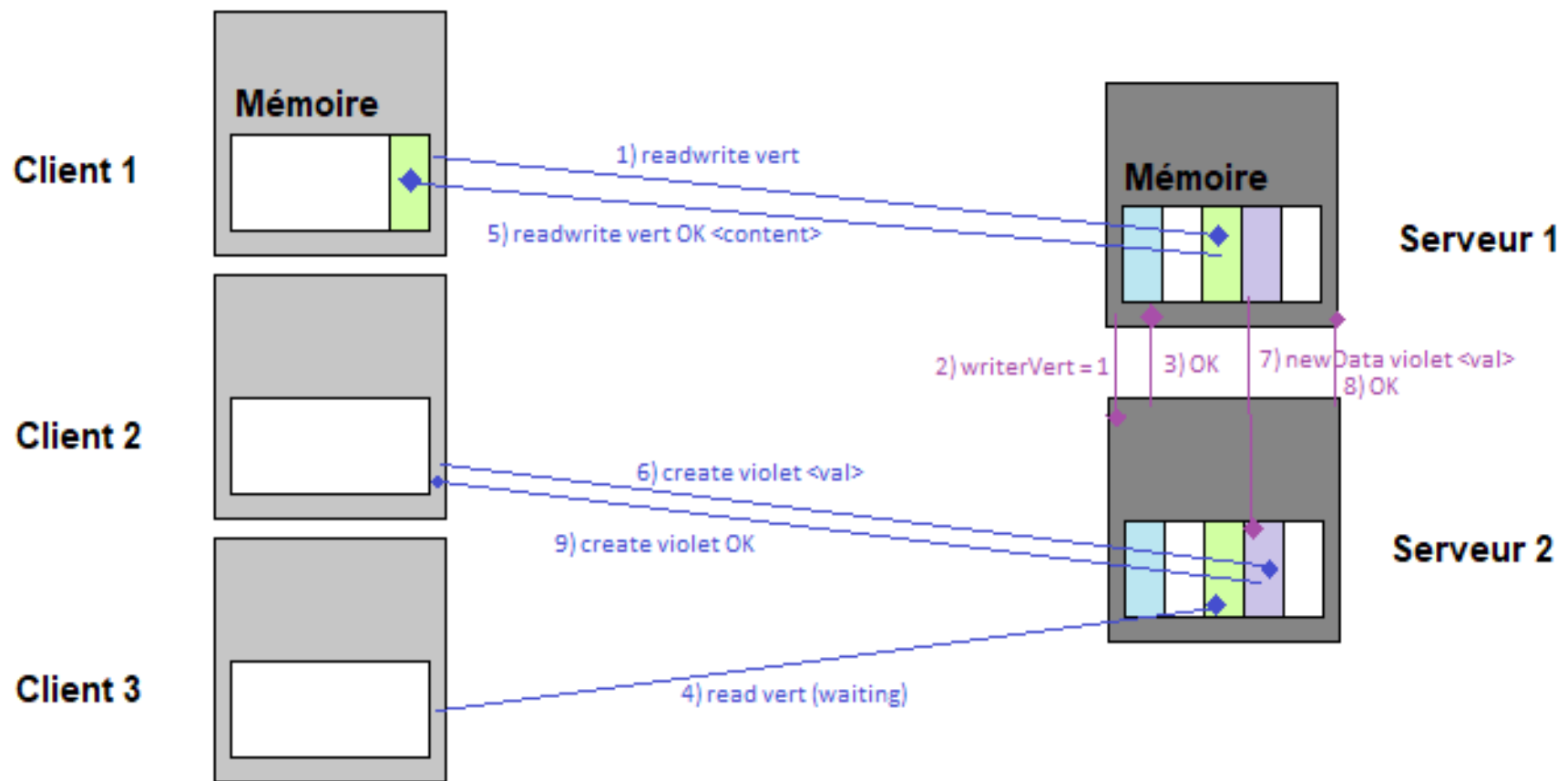


Schéma modélisant une écriture dans le système



Three black server racks are positioned on a dark, reflective surface. Each rack has several horizontal green light bars at the bottom, suggesting active cooling or power. The racks are slightly staggered, with the middle one being the most prominent. The background is a dark, gradient grey.

Quelques soucis