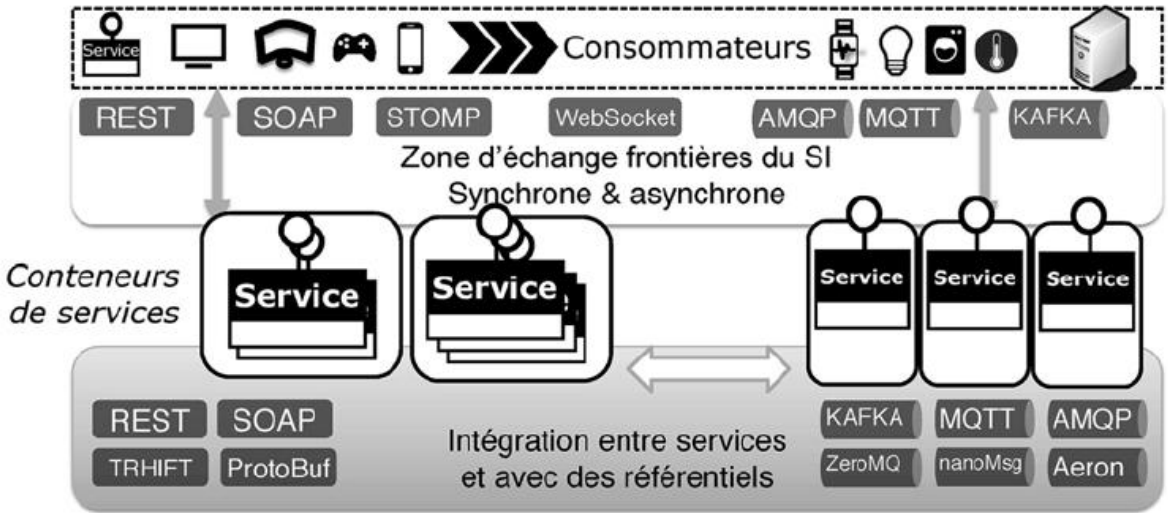
SOA,
micro services, API management
Le guide de l'architecture d'un SI AGILE
(PARTI II)

N° de la lecture individuelle :	5
Semestre	3
Étudiant	DAVID Guillaume, 803_1F
Sujet	SOA

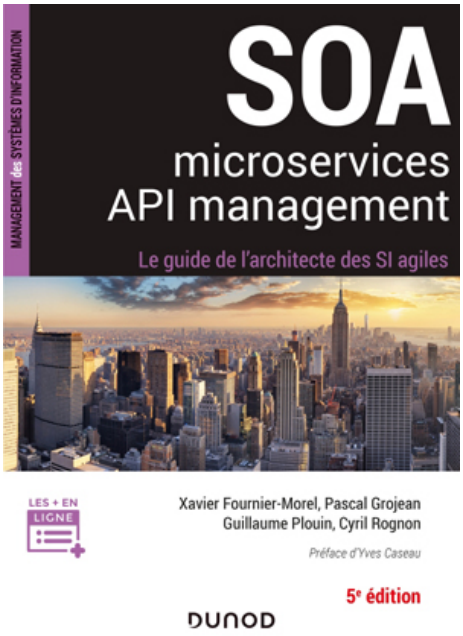


Support théorique

La recherche se base principalement sur le livre présenté ci-dessous. Des apports, de l'aide à la construction des exemples, et des compréhensions ont également réalisés avec ChatGPT.

Titre

SOA, microservices, API Management – Le guider de l'architecture d'un SI AGILE
Éditions DUNOD, 2020, ISBN : 978-2-10-080738-3 | 517 pages | 5ème édition



Auteurs

Xavier Fournier-Morel, Pascal Grojean, Guillaume Plouin et Cyril Rogon

Table des matières

PARTIE 4 – SOA et architecture d'entreprise

Cette partie vise à intégrer le style SOA dans une démarche d'architecture d'entreprise, dépassant ainsi le cadre de projets de modification d'applications « monolithiques ». Elle souligne l'importance de raisonner à l'échelle de l'entreprise, considérant des composants existants, des services SOA, la distribution sur plusieurs sites, voire dans le cloud, et la collaboration interentreprises. L'approche SOA nécessite une réflexion élargie, intégrant des éléments métier, techniques, informationnels, et organisationnels dans le cadre de l'architecture d'entreprise.

SOA et architecture d'entreprise

Les objectifs d'une architecture d'entreprise orientée SOA sont multiples :

- Définir un cadre pour la réalisation des services SOA, assurant cohérence et complétude.
- Améliorer l'alignement entre les besoins métier et les réponses techniques.
- Établir une communication efficace entre les différentes parties prenantes du système d'information.
- Mettre en place un outil d'évaluation de l'impact des demandes d'évolution des utilisateurs finaux.
- Planifier les évolutions du système d'information de manière stratégique.
- Assurer la conformité réglementaire des solutions.
- Corréler la mesure des objectifs, la priorisation des investissements avec l'usage, la monétisation, ou les incidents potentiels des services SOA.

Point de vue	Impact SOA
Architecture métier	> SOA apporte le concept de processus métier automatisés ou adaptables permettant de traiter des événements métier ou agrégés à l'échelle d'une ou plusieurs capacités métier. > SOA supporte le concept d'« entreprise étendue », en ouvrant le SI de l'entreprise aux clients, partenaires, fournisseurs, régulateurs, etc.
Architecture logique	> SOA propose une approche plus complète en distinguant processus, applications composites et services. > SOA propose une typologie des services (service métier, service technique) qui n'est pas prise en compte dans les approches classiques.
Architecture technique	> La mise en œuvre d'une approche SOA implique un choix raisonné d'outils (moteur d'orchestration, gestionnaire d'API, middleware de messages, etc.) et de standards adaptés au contexte de modélisation, de conception et d'implémentation (Archimate, WS*, BPMN, OpenAPI, etc.)..
Architecture opérationnelle	> L'apparition de processus automatisés renforce dans certains cas la nécessité d'opérer le système 24 h/24 h, 7 J/7 J. Ceci peut conduire à des problèmes organisationnels (et architecturaux) qui, sans être totalement nouveaux, peuvent apparaître ou s'exacerber dans le contexte de services et processus distribués. > La notion de niveau et qualité du service est au cœur de la valeur ajoutée de l'approche SOA. > Le déploiement des composants doit pouvoir se faire de manière plus modulaire et de plus en plus automatisée, notamment à travers le concept de conteneur, sans risque d'impacts entre de multiples services (cf. DevOps).

Point de vue	Impact SOA
Architecture physique	> SOA peut entraîner l'apparition d'un contexte multi-entreprise, voire le recours au *cloud* pour héberger des services en dehors du périmètre de responsabilité propre à l'entreprise. > L'approche SOA nécessite en conséquence une vision élargie de l'architecture physique du SI. > Cette vision élargie doit intégrer une dimension de mesure et réactivité pas forcément encore très usitée au sein des DSI.

Le point de vue « architecture métier »

Cette perspective d'architecture métier vise à définir et décrire les compétences nécessaires au fonctionnement de l'entreprise, en identifiant le cœur opérationnel des activités et les échanges avec l'extérieur. Les capacités métier, regroupées par domaines d'activité si nécessaire, permettent de mettre à disposition des consommateurs internes ou externes des fonctions métier à forte valeur ajoutée, favorisant ainsi l'économie d'échelle, l'amélioration des coûts, délais, performances, et qualité. La gestion centralisée des contrats de services métiers ou API facilite les ajustements fonctionnels et non fonctionnels en réponse aux besoins spécifiques des consommateurs, contribuant à l'optimisation globale de l'entreprise.

Le point de vue « architecture logique »

Du point de vue logique, chaque solution est généralement dédiée à une capacité métier spécifique, bien qu'elle puisse utiliser des services provenant d'autres capacités, même si cela diffère de l'approche microservices. À l'échelle du système d'information (SI), chaque solution représente une unité d'intégration composée de sous-systèmes ou modules, pouvant rester internes ou devenir des services réutilisables avec une API exposée. Chaque module ou service repose sur un socle technologique non décomposable et est idéalement aligné avec la décomposition des capacités métier. La vue systémique de la solution expose les sous-systèmes avec leurs points d'exposition, facilitant l'analyse des dépendances technologiques et des conventions de nommage pour les interfaces et composants.

Le point de vue « architecture technique »

Le point de vue technique dans l'architecture d'entreprise SOA définit les normes, standards et outillages nécessaires à la mise en œuvre des autres perspectives. Il englobe les outils socles de production, les frameworks SOA pour l'implémentation des composants logiques, les ateliers SOA pour la conception et le déploiement, ainsi que les normes et technologies assurant l'homogénéité du système d'information. Les frameworks SOA comprennent des composants génériques et des services techniques prêts à l'emploi, tandis que les ateliers SOA comprennent des outils de modélisation, d'édition de code, de tests, et d'intégration continue.

Le point de vue « architecture physique »

Le point de vue physique dans l'architecture d'entreprise SOA définit les composants matériels, logiciels et réseaux nécessaires pour exécuter les éléments de l'architecture logique. Il prend en compte la dimension géographique du déploiement, définissant les sites, matériels (serveurs, périphériques), matériels de stockage, logiciels de base (systèmes d'exploitation, SGBD), équipements réseau (WAN, LAN), centrales réseau (comme pour la VoIP dans les centres d'appels) et dispositifs de cybersécurité (protection des accès, cryptage des messages, pare-feu, etc.).

Le point de vue « architecture opérationnel »

Le point de vue opérationnel se concentre sur l'articulation entre l'organisation métier et l'exploitation du système d'information. Il considère les modes opérationnels, définissant différentes façons dont les équipes d'utilisateurs s'organisent pour assurer un fonctionnement continu du système, et les états techniques du système lorsqu'une panne ou une maintenance prolongée nécessite une reconfiguration de l'architecture physique. Les modes opérationnels peuvent être déclenchés par des contraintes métier, organisationnelles ou physiques, affectant l'architecture métier et logique. La prise en compte des états techniques implique la mise en œuvre d'un diagramme d'états pour chaque composant clé des architectures logique et physique.

PARTIE 5 – Robustesse et performance des services

Cette partie se concentre sur l'analyse de la robustesse d'une architecture orientée services (SOA) en mettant l'accent sur la disponibilité, explorant des solutions telles que la redondance des services et la réplication de l'état pour assurer une disponibilité permanente, tout en soulignant la priorité de la robustesse sur la performance.

Concevoir une SOA robuste

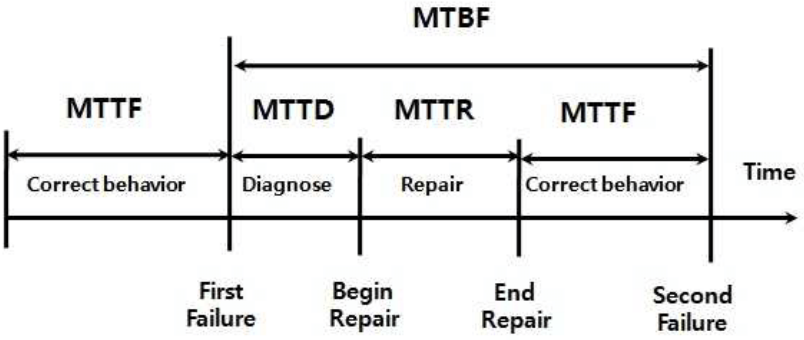
Introduction : robustesse et SOA

La robustesse d'une SOA s'exprime en termes de disponibilité.

- Le Mean Time Between Failure (MTBF) définit le temps écoulé entre deux pannes du service.
- Le Mean Time To Repair (MTTR) définit comme le temps écoulé entre la date d'arrêt du service et la date où les clients peuvent de nouveau accéder à ce service

Pour améliorer la disponibilité d'un système, il faut augmenter le MTBF (réduire les pannes) et/ou diminuer le MTTR (réduire le temps de réparation).

Disponibilité [%] = MTBF / (MTBF + MTTR) X 100



La politique de gestion de la disponibilité des services dans une architecture orientée services (SOA) est cruciale pour assurer la satisfaction des utilisateurs dans un environnement

numérique. **La solution la plus simple consiste à détecter les arrêts de service via un système de monitoring**, déclenchant manuellement le redémarrage du service par un opérateur humain. Cependant, cette approche manuelle est sujette à des retards en cas d'absence, de fatigue ou d'inexpérience de l'opérateur, affectant la disponibilité. Pour les services critiques en SOA, une politique manuelle s'avère insuffisante, notamment pour répondre aux exigences de disponibilité élevée dans des domaines tels que l'e-commerce, les jeux en ligne, les transactions financières, la cybersécurité, etc. La contrainte du « temps réel » généralisé, avec l'avènement de l'Internet des objets, rend la surveillance humaine impossible.

Afin d'atteindre une robustesse automatisée, la **redondance** des services critiques est proposée comme concept clé. La redondance implique le clonage d'un service, assurant qu'en cas de panne d'un clone actif, un autre reste opérationnel. La redondance peut être mise en œuvre au sein d'un centre de données ou entre deux centres de données géographiquement distincts, nécessitant une gestion complexe de la latence réseau.

Malgré la complexité, la redondance est essentielle pour obtenir une SOA robuste. La satisfaction de l'utilisateur dépend directement de la disponibilité du système, et adopter une approche SOA intensifie la nécessité d'analyser les exigences de robustesse et de performance. Les services critiques, même dans des systèmes monolithiques, ont historiquement investi dans la redondance pour assurer la continuité en cas de défaillance.

En conclusion, la robustesse n'est pas une option, surtout pour les services critiques en SOA, et la redondance automatisée émerge comme une solution essentielle pour assurer une disponibilité élevée dans un environnement numérique en constante évolution.

Les trois politiques de redondances

Trois politiques de redondance dans le cadre de la conception d'une architecture orientée services (SOA) permettent la mise en place structurée d’une redondance.

La politique « **actif / passif** » repose sur un clone actif traitant l'intégralité des requêtes ou événements tant qu'il est en bon état. Des clones passifs sont présents mais inactifs. En cas de panne du clone actif, l'un des clones passifs prend le relais. Bien que cette politique soit simple et assure la robustesse, elle peut être perçue comme inefficace car les clones passifs restent inutilisés.

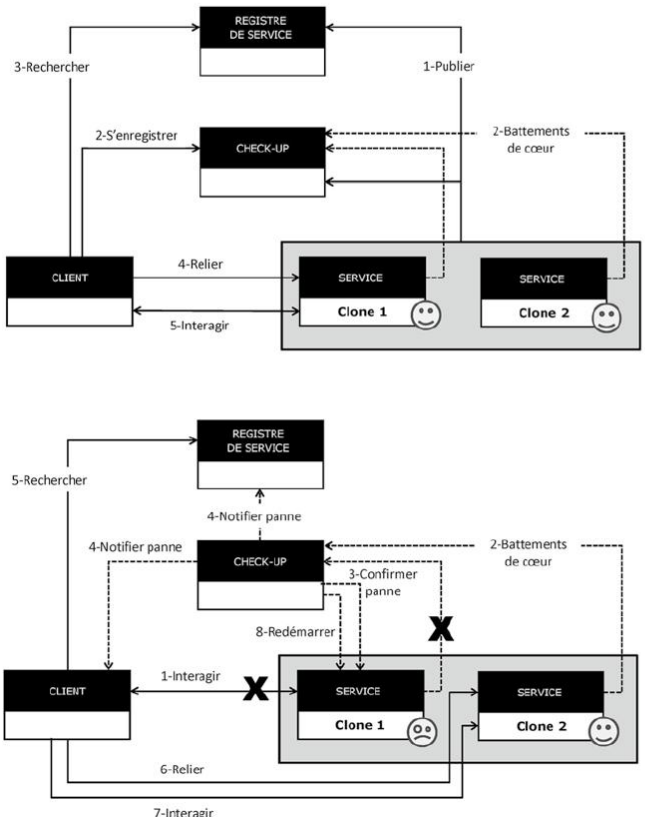
La politique « **tous au travail** » implique l'activation simultanée de tous les clones du service, chacun traitant une requête ou un événement différent. Cette approche garantit à la fois la robustesse et la performance, mais elle nécessite la mise en place d'un outil de répartition des requêtes (load balancing), devenant ainsi un point de fragilité. De plus, dans le cas d'un service "avec état", la complexité augmente car l'état du service doit être partagé entre les clones.

La troisième politique **« miroirs actifs** », impose que tous les clones soient actifs, mais ils traitent en parallèle la même requête ou événement, agissant comme des miroirs. Un seul clone est désigné pour fournir le résultat au client, et un mécanisme de vote peut être mis en place pour comparer les résultats. Bien que cette politique évite les problèmes de réplication d'état, elle nécessite une synchronisation rigoureuse entre les clones, excluant la redondance géographique entre les centres de données.

La redondance simple des services SOA (sans état)

Les clones déployés d'un même service sont enregistrés sur un « Registre de service » où un « Check-up » contrôle leur état en permanence. Lors d'une panne, le « check-up » notifie cette dernière au « Registre de service » ainsi qu'au « client » qui s'adapte afin d'utiliser le clone du service.

Le « battements de cœur » permet de notifier son fonctionnement et l'existence du « service » au « check-up ».



Cette architecture simple bénéficie d'une simplicité de compréhension et de mise en œuvre, notamment avec des services de check-up prêts à l'emploi. La principale contrainte réside dans le fait que le service check-up représente un point de fragilité du système, car en cas de défaillance de ce service, la vérification de la santé des services est compromise.

Cette solution s'adapte directement aux services "sans état" (stateless) et peut également être appliquée aux services "avec état". Cependant, dans ce dernier cas, une complexité supplémentaire survient : les clones du service redondé doivent partager le même état, sinon, le clone prenant le relais en cas de panne pourrait perdre la mémoire et ne serait pas en mesure de traiter les nouvelles requêtes ou événements.

La redondance des services à état

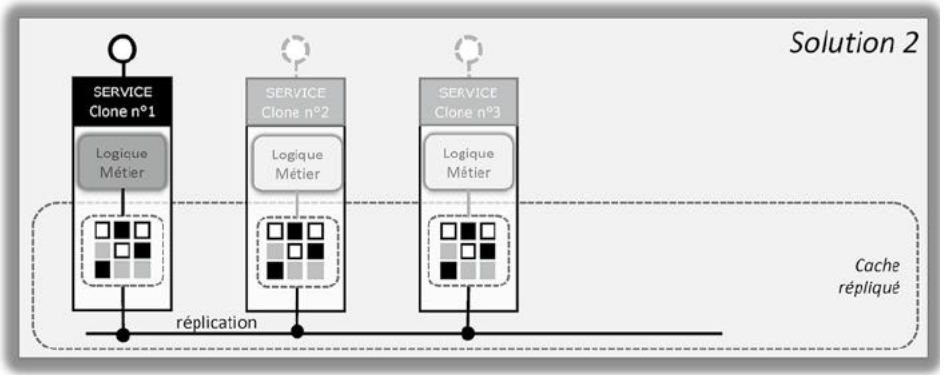
« Un service est dit stateful s'il doit conserver l'état d'un objet métier X entre deux sollicitations (requête ou événement) concernant cet objet métier. Pour réagir à cette requête

ou cet événement, le service doit accéder à l'état de cet objet X, état qu'il doit conserver lui-même, car l'application cliente (ou le service ou le système externe) qui lui envoie la requête ou l'événement ne lui fournit pas cet état »

Il existe diverses approches pour assurer la redondance d'un service avec état :

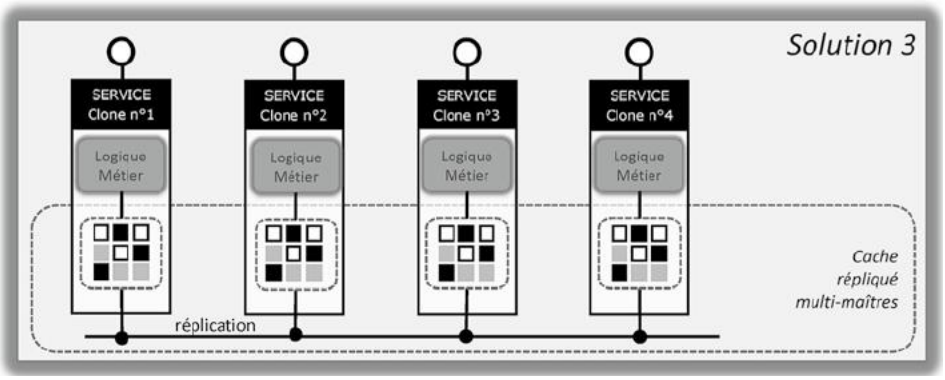
- La première solution « **rendre stateless »** implique le stockage de l'état du service métier dans une base de données SQL, transformant ainsi le service métier en un service sans état.
  Ses avantages incluent une mise en œuvre simple et la robustesse des services SOA 1.0. Cependant, cela déplace le problème vers la base de données, devenant un point de fragilité. Les inconvénients comprennent le temps de redémarrage potentiellement long en cas de panne. Des solutions émergent, notamment le clustering des bases de données SQL traditionnelles ou l'utilisation de bases NewSQL, bien que cette dernière nécessite une migration de données pour les services existants.

- Dans la deuxième solution « **cache maître esclave** », le service réplique son état entre ses clones, gérant cet état en mémoire via un cache distribué, tout en maintenant un seul clone actif. Il s'agit donc d'une stratégie de redondance simple appliquée aux services avec état.
  Cette solution repose sur l'utilisation d'un cache distribué où un seul clone actif, le "maître", lit et met à jour les informations d'état dans la mémoire du serveur hôte, avec le cache se chargeant de répliquer ces mises à jour sur les clones passifs. Le cache agit comme un espace mémoire distribué, masquant aux utilisateurs le partage de cet espace. Les avantages incluent une gestion simplifiée de l'accès aux informations, des performances élevées grâce à l'utilisation d'un cache en frontal d'une base de données, et des fonctionnalités supplémentaires telles que la persistance des données sur disque et la configuration pour des snapshots réguliers. Cependant, les inconvénients comprennent le gaspillage de ressources informatiques pour assurer la robustesse, la nécessité pour chaque cache d'avoir suffisamment de mémoire pour stocker l'ensemble des données d'état, et une limitation potentielle de la capacité mémoire. Certains outils, comme Redis, sont présentés comme des solutions types pour cette approche.

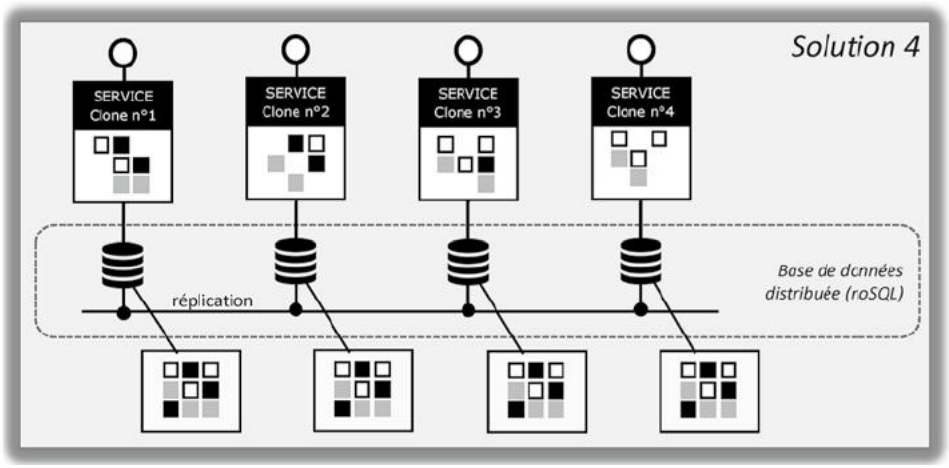


- La troisième solution consiste à répliquer l'état du service entre ses clones à l'aide d'un cache distribué **« cache multi-maîtres** », avec l'activité simultanée de tous les clones.

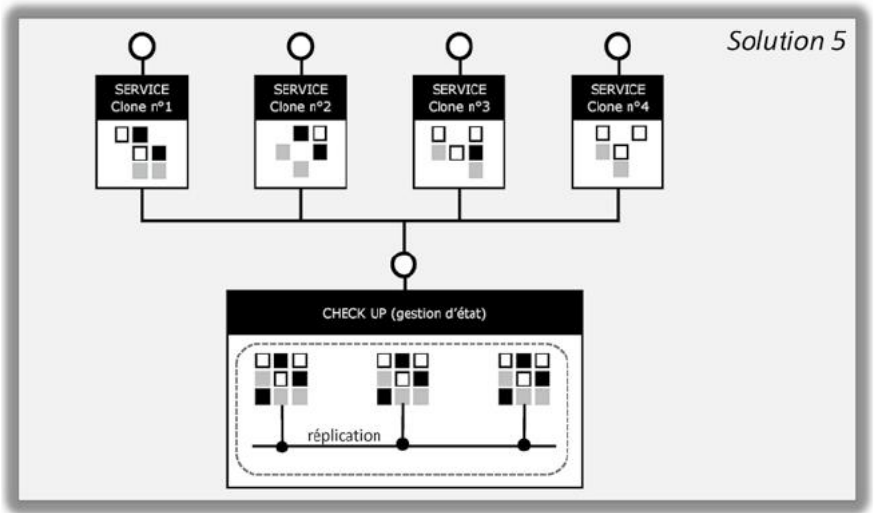
Dans cette situation, tous les clones du service sont actifs, chacun pouvant interagir avec son propre cache. Le service utilisateur communique avec le cache et ses API, la base SQL servant davantage de sauvegarde de persistance, tandis que dans la solution 4, le service utilisateur interagit directement avec l'API de la base de données distribuées, ignorant l'existence du cache sous-jacent.
Les avantages incluent l'absence de gaspillage de ressources par rapport à la solution 2, une offre d'outils variée et éprouvée telle qu'Apache Ignite, Ehcache, Hazelcast, ou NCache, et des performances techniquement compatibles avec les exigences "temps réel" des services liés à l'Internet des objets.
Cependant, les contraintes résident dans la nécessité de résoudre les problèmes traditionnels liés à ce type d'architecture, notamment en ce qui concerne la redondance locale, la redondance géographique en mode "point à point", et l'intégration du monitoring du cache distribué avec le service de check-up.



- La quatrième solution « **base de données distribuée** », implique que le service utilise un outil de gestion de base de données distribuée pour persister l'état, intégrant un protocole de réplication de données propre à la base de données.
Cette solution implique l'utilisation d'une base de données distribuée où chaque clone du service redondé est assigné à un nœud de la base de données. Chaque clone lit et met à jour son état dans le nœud qui lui est attribué, et la base de données assure la réplication de ces mises à jour sur les autres clones. Des bases telles que Cassandra, Riak, CouchBase, et Apache CouchDB, spécifiquement conçues pour les besoins du cloud et du Big Data, sont utilisées pour soutenir cette solution.
Les avantages comprennent la persistance apportée par ces bases, facilitant le redémarrage du service, et la possibilité de redondance locale et géographique entre centres de données.
Cependant, la mise en œuvre dépend du service redondé, avec des variations dans les modèles de données et la configuration de la réplication. Bien que les performances puissent être légèrement inférieures, la complexité accrue du mécanisme de réplication peut entraîner des coûts supplémentaires. L'abandon de SQL comme langage de requête peut être un frein, bien que les bases noSQL et newSQL supportent des sous-ensembles ou des langages similaires à SQL.



- La cinquième solution « **service checkup** », repose sur l'utilisation d'un service de check-up centralisé pour assurer le partage des données entre les clones du service.
  Cette solution repose sur l'utilisation du service technique de check-up pour stocker l'état du service métier. Ce service de check-up, généralement inclus dans le registre de services, est générique et indépendant des services qui l'utilisent. Il simplifie notablement l'architecture, le monitoring et le déploiement du système d'information. Sur le plan architectural, le service check-up se présente comme un service centralisé, libérant le client de toute préoccupation concernant la gestion des données.
  Cependant, cette solution a des contraintes, notamment en termes de capacité de stockage limitée par rapport à d'autres solutions. De plus, elle peut être légèrement moins performante en termes de disponibilité des données, bien que cela dépende de l'efficacité du cache local à chaque clone.



Les solutions présentées offrent des options variées pour la redondance des services "à état".

- La solution 1 convient aux services existants reposant sur une base de données SQL, avec une évaluation opportune de la migration vers une base newSQL.
- La solution 2 constitue un compromis solide entre la gestion de données complexes, le volume de données, la simplicité de mise en œuvre et les performances.

- La solution 3 optimise les ressources et améliore les performances, mais elle est plus complexe, adaptée à un contexte "temps réel".
- La solution 4 intègre la persistance et la réplication des données, idéale pour la gestion de gros volumes.
- La solution 5, orientée cloud, est intéressante si le volume de données n'est pas une contrainte majeure, offrant un déploiement "gratuit". La possibilité de répliquer les données entre serveurs et centres de données est un critère important pour le choix de l'outil, quel que soit la solution envisagée.

Concevoir une SOA performante

La performance d'une architecture orientée services (SOA) repose sur les temps de réponse individuels de chaque service, ainsi que sur la capacité globale de l'architecture à évoluer en fonction de la charge de travail de l'entreprise. La charge peut se manifester par des pics de messages, de requêtes ou d'événements, ou par une augmentation du volume d'informations à gérer. La capacité technique à faire face à ces variations est désignée sous le terme de « scalabilité" de l'architecture ».

On distingue plusieurs leviers pour augmenter la scalabilité.

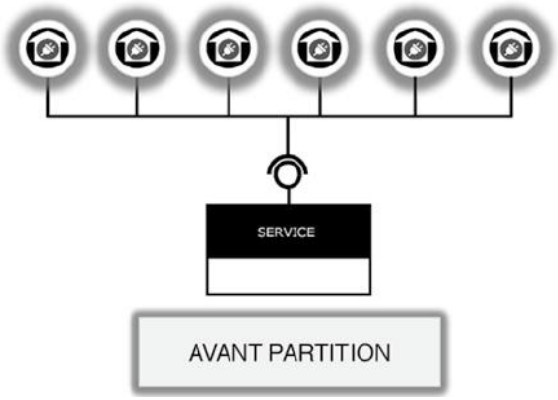
- **Scalabilité via le clonage d'un service**
  Gère les pics de charge en termes de nombre de requêtes ou d'événements en créant des clones du service.
- **Scalabilité via la répartition des données**
  Adresse l'augmentation du volume d'information et les pics de charge en répartissant les données.
- **Scalabilité via la décomposition d'un service**
  Traite les pics de charge en fragmentant un service en plusieurs services distincts.

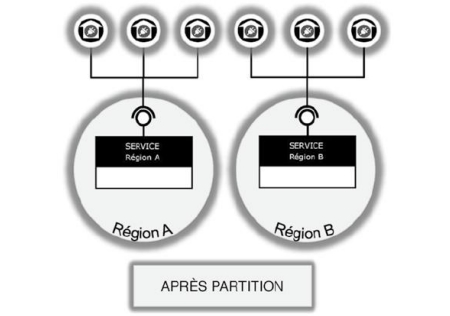
La scalabilité n'est pas une mesure absolue, mais plutôt relative au nombre de ressources nécessaires pour supporter un maximum donné. Cela inclut les serveurs, les data centers, la capacité de stockage, la bande passante réseau, les licences de base de données noSQL, etc. La décision de changer doit prendre en compte ces facteurs pour éviter des ajouts massifs de ressources sans réelle nécessité

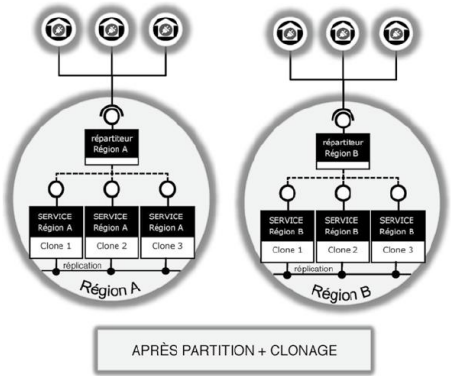
Scalabilité par les opérations

La scalabilité par les opérations **i**mplique le clonage du service pour distribuer les messages à traiter entre les clones, où chaque message est traité par un seul clone. L'ajout de nouveaux clones diminue la contrainte sur le temps de réponse unitaire du service.

Il y a deux niveaux de clonage possible : le clonage géographique du service et la duplication d'un service au sein d'un même nœud.

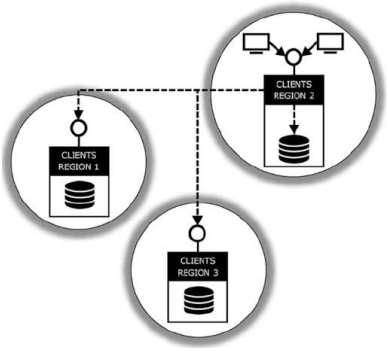






Scalabilité par les données

L'objectif est d'améliorer les performances de l'accès à une base de données volumineuse et fréquemment sollicitée via un service SOA. La « scalabilité horizontale » ou « scalabilité par les données » répartit les données sur différents serveurs ou nœuds en fonction d'une clé de répartition, telle que géographique ou alphabétique, pour éviter les goulets d'étranglement. Chaque instance du service SOA redirige les demandes vers les autres instances pour assurer l'accès complet à la base de données client



Scalabilité par les fonctions

Ce type de scalabilité repose sur l'identification des goulots d'étranglement au sein d'un service. Une fois ces traitements identifiés, on décompose le service en appliquant le pattern SEDA, émergeant ainsi un nouveau service isolant ces traitements. On améliore ensuite les performances de ce nouveau service, par exemple en l'isolant sur un serveur dédié ou en lui appliquant la scalabilité par les opérations.

PARTIE 6 – Monter en puissance sur SOA, micro-services et API ouvertes

Concevoir des API utilisables

Il existe deux approches majeures pour la gestion des API (Interfaces de Programmation d'Application)

- REST (Representational State Transfer)
- SOAP (Simple Object Access Protocol)

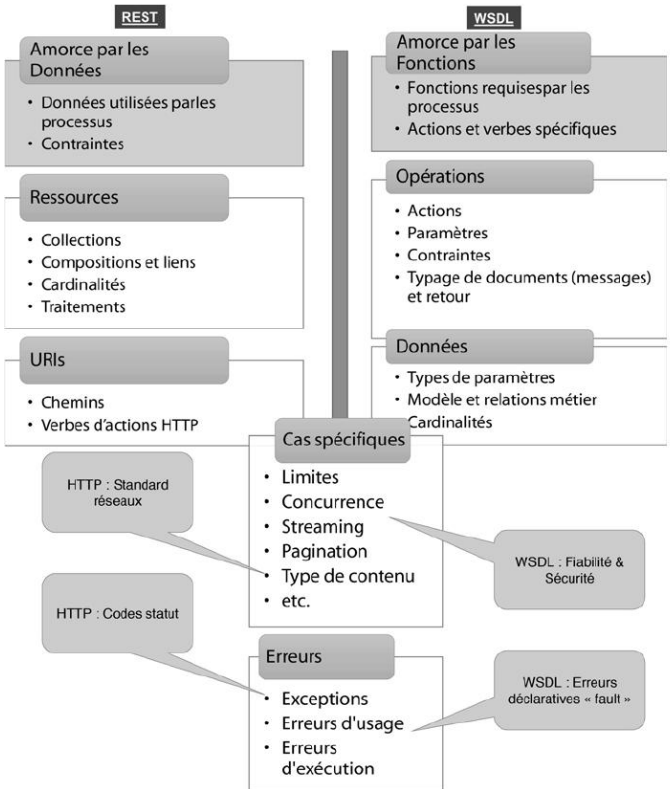
Standards d'interopérabilité :

- Les échanges synchrones sont généralement basés sur les standards HTTP.
- SOAP/WSDL (Web Services Description Language) est souvent associé aux approches structurées.
- REST est promu pour sa simplicité et est préféré par les adeptes de solutions légères.

Il n'y a pas de vainqueur clair dans le débat entre REST et SOAP mais les deux approches coexistent. REST intègre certaines contraintes de SOAP, comme un IDL (Langage de Définition d'Interface), nécessaire pour définir les interfaces de service.

REST repose sur le protocole HTTP, qui ne garantit ni la délivrance ni une sécurité avancée. Les services REST peuvent utiliser des contournements pour la fiabilité et la sécurité, par exemple, en utilisant WS-Security pour l'encryption ou la signature des messages.

Pour les expositions externes, la technologie REST est souvent choisie en raison de sa simplicité, favorisant notamment les clients mobiles.



Nommer une API et version

Dans le contexte de l'approche REST, il est préconisé d'adopter une dénomination au pluriel lorsqu'il s'agit de désigner les objets métiers ciblés par une opération de recherche, suivant le schéma ci-dessous.

```
http://adresse_du_service/Resource_Type/Resource_Id
http:// www.filrouge.com/api/consommation/clients/1x08
```

Savoir utiliser les opérations HTTP de base

Il est nécessaire de spécifier la version de l'API tel que des exemples ci-dessous.

- ✓ Directement dans l'URI :

```
http://www.filrouge.com/api/consommation/v2/clients/1x08
```

- ✓ Comme paramètre d'appel :

```
http://www.filrouge.com/api/consommation/clients/1x08?versionAPI=2
```

- ✓ En utilisant un en-tête http spécifique

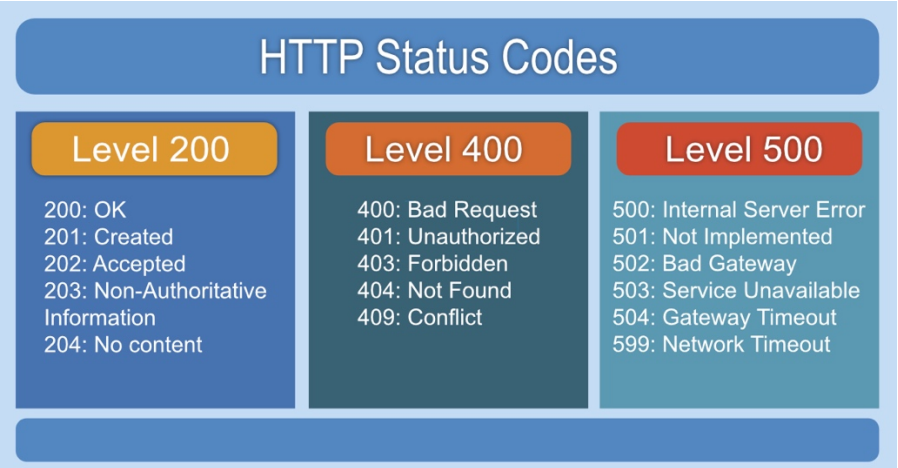
```
http://www.filrouge.com/api/consommation/clients
[en-tête]api-version: 2
```

Verbe	Opération	Sémantique
GET	Cette opération recherche un objet ou une liste d'objets	Cette opération est « sûre » : elle ne change pas l'état du système. Elle est utilisée pour les recherches et les lectures.
PUT	Cette opération met à jour une ressource si cette ressource existe. Sinon elle crée la ressource.	Cette opération est idempotente. Elle peut être utilisée pour la mise à jour.

Verbe	Opération	Sémantique
POST	Cette opération peut créer une nouvelle ressource. Chaque invocation peut créer une nouvelle ressource.	Cette opération change l'état du système. Elle n'est pas idempotente. Elle est utilisée pour la création, la mise à jour et les actions de traitement ou contrôlées.
DELETED	Cette opération détruit une ressource identifiée.	Cette opération est idempotente.
OPTIONS	Cette opération renvoie au client la liste des opérations autorisées sur un objet métier.	Cette opération est « sûre » : elle ne change pas l'état du système.
PATCH	Cette opération met à jour partiellement une ressource si cette ressource existe.	Cette opération change l'état du système et elle n'est pas idempotente. Elle est très rarement utilisée, et peu supportée par les existants.

Gestion des codes d'erreurs HTTP

Que l'on opte pour l'approche WSDL ou REST, il est impératif que le responsable de l'API définisse de manière explicite les différentes réponses envisageables pour cette API. En ce qui concerne l'approche REST, cela implique l'utilisation exhaustive des codes de réponse 20x et 40x.



Les opérations longues

Lorsqu'une opération prend du temps, le service ne doit pas bloquer l'utilisateur de l'API. Il est recommandé de répondre avec le code « 202 » et d'inclure dans cette réponse un lien (appelé callback) permettant au client de suivre l'avancement du traitement de la requête. C'est la manière appropriée de gérer, via REST, un traitement asynchrone.

Une alternative consiste à utiliser STOMP (Simple Text-Oriented Messaging Protocol), qui équivaut à HTTP pour les opérations asynchrones. STOMP offre une approche orientée

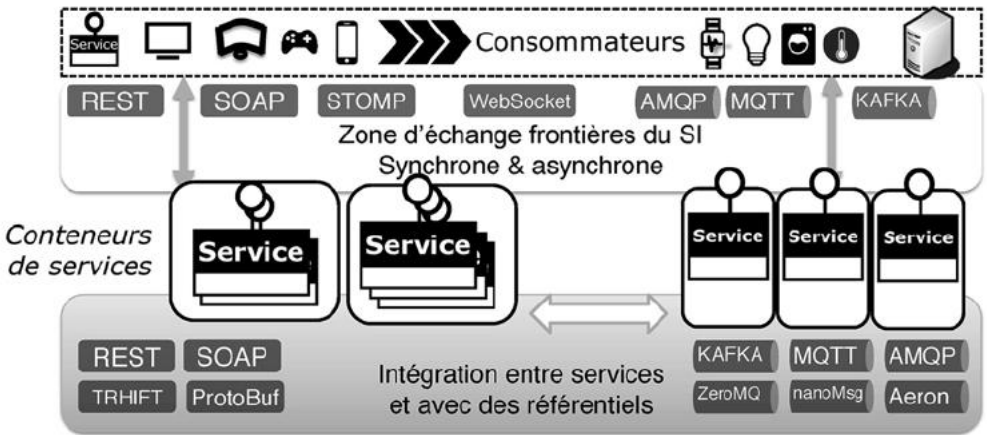
message plus efficace. Cependant, il est crucial de vérifier sa disponibilité si le client du service est une application au sein d'un navigateur.

Mettre en œuvre une architecture SOA

L'approche SOA nécessite la mise en œuvre d'un ensemble de technologies pour faciliter les communications via des middlewares, tant pour les appels synchrones que pour les appels asynchrones. Les gestionnaires d'API, les ESB, les MOM et les passerelles de services qui respectent les standards contribuent à favoriser la diversité technologique, un aspect apprécié dans l'approche des microservices. Cela ne suggère pas une multiplication obligatoire des technologies, mais souligne que l'objectif du système d'information n'est pas de privilégier l'hégémonie d'une technologie particulière. Il reste toujours possible de choisir une technologie de référence pour des considérations de main-d'œuvre et de coût de formation. Les organisations qui ont déjà adopté une approche SOA ont généralement fait ce choix et doivent gérer un portefeuille de services existants. Dans ce contexte, il est plus crucial de standardiser les outils et les normes d'échange que le langage de développement des composants et des services.

Les grands projets SOA qui optent pour la mise en place d'une plate-forme complète et complexe en une seule étape "Big Bang" sont risqués. La recommandation est plutôt d'enrichir progressivement une plate-forme de base, en suivant les priorités suivantes :

1. **Ouvrir le SI pour de nouveaux clients ou marchés**
   Monétiser ces échanges en offrant une visibilité et un accès simple et sécurisé à ses clients.
2. **Intégrer les services**
   Adapter les protocoles, formats divers, modes de transport, encodage, et synchronicités/asynchronicités, ainsi que le contenu métier des échanges.
3. **Couvrir les exigences de robustesse et de disponibilité**
   Gérer des événements très nombreux et remédier à la fiabilité limitée du réseau.
4. **Enrichir la gouvernance en place avec les services et la plate-forme SOA**



Déploiement des services

Le concept de conteneur a émergé comme une solution polyvalente pour le déploiement d'architectures de services et d'applications composites. Un conteneur, qui contient au moins une application ou un service, offre un environnement d'exécution indépendant de la machine hôte, garantissant une portabilité sans heurts entre différents environnements de déploiement. Contrairement aux machines virtuelles, plusieurs conteneurs peuvent s'exécuter simultanément sur une même machine hôte, offrant une isolation tout en partageant les ressources de l'OS.

L'histoire des conteneurs remonte à Java avec ses paquets .jar, .war et .ear, mais l'émergence des méthodologies DevOps a renforcé la nécessité d'une technologie de déploiement plus légère et flexible. Les conteneurs écuméniques ont gagné en popularité, permettant aux développeurs et aux équipes de production d'utiliser les mêmes versions exécutables de services à toutes les étapes du cycle de vie. Ces technologies facilitent le déploiement de services autonomes, s'alignant avec les principes de l'architecture orientée services (SOA).

Les micro-conteneurs ont émergé pour isoler efficacement les instances de service, démarrer instantanément et consommer moins de ressources. L'objectif est de proposer un format de paquetage unique, indépendant de la technologie des composants et de la cible hôte. Cette approche répond à la convergence des architectures de services distribuées, des méthodologies DevOps et de l'hébergement dans le cloud.

Pour SOA, le conteneur devient une unité de paquetage, de test, de déploiement, de gestion, d'exploitation et d'allocation de ressources. Cette approche simplifie l'organisation des développements, facilite les tests indépendants, permet des mises à jour et des rollbacks instantanés, homogénéise la gestion et l'exploitation, et s'aligne avec les besoins de performance grâce à des systèmes d'allocation de ressources comme Mesos ou Kubernetes.

La réussite de Docker depuis 2014 souligne l'impact de cette approche, notamment dans le contexte des architectures de services.

SOA et conteneur

Le conteneur se concentre sur les points d'entrée et de sortie permettant les interactions externes avec le service, tandis que le framework microservices se positionne davantage du côté de la conception et du développement des services. Bien que ces concepts soient étroitement liés, ils opèrent à des niveaux de granularité différents.

L'utilisation d'outils d'orchestration tels que Kubernetes ou Swarm, tout en simplifiant le déploiement de services en conteneurs, introduit des défis liés à la gestion des dépendances entre services. Les services contenus dans des conteneurs peuvent être déployés et redéployés dynamiquement, nécessitant un service de registre robuste pour gérer ces dépendances.

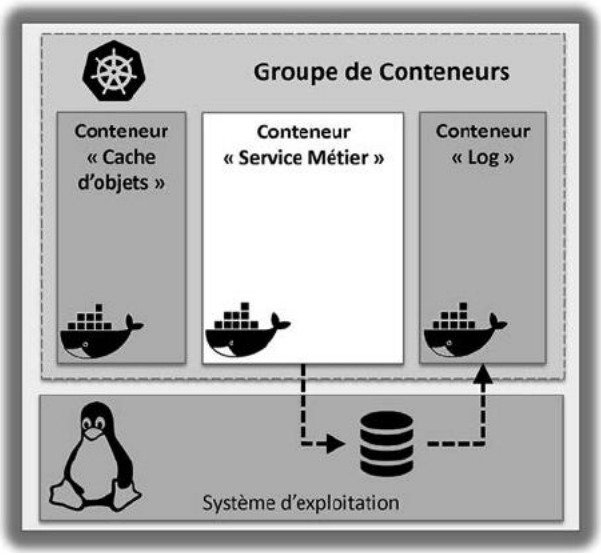
L'approche microservices préconise que chaque conteneur Docker contienne un seul processus élémentaire. Cela peut entraîner la découpe d'un service stateful utilisant une base de données en deux conteneurs distincts, l'un pour la logique métier et l'autre pour la base de données. Cependant, cette approche doit être utilisée avec prudence, et des considérations

plus générales sur l'architecture doivent être prises en compte. Actuellement, certaines limitations existent, notamment l'incapacité de Docker à gérer nativement la migration d'un conteneur de base de données d'un hôte à un autre.

Certains frameworks microservices, tels que WSF4J de WSO2, offrent plusieurs options de paquetage pour les services développés en Java, comme la création d'un jar autonome, d'un bundle OSGi ou directement d'une image Docker. Ces approches offrent des avantages en termes de gestion des dépendances, de performances et de facilité de déploiement, contribuant ainsi à l'évolution des pratiques de développement et de déploiement dans le contexte de SOA.

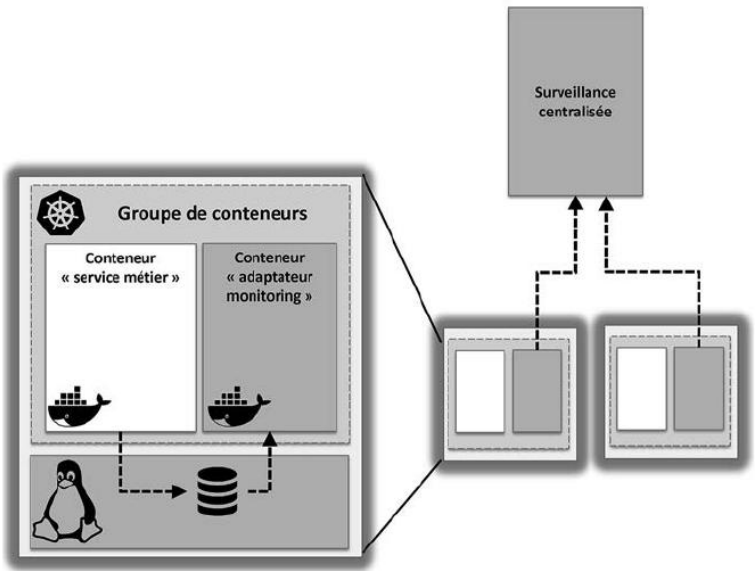
Pattern « side-car »

Le pattern side-car dans l'architecture orientée services (SOA) avec Docker consiste à encapsuler des fonctionnalités complémentaires ou des services auxiliaires dans des conteneurs distincts, agissant en tandem avec le conteneur principal du service. Ces « side-cars » fournissent des services additionnels tels que la gestion des dépendances, la sécurité, la surveillance, ou d'autres fonctionnalités spécifiques, sans altérer le code du service principal. Ce modèle offre une flexibilité et une modularité accrues dans le déploiement et la gestion des services au sein d'une architecture SOA.



Pattern « adapter »

Le pattern side-car dans l'architecture orientée services (SOA) avec Docker implique l'utilisation de conteneurs supplémentaires, appelés « side-cars », pour intégrer des fonctionnalités complémentaires aux services principaux sans altérer leur code. Ces side-cars peuvent fournir divers services tels que la gestion des dépendances, la sécurité ou la surveillance. Cette approche modulaire améliore la flexibilité et la gestion des services au sein de l'architecture SOA en permettant l'ajout ou la mise à jour indépendante des fonctionnalités associées à chaque service. En utilisant Docker, ces side-cars peuvent être déployés de manière efficace et isolée, offrant ainsi une solution souple et évolutive pour le développement et la maintenance des services dans un environnement SOA

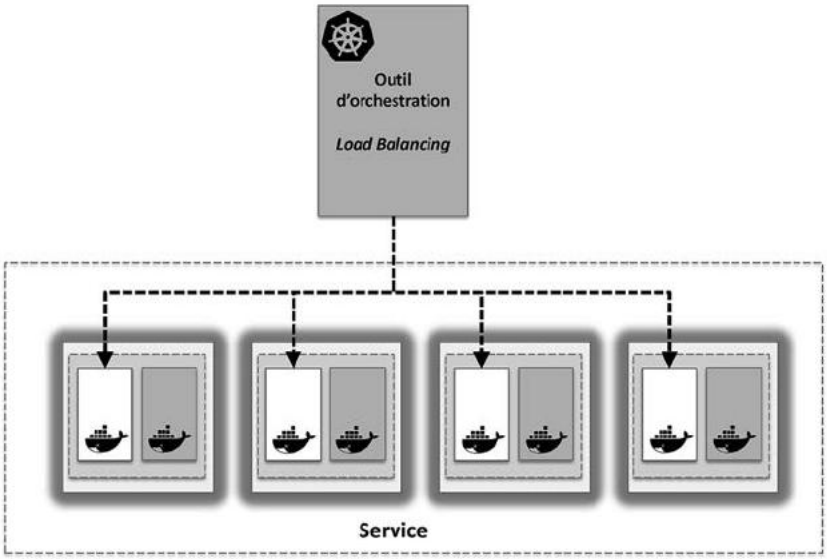


Pattern multi-nœud « load-balancing »

Le pattern multi-nœud « load-balancing » consiste à répartir équitablement la charge de travail sur plusieurs nœuds dans un système. Dans un contexte SOA (Service-Oriented Architecture) avec Docker, cela peut être réalisé en déployant plusieurs instances d'un même service sur différents nœuds, puis en utilisant un mécanisme de « load-balancer » pour distribuer les requêtes entre ces instances.

Avec Docker, chaque instance du service peut être encapsulée dans un conteneur, facilitant ainsi le déploiement, la gestion, et la mise à l'échelle des services. Le « load-balancer » peut être configuré pour répartir les requêtes entre ces conteneurs en fonction de différents algorithmes (round-robin, least connections, etc.).

Ce pattern améliore la disponibilité, la résilience et les performances du système en répartissant la charge entre plusieurs nœuds, assurant ainsi une utilisation plus efficace des ressources disponibles. En cas de montée en charge ou de défaillance d'un nœud, le « load-balancer » peut réajuster automatiquement la distribution des requêtes pour maintenir les performances globales du système.

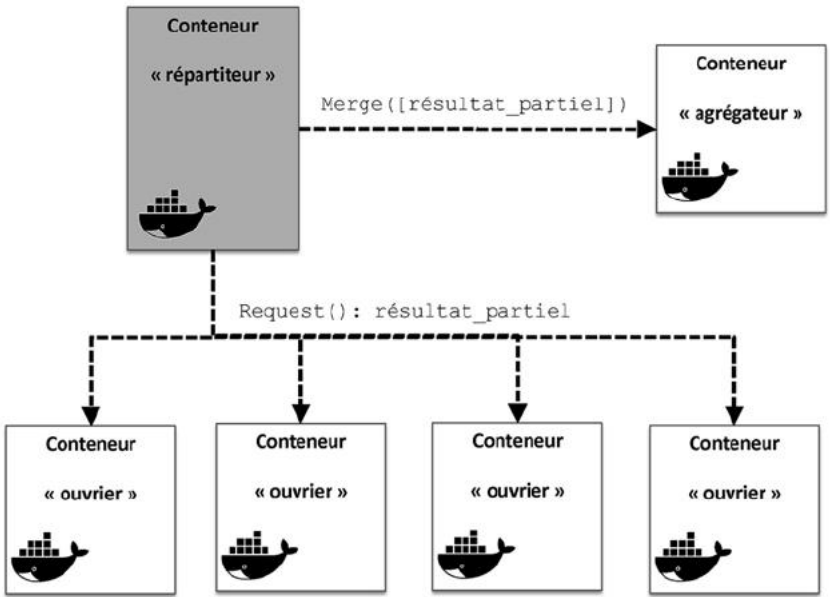


Pattern multi-nœud « scatter-gather »

Le pattern multi-noeud « scatter-gather » est une approche dans laquelle une requête initiale est diffusée (scatter) à plusieurs services distribués, puis les résultats de ces services sont agrégés (gather) pour former une réponse complète. Dans un contexte SOA (Service-Oriented Architecture), ce pattern est souvent utilisé pour paralléliser le traitement de requêtes complexes impliquant plusieurs services.

Avec Docker, chaque service peut être encapsulé dans un conteneur, facilitant ainsi le déploiement et la gestion de ces services distribués. L'idée est de distribuer une requête à plusieurs instances de services, potentiellement sur différents nœuds, pour exploiter au maximum les ressources disponibles et accélérer le traitement.

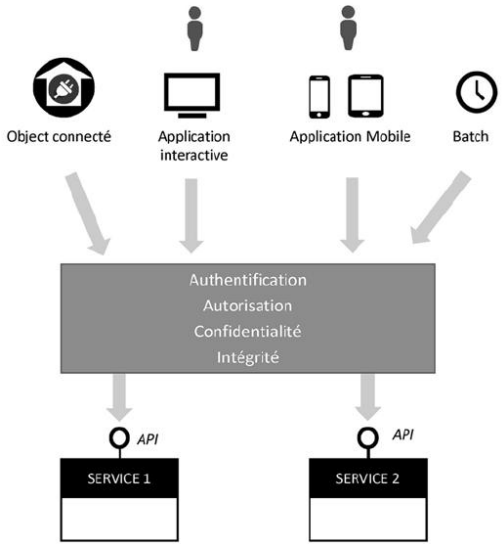
Le processus de « scatter » peut être géré par un composant central ou un orchestrateur qui envoie la requête à plusieurs services. Une fois que chaque service a traité sa partie de la requête, les résultats sont agrégés lors du processus de « gather » pour former une réponse complète qui est renvoyée à l'utilisateur.
Ce pattern est particulièrement utile pour les tâches qui peuvent être décomposées en sous-tâches indépendantes et qui peuvent être traitées de manière concurrente. Il améliore les performances globales du système en exploitant la parallélisation tout en maintenant la modularité des services individuels.



Sécuriser les services

Les spécialistes en sécurité se concentrent sur cinq fondamentaux pour analyser les besoins de sécurité d'une plate-forme SOA :

1. **Authentification :** Vérification de l'identité de l'utilisateur avant d'accorder l'accès au système ou à l'application.
2. **Confidentialité :** Prévention de la lecture d'informations par des personnes non autorisées.
3. **Intégrité :** Garantie technique de la non-altération des informations d'origine, qu'elle soit accidentelle ou malveillante.
4. **Disponibilité :** Assurance du bon fonctionnement d'une application, sa résistance aux pannes et aux attaques incapacitantes.
5. **Traçabilité :** Stockage des traces de toutes les interactions des utilisateurs avec les applications pour détecter des attaques ou des dysfonctionnements.



Dans le contexte des architectures orientées services (SOA), plusieurs mécanismes de sécurité sont utilisés :

1. **Fédération d'identité :** Utilisation de protocoles tels qu'OpenID Connect et SAML pour permettre la fédération des identités, facilitant l'authentification unique (SSO) entre différents systèmes.
2. **Autorisations :** Mise en œuvre de mécanismes tels qu'OAuth pour gérer les autorisations et l'accès aux ressources, ainsi que XACML pour la gestion fine des politiques d'autorisation.
3. **Provisioning :** Utilisation de standards tels que SPML (Service Provisioning Markup Language) et SCIM (System for Cross-domain Identity Management) pour automatiser et simplifier la gestion des identités et la synchronisation des informations utilisateur.
4. **Cryptographie en environnement SOAP :** Intégration de techniques de cryptographie spécifiques à l'environnement SOAP, notamment l'utilisation d'XML-encryption et XML-signature, ainsi que WS-Security pour renforcer la sécurité des échanges de messages.
5. **HTTPS :** Utilisation du protocole HTTPS (HTTP sécurisé) pour assurer la confidentialité, l'intégrité et l'authentification des communications au niveau du transport.

Ces mécanismes combinés contribuent à renforcer la sécurité des services web et à assurer la protection des données et des identités dans le contexte des architectures orientées services.

Bibliographie

A-schematic-diagram-of-MTTF-MTTR-and-MTBF_fig5_334205633. (s.d.). Récupéré sur https://www.researchgate.net/: https://www.researchgate.net/figure/A-schematic-diagram-of-MTTF-MTTR-and-MTBF_fig5_334205633

http-status-codes/. (2023, 12 27). Récupéré sur https://restfulapi.net: https://restfulapi.net/http-status-codes/

**Les images non référencées dans la bibliographie proviennent du livre étudié.*