Implantation d'un framework tolérant aux pannes pour la résolution de cas de calcul numérique

Sommaire

Afin de valider les concepts présentés dans les chapitres 2 et 3, nous avons réalisé le framework *MCA* (pour *M*obile *C*omputing *A*rchitecture), développé en Java, permettant la résolution de cas de calcul numériques dans un environnement distribué hétérogène.

Avant de présenter notre framework dans la suite de ce chapitre, il est important de souligner que les systèmes distribués sont fondamentalement différents des systèmes non-distribués. En effet,

dans un système distribué, il existe des situations dans lesquelles des membres du système ne sont plus en mesure de communiquer avec les autres membres du même système. Cela arrive en général pour deux raisons principales :
– Soit parce que l'un des membres de la communauté est en panne ;
– Soit parce que la connexion entre les membres de la communauté ne fonctionne plus.
Ce type de défaillance, dite partielle -c'est à dire la défaillance d'une partie du système-, peut survenir à tout moment et peut être intermittente ou de longue durée.

Un des principaux challenges de la mise en place d'un système distribué est d'être tolérant à ces différentes pannes. En effet, la propriété de réagir face à une défaillance dans un tel système complique considérablement le travail des développeurs. En contrepartie, rendre un système distribué tolérant aux pannes est une marque de qualité et un atout indéniable pour se différencier des autres systèmes du même type. Les composants mis en relation dans ce type de système proposent des ressources ou des services aux autres composants du système. La panne d'un ou plusieurs de ces composants peut conduire, par exemple, à des ressources inutilisées non libérées ou à des services qui continuent de s'exécuter alors que le client du service n'attend plus de réponse. Ces situations entrainent un système distribué vers une consommation inutile des ressources ou, plus grave encore, à un blocage total du système. Du point de vue de l'utilisateur final, cela signifie ne par voir ces pannes et donc ne pas perdre le temps de calcul déjà consommé (transparence de panne).

Dans ce chapitre, nous commençons par la présentation générale de l'architecture logicielle définie par notre framework *MCA* (Section 4.1). Ensuite, nous listons les différents outils et paradigmes proposés par l'API Jini™ et utilisés par notre framework MCA, comme la notion de services et de mobilité (Section 4.2) et la technologie *JavaSpaces* (Section 4.3). La section 4.4 détaille les différents éléments qui composent cette architecture. Enfin, la section 4.5 propose un bilan des solutions offertes par le framework *MCA*.

4.1 Présentation générale de l'architecture *MCA*

4.1.1 Apport de notre modèle formel

Les résultats obtenus dans le chapitre 2 ont abouti à la définition des exigences pour une plate-forme logicielle de calcul numérique où les transparences de localisation, d'échelle et de concurrence sont respectées. Ainsi, les différents termes π-calcul écrits dans le chapitre 2 se concrétisent par les composants présentées dans ce chapitre (cf. Figure 4.1) :
– Le terme $CaseDirectory$ (eq. 2.61) se concrétise par le service *MCAService* ;
– Le terme $Worker$ (eq. 2.64) se concrétise par un agent *MCAWorker* et le terme $Worker^m$ (eq. 2.37) se concrétise par l'ensemble des agents $MCAWorker$;
– le terme $ComputeAgent$ (eq. 2.41) se concrétise par un agent mobile de type *ComputeAgent* ;

– Le terme *ComputationCase* (eq. 2.58) se concrétise par le composant *ComputationSpace*. Celui-ci est un *space* et concrétise également les annuaires modélisés par les termes $TaskDirectory$ (eq. 2.20), $DHDirectory$ (eq. 2.40) et $PropertyDirectory$ (eq. 2.49). Le terme $Master$ (eq. 2.60) est lui concrétisé par un agent mobile d'un type de *ComputeAgent*. Enfin, le rôle modélisé par le terme $CaseHanler$ (eq. 2.59) se concrétise par les fonctionnalités offertes par l'utilisation d'un *space*.

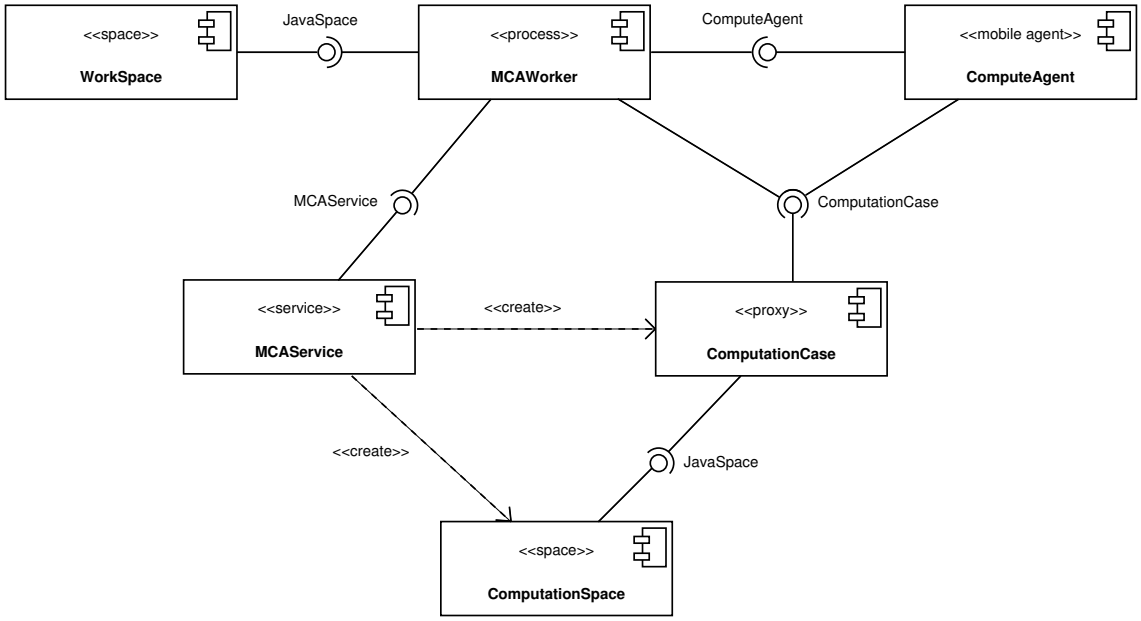


FIGURE 4.1 – Diagramme de composants de l'architecture logicielle définie par le framework *MCA*.

Notons que le composant $WorkSpace$ n'a pas été formellement modélisé mais celui-ci est utilisé dans notre architecture pour répondre à une exigence de tolérance aux pannes des agents *MCAWorker*.

4.1.2 Survol de notre architecture logicielle

Notre framework *MCA* permet de mettre en place une architecture logicielle dédiée à la résolution de cas de calcul numérique. Chaque cas de calcul est associé à un *ComputationSpace* qui contient les différents éléments nécessaires à la résolution du cas. L'ensemble des cas de calcul, chacun représenté par un *ComputationSpace*, forme la plate-forme de calcul MCA. Cette plate-forme est accessible par le service *MCAService* dont plusieurs instances sont actives afin de rendre ce service tolérant aux pannes. Ce service offre à un utilisateur la possibilité d'ajouter, de récupérer ou de supprimer un cas de calcul de la plate-forme. De plus, les agents *MCAWorker* se connectent à la plate-forme de calcul par le service *MCAService*. Ces agents ont alors la possibilité de participer à la résolution d'un cas de calcul via le composant *ComputationCase* qui joue le rôle de *proxy*. Si le nombre d'agents *MCAWorker* connectés à la plate-forme de calcul est supérieur à

la demande de ressources de calcul, alors les agents *MCAWorker* inactifs sont considérés comme disponibles (transparence de localisation). Ces derniers seront utilisés si un cas de calcul venait à demander de nouvelles ressources ou si un nouveau cas de calcul était ajouté à la plate-forme de calcul. La figure 4.2) propose une vue d'ensemble de cette architecture avec les composants *ComputationSpace*, *MCAService* et *MCAWorker*. Une vue plus précise est fournie par la figure 4.16 montrant les communications entre un agent *MCAWorker* et le service *MCAService*.

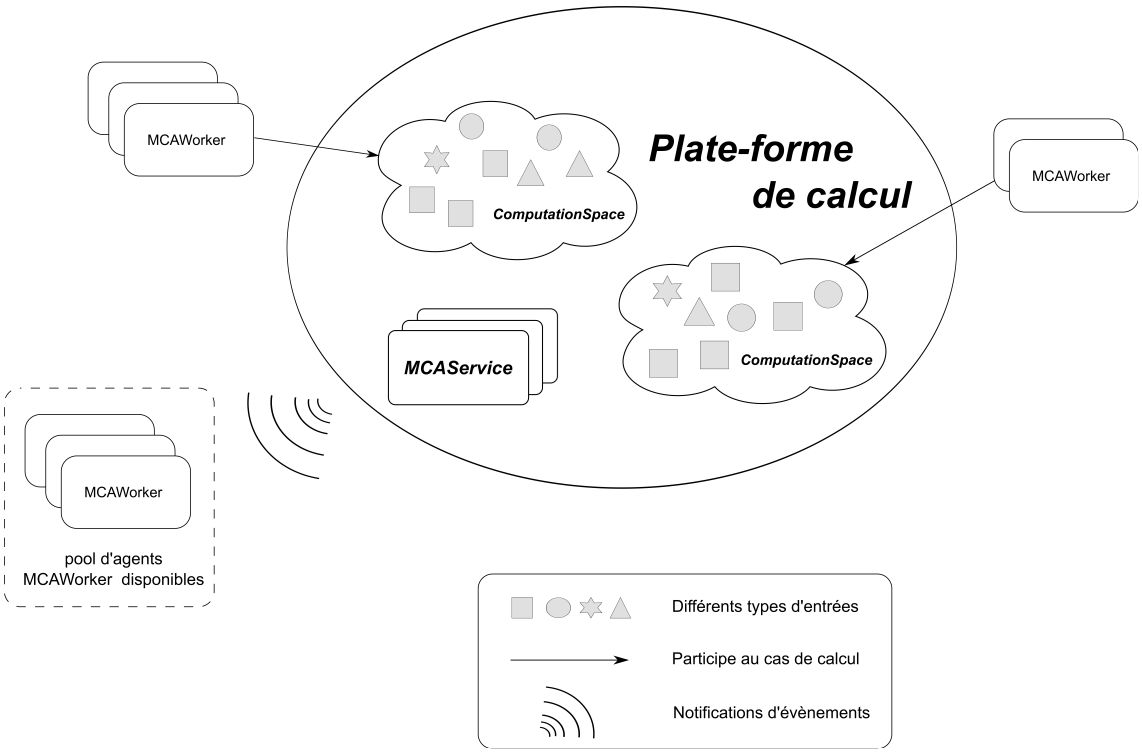


FIGURE 4.2 – Vue d'ensemble de l'architecture proposée par le framework *MCA*.

Les composants *ComputationSpace* et *WorkSpace* (mémoire locale d'un agent *MCAWorker*) sont des mémoires partagées, appelées « *spaces* » (cf. Section 4.3). Ils constituent le socle nécessaire pour rendre l'architecture définie par notre framework MCA tolérante aux pannes. La figure 4.1 présente les différents composants qui communiquent au sein de cette architecture.

Enfin, la résolution d'un cas de calcul est réalisée par les agents *MCAWorker*. Ceux-ci traitent les différentes tâches présentes dans le *ComputationSpace* associé au cas de calcul. À chaque tâche est associé un agent mobile de type *ComputeAgent* (cf. Section 4.4.3). Ce dernier contient le code à exécuter pour le traitement d'une tâche. Son aspect mobile lui permet de migrer vers la machine où est exécuté l'agent *MCAWorker* traitant la tâche. Le code est alors exécuté en local sur la machine de l'agent *MCAWorker*.

La suite du chapitre présente les différents éléments de notre framework en commençant par le socle logiciel sur lequel il se fonde.

4.2 Un environnement d'exécution adaptatif

Dans cette section nous présentons le socle logiciel sur lequel notre framework *MCA* se fonde. Dans un premier temps, nous abordons la technologie Jini™ (Section 4.2.1) car l'architecture *MCA* s'appuie essentiellement sur celle-ci avec l'utilisation des différents services de base qu'elle propose. Le service *MCAService* est un service *Jini* (Section 4.2.1.1) et, à ce titre, il dispose des mêmes caractéristiques que tous les autres services *Jini*, en particulier la possibilité de définir une politique de sécurité lors des invocations distantes de ses opérations (Section 4.2.1.4). Dans un second temps, nous présentons la mise en place d'un système d'agents mobiles utilisé pour les composants *ComputeAgent* et *ComputationCase* de l'architecture *MCA* (Section 4.2.2).

4.2.1 L'architecture orientée services offerte par la technologie Jini™

La technologie Jini™ [WT00] a été définie pour aider les développeurs de systèmes distribués à traiter de manière simple une des principales caractéristiques de ce type de système : l'adaptation au changement. Le changement dans une application distribuée se traduit par l'arrivée ou le départ de composants qui constituent le système. Un des problèmes qui en découle, et qui doit être absolument pris en compte, est la défaillance partielle, c'est à dire la défaillance d'une partie, et non de tout le système. En effet, dans un système distribué, des composants peuvent s'exécuter alors que d'autres sont arrêtés, ou alors tous les composants peuvent être opérationnels mais la connexion réseau peut être défaillante.

À ce jour, plusieurs implémentations de la spécification Jini™ sont disponibles : *JSC*, *Seven*, *Servicehost*, *Rio*, *Harvester*, *H2O* et *CoBRA*. L'ensemble de ces implémentations ont été présentées par Svetozar Misljencevic dans [Mis06]. Pour notre architecture logicielle, notre choix s'est porté vers l'implantation de référence car elle a déjà été utilisée par notre équipe sur d'autres travaux [Ber09] et l'expérience acquise ne peut être qu'un avantage. Cette implantation est développée par *Sun*, se nomme *Jini* [New06] et est disponible via le *Jini starter kit*.

4.2.1.1 Définition d'un service *Jini*

Jini™ est basé sur la notion de service et permet de créer des systèmes présentant une architecture orientée service. Un système adoptant l'architecture proposée par la technologie Jini™ est donc composé de divers services disponibles sur le réseau. Chaque service est accessible via un objet mandataire, appelé « *proxy* ». Lorsqu'un client désire utiliser un service, il doit en premier lieu obtenir un *proxy* de ce service. Le client invoque ensuite les méthodes à travers ce *proxy*. Ce dernier exécute alors la méthode désirée sur le service distant en prenant en charge les communications à travers le réseau.

L'arrivée d'un nouveau service, ou la migration d'un service vers une autre localité, sont des événements récurrents dans un système distribué. Malgré cela, un client doit pouvoir trouver

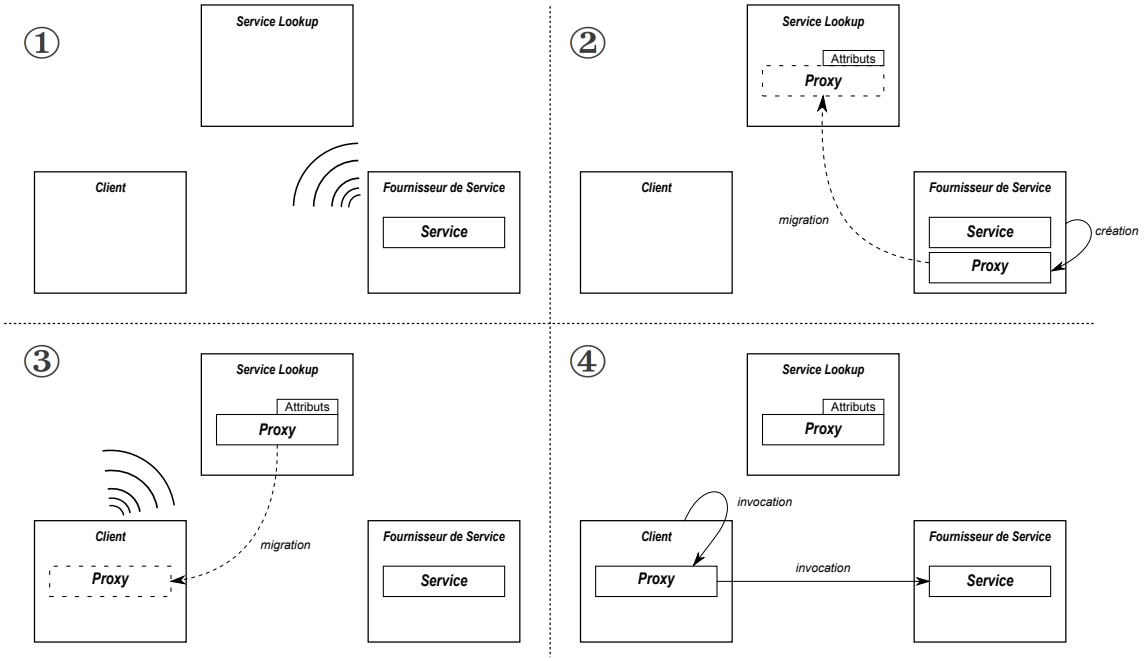


FIGURE 4.3 – Les protocoles mis en œuvre par la technologie Jini™ . ① *Découverte* : le fournisseur du service recherche le service *Lookup*. ② *Adhésion* : Le fournisseur crée et enregistre le proxy du service dans l'annuaire. ③ *Recherche* : Le client recherche le service en fonction de ses attributs. Le service *Lookup* lui transmet une copie du proxy associé. ④ *Invocation* : Le client invoque le service via le proxy associé.

facilement un service sur le réseau. Jini™ a une solution : le service *Lookup*. Celui-ci propose un annuaire dans lequel chaque service disponible sur le réseau enregistre son *proxy*. Le service *Lookup* est utilisé par un client pour découvrir des services sans connaître au préalable leur localisation. Si un client souhaite utiliser un service pour la première fois, il recherche le *proxy* correspondant à ce service dans un annuaire (*Lookup*). Soit le client connaît la localité d'un annuaire soit il utilise un ensemble de protocoles de découverte (*discovery protocols*) fournis par Jini™ pour trouver l'annuaire sur le réseau. Une fois que le client obtient le *proxy* du service désiré, il utilise ce *proxy* pour communiquer directement avec le service, et cette fois-ci sans la participation de l'annuaire. La figure 4.3 propose les différents protocoles que nous venons de décrire.

4.2.1.2 La notion de bail

Pour répondre aux possibles défaillances partielles d'un système distribué, la technologie Jini™ apporte une solution simple avec la notion de **bail**. Le concept de base est défini comme suit : il est préférable de définir un intervalle de temps restreint durant lequel un service peut être accessible plutôt que de laisser indéfiniment ce service accessible. Quand un client désire utiliser un service *Jini*, il doit demander au service de se maintenir dans un état durant lequel il pourra y accéder, le service fournit alors au client un bail. Ce bail est la période de temps durant laquelle le

service accepte de garder, dans la limite de ses possibilités, un état accessible pour le client. Cette période est déterminée par le service ou négociée entre le client et le service. Durant cette période, le client peut annuler le bail, ce qui permet au service de libérer les ressources qu'il utilisait pour maintenir un état correct vis-à-vis du client. Mais le client peut aussi vouloir renouveler le bail, dans ce cas le service peut renouveler le bail avec la période de temps demandée (ou avec une période plus courte[9]) ou refuser la demande de renouvellement. Si la période de temps du bail n'est pas renouvelée, alors le service est libre de libérer les ressources associées au maintien de l'état.

L'utilisation de baux évite à un service *Jini* d'utiliser des ressources inutilement. Par exemple, si un client demande à un service de se maintenir dans un état accessible et que ce même client a une panne, alors ce client sera incapable de renouveler le bail. Le bail expirera et le service pourra alors libérer les ressources utilisées. De la même manière, si le service et le client fonctionnent normalement mais que le réseau entre eux connait une défaillance, alors le client sera incapable de renouveler le bail ce qui permettra à l'expiration du bail de libérer les ressources utilisées inutilement.

La notion de bail est présente lors de l'enregistrement d'un service, comme par exemple le service *MCAService*, dans l'annuaire que propose le service *Lookup*. Lors de son enregistrement, le fournisseur du service obtient un bail créé par l'annuaire et lié au service qu'il vient d'enregistrer. Le service reste enregistré dans l'annuaire tant que le fournisseur de ce service renouvelle ce bail. Si le fournisseur du service vient à avoir une défaillance, le bail arrivera alors à expiration et le proxy du service sera supprimé de l'annuaire.

4.2.1.3 La définition d'un modèle d'invocation distante

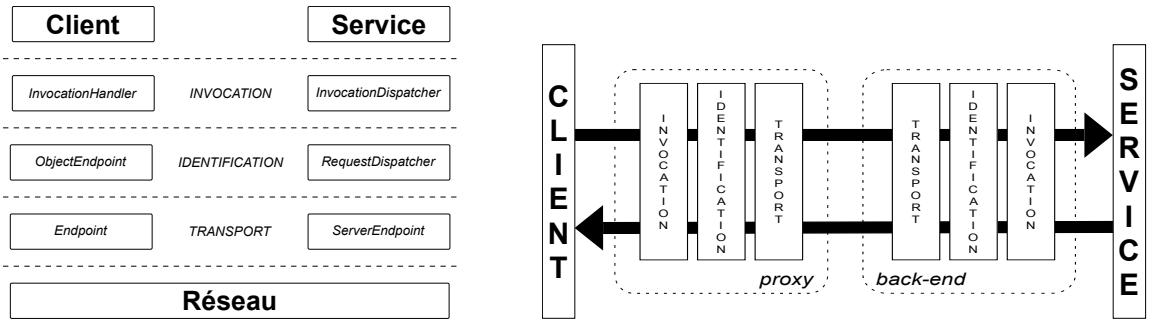
Le langage Java propose l'API *Remote Method Invocation* (*RMI*) pour effectuer des invocations de méthodes sur des objets distants. Cette API définit un modèle de programmation qui facilite la communication d'objets Java s'exécutant sur deux machines virtuelles distinctes. Cette communication est réalisée à l'aide de deux protocoles : le protocole *JRMP* (pour *Java Remote Method Protocol*) ou le protocole *IIOP* (pour *Internet Inter-Orb Protocol*).

Jini™ étend ce modèle avec le modèle *Jini ERI* (pour *J*ini *E*xtensible *R*emote *I*nvocation). Comme le montre la figure 4.4a, le modèle défini par *Jini ERI* se compose de trois couches : une couche d'invocation, une couche d'identification et une couche de transport.

– La *couche transport* fait communiquer les requêtes et les réponses à travers le réseau. Cette couche permet l'envoi d'une requête cliente au service à l'aide d'une instance de la classe `Endpoint` et contrôle ainsi son traitement par le service via une instance de la classe `ServerEndpoint` .

9. Nous ne rentrons volontairement pas dans le détail de la configuration Jini™ , mais lors de la configuration d'un service Jini™ il est possible de définir une période temps maximum pour un bail entre le service et un client.

– La *couche identification* permet de distinguer des objets distants lors de l'invocation de méthodes. Ainsi, coté client, une instance de `ObjectEndpoint` contient l'identifiant de l'objet distant ainsi qu'un point d'entrée pour communiquer les requêtes vers cet objet distant. Coté service, une instance de `RequestDispatcher` contient une table de correspondance associant un identifiant client au *proxy* utilisé par ce client.
– La *couche d'invocation* dirige le processus d'invocation distante. Coté client, une instance de `InvocationHandler` sérialise l'appel de la méthode avec ses paramètres éventuels et désérialise la réponse retournée par le service. Coté service, une instance de `InvocationDispatcher` désérialise les invocations faîtes par le *proxy* associé et sérialise la réponse qui est envoyée au client.



(a) (b)

FIGURE 4.4 – Les différentes couches du modèle *Jini ERI*.

Dans une application Jini™ , le client invoque une méthode du service désiré via un *proxy*. C'est le fournisseur du service qui lie le service avec un modèle d'invocation distante (*JRMP*, *IIOP* ou *JERI*) à l'aide d'un « service exporter ». Celui-ci crée, pour le modèle d'invocation désiré, deux parties distinctes qui permettent de faire communiquer un client avec le service :
– Une partie cliente, le *proxy* de la Figure 4.4b, qui est enregistrée sur un annuaire et utilisée par les clients du service ;
– Une partie serveur, le *back-end* de la Figure 4.4b, qui reste lié au service.

L'interface `net.jini.export.Exporter` généralise les différents moyens d'« exporter » un service. La création d'un *proxy* est définie par une implantation de cette interface. L'API proposée par Jini™ définit plusieurs implantations avec les classes `net.jini.jrmp.JrmpExporter` et `net.jini.iiop.IiopExporter` pour un modèle d'invocation distante utilisant respectivement les protocoles *JRMP* et *IIOP* définis par le modèle RMI.

Dans l'architecture proposée par le framework *MCA*, nous utilisons le modèle *Jini ERI* pour les invocations distantes des méthodes proposées par les services *MCAService*, *Lookup* et *JavaSpaces*. La classe `BasicJeriExporter` facilite la mise en place du modèle *Jini ERI* pour la couche d'invocation distante. Une instance de la classe `BasicJeriExporter` est créée pour chaque service proposé par l'architecture *MCA* et chaque « exporter » contient les informations permettant de contrôler l'implantation de chaque couche du modèle *Jini ERI* que ce soit du coté client ou du coté serveur.

4.2.1.4 Mise en place d'une politique de sécurité

Le modèle d'invocation de méthodes sur des objets distants demande de définir une politique de sécurité pour assurer les propriétés suivantes au sein de notre architecture :

- *Authentification* - Elle assure que chaque composant de l'architecture soit authentifié, qu'il soit un client ou un service. De cette façon, chaque composant est capable d'identifier le composant avec lequel il communique. Par exemple, un agent *MCAWorker* doit être authentifié avant de participer à la résolution d'un cas de calcul et un agent *MCAWorker* doit être sûr qu'il demande bien à un service *MCAService* authentifié.
- *Autorisation* - Un fois authentifié, un composant peut se voir accorder des droits. Par exemple, le service *MCAService* d'une plate-forme de calcul autorise seulement les utilisateurs authentifiés en tant qu'administrateur à ajouter un cas de calcul à la plate-forme.
- *Intégrité* - Elle assure que les données soient complètes et exactes après le transfert par le canal de communication. Comme par exemple, le résultat d'une tâche ne doit pas être corrompu.
- *Confidentialité* - Elle assure que les données ne soient lisibles uniquement par les composants autorisés. Par exemple, seuls les agents *Worker* participant à un cas de calcul pourront lire les données de ce cas.

Pour définir notre politique de sécurité lors de l'utilisation des différents services (*MCAService*, *ComputationSpace*, *WorkSpace*) au sein de notre architecture MCA, nous nous appuyons sur le protocole *Jini ERI* que nous venons de présenter. Jini™ part de l'hypothèse qu'un réseau n'est pas nécessairement sécurisé. Pour remédier à cela, Jini™ étend le modèle de sécurité de la plate-forme Java pour définir un nouveau modèle. Les services et les clients (de ces services) peuvent l'utiliser pour agir de façon sécurisée dans un réseau qui ne l'est pas.

Nous utilisons l'*API JAAS* (pour *Java Authentification and Authorization Service*) [JP01] pour autoriser un composant de s'exécuter sous une certaine identité (dite « *Principal* »). Chaque composant de notre architecture est donc exécuté dans un contexte authentifié. L'identité est définie par un certificat numérique *X.509*. Chaque certificat est signé par une autorité de confiance (CA), ce qui permet d'assurer la validité de l'identité du composant avec lequel on communique.

Sécurité coté serveur

Lors de la création d'un « service exporter » (voir le code de la figure 4.5), il est possible de définir la politique de sécurité utilisée pour l'invocation des méthodes de ce service. Deux éléments sont fournis pour paramétrer ce « service exporter » :

- Une fabrique de type `InvocationLayerFactory` (`BasicILFactory` dans la figure 4.5) qui définit comment créer un *proxy* (contenant un gestionnaire d'invocation de type `InvocationHandler`) destinés aux clients et un répartiteur d'invocation de type `InvocationDispatcher` pour le service.
- Un point d'entrée coté service (de type `ServerEndpoint`) pour la couche transport qui est aussi utilisé pour créer les points d'entrée (de type `Endpoint`) pour les clients de ce service.

Nous utilisons le protocole *TLS/SSL* (voir encadré sur le protocole *TLS/SSL*) dans la couche transport du modèle *Jini ERI*. La classe `SslServerEndpoint` permet d'utiliser *TLS/SSL* entre un client et le service désiré. Une instance de cette classe est passée au constructeur de la classe `BasicJeriExporter` afin que le *proxy* de chaque service de l'architecture *MCA* puisse utiliser *TLS/SSL*. Avec l'utilisation de ce protocole, les contraintes listées dans le tableau 4.1 sont implicitement respectées. Les communications entre les différents composants de l'architecture *MCA* s'appuient de ce fait sur les propriétés demandées : authentification, autorisation, intégrité et confidentialité.

```
1 SslServerEndpoint serviceEndpoint = SslServerEndpoint.getInstance(HOST,0);
2 MethodConstraints serviceConstraints = ...; // voir table 4.1
3 Class<?> permissionClass = MCASpacePermission.class;
4 BasicILFactory  serviceILFactory = new BasicILFactory(serviceConstraints, permissionClass);
5 Exporter exporter = new BasicJeriExporter(serviceEndpoint, serviceILFactory)
```

FIGURE 4.5 – Code source pour créer un « service exporter ».

Le protocole TLS/SSL

SSL (*Secure Socket Layer*) est un protocole standard fonctionnant sur TCP/IP. SSL suit le modèle client-serveur et fournit les services sécurisés suivants [FKK11] :
– Authentification du serveur et du client avec l'utilisation de certificat numérique *X.509*
– Confidentialité des données échangées à l'aide de chiffrement symétrique (comme par exemple l'algorithme *AES*[a]) et asymétrique (comme par exemple l'algorithme *RSA*[b])
– Intégrité des données échangées à l'aide d'une fonction de hachage comme le *SHA-1*[c]
Le protocole SSL peut aussi fonctionner sous d'autres protocoles de TCP/IP comme par exemple HTTP. SSL v3 est le standard SSL le plus généralement utilisé même si le protocole TLS (*Transport Layer Security*) élaboré par l'IETF (*Internet Engineering Task Force*) devient le nouveau nom du protocole SSL.

a. *AES* pour *Advanced Encryption Standard*
b. *RSA* pour *Rivest Shamir Adleman*
c. *SHA* pour *Secure Hash Algorithm*

Sécurité coté client

Avant de pouvoir invoquer des méthodes à travers un *proxy* reçu depuis le service *Lookup*, le client doit « préparer » ce *proxy* afin de s'assurer que les invocations se feront dans un contexte sécurisé. La « préparation » d'un *proxy* est effectuée par la méthode `prepareProxy` déclarée dans l'interface `ProxyPreparer` (cf. Figure 4.6) et suit les étapes suivantes :

– *Vérifier le proxy.* Le client commence par vérifier si le *proxy* est digne de confiance car un *proxy* peut avoir été reçu à partir d'une source non fiable ce qui peut représenter un danger. En effet si le client ne fait pas cette vérification alors il y a le danger que le proxy puisse ignorer les

contraintes imposées par le client et effectuer un transfert de données en clair (non chiffré) ou alors déformer l'identité du client ou du serveur par exemple. L'utilisation du protocole *TLS* dans l'architecture *MCA* assure la fiabilité des *proxies* utilisés.

– *Accorder des permissions.* Une fois que le client a vérifié le *proxy*, il peut lui accorder des permissions supplémentaires (en plus de celles déjà accordées au code non fiable), pour permettre aux futures invocations à travers ce proxy de fonctionner parfaitement. Par exemple, un *proxy* peut exiger une permission de type `AuthenticationPermission` afin qu'il puisse s'authentifier auprès du serveur. Il paraît donc évident qu'un client doit accorder des permissions supplémentaires à un *proxy* uniquement s'il lui fait entièrement confiance. Un *proxy* non fiable pourrait abuser de ses nouveaux droits pour causer de graves dommages sur la machine du client.

– *Définir des contraintes.* Le client peut appliquer des contraintes au *proxy*. Une contrainte est une exigence spécifique sur le comportement désiré lors de l'invocation d'une méthode sur le service distant à travers ce *proxy*. Il est important de noter que chaque contrainte exprime ce que représente la contrainte sur le contexte d'exécution mais non comment elle est vérifiée. C'est en effet le *proxy* qui est responsable d'utiliser les protocoles de transport appropriés pour satisfaire les exigences du client. Le tableau 4.1 liste les différentes classes de l'API qui permettent de définir des contraintes sur les invocations de méthode à travers un proxy.

```
1 KeyStore keyStore = KeyStores.getKeyStore("file:keystore.worker", null);
2 X500Principal clientUser = KeyStores.getX500Principal("server", keyStore);
3 ServerMinPrincipal smp = new ServerMinPrincipal(clientUser);
4 InvocationConstraints ics =
5   new InvocationConstraints( new InvocationConstraint[]{Integrity.YES,ServerAuthentication.YES,smp}, null);
6 ProxyPreparer preparer =
7   new BasicProxyPreparer(true, new BasicMethodConstraints(ics), new Permission[]{});
```

FIGURE 4.6 – Code source pour préparer un *proxy* avant l'invocation des méthodes du service associé.

4.2.2 Implantation d'un système d'agents mobiles

Le concept de base d'une architecture définie par Jini™ est la notion de service. Un service enregistre son *proxy* dans un annuaire, proposé par le service *Lookup*, et un client vient chercher ce *proxy* pour communiquer avec le service correspondant. Jini™ propose une autre façon d'enregistrer un service dans l'annuaire. Il est en effet possible d'enregistrer directement un service sans passer par la création d'un *proxy*. Dans ce cas, l'objet déposé dans l'annuaire est l'instance d'une classe implantant directement ou indirectement l'interface `java.io.Serializable`. Le client qui récupère cet objet, via la méthode `lookup`, exécute alors localement le code du service. Comme le code s'est déplacé, on parle de *code mobile*. L'instance qui contient ce code mobile est appelée

Classe	Valeurs	Description
`Integrity`	`YES/NO`	Exige l'intégrité du contenu du message. Ainsi lors de l'appel d'une méthode distante, le proxy va s'assurer que les données échangées demeurent intactes et que le code à télécharger est accessible par des URLs (*codebase*) assurant elles-mêmes une intégrité du contenu qu'elles proposent.
`Confidentiality`	`YES/NO`	Exige la confidentialité du contenu du message. Le proxy va ainsi s'assurer que le message est chiffré et qu'il ne peut être compris que par les deux parties de la communication.
`ClientAuthentication`	`YES/NO`	Exige du client d'être authentifié et empêche ainsi l'invocation d'une méthode par un client « anonyme ».
`ClientMinPrincipal`	`Principal[]`	Exige du client de s'authentifier comme l'une des identités (`Principal`) définies.
`ServerAuthentication`	`YES/NO`	Exige du serveur d'être authentifié et empêche ainsi l'invocation d'une méthode sur un serveur « anonyme ».
`ServerMinPrincipal`	`Principal[]`	Exige du serveur de s'authentifier comme l'une des identités (`Principal`) définies.

TABLE 4.1 – Classes appartenant au package `net.jini.core.constraint` utilisées pour définir des contraintes d'invocation sur un *proxy*.

« *agent mobile* », ce qui est le cas du *ComputeAgent* de l'architecture *MCA* (cf. Figure 4.1).

4.2.2.1 Migration réactive versus migration proactive

La section 1.4 nous a présenté que deux types de migration sont possibles pour un agent mobile : une migration *réactive*, c'est à dire que l'agent se déplace suite à la demande d'un autre agent, ou une migration *proactive*, dans ce cas l'agent mobile décide lui-même quand et où il doit se déplacer.

Une migration dite *réactive* est semblable à l'utilisation d'un service comme nous l'avons décrit dans la section 4.2.1. L'agent mobile est enregistré dans un annuaire via le service *Lookup* avec des attributs pour faciliter sa recherche. Le client voulant exécuter cet agent mobile récupère alors l'agent en local et non un *proxy* comme c'est le cas pour un service *Jini* classique. La figure 4.7 décrit les différentes étapes de la migration d'un agent mobile utilisant une migration réactive. Cette pratique est illustrée par l'utilisation d'un agent de type *ComputeAgent* faisant partie de notre architecture (cf. Section 4.4.3).

Jini™ ne permet pas de créer un agent mobile dit « *proactif* », mais la possibilité de surveiller l'activité d'un service *Lookup* permet de simuler ce type d'agent mobile. Deux composants sont ici mis en jeu : le premier est l'agent mobile lui même qui a une (ou parfois plusieurs) tâche à réaliser et le second est l'hôte qui accueille cet agent mobile dit « *proactif* ». Un processus hôte offre un environnement d'exécution à un agent mobile et ce dernier ne peut se déplacer uniquement vers une localité qui dispose de ce processus hôte. Ce dernier est composé d'un service *Lookup* et d'un *Agent Listener* (instance de la classe `MobileAgentListener`) qui surveille l'activité du service *Lookup*, en particulier l'arrivée d'un agent mobile (instance d'une classe héritant de la classe

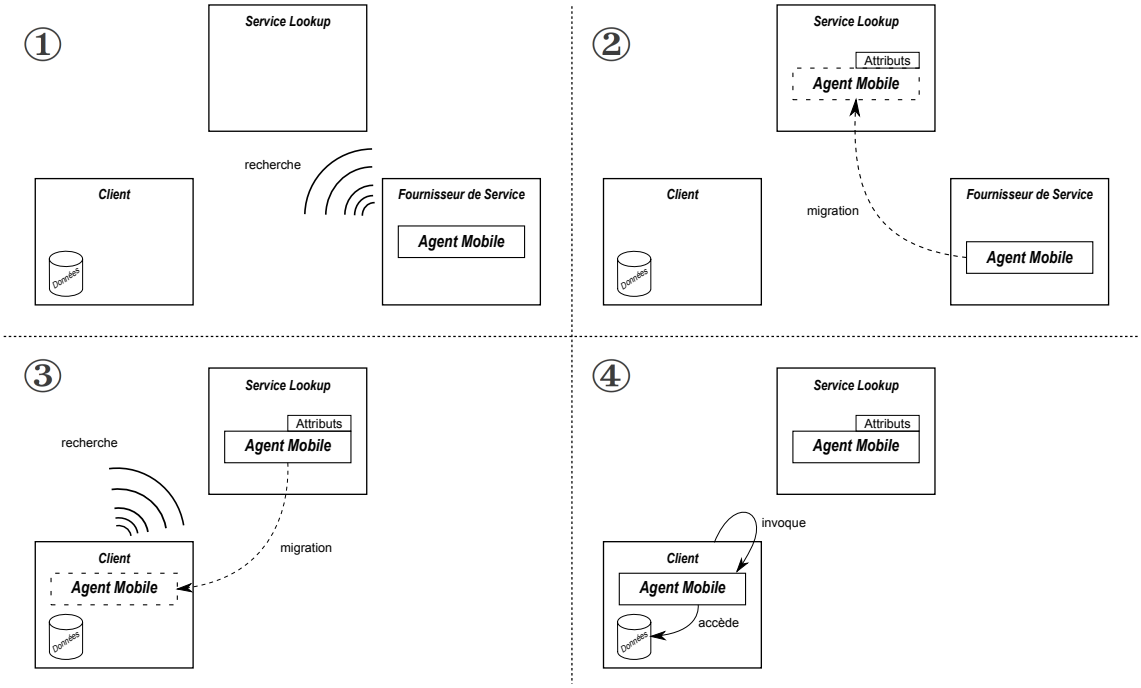


FIGURE 4.7 – Les différentes étapes d'une migration *réactive*. ① Le fournisseur du service recherche le service *Lookup*. ② Le fournisseur du service enregistre l'agent mobile dans l'annuaire en indiquant certains attributs pour faciliter sa recherche. ③ Le client recherche l'agent mobile en fonction de ses attributs. Le service *Lookup* lui transmet une copie de l'agent mobile associé. ④ Le client invoque une méthode de l'agent mobile. Le code est alors exécuté sur la machine du client.

abstraite `MobileAgent`). A l'arrivée d'un agent mobile, le processus *AgentListener* est prévenu (via la méthode `notify`) et peut exécuter le code de l'agent mobile via la méthode `execute`. A la fin de son exécution, l'agent mobile migre vers un autre hôte via sa méthode `move` (cf. Figure 4.8).

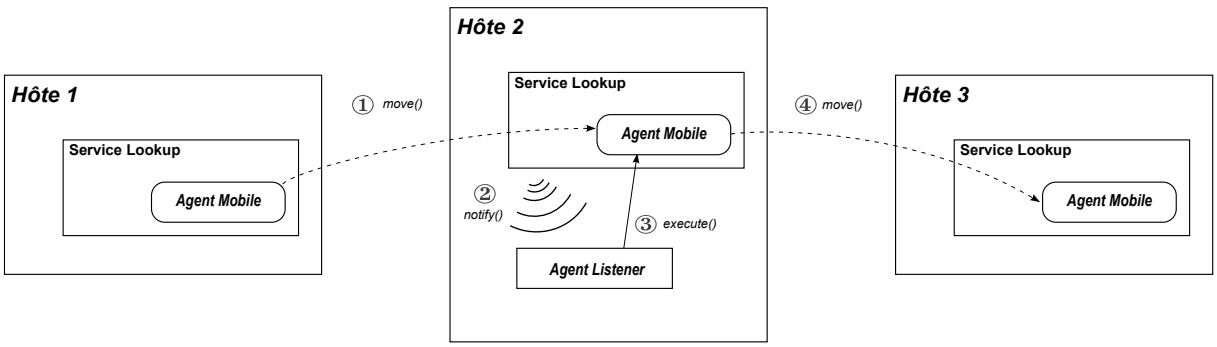


FIGURE 4.8 – Composants nécessaires à la mise en place d'un agent mobile *proactif*. ① L'agent mobile migre de l'*Hôte 1* vers l'*Hôte 2*. ② L'*AgentListener* est notifié de l'arrivée de l'agent mobile. ③ L'*AgentListener* de l'*Hôte 2* exécute le code de l'agent mobile. ④ A la fin de son exécution, l'agent mobile migre de l'*Hôte 2* ver l'*Hôte 3*.

4.2.2.2 Une politique de sécurité adaptée à la mobilité de code

L'accueil d'un agent mobile par un processus hôte est une action qui comporte un risque maximal (cf. Section 1.4.3). Parmi les différentes techniques proposées, la mise en place d'agents mobiles dans un environnement d'exécution comme celui proposé par *Java* permet de mettre en pratique les techniques suivantes :

La technique du bac à sable - L'utilisation du langage *Java* pour développer notre framework nous permet de profiter de la possibilité de pouvoir limiter les droits à un programme qui s'exécute dans une JVM. Ainsi, lors de l'exécution d'un agent mobile, l'hôte peut facilement limiter les possibilités de celui-ci (cf. Figure 4.9).

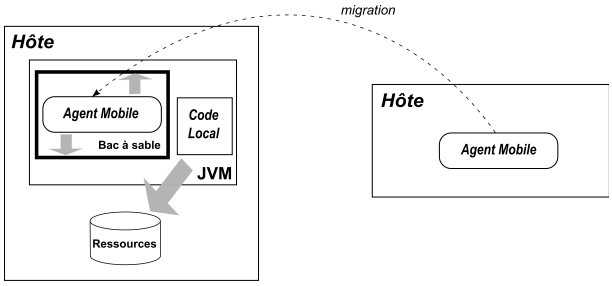


FIGURE 4.9 – La technique du bac à sable.

La signature du code - La signature du code intervient lors de la création d'un agent. Son créateur le signe numériquement. En réalité, il signe le fichier *jar* contenant la définition des classes Java. De cette façon, l'agent mobile est authentifié durant ses déplacements (cf. Figure 4.10). Dans notre cas, avec la technologie Jini™ , l'hôte télécharge la définition des classes de l'agent mobile à l'aide du *codebase* défini lors du déploiement de l'agent mobile via le service *Lookup*. Cette technique permet d'obtenir une authentification de haut niveau pour les hôtes. Elle assure aussi l'intégrité du code pour l'hôte visité. Une signature digitale sert donc de moyen de confirmation de l'authenticité de l'agent mobile, de son origine et de son intégrité.

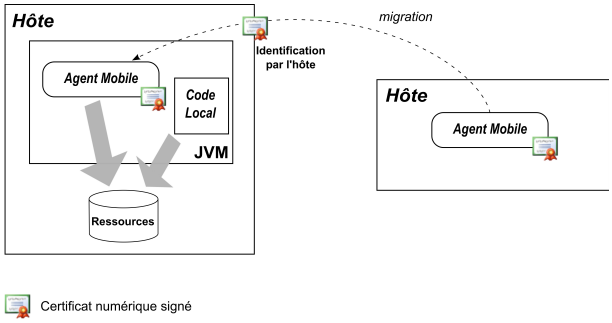


FIGURE 4.10 – Signature du code mobile.

Le contrôle d'accès - Un programme qui s'exécute sur un système doit accéder à des ressources pour réaliser sa tâche. Un agent mobile doit avoir un environnement d'exécution restreint pour des

raisons de sécurité. C'est pourquoi les accès aux ressources, locales à l'hôte, dont il a besoin pour traiter sa tâche, doivent donc être rigoureusement contrôlés. Pour améliorer les deux techniques précédentes, une politique de contrôle d'accès plus complexe est mise en place. La gestion des droits d'accès aux ressources pour un agent mobile passe par l'élaboration d'une politique de sécurité. Il est possible de choisir entre les politiques suivantes :

– L'hôte autorise l'accès à toutes les ressources pour tous les agents mobiles.
– Tous les agents mobiles sont soumis à la même politique sur l'hôte.
– Une négociation a lieu pour chaque agent mobile.

Il est essentiel pour un système hôte d'être capable d'authentifier les agents mobiles qu'il accueille. L'identité du signataire du code mobile permet de raffiner la définition de la politique de sécurité à l'aide des moyens offerts par la technique de clé publique. Cette identification est nécessaire pour lier des droits à un agent mobile identifié. Pour cela les droits et les permissions doivent être définis par l'hôte avant de recevoir un agent mobile. Le contrôle d'accès applique les droits et les restrictions avant l'exécution du code mobile afin de prévenir l'accès illégal aux ressources. La confidentialité est satisfaite si une ressource n'est accessible que par les agents autorisés. Cette politique de contrôle d'accès est un raffinement d'une politique de bac à sable par une politique spécifique à chaque application ou classe d'agents mobiles. En fonction des agents, l'hôte peut autoriser ou non l'accès à un ensemble précis de fonctionnalités ou de ressources. Le contrôle d'accès permet de combiner les deux premières techniques en offrant aux agents signés plus de fonctionnalités qu'un simple bac à sable sans pour autant accéder à toutes les fonctionnalités. En contrepartie, l'application d'un tel schéma d'accès a un coût puisque la négociation qui en découle est effectuée dynamiquement à l'exécution.

4.3 Une architecture logicielle basée sur les « *spaces* »

De nombreux modèles de conception et d'architectures logicielles ont été développés pour l'exécution d'applications parallèles sur des grilles de calcul (cf. Section 1.1). L'architecture proposée par notre framework *MCA* utilise les « *spaces* », notamment pour les composants *ComputationSpace* et *Workspace* (cf. Figure 4.1). Dans notre cas, ce type de composant facilite le partage de données dans le but de résoudre un cas de calcul. L'utilisation des *spaces* offre le modèle dit « *ferme de travailleurs* » (spécifié à la section 2.2.2.5 par l'équation 2.37). Celui-ci étend le paradigme Maître-Travailleurs, présenté dans la section 1.1, afin de l'adapter à un système distribué dans un environnement hétérogène. Dans cette section, nous commençons par définir la notion de « *space* » (Section 4.3.1) puis nous définissons le modèle « *ferme de travailleurs* » (Section 4.3.2) sur lequel se base l'architecture logicielle proposée par notre framework MCA. Enfin nous présentons les différents mécanismes rendant une application basée sur les *spaces* tolérante aux pannes (Section 4.3.3).

4.3.1 Définition d'un « *space* »

Au début des années 80, le professeur Gelernter mis les premières briques à la construction d'une architecture basée sur un « *space* » lorsqu'il développa le langage de programmation *Linda* [Gel93] dans le but de faciliter le développement d'applications distribuées. *Linda* était composé d'un ensemble réduit d'opérations combiné avec une mémoire globale, le fameux « *tuple-space* », permettant le stockage de « *tuples* ».

Ce type d'architecture fournit un modèle de développement simple qui remplace complètement le paradigme *RPC* (*R*emote *P*rocedure *C*all) [Mic88]. Le nombre d'opérations qu'il propose à travers un « *space* » est infime mais permet à une large gamme d'applications de profiter de ses avantages comme la modularité, l'évolutivité ou encore la simplicité du code source.

Un *space* est une mémoire virtuelle distribuée et partagée. L'interface de programmation destinée à utiliser un *space* doit contenir au minimum les quatre opérations suivantes : *write*, *read*, *take* et *notify*. Les opérations *write* et *read* permettent respectivement de stocker ou de lire des objets (appelés *entrées*) dans un « *space* » alors que l'opération *take* lit puis supprime une *entrée* de cette mémoire. Enfin, l'opération *notify* permet de s'enregistrer pour être notifié de l'activité du *space*.

Si le modèle RPC fait communiquer directement des objets entre eux à travers des appels explicites de méthodes, l'utilisation d'un *space* implique que seul ce dernier communique avec tous les autres participants. Il est possible de schématiser cela par une sorte de communication par tableau noir [10] où les participants n'ont aucune connaissance des autres participants. Dans ce type d'architecture chaque participant est indépendant des autres. Un *space* fournit un ensemble de caractéristiques permettant de mettre en place des systèmes distribués performants. Nous listons par la suite ces différents caractéristiques :

- *Les spaces sont des mémoires partagées* - De nombreux processus distants peuvent interagir simultanément avec un *space*. Un *space* gère lui-même les accès concurrents ce qui permet aux développeurs de se concentrer sur les données et non sur les accès.
- *Les spaces sont des mémoires dites « associatives »*- La recherche d'un objet dans un *space* se fait par association. Cela fournit un moyen très simple pour trouver un objet : il suffit de définir un modèle (*template*) avec les informations connues de l'*entrée* recherchée pour la retrouver dans un *space*.
- *Les spaces sont tolérants aux pannes* - Nous reviendrons sur ce point dans la section 4.3.3.
- *Les spaces permettent d'échanger du code exécutable* - Dans un *space* les *entrées* sont des données passives (impossible de les modifier ou d'invoquer une de leurs méthodes) mais une fois les objets récupérés en local (via les opérations *read* et *take*), il est possible de les modifier ou d'exécuter leurs méthodes.

10. [EHRLR80] enrichit le concept d'échange d'information en élaborant l'idée du tableau noir (*blackboard* en anglais) avec le projet *HERSAY-II*. Le tableau noir est alors défini comme une zone de travail commune, dévolue à la transmission d'information entre les différents agents

JavaSpaces [FHA99] est une spécification qui définit un service fournissant un mécanisme d'échange et de coordination distribuée pour des objets Java. Il reprend les principes du langage de programmation *Linda* vu précédemment. La spécification *JavaSpaces* fait partie de la technologie Jini™ . Le service éponyme propose une interface simple pour mettre en place des *spaces* et une API pour interagir avec eux. *Outrigger* est le nom donné au service par l'implémentation de référence de la spécification *JavaSpaces* et développée par *Sun*. Cette solution est livrée avec le *Jini starter kit* fourni par *Sun*. Il existe d'autres implémentations disponibles comme *Blitz* [Bli], projet *open source*, écrit et maintenu par Dan Creswell. Il paraît difficile de ne pas citer la plate-forme *GigaSpaces* [Gig11] tant elle propose une solution complète et très bien documentée [Gig]. A la différence des deux premières implémentations, celle-ci n'est pas libre mais propose une version allégée et gratuite.

Nous présentons dans la section suivante comment l'utilisation des *spaces* nous permet de définir un modèle d'architecture logicielle pour l'exécution d'une application distribuée autonomique (cf. Section 1.2) comme le propose notre framework *MCA*.

4.3.2 Implantation d'une « ferme de travailleurs »

Le paradigme *Maître-Travailleurs*, présenté dans la section 1.1.3, offre une solution performante pour les problèmes qui ont un ratio calcul-communication important : c'est à dire lorsque le temps de calcul réel des *travailleurs* dépasse de loin le temps pris pour communiquer avec le reste du système. Ce ratio est directement influencé par la taille d'une tâche relativement à la taille du problème complet auquel elle appartient. En effet, il est possible de diviser un calcul en un nombre important de petites tâches dont le temps de calcul est très court, mais les communications nécessaires entre toutes les tâches du calcul sont plus consommatrices de temps que le calcul réel de toutes les tâches. D'un autre coté, le partage d'un calcul en un petit nombre de grandes tâches peut entrainer une sous-utilisation des ressources de calcul avec un nombre de tâches plus petit que le nombre de *travailleurs* disponibles.

La problématique est alors de savoir comment rendre ce modèle capable de s'adapter aux changements (cf. Section 1.2) au cours de l'exécution d'un calcul. En effet, il est intéressant de pouvoir augmenter le nombre de *travailleurs*, pour répondre à une augmentation de la demande de calcul, ou de réduire le nombre de *travailleurs* si ces derniers ne sont pas utilisés. Comment ajouter ou diminuer le nombre de *travailleurs* de manière autonome, sans avoir à arrêter, reconfigurer et redémarrer le système ? Dans un système où la demande de ressources de calcul peut être variable, comment faire pour automatiser le passage à l'échelle ? Comment automatiser la variation de ressources de calcul en fonction de la demande de calcul sans une intervention humaine ? La suite de cette section montre comment l'utilisation des *spaces* répond à ces questions.

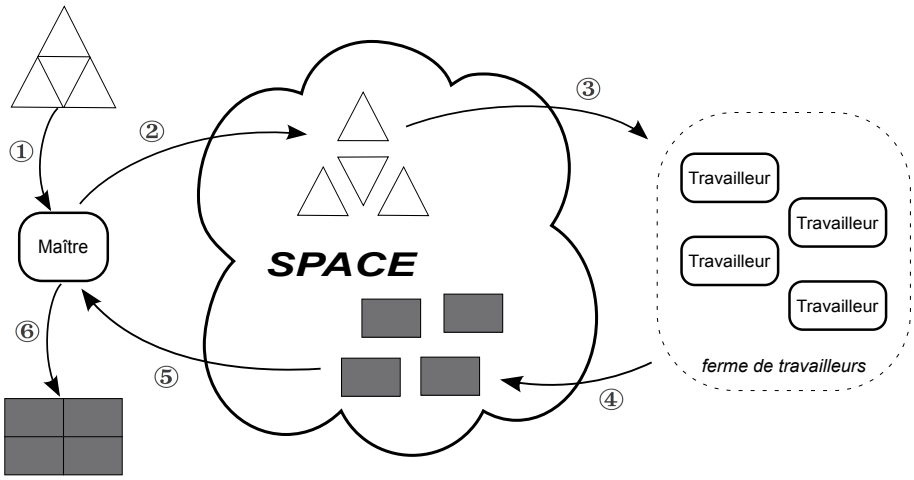


FIGURE 4.11 – Le paradigme « ferme de travailleurs » : ① Le Maître partage le calcul en plusieurs tâches. ② Il dépose les tâches dans le *space* ③ Les travailleurs récupèrent les tâches depuis le *space*. ④ Ils déposent ensuite les résultats dans le *space*. ⑤ Le Maître lit les résultats. ⑥ Pour finir il assemble tous les résultats pour finaliser le calcul.

4.3.2.1 Une équilibrage de charge autonomique

L'utilisation d'un *space* permet de mettre en œuvre le modèle *ferme de travailleurs*. Celui-ci définit un processus *maître* qui écrit des unités de calcul (les tâches) dans un *space*. Celles-ci sont récupérées, exécutées et les résultats retournés dans le *space* par des *travailleurs*, comme nous pouvons le voir dans la figure 4.11.

L'utilisation d'un *space* rend le modèle *Maître-Travailleurs* dynamique et adaptable à la demande de calcul [EG02]. En effet, le processus *maître* ne communique pas directement avec les processus *travailleurs*. Il ne connaît ni le nombre ni la localité des *travailleurs* disponibles car il n'assigne pas les tâches aux travailleurs comme dans le paradigme Maître-Travailleurs classique.

Pour une efficacité optimale, le traitement des tâches doit être équilibré entre les différents *travailleurs* afin que certains ne soient pas surchargés de travail alors que d'autres restent inactifs. Généralement des techniques d'équilibrage de charge sont utilisées pour distribuer la charge de calcul sur les différents *travailleurs* de manière à atteindre ce but. Malheureusement, les grilles de calcul sont le plus souvent composées de machines plus ou moins performantes ce qui rend encore plus difficile un équilibrage de charge efficace.

Les *spaces* apportent une solution simple à ce problème. Chaque *travailleur* obtient une tâche en effectuant une opération *take* sur le *space*. Cette opération facilite l'équilibrage de charge car le flux des tâches vers les *travailleurs* est régulé par les *travailleurs* eux-mêmes. Les *travailleurs* récupèrent les tâches disponibles dans le *space* uniquement quand ils sont prêts. La charge est alors équilibrée par le système lui-même. C'est le *space* qui garantit que chaque tâche n'est prise qu'une seule fois. Ainsi, les *travailleurs* les plus rapides prennent plus de tâches que les *travailleurs* plus lents.

4.3.2.2 Des travailleurs adaptables

Un axe important dans la recherche de l'optimisation des ressources disponibles est le temps d'inactivité des *travailleurs*. Un *travailleur* inactif est une ressource disponible non utilisée. Afin de garantir une activité optimale des *travailleurs*, il est important d'assurer qu'ils soient capables d'exécuter tout type de tâche. Ainsi, tant qu'il y a des tâches à exécuter, un *travailleur* n'est jamais inactif car il est capable d'exécuter toutes les tâches sans exception.

L'utilisation du protocole de découverte offert par Jini™ (cf. Section 4.2.1) permet aux *travailleurs* d'utiliser un *space* sans connaître sa localité sur le réseau. De cette façon, un *travailleur* nouvellement ajouté au système peut trouver le *space*, quel que soit l'emplacement du *travailleur* et du *space* sur le réseau.

L'architecture définie par notre framework *MCA* se base sur le modèle *ferme de travailleurs*. Notre framework propose un environnement d'exécution dans lequel la résolution d'un cas de calcul donne lieu à l'exécution d'un nouveau *space* représenté par le composant *ComputationSpace*. Chaque cas de calcul a un nombre non défini de *travailleurs* qui peut évoluer durant la résolution du cas de calcul. Ces *travailleurs*, représentés par le composant *MCAWorker*, sont tous identiques et sont capables d'exécuter toutes les tâches se trouvant dans un *space* indépendamment des cas de calcul auxquels ils participent. De plus, notre architecture ne définit pas de composant pour représenté un processus *maître* mais donne la possibilité de définir différents types de tâche qui sont exécutées par les *MCAWorker*. Il est donc possible de définir une tâche qui modélise un processus *maître*, celle-ci est traitée par un *MCAWorker* comme toutes les autres tâches.

4.3.3 Tolérance aux pannes dans les « *spaces* »

Nous avons vu précédemment qu'un composant de type *ComputationSpace* est un *space* dédié à la résolution d'un cas de calcul (cf. Section 4.1). Les agents de type *MCAWorker* partagent et modifient les *entrées* contenues dans un *ComputationSpace* afin de résoudre le cas de calcul associé. De ce fait, le *ComputationSpace* est un élément central de l'architecture proposée par notre framework MCA. Celui-ci doit donc être tolérant aux pannes [11], c'est pourquoi nous présentons dans cette section les différents mécanismes offerts par les *spaces* pour répondre à une défaillance des différents *spaces* présents lors de la résolution d'un cas de calcul. Nous abordons les notions de transactions distribuées, de réplication et de persistance de données.

4.3.3.1 Les transactions distribuées

Les transactions distribuées sont présentes dans la plupart des systèmes distribués. Appliquées à une architecture basée sur les *spaces*, comme celle que nous proposons avec le framework *MCA*,

11. Notons que le composant de type *Workspace* est aussi un *space* et jouit donc des mêmes mécanismes de tolérance aux pannes.

les transactions distribuées permettent de réaliser une suite d'opérations avec un *space* de manière transactionnelle. Par exemple, les différentes *entrées* contenues dans un *ComputationSpace* forment un ensemble cohérent dans le but de résoudre un cas de calcul. Si l'ensemble de ces entrées venait à être incohérent, cela pourrait rendre le *ComputationSpace* instable et mettre en péril la résolution du cas de calcul associé.

L'utilisation de transaction lors de l'exécution d'une suite d'opérations sur un *space* fournit alors un moyen de respecter l'intégrité des *entrées* contenues dans le *space*. La technologie *JavaSpaces* utilise le modèle de transaction offert par la technologie Jini™ . Celle-ci propose une solution très simple : toutes les transactions sont supervisées par un (ou plusieurs) service nommé « *Transaction Manager* ». Le *Jini starter kit* fournit une implémentation de ce service, nommée *Mahalo*. Une application qui souhaite exécuter des opérations dans un contexte transactionnel demande au service *Mahalo* de créer une nouvelle transaction. Cette transaction est alors utilisée lors de toutes les opérations. Une transaction peut se terminer de deux manières : par un succès (via la méthode *commit*) et dans ce cas toutes les opérations exécutées sous cette transaction sont effectives, ou par un échec (via l'opération *cancel*) et dans ce cas toues les opérations sont annulées.

Les transactions utilisées dans un environnement *JavaSpaces* observent les propriétés *ACID* (*A*tomicité, *C*ohérence, *I*solation et *D*urabilité) comme toutes transactions, c'est à dire :

- *Atomicité* : toutes les modifications réalisées sous une même transaction sont atomiques : soit toutes les modifications sont effectuées soit aucune d'entre elles. Une modification est l'effet de l'invocation des opérations *read*, *write*, *take* ou *notify*.
- *Cohérence* : toutes les modifications réalisées sous une même transaction ne doivent pas violer la cohérence des *entrées* contenues dans le *space*. Par exemple, l'unicité d'une tâche doit être respectée dans un *ComputationSpace*.
- *Isolation* : de nombreuses transactions peuvent s'exécuter simultanément, il est donc important que chaque transaction travaille dans un mode isolé. Par exemple, deux transactions ne doivent pas se perturber et les modifications faites dans un *space* sous une des transactions ne sont pas visibles par des opérations réalisées au même moment sous une autre transaction.
- *Durabilité* : la fin d'une transaction implique que le *space* se trouve dans un état stable et durable. Soit les modifications sont validées par l'opération *commit*, soit le *space* retourne dans un l'état stable antérieur avec l'opération *cancel* ou par la fin du bail de la transaction. Cette propriété peut résister à une défaillance partielle du système grâce à la possibilité de rendre les services *Jini* persistants (ici les services *Outrigger* et *Mahalo*).

Une transaction permet de regrouper une opération *take* et une opération *write*. Si la transaction échoue, toutes les *entrées* récupérées sous cette transaction (via l'opération *take*) sont remises dans le *space* à la fin de la transaction et toutes les *entrées* écrites (via l'opération *write*) sont supprimées. Le *space* se retrouve comme si les opérations n'avaient jamais été exécutées. Nous revenons sur l'utilisation des transactions lors du traitement d'une tâche par un agent *MCAWorker* (cf. Section 4.4.2.1).

4.3.3.2 La réplication

Le processus de réplication proposé par notre framework MCA permet de dupliquer les *entrées* d'un *space* « *source* » vers un (ou plusieurs) *space* « *cible* ». Ce processus est là pour répondre aux pannes pouvant arrivées durant l'exécution d'un *space*. Dans ce but, il est possible de regrouper plusieurs *spaces* ensemble afin de leur appliquer une politique de réplication commune. L'ensemble de ces *spaces* forme alors un *groupe de réplication*. Il est possible de définir un *ComputationSpace* comme un groupe de réplication pour le rendre tolérant aux pannes. Dans ce cas, un *ComputationSpace* est toujours vu par les agents *MCAWorker* comme un *space* alors qu'il contient en réalité plusieurs instances de *spaces*. Toutes ces instances suivent alors la même politique de réplication.

Une politique de réplication permet d'organiser les différentes instances de *spaces* au sein d'un groupe de réplication et définir un mode de réplication (*synchrone* ou *asynchrone*) pour la réplication de leurs entrées.

Les topologies

L'organisation des différentes instances de *spaces* au sein d'un groupe de réplication est définie par deux types de topologies : « *Actif - Passif* » ou « *Actif - Actif* ».

La topologie « *Actif - Passif* » demande de définir une instance dite « *active* ». Les interactions avec le groupe passent par cette instance *active* et toutes les autres instances dites « *passives* » sont des sauvegardes de l'instance active. Chaque instance passive est la copie exacte de l'instance active. Si l'instance active échoue, une des instances passives est élue par les autres comme la nouvelle instance active.

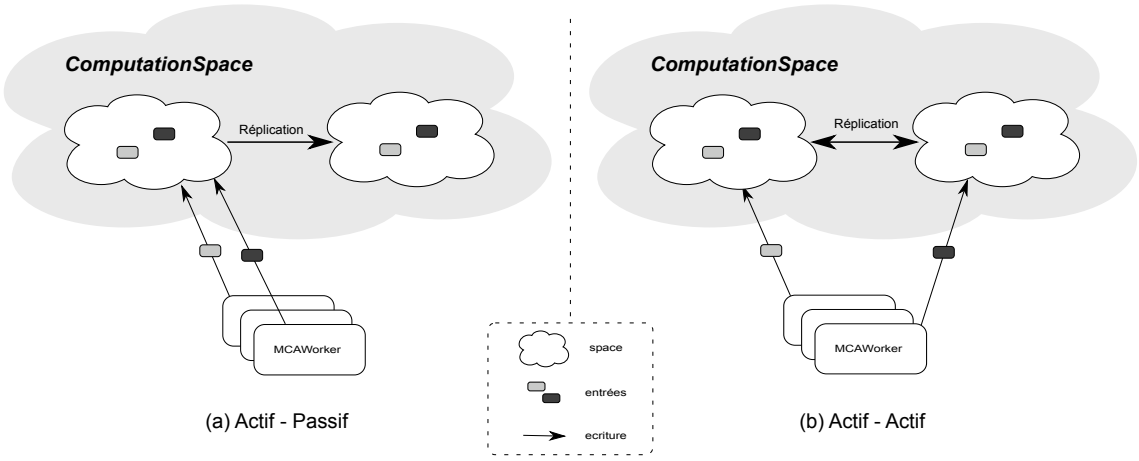


FIGURE 4.12 – Les deux types de topologies possibles au sein d'un *ComputationSpace* défini comme un groupe de réplication.

La topologie « *Actif - Actif* » définit toutes les instances d'un groupe de réplication comme des instances actives et les clients peuvent interagir avec toutes les instances. Chaque instance

réplique ses *entrées* sur toutes les autres instances, s'assurant ainsi que toutes les instances contiennent le même ensemble d'entrées. A son arrivée dans le groupe, une nouvelle instance recherche une instance active pour récupérer ses entrées avant de devenir elle-même active et disponible.

Les modes de réplication

La réplication des *entrées* d'une instance de *space* vers une autre instance à l'intérieur d'un même groupe de réplication peut être définie selon deux modes :

Une *réplication synchrone* assure que le client recevra la réponse de son opération sur l'instance source uniquement lorsque tous les autres instances du groupe de réplication auront exécuté cette opération. Ce mode de réplication est plus adapté à la topologie « *Actif - Passif* » lorsque l'application a besoin de garantir qu'aucune opération exécutée sur l'instance source ne sera perdue et non exécutée sur les autres instances du groupe. En contrepartie, ce type de réplication pénalise le niveau de performance des différentes opérations.

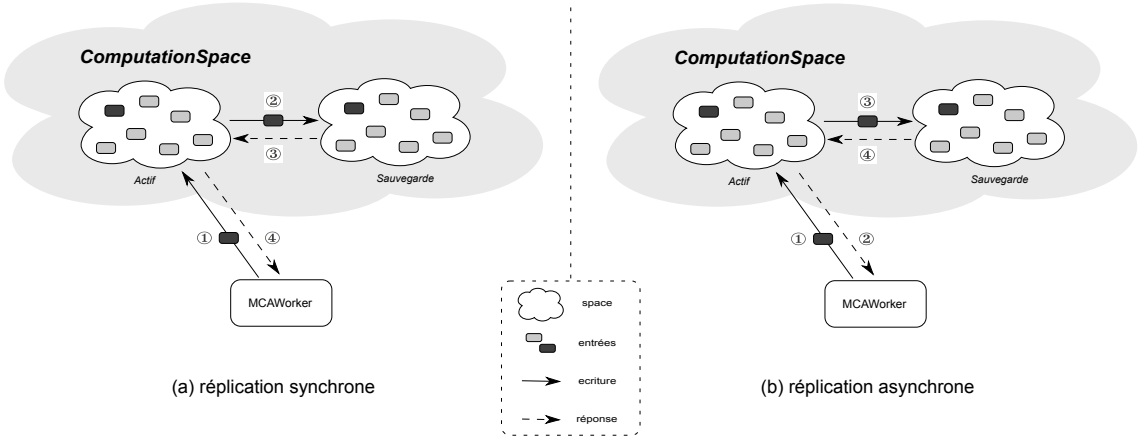


FIGURE 4.13 – Les deux modes de réplication possible au sein d'un *ComputationSpace* défini comme un groupe de réplication. Lors d'une réplication synchrone (a), l'agent *MCAWorker* est bloqué tant que le space initial (*Actif*) n'a pas répliqué la donnée sur le space de sauvegarde (*Sauvegarde*) alors que la réplication asynchrone (b) permet à l'agent *MCAWorker* de continuer son exécution avant la réplication.

Lors d'une *réplication asynchrone*, les opérations sont exécutées sur l'instance source et la réponse est immédiatement renvoyée au client. Toutes les opérations sont accumulées dans l'instance source et sont envoyées de manière asynchrone à l'instance cible, après une période de temps définie ou après un nombre défini d'opérations. Ce type de réplication offre de meilleures performances par rapport à une réplication synchrone mais des *entrées* peuvent être perdues si une panne de l'instance source survient lors de l'envoi des opérations en attente vers l'instance cible. Ce mode de réplication pose un autre problème : la cohérence des *entrées* contenues dans les instances *source* et *cible* n'est pas toujours assurée.

Propriété	Réplication synchrone	Réplication asynchrone
Perte de donnée	Aucune perte de données	Pertes de données possibles lors de la panne d'une instance source avant la réplication des entrées modifiées lors des dernières opérations.
Latence du réseau	Peu tolérant à une haute latence du réseau	Très tolérant à la latence réseau. À utiliser lorsque les instances sont situées dans différents sites géographiques.
Performance	Le client doit attendre la confirmation des instances source et cible. La performance est dépendante des ressources (CPU/Mémoire) des instances source et cible ainsi que de la qualité du réseau entre ces instances.	Le client reçoit une confirmation immédiatement après l'exécution de l'opération sur l'instance source. La performance est uniquement dépendante des ressources (CPU/Mémoire) de l'instance source.
Intégrité des données	Très précise	Peu précise

TABLE 4.2 – Tableau comparatif des deux modes de réplication.

Dans les deux modes de réplication, si une instance cible n'est pas disponible lors d'une opération sur l'instance source, le client reçoit une réponse de l'instance source. L'opération est réalisée sur l'instance cible uniquement lorsque l'instance source rétablit la connexion avec l'instance cible. L'instance source garde la liste de toutes les opérations non-répliquées jusqu'à ce qu'il soit en mesure de rétablir une connexion avec l'instance cible. Le tableau 4.2 présente une comparaison entre la réplication synchrone et asynchrone.

Lorsque la réplication est activée pour un *ComputationSpace*, l'ensemble des *spaces* qui le composent sont organisés en suivant une topologie de type « *Actif - Passif* » en utilisant un mode de réplication asynchrone. Il est tout à fait possible de configurer le *ComputationSpace* pour qu'il utilise une autre topologie et un autre mode de réplication. La section 5.3.2 évalue la performance d'un *ComputationSpace* en fonction des différentes configurations de réplication pour mettre en évidence les avantages et inconvénients de chacune d'entre elles.

4.3.3.3 La persistance de données

Une instance de *space* est une mémoire volatile car les *entrées* qu'elle contient sont perdues à la fin de son exécution. Pour permettre à une instance de *space* de recharger ses *entrées* après un redémarrage, notre framework *MCA* donne la possibilité de sauvegarder ses *entrées* dans une source de données externe. Nous avons choisi la solution offerte par les bases de type *NoSQL* (comprendre « *Not Only SQL* ») [Tiw11]. Cette solution est adaptée au stockage des *entrées* d'un *space* car ce type de base est « *schemaless* », c'est à dire qu'il n'est pas nécessaire de définir un schéma pour stocker des données. *MongoDB* [DC10] propose une base *NoSQL* orientée document

et stocke les données au format *BSON*[12] [Bso] (cf. Figure 4.14). La manipulation des données dans ce format est simple avec l'aide d'*API* fournies pour le langage Java.

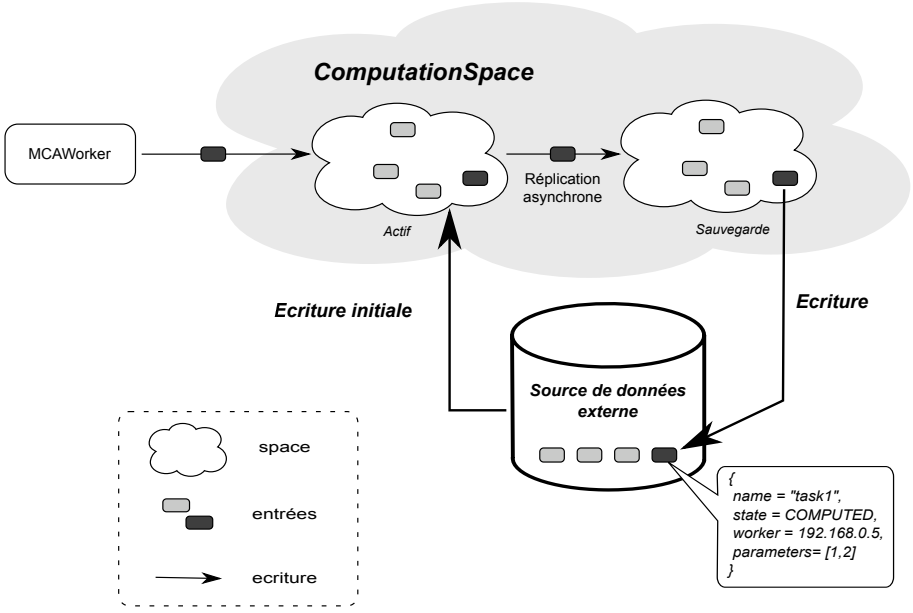


FIGURE 4.14 – La persistance des données d'un *ComputationSpace*.

Rendre une instance de *space* persistante par l'utilisation de cette persistance des données entraine une baisse de performance. Il est possible de réaliser une persistance dite « asynchrone ». Le même mécanisme que celui présenté pour la réplication asynchrone (cf. Figure 4.13) est utilisé. Dans ce cas, l'instance persistante redonne la main au client avant de sauvegarder ses entrées dans la source de données externe. La section 5.3.2 évalue la performance d'un *ComputationSpace* en fonction de l'activation ou non de la persistance de ses données.

4.4 Le framework MCA

Nous présentons dans cette section les différents composants présents au sein de l'architecture définie par le framework MCA. Les exigences imposées à ces composants ont été spécifiées au chapitre 2. Des propriétés temporelles ont été établies au chapitre 3 à propos de la disponibilité des agents *MCAWorker* et de l'évaluation des tâches de calcul (cf. Section 3.3.2). Nous commençons par décrire la plate-forme de calcul (Section 4.4.1) à laquelle se connectent les agents *MCAWorker* (Section 4.4.2) pour participer à la résolution de cas de calcul. Ensuite, nous abordons l'utilisation des agents mobiles de type *ComputeAgent* (Section 4.4.3) qui contiennent le code des différentes tâches d'un cas de calcul. Pour finir, nous présentons la définition et l'utilisation des structures de données distribuées (Section 4.4.4) proposées par notre framework MCA.

12. *BSON* pour *Binary JSON*, *JSON* (pour *JavaScript Object Notation*) étant un format de données dérivé de la notation des objets du langage *JavaScript*. L'avantage de ce format est qu'il est assez abstrait et générique pour pouvoir représenter n'importe quel type de données.

4.4.1 La plate-forme de calcul

Les composants *MCAService* et *ComputationSpace* constituent la plate-forme de calcul qui est l'élément central de l'architecture définie par notre framework MCA.

4.4.1.1 Le service *MCAService*

Le framework *MCA* utilise la technologie Jini™ (cf. Section 4.2.1) pour faire communiquer les différents composants de l'architecture. Il fournit le service *MCAService* (spécifié dans le chapitre 2 par le terme *CaseDirectory* (eq. 2.61)) qui est le point d'entrée des agents *MCAWorker* (cf. Section 4.4.2) pour se connecter à notre plate-forme de calcul. Le service *MCAService*, qui expose l'interface de la figure 4.15, est développé comme un service *Jini* (cf. Section 4.2.1). L'invocation de ses méthodes par les agents *MCAWorker* est réalisée via un *proxy* enregistré dans un annuaire. Les agents *MCAWorker* disposent des différents protocoles de découverte présentés par la figure 4.3 afin de trouver l'annuaire et de récupérer le *proxy* du service *MCAService*. L'utilisation du proxy respecte la politique de sécurité définie dans la section 4.2.1.4.

```
1 public interface MCAService extends Remote {
2   public ComputationCase addCase(String name, String description) throws MCASpaceException;
3   public ComputationCase addCase(String name, String description,
4             RecoveryTaskStrategy strategy) throws MCASpaceException;
5   public ComputationCase getCase(String name) throws MCASpaceException;
6   public Collection<ComputationCase> getCases() throws MCASpaceException;
7   public void removeCase(String name) throws MCASpaceException;
8   public EventRegistration register(MCASpaceEventListener listener, long leaseTime) throws MCASpaceException;
9 }
```

FIGURE 4.15 – Interface du service *MCAService*.

L'invocation d'une des méthodes `addCase` ajoute un cas de calcul à la plate-forme de calcul. Il est possible de définir une stratégie ou de laisser celle définie par défaut (cf. Section 4.4.1.2 avec les *entrées* de type *RecoveryTaskStrategy*). Cet appel entraine la création d'un nouveau *ComputationSpace* dédié spécialement à la résolution d'un cas de calcul. Ce *ComputationSpace* n'est pas accessible directement mais via un agent de type *ComputationCase* qui permet de communiquer avec lui. Ce composant, comparable à un *proxy*, est défini par l'interface `ComputationCase` et propose des méthodes permettant de réaliser des opérations sur le *ComputationSpace* associé. La figure 4.16 présente les échanges entre un agent *MCAWorker* et un *ComputationSpace*. Un agent *MCAWorker* obtient alors un agent *ComputationCase* via la méthode `getCase`. Il peut alors effectuer des opérations sur le *ComputationSpace*. Si la méthode `getCase` ne retourne aucun agent *ComputationCase*, l'agent *MCAWorker* peut demander au service *MCAService* d'être prévenu si un cas de calcul venait à manquer de ressources via la méthode `register`. Dans ce cas, l'agent

MCAWorker s'ajoute au pool d'agents *MCAWorker* disponibles (cf. Figure 4.2).

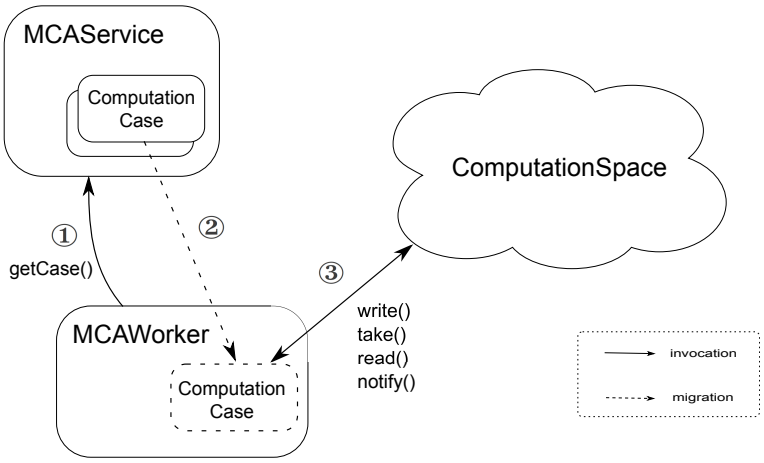


FIGURE 4.16 – Communications entre un agent *MCAWorker* et le service *MCAService*. ① L'agent *MCAWorker* demande au service *MCAService* si un cas de calcul a besoin de ressource. ② Un agent mobile *ComputationCase* migre vers l'agent *MCAWorker*. ③ L'agent *MCAWorker* peut maintenant communiquer avec le *ComputationSpace* via l'agent *ComputationCase*.

4.4.1.2 Le ComputationSpace : un *space* particulier

La particularité d'un *ComputationSpace*, par rapport à un *space* classique (cf. Section 4.3.1), se situe dans les types d'*entrées* qu'il peut recevoir. La figure 4.17 fournit le diagramme de classe des types d'*entrées* pouvant être écrites dans un *ComputationSpace*. Nous remarquons que ces classes ont toutes le stéréotype *« entry »* et leurs attributs sont tous publiques comme l'impose la spécification *JavaSpaces*. Nous listons ces types d'*entrées* :

State - Il n'y a qu'une seule entrée de ce type par *ComputationSpace*. L'état du cas de calcul (cf. Figure 4.18) est une donnée primordiale car chaque agent *MCAWorker* participant au cas de calcul associé au *ComputationSpace* surveille les modifications de cet état pour continuer ou non la résolution du cas de calcul.

Property - La résolution d'un cas de calcul impose, le plus souvent, la déclaration de propriétés (spécifiées à la section 2.2.3.4 par l'équation 2.48). Celles-ci sont des valeurs partagées par toutes les tâches. Ce type d'*entrée* possède deux attributs : une chaîne de caractères `name` pour identifier la propriété et `value` (instance de type `java.io.Serializable`) pour sa valeur.

Task - Définit une tâche du cas de calcul (spécifiée à la section 2.2.3.2 par l'équation 2.42). Une tâche est caractérisée par les propriétés suivantes :

- `name` : ce nom doit être unique pour un cas de calcul et permet d'identifier chaque tâche du cas de calcul. Cette information est utile, par exemple, lors de la recherche d'un résultat.
- `state` : l'état de la tâche (cf. Figure 4.19) permet, par exemple, aux agents *MCAWorker* de connaître les tâches en attente de traitement (IN_PROGRESS) pour les récupérer.

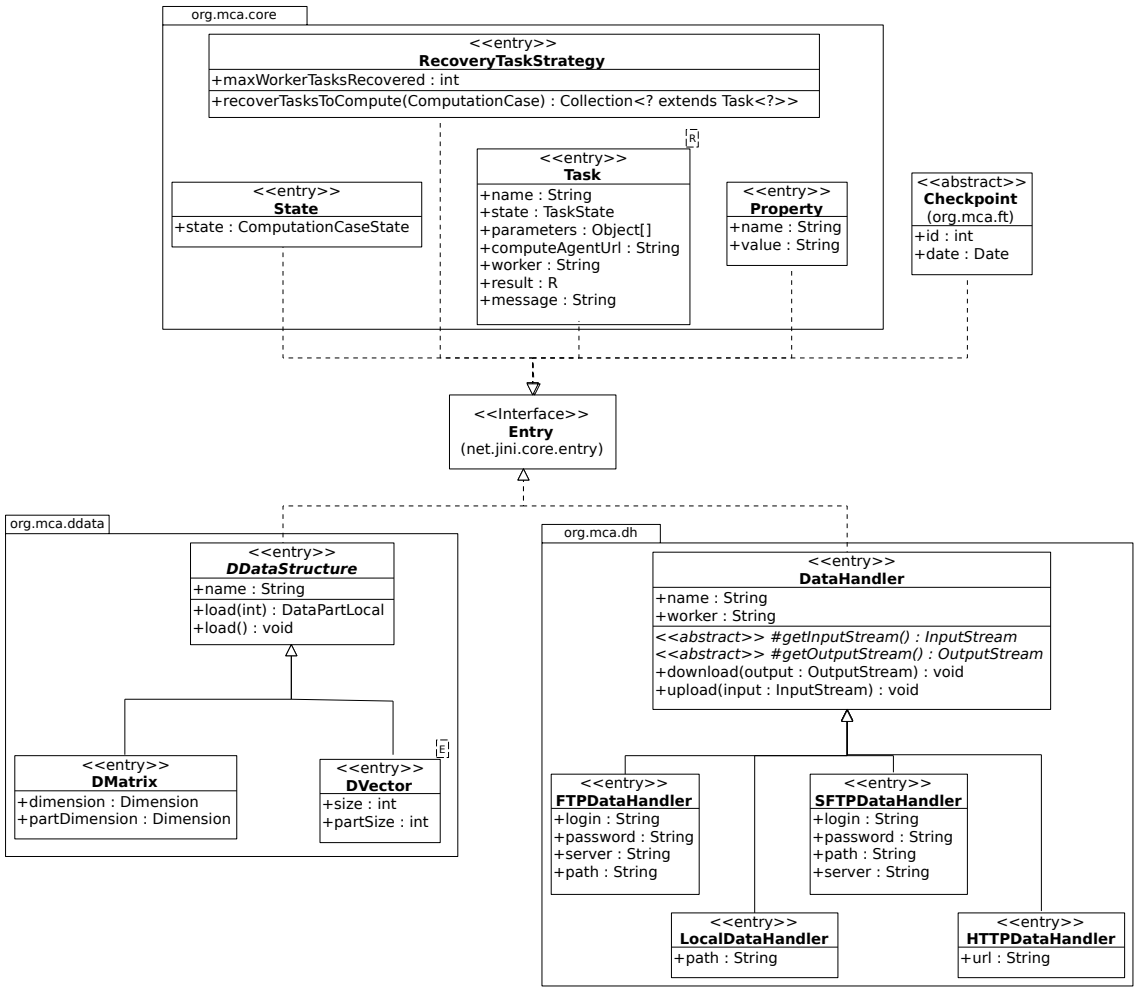


FIGURE 4.17 – Diagramme de classes des types d'*entrées* d'un *ComputationSpace*.

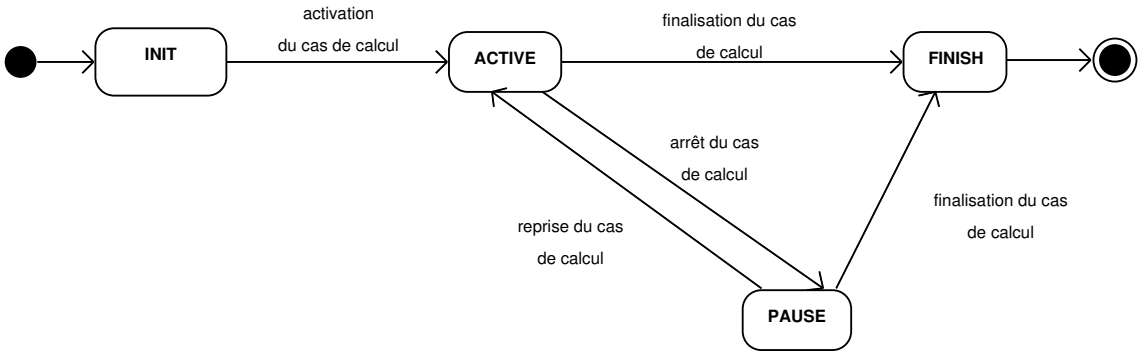


FIGURE 4.18 – Diagramme d'états-transitions d'un cas de calcul.

– `computeAgentUrl` : l'URL de l'agent mobile de type *ComputeAgent* associée à la tâche. Cette information permet à un agent *MCAWorker* de télécharger le code à exécuter correspondant à la tâche (cf. Section 4.4.3).
– `parameters` : à la différence d'une *entrée* de type `Property` qui est une propriété commune à

toutes les tâches du cas de calcul, la liste des paramètres (définie par un tableau) est spécifique à la tâche. Cette liste permet de paramétrer le *ComputeAgent* associé à cette tâche lors de l'exécution du code.

– `result` : le résultat d'une tâche peut être une instance de toute classe implémentant l'interface `java.io.Serializable`. Il peut être, par exemple, de type `DataHandler` si le résultat contient des données de taille importante. Sa valeur peut aussi être *null* si la tâche modifie uniquement des données accessibles par une entrée de type `DataHandler`.
– `message` : ce message décrit l'erreur lorsqu'une tâche se trouve dans l'état FAILURE.
– `worker` : il s'agit de l'adresse de l'agent *MCAWorker* qui traite ou a traité la tâche.

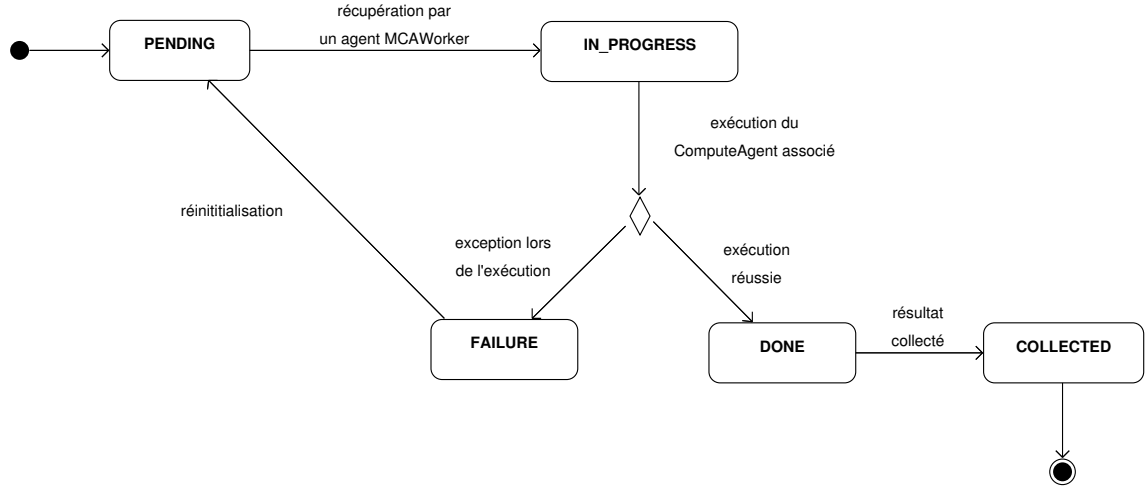


FIGURE 4.19 – Diagramme d'états-transitions d'une *entrée* de type *Task*.

RecoveryTaskStrategy - Il n'y a qu'une seule entrée de ce type par *ComputationSpace*. Celle-ci définit la stratégie utilisée par les agents *MCAWorker* pour récupérer les *entrées Task* à traiter. La stratégie par défaut consiste à récupérer `maxWorkerTasksRecovered` (ayant la valeur 1 par défaut) entrées de type *Task* dont l'état est PENDING. Pour modifier la stratégie par défaut, le développeur doit créer une classe héritant de la classe `RecoveryTaskStrategy` et donner une nouvelle implantation de la méthode `recoverTasksToCompute`. La section 5.3 présente la définition d'un stratégie spécifique à un cas calcul de type *Maître-Travailleurs*.

DataHandler - Les données utilisées pour le traitement des tâches du cas de calcul ne sont donc pas écrites dans le *ComputationSpace* (spécifiée à la section 2.2.3.1 par l'équation 2.39). En effet, ce sont seulement les accès à ces données qui sont présents. La classe abstraite `DataHandler` représente l'accès à des données et propose deux méthodes.

- `public void download(OutputStream output)throws IOException`
- `public void upload(InputStream input)throws IOException`

La méthode `download` permet de télécharger des données dans le flux `output` et la méthode `upload` sauvegarde le flux de données `input` vers la source de données représentée par le *DataHandler*.

Les différentes implantations de cette classe correspondent aux différents types d'accès offerts par

notre framework pour accéder aux données. Ces accès se font par des protocoles de communication (*FTP, SFTP*[13]*, HTTP*) ou directement par le système de fichiers via la classe `LocalDataHandler`. Il est possible de définir d'autres implantations si l'utilisation d'autres protocoles de communication était nécessaire pour l'accès aux données lors de la mise en place d'un cas de calcul.

DDataStructure - Ce type d'*entrée* permet l'utilisation de Structures de Données Distribuées. La mise en place de telles structures est décrite dans la section 4.4.4.

Checkpoint - L'architecture définie par le framework *MCA* est tolérante aux pannes. L'utilisation de points de reprise (*checkpoint*) est une pratique courante dans ce type de système. A partir de ces points de reprise le système doit être capable de reprendre la résolution d'un cas de calcul pour éviter de recommencer le cas de calcul depuis le début. Une entrée de type *Checkpoint* possède au minimum deux attributs : un identifiant et une date. Cette deux informations permettent de choisir un point de reprise lors de la relance d'un cas de calcul. Par défaut, lors de la relance d'un cas de calcul, c'est le point de reprise le plus récent qui est choisi mais il est possible de choisir un point de reprise plus ancien. La définition d'un point de reprise est spécifique à chaque cas de calcul car il contient des informations sur le traitement du cas de calcul lui-même. Pour rendre un cas de calcul tolérant aux pannes, le développeur doit définir un ou plusieurs nouveaux types d'*entrée* héritant de la classe `Checkpoint` spécifiques au cas de calcul. La section 5.2 présente l'utilisation de points de reprise dans un cas de calcul suivant le modèle *SPMD*.

4.4.2 Les agents *MCAWorker*

Les agents de type *MCAWorker* traitent les différentes tâches d'un cas de calcul. Ce type d'agent a été spécifié au chapitre 2 dans la section 2.2.4.2 par le terme $Worker$ (eq. 2.64). L'architecture d'un agent *MCAWorker* s'articule autour de 4 composants : un contexte d'exécution (un *space* de type *WorkSpace*), un moniteur d'activité (composé des processus *MCASpaceListener* et *WorkerMCASpaceEventListener*), un gestionnaire de cas de calcul (processus *CaseStateListener*) et un exécuteur de tâches (processus *TaskExecutor*). Notons que le contexte d'exécution d'un agent *MCAWorker* est actif durant toute l'exécution de l'agent, au contraire des autres composants qui sont eux actifs en fonction de l'état de l'agent. La figure 4.20 présente les différents états d'un agent *MCAWorker* et les différents processus démarrés en conséquence.

Le *contexte d'exécution* d'un agent *MCAWorker* est un *space* défini par le composant *WorkSpace* que nous retrouvons dans le diagramme de la figure 4.1. Ce composant bénéficie des mêmes propriétés de tolérance aux pannes que possède le composant *ComputationSpace*, comme la persistance de données et les transactions (cf. Section 4.3.3). Ce type de *space* se différencie d'un *ComputationSpace* par la définition de sa politique de sécurité. En effet, un *WorkSpace* n'autorise aucune modification provenant d'un processus autre que l'agent *MCAWorker* qui l'a lancé. De plus, le seul accès distant possible en lecture se situe lors de la manipulation de structures de données distribuées (cf. Section 4.4.4). Les types d'*entrée* qui sont acceptés dans un *WorkSpace*

13. *SFTP* pour *SSH File Transfer Protocol*

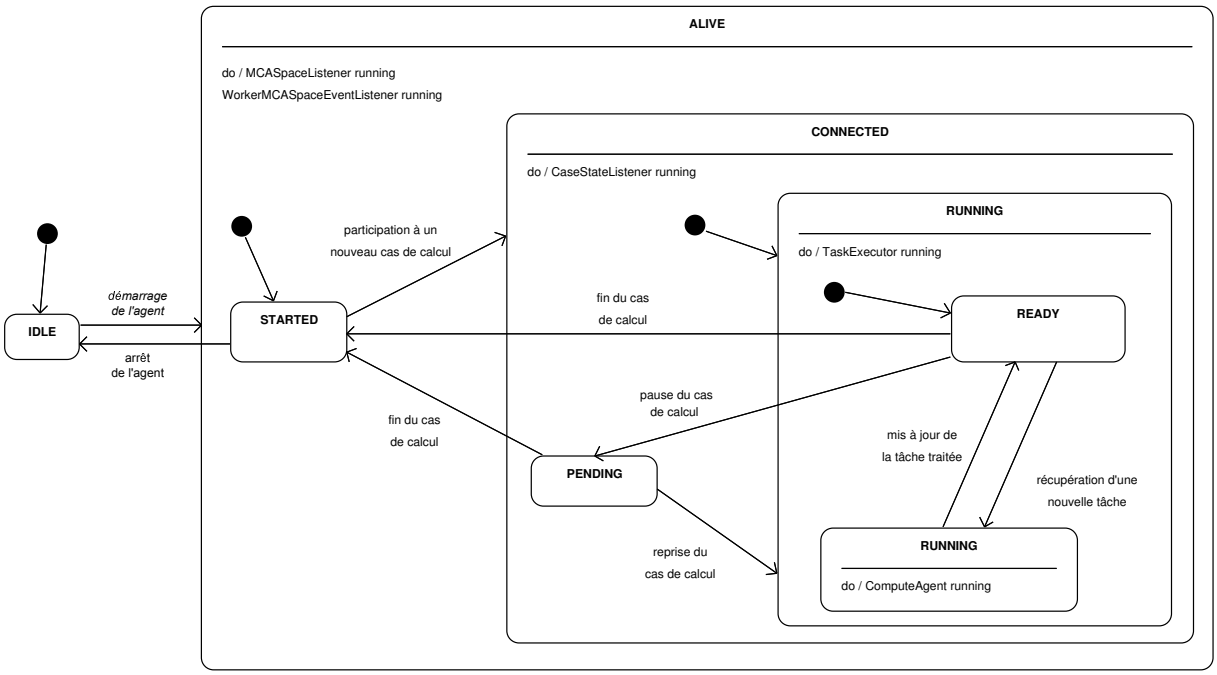


FIGURE 4.20 – Diagramme d'états-transitions d'un agent *MCAWorker*.

sont les types *Task* et *CheckPoint* (cf. Section 4.4.1.2), le type *DData* pour le partage de données avec les autres agents *MCAWorker* (cf. Section 4.4.4) et enfin le type *FTContext* qui contient des informations sur la cas de calcul auquel participe l'agent *MCAWorker*.

Le *moniteur d'activité* surveille l'activité de la plate-forme de calcul. Il se compose de deux processus : le processus *MCASpaceListener* et le processus *WorkerMCASpaceEventListener*. Ces deux processus sont lancés au démarrage de l'agent *MCAWorker*. Le processus *MCASpaceListener* découvre et surveille les différents services *MCAService* disponibles sur le réseau. Le processus *WorkerMCASpaceEventListener* surveille l'activité de chaque service découvert avec l'arrivée ou le départ de cas de calcul. Á l'arrivée d'un nouveau cas de calcul sur la plate-forme, l'agent *MCAWorker* qui ne participe à la résolution d'aucun cas de calcul (état ALIVE de la figure 4.20) est prévenu. Il est alors susceptible de récupérer un proxy de type *ComputationCase* (cf. Figure 4.16). Les informations de ce proxy sont alors sauvegardées dans le *WorkSpace* de l'agent *MCAWorker* (via une *entrée* de type *FTContext*). En cas de panne de l'agent *MCAWorker*, ces informations seront utilisées pour reprendre le cas de calcul auquel il participait lors du redémarrage de l'agent.

Le *gestionnaire de cas de calcul* surveille l'état du cas de calcul (cf. Figure 4.18) auquel l'agent *MCAWorker* participe. Le passage du cas de calcul dans l'état ACTIVE ou PAUSE entraine respectivement l'exécution ou l'arrêt de l'*exécuteur de tâches*. La fin d'un cas de calcul (passage dans l'état FINISH) entraine instantanément l'arrêt de l'exécuteur de tâches, sans attendre le résultat d'une tâche qui serait en cours de traitement. La fin d'un cas de calcul libère l'agent *MCAWorker* pour lui permettre de participer à un autre cas de calcul (état STARTED de la

Figure 4.20).

L'*exécuteur de tâches* récupère les tâches en attente de traitement se trouvant dans le *ComputationSpace*, c'est à dire celles qui sont dans l'état PENDING (cf. Figure 4.19). Pour cela, il utilise la stratégie définie par l'unique *entrée* de type *RecoveryTaskStrategy* qui est disponible dans le *ComputationSpace*. Il exécute les tâches avec la participation du *ComputeAgent* associé à chaque *entrée Task*. Ce processus est démarré par le gestionnaire de cas de calcul lors de la récupération du proxy de type *ComputationCase*. L'exécuteur de tâches est actif tant que le cas de calcul auquel participe l'agent *MCAWorker* est dans l'état ACTIVE. Si ce cas de calcul est mis en pause alors que l'agent est *MCAWorker* en train de traiter une de ses tâches (état RUNNING), ce dernier finit l'exécution de la tâche et sauvegarde le résultat dans son *WorkSpace*. Ainsi lors du redémarrage du cas de calcul, il pourra mettre à jour la tâche traitée sur le *ComputationSpace* du cas de calcul. Nous revenons sur le traitement d'une tâche par un agent *MCAWorker* dans la section suivante.

4.4.2.1 Traitement d'une tâche par un agent *MCAWorker*

Pour traiter une tâche du cas de calcul auquel il participe, un agent *MCAWorker* doit récupérer une *entrée* de type *Task*. Pour cela, il invoque la méthode `getTaskToCompute` (cf. Figure 4.21) sur l'agent *ComputationCase* correspondant (définie par l'interface `ComputationCase`). Cette méthode retire du *ComputationSpace* une *entrée* de type *Task* se trouvant dans l'état PENDING (cf. Figure 4.19). L'agent *MCAWorker* valorise alors les propriétés `state` (PENDING → IN_PROGRESS) et `worker` (avec l'adresse de l'agent *MCAWorker*) de l'*entrée* récupérée et met à jour cette *entrée* dans le *ComputationSpace*.

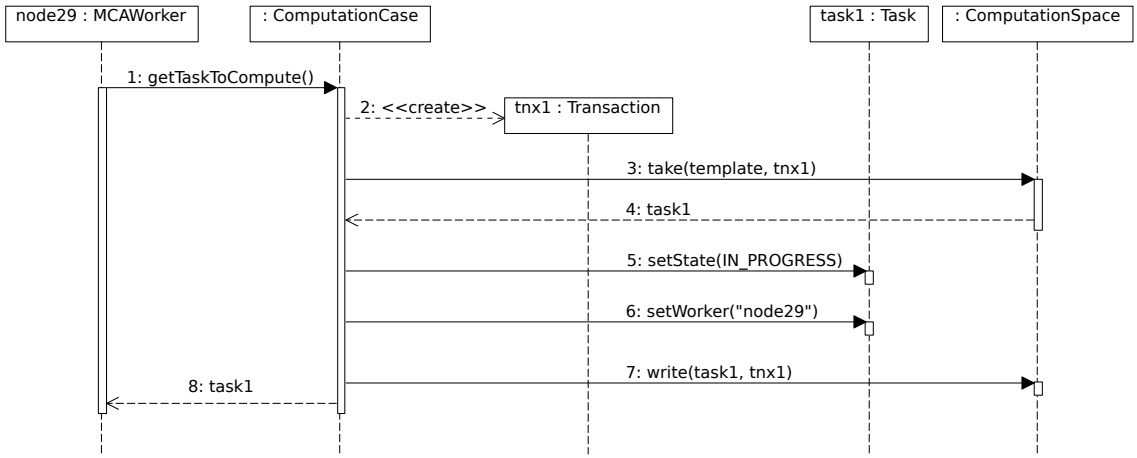
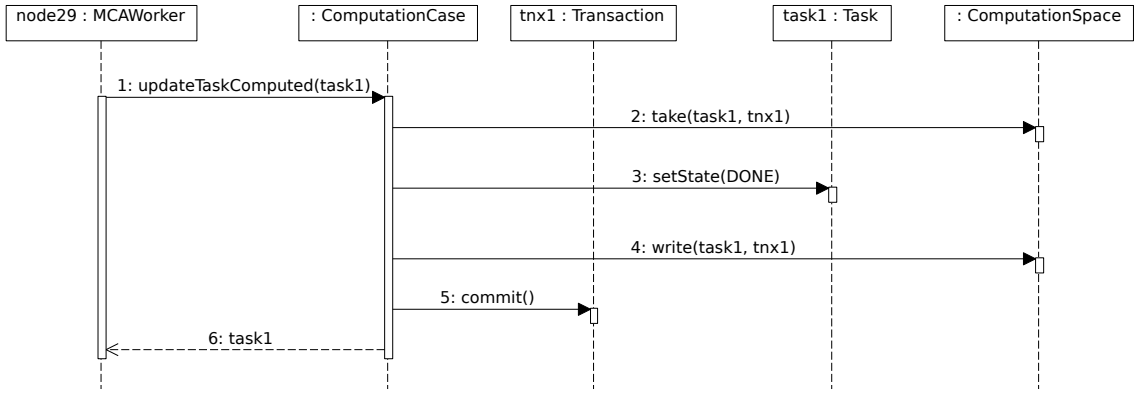


FIGURE 4.21 – Diagramme de séquence de la demande d'une tâche à traiter par un agent *MCAWorker* situé sur l'hôte *node29*. La variable `template` est une instance de la classe `Task` ayant la valeur de son attribut `state` égale à PENDING.

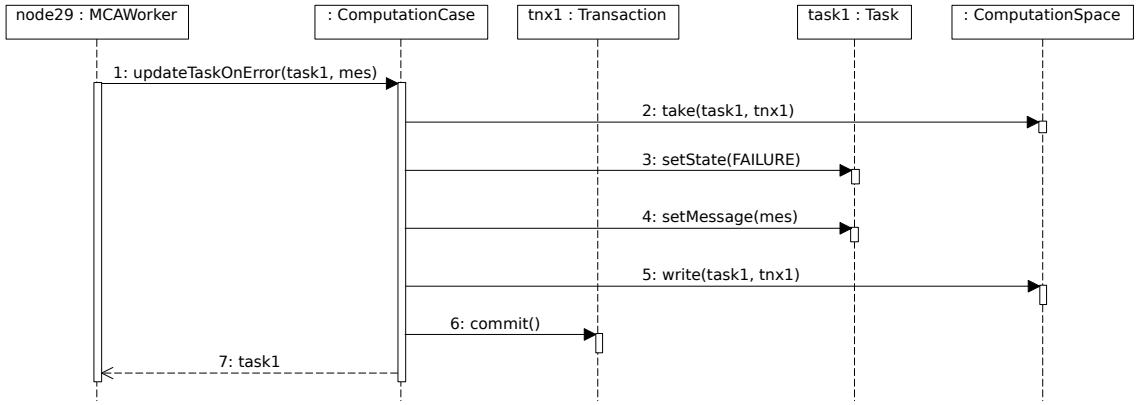
L'*entrée* de type *Task* récupérée est sauvegardée dans le *WorkSpace* de l'agent *MCAWorker*. Le traitement de la tâche correspondante est alors réalisé par le couple *MCAWorker-ComputeAgent*

(cf. Section 4.4.3.1). Deux cas sont alors possibles (cf. Figure 4.22) :

– le code défini par le *ComputeAgent* associé à la tâche s'exécute parfaitement : l'agent *MCAWorker* met à jour l'*entrée* correspondante à la tâche dans le *ComputationSpace* en invoquant la méthode `updateTaskComputed` sur l'agent *ComputationCase* (cf. Figure 4.22a).
– le code défini par le *ComputeAgent* associé à la tâche génère une exception : l'agent *MCAWorker* met à jour l'*entrée* correspondant à la tâche dans le *ComputationSpace* en invoquant la méthode `updateTaskOnError` sur l'agent *ComputationCase* (cf. Figure 4.22b).



(a) exécution réussie



(b) exécution échouée

FIGURE 4.22 – Diagrammes de séquence de la mise à jour d'une tâche par un agent *MCAWorker*.

L'exécution d'une tâche par un agent *MCAWorker* est une opération critique car elle modifie à plusieurs reprises l'*entrée* de type `Task` correspondante dans le *ComputationSpace*. La mise à jour d'une *entrée* est réalisée par deux opérations successives : une opération *take* qui retire l'*entrée* du *space*, et une opération *write* qui écrit l'*entrée* mise à jour dans la *space*.

Une entrée de type *Task* ne doit pas définitivement disparaître du *ComputationSpace* si l'agent *MCAWorker* qui la traite tombe en panne entre les actions *take* et *write*. De plus, elle ne doit pas être bloquée dans l'état IN_PROGRESS (cf. figure 4.19) si l'agent *MCAWorker* qui l'a récupéré pour la traiter tombe en panne. L'utilisation des transactions est donc indispensable pour assurer que cette suite d'opérations « take-write » soit réalisée de manière atomique. Sans

cet aspect transactionnel, plusieurs scénarios pourraient conduire un cas de calcul à être bloqué indéfiniment dans un état non final, c'est à dire dans un état différent de l'état FINISH de la figure 4.18.

4.4.3 L'agent mobile *ComputeAgent*

Un agent de type *ComputeAgent* est un agent mobile. Les contraintes imposées à ce type d'agent sont décrites au chapitre 2 dans la section 2.2.3.2 par l'équation 2.41. Cet agent est capable de migrer vers un agent *MCAWorker* en suivant le modèle défini par une migration réactive (cf. Section 4.2.2.1). Un *ComputeAgent* est donc enregistré dans un annuaire, via le service *Lookup* de Jini™ (cf. Section 4.2.1), pour être mis à disposition des agents *MCAWorker*[14]. Toutes les *entrées Task* d'un *ComputationSpace* sont associées à un type de *ComputeAgent* par son attribut `computeAgentUrl`. Cet attribut définit une URL de la forme `jini://<host>:<port>/<name>` avec `jini://<host>:<port>` pour indiquer l'URL d'un service *Lookup* et `name` pour indiquer le nom donné au *ComputeAgent* lors son enregistrement dans le *Lookup*.

Un *ComputeAgent* contient le code exécuté lors du traitement de la tâche auquel il est associé. Le code d'un *ComputeAgent* est implanté indépendamment de son contexte d'exécution (paramètres spécifiques à la tâche, propriétés du cas de calcul courant...). Deux tâches peuvent très bien être associées au même type de *ComputeAgent*. Par exemple, dans le cas d'un calcul suivant le modèle *SPMD* (cf. Section 5.2) le même code doit être exécuté sur différentes parties d'une structure de données. Dans ce cas, un seul type de *ComputeAgent* est implanté et cet agent est associé à toutes les tâches du cas de calcul. La figure 4.23 présente les différentes classes mises en œuvre lors de la définition d'un type de *ComputeAgent*.

Définir un nouveau type de *ComputeAgent* consiste à implanter une sous-classe de la classe `AbstractComputeAgent` (en implantant la méthode `execute`). Différentes sous-classes sont déjà implantées et fournies par notre framework : les classes `SPMDAgent`, `MasterAgent` et `SkeletonAgent`. Ces trois types de *ComputeAgent* offrent la possibilité de définir des cas de calcul basés sur des paradigmes de calcul parallèle comme *SPMD* (`SMPDAgent`), Maître-Travailleurs (`MasterAgent`) et les squelettes algorithmiques (`SkeletonAgent`). Le chapitre 5 utilise ces trois types de *ComputeAgent* pour évaluer l'architecture *MCA* à travers la résolution de différents cas de calcul.

4.4.3.1 Le couple *MCAWorker-ComputeAgent*

Tous les agents *MCAWorker* sont identiques et cela quelque soit le cas de calcul auquel ils participent. En effet, c'est seulement lors l'exécution d'un *ComputeAgent* qu'un agent *MCAWorker* se différencie des autres agents *MCAWorker*. Un *ComputeAgent* est récupéré grâce à la valeur de l'attribut `computeAgentUrl` défini dans l'*entrée Task* (cf. Section 4.4.1.2) associé à la tâche à traiter par l'agent *MCAWorker*. Après la migration de l'agent mobile vers la machine où se trouve

14. La section 5.1.3.1 présente la configuration et le déploiement d'un *ComputeAgent*.

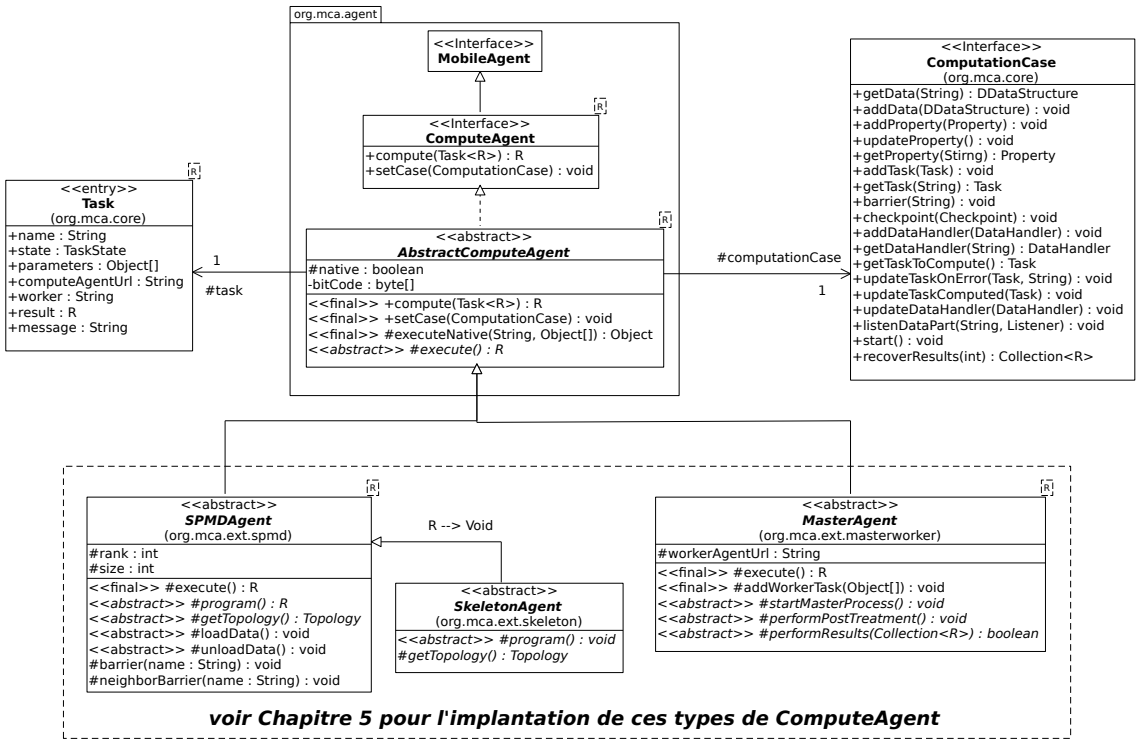


FIGURE 4.23 – Diagramme des classes utilisées pour définir un *ComputeAgent*.

l'agent *MCAWorker*, ce dernier s'assure qu'il peut exécuter le *ComputeAgent* en toute sécurité (cf. section 4.2.2.2).

Pour exécuter le *ComputeAgent*, l'agent *MCAWorker* lui fournit un lien vers le *ComputationSpace* associé au cas de calcul (via l'agent *ComputationCase*) et l'*entrée* de type *Task* correspondant à la tâche qu'il doit traiter. Le *ComputeAgent* exécute alors le code contenu dans la méthode `execute` déclarée dans la classe `AbstractComputeAgent` et implantée dans la classe réelle du *ComputeAgent* récupéré.

4.4.3.2 Exécution de code natif

L'une de propriétés souhaitées pour un outils de simulation numérique est la reprise de code existant (*legacy code*). C'est dans ce sens que le framework *MCA* offre la possibilité de définir un *ComputeAgent* dit « natif ». L'exécution de ce type d'agent ne modifie en rien la relation *MCAWorker-ComputeAgent* vue précédemment. Un agent *MCAWorker* reste identique qu'il exécute un *ComputeAgent* natif ou non. La totalité du code à exécuter doit rester entièrement mobile même si celui-ci est écrit à l'aide d'un langage natif.

Pour répondre à cette contrainte, nous avons choisi d'utiliser la suite d'outils proposée par

LLVM[15]. Elle permet à des fichiers sources écrits à l'aide de langages natifs (*C++, C, Fortran, ADA* ...) d'être compilés dans un format, appelé *bitcode*. Ce format est alors interprété par une machine virtuelle (fournie également par *LLVM*). Lors de la création d'un *ComputeAgent* natif, le code natif à exécuter est compilé en *bitcode* et injecté dans l'agent mobile associé via l'attribut `byteCode` de la classe `AbstractComputeAgent` (cf. Figure 4.23). Cette attribut permet de sauvegarder ce *bitcode* sous la forme d'un tableau d'octets. Ainsi le code natif devient mobile au même titre que le *ComputeAgent* qui le contient. *LLVM* offre la possibilité de manipuler la machine virtuelle qu'il fournit à l'aide d'un ensemble de librairies *C++*. Nous avons ainsi développé une librairie dynamique en *C++* (nommée *libMCA*) qui permet d'exécuter des fonctions définies dans un fichier *bitcode* (cf. Figure 4.24).

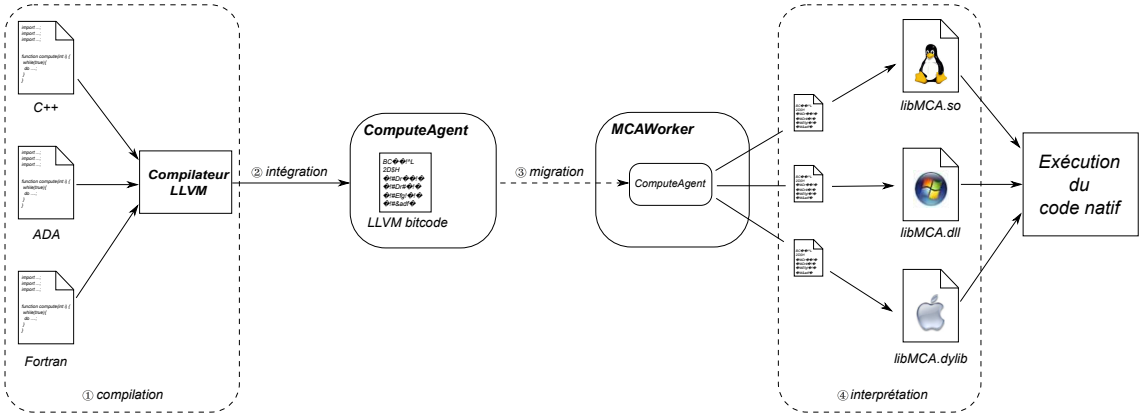


FIGURE 4.24 – Cycle de vie d'un *ComputeAgent* « natif ». ① Le code natif est compilé vers un fichier au format *bitcode*. ② Le *bitcode* est intégré au *ComputeAgent*. ③ Le *ComputeAgent* migre vers l'agent *MCAWorker* voulant exécuter le code. ④ Le *bitcode* contenu par le *ComputeAgent* est interprété grâce à la librairie *libMCA* présente sur la machine sur laquelle s'exécute l'agent *MCAWorker*.

La bibliothèque *libMCA* est installée sur chaque machine où s'exécutent les agents *MCAWorker*. Un *ComputeAgent* « natif » peut invoquer les fonctions proposées par cette librairie et ainsi utiliser les fonctions définies dans le *bitcode* qu'il contient. Nous pouvons citer en particulier les deux fonctions de la librairie *libMCA* suivantes :

- **void** load(**const char**∗ filename);

La première action d'un *ComputeAgent* « natif » est de sauvegarder le *bitcode* qu'il contient dans un fichier dans le système de fichier de la machine sur laquelle il s'exécute. La fonction `load` lui permet alors de charger le fichier sauvegardé en indiquant son emplacement par le paramètre `filename`. Il peut ensuite exécuter le code chargé via la fonction suivante :

- **const char**∗ execute(**const char**∗ **const** function, **const char**∗ **const** parameters[]);

15. *LLVM* (pour *L*ow *L*evel *V*irtual *M*achine) [Lat11] est une boîte à outils pour construire des compilateurs, des éditeurs de liens, des environnements d'exécution, des machines virtuelles et d'autres programmes d'exécution liés à ces outils.

Une fois chargé, le code du fichier bitcode est exécuté via la fonction `execute`. Les paramètres contenus dans le tableau `parameters` sont utilisés pour appeler la fonction `function`.

Un agent de type *ComputeAgent* est écrit en Java comme tous les autres composants de notre architecture. Si celui-ci est « natif », c'est uniquement par le fait qu'il exécute du code écrit dans un langage natif. Ainsi nous avons mis en place une solution afin de pouvoir utiliser notre librairie dynamique écrite en C++ (*libMCA*) à partir du code Java de la classe `AbstractComputeAgent`. Nous avons choisi d'utiliser le framework *JNI*[16] (cf. Figure 4.25) particulièrement adapté à ce type de configuration.

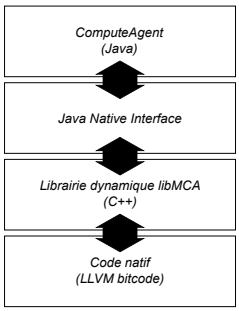


FIGURE 4.25 – Communications par couches lors de l'exécution de code natif.

4.4.4 Les Structures de Données Distribuées

Le dernier type de composant de l'architecture MCA que nous présentons est celui qui permet l'utilisation de Structures de Données Distribuées (*SDD*). Celles-ci profitent des multiples avantages offerts par l'utilisation des *spaces* dans les composants *ComputationSpace* et *WorkSpace* (cf. Figure 4.1).

4.4.4.1 Définition et utilisation de Structures de Données Distribuées

Parmi les différents types d'*entrée* disponibles dans un *ComputationSpace* (cf. Figure 4.17), le type *DDataStructure* permet de définir et d'utiliser des *SDD*. La classe éponyme fait partie d'un ensemble de classes abstraites et d'interfaces (cf. Figure 4.26) fournies par le framework *MCA*. Cette *API* spécifie la structure de classes pour la définition de nouveaux types de *SDD*. Celles-ci permet de partager des données structurées (matrices, vecteurs...) entre plusieurs agents *MCAWorker*. En réalité, ce ne sont pas les agents *MCAWorker* qui manipulent directement ces *SDD* mais les *ComputeAgent* exécutés par les agents *MCAWorker*. Pour simplifier la suite de la lecture, nous utilisons le terme *MCAWorker* pour désigner le couple *MCAWorker-ComputeAgent* qui s'exécute lors du traitement d'une tâche.

16. *JNI* (*J*ava *N*ative *I*nterface) [Lia99]. Celui-ci permet à du code Java d'appeler ou d'être appelé par des applications natives.

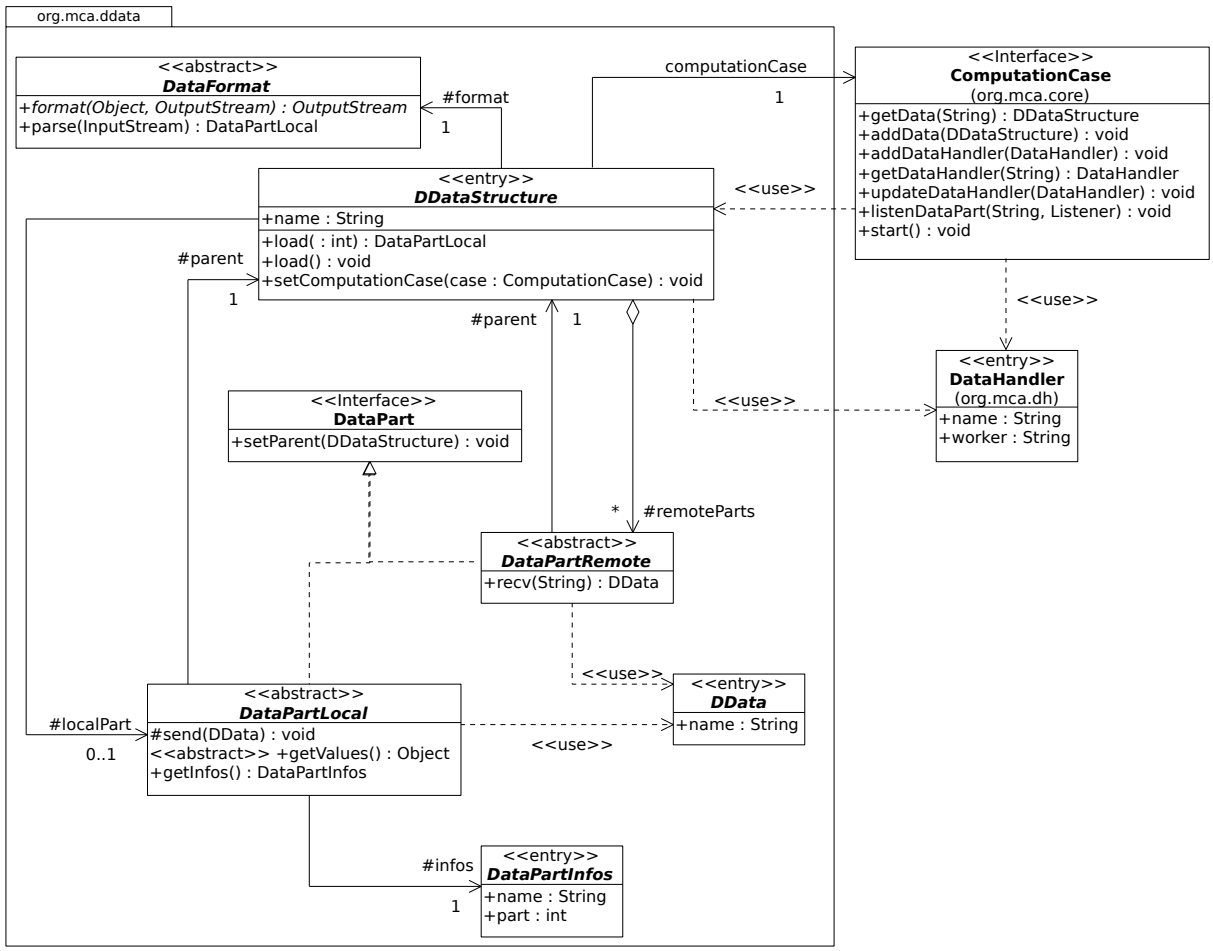


FIGURE 4.26 – Diagramme des classes du package `org.mca.ddata` définissant l'*API* dédiée à la définition de *SDD*.

Pour définir une *SDD* composée de n parties, il est nécessaire d'écrire $n + 1$ *entrées* dans un *ComputationSpace* :

– une *entrée* de type *DDataStructure*, comme par exemple avec le type `DMatrix` (cf. Section 4.4.4.2) ;

– n *entrées* de type *DataHandler*. Celles-ci permettent l'accès aux données de chaque partie de la *SDD*.

L'ensemble des *entrées* d'une même *SDD* respecte une convention de nommage. Par exemple, pour une *SDD* composée de n parties, si l'*entrée* de type *DDataStructure* a la valeur de sa propriété `name` égale à *dds*, alors la valeur de la propriété *name* de chacune des n *entrées* de type *DataHandler* est égale à $dds - i$, avec $i = 1, \ldots, n$.

La classe `DDataStructure` possède l'attribut `format` de type `DataFormat`. Cette attribut indique le format des données accessibles par les entrées de type *DataHandler*. Un instance de la classe `DataFormat` donne à chaque agent *MCAWorker* le moyen de lire, via la méthode `parse`, ou d'écrire, via la méthode `format`), les données d'une partie de la *SDD*. Par cette technique, un agent

MCAWorker peut lire et/ou écrire dans une *SDD* sans connaître à l'avance le format des données qu'elle contient.

Pour accéder aux données d'une *SDD*, un agent *MCAWorker* récupère l'*entrée* de type `DDataStructure` correspondante dans le *ComputationSpace*. Deux scénarios sont alors possibles, chacun correspondant à une méthode définie dans la classe `DDataStructure` :

– `public DataPartLocal load(int i)`
  Cette méthode permet de charger localement la partie i si elle n'a pas été déjà chargée par un autre agent *MCAWorker*. Elle est invoquée si la tâche récupérée par l'agent *MCAWorker* doit modifier les données de la *SDD*. En cas de succès, la méthode télécharge les données de la partie i dans le système de fichier local à la machine où s'exécute l'agent *MCAWorker*. En cas de succès, une *entrée* de type `DataPartInfos` est écrite dans le *WorksSpace* de l'agent *MCAWorker* (cette *entrée* est utile pour la gestion de la tolérance aux pannes dans les *SDD* (cf. Section 4.4.4.3) et retourne une instance de la classe `DataPartLocal`. Celle-ci représente un lien vers le *Workspace* de l'agent *MCAWorker*. La partie i est alors accessible en lecture et en écriture par cet agent *MCAWorker* et les autres parties de la *SDD* ne sont accessibles qu'en lecture via des instances de `DataPartRemote`. Ces instances sont des liens vers le *WorkSpace* de chaque agent *MCAWorker* ayant chargé une autre partie de la *SDD* et les données partagées sont lues via la méthode `recv` de la classe `DataPartRemote`. De la même façon, l'agent *MCAWorker* partage les données de la partie i avec les autres agents *MCAWorker*. Pour cela, il utilise son *WorkSpace* pour y déposer des entrées de type `DData` (cf. Figure 4.27) via la méthode `send` de la classe `DataPartLocal`.
– `public void load()`
  Cette méthode permet l'accès aux données de la *SDD* en lecture. Elle est invoquée si la tâche récupérée par l'agent *MCAWorker* ne doit pas modifier les données de la *SDD*. Dans ce cas, aucune partie n'est chargée localement par l'agent *MCAWorker* et toutes les données de la *SDD* sont accessibles en lecture via des instances de type `DataPartRemote`.

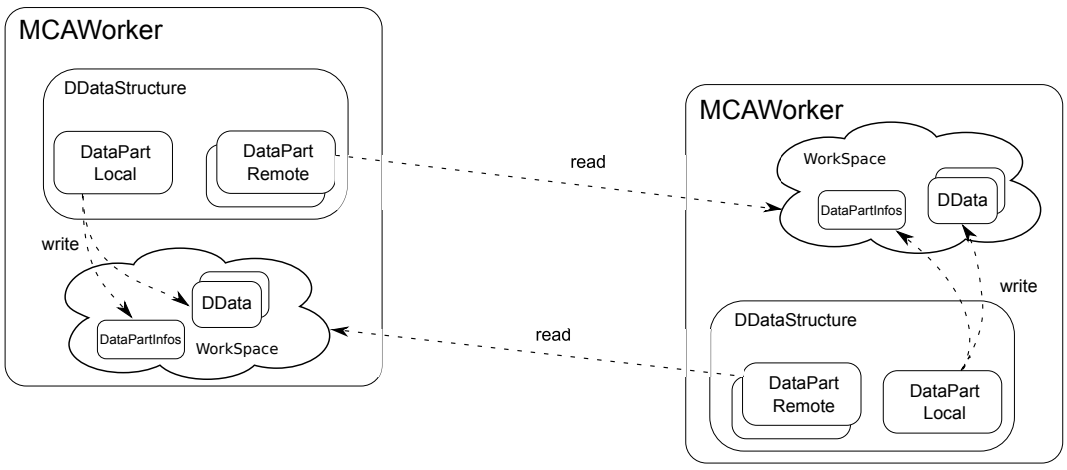


FIGURE 4.27 – Communication entre deux parties d'une Structure de Données Distribuée.

Notre framework *MCA* propose deux implantations de cette spécification sur les Structures de Données Distribuées : une pour la définition de matrices distribuées et une pour la définition de vecteurs distribués. Nous présentons dans la section suivante celle fournie pour utiliser des matrices distribuées.

4.4.4.2 Un exemple de *SDD* : une matrice distribuée

Le framework *MCA* propose l'implantation d'un type de *SDD* permettant de définir une matrice distribuée contenant des nombres réels de type `double`. L'implantation d'un type de *SDD* impose de définir une implantation pour chaque classe abstraite et interface de la figure 4.26. La figure 4.28 présente l'ensemble des classes permettant la définition et l'utilisation de matrices distribuées.

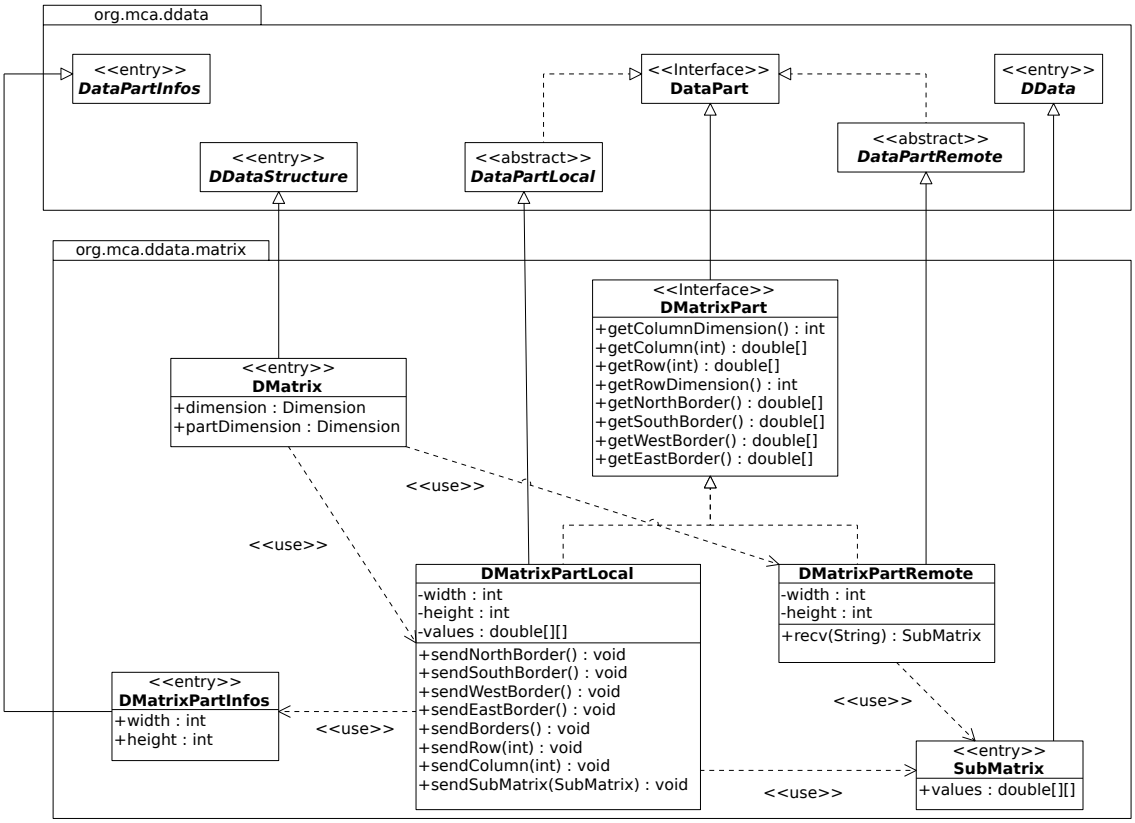


FIGURE 4.28 – Diagramme des classes pour la manipulation d'une matrice distribuée.

Dans le cas d'une matrice distribuée, la classe `DMatrix` définit un nouveau type d'*entrée* (c'est pourquoi ses attributs sont publics) qui représente ce type de *SDD*. Cette *entrée* permet de définir les dimensions [17] de la matrice entière (attribut `dimension`) et les dimensions de chaque partie (attribut `partDimension`). Chaque partie de ce type de *SDD* est accessible via une instances

17. La dimension d'une matrice se définit par le nombre de lignes et de colonnes de la matrice. La classe `Dimension` possède deux attributs : `width` pour le nombre de colonnes et `height` pour le nombre de lignes.

d'une classe implantant l'interface `DMatrixPart`. Cette interface définit des méthodes utiles pour récupérer des valeurs (colonnes, lignes, bordures, sous-matrices...) contenues dans la partie de la matrice correspondante . Deux classes implantent cette interface :

– la classe `DMatrixPartLocal` qui hérite de la classe `DataPartLocal`. Elle représente la partie locale à l'agent *MCAWorker*. Les valeurs de cette partie sont accessibles par l'attribut `values` et les implantations des méthodes définies dans l'interface `DMatrixPart` accèdent directement aux valeurs contenues dans cet attribut. La classe `DMatrixPartLocal` fournit également des méthodes pour partager les données (colonnes, lignes, bordures, sous-matrices...) locales avec les autres agents *MCAWorker*. Chacune de ces méthodes utilise des *entrées* de type `SubMatrix` qu'elles ajoutent au *WorkSpace* pour les mettre à disposition des autres agents *MCAWorker*.

– la classe `DMatrixPartRemote` qui hérite de la classe `DataPartRemote`. Elle représente une partie de la matrice prise en charge par un agent *MCAWorker* distant. Les implantations des méthodes définies dans l'interface `DMatrixPart` utilisent la méthode `recv` pour lire les *entrées* de type `SubMatrix` se trouvant dans le *WorkSpace* de l'agent *MCAWorker* responsable de cette partie distante.

La figure 4.29 modélise l'exemple d'une matrice distribuée sur 4 agents *MCAWorker*. Chaque agent *MCAWorker* a en charge une partie de cette matrice dont il sauvegarde les informations dans son *WorkSpace* à l'aide d'une *entrée* de type `DMatrixPartInfos`. Chaque partie est disponible via des instances de classes implantant l'interface `DMatrixPart`. Si la partie est locale à l'agent *MCAWorker* alors celui-ci utilise une instance de la classe `DMatrixPartLocal` pour accéder aux données de cette partie. Si la partie se trouve sur un autre agent *MCAWorker* alors elle est accessible par une instance de la classe `DMatrixPartRemote`. Chaque agent *MCAWorker* partage avec les autres agents *MCAWorker* les bords de sa partie locale via des *entrées* de type `SubMatrix`.

La section 5.2 utilise ce type de *SDD* et présente la configuration d'une matrice distribuée en vue de la résolution d'un cas de calcul suivant le modèle *SPMD*.

4.4.4.3 La tolérance aux pannes dans les structures de données distribuées

Comme tous les autres composants de l'architecture *MCA*, les *SDD* sont des structures tolérantes aux pannes. La panne ou l'indisponibilité d'un ou plusieurs agents *MCAWorker* en charge des parties d'une *SDD* ne doit pas rendre inaccessible les parties de la *SDD*. Il est important de garantir aux différents agents *MCAWorker* un accès permanent aux données d'une *SDD* partagée entre plusieurs agents *MCAWorker*.

La panne d'un agent *MCAWorker* entraine le passage de la partie de la *SDD* prise en charge par cet agent dans un état temporairement inaccessible. L'*entrée* de type *DataHandler* associée à cette partie est mise à jour (la valeur de son attribut `worker` devient nulle). Les autres agents *MCAWorker* n'ont plus accès à cette partie, elle devient alors libre pour qu'un autre agent *MCAWorker* puisse la prendre en charge. Deux scénarios sont alors possibles pour que les données de cette partie soient de nouveau accessibles :

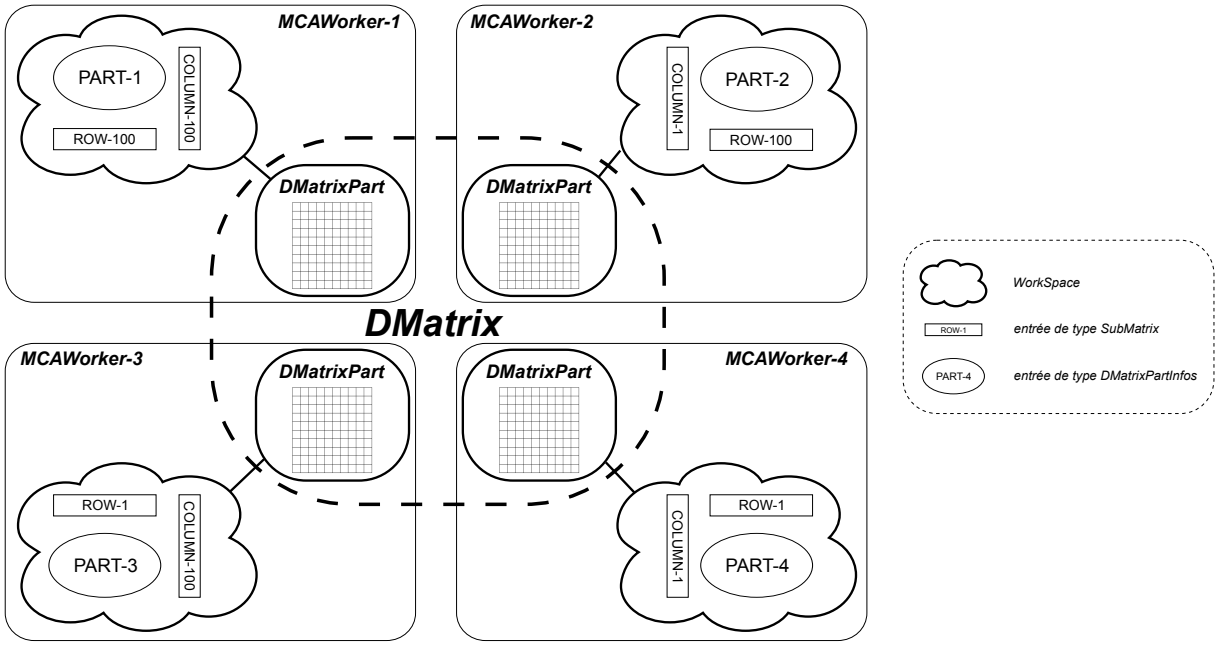


FIGURE 4.29 – Modélisation d'une matrice distribuée sur 4 agents *MCAWorker*.

– Le même agent *MCAWorker* redémarre avant qu'un autre agent *MCAWorker* ait pris en charge la partie devenue disponible. Grâce à l'entrée de type `DataPartInfos`, l'agent *MCAWorker* connaît la partie qu'il avait prise en charge avant son arrêt. Si la partie est disponible et n'a pas été modifiée depuis l'arrêt de l'agent, alors l'agent *MCAWorker* prend de nouveau en charge cette partie sans avoir à télécharger à nouveau les données.
– Un autre agent *MCAWorker* prend en charge la partie devenue libre. Ce cas est décrit dans la figure 4.30. Les instances de type `DataPartRemote` jouent un rôle important car elles surveillent les modifications apportées aux différents `DataHandler`, correspondant à chaque partie de la *SDD*, du *ComputationSpace*. Les agents *MCAWorker* qui ont un accès à cette partie peuvent à nouveau lire les données qu'elle contient.

4.5 Conclusion

Nous avons présenté dans ce chapitre le framework *MCA*. Celui-ci propose des outils pour la résolution de cas de calcul numérique dans un environnement distribué hétérogène. L'implantation de ce framework a été réalisée à partir de la modélisation formelle d'une architecture logicielle décrite dans le chapitre 2.

Notre architecture réalise nos deux objectifs clés : la transparence de panne et la transparence d'échelle (cf. Section 1.1.2). Effectivement, les composants *MCAWorker*, et leur *WorkSpace* associé, sont distribués de manière quelconque sur le réseau, leur nombre varie de manière dynamique sans nuire à la terminaison d'un cas de calcul. L'indisponibilité d'un agent *MCAWorker* est invisible à l'utilisateur. Les données d'un cas de calcul supportent deux autres transparences : la

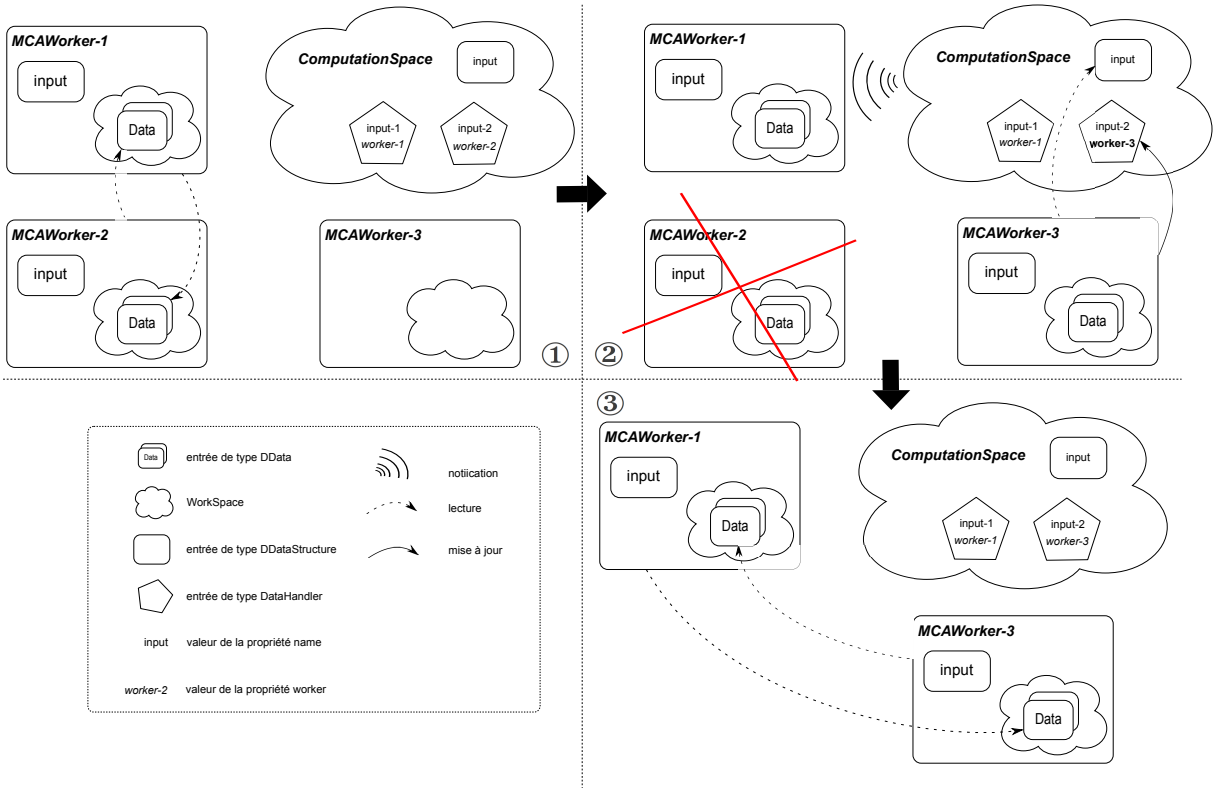


FIGURE 4.30 – Tolérance aux pannes dans les Structures de Données Distribuées. ① Partage d'une structure de données distribuées, nommée *input*, entre deux agents *MCAWorker* (*MCAWorker-1* et *MCAWorker-2*). ② Panne de l'agent *MCAWorker-2*, reprise de la partie disponible (*input-2*) par l'agent *MCAWorker-3* et notification des modifications à l'agent *MCAWorker-1*. ③ L'agent *MCAWorker-1* peut de nouveau lire les données contenues dans la partie *input-2*.

transparence d'accès, qui est mise en œuvre grâce à la notion de *DataPart* et de ses implantations locale et distante, et la transparence de localisation, mise en œuvre afin qu'un agent *MCAWorker* ne connaisse pas l'emplacement où les données sont conservées.

Pour la gestion des parties du code de calcul, nous avons mis en place une transparence de mobilité permettant au couple *MCAWorker-ComputeAgent* d'exécuter un code sans que la migration n'ait été visible des autres *MCAWorker*. Ainsi, la possibilité de déplacer du code natif est une avancée considérable par rapport aux outils de simulation numériques existants et apporte l'adaptation au contexte d'exécution.

Enfin, l'utilisation des *spaces* comme élément central de notre architecture supporte la transparence de réplication de données. Il apporte la persistance de ses données et l'utilisation de transactions distribuées.

Dans le chapitre suivant, nous évaluons le framework *MCA* en implantant différents cas de calcul et en les déployant sur la grille de calcul du LACL.