Découverte et analyse de dépendances dans les réseaux IP

Samer Mehri

Université Henri Poincaré – Nancy I

Découverte et analyse de dépendances dans les réseaux IP

MÉMOIRE

24 juin 2008

pour l'obtention du

Master Recherche de l'Université Henri Poincaré – Nancy I

(Spécialité: Informatique)

par

Samer Merhi

Composition du jury

Dominique MERY
Claude GODART
Didier GALMICHE
Noëlle CARBONELL
Guy PERRIER

Encadrant : Olivier FESTOR

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Je tiens à remercier tout particulièrement Monsieur Olivier Festor, responsable scientifique de l'équipe MADYNES, qui m'a guidé et suivi avec beaucoup de professionalisme tout au long du projet.

Je remercie chaleureusement l'ensemble de l'équipe MADYNES, pour l'ambiance conviviale qui y règne.

Je tiens à exprimer ma sincère gratitude envers Dominique Mery, Claude Godart, Didier Galmiche, Noëlle Carbonell et Guy Perrier pour avoir bien voulu faire partie de mon jury.

Table des matières

Table des figures

Introduction

La délivrance de tout service de communication s'appuie sur le fonctionnement harmonieux de multiples entités physiques et logiques qui contribuent à sa réalisation. Par conséquent, l'échec d'une de ces entités aboutit à des fonctionnements menant parfois à l'interruption totale du service. L'étude des dépendances est dans ce contexte indispensable pour analyser les causes potentielles de fautes, améliorer les performances et se protéger contre les attaques.

Les études montrent que 70 % du budget réservé à l'informatique dans une Entreprise est dépensé sur la maintenance. La capacité de créer un graphe de dépendances (GDD) d'une entreprise pourrait avoir un impact financier majeur sur ce poste en facilitant la réalisation des techniques suivantes pour la gestion et le diagnostic des pannes :

Localisation des fautes : C'est toujours une source de frustration pour les utilisateurs quand une application s'arrête temporairement pour une raison qui est non évidente. La partie la plus difficile dans la résolution des problèmes est en général la localisation du problème en premier lieu. Est-ce dans un serveur surchargé ? Une configuration de politique ? Un routeur ou une liaison en échec ? Le GDD pour une application donnée récapitule non seulement les composants qui sont impliqués dans sa réalisation, mais permet aussi aux informations de plusieurs clients d'être combinées pour définir exactement les sources des fautes.

Planification de la reconfiguration : Les entreprises ajoutent, réorganisent, ou consolident leurs services de façon continue. En général, ces changements causent des perturbations aux services en plus de celles causées par les interactions inattendues. La planification de ces changements et le diagnostic des problèmes qui en résultent inévitablement est côuteuse. On peut s'attendre à ce que le GDD aide de deux façons. D'abord, en détectant automatiquement les dépendances, les conséquences inattendues peuvent être identifiées et les failles anticipées. Deuxièmement, le GDD permet aux départements informatiques d'avertir d'avance les utilisateurs qui seront affectés par les changements.

Optimisation du help desk : Le fait que plusieurs utilisateurs sont actifs sur un service en même temps signifie que des échecs vont probablement impliquer plusieurs appels au help desk. L'introduction d'un nouvel effort de diagnostic pour chaque appel serait un gaspillage. La connaissance des dépendances entre les composants signifie que de nouveaux rapports peuvent être rapidement liés à la source directe de la faute : on élimine ainsi le temps passé à examiner des questions dépendantes. Il réduit aussi la probabilité de corrections inappropriées comme par exemple le reboot inutile de l'ordinateur de l'utilisateur.

Détection d'anomalies : Si le GDD est automatiquement construit en se basant sur le comportement observé des hôtes, les anomalies et les changements dans les graphes pointent vers les hôtes qui méritent une investigation plus approfondie. Par exemple, les différences entre des clients peuvent être utilisées pour trouver des questions de politique. Si un ensemble de clients ne peut pas atteindre un serveur tandis que tout est excellent pour un autre ensemble, les alogrithmes de détection de dépendances localiseront le problème pour les clients. La structure d'un GDD peut alors aider à guider l'homme pour déterminer si la cause est un pare-feu commun parmi les clients ou une politique sur les clients eux-mêmes.

Plusieurs techniques d'élaboration automatique du graphe de dépendances existent dans la littéra-

ture. Toutes requièrent soit une instrumentation des applications ou la disponibilité d'une inspection profonde des paquets. Ceci est difficilement réalisable dans de nombreux réseaux pour des raisons de débit souvent mais également pour des raisons légales de non accès au contenu applicatif des paquets. Notre objectif dans ce contexte est d'étudier l'applicabilité et l'adaptation des méthodes existantes de construction du graphe de dépendances en exploitant une donnée facilement disponible sur des réseaux d'entreprise à savoir les enregistrements de flux.

Nous proposons une technique de découverte des dépendances basée sur l'analyse des enregistrements des flux Netflow. Les techniques proposées par Microsoft dans [1] et [2] identifient les services et définissent la dépendance d'une manière différente de celle des chercheurs d'IBM[3]. Nous avons implémenté ces deux techniques, mais au lieu d'analyser les paquets IP (entêtes et données), nous proposons d'analyser les enregistrements NetFlow qui contiennent toutes les informations nécessaires pour l'analyse (les numéros des ports pour identifier des services et le FlowStartTime et le FlowEndTime pour déterminer le temps d'occurrence des flux).

Le mémoire est organisé comme suit. Le chapitre 1 introduit les différentes approches déjà proposées pour la détection des dépendances. Le chapitre 2 présente notre contribution et explique les besoins fonctionnels de l'implantation. Le chapitre 3 comprend l'évaluation de l'approche et le dernier chapitre synthétise les résultats obtenus et identifie quelques poste de poursuite de ces travaux.

Chapitre 1

Etat de l'art

Plusieurs techniques sont proposées dans la littérature pour automatiser la découverte des dépendances. L'objet de ce chapitre est de présenter et de comparer plusieurs techniques présentant chacune un algorithme et une architecture spécifiques, des données d'entrée différentes, et générant des graphes de dépendance avec des niveaux variés. La figure 1.1 illustre un shéma fonctionnel d'une application Web. Malgré la simplicité de l'exemple, il présente des aspects très intéressants et existants dans la plupart des applications Web. Cet exemple servira d'illustration tout au long de ce rapport. Les applications Web sont executées sur les hôtes (H1 et H2). Le rôle du proxy DNS est d'équilibrer la charge entre les deux serveurs de noms DNS1 et DNS2. Le serveur WEB dépend du proxy DNS et du serveur de base de données SQL.

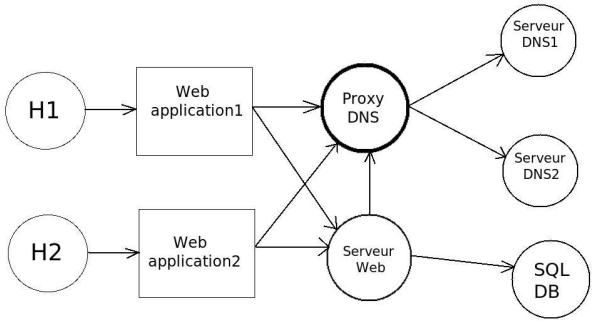


FIG. 1.1 – Schéma fonctionnel

1.1 Approches existantes

1.1.1 Sherlock

Le système Sherlock [1] construit le graphe de dépendances pour les services de communication par découverte et apprentissage automatiques basés sur l'analyse du trafic réseau. Il est composé d'un moteur d'inférence centralisé et d'un ensemble d'agents Sherlock distribués dans le réseau.

Le fonctionnement du système Sherlock est basé sur trois étapes principales. (1) Identification des dépendances des services, (2) construction du graphe des dépendances et (3) localisation des fautes.Nous détaillons ces trois étapes dans les sections suivantes.

Identification des dépendances des services

Le rôle des agents Sherlock est de capturer les paquets IP afin de les analyser pour trouver des corrélations utiles pour la détection des dépendances. Les agents sont installés sur les hôtes pour détecter les services exécutés par les clients et observer leurs temps de réponses. Ils peuvent aussi capturer des paquets traversant un lien.

Découverte des services : Un service est identifié par les numéros de ports et les adresses IP se trouvant dans les entêtes IP. Sherlock n'utilise aucune méthode d'inspection profonde de paquets afin d'identifier les services utilisant les ports non standards et les applications qui tentent de cacher leur trafic en ne communiquant que sur les ports connus.

Découverte des dépendances : Un service B est considéré comme dépendant de A si au moins un paquet de A apparaît avant B avec une probabilité assez intéressante pendant un intervalle de temps dit "Intervalle de dépendances". Le choix de cet intervalle est critique; un intervalle très large peut causer des faux positifs parce que deux services non dépendants auront plus de chance d'être éxecutés consécutivement quand l'intervalle de dépendances augmente. Par contre, un intervalle très court rend la découverte des dépendances assez difficile. A partir de ses paquets, chaque agent construit la matrice des dépendances des services et envoie ces résultats au moteur d'inférence.

Agrégation des probabilités : Lors de la réception des matrices des agents, le moteur d'inférence agrège les probabilités pour obtenir des résultats plus précis. L'aggrégation est très importante car elle contribue à éliminer les faux positifs.. Une dépendance non existante, fournie par un agent, va être négligée à cause de son absence dans les résultats des autres agents.

Construction du graphe :

Le graphe d'inférence ou le graphe de dépendances construit par Sherlock est un graphe orienté pondéré dont les arcs représentent les dépendances et les nœuds représentent les services et les liens physiques. Les nœuds sont de trois types :

1. Les nœuds 'Root-Cause' : Leurs échecs causent l'interruption de la tache du client.(ex : Liens physiques, serveurs des fichiers...)
2. Les nœuds d'Observation : représentent les services dont les performances sont mesurables par les agents Sherlock.
3. Les Meta nœuds : sont situés entre les nœuds d'observation et les nœuds 'Root-Cause'.

Tout service S appartenant à l'ensemble des services détectés par chaque agent est représenté par un méta nœud. Le moteur d'inférence ajoute au graphe un nœud d'observation pour chaque client accédant à un service et mesurant son temps de réponse. Ensuite, le moteur d'inférence commence à découvrir les dépendances entre les services en analysant les matrices des probabilités. Finalement, les nœuds 'Root-Cause' sont ajoutés au graphe pour représenter les liens physiques, les routeurs et les hôtes sur lesquels les services sont exécutés. Les liens physiques et les routeurs sont découverts à partir des traces des routes fournis par les agents Sherlock. Dans l'exemple de la figure 1, supposons qu'un hôte (H1) exécute l'application Web. Cette opération est représentée par un nœud d'observation car le temps de réponse de l'application peut être mesuré par les agents Sherlock. L'échec d'un serveur DNS ou du serveur Web cause l'interruption de l'application web. Pour cela, ils sont représentés par des nœuds 'root-cause'. Il en est de même pour les liens et les routeurs intermédiaires appartenant au chemin liant l'application au serveurs Web et DNS (e.g : Path H1−>Server Web).

L'état de chaque nœud est défini par le triplet $< P_{Up}, P_{Troubled}, P_{Down} >$ avec P_{Up} la probabilité que le nœud fonctionne normalement, $P_{Troubled}$ la probabilité que le nœud continue à fonctionner mais avec une dégradation des performances (ex : temps de réponse très long) et P_{Down} la probabilité que le service est interrompu. Ces probabilités sont calculées à partir de l'état des nœuds ancêtres, et suivant les degrés de dépendance.

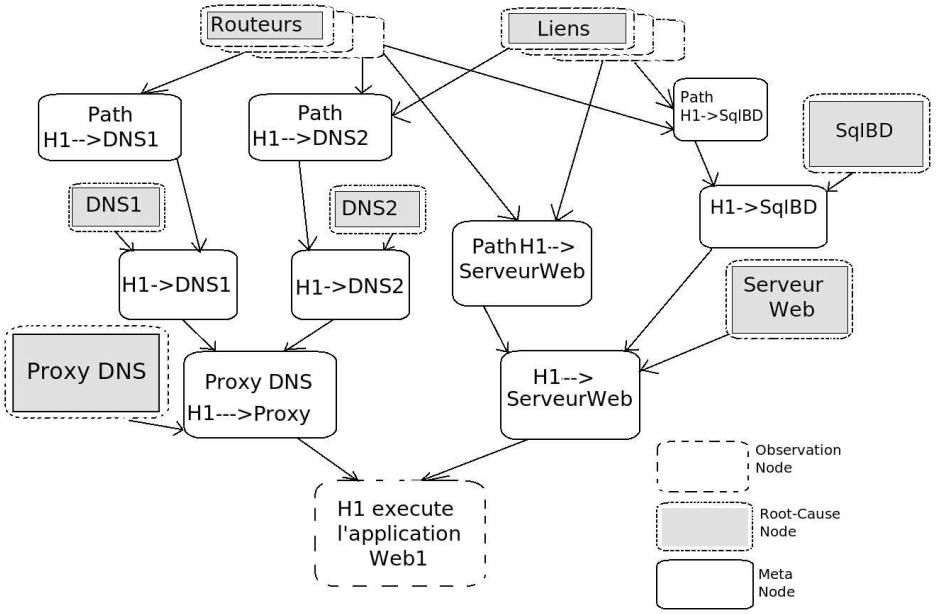


FIG. 1.2 – Graphe d'inférence du système Sherlock appliqué à l'exemple

Prenons l'exemple du proxy DNS qui est considéré comme un méta nœud entre les nœuds H1->DNS1 et H1->DNS2. Ces deux nœuds représentent les dépendances du proxy qui choisit un serveur DNS parmi les deux lors de chaque exécution d'une tache de l'application Web1. C'est pour cela que le méta nœud proxy est aussi appelé *Selector Node*. Supposons que la probabilité de dépendance du proxy DNS envers le serveur DNS1 vaut (d) , alors celle du serveur DNS2 sera (1-d).

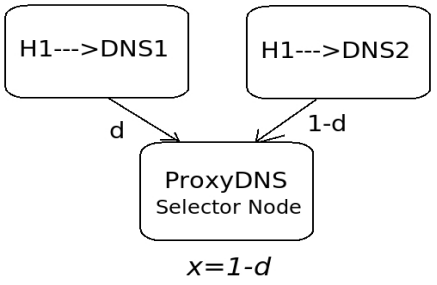


FIG. 1.3 – Tableau de probabilité du nœud Proxy

DNS1 / DNS2	UP	Troubled	Down
UP	1, 0, 0	x, 1-x, 0	x, 0, 1-x
Troubled	1-x, x, 0	0, 1, 0	0, x, 1-x
Down	1-x, 0, x	0, 1-x, x	0, 0, 1

La figure(1.3) présente comment l'état du nœud proxy ($< P_{Up}, P_{Troubled}, P_{Down} >$) est calculé à partir de l'état des nœuds ancêtres H1->DNS1 et H1->DNS2. Ces résultats peuvent être élaborés intuitivement dans le cas de deux antécédents (H1->DNS1 et H1->DNS2). Par contre, le calcul des probabilités sera plus compliqué dans le cas de n antécédents. Dans ce cas là, ces probabilités seront calculées à partir des formules suivantes :

$$P(up) = \prod_j \Big(\big(1 - d_j\big) * \big(p_j^{trouble} + p_j^{down}\big) + p_j^{up} \Big) \tag{1.1}$$

$$P(down) = \prod_j \Big(1 - p_j^{down} + \big(1 - d_j\big) * p_j^{down} \Big) \tag{1.2}$$

$$P(Troubled) = 1 - \Big(P\big(up\big) + P\big(down\big) \Big) \tag{1.3}$$

avec d_j la probabilité de dépendance de l'antécédent j.

Localisation des Fautes :

On définit un *'assignment vector'* comme étant un vecteur décrivant l'état de tous les nœuds 'Root-Cause' par Up, Troubled ou Down. Par exemple, l'état normal du système sera décrit par un vecteur dont tous les éléments sont des triplets (Up=1,Troubled= 0,Down= 0). A chaque fois que le temps de réponse d'un service sort de l'intervalle de tolérance, l'algorithme FERRET sera lancé pour détecter la source de la faute. FERRET consiste à chercher l'*assignment vector* le plus convenable pour expliquer l'observation. Cet a*ssignment vector* est trouvé à l'aide d'une fonction de pointage (scoring function) qui prend comme paramètres l'observation O, le graphe d'inférence G et retourne la liste des *assignment vectors* les plus convenables. Pour simplifier le calcul et le temps nécessaire pour la localisation des fautes on part du principe que les 'Root-Cause' sont souvent en état Up, alors dans le cas où le graphe d'inférence contient r 'Root-Cause' au lieu de chercher parmi les 3^r *assignment vectors* possibles, il est faisable de comparer les vecteurs ayant au plus k 'Root-Cause' en état Down ou Troubled. Par exemple, pour k=1, Ferret retourne les meilleurs *'assignment vectors'* parmi 2*r vecteurs.

Dans notre exemple, supposons que le serveur SQL tombe en panne. Celui ci va déclencher une exception à chaque fois que H1 essaye d'exécuter une instruction de manipulation de données (e.g : Select, update, etc ...). A cet instant, l'algorithme FERRET sera lancé pour détecter la source de la faute. D'après le graphe d'inférence, on trouve six nœud de type "root-cause" (ServeurDNS1, ServeruDNS2, Serveur Web, SqlBD, les routeurs et les liens). Le premier 'assignment vector' à évaluer sera le serveur DNS1 en état down et tous les autres en état UP. La fonction de pointage va évaluer ce vecteur. L'état de chaque nœud fils sera déduit à partir de ses ancêtres d'après les formules (1),(2) et (3) jusqu'à atteindre le nœud d'observation. Chaque vecteur aura un score décrivant le degré de similarité entre les probabilités calculées et les probabilités mesurées par les agents Sherlock. Le processus continue avec le deuxième 'assignment vector' ayant serveurDns2 en état down et tous les autres root-cause en état UP, et ainsi de suite. Dans notre cas, le vecteur qui va avoir le score le plus élevé est le vecteur décrivant SqlDB comme down et tous les autres en état UP. Par contre, le vecteur ayant le serveur Web en état down va avoir un score très bas puisqu'une telle assomption va causer l'interruption totale de l'application Web1, ce qui contredit les observations des agents Sherlock.

1.1.2 Constellation

Constellation [2] est une méthode qui construit le graphe de dépendances des services d'une manière complètement distribuée. Afin d'expliquer son fonctionnement, il faut tout d'abord définir les trois éléments de base suivants : "Un Canal - Un Patron d'activité - Un modèle d'activité".

* Canal : un canal est constitué de deux entités entre lesquelles existe un flux de messages. Suivant le sens du flux des messages, un canal sera considéré comme étant canal de sortie ou d'entrée.
* Patron d'activité : C'est une variable décrivant l'état d'un canal comme actif ou inactif. Un canal est inactif si aucun paquet ne le traverse dans un intervalle de temps bien précis.
* Modèle d'activité : C'est une fonction prenant un canal d'entrée et retournant un vecteur de probabilités attribué aux canaux de sortie. Les éléments de ce vecteur sont les probabilités que les canaux de sortie soient actifs sachant que le canal d'entrée est actif.

Construction du graphe :

Chaque hôte analyse son flux de paquets IP et construit son modèle d'activité par apprentissage (e.g : modèle bayésien, Noisy-Or...). Un hôte voulant construire le graphe de dépendances d'un service, envoie une requête à son modèle d'activité. Cette requête prend le canal d'entrée correspondant au service comme paramètre, et retourne l'ensemble des canaux de sortie actifs correspondant à ce canal d'entrée. Les canaux de sortie de cet hôte sont vus comme des canaux d'entrée par rapport aux hôtes voisins. Cette requête sera renvoyée à ces hôtes qui lancent à leur tour l'algorithme jusqu'à la construction complète du graphe.

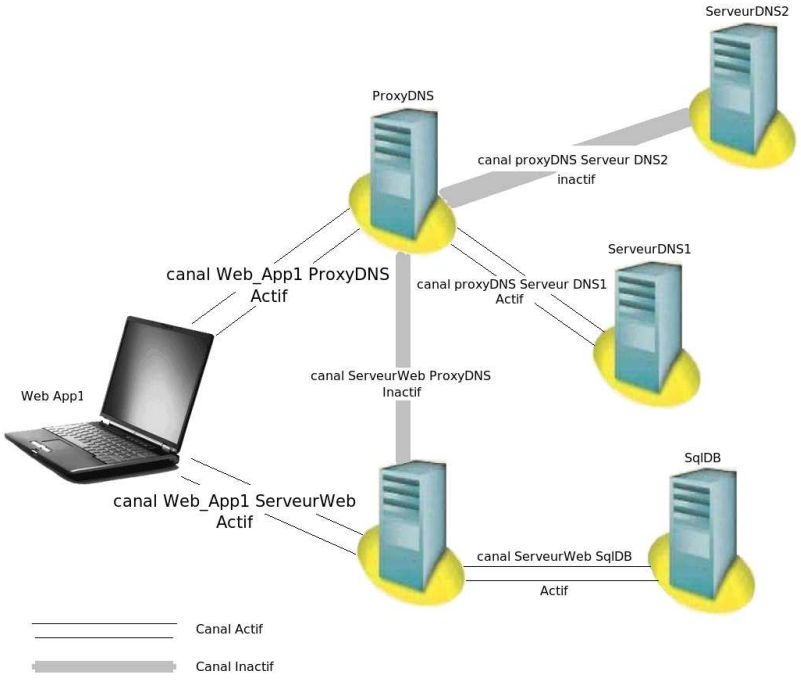


FIG. 1.4 – Les Canaux Actifs et Inactifs de l'application Web1

La figure 1.4 , présente les canaux actifs et inactifs entre les différentes entités de notre système. On remarque que le canal 'serveur Web - ProxyDNS' est marqué comme inactif (à cause du cache). De même, le canal 'ProxyDNS' - ServeurDNS2 est aussi inactif car il n'est traversé par aucun paquet pendant l'intervalle de temps prédéfini. Supposons que l'hôte H1 cherche le graphe des dépendances de l'application Web1. D'après la figure 1.4, le modèle d'activité de cette application contient deux canaux actifs (WebApp1-proxyDNS et WebApp1-ServeurWeb). Deux requête ayant comme destination

le proxyDNS et le serveur WEB1 sont envoyées afin de trouver les canaux actifs de ces deux entités lors de l'exécution de l'application WEB1. A cet instant, la dépendance serveur DNS1 - proxy ne doit pas être prise en considération à cause de l'inactivité du canal, ce qui est pareil pour la dépendance ServeurWeb-proxyDNS.

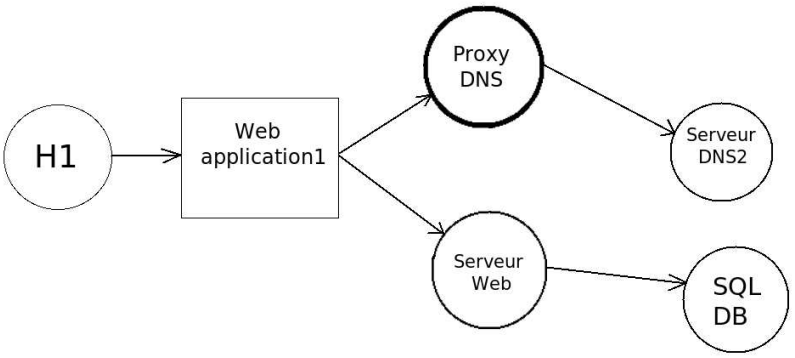


FIG. 1.5 – Le graphe de dépendances

Cependant, la technique "constellation" construit le graphe d'une manière réactive et totalement distribuée, celui-ci peut surcharger les hôtes et parfois manquer des dépendances existantes (figure 1.5).

1.1.3 IBM

Des checheurs d'IBM ont proposé une autre méthode pour la découverte des dépendances des services [3]. Cette méthode ne prend pas en considération la structure physique du réseau (routeurs et liens).

La découverte des dépendances est basée sur l'analyse de l'historique D des temps de début et de fin d'exécution des services. Cette opération se déroule en trois étapes, détection primaire puis élimination des faux positifs et enfin découverte des dépendances directes.

Construction du graphe :

Une transaction T(S,[s,t]) représente une instance de service S dont le temps d'exécution commence à l'instant s et se termine à l'instant t. Afin de trouver l'ensemble des transactions appartenant à un flux de paquets IP, IBM propose une approche d'inspection profonde puisque les temps de début et de fin d'exécution des services peuvent être déduits à partir des patrons bien définis (e.g. : l'instant de départ d'une transaction http peut être déduit par l'occurrence d'un paquet contenant 'GET' et la fin sera signalée par un autre contenant 'HTTP/1.1 200 OK').

On dit qu'une transaction $T_1(S_1[s_1, t_1])$ contient une transaction $T_2(S_2[s_2, t_2])$ si $s_1 \leq s_2 \leq t_2 \leq t_1$ (noté par $T_1 \geq T_2$). Si T_1 appelle T_2 on dit que T_1 dépend de T_2 (noté par $T_1 \rightarrow T_2$). T_1 est dite parent de T_2 si T_1 dépend de T_2, et dans ce cas on dit que T_2 est fils de T_1. On part du principe que si T_1 dépend de T_2, T_1 ne peut pas terminer son exécution avant T_2. Autrement dit, si $T_1 \geq T_2$ alors ($T_1 \rightarrow T_2$)

Prenons l'exemple de la figure 1.6 où le client H1 lance l'application Web1 qui dépend du proxyDNS avant de se connecter au serveur WEB, et l'application Web2 du client H2 se connecte directement au serveur Web (à cause du cache), et effectue une transaction à la base des données. la figure 1.7 représente l'espace temporel des exécutions des transactions. Une Transaction représente une instance d'exécution d'un service. Par exemple, la transaction T3 débute quand le proxy envoie un message de type 'Requête' et termine une fois qu'un message de type 'Réponse' est généré par le serveur DNS1.

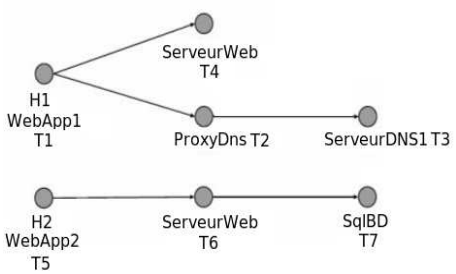


FIG. 1.6 – Graphe de dépendance

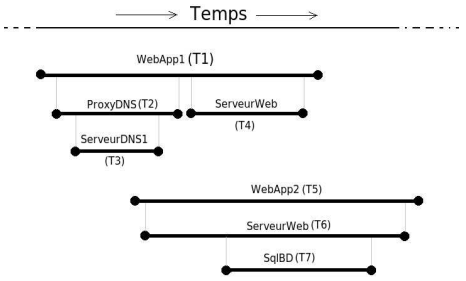


FIG. 1.7 – Execution concurrente de S1 et S2

En regardant la figure 1.7 qui représente l'espace temporel des exécutions des transactions, on voit que $T_1 \geq (T_2$ et $T_4)$, ce qui était prévu parce que T1 dépend de T2 et de T4 de même pour $T_2 \geq T_3$ et $T_5 \geq T_6 \geq T_7$. On remarque aussi que $T_5 \geq T_4$ mais ceci est produit par hasard.

Elimination des faux positifs et découverte des dépendances directes :

Pour avoir une détection primaire des dépendances il faut calculer $p((S_1) \rightarrow (S_2))$ qui est défini par $\#(S_1|S_1 \geq S_2)/\#(S_1)$ avec $\#(S_1)$ le nombre total des transactions $T(S_1)$ et $\#(S_1|S_1 \geq S_2)$ le nombre des transactions $T(S_1)$ contenant au moins une transaction $T(S_2)$. Cependant, même si S_1 ne dépend pas de S_2, S_1 peut contenir S_2 par hasard. Pour faire face à ces faux positifs on définit r comme étant $\#(S_2|S_1 \geq S_2)/\#(S_2)$ avec $\#(S_2|S_1 \geq S_2)$ le nombre des transactions $T(S_2)$ contenu par au moins une transaction $T(S_1)$.
Cette méthode est applicable dans les réseaux à faible débit. Une fois que le débit des paquets commence à augmenter, des faux positifs dans les dépendances commencent à apparaître. De même, le modèle précédent ne prend pas en considération le fait que chaque service admet au plus un parent (il faut ici choisir le parent de T_4 parmi T_3 et T_1). Pour apporter une solution à ces deux problèmes, une approche probabiliste pour trouver les faux positifs et un modèle compétitif pour inférer les dépendances directes sont proposés.

1.1.4 Technique de corrélation d'événements

Une nouvelle étude est proposée des chercheurs de Carnegie Mellon University (CMU) pour la découverte dynamique des dépendances physiques du réseau [4]. Elle consiste à analyser le fichier Syslog pour trouver des corrélations entre les différents événements enregistrés.

Le fichier Syslog contient des informations sur le lieu et sur le moment de l'occurrence de chaque événement (e.g :[2006-15-10T00 :04 :30| local6|err|int1|. pm3392.0.1. R64B66B : Loss of signal from the optics] est un message Syslog généré à 4 :30 le 15 octobre 2006 signalant la perte du signal optique de l'interface int1).

Construction du graphe :

La première étape de la construction du graphe de dépendances se fait en inférant les patrons d'événements (séquences d'événements qui apparaissent fréquemment). Ensuite, pour raffiner ces patrons, les moments d'occurrence sont pris en considération. Finalement, la création des relation se fait en analysant les lieux des événements.

Les moments d'occurrence : Pour sélectionner les patrons d'événements avec précision, le moment de l'occurrence peut être considéré comme une heuristique très intéressante. Supposons par exemple que l'occurrence de l'événement B après l'événement A se fait dans un intervalle de 5 secondes avec une probabilité de 99%. Par conséquent les patrons A-B séparés par un intervalle de 50 secondes

peuvent être éliminés. De même, cette étude temporelle peut nous aider à faire face à l'entrelacement des événements.

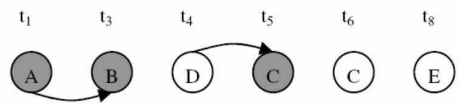


FIG. 1.8 – Corrélation incorrecte

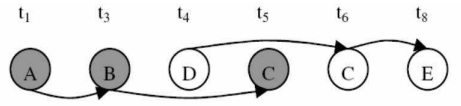


FIG. 1.9 – Corrélation correcte

L'absence d'une analyse temporelle peut aboutir à des fausses sélections des patrons à cause de l'entrelacement des événements. Prenons l'exemple de la figure 1.8 , les événements D et C sont séparés par un intervalle de (t5-t4) secondes. La première sélection ne respecte pas les moments d'occurrence et elle a signalé la présence de deux patrons A-B et D-C. La deuxième le fait (fig.1.9), et puisque l'occurrence de D-C dans un intervalle de (t5-t4) secondes se produit avec une probabilité minime, le patron D-C n'est pas pris en compte, et la vrai sélection est A-B-C et D-C-E.

Les lieux des événements : Pour créer le graphe des dépendances, on calcule la probabilité de l'occurrence de l'événement e_2 au lieu l_2 sachant que l'événement e_1 s'est produit au lieu l_1 (les lieux peuvent être des routeurs, des commutateurs, des hôtes ...). Le tableau ci-dessus, présente la probabilité

	P_1	P_2	P_3	P_4	Combined*
l_1 then l_2	1%	6%	94%	4%	94%
l_1 then l_3	0%	7%	0%	40%	0%
l_2 then l_1	95%	40%	97%	0%	96%
l_2 then l_3	5%	92%	98%	97%	95.6%
l_3 then l_1	3%	0%	20%	0%	0%
l_3 then l_2	96%	94%	95%	99%	96%

Pi = Patron d'evenement i
li = location i (ou routeur i)

FIG. 1.10 – Tableau des probabilités combinées

qu'un patron P_i apparaisse sur l_x puis sur l_y. Cela veut dire que si le patron P_i est constitué de deux événements A et B, la probabilité $P_{i->xy}$ sera le nombre de fois que B se produit sur l_y sachant que A s'est produit 100 fois sur l_x. L'étude se fait sur un simple réseau constitué de trois routeurs (l_1, l_2 et l_3). Le champ 'Combined' contient le moyen des pourcentages supérieurs à 90%.

En analysant ces résultats, on trouve des patrons qui apparaissent avec des probabilités élevées sur l_1 puis l_2 et sur l_2 puis l_3 (Fig. 2.2).

FIG. 1.11 – Graphe de dépendances

1.1.5 Technique XML/XPath

Les auteurs de [5] proposent une architecture distribuée pour construire le graphe de dépendances des applications et des services. Cette technique est basée sur le principe que chaque ressource (application ou service e.g : application Web, DNS, NFS etc.) est capable de générer une description XML de ses dépendances.

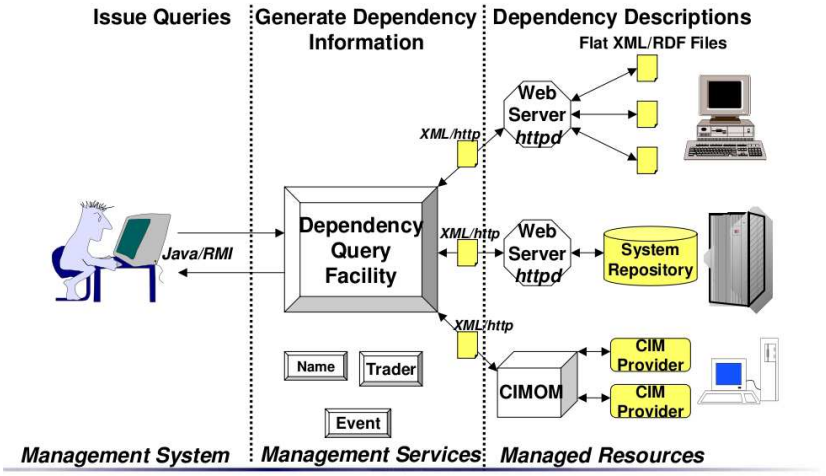


FIG. 1.12 – Architecture du système de dépendances[5]

L'architecture du système est composée de trois niveaux (figure 1.12). Au niveau 1, "Dependency description", chaque ressources génère un fichier XML décrivant ses dépendances. Au niveau 2, "Generation of dependency information", ces fichiers sont combinés afin d'avoir un document XML contenant des informations sur les dépendances entre les différentes ressources. Ensuite, au niveau 3 ,"Management System", des requêtes XMLpath seront envoyées au document XML afin de filtrer les résultats et trouver les dépendances d'un service donné.

Revenons à l'exemple de la figure 1.1. L'application Web du premier hôte va générer un fichier XML (Fig 9.2), ayant comme racine 'WebApplication1', et deux nœuds fils décrivant les dépendances 'service DNS' et 'Serveur Web'. D'après la figure 9.2 on remarque que le fichier contient le tag RDF. RDF (Ressource definition framework [8]) permet de rendre plus "intelligente" l'information nécessaire aux moteurs de recherche. Le service DNS génère un fichier décrivant ses dépendances des serveurs DNS1 et DNS2. Pareil pour le serveur Web, ses dépendances envers le service DNS et le serveur SQL seront mentionnées.

Fig 9.2 - Fichier XML decrivant les dépendances de l'application Web1

```
<?xml version ='1.0' encoding='UTF-8'?>
<rdf:RDF xmlns:ds='OurLab/DependecySchema#'
   xmlns:rdf='www.w3.org/1999/02/22-rdf-syntax-ns#'
   xmlns:rdfs='www.w3.org/2000/01/rdf-schema#'>

    <ds:Webapplication1>
    <ds:name>Webapplication1</ds:name>
    <ds:description>L'application Web de l'Hote H1</ds:description>
    <ds:version>1</ds:version>
    <ds:release>1</ds:relearse>

         <ds:dependency>
           <ds:ServiceDependency>
              <ds:antecedent rdf:resource='http://Lab2/xmlFiles/DNSproxy.xml'/>
              <ds:generated>automatic</ds:generated>
              <ds:label>WebApp_DNSproxy_Dep</ds:label>
           </ds:ServiceDependency>
         </ds:dependency>

         <ds:dependency>
           <ds:ServiceDependency>
              <ds:antecedent rdf:resource='http://Lab2/xmlFiles/ServeurWeb.xml'/>
              <ds:generated>automatic</ds:generated>
              <ds:label>WebApp_ServeurWeb_Dep</ds:label>
           </ds:ServiceDependency>
         </ds:dependency>

    </ds:Webapplication1>
</rdf:RDF>
```

Supposons que nous cherchons les dépendances de l'application Web1. Une simple requête XmlPath ((/descendant : :ds :WebApplication1/dependency)) retourne l'ensemble des dépendances de l'application Web1. Le résultat de cette requête va contenir deux nœuds, le proxy DNS et le serveur Web. Par la suite, des nouvelles requêtes XmlPath seront générées, ((/descendant : :ds :DNSProxy/dependency)) et ((/descendant : :ds :WebServer/dependency)) afin de révéler leurs dépendances.

1.1.6 ADD - Active Dependency Discovery

Toutes les techniques citées dans les sections précédentes sont passives. Les informations d'entrée sont les paquets IP et des pré requis sur la topologie du réseau. [6] et [7] présentent une approche différente, ADD (Active Dependency Discovery), basée sur le principe que la topologie du réseau et les dépendances des services peuvent être révélées d'une manière indirecte. L'injection d'une perturbation à une entité cause l'interruption d'une ou de plusieurs applications, ceci peut être considéré comme un témoin sur l'existence d'une dépendance entre l'entité perturbée et l'application affectée par la perturbation.

Une perturbation peut être injectée au niveau applicatif (e.g : mettre un verrou d'accès sur un tableau d'une base de données) ainsi qu'au niveau physique (e.g : injection d'agressions magnétiques au niveau des broches des puces électroniques afin d'introduire des erreurs au niveau binaire).

Le degré de dépendance d'un service S1 à un autre S2 est donné par :

$$\frac{(\#S1perturbe/S2pertube)}{\#S2pertube} \tag{1.4}$$

avec #S1perturbe/S2pertube représente le nombre de fois où S1 était perturbé durant la période de la perturbation de S2.

La technique des perturbations aux niveaux physiques (niveau des broches des puces) nécessite un dictionnaire de fautes [9]. Le rôle de ce dictionnaire est de présenter la correspondance entre les injections au niveau physique et leurs effets aux niveaux supérieurs (niveau réseau ou applicatif). Par exemple, perturber les broches d'entrée d'un additionneur aboutit à des résultats erronés, par suite des données violant parfois les contraintes d'intégrité d'une table de la base de données.

La procédure ADD est composée de 6 étapes :

1. Identification des nœuds :
   Cette étape spécifie l'ensemble des nœuds du graphe de dépendances. Les éléments de cet ensemble dépendent de la granularité du système. Un système désirant un graphe à faible granularité, va avoir un ensemble contenant les applications en cours d'exécution. En augmentant le taux de granularité du système, des nouveaux éléments peuvent être ajoutés à cet ensemble (e.g : les services desquels ces application dépendent, les routeurs intermédiaires liant les différents services, les hôtes sur lesquels les services sont exécutés...).

2. Surveillance du système :
   Prenons l'exemple de la figure 1.1. Chaque transaction exécutée par l'application Web d'un hôte sur la base de données Sql, nécessite un temps d'exécution. A chaque fois que le temps d'exécution sort de l'intervalle de tolérance, le système de surveillance lance une procédure de détection active pour trouver la cause du problème.

3. Découverte des dépendances :
   Supposons qu'une transaction T_1 subisse un retard de deux secondes. Avant de chercher la cause de ce retard, il faut tout d'abord trouver l'ensemble des ressources logiques (tableaux de la base de données, application Web...) et des ressources physiques (Serveur DNS, routeurs,...) dont T_1 dépend. Des perturbations seront injectées dans l'ensemble des ressources du système afin de révéler les degrés de dépendances.

Ressource/Transaction	T_1
tableau des clients	A
tableau des produits	S
DNS-proxy	S
Serveur DNS1	M
Serveur DNS2	M
Application Web	S
Serveur Web	S

Tab1 : S : Strong - M : Medium - W : Weak - A : Abscent

   D'après le tableau ci dessus, on trouve que la transaction T_1 dépend du tableau des produits appartenant à la base des données, du proxy DNS et de l'application Web. Puisque le rôle du proxy est d'équilibrer la charge sur les deux serveurs DNS, le degré de dépendance est marqué comme "Medium".

4. L'ensemble des ressources suspectes (ERS) : L'ensemble des ressources dont T_1 dépend sont considérées comme des ressources suspectes. Au moins, un élément de cet ensemble est le "Root Cause" du problème. Partons de notre exemple ERS = tableau des produits, DNS-proxy, Serveur DNS1, Serveur DNS2, Application Web, Serveur Web.

5. Ensemble des transactions suspectes (ETS) :
   Afin de raffiner l'ERS, ADD exécute d'autres transactions , par exemple T2, T3 et T4. Le tableau ci-dessous représente les résultats des quatres transactions :

Ressource/Transaction	T_1	T_2	T_3	T_4
tableau des clients	A	S	A	A
tableau des produits	S	A	S	S
DNS-proxy	S	W	S	S
Serveur DNS1	M	W	M	A
Serveur DNS2	M	A	A	M
Application Web	S	S	S	S
Serveur Web	S	S	S	S

Tab2

La différence entre les dépendances des transactions peut être évaluée. En affectant 5 unités à la différence (Strong-Abscent), cela veut dire que la différence (Strong-Weak) peut être mesurée par un nombre d'unités inférieur à 5.

L' ETS contient les transactions "proche" de T1. Une transaction est dite "proche" d'une autre si la différence entre elles est inférieure à un seuil prédéfini.

Revenons au tableau (Tab2), On remarque que T1,T3 et T4 sont assez proches, puisque les trois transactions dépendent du tableau Produits, Proxy-DNS, serveur Web et de l'application Web. Par contre, T1 et T2 sont moins proche, et par conséquent ETS va contenir T1 ,T3 et T4.

6. Détection du 'Root Cause' : Une fois que les transactions de l'ETS sont révélées, on les exécute individuellement l'une après l'autre. Supposons que T3 s'exécute normalement et T4 subit un retard identique à celui de T1. T3 sera retirée de l'ETS. L'importance de ces tests individuels est qu'ils nous aident à raffiner l'ERS parce que l'exécution normale de T3 montre que les ressources tableau des produits, DNS-proxy, serveur DNS1, Application Web, Serveur Web fonctionnent normalement et peuvent être retirées de l'ERS. Par la suite, ERS sera égale à ERS - [tableau des produits, proxy DNS, serveur DNS1, Application Web, Serveur Web]=[ServeurDNS2]. L'ensemble des ressources suspectes contient à cet instant un seul élément qui est le serveurDNS2 et c'est définitivement le "Root cause" du retard.

1.2 Analyse et comparaison

Les méthodes que nous avons étudiées pour la découverte des dépendances présentent différents algorithmes et architectures, nécessitent différentes données en entrée, et fournissent des graphes de dépendance avec des niveaux de granularité différents.

La méthode Sherlock présente deux architectures, l'une centralisée et l'autre distribuée. Malgré la distribution des agents Sherlock dans le réseau, la méthode ne peut pas être considérée comme totalement distribuée puisque la construction du graphe se réalise dans la machine d'inférence qui assemble les différentes observations collectées par les agents. L'aspect centralisé se manifeste dans les techniques proposées par [3] et par [4]. Par contre, La technique 'Constellation' est complètement distribuée comme celle du XML/XPath.

Les entêtes des paquets IP sont les données en entrée des techniques Sherlock et Constellation. Les numéros de ports indiquent les services, et les adresses IP sont indispensables pour l'identification des flux et des hôtes.

IBM propose une inspection profonde des paquets afin de déterminer le début et la terminaison des exécutions. L'avantage de l'inspection profonde se manifeste par rapport à l'analyse des entêtes dans la découverte des services communiquant sur des ports non standards (ex : les applications P2P).

L'algorithme de la technique des corrélations des événements prend en entrée les enregistrements des événements dans le fichier Syslog, ce qui est le plus simple parmi les cas précédents.

Toutes les techniques déjà citées sont complètement passives. Par contre, ADD est une approche active qui consiste à injecter des perturbations sur les ressources afin de révéler les dépendances. L'inconvénient de cette approche est qu'elle est intrusive.

Les dépendances sont exprimées par ces techniques avec des niveaux de granularité différents. Le tableau ci-dessous résume les caractéristiques de chaque méthode :

Table de Comparaison

Technique	Graphe de dépendance	Données en entrée	Architecture	Actif/Passif
Sherlock	(couche physique + services)	Entêtes des paquets IP	centralisée + distribué	Passif
Constellation	(services)	Entêtes des paquets IP	centralisée	Passif
IBM	(services)	Paquets IP Complets	distribuée	Passif
Corrélation des événements	(couche physique)	Syslog File	centralisé	Passif
XML/XPath	services	Fichiers Xml	distribuée	Passif
ADD	Couche Physique + Services	Résultats des perturbations	distribuée	Actif

Chapitre 2

Une approche basée sur l'analyse de flux

2.1 Contribution

Nous proposons une technique de découverte des dépendances basée sur l'analyse des enregistrements des flux Netflow et c'est le point fort de l'approche parceque les outils de mesure de flux sont nombreux et disponibles sur pratiquement tous les réseaux. Un flux Netflow est défini comme étant une séquence unidirectionnelle de paquets. Cela veux dire que chaque session aura deux flux, un du serveur vers le client et un du client vers le serveur. Chaque flux est identifié par 7 champs clé :

FlowStartTime
FlowEndTime
SourceIP
DestinationIP
SourcePort
DestinationPort
Protocol

Lorsqu'un paquet est reçu, le routeur examine les derniers 5 champs (SourceIP, DestinationIP, SourcePort, DestinationPort, Protocol). Si le paquet fait partie d'un flux existant, le champ FlowEndTime sera mis à jour, sinon, une nouvelle entrée de flux sera créée.

Les techniques proposées dans [1] et [2] identifient les services et définissent la dépendance d'une manière différente de celle des chercheurs de IBM[3]. D'après [1], un service est identifié par les champs (SourcePort, DestPort), et un service B est considéré comme dépendant de A si au moins un paquet de A apparaît avant B avec une probabilité assez intéressante pendant un intervalle de temps dit (intervalle de dépendance). Par contre, [3] identifie les services à l'aide d'une inspection profonde des paquets IP, et la définition de la dépendance part du principe que si un service B dépend d'un autre service A alors l'exécution de B commence nécessairement avant A, et ne peut s'arrêter avant que A termine son exécution. Nous avons implémenté ces deux techniques, mais au lieu d'analyser les paquets IP (entêtes et données), on propose d'analyser les enregistrements NetFlow qui contiennent toutes les informations nécessaires pour l'implémentation (les numéros des ports pour identifier des services et le FlowStartTime et le FlowEndTime pour déterminer le temps d'occurrence des flux). L'avantage de notre implémentation est sa simplicité, puisque l'ensemble des paquets appartenant à un même flux est présenté par un seul enregistrement contenant des informations utiles. En bref, les statistiques de NetFlow ont réalisé une partie du travail effectué par les techniques précédentes.

2.2 Modélisation et fonctionnement

2.2.1 Les besoins fonctionnels

Le système de détection de dépendances que nous voulons réaliser doit répondre aux besoins fonctionnels représentés sur la figure 2.1 :

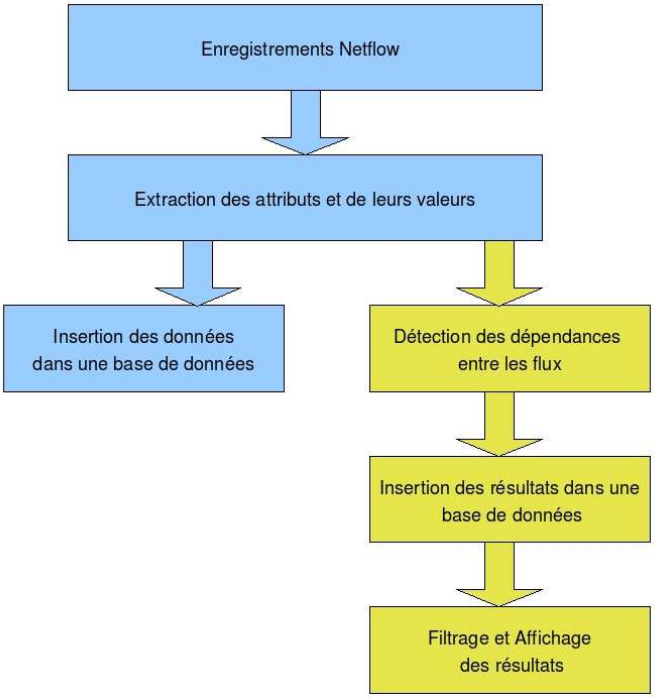


FIG. 2.1 – Besoins fonctionnels

2.2.2 Fonctionnement du système

Le processus de détection global comprend trois étapes : Collection des flux, Détection des dépendances, filtrage des résultats.

Collection des flux Nous avons installé Netflow (version 9) sur un routeur CISCO-2821 qui sert de passerelle par défaut pour 6 hôtes. Pour collecter les enregistrements, nous avons utilisé le collecteur nfdump pour deux raisons : La première est sa compatibilité avec NetFlow v9, et la deuxième est qu'il est capable de sauvegarder les données en format ascii dans un fichier texte.

Date flow start	Date flow end	Duration	Proto	Src IP	Src Pt	Dst IP	Dst Pt	Bytes	Packets
2008-04-21 19:38:07.212;	2008-04-21 19:38:07.212;	0.000;	UDP ;	152.81.48.65;	520;	152.81.48.66;	520;	132;	1 ;
2008-04-21 19:38:35.260 ;	2008-04-21 19:38:41.808;	6.548;	TCP ;	212.58.226.8;	80;	152.81.48.98;	34871;	19603;	18 ;

FIG. 2.2 – Des enregistrements NetFlow

Une fois qu'on a sauvegardé les enregistrements dans un fichier texte, ils seront insérés dans une table dans une base de données (mySQL).

Définition d'un flux : Le flux d'exécution défini par les chercheurs de IBM est un flux qui commence par le premier paquet envoyé du client vers le serveur (requête), et se termine par le dernier paquet reçu par le client (réponse) figure2.3.

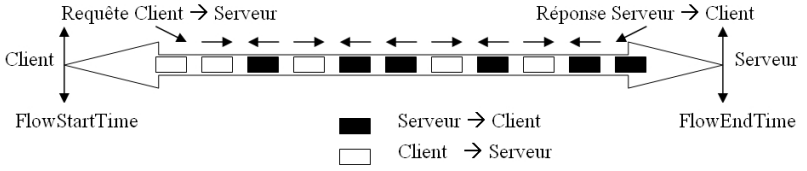


FIG. 2.3 – Définition d'un flux par IBM

Les flux de Netflow ne respectent pas cette définition, parce que chaque flux est unidirectionnel, et le StartFlowTime représente la date de transmission du premier paquet du client vers le serveur (ou du serveur vers le client), et le EndFlowTime représente le dernier paquet reçu par ce serveur (ou ce client) comme illustré dans la figure2.4.

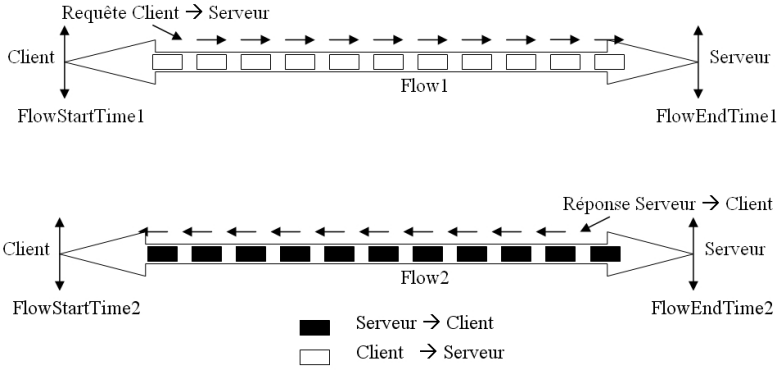


FIG. 2.4 – Un flux Netflow

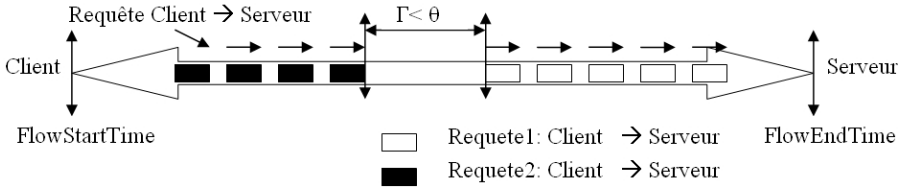
Nous définissons l'exécution d'un service par trois champs. Les flux orientés (Serveur->Client) sont définis par les champs SourceIP, DestinationIP et Source port et ceux qui sont orientés (Client->Serveur) le sont par les champs SourceIP, DestinationIP et DestPort. Par exemple, deux exécutions d'un service par un hôte causent la création de deux enregistrements Netflow puisque chaque exécution utilise un port source différent, alors que logiquement, ces deux enregistrements représentent l'exécution du même service malgré la différence dans les numéros de port source (figure 2.4).

Le contenu d'un flux Netflow : Les paquets d'un flux Netflow n'appartiennent pas nécessairement à une seule instance d'exécution. Ceci dépend de l'intervalle de temps séparant deux exécutions consécutives. Netflow crée un nouvel enregistrement dans deux cas : soit le paquet ne fait pas partie d'un flux existant, ou bien si le paquet appartient à un flux précédent et l'intervalle de temps séparant ce nouveau paquet du dernier était supérieur à une valeur donnée (θ). Si cet intervalle était court, deux instances d'exécutions seront incluses dans un seul enregistrement. Par exemple, lorsqu'un hôte ouvre une page web, deux enregistrements seront créés pour représenter la requête DNS transmise par le client et la réponse DNS reçue (figure 2.5).

Start Date flow	End Date flow	Duration	Proto	Src IP	Src Pt	Dst IP	Dst Pt	Bytes	Packets
X1;	Y1;	Y1-X1 ;	UDP;	ClientIP;	33700;	DNSServerIP;	53;	nbBytes	nbPackets;
X2;	Y2;	Y2-X2 ;	UDP;	DNSServerIP;	53;	ClientIP;	33700;	nbBytes	nbPackets;

FIG. 2.5 – Deux Flux Netflow- requête et réponse

Supposons qu'une nouvelle requête DNS est envoyée par le client à l'instant Y1 + Γ avec $\Gamma < \theta$ Dans ce cas, au lieu de créer un nouvel enregistrement pour représenter la nouvelle instance d'exécution, Netflow met à jour les champs EndDateFlow, Bytes et Packets (figure 2.6).



Start Date flow	End Date flow	Duration	Proto	Src IP	Src Pt	Dst IP	Dst Pt	Bytes	Packets
X1;	Y2;	Y2-X1 ;	UDP;	ClientIP;	33700;	DNSServerIP;	53;	nbBytes+x	nbPackets+y;

FIG. 2.6 – Flux Netflow mis à jour

Cela affecte les résultats de nos algorithmes puisqu'ils sont basés sur le principe que chaque flux représente les paquets d'une seule instance d'exécution d'un service, et par conséquent, des faux positifs apparaissent dans le premier algorithme (IBM), et des dépendances non révélées dans le second (Microsoft).

Les faux positifs : Puisque plusieurs flux sont agrégés sous un seul enregistrement ayant un temps d'exécution [StartDateFlow, EndDateFlow] assez large, celui ci augmente la probabilité d'avoir des flux inclus dans cet intervalle (figure 2.7).

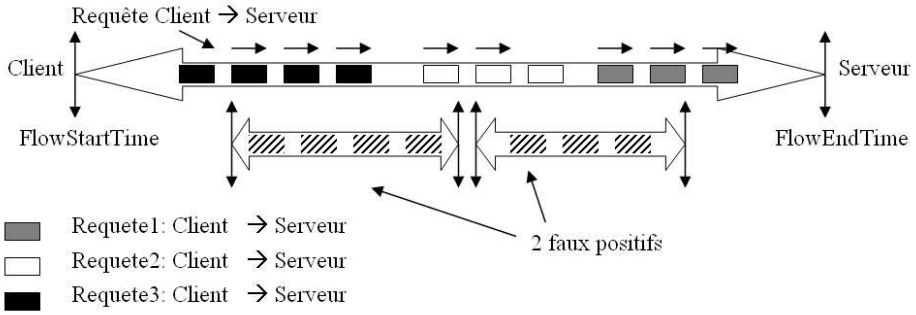


FIG. 2.7 – les faux positifs

Les dépendances non révélées : La définition de la dépendance proposée par Microsoft dans [1] et [2] consiste à regarder le champ StartDateFlow de chaque flux, et comme les flux des différentes instances sont parfois agrégés, seule la première instance sera prise en considération, et toutes les autres qui la suivent seront négligées (figure 2.8).

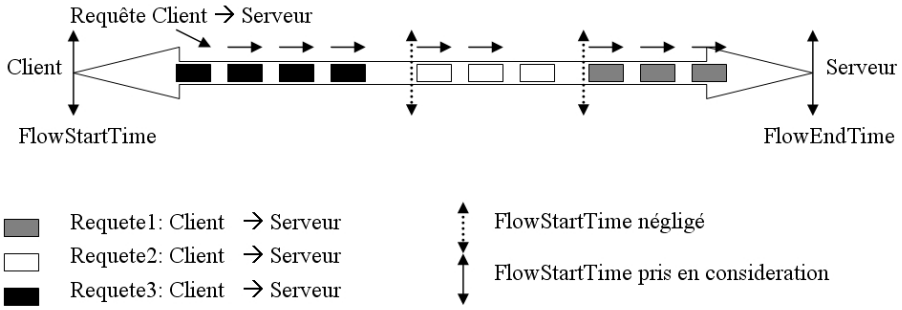


FIG. 2.8 – Des flux négligés

Détection des dépendances : Le fonctionnement de l'application est basé sur un ensemble de requêtes SQL, par exemple pour trouver l'ensemble des flux appartenant à l'intervalle [t1,t2] il suffit d'exécuter la requête :

```
Select * from flows where flows.StartDate>t1 and flows.EndDate<t2;
```

La figure 2.9 présente la matrice de probabilité décrivant les dépendances entre les différents flux.

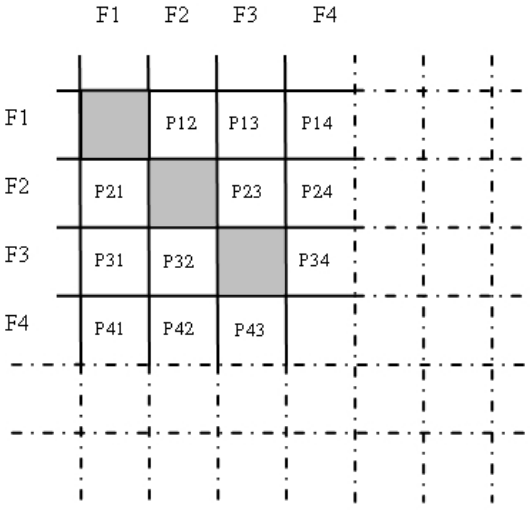


FIG. 2.9 – La Matrice des probabilités

Pij correspond à la probabilité que le flux Fi dépend du flux Fj. Notre implémentation vise à calculer les éléments de cette matrice de deux manières différentes, une respectant la définition de [1] pour la dépendance, et l'autre celle de [3].

Le diagramme d'activité La figure 2.10 présente le diagramme d'activité du programme implantant notre approche. Les enregistrements Netflow sont insérés dans une base de données avant de commencer l'étape de détection des dépendances.

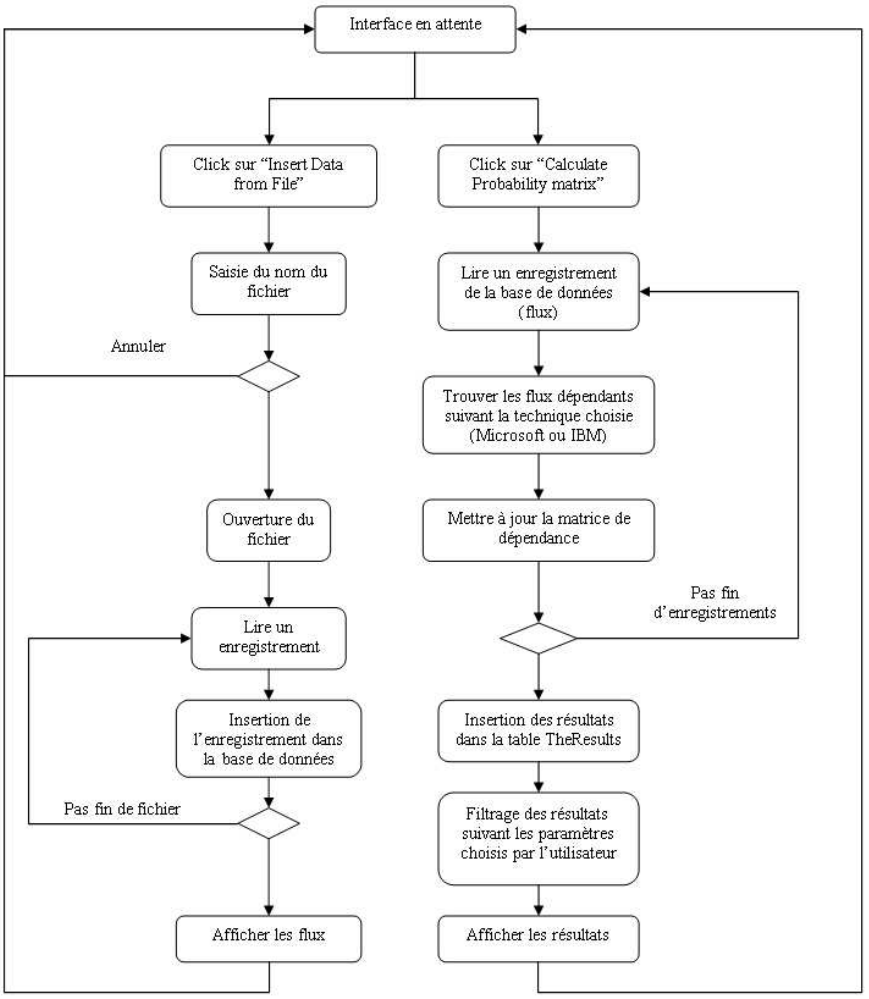


FIG. 2.10 – Le diagramme d'activité du programme

Détéction des dépendances : Comme nous l'avons présenté dans la section 1.1.3[1], il faut pour avoir une détection primaire des dépendances, calculer p($(S_1)\rightarrow(S_2)$) qui est défini par $\#(S_1|S_1{\geq}S_2)/\#(S_1)$. Cependant, même si S_1 ne dépend pas de S_2, S_1 peut contenir S_2 par hasard. Pour faire face à ces faux positifs on définit r comme étant $\#(S_2|S_1{\geq}S_2)/\#(S_2)$.

La fonction UpdateFrequencyMatrix() consiste à calculer les éléments de la matrice de probabilité selon la formule $P_{ij} = max(p, r) + p * r$ avec p=$\#(S_1|S_1{\geq}S_2)/\#(S_1)$ et r=$\#(S_2|S_1{\geq}S_2)/\#(S_2)$.

L'algorithme de calcul de la matrice de probabilité est présenté ci-dessous :

[1] Pour avoir une détection primaire des dépendances il faut calculer p($(S_1)\rightarrow(S_2)$) qui est défini par $\#(S_1|S_1{\geq}S_2)/\#(S_1)$ avec $\#(S_1)$ le nombre total des transactions T(S_1) et $\#(S_1|S_1{\geq}S_2)$ le nombre des transactions T(S_1) contenant au moins une transaction T(S_2). Cependant, même si S_1 ne dépend pas de S_2, S_1 peut contenir S_2 par hasard. Pour faire face à ces faux positifs on définit r comme étant $\#(S_2|S_1{\geq}S_2)/\#(S_2)$ avec $\#(S_2|S_1{\geq}S_2)$ le nombre des transactions T(S_2) contenu par au moins une transaction T(S_1).

Algorithme de détection des dépendances inspiré de [3]

```
Insert Flows into FlowTable
//Calcul du pFrequency matrix
For Each Flow in FlowTable
{

FlowIndex=Flow.ID
pResults=Select * from flows where flows.StartDate> Flow.StartDate
        and flows.EndDate<Flows.EndDate group by ID

For Each Flow in pResults
{
ChildIndex=Flow.ID
pFrequencyMatrix[FlowIndex][ChildIndex]+=1
}

For Each Flow in FlowTable
{
//Calcul du rFrequency matrix
FlowIndex=Flow.ID
rResults=Select * from flows where flows.StartDate< Flow.StartDate
        and flows.EndDate>Flows.EndDate group by ID
        //
For Each Flow in rResults
{
ParentIndex=Flow.ID
rFrequencyMatrix[FlowIndex][ParentIndex]+=1
}
UpdateFrequencyMatrix()
}
```

Filtrage des résultats : Plusieurs facteurs peuvent être pris en considération pour raffiner les résultats et pour différencier entre dépendances et faux positifs. Les facteurs utilisés dans notre implémentation sont :

1. La matrice de dépendance $P_{ij} = max(p, r)+p*r$ avec p=$\#(S_1|S_1 \geq S_2)/\#(S_1)$ et r=$\#(S_2|S_1 \geq S_2)/\#(S_2)$.
2. #F2/#F1 : Ce facteur est le rapport entre le nombre d'occurrences des flux (F1 dépend de F2).
3. #F1 : les facteurs précédents ne donnent pas toujours une idée claire sur la force des dépendances. Prenons l'exemple suivant :

	P_{ij}	#F2/#F1	#F1
F1->F2	0.8	0.3	200
F1->F3	0.8	0.7	2

   La table ci-dessus présente deux dépendances (F1->F2) et (F1->F3). D'après le champ #F1, on voit que la dépendance (F1->F2) est plus forte que celle du (F1->F3) malgré la valeur de 0.7 du champ #F2/#F1 pour la dépendance (F1->F3) ce qui est supérieur à 0.3.
4. Le filtrage des résultats peut être réalisé au niveau de l'algorithme. Comme nous l'avons présenté dans la section 2.1, les flux Netflow sont unidirectionnels et par conséquent, dans la majorité des fois, chaque flux orienté du client vers le serveur (requêtes) sera suivi par un autre du serveur vers le client (réponses) et celui-ci va générer des dépendances entre ces flux. Afin d'éliminer ces dépendances, notre algorithme néglige les dépendances entre les flux de même type. Ce filtrage nous aide à remédier au problème du scan. Un port scan affecte les enregistrements Netflow en causant la génération d'un grand nombre d'enregistrements dont chacun représente une requête suivie par une autre correspondant à la réponse.

Chapitre 3

Evaluation et analyse

Dans ce chapitre, nous testons notre modèle afin de déterminer sa capacité de détection et le maximum de perturbations qu'il peut tolérer. Pour cela, nous avons généré des flux simulant les dépendances entre une application Web et deux bases de données (flux de dépendance). La simulation se déroule comme suit ; un hôte génère des requêtes http, l'application Web reçoit ces requêtes et répond suivant leurs contenus, soit directement soit après l'accès à une des bases de données. Les flux sont paramétrables, le tableau ci-dessous présente la liste des paramètres affectant les flux générés.

Paramètres	Description
ExecTime	Durée d'exécution d'un flux. Suit une loi normale N(ExecTime, ExecTime/10).
NbFlux	Nombre de flux générés ou des instances exécutées par les hôtes.
NbHôtes	Nombre d'hôtes générant du trafic en parallèle.
%BD1	Probabilité avec laquelle une requête est envoyée vers la première base de données.
%BD2	Probabilité avec laquelle une requête est envoyée vers la deuxième base de données.
SimulationStartTime (SST) et SimulationEndTime (SET)	Tous les flux générés ont le champ [FlowStartTime] inclus dans l'intervalle [SST,SET].

Dans un premier temps, nous avons décidé de faire un 'merge' entre les dépendances générées et un fichier de flux Netflow contenant des enregistrements qui correspondent à un trafic réel pour savoir si on est toujours capable de détecter les dépendances injectées. Cette technique a donné des résultats positifs puisque le trafic réel contient des enregistrements correspondants à plusieurs services (http, https, DNS, ...). Par conséquent chaque service prend une partie des enregistrements ce qui explique l'effet négligeable sur nos dépendances.

Par contre, nous avons remarqué que nos résultats peuvent être affectés par des flux relatifs à un service qui s'exécute fréquemment et de manière quasiment périodique. Par exemple, les fichiers Netflow contiennent un grand nombre d'enregistrements NTP. NTP[9], ou protocole horaire en réseau, est un protocole qui permet de synchroniser les horloges des systèmes informatiques à travers un réseau de paquets. Ce type de service peut provoquer des faux positifs pendant la phase de détection puisque la génération d'un grand nombre de flux appartenant au même service augmente la probabilité de présenter des flux indépendants comme étant fortement dépendants.

Afin de déterminer la capacité de détection et le maximum de perturbations que notre technique peut tolérer, nous allons provoquer des flux de bruit d'une manière quasiment périodique appartenant à l'intervalle [SST,SET][2] comme le montre la figure 3.1. Ces flux générés n'ont alors aucune relation avec les flux sur lesquels nous travaillons.

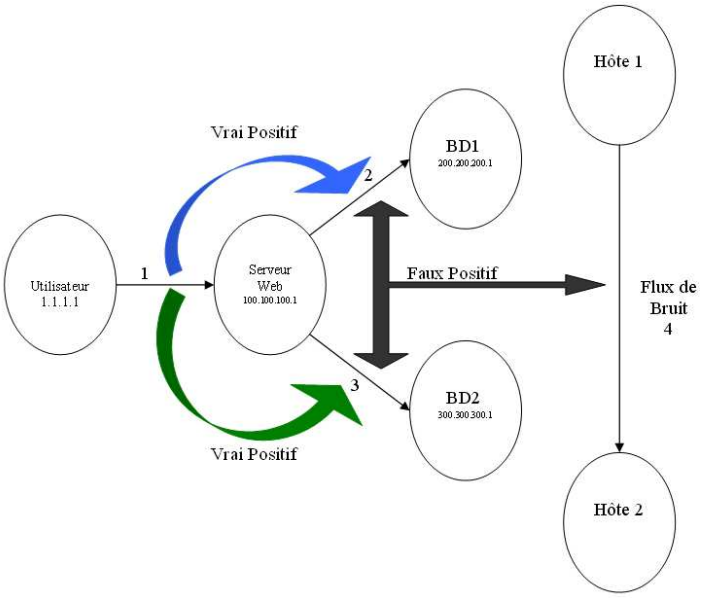


FIG. 3.1 – Modèle de simulation

Notre test se fera, en premier lieu, sur l'effet de la durée de l'exécution des flux de bruit injectés puis sur celui du nombre de ces flux sur la capacité de détection des vrais positifs. Nous allons faire ce test sur un réseau chargé (les flux de dépendance occupent 90% du temps de simulation) et sur un réseau moins chargé (les flux de dépendance occupent 15% du temps de simulation). Dans tout ce qui suit, un réseau est dit chargé s'il contient 500 flux de dépendances ayant chacun une durée d'exécution qui suit une loi normale centrée sur 100ms et d'écart type égal a 10. De même, un réseau non charge est un réseau qui contient 100 flux de dépendances ayant une durée d'exécution qui suit la même loi normale.

Avant de commencer le test, nous devons déterminer l'intervalle de dépendance optimal de la technique proposée par[1]. Rappelons que l'intervalle de dépendance est la durée maximale séparant l'occurrence de deux paquets (deux flux dans notre cas) considérés comme dépendantes. Pour ce faire, nous avons procédé par une suite de mesures des facteurs de dépendances des vrais et des faux positifs pour chaque valeur des intervalles de dépendances. Les résultats de ces mesures nous ont permis d'établir les courbes présentées par les figures 3.2. Dans le contexte de l'expérimentation, nous pouvons conclure que la valeur optimale recherchée correspond à 50 ms.

[2]SST : SimulationStartTime et SET : SimulationEndTime

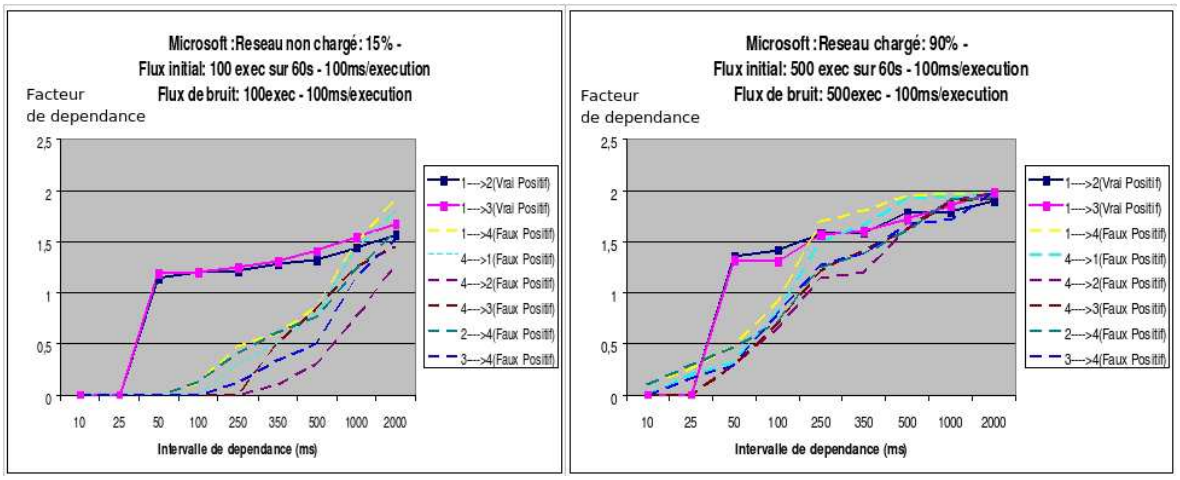


FIG. 3.2 – Effet de l'intervalle de dépendance

Durée d'exécution des flux de bruit : Le premier paramètre que nous avons étudié est la durée d'exécution des flux de bruit. Afin de tester l'impact de ce facteur sur la capacité de détection des dépendances, nous avons injecté des flux de bruit ayant une durée d'exécution qui suit une loi normale centrée sur 10ms et d'écart type égal à 1 et nous avons lancé nos algorithmes de détection de dépendance, ensuite nous avons répété le même test mais avec des flux de bruit ayant une durée d'exécution qui suit une loi normale centrée sur 25ms et ainsi de suite jusqu'à atteindre 3000ms. Nous avons introduit 500 flux de bruit dans un fichier contenant des enregistrements Netflow d'un réseau chargé, et 100 flux dans un autre fichier contenant des enregistrements Netflow d'un réseau non chargé.

D'après les figures 3.3 et 3.4 on remarque que ce paramètre n'a aucun impact sur la détection des dépendances par la technique de Microsoft, malgré la variation de la durée d'exécution des flux de bruit entre 10 et 3000ms, les vrais positifs sont toujours détectables avec un facteur variant entre 1 et 1.4 alors que le facteur des faux positifs n'a pas dépassé 0.58.

Par contre, la variation de la durée d'exécution des flux de bruit a affecté les dépendances révélées par la deuxième technique. Pour les réseaux non chargés, quand ce paramètre dépasse la valeur de 500ms, la différenciation entre vrai positifs et faux positifs paraît impossible. Le pire cas se manifeste dans les réseaux chargés puisqu'il suffit d'avoir des flux de bruit ayant des durées d'exécution supérieures à 100ms pour commencer à considérer des faux positifs comme étant des dépendances fortes. Ceci est du à cause de la définition d'une dépendance par les auteurs de [3].

Nous rappelons que [3] considère deux flux comme étant dépendants si la durée d'exécution de l'un d'eux est inclus dans l'autre avec une probabilité assez intéressante. C'est pour cela que les faux positifs apparaissent avec l'accroissement de la durée d'exécution des flux de bruit puisque la probabilité d'avoir des flux ayant des durées d'exécution inclues dans celles des flux de bruit augmente avec la croissance du paramètre étudié.

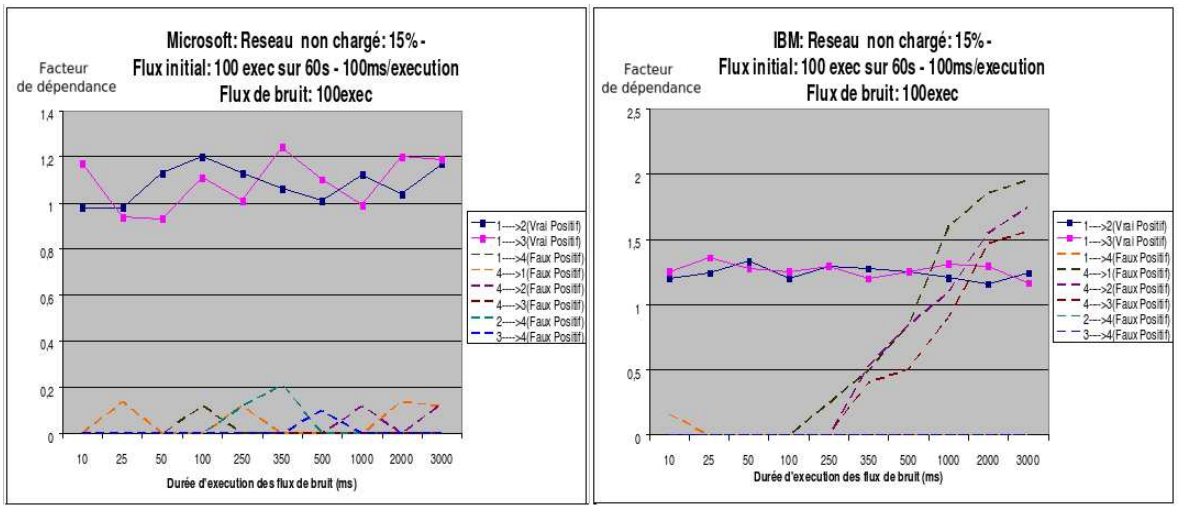


FIG. 3.3 – L'effet de la durée d'exécution des flux de bruit : sur un réseau non chargé

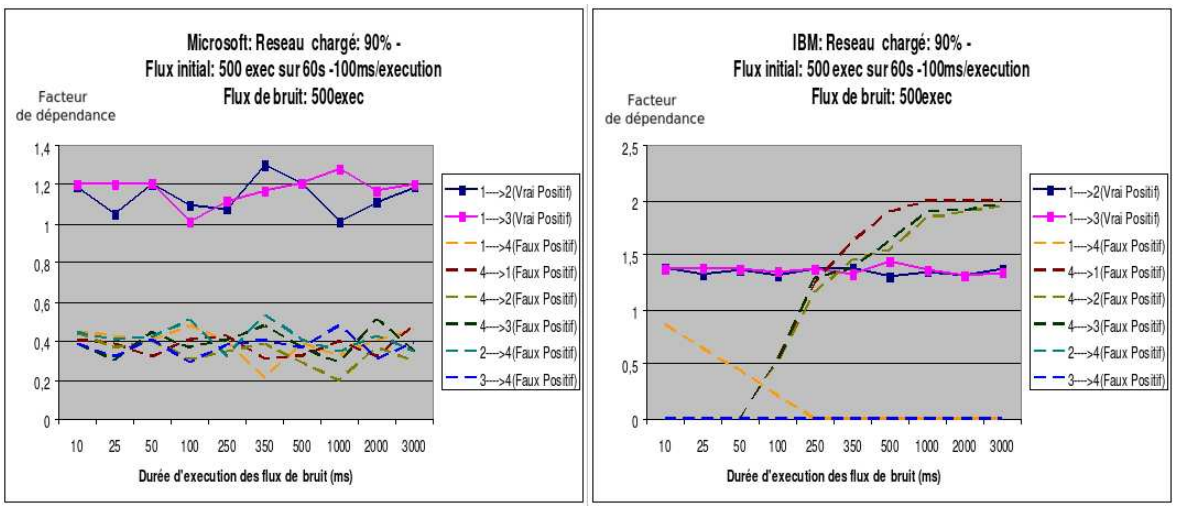


FIG. 3.4 – L'effet de la durée d'exécution des flux de bruit : sur un réseau chargé

Nombre de flux de bruit : Le second paramètre à étudier est le nombre d'instances de bruit injectées. Nous avons fixé le temps d'exécution des flux de dépendances et des flux de bruit à 100ms. Afin de tester l'effet de ce facteur sur la capacité de détection des dépendances nous avons commencé à injecter 10 flux de bruits puis nous avons lancé nos algorithmes de détection de dépendance ([1] et [3]), ensuite nous avons répété le même test mais avec 25 flux de bruit et ainsi de suite jusqu'à atteindre 3000 flux. Les figures 3.5 et 3.6 montrent que l'impact de ce paramètre sur la technique de [1] est plus intéressant que sur celle du [3].

Pour les réseaux non chargés, la technique IBM a pu détecter les vrai positifs avec un facteur variant entre 1.2 et 1.4 alors que le facteur des faux positifs n'a pas dépassé 0.23. L'effet de ce paramètre sur les dépendances révélées par Microsoft paraît clair puisque les facteurs des dépendances existantes et des faux positifs atteignent le même niveau en dépassant la valeur de 1500 flux de bruit injectés pour les réseaux chargés (2000 pour les réseaux non chargés).

La première technique de détection a montré une immunité à la durée d'éxecution des flux de bruit mais elle n'a pas pu faire face à l'augmentation du nombre des flux de bruit. Le cas inverse se produit pour la deuxième technique, où la détection des vrais positifs paraît impossible après l'injection des flux de bruits ayant des durées d'éxecution assez grandes alors que l'augmentation du nombre des flux de bruit l'affecte peu.

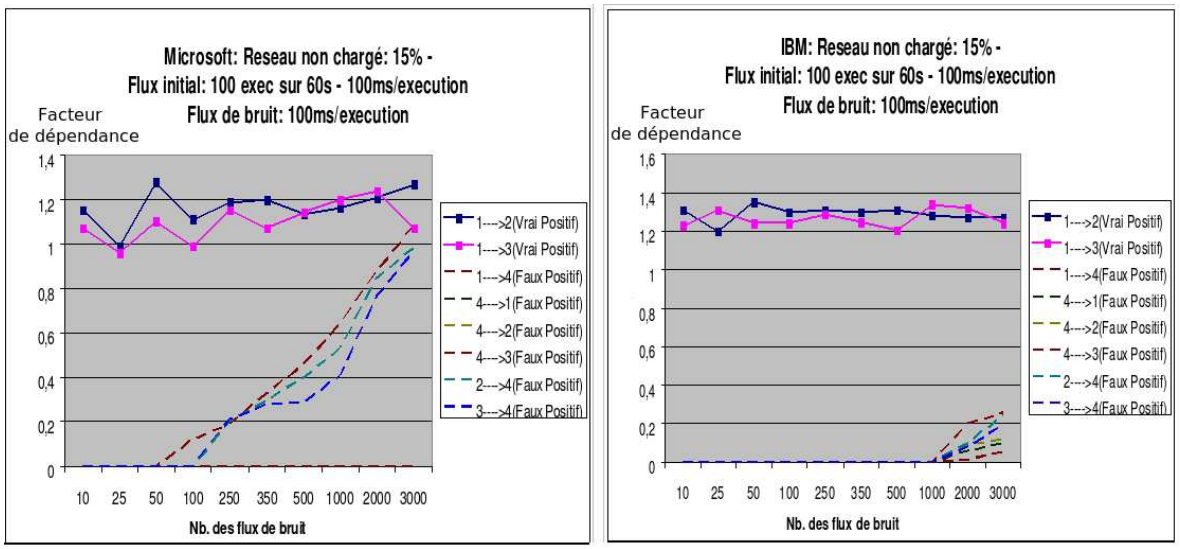


FIG. 3.5 – L'effet du nombre des flux de bruit sur un réseau non chargé

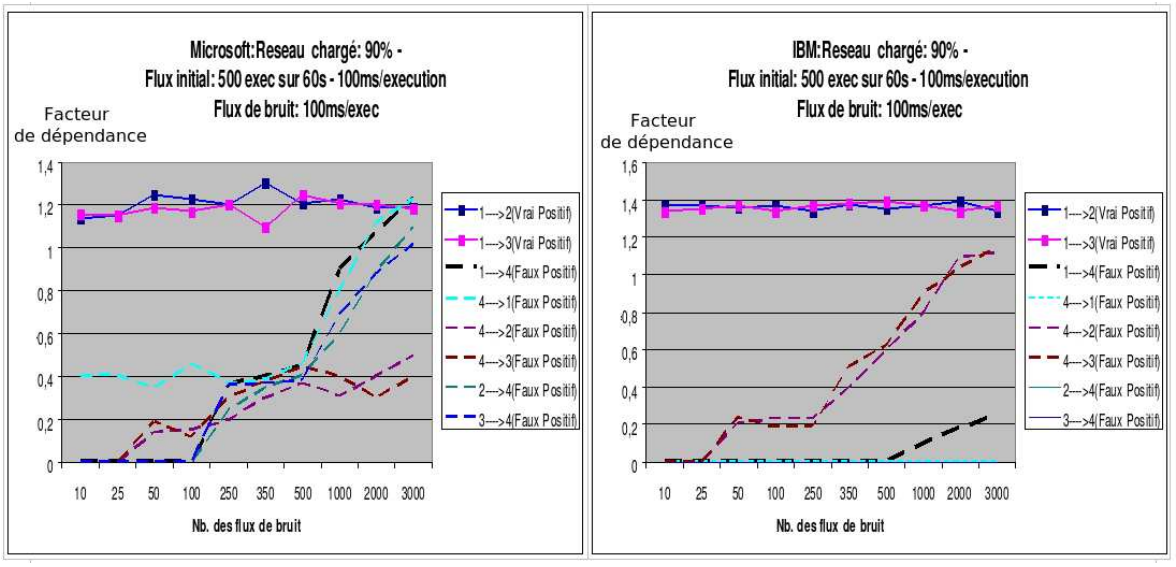


FIG. 3.6 – L'effet du nombre des flux de bruit sur un réseau chargé

En conclusion, on voit que la simulation a donné des résultats satisfaisants dans le cas d'un réseau non chargé (les flux de dépendance occupent 15% du temps de simulation) où nous avons pu détecter tous les vrais positifs et nos techniques de filtrage ont pu éliminer tous les faux positifs.

Cependant, ces résultats ont montré que notre méthode est moins performante pour certains cas de réseaux chargés (les flux de dépendances occupent 90% du temps de simulation) où les faux positifs acquièrent un facteur de dépendance proche de celui des vrais positifs ce qui rend difficile la différenciation entre flux dépendants et indépendants.

Chapitre 4

Conclusion

Dans ce mémoire, nous avons présenté et comparé différentes techniques de détection des dépendances liant les services exécutés à partir d'une analyse du trafic du réseau IP.
Les techniques proposées par Microsoft dans [1] et [2] identifient les services et définissent la dépendance d'une manière différente de celle des chercheurs de IBM [3]. A partir de l'étude de ces techniques, nous avons présenté, une nouvelle méthode basée sur l'analyse des enregistrements Netflow, remplaçant les paquets IP, ce qui présente un point fort compte tenu du fait que les sondes de Netflow, outre leur disponibilité sur de nombreux equipement, réalisent une partie du travail effectué par les techniques précédentes et contiennent les informations nécessaires pour l'implémentation (les numéros des ports pour identifier les services et le FlowStartTime et le FlowEndTime pour déterminer le temps d'occurrence des flux).

Dans un deuxième temps, nous avons implémenté et évalué notre méthode en simulant une communication entre un serveur web et deux bases de données en provoquant volontairement des enregistrements Netflow qui représentent ces dépendances. Nous avons alors étudié l'impact de deux paramètres sur la capacité de détection et le maximum de perturbations que notre technique peut tolérer. Pour ce faire, nous avons généré des flux de bruit paramétrables d'une manière quasiment périodique.

La simulation a donné des résultats satisfaisants dans le cas d'un réseau non chargé (les flux de dépendance occupent 15% du temps de simulation) où nous avons pu détecter tous les vrais positifs et nos techniques de filtrage ont pu éliminer tous les faux positifs.

Cependant, ces résultats ont montré que notre méthode est moins performante pour certains cas de réseaux chargés (les flux de dépendance occupent 90% du temps de simulation) où les faux positifs acquièrent un facteur de dépendance proche de celui des vrais positifs ce qui rend difficile la différenciation entre flux dépendants et indépendants.

Une suite naturelle à nos travaux porte sur l'évaluation à grande échelle de notre approche. Ceci implique cependant la disponibilité d'un grand nombre de traces réseau (de type enregistrement Netflow), traces issues d'un réseau réel. De plus ces traces doivent être annotées, i.e. les dépendances identifiées afin de permettre de mesurer la qualité de l'application de notre algorithme. Cette activité est possible dans le cadre du réseau d'excellence EMANICS dans lequel les chercheurs travaillent à la fois sur la collecte et l'annotation de traces. Cette mise en œuvre sera réalisée cet été en coopération avec l'Université de Twente aux Pays-Bas qui fournit les traces issues de son campus.

Une seconde poursuite porte sur l'étude d'une approche hybride permettant d'adapter dynamiquement le choix de la technique en fonction de la caractéristique du trafic (charge, nature des bruits, ...). L'objectif ici étant d'élaborer le modèle de sélection permettant de basculer d'une approche à l'autre.

Bibliographie

[1] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula,David A. Maltz, Ming Zhang, *Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies , SIGCOMM'07, August 27 -31, 2007*

[2] Paramvir Bahl, Paul Barham, Richard Black, Ranveer Chandra, Moises Goldszmidt, Rebecca Isaacs, Srikanth Kandula, Lun Li, John MacCormick, David A. Maltz, Richard Mortier,Mike Wawrzoniak, Ming Zhang. Microsoft Research ; also Caltech ,MIT and Princeton,*Discovering Dependencies for Network Management, 2007*

[3] Hisashi Kashima Tadashi Tsumura, Tsuyoshi Idé, Takahide Nogayama, Ryo Hirade, Hiroaki Etoh Takeshi Fukuda, *Network-based Problem Detection for Distributed Systems, IEEE, 2005*

[4] Andrew Turner, Hyong. S. Kim, Tina Wong,*Automatic Discovery of Relationships Across Multiple Network Layers, INM'07, August 27-31*

[5] Alexander Keller, Gautam Kar,*Determining Service Dependencies in Distributed Systems, Finland, Helsenki, June 2001*

[6] Saurabh Bagchi, Gautam Kar, Joe Hellerstein,*Dependency analysis in distributed systems using fault injection : Application to problem Determination in an e-commerce Environment, Hawthorne, New york*

[7] A.Brown, Alexander Keller, Gautam Kar,*An Active Approach to characterizing dynamic dependencies for problem determination in a distributed environment, may 2001*

[8] Resource Description Framework (RDF), http ://www.ietf.org/rfc/rfc3870.txt.

[9] Network Time Protocol (NTP), http ://www.faqs.org/rfcs/rfc1305.html.