

Livre blanc

*Plateformes web
Hautes Performances*
--o--
*Principes d'architecture
et outils open source*

Collection « système et infrastructure »

PREAMBULE

Smile

Smile est une société d'ingénieurs experts dans la mise en œuvre de solutions open source et l'intégration de systèmes appuyés sur l'open source. Smile est membre de l'APRIL, l'association pour la promotion et la défense du logiciel libre.

Smile compte 290 collaborateurs en France, 320 dans le monde (septembre 2009), ce qui en fait *la première société en France spécialisée dans l'open source*.

Depuis 2000, environ, Smile mène une action active de veille technologique qui lui permet de découvrir les produits les plus prometteurs de l'open source, de les qualifier et de les évaluer, de manière à proposer à ses clients les produits les plus aboutis, les plus robustes et les plus pérennes.

Cette démarche a donné lieu à toute une gamme de *livres blancs* couvrant différents domaines d'application. La gestion de contenus (2004), les portails (2005), la business intelligence (2006), les frameworks PHP (2007), la virtualisation (2007), et la gestion électronique de documents (2008), ainsi que les PGIs/ERPs (2008). Parmi les ouvrages publiés en 2009, citons également « Les VPN open source », et « Firewall est Contrôle de flux open source », dans le cadre de la collection « Système et Infrastructure ».

Chacun de ces ouvrages présente une sélection des meilleures solutions open source dans le domaine considéré, leurs qualités respectives, ainsi que des retours d'expérience opérationnels.

Au fur et à mesure que des solutions open source solides gagnent de nouveaux domaines, Smile sera présent pour proposer à ses clients d'en bénéficier sans risque. Smile apparaît dans le paysage informatique français comme le prestataire intégrateur de choix pour accompagner les plus grandes entreprises dans l'adoption des meilleures solutions open source.

Ces dernières années, Smile a également étendu la gamme des services proposés. Depuis 2005, un département consulting accompagne nos clients, tant dans les phases d'avant-projet, en recherche de solutions, qu'en accompagnement de projet. Depuis 2000, Smile dispose d'un studio graphique, devenu en 2007 Agence Media Interactive, proposant outre la création graphique, une expertise e-marketing, éditoriale, et

Architectures Hautes-Performances

interfaces riches. Smile dispose aussi d'une agence spécialisée dans la Tierce Maintenance Applicative, le support et l'exploitation des applications. Enfin, Smile est implanté à Paris, Lyon, Nantes, Bordeaux et Montpellier. Et présent également en Espagne, en Suisse, en Ukraine et au Maroc.

Quelques références

Intranets - Extranets

Société Générale, Caisse d'Épargne, Bureau Veritas, Commissariat à l'Energie Atomique, Visual, Vega Finance, Camif, Lynxial, RATP, SPIE, Sonacotra, Faceo, CNRS, AmecSpie, Château de Versailles, Banque PSA Finance, Groupe Moniteur, CIDJ, CIRAD, Bureau Veritas, Ministère de l'Environnement, JCDecaux, Ministère du Tourisme, DIREN PACA, SAS, Institut National de l'Audiovisuel, Cogedim, Ecureuil Gestion, IRP-Auto, AFNOR, Conseil Régional Ile de France, Verspieren, Zodiac, OSEO, Prolea, Conseil Général de la Côte d'Or, IPSOS, Bouygues Telecom, Pimki Diramode, Prisma Presse, SANEF, INRA, HEC, ArjoWiggins

Internet, Portails et e-Commerce

cadremploi.fr, chocolat.nestle.fr, creditlyonnais.fr, explorimmo.com , meilleurtaux.com, cogedim.fr, capem.fr, editions-cigale.com, hotels-exclusive.com, souriau.com, pci.fr, dsv-cea.fr, egide.asso.fr, osmoz.com, spie.fr, nec.fr, sogeposte.fr, nouvelles-frontieres.fr, metro.fr, stein-heurtey-services.fr, bipm.org, buitoni.fr, aviation-register.com, cci.fr, schneider electric.com, calypso.tm.fr, inra.fr, cnil.fr, longchamp.com, aesn.fr, Dassault Systemes 3ds.com, croix rouge.fr, worldwatercouncil.org, projectif.fr, editionsbussiere.com, glamour.com, fratel.org, tiru.fr, faurecia.com, cidil.fr, prolea.fr, ETS Europe, ecofi.fr, credit cooperatif.fr, odit france.fr, pompiersdefrance.org, watermonitoringalliance.net, bloom.com, meddispar.com, nmmedical.fr, medistore.fr, Yves Rocher, jcdecaux.com, cg21.fr, Bureau Veritas veristar.com, voyages sncf.fr, eurostar.com, AON, OSEO, cea.fr, eaufrance.fr, banquepsafinance.com, nationalgeographic.fr, idtgv.fr, prismapub.com, Bouygues Construction, Hachette Filipacchi Media, ELLE.fr, femmeactuelle.fr, AnnoncesJaunes.fr, Groupama, Macif, Le Furet du Nord, Camif-Collectivités.

Applications métier, systèmes documentaires, business intelligence

Renault, Le Figaro, Sucden, Capri, Libération, Société Générale, Ministère de l'Emploi, CNOUS, Neopost Industries, ARC, Laboratoires Merck, Egide, Bureau Veritas, ATEL-Hotels, Exclusive Hotels, Ministère du Tourisme, Groupe Moniteur, Verspieren, Caisse d'Epargne, AFNOR, Souriau, MTV, Capem, Institut Mutualiste Montsouris, Dassault Systemes, Gaz de France, CFRT, Zodiac, Croix-Rouge Française, Centre d'Information de la Jeunesse (CIDJ), Pierre Audoin Consultants, EDF, Conseil Régional de Picardie, Leroy Merlin, Renault F1, l'INRIA, Primagaz, Véolia Propreté, Union de la Coopération Forestière Française, Ministère Belge de la Communauté Française, Prodig

Ce livre blanc

Au cours des 10 dernières années, Smile a conçu et déployé quelques unes des plus grandes plateformes du web français. Des plateformes qui ont su évoluer en douceur pour accueillir un trafic toujours croissant, atteignant couramment plusieurs millions de pages vues par mois.

Les techniques et outils ont évolué, mais certains principes fondamentaux restent les mêmes.

Cet ouvrage vous fait partager l'expertise des équipes de Smile dans la mise en œuvre de ces grandes plateformes du web. Construit de manière didactique, il rappelle tout d'abord les concepts élémentaires, puis approfondit progressivement l'analyse jusqu'à présenter les techniques les plus avancées, permettant de viser une extensibilité réellement sans limite.

Bien entendu, une majorité des outils sur lesquels s'appuient les plus grandes plateformes du web sont des outils open source, dont Smile s'est fait une spécialité.

SOMMAIRE

| | |
|---|-----------|
| PREAMBULE | 2 |
| SMILE..... | 2 |
| QUELQUES RÉFÉRENCES | 3 |
| <i>Intranets - Extranets.....</i> | 3 |
| <i>Internet, Portails et e-Commerce.....</i> | 3 |
| <i>Applications métier, systèmes documentaires, business intelligence</i> | 3 |
| CE LIVRE BLANC | 4 |
| SOMMAIRE..... | 5 |
| PROBLEMATIQUE DES SITES HAUTES-PERFORMANCES | 10 |
| HAUTES-PERFORMANCES | 10 |
| PERFORMANCES ET ARCHITECTURE | 11 |
| L'EXTENSIBILITÉ | 12 |
| <i>L'extensibilité en trois dimensions</i> | 13 |
| <i>Extensibilité cellulaire</i> | 14 |
| <i>Extensibilité fonctionnelle, ou verticale</i> | 15 |
| <i>Extensibilité horizontale</i> | 15 |
| <i>Quelle cellule élémentaire, quelle brique de base ?.....</i> | 15 |
| AUDIENCE ET CAPACITÉ | 17 |
| <i>Chiffres clés.....</i> | 17 |
| <i>Heure de pointe.....</i> | 17 |
| <i>Connexions simultanées</i> | 18 |
| <i>Temps de réponse.....</i> | 19 |
| MÉGA-SERVEURS ? | 20 |
| COÛTS..... | 20 |
| QUELQUES PRINCIPES POUR LA HAUTE PERFORMANCE | 21 |
| <i>Share-Nothing</i> | 21 |
| <i>« Simple is beautiful »</i> | 22 |
| <i>Pas de solution unique</i> | 22 |
| <i>Les outils évoluent</i> | 22 |
| <i>Le méga-truc n'est pas la solution.....</i> | 22 |
| <i>Le poste client est plein de ressources.....</i> | 22 |
| <i>L'open source apporte beaucoup de solutions.....</i> | 22 |
| URBANISME ET SOA..... | 23 |
| URBANISME..... | 23 |
| ENCAPSULATION DES DONNÉES | 25 |
| L'URBANISME ET LES PLATEFORMES WEB..... | 26 |
| SERVICE ORIENTED ARCHITECTURE | 27 |
| UN SERVICE | 27 |
| MIDDLEWARE | 28 |
| LES MODES D'INTERACTION | 29 |
| <i>Synchrone</i> | 29 |
| <i>Asynchrone, one-way (aller-seul)</i> | 29 |
| <i>Asynchrone with callback</i> | 29 |

| | |
|---|-----------|
| <i>Asynchrone, publish / subscribe (publication / abonnement)</i> | 30 |
| LES TRAITEMENTS ASYNCHRONES | 30 |
| MOM ET MESSAGE QUEUES..... | 31 |
| DES SERVICES SANS ÉTAT..... | 33 |
| TROIS PROTOCOLES POUR LES SERVICES WEB | 33 |
| <i>XML-RPC</i> | 33 |
| <i>REST</i> | 34 |
| <i>SOAP.....</i> | 36 |
| <i>Services et interfaces</i> | 37 |
| <i>MOM open source – Apache ActiveMQ</i> | 39 |
| PERFORMANCES HTTP | 40 |
| CHRONOLOGIE DE CHARGEMENT DE PAGE | 40 |
| GESTION DU CACHE NAVIGATEUR..... | 42 |
| COMPRESSION DU FLUX | 43 |
| MOINS DE COMPOSANTS, MOINS DE REQUÊTES..... | 43 |
| <i>Quelques statistiques.....</i> | 43 |
| <i>Réduire le nombre de composants</i> | 46 |
| INFRASTRUCTURES GLOBALES ET CDN..... | 46 |
| REPARTITION DE CHARGE..... | 51 |
| PRINCIPE DE RÉPARTITION DE CHARGE | 51 |
| FINALITÉ ET LOGIQUE DE RÉPARTITION | 52 |
| <i>Augmenter la capacité.....</i> | 52 |
| <i>Equilibrer la charge</i> | 52 |
| <i>Résister aux pannes.....</i> | 52 |
| <i>Spécialiser des serveurs</i> | 52 |
| <i>Faciliter l'exploitation</i> | 52 |
| <i>Répartition de requêtes ou répartition de sessions ?.....</i> | 53 |
| <i>Répartition entre des serveurs ou entre des datacenters ?</i> | 53 |
| RÉPARTITION DE CHARGE DE NIVEAU DNS | 53 |
| <i>Principe</i> | 53 |
| <i>DNS-Round-Robin</i> | 54 |
| <i>GeoDNS.....</i> | 55 |
| <i>Anycast</i> | 56 |
| <i>Avantages et limites de la répartition DNS.....</i> | 56 |
| <i>Redirection applicative</i> | 57 |
| RÉPARTITION DE CHARGE DE NIVEAU TCP | 58 |
| <i>Quelques rappels</i> | 58 |
| <i>Répartition de charge TCP.....</i> | 58 |
| <i>Les algorithmes de répartition.....</i> | 59 |
| RÉPARTITION DE CHARGE DE NIVEAU 7 | 60 |
| <i>Répartition avec affinité de serveur</i> | 60 |
| <i>Principe de la répartition niveau 7</i> | 61 |
| <i>Spécialisation des serveurs.....</i> | 62 |
| <i>Répartition de charge et SSL</i> | 62 |
| GESTION DES SESSIONS | 63 |
| <i>Partage de sessions par cookies</i> | 63 |
| <i>Partage de contexte côté serveur.....</i> | 66 |
| <i>Partage de contexte en cache global</i> | 66 |
| <i>Synthèse</i> | 67 |
| CONFIGURATION RÉSEAU | 67 |

| | |
|--|-----------|
| Niveau 4, niveau 7, même configuration | 67 |
| Répartition de charge inter-datacenter | 68 |
| Configuration réseau et tolérance aux pannes de l'équipement | 69 |
| LES SOLUTIONS ET OUTILS | 71 |
| Solutions logicielles | 71 |
| Les boîtiers dédiés | 74 |
| Fonctionnalités associées | 75 |
| RÉPARTITION PEER-BASED | 75 |
| LOAD-BALANCING SUR DES SERVICES INTERNES | 78 |
| MAPREDUCE ET HADOOP | 79 |
| HAUTE DISPONIBILITE | 81 |
| HAUTE DISPONIBILITÉ | 81 |
| TOLÉRANCE AUX PANNEES | 81 |
| A tous les niveaux | 81 |
| BONNES PRATIQUES | 82 |
| Quelques bonnes pratiques de la haute disponibilité | 82 |
| Pannes logicielles | 82 |
| Exploitation | 83 |
| Changements de version | 83 |
| REDONDANCE ET SECOURS | 84 |
| Surveillance et passage en secours | 84 |
| Surveillance par l'homologue et heartbeat | 85 |
| Surveillance par l'étage amont | 85 |
| Secours passif | 86 |
| Secours actif | 86 |
| Gestion mutualisée du secours | 87 |
| Single Point of Failure | 87 |
| MTBF et probabilité de panne | 87 |
| La brique de base, le serveur élémentaire | 88 |
| MONITORING ET ALERTES | 89 |
| Service complet, scénarios applicatifs | 89 |
| Monitoring Http | 90 |
| Woozweb | 91 |
| LA GESTION DES DONNÉES | 92 |
| GESTION DES DONNÉES | 92 |
| Un problème difficile | 92 |
| Pensée unique ? | 92 |
| Modélisation objet et programmes | 92 |
| Les propriétés ACID | 93 |
| Le cluster | 94 |
| Lecture seule, extensibilité | 96 |
| Ecriture seule, extensibilité | 97 |
| Le partitionnement des données | 97 |
| Synthèse | 103 |
| BASE DE DONNÉES | 103 |
| La grosse base centrale | 103 |
| L'approche classique | 104 |
| La réPLICATION SGBD simple | 104 |
| RéPLICATION « manuelle » | 106 |
| RéPLICATION croisée, multi-maîtres | 106 |

| | |
|---|------------|
| <i>RAIDb et Sequoia</i> | 109 |
| LE MOTEUR D'INDEXATION-RECHERCHE | 111 |
| GESTION DE FICHIERS | 112 |
| <i>Une problématique différente</i> | 112 |
| <i>Des fichiers en base</i> | 112 |
| <i>La réPLICATION</i> | 113 |
| <i>Gestion de contenus et réPLICATION</i> | 114 |
| <i>SAN</i> | 116 |
| <i>Architecture NAS</i> | 118 |
| <i>DRBD</i> | 121 |
| <i>L'accès concurrent aux fichiers</i> | 121 |
| <i>Lustre</i> | 122 |
| <i>MogileFS</i> | 123 |
| <i>Hadoop HDFS</i> | 124 |
| LE CACHE | 125 |
| PRINCIPES DU CACHE | 125 |
| ACTIF / PASSIF, PULL / PUSH | 125 |
| CACHE EN MODE PULL | 126 |
| <i>Durée de vie</i> | 127 |
| <i>Le fonctionnement « MRU »</i> | 128 |
| LE CACHE HTTP | 128 |
| <i>Cache du navigateur</i> | 129 |
| <i>Un peu de mémoire suffit</i> | 129 |
| <i>La mémoire ne coûte pas cher</i> | 130 |
| <i>Mise en œuvre d'un cache HTTP</i> | 132 |
| <i>Les outils du cache HTTP</i> | 133 |
| CACHE PAR FRAGMENTS | 133 |
| <i>Sites personnalisés et contenus temps-réel</i> | 133 |
| <i>Introduction au cache par fragments</i> | 134 |
| <i>Agrégation de fragments et portails J2EE</i> | 135 |
| <i>Edge Side Include (ESI)</i> | 136 |
| <i>Web-scraping, web-clipping</i> | 137 |
| <i>Caches ESI Open Source</i> | 137 |
| <i>Le Web Assembling Toolkit</i> | 137 |
| <i>Agrégation de fragments côté client</i> | 139 |
| CACHE EN MODE PUSH | 139 |
| <i>Génération de pages statiques</i> | 140 |
| CACHE DE DONNÉES | 142 |
| <i>Cache de données</i> | 142 |
| <i>Approche ensembliste ou clé/valeur</i> | 142 |
| <i>Le cache gestionnaire de données</i> | 143 |
| <i>Memcached</i> | 145 |
| <i>EhCache</i> | 146 |
| QUELQUES CAS D'ECOLE | 148 |
| UNE MONTÉE EN PUISSANCE ORDINAIRE | 148 |
| <i>Le problème posé</i> | 148 |
| <i>Optimisation</i> | 148 |
| <i>Extension cellulaire</i> | 149 |
| <i>Extension verticale</i> | 149 |
| <i>Extension horizontale</i> | 151 |

| | |
|---|------------|
| <i>Extension en 2D</i> | 153 |
| <i>Spécialisation en écriture / lecture</i> | 155 |
| <i>Bases multiples en réPLICATION croisée</i> | 157 |
| <i>Serveur dédié à la contribution</i> | 158 |
| <i>Partitionnement des données</i> | 159 |
| ARCHITECTURE TYPE FACEBOOK | 159 |
| UNE PLATEFORME DE BLOGS DE TRÈS FORTE CAPACITÉ..... | 161 |
| <i>Le problème posé</i> | 161 |
| <i>Quelles options d'architecture ?</i> | 162 |
| <i>Un problème partitionnable</i> | 163 |
| <i>Répartition arbitraire</i> | 164 |
| <i>Fonctions centrales via webservices</i> | 164 |
| <i>Fonctions centrales via datawarehouse</i> | 165 |
| <i>Weservices + datawarehouse</i> | 165 |
| <i>Répartition de charge</i> | 166 |
| <i>Pas d'autre axe d'extensibilité</i> | 166 |
| <i>Secours</i> | 167 |
| <i>SAN</i> | 167 |
| SPORT 24 ET 01 INFORMATIQUE..... | 168 |
| <i>Le problème posé</i> | 168 |
| <i>Axes de solution</i> | 169 |
| <i>Agrégation côté client</i> | 170 |
| <i>Agrégation côté serveur</i> | 170 |
| <i>Cache Squid en frontal</i> | 170 |
| <i>Génération de pages statiques</i> | 170 |
| <i>Diffusion des pages vers les frontaux</i> | 171 |
| <i>Schéma d'ensemble</i> | 171 |
| WOOZWEB | 174 |
| <i>La problématique Woozweb</i> | 174 |
| <i>Partitionnement et consolidation</i> | 175 |
| <i>Architecture de chaque « NOD »</i> | 176 |
| <i>Architecture globale</i> | 177 |
| <i>Répartition de charge</i> | 177 |
| <i>Agrégation de contenus</i> | 178 |
| <i>Synthèse</i> | 179 |
| CONCLUSION | 180 |

PROBLEMATIQUE DES SITES HAUTES-PERFORMANCES

Hautes-performances

Dans le titre de cet ouvrage, *hautes performances* est une expression un peu vague. La performance a de multiples dimensions, dont nous ne traiterons que quelques-unes. Dans l'ensemble, nous présenterons les principes d'architectures qui permettent de construire des plateformes web à grande capacité d'accueil et haute disponibilité, mais surtout, des plateformes extensibles, capables d'accompagner la montée en puissance d'un site web.

Ici donc, « hautes performances » recouvrira différentes qualités :

La capacité d'accueil

La *capacité d'accueil* est la capacité à offrir un service à un grand nombre de visiteurs, et tout particulièrement lors de pointes de trafic.

La haute disponibilité

La *haute disponibilité* est la garantie que le service offert sera accessible sans interruption, ou avec un très faible taux d'interruption. La *tolérance aux pannes*, c'est à dire l'aptitude à offrir un service sans interruption en présence d'une panne, est l'une des conditions de la haute-disponibilité.

Les temps de réponse

Le *temps de réponse* est le délai entre une requête d'un visiteur et sa réponse. C'est un aspect important de la qualité de service, pour lequel l'architecture et l'infrastructure peuvent avoir un impact important.

La qualité de service

La qualité de service, porte en général sur la *satisfaction globale* du visiteur, dans son utilisation du site, le fait qu'il trouve aisément ce qu'il recherche, qu'il ait envie de revenir. La qualité de service recouvre donc des aspects divers : temps de réponse, ergonomie, disponibilité, etc.

Il faut citer d'autres qualités, moins en vue, mais également importantes.

L'exploitabilité

C'est la facilité à exploiter la plateforme, à assurer la supervision et les opérations d'entretien.

L'évolutivité

La capacité à évoluer, tant au plan fonctionnel qu'au plan technique.

L'extensibilité

La possibilité d'accroître la capacité d'accueil aisément et à moindre coût. Nous y reviendrons, c'est l'une des qualités fondamentales d'une architecture hautes-performances.

Performances et architecture

Une première précision à apporter, qui trace le périmètre de cet ouvrage, est une distinction entre les questions de performances et celles d'architecture.

Bien souvent, la performance observée sur un serveur peut être améliorée par différentes actions d'optimisation, des algorithmes, de la configuration d'une base de données, des index, voire même par le choix d'un langage plus rapide.

L'optimisation vise une économie *de moyens*, en cherchant à réduire le temps que met un programme à réaliser sa tâche, ou bien encore à réduire l'utilisation de ressources, en particulier CPU, dans cette exécution.

Certes, il est ridicule de commencer à monter un cluster et empiler les serveurs pour tenir la charge alors que de bons index dans la base de données feraient gagner un facteur 20, comme on le voit couramment. Rien ne sert de s'attaquer à l'architecture avant d'avoir traité un niveau élémentaire d'optimisation.

L'optimisation est un sujet évidemment primordial pour bâtir des plateformes hautes-performances, mais c'est un sujet différent. Il est transverse : on peut travailler la performance par optimisation d'une part, et travailler à améliorer l'architecture d'autre part. Et bien sûr

combiner les deux démarches, mais surtout mener une analyse de la valeur pour cibler ses efforts sur l'un ou l'autre de ces axes.

Mais les démarches d'optimisation ont toujours une limite – quand tout est optimal ! – tandis que les démarches portant sur l'architecture peuvent apporter une croissance sans limite, et c'est ce que nous rechercherons.

Dans le cadre de cet ouvrage, nous ne traiterons pas de l'optimisation des programmes, bases de données ou plus largement des composants unitaires, mais uniquement de l'optimisation générale de l'architecture, c'est à dire de l'agencement des composants et de leurs échanges.

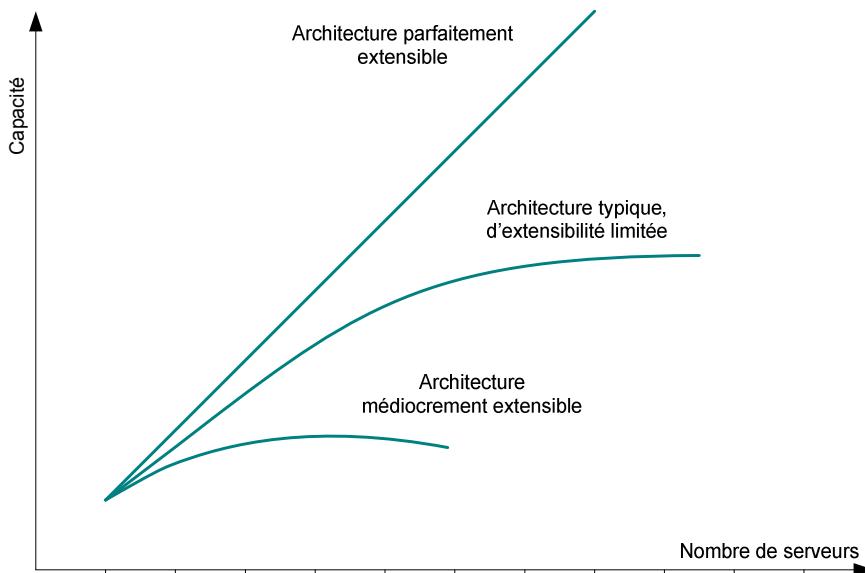
L'extensibilité

L'extensibilité, ou par anglicisme, « *scalabilité* », est la capacité d'une architecture à croître sans rupture, en ajoutant simplement du matériel.



L'extensibilité est la qualité principale d'une architecture web. On l'entend pour atteindre de fortes capacités, mais elle est importante aussi à l'autre extrême, pour commencer petit, et adapter la dépense en infrastructure à la croissance. C'est en quelque sorte le moyen de payer son infrastructure en *success-fee*, c'est à dire en cas de réussite. Et dans ce cas, on est toujours heureux de payer.

Architectures Hautes-Performances



Le schéma précédent représente différents cas de croissance de la capacité d'accueil d'une plateforme web, en fonction du nombre de serveurs qu'on y intègre :

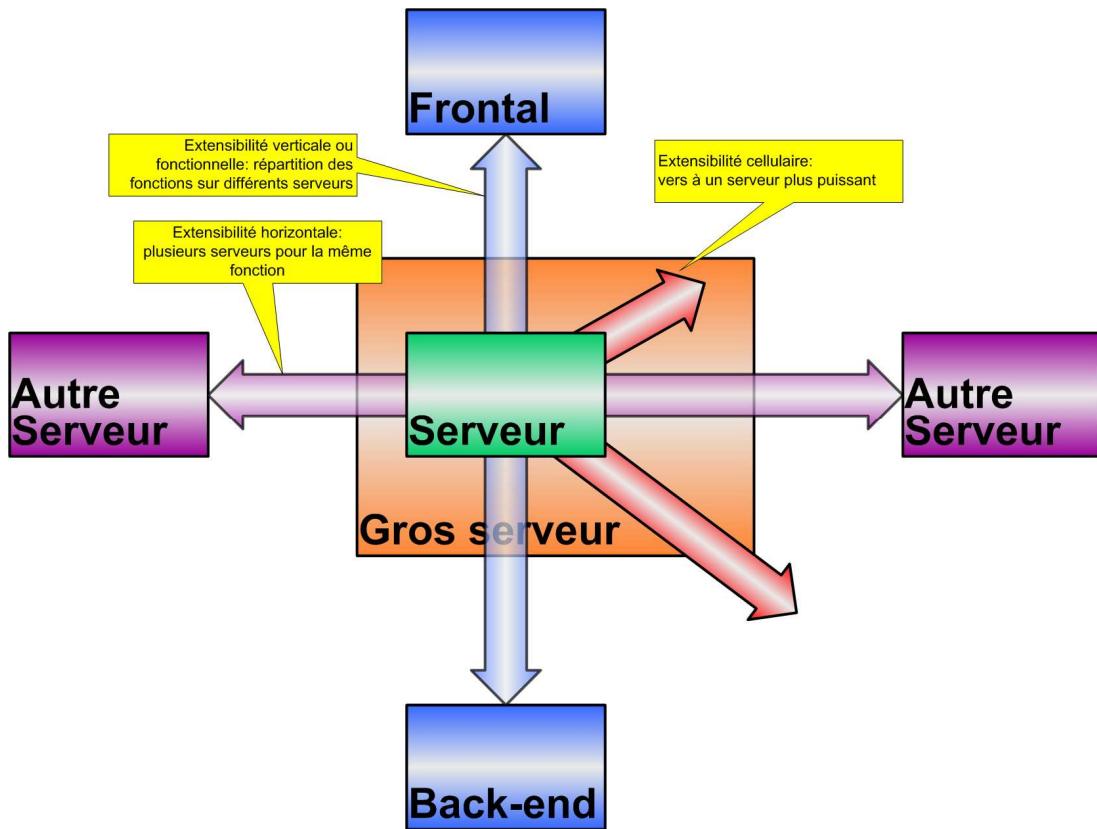
- En bas, une architecture médiocrement extensible : il n'y a guère à gagner en ajoutant des serveurs, et la capacité peut même rapidement se trouver dégradée, par exemple par des problèmes de synchronisation entre serveurs.
- En haut, l'extensibilité théorique idéale : la capacité d'accueil augmente de manière linéaire avec les machines.
- Et au milieu, le cas ordinaire type, une architecture qui est extensible jusqu'à un certain point, mais qui atteint une limite en asymptote horizontale.

L'extensibilité en trois dimensions

On peut travailler l'extensibilité selon trois dimensions, que nous appellerons ici l'extensibilité cellulaire, fonctionnelle et horizontale.

- L'extensibilité cellulaire consiste simplement à augmenter la puissance d'un ou des serveurs.
- L'extensibilité fonctionnelle, que l'on pourrait appeler aussi « verticale », consiste à *répartir les fonctions* de la plateforme sur des serveurs différents.
- L'extensibilité horizontale consiste à répartir les traitements sur différents serveurs homologues.

Nous analyserons les caractéristiques de chacune de ces démarches d'extensibilité, et nous verrons que seule l'extensibilité horizontale est pratiquement sans limite.



Extensibilité cellulaire

On appelle ici extensibilité « *cellulaire* », la capacité à faire gonfler la cellule élémentaire de l'architecture, le serveur de base : CPU, mémoire, disque, etc.

Il faut garder à l'esprit la fameuse loi de Moore, selon laquelle la puissance CPU disponible pour un prix donné est multipliée par deux tous les 18 mois.

C'est à dire que si votre plateforme a plus de 18 mois et commence à saturer, le moyen le plus simple et parfois le moins coûteux d'aller plus loin consiste à remplacer les serveurs par la dernière génération.

Mais une fois qu'on a fait cela, il faut trouver d'autres voies d'extensibilité, ou attendre à nouveau 18 mois.

Extensibilité fonctionnelle, ou verticale

Il s'agit ici de distinguer différentes *fonctions* dans la plateforme, et de répartir ces fonctions sur différents serveurs. Dans certains cas, ces fonctions peuvent s'organiser en *couches*, ou *tiers*, et on parlera d'architectures multi-tiers.

Ainsi on peut distinguer sur une plateforme web typique, les fonctions suivantes :

- Le serveur HTTP, typiquement Apache,
- Le serveur d'application, par exemple Tomcat ou Jboss,
- Les applications web
- La base de données.

Mais on identifie parfois d'autres fonctions, pas nécessairement en couches. Par exemple un moteur d'indexation-recherche, ou bien la distinction d'une application de back-office par rapport à un front-office.

Ces différentes *fonctions*, que l'on pourra appeler *services* dans une logique SOA, peuvent être initialement disposées sur un même serveur. Et au fur et à mesure de la montée en charge, elles peuvent être réparties sur différents serveurs spécialisés, ce qui évidemment permet de donner plus de ressources physiques, particulièrement de CPU, à l'ensemble de la plateforme.

Cette forme d'extensibilité est limitée, bien sûr, par le nombre de fonctions identifiées dans l'architecture.

Extensibilité horizontale

Enfin, comme on l'a vu, l'extensibilité horizontale consiste à répartir une même typologie de traitements, une même fonction, sur différents serveurs homologues. Nous verrons au chapitre « Répartition de Charge », page 51, les principes et outils de ce type de répartition.

Des trois axes, c'est le seul qui soit véritablement sans limite. Mais bien sûr, les trois peuvent être combinés.

Quelle cellule élémentaire, quelle brique de base ?

En supposant que l'on soit parvenu à un degré satisfaisant d'extensibilité, on a gagné la possibilité de choisir entre N_1 serveurs de capacité unitaire C_1 et de prix unitaire P_1 , ou bien N_2 serveurs de capacité unitaire C_2 et de prix unitaire P_2 , pour une même capacité totale, c'est à dire $N_1C_1 = N_2C_2$.

Architectures Hautes-Performances

On peut traiter la question par le simple prix unitaire de la capacité de traitement : quel est le meilleur rapport C/P, et donc le moindre coût d'acquisition de la plateforme. Cette simple analyse conduirait le plus souvent vers les serveurs les plus bas de gamme du marché.

Mais plusieurs autres considérations sont à prendre en compte :

- Les coûts d'hébergement dépendent pour une part importante du nombre de serveurs, indépendamment de leur puissance : espace de racks, ports des switchs, etc.
- Toute l'administration et l'exploitation prennent une complexité croissante, et donc un coût supérieur, avec le nombre de serveurs. Cette augmentation n'est cependant pas linéaire : passer de 3 serveurs à 10 serveurs peut doubler le coût d'exploitation, mais passer de 10 à 20 n'augmentera peut être que de 50%.
- Le nombre de pannes est proportionnel au nombre de serveurs – à qualité égale – et chaque panne engendre un coût spécifique de traitement.
- Le coût de l'électricité n'est pas négligeable, et il importe de considérer aussi le rapport C/E, capacité par Watt, qui varie de manière importante entre les serveurs.

Il s'ensuit que, à rapport puissance/prix égal, on préfèrera minimiser le nombre de serveurs. C'est à dire typiquement que si un serveur bi-processeur coûte deux fois le prix d'un serveur mono-processeur, alors on préfèrera mettre en œuvre des serveurs bi-processeurs.

Chaque étage ne s'analyse pas de la même manière, précisément parce que l'impact du nombre de processeurs peut être très différent. Ainsi, l'étage base de données est toujours moins extensible, plus difficile à paralléliser. Plutôt que de mettre en œuvre une base répartie sur 4 serveurs, il sera sans doutes moins coûteux de n'administrer qu'un serveur unique, quadri-processeurs. Mais bien sûr, on sera alors en butée, sans plus d'extensibilité horizontale.

Les serveurs multi-processeurs apportent une certaine extensibilité de manière quasi-transparente. Mais quelques-uns des problèmes liés à la parallélisation, que l'on traitera avec l'extensibilité horizontale se retrouveront déjà à l'échelle d'un serveur multi-processeurs.

Une autre considération importante en matière d'acquisition de serveurs est que l'on ne peut pas espérer une plateforme homogène en termes de configuration. Une plateforme monte en gamme progressivement, et l'on ne peut pas passer commande d'un coup pour les 3 années à venir. Sur le plan logiciel, en revanche, on s'attachera à avoir des configurations identiques, mais les écarts matériels, principalement au niveau du processeur, auront des impacts en

particulier dans les dispositifs de répartition de charge, qui devront prendre en considération la puissance spécifique de chaque serveur. On verra plus loin que l'un des apports majeurs des solutions de virtualisation est justement de mieux masquer les petites différences du matériel, et donc de faciliter la gestion d'un parc hétérogène.

Audience et Capacité

Chiffres clés

Lorsque l'on parle de capacité d'accueil, il faut souvent manipuler des notions qui s'entremêlent. Marketing, webmestres et architectes ne parlent pas le même langage. Voyons rapidement comment construire des équivalences.

| | | |
|------------------------------|-------------------|---|
| Visites par mois | V_{mois} | Une visite est une session, une suite de requêtes plus ou moins enchaînées. On parle aussi de visiteurs uniques, mais cela n'a pas d'intérêt au plan technique. Un même visiteur (unique) peut effectuer plusieurs visites dans le mois. |
| Pages par visite | P | Le nombre de pages parcourues lors d'une visite. On parle d'une moyenne sur l'ensemble du trafic, bien sûr. Il se situe typiquement entre 5 et 15 pages par visite. C'est une grandeur qui est en général assez stable, pour un site donné. |
| Délai moyen entre deux pages | D | Le délai moyen entre deux requêtes de pages. Il peut être typiquement de l'ordre de 15-30 secondes, mais dépend fortement bien sûr de la quantité d'information de la page, et plus encore des phases de saisies. |

Heure de pointe

Le trafic n'est pas régulier, il y a des jours de pointe, et au sein d'une journée typique, il y a des heures de pointe. Pour le dimensionnement d'une plateforme, c'est uniquement l'heure de pointe qui importe.

On considère parfois que le jour de pointe représente 1/20ème du trafic du mois, soit $V_{\text{jour}} = V_{\text{mois}}/20$. Et de même, on peut estimer que l'heure de pointe concentre 1/5ème du trafic de la journée, $V_{\text{heure}} = V_{\text{jour}}/5$. Et au total donc $V_{\text{heure}} = V_{\text{mois}}/100$.

En fait ces ratios dépendent fortement de la typologie de l'application, de sa cible, grand public ou professionnelle. Le site d'une émission de télévision, ou d'événements sportifs, pourront avoir un trafic bien plus concentré encore. Mais du moins en l'absence de plus d'information, on peut utiliser ces ratios.

Pour la plateforme, la notion de *visite* n'est pas la plus importante. Sur un site de publication, on s'intéressera plutôt au nombre de pages servies.

Si à l'heure de pointe, on reçoit V_{heure} visites, cela correspond à $P.V_{\text{heure}}$ pages servies, que l'on peut ramener à la seconde : $W = P.V_{\text{heure}}/3600$ pages servies par seconde. Pour fixer les idées, un serveur sert typiquement entre 10 et 1000 pages par seconde. Oui, l'écart est très grand, mais c'est un fait : la capacité unitaire dépend de l'optimisation, du langage et de nombreux paramètres internes.

Application numérique : 100 000 visites par mois, 10 pages par visite, on obtient $W = 2,77$ pages par seconde à l'heure de pointe, ce qui devrait être supporté sans difficulté par un serveur unique.

Nous sommes passés des visites aux pages de manière un peu rapide. En fait, tout dépend ici encore de la typologie du service. Si une visite type se compose d'une page de recherche, de trois pages de saisie et de 5 pages de consultation, alors c'est ce scénario qui importe. Et l'on calculera plutôt le nombre de recherches par seconde, le nombre de saisies par seconde et le nombre de consultations par seconde, avec des impacts très différents sur la base de données par exemple.

Notons que dans ces calculs, le délai moyen entre deux requêtes n'a pas été utilisé. En effet, il n'intervient pas à ce niveau : si les internautes naviguent plus rapidement, ils sollicitent davantage le serveur, mais pendant moins longtemps.

Connexions simultanées

On entend parfois raisonner en termes de connexions simultanées. C'est une notion qui n'est pas la plus appropriée pour le web, mais qui peut avoir un sens, par exemple s'il y a des ressources mobilisées pour la durée d'une visite, d'une session. Chaque visite a une durée égale à $D.P$, délai moyen entre pages multiplié par nombre de pages par visite.

A l'heure de pointe, on a donc $V_h.D.P$ « secondes de visite », sur 3600 secondes, donc un nombre moyen de sessions simultanées égal à $S = V.D.P/3600$, c'est à dire aussi $S = W.D$.

Application numérique : dans le cas précédent, et pour un délai moyen de 30 secondes, on trouve $S = 2,77 \times 30 = 83$ sessions simultanées.

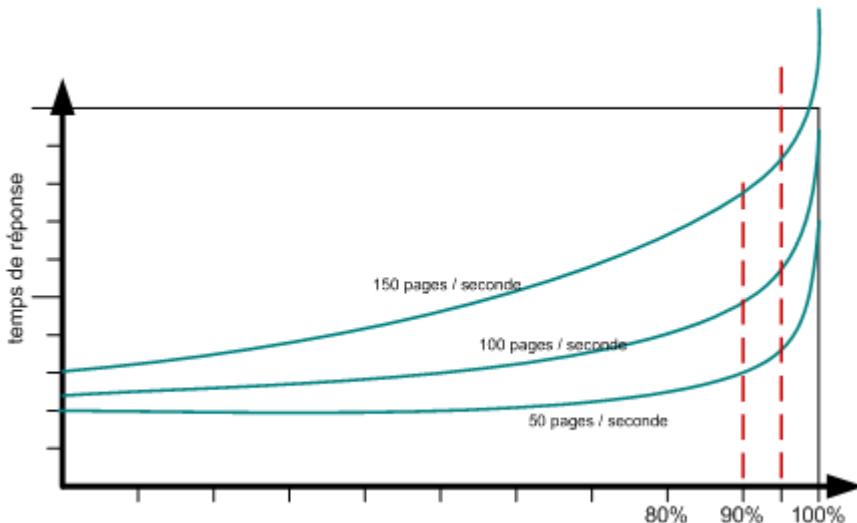
Temps de réponse

Lorsque l'on parle de temps de réponse observés, il faut toujours préciser « à XX% », typiquement à 90 ou 95%.

En effet, lorsqu'un grand nombre d'internautes soumettent des requêtes, de manière totalement indépendante, il est possible que 10 d'entre eux cliquent sur un lien dans un même petit intervalle de temps, disons 1/100^{ème} de seconde. S'il y a peu de trafic au total, alors la probabilité de cette coïncidence est faible, mais toujours non nulle. Et si cela se produit, alors au moins une partie de ces 10 requêtes sera servie plus lentement.

Sans entrer ici dans des calculs de probabilité poussés, on voit bien qu'on ne peut jamais garantir un temps de réponse à 100%, car il peut toujours y avoir des coïncidences rares de trafic qui dégradent exceptionnellement le temps de réponse.

Si l'on conduit un test de forte charge, que l'on relève chacun des temps de réponse observés, et que l'on range ces temps de réponse du plus court au plus élevé, alors on obtient en général une courbe qui a sensiblement la forme suivante :



Où l'on voit que les courbes tendent vers une asymptote verticale vers la droite, correspondant aux conjonctions extrêmes. Ces cas extrêmes ne sont pas les plus intéressants, ce qui évalue le mieux la qualité de service est le temps de réponse à 90%, ou encore à 95%.

Méga-serveurs ?

En matière de hautes performances, on peut distinguer schématiquement deux voies : une voie centralisée et une voie distribuée.

La voie centralisée consiste à rechercher un serveur ayant une très grande capacité de traitement, donc un très grand nombre de processeurs très rapides, beaucoup de mémoire et de disques. C'était la voie privilégiée dans les années 90, et elle correspond encore à une offre aujourd'hui, de la part des plus grands constructeurs. On peut trouver ainsi des serveurs de 64 processeurs, à des prix astronomiques bien sûr.

La voie distribuée consiste au contraire à bâtir son architecture à base d'un grand nombre de serveurs peu coûteux, relativement indépendants les uns des autres.

On a vu la même scission se produire dans le calcul scientifique, où l'approche « Cray » des années 90 a finalement été surpassée par l'approche distribuée.

S'il peut subsister quelques domaines où le méga-serveur central a encore un marché, il est rarement approprié dans les grandes architectures web. Les plus grands acteurs (Google, Amazon, eBay, ...) montrent le chemin, et aucun d'entre eux ne choisit de s'appuyer sur un ou plusieurs de ces méga-serveurs.

Nous ferons le même choix, et ne nous intéresserons pas à ce type d'architectures dans cet ouvrage.

L'approche méga-serveur peut être la plus simple pour obtenir des hautes-performances sans se préoccuper d'architecture. Mais c'est de loin la démarche la plus coûteuse. Par ailleurs, ces configurations sont trop rares pour être bien rodées, et surtout les compétences pour les exploiter sont trop rares, de sorte qu'elles peuvent amener des difficultés spécifiques par suite d'une maîtrise insuffisante.

Les méga-serveurs ne sont pas véritablement extensibles, ou du moins ils ne le sont que dans la mesure des « slots » prévus pour ajouter des processeurs. Au delà, on rencontre une limite, et c'est alors une limite « dure », infranchissable.

Coûts

On ne peut pas traiter de l'architecture en faisant abstraction des coûts : en général toute la réflexion est sous la contrainte du coût, c'est à dire vise à optimiser le rapport capacité/prix global.

L'analyse des coûts inclut en premier lieu :

- Les coûts matériel
- Les coûts d'hébergement

Mais aussi :

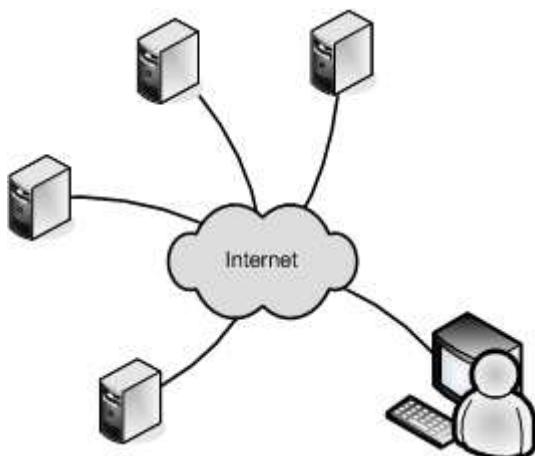
- Les coûts d'exploitation
- Le prix de l'indisponibilité ou plus largement de la moindre qualité de service
- Les coûts de développement et de maintenance logicielle.
- Les prix de licences, le cas échéant.

Nous verrons plus loin que l'on privilégie le plus souvent les architectures à base de composants ordinaires, peu coûteux. Mais aussitôt qu'il y a arbitrage entre coût matériel et coût de prestation, le prix du matériel devient vite négligeable.

Quelques principes pour la haute performance

Voyons quelques principes généraux de la haute-performance, que nous retrouverons ensuite au travers des différents chapitres.

Share-Nothing



« *Share nothing* », signifie « ne rien partager ». Ce que l'on pourrait représenter schématiquement comme ci-dessus, où l'utilisateur internaute peut obtenir le service attendu en s'adressant à n'importe lequel de ces serveurs, indifféremment, et sans que ceux-ci ne partagent quoi que ce soit.

Le principe de *share-nothing* est un but théorique, pas toujours atteignable. C'est la voie vers une extensibilité sans limite, ainsi que la voie vers la très haute disponibilité à moindre coût.

« Simple is beautiful »

« *Le simple est beau* », c'est à dire que l'on doit toujours rechercher les solutions les plus simples, qui seront généralement les moins coûteuses, mais aussi les plus fiables.

Pas de solution unique

L'architecte ne doit pas se rabattre toujours sur les mêmes recettes, toujours utiliser les mêmes outils. Même s'il vise une certaine uniformité de sa plateforme, il doit avoir une palette de solutions, et savoir choisir la plus appropriée. Il ne doit pas, par exemple, utiliser une base de données là où un système de gestion de fichiers conviendrait mieux ou encore un transfert de fichier lorsque son besoin est d'un vrai middleware.

Les outils évoluent

Les architectes doivent trouver un équilibre entre bonnes vieilles recettes et solutions nouvelles. Certes, certaines solutions anciennes sont encore appropriées, mais l'informatique bouge vite, et des outils plus modernes arrivent rapidement à maturité, apportant de réels progrès, de meilleures solutions.

Le méga-truc n'est pas la solution

Le méga-serveur avec des dizaines de processeurs, n'est pas la bonne voie pour une plateforme web hautes-performances. De même que le méga-switch, ou le méga-SAN. En règle générale, le méga-truc n'est pas la bonne voie.

Le poste client est plein de ressources

Le poste client, c'est à dire le poste de l'utilisateur, avec son navigateur, a beaucoup de possibilités mal exploitées. Il dispose de beaucoup de CPU, de la mémoire, des possibilités de stockage, etc. Certaines des choses que l'on fait côté serveur peuvent être déportées côté client, avec quelques bénéfices.

L'open source apporte beaucoup de solutions

Dans le domaine de l'infrastructure, l'open source règne en maître. Quand on exploite des milliers de serveurs, comme les plus grands acteurs du web, on est évidemment sensible aux solutions open source, au plan économique d'une part, mais aussi pour leur robustesse à toute épreuve.

URBANISME ET SOA**Urbanisme**

L'urbanisme est une démarche d'architecture apparue dans les années 90, qui vise à mieux maîtriser un vaste système d'information (SI) en le décomposant en sous-systèmes indépendants, échangeant par des interfaces bien définies et stables.

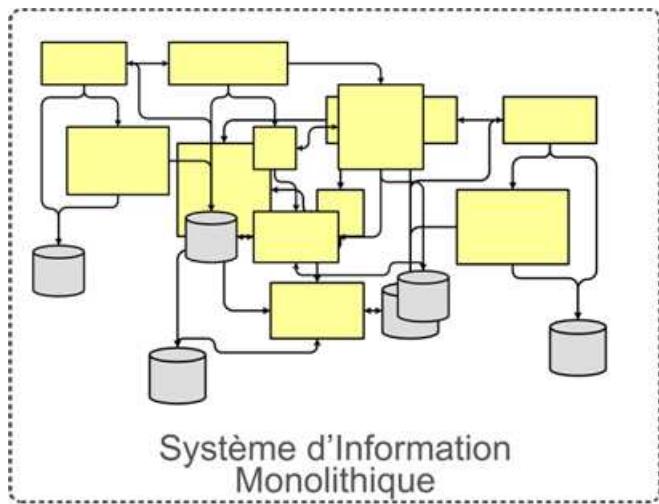
L'objectif principal de l'urbanisme est donc de *maîtriser la complexité et l'immensité*, et de rendre possibles des refontes partielles, portant sur un sous-système à la fois.

En effet, les projets de refonte de grands systèmes d'information échouent, non pas parfois, mais *le plus souvent*. Simplement parce que ce sont des projets trop complexes et surtout trop longs. Pourtant, les grands SI ne peuvent pas rester, tels de grands dinosaures, à l'écart de l'évolution.

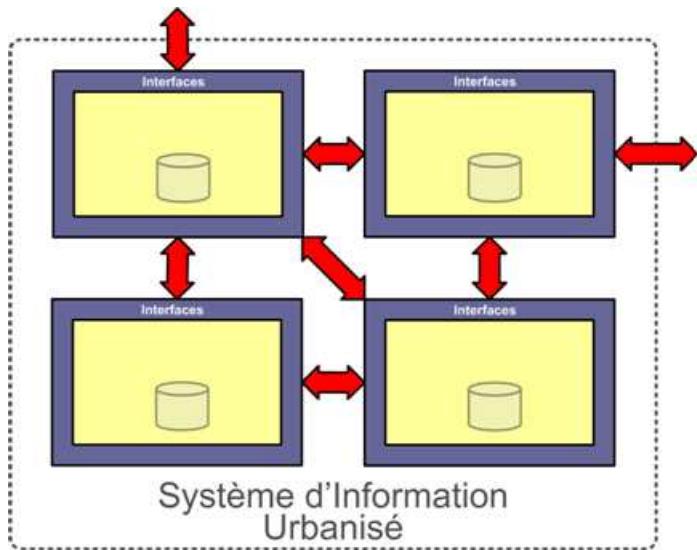
Certes, le rêve d'un architecte en systèmes d'information est toujours d'atteindre la parfaite homogénéité technologique, en même temps qu'une modélisation globale unifiée. Mais c'est un but impossible, auquel il vaut mieux renoncer.

L'urbanisme consiste en somme à prendre de la hauteur, et à délimiter des périmètres, des « quartiers », au sein desquels la complexité est maîtrisable, et pour lesquels on peut viser une certaine homogénéité locale.

Ce que l'on peut représenter comme suit :



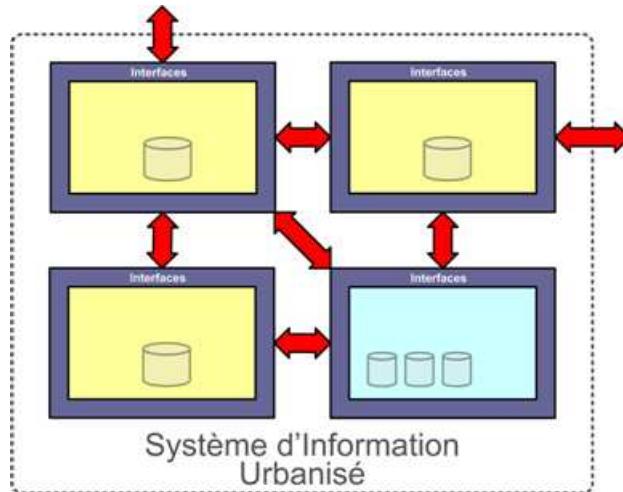
Ici, le système d'information monolithique, avec ses programmes et bases d'information entremêlés, interfacés de manière croisée, tel un plat de spaghetti. On ne peut refondre un programme ou changer la modélisation d'une base d'information sans des impacts nombreux, parfois mal définis, sur de multiples autres modules.



Ici, le système d'information urbanisé : on y distingue des sous-systèmes indépendants, échangeant uniquement selon leurs interfaces.

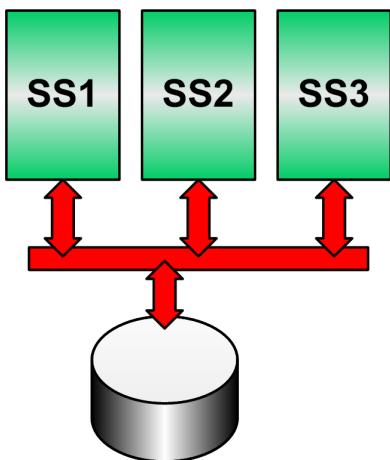
Dans cette architecture, on peut envisager la refonte d'un sous-système que ce soit pour adopter de nouvelles technologies ou pour répondre à un besoin fonctionnel nouveau. Du moment que ses interfaces avec les autres sous-systèmes sont préservées, le périmètre de la refonte est maîtrisé.

Ce que l'on peut représenter comme suit, avec l'un des sous-systèmes refondus :



Encapsulation des données

La base de données a longtemps été vue comme l'articulation, le moyen de partage et même d'échange, entre sous-systèmes, ce que l'on peut représenter comme suit :

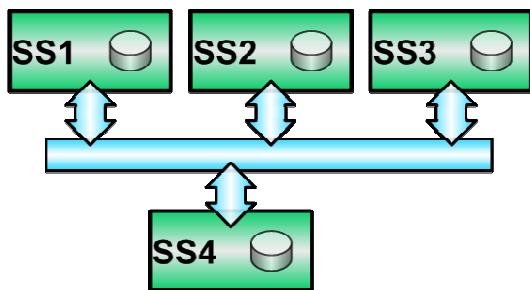


Dans ce modèle, les interfaces entre les sous-systèmes, ici SS1, SS2, SS3, transitent par la base de données, qu'ils partagent, l'un y versant des données, l'autre les lisant.

Ce modèle n'est pas bon, il crée des dépendances bien trop fortes entre les sous-systèmes et la base de données, donc entre les sous-systèmes eux-mêmes.

Le langage SQL et les protocoles d'interrogation de la base sont certes une forme d'interface, mais une interface bien trop vaste, et trop dépendante de l'implémentation et par ailleurs trop peu standardisée.

Les architectures modernes retiennent au contraire le principe d'encapsulation des données : les données, leur modélisation et leurs outils de stockage sont purement internes à un sous-système, totalement invisibles de l'extérieur, ce que l'on peut représenter comme suit.



Dans ce modèle, les données de chaque sous-système ne peuvent pas être manipulées ou accédées par d'autres sous-systèmes autrement qu'au moyen des interfaces, des services qui sont exposés.

Il peut s'agir de services simples, de type « CRUD », Create, Read, Update, Delete, ou bien de services de plus haut niveau, davantage orientés métier.

L'urbanisme et les plateformes web

En général, les plateformes web sont plus simples que des systèmes d'information d'entreprise. Certes, le SI de Amazon.com est plus vaste que celui d'une PME, mais encore sensiblement moins que celui d'une grande banque par exemple.

Par ailleurs, les plateformes web sont plus récentes, et donc plus homogènes que la plupart des systèmes d'information. Elles ne traînent pas un patrimoine historique de 10 ans d'âge et plus.

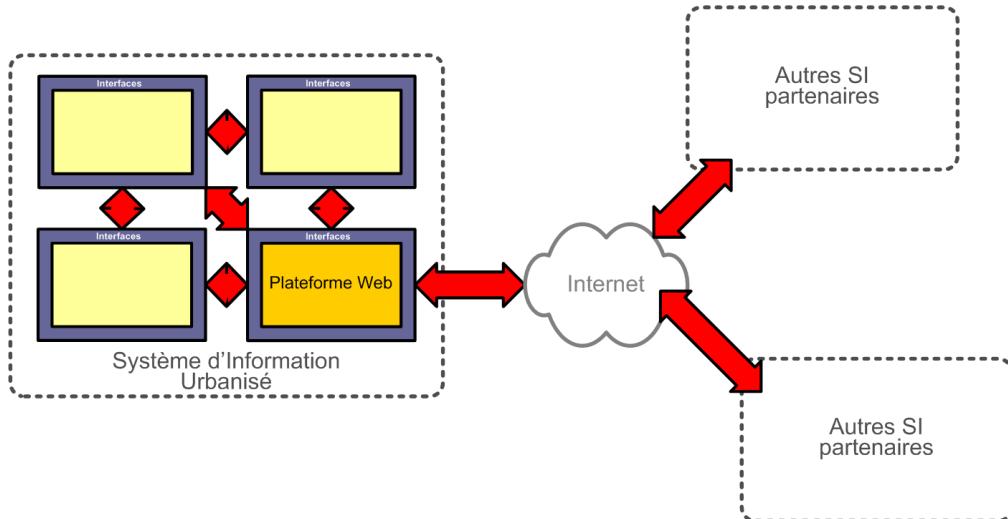
Ainsi, une plateforme web n'est pas confrontée aux mêmes problèmes qu'un système d'information complet, et n'a pas l'usage des mêmes axes de solutions. Néanmoins, après quelques années, une plateforme web peut-être confrontée au même problème : on a construit une sorte de Tour de Babel, et on ne parvient plus à la faire évoluer.

A partir d'une certaine dimension, la refonte d'un grand site web est un projet de plus d'un an. Pendant la refonte, les affaires continuent, et les évolutions continuent également, de sorte que l'on est tenu d'avancer avec des spécifications mouvantes, et de faire bouger simultanément l'ancien et le futur. Le résultat est que les refontes sont difficiles, voire douloureuses, et il arrive souvent que l'on n'ose plus engager une refonte, laissant son socle technologique vieillir peu à peu.

Notre conclusion est que les plateformes web ont un réel besoin des principes d'urbanisme. De manière moins systématique ou moins ambitieuse peut-être, mais construire une plateforme urbanisée est le gage d'une réelle évolutivité.

A cela il faut ajouter que la frontière entre « Système d'Information » et « Plateforme web » devient de moins en moins marquée. Dans une logique d'extranets, ce sont tous les systèmes d'information d'une

entreprise et de ses partenaires économiques qui travaillent en réseau, et la plateforme web n'est alors qu'un sous-système parmi d'autres.



Service Oriented Architecture

SOA ou « *Service Oriented Architecture* » est un modèle d’architecture fondé sur la notion de service.

Un service est une fonction qui peut être invoquée à distance, par un humain ou bien par un programme.

SOA et urbanisme sont indissociables, les *services* étant une représentation des interfaces. SOA est une déclinaison des principes d’urbanisme, à une échelle intermédiaire.

Comme l’urbanisme, l’approche SOA est souvent ignorée par les plateformes web, et c’est regrettable. Le seul middleware mis en œuvre est celui qui interface la base de données, et trop souvent on ne connaît qu’appels de fonctions ou méthodes.

Pourtant, comme l’urbanisme, SOA est un modèle qui doit trouver sa place dans une plateforme web un peu ambitieuse, apportant de réels bénéfices, en particulier d’évolutivité, fonctionnelle et technologique.

Un Service

Qu'est-ce précisément qu'un service ?

Dans le contexte SOA, un service a les caractéristiques suivantes :

Architectures Hautes-Performances

- Il est d'une granularité moyenne : ce n'est pas une simple routine, mais pas non plus un applicatif complet.
- Un service a vocation à être réutilisé, il n'est pas dédié à une utilisation unique.
- Un service est technologiquement neutre, il peut être invoqué par toutes sortes de programmes.
- Un service est potentiellement accessible depuis l'extérieur du système d'information, depuis l'Internet. Cela sous réserve bien sûr de permissions et contrôle d'accès.

Ce dernier point est particulièrement important au sein des plateformes web :

Il faut partir du principe que l'on ne sait pas à l'avance quels services devront être accessibles depuis l'extérieur de la plateforme, et donc implémenter tous les services de manière à ce qu'ils puissent l'être.

Un service est défini par son *contrat de service* et *d'interface*, c'est-à-dire ce qu'il promet au reste du monde. Le service masque totalement son implémentation, il fonctionne en mode boîte noire. Il peut donc changer son implémentation sans impact pour ses clients, tant que son interface est inchangée.

Le service ne doit pas être vu uniquement comme « plus ou moins un appel de fonction » ; en particulier il peut être synchrone ou asynchrone, ou bien adopter d'autres modèles d'appel comme on le verra plus loin.

Middleware

Le middleware est l'outil qui permet d'implémenter les interfaces de manière standard.

Il est extraordinaire que beaucoup de plateformes web ne se préoccupent pas de middleware, et ne connaissent que les appels à leur base de données.

Pourtant, il est temps d'introduire du middleware dans les plateformes web comme on l'a fait dans les SI.

L'une des exigences en matière de middleware est la neutralité technologique. Au sein d'un unique environnement technologique, on dispose d'outils spécifiques pour mettre en œuvre des appels distants, par exemple le RMI dans le cas de l'environnement Java. Mais dans une architecture SOA, les modalités d'appel doivent être

technologiquement neutres, c'est-à-dire qu'un service doit pouvoir être appelé depuis n'importe quel environnement (langage, système d'exploitation, *framework*).

Par essence, le middleware doit être interfacé avec toute forme de composants, c'est donc un domaine où les standards sont d'une importance extrême. Un excellent middleware propriétaire, aux interfaces non-standard, serait sans utilité. Le middleware est nécessairement structurant vis à vis des programmes qui l'utiliseront, mais s'il est respectueux de standards, il ne crée du moins pas de dépendance vis à vis d'un vendeur ou d'un produit unique.

Par ailleurs, le middleware vise comme on l'a vu à permettre des interfaces externes aussi bien qu'internes. Les interfaces externes ont des exigences bien sûr en termes de sécurisation, mais aussi de standardisation des formats, des représentations de l'information.

Les modes d'interaction

Les développeurs, qui ont d'abord appris le bon vieil appel de fonction ont tendance à raisonner uniquement en termes d'appels synchrones. Mais l'architecte et le concepteur d'application doivent avoir à leur disposition une diversité de logiques d'échange, et c'est ce que doit leur offrir le middleware.

Passons en revue les principales possibilités.

Synchrone

En mode synchrone, l'appelant est bloqué en attente de la réponse à son message, ce qui correspond à un appel de fonction à distance. Bien sûr, on est généralement dans un contexte multi-process ou multi-thread, de sorte qu'il y a toujours du travail pour occuper le processeur. Mais du moins la logique de l'interaction est bien synchrone, bloquante.

Asynchrone, one-way (aller-seul)

Dans ce mode, l'appelant adresse son message de requête et poursuit son traitement. Il peut demander un acheminement garanti ou non. C'est par exemple le cas d'une notification vers un système de supervision.

Asynchrone with callback

Ici, l'appelant adresse son message et poursuit son traitement sans attendre, mais il spécifie un service à invoquer lorsque la réponse parviendra, c'est ce service qu'on appelle « callback ». C'est donc un

mode asynchrone aller-retour, qui permet de ne bloquer ni l'appelant ni l'appelé.

Asynchrone, publish / subscribe (publication / abonnement)

Dans ce mode, un ou plusieurs services dits « consommateurs » s'abonnent à un flux de messages et le service « producteur » publie ses messages sur le flux. C'est en quelque sorte de l'asynchrone one-way, mais à destinataires multiples. Mais ce n'est pas le service émetteur qui désigne les destinataires, ce sont les destinataires qui s'inscrivent, ce qui est très différent.

Les traitements asynchrones

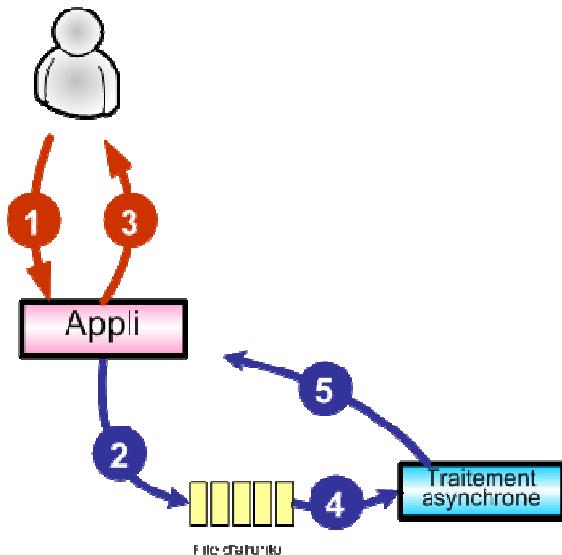
Dans le web, la tentation est de tout faire en synchrone, que ce soit par souci de rendre compte à l'internaute, ou bien – plus encore – parce que c'est plus simple pour le développeur.

Pourtant, une des voies vers la haute-performance consiste à accepter un peu d'asynchronisme dans l'exécution.

Que ce soit avec ou sans middleware (mais on préférera avec), le principe est simple :

- On met en place une file d'attente (FIFO, « *First In, First Out* ») des tâches à exécuter
- Le traitement synchrone ajoute une tâche à la liste
- Un ou plusieurs programmes « déplient » les tâches et les exécutent.
- Selon les cas, il peut être nécessaire de gérer des compte-rendu d'exécution, et de les présenter à l'utilisateur, que ce soit au sein des interfaces web, ou bien par d'autres canaux, e-mail par exemple.

Ce que l'on peut représenter comme suit :



L'asynchronisme permet à la fois de rendre l'interface utilisateur plus réactive, puisqu'on ne laisse jamais l'utilisateur en attente, et permet également de *lisser le travail*, et donc de beaucoup mieux gérer des pointes de charge, sans être obligé de surdimensionner l'architecture.

L'asynchronisme limite aussi les dépendances, et peut donc amener une architecture plus robuste.

Bien sûr, si l'on a mis en place un middleware qui prend en charge cette gestion, tout cela deviendra facile. Néanmoins, il arrive que l'on gère la file d'attente au sein de la base de données, avec des process qui écrivent dans une table des tâches, tandis que d'autres process lisent les paramètres de la tâche, l'exécutent et la suppriment.

Un cas particulier classique d'une gestion qui ne doit pas être synchrone est l'envoi de mail. Même si l'on dispose de fonctions synchrones d'envoi de mail, très simples à utiliser, il n'est jamais bon de les appeler en synchrone. C'est une règle d'or des plateformes web : les mails doivent être émis par une tâche asynchrone.

MOM et Message Queues

L'une des formes intéressantes de middleware est la famille des MOM, *Message-Oriented Middleware*, middleware à base de messages donc.

Dans ce modèle, une application interagit avec une autre application en lui adressant des messages, et le MOM se charge d'acheminer les messages de l'une à l'autre.

Architectures Hautes-Performances

En fait, ce n'est pas tant l'aspect *message* qui caractérise le MOM, que l'aspect asynchrone, et quelques autres possibilités, que nous verrons plus loin.

Une caractéristique déterminante d'un MOM est la fiabilité, c'est à dire la garantie d'acheminement. Pour l'application émettrice, c'est un confort : elle remet le message au MOM, et elle passe à autre chose, en ayant l'assurance que le message parviendra à son (ses) destinataire(s). Les MOM ont donc presque toujours une forme de stockage persistant sécurisé, qui peut être une base de données. Même si le serveur de l'application émettrice s'arrête, les messages seront conservés jusqu'à ce qu'ils puissent finalement être acheminés. Le dispositif a donc par construction une bonne tolérance aux pannes.

La principale finalité du MOM est de permettre ce qu'on appelle un *couplage lâche*, ou *faible*, entre les applications qui interagissent, c'est-à-dire qu'elles dépendent peu les unes des autres. L'asynchronisme en est un aspect, mais l'identification de la file d'attente est également importante, elle crée une indirection dans la relation entre producteur et consommateur de messages : l'un et l'autre identifient la file d'attente, mais ni l'un ni l'autre n'a besoin de connaître son 'partenaire'.

Il faut souligner aussi que même si *dans le principe* le message peut mettre un temps indéfini à être remis, *dans la pratique* cela peut être extrêmement rapide, quelques millisecondes à peine. Mais même si l'aller-retour en MOM peut se faire de manière quasi-immédiate, on ne doit jamais compter dessus, et par exemple laisser un internaute en attente d'un aller-retour MOM.

Le MOM est un outil souvent méconnu et donc sous-utilisé au sein des plateformes web.

Les messages ne transitent pas d'un MOM à un autre de manière transparente. Il n'y a guère de standard en matière de MOM, de sorte qu'il devient structurant pour une architecture. Par ailleurs, contrairement à REST (que l'on évoquera plus loin), qui ne requiert qu'un serveur web ordinaire, un MOM est un vrai produit, un composant supplémentaire à déployer dans votre infrastructure.

Des services sans état

Un service sans état, *stateless*, est un service *qui ne se souvient de rien entre deux appels*. C'est le cas du HTTP, nous y reviendrons plus loin.

Un fonctionnement sans état présente de nombreux avantages :

- Il n'implique pas de conserver de l'information, donc de réserver des ressources, sur des durées indéterminées.
- Il n'est pas nécessaire de fermer une session, et donc il n'y a pas de risque d'oublier de fermer une session qui devrait l'être.
- Chaque invocation du service étant totalement indépendante des précédentes, on a une grande flexibilité dans la gestion du *load-balancing* et du *failover*.

Ce sont des arguments forts, et l'on peut retenir qu'il faut toujours privilégier un fonctionnement sans état.

Il y a malgré tout certains services qui, au plan fonctionnel, ne sont pas compatibles avec un tel fonctionnement, mais ils sont très rares.

Trois protocoles pour les services web

Il existe principalement trois protocoles standards pour mettre en œuvre des services web : XML-RPC, REST et SOAP.

Soulignons tout d'abord qu'en matière de middleware – comme dans d'autres domaines d'ailleurs – on recherche un compromis entre la perfection théorique du protocole et sa simplicité. C'est ainsi que CORBA, un middleware d'interopérabilité des années 90, n'a pas eu le succès escompté, car manquant de légèreté. Les architectes du web, en particulier, privilégient la simplicité et la performance (voir par exemple la défense des *Plain Old Java Objects* sur les EJB) et il est fréquent que le marché leur donne raison.

XML-RPC

XML-RPC est historiquement le premier des protocoles de services web, défini dès 1998.

Il est construit sur HTTP, et s'appuie sur une mise en forme Xml de tous les paramètres de l'invocation du service, et de la réponse, selon une syntaxe générique très simple.

Les variables supportent une dizaine de types, tels que *int*, *float*, *double*, *string*, *date-time*, *boolean*, ainsi que des types plus complexes, matrices et structures.

Considérons un exemple d'appel XML-RPC :

```
<?xml version="1.0"?>
<methodCall>
    <methodName>examples.getDeptName</methodName>
    <params>
        <param>
            <value><i4>27</i4></value>
        </param>
    </params>
</methodCall>
```

Ici, on appelle un service « *getDeptName* » qui retournera le nom d'un département en fonction de son numéro. Ce service accepte un paramètre unique : le numéro du département, ici le 27.

Et voyons sa réponse :

```
<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value><string>Creuse</string></value>
        </param>
    </params>
</methodResponse>
```

Ici le service a fourni une réponse constituée d'une simple chaîne de caractères : « *Creuse* ».

On constate sur cet exemple que, tant dans la requête que dans la réponse, la modélisation de l'information est générique, n'utilisant que des balises *<param>*, *<value>*, etc, et non pas une syntaxe adaptée à la sémantique du service. C'est un choix simplifiant, mais limitant. Typiquement, il n'est pas possible de valider la syntaxe d'une requête, ou bien d'une réponse, autrement que dans le service lui-même.

REST

REST est un modèle d'échange basé également sur HTTP, le protocole du web. Il n'utilise que les verbes standards du HTTP : GET, POST, PUT, DELETE.

REST est un protocole sans état. Le fait qu'il soit basé sur HTTP le rend très facile à mettre en œuvre dans une infrastructure web, puisqu'il ne demande rien d'autre qu'un serveur web ordinaire. Autrement dit : n'importe quelle application web, quel que soit le langage, peut implémenter un service REST.

Chaque service REST est accédé au moyen d'une URI, qui comprendra l'indication du service et ses paramètres, par exemple :

```
HTTP://www.woozweb.com/lirecompte/account-id/8992871100
```

Ici, on invoque un service LireCompte de l'application « Woooze.com », le service fournit des informations relatives à un compte utilisateur, dont l'identifiant est spécifié.

Notons que la syntaxe du REST en lecture masque parfois l'aspect « invocation d'une fonction », pour une logique davantage orientée « accès à une ressource ». Dans cette voie, l'URI précédente pourra devenir simplement :

```
HTTP://www.woozweb.com/accounts/8992871100
```

On voit que c'est sensiblement plus simple qu'un appel XML-RPC.

Cette simplicité, et la relative lisibilité des interfaces, le rend également très simple à tester, que ce soit « à la main », en saisissant des URIs, ou bien au moyen d'un outil de test ou d'un robot.

REST est un protocole qui vise la performance, et à ce titre il est plus adapté aux plateformes web, en particulier par rapport à SOAP. C'est par ailleurs un protocole « cachable », c'est-à-dire que le résultat d'un GET (lecture) présentant les mêmes paramètres, donc la même URI, pourra être considéré valable pendant la durée de vie spécifiée, exactement à la manière d'une page web.

Voyons maintenant les formats de réponse.

En fait, REST ne donne pas de règle concernant le format de réponse. Dans certains cas, un service REST peut fournir différentes représentations possibles de sa réponse, et c'est le client qui spécifie la représentation demandée, en renseignant l'attribut « accept » du header HTTP, par exemple accept=text/html, ou bien accept=text/xml.

Mais ce n'est pas vraiment recommandé. Plutôt que de faire des services multi-interfaces, on préférera construire des services purement « orientés machines », pour lesquels on choisira évidemment une représentation Xml, ou dans quelques cas JSON pour s'interfacer directement à un client en Ajax. Et l'on pourra ensuite construire une couche d'interface par-dessus ce service « orienté machine », couche d'interface qui peut être en forme de service REST également (voir page 34).

Notons que ni REST ni XML-RPC n'exposent leurs interfaces, c'est-à-dire ne fournissent un moyen, pour un programme client, de découvrir les spécifications de l'interface. En interne à une plateforme, c'est une limitation peu contraignante, les développeurs peuvent lire les spécs. Mais il est vrai que dans une approche SOA généralisée, ouverte sur l'extérieur, la publication du service et l'exposition de ses interfaces

sont importantes. Dans les faits toutefois, il est pratiquement impossible de construire un programme client qui ait l'intelligence requise pour (a) découvrir les services proposés et leurs interfaces, (b) choisir un service approprié, (c) construire une requête adaptée à l'interface de ce service, et (d) analyser la réponse et utiliser les informations fournies. L'exposition des interfaces n'est pas tant tournée vers des programmes aussi intelligents, elle est plutôt destinée à s'intégrer à des processus et outils de développement.

SOAP

Le protocole SOAP est le descendant de XML-RPC, auquel il ajoute quelques qualités, en premier lieu la diversité des protocoles sous-jacents, qui peuvent être SMTP, HTTP, ou bien des protocoles de type MOM, *Message Oriented Middleware*. C'est donc le plus compatible avec un fonctionnement asynchrone, même si on peut créer des interfaces REST vers un MOM.

SOAP se distingue aussi de XML-RPC par la représentation des données. Elle est aussi basée sur XML, mais peut utiliser des formats plus complexes ainsi que des *namespaces* spécifiques, ce qui permet une représentation sémantique des données, ainsi qu'un contrôle de conformité intégré.

Un exemple de format de requête :

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="HTTP://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="HTTP://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <m:GetAvailability xmlns:m="http://www.smile.fr/xmlns/woozweb">
      <m:Ressource>www.voyages-sncf.com</m:Ressource>
    </m:GetAvailability>
  </soap:Body>
</soap:Envelope>
```

On observera en particulier l'introduction d'un namespace spécifique à l'application, ici « m : », dont la définition est donnée en référence, <http://www.smile.fr/xmlns/woozweb>.

Et sa réponse:

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="HTTP://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="HTTP://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <m:GetAvailabilityResponse
      xmlns:m=" http://www.smile.fr/xmlns/woozweb ">
      <m:Availability>99.96</m:Availability>
    </m:GetAvailabilityResponse >
  </soap:Body>
```

```
</soap:Envelope>
```

De la même manière ici, le prix n'est pas juste un entier, c'est une variable de type `m:Availability`.

On observe aussi, sur cet exemple, que tout cela est un peu verbeux, mais c'est le prix à payer pour la flexibilité, l'extensibilité, et l'interopérabilité.

Il faut souligner toutefois que cette syntaxe des messages n'est pas à gérer par le programmeur lui-même. C'est le bénéfice de la standardisation : les outils de développement prennent en charge le plus gros du travail. Ainsi dans de très nombreux environnement, transformer une méthode ordinaire en un service SOAP est pratiquement immédiat et transparent.

SOAP est un protocole plus complet et plus complexe que les précédents. Il requiert un serveur spécifique, par exemple AXIS de Apache, et non un simple serveur web. Il est en général moins performant que REST.

SOAP est en particulier peu approprié pour des échanges internes à une plateforme, et convient davantage aux interfaces externes. Il est vrai – on l'a souligné plus haut – que l'on ne sait pas toujours quelles interfaces que l'on croyait internes seront un jour à exposer en externe.

Services et interfaces

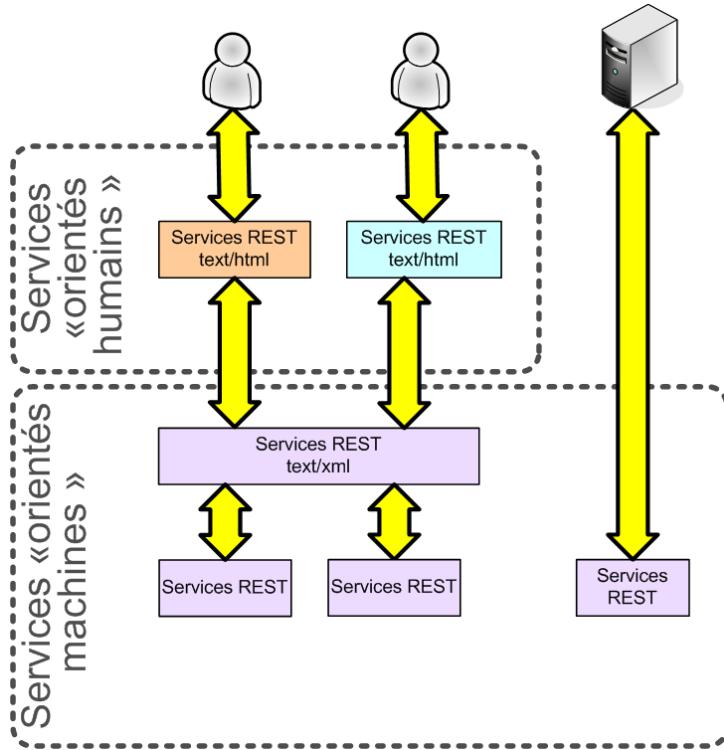
Dans les débuts du web, on ne s'adressait pratiquement qu'aux humains. Il arrivait parfois que des « machines », i.e. des programmes, aient besoin de l'information fournie par des sites ou applications web. Les machines se faisaient alors passer pour des humains, à la manière de Kelkoo allant collecter des informations de prix dans les pages web des sites marchands. Cette technique, de *screen scraping*, ou *web scraping*, ce qui signifie « gratter les pages web » pour y trouver des données, était laborieuse, instable et fragile. Le moindre changement de charte graphique obligeait à reconfigurer la collecte.

Ce besoin d'utilisation du web par des machines et non par des humains s'est étendu, et aujourd'hui, un principe de base des architectures web est que n'importe quel service devrait pouvoir être appelé par un programme.

Plutôt que de bricoler des interfaces « orientées machines » au dessus d'interfaces natives « orientées humains », on préfère faire le contraire, c'est-à-dire implémenter d'abord des interfaces nativement orientées machines, et y ajouter si besoin des interfaces destinées aux humains.

Architectures Hautes-Performances

Ce qui peut se représenter sur le schéma suivant :



Dans le périmètre du bas, nous avons une application web, avec des services internes REST, tous purement orientés machines (text/xml). Et au dessus de ces services on construit une couche d'interface, avec des services orientés humains.

L'application web moderne est donc construite en deux couches (au moins) :

- Une couche *services*, aux interfaces « orientées machines »
- Une couche *interfaces*, habillant la couche services, à destination des humains.

La couche service est généralement REST ou bien SOAP.

C'est en particulier le modèle retenu par les progiciels modernes, qui se généralise depuis quelques années :

- Les progiciels encapsulent totalement leurs données – il est donc interdit d'accéder à leur base de données.
- Ils fournissent des services
- Ils peuvent être utilisés aussi bien par des humains que par des programmes.

Il faut le souligner, c'est la condition d'une réelle capacité à intégrer ces progiciels au reste du système d'information. S'il s'agit d'une application de RH, elle exposera des services permettant d'accéder au dossier d'un employé, de créer un employé, de modifier le poste ou le salaire, etc.

C'est aussi ce qui permet de totalement refondre la couche interface d'un progiciel, que ce soit pour une utilisation requérant une ergonomie spécifique, ou bien simplement pour une harmonisation graphique.

MOM open source – Apache ActiveMQ

Il existe un petit nombre d'outils MOM de qualité en open source. On peut citer ActiveMQ de Apache, JBoss Messaging, maintenant de Redhat, ou encore Joram, de ObjectWeb, ou encore Glassfish Open Message Queue, de SUN.

Lorsque Apache propose un bon produit, il s'impose souvent. Intéressons nous donc plus particulièrement à ActiveMQ.

ActiveMQ présente des APIs pour pratiquement tous les langages et environnements : C, C++, C# / .Net, Delphi, Flash / ActionScript, JavaScript, Perl, PHP, Python, etc.

ActiveMQ offre aussi des interfaces REST pour l'émission et la réception de messages, ce qui est facile à interfaçer dans n'importe quel environnement, mais n'offre pas la richesse d'une interface de type JMS. La publication d'un message utilise un POST http, et la lecture un GET ou un DELETE.

La persistance des messages peut s'appuyer sur n'importe quelle base de données, accédée en JDBC, avec par défaut Apache Derby, qui est une base full-java. Depuis la version 5, la persistance est assurée par défaut sur un *message store* spécifique, AMQ Message Store, qui est plus rapide qu'une base de données, car spécialisé sur sa mission plus simple.

PERFORMANCES HTTP

En matière d'architecture de plateforme web, on s'arrête généralement aux frontières du datacenter. Il s'agit de servir des pages le plus rapidement possible, et ce qu'il advient de ces pages passé le routeur ne nous intéresse guère.

Pourtant c'est loin d'être négligeable, et c'est pourquoi nous accordons un chapitre à la question des performances des échanges HTTP entre plateforme web et navigateur, et même jusqu'à la restitution finale à l'utilisateur.

Certaines de ces considérations ont uniquement un impact sur la performance perçue, mais d'autres peuvent impacter aussi la capacité d'accueil de la plateforme.

Chronologie de chargement de page

C'est une chose généralement connue, mais dont les détails et les implications ne sont pas toujours bien mesurés : une page web est constituée de nombreux composants, qui sont chargés les uns après les autres par le navigateur.

Au départ, le plus souvent, il y a une page faite de Html. Cette page contient des références à d'autres objets, chacun identifié par son URI : des images, des feuilles de style, des fichiers javascript, des objets Flash, d'autres fichiers Html, ou encore des fichiers Xml utilisés par des composants Ajax, et quelques autres genres de composants. Certains de ces objets sont cherchés sur d'autres serveurs, typiquement les bannières de publicité ou bien les invocation des services de mesure d'audience.

Comment tous ces objets sont-ils cherchés et chargés ? La chronologie exacte du chargement dépend du navigateur, et de sa configuration, mais on peut citer quelques règles générales :

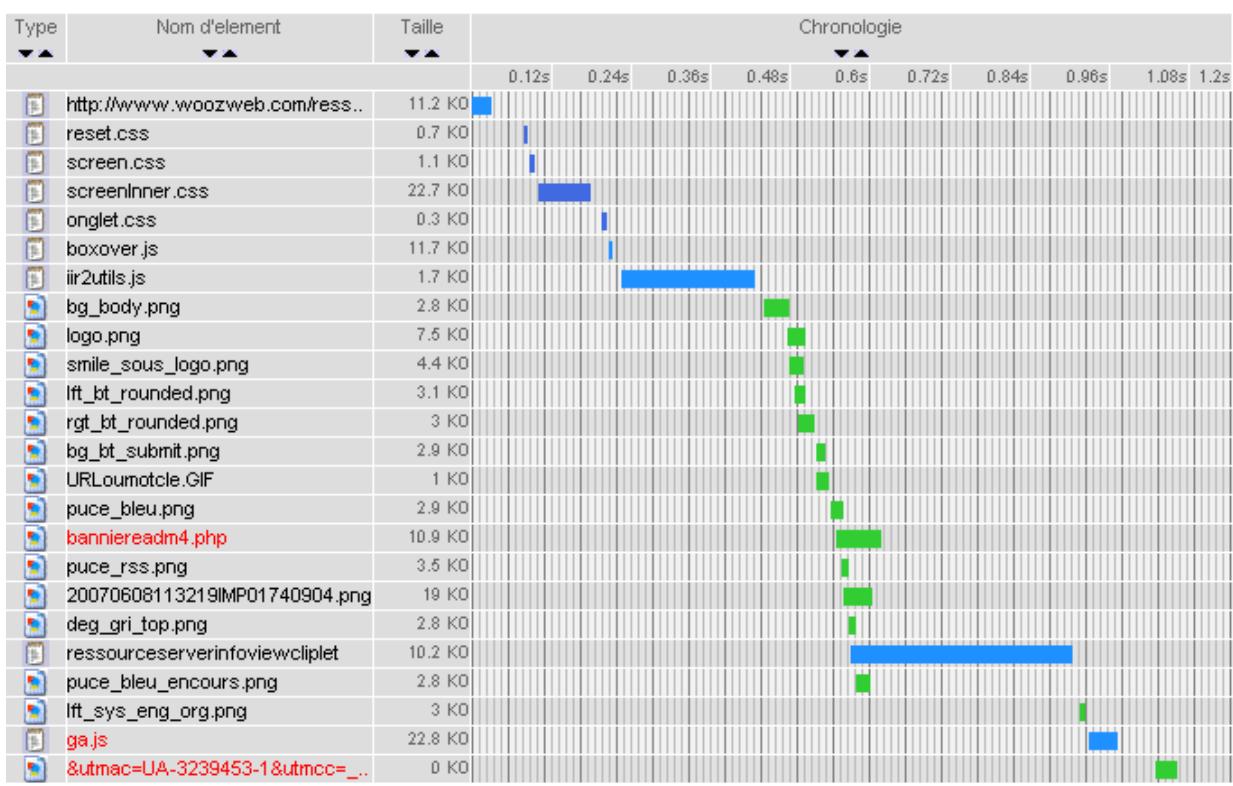
- En premier lieu, il y a des *dépendances de références* entre les objets : ce n'est pas la page Html de départ qui inclut toutes les références aux autres objets. Elle peut inclure une *iframe* Html, qui inclut un Javascript, qui en s'exécutant va demander d'autres objets. Dans ce cas bien sûr, la chronologie du téléchargement épouse cette dépendance.

Architectures Hautes-Performances

- Le chargement des images est en partie parallélisé. Si une page contient 10 images, elles ne seront pas chargées l'une après l'autre. Mais elles ne seront pas non plus chargées toutes les 10 ensemble. Selon la version de navigateur et sa configuration, on aura de l'ordre de 3 téléchargements simultanés par serveur cible.
- Le chargement des fichiers Javascript n'est pas du tout parallélisé : pendant qu'un fichier Javascript est téléchargé, rien d'autre ne l'est.

La perception de l'utilisateur dépend de cette chronologie de chargement de la page. Selon les cas, l'utilisateur peut commencer à lire la page de manière confortable avant la fin du chargement complet. Dans d'autre cas au contraire, la page reste instable, et donc illisible.

Nous verrons qu'il y a beaucoup de moyens de rendre plus rapide ce chargement global de la page, et donc d'améliorer le confort de l'utilisateur, et dans certains cas les performances du serveur.



La figure précédente représente la chronologie d'un chargement de page typique.

Gestion du cache navigateur

Nous accorderons plus loin un chapitre entier aux outils de cache côté serveur. Les navigateurs web gèrent aussi leur propre cache. L'utilisateur a quelques options pour configurer le fonctionnement de ce cache, mais pour l'essentiel, le fonctionnement dépend des directives fournies par le serveur, ou par l'application. Il s'agit principalement des directives¹ « *expires* » et « *cache-control* ».

Un très grand nombre d'applications ne gèrent pas bien ces directives, et donc utilisent très mal les possibilités de cache des navigateurs.

Il peut y avoir deux stratégies dans cette gestion. La première consiste à spécifier une durée de vie « raisonnable », c'est à dire correspondant au délai pendant lequel on peut accepter que la ressource soit obsolète. Ainsi, on peut spécifier un paramètre « *expires* » à $T_0 + 24h$ pour des fichiers CSS, et accepter qu'un changement mette 24 heures à être visible par tous les internautes.

Mais la stratégie la plus recommandée est de spécifier une durée de vie quasi infinie (c'est à dire un *expires* à $T_0 + 10$ ans), et à renommer le fichier lorsqu'il vient à changer.

En effet, il faut se souvenir que tous ces fichiers auxiliaires, css, js ou images, sont toujours spécifiés *au sein de la page html*, de manière directe ou indirecte. Si chaque fichier porte en suffixe son numéro de version, par exemple *common_styles_3.2.css*, alors le jour où ces styles changent, on fera référence au fichier *common_styles_3.3.css*, qui bien sûr ne pourra pas être en cache. Le petit inconvénient serait que le 3.2 va rester en cache à jamais. Mais ce n'est pas grave, tant qu'il y a de la mémoire. Et s'il n'y en avait plus, alors il serait purgé en tant que « *Least Recently Used* ».

Cette seconde stratégie est meilleure car elle n'implique pas un compromis entre réactivité et performance. Autant un cache de niveau serveur peut être efficace avec des durées de vie d'à peine un quart d'heure, autant pour un cache de niveau navigateur, une durée de vie de 24 heures n'est pas vraiment suffisante.

Il faut souligner qu'une bonne gestion des directives de cache bénéficie au confort de l'utilisateur – sa page s'affiche beaucoup plus vite – mais aussi à la capacité du serveur : il y a moins de composants à servir, donc une économie tant de bande passante que de CPU.

¹ Pour plus de précisions quant à leur utilisation : [HTTP://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html)

Cette stratégie de gestion du cache navigateur se définit le plus souvent au niveau du serveur Apache, et donc de manière transparente pour les applications.

Compression du flux

Tout le monde connaît les outils de compression Zip, et la réduction de volume qu'ils procurent, souvent de plus de 60%. Les fichiers Html, Css et Javascript sont des fichiers texte particulièrement adaptés à une bonne compression. Et les navigateurs supportent la compression Gzip, c'est à dire qu'ils savent décompresser les flux.

La compression consomme de la CPU, en échange d'une économie de bande passante. La CPU est consommée côté serveur, pour la somme de tous les flux, et côté client, pour les flux d'un client seulement. Les postes client d'aujourd'hui sont largement dimensionnés, et le traitement de décompression est quasiment négligeable de ce côté.

Dans l'ensemble, le gain de temps sur les flux est très supérieur au petit délai de compression, et il convient donc de mettre en place la compression de manière systématique.

Du côté serveur, la compression est prise en charge de manière tout à fait transparente, au niveau du serveur Apache, avec les modules *mod_gzip* ou *mod_deflate*. La configuration permet de spécifier les types de contenus (type de fichier ou bien type mime) qui sont à compresser.

Et bien sûr, il y a une économie globale de bande passante du côté du datacenter.

Moins de composants, moins de requêtes

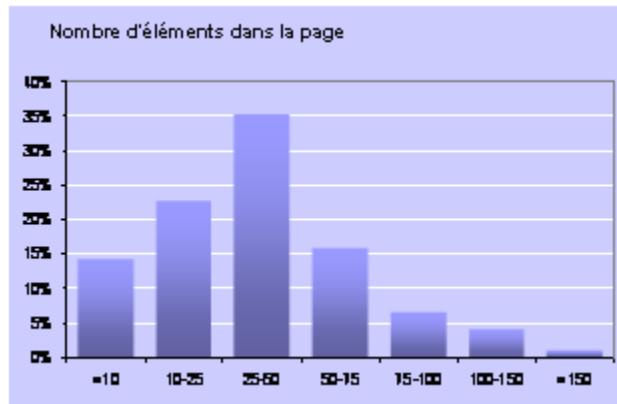
Une règle simple est qu'il faut réduire le nombre de composants intervenant dans une page web.

On ne demande pas pour autant de revoir l'esthétique ou l'ergonomie des sites. Non, l'idée c'est que *pour le même rendu*, il faut parvenir à élaborer la page avec moins de composants.

Quelques statistiques

Etudions quelques statistiques des plus grands sites web français, sous monitoring sur l'outil Woozweb, à partir d'un échantillon de 6000 sites :

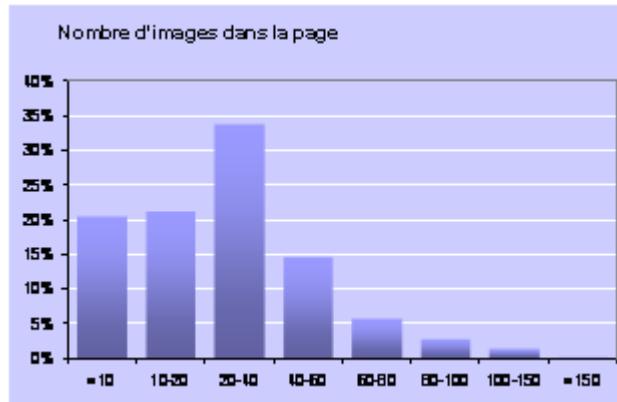
Composants de la page



Ici le pourcentage de sites ayant tel nombre de composants dans leur home page. Par exemple 35% des sites ont entre 25 et 50 composants sur leur page, et au total 30% des sites ont plus de 50 composants.

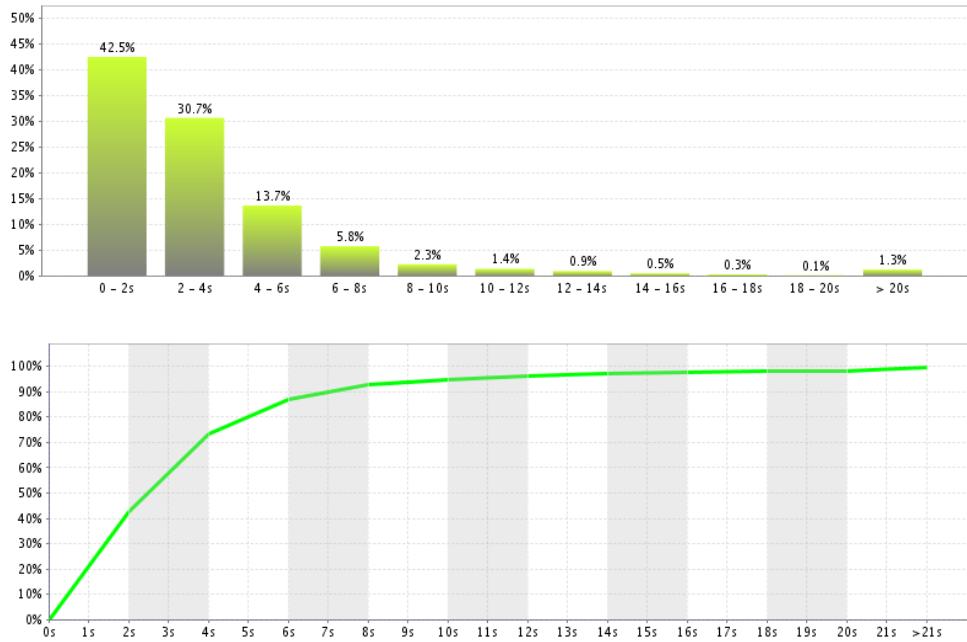
Ces composants peuvent être, comme on l'a vu plus haut, des images, des fichiers javascript, css, objets flash, etc.

Images dans la page



Sur la figure précédente, nous avons un focus sur les images au sein de la page, toujours sur le même échantillon. Où l'on voit que 10% des sites ont plus de 60 images dans leur page.

Chargement de la page



La figure précédente représente le temps total de chargement de la page, dans un navigateur à très haut débit. Le second graphe est une courbe cumulé, où l'on peut lire par exemple que 72% des sites environ chargent la totalité des composants de la page en moins de 4 secondes ; Cela en laisse quand même 28% qui mettent plus de 3 secondes, ce qui est trop.

Poids total de la page



Enfin sur ces derniers graphes, nous avons représenté le poids total de la page, tous composants compris. La moitié des sites ont un poids total de plus de 250 kO.

Réduire le nombre de composants

Il faut bien comprendre que, au delà du volume total échangé, c'est vraiment le nombre de requêtes, donc de composants, qui impacte le plus les performances.

On cherchera donc à adresser les mêmes fichiers au navigateur en moins de requêtes. Comment faire cela ? Plusieurs voies :

- Plusieurs fichiers Javascript ou bien CSS peuvent souvent être réunis en un seul. On pourra également supprimer les commentaires de ces fichiers lors de la mise en production.
- Fusionner les fichiers image également, selon la technique du « CSS Sprite », qui consiste à former une image unique en juxtaposant différentes petites images, puis à sélectionner la petite image côté client, en indiquant le coin supérieur gauche et la dimension. C'est un peu sophistiqué, mais il existe quelques outils pour rendre cela facile.

Dans l'ensemble toutefois, ces techniques sont moins déterminantes si l'on a par ailleurs une bonne configuration du cache. Sauf que, à la différence du cache, elles fonctionnent au premier chargement et non au second.

Infrastructures Globales et CDN

Nous nous intéressons ici aux sites qui servent des internautes dans un grand nombre de pays, qui s'adressent à toute la planète. Nous verrons que ces sites requièrent des solutions spécifiques pour assurer une qualité de service homogène sur tous les continents.

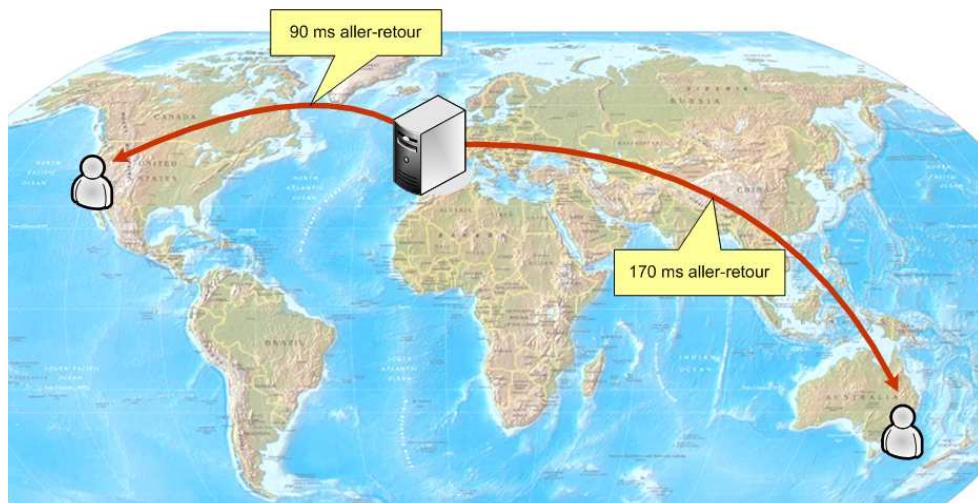
Un monde tout petit ?

On dit parfois que l'Internet supprime les distances, mais ce n'est pas tout à fait exact. La distance physique, géographique, séparant un serveur web et l'internaute a une réelle influence sur les performances perçues, même si elle n'a aucune influence sur les performances du serveur lui-même. Il arrive couramment que des sites qui s'ouvrent à un public international se plaignent de la lenteur perçue par des internautes chinois, australiens ou bien brésiliens, alors que tout allait bien lorsqu'ils ne s'adressaient qu'aux européens.

En matière de télécommunication, deux grandeurs essentielles sont à distinguer : le débit et le temps de latence. Le débit, c'est le nombre de bits ou d'octets par seconde que l'on peut transporter. Le temps de latence c'est le temps que met le premier octet à arriver. Ou bien on parle parfois de temps d'aller-retour (« RTT ou *Round Trip Time* »), le temps que met un octet à faire un aller-retour, plus facile à mesurer par un simple « ping » que le seul temps aller.

Or on s'intéresse le plus souvent au débit, en supposant le temps de latence négligeable. Mais à l'échelle de la planète, ce n'est pas le cas.

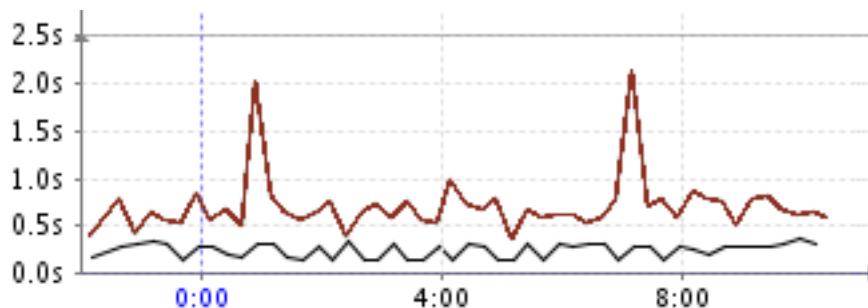
La limite inférieure au temps de latence est liée à la vitesse de la lumière. La vitesse de la lumière dans les fibres optiques est environ 2/3 de la vitesse de la lumière dans le vide, c'est à dire $0,66 \times 300 \times 10^6 \text{ m.s}^{-1}$, soit $200 \times 10^6 \text{ m.s}^{-1}$ (200 000 km par seconde). C'est rapide, mais la terre est grande. Il y a 9 000 km à vol d'oiseau entre Paris et la Californie, ce qui représente un temps de latence minimal de 45 ms. 17 000 km entre Paris et Sidney, soit 85 ms au minimum.



De plus, les fibres optiques sont posées au fond des mers et non à vol d'oiseau, et chaque routeur traversé ajoute un petit temps de latence, de sorte que le temps de latence réel peut être facilement double du minimum théorique. Et ce temps de latence croissant avec les distances tend à faire baisser les débits également. Voici une représentation de la capacité installée effective.



www.smile.fr



L'image précédente, issue de www.woozweb.com, fait apparaître le temps de réponse d'une page html de 380 ko, tel qu'il est perçu depuis la France (courbe noire en bas) et depuis les Etats-Unis (courbe rouge en haut). L'axe des x représente le temps en heures, avec une mesure tous les ¼ d'heures. On constate que le temps de réponse perçu pour ce seul fichier est environ double depuis les Etats-Unis, avec aussi une moindre stabilité c'est à dire des pics plus importants.

En synthèse donc : la distance n'est pas négligeable sur Internet.

Les chinois devront-ils se résigner ?

La seule solution pour obtenir d'excellentes performances en tous points du globe est de servir les contenus depuis différents serveurs, en choisissant le serveur le plus proche de l'internaute. Lorsque le service est lui-même localisé par nature, c'est à dire lorsque les chinois et les européens n'accèdent pas aux mêmes pages, n'achètent pas les mêmes produits, alors le plus simple est de prendre un hébergement dans différentes zones géographiques, au minimum Amérique, Europe, Asie,

avec des plateformes totalement distinctes, n'échangeant que quelques flux de synthèse au niveau back-office (états des ventes, tables de référence...).

Dans certains cas toutefois, cette approche ne convient pas. Soit le service est intrinsèquement global et ne peut pas être découpé en sous-domaines indépendants. Ce peut être le cas d'un FaceBook par exemple. Soit au contraire le service n'est pas assez global pour pouvoir se permettre un hébergement multiple.

Content Delivery Network

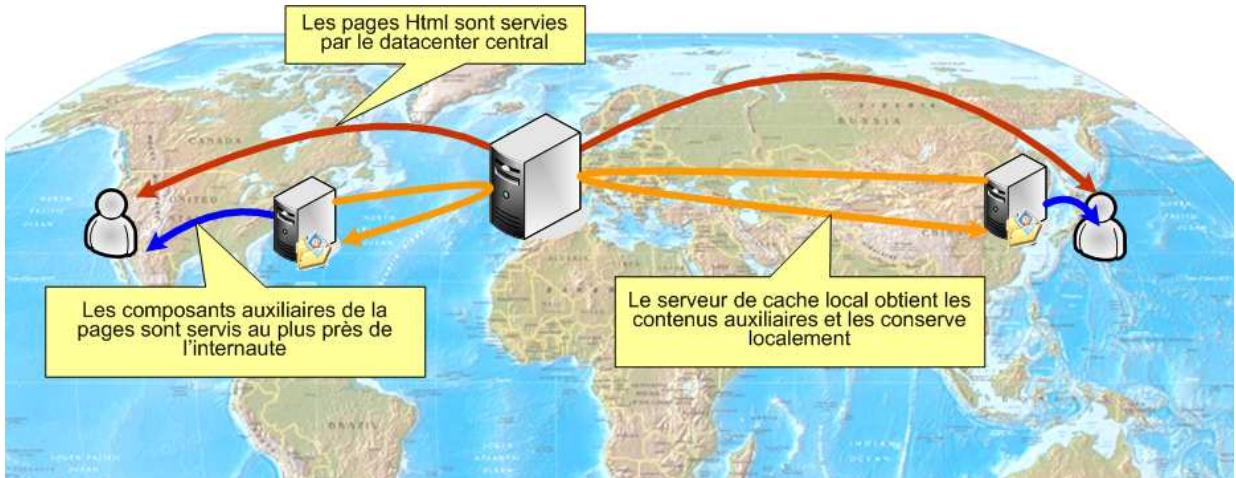
On appelle CDN, *Content Delivery Network*, un dispositif qui permet de servir les mêmes contenus depuis différents serveurs, en sélectionnant le serveur le plus proche de l'internaute. Les plus grands acteurs de l'Internet ont construit leur propre CDN, mais les autres ont recours à des prestataires commerciaux.

Un CDN maison pour pas cher

Prenons un cas très simple. Vous avez une plateforme web hébergée en France, qui adresse des clients dans le monde entier. Vos pages, comme celles d'un site moyen, se chargent en 3-5 secondes, dont disons 4 pour les différents composants statiques, images et autres. Depuis l'Asie, ces 4 secondes deviennent 8 ou 10.

Comment y remédier simplement ? Il suffit de louer un petit serveur pas cher dans un bon datacenter local, par exemple en Corée du Sud, et d'y servir tous vos fichiers statiques. Pour cela, on mettra en place un serveur de cache de type Squid, qui conserve localement une copie des fichiers. Ensuite, il faut encore que, pour vos internautes asiatiques, les images soient servies depuis le serveur coréen. Soit vous gérez cet aiguillage par vous mêmes, en détectant l'origine de l'internaute et en insérant des images `` au lieu des références d'images précédentes. Ca demande un peu de code, mais en termes d'architecture c'est assez simple. L'alternative serait de conserver le même nom de serveur `www-static.smile.fr` indépendamment de la géolocalisation, mais de lui faire correspondre différentes IP, donc différents serveurs, comme on le verra plus loin (cf « Répartition de charge de niveau DNS », page 53).

Vous aurez alors construit votre propre CDN et amélioré sensiblement les performances perçues par vos internautes chinois, pour un coût très raisonnable.



Prestataires CDN

On peut distinguer deux niveaux dans la mise en œuvre d'un CDN. Soit le CDN ne distribue que certains types de composants, typiquement les images, vidéos, css, etc. Soit le CDN va plus loin et joue un rôle de cache sur des fragments de Html, produits par des applications. Sur ce dernier aspect, on se reportera à « Cache par fragments », page 133.

La mise en œuvre du CDN pour servir localement les composants statiques seulement peut être transparente et très rapidement mise en œuvre. Outre l'option « maison » évoquée ci-dessus, il existe quelques prestataires proposant un CDN prêt à l'emploi. Dans une approche un peu plus « pro », on ne bricolera pas les tags ``, on comptera plutôt sur le DNS pour diriger les requêtes d'images vers le serveur le plus proche. Outre la quasi-transparence de la mise en œuvre, les CDN commerciaux ont bien sûr une couverture mondiale difficilement égalable.

Ces dernières années, le marché avait été écrasé par l'acteur dominant Akamai, qui avait de plus racheté quelques-uns de ses concurrents et dont l'action a été multipliée par 4 en 2006. Mais l'arrivée massive des vidéos a donné un nouvel élan à ce secteur, et fait apparaître de nombreux concurrents, qui ont ouvert une guerre des prix sévère ; en particulier Limelight Networks, Level 3, Internap, CDNetworks, ou encore Panther Express.

Qu'on ne se trompe pas : l'approche CDN conserve *une logique fondamentale centralisée*. Il y a une plateforme centrale, mais un réseau global de distribution des contenus. Mais pour les applications de la plateforme centrale, l'existence du CDN est pratiquement transparente.

REPARTITION DE CHARGE

Principe de répartition de charge

Chaque serveur, chaque composant, ayant unitairement une capacité de traitement limitée, il est évidemment nécessaire de mettre en œuvre plusieurs serveurs pour dépasser cette limite, et cela implique d'être capable de répartir le travail entre ces serveurs.

C'est l'objet de la répartition de charge. La répartition de charge s'entend le plus souvent au niveau des serveurs HTTP, c'est à dire en frontal sur une plateforme web. Mais le même principe peut s'appliquer sur n'importe quel service. C'est à dire que l'on peut répartir la charge entre plusieurs bases de données, entre plusieurs serveurs de fichiers, entre plusieurs serveurs offrant un même service.

Ces invocations de services s'appuient sur différents protocoles, tous intervenant au dessus de TCP/IP. C'est pourquoi on s'intéressera tout particulièrement à la répartition de charge au niveau TCP/IP.

Mais il y a fondamentalement trois niveaux de répartition de charge pertinents dans le monde du web :

- Le niveau DNS
- Le niveau HTTP, « applicatif »
- Le niveau TCP/IP.

Au delà de ces « niveaux », sur lesquels nous reviendrons, on peut analyser la question selon différents axes :

- Quelle est la finalité de cette répartition, l'objectif visé ?
- Que répartit-on ? Des requêtes, des sessions, des internautes ?
- Selon quelle logique s'effectue cette répartition ?
- Et enfin, quels sont les techniques et outils de cette répartition ?

Finalité et logique de répartition

Nous voulons distinguer clairement les *principes et logiques* de répartition des *techniques et outils*, qui seront traités plus loin.

Pourquoi veut-on répartir la charge ?

Augmenter la capacité

En premier lieu, on l'a dit, pour atteindre une plus grande capacité de traitement, en mettant en œuvre plusieurs serveurs physiques.

Mais ceci avec quelques objectifs secondaires.

Equilibrer la charge

La logique de répartition de charge vise le plus souvent un équilibrage : on souhaite que chaque serveur soit utilisé de manière optimale, qu'il n'y en ait pas un surchargé tandis qu'un autre est sous-utilisé. Dans cette finalité de répartition équilibrée, nous verrons qu'il y a différents algorithmes possibles, depuis l'aléatoire jusqu'au plus « intelligent ».

Résister aux pannes

Nous verrons que l'objectif de répartition est souvent corrélé à celui de haute disponibilité : on veut aussi répartir la charge entre des serveurs *parce que c'est un bon moyen de résister à la panne de l'un d'entre eux*. Ceci à condition que les dispositifs de répartition puissent détecter une panne et exclure le serveur défaillant.

Spécialiser des serveurs

La répartition peut avoir d'autres logiques que celle de l'équilibre. Elle peut viser aussi la spécialisation des serveurs, faisant traiter certaines typologies de requêtes, ou bien d'internautes, par certains serveurs.

Faciliter l'exploitation

Différentes solutions de répartition de charge permettent d'agir à chaud sur les paramètres de la répartition, et en particulier d'ajouter ou de retirer un serveur. Il est possible ainsi d'arrêter un serveur en douceur, en le retirant de la répartition, puis en attendant que les connexions qui lui sont affectées soient fermées. C'est le moyen de gérer une opération de maintenance en assurant une parfaite continuité du service, et à cet égard, la flexibilité d'exploitation est aussi une des finalités de la répartition de charge.

Répartition de requêtes ou répartition de sessions ?

Un serveur web traite un très grand nombre de requêtes, adressées par un très grand nombre d'internautes. On peut analyser la répartition de charge en termes de granularité : répartir « la charge » est un peu vague, que va-t-on répartir exactement ? La requête HTTP est ici l'entité minimale, atomique. Le traitement complet d'une requête HTTP est une petite tâche, qui demandera quelques centièmes ou quelques dixièmes de secondes, et sera traitée par un même serveur.

Certaines techniques de développement obligent à traiter toutes les requêtes constituant une même session, d'un même internaute, sur le même serveur, ce qui présente des contraintes spécifiques quant à la répartition.

Répartition entre des serveurs ou entre des datacenters ?

Enfin, on peut répartir la charge entre différents serveurs d'un même datacenter, sur un même réseau LAN et partageant un ensemble de ressources en amont (routeurs, switchs, load-balancer), et en aval (base de données, serveur de fichiers, SAN, ...), ou bien on peut répartir la charge entre plusieurs datacenters, qui ne partagent rien.

Les techniques utilisées sont alors très différentes.

Répartition de charge de niveau DNS

Principe

La répartition de charge de niveau DNS intervient dans l'association d'une adresse IP à un nom de serveur.

Lorsque le navigateur doit accéder à un serveur dont il connaît le nom, par exemple www.smile.fr, il commence par rechercher l'adresse IP correspondant à ce serveur www sur le domaine smile.fr, en adressant une requête à son serveur DNS, qui lui-même, s'il ne dispose pas de l'information, interrogera d'autres serveurs DNS, de manière récursive. Une fois que le poste client a obtenu l'adresse IP, il la conserve en cache selon différentes règles, généralement de l'ordre d'une demi-heure.

C'est dans cette phase d'association entre un nom de serveur et une adresse IP, c'est à dire un serveur, qu'intervient la répartition de charge de niveau DNS, simplement en fournissant différentes adresses IP pour un même nom de serveur.

On utilise assez peu cette technique pour répartir la charge entre serveurs d'un même datacenter, simplement parce que les répartition réseau, que nous verrons plus loin, sont plus flexibles et plus réactives.

La répartition de niveau DNS est donc plutôt utilisée pour répartir la charge entre plusieurs datacenters.

Quelle en est la finalité ?

- La tolérance aux pannes : un datacenter peut se trouver globalement indisponible, pour différentes raisons, et toutes les mesures internes à la plateformes seront alors inopérantes.
- La proximité physique des internautes, dans une approche de type « Content Delivery Network », telle que décrite plus haut (cf « Infrastructures Globales et CDN », page 46).
- La recherche d'un point d'équilibre entre mesures de haute-disponibilité internes à une plateforme, et haute-disponibilité globale multi-plateformes, pour viser une haute-disponibilité globale au moindre coût.

Nous passerons en revue trois types de solutions de répartition de niveau DNS : DNS Round-Robin, GeoDNS et Anycast.

La première est la seule qui soit totalement standard, compatible avec n'importe quel DNS ; les deux suivantes requièrent un serveur DNS spécifique et sont donc sensiblement plus complexes à mettre en œuvre.

DNS-Round-Robin

La technique de répartition DNS la plus simple et la plus couramment utilisée est celle du *DNS Round-Robin*, ou DNS-RR.

Lorsqu'un serveur DNS répond à un client, il peut fournir non pas une adresse IP, mais une liste d'adresses IP, dans un certain ordre. La première adresse est celle qu'il faut utiliser en premier lieu, les autres sont des adresses de secours. Si l'on dispose de trois serveurs S_1, S_2, S_3 capables de traiter les requêtes, alors le serveur DNS pourra répondre $\{S_1, S_2, S_3\}$, dans cet ordre. Rappelons qu'ici S_i désigne l'adresse IP d'un serveur. Le principe du DNS-RR consiste à répondre en indiquant les serveurs en permutation circulaire : $\{S_1, S_2, S_3\}$, puis $\{S_2, S_3, S_1\}$, puis $\{S_3, S_1, S_2\}$ et ainsi de suite. Le premier client, s'adressera donc à S_1 , le second à S_2 , le troisième à S_3 . Et si S_1 ne répond pas, alors le premier s'adressera à S_2 en secours.

Cette technique a le mérite d'être très simple à mettre en œuvre, et ne requiert pas un DNS spécifique, le Round-Robin est supporté par *tous* les serveurs DNS de l'Internet, il n'y a qu'à demander. La solution peut

convenir lorsqu'on dispose de 2 ou 3 datacenters d'une même région géographique, d'un même continent. Elle convient beaucoup moins bien au niveau global, avec des datacenters sur différents continents, car elle peut amener l'internaute européen sur un datacenter asiatique, l'internaute américain sur un datacenter européen, et ainsi dégrader la qualité de service de tout le monde. On a vu déjà qu'à l'échelle de la planète la proximité géographique a un impact important sur la qualité de service.

La répartition par DNS-RR est assimilable au global à une répartition aléatoire. En particulier, elle ne s'appuie sur aucune connaissance de la charge respective des serveurs. Néanmoins, avec un trafic important, l'aléatoire procure une répartition tout à fait satisfaisante, avec des écarts en général de moins de 5% entre les serveurs.

On met en œuvre également le DNS-RR lorsqu'on vise un hébergement low-cost, qui ne permet pas d'insérer de load-balancer.

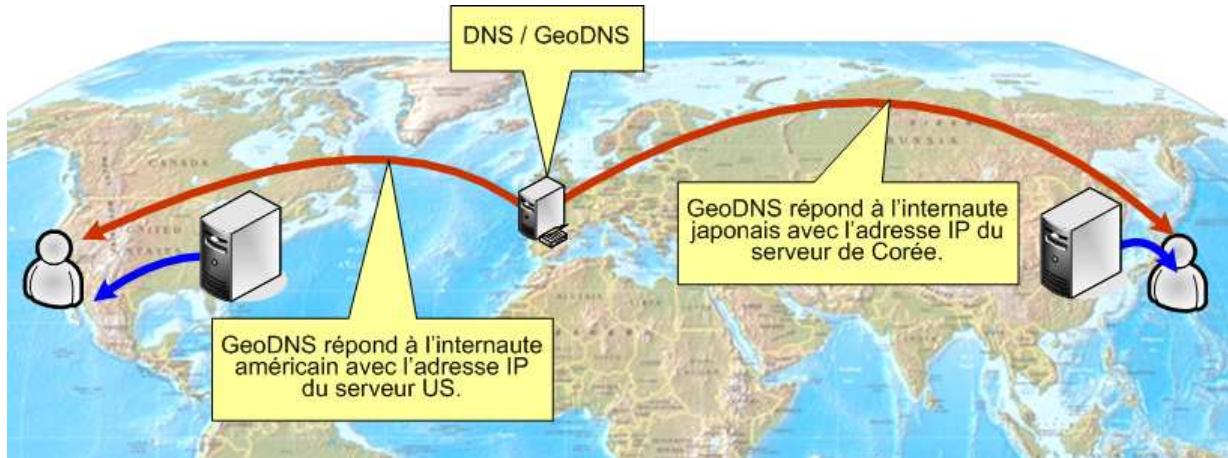
GeoDNS

Au lieu de répondre en permutation circulaire, le serveur DNS peut essayer de répondre intelligemment, en fournissant en premier le serveur le plus proche de l'internaute (cf « *Un monde tout petit ?* », page 46), après avoir localisé l'internaute, au moyen de son adresse IP. C'est l'objet de l'extension GeoDNS, qui ajoute cette fonctionnalité au serveur BIND, le DNS plus utilisé sur l'Internet.

Bien entendu, cela suppose de gérer son propre DNS sur son domaine.

L'extension GeoDNS permet de définir, pays par pays, la réponse spécifique du DNS à une demande de l'internaute, et donc de répondre { S_{FR}, S_{US}, S_{KR} } à un internaute européen, et { S_{KR}, S_{US}, S_{EU} } à un internaute chinois, où S_{FR}, S_{US}, S_{KR} sont les adresses des serveurs hébergés respectivement en France, Etats-Unis et Corée du Sud.

C'est donc une solution qui convient bien au niveau global.



Anycast

La dernière solution, utilisée par les plus grands sites globaux, s'appelle « Anycast ». Elle consiste à utiliser la même adresse IP pour différents serveurs DNS en charge du domaine, chaque serveur étant hébergé sur le même datacenter que l'une des plateformes web. Ainsi si l'on dispose de trois plateformes dans trois datacenters, en Europe, Asie et Amérique, on mettra en place trois DNS pour mondomaine.com, sur ces mêmes plateformes.

Comme ces trois serveurs DNS partagent la même adresse IP, on compte sur les algorithmes de routage IP pour trouver automatiquement le serveur DNS le plus proche de l'internaute. Chaque DNS répond à la requête de nom de domaine de manière légèrement différente, en plaçant l'IP de son propre datacenter en tête de liste.

Le génie de cette solution, c'est de trouver de manière naturelle le serveur le plus proche, au sens de la topologie de l'Internet, et non au sens géographique, en utilisant les mécanismes standards du réseau.

Partager une adresse IP est en général hasardeux, et effectivement, dans une connexion TCP/IP, il se pourrait que certains paquets arrivent à un serveur et d'autres à un autre serveur, ce qui rendrait la communication impossible. Mais l'interrogation DNS n'utilise que le protocole UDP, qui est sans session, de sorte qu'il est compatible avec le mode Anycast.

Avantages et limites de la répartition DNS

Comme on l'a vu, la répartition de niveau DNS, quel qu'en soit l'outil, est pratiquement la seule qui permette de répartir sur différents datacenters, ce qui permet à la fois une meilleure tolérance aux pannes, et dans certains cas une moindre latence donc une meilleure qualité de service.

Dans le mode *Round-Robin*, elle est très facile à mettre en place, et peu coûteuse, pour autant que l'application soit compatible, c'est à dire qu'elle ne suppose aucune ressource partagée, un « *share nothing* » de niveau datacenter.

Elle présente toutefois les limitations suivantes :

- La gestion de la détection de panne et du secours est laissée à la charge du navigateur. Si le serveur HTTP répond une erreur 500, par exemple, le navigateur ne passe pas sur la seconde adresse.
- L'équilibrage de la charge n'est pas géré de manière fine, typiquement avec des capacités d'accueil différentes selon les plateformes, ou bien l'affectation au datacenter le moins chargé.
- Elle ne permet pas de mettre en œuvre l'affinité de serveur de manière satisfaisante.

A la différence des solutions de répartition de charge réseau, que l'on verra plus loin, la répartition de niveau DNS ne met pas en œuvre une surveillance des plateformes, et n'adapte pas la répartition à la disponibilité ou à la charge.

Notons qu'il existe malgré tout quelques solutions gérant la haute-disponibilité au travers d'une reconfiguration DNS : on met en œuvre un monitoring des différents serveurs, et si l'un d'entre eux ne répond pas, on met à jour les DNS pour exclure ce serveur.

Tant qu'on utilise les DNS de manière standard, c'est le cas pour le mode round-robin, la tolérance aux pannes du dispositif de répartition lui-même est excellente. En revanche, pour les deux autres modes, qui demandent un DNS spécifique, c'est à vous d'assurer son secours et sa disponibilité.

Redirection applicative

Il faut évoquer aussi la possibilité de redirection applicative, vers un serveur ou un datacenter proche de l'internaute. C'est une version très rustique du GeoDNS : l'internaute se connecte à un premier serveur www.smile.fr, l'application localise l'internaute, en déduit le serveur le plus approprié, et adresse un *redirect* HTTP vers ce serveur, par exemple www-us.smile.fr, qui assurera la suite de la session.

Répartition de charge de niveau TCP

Quelques rappels

On l'a vu, presque tous les échanges sur une plateforme web s'appuient sur des protocoles construits sur IP et le plus souvent sur TCP/IP.

TCP est un protocole de niveau 4, de niveau transport. Il implique l'établissement d'une connexion entre deux serveurs, puis l'échange bidirectionnel de messages sur cette connexion, puis la fermeture de la connexion. Chacun des message est ensuite décomposé en paquets IP, et chaque paquet est routé indépendamment. TCP se charge de réémettre les paquets perdus, le cas échéant, et de les remettre dans l'ordre à l'arrivée si besoin.

Le protocole HTTP est construit au dessus de TCP. C'est à dire que les requêtes et réponses HTTP sont échangées sur une connexion TCP. Le navigateur ouvre une connexion TCP, envoie une requête, par exemple une requête GET portant sur une URL donnée, puis attend la réponse sur la même connexion. Avec HTTP 1.0, la connexion TCP est fermée une fois la réponse reçue. Avec HTTP 1.1, la connexion peut être maintenue ouverte pour une autre requête.

Le protocole HTTP est, à la base, un protocole sans état, c'est à dire qu'il n'y a pas d'information conservée relativement aux échanges précédents, chaque couple requête/réponse est indépendant des précédents. Cela du moins du point de vue du protocole, car le besoin de gérer des sessions conduit parfois à construire une notion d'état au dessus du protocole, au niveau applicatif.

Répartition de charge TCP

La répartition de charge TCP, ou encore « de niveau 4 », permet de faire apparaître N serveurs comme une seule adresse IP vue de l'extérieur. La répartition de charge est associée à une translation d'adresse (NAT), qui permet de totalement décorréliser les adresses IP internes de celles qui sont vues de l'extérieur. Sur le réseau interne, chaque serveur dispose de sa propre adresse IP.

La répartition de charge de niveau TCP intervient dans la phase d'établissement de la connexion. Une demande de connexion est adressée par un serveur ; elle parvient à l'équipement de répartition de charge, qui détermine le serveur auquel il va affecter la connexion, parmi les serveurs disponibles. Une fois la connexion TCP établie, l'équipement de répartition de charge devient pratiquement transparent : il pousse les paquets IP de la connexion vers le serveur

sélectionné, dans un sens et dans l'autre. Ceci jusqu'à fermeture de la connexion.

Les algorithmes de répartition

Il existe une variété d'algorithmes pour gérer la répartition de charge, c'est à dire *choisir le serveur* lorsqu'une nouvelle connexion est à affecter.

- **Round-robin.** L'expression signifie une ronde, une permutation circulaire. Il s'agit d'une affectation cyclique : A puis B puis C puis A puis B puis C, ...
- **Weighed round-robin.** Une affectation cyclique avec pondération. La pondération permet de prendre en compte la capacité différente des serveurs. Si un serveur a une capacité 1,5 par rapport aux autres, il est sélectionné 1,5 fois plus souvent. Les plateformes ne sont pas toujours mises en place d'un seul coup, et il est donc courant que des serveurs de puissance différentes soient utilisés.
- **Least-connection.** Affectation au serveur qui a le moins de connexions ouvertes, c'est à dire donc en équilibrant le nombre de connexions entre les serveurs.
- **Weighed-least-connection.** Même principe, mais avec pondération selon la capacité du serveur.
- **Priority activation.** Une logique selon laquelle certains serveurs ne sont utilisés que si les serveurs principaux dépassent une charge définie comme seuil.
- Algorithmes basés sur **l'IP source**. Un algorithme de hashage est appliqué à l'IP source pour déterminer le serveur cible. Une même IP est toujours connecté au même serveur. On pourrait penser que cela procure une répartition de charge avec affinité de serveurs (cf. « Répartition avec affinité de serveur », page 60) , mais il existe des cas où un internaute peut changer d'adresse IP.
- **Aléatoire.** Une répartition purement aléatoire, qui peut donner un équilibrage satisfaisant, sur des volumes importants.

Notons que le *round-robin*, comme le *weighed-round-robin*, a le mérite de la simplicité, mais peut en théorie répartir médiocrement la charge car la durée de vie des connexions TCP peut être très variable, entre quelques centièmes de secondes et plusieurs heures. Si le hasard fait que l'un des serveurs reçoit plusieurs connexions de longue durée, il se retrouve avec un bien plus grand nombre de connexions ouvertes. Mais ce n'est pas tout, sur une connexion ouverte, il peut y avoir plus

ou moins de trafic, de messages échangés, de sorte que même l'équilibrage des connexions peut être imparfait. Et enfin, les requêtes peuvent solliciter plus ou moins les ressources du serveur.

Néanmoins, sur un serveur web, ces subtilités théoriques sont le plus souvent balayées par la loi des grands nombres et la relative homogénéité du trafic, de sorte que les algorithmes les plus simples conviennent.

Si l'on cherchait à amener chaque serveur au plus près de sa capacité, alors le parfait équilibre pourrait être important. Mais ce n'est pas le cas. Il faut garder à l'esprit que, sur une plateforme bien dimensionnée, les serveurs ne dépassent pratiquement jamais 30% de leur capacité, avec de très brèves crêtes à 70-80%, lors de phénomènes transitoires planifiés (redémarrage, vidage de cache, opération d'exploitation), ou non planifiés. On ne cherche jamais à utiliser les serveurs au maximum en les chargeant au delà de 50-70% car alors la moindre crête transitoire se traduirait par une saturation et donc indisponibilité.

C'est pourquoi, si un phénomène transitoire dans la répartition amène un serveur à 25% et l'autre à 35% pendant quelques secondes ou minutes, cela ne pose aucun problème.

Répartition de charge de niveau 7

Répartition avec affinité de serveur

Les échanges sur une même connexion TCP, une fois établie, sont bien sûr avec le même serveur. Mais le protocole HTTP utilise de nombreuses connexions TCP pour la session d'un même internaute. De sorte que, avec la répartition TCP décrite plus haut, les requêtes d'un même internaute sont réparties entre les différents serveurs.

Certaines applications sont construites de telle manière qu'elles exigent que les requêtes d'un même utilisateur soient adressées à un même serveur. Généralement parce que les applications conservent en mémoire des informations relatives à la session, par exemple le contenu du panier dans un site de e-commerce. Pour construire des plateformes à très haute capacité d'accueil, c'est une pratique qui est à éviter, mais il arrive souvent que la conception des applications ne prenne pas en compte les exigences d'architecture.

On appelle « *répartition avec affinité de serveur* », une répartition de charge qui dirige toutes les requêtes d'une même session d'un internaute, à un même serveur. On parle parfois de « *sticky sessions* », des sessions « collantes ».

La solution la plus utilisée pour parvenir à une répartition de ce type consiste à utiliser un identifiant de session placé dans un cookie. Soit l'application a placé un cookie identifiant la session applicative et l'équipement de répartition lit ce cookie, puis affecte toutes les requêtes portant ce cookie au même serveur. Soit l'équipement insère lui-même son cookie pour reconnaître les requêtes de la même session.

Nous étudierons ceci de manière plus spécifique au chapitre « Gestion des sessions », page 63.

Principe de la répartition niveau 7

« Niveau 7 » signifie niveau « application » en référence au modèle OSI. En fait, dans la pile TCP/IP on distingue moins de niveaux que dans le modèle OSI, et donc on passe directement du niveau 4, transport, au niveau 7, application. HTTP, comme SMTP, POP, SSH, Telnet et bien d'autres sont donc des protocoles de niveau 7.

La répartition de niveau 7 est une répartition dont les mécanismes impliquent l'analyse des requêtes HTTP. On a vu plus haut que le niveau 4 n'intervenait pratiquement qu'à l'établissement de la connexion TCP. Une fois la connexion établie, il suffit de transférer les paquets.

Dans un mécanisme de répartition de niveau 7, on analyse *le contenu de chaque requête*, pour décider du routage. En pratique, deux choses sont recherchées et analysées :

- Les cookies, qui figurent dans l'entête HTTP.
- L'URI, c'est à dire l'URL et l'ensemble de ses paramètres.

L'analyse de l'URI permet de mettre en place une répartition avec *spécialisation des serveurs* ; nous y reviendrons. Elle permet aussi d'assurer l'affinité de serveur, dans le cas où un jeton de session est inséré dans l'URI.

Notons que la différence entre niveau 4 et niveau 7 n'est pas négligeable. Pour intervenir au niveau 7, on doit scruter tout le contenu des échanges, et analyser chaque message pour identifier les requêtes HTTP, puis analyser encore la requête pour trouver le cookie. C'est un travail bien plus important que d'aiguiller des paquets. Et ce n'est pas tout, s'il ne peut pas aiguiller la requête avant de savoir ce que contient le cookie, il induit forcément un délai supplémentaire.

En résumé, la répartition de charge de niveau 7, qui permet l'affinité de serveur, est parfois rendue nécessaire par les applications, mais ne doit pas être vue comme le *nec plus ultra* de la répartition.

Spécialisation des serveurs

L'affinité de serveur n'est pas la seule raison de gérer la répartition de charge au niveau 7. Une utilisation courante consiste à répartir la charge au moyen d'une analyse de l'URL, de manière à ce que les requêtes portant sur les mêmes ressources soient adressées aux mêmes serveurs. Cette affectation peut être définie soit de manière explicite, par des expressions régulières portant sur l'URL, soit de manière automatisée, en utilisant un hashage sur l'URL. Dans le premier cas, l'administrateur maîtrise la répartition, dans le second, la répartition est indifférente, mais elle est stable, un même serveur reçoit toujours les mêmes URLs.

A quoi sert ce type de répartition ? Dans le cas d'une répartition explicite, on peut définir une spécialisation des serveurs, correspondant à des configurations spécifiques. Par exemple les serveurs délivrant les images ou autres petits fichiers statiques pourront avoir une configuration parfaitement dédiée à cet usage. Mais la spécialisation des serveurs a aussi des inconvénients, et complique en particulier la question de la tolérance aux pannes.

Dans une répartition automatique par hashage, chaque serveur doit pouvoir répondre à toute forme de requête, mais les requêtes semblables tendent malgré tout à être adressées aux mêmes serveurs. On a donc une bonne flexibilité et une gestion satisfaisante de la tolérance aux pannes. A quoi bon cette affinité ? Principalement pour permettre aux caches des serveurs de se spécialiser. Si l'on dispose de 5 serveurs ayant chacun un cache d'une capacité de 2 GO, mais un volume total de contenus de 10 GO pour 80% des pages, alors on aura un hit-ratio trop faible sur chacun des serveurs. Si on parvient à séparer le trafic en 5 sous-ensembles de 2 GO chacun, on obtiendra au contraire un excellent hit-ratio sur chacun d'entre eux. C'est un peu comme si l'on avait mis en commun les 5 fois 2 GO au lieu de les laisser se disperser.

Répartition de charge et SSL

Lorsque la session est sécurisé, en HTTP-S ou SSL, alors le contenu des échanges HTTP est totalement crypté. Il est impossible de gérer une répartition de charge de niveau 7 sur une session cryptée, alors que l'on peut gérer une répartition au niveau 4.

Il est donc nécessaire de commencer par terminer la session SSL, en amont du load-balancer. Quitte le cas échéant à établir une nouvelle session sécurisée en aval, mais en général on considère que au sein du datacenter le besoin de sécurisation est moindre.

Dans ce cas, on combine généralement les deux fonctions, terminaison SSL et load-balancing, sur le même équipement frontal. En outre, cela décharge un peu de la CPU du serveur.

Gestion des sessions

La question de la gestion des sessions mérite une attention particulière. Nous avons dit plus haut que, lorsqu'une application a besoin de retrouver en mémoire des informations de *contexte*, relatives aux échanges précédents de l'internaute, de la même *session*, cela amène souvent à mettre en œuvre une répartition de charge avec affinité de serveur.

C'est un mode de fonctionnement qui est souvent retenu, non pas par choix, mais parce que la conception de l'application n'a pas pris en compte le besoin d'extensibilité.

Un fonctionnement sans affinité de serveur est supérieur en termes d'équilibre de charge, de flexibilité, de haute-disponibilité. Lorsqu'on le peut, c'est ce fonctionnement qui doit être visé.

Voyons quelles sont les alternatives à une gestion de contexte.

La première chose à dire est que le contexte de *session* doit être réduit au maximum. Et en particulier il doit comporter de vraies informations de *session* et non des informations de *transaction*. Si l'internaute est en train de réserver un billet d'avion dans un processus à plusieurs étapes, les saisies précédentes relèvent du contexte de *transaction*. Il est préférable que les informations de *transaction* ne soient pas en session. L'alternative est de les intégrer, soit dans les paramètres d'URL, soit éventuellement dans des champs cachés des formulaires.

Le second point est qu'il existe différentes solutions permettant une gestion de contextes de session sans affinité de serveur. Passons-les en revue.

Partage de sessions par cookies

Principes de fonctionnement

Comme on le sait, le cookie est une donnée conservée sur le navigateur, à la demande du serveur. Le cookie est retourné au serveur avec chacune des requêtes HTTP. Les cookies ont une durée de vie, ou plutôt une date d'expiration, comme les objets en cache. Si la date d'expiration n'est pas indiquée, le cookie est « volatile », il est purgé à la fin de la session, c'est à dire à la fermeture du navigateur. Au contraire les cookies qui restent jusqu'à une certaine date sont des cookies persistants.

Le cookie est retourné soit au serveur qui l'a envoyé (on parle ici de serveur virtuel : en cas de répartition de charge, le cookie est retourné à n'importe lequel des serveurs physiques), soit éventuellement à tous

les serveurs du domaine. Mais en aucun cas à un serveur hors du domaine.

Chaque cookie a un identifiant, choisi par le serveur, et tous les couples (identifiant, valeur) de tous les cookies existants pour ce serveur ou son domaine, sont envoyés à chaque requête. Notons enfin qu'il y une limite de taille de 4 Ko pour un cookie. Sur Internet Explorer, la limite est de 4 Ko pour tout un domaine.

Maintenant que l'on a rappelé les grands principes des cookies, voyons comment les utiliser dans le contexte d'architectures hautes performances.

Mauvaise réputation

Les cookies ont mauvaise réputation. D'abord parce qu'ils sont utilisés par différents sites et régies publicitaires pour consolider l'information recueillie sur le comportement et les préférences des internautes. Du coup, certains peuvent même choisir de désactiver ou restreindre la gestion des cookies sur leur navigateur, mais c'est encore une faible proportion des internautes.

La mauvaise réputation des cookies est aussi liée à des questions de sécurité : confidentialité, intégrité, possibilité de vol de cookie. Nous reviendrons sur ces questions plus loin.

Cookie identifiant de session

La plupart des dispositifs de gestion de sessions reposent sur un cookie qui ne porte qu'un identifiant de session. Cet identifiant est la clé permettant de retrouver un contexte de session, conservé côté serveur. Si ce contexte est en mémoire de l'application, il faut que les requêtes d'un même internaute parviennent au même serveur, c'est à dire qu'il faut une répartition de charge « avec affinité de serveur » ou avec « *sticky sessions* ».

Eviter l'affinité de serveur

Les sites à très forte audience évitent, le plus souvent, d'utiliser des *sticky sessions*. Pour plusieurs raisons : en premier lieu, la répartition est imparfaite, car gérée sur des durées trop longues, mais surtout la tolérance aux pannes est médiocre puisque si un serveur tombe les sessions qu'il gérait sont perdues, c'est à dire que les utilisateurs sont brutalement ramenés à la phase d'identification.

Sessions et outils de développement

Très souvent, ces questions ne sont pas étudiées en tant que telles : c'est l'environnement de développement qui dicte sa loi. Il gère les contextes de cette manière, et dicte ses contraintes pour l'architecture. Architecture et scalabilité sont vus, trop souvent, comme des problématiques aval : j'ai développé mon application, maintenant quelle architecture vais-je donc choisir ?! C'est la manière de procéder la plus usuelle. Elle convient pour du bas et du milieu de gamme, mais elle ne convient plus pour des plateformes à très haute performance.

Gérer des données applicatives en cookie

Au lieu de stocker un identifiant seulement, les cookies peuvent être utilisés comme gestionnaire de données de session. D'une certaine manière les cookies peuvent aussi un rôle de cache, dans la mesure où l'on y stocke des données issues des bases de données, afin de ne pas avoir à les rechercher lors d'une prochaine requête.

Le cookie en tant que gestionnaire de données a plusieurs avantages :

- Il est intrinsèquement extensible : que vous ayez 100 utilisateurs ou 1 million, vous ne rencontrerez aucune limite de ce côté-ci.
- Il ne pose aucune contrainte sur les techniques de répartition de charge, y compris de niveau DNS. Et si l'on utilise un cookie de domaine, alors tous les serveurs du domaine partagent l'information du cookie.
- L'information placée dans le cookie peut être utilisée aussi bien côté serveur que côté client, puisqu'un petit programme Javascript peut accéder à ces informations.
- Enfin, il peut être persistant, au delà d'une session, c'est à dire que c'est une sorte de cache que l'on peut retrouver une semaine plus tard, pour autant bien sûr que l'utilisateur utilise le même poste.

A l'inverse il a quelques inconvénients :

- Il induit un petit poids sur le trafic réseau, quelques kilo-octets par requête. C'est peu de chose comparé aux poids de pages usuels, mais il faut se souvenir que la bande passante dans le sens client vers serveur est sensiblement moindre en ADSL.
- Il faut analyser la question de la cohérence : lorsqu'une application reçoit des données stockées en cookie sur un poste, elle ne sait pas s'il s'agit de la dernière version de ces données.

- Différents problèmes de sécurité sont à traiter enfin, pour garantir l'intégrité, voire la confidentialité du cookie, ainsi que l'impossibilité de voler un cookie.
- Il existe des risques d'incohérence, par exemple si l'utilisateur fait un 'back' (précédent) sur son navigateur, le cookie reste inchangé.

Revenons sur ces difficultés. La question de la cohérence se règle partiellement en se limitant à des durées de vie réduites. C'est à dire que l'utilisation du cookie persistant pour stocker de l'information est rarement possible car on ne sait pas dans quelle mesure cette information est valide une semaine plus tard, et donc le serveur ne peut rien en faire avant d'avoir récupéré l'information de référence, sans doutes auprès d'une base de données.

Les différents problèmes de sécurité peuvent être résolus par un peu de cryptographie. Typiquement, on ajoutera aux données un horodatage, et on cryptera le tout avec un simple algorithme symétrique, ou bien un algorithme de scellement tel que SHA-5 portant sur le contenu du cookie PLUS une clé secrète uniquement connue côté serveur.

Partage de contexte côté serveur

Nous avons vu que l'une des voies pour éviter de conserver un contexte de session au niveau des frontaux web est d'en transférer la gestion côté client, aux navigateurs, via les cookies. L'autre possibilité est de la transférer côté serveurs, à l'étage de la gestion des données, qui est l'étage naturellement dédié au partage de données.

Si le contexte de session est conservé en base de données, il peut être accédé et mis à jour à partir de n'importe quel serveur frontal. C'est donc une voie qui permet à la fois une répartition de charge sans affinité, et néanmoins un partage de contexte. Et différents outils de développements permettent de rendre cela transparent pour le développeur.

L'inconvénient est un petit impact en performances : un accès à la base de données, même très simple, est beaucoup plus coûteux qu'une lecture en mémoire.

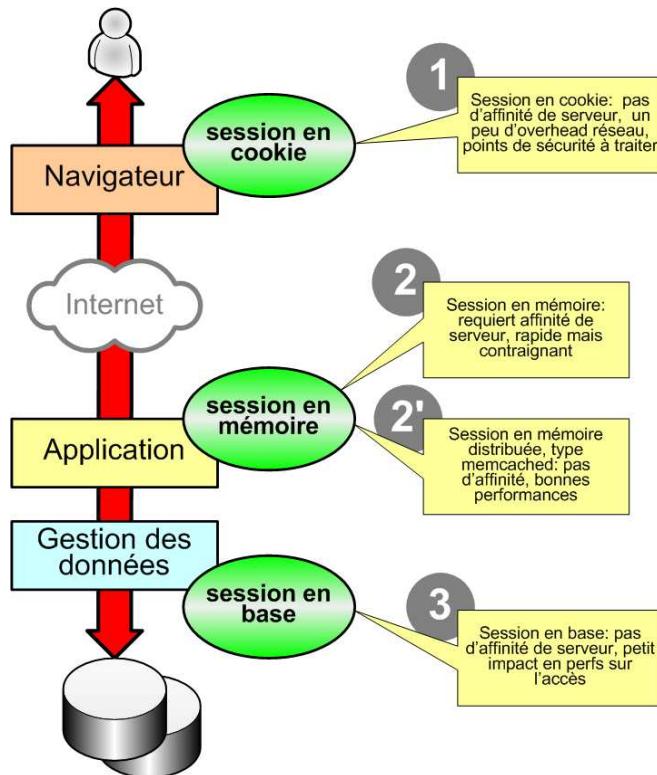
Partage de contexte en cache global

Une autre alternative consiste à partager les contextes de sessions au moyen d'un outil de cache global, tel que *memcached* (cf. page 145). Comme on le verra ce type d'outil, bien que distribué sur un ensemble de serveurs, maintient une copie unique de chaque objet qui lui est confié. N'importe quel serveur frontal peut donc récupérer le contexte de session et le mettre à jour. L'accès est distant, mais purement en

mémoire, avec donc des performances à mi-chemin entre celles d'une gestion locale, et celles d'une gestion en base.

Synthèse

La figure suivante fait apparaître les différentes solutions de gestion de session évoquées :

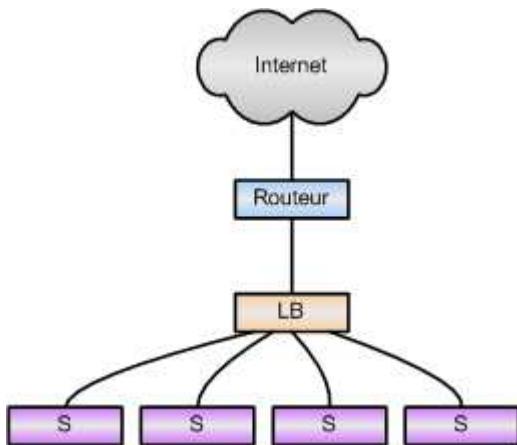


Configuration réseau

Niveau 4, niveau 7, même configuration

Du point de vue de l'architecture réseau, la répartition de charge de niveau 7 est très semblable à celle au niveau 4 : un équipement est placé en amont des serveurs, analyse le flux, et répartit les connexions. Et d'ailleurs, ce sont souvent les mêmes équipements qui peuvent répartir au niveau 4 ou au niveau 7.

Le schéma de principe de la répartition de charge est donc comme suit :



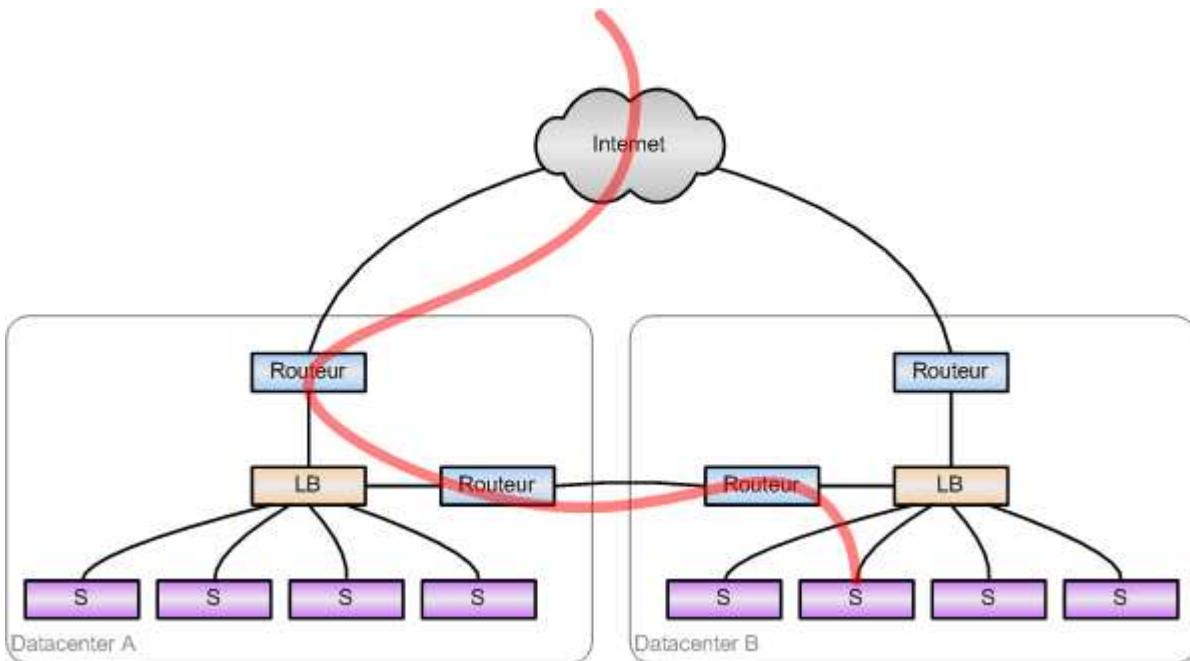
On représente ici un petit nombre de serveurs, mais on peut en intégrer plusieurs dizaines sans difficulté.

Répartition de charge inter-datacenter

Les dispositifs cités ici n'impliquent pas que tous les serveurs soient dans un même datacenter. Rien n'interdit de ressortir sur l'Internet pour trouver un serveur ailleurs. Mais bien sûr, cela impliquerait une forte consommation de bande passante, essentiellement inutile.

C'est pourquoi pour des répartitions intercentres, on préfère généralement les dispositifs de niveau DNS, cités plus haut.

Néanmoins, dans certains cas on peut disposer d'un débit dédié sur fibre optique entre deux centres. Du moins jusqu'à des distances de quelques dizaines de kilomètres, car à l'échelle des continents, le très haut débit coûte très cher, et induit inévitablement un temps de latence important. Si un internaute est d'abord dirigé sur un datacenter DC_A, et que toutes ses requêtes transitent ensuite par ce datacenter A pour atteindre un datacenter DC_B situé dans un autre pays, alors la qualité de service ne pourra être que médiocre. Et la tolérance aux pannes sera médiocre aussi, puisque DC_A reste point de passage obligé.



Ici, typiquement, le load-balancer du datacenter DC_A serait configuré en mode « *priority activation* », de sorte qu'il commence par charger ses propres serveurs, puis au delà d'un certain seuil, il part en débordement sur le second datacenter.

Configuration réseau et tolérance aux pannes de l'équipement

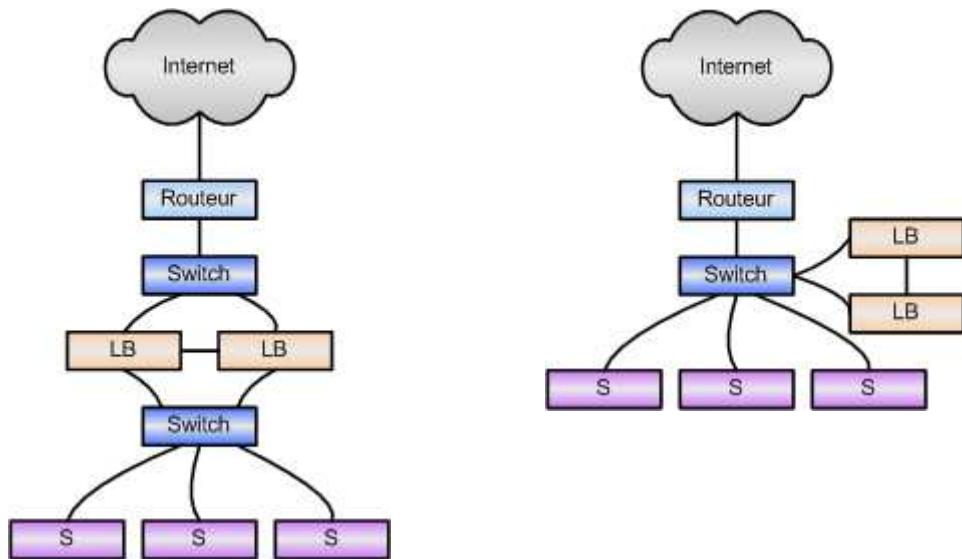
Etant seul, en frontal de tous les serveurs de la plateforme, l'équipement de répartition de charge est un point de fragilité de l'architecture, un « *Single Point of Failure* ». Il est donc nécessaire qu'il soit secouru, avec un dispositif transparent de bascule en secours.

Toutes les solutions de répartition de charge intègrent un tel mécanisme, y compris celles basées sur OpenBSD ou Linux, citées plus loin (cf « Solutions logicielles », page 71).

On met classiquement en place deux équipements en parallèle, dont l'un est actif et l'autre est passif, en *standby*. Les deux équipements sont interconnectés, soit par le switch, soit par un câble croisé, soit par une liaison série dédiée. Sur ce lien, le serveur passif surveille le serveur actif par un échange de « *heartbeat* », de battements de cœur. S'il détecte une panne, le serveur passif devient actif, et reprend l'adresse IP à son compte.

En fait, pour un passage en secours véritablement transparent, les deux équipements doivent échanger plus qu'un simple heartbeat, ils doivent partager la table d'allocation des connexions, de sorte que même les connexions TCP/IP ne soient pas cassées lors du passage en secours.

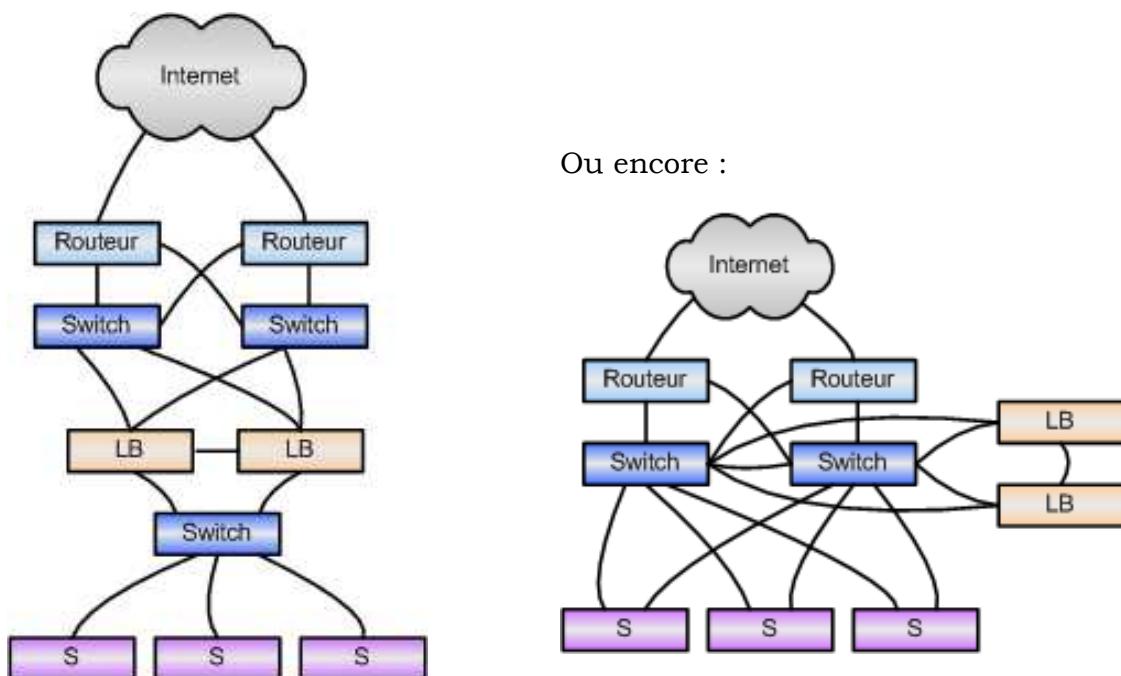
Architectures Hautes-Performances



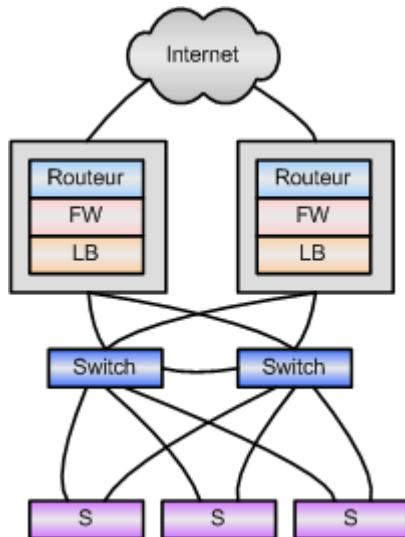
On a fait figurer ici un switch en amont et un autre en aval.

Ici, la configuration avec un switch unique.

Pour une réelle tolérance à la panne d'un équipement, on aura la configuration suivante :



Enfin, dans le cas de load-balancer logiciels, on peut aisément combiner plusieurs fonctions sur les mêmes serveurs, typiquement routeur, firewall et load-balancer, comme suit :



Les solutions et outils

Il y a deux types de solutions pour mettre en œuvre une répartition de charge de niveau 4 ou 7 : solutions logicielles et solutions matérielles.

Les solutions logicielles tournent sur de l'infrastructure de serveur standard, soit sur une base OpenBSD, soit sur une base Linux. On les met en œuvre sur du matériel dédié et peu coûteux.

Les solutions dites matérielles sont des boîtiers dédiés, prêts à l'emploi. Bien sûr, ils incluent du logiciel, mais les boîtiers haut de gamme ne sont pas des serveurs standards, ils utilisent des circuits intégrés spécialisés pour de plus hautes performances.

Solutions logicielles

Des solutions logicielles de grande qualité et d'une robustesse extrême sont disponibles en open source.

Au niveau 4, deux solutions dominent le paysage :

- OpenBSD avec relayd
- Linux avec IPVS

La répartition de charge au niveau 4 appliquant des algorithmes très simples, et uniquement pour la phase de connexion, elle ne requiert que très peu de ressources CPU, et donc un matériel peu coûteux.

A partir des produits cités, on met en place un boîtier de répartition de charge en prenant un hardware relativement bas de gamme mais très fiable, sans disque dur, ni ventilateur, un équipement à moins de 500€.

IPVS

IPVS est une des composantes du projet LVS, Linux Virtual Server, qui réunit différentes fonctionnalités autour du load balancing, afin de construire des serveurs virtuels sur un ensemble de serveurs physiques.

IPVS est intégré au noyau Linux, depuis la version 2.5, et assure une répartition de charge au niveau 4 seulement. Rappelons que la répartition de niveau 4 est compatible avec une diversité de protocoles et donc d'applications au-delà du web.

Il est associé à d'autres outils tels que *mon* pour tester la disponibilité des serveurs et services sous-jacents, *ldirectord* ou *heartbeat* pour tester le load-balancer (*director*) homologue.

LVS supporte tous les algorithmes de répartition évoqués plus haut : round-robin, weighted round-robin, least-connection scheduling, weighted least-connection, locality-based least-connection, locality-based least-connection with replication, destination hashing, source hashing, shortest expected delay, never queue.

Le load-balancer fournit un certain nombre de compteurs statistiques en temps réel, et peut être reconfiguré à chaud.

Deux load-balancer IPVS en normal/secours synchronisent les informations d'état associées aux connexions, de manière à offrir un passage en secours réellement transparent, sans nécessité de ré-établir les connexions en cours.

LVS offre aussi des outils de protections contre les attaques DoS.

Relayd

Si OpenBSD est un Unix moins répandu que Linux sur les serveurs applicatifs, il est très populaire pour construire des boîtiers *appliance*, du fait de sa grande robustesse et d'excellents outils de networking.

Relayd en fait partie. C'est un outil de routage et de load-balancing de niveau 4 et 7.

Pour le niveau 4, il s'appuie en fait sur un autre outil BSD : *pf, packet filter*, qui prend en charge le routage au niveau du noyau.

Relayd permet de définir des pools de serveurs, des « *tables* », qui sont surveillés.

Quelques fonctionnalités :

- Il permet différents modes de répartition : round-robin, équilibrage des connexions, hashage sur des paramètres de la requête Http.
- Il permet différents modes de test des serveurs : ICMP, simple connexion TCP, requête-réponse sur port TCP, requête HTTP. Possibilité de timeout sur les requêtes de tests.
- Il peut adresser des alertes SNMP à un système de supervision.

HAProxy

HAProxy est l'une des solutions logicielles les plus utilisées au niveau 7. Solution très complète au plan fonctionnel, extrêmement robuste et performante, c'est de surcroît un projet particulièrement dynamique.

HAProxy peut fonctionner aussi au niveau 4, mais on l'utilise plutôt au niveau 7.

Quelques fonctionnalités :

- Répartition sur la base d'un cookie existant, ou bien insertion de son propre cookie pour gérer l'affinité de serveur.
- HAProxy peut tester la disponibilité des serveurs, soit en établissant une connexion TCP, soit en adressant une requête HTTP. En mode HTTP, la vérification peut porter sur une URI particulière, qui teste tous les composants requis au bon fonctionnement.
- Le seul mode de répartition possible pour l'instant est le mode *round-robin* (permutation circulaire), avec toutefois la possibilité de permutation et de limitation du nombre de connexions par serveur.
- La reconfiguration est possible à chaud.
- Des statistiques sont disponibles en temps-réel.
- La version 1.3, en développement apportera d'autres logiques de répartition et des fonctionnalités complémentaires telles que le *content-switching*, aiguillage basé sur l'analyse des URI.

HAProxy est généralement mis en œuvre couplé à *heartbeat*, un composant du projet Linux-HA, qui permet d'associer un HAProxy de secours, qui surveille le load-balancer principal, et reprend sa fonction, par son adresse IP, s'il détecte une défaillance.

Apache mod proxy

L'extension Apache *mod_proxy_balancer* est aussi couramment utilisée dans une fonction de *load balancer*.

Il s'agit bien sûr d'une répartition de niveau 7, et il est possible en particulier de maintenir les sessions sur la base des cookies placés par les applications, de type JSessionId ou PHPSESSIONID.

Il permet différents modes de répartition :

- Répartition simple des requêtes entre les serveurs, avec pondération.
- Répartition vers le serveur qui a le moins de requêtes en cours (*pending request counting*)
- Répartition visant à équilibrer le flux traité en termes de nombre d'octets en requête/réponse, avec pondération éventuelle. Ce type de répartition équilibre la bande passante, mais pas nécessairement la charge.

Dans l'ensemble, la répartition de niveau Apache est moins performante que celle d'un outil spécialisé tel que HAProxy. Mais si l'on dispose déjà d'un serveur Apache, par exemple pour servir les contenus statiques, alors on peut choisir de lui confier aussi la répartition de charge.

Les boîtiers dédiés

Les boîtiers dédiés sont plus robustes, ils peuvent atteindre de meilleures performances et offrir quelques fonctionnalités supplémentaires.

L'offre du marché est assez vaste (Foundry ServerIron, F5 BigIP, Citrix Netscaler, Cisco ACE, Nortel Alteon), mais ce sont des équipements très coûteux, par exemple de l'ordre de 15 k\$ pour un boîtier Foundry ServerIron 4G-SSL.

La performance n'est pas le critère de choix premier, car les solutions logicielles ont déjà des performances satisfaisantes. La question est un peu différente pour une répartition de niveau 7, qui consomme plus de CPU.

Les boîtiers dédiés ont aussi en général une meilleure interface de configuration et de gestion.

Les boîtiers de répartition de charge tels que BigIP offrent quelques possibilités supplémentaires, telles que la sélection du serveur dont le temps de réponse observé est le plus bas, ce qui est interprété comme une moindre charge réelle (il s'agit d'une analyse de niveau 7). Mais là aussi, il peut y avoir des anomalies dans la mesure, un cas de plantage classique est celui où le temps de réponse d'un serveur est très faible *parce que* l'application est plantée et répond immédiatement par une page d'erreur. Ainsi, si le load-balancer est mal configuré, il dirige tout le trafic vers le serveur en panne, de sorte qu'une panne locale devient globale.

Fonctionnalités associées

En plus de sa fonction de répartition de charge, le composant de load balancing, qu'il soit logiciel ou boîtier, de par son positionnement en coupure du trafic HTTP, peut assurer quelques autres fonctions, et décharger du même coup le serveur web. Typiquement il s'agit des fonctionnalités de :

- Cryptage SSL, qui est assez consommateur de CPU. On a vu qu'il était obligatoire pour répartir au niveau 7.
- Cache. La gestion du cache fait l'objet d'un chapitre dédié (cf. page 123). Côté serveur, elle peut être mise en œuvre au même niveau que la répartition de charge.
- GZip. Nous avons décrit déjà l'intérêt de compresser les flux jusqu'au navigateur (cf. « *Compression du flux* », page 43). C'est aussi une tâche qui consomme un peu de CPU et qui peut être prise en charge à un niveau frontal.
- Maintien des connexions. Enfin, les boîtiers frontaux gèrent parfois le partage des connexions, en particulier pour les clients utilisant du HTTP 1.0. Dans ce cas, plusieurs requêtes HTTP, parvenant sur des connexions TCP distinctes, sont multiplexées sur une même connexion à destination du serveur web.
- Protection contre les attaques « DOS », déni de service, en limitant le nombre de connexions autorisées.

Répartition peer-based

Il existe un dernier mécanisme de répartition, moins couramment déployé, mais qui présente des caractéristiques uniques, que l'on

appelle le *peer-based load-balancing*, c'est à dire de la répartition de charge entre pairs, entre homologues.

Une de ses caractéristiques est de ne pas faire intervenir d'équipement spécifique pour gérer la répartition, ce qui procure une excellente résistance aux pannes.

En fait, le mécanisme est indissociable d'un premier niveau de répartition en DNS round-robin. On a vu que le DNS round-robin était satisfaisant pour ce qui est de la répartition, mais moins pour ce qui est de la tolérance aux pannes. Le *peer-based load-balancing* ajoute la tolérance aux pannes.

L'une des caractéristiques de ce mode est de nécessiter un noyau d'OS spécifique, c'est probablement ce qui en a limité l'utilisation. La solution implémentant le *peer-based load-balancing* est Wakamole².

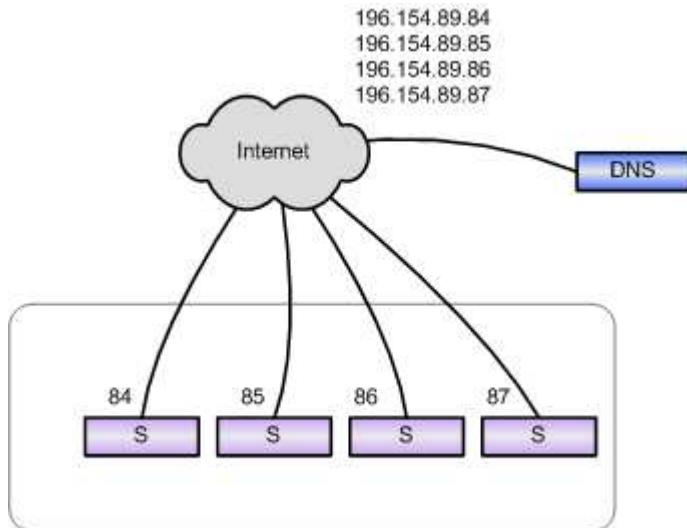
Selon le principe du DNS-Round-Robin, le nom du serveur est mis en correspondance avec différentes adresses IP, de manière cyclique. Sur la masse des internautes, cela crée donc une répartition de charge équilibrée entre les N adresses IP. Ce nombre N d'adresses n'est pas égal au nombre P de serveurs physiques. On peut utiliser ainsi par exemple 20 adresses pour 5 serveurs.

Les P serveurs physiques échangent des messages entre eux pour se surveiller et négocier la répartition des N adresses IP sur les P serveurs physiques. A priori, chaque serveur prend en charge N/P adresses. Mais si l'un des serveurs est indisponible, la répartition est redéfinie entre les P-1 serveurs restants.

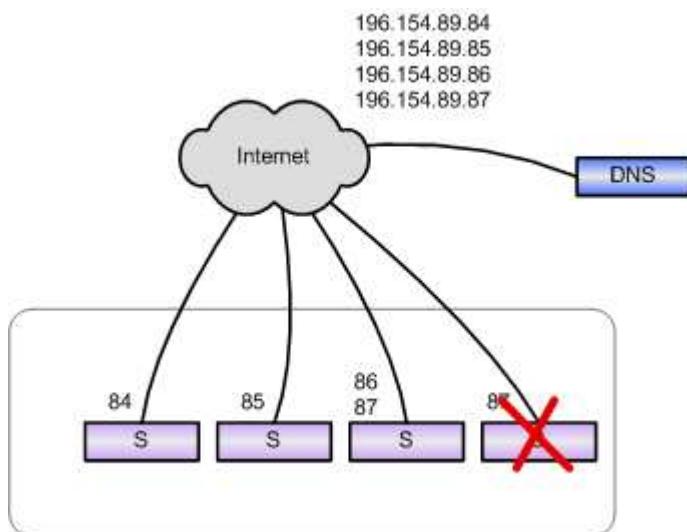
C'est donc un mécanisme qui ressemble un peu à celui utilisé pour la gestion du secours, dans lequel un serveur surveille l'autre et reprend son adresse IP, donc sa fonction, en cas de panne. Ici, le principe est le même, mais au lieu d'être un jeu à 2, c'est un jeu à N serveurs.

² <HTTP://www.backhand.org/wackamole/>

Architectures Hautes-Performances



Dans le cas d'une panne du serveur de droite, l'IP 87 est réaffectée sur l'un des autres serveurs :



Ici, on a du coup une charge double sur le troisième serveur, ce qui n'est pas bon. Il faut en fait brasser un nombre plus important d'IPs pour avoir plus de flexibilité dans la réaffectation : si chaque serveur avait initialement 5 IPs, alors on aurait réparti les 20 adresses sur 3 serveurs en 7, 7 et 6.

Le principe du peer-based va dans le sens du *share-nothing*, ce qui est particulièrement intéressant, mais il faut reconnaître que sa mise en œuvre est restée plutôt confidentielle.

Load-balancing sur des services internes

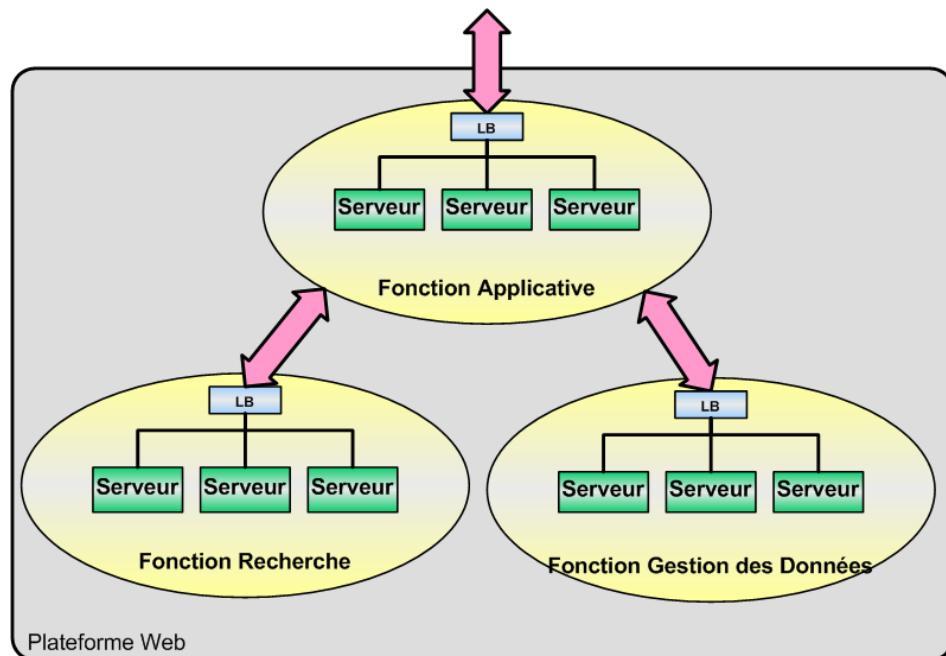
Nous avons beaucoup parlé du load-balancing sur l'étage frontal HTTP, mais le même principe peut s'appliquer en interne d'une plateforme web.

Les interfaces de type REST, que nous avons décrites plus haut, sont particulièrement adaptées à ce type de répartition, puisqu'elles sont sans état et construites sur du HTTP, et donc peuvent bénéficier exactement des mêmes mécanismes.

Imaginons ainsi une plateforme web qui utilise un service web interne, par exemple pour la recherche. On a donc mis en place un service REST de recherche, auquel on passe une série de mots-clés, et qui répond par une liste d'objets. Dans un premier temps ce service interne a pu être implémenté sur un unique serveur. Mais l'audience grimpe, et voilà que la recherche devient le maillon critique de la plateforme.

Rien n'est plus facile alors que de répartir le service de recherche REST sur différents serveurs ; on obtiendra alors à la fois une capacité accrue, et une bonne tolérance aux pannes.

Ce que l'on peut représenter ainsi :



Bien sûr, c'est une vision fonctionnelle, du point de vue réseau, les trois domaines fonctionnels peuvent partager le même équipement de load-balancing.

MapReduce et Hadoop

Nous avons parlé jusqu'ici de la répartition de charge temps-réel, c'est-à-dire portant sur le traitement synchrone de requêtes. Mais certains travaux batch, portant sur des téraoctets de données, doivent également être partitionnés.

MapReduce est un mécanisme de partitionnement de tâches en vue d'une exécution distribuée sur un grand nombre de serveurs. Le principe est simple : il s'agit de décomposer une tâche en tâches plus petites, ou plus précisément découper une tâche portant sur de très gros volumes de données en tâches identiques portant sur des sous-ensembles de ces données. Les tâches (et leurs données) sont ensuite dispatchées vers différents serveurs, puis les résultats sont récupérés et consolidés. La phase amont, de décomposition des tâches, est la partie *map*, tandis que la phase aval, la consolidation des résultats est la partie *reduce*.

Le principe est simple donc, mais l'apport de l'algorithme MapReduce est de bien conceptualiser, en vue de normaliser les opérations d'intendance de sorte qu'un même framework puisse prendre en charge une diversité de tâches partitionnables, ce qui permettra aux développeurs de se concentrer sur les traitements proprement dits, tandis que le framework prend en charge la logistique de la répartition.

La normalisation autour de ces principes s'appuie sur la manipulation de couples (clé, valeur). La tâche initiale est un couple (clé, valeur), et chacune des tâches intermédiaires également. Chaque tâche intermédiaire retourne un résultat sous la forme d'un couple (clé, valeur), et la fonction *reduce* combine tous ces résultats en un couple (clé, valeur) unique.

L'algorithme MapReduce est principalement destiné à des tâches de type batch, de très grande ampleur, portant sur de très grands volumes de données. La décomposition typique consiste à découper le volume de données initial en N volumes plus petits, qui peuvent être traités séparément. L'exemple souvent pris pour illustration est le comptage du nombre d'occurrence de chaque mot d'un très grand fichier. Le grand fichier est décomposé en plus petits fichiers, et les occurrences sont comptées sur chacun d'eux. La fonction *reduce* additionne les décomptes obtenus sur chaque fichier intermédiaire.

La modélisation a été proposée initialement (2004) par des informaticiens de Google pour mieux gérer les gigantesques tâches portant sur des téra- ou péta-octets de données, et les distribuer sur des dizaines de milliers de serveurs. Puis Doug Cutting, le créateur de Lucene, s'en saisit et lance le projet Hadoop, dans le cadre de la fondation Apache. Cutting sera ensuite embauché par Yahoo pour

Architectures Hautes-Performances

refondre toute la technologie de recherche du portail, en intégrant Hadoop et HDFS, qui restera intégralement open source tout en intégrant l'essentiel des développements financés par Yahoo.

Dans Hadoop, toutes les entités manipulées, les couples (clé, valeur), doivent être sérialisables, de sorte que l'on puisse les transmettre d'un serveur à un autre. Mais, puisqu'on a dit qu'il s'agissait de très gros volumes, il faut aussi optimiser l'utilisation de la bande passante sur le réseau. C'est pourquoi MapReduce est généralement utilisé en combinaison avec un système de gestion de fichiers distribué, dans le cas de Hadoop il s'agit de HDFS. Dans cette logique, chaque serveur est à la fois un outil de calcul et un outil de stockage. La fonction de mapping cherchera alors à attribuer chaque tâche à un serveur qui stocke *déjà* les données correspondantes.

Enfin, puisqu'il est destiné à fonctionner sur des milliers de serveurs, Hadoop gère également la tolérance aux pannes, c'est-à-dire qu'il réaffecte les tâches de manière transparente en cas de panne d'un serveur.

HAUTE DISPONIBILITE

Haute disponibilité

La disponibilité qui compte n'est pas celle de tel ou tel équipement, mais celle vécue par l'utilisateur utilisant le service, indépendamment des moyens mis en œuvre pour l'obtenir.

Tous les sites rencontrent des pannes ... Même les plus grands. Rien qu'en 2008, Amazon.com ou bien Voyages-Sncf ont connu des interruptions importantes. Est-ce une fatalité ?

Les plateformes web ont en général des exigences de disponibilité très supérieures à celles des applications d'entreprise, même critiques. D'une part parce qu'à toute heure il reste quelques internautes connectés. D'autre part, et surtout, parce que c'est la réputation de l'entreprise qui est en jeu. Sur une application d'entreprise, un arrêt d'un quart d'heure est généralement toléré, surtout s'il a pu être anticipé. Sur un portail de l'Internet, la moindre indisponibilité est perçue comme une forme d'incompétence : « *Si ma banque ne sait pas offrir un service web qui tourne parfaitement, alors comment saura-t-elle gérer mon argent ?!* ».

Tolérance aux pannes

A tous les niveaux

Bien sûr, l'une des causes d'indisponibilité est la panne d'un composant, qu'il soit matériel ou bien logiciel. Et c'est pourquoi on parle souvent de *tolérance aux pannes*, c'est à dire de la capacité d'une plateforme à continuer de fonctionner en présence d'une panne.

La tolérance aux pannes peut être gérée à différents niveaux :

- Composants élémentaires : disques, mémoire, etc.
- Serveur, routeur, autres équipements de type *appliance*.
- Ressources partagées de niveau datacenter

Pour la meilleure disponibilité, il faut en pratique combiner la tolérance aux pannes sur ces différents niveaux, en visant le meilleur rapport disponibilité / prix global.

Typiquement, vaut-il mieux généraliser des disques RAID sur tous ses frontaux pour en réduire la probabilité de panne unitaire, ou bien accepter la panne disque et traiter la redondance au niveau des serveurs ?

Bonnes pratiques

Quelques bonnes pratiques de la haute disponibilité

Une première règle en matière de haute disponibilité est de *ne jamais compter sur le support du fabricant*, quel que soit le délai d'intervention garanti.

Un technicien sera peut être là sous 4 heures, mais 4 heures c'est très long, et de plus il est probable qu'il ne pourra réparer immédiatement.

Il faut donc à minima disposer de tous les composants de secours sur place. On peut en revanche, compter sur un certain niveau de mutualisation des composants de secours, et à ce titre, plus la plateforme est grande, moins le secours coûte cher.

Il convient également de disposer d'une gestion de configurations virtualisées, permettant d'installer n'importe quel serveur sur n'importe quel hardware en quelques minutes.

Enfin, on peut considérer que, à partir d'une dizaine de serveurs, la panne doit être considérée comme un événement anodin, avec un secours totalement transparent.

Pannes logicielles

Il faut bien garder à l'esprit que les pannes dues au logiciel sont bien plus nombreuses que celles dues au matériel défaillant.

Et pourtant il est courant que l'on investisse davantage sur la redondance matérielle que sur les process assurant la qualité du logiciel. Typiquement, on prévoira un serveur de secours, mais on trouvera trop coûteux de disposer d'une plateforme de préproduction. De sorte que les nouvelles versions d'applications ne seront pas testées dans un environnement suffisamment représentatif, ce qui causera des anomalies et indisponibilités. Redisons-le : il est souvent plus important d'investir pour se prémunir des défauts logiciels que des pannes matérielles. D'autant que lorsqu'une panne logicielle survient,

Architectures Hautes-Performances

la redondance le plus souvent ne sert à rien car le même logiciel est déployé sur les composants de secours.

La haute disponibilité est affaire de process, bien plus que d'infrastructure : process de développement et de déploiement, mais aussi process de supervision et d'exploitation.

Enfin, en matière de haute disponibilité comme ailleurs, la simplicité doit prévaloir. Il arrive couramment qu'une configuration visant la disponibilité au prix d'une complexité trop grande subisse des arrêts *à cause même* de cette complexité ou mauvaise maîtrise.

C'est pourquoi il faut le redire, le marteler : 9 fois sur 10 les interruptions de service proviennent du logiciel et non du matériel. Certes, à l'intérieur de ces 9 fois, une bonne part correspondra à une mauvaise réaction du logiciel à un incident d'origine matérielle ou un événement extérieur exceptionnel. Ce pourra être typiquement un fichier de log devenu trop gros, ou un administrateur qui lance un job, disons de purge par exemple, qui va bloquer la base de données.

Exploitation

Beaucoup d'organisations ont des processus d'exploitation qui portent encore la marque d'une époque ancienne, où l'on pouvait arrêter le service, la nuit vers 3 heures du matin, de manière à mettre en place différents travaux batch, et à l'occasion, relancer les serveurs. A l'évidence, le web a changé la donne. Non seulement parce qu'il y a toujours des internautes en ligne, à toute heure de la nuit, mais aussi parce que la plateforme web sert des internautes du monde entier, y compris des internautes pour lesquels il est 9h du matin. Tout cela semble évident, et pourtant on rencontre encore trop souvent des sites qui ferment une demi-heure chaque nuit, ou bien une fois par semaine.

C'est en fait dès la conception des applications qu'il faut intégrer l'exigence d'une exploitation 24/7.

Changements de version

La règle générale en informatique est que *ce qui a marché continue de marcher*, du moins tant qu'on ne touche à rien, et qu'il n'y a pas d'événement extérieur nouveau.

Une grande majorité des interruptions de service se produit à l'occasion d'un changement de version.

Il est extrêmement difficile de valider une version pour le monde réel. Les scénarios de test, même si l'investissement est important, n'auront parcouru que quelques centaines de cas de figure, et d'un seul coup

l'application sera confrontée à quelques centaines de milliers, voire millions de cas de figure. Il y a là un saut quantique auquel peu d'applications résistent.

Nous ne pouvons pas donner ici toutes les bonnes pratiques de validation du logiciel, mais trois points essentiels seulement :

- Pour une plateforme qui est vivante, et reçoit régulièrement de nouvelles versions, *la pratique de l'intégration continue et des tests automatisés* est déterminante pour assurer la non-régression.
- La validation n'étant jamais assez complète, une bonne pratique est *la mise en service progressive*, que ce soit sur une typologie d'utilisateurs, sur une région, sur une courte période.
- Il faut pour cela concevoir des mises en service avec possibilité de *retour arrière*, et si possible faire *une distinction entre les changements techniques et les changements visibles*.

Redondance et secours

D'une manière générale, la tolérance aux pannes consiste à mettre en œuvre, pour assurer une fonction, N équipements dont P suffisent.

On peut parler soit :

- De secours actif, dans le cas où les N équipements sont opérationnels en nominal
- De secours passif, lorsque P équipements sont opérationnels, et N-P sont en réserve, en secours, en « *spare* ».

Les questions clés sont bien sûr : Quel process, quel délai, pour que le secours intervienne, et quel impact visible l'événement a-t-il en termes de disponibilité ?

Surveillance et passage en secours

La première étape est la détection de la panne et l'activation du secours. On peut distinguer :

- La surveillance *par l'équipement homologue*, et la reprise de la fonction par l'homologue.
- La surveillance *par l'étage amont*, par l'appelant, et la réaffectation de la fonction à un homologue, qu'il soit actif ou passif.

- La surveillance *par la supervision*, et la réaction prise en charge à ce niveau.

Surveillance par l'homologue et *heartbeat*

C'est le cas de deux équipements placés en parallèle, que ce soit avec un secours actif ou passif.

Typiquement, avec secours passif, on a un équipement de secours qui teste en permanence l'équipement principal pour vérifier son bon fonctionnement. Ce test est l'appel périodique d'un service qui témoigne du bon fonctionnement. On parle alors de *heartbeat*, c'est à dire de battement de cœur : tant qu'on entend battre son cœur, c'est qu'il va bien.

Pour plus de robustesse, l'échange de *heartbeat* peut se faire hors du réseau local, sur un câble Ethernet croisé, ou une liaison série.

S'il détecte la panne de l'équipement actif, l'équipement de secours reprend la fonction, en reprenant l'adresse IP. Il s'affecte l'adresse IP, puis envoie en *broadcast* un message d'effacement du cache ARP, afin que les switchs puissent associer son adresse physique (MAC) à l'adresse IP.

Dans certains cas, l'échange entre les équipements peut porter sur davantage qu'un simple signe de vie, et inclure une vraie synchronisation de données, afin de reprendre la fonction *avec son contexte*, et donc de manière transparente pour les clients.

Surveillance par l'étage amont

C'est typiquement le cas où l'on a un *load-balancer* en amont, affectant les requêtes à différents serveurs.

Le *load-balancer* teste lui-même le bon fonctionnement des serveurs. Cela peut se faire à différents niveaux : ICMP, test de connexion TCP, test au niveau HTTP, ou bien appel d'un service spécifique de surveillance.

Lorsqu'il détecte un équipement défaillant, le *load-balancer* le sort de la répartition, et peut adresser une alerte à la supervision.

Soulignons que la qualité de ce test est primordiale. Un cas d'effet de bord très classique est celui où un serveur en panne répond plus vite que tous les autres, précisément parce qu'il est en panne. Il répond au niveau HTTP, mais répond une erreur, ou pire encore une page bien formée mais comportant un message d'erreur. Dans certains cas, le *load-balancer* dirigera les requêtes de manière privilégiée vers ce serveur défaillant, transformant une panne partielle en indisponibilité globale.

Secours passif

Sur une plateforme web, on vise le plus souvent un secours transparent (« *transparent failover* »), c'est à dire s'opérant de manière automatique.

Le secours à froid, le secours passif, échoue très souvent.

Il échoue pour différentes raisons :

- La configuration du secours est obsolète ou erronée
- Le mode opératoire du passage en secours est mal maîtrisé
- La bonne personne n'est pas là.

On priviliege donc un secours transparent, mais il doit aussi être revalidé très régulièrement, et l'on préfère un secours actif, c'est à dire qui participait au travail en temps normal.

Secours actif

Dans un principe de secours actif, le composant de secours est déjà opérationnel avant d'être requis. Il est en surnombre, mais assure malgré tout une part de la fonction. Le secours actif est associé aussi à un dispositif de passage en mode secours transparent ou automatisé. On parle d'un secours « à chaud ». Les disques RAID, typiquement, fonctionnent selon un principe de secours actif.

En règle générale, le secours actif a beaucoup d'avantages :

- Il réduit considérablement les risques. Risque que le composant de secours lui-même soit défaillant ou bien mal configuré, risque d'une procédure de passage en secours mal maîtrisée.
- Il permet de tirer parti du surplus de matériel pour disposer d'une meilleure qualité de service même en l'absence de secours.
- Et bien sûr, le passage en secours automatisé réduit la durée des incidents, et limite les besoins humains, pour autant que tout se passe comme prévu.

Aux rangs des inconvénients, on peut citer :

- Une moindre flexibilité, et donc un coût matériel potentiellement supérieur : chaque plateforme d'un datacenter doit avoir son serveur de secours, (voire même un pour chaque grande fonction de la plateforme) alors qu'un hébergeur pourrait choisir de mutualiser un petit nombre de serveurs de secours pour différents clients.

Gestion mutualisée du secours

Une gestion mutualisée du secours consiste à disposer d'un ensemble (« *pool* ») de serveurs en *spare*, qui peuvent être utilisés pour remplacer différentes fonctions de la plateforme.

C'est une voie qui peut être nécessaire lorsqu'on a spécialisé les serveurs, que ce soit une spécialisation fonctionnelle, ou une spécialisation par partitionnement des données.

Bien sûr, cela suppose que la même configuration hardware puisse convenir aux différentes fonctions. Et cela implique aussi un process de passage en secours plus long, puisqu'il implique l'installation de la configuration spécifique à la fonction secourue.

La généralisation de la virtualisation des serveurs facilite grandement cette installation rapide du secours : il convient de disposer des VMs correspondant à toutes les fonctions de la plateforme, prêtes à l'emploi.

C'est un secours qui pourra être rendu automatique, mais ne sera pas instantané. A moins de disposer d'un SAN qui permette de faire repartir immédiatement un nouveau serveur sur une VM déjà installée sur une partition disque.

Single Point of Failure

« SPOF », c'est un acronyme célèbre en matière de disponibilité : le *Single Point Of Failure*, *Point Unique de Panne*, est un composant critique qui n'est pas secouru, un point de fragilité de la plateforme.

Dans une architecture, il est essentiel évidemment de l'avoir parfaitement analysé. Pour autant, cette analyse doit faire intervenir les probabilités de panne très différentes des composants : câble, switch, routeur, cpu, alimentation, disque, ...

Il faut garder à l'esprit aussi que même si un composant est secouru, il devient critique dès la première panne, et jusqu'à réparation. Le temps moyen de réparation (MTTR *Mean Time To Repair*) est donc un facteur essentiel.

MTBF et probabilité de panne

On appelle MTBF, *Mean Time Between Failures*, le temps moyen entre pannes. C'est une caractéristique essentielle exprimant la fiabilité d'un composant. Ce n'est pas tout à fait la *durée de vie moyenne*, qui est une autre notion. L'idée est que à l'intérieur de la *durée de vie de l'équipement*, le MTBF exprime la probabilité de panne. Précisément, le MTBF est l'inverse de l'espérance de panne. Par exemple, si le MTBF

est de 10 ans, l'espérance de panne sur un an est de 0,1 : il y aura en moyenne 0,1 panne par an.

Les opérateurs qui exploitent plusieurs milliers de serveurs disposent de bonnes statistiques quant aux pannes. Le MTBF d'un serveur se situe entre 5 et 10 ans. 10 ans, cela signifie que si vous exploitez 10 serveurs, il vous faut compter sur une panne par an. Si vous exploitez 100 serveurs, environ une panne par mois. On voit bien qu'à partir de cette dimension, la panne est totalement banalisée. Et c'est lorsqu'elle est banalisée que l'on atteint la meilleure disponibilité.

Il existe différentes formules statistiques pour brasser les MTBF et les probabilités de panne. Sous Excel, vous disposez d'une fonction LOI.POISSON, qui vous donne la probabilité de N pannes pour une espérance donnée.

Il est intéressant par exemple de calculer ainsi la probabilité de double panne, comme sur le tableau suivant :

| MTBF (ans) | 3 | 7 | 31 |
|--|---------------|---------------|---------------|
| Espérance de panne sur 1 an | 0,333 | | |
| Probabilité de zéro panne sur 1 an | 0,717 | | |
| Probabilité d'une panne ou plus sur 1 an | 0,283 | | |
| Temps pour réparer, en jours | 2 | 7 | 31 |
| Espérance de panne unitaire sur ce délai | 0,002 | 0,006 | 0,028 |
| Probabilité de zéro panne sur ce délai | 0,998 | 0,994 | 0,972 |
| Probabilité d'une panne ou plus sur ce délai | 0,002 | 0,006 | 0,028 |
| Probabilité de double panne sur l'année | 0,0005 | 0,0018 | 0,0079 |
| Probabilité de double panne sur l'année (%) | 0,052% | 0,181% | 0,791% |
| Disponibilité sur l'année | 0,9995 | 0,9937 | 0,8774 |

Si l'espérance d'occurrence d'un événement sur une période donnée est de E, alors la probabilité de N occurrence est LOI.POISSON(N, E, FAUX), et la probabilité de N occurrences ou moins est de LOI.POISSON(N, E, VRAI).

On voit ici que, pour un MTBF de 3 ans, la disponibilité malgré la redondance tombe en dessous de 0.995 pour un temps de réparation de 7 jours ce qui n'est pas négligeable, et en dessous de 0,9 s'il faut un mois pour réparer.

La brique de base, le serveur élémentaire

Supposons que les mesures de dimensionnement conduisent à une capacité de 100 pages par seconde sur un serveur, et que votre audience maximum à l'heure de pointe correspond à 150 pages par seconde. Il vous faut donc idéalement 1,5 serveurs, ce que raisonnablement on arrondit à 2 serveurs. Mais pour résister à une panne, on doit donc mettre en place 3 serveurs, c'est à dire deux fois la capacité nécessaire.

Si l'on pouvait disposer de serveurs ayant la moitié de cette capacité, pour la moitié du prix, alors on mettrait trois serveurs pour tenir 150 pages par seconde, un quatrième pour le secours. Soit au final 4 serveurs, mais pour les 2/3 du prix.

D'une manière plus générale, en matière de disponibilité, plus les unités élémentaires sont petites, moins la tolérance aux pannes est coûteuse, puisque le coût relatif de l'équipement de secours est de $1/N$, où N est le nombre de serveurs en nominal.

Par ailleurs le meilleur rapport performance/prix est généralement obtenu vers le bas de la gamme.

C'est une analyse que nous avons déjà citée en introduction (cf. « Quelle cellule élémentaire, quelle brique de base ? », page 15).

Monitoring et alertes

Service complet, scénarios applicatifs.

Les internautes s'énervent, pestent, maugréent, et vont voir ailleurs, mais très rares sont ceux qui se donneront le mal de vous remonter les problèmes qu'ils rencontrent, et encore moins leurs vagues insatisfactions. Lorsqu'un internaute proteste, il faut bien comprendre qu'il est le porte-parole de 1000 internautes silencieux. On ne peut donc pas balayer sa critique en se disant « les autres ne semblent pas avoir de problème... ». Il faut au contraire saisir l'occasion de chaque remontée des internautes, dysfonctionnement ou simple critique d'ergonomie.

On ne s'intéresse pas ici à la qualité générale de votre service ni à son ergonomie, mais à sa disponibilité seulement. Si votre serveur ne répond pas, vous pouvez être certains que personne ne va vous appeler, d'autant que même le numéro de téléphone ne sera sans doute pas disponible.

Il est donc impératif de mettre en place un dispositif de surveillance.

Nous parlerons ailleurs de la supervision de plateforme, qui est un sujet un peu différent. Nous traitons ici de la surveillance du service au niveau fonctionnel, c'est à dire tel qu'il est vu par ses utilisateurs. On parle parfois de surveillance fonctionnelle versus surveillance organique. La surveillance organique s'intéresse aux organes, c'est à dire aux composants, à différentes échelles : un simple disque ou bien une base de données. La surveillance fonctionnelle s'intéresse à la fonction, au service rendu.

Monitoring Http

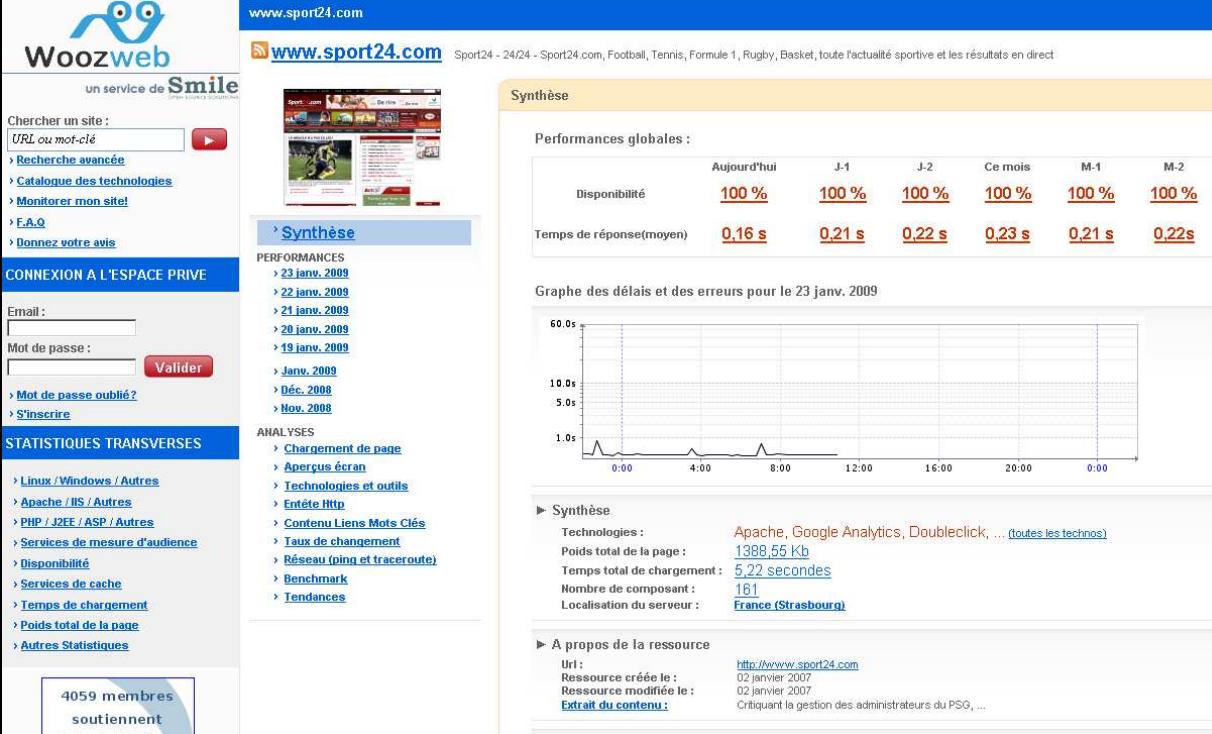
Il existe différentes solutions de surveillance, ou monitoring. Les plus simples consistent à demander périodiquement la page correspondant à une URL donnée. Il convient de vérifier ensuite si le serveur répond, s'il n'y a pas de code d'erreur, mais aussi si la page retournée est conforme, et enfin si le temps de réponse est acceptable. Dans tous les cas d'erreur, le service de monitoring pourra alerter un exploitant, que ce soit par email ou bien par SMS.

Pour surveiller avec un minimum de profondeur, on évite généralement de demander une page d'accueil, ou bien de simple menu, ou bien une page qui sera servie depuis le cache. On s'attache au contraire à demander une page qui va solliciter toutes les ressources de la plateforme. Ce peut être un résultat de recherche par exemple, mais on choisit parfois de réaliser une page spécifique, qui teste un ensemble de composants, et affiche un compte rendu global.

Pour des services de haut niveau, la surveillance fonctionnelle ne se satisfait pas d'une page unique, même complexe. Il faut valider l'ensemble du service au moyen de *scénarios d'utilisation* : l'achat d'un billet d'avion, le passage d'un virement, l'inscription, la saisie d'une contribution, etc. Des processus qui impliquent une suite de requêtes exécutées dans le cadre d'une session. Ce genre de scénarios ne peut être exécuté qu'au moyen d'un vrai navigateur. En effet, il y a trop de choses dans un site web moderne qui peuvent échouer et ne relèvent pas du Html pur : javascript, cookies, ajax, flash, etc. On utilisera pour cela des outils tels que Selenium, intégré aux navigateur Firefox.

Architectures Hautes-Performances

Woozweb



The screenshot shows the Woozweb monitoring interface for the website www.sport24.com. The main page displays a summary of site availability and response times. On the left, there's a sidebar with various monitoring categories such as CONNEXION A L'ESPACE PRIVE, STATISTIQUES TRANSVERSES, and ANALYSES. The right side provides detailed performance data, including a graph of response times over time and specific metrics for the day of January 23, 2009.

Woozweb est un outil de monitoring gratuit, qui permet :

- De surveiller un ou plusieurs sites, c'est-à-dire plus précisément des URLs.
- D'alerter en cas d'indisponibilité, de temps de réponse excessif ou de page non-conforme (non-présence d'un mot clé attendu dans la page).
- De suivre la qualité de service dans le temps par différents graphes : temps de réponse, disponibilité, par jour / par mois.
- De recevoir des états de synthèse.

Woozweb teste les sites depuis deux serveurs sondes, l'un en France, l'autre aux Etats-Unis. Chaque sonde teste chaque site une fois toutes les 15 minutes.

Il existe d'autres services de monitoring, mais Woozweb est celui qui offre le service le plus complet, et la surveillance la plus rapprochée, parmi les outils gratuits.

LA GESTION DES DONNEES

Gestion des données

Un problème difficile

Nous avons vu que la gestion des traitements était finalement assez facile à partitionner, et donc à rendre extensible. Simplement parce que dans un paradigme (traitement, données), c'est à l'étage données que se situe le besoin de partage, d'interaction et de synchronisation.

Nous éviterons de parler trop vite de *base de données*, pour mieux analyser les problématiques relevant plus largement de la gestion des données.

Pensée unique ?

En matière d'architectures hautement extensibles, il faut en premier lieu cesser de considérer la base de données comme solution unique à toutes formes de besoins.

Les SGBD sont des outils très complets et robustes, dont l'usage s'est généralisé au fil des années, à tel point que, pour beaucoup d'architectes, la question ne se pose même plus : les données doivent être gérées dans un SGBD relationnel. Et une fois ce postulat posé, on réfléchit au moyen de rendre cette gestion extensible.

De nombreux architectes ont critiqué cette approche dite du « *one size fits all* » (*une taille unique convient à tout le monde*), qu'on pourrait traduire en français branché par *pensée unique*.

Car rendre une base de données extensible est complexe, moyennement performant, et finit toujours par buter sur une limite.

Ainsi, les très grandes plateformes de l'Internet ont toutes renoncé non pas aux bases de données en général, mais au principe d'une base centralisée tournant sur un méga-cluster.

Modélisation objet et programmes

Par rapport aux années 90, un changement fondamental est intervenu dans le développement d'applications. L'application des années 90 construisait une requête SQL, qu'elle adressait à un SGBD. Ou bien dans certains cas, elle invoquait une procédure stockée du SGBD.

Au sein de l'application, la modélisation d'entités en forme d'objets métier s'est répandue. Dans un premier temps, cette approche était générée par la nécessité de convertir les interfaces relationnelles ensemblistes de la base de données vers les objets de l'application, et réciproquement. Rapidement sont apparus des frameworks qui ont pris en charge ce travail. Et la généralisation de ces couches ORM (*object-relational mapping*), a modifié en profondeur la relation entre une application et son SGBD. L'application ne prépare plus des requêtes SQL ensemblistes, l'application n'a plus une approche ensembliste de la gestion des données, elle est focalisée sur son paradigme objet.

Ainsi, les couches ORM telles que Hibernate, *ont beaucoup réduit le spectre de fonctionnalités SGBD effectivement utilisées*. A la fois parce que le paradigme objet n'a pas grand usage des possibilités ensemblistes, et également parce que ces couches visaient une totale indépendance par rapport à la base de données, et devaient donc se satisfaire du plus petit dénominateur commun aux différents SGBD.

C'est ainsi par exemple que les langages de procédures stockées, qui étaient un *must* dans les années 90 pour un SGBD sérieux, sont tombés en désuétude, à la fois par manque de standard, et par incompatibilité avec le développement objet.

Les propriétés ACID

Pour une gestion sûre et cohérente des données, en présence de multiples processus effectuant des mises à jour de manière concurrente, un système de gestion des données doit respecter les propriétés dites « ACID » :

- **Atomicité.** Dans une séquence d'opérations liées, une transaction, on doit avoir l'assurance que toutes les opérations ont été exécutées, ou qu'aucune n'a été exécutée.
- **Cohérence.** Les données sont toujours dans un état cohérent, il n'y a pas d'états transitoires incohérents qui soit visible.
- **Isolation.** Les autres processus ne voient que l'état avant et l'état après une transaction, ils sont isolés des états intermédiaires.
- **Durabilité.** Une fois la transaction terminée avec succès, elle est irréversible.

Bien entendu, ces propriétés doivent être vérifiées en toutes circonstances, c'est-à-dire :

- Quels que soient le nombre de processus concurrents et la chronologie de leurs transactions ;

- Y compris en cas de panne soudaine d'un composant

Les bonnes bases de données savent assurer les propriétés ACID. Cela inclut MySql, avec le moteur InnoDB.

Pour bien fixer les idées, il est utile de considérer un exemple typique de transaction. Il s'agit d'enregistrer un virement d'un montant M d'un compte S vers un compte D, qui implique la suite d'opérations suivante :

- Vérifier que le solde du compte S est supérieur ou égal à M.
- Soustraire M du solde du compte S.
- Ajouter une ligne dans la table des mouvements associés à S.
- Ajouter M au solde du compte D.
- Ajouter une ligne dans la table des mouvements associés à D.

On voit aisément les problèmes qui surviendraient en l'absence de propriétés ACID. Par exemple si la transaction est interrompue après avoir soustrait et avant d'avoir additionné...

La mauvaise nouvelle, c'est qu'il est très difficile de réaliser une gestion de données assurant les propriétés ACID et qui soit par ailleurs extensible sans limite.

Nous verrons qu'il faut en général accepter quelques compromis, et bien mesurer le risque d'une gestion non-ACID. Bien sûr, s'il s'agit d'argent ou de sécurité, le compromis n'est pas possible. Mais s'il s'agit des commentaires ajoutés à un blog, on peut se passer de ces propriétés, et accepter des états transitoires incohérents, ou d'une manière plus large, négliger les cas exceptionnels, y compris même la perte de quelques commentaires dans le cas d'une panne disque, si elle est rare.

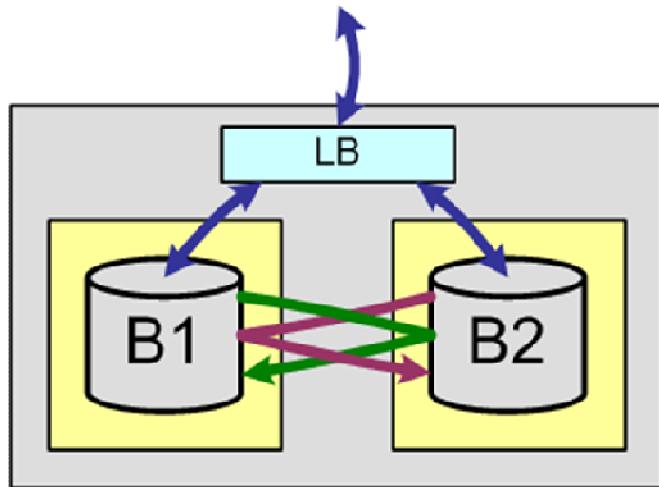
Pour autant, on ne peut y renoncer à la légère. *Une base de forte volumétrie qui se retrouve dans un état incohérent peut être un problème très difficile à résoudre.*

Le cluster

Le cluster de base de données est la seule solution qui garantisse la cohérence des données et les propriétés ACID entre plusieurs serveurs. Cela implique des échanges relativement complexes entre les serveurs, c'est ce qu'on appelle le commit à deux phases, ou *two-phase commit* (2PC) : avant de valider (« commiter ») une transaction, chaque serveur doit vérifier que tous les autres pourront la valider également. Entre cette première vérification et la confirmation de validation, les autres

serveurs ne doivent évidemment rien faire qui puisse interdire la transaction.

Ce que l'on peut représenter schématiquement comme suit :



Ces échanges ont un impact sur les performances, de sorte que le mécanisme n'est pas très extensible.

Et en cas d'arrêt brutal au milieu de ces échanges, il faut encore des échanges complexes pour « démêler » les serveurs, débloquer les verrous et s'assurer de la cohérence.

Le cluster de base de données forme un tout, incluant son load-balancing, de sorte qu'il est vu du reste de la plateforme comme un serveur unique. Les deux bases sont à tout instant identiques et une transaction n'est validée sur l'une que si elle peut l'être également sur l'autre. En cas de panne de l'un des serveurs, aucune donnée n'est perdue, le second traite immédiatement toutes les requêtes.

Pour une vraie haute-disponibilité, chaque serveur doit donc avoir une capacité suffisante pour traiter toutes les requêtes. Avec deux serveurs, on ne peut donc pas avoir simultanément haute-disponibilité et extensibilité. C'est à partir de trois serveurs que l'on pourrait avoir à la fois une capacité nominale double et la tolérance à une simple panne. Mais on met assez rarement en œuvre un cluster au-delà de deux serveurs, et donc plus souvent dans une finalité de haute disponibilité que de haute capacité.

En conclusion, il faut retenir que le cluster n'est pas le nec-plus-ultra de la gestion de données dans une plateforme web hautes-performances.

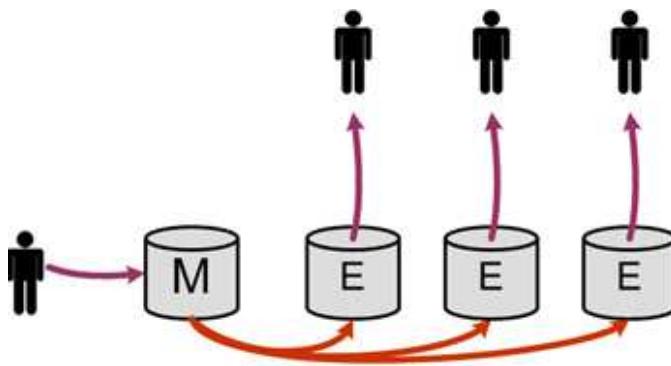
Lecture seule, extensibilité

Beaucoup de plateformes web sont, sinon en lecture seule, du moins en lecture majoritaire.

Or en lecture, l'extensibilité est facile à obtenir :

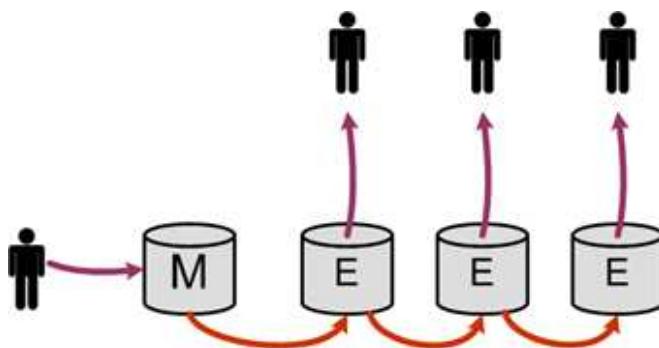
- On copie les données à l'identique sur un grand nombre de serveurs équivalents
- On lit les données sur n'importe lequel de ces serveurs.

C'est un principe de *réparation*, que l'on peut représenter comme ceci :



Où M est une base Maître, et E sont des bases Esclaves. Toutes les bases esclaves sont ici en lecture seule.

On peut également mettre en œuvre cette réparation en cascade, de la manière suivante :



Lorsqu'une modification est opérée sur la base maître, elle doit être propagée sur les bases esclaves. Nous verrons plus loin les outils de cette propagation.

La réparation est extensible, simplement et sans limite, mais elle implique un délai, et donc des états transitoires incohérents, aussi bien entre la base maître et une base esclave, mais aussi entre deux bases esclaves.

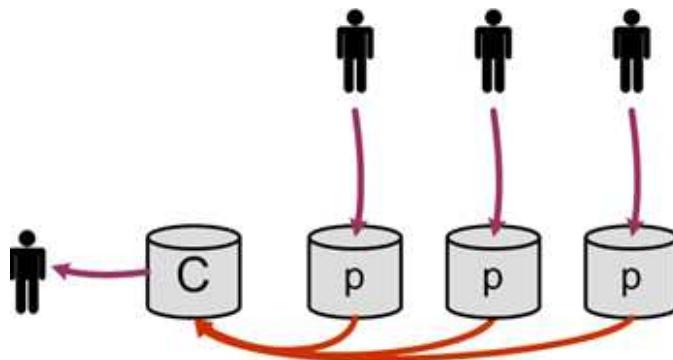
Ainsi, si un même utilisateur est amené à lire des données depuis une base E₁ d'abord, puis depuis une base E₂, il est possible qu'il lise des choses différentes, du moins en présence d'écritures exécutées sur M, et en cours de réPLICATION.

En présence d'un flux d'écriture réduit, ces états transitoires peuvent souvent être négligés, mais ils doivent malgré tout être analysés, afin de bien en mesurer les conséquences possibles au plan fonctionnel.

Ecriture seule, extensibilité

Le cas de l'écriture seule est beaucoup plus rare sur des plateformes web, mais peut arriver. C'est typiquement le cas de l'écriture d'un fichier de log : chaque serveur écrit sa propre log, qu'il ne lit jamais. Les fichiers de log sont acheminés de manière asynchrone sur un serveur où ils sont consolidés en une log unique.

Ce que l'on peut représenter comme ceci :



Le partitionnement des données

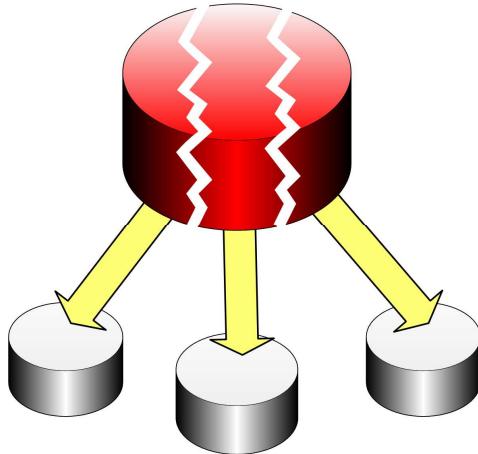
Principe du partitionnement

Un autre cas de figure qui permet une gestion extensible des données est le partitionnement.

Partitionner, c'est répartir les données sur N serveurs, de sorte que :

- Chaque entité partitionnée est gérée sur un serveur et un seul
- Elle emmène avec elle différentes entités liées, au regard du modèle des données
- Certaines entités de référence peuvent être répliquées sur les différents serveurs.

Ce qu'on peut représenter symboliquement comme ceci :



Bien entendu, en présence d'une gestion de données partitionnée, les applications doivent savoir facilement sur quel serveur chercher une entité. Elles pourraient adresser des requêtes en parallèle à tous les serveurs, mais dans ce cas la charge ne serait pas partagée, et l'on n'aurait obtenu qu'une extensibilité en termes de volumétrie, non d'accès.

Quelle logique de répartition ?

Dans une gestion de données partitionnée, il faut absolument éviter les règles de partitionnement fonctionnelles, telles que segmentation alphabétique (A-F sur le serveur S₁, G-N sur le serveur S₂, ...) ou encore chronologique.

Ce type de répartition manque beaucoup trop de flexibilité. En effet, comment réagencer ces gros volumes de données en cas de saturation d'un serveur, ou bien pour introduire un nouveau serveur ? On voit qu'il est difficile d'accompagner ainsi une montée en charge progressive.

La bonne pratique est donc plutôt d'adopter un partitionnement arbitraire, sans autre critère que le taux de remplissage. Une entité est gérée sur un serveur non pas parce qu'elle vérifie telle ou telle propriété, mais simplement parce qu'on en a décidé ainsi, parce qu'il y avait de la place.

Cela requiert bien sûr une table d'allocation, qui gère la correspondance (entité → serveur), et cette table doit être mise à disposition des applications, et tenue scrupuleusement à jour. Mais en échange de cela, on obtient une réelle flexibilité dans le remplissage, même si les opérations de réagencement général restent difficiles.

Requêtes transverses et datawarehouse

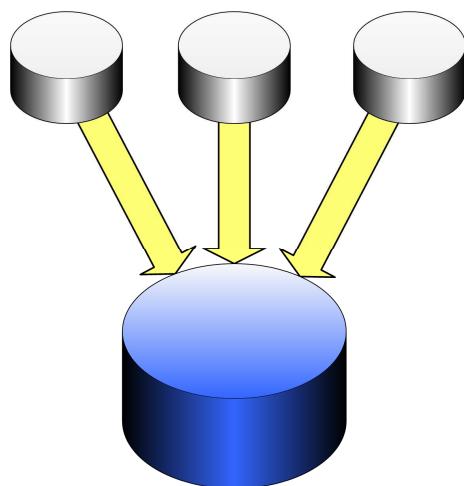
En présence d'une gestion de données partitionnée, il demeure presque toujours quelques besoins de requêtes transverses, des requêtes qui ne peuvent pas être traitées par un seul serveur. Par exemple, rechercher tous les internautes inscrits qui habitent en région parisienne.

Il y a deux voies pour obtenir cela :

- Soit des requêtes adressées en parallèle à tous les serveurs, et un traitement de consolidation des réponses ;
- Soit la création d'une base consolidée permanente, à la manière d'un datawarehouse.

C'est en général plutôt cette seconde voie que nous préconisons. En effet une fois le datawarehouse central mis en place, toutes sortes de requêtes, d'extractions ou de traitements statistiques de type décisionnel pourront y être opérés. Alors que dans le cas de la parallélisation / consolidation, il faut explicitement réaliser chaque type de traitement.

Le datawarehouse et le partitionnement sont donc complémentaires.



En voyant cette figure, on pourrait se dire : à quoi bon partitionner, pour ensuite consolider ? Mais en fait :

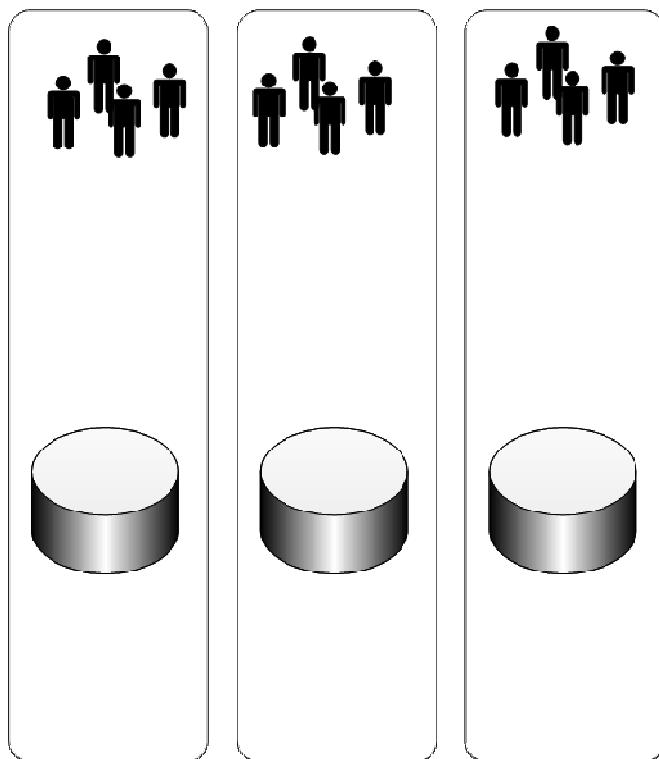
- Le datawarehouse ne consolide que l'information *utile en transverse*, le plus gros de l'information reste purement partitionné ;
- Le datawarehouse peut gérer de gros volumes, mais avec un taux d'accès faible, il est utilisé pour des requêtes occasionnelles, relevant le plus souvent de l'administration de la plateforme ;

- Le datawarehouse peut avoir une modélisation et des outils différents, plus adaptés à sa fonction et typologie d'accès.
- Enfin, puisqu'on a dit qu'il nous fallait une table d'allocation à jour, le datawarehouse est le lieu naturel d'élaboration de cette table.

Partitionnement par user

Dans une plateforme web, l'entité appropriée pour le partitionnement est généralement l'utilisateur, pour autant qu'il soit identifié. C'est bien souvent l'entité autour de laquelle gravitent les données : l'utilisateur est inscrit, ou « membre », et différentes données lui sont rattachées : son profil, ses préférences, ses commandes, etc.

Ce que l'on peut représenter comme suit :



Si l'entité de partitionnement est l'utilisateur, alors on peut envisager de répartir dès le niveau frontal :

- Un même internaute est alors géré sur un même frontal, non pas l'espace d'une session, mais toujours, pour toutes ses visites.
- Le frontal est en relation avec l'une des partitions de base, gérant les données de l'utilisateur.

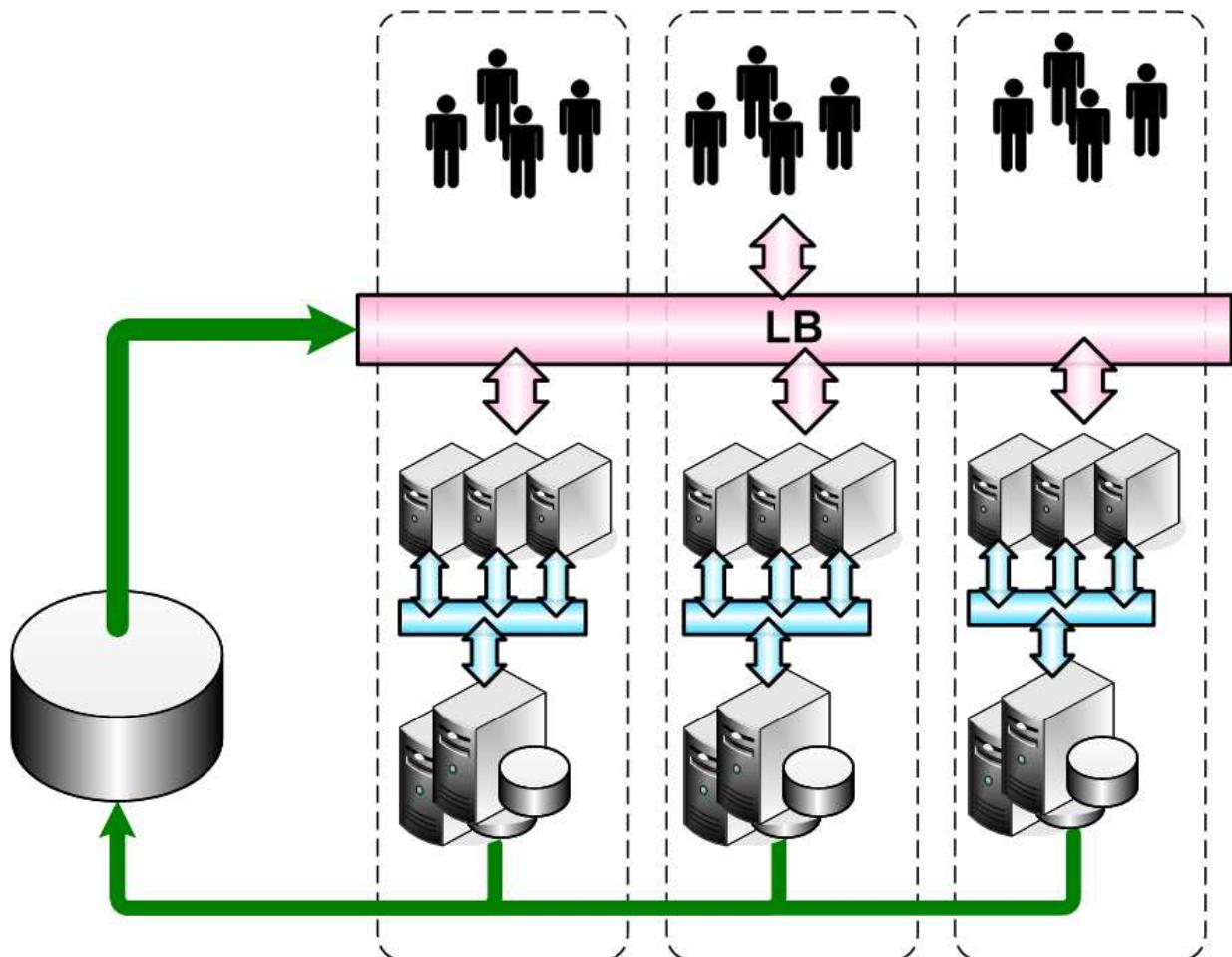
Architectures Hautes-Performances

- Dans ce cas, on gèrera le load-balancing sur la base de l'identification.

L'un des avantages est une certaine transparence dans l'accès aux données : *les applications n'ont pas à connaître le partitionnement*. C'est en particulier approprié lorsque l'*application est un progiciel*.

En revanche, ce type de partitionnement rend plus difficile la gestion du secours : chaque frontal, chaque partition de base, doit disposer de son propre secours, ou alors il faut mettre en œuvre un secours sur du matériel mutualisé, mais cela implique un processus d'activation du secours qui installe la bonne configuration et la bonne sauvegarde.

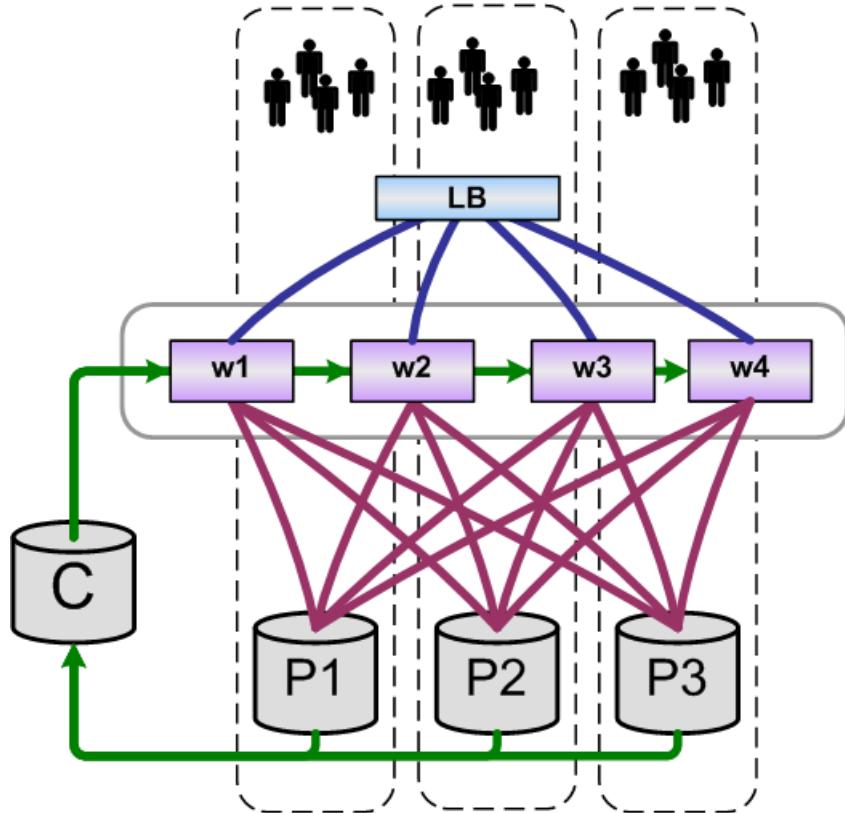
En constituant ainsi des plateformes indépendantes, assurant chacune sa haute disponibilité, on obtient une architecture de ce type :



On a fait figurer un lien entre la base consolidée, qui porte la table d'allocation, et le dispositif de load-balancing, pour signifier une répartition fonction de l'utilisateur. Dans ce cas de figure, chacun des frontaux ne voit qu'une seule base de données, la sienne, correspondant à la partition qui porte ses utilisateurs.

L'alternative est de gérer l'aiguillage vers la bonne partition en aval des frontaux web, au niveau de l'accès aux données. C'est plus naturel, plus flexible, et plus robuste, et le load-balancing en amont plus simple. *Mais le partitionnement n'est alors pas transparent, il doit être géré par l'application.*

On peut représenter cette configuration comme suit :



Ici, le load-balancer répartit la charge sans se préoccuper des utilisateurs, et pas même des sessions. C'est au niveau des frontaux qu'intervient la table d'allocation qui définit le partitionnement, et qui peut être répliquée ou cachée sur les frontaux. Et chaque frontal adresse ses requêtes aux différentes partitions selon le besoin. Le secours à l'étage frontal est simplifié puisque les frontaux sont tous équivalents. Il n'y a pas d'ailleurs de relation entre le nombre de frontaux et le nombre de partitions.

En général, comme toute forme de *spécialisation des ressources*, le partitionnement rend plus difficile la gestion du secours, puisque chaque partition doit bénéficier d'un secours.

En résumé, le partitionnement est souvent la voie la plus extensible dans la gestion des données. Il présente quelques contraintes, mais couplé à une consolidation au sein d'un datawarehouse, c'est une configuration qui répond à un grand nombre de besoins.

Synthèse

Nous avons passé en revue 4 variantes de gestion de données visant l'extensibilité :

- Gestion ACID centralisée ou en cluster, pour les données critiques.
- Réplication pour les données en lecture seule, ou en lecture majoritaire.
- Consolidation asynchrone pour les données en écriture seule.
- Partitionnement, pour les données partitionnables, associé le cas échéant à un datawarehouse.

Ces quatre voies peuvent évidemment être combinées, pour couvrir le besoin d'une plateforme web.

Base de données

La grosse base centrale

Le chapitre précédent donnait des axes de solution vers une extensibilité à peu près sans limite au plan théorique.

Dans la pratique toutefois, il faut garder à l'esprit que la base de données centrale, unique, *convient encore à une majorité des sites*.

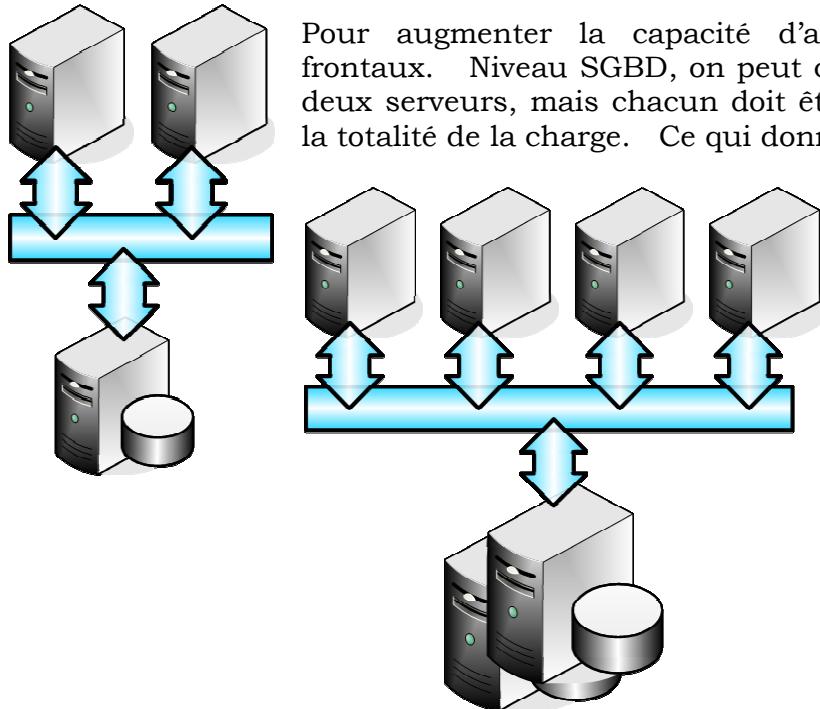
Elle est difficilement contournable lorsqu'on déploie un progiciel : les architectures extensibles sur la gestion des données sont rarement transparentes, et l'on ne peut pas prendre un ERP quelconque par exemple, et lui coller une gestion de données extensible.

En matière de base de données, avant d'être aux limites, on peut trouver de très importants facteurs de gain dans l'optimisation des requêtes, des index, des paramètres généraux ou du cache. Une fois la base optimisée, on peut encore trouver un gain important dans la configuration matérielle : disques rapides, processeurs, mémoire.

Au total, il faut retenir qu'une base de données bien conçue et bien « tunée », sur un serveur puissant, peut servir une très grosse plateforme web. A titre d'exemple, le site cadremploi.fr, qui reçoit 3,2 millions de visites par mois, sur des processus de consultation relativement complexes, s'appuie sur une base Oracle unique tournant sur un bi-processeur qui a quelques années déjà, une base qui est loin d'être surchargée. Avec des applications bien conçues pour la performance, une base centrale peut aller très loin.

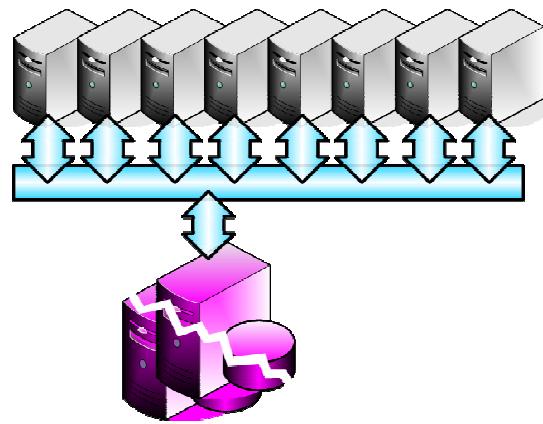
L'approche classique

L'approche traditionnelle est un modèle à deux étages : frontal / sgbd, qui se représente comme suit :



Pour augmenter la capacité d'accueil, on ajoute des frontaux. Niveau SGBD, on peut construire un cluster de deux serveurs, mais chacun doit être en mesure de traiter la totalité de la charge. Ce qui donne ceci :

Mais on ne peut pas ajouter indéfiniment des frontaux, il vient un stade où la base de données sature, elle est le point de contention :



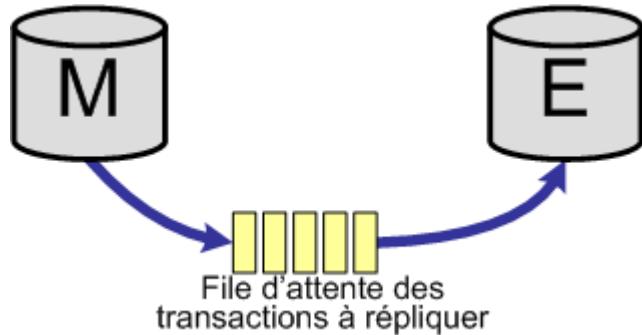
La réPLICATION SGBD simple

L'un des outils puissants à notre disposition pour étendre la capacité de la gestion de données est la réPLICATION.

Voyons d'abord le cas de la réPLICATION maître-esclave.

Architectures Hautes-Performances

Le principe est simple : toutes les écritures appliquées sur la base maître sont propagées et exécutées sur la ou les bases esclaves. Il peut y avoir un grand nombre de bases esclaves. La base maître est la seule qui accepte les écritures, les autres bases sont en lecture seule.



La réPLICATION maître-esclave est offerte par la plupart des SGBD, et facile à mettre en œuvre.

La réPLICATION est *transactionnelle*, c'est à dire que seules les transactions commitées sur la base maître sont répliquées sur la base esclave, et elles sont alors répliquées totalement.

La réPLICATION est *fiable* : aucune transaction ne peut être perdue, même en présence d'incidents, que ce soit sur le réseau ou bien sur l'une des bases esclaves. Si la base esclave est indisponible, par exemple arrêtée pour maintenance, alors les transactions s'accumulent en file d'attente, et seront jouées lorsque la base esclave redeviendra disponible. Cette fiabilité théorique est toutefois dépendante de la configuration matérielle dans la pratique : si la file d'attente est sur un disque non secouru qui vient à crasher, les transactions non transmises seront perdues.

La réPLICATION est *asynchrone* : la base esclave peut être en retard de quelques transactions par rapport à la base maître. Ce retard dépend de la configuration, et peut être réduit à quelques secondes. Mais au plan théorique, il faut compter sur la possibilité d'un décalage plus grand.

Enfin, le couple formé de la base maître et d'une base esclave en lecture ne respecte pas les propriétés ACID, comme vues plus haut.

Comme pour tout dispositif incrémental, fonctionnant par « deltas » c'est à dire ne portant que sur la propagation des changements, la fiabilité est absolument critique. Un taux même très faible de transactions perdues engendrerait une divergence entre les bases qui ne pourrait aller qu'en s'accentuant. Et une fois que les bases ont divergé, les resynchroniser peut être particulièrement délicat.

On peut distinguer deux modes d'utilisation de la réPLICATION :

- En mode secours seul. La base esclave est une copie, légèrement décalée, de la base maître. En cas de panne de la base maître, on passe l'esclave en maître.
- En mode partage de charge. Dans ce mode, la base esclave est utilisée en lecture. Les requêtes en lecture peuvent être réparties entre la base maître et la base esclave ou plusieurs bases esclaves. C'est un schéma que nous avons déjà évoqué.

Il faut garder à l'esprit que, dans tous les cas, l'activité en écriture sur la base esclave, ou bien les bases esclaves, est la même que sur la base maître. C'est donc une configuration qui n'apporte rien si la base maître est saturée en écriture.

RéPLICATION « manuelle »

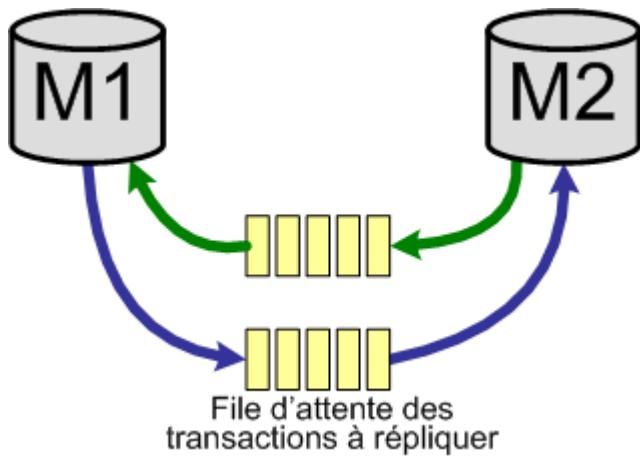
La réPLICATION suppose des modèles de données très proche, sinon identiques. Elle crée donc une forte dépendance entre les sous-systèmes concernés. Lorsqu'il s'agit de sous-systèmes homologues, comme dans le cas d'une répartition de charge, la dépendance ne pose pas de problème : quoi qu'il en soit on souhaite que les bases soient identiques.

Mais on peut aussi mettre en œuvre la réPLICATION entre sous-systèmes distincts, qui doivent échanger des données. Dans ce cas, la dépendance n'est pas bonne, et viole le principe d'encapsulation des données. C'est à dire que la réPLICATION est utilisée à la manière d'un middleware, ce qu'elle n'est pas. Il convient plutôt :

- Soit d'utiliser un vrai middleware de type MOM, asynchrone et fiable (cf. « MOM et Message Queues », page 31).
- Soit de mettre en place un dispositif de collecte applicatif, qui pourra passer par l'invocation d'un webservice de collecte.

RéPLICATION croisée, multi-maîtres

La réPLICATION multi-maîtres consiste « simplement » à croiser deux réPLICATIONS maître-esclave, comme on peut le représenter sur le schéma suivant :



Dans cette configuration, les deux bases sont en lecture-écriture. Elles sont identiques à un délai près, c'est à dire qu'en l'absence d'écriture, elles redeviennent identiques.

La réPLICATION multi-maîtres est d'une mise en œuvre délicate, non pas tant au plan technique, mais au plan fonctionnel :

- Des cas d'incohérences transitoires entre les deux bases doivent être analysés, et parfois traités par des règles de gestion spécifiques.
- Une même transaction étant exécutée sur une même base, chaque base a un comportement ACID, mais ce n'est pas le cas du couple de bases, ce n'est pas un cluster.

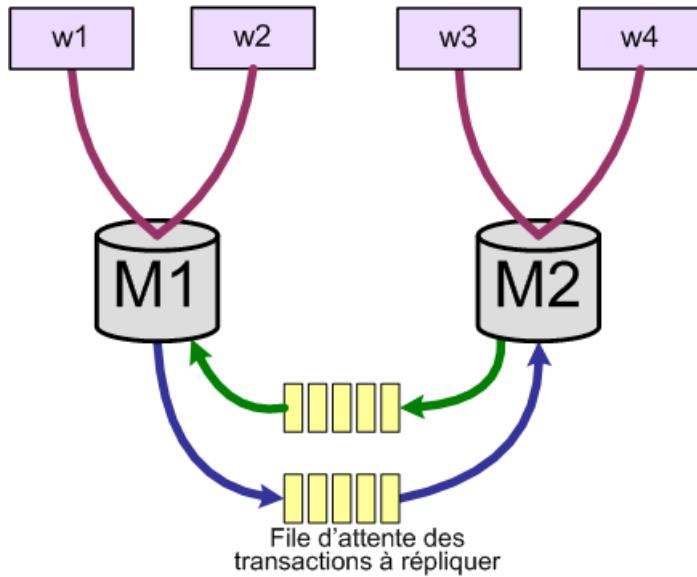
Par ailleurs, comme le cluster, c'est une configuration qui dans la pratique peut aller jusqu'à 3 serveurs, mais guère au delà.

Si les requêtes à la base de données sont réparties unitairement entre les bases, alors le risque est fort de souffrir d'incohérences. Un même internaute pourrait avoir demandé une modification de ses données, traitée sur l'un des serveurs, et immédiatement accéder en lecture sur l'autre serveur et ne pas voir ses changements. On voit que pour assurer qu'un même internaute a une vision cohérente et stable des données, il faut faire en sorte qu'il n'accède qu'à une seule des bases.

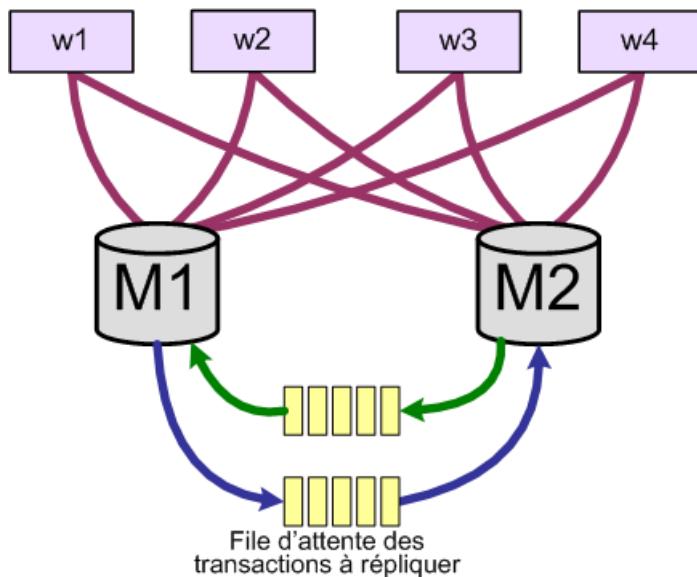
Pour cela, on peut envisager :

- Soit une répartition statique des frontaux vers les bases de données, associée à une répartition de charge avec affinité de serveurs, ce qui peut se représenter comme suit :

Architectures Hautes-Performances



- Soit une répartition sans affinité à l'étage frontal, mais une mémoire du serveur affecté à une session. Ce qui a le bénéfice d'un load-balancing plus flexible en amont, mais oblige malgré tout à une gestion de données de session, qui bien sûr ne pourra pas ici être en base de données, mais pourrait être conservée dans un cookie par exemple.



Notons qu'ici aussi, comme pour le cluster, si une des bases est indisponible, la capacité globale est divisée par deux

RAIDb et Sequoia

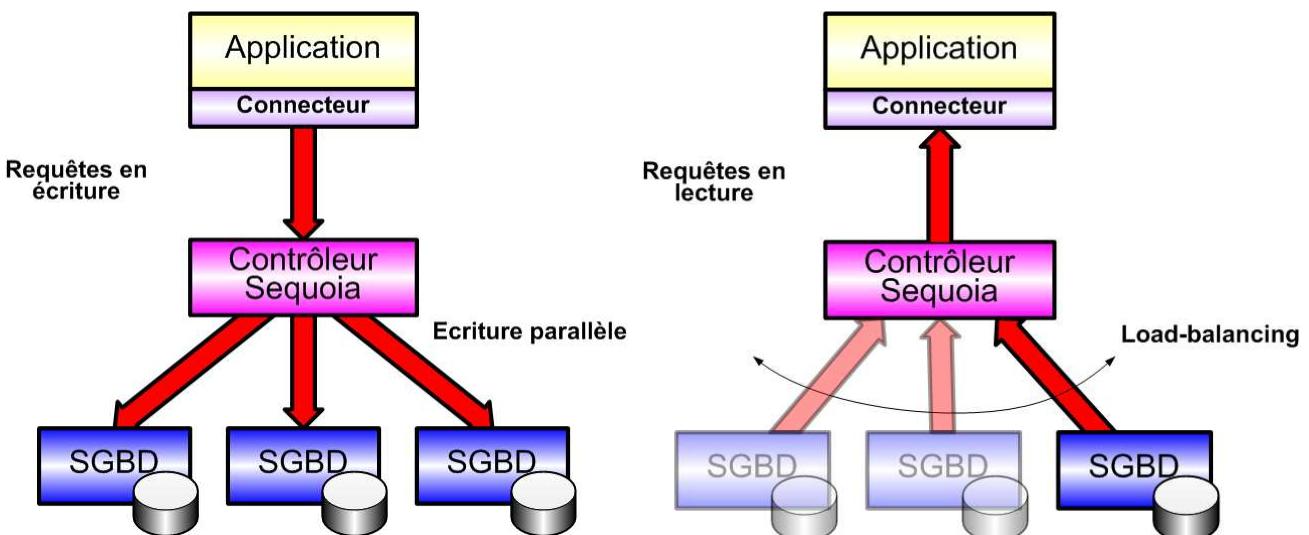
En matière d'extensibilité de la gestion des données, il faut citer aussi le concept relativement récent de RAIDb, « *Redundant Array of Inexpensive Databases* », inspiré évidemment du RAID pour les disques.

Le concept est issu des laboratoires de l'INRIA, exposé dans un article de 2003, et implémenté d'abord sous la forme d'un middleware initialement nommé C-JDBC, repris ensuite par Continuent et renommé Sequoia.

Comme pour le RAID disque, le principe de Sequoia est l'utilisation de multiples bases de données, de dimension et de niveau de fonctionnalités ordinaires, pour construire un cluster de bases de données, visant à la fois la haute disponibilité de l'ensemble et l'extensibilité en capacité. Sequoia implémente une « *base de données virtuelle* », appuyée sur des bases de données effectives, qui peuvent être hétérogènes, appelées ici « *backend* ».

Pour les applications, l'utilisation d'un cluster Sequoia au lieu d'une base de données ordinaire est totalement transparente. Les applications adressent la base de données par le middleware standard JDBC, pour les applications Java, mais des APIs sont maintenant disponibles également pour les applications PHP ou d'autres environnements.

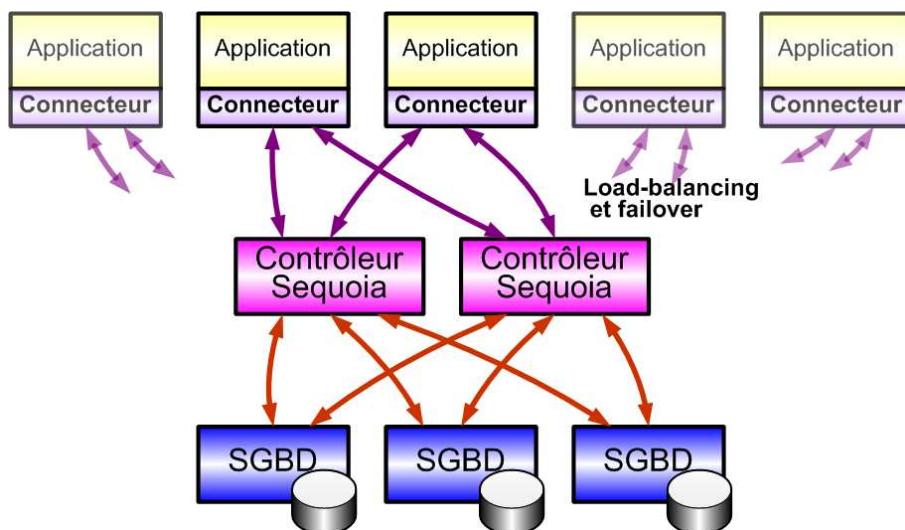
Il y a plusieurs configurations possibles de Sequoia, mais la plus utile est celle qui gère N bases de données identiques, entre lesquelles les requêtes sont distribuées. Les requêtes d'écriture sont adressées en parallèle à toutes les bases, tandis que les requêtes de lecture sont load-balancées. On n'obtient donc un bénéfice réel que s'il y a relativement moins d'écritures que de lectures, ce qui est courant.



Architectures Hautes-Performances

Cette configuration apporte des bénéfices semblables à ceux de la réPLICATION, mais avec une différence essentielle : les écritures sont exécutées de manière synchrone sur toutes les bases, de sorte qu'il n'y a plus de problème lié aux incohérences transitoires.

Sequoia traite également de la tolérance aux pannes, puisque les bases de données indisponibles sont automatiquement sorties de la répartition. Bien entendu, le composant middleware lui-même ne doit pas devenir « SPOF », point de fragilité. C'est pourquoi la configuration type met en œuvre deux ou plusieurs contrôleurs Sequoia, et les connecteurs (du côté des applications clientes), gèrent une répartition de charge (random, round-robin, séquentielle), entre les contrôleurs, sachant eux-mêmes éliminer un contrôleur défaillant.



Une même connexion applicative est stable, sur le même contrôleur, mais le contrôleur répartit ensuite les requêtes entre les différentes bases. Il utilise lui-même différents algorithmes : *least-pending-requests* (la base qui a le moins de requêtes en attente), *round-robin* (permutation circulaire), *weighed-round-robin* (idem avec pondération selon la capacité).

Une fonctionnalité importante est la gestion d'une log des transactions au niveau du contrôleur Sequoia lui-même, indépendamment des bases backend. Ainsi Sequoia permet également de désactiver une base de données backend pour une opération de maintenance. Dans ce cas, le contrôleur définit un point de contrôle dans sa log, de manière à pouvoir rejouer les transactions manquantes lorsque la base sera réinsérée.

En conclusion, Sequoia et le concept du RAIDb est une voie particulièrement intéressante vers l'extensibilité de la gestion des données, respectant le principe que nous avons posé en introduction, de faire appel à des composants ordinaires et peu coûteux. Son

caractère transparent pour l'application le rend en particulier compatible avec des progiciels standards.

On a dit plus haut à quel point la gestion des données était difficilement extensible. A cet égard, Sequoia est l'une des voies les plus intéressantes. Sequoia est une solution open source, qui a déjà suffisamment de références opérationnelles pour avoir démontré sa stabilité.

Le moteur d'indexation-recherche

Il existe des moteurs d'indexation-recherche uniquement consacrés aux pages web ; ils ne nous intéressent pas beaucoup en termes d'architecture.

Il existe aussi des moteurs d'indexation-recherche plus génériques, qui peuvent indexer n'importe quoi. Le plus célèbre, et sans doutes le plus puissant, est le moteur Lucene, de la fondation Apache, qui a été adopté par pratiquement tous les produits de gestion de contenus et de gestion de documents, qu'ils soient open source ou non.

Ce qui nous intéresse ici, c'est l'utilisation d'un moteur tel que Lucene dans la gestion des données d'une plateforme web. Un tel moteur a de nombreux atouts :

- Il est plus performant qu'un SGBD sur certaines typologies de requêtes complexes ;
- En particulier, il excelle dans des requêtes qui réunissent contenus structurés et contenus non-structurés. Par « contenus non-structurés », on entend les textes et documents.
- Il supporte de très gros volumes sans dégradation des performances. Typiquement plusieurs dizaines de millions d'items sont monnaie courante.
- Sa fonction n'est pas de stocker, ni de gérer l'information, il donne juste un moyen de la retrouver par la recherche.

Intégrer un moteur de recherche en complément du SGBD est dans la logique que nous citions plus haut, de ne pas s'appuyer systématiquement sur la même petite panoplie d'outils, mais d'utiliser au contraire le meilleur outil pour chaque fonction.

Sur plusieurs grands sites d'annonces, nous avons ainsi intégré des moteurs de recherche tels que Lucene. La base de données reste le

lieu de référence de gestion de l'information, mais les objets à rechercher sont passés au moteur pour indexation. Ils peuvent être par exemple exportés au format Xml, et analysés dans cette forme par l'indexation. Ou bien les APIs d'indexation peuvent être appelées directement par un traitement batch lisant dans la base. Mais on préfèrera nettement considérer la base de données et la fonction d'indexation-recherche comme deux sous-systèmes disjoints, qui doivent interagir uniquement au travers du middleware ou bien par des APIs bien définies, et non un traitement batch « à cheval » sur les deux.

Gestion de fichiers

Une problématique différente

S'il y a des similitudes entre le partage de données gérées en base, et le partage de fichiers, il faut garder à l'esprit aussi les différences essentielles :

- Un serveur de base de données est à la fois une unité de stockage et une unité de traitement. Pour répondre à une requête, il doit effectuer des traitements, qui peuvent être complexes et longs.
- Au contraire, un serveur de fichiers n'a que des traitements très simples à effectuer pour savoir quels blocs il doit accéder sur quels disques. Il a donc une plus grande capacité à servir des requêtes, et il est plus souvent limité soit par le débit de ses disques, soit par le débit de son interface réseau.

Il est assez aisément d'augmenter les capacités d'un serveur de fichiers, en premier lieu avec des disques plus rapides, puis avec une configuration RAID avec stripping et des interfaces réseau Gigabit.

Malgré tout, comme tout composant central, le serveur de fichiers finira par devenir facteur limitant de l'architecture.

Etudions donc les différents moyens de gérer des fichiers partagés entre différents serveurs. On va d'ailleurs retrouver finalement les différentes voies étudiées pour les bases de données, en particulier la réPLICATION, le partage et le partitionnement.

Des fichiers en base

Les bases de données acceptent des objets binaires (BLOB) de grande taille, qui peuvent stocker des fichiers. Gérer des fichiers en base de données peut être une option, lorsque la volumétrie est faible.

Il y a quelques bénéfices à en attendre :

- L'homogénéité des accès : données structurées et fichiers sont gérés de la même manière.
- La cohérence, et le bénéfice des propriétés ACID des bases de données. Typiquement, aussitôt qu'une application manipule à la fois une référence à un fichier, dans la base de données, et le fichier proprement dit, il y a des risques d'incohérences entre l'un et l'autre.
- L'homogénéité et la cohérence des sauvegardes : en une seule opération, on sauvegarde l'état courant de la base et des fichiers associés.

C'est par exemple la solution retenue par différents outils CMS ou GED pour adopter un fonctionnement dit en cluster.

L'un des inconvénients est que le fonctionnement n'est pas transparent pour les applications, qui n'accèdent pas aux fichiers en base par les APIs ordinaires, et ne peuvent pas faire de l'accès direct.

Par ailleurs, dès que la volumétrie grandit, c'est un mauvais choix :

- La base de données devient énorme et les opérations d'exploitation deviennent difficiles.
- Les performances sont inférieures à celles d'un *file system*.
- Il y a une certaine opacité du dispositif pour les exploitants

La réPLICATION

Comme pour les bases de données, lorsque des fichiers sont très majoritairement en lecture, le plus simple et le plus efficace est de les répliquer vers les différents serveurs qui en ont usage.

L'important, comme toujours, est de bien identifier *quelle est la version maître des fichiers*, celle qui est à jour à un instant donné. C'est à partir de cette version maître que l'on organisera la réPLICATION.

La réPLICATION est une solution fiable, extensible sans limite, tolérante aux pannes et simple à déployer.

L'outil magique de la réPLICATION de fichiers s'appelle Rsync. Il existe depuis une douzaine d'années, et s'est bonifié avec le temps, atteignant aujourd'hui une robustesse à toute épreuve.

Rsync ne réalise pas une copie massive des fichiers, de la source vers la destination. Au contraire, il commence par analyser les différences entre destination et source, afin de réduire les échanges au strict minimum requis. Non seulement les fichiers inchangés ne sont pas

envoyés, mais Rsync peut aussi découper les fichiers en morceaux, et ne renvoyer que les morceaux modifiés.

Du fait de cette économie de moyens, Rsync peut être lancé périodiquement, de manière assez fréquente, sans mémoire de son exécution précédente. Si rien n'a été modifié depuis la dernière synchronisation, Rsync ne transfère rien.

Notons que dans les environnements locaux, on utilise RSync en mode « -W » pour « *whole file* » (fichier entier) car le débit n'est pas contraint et cette technique utilise moins de CPU.

Gestion de contenus et réPLICATION

Considérons le cas d'un site web entièrement statique, constitué d'une arborescence de fichiers Html et autres composants. Si ce site doit supporter une charge très importante, le moyen le plus efficace pour en assurer l'extensibilité, est de simplement répliquer l'arborescence de fichiers vers N serveurs, à partir d'un serveur maître.

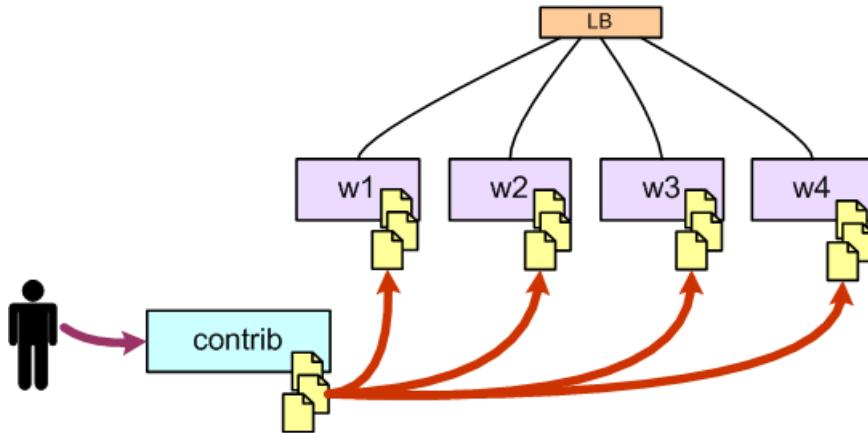
Lorsque des modifications sont requises, elles sont opérées sur le serveur maître. Lorsque le RSync est lancé par l'ordonnanceur, il détecte les changements et synchronise chacun des serveurs. Cette synchronisation peut intervenir toutes les 15 minutes, de sorte que les modifications sont rapidement visibles.

Certes, durant la synchronisation, différentes petites incohérences transitoires peuvent intervenir. Les différents serveurs peuvent ne pas être exactement au même niveau, de sorte que si on a un load-balancing sans sessions, un même internaute pourrait apercevoir des pages différentes sur une même URL. Et sur un même serveur, une page de sommaire peut avoir été mise à jour avant une page référencée, de sorte que l'on courrait le risque d'un lien cassé. Ce sont de vrais problèmes, mais ces incohérences sont extrêmement transitoires, de sorte que la probabilité d'anomalie est faible, et les conséquences non critiques. En général, il suffira à l'internaute de rafraîchir la page pour voir disparaître l'anomalie.

Si l'on veut malgré tout éviter cela, alors il suffit de mettre en place un script un peu plus sophistiqué, qui prépare la nouvelle arborescence à côté de la précédente, puis bascule le serveur une fois synchronisé.

Cette forme de réPLICATION est représenté sur le schéma suivant :

Architectures Hautes-Performances



A gauche on distingue un « contributeur », celui qui est susceptible de modifier les fichiers, par l’intermédiaire d’une application. Les fichiers sont répliqués sur N frontaux, ici W₁, W₂, W₃, W₄.

On a là une architecture réellement, totalement, extensible. En fait non seulement on pourra aisément ajouter autant de frontaux que l’on souhaite, mais de plus la capacité de chacun, servant des fichiers statiques, est extrêmement élevée, typiquement de plusieurs milliers de pages par seconde. Et ces frontaux peuvent aussi être répartis dans différents *datacenters*.

Il faut souligner aussi que ce n’est pas parce que l’on parle de pages statiques, de fichiers, que l’on ne peut pas bénéficier d’outils de gestion de contenus. Il est possible en effet de mettre en place un CMS sur le serveur de contribution, et donc de générer les pages à base de gabarits comme sait le faire un CMS. On exportera ensuite les pages produites dynamiquement par le CMS, sous la forme de fichiers statiques. Cet export peut être plus ou moins facile à réaliser selon les outils, mais il est généralement possible.

Quels sont les inconvénients de cette architecture sublime ?

- Un peu de latence dans la mise en ligne de modifications, mais elle peut être réduite à quelques minutes ;
- Un site qui est strictement en lecture seule, ce qui va à l’encontre des tendances actuelles web 2.0, où les aspects participatifs et de contributions communautaires deviennent prépondérants, sans parler des aspects véritablement applicatifs.

La réPLICATION de contenus, de pages Html ou autres composants, sur des serveurs n'est toutefois qu'un des cas de figure de la réPLICATION de fichiers.

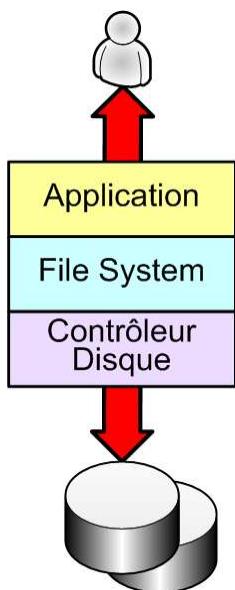
On peut avoir à réPLIQUER de simples fichiers de configuration, ou encore des programmes, ou des ressources utilisées par les applications.

Les outils de gestion de contenus manipulent également un grand nombre de fichiers : images, sons, flash, css, javascript, etc, mais aussi fichiers à usage interne : configuration ou templates, ou parfois fichiers de cache. Selon les outils, et selon les choix de configuration, ces fichiers peuvent être gérés en base de données ou sur le *file system*.

SAN

La technologie SAN (*Storage Area Network*) permet le partage d'un système de disques – mais non pas de fichiers – entre différents serveurs.

Pour bien comprendre son positionnement, il faut rappeler les trois couches intervenant dans l'accès aux données :

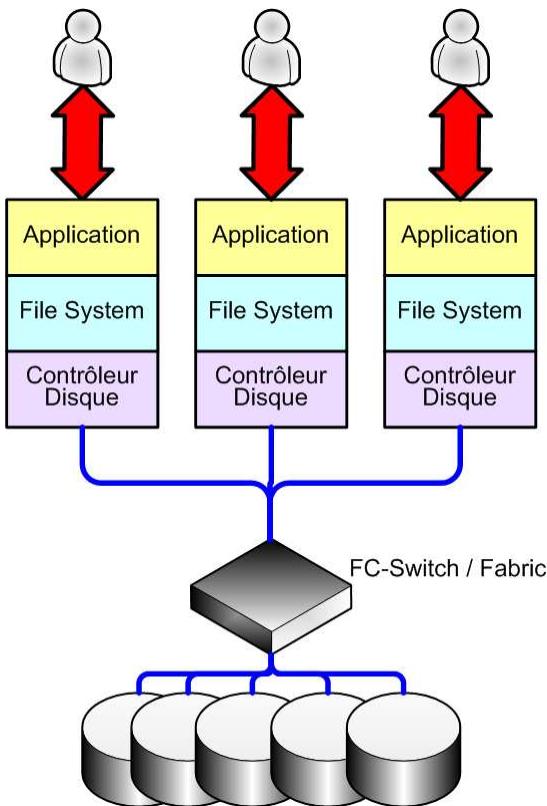


L'application, qui est le « client ».

Le file system, système de gestion de fichiers, qui reçoit des requêtes des applications, les traite et leur répond. Le file system tient à jour une table d'allocation des fichiers, qui définit de quels blocs sur quels disques chaque fichier est constitué.

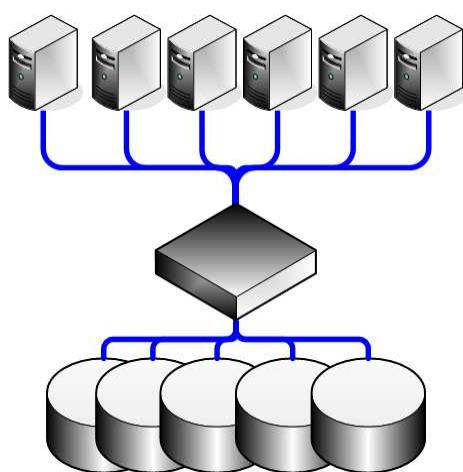
Le contrôleur disque, qui reçoit des requêtes du file system, les traite en accédant aux disques et répond. Le contrôleur disque ne connaît pas les notions d'utilisateur ni même de fichier. Il ne connaît que des disques et des blocs.

Architectures Hautes-Performances



Une configuration SAN intervient *en aval* du contrôleur disque et du file system, de la manière figurée ci-contre.

Les disques sont empilés dans une baie, raccordés le plus souvent par une liaison *fiber channel* à haut débit, sinon iSCSI, par l'intermédiaire d'un switch, qui permet de redéfinir à la demande la connexion des disques aux serveurs. Parce que les configurations disques sont gérées de manière globale et mutualisée, l'investissement est réparti, et l'on peut viser des configurations de très hautes performances, en termes de capacité, de débit et de tolérance aux pannes.



Le principe général est que tous les disques d'une plateforme, ou d'un datacenter, sont gérés de manière globalisée, ce que l'on peut représenter comme ci-contre.

Le SAN ne permet pas le partage de données.

En effet, on ne peut pas partager des disques en aval de la table d'allocation, c'est-à-dire du *file system*. Autrement dit, deux *file*

Architectures Hautes-Performances

systems, gérant chacun sa table d'allocation, ne peuvent pas partager un disque.

Le SAN n'a donc pas une finalité de *partage*, mais plutôt de *mutualisation de ressources*, visant :

- Une meilleure flexibilité dans l'allocation des ressource disque
- De très hauts débits IO obtenus par stripping, répartition des bits de chaque octet en parallèle sur N disques.
- Haute-sécurisation des données par RAID, répartition sur différents disques avec redondance permettant de tolérer une panne.
- Gestion centralisée des sauvegardes.
- Gestion rapide du secours, avec la possibilité de redémarrer un nouveau serveur sur la même unité logique. Ce qui est particulièrement efficace dans une architecture totalement virtualisée.
- Et finalement, allègement de la configuration des serveurs, qui peuvent être sans disque.

Dans les plateformes web, le SAN est d'un usage réduit. C'est plutôt un outil au service de grandes infrastructures diversifiées, telles qu'un datacenter dans sa globalité. Dans ce contexte, la mutualisation peut compenser le prix élevé.

Le SAN est parfois incontournable, dans des domaines d'application qui requièrent de très hauts débits, et de très gros volumes, typiquement le brassage de flux vidéo.

Mais en conclusion, le SAN n'est pas la clé de l'extensibilité.

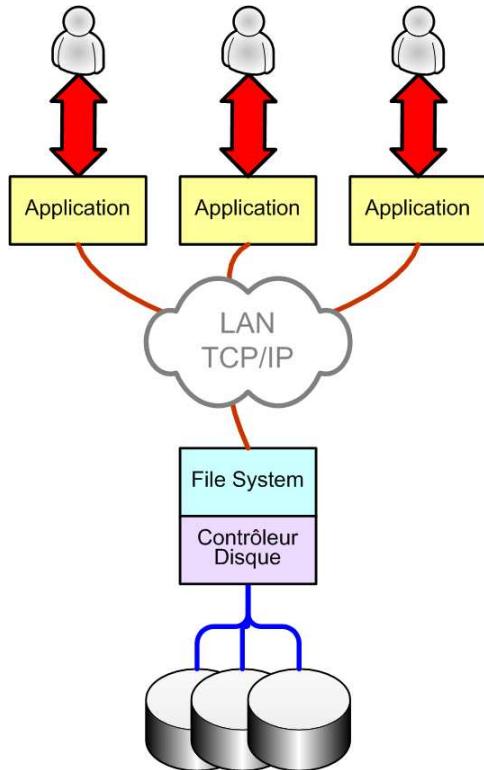
Architecture NAS

Un principe de partage

Au contraire du SAN, le NAS (*Network-Attached Storage*) est fait pour le partage de données entre différentes applications clientes.

Dans le cas du NAS, le *file system* est unique, et les questions de cache, synchronisation et cohérence sont donc traitées de manière centrale.

On le représente comme ceci :



Les applications accèdent au file system réseau sur du TCP/IP, qui peut donc traverser des routeurs et switchs ordinaires. Les unités de stockage réseau peuvent être montées dans le file system local des serveurs clients, et donc être accédées de manière transparente par les applications, comme des données locales.

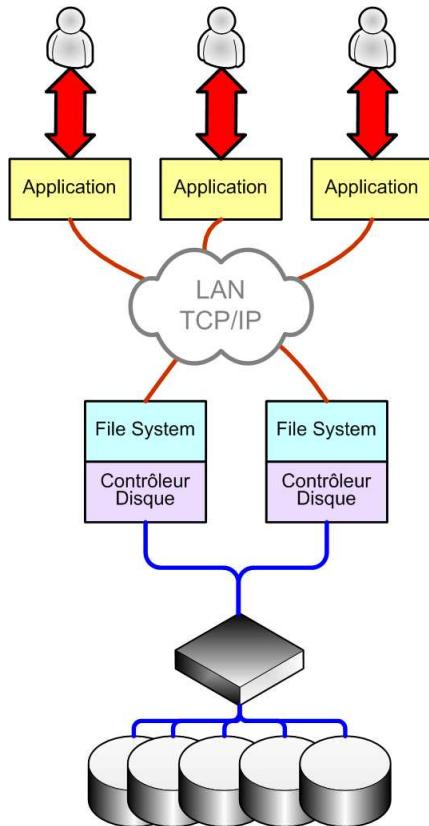
Bénéfices et limites

L'accès à un serveur de fichier de type NAS, typiquement NFS, est le moyen de partager des ressources fichiers entre différents serveurs, par exemple entre des frontaux web.

- Il est approprié pour des ressources relativement statiques, mais dans ce cas la réPLICATION peut être une alternative également.
- Il convient moins à des ressources partagées en écriture/lecture de manière intensive. Dans ces cas d'utilisation, les problèmes de verrouillage peuvent provoquer des ralentissements importants.
- Le débit IO est souvent limité par le réseau.
- Les requêtes d'IO disques des différentes applications sont bien sûr additionnées sur le serveur NAS. L'extensibilité atteindra donc une limite.
- Enfin, on voit que le serveur NAS est un point de fragilité (« SPOF ») de l'architecture, et nécessite donc un dispositif de failover.

NAS et SAN combinés

Servant, comme on l'a vu, des finalités très différentes, NAS et SAN peuvent tout à fait se compléter, avec une architecture de ce type :



NFS

En matière de serveur de fichier réseau, une solution domine : le *Network File System* de SUN, NFS.

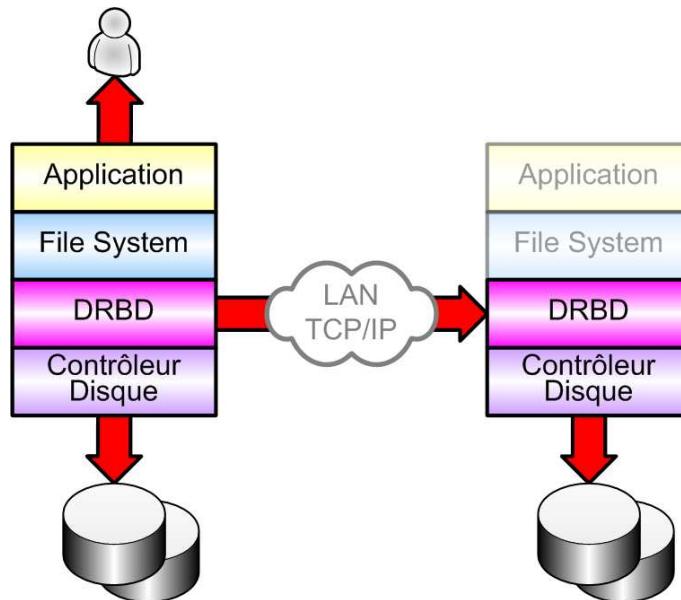
NFS est un protocole d'accès aux fichiers sur le réseau introduit par SUN au début des années 90, et devenu un standard de fait dans le monde Unix puis Linux. Il passe sur TCP/IP en 93, ce qui le rend compatible avec des réseaux de type WAN, mais malgré tout l'accès à un serveur de fichiers requiert des débits sensiblement plus élevés que l'accès à un serveur HTTP.

NFS est particulièrement approprié pour gérer le déploiement d'applications ou assurer l'unicité des données partagées. Mais il peut rapidement devenir point de contention de l'architecture en termes d'IO. Un partage NFS (ou CIFS dans le monde Windows) est en général peu approprié au-delà de 4 à 8 serveurs clients, selon le taux d'accès.

Un serveur NFS sera généralement doublé avec un spare, synchronisé par DRBD (voir plus loin), avec *heartbeat* et secours automatique

DRBD

DRBD est un outil de réPLICATION DISQUE qui s'insère entre le file system et le contrôleur disque, de la manière suivante :



Il assure une réPLICATION DISQUE transparente, vers un serveur de secours passif.

Il supporte deux modes de fonctionnement :

- Synchrone : chaque opération d'IO ne se termine qu'après double écriture
- Asynchrone : l'écriture répliquée est légèrement différée.

Le mode synchrone assure une parfaite cohérence des deux disques, même en cas d'arrêt brutal, tandis que le mode asynchrone assure de meilleures performances.

Il faut souligner que le *file system* de secours est totalement inutilisable, il ne peut pas être monté, même en lecture seule. C'est donc une configuration exclusivement réservée au secours.

L'accès concurrent aux fichiers

Les systèmes de gestion de fichiers réseau gèrent les accès concurrents et le verrouillage de fichiers, et même de blocs au sein d'un même fichier.

En fait, les applications modernes utilisent de moins en moins les fichiers, et encore moins en accès direct, ou en accès concurrents. Dans tous les cas, l'absence de transactions dans l'accès aux fichiers présente des risques d'incohérences. On ne peut pas englober 3 écritures sur 3 fichiers en une transaction insécable qui sera soit totalement exécutée, soit nullement exécutée.

L'utilisation typique est plus souvent du type : lecture globale du fichier, modification, écriture globale. C'est ce qui se passe lorsque le fichier représente un objet, sérialisé, ou bien stocké en Xml.

Lustre

Beaucoup croient que la problématique du *file system* est un sujet clos, une question réglée depuis 20 ans. On ouvre un fichier, on lit, on écrit, on ferme... rien de bien complexe là-dedans.

Mais il n'en est rien. Comme les SGBD, les gestionnaires de fichiers n'ont pas réglé tous leurs problèmes, et restent un domaine de recherche très actif. Parmi leurs problèmes figure, précisément, la question de l'extensibilité.

Lustre est un système de gestion de fichiers distribué, conçu pour une extensibilité maximale. Porté par une startup, il a été repris en 2007 par SUN, qui en a fait un produit stratégique dans son offre.

Lustre est un produit haut de gamme, open source, parfaitement robuste et « *production ready* », prêt pour une utilisation en production. Il est utilisé par plus de la moitié des 30 plus grands centres de calcul parallèle du monde. C'est d'ailleurs le domaine d'application favori.

Lustre peut gérer littéralement plusieurs peta-octets de fichiers, avec des débits globaux de plusieurs centaines de Gbps. Lustre traite en premier lieu d'extensibilité, la haute-disponibilité quant à elle, est traitée au niveau de chaque nœud de l'architecture.

Les principes généraux de Lustre sont les suivants :

- Une gestion centrale des métadonnées. Les métadonnées, c'est tout ce qui caractérise le fichier : son nom, son répertoire, ses droits, dates de modification, ... mais aussi sa répartition sur différents serveurs. Dans Lustre cette gestion des métadonnées est centralisée sur le MDS, MetaData Server.
- Chaque fichier est réparti sur un ou plusieurs *Object Storage Targets (OST)*, chacun composé de plusieurs *Object Storage Server (OSS)*.

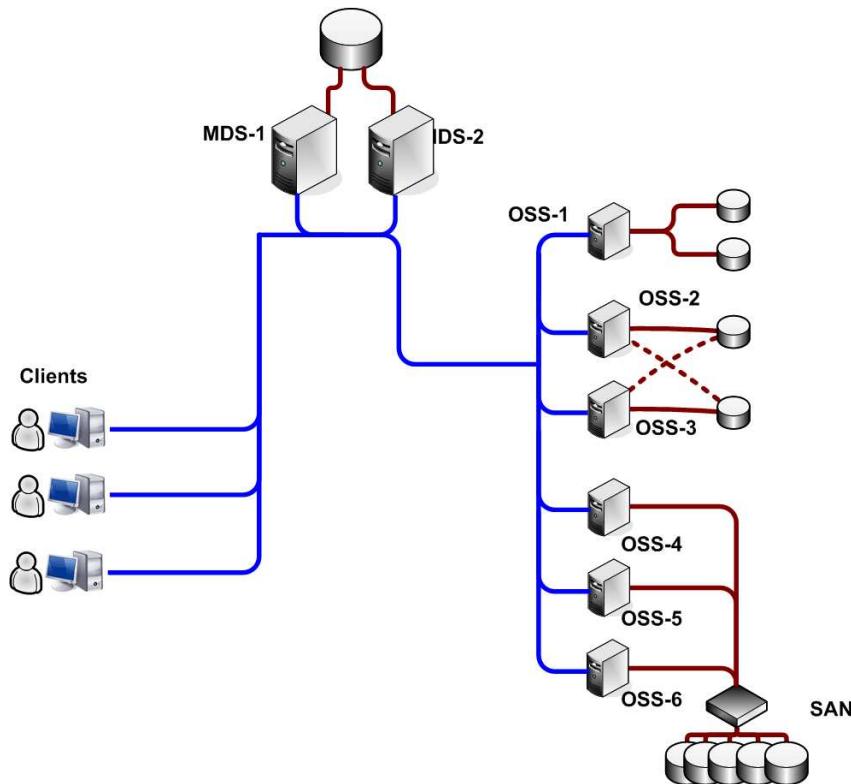
Architectures Hautes-Performances

- Une fois que le client (l'application accédant au fichier) a obtenu la table d'allocation du fichier sur le MDS, elle adresse des requêtes aux OST concernés.

Le caractère centralisé du MDS n'est pas un blocage pour l'extensibilité, puisqu'il y a un facteur 1000 ou plus entre les volumétries de fichiers et celles des métadonnées. Nous avons vu (cf « Le partitionnement des données », page 97) que d'une manière générale, le principe d'un partitionnement couplé à une forme d'annuaire global est une voie classique de la haute extensibilité.

Lustre met donc en œuvre un principe de partitionnement / consolidation, comme on l'a évoqué plus haut, mais ceci de manière totalement transparente pour les applications, qui ne voient que des APIs POSIX d'accès aux fichiers.

➔ Si vous avez de très fortes volumétries de données au sein de votre plateforme web, alors Lustre est une solution à considérer.



On voit sur le schéma ci-contre une variété de cas de figure, un serveur simple OSS-1, deux serveurs OSS-2 et 3, avec un DRBD croisé, leur permettant d'être en secours réciproque, et finalement un ensemble de serveurs partageant une baie SAN.

MogileFS

Dans la catégorie des systèmes de gestion de fichiers distribués et à tolérance de panne, il faut citer également MogileFS, qui a certes moins de références gigantesques, mais est d'une mise en œuvre plus simple, et constitue donc une bonne alternative pour une architecture web. MogileFS est un projet de Dange Interactive, les créateurs de

LiveJournal, qui ont également apporté le gestionnaire de cache distribué MemCached.

Hadoop HDFS

Parmi les principaux systèmes de gestion de fichier distribués il faut citer également HDFS, associé au projet Apache Hadoop, dont nous avons parlé plus haut.

LE CACHE

Principes du cache

Le cache est l'outil miracle en informatique, omniprésent au service des performances. On le rencontre à tous niveaux, tant dans le matériel que dans le logiciel, tant dans l'accès aux données que dans l'accès aux pages Html.

Lorsque l'accès à une ressource est trop lent, le principe du cache consiste à lui substituer un accès plus rapide, à une copie de cette ressource.

L'accès à la copie est plus rapide, mais ce n'est qu'une copie, et il convient de la mettre à jour si l'original est modifié. C'est là toute la problématique du cache.

Selon les cas, on peut tolérer un délai plus ou moins grand entre la modification de l'original, la donnée *de référence*, et la modification de la copie.

Le cache est avant tout un outil au service des performances. Il n'est pas particulièrement utile à l'extensibilité. Au contraire, même, il rend parfois l'extensibilité plus difficile. On pourrait ranger le cache dans la famille des *techniques d'optimisation*, indépendante des questions d'architecture. Mais ce n'est pas tout à fait le cas ; dans les faits, le cache est très lié à l'architecture, comme on le verra.

Actif / Passif, Pull / Push

On peut distinguer deux types de cache, en fonction de la manière de rafraîchir les données :

- Soit c'est le dispositif de cache qui *va chercher* une copie de l'original auprès de sa source (disque, réseau, serveur, ...)
- Soit c'est la source qui *pousse* une copie vers le dispositif de cache.

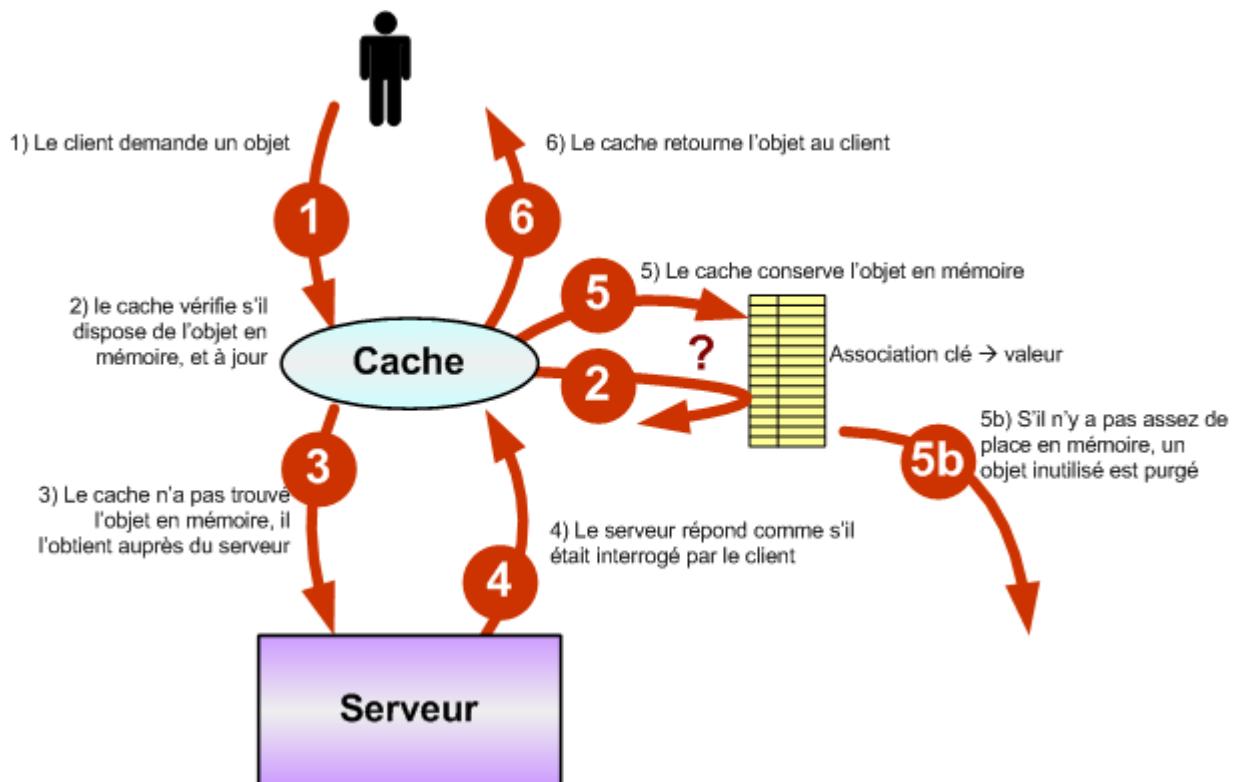
On appellera le premier mode *pull* (tirer), et le second *push* (pousser). Pour certains l'appellation cache évoque implicitement le mode *pull*, tandis qu'un mode *push* est assimilé plutôt à de la *réplication*. Mais il est intéressant d'étudier les deux conjointement.

Le cache en mode *pull* est parfois aussi appelé *passif*, tandis que le mode *push* peut être appelé *actif*.

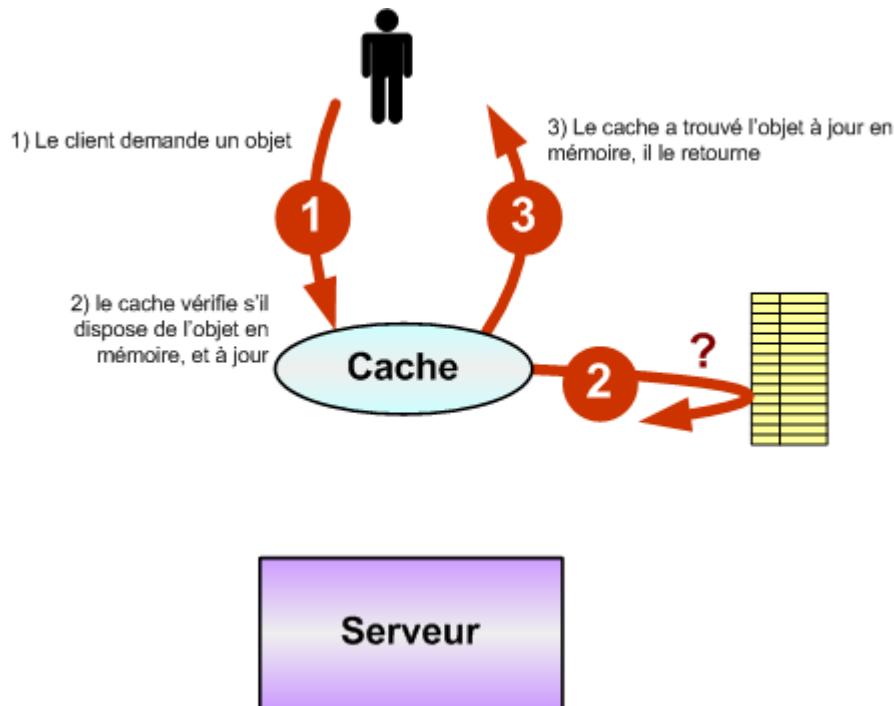
Cache en mode *pull*

Dans le mode *pull*, le cache *va chercher* les objets lorsqu'il en a besoin. Un « consommateur » demande une « ressource » au cache, le cache demande la ressource au « producteur », puis la conserve pour pouvoir la fournir à nouveau sans la redemander, et donc beaucoup plus rapidement.

C'est ce qui est représenté sur les schémas suivants, avec tout d'abord le cas où l'objet n'était pas trouvé dans le cache, et doit être obtenu auprès de son producteur, son serveur :



Et sur la figure suivante, le cas d'un appel suivant, où l'objet est trouvé en cache, et où il n'y a donc aucune sollicitation du serveur :



Durée de vie

Le mode *pull* est indissociable de la notion de *durée de vie*, ou *de validité*, en anglais « *time to live* » (TTL).

En effet, dans le schéma précédent, la ressource peut être modifiée à tout instant du côté du producteur. Dans le mode *pull*, le cache n'en est pas averti. En revanche, chaque ressource qui lui a été confiée est caractérisée par une *durée de validité*, durée pendant laquelle on peut faire l'hypothèse qu'elle n'a pas changé, ou plutôt que si elle a changé, on peut malgré tout utiliser la valeur précédente sans dommage.

Sur certains outils de cache, on configure aussi le « *time to idle* », le temps maxi écoulé depuis *la dernière utilisation* de l'objet.

On peut distinguer trois modes de gestion de la durée de vie des objets :

- Durée de vie illimitée, on changera le nom de l'objet lorsqu'il sera modifié
- Durée de vie unique : c'est un paramètre du cache, transparent pour le fournisseur

- Durée de vie différenciée, spécifiée par le fournisseur

Le fonctionnement « MRU »

Le gestionnaire de cache n'a pas toujours une mémoire suffisante pour tout conserver, et il doit alors trouver une stratégie pour garder en mémoire ce qui a le plus de chances d'être demandé à nouveau.

Un dispositif de cache fonctionne le plus souvent selon un principe de MRU, « *Most Recently Used* », c'est à dire que les contenus les plus récemment utilisés sont conservés en mémoire. Si la mémoire allouée ne suffit plus, alors on laisse tomber les contenus qui n'ont pas été utilisés depuis longtemps, les « *Least Recently Used* ». Le principe est bien sûr que les contenus les plus *souvent* utilisés seront toujours statistiquement dans les plus *récemment* utilisés, et resteront donc presque toujours en cache. Cette gestion MRU permet d'allouer une certaine quantité de mémoire à la gestion du cache, et d'obtenir le meilleur taux de succès, c'est à dire le pourcentage des pages qui peuvent être servies depuis le cache, que l'on appelle « *hit ratio* ». Bien entendu, plus la mémoire allouée augmente, plus le hit ratio augmente, ceci jusqu'à ce que la mémoire allouée au cache atteigne la somme totale de tous les contenus, et donc un hit ratio de 100%.

Certains dispositifs de cache introduisent une notion de « *Most Frequently Used* », potentiellement supérieure, mais bien plus difficile à gérer.

Le cache HTTP

En grande majorité, les sites servent les mêmes pages à tout le monde, et en grande majorité, ces pages sont construites par un CMS ou une application dédiée, au moyen d'un gabarit, dans lequel sont insérés les contenus extraits d'une base de données. L'élaboration des pages est un travail important, qui consomme beaucoup de ressources, et peut prendre jusqu'à 0,1 seconde, voire 1 seconde. Mais puisque la page a déjà été construite quelques secondes plus tôt pour un autre internaute, inutile de tout recommencer, il suffit de l'avoir conservée en mémoire, prête à l'emploi.

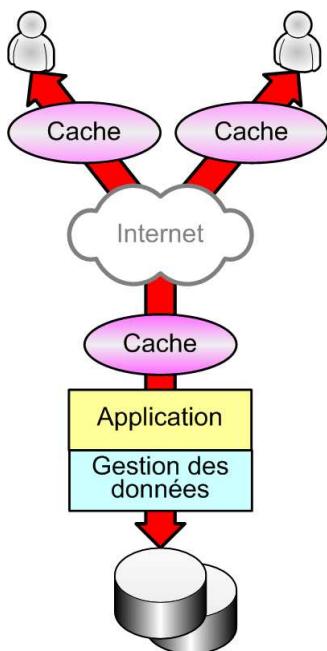
C'est donc là le principe du cache HTTP, qui peut multiplier par 100, voire 1000, la capacité d'accueil d'une plateforme Internet, et il existe depuis longtemps d'excellents outils open source pour mettre en place un cache, ceci quel que soit le CMS, ou bien l'application en général, qui construit les pages.

Le gestionnaire de cache est placé *devant le serveur HTTP*, c'est lui qui reçoit les requêtes des internautes. Si la page demandée est dans son cache, il la sert directement, sinon, il la demande au serveur, puis la

range dans son cache, pour un éventuel usage futur. Dans ce dispositif, chaque page peut comporter sa propre indication de durée de vie, durée pendant laquelle elle peut être servie à partir du cache, avant d'être rafraîchie.

Cache du navigateur

Nous l'avons évoqué déjà, les navigateurs web intègrent tous une gestion du cache sur les pages html et les composants qu'elles intègrent. Le protocole HTTP comporte différentes directives permettant de contrôler cette gestion du cache, et la bonne gestion de ces directives, souvent négligée, peut améliorer très sensiblement les performances d'un site. Cf. « Performances HTTP », page 40.



Le principe général est le même et repose également sur la bonne indication de durée de vie.

Pourquoi mettre en place un cache Html/HTTP côté serveur puisqu'on peut compter déjà sur un cache côté navigateurs ?

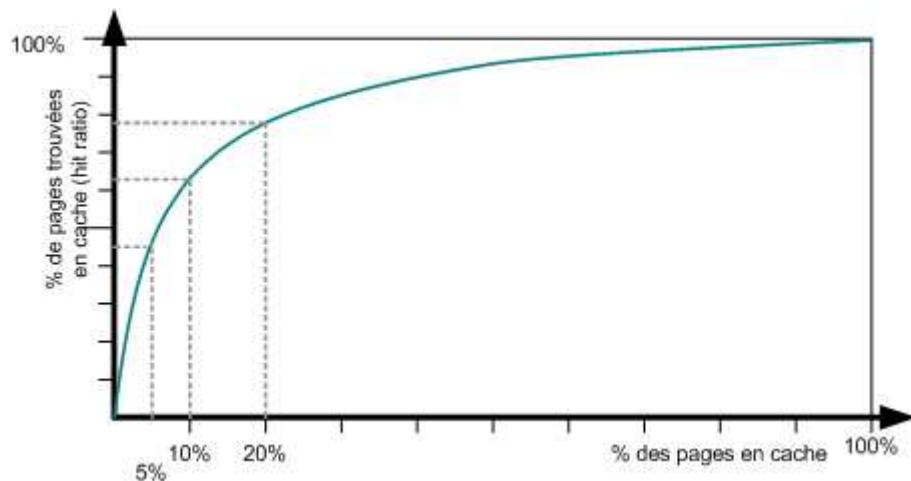
C'est très simple : le cache du navigateur ne peut servir que si un même internaute demande deux fois la même ressource. Un cache côté serveur est utile aussitôt que deux internautes différents demandent la même ressource. Et ce n'est pas tout : le cache côté serveur est parfaitement maîtrisable, configuré précisément pour nos besoins, tandis que le cache des navigateurs a une configuration que nous ne maîtrisons pas.

Comme représenté sur la figure ci-contre, le cache HTTP du côté serveur est partagé par tous les internautes, son efficacité potentielle est donc plus grande.

En revanche, le cache côté client soulage *toute la plateforme*, y compris sa bande passante et ses routeurs.

Un peu de mémoire suffit

Sur un site, il est courant que la *home page* représente jusqu'à 30% des accès, et une vingtaine de pages peuvent concentrer plus de 50% des accès, de sorte qu'un peu de mémoire suffit pour un cache efficace.



La figure précédente représente l'évolution typique du *hit-ratio*, le taux de page servies à partir du cache, en fonction du pourcentage de pages que peut contenir le cache. Où l'on voit qu'avec 10% des pages en cache, on peut déjà espérer plus de 50% de succès.

La mémoire ne coûte pas cher

En fait la mémoire ne coûte pas cher aujourd'hui, et il est souvent possible de disposer de la totalité des contenus en mémoire.

Considérons un site qui comporte 10 000 pages, chaque page ayant un poids de 100 ko, pour la partie Html du moins. Ce sont des chiffres assez représentatifs d'un site plutôt haut de gamme. Est-il possible de mettre tout cela en cache ? $10\,000 \times 100\text{ ko} = 1\text{ Go}$. Un giga-octet de mémoire, c'est très peu de chose aujourd'hui. Noter qu'on ne parle pas ici des images, qui ne seront guère plus volumineuses, et moins changeantes et sont donc particulièrement prédisposées à une bonne gestion de cache.

Lorsque tout peut tenir en mémoire, il n'est plus question de MRU, on garde tout. Ce qui limitera le hit ratio, malgré tout, c'est la durée de vie des contenus. Si elle est suffisamment longue, par exemple une journée, alors effectivement, chaque contenu ne sera produit qu'une seule fois par jour, puis servi uniquement depuis le cache. Mais si l'on considère que les contenus doivent être rafraîchis toutes les 10 minutes, alors il est possible que certains contenus ne soient jamais resservis depuis le cache.

En supposant la capacité du cache suffisante pour la totalité des contenus, on peut se livrer à un calcul simple, faisant intervenir deux paramètres : le nombre de pages servies par seconde et la durée de vie des pages.

Architectures Hautes-Performances

Soit D , la durée de vie des pages en cache ; P , le nombre de pages servies par seconde à l'heure de pointe ; N le nombre total de pages.

Faisons l'hypothèse typique que 80% des requêtes portent sur 20% des pages.

Pour que le cache commence à être utile, il faut que sur la durée de vie D , on ait servi plus de pages que $N \times 0,2$, les 20% de pages les plus utilisées.

C'est à dire :

$$P \times D \times 0,8 > N \times 0,2$$

et donc :

$$D > (N \times 0,2) / (P \times 0,8)$$

ou encore :

$$D > N / 4P.$$

Attention, on calcule ici le seuil à partir duquel le cache commencerait à être utile. Ce n'est pas par cette formule qu'il faut calibrer la durée de vie en cache des contenus. Plus la durée de vie est importante, plus le bénéfice du cache est grand. En fait le réglage est d'ordre fonctionnel et non technique : quelle est la durée de validité d'un contenu, quelle est la réactivité requise en cas de changement ? C'est affaire de compromis entre efficacité du cache et réactivité face aux changements.

Application numérique typique : $N = 10\,000$ pages, $P = 10$ pages par seconde : $D = 3$ minutes environ. 10 pages par seconde, c'est environ 40 000 visites par jour, c'est donc un site moyen en termes d'audience. Si l'on prend $P = 50$ pages par seconde, on est au niveau d'un site plus sérieux, à 200 kv/j, et le cache commence à être utile à partir de 0,5 minutes. Le calcul est simplifié, évidemment, car on pourrait s'intéresser aussi aux 10 pages qui représentent peut-être 20% du trafic total. Pour ces pages-ci, on trouverait $D > 10 / (P \times 0,2)$, c'est à dire que le cache serait utile encore plus tôt.

On en déduit plusieurs choses :

- Plus l'audience est forte plus le bénéfice du cache est important, c'est assez évident.
- Mais surtout: *pour un site à forte audience, même un cache de courte durée de vie commence à être utile*, c'est à dire que le bénéfice du cache n'est pas incompatible avec une haute fréquence de mise à jour.

Mise en œuvre d'un cache HTTP

Nous avons traité des différents aspects théorique du cache, voyons-en les aspects plus pratiques.

Le gestionnaire de cache (en mode *pull*), est une application, un outil, qui se place *devant* l'application web. Ce peut être sur le même serveur physique, ou bien sur un serveur physique (ou virtuel) dédié. Le gestionnaire de cache demande peu de ressources, si ce n'est de la mémoire, de sorte qu'il peut souvent être placé sur le même serveur que l'application web.

Pour une application web complexe, par exemple une gestion commerciale, ou bien un ERP, ou encore le back-office d'un outil CMS, le degré de variation des pages est en général trop important pour qu'il y ait beaucoup à gagner avec un gestionnaire de cache. Du moins pour les parties applicatives des pages, car on peut toujours gagner sur les composants auxiliaires tels que images, css ou bien javascript. Mais s'il ne s'agit que de ces composants, le cache natif du serveur Apache suffira en général. On veillera simplement à ce que ces composants soient servis directement par le serveur Apache, et non par le serveur d'application.

Dans beaucoup de cas, la mise en place du cache devant un CMS est pratiquement transparente, c'est à dire qu'on ne change rien du côté du CMS, et que l'on bénéficie immédiatement de performances décuplées. Pourquoi s'en priver ?

La transparence n'est pas toujours totale. La première adhérence réside dans la bonne gestion des durées de vie, et de durées de vie différencieres selon les composants, comme on l'a vu plus haut. Comme on l'a vu plus haut, la durée de vie d'une page (ou plus largement d'une « ressource ») est spécifiée par le producteur, dans l'entête HTTP. Or il est fréquent que les CMS ne laissent pas une maîtrise fine des paramètres de l'entête HTTP.

La seconde difficulté est la personnalisation des pages qui, comme on l'a dit, est de plus en plus importante sur les sites modernes. Ici deux approches sont pratiquées :

- Soit une distinction globale entre pages génériques et pages personnalisées, les premières étant gérées en cache, et les secondes non.
- Soit une gestion de cache par fragment, comme on le verra (cf « Cache par fragments », page 133), mais qui présente l'inconvénient de ne pas être transparente.

Les outils du cache HTTP

L'outil de cache le plus célèbre est Squid, un serveur proxy utilisé le plus souvent en mode « *reverse proxy* » c'est à dire en tant que serveur de cache. Squid remonte aux débuts du web, avec une version 1 qui date de 1996. C'est un projet qui a peu évolué, pendant pas mal d'années, mais qui s'est réveillé depuis 2008.

Squid permet de configurer la quantité de mémoire allouée au cache ; elle est gérée en MRU, mais les objets évacués du cache sont écrits sur disque, d'où ils peuvent être réutilisés dans les limites de leur durée de vie. De plus lorsqu'on arrête un Squid, il écrit sur disque tous les objets qui étaient en mémoire, et qui seront rechargés lors de la relance, ce qui est important car le redémarrage avec un cache vide peut amener une très forte sollicitation initiale du serveur source.

La gestion des durées de vie est essentiellement dictée par les consignes des entêtes HTTP, c'est-à-dire par l'application source. C'est au niveau de la source que se fait la configuration, qui permet donc de distinguer différentes typologies d'objet, aux durées de vie différentes.

Notons que dans sa version 3, Squid annonce le support ESI, mais il est encore en rodage pour l'instant.

Varnish est un outil plus récent, également open source, qui cible directement la fonction de cache, et annonce des performances supérieures.

Un cache HTTP est souvent mis en œuvre seulement pour les objets média, ou composants relativement statiques : images, css, flash, etc. Dans cette configuration, sa mise en œuvre est réellement transparente. Il peut apporter beaucoup sur les pages elles-mêmes, mais à condition de bien maîtriser les durées de vie de côté de l'application.

Cache par fragments

Sites personnalisés et contenus temps-réel

Un bon outil de cache peut servir quelques milliers de pages à la seconde, sur un unique serveur. Et ce n'est pas tout : le cache contribue aussi à la haute disponibilité puisque si l'application ne répond pas, le cache peut toujours fournir une page, peut être un peu ancienne, mais c'est mieux que rien.

Tout cela est magique, du moins tant que l'on sert les mêmes pages à tout le monde, au moins pour quelques temps. C'est le cas de beaucoup de sites, mais depuis quelques années déjà, on observe une tendance puissante vers la personnalisation des sites : chaque internaute est

identifié, et reçoit une page spéciale, dépendant de ses préférences et options.

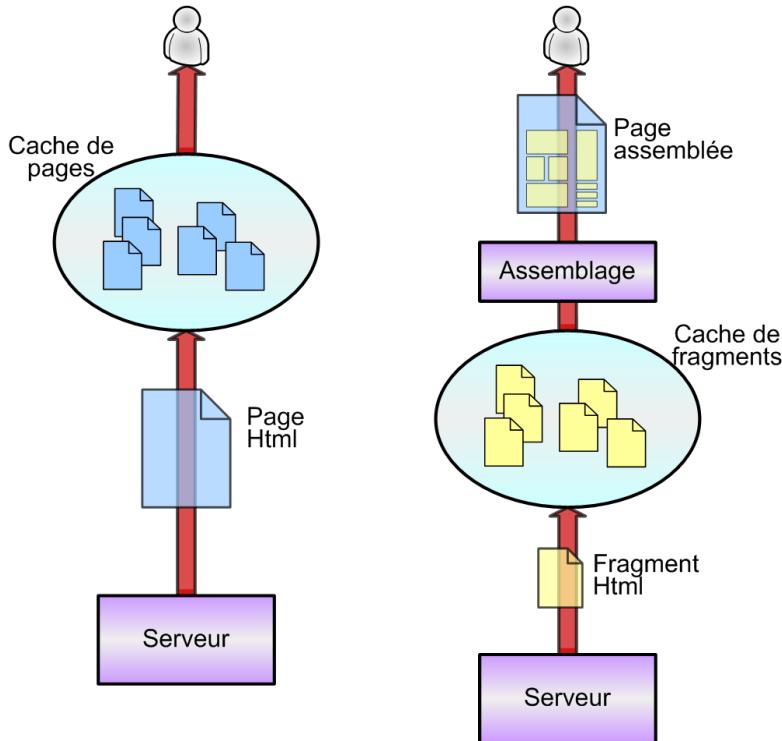
Si chacun reçoit une page unique, c'en est fini du cache et de ses milliers de pages par seconde. Faut-il en rester là ? Soit renoncer à des sites personnalisés, soit empiler une trentaine de serveurs pour tenir la charge ?

Non, heureusement, il nous reste quelques armes.

Introduction au cache par fragments

A bien y regarder, même si chaque page est unique, de nombreux morceaux de la page sont les mêmes pour différents internautes. A défaut de mettre la page entière en cache, on peut donc tenter de mettre des morceaux de pages en cache, des fragments. Typiquement, beaucoup d'éléments de navigation, mais aussi des blocs entiers de contenus, seront intégrés à l'identique dans les pages de tels et tels internautes, même si c'est éventuellement à des emplacements différents.

Cette réflexion amène à la notion de cache par fragments : on ne gère plus en cache des pages web entières mais des fragments de pages. Et le principe est que chaque fragment peut avoir une durée de vie particulière : des parties fixes de la page, un pied par exemple, ne seront rafraîchies que toutes les 4 heures, tandis que des actualités récentes seront rafraîchies tous les $\frac{1}{4}$ d'heure, et certains fragments, eux, ne seront pas du tout mis en cache, ils seront systématiquement demandés au serveur.



La figure précédente compare le cache de pages, à gauche, et le cache de fragments, à droite. Le cache de fragments implique une phase d'assemblage, d'agrégation, en aval du cache. Mais il permet de produire une variété de pages à partir d'un ensemble de blocs, de fragments, qui ont des durées de vies variées.

Agrégation de fragments et portails J2EE

La gestion d'un cache par fragment combine donc deux fonctions distinctes : la fonction d'agrégation et la fonction de cache. La fonction d'agrégation consiste à produire une page à partir de différents fragments.

C'est une fonction qui correspond à un besoin usuel, indépendamment de la problématique du cache, puisque c'est la fonction principale des outils de portail, des outils tels que Jetspeed, Liferay, Uportal, ou encore Websphere Application Portal. Ce sont les portails du monde J2EE, les portails répondant à la norme JSR168-JSR286. Cette norme définit des APIs entre le portail agrégateur et les applications fournissant les contenus agrégés. Ces APIs sont définies en Java, et donc ne conviennent que pour cet environnement, elles ne sont pas technologiquement neutres.

C'est là une limitation intrinsèque des portails J2EE : ils sont faits pour agréger des contenus fournis par des applications Java. Et ce n'est pas tout : des applications Java *écrites pour supporter les APIs du portail*. C'est beaucoup demander et dans la pratique, le besoin est très souvent

d'agrégger des contenus issus d'applications d'une part hétérogènes, d'autre part pré-existantes, et pas uniquement Java. En d'autres mots : il faut prendre les applications comme elles sont.

Or la fonction du cache agrégateur de fragments reprend, pour partie, cette mission d'agrégation du portail, de manière à la fois plus simple et technologiquement neutre.

Edge Side Include (ESI)

Akamai a défini le *Edge Side Include* (ESI), devenu une norme du W3C, et adopté par quelques autres grands acteurs. ESI permet d'inclure dans une page web des fragments html obtenus en HTTP, en interrogeant différents serveurs. Pour spécifier cela, on inclut dans la page principale des tags de la forme

```
<esi:include src= "HTTP://www.mysite.com/fragment01" />
```

Cette balise signifie qu'à cet endroit de la page doit être inséré le fragment obtenu à l'adresse indiquée. Il appartient alors au dispositif ESI d'aller chercher cette ressource à l'adresse HTTP indiquée, et de l'insérer à l'endroit du tag `<esi/>`.

Un point important est que pour le fournisseur de la ressource, aucune interface spécifique n'est requise, il s'agit simplement de servir un morceau de html sur le protocole HTTP.

Dans un système de cache par fragment, le gestionnaire de cache est donc en charge d'élaborer la page, en insérant tous les fragments à la position appropriée. Lorsque tous les fragments sont en cache, c'est immédiat, et cela ne requiert aucun accès au serveur HTTP. Si un fragment n'est pas dans le cache, ou bien si la version en cache est périmée, trop ancienne, alors ce fragment seulement est obtenu auprès du serveur.

Un tel dispositif de cache par fragments permet donc de faire cohabiter dans les pages, des parties stables et des parties dynamiques, des parties identiques pour tous et des parties personnalisées, chacune bénéficiant d'un cache conforme à son besoin.

L'un des inconvénients du cache par fragment, qu'il s'appuie sur ESI ou non, est **qu'il n'est pas transparent pour les applications**. La structure même de l'application est impactée : au lieu de servir des pages, elle doit servir des fragments. Ce n'est pas anodin car toutes les applications, tous les outils de développement, sont conçus pour élaborer des pages web. A commencer par les outils de gestion de contenus, qui sont derrière la plupart des sites.

Sur ce thème, voir également « Infrastructures Globales et CDN », page 46.

Web-scraping, web-clipping

Un cache ESI est donc un bel outil, mais il implique en général de recevoir des fragments élaborés spécifiquement à cet usage. Dans certains cas, on souhaite plutôt manipuler des fragments directement récupérés d'applications existantes, c'est à dire des morceaux extraits de pages entières, typiquement un bloc <div> au sein de la page.

La technique que l'on appelle *web scraping*, ou *web-clipping*, consiste à récolter des données au sein de pages html. On peut pratiquer le *web scraping* avec une participation de l'application producteur, ou bien à son insu. Lorsque l'on maîtrise l'application qui produit la page, alors on pourra soit lui demander de fournir ses contenus d'une manière plus structurée que sur du simple Html (par exemple une interface de type REST/XML), ou bien lui demander plus simplement encore de placer des balises particulières au sein du Html, qui permettront au *consommateur* de repérer à coup sûr la partie utile. Lorsqu'on ne maîtrise pas l'application *producteur*, alors on ne peut qu'essayer de se repérer par rapport aux balises du Html, typiquement rechercher un « div » particulier, puis prendre le second tableau, la troisième ligne, etc. C'est cela qu'on appelle le plus souvent *web scraping*, et l'on voit bien que c'est une technique assez fragile, car le moindre changement de montage html casser l'identification des fragments.

Caches ESI Open Source

Malheureusement, il n'existe pas d'outil open source solide pour la gestion d'un cache ESI. C'est une fonctionnalité prévue de longue date dans Squid v3, mais Squid v3 n'est pas « *production ready* » à ce jour, selon l'aveu même de ses développeurs : « la plupart des bugs restant de Squid v3 concernent le ESI ».

Varnish en est à peu près au même stade, avec une implémentation d'un sous-ensemble de la norme ESI, mais ici aussi, l'avertissement est « attendez-vous à des bugs » !

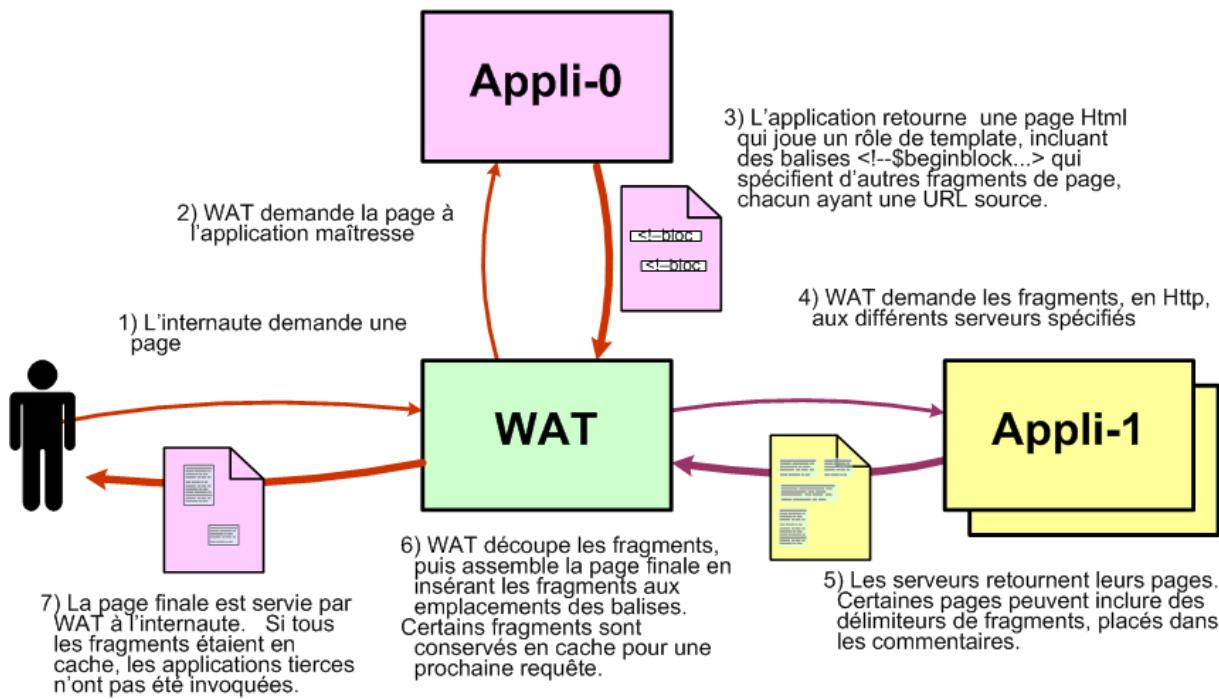
Notons que l'on peut obtenir un service d'agrégation semblable en utilisant le module *Server Side Include (SSI)*, de Apache, mais il faut encore y ajouter une gestion de cache ordinaire par Squid.

Le Web Assembling Toolkit

Pour d'autres types de projets, Smile a mis au point le Web Assembling Toolkit (WAT). Il s'agit d'un ensemble de taglib JSP, qui permettent de constituer des pages en insérant du contenu obtenu en HTTP à partir d'autres serveurs. Et même si l'agrégation est sa mission première, WAT gère aussi un cache, sur chacun des fragments ainsi obtenus.

Comme on l'a vu, la clé de la haute performance c'est de réunir cache et agrégation. WAT combine ainsi les deux rôles : outil d'agrégation, et outil de cache par fragments.

Un fonctionnement général que l'on peut représenter comme suit :



Et WAT a une autre caractéristique : lorsqu'il appelle un serveur pour obtenir un fragment, il est capable d'extraire un petit morceau de la page obtenu. C'est du web-clipping : on obtient une page, et on découpe le morceau qui nous intéresse, qui constituera le fragment. Cette technique permet en particulier d'obtenir des fragments à partir d'applications existantes, qui n'auraient pas été conçues spécifiquement pour être agrégées ainsi au sein d'un portail. Pour certains, le web-clipping est une solution un peu rustique, du bricolage presque. Ne vaudrait-il pas mieux invoquer un superbe web-service ? C'est discutable. Le bénéfice essentiel du web-clipping est le caractère non-structurant : il ne demande rien de particulier aux applications intégrées.

Placé en frontal des applications, WAT joue en quelque sorte le rôle d'un portail. Mais dans certains cas, il est en fait combiné à un véritable portail, par exemple Jahia. C'est le cas du nouveau site d'assurance idmacif.com, de la Macif, ou du portail de Bouygues Immobilier.

WAT est également utilisé par Editions Francis Lefebvre, Bluestar Silicones, La Poste, et le Conseil Supérieur du Notariat. C'est un projet

open source sous licence Apache³. Et depuis début 2009, WAT supporte les tags ESI les plus importants.

Agrégation de fragments côté client

Si l'on parle d'agrégation de fragments, il est important de citer aussi la pratique d'une agrégation « côté client », c'est à dire sur le navigateur de l'internaute.

Il existe différentes techniques pour cela : iframes, inclusion de javascript générant des fragments Html au sein de la page, et enfin véritable Ajax.

L'agrégation côté client a de multiples avantages :

- Elle permet d'agréger des contenus venant de *differents serveurs*
- Le traitement ne demande *aucun outil spécifique* côté serveur
- Le traitement ne consomme *aucune ressource* côté serveur

C'est en particulier une excellente technique lorsqu'un site produit des pages *partiellement personnalisées*. Typiquement avec un bloc personnalisé de type « mon profil », « mes préférences », etc. Dans ce cas, on peut bénéficier pleinement de la gestion de cache serveur (et client), en ne traitant de manière dynamique que le bloc considéré.

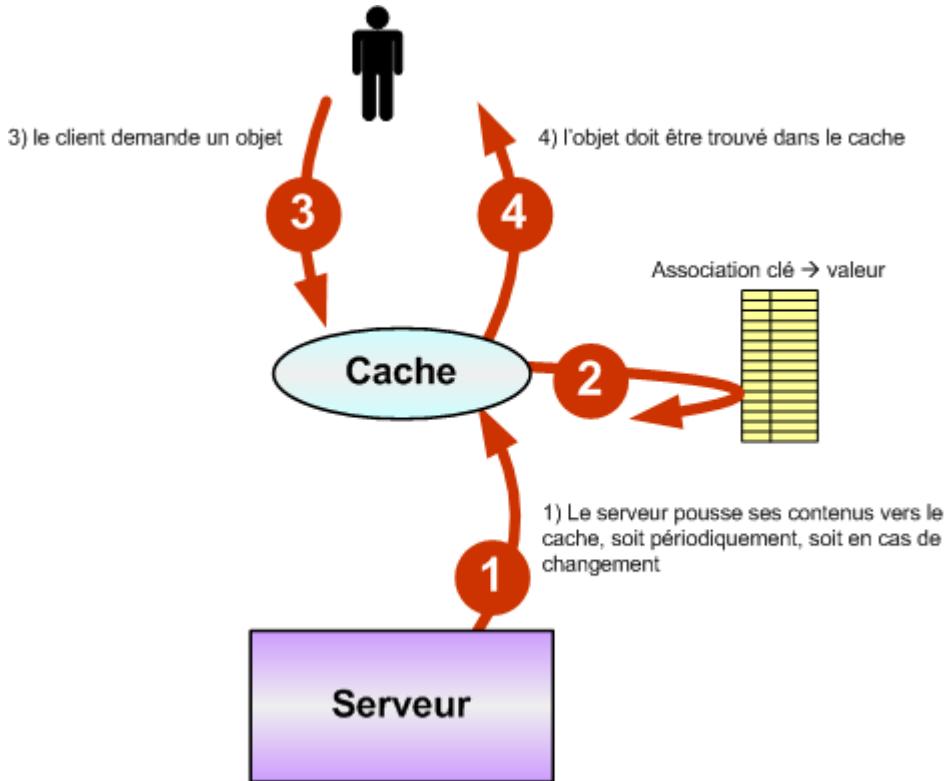
Cache en mode *push*

Nous avons défini plus haut le principe du cache en mode *push* : c'est le producteur qui pousse les objets, les ressources, vers le cache.

A la différence du consommateur, le producteur, qui gère la version originale de l'information, sait quand l'information a changé. Ainsi à la manière des dispositifs de réPLICATION des bases de données, on peut faire en sorte que ce soit *l'événement de changement qui déclenche l'envoi* de l'information aux différents caches. Cela suppose une forme d'inscription, d'abonnement, du cache auprès de la source de données. Et cela implique aussi une phase d'initialisation. Dans un cache en mode *push*, on suppose généralement que le cache possède une copie de l'ensemble des données et non un sous-ensemble particulier.

C'est ce que l'on peut représenter comme suit :

³ WAT : <http://sourceforge.net/projects/webassemblytool>



On voit sur ce schéma que le point de départ du processus est cette fois-ci le serveur producteur de l'information, qui dépose, pousse, les objets sur le cache.

On peut aussi mettre en place des modes mixtes, où le producteur ne pousse pas la ressource vers le cache, mais déclenche son invalidation, de sorte qu'elle sera rafraîchie à la prochaine utilisation. C'est ce que permet Squid par exemple, avec un service d'invalidation, qui peut être appelé par simple invocation d'une URL.

Génération de pages statiques

La génération de pages statiques est un cas particulier du cache en mode *push*.

A l'origine du web, il y avait des fichiers html posés dans les répertoires d'un serveur. Le serveur HTTP a été initialement conçu pour analyser les requêtes des internautes et servir ces pages. On les a appelées pages statiques plus tard, par opposition aux pages dynamiques élaborées par des applications.

Aujourd'hui encore, rien n'est plus efficace que de servir des pages statiques. Le serveur Apache gère nativement un cache pour ce type de pages, et peut en servir plusieurs milliers à la seconde, pour autant qu'on lui donne la mémoire qu'il lui faut.

Architectures Hautes-Performances

Les pages statiques, ce n'est pas juste plus rapide, c'est aussi plus fiable, bien sûr.

Mais le tout-statique a aussi beaucoup d'inconvénients, en particulier le manque de flexibilité, la difficulté à traiter chaque internaute de manière spécifique, et à gérer de l'information changeante. Et la difficulté à assurer une parfaite cohérence des liens en présence de déplacements ou renommages de rubriques, et à assurer une parfaite cohérence graphique.

C'est pourquoi le web est passé en mode dynamique. Avec un niveau de service très supérieur, mais aussi de moindres performances.

Ne peut-on pas avoir *à la fois* les hautes performances *et* l'aspect dynamique ?

Dans certains cas, on le peut. C'est le propre de la génération de pages statiques. Le principe est celui d'une génération de page dynamique, mais la page une fois générée est écrite sur disque en un fichier Html, et c'est ce fichier qui est servi par le serveur HTTP.

Beaucoup de sites ont un taux de changement très faible. Il s'écoule souvent plusieurs jours, voire une semaine entière sans le moindre changement. A quoi bon régénérer des pages toujours identiques ?

Bien sûr, les dispositifs de cache, déjà évoqués, répondent aussi à cette problématique : éviter de recalculer une page qui n'a pas changé. Et effectivement, génération statique et solution de cache ont la même finalité. Mais des atouts spécifiques.

Une solution de cache en mode *pull*, invoque son application moins souvent, mais l'invoque quand même régulièrement. Si l'application est en dysfonctionnement, l'impact est retardé, mais demeure.

Par rapport à un dispositif de cache, la génération de pages statiques permet de découpler bien plus fortement le système d'édition, le back-office, et le système de publication, le front-office. Cela permet de mettre en place des architectures dans lesquelles la plateforme de production est minimalist et simplifiée au maximum. Servir des pages statiques est non seulement ultra-performant, mais aussi ultra-extensible.

Mais s'il y a plusieurs dizaines de milliers de pages, voire centaines de milliers, la régénération peut être un processus de plusieurs heures. La solution serait une régénération partielle, qui ne porterait que sur ce qui a changé. Mais l'analyse des impacts peut être un problème complexe. Certains petits changements peuvent impacter un très grand nombre de pages, voire toutes les pages ;

Il n'y a pas d'outil standard de génération statique, c'est à dire que c'est une technique encore assez artisanale, mais les bénéfices peuvent être énormes. Smile a été amené à mettre en place un principe de génération de pages statiques à partir d'un CMS à de nombreuses reprises. Citons les cas de la plateforme voyages-sncf.com, de Bouygues Telecom, ou encore de Sport24.com.

A la base de ces techniques, il y a un principe semblable à celui de l'« aspiration » de site : un petit programme externe au CMS va demander une page, et la stocker sur le disque. La commande `wget` est généralement utilisée à cet effet. Il est possible de procéder vraiment comme un aspirateur de sites, c'est à dire de rechercher les liens dans la page obtenue, et de requêter ensuite les pages correspondant à ces liens. Mais c'est une technique trop grossière, qui ne permet pas de corrélérer la génération de page avec les changements effectifs, et oblige donc à tout regénérer périodiquement.

Pour ne regénérer que le minimum, et pouvoir donc opérer en quasi temps-réel, il faut d'une part déclencher l'export statique sur l'événement de changement d'un contenu et d'autre part bien identifier les impacts d'un changement.

Cache de données

Cache de données

Nous avons dit plus haut que les mécanismes de cache se retrouvaient à tous les niveaux. Nous avons consacré un chapitre au cache de pages, mais le cache positionné entre application et gestion des données est essentiel également, particulièrement dans les architectures les plus modernes.

Approche ensembliste ou clé/valeur

Le cache de données n'est pas juste une gestion de données en mémoire, il fonctionne selon un paradigme différent, infiniment plus simple et donc plus performant. A l'inverse bien sûr, il lui manque énormément de fonctionnalités, à commencer par les propriétés ACID.

Le concept du SGBD est de nature *ensembliste*, c'est à dire que les requêtes que l'on adresse au SGBD sont *en référence à un ensemble d'entités*, défini par la clause WHERE, ou bien par une jointure.

C'est la force et la puissance des SGBD, qui permet en quelques mots, d'exprimer une sélection ou une mise à jour qui correspondraient à un traitement complexe.

Mais cette approche ensembliste présente aussi des inconvénients.

D'une part, elle est d'un maniement délicat. Les requêtes complexes sont pratiquement imprévisibles en termes de performances, et doivent subir un tuning spécifique pour être utilisables en production.

Plus important, l'approche ensembliste est difficilement compatible avec un partitionnement. Or le partitionnement est l'une des meilleures voies vers l'extensibilité.

Dans la pratique, il est fréquent qu'un SGBD ne soit pas utilisé de manière ensembliste, mais selon un paradigme clé-valeur, moins puissant, mais beaucoup plus simple. Et la programmation objet a accentué cette tendance.

Fondamentalement, le paradigme clé-valeur tient en deux primitives :

- Lire *valeur* correspondant à *clé* : `value=get(key);`
- Ecrire valeur correspondant à clé : `set(key,value);`

Pour ce type d'utilisations, le SGBD n'est parfois pas le bon outil.

Le cache gestionnaire de données

Comme on l'a vu en introduction, la vision traditionnelle est que la mémoire est certes très rapide, mais (1) trop chère donc trop rare et (2) non persistante donc trop fragile. Le modèle classique du cache part de ces deux hypothèses, supposant que (1) on ne peut pas compter avoir toutes les données en mémoire et (2) même si on le pouvait ce serait trop volatile.

Donc le dispositif de référence pour stocker les données est le disque, que ce soit géré par un SGBD ou directement en fichiers. Et on utilise le peu de mémoire dont on dispose en mode cache MRU.

Aujourd'hui, la mémoire est toujours aussi peu persistante, mais elle n'est plus rare. Elle peut donc devenir le moyen de stockage primaire, le gestionnaire de données. Non pas des données les plus utilisées, mais de toutes les données.

Si on a un millions de membres inscrits et 50 KO de données sur chacun d'eux, ça ne fait finalement que 50 GO de RAM à prévoir, ce qui n'est pas une grosse dépense. Surtout si c'est pour servir 200 000 requêtes par seconde. Donc on peut dire: *la totalité de mes données sont en mémoire, tout le temps*. Même si un utilisateur n'est pas venu depuis une semaine ou un mois, ses données sont en mémoire, comme les autres.

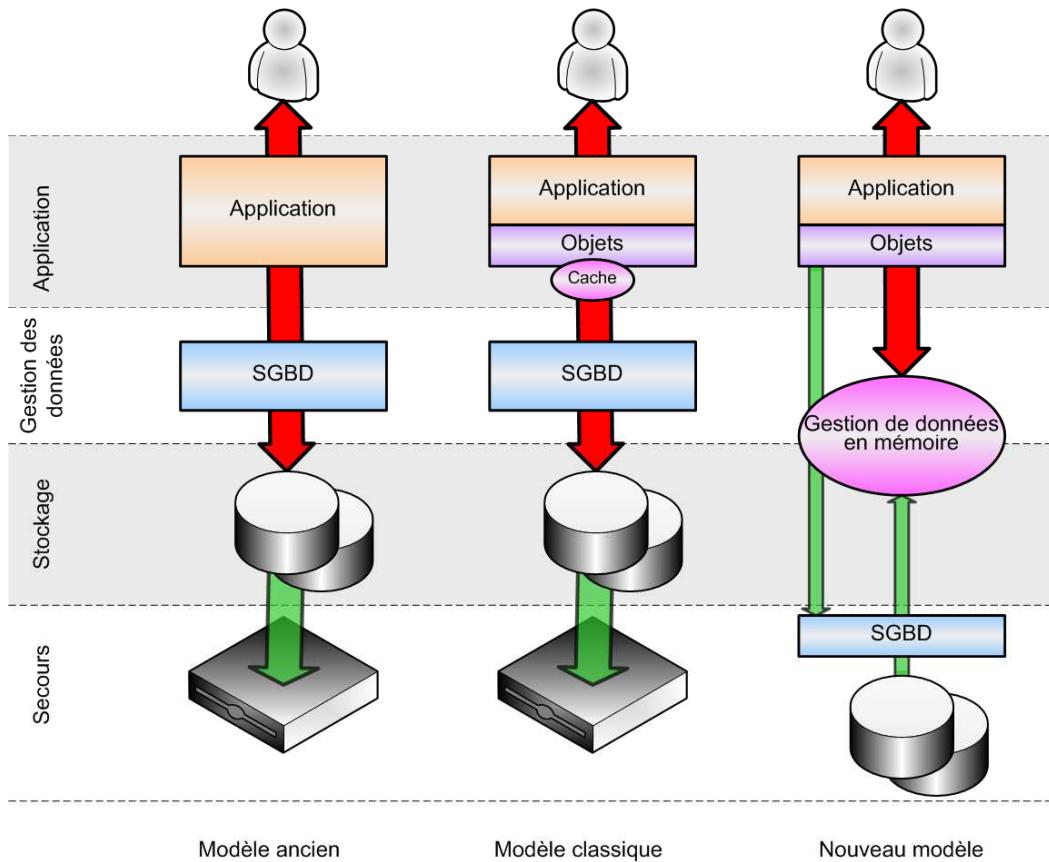
Malgré tout, la mémoire reste fragile, et il faut donc une sauvegarde persistante. C'est là qu'intervient le disque. Autrement dit, le disque joue dans ces architectures le rôle que jouaient les bandes auparavant:

Architectures Hautes-Performances

c'est l'ultime recours, mais en temps ordinaire on n'en a pas besoin. Avec quelques différences tout de même. La « sauvegarde » sur le disque s'effectue en temps-réel, à chaque modification. Le disque reste en fait le lieu « de référence » pour l'information, mais non le lieu « de gestion ordinaire ».

Dans ce modèle, la gestion MRU demeure, mais elle n'est plus d'une grande importance. En revanche, le principe de l'alimentation du cache en mode *pull*, reste essentiel, car le principe continue d'être que l'on peut perdre tout ou partie du cache sans dommage. Dans ce cas, le cache sera reconstitué au fur et à mesure de la demande.

C'est une réflexion que l'on a représenté sur le schéma suivant :



A gauche, l'ancien modèle, disons des années 90 : l'application s'adresse au SGBD qui stocke ses données sur disque, avec une certaine utilisation du cache, que nous n'avons pas représentée.

Au milieu, le modèle aujourd'hui classique. L'application manipule des objets, ces objets sont construits par la couche ORM à partir des

données gérées toujours dans le SGBD, avec un dispositif de cache entre les deux.

Dans les deux cas, le stockage disque est secouru sur des supports magnétiques offline.

A droite, le nouveau modèle, à la fois de très hautes performances et très haute extensibilité. On a placé ici le SGBD au niveau « secours », en même temps que ses disques. Il peut y avoir encore un secours offline de second niveau, mais la tendance est plutôt à gérer la totalité de l'archivage sur disque.

Dans les couches supérieure, on retrouve la modélisation objet, mais le cache prend une place nouvelle, il est le gestionnaire principal des données. A tout instant, toutes les données sont en mémoire quelque part dans le cache, sauf du moins dans quelques états transitoires.

Memcached

Memcached est un outil de cache open source, qui a la particularité d'être un cache distribué. C'est un cache générique, c'est à dire qu'il peut stocker n'importe quels objets, de 1 octet à 1 mégaoctets environ.

Memcached fonctionne comme un serveur, au sens applicatif du terme. C'est à dire que, à la manière d'une base de données, les applications adressent des requêtes (sur TCP/IP) à memcached, qui leur répond.

Memcached fonctionne selon le paradigme du dictionnaire, c'est à dire une correspondance clé-valeur, à la manière des *hashtable*.

Les requêtes que les applications adressent à memcached sont très simples : *lire valeur* correspondant à *clé*, *écrire valeur* correspondant à *clé*.

Puisque memcached fonctionne pratiquement comme une *hashtable*, une simple fonction d'association (clé, valeur), pourquoi ne pas utiliser celle-ci ?

C'est là qu'intervient le caractère distribué. On peut lancer autant d'instances de memcached, sur autant de serveurs que l'on veut, elles se répartiront le travail, et le stockage, de manière totalement automatique et transparente.

Chaque paire clé-valeur ne sera stockée que dans une seule des instances de memcached. Le fonctionnement est d'une simplicité superbe. Lors de chaque accès, la clé est *hashée*, c'est à dire qu'on lui applique un petit algorithme très rapide, qui aboutit à un nombre, et à ce nombre on applique un modulo N, où N est le nombre d'instances. Le résultat de ce calcul indique l'instance à laquelle s'adresser. Une fois que la requête parvient à la bonne instance, le reste n'est qu'un

accès *hashtable* ordinaire. On voit bien que ce fonctionnement est extensible à l'infini.

Pour autant, avant de multiplier les instances, il faut analyser le besoin : tant qu'une instance unique n'est pas saturée, on n'a pas besoin de plus.

Comme tous les dispositifs de cache partiels, le cache distribué de memcached est naturellement robuste, tout simplement parce que si une donnée n'est plus disponible, on va la chercher à la source. Le cache ne porte aucune donnée de référence.

Du moins, c'est la manière recommandée de l'utiliser, mais rien n'interdit d'utiliser memcached comme seul gestionnaire des données, du moment que l'on a analysé les conséquences d'une perte.

Memcached est utilisé en particulier par Facebook, qui gère 800 serveurs de cache, totalisant 28 tera-octets de mémoire. Avec quelques optimisations de memcached, Facebook annonce des performances de 200 000 requêtes par seconde par serveur.

EhCache

Ehcache fonctionne également en mode serveur ; il n'est pas distribué, mais plutôt répliqué.

Avec Ehcache, chaque objet en mémoire est répliqué sur les différentes instances Ehcache, contrairement au principe de memcached. Cette réPLICATION présente nécessairement des limites en termes de scalabilité : le nombre de messages à échanger entre les instances croît plus ou moins avec le carré du nombre d'instances.

Pour cette réPLICATION, Ehcache peut utiliser différents modes : lors de la modification d'un objet, les autres instances peuvent seulement invalider l'objet en cache, ou bien en recevoir la copie. C'est dans ce second cas que l'on peut réellement parler de réPLICATION.

De même, la réPLICATION peut être synchrone ou bien asynchrone.

Ehcache peut s'utiliser en temps que librairie simple, pour une application unique en environnement J2EE donc, ou bien en temps que serveur de cache, partagé par plusieurs applications.

Une nouvelle instance de Ehcache peut initialiser son cache auprès du cluster d'instances actives.

Ehcache supporte la persistance, c'est à dire qu'il peut conserver ses données sur disque lors de l'arrêt, et les reprendre à la relance.

Il supporte l'API java JSR107, JCache, mais peut également être utilisé en tant que serveur, sur des APIs REST ou SOAP. Le mode serveur le rend accessible à des applications de tous environnements techniques.

Les valeurs de *time to live* et *time to idle* peuvent être définies pour chaque instance, mais des valeurs *par objet* peuvent aussi être spécifiées, qui prévalent.

Il est capable d'avoir une stratégie d'éviction des *moins fréquemment utilisés*, et non simplement des *moins récemment utilisés*. Il y a une petite nuance : un objet peut avoir été utilisé 100 fois à un instant T_0 , puis plus du tout pendant 10 secondes. Sa fréquence d'utilisation sur les 10 dernières secondes reste de 10 fois par seconde. Un second objet peut n'avoir été utilisé que 2 fois sur ces 10 secondes, mais la dernière fois était à la dixième seconde. S'il faut en purger un, alors la politique LRU purge le premier, tandis que la politique LFU purge le second.

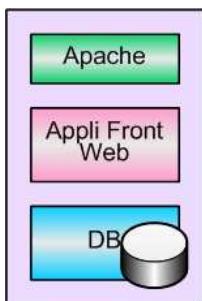
QUELQUES CAS D'ECOLE

Une montée en puissance ordinaire

Le problème posé

Prenons un petit site web ordinaire, qui tourne sur un serveur unique. Peu importe le système d'exploitation, les outils de développement et frameworks ou le SGBD retenu, nous ne parlons ici que d'architecture.

On peut le schématiser comme suit :



Un serveur HTTP, Apache sur le schéma, une application frontale web et une base de données. Le bloc « Appli Front Web » pourrait, selon les cas, se décomposer en différentes couches, en particulier un serveur d'application tel que Tomcat ou JBoss, ou encore un interpréteur de langage tel que PHP ou Perl. Mais ce découpage ne nous est pas utile ici.

Il n'y a aucune métrique pour nous dire combien de pages par seconde cette application pourra servir, cela dépend de trop de paramètres, liés tant à la manière dont elle est codée qu'à son utilisation de la base de données.

Néanmoins, le service tournait de manière satisfaisante. Le site est un succès, l'audience augmente, et petit à petit les performances se dégradent. En général, le premier signal n'est pas juste un temps de réponse qui augmente légèrement. La première alerte vient sur des heures de pointes où le temps de réponse augmente non pas un peu, mais énormément. Cela dure quelques minutes, puis le trafic retombe et tout semble rentrer dans l'ordre. La sollicitation est auto-régulée, c'est à dire que le ralentissement dégoûte une partie des internautes.

A ce stade, il est essentiel d'avoir mis en place un monitoring qui nous permettra d'identifier ces crêtes mal supportées.

Optimisation

On peut hésiter souvent entre une démarche d'optimisation et une augmentation des ressources. D'un point de vue écologique,

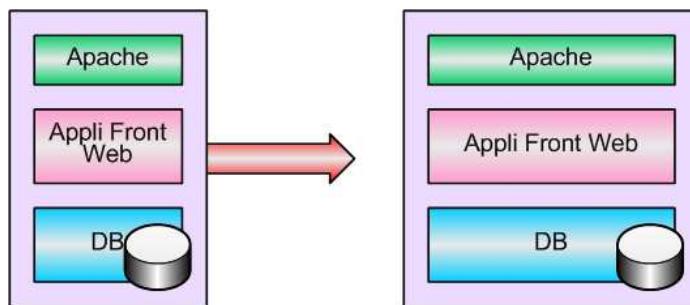
l'augmentation des ressources sans optimisation est une aberration, comparable à monter le chauffage au lieu d'isoler les fenêtres.

Néanmoins l'optimisation peut être coûteuse, et de plus, ses résultats ne sont pas garantis, de sorte que d'un point de vue économique, un simple changement de serveur peut être une meilleure voie.

Extension cellulaire

Une première étape donc, souvent la plus simple, consiste à basculer sur un serveur plus puissant. Il y a quantité de manière d'être plus puissant, et cet upgrade requiert une analyse : processeur multi-coeurs, disques plus rapides, mémoire augmentée, bi-processeur, ... Pour que l'opération soit justifiée, il faut qu'il y ait au moins un facteur 2 à gagner, c'est-à-dire que – selon la fameuse loi de Moore – le serveur précédent ait environ 18 mois d'âge.

L'upgrade ne relèvent pas de l'architecture, néanmoins, pour mémoire, représentons le ainsi :

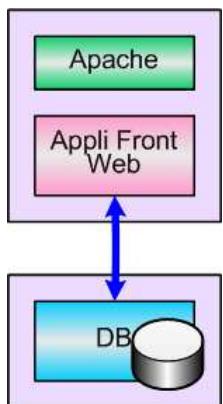


Bien, maintenant on arrive à un bi-processeurs de dernière génération, le succès croît toujours, et ça rame de nouveau.

Extension verticale

L'étape suivante la plus simple est l'extension verticale, ou encore *fonctionnelle*. Nous avons identifié deux *fonctions* principales dans cette architecture, l'application frontale et la base de données. L'extension fonctionnelle consiste à affecter des serveurs spécialisés sur chacune de ces fonctions. Ce qui donne le schéma suivant :

Architectures Hautes-Performances



L'avantage du découpage fonctionnel c'est qu'il est presque toujours transparent pour les applications.

Ainsi la base de données est *déjà* accédée par des protocoles réseau, qu'elle soit sur le même serveur ou bien sur un autre. Le fait de la déporter est totalement transparent pour l'application.

On a maintenant globalement deux fois plus de ressources physiques (CPU, Mémoire, Disques), au service de notre site, ce qui ne peut faire que du bien.

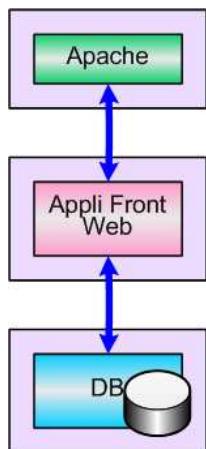
Avant d'aller plus loin, analysons un peu mieux cette configuration.

En premier lieu, on remarque que si n'importe lequel des deux serveurs tombe en panne, le site est arrêté. La probabilité de panne a doublé, et donc le taux d'indisponibilité a doublé. Ce n'est pas bon, mais pas forcément éliminatoire. On peut toujours envisager un secours à froid, disponible sur la plateforme.

Une autre remarque. La consommation de ressources, disons par exemple de CPU, par chacune des fonctions, par chacun des « étages » de l'architecture, est tout à fait corrélée. Une requête typique va induire une charge C_F sur le frontal, et une charge C_{DB} sur la base, et l'on peut définir un coefficient $k = C_F / C_{DB}$, sans présager de sa valeur.

Où veut-on en venir ? Supposons maintenant que $k=0,5$, c'est à dire que le trafic induit une charge deux fois plus forte sur la base de données que sur le frontal. La base de données arrivera à saturation alors que le frontal sera à la moitié de sa capacité. Mais si $k=2$, alors c'est le contraire, le frontal est saturé longtemps avant la base. D'une manière ou d'une autre, il y a de la ressource gaspillée. Cette architecture manque de flexibilité.

On voit parfois des plateformes allant plus loin encore dans le découpage fonctionnel, en détachant l'étage Apache :

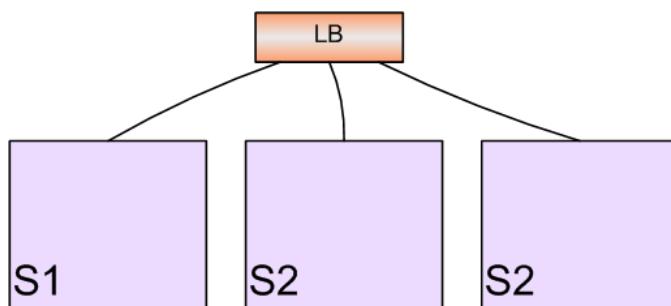


On est content alors d'annoncer une architecture vraiment multi-tiers, comme si le but était d'avoir le plus de « tiers », de couches, possible ! Pourtant, cette architecture est souvent peu intéressante, car d'une part la tolérance aux pannes doit être gérée sur chacun des trois niveaux, et d'autre part chacun des niveaux doit être dimensionné séparément, comme évoqué plus haut.

Clairement, si l'on a un dimensionnement 1-1 entre l'étage HTTP frontal et l'étage applicatif, alors on gaspille de la ressource : l'étage Apache n'a presque rien à faire. A la rigueur, on peut lui faire servir les pages et composants statiques, et donc le configurer de manière optimisée pour cette tâche. On peut aussi positionner un outil de cache frontal sur ce serveur, ou lui faire jouer un rôle de load-balancer de niveau 7.

Extension horizontale

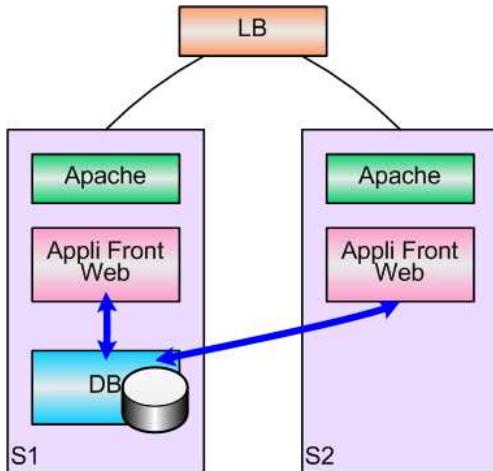
Une étape intermédiaire possible également, est une extension horizontale d'abord, de la manière suivante :



On voit apparaître le « Load Balancer » (LB), qui assure la répartition de charge. On se réfèrera au chapitre traitant des techniques et algorithmes de la répartition de charge.

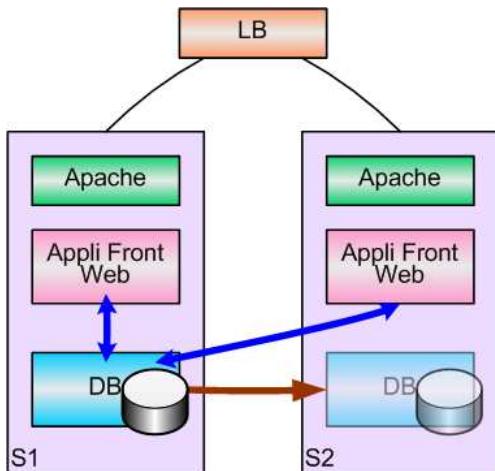
La base de données présente une contrainte particulière : elle doit le plus souvent être centralisée, afin d'assurer la cohérence des données.

En restant sur une seule « couche », on peut tout à fait partager la base de données, placée sur l'un des serveurs :



Ce serveur S1 a davantage de travail pour servir une requête, on peut donc configurer le load-balancer pour lui attribuer un peu moins de requêtes.

Et, pour ne pas ajouter un serveur supplémentaire si l'on peut s'en passer, on utilisera S2 comme secours du SGBD, avec une réplication entre S1 et S2.



En général, la réplication sollicite peu le serveur cible, de sorte qu'il conserve malgré tout une capacité supérieure.

Une alternative à cette réplication de niveau SGBD est une réplication de niveau disque, avec DRBD, déjà vu plus haut.

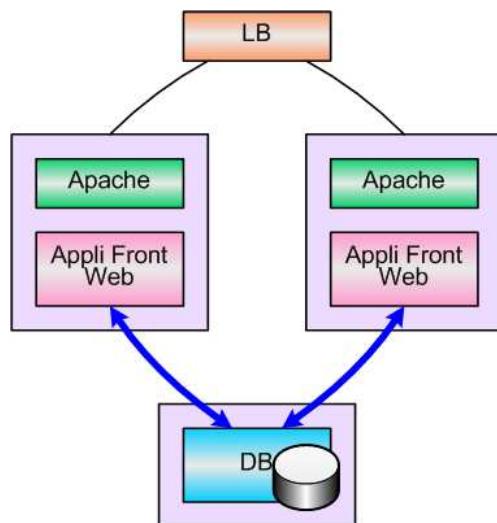
Dans cette configuration, on choisira de préférence de placer la base de données dans une VM distincte, dans une architecture virtualisée. Ainsi, il sera facile de la déplacer plus tard.

La capacité d'accueil est pratiquement doublée, en temps normal, mais en cas de panne d'un serveur elle revient à ce qu'elle était : on n'a pas

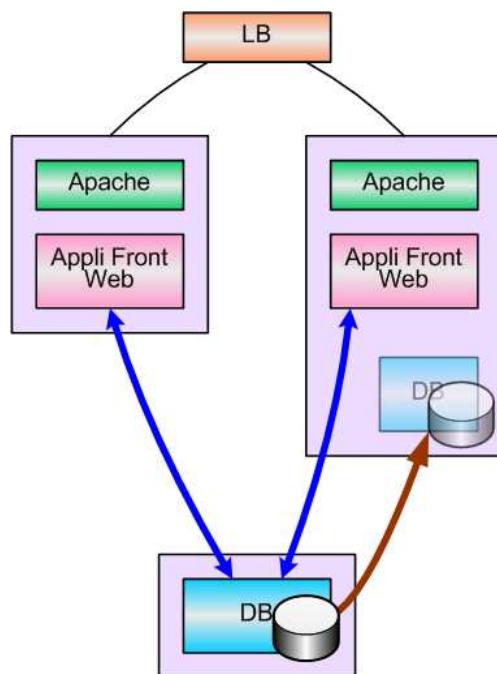
simultanément tolérance aux pannes et capacité doublée. Malgré tout, cette configuration horizontale est supérieure à la configuration verticale précédente, qui au contraire dégradait la disponibilité.

Extension en 2D

L'étape suivante, maintenant, consiste à avoir plus de frontaux, tout en détachant l'étage SGBD.



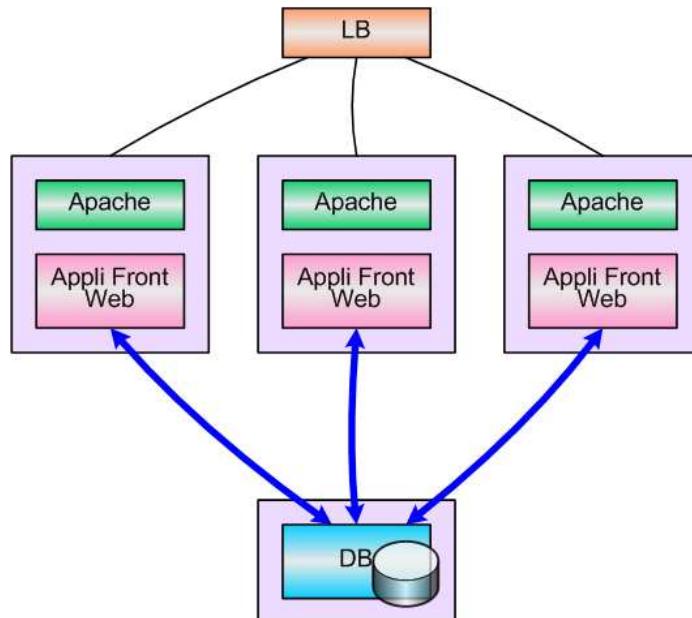
On pourra, ici aussi, répliquer la base de données sur l'un des serveurs disponibles, de manière à disposer d'un secours, ou au minimum d'une sauvegarde en continu.



Architectures Hautes-Performances

Ici, on est encore à un stade « modeste », ou « low-cost », où l'on essaye de calibrer le nombre de serveur au plus juste.

Et bien sûr, une fois à 2 frontaux, on peut passer à 3, 4, ... N frontaux pour une même base de données, selon le même principe de répartition :

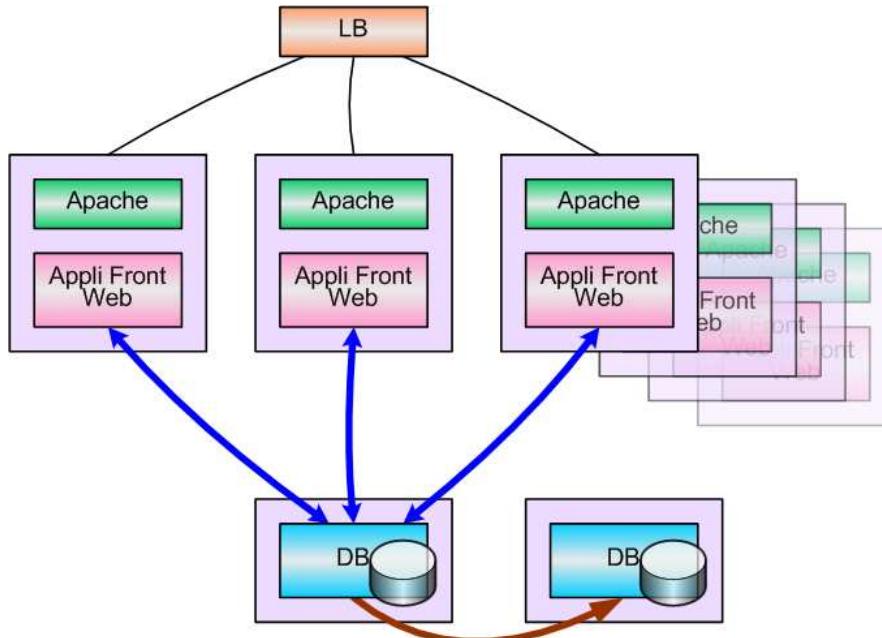


Nous avons là un grand classique des architectures web ordinaires.

On en voit bien les limites : la base de données unique et centrale finira tôt ou tard par être le goulot d'étranglement. A quel stade ? Combien de frontaux pour une base ? Il n'y a aucune règle en la matière. La sollicitation de la base de données, et donc le ratio k cité plus haut, dépend entièrement de la typologie de l'application, de sa bonne utilisation de la base, des couches d'abstraction type Hibernate, d'un cache sur les données, et bien sûr du bon tuning des requêtes. On peut rencontrer des architectures mettant en œuvre une dizaine de frontaux partageant une base de données.

Comme évoqué plus haut, la base est ici aussi point de fragilité, « SPOF » comme on dit. Sur une architecture d'envergure, on n'utilisera pas un frontal comme secours, ce qui est un peu mesquin, on préférera privilégier l'homogénéité des configurations et des fonctions.

On met donc en place, a minima, un secours par réPLICATION, comme suit :



Le passage en secours, dans une architecture de ce type, peut être rendu pratiquement transparent. Il faut quelques secondes pour que la base de secours prenne la fonction et les applications doivent se reconnecter sur cette base. Les connexions SGBD en cours sont perdues, mais certains pools de connexions savent gérer une reconnexion automatique, dans un mode « *test on borrow* ». Lorsque l'application redemandera une connexion, elle obtiendra une connexion valide, sur la nouvelle base.

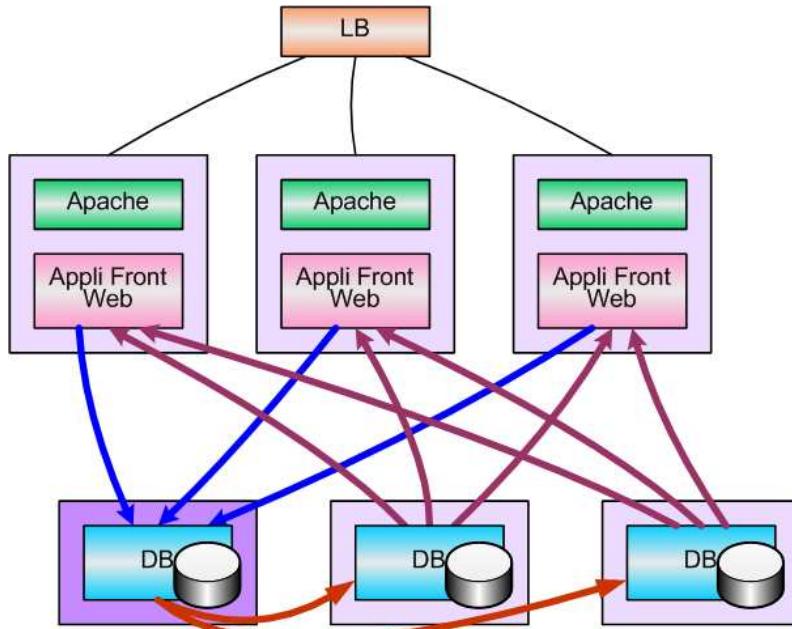
Spécialisation en écriture / lecture

Mais le succès ne se dément pas, et bien que nous ayons ajouté de nombreux frontaux, ce qui devait arriver arrive : nous avons maintenant 8 frontaux et le serveur de base de données ne tient plus la charge. On a gonflé la configuration au max, on est passé sur un bi-pro, puis un quadri-pro, rajouté des disques, de la mémoire, mais ça ne suffit plus...

Pourra-t-on aller plus loin ?

Nous avons vu que beaucoup de plateformes web ont un taux d'écritures base bien plus faible que de lectures. Dans ce cas, il est intéressant, de répartir les lectures sur différents serveurs, tout en concentrant les écritures sur un serveur unique.

Architectures Hautes-Performances

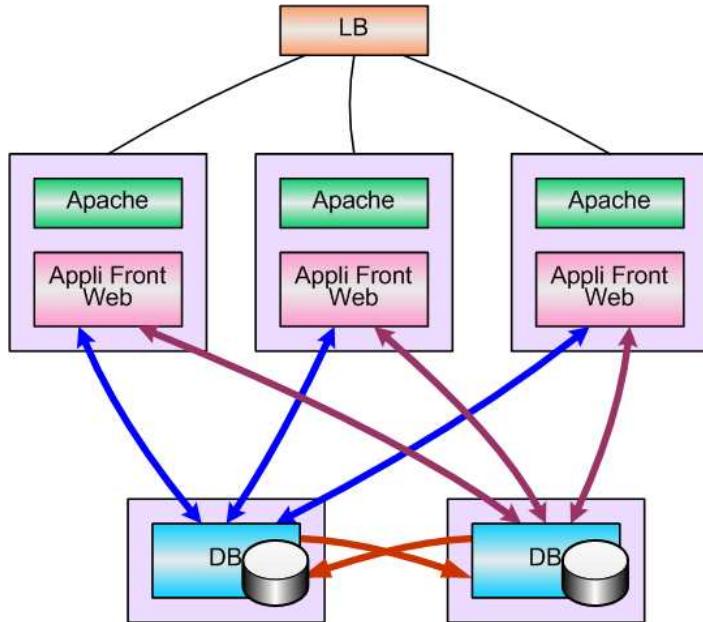


Ici, les frontaux répartissent les lectures sur les différentes bases, mais les écritures ne sont adressées que sur la base de gauche. Les transactions sont répliquées depuis celle-ci, vers toutes les autres bases.

Il y a quelques inconvénients à cette configuration :

- Elle n'est pas transparente pour les applications, qui doivent distinguer explicitement leurs écritures et leurs lectures.
- Il y a un petit de propagation avant qu'une écriture ne soit visible sur les autres bases, ce qui peut engendrer des incohérences transitoires.

Bases multiples en réPLICATION croisée



Dans l'architecture précédente, nous avons deux bases de données, qui se partagent la charge, certainement par l'intermédiaire d'un double pool de connexions. Ici les deux bases sont actives, il n'y a pas une base principale et une base de secours.

C'est l'architecture que nous avons mise en place pour un des plus grands sites français autour de 2004, et qui a parfaitement répondu à des charges de plus de 150 000 visites par jour.

Néanmoins, elle présente quelques inconvénients encore :

- Elle a des impacts fonctionnels, liés au petit décalage entre les serveurs ;
- Elle a des impacts également dans le modèle des données, l'utilisation des séquences (valeur auto-incrémentée) par exemple.
- Elle peut faire apparaître des incohérences, dans le cas où deux mises à jour de la même entité sont demandées de part et d'autre.
- Elle n'est pas transparente pour les applications, qui doivent se préoccuper du bon choix de serveur et du passage en secours ;
- La tolérance aux pannes de l'étage base de données est imparfaite. On a intégré deux serveurs parce qu'il en fallait deux, mais si l'un tombe en panne, le serveur restant est insuffisant pour satisfaire le besoin. Il faut donc prévoir également un secours tiède de ces serveurs.

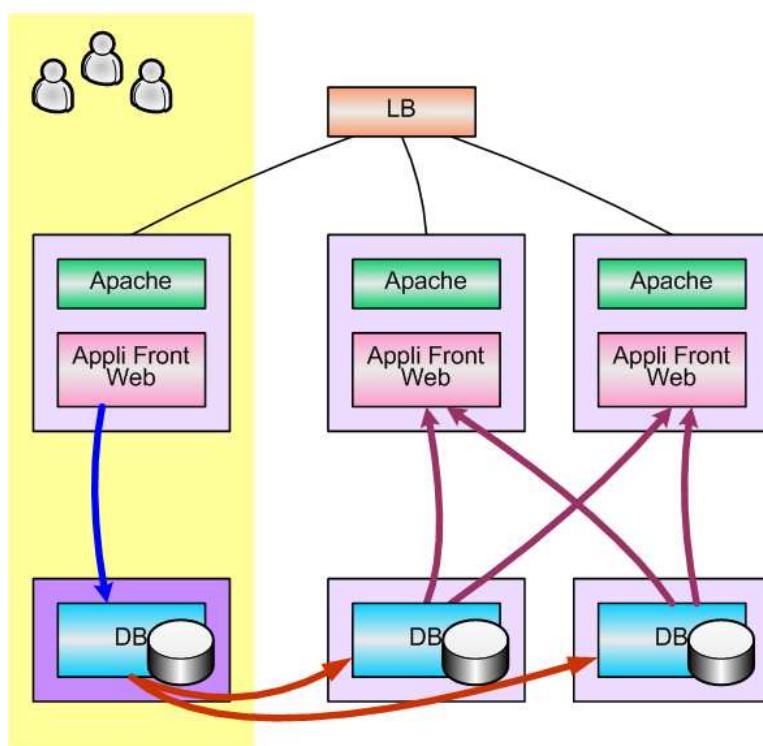
- Et finalement, elle n'est pas extensible à l'infini. Même s'il n'est pas impossible de passer à trois serveurs, les mécanismes et flux de réPLICATION croissent selon le carré du nombre de serveurs.

Serveur dédié à la contribution

Il est assez courant que les mises à jour soient réservées à des internautes identifiés. Dans ce cas, la règle d'affectation est assez simple : les internautes identifiés sont gérés sur la base maître.

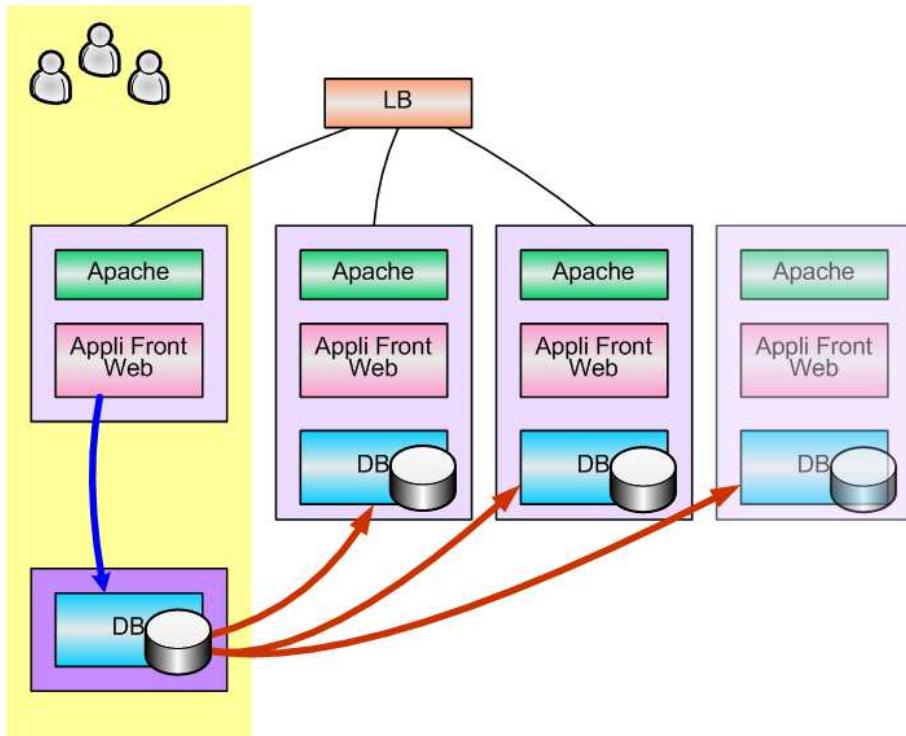
Un cas particulier usuel est celui d'un site utilisant un outil de gestion de contenus. On distingue alors des contributeurs et de simples lecteurs. Les contributeurs agissent alors sur la base maître, tandis que les internautes lecteurs accèdent à l'une des bases répliquées.

Dans certains cas, on trouvera même plus simple de dédier un couple frontal / base de données pour les contributions :



Bien sûr, cela pose à nouveau le problème de la tolérance aux pannes, tout particulièrement pour la contribution. Mais on peut admettre parfois que la disponibilité de la contribution est moins critique que celle de la lecture.

A vrai dire, dans le schéma précédent, la flexibilité de l'aiguillage entre frontaux et bases, pour la partie *lecture* n'est pas fondamentale. Pour le même nombre de serveurs, on pourra envisager de réunir à nouveau étages frontal et base de données.



En rassemblant les étages, on obtient une répartition de ressources transparente entre frontal et SGBD.

L'inconvénient majeur de ce type d'architecture, c'est la prise en compte des « *User Generated Content* » (*UGC*) de toutes formes : commentaires, notation, participation à des forums, etc. Les *UGC* sont une tendance majeure du web moderne, et il n'est plus possible de distinguer aussi clairement *contribution* et *publication*.

Partitionnement des données

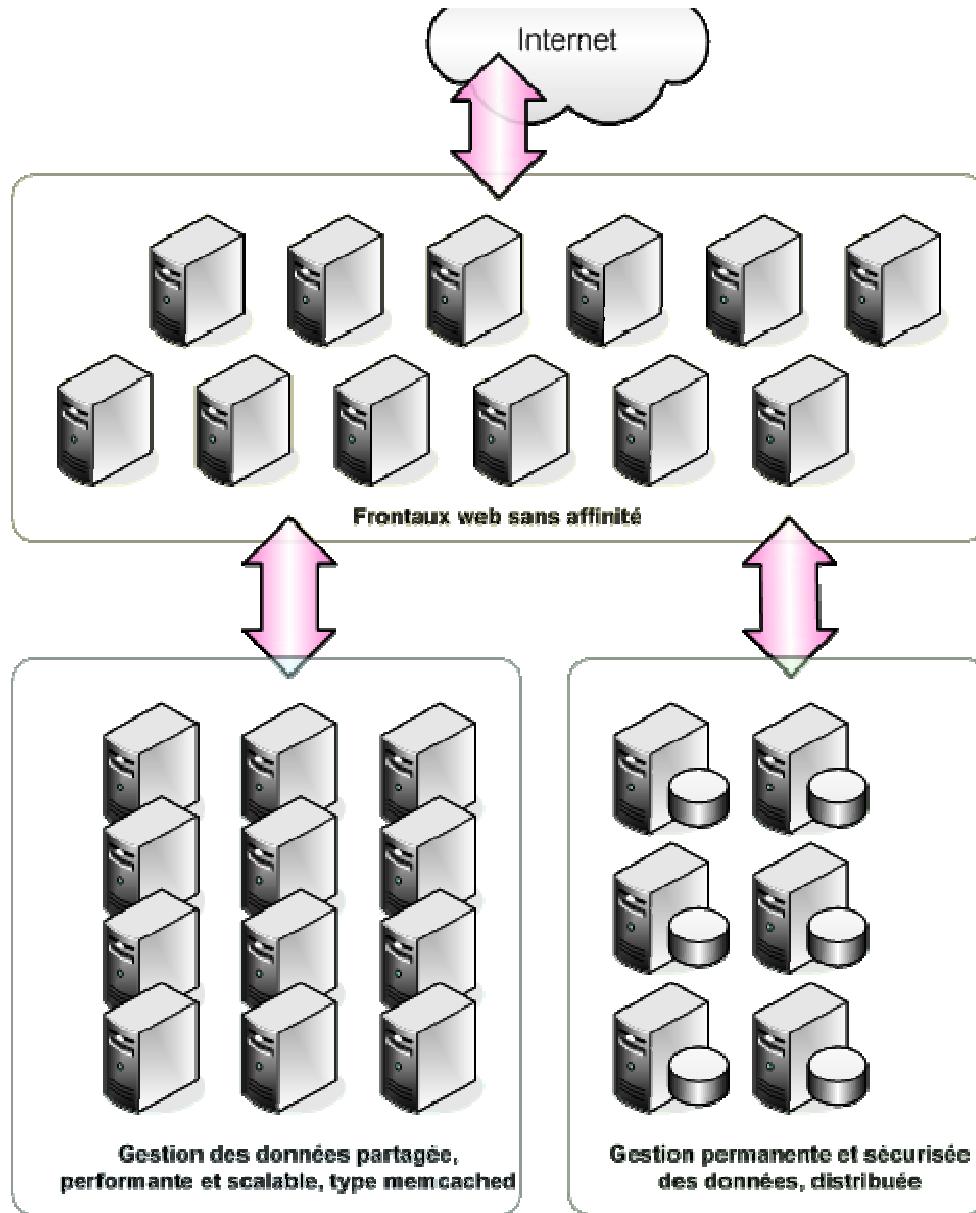
Enfin, le *nec plus ultra* de l'extensibilité sera atteint avec le partitionnement des données, que l'on a évoqué déjà. Comme on l'a dit, le partitionnement n'est pas compatible avec toutes les typologies d'application. On l'évoquera davantage dans notre *cas d'école* suivant, traitant d'une plateforme de blog.

Architecture type Facebook

Nous avons évoqué plus haut l'usage généralisé de memcached sur la plateforme Facebook, et nous avons vu également les principes d'une architecture urbanisée, dans laquelle chaque domaine fonctionnel échange avec les autres en invoquant des services, et chaque domaine est totalement extensible, de manière indépendante.

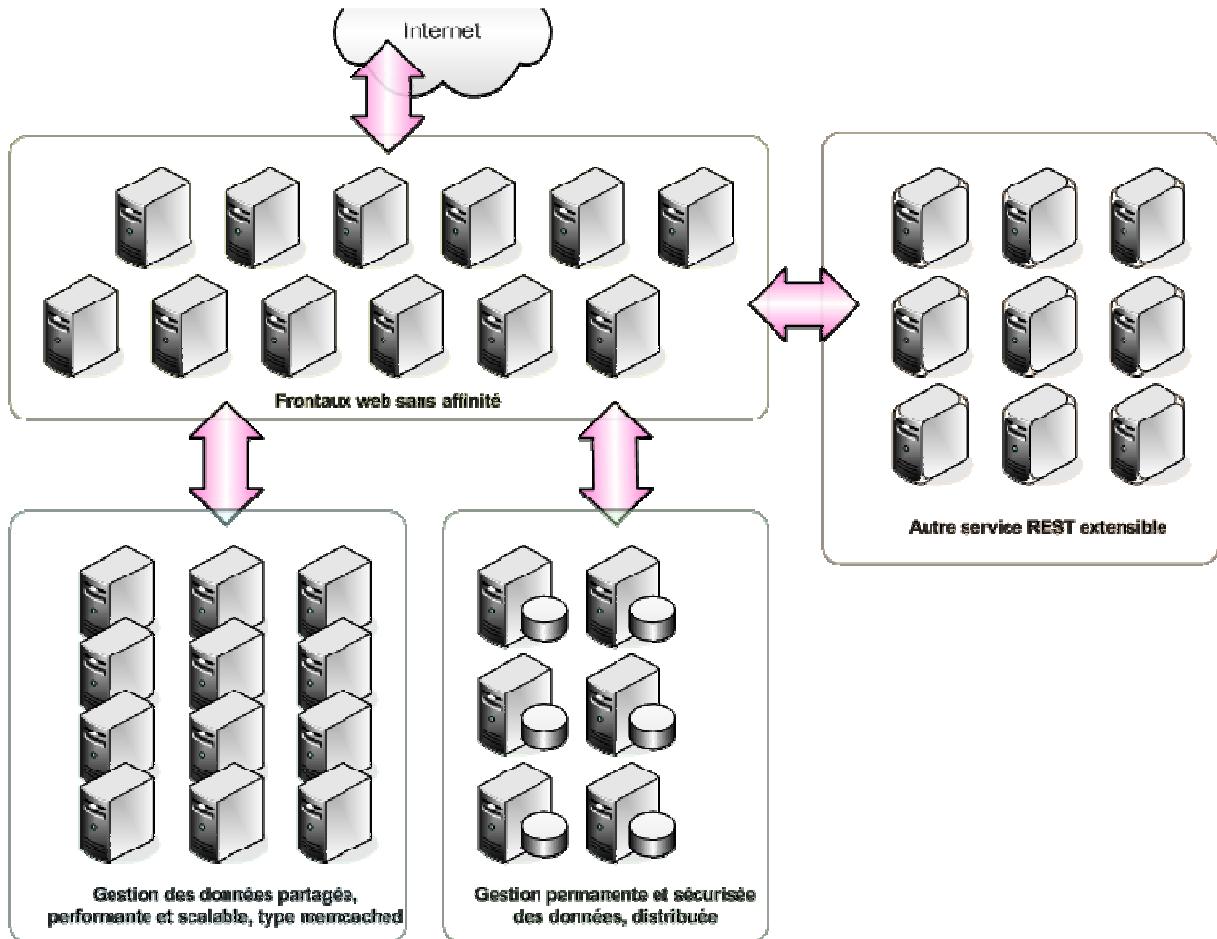
Architectures Hautes-Performances

C'est ce que l'on peut représenter typiquement de la manière suivante :



En référence au schéma de la page 143, le sous-système bleu représente le gestionnaire de données principal, entièrement géré en mémoire et entièrement distribué, au moyen de l'outil Memcached. C'est-à-dire que les applications viennent accéder la totalité de leurs données en invoquant le service de ce sous-système. Occasionnellement, une donnée peut ne pas y figurer, et dans ce cas elle est obtenue sur le sous-système vert, qui met en œuvre une gestion sécurisée et permanente des données, que ce soit au moyen d'une base de données partitionnée sur différentes instances, ou bien au moyen d'un système de gestion de fichiers extensible tel que MogileFS ou HDFS, comme mentionné page 124.

Sur ce même principe d'architecture, on peut aisément ajouter d'autres sous-systèmes accédés par web services, ainsi :



On veillera à ce que chacun de ces services soit accessible autant que possible en mode REST, sans session, et éventuellement de manière cachable.

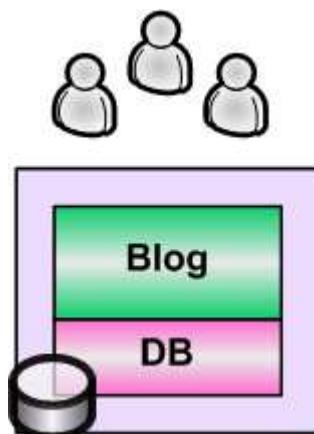
Une plateforme de blogs de très forte capacité

Le problème posé

Considérons un autre cas d'école : nous avons à mettre en place une plateforme de blog de très forte capacité, capable d'accueillir plus d'un million de blogueurs. En fait, nous disons « plus d'un million », mais la véritable exigence est plutôt « un nombre illimité de blogueurs », que ce soit un million, 20 millions ou 200 millions.

Les internautes peuvent s'inscrire pour devenir blogueurs, et une fois passées les quelques formalités d'inscription, ils peuvent commencer leur blog. Chaque blogueur poste des articles sur son blog, qui sont mis en ligne a priori avec ou sans modération. Les autres internautes, lisant un article, peuvent eux-mêmes y poster des commentaires, qui seront ou non modérés par l'auteur du blog.

On ne s'attardera pas ici sur les fonctionnalités cosmétiques ou sans impact sur l'architecture. En revanche, il faut bien analyser les besoins en matière d'administration. Un administrateur de la plateforme doit avoir une vision globale des blogs. Autant du point de vue des fonctions liées à la gestion des blogs que du point de vue des statistiques (nombre de blogueurs inscrits, nombre d'inscriptions du jour, de la semaine, du mois, nombre d'articles nouveaux écrits, de commentaires postés, etc.). L'administrateur doit pouvoir retrouver n'importe quel blogueur, sur différents critères de recherche, et modifier ou bien supprimer son compte. L'administrateur doit avoir des interfaces de modération centralisées, qui lui présentent par exemple les derniers articles, ou bien les derniers articles comportant tels et tels mots à surveiller. Et cette tâche de modération doit également pouvoir être répartie entre différents intervenants.



Quelles options d'architecture ?

Si le problème était d'entrée de jeu posé avec des limites de volumétrie définitives, on pourrait sauter sur l'architecture classique « deux tiers », incluant quelques frontaux et une base de données. Il est certain que ce choix impliquerait une limite sur la capacité globale de la plateforme. Si 100 000 blogueurs et lecteurs sont connectés et postent leurs articles, à raison disons de une contribution toutes les 5 minutes, alors nous avons 300 insertions par seconde dans la base de données, sans compter bien sûr toutes les mises à jours périphériques. On aura beaucoup de mal à soutenir ce rythme, même sur une configuration matérielle haut de gamme. Et quand bien même on y parviendrait, on ne tiendra pas un million de blogueurs en ligne.

On peut dire : peu importe où se situera exactement la limite, l'essentiel est qu'il y aura une limite, et c'est ce que ne voulons pas. Car lorsque la limite sera atteinte, ce sera une limite « dure », très difficile à franchir, nécessitant une révision d'ensemble de l'architecture.

Un problème partitionnable

Un blogueur n'agit que dans le périmètre de son blog. En première analyse, le problème posé est naturellement partitionnable. Si l'on sait mettre en place une plateforme relativement simple pour accueillir disons 10 000 blogueurs, alors qu'est-ce qui nous empêchera d'aligner autant de ces plateformes qu'il faudra, 10 pour 100 000 blogueurs, 100 pour un million ?

Faisons la liste de ce qui nous poserait problème dans cette simple *juxtaposition de petites plateformes*.

- La répartition des internautes blogueurs : faudrait-il dire à certains d'accéder à www-01.mesblogs.fr, d'autres à www-02.mesblogs.fr, d'autres finalement à www-10.mesblogs.fr ? Ce n'est pas très élégant assurément.
- La gestion administrative des blogueurs : lorsqu'un administrateur recherche un blogueur, devra-t-il adresser une requête à chacune des N plateformes ?
- De même pour la gestion de la modération : les modérateurs devront-ils se connecter à chacune des N plateformes ?
- A ces problèmes d'organisation on pourrait ajouter des problèmes de coût. On sait que plus la tolérance aux pannes est gérée à grande échelle, moins elle est coûteuse. Faudra-t-il ici que chacune des petites plateformes élémentaires assure sa haute disponibilité de manière totalement autonome ?

Malgré ces questions à traiter, la remarque initiale reste prépondérante : il y a dans cette problématique, une capacité assez naturelle à diviser le problème, diviser pour régner. Ce sera donc bien le principe de base de notre architecture. Les problèmes cités ne sont pas suffisants pour remettre en question ce principe, nous leur trouverons des solutions.

Répartition arbitraire

Nous avons expliqué (cf. « Quelle logique de répartition ? », page 98), que dans un partitionnement, la répartition ne doit pas être basée sur des règles fonctionnelles. Elle doit être arbitraire, ou plutôt, fondée uniquement sur des critères techniques : un blogueur nouveau est affecté à tel serveur *parce que ce serveur a de la capacité disponible*.

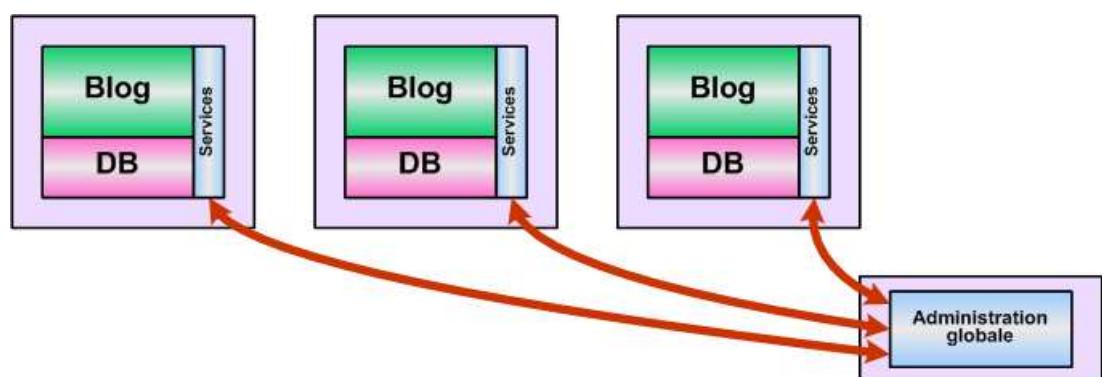
Cela implique, comme on l'a vu, de tenir à jour une table d'affectation, qui sera utilisée pour la répartition des internautes.

Cette logique de remplissage des serveurs permet d'étendre la plateforme au fur et à mesure du besoin. Lorsque les serveurs en place manquent de capacité, on en ajoute quelques-uns. Des blogs peuvent aussi être supprimés, de sorte que de la capacité peut apparaître sur des serveurs anciens, qui sera utilisée par de nouvelles affectations.

Fonctions centrales via webservices

Pour les fonctions centrales, en particulier d'administration, une première voie consiste à s'appuyer une application d'administration globale échangeant avec chacune des plateformes de blog indépendantes en invoquant des webservices. Ce peut être aussi bien pour collecter de l'information de manière centralisée que pour adresser des commandes.

Ce qui est représenté ci-après :



Si notre application de blogs est moderne et bien conçue, elle aura déjà implémenté des interfaces REST pour chacune de ses fonctionnalités. Sinon, dans le cas d'un progiciel en général, mettre en place ces webservices pourrait être difficile, mais pour un produit open source, on y parviendra généralement.

Malgré tout, dans ce mode, l'administration ne sait pas *quel blog est sur quel serveur*, de sorte qu'elle doit fréquemment adresser sa requête en parallèle à la totalité des serveurs, alors qu'un seul est concerné.

Lorsque le nombre de ces serveurs augmente, ces invocations pèsent de plus en plus lourd, et deviennent un réel frein à l'extensibilité.

Fonctions centrales via datawarehouse

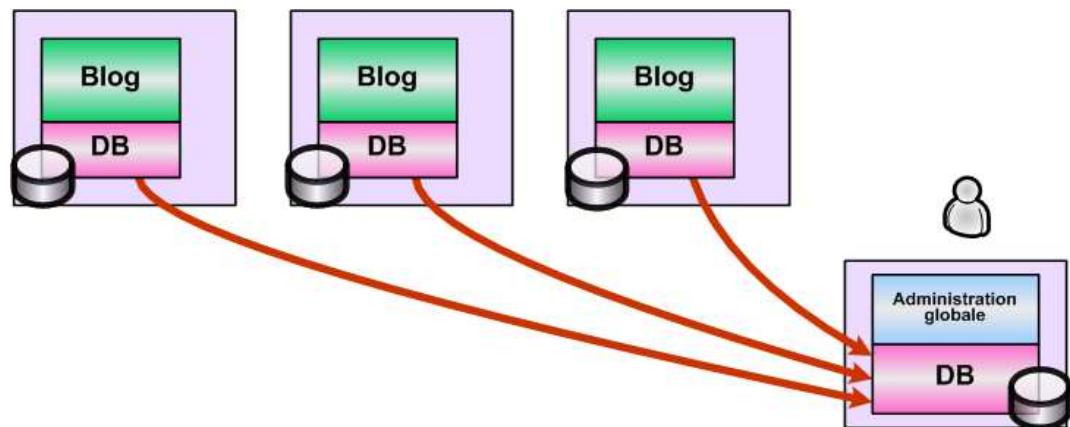
On a dit déjà que le datawarehouse est le complément naturel du partitionnement.

Ici, il s'agit de constituer une base centrale consolidant une partie des données portées par chacun des serveurs, les données utiles à l'administration.

Cette base consolidée est utilisée :

- Pour faire des recherches transverses, multi-critères, sans solliciter tous les serveurs.
- Pour construire la table d'affectation, qui permettra ensuite d'aiguiller les requêtes.
- Pour disposer de statistiques globales.

Ce que l'on représente comme suit :



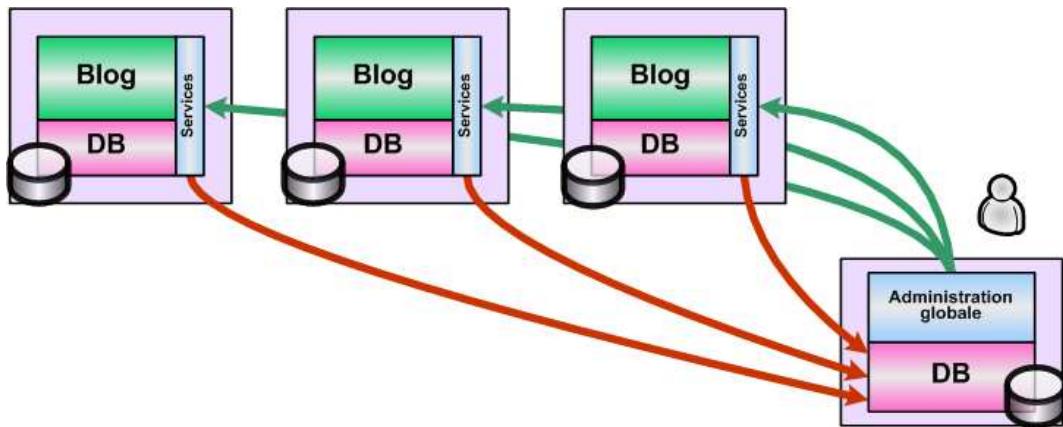
Les outils de cette consolidation peuvent varier. On a dit déjà que la réPLICATION de base de données devait être utilisée seulement *entre systèmes homologues*, ce qui n'est pas le cas ici. On préférera donc appuyer cette collecte sur un middleware, que ce soit en mode *push* sur une *message queue*, ou bien en mode *pull*, par interrogation d'un webservice spécifique.

Webservices + datawarehouse

En fait, nous avons besoin de combiner les deux approches :

- La base consolidée, afin de ne pas invoquer tous les serveurs pour la moindre interrogation, et pour tenir à jour la table d'allocation ;
- Les webservices afin d'agir sur les serveurs.

Ce que l'on représentera comme ceci :



On a ici un dispositif extrêmement extensible, qui s'accommadera d'une centaine de serveurs si besoin.

Répartition de charge

Dans cette architecture partitionnée, chaque entité, chaque sous-plateforme est à la fois une entité de stockage et une entité de traitement. En tant qu'entité de stockage, elle a la charge d'un certain nombre de blogueurs et de toutes les informations associées : leur personnalisation, leurs articles, les commentaires, etc. En tant qu'entité de traitement, elle a la charge de gérer toutes les interactions avec ces blogueurs, de présenter les articles en lecture aux internautes, et d'accepter leurs contributions. Ce double rôle, stockage et traitement, implique une bonne analyse du dimensionnement selon ces deux besoins.

Dans cette architecture, la répartition des internautes ne se gère pas selon la charge, mais selon la cible. Imaginons que le blog du blogueur Jean-Paul soit publié à l'adresse <http://www.mesblogs.fr/blog-de-jean-paul/>. On mettra en place une répartition de niveau 7 qui, après analyse de l'URL et consultation de la table d'affectation, dirigera toutes les requêtes du type /blog-de-jean-paul/* vers le serveur approprié.

Pas d'autre axe d'extensibilité

Une fois qu'on a retenu le principe de partitionnement, est-il utile de le combiner avec d'autres axes d'extensibilité ?

On pourrait en effet augmenter la capacité de chacune des plateformes élémentaires, que ce soit verticalement, en séparant application et base, ou horizontalement, en disposant plusieurs frontaux.

Du point de vue de l'extensibilité, ce n'est pas nécessaire : le partitionnement nous apporte toute l'extensibilité désirée, et il sera plus simple d'avoir 100 petits serveurs de blog autonomes, plutôt que 20 plateformes de 5 serveurs.

Mais il reste la question du secours.

Secours

Nous avions signalé déjà que le partitionnement rendait le secours plus difficile.

Si nous alignons 100 serveurs de blog autonomes, comment offrir de la haute disponibilité pour chacun d'eux ? En le dupliquant ?

Clairement, doubler tous nos serveurs serait extrêmement coûteux.

Il nous reste deux voies :

- Soit construire, comme on l'a évoqué ci-avant, 20 petites plateformes de 5 serveurs (ou une autre combinaison), chacune disposant des solutions classiques de haute disponibilité, déjà recensées.
- Soit mettre en œuvre un secours mutualisé pour les 100 serveurs : on dispose d'un petit lot de serveurs de secours, en *spare*, qui peuvent servir à remplacer n'importe lequel des serveurs défaillants. Si l'infrastructure est totalement virtualisée, on pourra restaurer l'ensemble de la VM au dernier point de sauvegarde, en quelques secondes.

SAN

Nous avons dit plus haut que le SAN n'était en général pas la solution pour construire une plateforme de très haute capacité. Néanmoins, ici, ce pourrait être une voie intéressante.

En disposant d'un système de stockage qui soit à la fois :

- De très haute disponibilité, car intrinsèquement redondant
- Et qui puisse être associé à n'importe quel serveur

On peut gérer un secours mutualisé pour l'ensemble de la plateforme. En cas de panne d'un serveur, on alloue un serveur parmi le pool de spare, et on lui associe le système disque du serveur qu'il remplace.

Sport 24 et 01 Informatique

Le problème posé

Nous étudions ici deux sites différents, réalisés par Smile, qui mettent en œuvre des principes d'architecture semblables.

Le contexte est le suivant :

- Ces sites ont une large part de gestion de contenus : des journalistes écrivent des articles, qui sont mis en forme au moyen de templates, puis mis en ligne. Il existe d'excellents outils open source prêts à l'emploi qui couvrent toutes les fonctionnalités requises à cet égard.
- Ces sites ont une forte audience. Dans le cas du site Sport24, on est à 5 millions de visites sur le mois de septembre 2008. Mais plus important encore : le site observe de très forts pics de trafic à l'occasion des événements sportifs.
- Ces sites ont une part de contenus personnalisés, c'est-à-dire qui changent selon les préférences de l'internaute.
- Ces sites n'intègrent pas que de la gestion de contenu, mais de nombreuses fonctionnalités périphériques relevant d'une dimension communautaire : blogs, forums, chat, etc., ainsi que des commentaires utilisateurs, qui plus largement sont dans la catégorie des « UGC », « *User Generated Content* », contenus produits par les utilisateurs.
- Enfin, dans le cas de Sport24, on a également une fonctionnalité de *match en temps réel*, où l'historique du match est rafraîchi de manière très rapide.



Axes de solution

Pour de tels sites, le socle est obligatoirement un outil de gestion de contenus, couvrant pour au moins les 2/3 les fonctionnalités attendues.

Il existe une offre très diversifiée de bons outils CMS, « *Content Management System* », et l'open source domine clairement ce marché. On ne va pas détailler ici l'analyse du choix de solution, sur la base des qualités ou contraintes propres de chaque produit. Pour ces deux sites, le choix s'est porté sur le produit eZ Publish, un excellent CMS en environnement PHP.

eZ Publish offre une configuration dite « en cluster », qui a les caractéristiques suivantes :

- Les contextes de session sont partagés en base de données, ce qui permet une répartition de charge sans affinité de serveur.
- Les fichiers média sont également conservés en base de données, donc peuvent être partagés par N frontaux.

eZ Publish présente des performances assez bonnes, mais en présence de pages personnalisées, le cache ne peut pas être très efficace, et l'on peut descendre jusqu'à une dizaine de pages par processeur seulement.

Dans le cas de Sport24, on veut tenir des charges en crête de quelques milliers de pages servies par seconde. On voit qu'en configuration

ordinaire, il faudrait plusieurs centaines de serveurs pour y parvenir ! Il va donc falloir travailler un peu l'architecture.

Agrégation côté client

Pour gérer la personnalisation, ces sites utilisent le mécanisme d'agrégation côté client, c'est-à-dire que certains morceaux de la page sont insérés dans la page côté navigateur par un peu de javascript.

Il s'agit des blocs personnalisés, qui ne peuvent être gérés en cache. Ainsi la plus grande partie de la page bénéficie du cache – comme on le verra plus loin – et seuls les blocs personnalisés sont constitués dynamiquement.

De même, les matchs en temps réel sont rafraîchis à une fréquence élevée en javascript, sans recharger la page.

Agrégation côté serveur

Pour l'agrégation côté serveur, nous avons utilisé le module *Server-Side Include (SSI)*, de Apache, qui précisément permet de constituer des pages en insérant différents fragments aux positions indiquées. C'est finalement une sorte de dispositif de templating, rudimentaire.

Cache Squid en frontal

Bien entendu, nous avons positionné un serveur de cache, ici Squid, en frontal du serveur Apache. Pratiquement toute plateforme web doit avoir un serveur de cache en frontal.

Un unique Squid permet de servir plus de 2000 pages par seconde.

Comme on l'a vu plus haut, la gestion des durées de vie en cache (TTL) peut soit être configurée globalement, soit plutôt spécifiée par le serveur lui-même.

On compte en particulier sur le Squid pour servir tous les fichiers de ressources autres que Html, qui varient peu.

Génération de pages statiques

C'est l'originalité de ces deux plateformes : le CMS ne sert pas directement les pages aux visiteurs, on lui fait générer des pages Html statiques, des fichiers, qui sont ensuite servis par Apache.

C'est un moyen de bénéficier à la fois de toutes les fonctionnalités d'un CMS – et elles sont très nombreuses – et de l'extraordinaire robustesse et performance du statique.

Les journalistes contributeurs utilisent le back-office du CMS pour créer et modifier leurs articles, et gérer les médias.

De manière assez fréquente, un petit script provoque la régénération des pages, ou des fragments HTML.

Obtenir des pages statiques, des fichiers HTML, à partir d'un CMS est assez facile : il suffit de se faire passer pour un navigateur web, et demander la page en HTTP, puis écrire sur disque le fichier ainsi obtenu. Il faut veiller bien sûr à ce que les liens hypertexte restent corrects dans un contexte statique.

La principale difficulté est ailleurs : on ne peut pas régénérer, par une sorte d'aspiration, l'ensemble du site à haute fréquence. *Il faut donc déterminer qu'est-ce qui a changé.* Ce n'est pas chose facile, car dans un CMS sophistiqué, il n'y a pas correspondance entre un contenu et une page : le journaliste modifie un article, mais cet article peut apparaître dans différentes pages, sous différents formats. On peut même modifier un pied de page par exemple, qui impactera la totalité des pages.

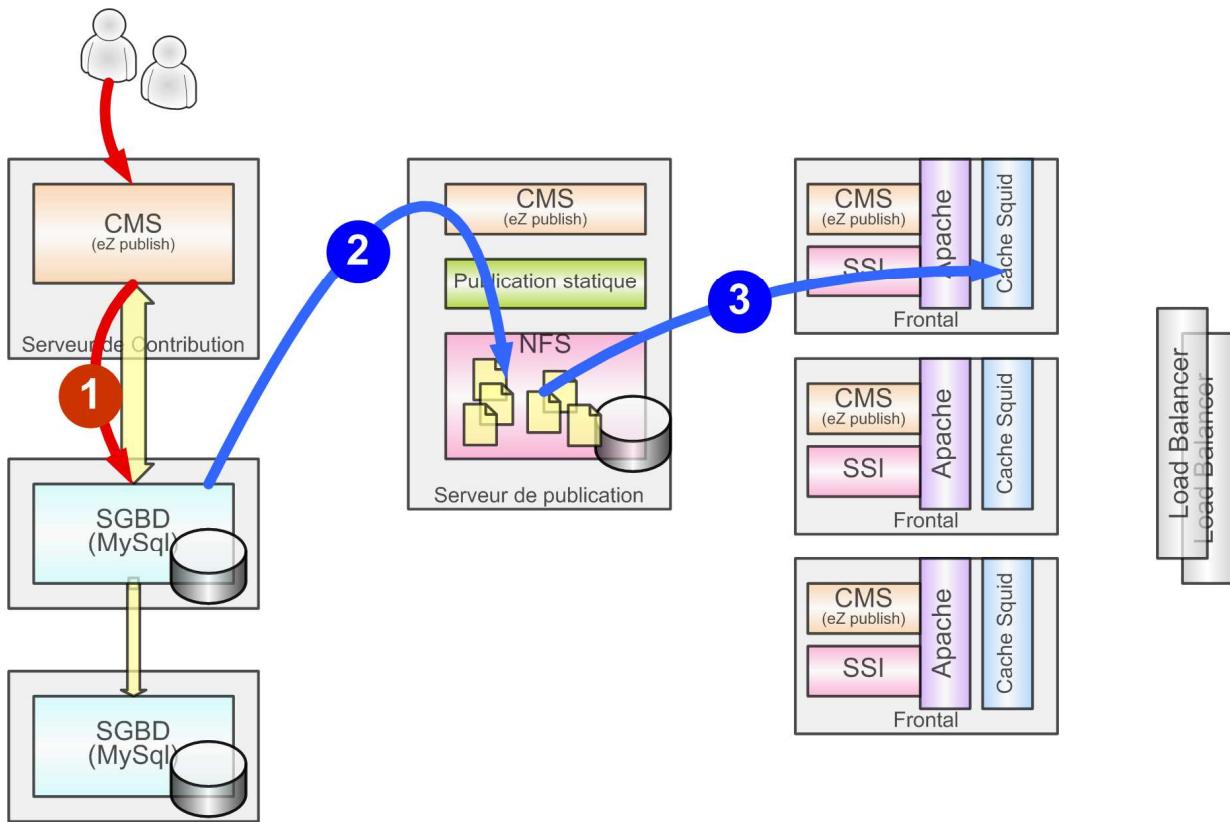
Nous avons donc réalisé un job qui analyse les impacts des derniers changements, afin de ne régénérer que ce qui est requis.

Diffusion des pages vers les frontaux

Une fois générés les fichiers statiques, fragments de pages HTML, on peut les mettre à disposition des frontaux soit par un partage NFS, soit par une réPLICATION RSync. Pour Sport24, nous avons utilisé un NFS, pour O1Informatique, nous avons préféré la réPLICATION.

Schéma d'ensemble

Architectures Hautes-Performances

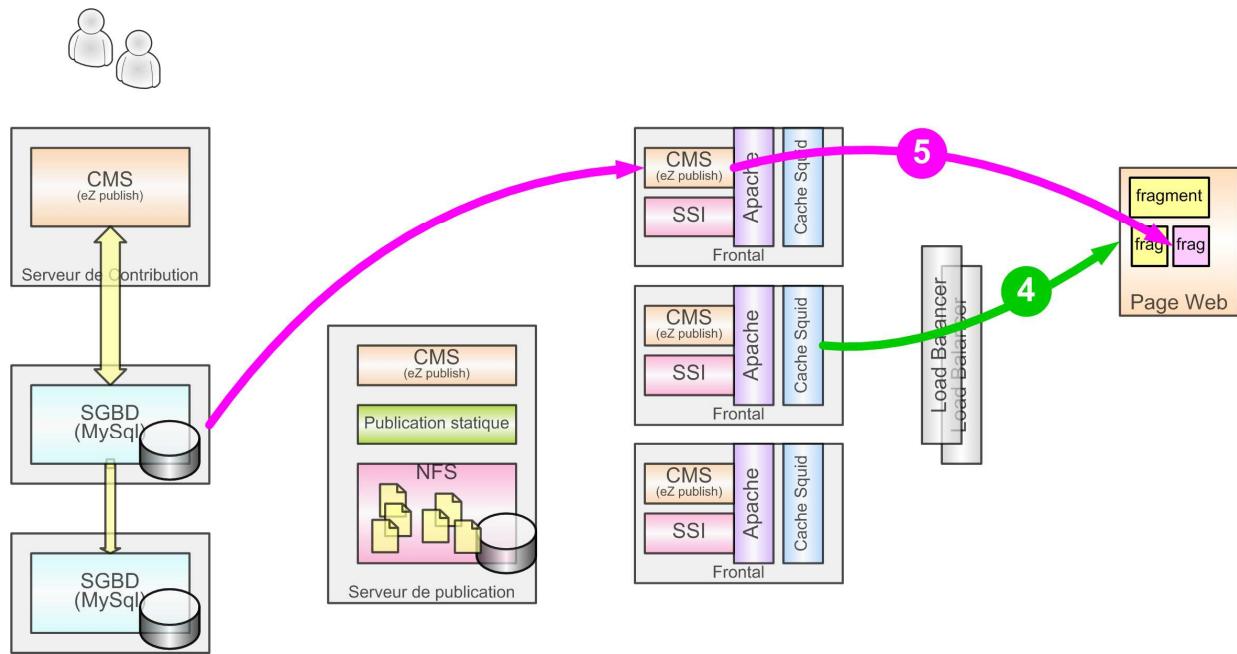


Sur le schéma précédent, on distingue :

- A gauche, la plateforme de contribution destinée aux journalistes
- Au centre, le serveur de publication, sur lequel un job spécifique vient analyser les changements et aspirer les pages ou fragments Html.
- A droite, la plateforme de publication, sur laquelle on distingue :
 - Le cache Squid en frontal
 - Apache bien sûr
 - Le module SSI, qui va réaliser l'agrégation des fragments Html obtenus depuis le serveur NFS.

On a fait figurer également un CMS en production, qui est en charge des blocs personnalisés, insérés dans les pages du côté navigateur.

Architectures Hautes-Performances

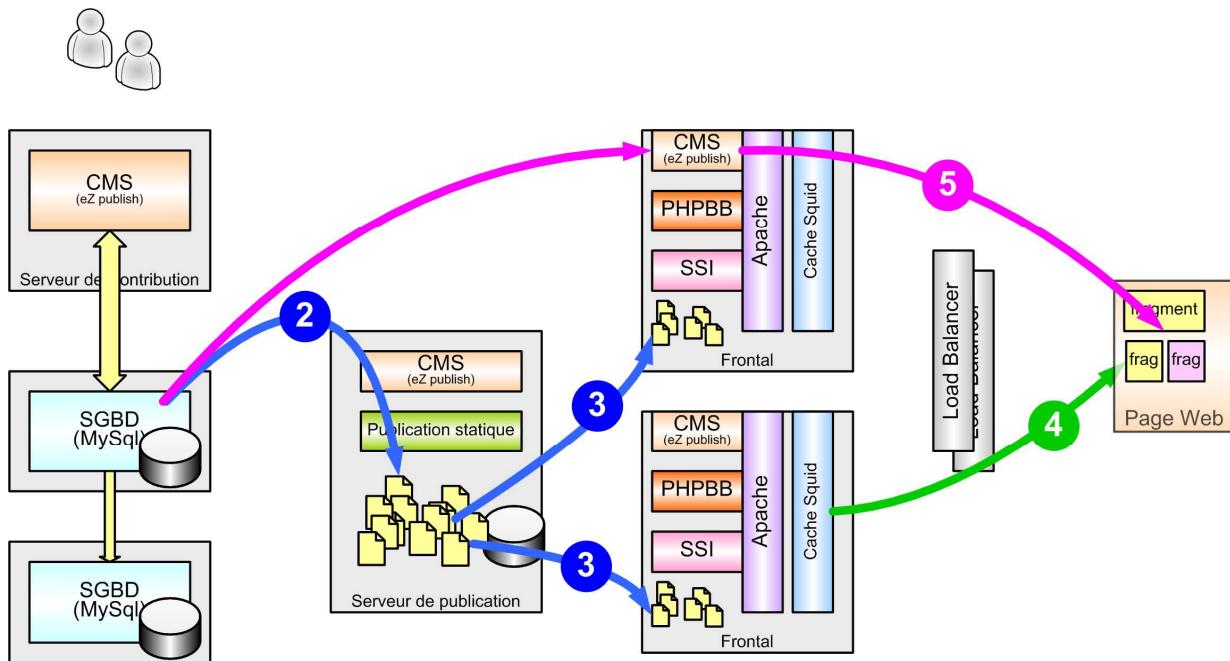


Sur la figure précédente, on distingue :

- Tout à fait à droite, l'agrégation côté client, insérant un bloc de contenu au sein d'une page, en invoquant le CMS, qui accède à la même base que les serveurs de contribution.
- Les contenus média servis directement depuis le Squid.

Enfin, la figure suivante représente la variante 01 Informatique

Architectures Hautes-Performances



Outre la réplication RSync qui remplace le partage NFS pour un résultat équivalent, on a aussi intégré ici la plateforme de Forum PHP-BB, qui elle est en direct.

Woozweb

Nous avons évoqué Woozweb en tant qu'outil de monitoring et d'observatoire du web, mais il peut être aussi un cas d'école pour étudier l'extensibilité d'une plateforme.

Car Woozweb est une plateforme que l'on a voulu extensible à l'infini.

La problématique Woozweb

En quelques mots, les grandes fonctionnalités de Woozweb :

- Woozweb surveille des « ressources » du web. Une « ressource » est caractérisée par son URI.
- Des serveurs spécifiques, appelés « Sondes », adressent des requêtes afin de tester les ressources.
- On distingue deux types de sondes : des sondes *haute-fréquence (HF)*, qui testent chaque ressource toutes les 15 minutes, et des sondes *basses-fréquence (LF)*, qui testent chaque ressource toutes les semaines.

- Les sondes HF ne font que charger la page Html correspondant à l'URI, et vérifier la conformité de la page, le code retour Http, et le temps de réponse observé.
- Les sondes LF, elles, effectuent un chargement complet de la page, et de toutes ses ressources, au moyen d'un vrai navigateur, exécutant aussi bien le javascript que le Flash, ou tous autres objets embarqués. Elles relèvent donc une information différente, qui va au delà de la seule disponibilité ou qualité de service.
- Toutes les informations relevées par les sondes sont conservées et mises à disposition au travers d'interfaces web.

Partitionnement et consolidation

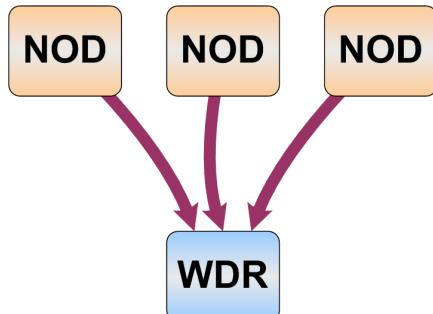
Woozweb a recours au principe déjà évoqué, combinant partitionnement et consolidation :

- On découpe la plateforme Woozweb en sous-systèmes que l'on appelle *nœuds*, ou « nodes », parfois abrégé en « NOD ». Un NOD est un sous-système qui est en charge de N ressources. Le NOD est constitué de plusieurs serveurs, nous y reviendrons plus loin. Le NOD assure à la fois le monitoring LF et HF des ressources dont il a la charge, le stockage et la restitution de ces données.
- Nous avons besoin de requêtes transverses, pour deux raisons. D'une part Woozweb autorise des recherches multi-critères sur l'ensemble des ressources. Indépendamment donc, de quel NOD en a la charge. D'autre part Woozweb met à disposition des statistiques transverses, portant sur la totalité des ressources. Par exemple quelle est la part des sites qui utilisent du PHP. Comme on l'a vu plus haut, les fonctionnalités de ce type sont mieux mises en œuvre sur un datawarehouse consolidant les données.

Dans l'architecture Woozweb, le datawarehouse est appelé « WDR », *Woozweb Data Repository*.

Rappelons que le datawarehouse n'est pas la somme de toutes les données de tous les NODs : il ne consolide qu'un petit échantillon de données, celles utiles aux fonctionnalités transverses.

On a donc le schéma global classique :



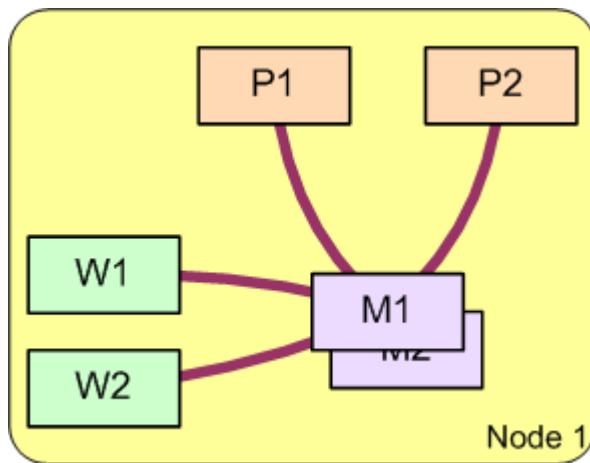
Information consolidée

Architecture de chaque « NOD »

Chaque NOD assure à la fois les fonctionnalités :

- De monitoring, par les sondes LF et HF adressant des requêtes aux ressources sous surveillance.
- De stockage de l'information de monitoring recueillie, qui peut être volumineuse.
- D'interface de consultation, elle peut accueillir un visiteur et gérer les interfaces web de sa session.

Chaque sous-système NOD a l'architecture suivante :



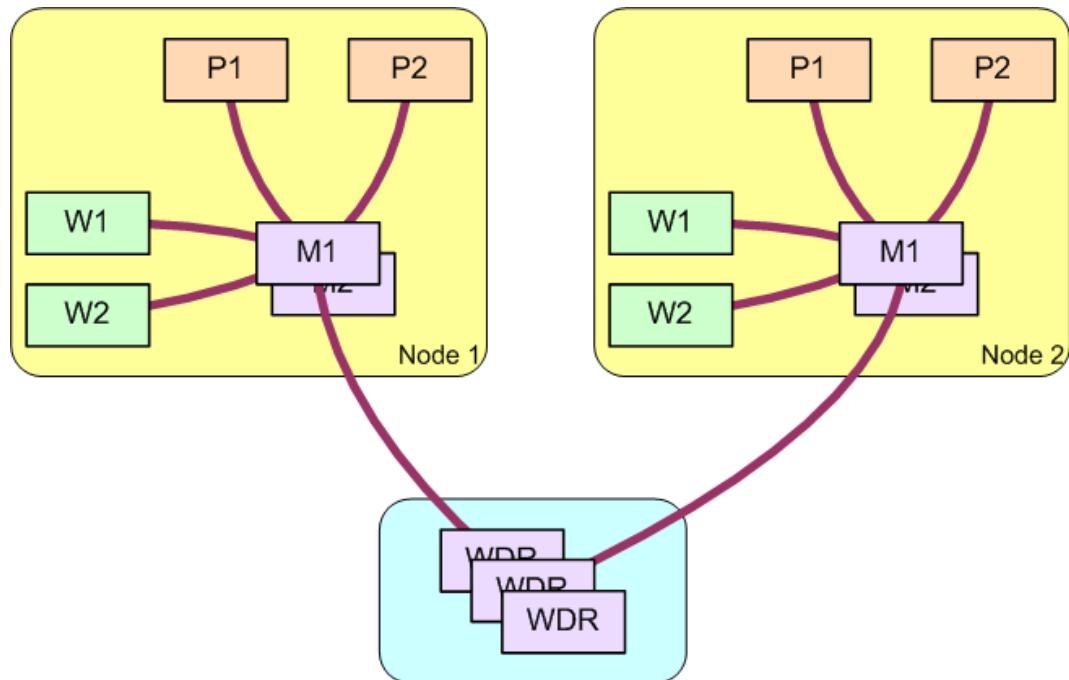
Où M1 est un serveur dit « Manager », noté « MGR », qui porte la base de données du NOD. M2 est le secours de M1, en réplication.

P1 et P2 sont des sondes, que ce soit LF ou HF. Un même MGR peut être configuré pour gérer plusieurs sondes, selon la capacité unitaire d'une sonde.

W1 et W2 sont des frontaux web associés au NOD, utilisés en répartition de charge, partageant la base de MGR.

Architecture globale

On peut donc représenter l'architecture globale, combinant un certain nombre de NODs, et un WDR, comme suit :



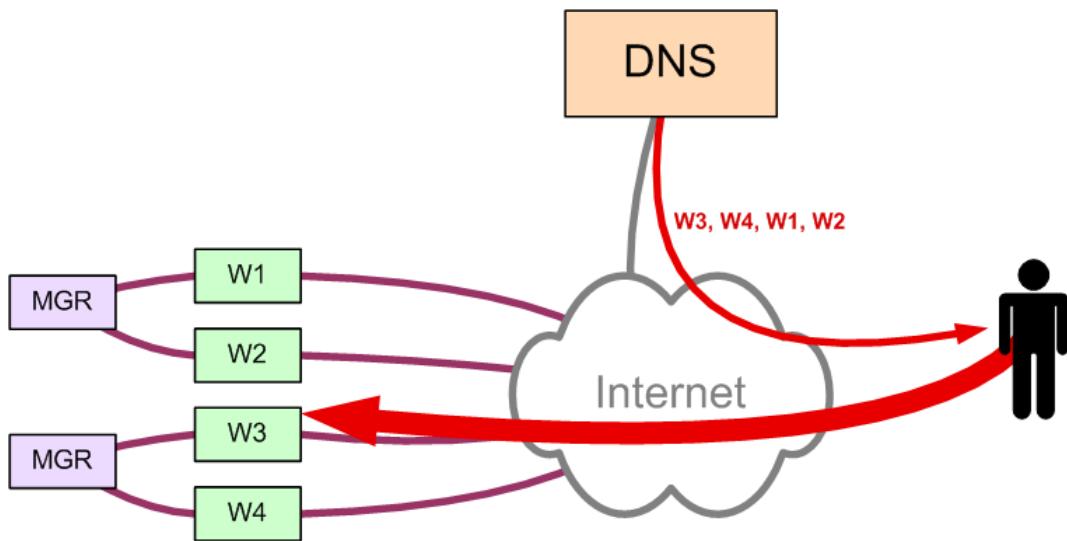
On note que le sous-système WDR peut lui-même être constitué de N serveurs en cluster. WDR n'utilise pas de base de données, il utilise seulement un moteur de recherche, qui est Lucene/SolR. SolR peut fonctionner en cluster, et gérer de très gros volumes et de très fortes capacité d'accueil.

Répartition de charge

Pour Woozweb, nous avons adopté les principes suivants :

- Il n'y a *pas de contexte de session*, et donc *pas d'affinité de serveur* ; n'importe quel internaute peut être traité par n'importe quel frontal web, et il peut changer de frontal au cours d'une même session.
- Cela nous permet de retenir un principe de load-balancing par DNS round-robin, qui est économique en infrastructure, et compatible avec de l'hébergement peu coûteux et multi-datacenters.

- Malgré tout, certains internautes membres sont identifiés. Leur session est alors gérée dans un cookie de domaine, qui est adressé à n'importe quel frontal avec chacune des requêtes. Le cookie porte un cryptogramme qui permet de valider l'authentification et la durée de vie de la session.



Agrégation de contenus

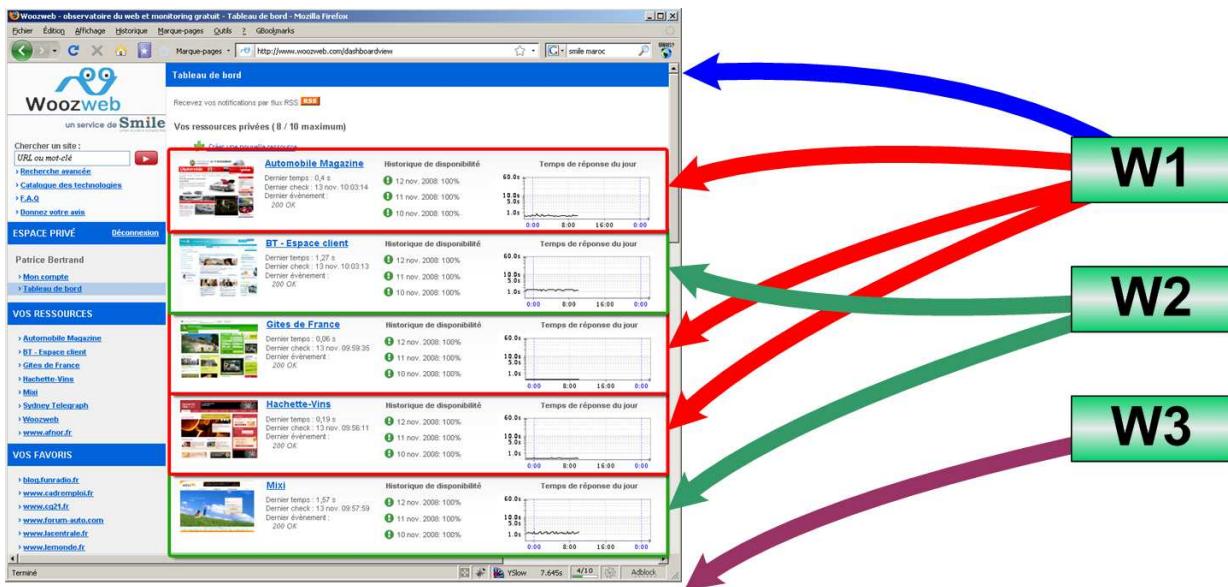
Il n'y a pas d'affectation des *users* aux NODs, mais comme on l'a vu chaque *ressource* est affectée à un NOD. Ainsi les ressources préférées ou privées d'un utilisateur peuvent être réparties sur différents NODs. Néanmoins, il faut être en mesure de lui présenter un tableau de bord consolidé, qui réunisse des informations relatives à différentes ressources.

Pour élaborer ces pages « tableau de bord », il faut donc réunir des fragments d'informations issues de différents serveurs.

Nous avons vu plus haut les différentes options qui se présentent pour réaliser cela. L'une des plus simples est l'agrégation *côté client*, c'est à dire réalisée sur le poste client, au sein du navigateur.

C'est la voie qui a été retenue pour Woozweb, que l'on peut représenter comme ceci :

Architectures Hautes-Performances



A gauche, on distingue une page tableau de bord vue par un utilisateur de Woozweb. On voit qu'elle réunit des fragments d'information correspondant à différentes ressources. Ces ressources sont gérées par différents NODs.

C'est au niveau du navigateur, sur le poste de l'utilisateur, que les différents frontaux sont invoqués pour obtenir les fragments correspondant à chacune des ressources, afin de constituer la page finale.

Synthèse

Woozweb est un cas d'école intéressant, car il réunit une diversité de techniques que nous avons passées en revue :

- Partitionnement et consolidation
- Découpage fonctionnel
- Absence de contextes
- Répartition par DNS-RR
- Agrégation de fragments côté client.

Au final, nous avons une plateforme réellement extensible, qui pourra montrer plusieurs millions de ressources.

CONCLUSION

Au travers de ce petit ouvrage, nous vous avons présenté les fondements architecturaux des plateformes web hautes-performances, en même temps que les outils intervenant dans leur mise en œuvre.

L'open source tient une place toute particulière dans ces infrastructures, tant du fait de l'extrême robustesse de ses outils, que par le moindre coût de possession, en particulier à grande échelle.

Chaque plateforme est une problématique unique, mais comme on l'a vu, on retrouve souvent le même ensemble de bons principes, de bonnes pratiques et de bons outils.

Ayant mis en œuvre un très grand nombre de plateformes web, dont beaucoup accueillent plusieurs millions de visiteurs par mois, et dont certaines comptent jusqu'à 50 serveurs, Smile possède un réel savoir-faire dans la conception de ces plateformes, leur mise en œuvre et leur exploitation. Ceci, qu'elles soient issues de développement d'applications spécifiques, conçues pour l'extensibilité, ou bien construites à base de progiciels.

Nous espérons que cette présentation vous aura été utile, et sommes à votre disposition pour passer à la phase pratique !