

Projet ROSES
Programme MDCO – Edition 2007

Livrable no D2.4
RSS-XML freshness measures

Identification

Acronyme du projet	ROSES
Numéro d'identification de l'acte attributif	ANR-07-MDCO-011-01
Coordonnateur	Paris 6
Rédacteur (nom, téléphone, email)	Dhia Hedfi et Cécile Le Pape, (+33) 01 44 27 88 75, Cecile.Lepape@lip6.fr
No. et titre	RSS-XML freshness measures
Version	
Date de livraison prévue	31 décembre 2008 / t0+12
Date de livraison	31 décembre 2008 / t0+12

Résumé

Ce document décrit l'étude et l'implantation d'un moniteur de flux RSS pour ROSES. Ce moteur est fondé sur un nouveau protocole de syndication adaptatif (WSP) pour le rafraîchissement de flux RSS/ROSES. On décrit également les résultats de premières expériences pour valider un modèle théorique d'estimation de l'influence de différents paramètres (fréquence de rafraîchissement, taille de fenêtre) sur la qualité d'un moniteur de flux RSS (fraîcheur, complétude).

Table des matières

1	Introduction	5
2	Etude des approches de synchronisation des flux	9
2.1	Synchronisation avec serveurs indépendants	10
2.1.1	Synchronisation d'un seul flux	10
2.1.2	Synchronisation de plusieurs flux	11
2.2	Synchronisation coopérative	14
2.2.1	Pull coopératif	14
2.2.2	Push coopératif	15
2.3	Analyse	16
3	Architecture	21
3.1	Vue globale de l'architecture	21
3.2	La couche contrôle système	22
3.2.1	Le module <i>Subscription Manager</i>	22
3.2.2	Le module <i>Access Manager</i>	23
3.2.3	Le module <i>synchronization Mode Manager</i>	23
3.3	La couche logique	23
3.3.1	Le module <i>Synchronizer</i>	24
3.3.2	Le module <i>Trigger</i>	25
3.3.3	Le module <i>Treatment</i>	25
3.4	La couche physique	25
4	Système de syndication pour le Web	27
4.1	WSP : Protocole de Syndication dans le Web	27
4.1.1	Définitions et propriétés	27
4.1.2	Modèle : abstraction et besoins	27
4.1.3	Définition et formalisation des primitives	28
4.1.4	Diagramme de séquence de WSP	30
4.2	Modèle d'apprentissage et décision	32
4.2.1	Concepts de décision	32
4.2.2	Règles de qualité de service	32
4.2.3	Méthodes de décision	34
5	Résultats expérimentaux	39
5.1	Environnement et éléments de configuration	39
5.1.1	Scénario global	39
5.1.2	Paramètres de qualité	40
5.2	Analyse des résultats	40
6	Conclusion	43
	Bibliographie	45
	Annexe	57

Table des figures

1.1	Architecture en couches	5
1.2	Exemple simplifié	6
2.1	Vue globale du paysage du Web	9
2.2	Architecture Pull	10
2.3	Architecture Push	11
2.4	Architecture 1 Flux client - N Flux serveur	12
2.5	Architecture N Flux client - 1 Flux serveur	12
2.6	Architecture N Flux client - N Flux serveur	13
2.7	Comparaison Pull / Push	13
2.8	Architecture du pull coopératif	14
2.9	Architecture du FeedEx	15
2.10	Architecture du Push coopératif	16
2.11	Représentation en niveaux d'abstraction	17
2.12	Grille d'analyse	18
2.13	Répartition en deux dimension : Mode / classe	19
3.1	Architecture générale	21
3.2	Configuration en Pull	22
3.3	Configuration en Push	22
3.4	Architecture de la couche contrôle système	23
3.5	La couche logique	24
4.1	Architecture	28
4.2	Le diagramme de séquence de WSP	31
4.3	Les paramètres d'évaluation	32
4.4	Exemple simple de calcul du mode initial	34
5.1	Le scénario de la simulation	39
5.2	Les courbes de la simulation	41
5.3	Le système personnalisable WSS	42
6.1	Les cas d'utilisation du système	49
6.2	Diagramme de séquence du système proposé	50
6.3	Modélisation du système en MAP	51
6.4	Diagramme de séquence du mécanisme Pull	52
6.5	Diagramme de séquence du mécanisme Push	53
6.6	Modèle conceptuel de la base de données serveur	54
6.7	Modèle conceptuel de la base de données client	55
6.8	L'API serveur	55
6.9	L'API producteur	56
6.10	L'API client	56

Chapitre 1

Introduction

La syndication dans le web consiste à produire, diffuser et partager des informations dans le web. Elle utilise plusieurs techniques telles que RSS et ATOM [15]. La technologie RSS (Really Simple Syndication) [Wit06] [22], plus communément appelée fil RSS ou encore fil de syndication, est actuellement de plus en plus populaire [dL08] [2] [5] [6] [9] [14] [15] [18] [20] [21]. Elle est habituellement utilisée pour signaler les mises à jour des sites dont le contenu change fréquemment, typiquement les sites d'information ou les blogs [Édi06] [17]. L'utilisateur peut s'abonner aux flux, ce qui lui permet de consulter rapidement les dernières mises à jour sans avoir à se rendre sur le site. Concrètement, un flux RSS est un document XML mis à disposition par un serveur Web, structuré comme une liste d'éléments, chaque élément contenant une date de publication, un résumé et un lien vers une page Web. Le client s'abonne au flux via un lecteur de flux. Il existe des sites permettant de s'abonner et de visualiser ses abonnements (igoogole, netvibes) [Cou07] [7] [12] [16]. Il existe également divers lecteurs de flux. Cette fonctionnalité est intégrée dans la plupart des navigateurs (Opera, IE, Firefox). Le lecteur de flux rafraîchit ensuite régulièrement le flux selon la période spécifiée par le client en réinterrogeant le serveur web mettant à disposition le flux.

Dans ce cadre, le projet ROSES (Really Open and Simple Web Syndication) [4] a pour objectif la définition d'un ensemble de services de syndication pour la localisation, l'interrogation, la génération, la composition et la personnalisation de flux RSS sur le Web. Un flux de données représente, principalement, une vue continue sur les données. Un flux de syndication est défini comme étant une séquence infinie des fenêtres sur des données. La fenêtre courante est une vue sur la portion de données la plus récente 1.2. La séquence infinie des fenêtres représente donc le flux de syndication. D'autre part, le flux RSS se caractérise par la notion de mise à jour permanente. En effet, le flux est par définition une source de données qui change d'une manière régulière. Il sera impérativement utile que le flux de syndication présente une propriété pareille qui garanti la variabilité durable des informations exposées. Par ailleurs, la disponibilité du flux représente une nécessité pour la source de flux. Donc, le détenteur de flux doit assurer la disponibilité permanente de son flux pour les demandeurs abonnés ainsi que pour les nouveaux demandeurs des nouvelles offertes 1.2. Dans ce cadre, ROSES permet de trouver des solutions afin de localiser les flux, les interroger et les composer avec d'autres informations.

Dans le cadre du projet ROSES, ce mémoire traite l'aspect synchronisation des flux RSS 1.1. Notre apport consiste à étudier et remédier aux problématiques qui peuvent régir le rafraîchissement de flux.

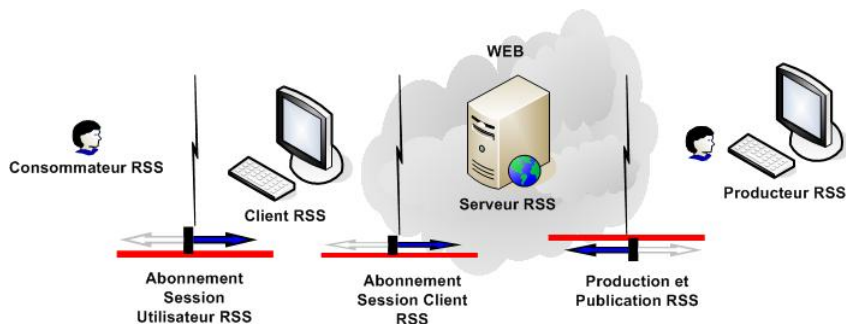


FIGURE 1.1 – Architecture en couches

A partir de cette vue généralisée illustrée dans la figure 1.1, nous identifions les éléments de base pour notre

étude. Un système de flux peut être conçu comme étant une architecture client-serveur. Le producteur du flux assure la production du flux (ex : fnac) et la propriété de mise à jour permanente du flux. Le flux est créé à partir d'un serveur producteur et hébergé dans un serveur de flux. Un serveur peut supporter plusieurs flux à la fois. Le mécanisme de la consultation du flux continu se fait dans la couche consommatrice et à travers la couche communication. En outre, le mécanisme de transport et l'intercommunication, entre le producteur et le consommateur, est assuré par plusieurs protocoles notamment le *HTTP*. Une étude de la spécification du système, basée sur la communication entre les différents éléments dans cette architecture, permet de définir trois processus de base qui nous concernent dans ce mémoire :

- Processus d'abonnement : Un client peut s'abonner à n'importe quel flux disponible chez un serveur ;
- Processus de syndication de données : i.e. la production du continu et la mise en disponibilité. Un serveur met en disposition un ou plusieurs flux et doit répondre aux requêtes clients ;
- Processus de transport de requêtes : i.e. la transmission des requêtes entre le client et le serveur. Les requêtes concernent la demande de mise à jour et les réponses éventuelles.

Ces processus permettent de déduire l'objectif principal du système de flux. En effet, un tel système aura comme but primordial la synchronisation entre le flux disponible de côté serveur et celui émis à un client abonné [CGM03], [WA]. En d'autre terme, la vue cliente du flux doit être synchronisée avec le serveur qui détient le flux pour assurer la fraîcheur des informations mises à disposition. La problématique générale de notre mémoire est liée à la performance et l'efficacité des systèmes gérant les flux RSS dans la synchronisation des flux. Plus particulièrement, la montée à l'échelle de ces architectures avec la dimension des flux dans le web aujourd'hui. Cette problématique s'inspire du facteur performance. La performance, dans notre cas, sera évaluée par la capacité de répondre à tout type de requêtes RSS ayant des finalités différentes. En effet, les requêtes RSS sont des requêtes utilisées dans les architectures basées sur les flux RSS comme moyen d'interrogation, de gestion et de contrôle des flux. Nous démontrons que la problématique évoquée dans ce mémoire s'articule autour de trois facettes :

- La fraîcheur du flux : le client abonné demande les dernières nouvelles fraîches mise en disposition à partir du flux.
- La complétude du flux : le processus de synchronisation entre la source de flux et le consommateur doit faire en sorte que le client ne rate pas des nouvelles.
- La charge et le coût : le processus de synchronisation demande un coût élevé de transmission de requêtes ainsi que dans certaine circonstances augmente la charge des côtés client et serveur.

A travers la figure 1.2, nous présentons un exemple simplifié afin de mieux expliciter la problématique. Dans ce cas de figure, on est en présence d'un processus de synchronisation d'un flux entre client et serveur. Le flux

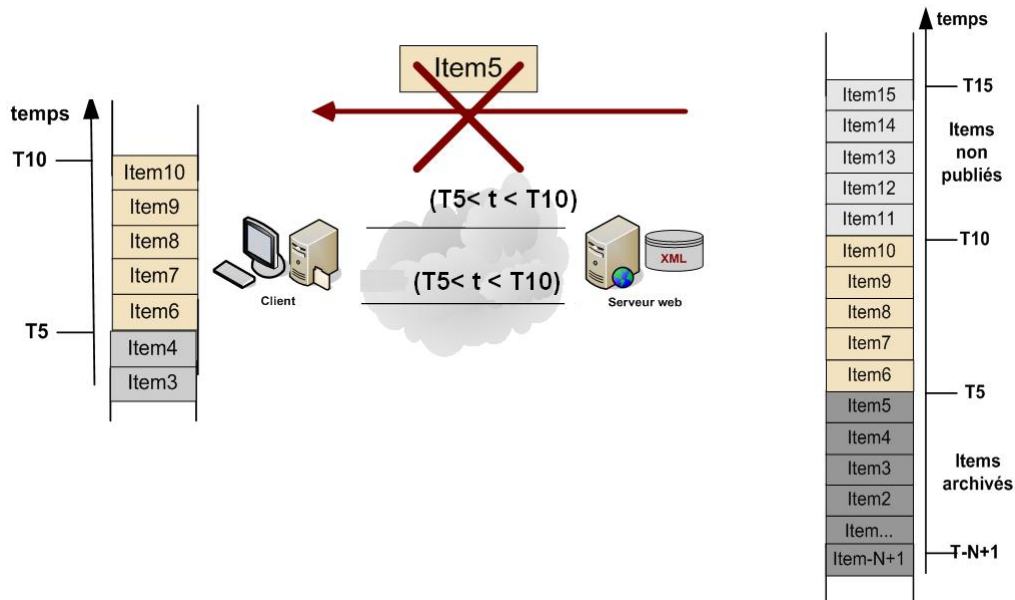


FIGURE 1.2 – Exemple simplifié

disposé dans le serveur est une séquence des fenêtres (F1, F2, etc.). Chaque fenêtre présente une suite des Items (Item1, Item2, ..., Item15). Un item est une entrée (entité) qui encapsule l'information consultable. Pour chaque laps de temps ($T0, T1, T2, \dots$), le système est en présence d'une fenêtre dite actuelle.

La problématique s'explique donc par trois facettes. D'une part, le coût présente un paramètre essentiel dans la

synchronisation. Le système doit s'adapter aux performances disponibles soit du serveur soit même du réseau. Il doit utiliser efficacement la charge disponible à tout instant. D'autre part, la vue cliente doit être fraîche par rapport à celle du serveur. Cela permet le suivi permanent d'actualité ainsi, par la suite, une prise en connaissance instantanée avec les dernières mises à jour. La dernière facette dévoilée pour la problématique de la synchronisation consiste à avoir une vue cliente complète du flux. Autrement dit, le client, dans certaines situations sensibles, n'est pas autorisé de rater des informations disponibles à partir du flux. Dans ce cas, le système de syndication sera le responsable à garantir le côté complétude du flux.

Plusieurs travaux proposent des systèmes de syndication capables de passer à une large échelle avec la dimension du flux. Un système de syndication est un système capable de diffuser le contenu dans le web pour les clients à partir des serveurs. Ces derniers représentent les détenteurs des flux alors que les clients sont les abonnés au flux. Ces approches étudiées sont classifiées en deux catégories : La première étant relative à la synchronisation dans le contexte indépendant où les serveurs correspondent à la couche productrice, les flux et les clients représentent la couche consommatrice. Alors que dans la deuxième classification, les clients peuvent jouer aussi le rôle d'un détenteur du flux. L'aspect travail coopératif est mis en oeuvre dans cette catégorie. Pourvu que les approches de synchronisation prouvent leurs succès à remédier à la globalité des problèmes qui se pose dans ce contexte, elles présentent quelques limites et insuffisances. Elles restent disjointes et ne sont pas capables d'offrir des solutions efficaces pour notre problématique.

Ce mémoire est composé de quatre chapitres. Nous parcourons, à travers la première partie de ce mémoire, plusieurs approches proposées afin de remédier à cette problématique. Il permet aussi d'évaluer ce qui a été proposé en termes de capacité au passage à une large échelle, plus précisément dans le cas d'une activité intense de flux. Notre contribution dans ce mémoire consiste à proposer une approche pour remédier aux problématiques de la synchronisation des flux au sein du projet ROSES. Le deuxième et le troisième chapitre permettent de présenter l'approche proposée que nous estimons capable de remédier à la problématique posée. Ensuite, nous présentons un système de syndication illustré par un protocole de synchronisation et un module de décision. Le dernier chapitre est consacré pour l'évaluation de cette approche.

Chapitre 2

Etude des approches de synchronisation des flux

Actuellement, les typologies du Web ont enrichi les architectures qu'on dispose pour les systèmes de flux syndiqués. Une vue globale du web, illustrée dans la figure 2.1, permet de montrer une multitude de sources de données. Comme la même information peut être trouvée dans plusieurs sources à la fois, il existe plusieurs chemins pour la recherche d'une information [KMMS07] [PV06].

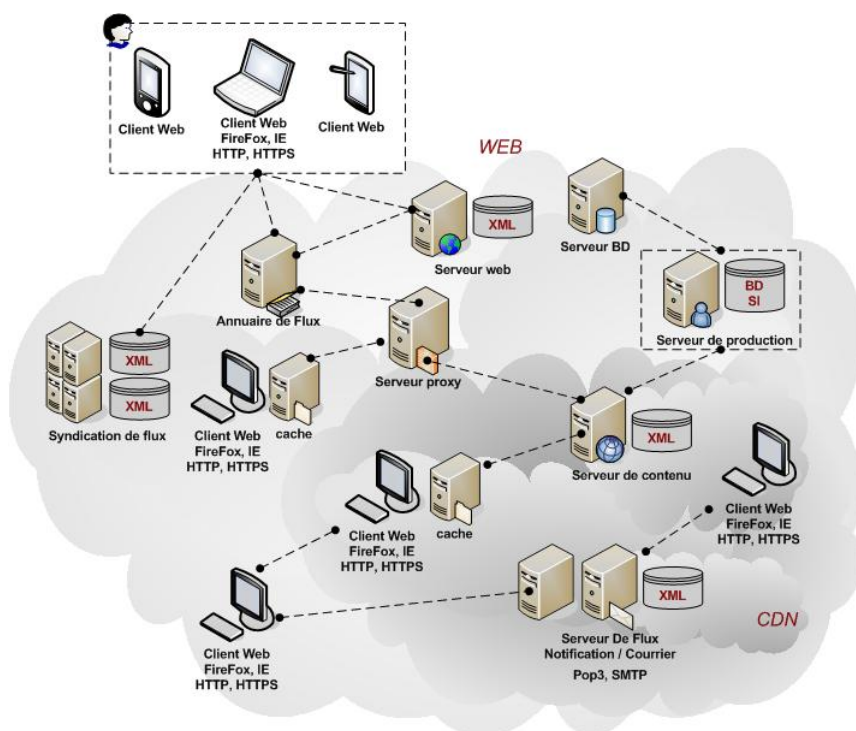


FIGURE 2.1 – Vue globale du paysage du Web

A travers cette figure, nous relevons également une diversité des consommateurs de flux. Ces derniers peuvent être des simples clients (client web). Ils peuvent également dès lors jouer le rôle d'un détenteur d'information grâce à leurs caches. Ils peuvent redistribuer leur contenu à d'autres clients (ex : agrégateur). Ainsi, nous distinguons deux typologies d'architectures fondamentales : les architectures basées sur les serveurs indépendants et celles dites architectures coopératives. Dans les architectures indépendantes, la source de l'information est unique et le serveur est le seul détenteur de flux. Les architectures coopératives, par ailleurs, présentent une multitude de sources dans lesquelles un client peut jouer le rôle de serveur.

Ce chapitre est consacré à un état de l'art sur les politiques de synchronisation des flux de données syndiquées dans le web. Dans la littérature, les approches de synchronisation des flux proposées dépendent de l'architecture en question. Donc, il est recommandé de présenter ces approches au sein de leurs architectures. Dans ce qui suit, nous commencerons par présenter les approches de synchronisation dans un contexte indépendant aussi bien pour un flux unique que dans le cas de plusieurs flux à la fois. Ensuite, nous exposerons plusieurs systèmes dans le cadre de la synchronisation coopérative.

2.1 Synchronisation avec serveurs indépendants

Cette section présente une architecture simplifiée présentant un flux centralisé en une unique source de flux. Dans ce contexte, les sources ne sont pas en coopération. L'étude consiste à étudier la synchronisation dans deux cas de figure : la synchronisation en présence d'un seul flux et celle dans le cas de plusieurs flux à la fois.

2.1.1 Synchronisation d'un seul flux

Le premier cas consiste à avoir un seul flux c'est-à-dire que la source détient un et un seul flux. Ce cas étant le cas le plus simple pour la synchronisation et les techniques de synchronisation sont variées. Dans un tel mécanisme, nous distinguons entre deux modes de synchronisation selon l'initiateur de rafraîchissement : le mode Pull et Push. Nous analysons en détails dans ce qui suit ces deux techniques.

2.1.1.1 Synchronisation par Pull

Avec cette technique, c'est le client qui sollicite le rafraîchissement de sa vue au près du serveur. On dit que le client est actif et que le serveur est passif. La figure 2.2 illustre ce mécanisme.

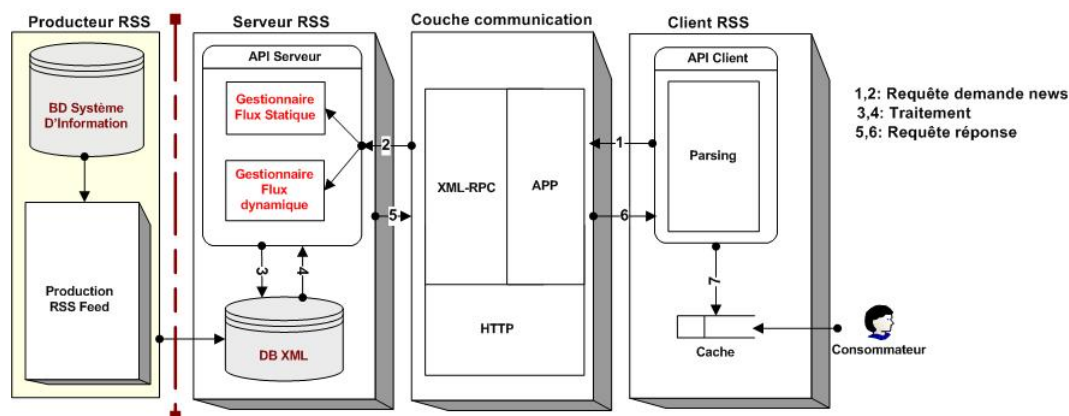


FIGURE 2.2 – Architecture Pull

Le scénario, tel qu'il est présenté dans la figure 2.2, étant le déroulement standard de Pull dans une architecture en couche. Un client actif envoie périodiquement des requêtes de demande des news au serveur passif. Le serveur attend simplement les demandes des clients pour diffuser la mise à jour de son flux. La couche communication garantit à travers des protocoles de communication l'interaction entre les deux entités client et serveur. Les protocoles de communication tel que l'*APP* (Atom Publishing Protocol) et *XML-RPC* sont des protocoles de syndication existants aujourd'hui et se basent sur l'*HTTP*.

Plusieurs techniques utilisant ce mécanisme pour la synchronisation des flux. Le *HTTPconditionalGET* et *HTTPentityTags(ETags)* sont deux techniques *HTTP* de type *Pull*. Le client envoie une requête de type *HTTPconditionalGET* pour demander la mise à jour disponible chez le serveur. La condition est d'avoir une nouvelle version par rapport à celle courante. La version courante est la version préalablement chargée. La technique *HTTPentityTags(ETags)*, lors de l'expiration d'un *timer* (temporisateur) côté client, déclenche une requête de demande de mise à jour.

Le serveur manipule le temporisateur pour chaque flux. Il désigne, à chaque envoi de mise à jour aux clients, la valeur du *timer*. Cette valeur est passée dans le paramètre *RSSTimeToLiveTTL* dans la requête réponse.

2.1.1.2 Synchronisation par Push

Dans ce cas de figure, c'est le client qui s'abonne au flux. Le serveur rafraîchit les vues clients à chaque fois une nouvelle mise à jour soit disponible. Dans la figure 2.3, le module *Gestionabonn* permet au serveur de connaître tous ses abonnés.

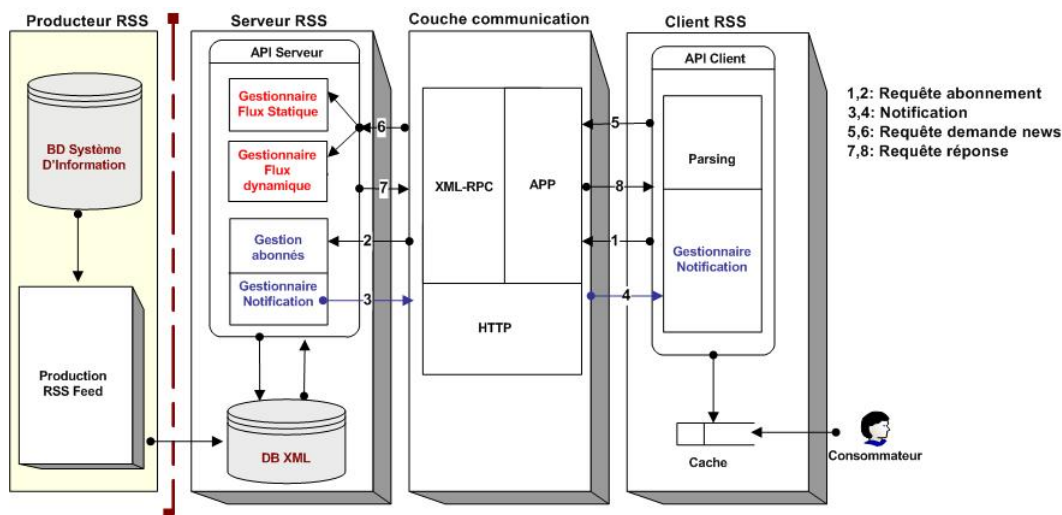


FIGURE 2.3 – Architecture Push

Comme figuré dans 2.3, un mécanisme de notification gère des alertes quand un flux est mis à jour et notifie tous les abonnés afin qu'ils accèdent à la mise à jour. Le mécanisme *Push* permet de réagir en temps réel aux événements. Ce qui permet une efficacité d'accès à l'information. Le mode *Push* est plus approprié dans le cas de besoin d'information d'une manière continue. Ceci est préconisé par le processus de souscription. Le client aura souvent l'information souhaité sans être obligé d'aller chercher à chaque fois. La technique *DeltaEncoding* est une techniques utilisant ce mécanisme. Elle se base sur l'envoi de mise à jour au client une fois que le serveur détecte une modification récente au niveau du flux. La technique *EntityTags(LastModified, IfModifiedSince)* utilise aussi *Push* grâce à la capacité de *HTTP* de gérer les requêtes de mise à jour.

2.1.2 Synchronisation de plusieurs flux

Après avoir étudié quelques méthodes de synchronisation des flux en présence d'un flux unique, cette section présente une étude des approches proposées dans le cadre de plusieurs flux à la fois. Nous adopterons dans cette section une classification selon la dimension du flux côté client et côté serveur. Dans chacun des cas, en se basant toutefois sur une architecture de référence, nous déterminerons les paramètres dont dépend la synchronisation. Nous présenterons ainsi quelques approches dans le cas de Pull et de Push.

Cas d'un flux côté client et N flux côté serveur : Nous nous basons principalement sur cette architecture représentée dans la figure 2.4. Dans cette architecture, le serveur détient N flux et chaque client s'abonne à un et un seul flux depuis le serveur désigné. Nous étudions la synchronisation côté serveur car aucun nouveau problème ne se présente côté client. Trois paramètres déterminent la manière de synchroniser les flux côté serveur :

- Le nombre de flux détenus par le serveur
- Le nombre de clients abonnés à chaque flux
- La fréquence de mise à jour pour chaque flux

La figure 2.4 illustre l'utilisation des lecteurs/agrégateurs de flux [8]. Les flux RSS menacent d'augmenter considérablement la bande passante des serveurs qui les diffusent. Cela revient au nombre important des requêtes adressées au serveur diffuseur. Il est ainsi possible qu'un site ne soit plus mis à jour aussi souvent qu'avant, mais que l'augmentation du nombre d'abonnés à son flux RSS et le requêtage automatique des lecteurs/agrégateurs induisent une hausse conséquente de la bande passante pour le diffuseur.

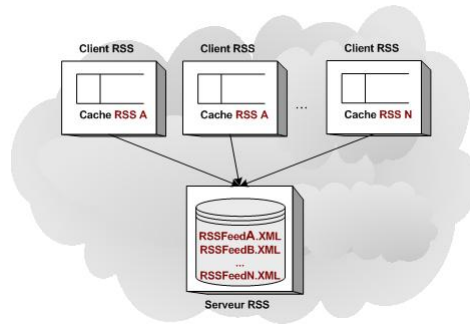


FIGURE 2.4 – Architecture 1 Flux client - N Flux serveur

Dans ce cas, les agrégateurs de flux utilisent les mêmes techniques de synchronisation vues dans la section précédente sauf qu'ils gèrent plusieurs flux à la fois. Un problème de montée en charge dans le cas du mécanisme " Push " peut forcément se présenter vu la dimension des flux à gérer et le nombre de flux par client.

Cas de N flux côté client et 1 seul flux côté serveur : Un client gère N flux où chaque flux est détenu chez un serveur différent. Chaque serveur détient un et un seul flux et la synchronisation se fait donc côté client.

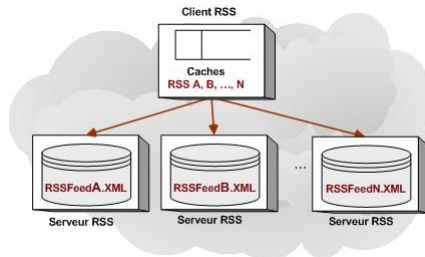


FIGURE 2.5 – Architecture N Flux client - 1 Flux serveur

La figure 2.5 illustre globalement le fonctionnement des annuaires RSS [1] [10] [11] [21] et les portails de flux. Il est possible (via des annuaires tiers par exemple) de s'abonner à un flux RSS sans vraiment connaître le site qui le diffuse (et donc sans savoir s'il est fréquemment mis à jour). Nous déterminons deux paramètres qui déterminent la manière de synchroniser les flux côté client :

- Le nombre de flux à gérer coté client.
- La fréquence de mise à jour pour chaque client.

Le *RollerWebBlogger* présente une architecture similaire et utilise les techniques de base de synchronisation sauf que le mode *Pull* est le mécanisme standard adopté. *FeedSync* est également un système de publication qui se base sur cette architecture. Il modifie la spécification du flux en ajoutant d'autres paramètres désignant la période dynamique de synchronisation.

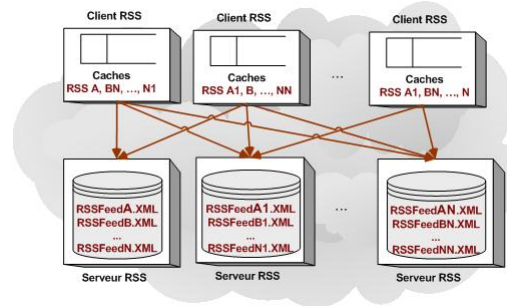
Cas de N flux côté client et N flux côté serveur :

FIGURE 2.6 – Architecture N Flux client - N Flux serveur

Dans cette architecture comme figurée dans 2.6, chaque serveur détient N flux et chaque client s'abonne à N flux à la fois depuis ces serveurs désignés. Nous évaluons la synchronisation des deux côté client et serveur. Deux paramètres déterminent la manière de synchroniser les flux côté client :

- Le nombre de flux à gérer coté client
- La fréquence de mise à jour pour chaque client

Et trois paramètres pour la synchronisation des flux côté serveur :

- Le nombre de flux détenus dans chaque serveur
- Le nombre de clients abonnés pour chaque flux
- La fréquence de mise à jour pour chaque flux

Cette architecture illustre aussi le fonctionnement mixte dans un système à base de flux. Au sein de ce système, le problème de synchronisation s'aggrave étant donné la présence de plusieurs typologies de client et de serveurs dans un contexte indépendant [KLJS04]. C'est ce dernier principe qui préside au calcul théorique effectué pour la fréquence de mise à jour pour chaque flux. Nous sommes en présence d'une menace que représente les flux RSS sur la consommation de bande passante des serveurs diffuseurs et des clients abonnés. Les choses ne sont donc pas si simples : il faut énormément plus de requêtes RSS que d'accès à une page du site correspondant pour que les consommations de bande passante induites par les uns ou les autres s'égalent.

Conclusion

La figure 2.7 compare les deux mécanismes Pull et Push selon une liste de critères fonctionnels.

Critère fonctionnel	Pull	Push
Pertinence des informations reçues	★	
Réquisition d'abonnement		★
Nécessite plus de temps	★	
Risque de veille irrégulière	★	
Économie de temps		★
Recherche automatique		★
Disponibilité des services de signalement rapide		★
Résultats de recherche volumineux		★

FIGURE 2.7 – Comparaison Pull / Push

Ces deux mécanismes ont tous les deux quelques points faibles pour certains scénarios. La mise en oeuvre de certaines approches de synchronisation proposées dans le cadre de la synchronisation à serveurs indépendants, vues au sien de leurs architectures, prouvent évidemment leurs performances dans le cas d'une activité de flux réduite. Dans le cas d'une activité intense, cette étude montre les limites de ces approches pour un passager à l'échelle en terme de différents clients, de messages, de bande passante...

2.2 Synchronisation coopérative

Dans la section précédente nous avons présenté quelques approches de synchronisation à ressources uniques, c'est-à-dire qu'il n'y avait aucune sorte de travail coopératif entre les différents éléments du système de flux. Dans cette section, nous nous intéressons à un système de flux en multi-sources dans lequel ceux-ci peuvent être consultés à partir de plusieurs sources réparties. A partir du moment où un client présente les propriétés d'un serveur, il pourra jouer le rôle d'une source secondaire. Le cache client est donc devenu actif au sein d'une telle architecture coopérative [3]. Afin d'étudier la synchronisation de flux dans ces architectures, nous présentons les approches proposées et leurs manières de synchroniser le flux consommateur avec celui de la source. Nous distinguerons en premier lieu le pull coopératif comme étant l'ensemble des architectures coopératives basées sur Pull, et en second lieu celles basées sur le push.

2.2.1 Pull coopératif

L'interrogation des fils de news dans une architecture coopérative dans le but de la synchronisation fait l'objet d'étude dans cette section.

2.2.1.1 Principes généraux

Le flux dans le système ne peut être fourni à la couche consommatrice qu'après une demande envoyée à la source. Le système de flux distribué coopératif est défini non seulement comme étant plusieurs sources à la fois en coopération mais en plus, chaque abonné interagit avec le système comme s'il s'agissait d'une seule et unique source. De ce fait, la caractéristique la plus importante de ce type de système est que le client ne doit avoir aucune idée du fonctionnement interne du système, du fait même qu'il est géré par plusieurs sources en coopération.

2.2.1.2 Architectures proposées

Nous présenterons à travers la figure 2.8 l'architecture générale de ces systèmes en pull coopératif.

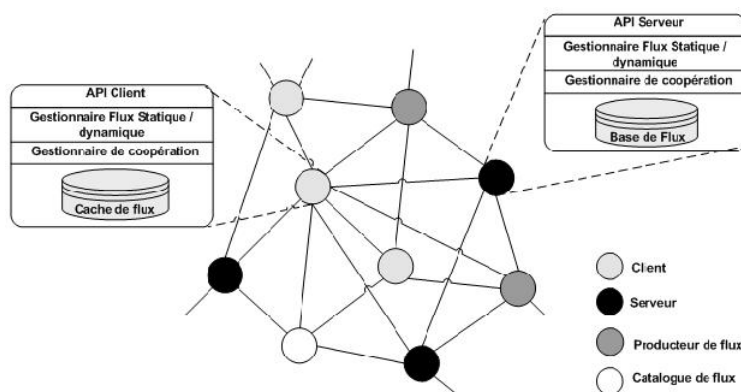


FIGURE 2.8 – Architecture du pull coopératif

Au sein de cette architecture, un client peut être à la fois un simple demandeur d'actualité et un détenteur de flux. Cette architecture se base sur *Pull* comme mécanisme de synchronisation. Le consommateur de flux, ainsi, peut accéder à l'information à partir de plusieurs alternatives. Il se peut qu'un client, après avoir eu les dernières actualités sur le flux, puisse servir de son cache pour se comporter comme étant un serveur pour d'autre client. Les architectures conçues pour le client et le serveur sont pareil sauf que le premier détient un cache au lieu d'une base de flux.

2.2.1.3 Pull à travers des architectures coopératives

Deux approches principales de synchronisation font l'objet de discussion dans cette partie. Le système *FeedEx* [JA06] est un système qui se base sur la notion de pull coopératif dans le contexte de l'échange collaborative des fichiers d'alimentation des flux RSS. En particulier, l'exploration du potentiel de partage des fils d'alimentation. Il traite l'accélération de la diffusion et la réduction des charges des serveurs. Feedex, formant un réseau de distribution, ses noeuds cherchent non seulement les fichiers d'alimentation à partir des serveurs, mais aussi les

échanger avec les voisins. Les noeuds forment une superposition de distribution réseau de fils de nouvelles qui sont échangés.

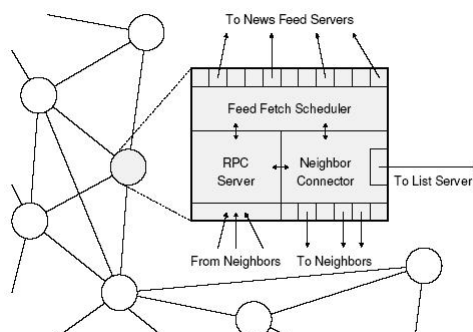


FIGURE 2.9 – Architecture du FeedEx

Cet échange décrit dans la figure 2.9 permet aux noeuds de réduire la fréquence de la récupération documents à partir des serveurs, il peut diminuer donc la charge du serveur. Dans ce sens, FeedEx construit un système efficace pour la technologie d'alimentation des actualités. En raison de cette efficacité, les noeuds se bénéficient des délais de livraison et de haute disponibilité des fichiers d'alimentation. Un mécanisme d'incitation a été proposé pour FeedEx afin que les noeuds soient encouragés à être des collaborateurs plutôt que de resquilleurs. Depuis FeedEx ne nécessite aucune modification de l'alimentation actuelle des serveurs ou des formats de document, il peut être déployé facilement.

Dans le cadre ces approches de synchronisation basé sur le pull coopératif, FeedTree [WA] est une amélioration du processus *Pull* utilisé dans l'architecture distribuée. FeedTree part de l'hypothèse que l'architecture actuelle centralisée ne pourra pas soutenir le phénomène de croissance RSS, et qu'une approche distribuée est nécessaire. Une des raisons est que dans les architectures centralisées lequel tous les clients utilisent le *pulling* périodique sur un serveur central procure une bande passante qui croit linéairement avec le nombre d'abonnés. Les propriétés des superpositions en peer-to-peer (p2p) peuvent être complètement formalisées pour ce problème. En effet, les systèmes *p2p multicast* présente une approche logarithmiquement proportionnelle à la dimension du flux et devraient soutenir la participation des millions de nœuds. Par conséquent, l'approche proposée estime que les flux RSS peuvent être distribués d'une manière qui partage les coûts entre tous les participants.

2.2.2 Push coopératif

Après avoir étudié les mécanismes proposés dans quelques approches de synchronisation de flux dans les architectures coopératives en Pull, nous détaillerons dans cette section ce qui a été proposé pour le cas de Push coopératif.

2.2.2.1 Principes généraux

Nous rappelons qu'à part l'accès direct au serveur déployant le flux, celui-ci peut être accédé aussi à travers le cache d'un client abonné au même flux. Sachant que nous proposons d'étudier la synchronisation des Flux dans les architectures distribuées coopératives en Push, nous devons admettre aussi la présence d'un mécanisme de notification déclenché à chaque mise à jour du flux dans la source principale.

2.2.2.2 Architecture

La figure 2.10 illustre l'architecture générale des systèmes de flux répartis coopératifs en Push.

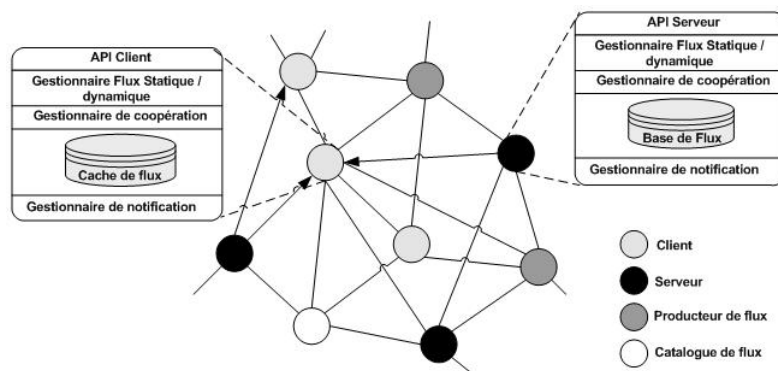


FIGURE 2.10 – Architecture du Push coopératif

Ces architectures, pareilles à celles en pull, ne distinguent pas entre un client et un serveur. L'apport concernant ces architectures par rapport à celles de pull coopératif est le service de notification. A travers la figure, nous démontrons l'ajout du service *Gestionnaire de notification* côté serveur et client.

2.2.2.3 Push à travers des architectures coopératives

Dans la littérature, rares sont les approches qui ont proposé des solutions pour la synchronisation en Pull coopératif. Nous pouvons citer le cas de FeedTree [SMPD05] qui utilise la notification p2p d'événement pour la distribution des MicroNews. La charge placée sur les éditeurs est réduite, ainsi, de façon spectaculaire tout en fournissant le service à la clientèle. Nous proposons d'étudier, aussi, en particulier une approche qui utilise le Pushing à travers des architectures coopératives et plus précisément dans les CDN (content delivery networks). Les CDN sont des réseaux spécifiques représentant une plateforme favorable pour le travail coopératif dans le web. Ils présentent aussi un environnement de coopération pour le système de Freedom [14]. *FeedRoom Distribution System* [13] est un système qui se base sur le processus de livraison de contenu. Ce système utilise le Push dans les réseaux de distribution (tels que les réseaux par satellites exploitant des systèmes de prestation, peer-to-peer, LAN et WAN sans fil etc.). Le système de distribution génère des fichiers XML en différents formats pour faciliter l'échange de métadonnées, la syndication des informations.

2.3 Analyse

Dans une première perspective, à partir de l'étude des approches de synchronisation, nous arrivons à dégager les notions suivantes :

- Protocoles de communication : les protocoles peuvent résoudre la problématique de la synchronisation (exemple : *APP*).
- Modèle du flux (la spécification du flux) : plusieurs techniques se basent sur la spécification du document du flux pour gérer les paramètres de synchronisation.
- Systèmes de publication de flux : Nous trouvons aussi plusieurs systèmes qui proposent des solutions diverses pour la synchronisation (*FeedSync ; system*) [19].
- Algorithmes explicites de rafraîchissement des flux : Nous distinguons des algorithmes de synchronisation qui sont inclus dans des systèmes de syndication (exemple : *ROMES ; Fetcher*)

Ces concepts reflètent une autre classification des différentes approches de synchronisation. Cette classification nous permet de séparer fonctionnellement entre ces politiques et les représenter comme l'illustre le schéma 2.11 en niveaux d'abstraction.

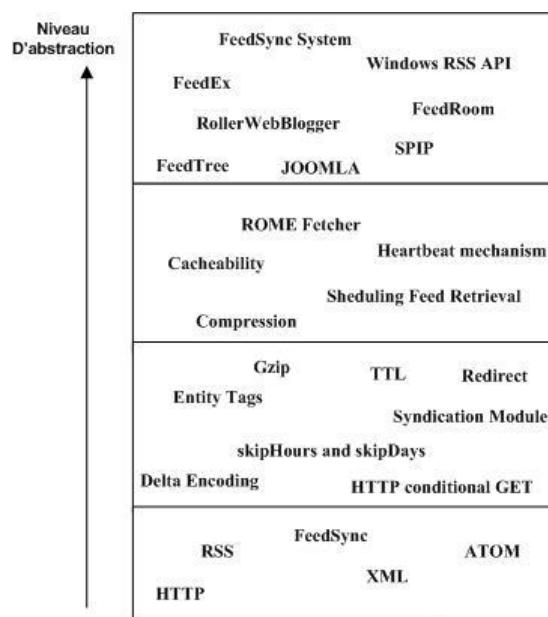


FIGURE 2.11 – Représentation en niveaux d'abstraction

Les niveaux d'abstraction désignent que si une approche rejoint un niveau plus haut, alors elle utilise toutes les techniques de niveau plus bas. Nous distinguons donc entre :

- Les techniques de bases.
- Les méthodes proposant des algorithmes utilisant ces techniques.
- Les systèmes de flux basés sur ces méthodes.

Nous utilisons la méthode QQQQCP (Qui, Quoi, Où, Quand, Comment, Pourquoi) afin d'évaluer selon plusieurs critères les approches présentées. Cette grille permet aussi de définir les modalités de mise en oeuvre d'une solution pour la synchronisation des flux. Cette grille 2.12 illustre une comparaison fonctionnelle entre les approches réparties en catégories. Plusieurs techniques sont proposées afin de remédier à la problématique de la synchronisation. Ces approches sont disjointes et ne peuvent être efficaces dans plusieurs situations. Nous distinguons entre des techniques qui sont simplement des algorithmes de synchronisation (tel que ROME Fetcher) et d'autres qui sont intégrés au sein des systèmes de publication (ex : FeedSync). Les premières ont une insuffisance au niveau de la gestion du coût vu qu'elles n'ont pas une vue globale sur le système. Les systèmes de syndication existant aujourd'hui ne contrôlent pas efficacement le coût. *FeedEx* est le seul système qui a géré la bande passante parmi tous les systèmes présentés. Nous distinguons aussi à travers cette grille des techniques de rafraîchissement qui sont relatives à la spécification RSS (ex : SkipHours) et d'autres qui se basent sur les primitives développées dans le protocole *HTTP* (ex : http Conditional Get). Ces techniques sont variées et ne peuvent pas être applicables dans toutes les situations (répartie, centralisé...). Nous déduisons qu'il y a deux dimensions intéressantes qui permettent de mieux positionner notre approche. Pour mieux analyser ces approches de synchronisation, nous ne conservons que les deux dimensions d'étude suivantes :

- Synchronisation coopérative - Synchronisation avec serveurs indépendants.
- Pull - Push.

Critères d'évaluation	Quoi		Temps	Qui		Où		Quand	Comment	Combien	Pourquoi			
	Données	Événement		Ressources centralisées	distribuées	Coopération	Client				Serveur	Pull	Push	Requête
Algorithme explicites de rafraîchissement des flux														
Scheduling Feed Retrieval	*	*	*	*			*	*	*	*				
Heartbeat mechanism	*	*	*	*			*	*	*	*				
ROME Fetcher			*	Peu importe				*	*	*	*			
Compression	*			*		Peu importe	*	*	*	*				
Caching								*	*	*	*			
Méthodes de rafraîchissement dans les systèmes de publication des flux														
SPiP				*			*	*	*	*				
RollerWeb Blogger				*			*	*	*	*	*			
Feed Sync	*			*		*	*	*	*	*	*	*		
Feed Ex	*			*		*	*	*	*	*	*	*	*	
Feed Tree					*	*	*	*	*	*	*	*		
Feed Room				*			*	*	*	*	*	*		
JOOMLA				*			*	*	*	*	*	*		
Windows RSS API				*			*	*	*	*	*	*		
Techniques de rafraîchissement de Base														
Méthodes de rafraîchissement basées sur le modèle de flux (la spécification RSS)														
skip Hours and skip Days		*	*	Peu importe		Peu importe	Obligation	Pouvoir	*			*		
TTL			*	Peu importe		Peu importe	Devoir	Pouvoir	*			*		
Syndication Module (update period, frequency, base)		*	*	Peu importe		Peu importe	Devoir	Pouvoir	*			*		
Méthodes de rafraîchissement basées sur le protocole de communication														
HTTP conditional GET (Last-Modified)	*		*	Peu importe		Peu importe	Obligation	Devoir	*			*		
Entity Tags (Last-Modified, If-Modified-Since, If-None-Match)	*			Peu importe		Peu importe	Obligation	Devoir	*			*		
Delta Encoding	*			Peu importe		Peu importe	Devoir	Devoir	*			*		
gzip	*			Peu importe		Peu importe	Devoir	Pouvoir	*			*		
Redirect		*		Peu importe		Peu importe	Obligation / Devoir	Pouvoir	*			*		
Gone		*		Peu importe		Peu importe	Obligation	Pouvoir	*			*		

FIGURE 2.12 – Grille d'analyse

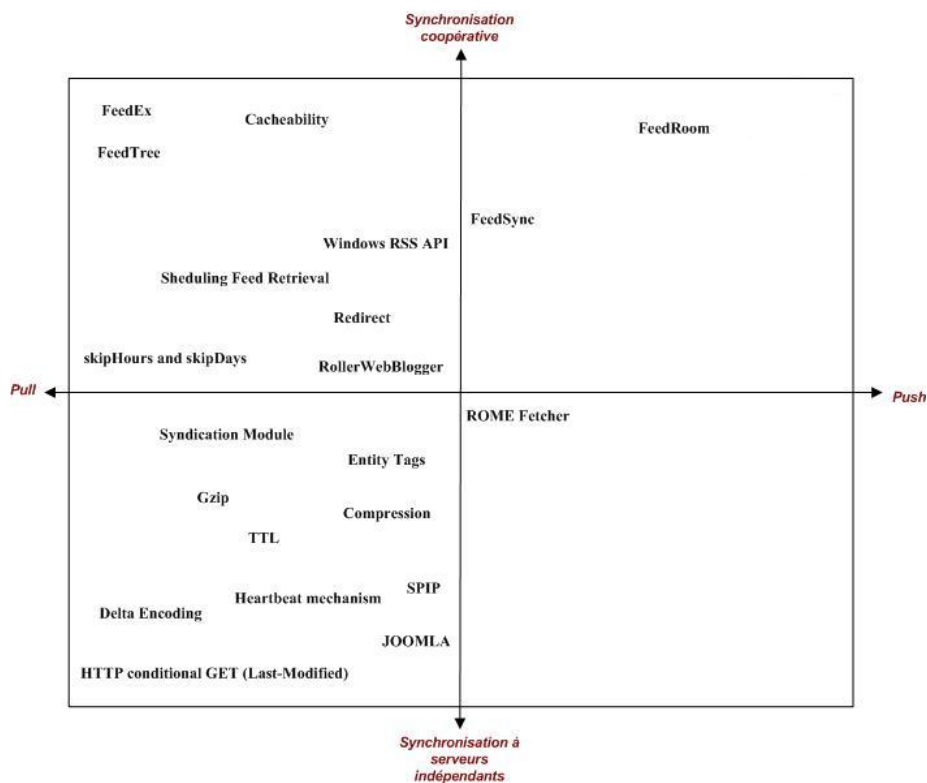


FIGURE 2.13 – Répartition en deux dimension : Mode / classe

Une répartition dans ces deux dimensions que nous proposons dans la figure 2.13 permet de voir une répartition des approches présentées.

2.3.0.4 Conclusion

Actuellement, le web à base de flux présente une richesse au niveau de la diversité d'architectures ainsi qu'une multitude d'approche traitant le processus de synchronisation des flux. Cependant, le problème le plus répandu dans toutes ces architectures est, malgré la puissance relative des approches proposées en présence d'activité faible et moyenne de flux, qu'elles restent incapables de passer à une large échelle. Finalement, les deux dimensions Pull et Push nous paraissent intéressantes afin de déterminer la capacité ou non des méthodes proposées de passer à une grande échelle.

Chapitre 3

Architecture

Ce chapitre présente l'architecture proposée pour synchroniser les flux dans un système client et serveur et dans le contexte de serveurs indépendants. L'objectif de cette architecture est de pouvoir synchroniser un flux entre un client demandant d'actualité et un serveur disposant d'un flux continu en mise à jour. Notre architecture permet de supporter un protocole de synchronisation se basant sur une combinaison de demande de mise à jour à partir du client (Pull) et la notification serveur des clients pour un flux désigné (Push). Ainsi, notre architecture présente un sous système de décision utilisant des techniques d'apprentissage afin de contrôler le mode de synchronisation pour une meilleure efficacité. Nous aborderons dans la première section la présentation générale de l'architecture de synchronisation proposée. Nous détaillons cette architecture dans la seconde section et nous présenterons ainsi une mise en oeuvre des modules dans ce cadre architectural.

3.1 Vue globale de l'architecture

L'architecture illustrée dans la figure 3.1, se présente comme une superposition de trois couches. Ces couches représentent l'architecture standard de référence dans notre travail.

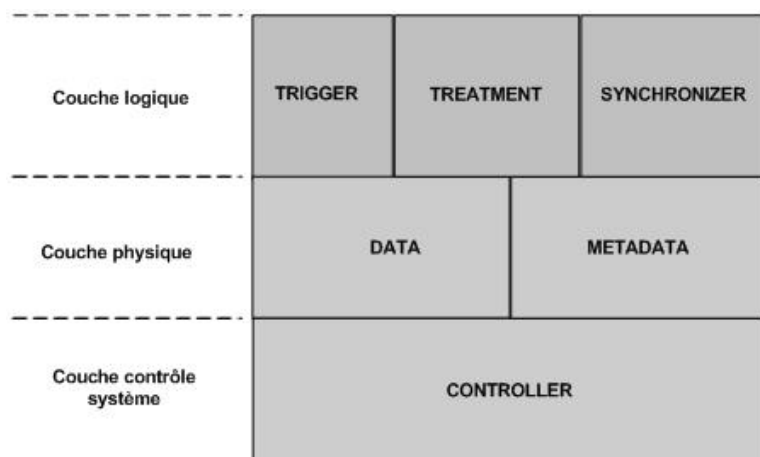


FIGURE 3.1 – Architecture générale

L'architecture proposée est composée de trois couches : logique, physique et contrôle système. Nous adoptons la même architecture de synchronisation pour les deux côtés client et serveur dans le contexte des serveurs indépendants. Deux sorte d'interactions sont conçues dans cette architecture : L'interaction entre les couches au sein de la même architecture et l'interaction entre les couches similaires des architectures client et serveur. Cette architecture propose deux modes de synchronisation :

- Le mode Pull : interrogation de la mise à jour de flux se fait à partir du client
- Le mode Push : le serveur notifie le client de la mise à jour de flux

Nous distinguons trois configurations à partir de cette architecture :

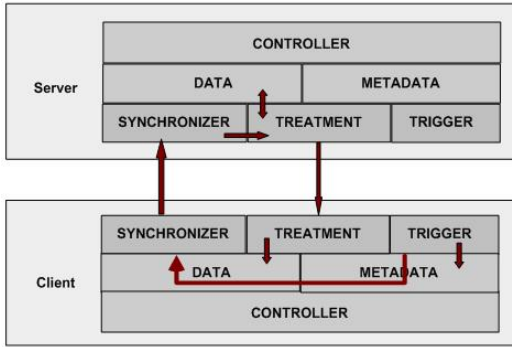


FIGURE 3.2 – Configuration en Pull

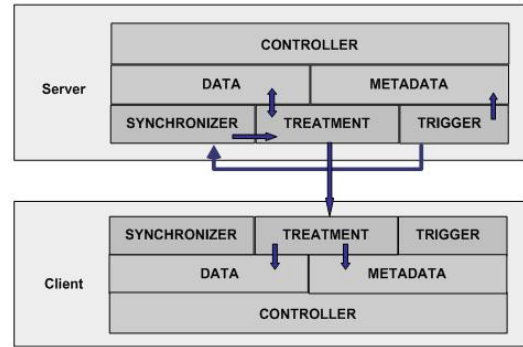


FIGURE 3.3 – Configuration en Push

La configuration pour le mode Pull : Cette configuration, comme elle est illustrée dans la figure 3.2, permet d'acquies le déroulement du mode Pull. La communication entre les modules suit les traits en rouge dans la figure. Nous remarquons que le client déclenche sa demande de mise à jour et le serveur répond ce dernier avec la mise à jour du flux.

La configuration pour le mode Push : La figure 3.3 illustre la configuration qui est capable de dérouler le mécanisme Push. Le serveur joue le rôle de l'initiateur du rafraîchissement. Ce dernier déclenche le processus de synchronisation en envoyant la mise à jour au client.

La configuration pour le mode Mixte : Pour cette configuration, notre architecture utilise les deux configurations présentées pour Push et Pull. Ce mode est garanti par notre architecture afin de créer la notification client au premier lieu, et permettre à ce dernier d'aller chercher éventuellement la mise à jour en second lieu. Une mise en oeuvre des couches de l'architecture proposée est présentée d'une façon plus détaillée dans ce qui suit.

3.2 La couche contrôle système

Cette couche est totalement représentée par le *Controller* illustré dans la figure 3.4. Le contrôleur est responsable de la souscription, du contrôle d'accès et du contrôle du mode de synchronisation. La couche contrôle système est composé donc du module *Subscription Manager*, l'*Access Manager* et le *synchronization Mode Manager*

3.2.1 Le module *Subscription Manager*

Le client envoie, d'après le scénario dans 3.4, au module *Subscription Manager* une demande d'abonnement à un flux disponible dans le serveur. La requête de demande d'abonnement comprend plusieurs paramètres indiquant des informations concernant le flux et le client. L'identification du flux est représentée à travers le paramètre *idDataStream*. Au moment où *idClient* désigne l'identifiant client, *clientState* représente l'état du client au moment de la demande de souscription. Le *userProfile* est le profil utilisateur désignant les préférences de l'utilisateur pour ce flux (le besoin en fraîcheur *freshNeed*, *completenessNeed* qui désigne le besoin en complétude du flux et *importance* reflétant l'importance du flux estimé par l'utilisateur initialement et qui sera réévalué dynamiquement).

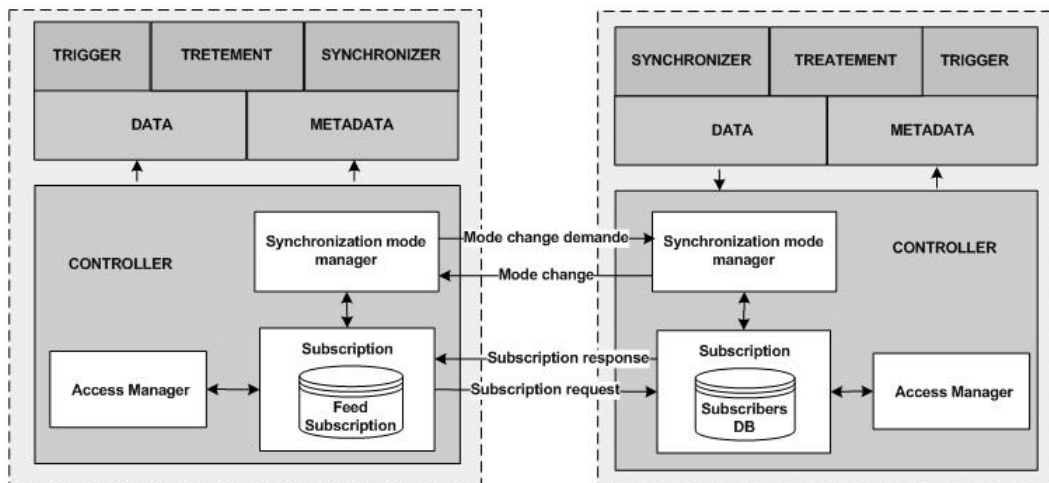


FIGURE 3.4 – Architecture de la couche contrôle système

Le module *Subscription Manager* côté serveur reçoit cette demande, la traite et renvoi à ce client une réponse à sa demande. La réponse peut désigner :

1. Le refus de la demande d'abonnement : dans le cas de l'incapacité de serveur d'accepter l'abonnement client faute de la surcharge actuelle ou même en cas de difficulté de mettre en disponibilité le flux désigné.
2. L'acceptation de l'abonnement : Le serveur accepte la demande client et lui répond avec une requête d'acceptation comprenant trois paramètres. Le *dataStreamWindow* représente le premier paramètre dans cette requête, il désigne la fenêtre actuelle du flux disponible chez le serveur au moment de l'abonnement. Le paramètre *SynchronisationMode* désigne le mode de synchronisation qui va être initialement utilisé. Le troisième paramètre sera relatif au contrat crée pour cet abonnement et qui décrit les obligations du serveur envers le client.

Dans le cas d'acceptation de la demande d'abonnement et au niveau serveur, les coordonnées du client seront capturés dans la base des abonnés. Alors qu'au niveau client, à la réception de la réponse, il enregistre la souscription au part de ce serveur pour le flux désigné.

3.2.2 Le module *Access Manager*

Dans certains cas où le flux présente un critère privé, ce module valide la souscription client. L'*Access Manager* évalue les droits du client qui est en cours de souscrire. Par exemple, dans cetrain cas, le flux présente des aspects confidentiels et que certains clients ont les droits d'accès sur ces informations.

3.2.3 Le module *synchronization Mode Manager*

Le module côté serveur *synchronization Manager* (3.4) est responsable de la prise de la décision du mode de synchronisation. Une première décision initialement calculée par ce module informe le client du mode de départ. Cette décision transmise à travers la réponse à la demande d'abonnement sera passée au module *Sychronization Manager* côté client. Ce module à son tour autorise l'un des deux modes de synchronisation : Synchro Pull ou Synchro Push.

3.3 La couche logique

La couche logique comme illustré dans la figure 3.5 est composée de trois modules à savoir le *Synchronizer*, le *Treatment* et le *Trigger*

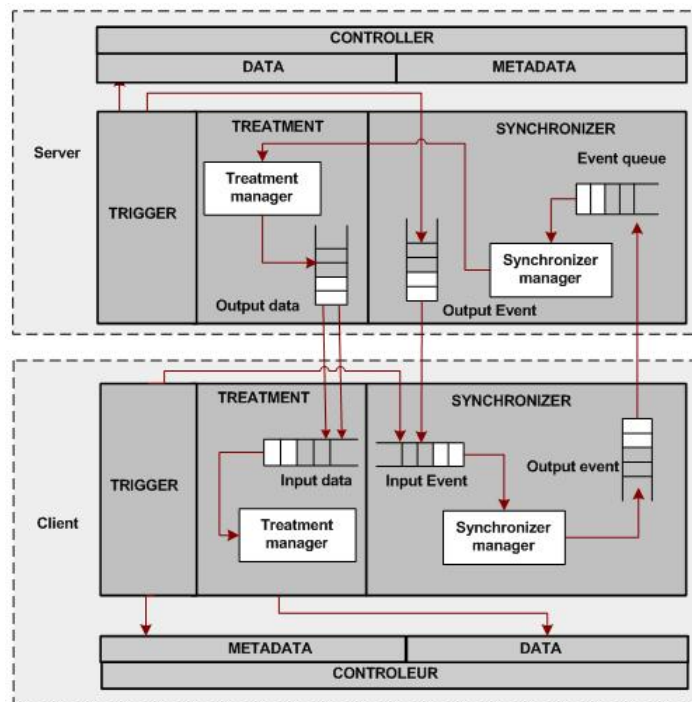


FIGURE 3.5 – La couche logique

3.3.1 Le module *Synchronizer*

La figure 3.5 illustre la vue détaillée du module *Synchronizer*. Le *synchronizer* est composé de deux files d'attente en entrée et sortie et le *Synchronizer ; Manager*. Les deux premières gèrent l'ordonnancement des requêtes alors que le deuxième exécute ces dernières. Ce module supporte les mécanismes Pull et Push ainsi que le mode mixte. Nous distinguons entre le fonctionnement à temporisateur statique et celui dynamique. Ces deux concepts correspondent à deux modes de fonctionnement relatifs au *Synchronizer* côté client et serveur. À l'établissement du contrat lors de la souscription, le mécanisme de décision chez le serveur informe le client du mode du fonctionnement adopté pour chaque son *Synchronizer*. À partir de cette décision, le client adopte l'une des deux stratégies suivantes :

Stratégie avec Temporisateur statique : Cette stratégie correspond au cas où le *Synchronizer* client est en mode passif et celui côté serveur est en mode actif. Le *Synchronizer* serveur étant le responsable du calcul du temporisateur. Le temporisateur est donc imposé par le serveur à chaque fois la mise à jour est envoyée au client. Nous proposons les deux méthodes suivantes.

- TTL
- Syndication module

Stratégie avec Temporisateur dynamique : Cette stratégie correspond au cas où le *Synchronizer* client est en mode actif et celui côté serveur est en mode passif voir même désactivé. Le *Synchronizer* client étant le responsable en plus de sa fonction de base qui est le polling une fois un événement sélectionné, il assume le calcul du temporisateur pour chaque flux. Le temporisateur étant dynamiquement calculé à partir d'une fonction optionnelle dans le *Synchronizer* et sera sauvegardé automatiquement comme paramètre de configuration pour chaque flux. L'apport de cette stratégie par rapport à celle statique est résumé dans le fait d'avoir un temporisateur dynamiquement calculé. Nous parlons donc de deux approches pour ce cas dynamique :

- Dynamic TTL : TTL + Heartbeat : déroulement de Heartbeat pour la détermination du *ttd*.
- Dynamic Syndication module : Syndication module + Heartbeat : déroulement de Heartbeat pour la détermination des paramètres *sy :updatePeriod*, *updateFrequency*...

3.3.2 Le module *Trigger*

L'objectif du *Trigger* est de créer les événements qui déclenchent la synchronisation. Nous distinguons deux types d'événements : ceux relatifs au client et désignant l'expiration du *timer* d'une part, et les événements créés au sein du serveur en but de notification d'autre part. Suite à cette formalisation de événements, nous spécifions le trigger client et celui du serveur.

Le *Trigger* client : Ce module fonctionne en cas du mode Pull et crée, sur la partie persistance des flux côté client, des événements appelés *Feed Events*. Ces *Feed Events* représentent une demande de mise à jour sur un flux désigné lorsque le *timer* expire au niveau client. Une fonction d'optimisation et de personnalisation est proposée pour la création des événements qui a pour objectif la sélection dynamique de la durée inter-fraîcheur. Ce paramètre représente la durée entre deux demandes de mises à jour sur un même flux.

Le *Trigger* serveur : En cas du mode Push, ce module étant un listener sur la couche *Metadata* côté serveur détectant les mises à jour sur les flux. Le trigger serveur a pour fonction donc la création des *Feed Events* qui sont des événements sur les mises à jour détectées. Ces événements auront comme information, les clients à notifier ainsi que le type de notification. Une notification pourra être soit une simple information d'une mise à jour faite, soit un polling même de cette nouvelle fenêtre du flux détectée. Une fonction de décision est proposée afin de choisir le type de notification chaque fois une mise à jour détectée. Les événements subissent une fonction d'optimisation avant d'être chargés dans une file d'attente dans le module *SynchroPush*.

3.3.3 Le module *Treatment*

Ce module est le composant responsable d'envoyer la mise à jour aux clients. Il est composé d'une file d'attente pour gérer la liste des requêtes et du *Treatment Manager* le module d'exécution. Ce dernier est le coeur de ce module.

3.4 La couche physique

La couche physique est essentiellement composée de deux modules : le *Data* et le *Meta Data*. Le module *Data* représente la couche de stockage des données relatives au flux. La partie *Meta Data* représente les données de configuration. Ces données sont évolutives à savoir l'état du serveur ou bien celle du client, le profil utilisateur etc...

Conclusion

Dans ce chapitre, nous avons proposé une architecture qui assure les modes Pull, Push et un mode mixte qui bascule entre les deux. L'architecture proposée est une architecture client - serveur. Dans le chapitre suivant, nous détaillons le système de syndication proposé.

Chapitre 4

Système de syndication pour le Web

Nous proposons dans ce chapitre le système de syndication pour le web WSS (Web Syndication System) à travers une spécification d'un protocole de syndication et d'un modèle de décision. WSS est un système de syndication qui a comme objectif principal la synchronisation du flux. Il s'appuie sur l'architecture présentée dans le chapitre précédent. Le protocole de syndication dans le Web WSP (Web Syndication Protocol) proposé au sein de WSS définit les différentes primitives de synchronisation et normalise les requêtes à utiliser. Dans une optique complémentaire, l'adoption d'un composant d'apprentissage et de décision vise à caractériser les paramètres déterminants dans notre système.

4.1 WSP : Protocole de Syndication dans le Web

4.1.1 Définitions et propriétés

Nous définissons le protocole WSP (*Web Syndication Protocol*) comme étant un protocole de syndication dans le web qui offre une spécification de plusieurs règles pour les primitives de synchronisation des flux. WSP se base sur XML comme norme d'échange de données pareil à *Atom Publishing Protocol* APP (Protocole de Publication de documents Atom). Ce dernier est un protocole de publication qui permet de mettre à jour les informations dans le Web. Il utilise le format Atom pour transporter des informations et repose sur l'architecture REST (Representational state transfer)[24] qui est une manière de construire une application pour les systèmes distribués comme le Web à la manière de *SOAP* [25] et de *XML – RPC*. Dans son optique spécifique à la synchronisation des flux, WSP se base sur l'architecture que nous avons définie dans le chapitre précédent.

Un client APP peut ainsi créer, modifier ou supprimer une ressource située sur un serveur APP en se basant sur le protocole de référence HTTP. Afin de synchroniser sa vue du flux, un client WSP peut soit accéder à une ressource disponible sur le serveur soit l'acquiescer selon le mode de synchronisation. Nous définissons ainsi des primitives spécifiques pour la synchronisation pour le protocole WSP.

4.1.2 Modèle : abstraction et besoins

WSP est positionné dans la couche applicative au sein des niveaux d'abstraction représentés dans la figure suivante. Il est modélisé avec ses trois primitives de synchronisation et se repose sur une diversité de protocoles au niveau plus bas. Cela désigne la capacité de ce protocole d'intégrer les méthodes offertes par les protocoles de base de la communication, la syndication et la messagerie. Nous déclarons aussi que WSP est un protocole générique de syndication, qui sera adapté à toute technologie utilisant les protocoles. Cet esprit de généricité se base sur l'intégration de la couche WSP-Divers qui permet l'adaptation des primitives WSP aux différents formats et normes. Cette couche offre des services de configuration et de paramétrage pour mieux passer les primitives WSP en mode fonctionnel tout en l'adaptant au protocole utilisé.

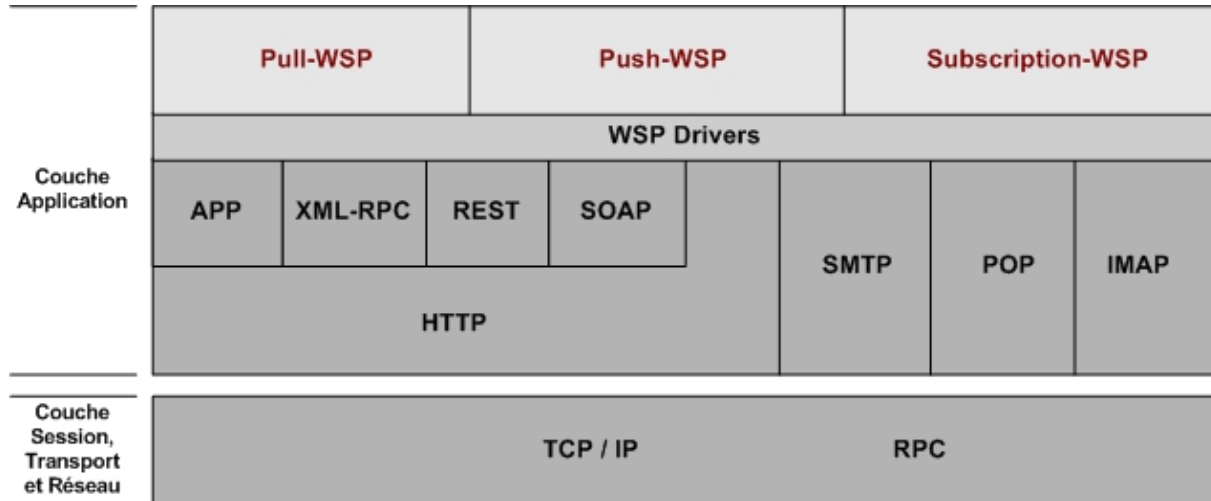


FIGURE 4.1 – Architecture

Il s'agit donc d'un modèle de protocole générique pour la syndication dans le web. Il est caractérisé par :

- la granularité de primitives riches.
- l'indépendance de toute plateforme.
- l'adaptabilité et la flexibilité à travers les WSP Drivers.

4.1.3 Définition et formalisation des primitives

La synchronisation des flux demande plusieurs services et fonctionnalités complémentaires ainsi qu'un protocole qui normalise les règles et les moyens de communication. Pour satisfaire cela, WSP présente plusieurs primitives qui diffèrent en leurs spécifications mais qui sont complémentaires en globalité.

Afin de mieux formaliser le protocole proposé, nous supposons le système formel suivant :

$$WSP = PullWSP + PushWSP + SubscriptionWSP$$

- SubscriptionWSP := { REST, HTTP, SOAP, SMTP, ... } (a)
- PullWSP := { REST, APP, POP, SMTP, HTTP, ... } (b)
- PushWSP := { REST, SMTP, HTTP, SMTP + POP, ... } := { C-Push-WSP + NC-Push-WSP } (c)
 - C-PushWSP := { REST, SMTP, HTTP, ... } (d)
 - NC-PushWSP := { REST, SMTP + POP, HTTP + IMAP, ... } (e)

À partir de ce système, le protocole WSP est représenté par les trois primitives citées. Les primitives associées à la souscription et au mode Pull peuvent être définies à travers plusieurs protocoles (a) et (b) respectivement. Cependant, la primitive du mode Push se subdivise en C-Push-WSP relative au mode connecté et NC-Push-WSP (c) qui est une primitive en mode non connecté. Chacune de ces deux sous-primitives peut être intégrée avec plusieurs protocoles spécifiques (d) et (e).

Ces fonctions sont définies à partir des requêtes ayant des paramètres spécifiques en demande et en réponse. Par ailleurs, les objectifs de ces requêtes diffèrent selon la primitive associée. Dans la section suivante, nous définissons les objectifs et les paramètres des différentes requêtes spécifiées pour les primitives WSP.

4.1.3.1 La primitive *Subscription – WSP*

Nous présentons à travers le tableau 4.1 les différentes requêtes proposées pour la souscription. Une première primitive visant le concept de souscription sera traitée par la primitive Subscription-WSP. Les concepts génériques sont les suivants :

- gestion de la découverte des flux ;
- gestion de la souscription à un flux ;
- gestion des contrats (granularité riche)

Cette primitive représente un service de base offert par ce protocole, c'est-à-dire que n'importe quelle utilisation du processus de synchronisation fasse un appel à cette primitive de souscription. Le tableau suivant illustre les requêtes proposées pour le *Subscription – WSP*.

Ref	Objectif	Requête	Request Params	Response Params
S1	Découverte des flux disponible dans un serveur (annuaire)	WSP-Discover	ServerId, ClientId, Criterion	[DataStreamState DataStreamId, DatastreamProdFreq]*, ServerState bandwidth, charge, ResponseTime
S2	Souscription à un flux	WSP-Subscribe	– StaticParams : DataStreamId, ClientId. – DynamicParams : ClientState bandwidth, charge, UserProfile freshNeed, completenessNeed, importance, SynchroConfig mode, TimerKind, AlertType	– StaticParams : DataStreamId. – DynamicParams : DataStreamWindow, DatastreamMetaData TTL,..., ServerState bandwidth, charge, ResponseTime, SynchroConfig mode, TimerKind, AlertType
S3	Changer l'état du client	WSP-ChangeClientState	ServerId, ClientId, ClientState bandwidth, charge	
S4	Changer l'état du serveur	WSP-ChangeServerState		ClientId, ServerId, ServerState bandwidth, charge, ResponseTime
S5	Changer le profil de l'utilisateur	WSP-ChangeUserProfile	ServerId, ClientId, UserProfile freshNeed, completenessNeed, importance	
S6	Demande de changer la configuration de la synchronisation	WSP-DChangeSynchroConfig	ServerId, ClientId, SynchroConfig mode, TimerKind, AlertType	
S7	Changer la configuration de la synchronisation	WSP-ChangeSynchroConfig		ClientId, ServerId, SynchroConfig mode, TimerKind, AlertType
S8	Changer la metadata du flux	WSP-ChangeDTMetaData		ClientId, ServerId, [DataStreamMetaData TTL,..]

TABLE 4.1 – Subscription-WSP Queries

4.1.3.2 La primitive *Pull-WSP*

La deuxième primitive est relative au mode de synchronisation *à Pull*. WSP intègre la primitive *Pull-WSP* afin de gérer l'opération de demande de mise à jour. A travers le tableau 4.2, Nous présentons les différentes requêtes proposées pour la primitive *Pull-WSP*.

Ref	Objectif	Requête	Request Params	Response Params
PL1	Demander la fenêtre disponible actuellement pour un flux	WSP-ActualWindowPULL	ServerId, ClientId	DataStreamWindow, [DataStreamMetaData]
PL2	Demander la fenêtre actuelle si elle est changée par rapport à celle du client (<i>Last – Modified</i>)	WSP-WindowConditional-PULL	ServerId, ClientId, If-Modified-Since	DataStreamWindow, Last-Modified, [DataStreamMetaData]
PL3	Demander une fenêtre limitée entre deux Items	WSP-WindowPULL	ServerId, ClientId, BeginItem, EndItem	DataStreamWindow, [DataStreamMetaData]
PL4	Demander les derniers Items ajoutés depuis <i>LastRefresh</i>	WSP-NewItemsPULL	ServerId, ClientId, LastRefresh	DataStreamWindow, [DataStreamMetaData]
PL5	Demander la mise à jour depuis une certaine date <i>FromDate</i> ou (<i>LastRefresh</i>)	WSP-FromDatePULL	ServerId, ClientId, LastRefresh	DataStreamWindow, [DataStreamMetaData]
PL6	Demander la mise à jour depuis un Item <i>FromItem</i>	WSP-FromItemPULL	ServerId, ClientId, FromItem	DataStreamWindow, [DataStreamMetaData]
PL7	Demander une mise à jour paramétrée sur un flux	WSP-ParameterPULL	ServerId, ClientId, [params]*	DataStreamWindow, [DataStreamMetaData]

TABLE 4.2 – Pull-WSP Queries

4.1.3.3 La primitive *Push-WSP*

Le mode push est défini dans le protocole par la primitive *Push-WSP* qui gère les opérations de diffusion de la mise à jour. Le tableau 4.3 illustre les requêtes proposées pour le Push-WSP.

Ref	Objectif	Requête	Response Params
PH1	Diffuser la fenêtre disponible actuellement pour un flux	WSP-ActualWindowPUSH	ClientId, ServerId, DataStreamWindow
PH2	Diffuser les derniers Items ajoutés depuis une date <i>FromDate</i>	WSP-NewsByDatePUSH	ClientId, ServerId, DataStreamWindow, FromDate
PH3	Diffuser les derniers Items ajoutés dans la fenêtre depuis <i>FromItem</i>	WSP-NewsByItemPUSH	ClientId, ServerId, DataStreamWindow, FromItem
PH4	Diffuser une fenêtre limitée entre deux Items	WSP-WindowPUSH	DataStreamWindow, BeginItem, EndItem
PH5	Diffuser une mise à jour paramétrée sur un flux	WSP-ParameterPUSH	ClientId, ServerId, DataStreamWindow

TABLE 4.3 – Push-WSP Queries

4.1.4 Diagramme de séquence de WSP

La figure 4.2 illustre un scénario en utilisant les primitives de WSP.

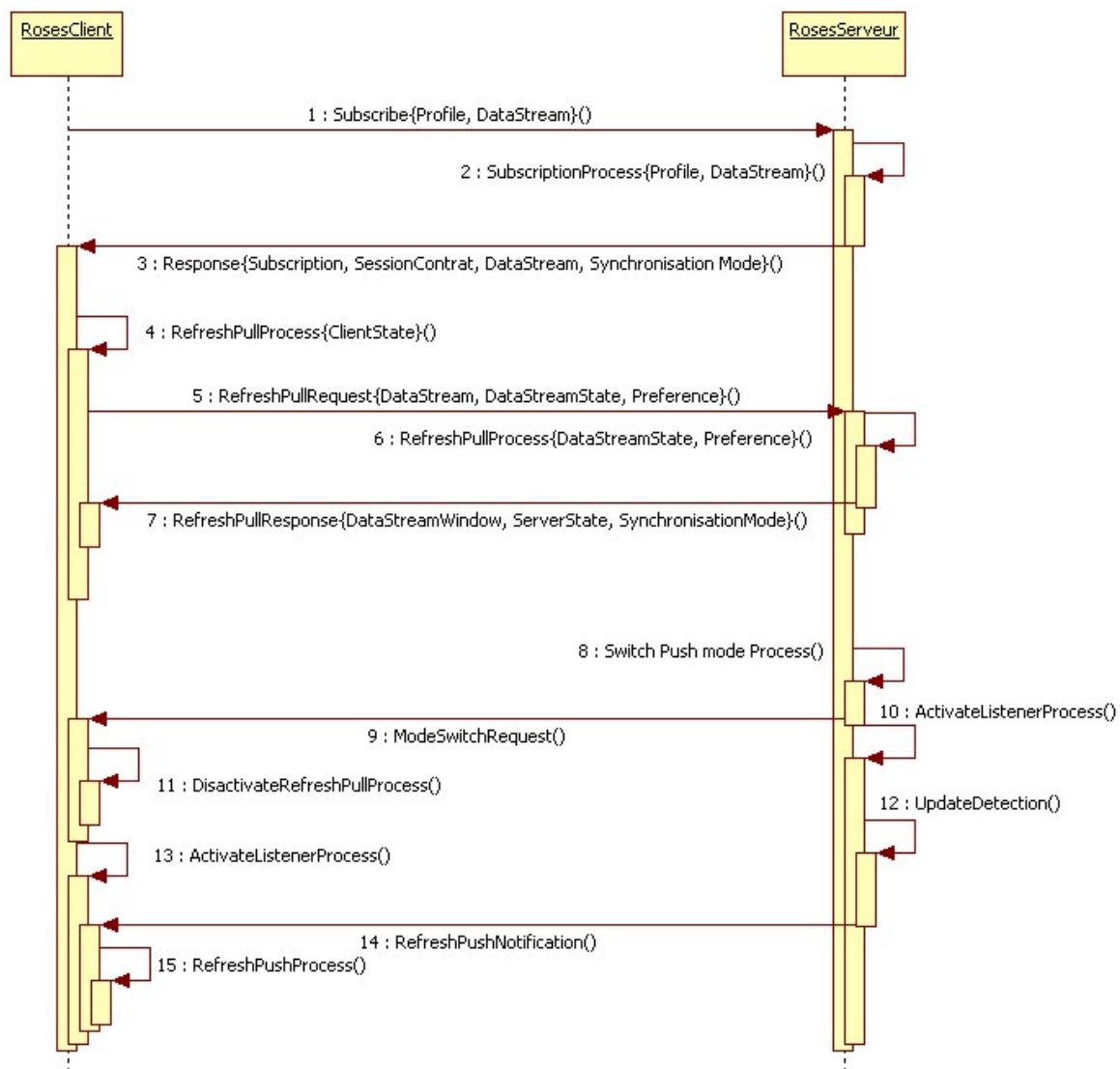


FIGURE 4.2 – Le diagramme de séquence de WSP

Concrètement, à travers le tableau 4.4, nous représentons les requêtes qui peuvent être utilisées dans chaque primitive.

Numéro séquence	Requête
1	S2
3	S2
5	PL1, PL2, PL3, PL4, PL5, PL6, PL7
7	PL1, PL2, PL3, PL4, PL5, PL6, PL7
9	S6, S7
14	PH1, PH2, PH3, PH4, PH5

TABLE 4.4 – Spécification des primitives de WSP

4.2 Modèle d'apprentissage et décision

4.2.1 Concepts de décision

Dans la section précédente, nous avons défini en premier lieu les primitives principales pour le protocole WSP et développé en second lieu les requêtes qui normalisent les typologies de messages utilisés. Nous avons donc bâti un protocole pour la synchronisation de flux qui se repose sur une architecture similaire côté client et serveur. Ce protocole permet de spécifier les requêtes normalisées utilisées dans le processus de synchronisation. Nous sommes demandés en seconde position de planifier un module de décision et d'apprentissage. Ce module aura comme objectif principal l'ordonnancement, la prise de décision et la mise en fonctionnement du processus de synchronisation.

Ce module en question est vu ainsi comme étant une troisième couche qui détermine des décisions. Ces dernières seront déroulées par le biais des primitives du protocole WSP représentant la deuxième couche. La plateforme d'exécution sera représentée par l'architecture de référence. Le système de décision que l'on propose se base sur trois règles. La définition de ces règles est indispensable pour caractériser les paramètres de la prise de décision dans WSS. La figure 4.3 illustre les paramètres de synchronisation. Nous définissons trois indices pour la qualité de service. L'indice de coût désigne le nombre de messages inutiles. L'indice de la complétude est proportionnel au nombre des items manqués. Le paramètre fraîcheur est évalué par le nombre de mises à jour faites entre deux rafraîchissements. A travers la spécification de la qualité de service, nous formalisons trois règles.

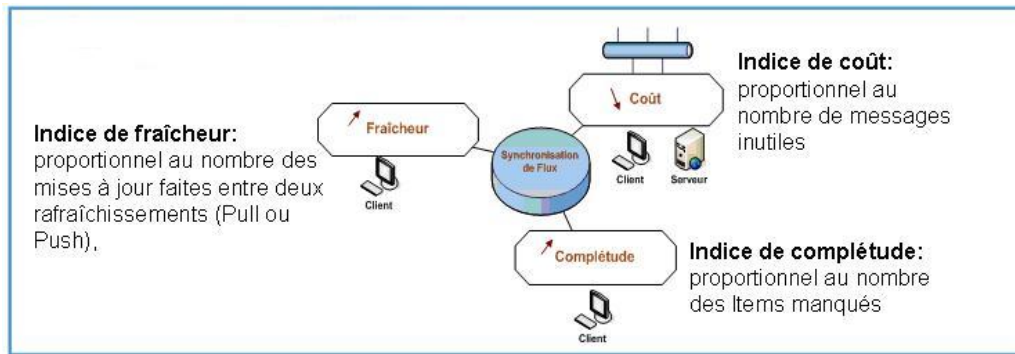


FIGURE 4.3 – Les paramètres d'évaluation

4.2.2 Règles de qualité de service

La qualité de service est définie par les critères coût, fraîcheur et complétude. Nous définissons des règles de calcul pour chaque critère. Nous définissons tout d'abord l'indice X par le rapport du nombre de mises à jour du flux par le nombre de rafraîchissements. Autrement, X est égale au nombre de mise à jour entre deux rafraîchissements. Principalement, X est définie par le rapport instantané de la fréquence de mise à jour et la fréquence de rafraîchissement.

$$X = \text{UpdateFreq} / \text{RafFreq} (= \text{UpdateNombre} / \text{RafNumber}) \quad (1)$$

(Exemple : pour Pull, $X \text{ t (t : temps) } = \text{UpdateFreq} / \text{PullFreq} = \text{Nombre de mise à jour par rapport au nombre de Pull dans un cycle} = \text{Nombre de mise à jour entre deux Pull}$).

Le coût : Nous formalisons, en premier lieu, la première règle relative à l'indice du coût. Ce dernier doit renseigner sur le nombre de messages inutiles dans une itération. Le but étant d'avoir une proportionnalité avec le nombre de mises à jour côté serveur. Y_{ct} , le paramètre du coût pour un cycle, est défini comme étant le rapport du nombre de rafraîchissement et le nombre de mises à jour.

$$Y_{ct} = \text{Cout}_i = \text{RafNumber}_i / \text{UpdateNumber} = 1 / X_i \quad (1)$$

Finalement, à partir de (1), Y_{ct} est égale à l'inverse de X .

La complétude : Rappelons que lors du mécanisme de synchronisation, un ou plusieurs processus de rafraîchissement peut se faire. Un rafraîchissement étant défini à un instant donné par le processus de transformation de la fenêtre disponible chez le serveur au client associé. C'est-à-dire, cela indique la mise à jour de la vue cliente à partir de la fenêtre disponible chez le serveur à cet instant. Etant donné que la fenêtre est un nombre fini des

items, un rafraîchissement donc est défini par une mise à jour de la fenêtre cliente par un ensemble des items disponible chez le serveur. A partir de ces rappels, le paramètre qui désigne la complétude peut être formalisé par le nombre moyen des Items perdus. Le système est en présence des items perdus si le serveur présente une fréquence de mise à jour très grande. Dans ce cas, ces items risquent de passer rapidement dans la fenêtre archive du serveur. Le rafraîchissement est incapable à ce moment de transférer tout ces items alors qu'ils étaient auparavant disponibles. On dit que le système est en présence d'un état d'incomplétude.

En supposant le déroulement du processus de synchronisation dans plusieurs itérations, nous pouvons mesurer cet état par le biais du paramètre Ycm avec la formule suivante :

$$Ycm = Completeness = 1 - \frac{\max(n_j - windowWidth), 0}{UpdateNumber}$$

$$n_j = ItemStartOfNewWindow(Raf_j) - ItemStartOfTheLatestWindowSaved(Raf_j - 1)$$

(par exemple : $n_j = ItemStartOfNewWindow(Pull_j) - ItemStartOfTheLatestWindowSaved(Pull_j - 1)$)

Nous pouvons l'écrire autrement avec : $Ycm = \min \left\{ 1, \frac{WindowWidth}{X} \right\}$

Nous adoptons que windowWidth designe la taille de la fenêtre du flux et la sommation s'effectue sur le nombre de rafraîchissement. Donc, nous montrons que :

$$Ycm = Completeness_i = 1 - \frac{\sum \max(n_j - windowWidth), 0}{UpdateNumber}$$

$$= 1 - \frac{\sum NombreItemsManques}{UpdateNumber}$$

(si on raisonne en terme de moyenne par rapport au nombre de Rafraîchissement)

$$= 1 - \frac{RafNombre * NombreMoyenItemsManques}{UpdateNumber}$$

$$= 1 - \frac{RafNombre * (X - WindowWidth)}{UpdateNumber}$$

$$= 1 - \frac{RafNombre * X}{UpdateNumber} + \frac{RafNombre * WindowWidth}{UpdateNumber}$$

$$= 1 - 1 + (RafNombre * WindowWidth / UpdateNumber)$$

$$= \frac{RafNombre * WindowWidth}{UpdateNumber}$$

$$= WindowWidth / X (Ycm \geq 1)$$

$$Ycm = Completeness_i = \min \left\{ 1, \frac{WindowWidth}{X} \right\}$$

Nous déduisons qu'à un instant t et pour un client, $Ycm = \min \left\{ 1, \frac{WindowWidth}{X} \right\}$ (2)

La fraîcheur : En plus des deux paramètres coût et complétude, le processus de synchronisation est évalué selon la fraîcheur de la fenêtre cliente. Nous définissons la fraîcheur par le nombre de mise à jour faites entre deux mises à jour successives. Afin de pouvoir exprimer ce paramètre, nous supposons qu'on est en présence toujours d'un déroulement successif du processus de rafraîchissement. A Partir de cette formule, nous désignons par Yfr le paramètre de la fraîcheur.

$$Yfr = Fraicheur_i = 1 - \sum \frac{\frac{PercentageOfLostWindow_j}{RafNumber_i}}{UpdateNumber}$$

$$PercentageOfLostWindow_j = \frac{LostUpdateNumber_j}{WindowWidth}, \sum jCycle$$

$$\text{Une autre écriture de cette règle : } Yfr = \max \left\{ 0, 1 - \frac{X-1}{WindowWidth} \right\}$$

Pendant chaque mise à jour, nous sommes supposés de calculer PercentageOfLostWindow. Donc, nous montrons que :

$$Yfr = Fraicheur_i = 1 - \sum \frac{\frac{PercentageOfLostWindow}{UpdateNumber}}{RafNumber} \text{ (si on raisonne en terme de moyenne par rapport au nombre des Updates)}$$

$$Yfr = Fraicheur_i = 1 - \frac{UpdateNumber * PourcentageMoyenOfLostWindow}{RafNumber}$$

$$\begin{aligned}
&= 1 - \frac{\frac{LostUpdateNumber}{WindowWidth}}{RafNumber} \\
&= 1 - \frac{\frac{UpdateNumbre - RafNumber}{WindowWidth}}{RafNumber} \\
&= 1 - \frac{UpdateNumbre - RafNumber}{RafNumber * WindowWidth} \\
&= 1 - \frac{\frac{UpdateNumbre}{RafNumber} - 1}{WindowWidth} \\
&= 1 - \frac{X-1}{WindowWidth} (Yfr \geq 0)
\end{aligned}$$

$$Yfr = Fraicheur_i = \max \left\{ 0, 1 - \frac{X-1}{WindowWidth} \right\}$$

Nous concluons qu'à un instant t on aura $Yfr = \max \left\{ 0, 1 - \frac{X-1}{WindowWidth} \right\}$ (3)

4.2.3 Méthodes de décision

Le système de décision sera responsable des tâches suivantes :

- le choix du mode de synchronisation initial (pendant la souscription) ;
- la date de rafraîchissement ;
- la décision de modification du mode (en cours de fonctionnement) ;
- la taille de fenêtre optimale demandée.

4.2.3.1 Mode de synchronisation initial

Le mode de synchronisation initial est précisé au moment de la souscription à travers l'établissement du contrat client. Nous concevons pour ce faire une fonction qui calcule le choix du mode de synchronisation. Ce choix est déterminé à partir de la situation actuelle du serveur détenteur du flux, de la proposition du client ainsi que de l'état du flux lui-même.

$PrefUser_i$ est une fonction spécifique selon le contexte utilisé. C'est-à-dire qu'elle a comme signature standard

$$PrefUser_i(FreshnessNeed, CompletenessNeed, importance)$$

$PrefUser_i$ a pour valeurs dans $[0..1]$ mais son comportement sera déterminé selon l'utilisation.

ModeFromPref est une fonction spécifique au contexte aussi avec la signature suivante :

$$ModeFromPref(pref - User - i)$$

Ses valeur appartiennent à $[0..1]$

Exemple :

$$PrefUser_i(FreshnessNeedfn, CompletenessNeedcn, importanceim) = (fn * 40 + cn * 50 + im * 10) / 100$$

Et ModeFromPref aura comme valeurs (déduction à partir des intervalles) :

- $[0..0, 3]$: Pull
- $[0, 3..0, 7]$: Push
- $[0, 7..1]$: Mixte

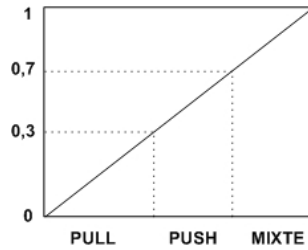


FIGURE 4.4 – Exemple simple de calcul du mode initial

La fonction *SynchroInitialModeDcision* reçoit en entrée une proposition du mode client. Son objectif essentiel, d'une part, est de tester la capacité du système à adopter ce mode en fonction du mode adopté en moyenne pour ce flux et la capacité du serveur. D'autre part, d'adopter une mesure de similitude des préférences du nouvel utilisateur avec celles des autres abonnés pour ce flux.

Algorithme 1 SynchroInitialModeD cision : Choix du mode initial de synchronisation

ENTR ES: Sub_i : la nouvelle souscription, F_j flux en question, $ModeProp_i$, le mode propos  par le nouvel abonn , $Mode_{F_j}$, le mode le plus utilis  par les abonn s de ce flux, $prefAllUsers_j$, la moyenne des pr f rences utilisateurs pour tous les abonn s de ce flux

SORTIES: $InitialSynchroMode_{i,j}$: le mode initial de synchronisation pour le nouvel abonn 

```

si  $Mode_{F_j} = \emptyset$  alors
  // cas de nouveau flux ou pas de mode plus utilis  en moyenne
  si  $chargeServeurForMode(ModeProp_i) = OK$  alors
     $InitialSynchroMode_{i,j} = ModeProp_i$ 
  sinon
     $InitialSynchroMode_{i,j} = PULL$ 
  fin si
sinon
  // on regarde le cas ou il y a un mode par d faut pour ce flux
  si  $Mode_{F_j} = ModeProp_i$  ou  $prefUser_i$  proche  $prefAllUsers_j$  alors
    si  $chargeServeurForMode(Mode_{F_j}) = OK$  alors
       $InitialSynchroMode_{i,j} = Mode_{F_j}$ 
    sinon
       $InitialSynchroMode_{i,j} = PULL$ 
    fin si
  sinon
    si  $Mode_{F_j} \neq ModeProp_i$  alors
      si  $chargeServeurForMode(ModeFromPref(prefUser_i)) = OK$  alors
         $InitialSynchroMode_{i,j} = ModeProp_i$ 
      sinon
         $InitialSynchroMode_{i,j} = PULL$ 
      fin si
    fin si
  fin si
return  $InitialSynchroMode_{i,j}$ 

```

4.2.3.2 La date de rafraîchissement

Le calcul de la date du prochain rafraîchissement se base sur deux approches. La première est utilisée en cas d'un temporisateur statique et la seconde pour le contexte dynamique. Un temporisateur est dit statique si la date de rafraîchissement sera statiquement calculé durant tout le contrat établi entre le client et le serveur. Et le temporisateur désigne le changement de cette date durant le processus. Dans ce cas, le recalcul de la date aura pour objectif la flexibilité du système pour avoir une meilleure performance.

Comme l'indique le tableau suivant, la méthode statique *DateRefreshStaticDecision* est destinée pour le serveur vu qu'il est le seul à indiquer la date de rafraîchissement statique dans ce cas. La méthode de détermination dynamique de la date du prochain rafraîchissement sera utilisée par le serveur et tout de même par le client. Ceci est dû au fait que ces deux derniers peuvent décider de changer leur instant de rafraîchissement selon plusieurs critères à définir dans la suite.

Calcul statique avec la méthode *DateRefreshStaticDecision* : Il s'agit donc de prédire la date de rafraîchissement (*RefreshDate* : variable à prédire) en fonction des données en entrées suivantes : l'état du serveur (*ServerStatebandwidth*, charge), l'état du flux (*DatastreamStateidDatastream*, *modifiedItems*, importance), l'état et le profil du client (*usersState[ClientDecisionalertType*, frequency, *ClientProfilefreshNeed*, completeness-Need,importance]).

Cette méthode adopte l'apprentissage supervisé pour déterminer la date du prochain rafraîchissement. L'apprentissage supervisé qui est une technique d'apprentissage automatique où l'on cherche à produire automatiquement des règles à partir d'une base de données d'apprentissage contenant des exemples de cas déjà traités. Le but de la méthode d'apprentissage supervisé est alors d'utiliser cette base d'apprentissage afin de déterminer une représentation compacte de f , une fonction de prédiction, qui à des nouvelles entrées $x[]$ associe une sortie $f(x[])$: la date du prochain rafraîchissement. Le but de cet algorithme d'apprentissage supervisé est donc de généraliser pour des entrées inconnues ce qu'il a pu apprendre grâce aux données déjà traitées par des simulations déjà faites, ceci de façon raisonnable. Nous choisissons l'arbre de décision comme outil d'aide à la décision dans le cadre de l'apprentissage supervisé. Il permet de modéliser simplement, graphiquement et rapidement un phénomène mesuré plus ou moins complexe. Sa lisibilité, sa rapidité d'exécution et le nombre réduit d'hypothèses nécessaires a priori justifient notre choix.

Calcul dynamique avec la méthode *DateRefreshDynamicDecision* : Pour cette méthode, nous adoptons l'apprentissage par renforcement. Ce type d'algorithme fait référence à une classe de problèmes d'apprentissage automatique. Le but est d'apprendre, à partir d'expériences, le meilleur choix de la date de rafraîchissement, de façon à optimiser les paramètres de qualité (coût, fraîcheur et complétude).

Un paradigme classique pour présenter les problèmes d'apprentissage par renforcement consiste à considérer un agent autonome, plongé au sein d'un environnement, et qui doit prendre des décisions en fonction de son état courant. À partir de cela, vu que cette méthode est utilisée par le client ainsi que pas le serveur, ces deux derniers cherchent un comportement décisionnel. Au travers d'expériences itérées, ce comportement est initialement résultant d'une perception de l'état courant. Suite à une évaluation de ce comportement, l'agent exécute une action de récompense qui est le changement de la date de rafraîchissement. Cette méthode est servie pour améliorer le processus de synchronisation. Cette méthode prend exactement les mêmes paramètres que la méthode statique à l'exception de l'état du serveur. Cet état est représenté par sa fréquence de production du flux et l'historique de son temps de réponse (*ServerStateDatastreamProductionFrequency*, *ResponseTime(History)*).

4.2.3.3 Modification du mode

La modification du mode dans notre système est un résultat immédiat à l'incapacité du système, pour un instant donné, de fournir une bonne qualité de service. Garantir une telle qualité de service étant l'objectif essentiel de l'adoption d'un tel mode par rapport à un autre. Nous rappelons ainsi que la réalisation du processus de synchronisation doit assurer un coût minimum, une grande fraîcheur et une haute complétude. L'atteinte d'une certaine qualité de service souhaitée ne sera pas garantie simplement par l'adoption d'un seul mode de synchronisation. Etant donné la plateforme d'exécution ainsi que l'état courant du serveur, la réalisation d'une telle qualité de service devient difficile et même impossible avec le mode utilisé en cours.

À partir de ce moment, le système doit avoir une réaction d'autocorrection et de récompense. Nous avons conçu pour cela une méthode de modification du mode. Cette méthode sera adoptée, vu leurs droits de changer le mode courant, par le client ainsi que par le serveur. Nous nous basons aussi sur les primitives du protocole WSP qui sont relatives au changement du mode de synchronisation. Cette méthode se base sur l'apprentissage par renforcement. Le client et le serveur sont représentés comme des agents. Nous appliquons spécifiquement l'algorithme standard de l'apprentissage par renforcement. L'amélioration de la qualité de service est relative au changement du mode utilisé. Donc, c'est la capacité de perception qui déduit cette nécessité. Par la suite, après une évaluation du bon

choix du mode, le système décide de se basculer vers le mode nouvellement. Cela peut être résumé par l'action de récompense offerte par cette méthode afin d'améliorer la qualité de service du processus de synchronisation.

4.2.3.4 Taille de la fenêtre optimale demandée

La fenêtre du flux représente le paramètre principal à gérer dans tout le processus de synchronisation. Nous avons distingué entre trois types de fenêtres à gérer côté serveur, la fenêtre disponible, celle non encore publiée et la fenêtre archivée. Pour le client, nous avons conçu un seul type qui est la fenêtre disponible. Le processus de synchronisation vise à rafraîchir cette fenêtre disponible vue client à partir de celle du serveur. La taille de la fenêtre paraît important dans le processus de synchronisation à partir du moment où elle figure dans les formules des indices de la fraîcheur et de la complétude. Cela nous ramène à prouver théoriquement que la taille de la fenêtre est un paramètre obligatoire dans le processus de synchronisation. Il joue un rôle essentiel dans la qualité de service offerte par le processus en question. Ce dernier demande donc une taille appropriée de la fenêtre permettant l'atteinte de la qualité optimale. Nous pouvons déduire ainsi que cette taille peut être préalablement calculé à partir des règles définies précédemment. D'une part, à partir de 2 et 3, ce paramètre peut être écrit sous forme :

$$WindowWidth = \frac{Completeness}{Completeness * (Freshness - 1) + 1}$$

D'autre part, 2 permet de déduire que :

$$WindowWidth = Completeness * X$$

Ou encore avec 3 :

$$WindowWidth = \frac{X - 1}{1 - Freshness}$$

La première formule signifie que la taille de la fenêtre peut être calculée à partir des deux paramètres de la qualité, tandis que les deux autres indiquent qu'il est relatif avec les fréquences serveur et client.

Conclusion

Dans ce chapitre, nous avons pu ajouter à l'architecture de référence notre système de syndication WSS à travers un protocole et un module de décision. Nous avons normalisé avec *WSP* les primitives de synchronisation utilisées dans WSS. Le module prise de décision gère plusieurs fonctions permettant d'améliorer la qualité de service souhaitée à travers le processus en question. Dans cette perspective, les règles de syndication trouvées permettent évidemment de préconiser un système intelligent de syndication. Dans la suite de ce mémoire, nous allons tester, prouver et analyser ces résultats théoriques à travers des simulations. Notre objectif sera de prouver que *WSS* est un système configurable qui, selon une qualité demandée, sera capable d'établir un contrat en résultat.

Chapitre 5

Résultats expérimentaux

Dans le chapitre précédent, nous avons clarifié les concepts du système de syndication proposé et appliqué sur l'architecture de référence. WSS est essentiellement utilisé pour la synchronisation des flux. Il est basé sur un protocole de syndication et un module d'apprentissage et de décision. Le protocole se base sur des primitives de services destinées à la synchronisation. Le module de prise de décision de bases sur des fonctions et des règles qui consistent à améliorer le processus de synchronisation. Ces règles traitent la qualité de services du processus de synchronisation. Elles sont déterminées pour les paramètres d'évaluation suivant : coût, fraîcheur et complétude. Dans cette perspective, notre objectif à travers ce chapitre consiste à prouver expérimentalement notre approche.

5.1 Environnement et éléments de configuration

5.1.1 Scénario global

Nous essayons à travers une simulation de valider l'approche théorique élaborée dans les deux chapitres précédents. Nous visons déterminer une analyse du système s'il est en présence d'un processus de synchronisation. Cette analyse se base essentiellement sur l'évaluation des différentes règles relatives à la qualité de service. Nous avons implémenté donc l'architecture de référence en utilisant les services web qui réalisent une plateforme de communication orientée service (SOA). (Une mise en oeuvre sera mieux détaillée en Annexe A). Le scénario général de la simulation consiste à avoir un seul flux actif (mise à jour du flux fréquente). Pour ce faire, nous avons opté à implémenter une moulinette de production de flux. Elle produit un item par un laps de temps. Ce laps déterminera la fréquence de mise à jour du flux côté serveur. Ce dernier étant le détenteur du flux en question. Nous supposons adopter ainsi le schéma proposé au début du mémoire représentant les trois types de fenêtres chez le serveur.

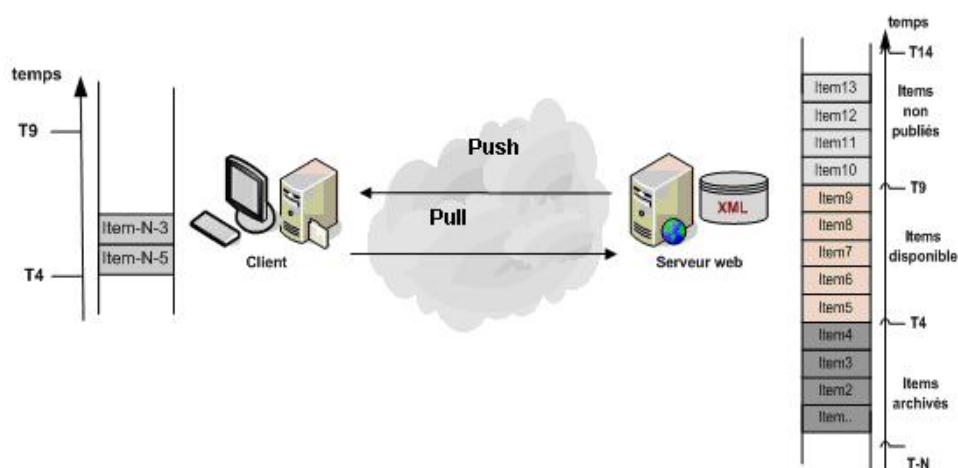


FIGURE 5.1 – Le scénario de la simulation

La simulation comprend aussi un client qui souscrit à ce flux. Il présente une vue cliente représentée par une fenêtre sur le flux. La synchronisation de la vue cliente pour ce flux se base sur les trois modes proposés auparavant.

Le processus de rafraîchissement se fait par rapport à l'ensemble des Items disponible à chaque instant chez le serveur. La simulation consiste donc à exécuter un nombre au choix de cycles. Un cycle représente un nombre fini de plusieurs itérations. Notons que dans un cycle, le nombre de rafraîchissements désigne le nombre de mises à jour effectuées sur la fenêtre actuelle du flux côté client. Ce nombre correspond aussi au nombre d'itération dans un cycle qui est supposé fixé dans toute la simulation. On dit que la taille d'un cycle est constante durant l'évaluation. Une itération représente donc un déroulement répétitif du mécanisme de rafraîchissement (pull, push ou mixte) en parallèle avec le mécanisme de mise à jour du flux. Dans chaque cycle, nous modifions le rapport entre la fréquence de rafraîchissement et la fréquence de mise à jour du flux. Nous adoptons les variables de configuration suivantes :

- Taille de la fenêtre disponible : `windowWidth` ;
- La fréquence de mise à jour de flux côté serveur : `updateFreq` ;
- La fréquence de pull : `pullFreq` ;
- La fréquence de Push : `pushFreq` ;
- Taille d'un cycle : Nombre de mise à jour de flux par cycle : `UpdateNumber` ;

Après avoir défini l'environnement technique de la simulation, nous associons les valeurs suivantes aux variables de configuration :

- `updateFreq` = 1/10 s
- `WindowWidth` = 10 Items = fixe
- `UpdateNumber` = `cycleWidth` = 200

5.1.2 Paramètres de qualité

Nous rappelons les règles de qualité de services créés dans le chapitre précédent. La formule initiale est $X = \text{UpdateFreq} / \text{RafFreq}$. Dans un cycle, le nombre d'itération fait que $X = \text{UpdateNumber} / \text{RafNumber}$.

- Le paramètre coût est évalué par la formule : $Yct = \text{Coût}_i = \text{RafNumber}_i / \text{UpdateNumber} = 1/X_i$, avec i référence le cycle.
- La complétude est définie par $Ycm = \text{Complétude}_i = 1 - [\max(nj - \text{windowWidth}, 0) / \text{UpdateNumber}]$ avec $nj = [\text{ItemStartofnewwindow}(\text{Refresh}_j) - \text{ItemStartofthelatestwindowsaved}(\text{Refresh}_j - 1)]$, la sommation se fait par rapport au nombre d'itérations dans un cycle. Nous pouvons écrire autrement $Ycm = \min(1, \text{WindowWidth} / X)$.
- La fraîcheur est évaluée par $Yfr = \text{Fraîcheur}_i = 1 - [((\text{PercentageOfLostWindow}_j / \text{RafNumber}_i) / \text{UpdateNumber})]$, la mesure de la fraîcheur avec $\text{PercentageOfLostWindow}_j = (\text{LostUpdateNumber}_j / \text{WindowWidth})$, et la sommation s'effectue par rapport au nombre de mise à jour. Yfr peut s'écrire aussi $Yfr = \max(0, 1 - [(X - 1) / \text{WindowWidth}])$.

5.2 Analyse des résultats

La figure 5.2 illustre les courbes résultantes trouvées suite à notre simulation faite avec la configuration montrée préalablement. Nous trouvons les mêmes courbes à une mesure expérimentale près en utilisant les deux modes de synchronisation pull et push. Nous évaluons ainsi les éventuels impacts sur les courbes en jouant sur les paramètres de configuration. Désormais, nous concluons que tout changement des paramètres relatifs aux fréquences n'aura pas d'influence sur les courbes. Cela est justifié vu la récompense des fréquences à travers le rapport $X = \text{UpdateFreq} / \text{RafFreq}$. Néanmoins, le paramètre taille de la fenêtre disponible est un paramètre déterminant de la synchronisation. En effet, tout changement de `WindowWidth` entraîne une translation horizontale du point A dans la courbe complétude et un changement au niveau des pentes des deux autres courbes. A partir de cette diduction, nous aurons une possibilité d'ajuster ce paramètre en fonction de notre préférence des courbes. L'analyse de ces courbes consiste aussi à dire que nous pouvons garantir une efficace synchronisation et cela selon 3 situations possibles :

1. Si nous souhaitons une fraîcheur $\text{FraîchIndex} = \text{FraîchValeur}$, à partir des courbes :
 - En projetant la parallèle à l'axe des abscisses ayant comme valeur FraîchValeur en ordonnée sur la courbe du coût, nous déterminons la valeur CoûtValeur nécessaire.
 - En ajustant `WindowWidth` pour avoir l'intersection entre la courbe de fraîcheur et celle de la complétude en ordonnée FraîchValeur , nous déterminant ainsi une valeur maximale de la complétude dans ce cas.
2. Si nous souhaitons une complétude $\text{ComplétudeIndex} = \text{ComplétudeValeur}$, à partir des courbes :
 - Nous pouvons déterminer à partir de la courbe de complétude la valeur `WindowWidth` nécessaire dans ce cas (si $\text{ComplétudeValeur} = 100\%$ alors `WindowWidth` = [1..Infini])
 - En projetant la parallèle à l'axe des abscisses ayant comme valeur ComplétudeValeur en ordonnée sur la courbe du coût, nous déterminons la valeur CoûtValeur nécessaire.
 - En projetant la même droite sur la courbe de fraîcheur, nous déterminons la valeur maximale de la fraîcheur dans ce cas. (si $\text{ComplétudeValeur} = 100\%$ alors $\text{fraîcheurValeur} = 100$)

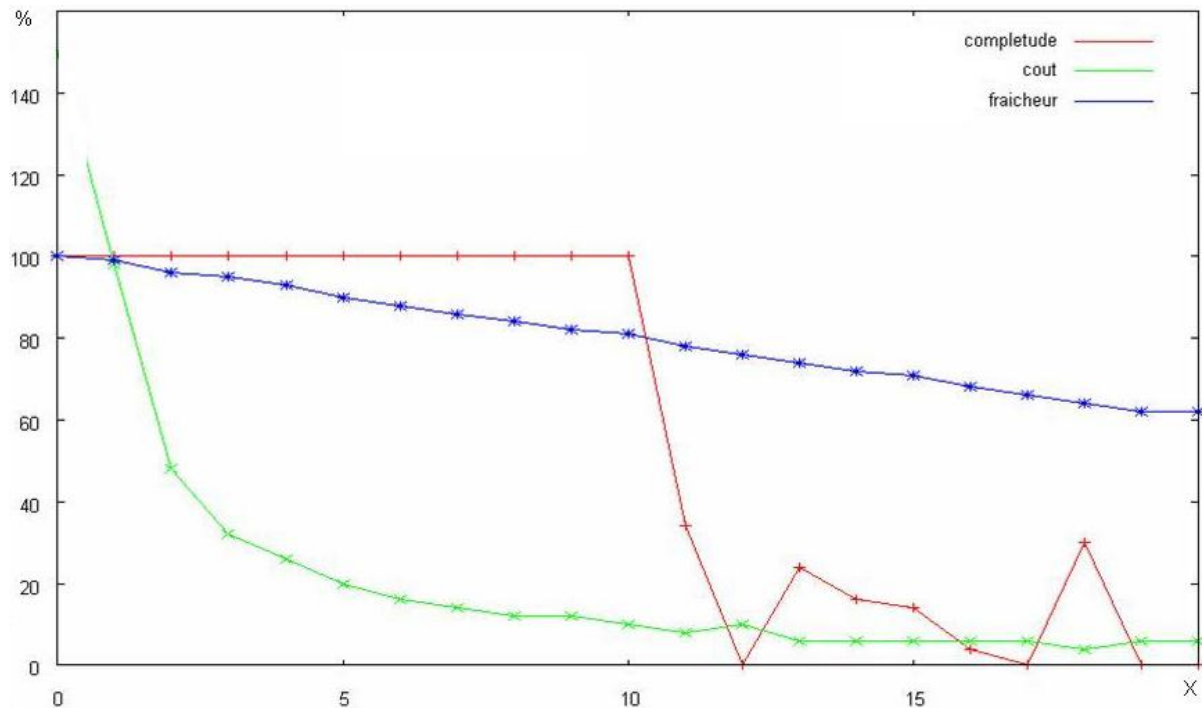


FIGURE 5.2 – Les courbes de la simulation

3. Si nous connaissons le coût disponible $\text{CoûtIndex} = \text{CoûtValeur}$, à partir des courbes :
 - En projetant la parallèle à l'axe des abscisses ayant comme valeur FraîchValeur en ordonnée sur la courbe du coût, nous déterminons la valeur FraîchValeur maximale pour ce coût.
 - En ajustant WindowWidth pour avoir l'intersection entre la courbe de coût et celle de la complétude en ordonnée FraîchValeur , nous déterminons une valeur maximale de la complétude dans ce cas. (Si $\text{CoûtValeur} = 100\%$ alors n'importe quel valeur de WindowWidth peut nous satisfaire)

A partir de ces situations, nous déduisons que le système WSS aura cette analyse comme une référence pour son système d'apprentissage et décision. La création d'un système à base de règles sera facile en adoptant les différents scénarios présentés auparavant et de formaliser les règles d'exécution appropriées. Nous remarquons que les deux modes Pull ou Push peuvent valider ces règles. A partir d'un système à base de règle, nous pouvons déterminer le rapport fréquence de rafraîchissement et fréquence de mises à jour en tenant compte des préférences utilisateur (coût, fraîcheur et complétude). Dans une autre perspective, pour Push, vu que le système connaît updateFreq , nous pouvons déterminer la date du prochain rafraîchissement. Par ailleurs, en cas de pull, vu que le système n'a pas de connaissance sur updateFreq , nous pouvons déduire la nécessité du système d'apprentissage automatique proposé dans le système WSS. Nous dégageons ainsi l'utilité de changement du mode. Dans certains cas, il est utile d'utiliser un mode et non pas l'autre. Nous énumérons ces cas dans ce qui suit :

- Dans la situation (1), si $\text{FraîchValeur} = 100\%$, sauf Push peut garantir ce cas (vu que dans pull, le système ne connaît pas updateFreq).
- Dans la situation (2), si $\text{ComplétudeValeur} = 100\%$, seulement Push peut garantir ce cas (toujours c'est la même raison).
- Dans la situation (3), Si $\text{CoûtValeur} = 100\%$ et WindowWidth est infinie, il sera utile d'adopter Push.

Conclusion

Nous concluons que la fenêtre disponible représente un paramètre déterminant dans la synchronisation des flux. Nous pouvons ainsi prévoir le comportement de système et calculer à priori les variables de configuration selon nos besoins en qualité de service :

- Paramètres de configuration = Fonction (windowWidth , updateFreq , FreshFreq)

Nous pouvons ainsi personnaliser la configuration du système qui satisfait nos besoins :

- Qualité de service = Fonction (Paramètres de configuration).

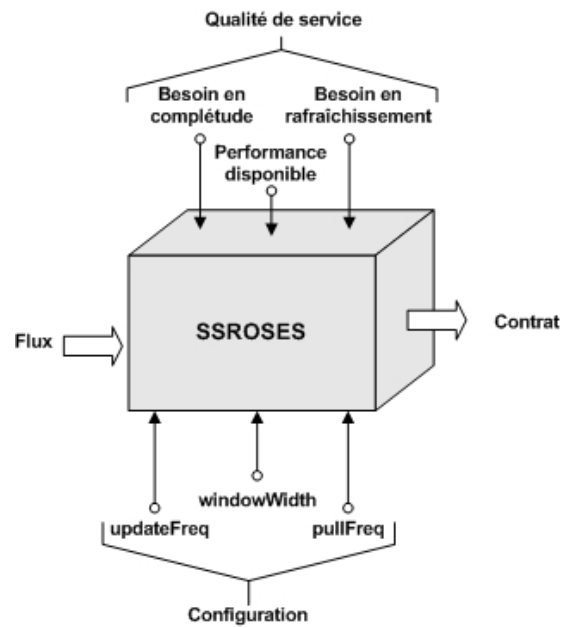


FIGURE 5.3 – Le système personnalisable WSS

Nous obtenons finalement un système personnalisable (voir figure 5.3) qui permet d'obtenir les paramètres de configuration à partir des préférences en qualité de service. Ce système consiste à générer un contrat utilisateur pour le client abonné à ce système.

Chapitre 6

Conclusion

Notre approche se positionne dans le contexte indépendant. Elle consiste en l'utilisation de la diversité des modes de synchronisation.

- Nous avons proposé une architecture de référence pour la synchronisation des flux. C'est une architecture standard qui est utilisable côté client et serveur. Elle présente par la suite une plateforme de base pour le système WSS (Système de Syndication dans le Web) proposé dans le troisième et le quatrième chapitre. WSS est le système de syndication proposé qui se base sur un protocole WSP (Web Syndication Protocol) et un module d'apprentissage et de décision.
- Les spécificités de WSP se résument au fait qu'il est un protocole générique pour la synchronisation de flux. Il présente des primitives spécifiques à WSS pour gérer la souscription et les modes de rafraîchissement.
- Le module décisionnel étant aussi spécifique en termes de méthodes proposées. La décision concerne évidemment le choix de la date de rafraîchissement, le choix et le changement du mode de synchronisation ainsi que l'ajustement de la fenêtre de rafraîchissement.
- Nous avons pu aussi simuler et évaluer l'approche proposée et tester le système WSS. Les tests sont basés essentiellement sur les différentes règles de qualité de service proposées. L'étude a montré que WSS est un système pouvant être personnalisable selon nos besoins. Il présente deux modes de fonctionnement. Le premier étant le fonctionnement pour la génération de la configuration nécessaire pour garantir la qualité de service demandée. Le deuxième est relatif à la gestion efficace de la qualité de service en utilisant la configuration système courante. Avec le deuxième mode de fonctionnement, le système remède à la problématique de départ avec ses trois facettes.
- A travers le module décisionnel, WSS permet le choix du meilleur mode initial de synchronisation selon la performance disponible. Le changement du mode permet de garantir la vérification des trois paramètres de qualité. Le module d'apprentissage sera donc le responsable de la rectification des positions client et serveur.

Nous concluons que WSS permet d'être à la hauteur dans le cas d'une activité importante de flux avec une montée en charge énorme au niveau serveur. Cela peut se faire intuitivement par l'ajustement de la fenêtre de rafraîchissement. Un autre scénario permet aussi de valider la capacité de WSS : c'est dans le cas de plusieurs clients qui utilisent le même serveur. Dans cette situation, le module d'apprentissage sera déclenché en parallèle avec celui de la décision. Cela permet, d'abord, en utilisant le premier mode de fonctionnement, de mesurer selon les indices maximums de qualité de service demandés, la configuration du système nécessaire. Ensuite, de régulariser la configuration du système pour atteindre les indices de qualité de services calculés. Ceci est garanti par le deuxième mode de fonctionnement de WSS.

A travers ce scénario, nous voulons prouver la capacité de ce système à faire face aux problèmes de la synchronisation dans le contexte indépendant. Les mêmes problèmes peuvent être aussi résolus dans le cas de la synchronisation coopérative vu que c'est la même architecture utilisée côté serveur et client. Donc, un client peut facilement être un serveur, le même fonctionnement de WSS sera utilisé aussi pour faire face aux problèmes déjà présentés. Mais, WSS nécessite seulement un module de gestion de cache coopérative qui permet de résoudre l'aspect de l'utilisation du cache coopératif.

Bibliographie

- [CGM03] Junghoo Cho and Hector Garcia-Molina. Effective page refresh policies for Web crawlers. *ACM Transactions on Database Systems*, 28(4) :390–426, December 2003.
- [Cou07] Serge Courrier. *Utiliser les fils RSS et Atom*. ADBS éditions, 2007.
- [Édi06] MM2 Éditions. *RSS, blogs*. DL, 2006.
- [dL08] Guy de Lussigny. *Flux RSS et podcast*. Éditions ENI, 2008.
- [JA06] Seung Jun and Mustaque Ahamad. Feedex : collaborative exchange of news feeds. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *WWW*, pages 113–122. ACM, 2006.
- [KLJS04] Su Myeon Kim, Jinwon Lee, SungJae Jo, and June-hwa Song. Content aggregation middleware (CAM) for fast development of high performance web syndication services. In Hai Jin, Guang R. Gao, Zhiwei Xu, and Hao Chen, editors, *NPC*, volume 3222 of *Lecture Notes in Computer Science*, pages 195–204. Springer, 2004.
- [KMMS07] Firat Kart, Louise E. Moser, and P. Michael Melliar-Smith. Reliable data distribution and consistent data replication using the atom syndication technology. In Hamid R. Arabnia, Victor A. Clincy, Joan Lu, and Jan Smid, editors, *International Conference on Internet Computing*, pages 124–132. CSREA Press, 2007.
- [PV06] George Pallis and Athena Vakali. Insight and perspectives for content delivery networks. *Commun. ACM*, 49(1) :101–106, 2006.
- [SMPD05] Daniel Sandler, Alan Mislove, Ansley Post, and Peter Druschel. Feedtree : Sharing web micronews with peer-to-peer event notification. In Miguel Castro and Robbert van Renesse, editors, *IPTPS*, volume 3640 of *Lecture Notes in Computer Science*, pages 141–151. Springer, 2005.
- [WA] Van Wanrooij and A.Pras. Scheduling feed retrieval. *IEEE International Conference on*, 28(4) :2064 – 2069.
- [Wit06] Heinz Wittenbrink. *RSS et Atom*. Eyrolles, 2006.

Nétographie

[NG-1] <http://www.rss-one.com/>
[NG-2] <http://www.kbcafe.com/rss/rssfeedstate.html>
[NG-3] http://www.mnot.net/cache_docs/index.fr.html
[NG-4] <http://www.bd.lip6.fr/roses/doku.php>
[NG-5] http://www.actifpub.com/code_blog.php
[NG-6] http://www.monsitegratuit.com/services/rss_manager/index.php
[NG-7] <http://creer-un-site.fr/lecture-d-un-flux-rss.php>
[NG-8] <http://www.metacites.net/mot25.html>
[NG-9] <http://www.sebsauvage.net/comprendre/rss/creer.html>
[NG-10] <http://reflink.fr/>
[NG-11] <http://creer-un-site.fr/creer-un-annuaire-gratuit.php>
[NG-12] <http://developer.yahoo.com/rss/>
[NG-13] <http://www.feedroom.com/product.aspx?mode=products>
[NG-14] <http://www.strongangel3.net/sse>
[NG-15] http://www.intertwingly.net/wiki/pie/SamRuby_2fMoinEditorBackup?action=highlight&value=Rss20AndAtom10Compared
[NG-16] <http://www.clubic.com/telecharger/windows/flux-rss/>
[NG-17] <https://blogapps.dev.java.net/>
[NG-18] http://feedvalidator.org/docs/howto/install_and_run.html
[NG-19] <http://dev.live.com/feedsync/>
[NG-20] <http://rss.indexweb.info/>
[NG-21] http://www.dmoz.org/Computers/Internet/On_the_Web/Syndication_and_Feeds/RSS/
[NG-22] <http://www.xul.fr/xml-rss.html>
[NG-23] <http://crinfo.univ-paris1.fr/id64.htm>
[NG-24] http://www.journaldunet.com/developpeur/tutoriel/xml/030707xml_rest1a.shtml
[NG-25] <http://www.w3.org/TR/soap/>

Annexe A

Mise en oeuvre de l'approche proposée

Introduction

Dans cette annexe nous présentons la mise en oeuvre de notre approche. Nous commençons par présenter la spécification du système avec les diagrammes de cas d'utilisation, la modélisation avec la carte [23] et le diagramme de séquence global. Nous présentons ensuite les scénarii du protocole WSP à travers quelques diagrammes de séquence. Nous montrons ainsi une conception de la partie *persistance* (Data et MetaDate) de l'architecture proposée avant de présenter l'hierarchie conçue pour l'implémentation.

Spécification de l'approche

La figure 6.1 illustre les cas d'utilisation de notre approche. Nous distinguons plusieurs processus de base dans le système.

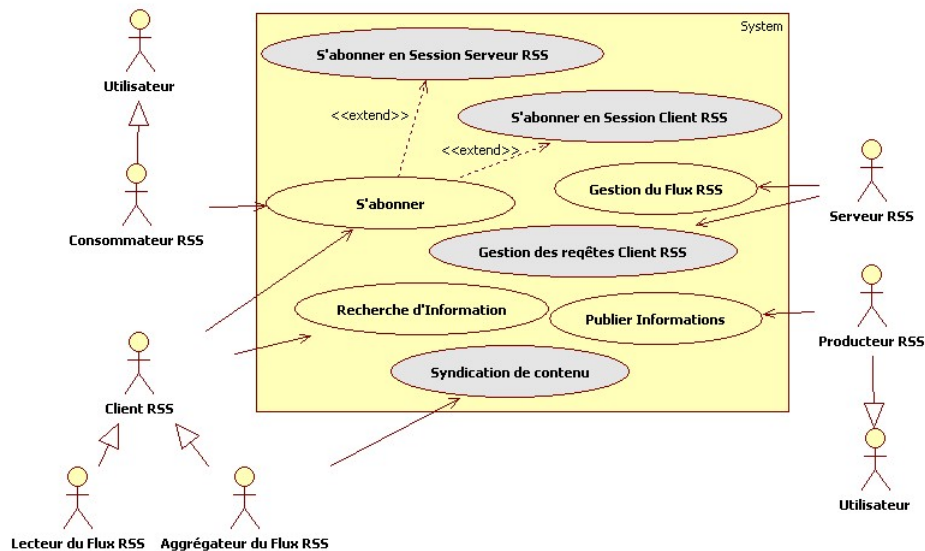


FIGURE 6.1 – Les cas d'utilisation du système

Ces processus sont mieux explicites à travers la figure 6.2.

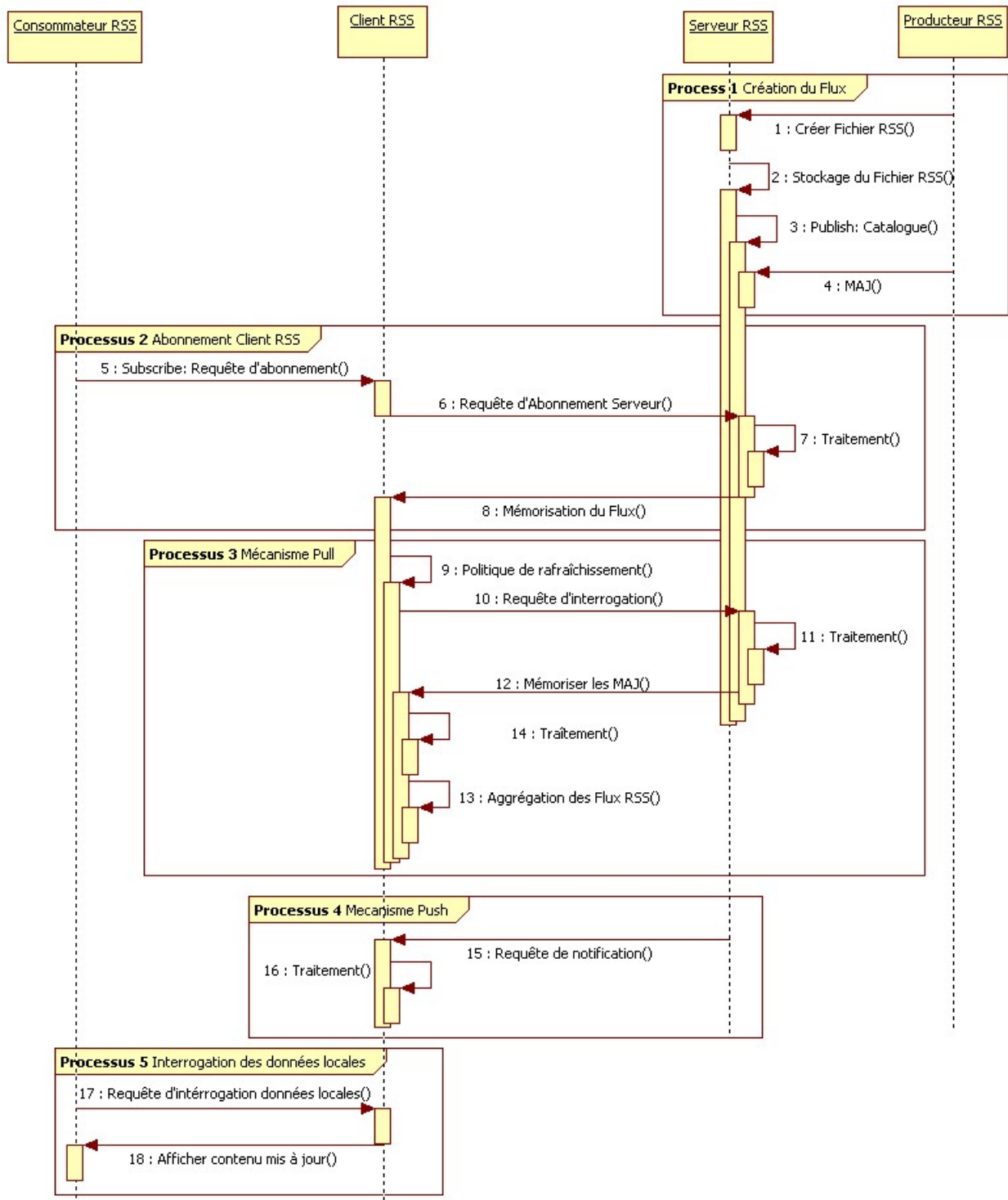


FIGURE 6.2 – Diagramme de séquence du système proposé

Nous resumons avec la modélisation dans la figure 6.3, à travers l'ingénierie des processus, le système de syndication proposé.

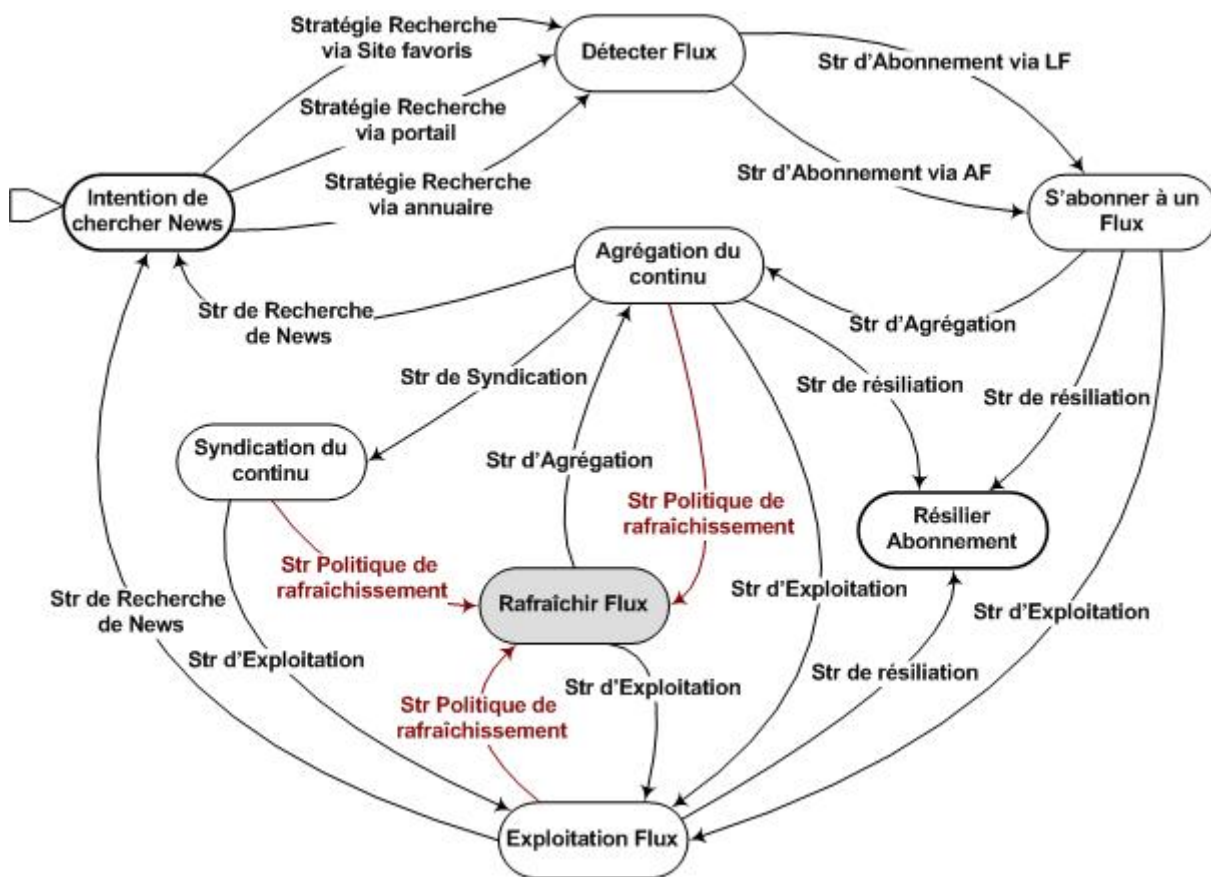


FIGURE 6.3 – Modélisation du système en MAP

Mise en oeuvre des mécanismes de synchronisation avec WSP

Comme nous avons spécifié pour le protocole WSP, il permet de dérouler les deux mécanismes Pull et Push. Nous spécifions à travers ces deux figures 6.4 et 6.5 ces deux mécanismes.

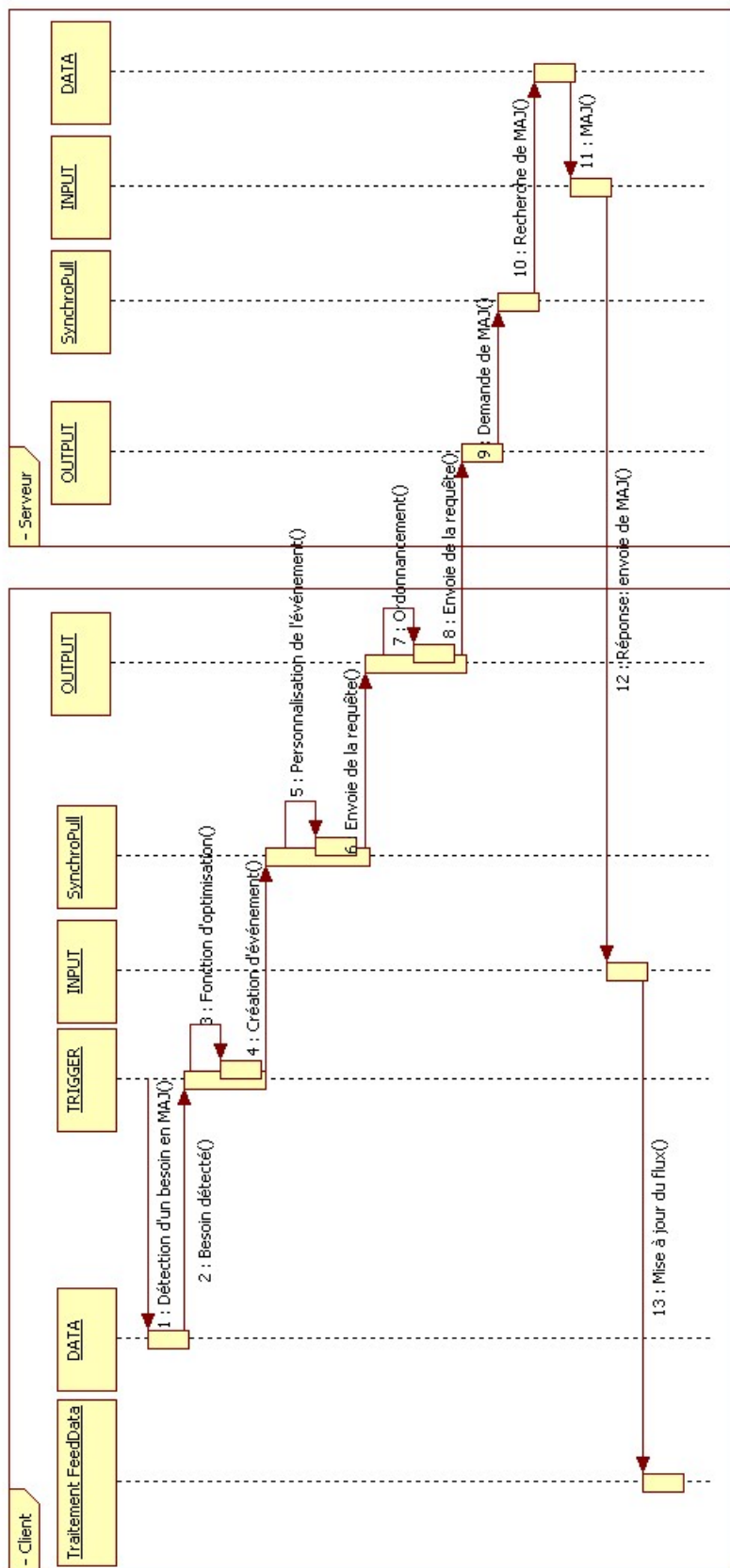


FIGURE 6.4 – Diagramme de séquence du mécanisme Pull

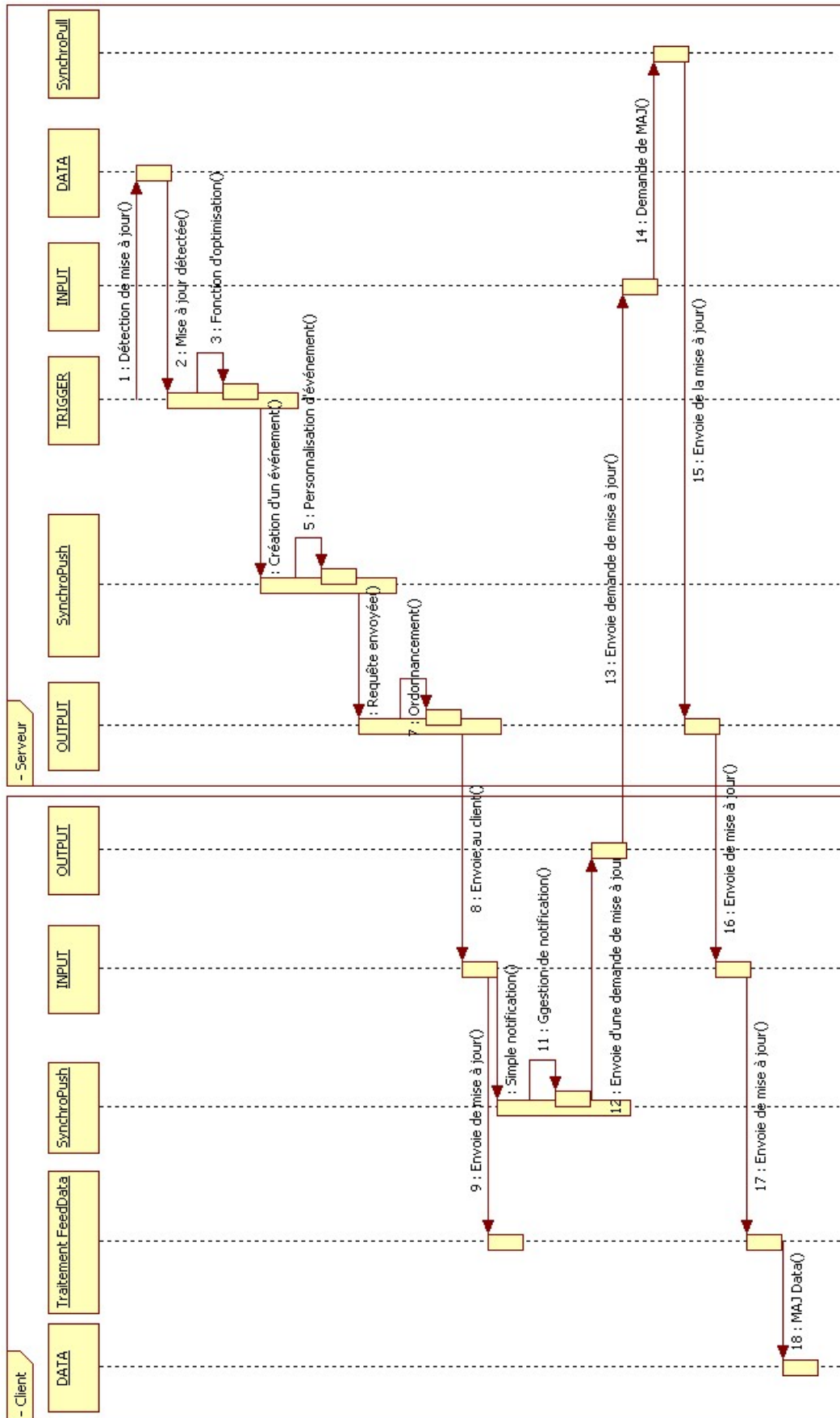


FIGURE 6.5 – Diagramme de séquence du mécanisme Push

Conception de la partie *Persistence* dans le système

La partie persistance dans l'architecture de référence proposée peut être conçue à travers le modèle conceptuel de donnée. La figure 6.6 illustre la conception des données du serveur.

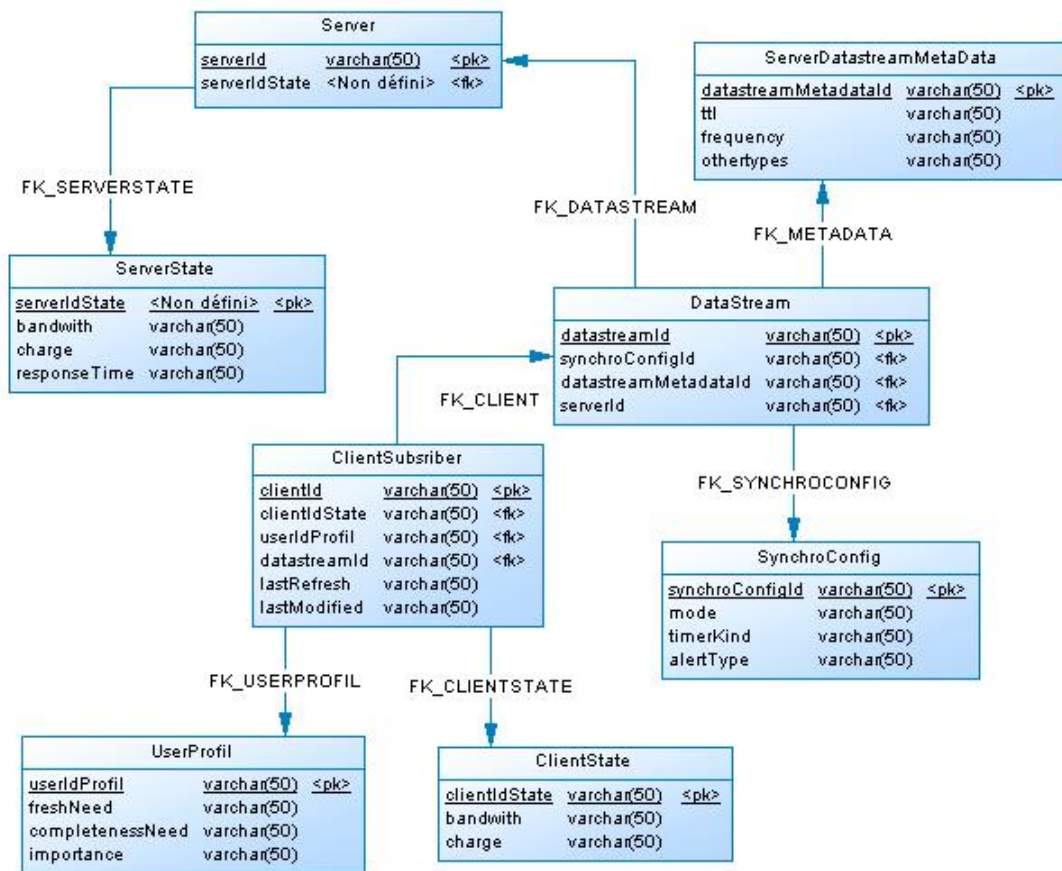


FIGURE 6.6 – Modèle conceptuel de la base de données serveur

La figure 6.7 illustre la conception des données du client.

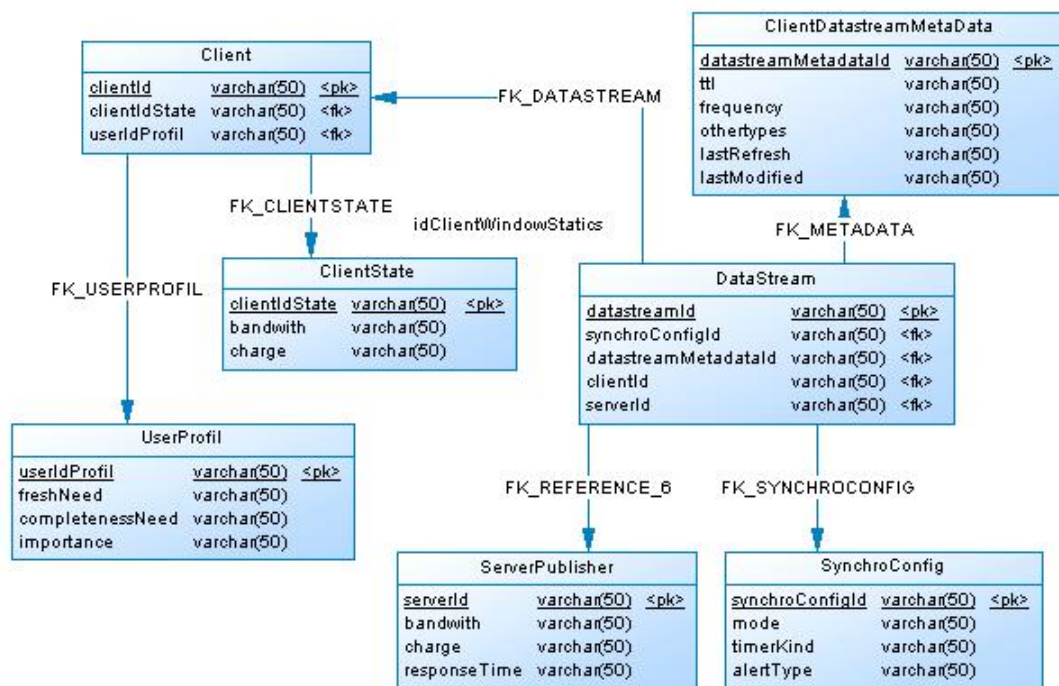


FIGURE 6.7 – Modèle conceptuel de la base de données client

Implémentation et hiérarchie des packages (services)

Les figures 6.8, 6.9 et 6.10 illustrent respectivement l'hiérarchie de packages pour le serveur, le producteur et le client. Nous concevons un package *commun* pour la gestion des fonctionnalités communes pour ces différents acteurs. Nous adoptons aussi une hiérarchie organisée dans l'implémentation pour permettre la simplicité de maintenir le code.

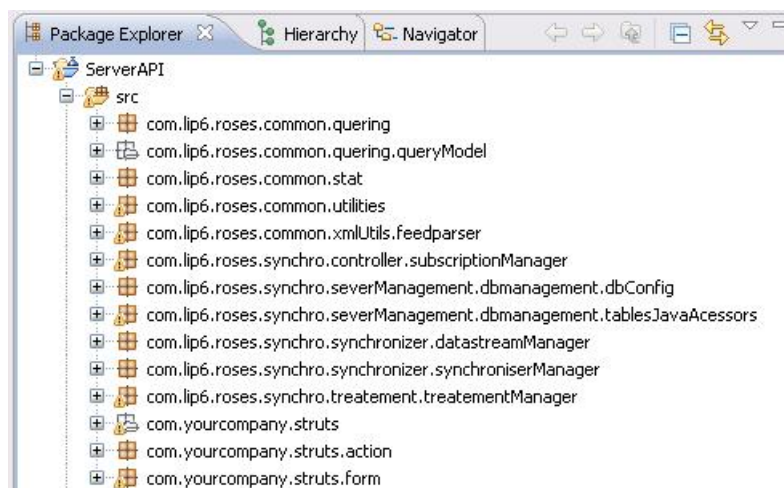


FIGURE 6.8 – L'API serveur

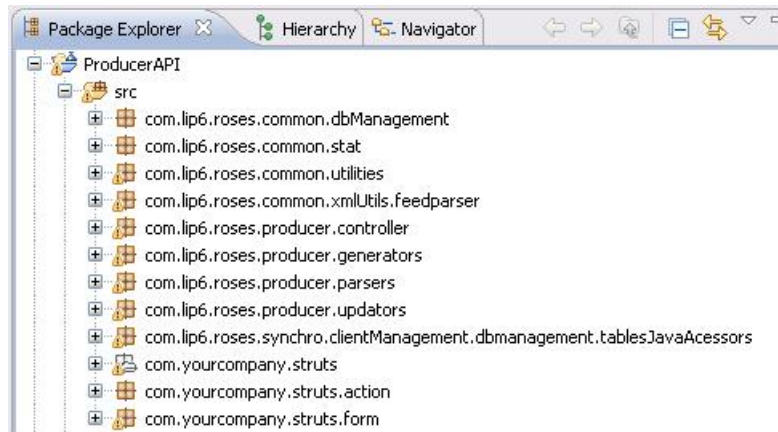


FIGURE 6.9 – L'API producteur

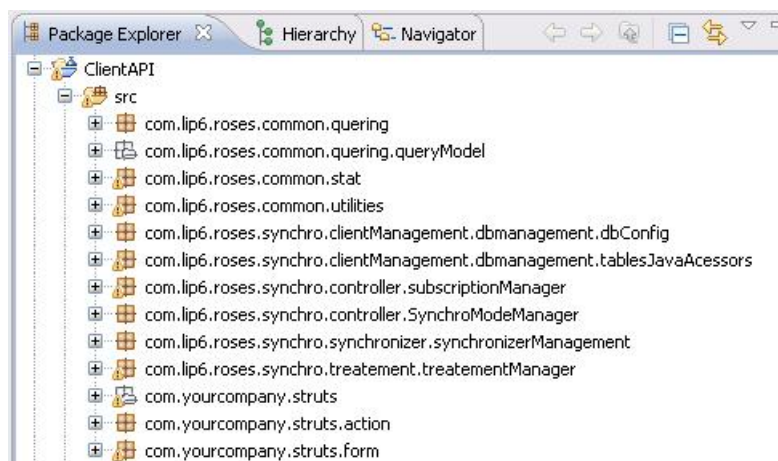


FIGURE 6.10 – L'API client

