https://blog.xebia.fr/2015/03/02/microservices-les-concepts/

**Microservices – Les concepts**

Microservices. C’est une architecture dont on entend beaucoup parler, mais que se cache-t-il derrière ce terme ?

Avec une série de trois articles, nous allons tenter de découvrir ce qu’est une architecture microservices et ce qu’elle change par rapport à une architecture « classique ».

* Ce premier article s’intéressera tout d’abord aux concepts de ces architectures
* Le deuxième article illustrera ensuite [des exemples possibles d’architectures microservices](http://blog.xebia.fr/2015/03/09/microservices-des-architectures/)
* Le troisième article s’attachera, quant à lui, [aux pièges de ces architectures et aux pistes qui permettent de les éviter](http://blog.xebia.fr/2015/03/16/microservices-des-pieges/)

Un SI est un système complexe qui se développe, évolue, vit et meurt. A la manière d’un être vivant, un SI est un assemblage de plus petites unités qui fonctionnent de concert et qui se répartissent des fonctions clés au sein du système. De nombreux styles d’architectures permettent de réaliser de tels systèmes.

Nous proposons, dans cet article, de découvrir les architectures microservices et plus précisément les concepts qui s’y rattachent. On tâchera de comprendre ce qui se cache derrière cette dénomination qui, au premier abord, doit se comprendre comme ceci : un système complexe de petits services simples. Nous verrons également comment ce style d’architecture répond aux différents besoins du SI.

Nous avons évoqué un assemblage de petites unités au sein d’un système complexe. Ces petites unités, vous l’aurez compris, constituent lesdits services d’une architecture microservices. Quelles sont les concepts qui se cachent derrière ces services ?

**Des unités fonctionnelles du système**

Un microservice est avant tout une unité fonctionnelle. Il correspond à une fonctionnalité précise, logique et cohérente du système. Un microservice est donc autonome vis à vis de la fonctionnalité qu’il réalise. Il a son propre code, gère ses propres données et ne les partage pas – pas directement en tout cas – avec d’autres services.

Dans ce style architectural un service correspond, de plus, à un processus système indépendant. Dans certains cas, un microservice peut être constitué de plusieurs processus mais l’inverse n’est pas vrai. Une conséquence de ce constat est que les services communiquent entre eux par des appels réseaux et non par des appels de fonctions en interne dans un processus.

Un microservice est donc une unité de service fonctionnelle qui se développe, se déploie, s’exécute et gère ses données indépendamment des autres services du système. Quels sont les bénéfices de ce découpage en services ?

**La modifiabilité avant tout**

Lorsqu’une fonctionnalité précise du SI doit être modifiée, l’évolution en question ne porte que sur le code d’un service en particulier. Cela implique que les autres services ne sont pas directement impactés. Ainsi, la modification du code ou des données d’un service provoque uniquement la mise à jour et le redéploiement d’un service car celui-ci s’exécute dans un processus isolé. Il n’y a, de plus, pas d’obligation d’embarquer des mises à jour d’autres fonctionnalités qui auraient été intégrées en parallèle à l’évolution de la fonctionnalité, car le code est séparé. De la même façon, la présence d’une anomalie sur une fonctionnalité ne bloque pas l’évolution ou le déploiement d’une autre fonctionnalité.

On peut, grâce à ces caractéristiques, facilement envisager de prototyper, déployer et tester une nouvelle fonctionnalité sans remettre en cause tout le système. Le système est comme un assemblage de pièces de LEGO. Une brique ou une autre peut être ajoutée ou retirée facilement.

Le système est construit à une échelle qui permet de modifier une fonction du système sans avoir peur de créer des effets de bord sur d’autres fonctions. Dans cette architecture, l’échelle de la modification est donc, non pas un ensemble de fonctionnalités assemblées dans une “application”, mais la fonctionnalité, portée par le service. Cette échelle de la modification confère d’autres avantages.

**Une architecture à l’échelle du service**

S’agissant de processus isolés, l’augmentation ou la diminution des ressources allouées à un service ou un ensemble particulier de services peut se faire de manière indépendante. On peut choisir individuellement les services dont on veut modifier le nombre d’instances. L’unité de mise à l’échelle est bien réduite au service et non pas à une “application” ou un ensemble technologique de fonctions.

Chaque service est différent et sert des besoins différents. Chaque service a donc son propre cycle de vie et in fine une durée de vie qui lui est propre. Là encore la granularité de déploiement et d’exécution d’un service permet de contrôler le cycle de vie d’une fonction de manière indépendante des autres fonctions du système. Un service peut ainsi être suspendu ou remis en fonction indépendamment.

Selon le même principe, les données d’un service peuvent aussi être mises à jour, migrées, traitées sans risquer d’impacter les autres fonctionnalités du système. Dans une architecture microservices chaque service est différent et évolue donc selon son propre rythme.

Les services sont donc autonomes mais fonctionnent tout de même de concert entre eux, car ils forment un système fonctionnel et cohérent dans sa globalité. Cela est possible par une communication à base de messages.

**Des services et des messages**

Une architecture microservices est avant tout un système de services qui communiquent entre eux. Comme on vient de l’évoquer, chaque service réalise une fonction précise dans un système. Chaque fonction a néanmoins besoin d’être déclenchée, régulée ou inhibée par d’autres fonctions. Pour y parvenir les services échangent, par le biais d’interfaces qui leurs sont propres, des messages entre eux.

Je ne suis que le messager

Dans ce style architectural, le vecteur de la communication se contente de transporter les messages entre les services. Il n’y a pas de transformation, agrégation ou enrichissement des messages. Autrement dit, il n’y a pas “d’intelligence” dans le vecteur. Cette intelligence, si nécessaire, est bien sûr portée par les services qui réalisent les fonctions. Le vecteur ne sert que de tuyau et est “aveugle” vis-à-vis du contenu des messages. Il peut par conséquent être quelconque mais est avant tout focalisé sur son rôle premier : véhiculer des messages de manière fiable.

Deux modes de communication sont couramment employés dans ce style d’architecture. On retrouve, sans surprise, une communication de type REST qui s’appuie généralement sur le protocole HTTP et une communication de type BUS qui centralise les échanges de messages. L’utilisation de ces deux modes de communication dans une architecture microservices sera étudiée dans le prochain article.

Le contenu des messages n’est pas pollué par une multiplication d’enveloppes techniques. Les messages véhiculent une information pertinente destinée à la réalisation des fonctions des services. Les mises à jour d’interfaces sont en ce sens plus souples, car elles portent sur une modification de contenu imposée par la modification d’un service et non sur une modification liée aux enveloppes techniques.

Dans le cas général, un service consomme en entrée des informations d’autres services et produit en sortie des informations qu’il communique à son tour à d’autres services. La communication d’un service en amont peut être rendue fortement dépendante d’un nombre important de services en aval. La communication est donc prioritairement asynchrone. Chaque service pouvant poster un message et écouter des messages d’autres services au fur et à mesure de leur disponibilité. Le synchronisme de communication entre services augmente les sources de pannes et de lenteurs, qui se répercutent directement sur les services en amont.

Une architecture microservices tend donc plutôt vers une architecture réactive, à base d’évènements, déclenchés et écoutés par les services. Pour que ces échanges entre services puissent s‘établir, ils doivent toutefois démontrer une certaine tolérance.

**Des services tolérants**

Si une catastrophe peut arriver, elle finira obligatoirement par arriver

Des services seront indisponibles, des données seront manquantes, peut-être même incohérentes. Des services peuvent être volontairement éteints, remplacés ou mis à jour. Tous ces aléas font la vie d’un SI. Plutôt que de chercher à se protéger, par le rejet, de ces problèmes qui se produiront invariablement, l’architecture microservices privilégie plutôt la tolérance et l’adaptation.

Chaque service est conçu au sein d’un système avec ses interactions. Il est également conçu en étant conscient que les services dont il dépend ou qu’il influence peuvent être indisponibles. Un service adapte donc son fonctionnement en fonction de l’état du système dans lequel il évolue. Cela se traduit par exemple par une remontée d’alertes, un fonctionnement en mode dégradé ou encore par une reprise de fonctionnement lors du retour à une situation nominale.

Cette acceptation de la panne va au-delà des dépendances entre services. Chaque service doit être conçu pour tolérer par exemple une donnée manquante ou incohérente. Cela suppose une certaine souplesse des interfaces. Un service doit tolérer, par exemple, des messages même s’ils comportent des données dont il n’a pas besoin. D’un autre côté un service doit pouvoir fonctionner dès lors qu’un message comporte les données suffisantes à son fonctionnement et tolérer l’absence d’informations optionnelles. Dès lors qu’une rupture du contrat envers un client est obligatoire pour l’évolution d’un service, on peut par exemple, envisager la coexistence de versions différentes du service pendant la période de migration des services consommateurs.

Les services sont donc tolérants vis-à-vis de leurs échanges en messages. Cette tolérance implique tout de même une certaine surveillance du système.

**Un système sous surveillance**

Le système doit être, en effet, en permanence surveillé. Même si les services sont tolérants à des pannes, celles-ci peuvent compromettre le fonctionnement du système global. Il est donc important d’être capable de détecter rapidement une panne dans le système pour pouvoir intervenir rapidement : remettre en route, corriger une défaillance, redémarrer un service sur une autre machine, augmenter le nombre de ressources en cas d’un pic de trafic, de lenteur ou de saturation d’un service. Il s’agit là d’une surveillance technique du bon fonctionnement du système.

Il est également vital de surveiller le système et les services d’un point de vue fonctionnel. Le bon fonctionnement du système passe par la surveillance d’indicateurs métiers : transactions validées, en erreur, quantité de commandes, envois de messages en erreur, nombre d’inscriptions, etc. Tous ces indicateurs métiers sont les premiers révélateurs d’un problème. Ils peuvent également aider à distinguer les fonctions du système les plus sollicitées de celles qui sont inutilisées.

Dans une architecture microservices la surveillance est d’autant plus importante que les services sont nombreux et les communications complexes. Cette complexité nécessite et reflète bien sûr une organisation, humaine notamment, à l’origine de ces architectures.

**Des équipes centrées autour d’un objectif**

Une équipe qui sait ce qu’elle fait et pourquoi, le fait bien.

Tout comme un service se concentre sur la réalisation d’une fonction, les équipes s’organisent autour de fonctionnalités métier et non pas de technologies. Les équipes sont multidisciplinaires et se complètent pour la réalisation d’un objectif commun : faire fonctionner un système.

You build it, you run it

La surveillance que l’on évoquait précédemment fait partie intégrante du rôle de cette équipe. Il ne s’agit pas d’un rôle porté par une équipe distincte et dédiée. Le service est donc avant tout un produit, issu d’une équipe, qu’elle s’approprie, réalise et accompagne durant toutes les étapes de la vie du produit.

L’appropriation du produit passe également par la taille des équipes. Un service est d’autant plus facile à s’approprier qu’il est petit, par définition, mais aussi que le nombre de personnes qui en sont responsable est petit.

De la même manière que les équipes sont diversifiées dans leurs rôles, leurs objectifs et leurs compétences, les services et les technologies dont ils sont composés le sont aussi.

**Des services diversifiés**

Une équipe qui utilise des outils adaptés, le fait mieux.

Nous l’avons déjà vu, chaque service répond à un problème particulier. Il n’y a donc a priori pas de raison d’adresser des problématiques différentes avec une seule et même pile technologique.

Chaque technologie ou outil est plus adapté à résoudre un certain type de problème. Avoir la possibilité d’utiliser un outil adapté à la résolution d’un problème est donc un facteur de succès pour la mise en place d’un service. Par ailleurs, l’isolation des services permet de basculer facilement d’une technologie à une autre.

Cela va également dans le sens d’équipes responsabilisées qui choisissent les outils qu’elles utilisent et qu’elles sont le plus à même de juger pertinents.

Un facteur de succès qui se révèle également important après cet aspect sur la diversification est bien sûr la qualité.

**Des procédures automatisées, des services de qualité**

On vient de voir que chaque service est façonné avec des outils adaptés et donc diversifiés.

Cela implique dans un premier temps que la qualité des services doit être mesurée en permanence : tests automatisés et analyse du code – intégration continue. Par ailleurs l’aspect “système complexe” de cette architecture, implique que les tests d’intégration et les tests globaux, sont aussi importants que les tests des services isolés. Ils ont donc tout autant vocation à être automatisés.

Cela implique ensuite, pour la viabilité du système en production, que les déploiements sont au maximum automatisés – déploiement continu. Les procédures doivent être reproductibles, répétables et doivent contenir le moins de sources d’erreurs possible – intervention manuelle notamment – car elles seront potentiellement jouées un grand nombre de fois.

**Conclusion**

La seule chose qui ne change pas, c’est le changement

Les SI changent. Certaines fonctionnalités sont volatiles, d’autres persistantes, d’autres encore sont testées puis abandonnées et quelques unes sont fréquemment modifiées. En d’autres termes le SI réagit en permanence à des besoins changeants et ces changements sont de plus en plus fréquents. L’architecture mise en place est donc fortement contrainte sur sa dimension “modifiabilité”.

Nous venons de voir que les architectures microservices adressent directement cette problématique de la modifiabilité. L’échelle de l’architecture est la fonctionnalité, le service. Cela rend possible une dynamisation de l’architecture à l’échelle de la fonctionnalité. Les fonctionnalités sont en effet isolées les unes des autres au sein du système.

Cette isolation à des revers. Elle implique notamment une communication plus pertinente, organisée autour de messages, mais aussi plus coûteuse. Pour la viabilité du système dans sa globalité, les services doivent être tolérants à des défauts d’autres services et de leurs messages. Le système doit être également surveillé, aussi bien sur des aspects techniques que fonctionnels.

Les organisations à l’origine de ces architectures doivent également refléter le système en lui-même (i.e. [la loi de Conway](http://en.wikipedia.org/wiki/Conway%27s_law)). Les équipes et les services sont diversifiés et organisés avant tout autour des fonctionnalités du système et pas des technologies.

Un second revers de cette organisation décentralisée concerne la complexité de la compréhension du système dans sa globalité. On peut certes comprendre le fonctionnement d’un service isolé, mais il faut aussi et surtout être capable d’intégrer son fonctionnement dans l’ensemble du système.

Une difficulté se situe aussi dans la définition même des services et des fonctions qu’ils réalisent. Cette définition n’émerge pas nécessairement immédiatement et peut faire l’objet d’ajustements progressifs.

De nouvelles possibilités sont donc offertes par cette architecture à l’échelle de la fonctionnalité. Mais des pièges, à éviter, émergent tout de même des quelques concepts que l’on vient de dégager. Ces pièges doivent bien sûr être spécifiquement adressés dans le cadre d’une architecture microservices.

Nous verrons, dans le prochain article de cette série, des exemples possibles d’architectures microservices. Les pièges de ces architectures, feront quant à eux, l’objet du dernier article.

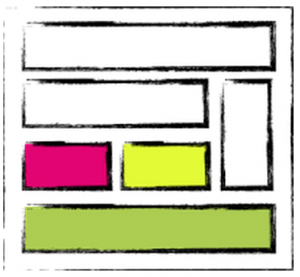
**Microservices – Des architectures**

Microservices. C’est une architecture dont on entend beaucoup parler, mais que se cache-t-il derrière ce terme ?

Avec une série de trois articles, nous allons tenter de découvrir ce qu’est une architecture microservices et ce qu’elle change par rapport à une architecture « classique ».

* Le premier article s’intéressait tout d’abord aux [concepts de ces architectures](http://blog.xebia.fr/2015/03/02/microservices-les-concepts/)
* **Ce deuxième article illustrera ensuite des exemples possibles d’architectures microservices**
* Le troisième article s’attachera, quant à lui, aux pièges de ces architectures et aux pistes qui permettent de les éviter

Avant la présentation de ces architectures, intéressons nous rapidement à une application monolithique comme on en rencontre beaucoup actuellement. Voici un schéma présentant ce type d’application:

[](http://blog.xebia.fr/wp-content/uploads/2015/03/Capture-d’écran-2015-03-09-à-14.44.12.png)

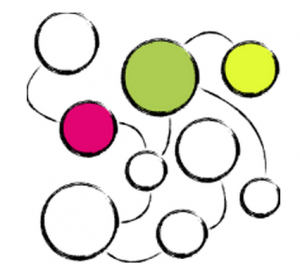
Notre application est faite d’un seul package, toutes les couches communiquent entres elles via des appels de méthodes. Ce constat va nous servir de point de départ pour notre premier exemple d’architecture: les microservices en REST.

**Microservices en REST**

Comme expliqué dans les concepts, une architecture microservices est un système de services communicant entre eux.

Un des moyens les plus simples de faire communiquer ces services entre eux est de le faire à travers le protocole HTTP via une API REST. En reprenant l’exemple précédent, on peut imaginer isoler chaque partie métier de notre application pour la transformer en un service indépendant exposant sa propre interface. Les autres services du système communiqueront avec lui.

L’organisation de notre application ressemblera à ceci :

[](http://blog.xebia.fr/wp-content/uploads/2015/03/Capture-d’écran-2015-03-09-à-14.44.23.png)

Chacune des parties métier de notre application est devenue un service indépendant, ce qui présente plusieurs avantages:

* Chaque service a une base de code plus petite, ce qui le rend plus lisible et facilite sa maintenance.
* Chaque service étant indépendant du reste, il peut évoluer plus rapidement. Si le contrat d’interface est constant, les autres services ne seront pas impactés par les changements internes. De plus, toujours grâce à cette indépendance, il est possible d’écrire chaque service dans le langage qui correspond le mieux à l’équipe dédiée à son développement ou au besoin précis du service. Il est ainsi plus simple de focaliser les efforts de performance sur un service en particulier qui sera écrit en langage C, adopter l’isomorphisme javascript pour une interface web ou encore profiter de la richesse des bibliothèques java pour l’intégration avec le SI.
* Chaque service peut travailler avec ses propres bases de données, du type qui correspond le mieux à son usage. Par exemple, il pourra utiliser une base NoSQL sans que les autres services soient obligés d’évoluer.

Un des gros avantages de cette architecture est de permettre aux services de *scaler* facilement et distinctement. Un service peu sollicité peut rester sur une instance unique tandis qu’un service devant supporter une charge importante peut multiplier ses instances d’exécutions de manière transparente.

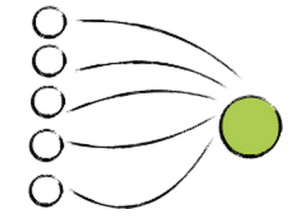
Pour réaliser cela, il faut une méthode simple permettant aux services de se découvrir entre eux.

Chaque service doit connaitre les adresses de l’ensemble des services qu’il utilise. Une façon de faire serait d’indiquer l’adresse IP de chaque API dans la configuration du service. Cette pratique pose toutefois plusieurs problèmes:

* Si l’adresse IP d’un service change, vous devrez changer l’intégralité des configurations utilisant ce service
* Gérer les multiples instances d’un service avec une configuration par adresse IP est très compliqué. Cela oblige à maintenir manuellement l’inventaire des services utilisés

Etant donné que nous sommes dans un univers web, autant utiliser les moyens du web. La solution la plus immédiate consiste en la mise en place d’un loadbalancing qui va répartir la charge en entrée sur plusieurs instances du service. Quelle que soit l’implémentation technique retenue (IP, DNS, proxy, …), elle permet de *scaler* horizontalement, facilite la montée en version du service tout en offrant aux services consommateurs un point d’entrée constant.

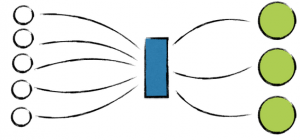
Faisons un zoom sur le comportement d’un service fortement chargé où la communication est directe entre les services :

[](http://blog.xebia.fr/wp-content/uploads/2015/03/Capture-d’écran-2015-03-09-à-14.44.28.png)

Chacune des applications a été configurée pour accéder directement à un service en particulier. Une telle typologie, bien que simple à mettre en œuvre présente plusieurs inconvénients. D’une part il est impossible d’ajouter une instance d’un service consommé et d’autre part la mise à jour du service entraine nécessairement une rupture dans la continuité de service.

L’utilisation d’un *loadbalancer* pour supporter la montée en charge n’est pas spécifique aux architectures microservices mais il permet d’orchestrer efficacement la montée en version. Le nouveau service est démarré, le *loadbalancer* redirige les requêtes vers le nouveau service puis l’ancien service peut être décommissionné.

Maintenant si nous glissons un *loadbalancer* dans le système, il est facile d’ajouter de nouvelles instances de notre service (vert) sans bouleverser le système :

[](http://blog.xebia.fr/wp-content/uploads/2015/03/Capture-d’écran-2015-03-09-à-14.44.32.png)

Le *loadbalancer* est une bonne solution pour faciliter la *scalabilité* de votre système. Néanmoins, il faut être conscient que cela ajoute un élément actif sur l’acheminement de la requête et donc une latence supplémentaire ou une potentielle panne. Cet élément devient donc critique. Il s’avère être une solution simple pour quelques services mais peut s’avérer complexe à une échelle plus importante.

Comme nous venons de le voir, ce type d’architecture permet de mettre en place simplement des microservices grâce à un système de communication couramment utilisé, même dans des applications monolithiques. Néanmoins ce système révèle plusieurs problèmes:

* Les appels REST sont synchrones, si un service ralentit, l’ensemble du système s’en trouve impacté
* Certains appels REST ne sont pas idempotents (principalement POST et PUT) et ne peuvent être rejoués plus tard au premier échec
* La multiplication des services peut entraîner un effet « spaghetti » entre les liens de communication
* Transposer une application monolithique en microservices en remplaçant un appel de méthode par son équivalent en RPC (REST ou autre) limite les performances globales du système.

Résumer les services REST aux seuls appels synchrones serait un raccourci maladroit. Rien n’interdit en effet de concevoir une architecture reposant sur des messages asynchrones tout en utilisant le protocole HTTP. Certes ce dernier est synchrone par définition mais il est tout à fait possible qu’un service consomme un autre service en communicant une adresse de callback, laquelle sera appelée lorsque le traitement sera terminé. Nous rencontrons fréquemment cette séquence d’échanges dans les protocoles SSO par exemple. Le terme d’asynchronisme peut ainsi avoir une signification différente selon que l’on se place au niveau du protocole d’échange ou au niveau de l’orchestration des échanges.

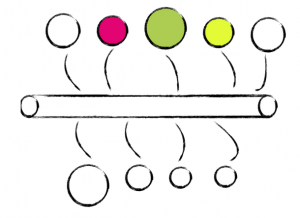
Pour palier aux différents problèmes évoqués précédemment, un autre système de communication existe : la communication par bus de messages.

**Microservices à travers un bus**

La communication par bus de messages est fortement poussée dans les architectures microservices. Elle apporte une réelle souplesse que nous allons détailler par la suite.

Néanmoins, ce type de communication change complètement la manière de penser les applications. Il faut être conscient de ce fait avant de se lancer dans la conception d’une telle architecture.

Comme exposé précédemment, la communication par services REST peut devenir une faiblesse lorsque le nombre de liens se multiplie. Dans la communication par messages, les liens entre services sont remplacés par un bus central :

[](http://blog.xebia.fr/wp-content/uploads/2015/03/Capture-d’écran-2015-03-09-à-14.44.44.png)

Chaque service envoie des messages sur le bus qui seront consommés ensuite par d’autres services. On remarque tout de suite un avantage dans les échanges: chaque service ne connait que le bus de messages comme intermédiaire. Les liens directs entre services sont coupés. Naturellement, chaque service doit connaitre la typologie des messages qu’il souhaite utiliser.

La responsabilité d’un bus est volontairement limitée: transmettre les messages. Suivre cette approche permet de réduire au maximum l’impact du bus de communication sur le métier. Il s’agit d’un composant d’infrastructure sans intelligence particulière. Cette intelligence business est concentrée aux extrémités, dans les (micro-)services. Les critères de sélection d’une solution technique sont ainsi simplifiés : le bus doit faire peu de choses mais bien. Parmi ces critères retenons la capacité de débit, la garantie d’acheminement, la latence ou la typologie réseau.

Pour qu’un service reçoive des messages, il doit simplement écouter sur le bus ceux qui l’intéressent. Les services sont ainsi réellement isolés du reste du système. Leurs seuls liens vers l’extérieur étant la typologie des messages (reçus ou émis), ainsi que le bus.

L’utilisation d’un bus de messages induit un nouveau concept : tout ce qui se passe dans le système est **asynchrone**. En effet, une fois un message envoyé sur le bus, il n’y a aucune garantie qu’il sera lu et surtout, quand il sera lu.

Ceci oblige à penser l’architecture de manière totalement asynchrone, ce qui peut être déroutant au départ. Néanmoins, cet asynchronisme présente plusieurs avantages :

* Le système devient résistant à la lenteur. Une fois le message envoyé, le service n’attend pas de réponse immédiate. Si le service répondant devient lent, l’appelant n’est pas impacté
* Le système devient tolérant à la panne. Si un service tombe, tous les messages qui lui sont destinés sont gardés dans le bus. Il pourra reprendre le travail une fois redémarré, sans perte.
* Le système devient facilement *scalable*. N’importe qui peut lire dans le bus, qu’il y ait une ou dix instances du même service ne change rien pour le bus et cela répartit la charge entre chaque instance

Ce type d’architecture permet de réellement découpler les services et de les faire évoluer de manière totalement indépendante. Tant que la typologie des messages ne change pas, l’évolution des services sera transparente pour le système.

Comme vous l’avez sans doute remarquer, le point sensible des architectures avec communication par messages est le bus lui même. Il devient le goulot d’étranglement du système.

Heureusement, aujourd’hui, les technologies de bus supportent particulièrement bien la charge, permettent de *scaler* le bus facilement et si besoin de rejouer les messages. Le rejeu d’anciens messages conduit vers le principe d’*[event sourcing](http://martinfowler.com/eaaDev/EventSourcing.html).*

Les bus se doivent également d’être des « *dumb pipe »*. Autrement dit, un bon bus ne porte aucune intelligence de routage. Il reçoit des messages que d’autres services viennent lire. Le bus est un élément technique du système, pas un élément fonctionnel. C’est un point essentiel qui permet d’éviter d’avoir des bus très complexes contenant des tables de routages encore plus complexes.

Un autre point également très important dans ce type d’architecture est le monitoring. L’ensemble étant totalement asynchrone et découplé, un service peut tomber sans que cela soit visible immédiatement. Les autres services continueront d’envoyer leurs messages sans ralentissement visible. Il est donc primordial de penser à un bon système de monitoring. Que les données collectées soient business ou techniques, il est impératif de savoir ce qui se passe dans le système. Mettre en place des *dashboards* simples et visuels permettront de détecter rapidement les problèmes.

**Conclusion**

Nous venons de voir deux types d’architectures microservices. La première avec communication par service REST est certainement la plus simple à mettre en oeuvre. Elle est efficace lorsqu’il s’agit de migrer depuis une application monolithique.

La deuxième architecture avec le bus de messages est la plus intéressante. L’asynchronisme rend plus robuste et plus tolérant le système. Le bus permet de *scaler* horizontalement tout type de services simplement et efficacement.

Il est par ailleurs tout à fait envisageable de mixer les deux types d’architecture afin d’obtenir une architecture hybride. Ceci peut par exemple être envisagé lors d’une migration progressive vers du tout asynchrone.

Ces architectures sont surtout utilisées dans des environnements complexes devant absorber une charge importante.

Le lot de nouveautés des microservices apporte également de nouvelles problématiques : sécurité du bus de messages, gestion de l’asynchronisme, débuggage de systèmes complexes. Le monitoring de ces systèmes est un élément très important à prendre en compte. Sans un bon monitoring, les systèmes complexes peuvent devenir un enfer à gérer en production.

Heureusement, de nouveaux outils sont apparus pour aider à la mise en place d’une architecture microservices. Des frameworks comme vert.X avec un bus intégré ou encore Spring Boot permettent de créer rapidement de nouveaux services.

L’orchestration du système est également un point essentiel. Le nombre de services se multipliant, il devient important d’automatiser les déploiements et la gestion de l’ensemble. Là aussi, des outils font leur apparition comme Marathon et Mesos. Les microservices se marient idéalement avec Docker.

Dans notre prochain article, nous verrons plus en détails les pièges possibles des microservices et comment les éviter.