الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Systèmes Informatiques

Thème :

Proposition d'un modèle basé sur les réseaux de Petri pour modéliser les microservices

Encadré Par :

Dr. Benamira Adel

Présenté par :

Rouzlani lilia

Juin 2022

REMERCIEMENTS

C’est avec un grand plaisir que je consacre ces quelques lignes en signe de gratitude et de reconnaissance à tous ceux qui ont contribué à l’élaboration de ce mémoire. Je tiens à rendre un grand hommage à mon encadreur : Mr. Benamira adel qui a suivi de près ce projet avec le sérieux et la compétence qui le caractérisent. Qu’il trouve ici l’expression de ma profonde gratitude pour l’intérêt qu’il n’a jamais cessé de porter au projet, pour sa disponibilité et pour le soutien qu’il m’a prodigué. Je tiens finalement à remercier ma famille et tous mes amis pour leurs soutiens et leurs encouragements.

Résumé

Ce présent projet s'inscrit dans le cadre de la conception basée sur une architecture dite microserices. Dans ce cadre, l'interdépendances entre les différents microservices est cruciale pour assurer une bonne distribution des charges « load-balancing ». Avoir le calcul d'une telle relation dans un contexte distribué est un vrai chalenge. Afin d'échapper aux détailles de la distribution et ses problèmes, nous avons aborde le problème dans un niveau d'abstraction plus élevé et avec crédibilité. En effet, nous avons fait appel aux modèles de description formelle, précisément, nous avons proposé un modèle, basé sur les réseaux de Petri, qui permet de capter la relation d'interdépendance en appliquer la notion de jetons individuels au model LSGA. Ce dernier est une sous-classe des réseaux de Petri permet la modélisation des systèmes distribués. Finalement, nous avons proposé une méthode d'immigration vers une architecture microservices à partir d'un système monolithique existant modélisé par les réseaux de Petri.

Table des matières

Liste des figures

Chapitre 01
Introduction générale

En 2011, lors d'une conférence pour les architectures logicielles, un des problèmes évoqués est comment assurer la mise à l'échelle (scaling) plus performant et surtout plus flexible pour faire face à l'accroissement du nombre de visites des sites web les plus populaires (Amazon, eBay, Google, Netflix). C'est comme ça le terme microservices a été apparu.

Au lieu de dupliquer en entier des applications monolithiques (voir Figure 1.1), il s'agit désormais de distribuer, et même de dupliquer, certains micro-services uniquement, sur différents serveurs. Tout en gagnant en flexibilité et en agilité, pour effectuer plus rapidement des changements, le plus souvent des améliorations.

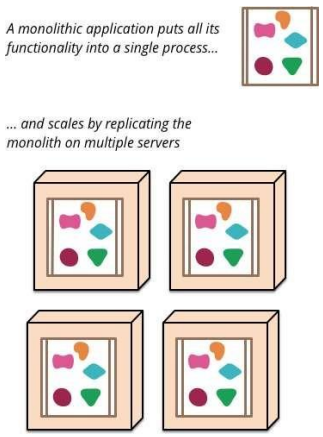


Figure 1.1 : *Duplication des applications monolothiques.*

Une application partitionnée en micro-services est plus robuste qu'une application monolithique. Un micro-service en panne ne bloque pas forcément toute l'application, qui devient alors résiliente. C'est une évolution de la SOA (Service Oriented Architecture) : certains parlent de l'Architecture Micro-Services comme la SOA ***à grains fins***. Elle ne remet pas en cause la SOA, mais propose des solutions techniques plus légères, souvent *Open Source*, comme RESTful notamment.

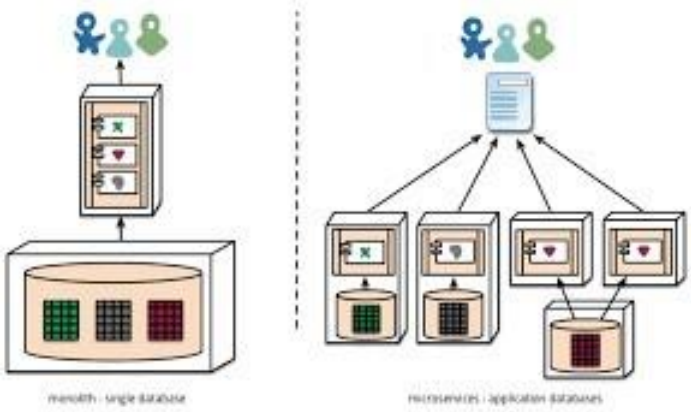


Figure 1.2 : *Monolithique vs Microserices.*

Comme nous l'avons déjà dit, les microservices n'ont évolué que récemment, ce qui rend difficile l'obtention de renseignements complets. En conséquence, une variété de ressources ont été recueillies et recherchées, y compris non seulement des articles publiés, des revues, des livres et des articles, mais aussi des conférences technologiques, des articles Web et des blogues, afin de combiner un certain nombre de questions principales dans le déploiement des microservices et la migration. Ce présent travail cherche à contribuer dans ce champ en proposant un modèle formel pour représenter les microservices en garantissant les résultats suivants :

1. La proposition d'un modèle formel supporte la mise à l'échelle (la scalabilité) d'une architecture microservices.
2. Fournit un calcul formel, basé sur la sémantique de causalité, qui peut être utilisé dans la phase du déploiement, précisément dans la distribution des charges.
3. La proposition d'une méthode de transformation formelle pour avoir une architecture microservices à partir d' un système monolithique existant modélisé par les réseaux de Petri.

Ce présent document est organisé comme suite. Le chapitre 02 évoque un bref aperçu de l'architecture microservices et de son état actuel de développement. La démarche adopte dans notre travail est introduite dans le chapitre 03, c'est un démarche dite formelle, autrement dit, en va utiliser les réseaux de Petri comme modèle de base pour notre proposition et la sémantique de causalité comme sémantique sous-jacente. Le chapitre 04 est notre contribution, dans lequel on va montrer comment en va assurer la scalabilité des microservices, le calcul de l'interaction et comment assurer l'immigration d'une application monolithique vers les microservices. Le document est clôturé par une conclusion générale.

Chapitre 02

Architecture Microservices

Ces dernières années, l'architecture microservices a gagné en popularité, offrant un certain nombre d'avantages par rapport aux architectures traditionnelles. En effet, elle s'attaquant de plus en plus à un certain nombre de problèmes critiques en génie logiciel. Dans ce chapitre, nous commencerons par donner un bref aperçu de l'architecture microservices et de son état actuel de développement. Après la liste de ses avantages, les obstacles à la mise en œuvre anticipés seront discutés, ainsi que des stratégies alternatives pour faire face à ces défis, en utilisant des méthodes empiriques et conceptuelles.

1 Définition

Les microservices sont un concept relativement récent dans le monde des modèles d'architecture logicielle. L'architecture de microservice voit la conception d'une application comme une collection de petits services autonomes. Chaque service s'exécute dans son propre processus distinct. Les services peuvent communiquer en utilisant des protocoles légers (souvent basés sur HTTP) **[DHM11]**. De tels services pourraient être lancés entièrement par eux-mêmes. De plus, la gestion centralisée de ces services est un service distinct. Il peut être écrit dans une variété de langages de programmation, avec ses propres modèles de données, etc. **[DM14]**

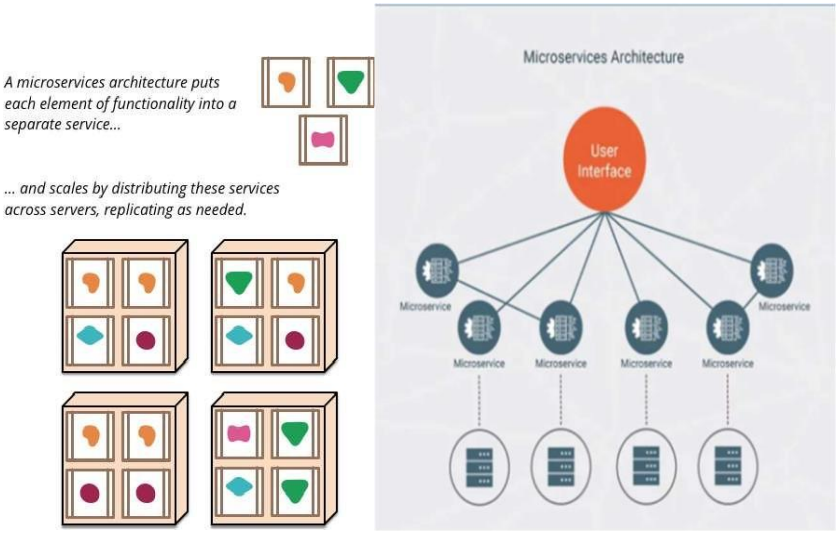


Figure 2.1 : *Architecture de microservices.*

2 Evolution

Comme le montre la figure 2.2, la conception monolithique a évolué vers une SOA. Pour mieux planifier, créer, gérer et entretenir de grands systèmes complexes, SOA a établi un concept de services autonomes avec des limites claires basées sur l'idée de séparation des préoccupations. Comme conséquence, dans telle architecture l'évolutivité dynamique (la possibilité de redéployer des instances de services) et la réutilisabilité des services sont possibles. Cependant, dans la pratique, les services SOA sont généralement lourds et s'appuient sur des intergiciels, des cadres ou des outils avec des bases de données partagées, ce qui contredit dans une certaine mesure le concept de services simples et séparés **[D et al 17]**, **[J et al 18]**. Les microservices ont récemment évolué. Le mot a été inventé à l'origine dans un atelier en 2011.

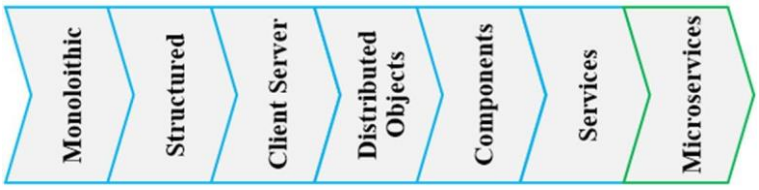


Figure 2.2 : *Évolution de l'architecture des microservices.*

Monolithique

L'architecture monolithique figure 3.2 est considérée comme la méthode traditionnelle de développement d'une application. Une application monolithique est créée comme une unité unique, indivisible et fortement couplée. En règle générale, une telle solution repose sur une architecture à trois niveaux qui comprend

1. Une interface utilisateur côté client,
2. Une application côté serveur,
3. Une base de données.

Elle est unifiée, avec toutes les fonctions gérées et servies à partir d'un seul endroit. Les applications monolithiques manquent généralement de modularité et le couplage entre les différentes fonctions est assez étroit. Par conséquent, les mises à jour logicielles sont extrêmement risquées car toute modification endommagera tous les fichiers liés.

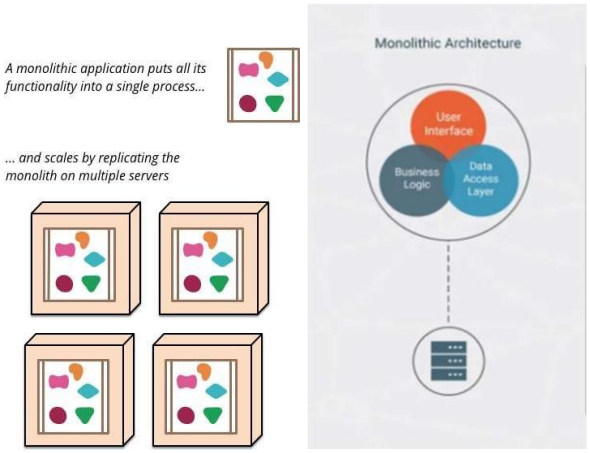


Figure 2.3 : *Architecture monolithique.*

Avantages

- Simple à développer : Au début d'un projet, la conception monolithique est beaucoup plus facile à mettre en œuvre**.**
- C'est simple à mettre sur l'échelle horizontale en exécutant de nombreuses copies derrière un équilibreur de charge. Cependant, si une seule caractéristique monolithique est la cause de la mise à l'échelle, l'ensemble du monolithe doit être redéployé sur un nouvel ordinateur pour répondre aux demandes. Ce n'est pas la meilleure option en ce qui concerne les coûts d'infrastructure.
- Simple à déployer : Un autre avantage associé à la simplicité des programmes monolithiques est la facilité avec laquelle ils peuvent être déployés. En ce qui concerne les applications monolithiques, vous n'aurez pas à faire face à de nombreux déploiements.

Désavantages

- Difficile à comprendre : dans un programme monolithique, un couplage étroit et de nombreuses interconnexions entre les modules peuvent entraîner une grande quantité de code et des interdépendances difficiles à comprendre dans leur ensemble. Les nouveaux développeurs doivent avoir une compréhension approfondie de la façon dont l'ensemble du programme est organisé. Il est possible que cela rende difficile l'intégration de nouveaux talents**.**
- Les changements sont difficiles à mettre en œuvre : il est encore plus difficile de mettre en œuvre des changements dans une application aussi vaste et compliquée avec un couplage aussi étroit. Chaque mise à jour de code a un impact sur l'ensemble du système, elle doit donc être bien coordonnée. En conséquence, le processus de développement global prend beaucoup plus de temps.
- La capacité d'évoluer : Vous ne pouvez pas changer les composants seuls, mais vous pouvez changer l'ensemble de l'application.
- Fiabilité : un bogue dans n'importe quel module (par exemple, une défaillance de la mémoire) peut entraîner le blocage de l'ensemble du processus. De plus, comme toutes les instances du programme sont identiques, ce bug a une influence sur la disponibilité globale de l'application.

SOA

Comme le montre la figure 2.4, SOA est une architecture qui implémente le concept de séparation des préoccupations, dans lequel les capacités ou services métier d'un système logiciel sont fournis par divers modules ou composants appelés services via un protocole de communication et une gestion centralisée appelée Enterprise Service Bus (ESB) **[T18]–[XWQ16]**.

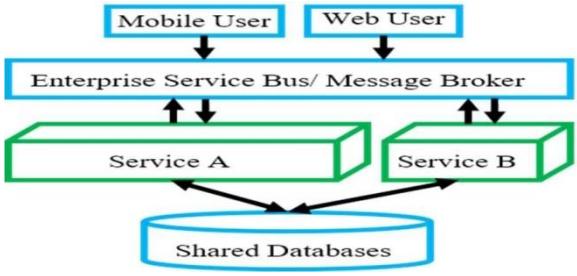


Figure 2.4 : *Architecture orientée services (SOA) : seuls les composants pertinents pour le contexte de l'article sont affichés.*

Microservices

L'architecture des microservices divise les services métier d'un système logiciel en tâches ou fonctionnalités discrètes, généralement une tâche par microservice - **[T18]**. Ces microservices sont de petite taille et indépendants (fonctionnalité et déploiement), et ils communiquent entre eux à l'aide de protocoles de communication légers **[FL18]**, comme illustré à la figure 2.5.

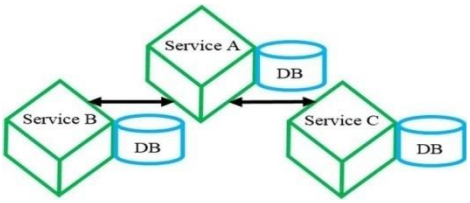


Figure 2.5 : *Microservices.*

SOA Vs Microservices SOA :

Librement couplé, cohésif.

- La gestion /gouvernance Centralisé,
- Déploiement monolithique (tout à la fois),
- Multiple ou comparativement grandes équipes,
- Technologie homogène,
- Stockage de données,
- Stockage de données partagé,
- Communications centralisées (ESB),
- Comparativement moins évolutif.

Microservices : Indépendante.

- La gestion /gouvernance distribuée,
- Déploiement de microservices individuels,
- Petites équipes individuelles,
- Technologie : Hétérogène,
- Stockage de données Par microservice,
- Communications légères entre les microservices,
- Communications légères entre les microservices.

Les conceptions SOA et microservices divisent un système logiciel en modules ou composants appelés services ou microservices, bien qu'ils le fassent de manière distincte. Les microservices, quant à eux, visent à bien faire une chose. Les services sont basés sur la fonctionnalité métier et la portée de l'entreprise. La gestion des communications ou des interactions entre services est centralisée, mais les microservices favorisent une gouvernance dispersée. Du point de vue du déploiement, la SOA reste monolithique **[CDT18]**, tandis que les microservices permettent un déploiement autonome avec une évolutivité dynamique.

3 Avantages envisagés des microservices

- Évolutivité : étant donné que les petits microservices sont simples à créer, à déployer et à exploiter, l'évolutivité d'énormes systèmes complexes peut être considérablement améliorée.
- Livraison continue : il est beaucoup plus facile de déployer de nouveaux microservices et de redéployer des microservices existants avec des fonctionnalités améliorées **[FL18].**
- Développement : les microservices individuels qui implémentent des activités ou une logique unique nécessitent moins de travail de développement et de déploiement.
- Indépendance : les microservices sont considérés comme construits et déployés de manière à pouvoir fonctionner de manière indépendante **[PJ16]**, éliminant les difficultés telles que la succession d'erreurs dans d'autres activités.
- Maintenabilité : En raison de sa petite taille, il est moins sujet aux bogues et est simple à entretenir et à mettre à jour.
- Hétérogénéité technologique : capacité à utiliser n'importe quelle technologie pour créer des microservices discrets **[D et al17]**, **[FL18]**.

4 Challenges

Microservices et interactions client

Le premier problème dans le développement de systèmes logiciels avec une architecture de microservices est de déterminer comment l'interface utilisateur client (UI) interagira avec le système logiciel basé sur les microservices. Si nous analysons les interactions directes entre l'interface utilisateur et les microservices, un trop grand nombre d'appels entre eux sur le réseau public entraînera une congestion **[RS16]**, **[AAE16]**, réduisant les performances, comme le montre la figure 2.6.

De plus, la simplicité du logiciel client sera compromise en raison de la grande base de code utilisée pour implémenter la logique des opérations/transactions de l'utilisateur qui couvrent de nombreux services.

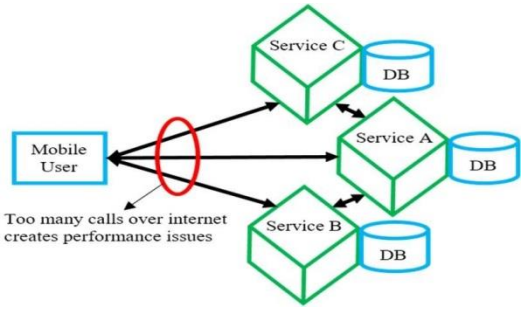


Figure 2.6 : *Interactions directes entre l'utilisateur et les microservices.*

Interactions microservices (communications intra-microservices)

Étant donné que les microservices sont de petits composants autonomes avec des fonctionnalités limitées (généralement une seule tâche), la plupart des opérations/transactions des utilisateurs en temps réel sont liées sur plusieurs microservices, ce qui nécessite des interactions entre eux, comme le montre la figure 2.7, ce qui ne peut être réalisé par la mise en place d'interfaces compatibles et l'utilisation de protocoles de communication légers. La conception d'une architecture de microservices nécessite de déterminer la meilleure approche de communication en fonction des temps de réponse, de la taille des données à transférer et de la compatibilité des services.

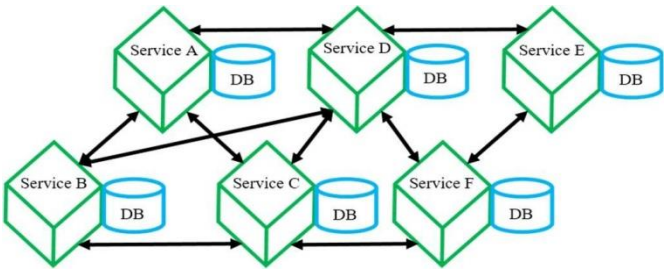


Figure 2.7 : *Communications entre microservices.*

Transactions ACID dans les bases de données de microservices

Les microservices sont destinés à fonctionner comme une entité de travail distincte avec sa propre partie de données dédiées, ce qui pose le problème de garantir les transactions ACID (atomicité, cohérence, isolation et durabilité) sur plusieurs bases de données par microservice. Pour appliquer les transactions ACID, une validation en deux phases (2PC) ou des sagas développées à partir d'un style d'architecture piloté par les événements peuvent être utilisés. Cela implique qu'une entité doit suivre les transactions de longue durée sur des bases de données dispersées par microservices et assurer la réussite de la transaction/l'engagement ou l'annulation de toutes les activités associées montre la figure 2.8.

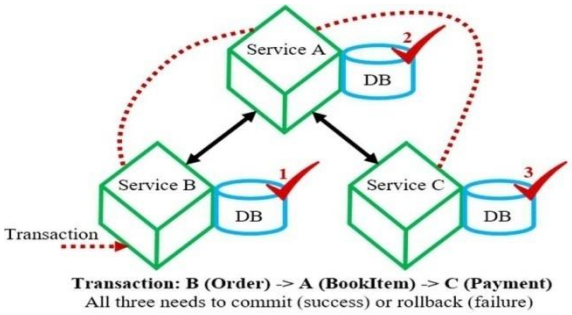


Figure 2.8 : *Transactions dans les bases de données par microservices.*

Découvrir les services

Avec la réduction de la taille des microservices, le nombre de microservices dans les grands systèmes logiciels sophistiqués va augmenter. Il existe une exigence pour un composant ou un module standard et cohérent qui devrait enregistrer tous les microservices avec leurs détails et l'état du service afin de découvrir le bon service **[RS16]**, **[AAE16]** pour une certaine activité. Cela facilitera la découverte et la disponibilité des services pour les microservices et les passerelles API **[XWQ16]**. La figure 2.9 illustre un cas dans lequel le microservice A a été redéployé/mis à niveau avec une nouvelle adresse, provoquant un problème avec sa découverte.

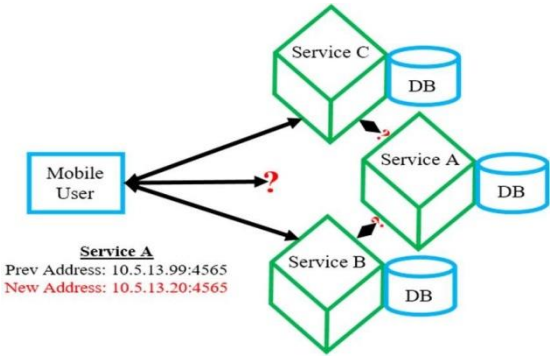


Figure 2.9 : *Problème de découverte de service.*

Tolérance aux erreurs

Étant donné que les opérations en temps réel reposent sur plusieurs microservices fonctionnant ensemble, un grand nombre de microservices peut poser un problème de défaillance partielle. Une seule défaillance d'un microservice peut entraîner une défaillance complète ou un impact d'entraînement sur les activités du système logiciel. Comme décrit dans la section IV-C, non seulement des échecs mais aussi des transactions incohérentes peuvent se produire. Comme illustré à la figure 2.10, il devrait y avoir un mécanisme approprié pour gérer les défaillances partielles dans les opérations afin que l'ensemble du système puisse gérer/récupérer avec élégance **[AAE16]** ou empêcher la défaillance de tout microservice individuel de cascader et d'affecter les opérations et les performances d'autres modules du système. En affichant un avis d'erreur approprié ou en redirigeant vers des instances mises en cache

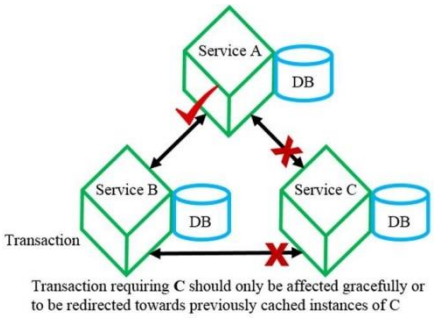


Figure 2.10 : *Tolérance aux pannes.*

Surveillance des performances/analytique

Les systèmes logiciels basés sur les microservices posent une difficulté en termes de résolution des problèmes de performances et d'analyse des microservices **[XWQ16]**. Parce qu'il y a tellement de microservices qui communiquent entre eux pour accomplir certaines tâches. Un décalage de microservice individuel ou une réaction/performance retardée peut dégrader les performances globales du système logiciel. Cela nécessite l'utilisation d'un module de surveillance actif pour suivre les microservices qui rencontrent des problèmes de performances. Cette surveillance active peut indiquer la nécessité de mettre à niveau ou de supprimer tout microservice, ainsi que d'examiner la raison à l'origine d'un problème de performances. Il est également nécessaire de surveiller diverses vulnérabilités liées à la sécurité (ce qui est en soi une difficulté) qui peuvent survenir lors des opérations de microservices.

Logs

La livraison continue est une fonctionnalité de Logs Microservices, ce qui signifie que la mise en œuvre et le développement de microservices sont simples et rapides. Seules certaines informations sur les opérations du système et les commentaires fournis par certains modules de surveillance actifs peuvent déterminer le besoin de nouveaux services ou l'évolutivité et l'équilibrage de charge de tous les services actuels. Le débogage, l'audit et le traitement de divers problèmes de performances et de sécurité nécessitent tous un traçage et une journalisation. Un module de journalisation sera nécessaire pour accomplir cette tâche, qui peut être central ou distribué dans plusieurs composants en fonction des demandes de l'entreprise et des exigences de fonctionnement des microservices.

Sécurité

La sécurité est une grande difficulté **[AAE16]** dans tout système logiciel, mais avec autant de microservices, limiter l'accès non autorisé aux processus métier est une tâche sérieuse à réaliser à de nombreuses couches du système (applications, plates-formes de déploiement, etc.). Grâce à une méthode et à un module d'application, chaque microservice doit vérifier et confirmer la validité des clients/microservices accédant à ses processus métier. Pour assurer la sécurité, un module identifiant les utilisateurs autorisés et appliquant des sessions sécurisées entre les utilisateurs autorisés et les microservices est requis.

Intégration et déploiement

Le déploiement d'un grand système compliqué est toujours une tâche, mais dans le cas d'une architecture de microservices, l'intégration entre les microservices et le déploiement de microservices est un défi clé qui doit être relevé dès le départ. Pour une intégration réussie de nombreux microservices pour mener des activités commerciales, il est nécessaire de fournir une stratégie et des interfaces de communication standard/appropriées, comme indiqué dans la section IV-B. Afin de déployer des microservices dans un contexte cloud (distribué), nous devons les créer de manière à pouvoir les déployer en tant qu'instances distinctes dans un environnement cloud (distribué) à l'aide de VM ou de l'idée de conteneurisation **[J et al 18]**, **[BHJ15]**.

5 Exemple d'une conception microservices

Passerelle API

En commençant par le niveau abstrait de l'utilisateur et des microservices, le premier problème consiste à décider comment ils communiqueront entre eux, comme indiqué dans la section IV-A. Une passerelle API sera ajoutée pour servir de point d'interaction entre l'interface utilisateur client et les microservices **[R18]**, **[RS1618]**. Les API Gates peuvent être divisées en sous-passerelles pour différents types d'appareils (passerelle pour les clients Android, par exemple) ou de services (passerelle pour les requêtes ftp), toutes basées sur la philosophie des microservices consistant à éviter la congestion et les points de défaillance uniques. Les passerelles doivent également avoir un module d'équilibrage de charge pour gérer dynamiquement différentes instances de différentes passerelles en fonction du nombre de demandes des utilisateurs. Comme illustré dans API Gateway Figure2.11, un module de cache supplémentaire peut être ajouté pour améliorer les performances.

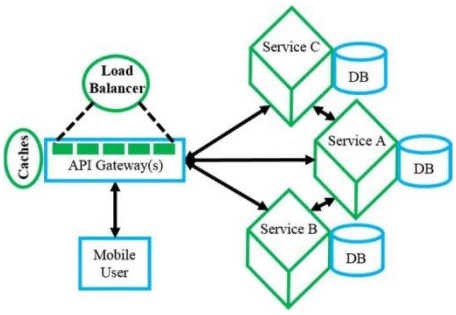


Figure 2.11 *: Passerelles API.*

Registre des services

La prochaine étape après la passerelle API consiste à sélectionner un microservice approprié auquel transmettre la demande de l'utilisateur. L'enregistrement d'adresses/d'enregistrements avec API Gateway ne sera pas un problème pour un petit nombre de microservices, mais à mesure que le système se développera avec des milliers de microservices, nous aura besoin d'un module séparé pour garder une trace de chacun d'eux, y compris les microservices actifs, ceux qui sont temporairement inactifs en raison d'un problème, les détails de version, les versions précédentes mises en cache et les adresses et interfaces de ces microservices. Comme le montre la figure 6.2, le registre des services fonctionnera comme un système d'enregistrement de microservices auquel les passerelles API et les microservices peuvent faire référence pour la découverte de services **[BHJ15]**. Cela servira de référentiel pour toutes les métadonnées des microservices.

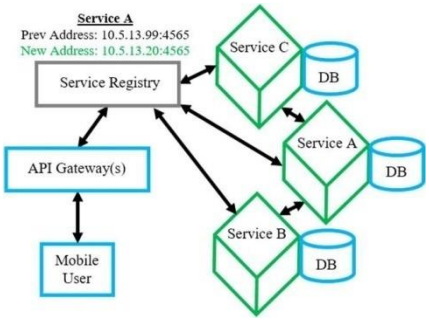


Figure 2.12 : *Registre des services.*

Authentification et sécurité

Pour créer une sécurité indispensable et accorder uniquement aux clients autorisés l'accès aux services commerciaux couvrant un grand nombre de microservices. Pour l'autorisation et l'authentification, cela peut impliquer la mise en œuvre de chiffrements de messages, de jetons d'accès et de cookies pour diverses sessions ou transactions utilisateur. D'autres logiques de sécurité pour bloquer les transactions ou les clients douteux peuvent également être mises en œuvre. Les authentifications client et système, les authentifications de microservices et les authentifications tierces sont toutes couvertes**.**

Surveillance et journalisation

Pour suivre l'état des microservices ainsi que les performances globales du système logiciel. Une vérification en direct des microservices et des activités en cours fait partie de la surveillance. Surveillance de l'état des microservices, des transactions, de la charge sur les microservices individuels et de tous les autres composants du système. Il peut également inclure le module de disjoncteur **[N15]**, qui gérera avec élégance les microservices défaillants, non réactifs ou en retard pendant les opérations actives. Le module de surveillance peut comprendre un module d'analyse qui affiche des informations de microservice et d'exécution d'opération. Ces analyses, qui comprennent à la fois les transactions réussies et non réussies, les sessions utilisateur et les données liées au suivi des divers échecs / échecs partiels dus à certaines erreurs, ainsi que leurs détails associés, doivent être conservées pour une analyse détaillée, ce qui permettra la maintenance et l'évolution. Décisions à prendre. La journalisation centrale et la journalisation distribuée ont leur propre ensemble de problèmes et peuvent être implémentées en fonction des exigences.

SAGA

Pour contrôler l'activité de restauration des transactions et la cohérence de la base de données entre les microservices. SAGA **[R18]** peut être considéré comme un système à sécurité intégrée. Comme le montre la figure 6.3, il garde essentiellement une trace de la séquence de toutes les transactions / activités locales à effectuer par divers microservices, ainsi que de leur état pour terminer une transaction couvrant de nombreux microservices. Lorsqu'une activité est terminée avec succès, le microservice envoie un message ou publie un événement, qui lance l'activité suivante dans la séquence. SAGA garantit la restauration de l'activité partielle des transactions en cours en cas de défaillance d'un microservice.

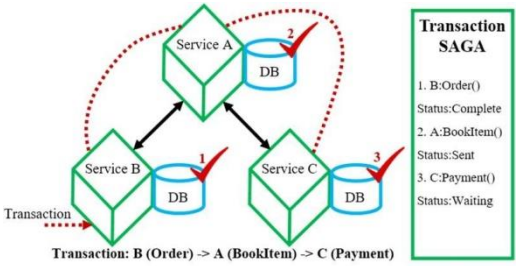


Figure 2.13 : *Concept SAGA.*

Microservices

Les microservices sont les entités qui effectuent le travail de base ou fournissent des fonctionnalités fondamentales. Les microservices doivent implémenter un seul élément de logique ou effectuer une seule opération en utilisant quelques centaines de lignes de code. La difficulté d'atteindre la modularisation correcte **[DS14]**, **[J et al 18]** tout en gardant à l'esprit le principe d'exécution efficace d'une tâche continue, de sorte que les communications inutiles entre les microservices peuvent être minimisées. Les microservices doivent avoir leurs propres bases de données/stockages de données pour profiter de l'hétérogénéité technologique, car les microservices peuvent utiliser plusieurs types de bases de données en fonction de leurs besoins (par exemple, NoSql).

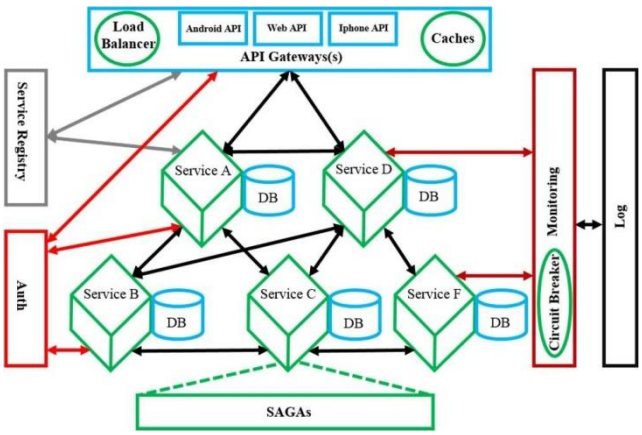


Figure 2.14 : *Conception conceptuelle proposée.*

6 Conclusion

L'architecture de microservices est un concept relativement nouveau qui est rapidement adopté par l'industrie. En fait, il présente des avantages par rapport aux conceptions précédentes allant du monolithique au SOA. Malgré sa popularité croissante dans une variété de domaines logiciels, l'industrie et la littérature sont divisées sur les concepts et la conception de l'architecture des microservices.

Chapitre 03

Modélisation formelle des systèmes concurrents

Durant la dernière décennie, les méthodes formelles sont devenues une activité indispensable intégrée au processus de conception des systèmes complexes à caractère critique, telles que les protocoles de télécommunication, les systèmes répartis et les architectures multiprocesseurs.

En effet, la complexité de ces systèmes rendant leur analyse classique (prototypage et test) extrêmement ardue, leur fiabilité ne saurait être garantie autrement qu'en employant des méthodes de spécification et vérification formelle, assistées par des outils informatiques performants.

Une approche de vérification offrant un bon compromis coût performances est la vérification basée sur les modèles (model-checking), connue aussi sous le nom de la vérification énumérative.

Cette approche consiste à traduire le système, préalablement d´écrite dans un langage appropriée « Description du système », vers un modèle « sémantique », généralement un graphe d'états, sur lequel les propriétés de correction attendues « Spécification » sont vérifiées au moyen d'algorithmes spécifiques (Figure 3.1).

Bien que limitée aux applications ayant un nombre fini d'états, la présente approche est particulièrement utile dans les premières phases du processus de conception, permettant une détection rapide et économique des erreurs.

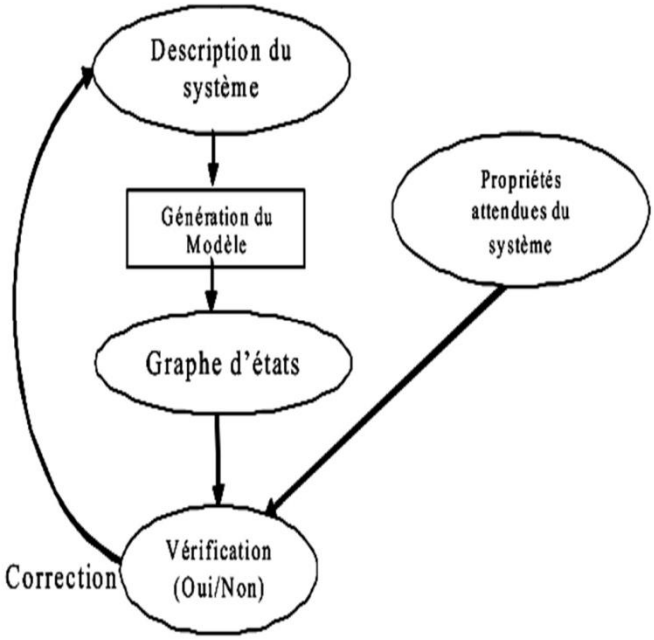


Figure 3.1 : *Approche Formelle.*

1 Fondement de la vérification formelle

La vérification formelle fondée précisément sur les quatre éléments suivants :

- Un langage de description formelle du système, c'est un formalisme du comportement de systèmes de haut niveau, il possède à la fois une syntaxe et une sémantique bien définie (dit formel). Parmi ces langages, nous pouvons citer les réseaux de Pétri, les algèbres de processus (ACP **[BK85]**, CCS **[Mil80] [Mil83] [Mil89]**, CSP **[Hoa85]**) et les techniques de description formelle (ESTELLE **[ISO88]**, LOTOS **[BB87][ISO88]**, SDL**[CCI88]**).
- Un modèle sémantique du parallélisme (modèle de bas niveau), comme son nom l'indique, ce modèle est utilisé pour exprimer la sémantique du parallélisme des langages de description formelle. Nous distinguons deux grandes familles, les modèles d'entrelacement(Arbres de synchronisation**[Mil80]**, STE **[Arn92]**,...) et les modèles de non entrelacement (dit vrai parallélisme) (RDP **[Rei85a]**, SEP **[NPW81]**, STA **[Bed87] [Shi85a] [Shi85b]**,. . .).Leur principale déférence réside dans l'adoption ou non de l'hypothèse de l'atomicité temporelle et structurelle des actions (les actions sont de durée nulle et indivisibles).Les modèles de vrai parallélisme nous permettent d'éviter les difficultés liées aux modèles entrelacés d'une part, et d'une autre part nous permettent d'introduire une méthode de conception formelle par raffinement successif (remplacer une action par un processus).
- Un langage de spécification, c'est un formalisme dédié à la description de propriétés attendues du système. Nous trouvons dans la littérature deux types de langages de spécification : spécification logique et spécification comportementale. La spécification logique (HML **[HM80]**, CTL **[CE81] [CES83]**, CTL**[EH86]**, . . .) est généralement fondée sur des formules d'une logique temporelle modale qui sont interprétées sur le modèle sémantique sous-jacent. Cependant les propriétés du deuxième type sont exprimées dans le même modèle sémantique de celui de la description du système.
- La vérification, c'est une relation de satisfaction qui définit la comparaison entre la description du système et sa spécification. Selon le formalisme de la spécification employé, il existe généralement deux types de vérification ; le premier type consiste de vérifier les formules logiques sur le graphe d'états du système (dit vérification logique) et l'autre type est une comparaison entre les modèles sémantiques s'effectue au moyen d'une relation d'équivalence ou de préordre **[Mil80] [Par81]** (dit vérification comportementale).

Pour prévenir ainsi d'éventuels dysfonctionnements, une pléthore de technique ont été développées. Parmi celles-ci, les tests et la simulation sont les plus connus et probablement les plus largement utilisés. Dans les vérifications basées sur les tests, le système à vérifier est testé pour révéler d'éventuelles erreurs sur un ensemble de situations choisies. Faute de temps, il est souvent impossible de tester ce système pour toutes les situations possibles. L'ensemble des cas à tester doit être choisi avec une attention particulière pour couvrir le maximum de scénarios distincts possible. Néanmoins, il est impossible de garantir que le produit sera exemple de toute erreur.

Une autre technique utilisée pour assurer la validation de systèmes est la méthode formelle. Pour pouvoir être réalisée, la vérification impose une description formelle du système, ainsi qu'une spécification formelle des propriétés.

De nombreux formalismes de spécification dédiés aux systèmes concurrents ont été proposés en littérature tels que les automates communications, systèmes de transitions, algèbres de processus CCS, CPS, LOTOS, …

Parmi les formalismes les plus utilisés, les réseaux de Pétri occupent une place prépondérante et se trouvent être l'outil ayant été le plus largement étudié si l'on considère la diversité des techniques automatiques de vérification qui leurs sont associés. Ils offrent une structure très simple et forment toujours le support de nombreuses études théorique des systèmes concurrents et de leurs propriétés

2 Les vérifications basées sur le model-cheking

Cette dernière approche et la plus utilisées, est basées sur les modèles, permet une vérification simple et efficace et elle est complément automatisable. La vérification basée sur les modèles ou model-cheking est surtout applicable pour les systèmes ayant un espace d'états fini.

Les algorithmes de vérification dans cette méthode utilisent l'ensemble des états que le système peut atteindre pour prouver la satisfaction ou la non-satisfaction des propriétés. Cependant, le problème majeur de ce type de vérification est la taille souvent excessive de l'espace d'états. En effet elle peut être exponentielle par rapport à la taille de la description du système. Une des causes principales de cette explosion combinatoire du nombre d'états et le fait que l'explosion causes principales de cette explosion combinatoire du nombre d'états et le fait que l'exploration est réalisée en prenant en compte tous les entrelacements possibles d'événement concurrents.

Pour pallier à ce problème de l'explosion combinatoire, différentes solutions dont l'objectif et de réduire la taille de l'espace d'états, ont été proposées dans la littérature. Elles sont généralement basées sur :

- L'utilisation de structure de données particulières.
- L'exploitation de l'ordre partiel sur les occurrences des événements.
- L'exploitation de symétries du système.
- L'exploitation de la modularité.

3 Réseaux de Petri

Les réseaux de Pétri ont été introduits en 1962 par Carl Adam Pétri **[Pét62]** dans sa thèse "Kommunikation mit Automaten". Possédant une représentation graphique simple, les réseaux de Pétri permettent de modéliser des systèmes dynamiques échangeant des ressources. Ainsi, à l'aide des réseaux de Pétri on peut modéliser et analyser les systèmes discrets, particulièrement les systèmes concurrents, représenter les concepts de parallélisme, de synchronisation (rendez-vous, précédence...), de partage de ressources, de communication, de causalité.

Informellement, les réseaux de Pétri sont des graphes orientés composés d'un nombre fini de nœuds places et d'un nombre fini de nœuds transitions. Des arcs orientés peuvent relier des nœuds places à des nœuds transitions ou des nœuds transitions à des nœuds places. Les nœuds places, représentés par des cercles, représentent des ressources disponibles.

Le nombre de ressources disponibles est représenté par des points (appelés jetons) à l'intérieur de ces nœuds places. Un jeton représente donc une ressource.

Les nœuds transitions, représentés par des rectangles pleins, représentent des actions pouvant effectuer des échanges de jetons entre différentes places. L'orientation des arcs indique les échanges pouvant être effectués. Chaque échange consomme un jeton dans toutes les places pointant vers une transition et produit un jeton dans les places pointées par cette transition.

Concepts de base

Définition formelle :

Un réseau de Pétri (RDP) **[Rei85b] [Vid92]** est la donnée d'un ensemble fini de places, d'un ensemble fini de transitions et d'une fonction dite fonction de poids. Ceci définit la structure statique du système. L'état de celui-ci se modélise à l'aide d'un marquage que l'on fait évoluer en franchissant des transitions, ce qui correspond à exécuter les actions qui leur sont associées.

Formellement, un réseau de Pétri peut être représenté par un quadruplet $\mathrm{N} = (\mathrm{S}, \mathrm{T}, \mathrm{W}, \mathrm{M})$ où:

- ✓ P est un ensemble fini non vide de places : $S = \{s_1, s_2, \dots, s_n\}$.
- ✓ T est un ensemble fini non vide de transitions. $\mathrm{T} = \{t_1, t_1, \dots t_m\}$.
- ✓ $w: (S \times T) \cup (T \times S) \to N$ est la fonction de poids.
- ✓ $M: S \to N$ est la fonction de marquage.

Le marquage initial est donné par M_0. M(s) = k signifie que la place contient k marques (jetons). On dit aussi que le marquage de s est k.

$\mathrm{W}(\mathrm{s}, \mathrm{t}) =$ k signifie que la transition *t* utilise k jetons dans la place s.

De façon duale, $\mathrm{W}(\mathrm{t}, \mathrm{s}) =$ k signifie que la transition t crée k jetons dans la place **[Jar95]**.

Exemple : Soit le réseau de Pétri R = (S, T, W, M_0) où P = {p1, p2, p3} et T = {t1, t2, t3, t4}.

Définissons W et M_0 de la façon suivante :

W (p1, t2) = 1.W (p2, t1) = 1. W (p2, t3) = 3. W (p3, t4) = 1.

W (t1, p1) = 1. W (t2, p2) = 1. W (t3, p3) = 1. W (t4, p2) = 3.

M_0 (p1) = 0; M_0 (p2) = 3; M_0 (p3) = 0:

Pour tous les couples (x, y) non spécifies ci-dessus, W (x, y) = 0.

Par exemple : W (p1, t3) = W (t3, p1) = 0.

Représentation graphique

Un Réseau de Pétri est un graphe orienté comprenant deux types de sommets :

- ✓ Les places sont représentées par des cercles.
- ✓ Les transitions par des rectangles (ou traits).
- ✓ Ils sont reliés par des arcs orientés : un arc relie soit une place à une transition, soit une transition à une place mais jamais une place à une place ou une transition a une transition **[Jarras95]**.

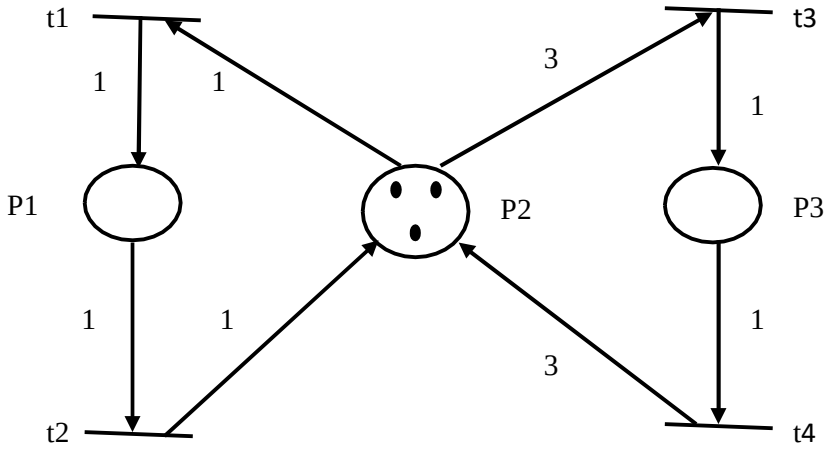


Figure 3.2*: Représentation graphique d'un réseau de Pétri.*

Cas particuliers :

« Pas de place en entrée de la transition »

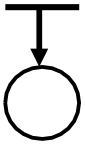


Figure 3.3 : *Transition.*

« Pas de place en sortie de la transition »

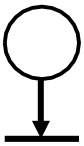


Figure 3.4 : *Transition puits.*

Exemples de réseau de Pétri

Système de quatre saisons

On considère un système d´écrivant quatre saisons. S'il est printemps et l'événement "début d'été" se produit, la saison changea l'été. Ensuite, si l'événement "début de automne" se produit, la saison est automne, etc. Le système modélise le changement séquentiel entre les saisons.

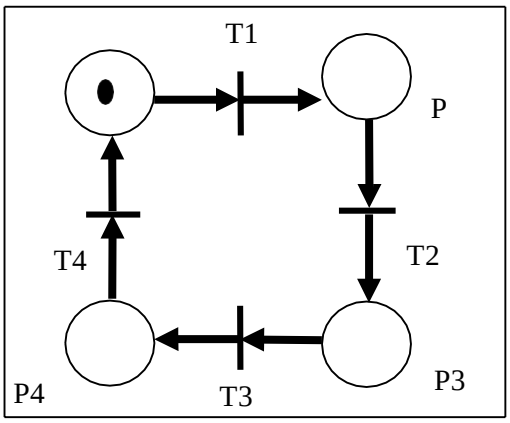


Figure 3.5 : *Réseau de Pétri pour le système de quatre saisons.*

Système de producteur-consommateur

On décrit le système d'un producteur et un consommateur utilisant un tampon comme leur boutique pour échanger les items (ou les marchandises). Le producteur produit des items (représenté par des jetons dans le tampon). Le consommateur peut les acheter en supprimant des items dans le tampon. Un franchissement d'une transition supprime un jeton de l'entrée et ajoute un jeton à chaque place de sortie à la fois. Cela correspond à l'échange des marchandises dans la réalité.

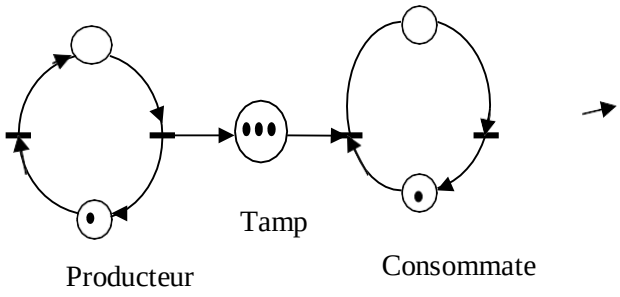


Figure 3.6 : *Réseau de Pétri pour le système de producteur- consommateur.*

Marquages Accessibles et Graphe des Marquages

Pour définir l'état d'un système modélisé par un réseau de Pétri, il est nécessaire de compléter le réseau de Pétri par un marquage. Ce marquage consiste à disposer un nombre entier (positif ou nul) de jetons dans chaque place du réseau de Pétri.

L'ensemble des marquages accessibles d'un RdP à partir d'un marquage initial donné, correspond à l'ensemble des marquages atteint après franchissement de transitions sensibilisées les unes après les autres ; ce qui correspond à toutes les situations possibles du RdP au cours de son évolution à partir du marquage initial. On appelle marquage M d'un RdP le vecteur dont les composantes représentent le nombre de jetons dans chaque place : la ième composante de ce vecteur correspond au nombre de jetons dans la ième place. Il indique à un instant donné l'état du RdP.

On note le marquage initial, M_0, le marquage à l'instant initial (t =0).
L'ensemble des marquages accessibles, A(R;M0), pour le RdP ci-dessous est: A(R;M0) = {M0, M1, M2, M3, M4}.

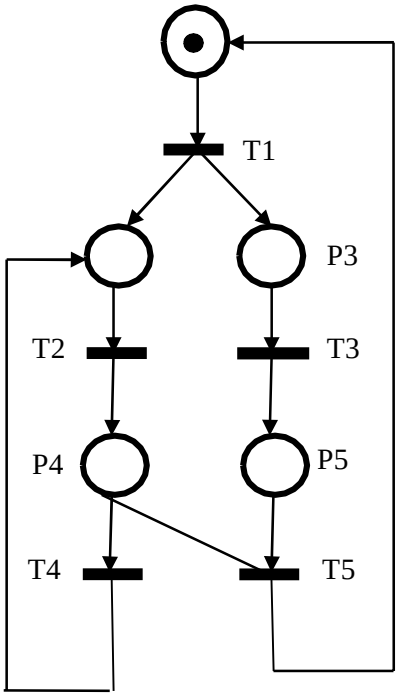


Figure 3.7 : *Un réseau de Petri.*

L'évolution du RdP est représentée ci-dessous avec les marquages représentés sous la forme de vecteurs colonnes L'évolution du RdP est représentée ci-dessous avec les marquages représentés sous la forme de vecteurs colonnes.

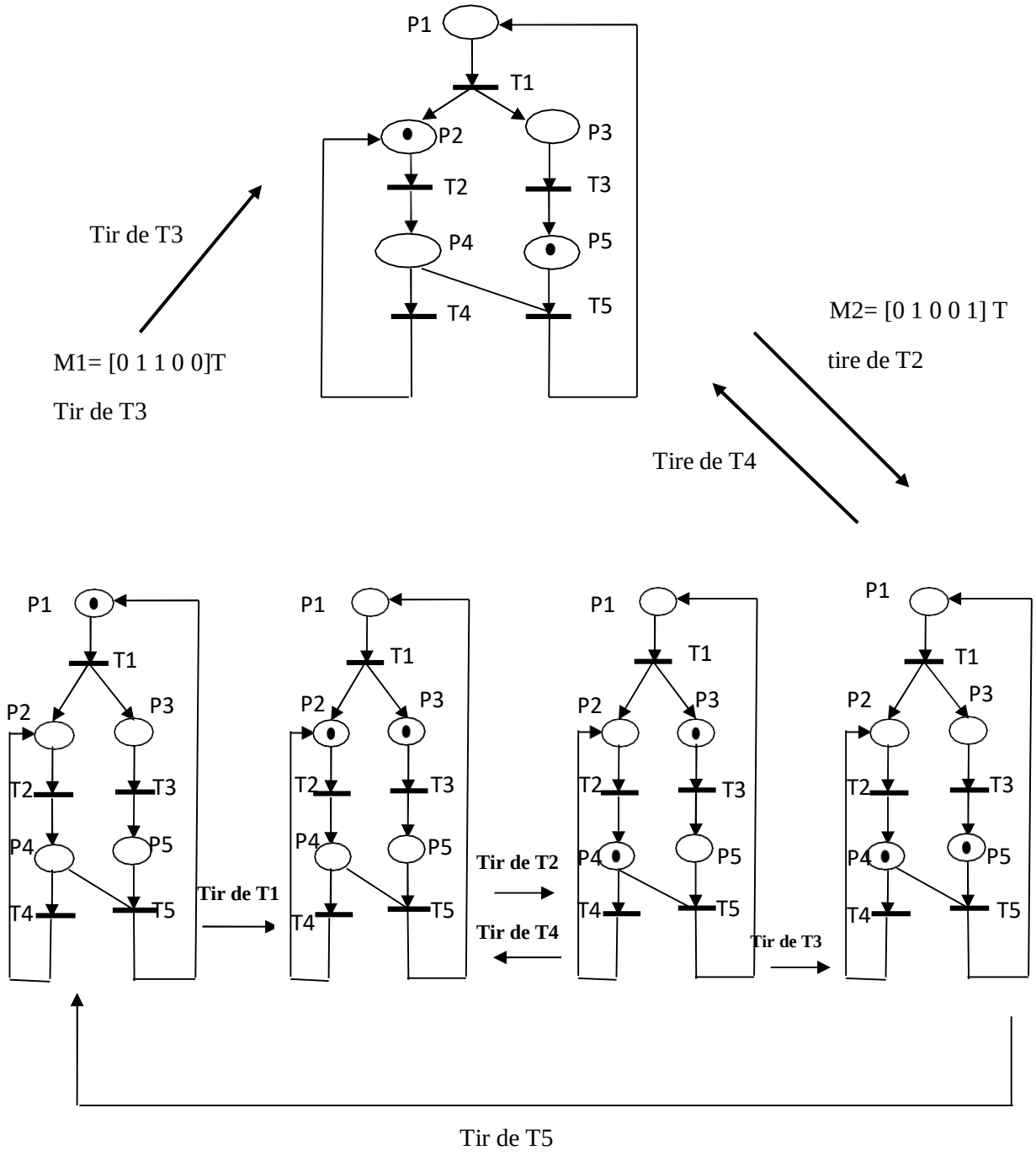
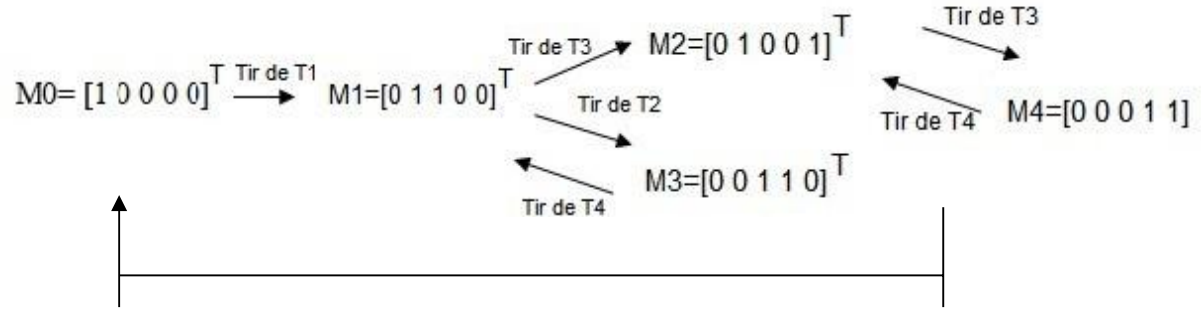


Figure 3.8 *: Ordre de tire des transitions.*

L'évolution du RdP peut être représentée sous la forme d'un **graphe des marquages GA (R ; M0)**, dont les sommets correspondent aux marquages accessibles.



Enfin, il est possible de représenter un graphe de marquage sous forme d'un « organigramme ». Ainsi pour l'exemple ci-dessous, on aura :

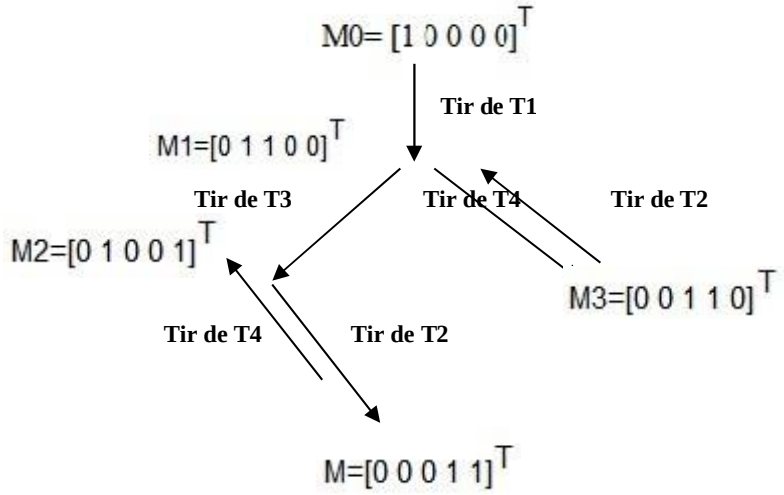


Figure 3.9 *: Graphe de marquage.*

Le marquage dit initial décrit l'état initial du système. Ainsi, l'ensemble des marquages accessibles à partir du marquage initial par franchissement d'une séquence de transition correspond à l'ensemble des états du système. Contrairement aux graphes d'état, une place d'un RdP ne correspond pas à un état du système mais participe, à travers son marquage, à la description d'un ou de plusieurs états du système. L'ensemble des marquages accessibles est équivalent au graphe d'état représentant le comportement du système.

Remarque :

- On utilise le graphe de marquages quand le nombre de marquages accessibles est fini.
- La représentation graphique d'un graphe de marquage permet de déterminer certaines propriétés de celui-ci. Par exemple si le graphe présente une zone non bouclée, cette partie du marquage une fois atteinte constitue un arrêt de l'évolution du RdP et celui-ci sera déclaré avec blocage.

4 Interprétations de jetons individuels

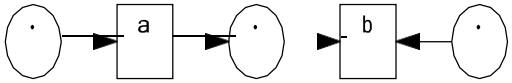
Dans la littérature, il existe un autre type d'interprétation de jeton est celle de jetons individuels.

Interprétations symboliques, individuelles et collectives

L'interprétation des jetons individuels des réseaux de Petri identifie différents jetons existant dans la même place tout en gardant une trace de leurs origines (place d'origine). Il existe un lien de causalité entre les deux jetons si une transition est résolue en utilisant un jeton émis par une autre transition. En conséquence, les relations causales entre les transitions peuvent toujours être caractérisées à l'aide d'un ordre partiel.

Les jetons ne peuvent pas être distingués dans l'interprétation collective des jetons, en revanche : s'il y a deux jetons dans un emplacement, tout ce qui s'y trouve est le numéro 2. Cela se traduit par des liens de causalité plus nuancés entre les transitions qui ne peuvent pas être énoncés à l'aide d'ordres partiels.

La différence entre les deux lectures est visible dans l'exemple suivant.

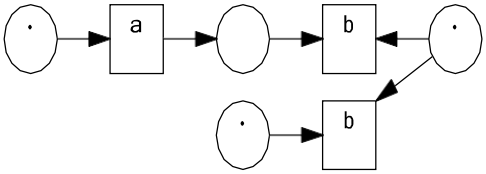
A : 

Les transitions a et b dans ce réseau peuvent s'exécuter une seule fois chacune d'elles. Il y a deux jetons au milieu après que a soit exécutée, n'importe lequel de ces jetons sera utilisé pour exécuter b. Les transitions a et b sont causalement indépendantes si le jeton qui était déjà présent est utilisé (ce qui doit être le cas si b se produit avant que le jeton de a n'arrive). Lorsque le jeton créé par a est utilisé, b devient causalement dépendant de a. En conséquence, le réseau A ci-dessus a deux exécutions maximales, qui sont désignées par les ordres partiels b^a et $a \rightarrow b$. Le nombre 2 est tout ce qui est présent en place médiane après l'apparition de a, selon la théorie des jetons collectifs. Parce que les conditions préalables pour que b soit restent constantes, b est toujours causalement indépendant de a.

Les deux philosophies fournissent des idées incomparables d'équivalence, comme indiqué ci-dessous.

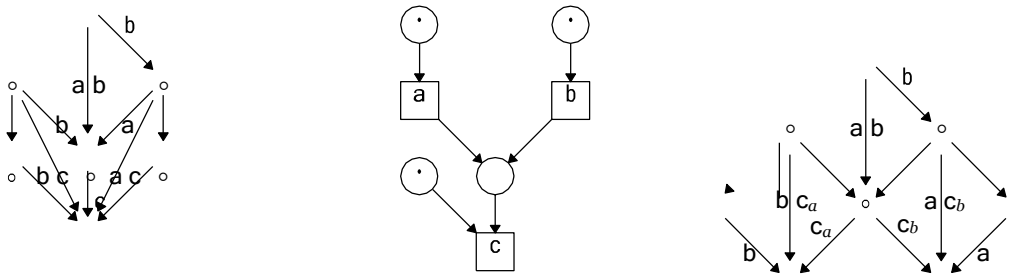
B : 

L'exigence de b représentée par la place médiane est redondante dans la théorie des jetons collectifs, donc A doit être comparable à B. Cependant, parce que B n'a pas l'exécution $a \rightarrow b$, A et B ne sont pas complètement comparables à la bisimulation simultanée (une causalité respectant équivalence basée sur la méthode du jeton individuel **[BDKP91]**). A, en revanche, est une bisimulation entièrement simultanée équivalente à C ci-dessous.

C : 

En réalité, en dépliant **[Eng91]**,**[MMS97]** C est le réseau d'occurrences produit à partir de A.

A et C ont les exécutions $a \rightarrow b$ et a b dans la philosophie individuelle des jetons. Cependant, A n'a pas de course $a \rightarrow b$ dans la philosophie symbolique collective, et ne peut donc pas être équivalent à C dans un sens préservant la causalité.



Le réseau de Petri D (ignorez $H_{\mathrm{ct}}(D)$et $H_{\mathrm{it}}(D)$ pour le moment) montre comment l'interprétation collective des jetons génère des liens de causalité qui ne peuvent pas être énoncés par un ordre partiel. Ce réseau a une causalité disjonctive selon l'interprétation symbolique collective : c dépend causalement de $a \vee b$. L'interprétation individuelle des jetons D, en revanche, permet deux exécutions : une dans laquelle c dépend uniquement de a, et une autre dans laquelle c dépend uniquement de b.

Une règle de tir pour l'interprétation individuelle des jetons

La question qui se peut être posée comment distinguer les jetons. À cette fin, chaque occurrence de jeton concevable recevant un nom unique. Un jeton sera un triplet (t', k, s) avec s indiquant l'origine (place) du jeton et t' indiquant la transition qui l'y a amené. La notation $t' = *$ pour les jetons qui sont en s à l'état initial. Lorsque le nombre de jetons que t' dépose en s en n ces jetons-là seront identifiés en leur fournissant des valeurs ordinales k = 0, 1, 2, . . . , n−1.

Pour définir des jetons comme indiqué précédemment, il faut voir le tir des transitions comme des paires $(X; t)$, avec t la transition de tir et X les jetons utilisés pendant le tir. Les transitions t qui ne consomment pas de jetons peuvent se déclencher plusieurs fois sur la même entrée (vide) ; ces tirs seront appelés (k, t) avec $k \in \mathbb{N}$ au lieu de $(\emptyset, t)$.

La fonctions β fournit l'origine d'un jeton $\beta(x, k, s) = s$ et η fournit la transition d'un tir $\eta(x, t) = t$. Par $\beta(X)(s) = |\{s' \in X \mid \beta(s') = s\}|$, Par la fonction β se prolonge en une fonction allant d'ensembles de jetons X à des multiensembles d'emplacements $\beta(X)$: $S \rightarrow IN$.

Définition : Étant donné un réseau de Petri $N = (S\ ;\ T\ ;\ F\ ;\ I\ ;\ l)$, les ensembles de jetons $S\bullet$ et de tirs de transition $T\bullet$ de N sont définis récursivement par

– $(*, k, s) \in S\bullet$ pour $s \in S$ et $k < I(s)$;
– $(t', k, s) \in S\bullet$ pour $s \in S, t' \in T\bullet$ et $k < F(\eta(t'), s)$;
– $(X, t) \in T\bullet$ pour $t \in T$ et $X \subseteq S\bullet$ tells que $\beta(X) = \bullet t \neq 0$;
– $(k, t) \in T\bullet$ pour $k \in IN$ et $t \in T$ tel que $\bullet t = 0$.

Aux tirs de transition, la fonction d'étiquetage $l\bullet\colon T\bullet \to A$ est assurée par $l\bullet(t) = l(\eta(t))$. Un multi ensemble $M : S\bullet \to IN$ de jetons constitue un marquage individuel de N. $I\bullet(*, k, s) = 1\ and\ I\bullet(t', k, s) = 0$. Offrent le premier marquage individuel $\bullet\colon S\bullet \to IN$.

Réseaux standards : Un réseau standard est un réseau N avec au moins un arc entrant à chaque transition : $\forall t \in T.\ \bullet t > 0$. Si la collection de transitions spontanées d'un réseau $T_\circ = \{(k, t) \in T. \mid k \in \text{IN}\}$ est vide, elle est considérée comme standard. Tout d'abord, je définis la règle de déclenchement qui encapsule l'interprétation individuelle des jetons pour les réseaux conventionnels.

Définition : Dans un réseau typique, pour un ensemble fini $U \subseteq T.$d'anneaux de transition,

$$\bullet U = \sum_{(K,t)\in U} X\ and\ U\bullet = \{(t', k, s) \mid t' \in U \wedge k < F(\eta(t'), s)\}$$

Soit l'ensemble des jetons d'entrée et l'ensemble des jetons de sortie du multi ensemble de jetons d'entrée U. Si U$\leq M$, l'ensemble U est activé sous un marquage individuel $M \in \text{IN}^{S\bullet}$. Dans une telle situation, U peut tirer sous M, résultant en $M' = M - {}^\bullet U + U^\bullet \in \text{IN}^{S\bullet}$, représenté par $M \xrightarrow{U} M'$.

Une séquence de tirs $I. \xrightarrow{U1} \xrightarrow{U2} \cdots \xrightarrow{Un} M_n$ est appelée chaîne. Si une séquence se terminant par $M = Mn$ existe, un marquage individuel $M \in \text{IN}^{S\bullet}$peut être atteint.

Pour un tir $u \in T.$ tel que $u = (X, t)$, soit $\cdot u = X$ et $u^\cdot = \{(u, k, s) | K < F(\eta(u), s)\}$ soit l'ensemble des jetons d'entrée et l'ensemble des jetons de sortie de u.la transition t est activée sous un marquage individuel ($M \in S.$) si $\cdot u \subseteq M$, dans ce cas , t peut se déclencher sous M, donnant $(M' = (M \setminus \cdot u) \cup u^\cdot)$, écrit comme $(M \to. M')$ ou $(M[u > M'[M >$ est l'ensemble des marquages accessibles depuis M pour chaque marquage M. Par conséquent, $[I. >$ est l'ensemble des marques accessibles depuis $I.$, ou en d'autres termes, c'est l'ensemble de toutes les marques accessibles depuis N.

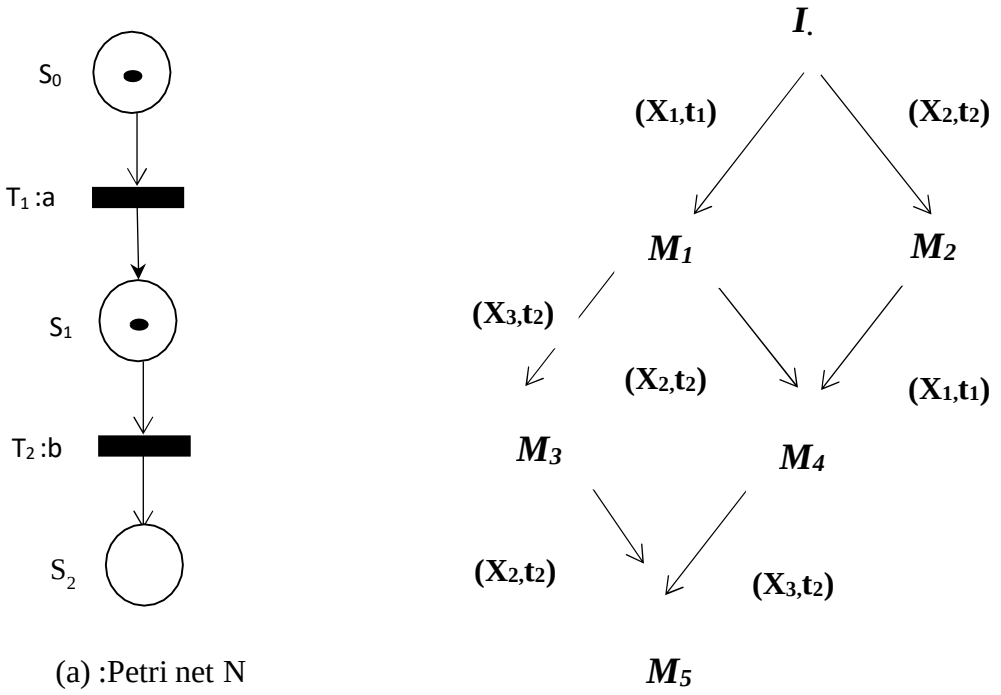


Figure3.10 : *graphique de Marquage dans l'interprétation individuelle des jetons*

Étant donné le réseau de Petri N de la figure 3.10.(a), où le marquage individuel initial I est l'ensemble $\{(*\ ,0,\ s_0),\ (*\ ,0,\ s_1)\}$, cet exemple explique comment les IPN$_s$ peuvent préserver une relation causale entre les actions. Transition t_1 (respectivement, t_2) est activé à partir de l'état initial; en conséquence, nous avons le tir $u_1 = (\{(*\ ,0,\ s_0)\},\ t_1)$ (respectivement, $u_2 = (*\ ,0, s_1),\ t_2$). u_1 établit l'état $M1$, tel que $M1 = (u_1, 0, s_1\ ), (*$ $,0,\ s_1)$ qui a deux tir possibles : le premier est produit par le jeton de départ $(*\ ,0,\ s_1)$, et le second est causé par la conséquence de u_1; c'est-à-dire que le jeton $(u_1,\ 0,\ s_1\ )$, notés respectivement par u_3 et u_4 comme $u3 = (\{(*\ ,0,\ s_1)\},\ t_2)$ et $u4 = (\{(u_1,\ 0,\ s_1\ )\},\ t_2)$. Cela signifie que nous pouvons distinguer le tir de t_2 causé par l'exécution de t_1 et le tir de t_2 causé par le jeton initial de s_1. Nous n'avons pas cette option dans l'interprétation collective des jetons. La figure 3.10.(b) représente le graphique de marquage de N, qui indique :

- $X_1 = \{(*\ ,0s_0)\}$ et $M_1 = \{((X_1, t_1), 0, s_1), (*\ ,0, s_1)\}$,
- $X_2 = \{(*\ ,0, s_1)\}$ et $M_2 = \{((X_2, t_2), 0, s_2), (*\ ,0, s_0)\}$,
- $X_3 = \{((X_1, t_1), 0, s_1)\}$, et $M_3 = \{((X_3, t_2), 0, s_2), (*\ ,0, s_1)\}$,
- $M_4 = \{((X_1, t_1), 0, s_1)((X_2, t_2), 0, s_2)\}$,
- $M_5 = \{((X_3, t_2), 0, s_2)((X_2, t_2), 0, s_2)\}$,

5 Les systèmes distribués et Réseaux LSGA

Dans cette section et selon **[VGS12]**, nous présentons ce que nous entendons par un système distribué, puis utilisons des réseaux de Petri pour formaliser un modèle de systèmes distribués.

- Un système distribué est constitué de composants situés à différents endroits.
- Les composants fonctionnent en parallèle. Les communications est explicites. La communication entre les composants est lente et asynchrone.
- Dans un système distribué, la communication asynchrone est le seul moyen de partager des signaux ou des informations.
- L'envoi d'un message a toujours lieu avant sa réception (il existe une relation causale entre l'envoi et la réception d'un message).
- Un composant émetteur communique sans se soucier de l'état du récepteur ; il n'est pas nécessaire de se synchroniser avec un composant récepteur en particulier. Le comportement de l'expéditeur continue après l'envoi du message, que le message soit reçu ou non.
- Les composants ne doivent permettre qu'un comportement séquentiel.

En termes de places d'entrée et de sortie, les systèmes distribués sont représentés comme des réseaux constitués de réseaux de composants à comportement et interfaces séquentiels.

Définition 1.1. Soit $N = (S, T, F, M_0, \ell)$ un réseau de Petri, $I, O \subseteq S, I \cap O = \emptyset$, et $O^{\bullet} = \emptyset$.

1. Le composant (N, I, O) possède l'interface (I, O).
2. Si $\exists Q \subseteq S \backslash (I \cup O)$ avec $\forall t \in T. | {}^{\bullet}t \upharpoonright Q| = 1 \wedge |t^{\bullet} \upharpoonright Q| = 1)$ et $|M_0 \upharpoonright Q| = 1, (N, I, O)$ est une composante séquentielle avec interface (I, O)

Une boîte aux lettres de C pour un type de message donné peut être considérée comme une place d'entrée $i \in I$ d'un composant $C = (N, I, O)$. En revanche, un lieu de sortie $o \in O$ est une adresse extérieure à C à laquelle C peut envoyer des messages. Placer un jeton en o revient à envoyer une lettre. La condition $o^{\bullet} = \emptyset$ indique que le composant ne peut pas récupérer un message une fois qu'il a été posté. Un $S - invariant$ est une instance spécifique d'un ensemble d'emplacements comme Q. Étant donné que les critères garantissent que le nombre de jetons dans ces emplacements reste constant, dans cet exemple 1, aucune transition ne peut jamais se déclencher en même temps (en une seule étape). Lorsqu'un réseau est séquentiel, c'est-à-dire que deux transitions ne peuvent pas se déclencher dans la même étape, il est simple de le transformer en un réseau comportementalement identique avec $l'invariant\ S$ requis, ce qui se fait en ajoutant un seul emplacement marqué avec un auto-boucle à toutes les transitions. Ce changement préserve presque toutes les équivalences sémantiques sur les réseaux de Petri trouvées dans la littérature, y compris $\approx \Delta_{bSTb}$.

Puis, en fusionnant les places d'entrée et de sortie, nous construisons un opérateur pour combiner des composants avec une communication asynchrone.

Définition : Considérons l'ensemble d'index $\mathfrak{K}$.

Soit $((S_k, T_k, F_k, M_{0k}, \ell_k), I_k, O_k)$ avec $(k \in \mathfrak{K})$ des composants avec des interfaces telles que $(S_k \cup T_k) \cap (S_l \cup T_l) = (I_k \cup O_k) \cap (I_l \cup O_l)$ pour tous les $k, l \in \mathfrak{K}$ avec $(k \neq l)$ (les composants sont disjoints sauf pour les emplacements des interfaces) et $I_k \cap I_l = \emptyset$ pour tous les $k, l \in \mathfrak{K}$ avec $k \neq l$ (les boîtes aux lettres ne peuvent pas être partagées ; tout message a un destinataire unique).

La composition parallèle asynchrone de ces composants est donc définie comme suit :

$\|_{i\in\mathfrak{K}}((S_k,T_k,F_k,M_{0k},\ell_k),\ I_k,O_k) = ((S,T,F,M_0,l),I,O),$

Avec $S = U_{k\in\mathfrak{K}}S_k$, $T = U_{k\in\mathfrak{K}}T_k$, $F = U_{k\in\mathfrak{K}}F_k$,$M_0 = \Sigma_{k\in\mathfrak{K}}M_{0k}$,$\ell = U_{k\in\mathfrak{K}}\ell_k$ (union par composants de tous les réseaux) ,$I = U_{k\in\mathfrak{K}}I_k$ (nous acceptons les entrées supplémentaires de l'extérieur), et $O = O_{k\in\mathfrak{K}}O_k\backslash U_{k\in\mathfrak{K}}I_k$ (une fois fusionné avec une entrée, $o \in O_I$ n'est plus une sortie).

Il convient de noter que la composition parallèle asynchrone de composants avec interfaces produit un autre composant avec interface.

Observation : || est associatif.

Cela découle immédiatement de l'associativité de l'opérateur d'union (multi)ensemble. Nous pouvons maintenant définir la classe net qui représente les systèmes avec des composants séquentiels communicants de manière asynchrone.

Definition : Un réseau de Petri N est un réseau LSGA (un réseau localement séquentiel globalement asynchrone) s'il possède un ensemble d'indices $\mathfrak{K}$et des composantes séquentielles d'interface C_k, $k \in \mathfrak{K}$, tels que $(N, I, O) = \|_{kG\mathfrak{K}}\ C_k$ pour certains I et $(N, I, O) = \|_{kG\mathfrak{K}}\ C_k$ pour certains O.

La même classe de systèmes LSGA aurait été obtenue jusqu'à $\approx \Delta_{bSTb}$ ou toute équivalence raisonnable préservant la causalité et le temps de branchement mais faisant abstraction de l'activité interne— si nous avions imposé, dans la Définition (4.1) des composantes séquentielles, que I, O, et Q forment une partition de S et que $\bullet I = \emptyset$.[4] . Cependant, il est essentiel que notre spécification permette à plusieurs transitions de composants de lire à partir du même emplacement d'entrée.

La figure 3.11 montre un LSGA formé par deux composants Sub1 et sub2.

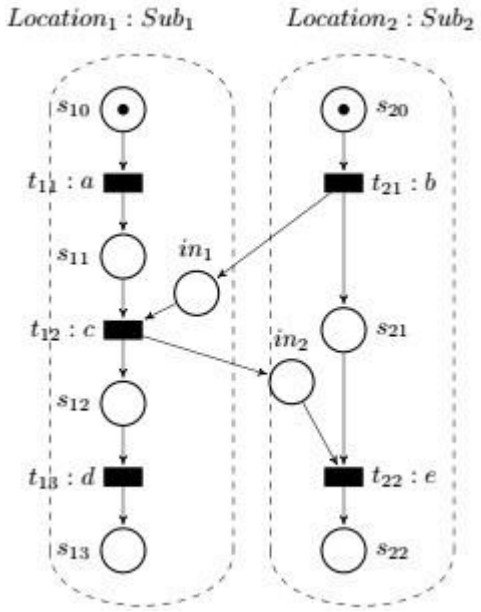


Figure 3.11 : *Un LSGA.*

6 Conclusion

Notre objectif est comment modéliser les microservices en assurant (i) la scalabilité, (ii) calcul de l'interaction (iii) une méthode d'immigration d'une application monolithique vers les microservices. Ce chapitre présente notre cadre théorique qui rependre aux trois point-là.

Chapitre 04

La proposition d'un modèle pour les microservice

Ce présent travail cherche à donner une représentation de haut niveau d'une architecture microservices :

1. En conservant l'historique de l'interaction entre les différents microservices,
2. En assurant la scalabilité.
3. En offrant une méthode d'immigration.

Pour assurer les deux premiers points, Nous allons proposer une version d'un LSGA avec la sémantique de causalité sous-jacente, nommée PN4M (Petri Net for Microservices). Cette sémantique est fournie par la représentation individuelle des jetons. Pour l'immigration, en va utiliser la méthode de distribution d'un réseau de Petri définie dans **[VGS12]**.

L'architecture de microservice voit la conception d'une application comme une collection de petits services autonomes où chaque service s'exécute dans son propre processus distinct et les services peuvent communiquer en utilisant des protocoles légers (souvent basés sur HTTP). Donc chaque microservices k est un composant LSGA : $((S_k, T_k, F_k, M_{0k}, \ell_k), I_k, O_k)$où la communication est assurée par le couple (I_k, O_k).

1. La communication

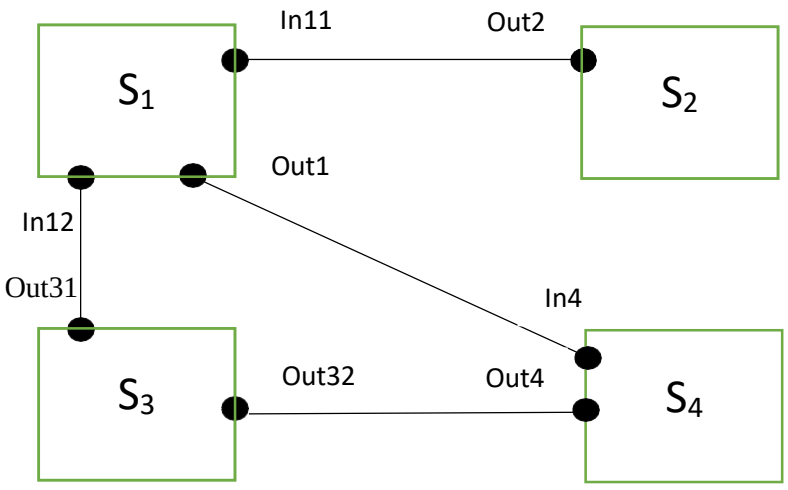
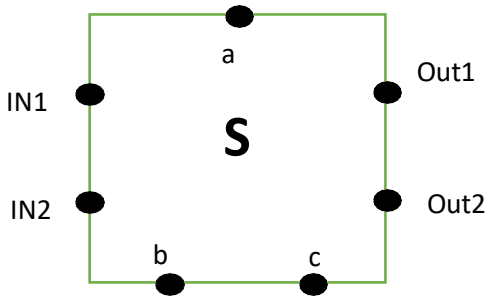


Figure 4.1 : *Point de communication*.

Dans l'exemple de la figure 4.1, on a quatre microservices à savoir :

1. $((S_1, T_1, F_1, M_{01}, \ell_1), I_1, O_1)$ où l_1={In_{11},In_{12}} et O_1={Out_1}.
2. $((S_2, T_2, F_2, M_{02}, \ell_2), I_2, O_2)$ où l_2=∅ et O_2={Out_2}.
3. $((S_3, T_3, F_3, M_{03}, \ell_3), I_3, O_3)$ où l_3=∅ et O_3={Out_{31},Out_{32}}.
4. $((S_4, T_4, F_4, M_{04}, \ell_4), I_4, O_4)$ où l_4=={In_{41}} et O_4={Out_4}.

Où chaque composant en peut le voir de la forme ci-dessous :



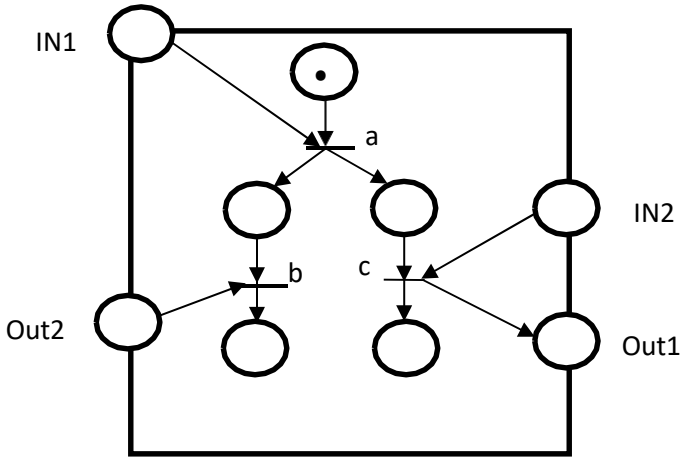


Figure 4.2 : *L'interface d'un microservices.*

2. La scalabilité

La scalabilité est assurée par l'ajout d'une place spécifique dans laquelle en introduire le nombre des instance souhaitées.

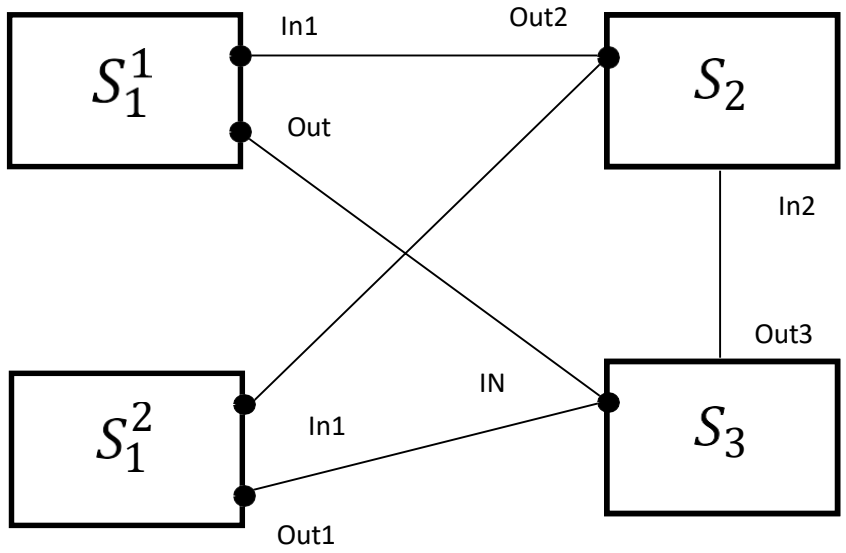


Figure 4.3 : *Un système avec deux instances d'un composant donné.*

Dans l'exemple de la figure 4.3, on a quatre microservices à savoir :

1. $((S_1, T_1, F_1, M_{01}, \ell_1), I_1, O_1, Ins_1)$ où l_1*={In₁,} et* O_1*={Out₁}.*
2. $((S_2, T_2, F_2, M_{02}, \ell_2), I_2, O_2, Ins_2)$ où l_2*={In2,} et* O_2*={Out₂}.*
3. $((S_3, T_3, F_3, M_{03}, \ell_3), I_3, O_3, Ins_3)$ où l_3*=={In₃} et* O_3*={Out₃}.*

Un microservice sera défini formellement comme suite :
$((S_k, T_k, F_k, M_{0k}, \ell_k), I_k, O_k, Ins_k)$ où $Ins_k \in S$.

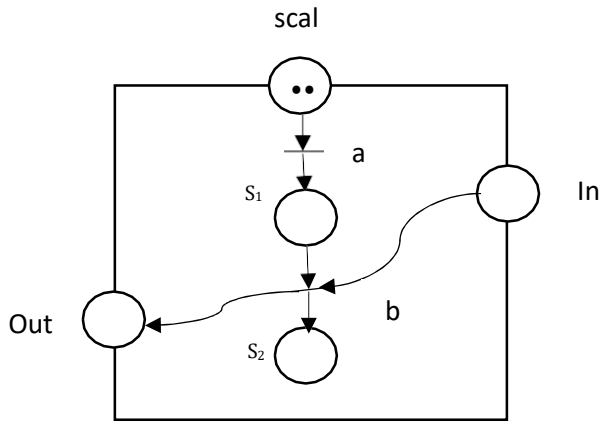


Figure 4.4 : *Un microservices supporte la mise en échelle.*

Pour l'exemple de la figure 4.4, le microcervice sera représenté comme suite :

$((S, T, F, M_0, \ell), I, O, Ins)$ où :

1. S={scal,s*1*,s*2*,In,Out},
2. I={In},
3. O={Out},
4. Ins={(*,0,scal),(*,1,scal)}.

La définition complète de PB4M est donnée comme ci-dessous.

Définition : Considérons l'ensemble d'index $\mathfrak{K}$.

Soit $((S_k, T_k, F_k, M_{0k}, \ell_k), I_k, O_k, Ins_k)$ avec $(k \in \mathfrak{K})$ des composants avec des interfaces telles que $(S_k \cup T_k) \cap (S_l \cup T_l) = (I_k \cup O_k) \cap (I_l \cup O_l)$ pour tous les $k, l \in \mathfrak{K}$ avec $(k \neq l)$ (les composants sont disjoints sauf pour les emplacements des interfaces) et $I_k \cap I_l = \emptyset$ pour tous les $k, l \in \mathfrak{K}$ avec $k \neq l$ (les boîtes aux lettres ne peuvent pas être partagées ; tout message a un destinataire unique) et $Ins_k \in S$. .

La composition parallèle asynchrone de ces composants est donc définie comme suit :

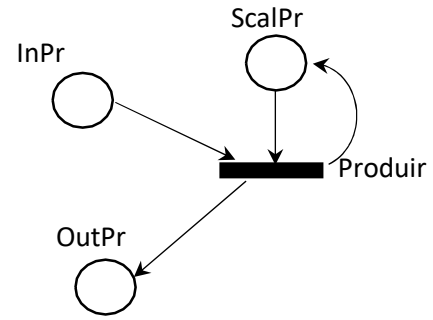
$\|_{i \in \mathfrak{K}}((S_k, T_k, F_k, M_{0k}, \ell_k), I_k, O_k, Ins_k) = ((S, T, F, M_0, l), I, O, Ins)$,

Avec $S = U_{k \in \mathfrak{K}} S_k$, $T = U_{k \in \mathfrak{K}} T_k$, $F = U_{k \in \mathfrak{K}} F_k, M_0 = \Sigma_{k \in \mathfrak{K}} M_{0k}, \ell = U_{k \in \mathfrak{K}} \ell_k$ (union par composants de tous les réseaux) ,$I = U_{k \in \mathfrak{K}} I_k$ (nous acceptons les entrées supplémentaires de l'extérieur), et $O = O_{k \in \mathfrak{K}} O_k \backslash U_{k \in \mathfrak{K}} I_k$ (une fois fusionné avec une entrée, $o \in O_I$ n'est plus une sortie) et $Ins = U_{k \in \mathfrak{K}} Ins_k$.

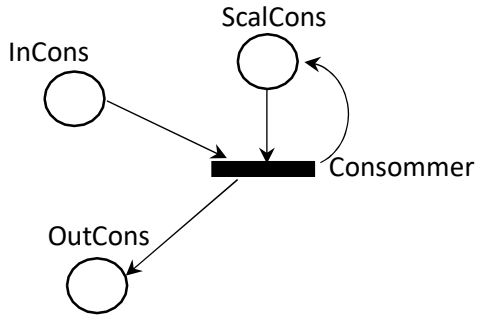
3. Cas d'étude

Le problème des producteurs et des consommateurs est un exemple informatique de synchronisation de ressources, qui peut s'envisager dans différents contextes de programmation concurrente, notamment en environnement multi-thread. Il s'agit de partager entre deux tâches, le producteur et le consommateur, une zone de mémoire tampon utilisée comme une file. Le producteur génère un élément de données, l'enfile sur la file et recommence ; simultanément, le consommateur retire les données de file. Ce problème peut être généralisé à plusieurs producteurs ou consommateurs.

Dans notre contexte, on peut voir le système comme deux types de microservices à savoir les producteurs et les consommateurs. Graphiquement en peut voir les microservices Producteur et consommateurs (séparés) comme la figure 4.5.



(a). *Producteurs.*



(b). *Consommateurs.*

Figure 4.5 *: Les microservices séparés.*

On peut noter les deux composant comme ci-dessous où on a deux instances de Producteur et trois instances de Consommateur.

1. Producteurs=$((S_p, T_p, F_p, M_{0p}, \ell_p), I_p, O_p, Ins_p)$ où :
    1. S_p={scalPr,InPr,OutPr},
    2. I_p={InPr},
    3. O_p={OutPr},
    4. Ins_p={(*,0,scalPr),(*,1,scalPro)}.
2. Consommateurs =$((S_c, T_c, F_c, M_{0c}, \ell_c), I_c, O_c, Ins_c)$ où :
    1. S_c={scalCn,InCn,OutCn},
    2. I_c={InCn},
    3. O_c={OutCn},
    4. Ins_c={(*,0,scalCn),(*,1,scalCn),),(*,2,scalCn)}.

La figure 4.6 montre comment construire le système complet avec un tampon de capacité 2.

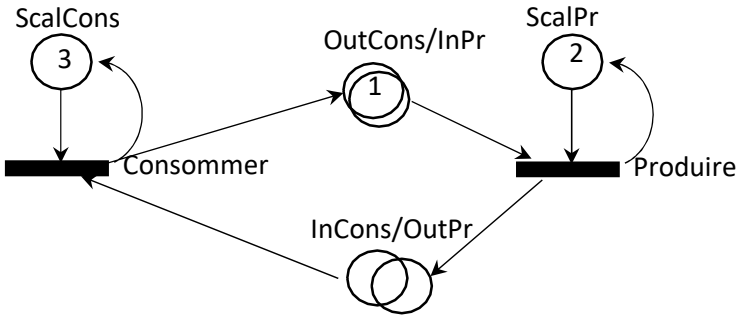


Figure 4.6 : *Producteurs/Consommateurs.*

4. La sémantique

La sémantique adoptée ici c'est la sémantique de causalité qui sera assurée par l'interprétation individuelle des jetons.

Définition : Étant donné un réseau de Petri $N = ((S, T, F, M_0, l), I, O, Ins)$, , les ensembles de jetons $S\bullet$ et de tirs de transition $T\bullet$ de N sont définis récursivement par

– $(*, k, s) \in S\bullet$ pour $s \in S$ et $k < I(s)$;
– $(t', k, s) \in S\bullet$ pour $s \in S, t' \in T\bullet$ et $k < F(\eta(t'), s)$;
– $(X, t) \in T\bullet$ pour $t \in T$ et $X \subseteq S\bullet$ tells que $\beta(X) = \bullet t \neq 0$;
– $(k, t) \in T\bullet$ pour $k \in IN$ et $t \in T$ tel que $\bullet t = 0$.

Aux tirs de transition, la fonction d'étiquetage $l\bullet: T\bullet \to A$ est assurée par $l\bullet(t) = l(\eta(t))$. Un multi ensemble $M: S\bullet \to IN$ de jetons constitue un marquage individuel de N. $\boldsymbol{I}\bullet(*, k, s) = 1$ and $\boldsymbol{I}\bullet(t', k, s) = 0$. Offrent le premier marquage individuel $\bullet: S\bullet \to IN$.

5. Immigration vers les microservices

Cette section présente une méthode qui fournit un PN4M à partir d'un réseau de Petri donné. L'idée est d'utiliser la définition **[VGS12]**.

Définition : Un réseau de Petri N = (S, T, F, M0, ℓ) est distribuable si seulement s'il existe une distribution D telle que :

1. ∀s ∈ S, t ∈ T. s ∈ •t ⇒ D(t)=D(s),
2. ∀t, u ∈ T. t ⌣ u ⇒ D(t) D(u). avec t ⌣ u si seulement si ·t ∩ ·u = ∅

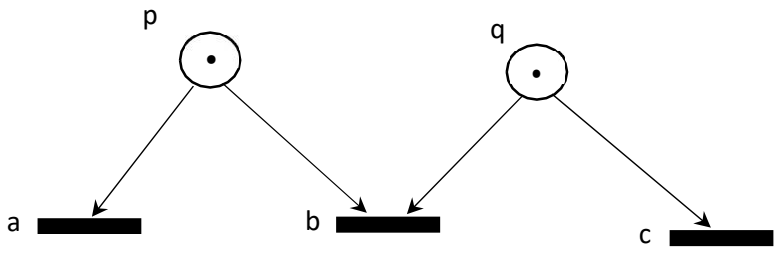


Figure 4.7 : *Un système non distribuable.*

La figure 4.7 montre un système non distribuable. Le problème est qu'il y a un conflit au niveau de la place p (respectivement q) entre la transition a et b (respectivement entre b et c).

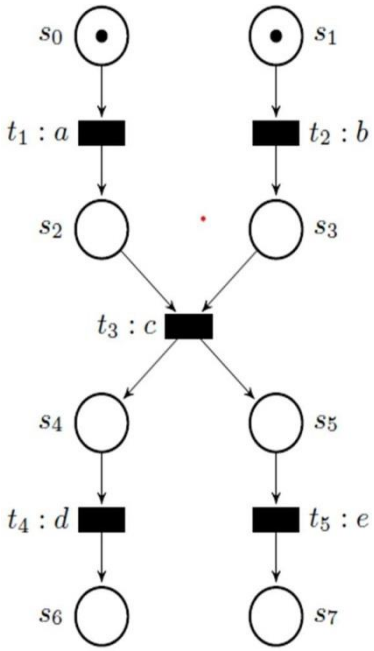


Figure 4.8 : *Un système distribuable.*

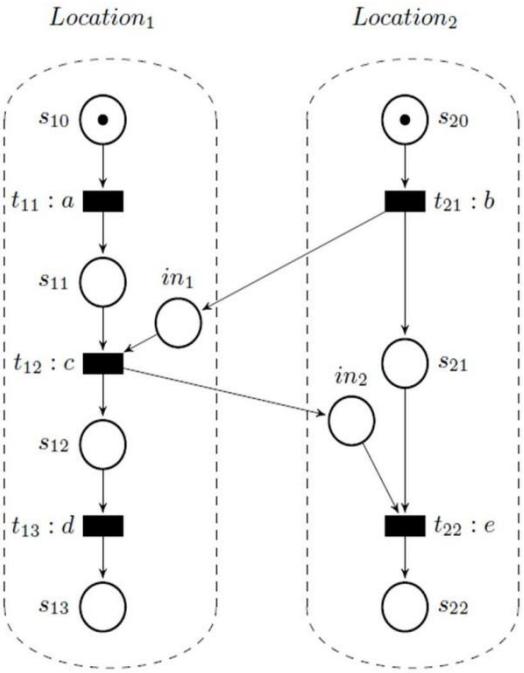


Figure 4.9 : *Un PN4M immigré à partir de Figure 4.8.*

La figure 4.8 représente un réseau de Petri d'un système monolithique. En appliquant la définition de la fonction de distribution en trouve que :

1. $D(s_0)=D(a)=D(s_2)=D(s_3)=D(s_4)=D(d)=D(s_6)$.
2. $D(s_1)=D(b)=D(s_3)=(s5)=D(e)=D(s_7)$.

Donc, on a deux microservices.

1. $((S_1, T_1, F_1, M_{01}, \ell_1), I_1, O_1, Ins_1)$ où $l_1=\{In_1,\}$ *et* $O_1=\emptyset$.
2. $((S_2, T_2, F_2, M_{02}, \ell_2), I_2, O_2, Ins_2)$ où $l_2=\{In2,\}$ *et* $O_2=\emptyset$.

La méthode sera définie comme suite.

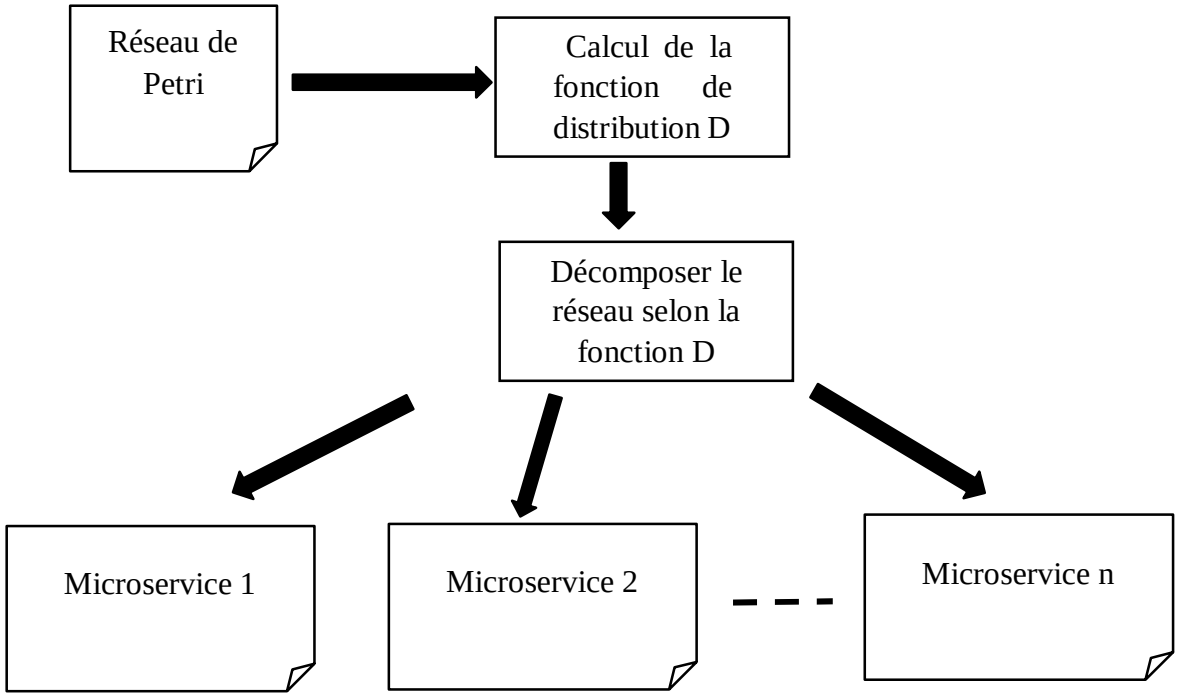


Figure 4.10 : Une méthode d'*immigration à partir d'un réseau de Petri.*

6. **Conclusion**

Dans ce chapitre nous avons donné une représentation de haut niveau d'une architecture microservices en conservant l'historique de l'interaction entre les différents microservices, et la scalabilité et donner une méthode d'immigration d'une application monolithique vers les microservices.

Conclusion générale

Même s'il y a beaucoup à dire sur la construction de systèmes basés sur des architectures de microservices, l'approche moderne n'est pas nécessairement le meilleur choix pour chaque entreprise ou projet. La mise en œuvre des microservices peut prendre plus de temps, en particulier pour les petits programmes informatiques qui ne peuvent gérer que quelques tâches dans tous les cas. De plus, la création de services, ainsi que la maintenance, le développement futur et la surveillance, prennent beaucoup de temps. Il est important d'évaluer soigneusement les résultats des microservices, en particulier lors de la surveillance des processus : d'une part, les microservices individuels sont très faciles à analyser et à mesurer. En raison du grand nombre de microservices, cette tâche s'est considérablement développée.

Une solution consiste à offrir un cadre formel pour la conception de ces systèmes. L'objectif étant d'offrir un modèle supporte la mise à l'échelle (la scalabilité) d'une architecture microservices et fournit un calcul formel, basé sur la sémantique de causalité, qui peut être utilisé dans la phase du déploiement, précisément dans la distribution des charges. Plus, nous avons proposé une méthode de transformation formelle pour avoir une architecture microservices à partir d'un système monolithique existant modélisé par les réseaux de Petri.

Ce présent travail est loin d'être terminé. En effet, nous avons seulement donner une réflexion sur l'utilisation da la sémantique de causalité afin de capte les interactions entre les microservices qui sera utile pour calculer la distribution de charge (load-balancing) dans la phase de déploiement. Il est très intéressant d'approfondir cette sémantique pour régler d'autre chalenges. Un Framework pour la conception des microservices supporte les notions évoquées dans notre travail aura une grande utilité.

Références

[AAE16] N. Alshuqayran, N. Ali & R. Evans, ”A Systematic Mapping Study in Microservice Architecture,” In Proc. IEEE 9th International Conference on SOCA, Macau, 2016, pp. 44-51.

[Arn92] : A. Arnold. Systèmes de transitions finis et sémantique des processus ommunicants. (1992).

[BB87]: T. Bolognesi and E. Brinksma. “Introduction to the ISO Specification Language LOTOS”, volume 14.Computer Networks and ISDN Systems (1987).

[Bed87]: M. A. Bednarczyk. “Categories of Asynchronous Systems”.PhD thesis, Univ Sussex (1987 1987).Available as CS R 1/88.

[Bel10] : Nabil Belala. Modèles de Temps et leur Intérêt à la Vérification Formelledes Systèmes Temps-Réel. PhDthesis, Laboratoire MISC, Université deMentouri, 25000 Constantine, Algérie, 2010.

[BDKP91] E. Best, R. Devillers, A. Kiehn & L. Pomello (1991): Concurrent bisimulations in Petri nets. Acta Informatica 28, pp. 231-264.

[BK85]:J. A. Bergstra and J. W. Klop. Algebra of communicating processes withabstraction. TCS, 37:77–121, 1985.

[BHJ1518] A. Balalaie, A. Heydarnoori, P. Jamshidi, “Microservices Migration Patterns,” Technical Report No. 1, Automated Software Engineering Group, Sharif University of Technology, October, 2015. [Online]. Available: http://ase.ce.sharif.edu/pubs/techreports/ TR-SUT-CE-ASE-2015-01-Microservices.pdf [Accessed Nov. 15,2018]

[CCI88] : CCITT88. SDL, recommendation z.100-z.104.CCITT (1988).

[CDT18] T. Cerny, M. J. Donahoo & M. Trnka, “Contextual understanding of microservice architecture: current and future directions”, ACM SIGAPP Applied Computing Review, 2018.

[CE81]: E. M. Clarke and E. A. Emerson. Design and synthesis of synchronizations keletons using branching time temporal logic. In “Logics of Programs Workshop”,volume 131, pages 52—71. LNCS (May 1981).

[CES83]: E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verificationof finite-state concurrent systems using temporal logic specification: A practicalapproach. In “10th ACM Symp.Principles of Programming Language”, pages 117—126, Austin, Texas (January 1983).

[DE.Saidouni] : Djamel –Eddine SAIDOUNI MODELES DU PARALLELISME5ème Année Ingénieur en InformatiqueOption : Systèmes Parallèles et DistribuésAnnée universitaire 2004/2005Présenté.

[DHM11] Uckelmann, Dieter, Mark Harrison, and Florian Michahelles. "An architectural approach towards the future internet of things." Architecting the internet of things. Springer Berlin Heidelberg, 2011. 1-24.

[DM14] Dmitry, N., & Manfred, S. S. On micro-services architecture. International Journal of Open Information Technologies, 2(9), 24-27. (2014).

[D et al 17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara F. Montesi, R. Mustafin & L. Safina, "Microservices: Yesterday, Today, and Tomorrow," In: Manuel Mazzara, Bertrand Meyer (eds), Present and Ulterior Software Engineering. Springer, Cham,2017.

[DS14] N. Dmitry & M. Sneps, "On Micro-Services Architecture," International Journal of Open Information Technologies, vol. 2, no. 9, 2014.

[ELMansouri] : RaidaELMansouri, modélisation et vérification des processus métiers dans les entreprises virtuelles : Une approche basée sur la transformation de graphes , université Mentouri Constantine.

[EH86]: E. A. Emerson and J. Y. Halpern. "sometims" and "not never" revisited: Onbranching versus linear time temporal logic. Journal of the ACM 33(1), 151—178(1986).

[Eng91]J. Engelfriet (1991): Branching processes of petri nets. Acta Informatica 28(6), pp. 575-591.

[FL18] M. Fowler, J. Lewis, "Microservices". [Online]. Available: https://www. martinfowler.com/microservices/ [Accessed Nov. 1, 2018]

[GP95] R.J. van Glabbeek & G.D. Plotkin (1995): Con_guration structures (extended abstract). In D. Kozen, editor: Proceedings 10th Annual IEEE Symposium on Logic in Computer Science, LICS'95, San Diego, USA, IEEE Computer Society Press, pp. 199-209. Available at http://boole.stanford.edu/pub/conf.ps.gz.

[HM80]: M. Hennessy and R. Miner. On observing nondeterminism and concurrency.In "Proc 7th ICALP", pages 299—309. Spriner-Verlag (July 1980).

[Hoa85]: C. A. R. Hoare. "Communicating Sequenctial Processes".Prentice Hall (1985).

[ISO88]: ISO8807. LOTOS, a formal description technique based on the ordering of observation behaviour. ISO (November 1988).

[Jar95] :Imed Jarras Vérification et synthèse d'un réseau de pétri relativement à une spécification logique temporelle, université JAVAL, juillet 1995.

[J.Eng91]: Branching processes of petri nets.Acta informatica 28(6) pp.575-591.

[J et al 18] P. Jamshidi, C. Pahl, N. C. Mendonc¸a, J. Lewis & S.Tilkov, "Microservices: The Journey So Far and Challenges Ahead," in IEEE Software, vol. 35, no. 3, pp. 24-35, May/June 2018.

[Jmes, Umon, Vsan97]:J.MESEGVER, U.MONTANARI, V.SASSONE, on the semantics of place/transition and computation 57, pp, 125-147.Mathematical Structures in computer science 7, pp.125-147

[Mil80]:R. Milner. Communication and Concurrency, volume 92 of LNCS.Springer-Verlag, 1980.

[Mil81]: Robin Milner. A modal characterisation of observable machine-behaviour.In CAAP, pages 25–34, 1981.

[Mil83]: R. Milner. Calculus for synchrony and asynchrony.TCS 25, 267—310 (1983).

[Mil89]: R. Milner. "Communication and Concurrency".Prentice Hall (1989).

[MMS97] J. Meseguer, U. Montanari & V. Sassone (1997): On the semantics of place/ transition Petri nets. Mathematical Structures in Computer Science 7, pp. 359-397.

[N15] Sam Newman, Building Microservices. O'Reilly Media Inc., 2015.

Pet62] : C. A. Petri. KommunikationMitAutomaten, Institute f urinstrumentelle mathematik, schriften des IMM, 1962

[PJ16] C. Pahl &P. Jamshidi, "Microservices: A Systematic Mapping Study" In Proc. 6th International Conference on Cloud Computing and Services Science, 2016.

[R18] C. Richardson, "Microservices Architecture". [Online]. Available: http://microservices.io/ [Accessed Oct. 21, 2018]

[Rei85(a)]: W. Reisig. Petri nets. In "EATCS Monofraphs on Theoretical Computer Science", Berlin, Heidelberg, New York, Tokyo (1985).Springer Verlag

[Rei85(b)]:W. Reisig, Petri nets : an introduction, EATCS monographs on theoretical computer science, Springer-Verlag, 1985.

[RS16] C. Richardson & F. Smith, Microservices From Design to Deployment, May 18, 2016. [E-Book] Available: https://www.nginx.com/resources/ library/designing-deploying-microservices/

[RVG13] : Rob Van Glabbeek national ITC australia and school of computer science engineerring the university of new south wales . the individual and collective interpretation of petri nets. année 2013

[Shi85a]: M. W. Shields. Concurrent machines. The Computer Journal 28(5), 449—465(1985).

[T18] J. Thones, "Microservices," in IEEE Software, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015. [Online]. Available: https://ieeexplore.ieee.org/ stamp/stamp.jsp?tp=&arnumber=7030212 [Accessed Oct. 15, 2018]

[Vidal92] : G. Vidal-Naquet et A. Choquet Geniet, R eseaux de Petri etsyst emes parall eles, Armand Colin, 1992

[XWQ16] Z. Xiao, I. Wijegunaratne & X. Qiang, ”Reflections on SOA and Microservices,” In Proc. 2016 4th International Conference on Enterprise Systems (ES), pp. 60-67.