UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DIAGNOSTIC AUTOMATISÉ DE PROBLÈMES RÉSEAU

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

STEVE BOUCHER

SEPTEMBRE 2019

REMERCIEMENTS

Ce travail de recherche n'aurait pas eu lieu sans la collaboration de plusieurs personnes à qui j'aimerais adresser des remerciements.

Je tiens tout d'abord à remercier mon directeur de recherche, M. Roger Villemaire. Il a été disponible pour mes questions, et ses précieux conseils m'ont guidé durant toute ma maîtrise.

Je veux aussi remercier mon patron et la direction de Rheinmetall Canada inc, où je travaille depuis presque quinze ans. Ils m'ont donné le financement et le temps nécessaires pour faire mes études, et ce depuis le début de mon parcours universitaire. Ils sont toujours ouverts à aider leurs employés à continuer leur formation peu importe le niveau, et je leur en suis très reconnaissant.

Je tiens à remercier M^{me} Julie Lalancette pour ses précieux conseils sur la langue française.

Enfin, un gros merci à M. André Poirier et ses collègues du Géotop avec qui, autour de solutions houblonnées, j'ai passé plusieurs heures à philosopher sur le processus scientifique.

Merci à tous.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Les réseaux informatiques font partie intégrante des entreprises d'aujourd'hui. Un réseau local typique sera composé de plusieurs systèmes différents dont un ou plusieurs de ces systèmes seront dépendants d'un ou de plusieurs autres systèmes, et qui, eux-mêmes, seront dépendants d'autres systèmes, et ainsi de suite. Le diagnostic de problèmes reliés à ces systèmes fait partie du quotidien des administrateurs.

Le but de ce travail est d'élaborer et d'évaluer une méthode pour automatiser la détection et la modélisation des services et de leurs dépendances dans un parc de serveurs Windows et, ensuite à partir de ce modèle, de faire la détection d'anomalies qu'on peut rencontrer à l'intérieur d'un réseau local typique d'aujourd'hui. La détection des services et de leurs dépendances a été comparée avec NSDMiner [1], une méthode utilisant un analyseur de paquet afin de détecter les dépendances du réseau.

Les résultats ont démontré que notre méthode peut être nettement supérieure à la méthode utilisant un analyseur de paquet et que la détection d'anomalies est possible avec le modèle généré.

Mots-clés : diagnostic automatisé, détection de dépendances

INTRODUCTION

Les réseaux informatiques jouent un rôle important dans la société d'aujourd'hui. Ils constituent généralement un aspect essentiel des entreprises et toutes ont leurs besoins et leurs infrastructures informatiques bien spécifiques. Le choix technologique et la complexité de l'infrastructure informatique peuvent être différents d'une entreprise à l'autre pour toutes sortes de raisons et dépendent en général de l'historique des décisions de l'entreprise. Le réseau informatique fait partie du quotidien de l'entreprise et, comme pour toute chose, il doit être maintenu en état fonctionnel.

Au cœur de ces réseaux, il y a les systèmes informatiques qui sont utilisés. Ces systèmes sont des services ou des ensembles de services qui remplissent une fonction précise. Chaque réseau peut être constitué d'un ou de plusieurs systèmes. Les systèmes peuvent être d'importances différentes et être installés en utilisant différentes plateformes de serveurs (la quincaillerie où les systèmes sont installés). Ils peuvent être centralisés ou non, être hébergés dans un nuage ou isolés dans une salle de serveur. Ces systèmes peuvent être installés sur une machine physique ou virtuelle. L'accès réseau de ces systèmes entre eux peut aussi se faire de façon différente de réseau en réseau.

Un systèmes a toujours des dépendances directes comme être installé sur du matériel fonctionnel qui a une connexion réseau fonctionnelle elle aussi. Chaque système n'est pas nécessairement seul dans son monde. Il peut dépendre du bon

fonctionnement d'un autre système. Le bon fonctionnement de plusieurs systèmes peut être dépendant du bon fonctionnement de plusieurs autres qui, eux-mêmes, sont dépendants d'autres.

Un exemple simple serait une application A_1 sur un serveur S_1 qui utilise une base de données sur un serveur S_2. Que se passe-t-il, si pour une raison inconnue, le serveur S_2 n'est plus disponible et que la base de données n'est plus accessible ? Il est clair que cela affecterait l'application A_1 qui est dépendante de la base de données, même si les deux serveurs S_1 et S_2 sont installés à des kilomètres l'un de l'autre.

Le diagnostic de problèmes reliés à ces systèmes est la responsabilité des administrateurs. Lorsqu'un problème survient, le diagnostic peut se faire en plusieurs étapes, de la lecture de journaux d'événements jusqu'à la découverte de matériel défectueux, en passant par des tests systématiques sur chacun des maillons d'une longue chaîne de dépendances. Le processus peut prendre un certain temps et, bien entendu, ces dépendances doivent être connues. Prendre connaissance des dépendances entre les systèmes est aussi une des responsabilités des administrateurs. Souvent, lors du diagnostic d'un problème, découvrir les dépendances d'un système peut faire partie du diagnostic même.

Le but de ce travail est d'élaborer et d'évaluer une méthode permettant de réduire le temps d'attente pour poser un diagnostic de problèmes réseau. Tout d'abord, notre méthode fait la détection des dépendances des services du réseau de façon automatique et modélise le tout sous forme de séries de graphes. Ensuite, des algorithmes utilisant ces graphes ont été développés afin d'automatiser les diagnostics.

Ce mémoire est structuré de la façon suivante.
Au chapitre 1, nous présentons l'état de l'art sur le diagnostic réseau et la détection de dépendances. Nous aborderons ensuite les différentes problématiques qu'on peut identifier dans les récents travaux de recherche. Nous élaborerons aussi sur les objectifs de ce mémoire.

Au chapitre 2, nous aborderons la notion d'architecture d'un parc de serveurs et présenterons quelques outils utiles à la collecte d'information dans un réseau local. Ensuite, nous aborderons quelques notions de la théorie des graphes, qui sont importantes pour comprendre les algorithmes présentés plus loin dans le document.

Le chapitre 3 sera consacré à l'explication de toutes les étapes de la construction du modèle utilisé dans notre méthode, en partant de la détection automatique des serveurs du réseau jusqu'à sa représentation interne.

Le chapitre 4 présentera le détail des algorithmes utilisés dans le projet afin de construire un modèle fidèle du réseau et faire la détection d'anomalies.

Le chapitre 5 présentera des scénarios de panne ainsi que les résultats obtenus lors de la détection du réseau et des anomalies à l'aide de notre outil.

La conclusion de ce mémoire portera sur l'évaluation générale des résultats obtenus ainsi que sur de futures avenues de recherches.

CHAPITRE I

MISE EN CONTEXTE

Dans ce chapitre, nous présentons l'état de l'art sur le diagnostic réseau et la détection de dépendances. Nous aborderons ensuite les différentes problématiques qu'on peut identifier dans les récents travaux de recherche. Nous élaborerons aussi sur les objectifs de ce mémoire.

1.1 Diagnostic de problèmes réseau

En informatique, le mot *réseau* est utilisé à toutes les sauces. Il peut vouloir dire un réseau de télécommunications, les mécanismes de transmission de paquets TCP/IP, l'architecture de routeurs et de commutateurs, un parc de serveurs informatiques, etc. Notre recherche se limite uniquement aux parcs de serveurs Windows, car les méthodes de diagnostic sont différentes lorsque l'on diagnostique un problème sur un serveur Windows que sur un commutateur. Aussi, le choix de se limiter au serveurs Windows a été fait simplement pour limiter le temps de développement. Cependant, la revue de littérature a couvert beaucoup plus large, car les méthodes présentées pour tous les types de réseau sont intéressantes.

Les deux articles suivants [2, 3] sont des revues d'articles qui nous ont servi de point de départ dans le domaine du diagnostic réseau. Ils nous proposent un

survol de plusieurs approches algorithmiques telles que les arbres de défaut, la logique floue, les réseaux de neurones, les réseaux bayésiens, les arbres de décision, etc. Toutes ces approches peuvent être utilisées lors du diagnostic d'un problème (*fault localization*) et donnent des indications sur la façon dont pourraient être fais la représentation interne des données de diagnostic et, surtout, le diagnostic proprement dit.

L'article suivant [4] présente d'ailleurs plusieurs distinctions entre différents concepts tels qu'une anomalie (*fault*), une erreur et un symptôme. Les anomalies sont un problème ou la source d'un problème. Une erreur représente une différence entre un comportement attendu et le comportement réel. Les symptômes sont les manifestations d'une anomalie. Dans le même article, on fait aussi la différence entre la détection d'une anomalie et la localisation d'une anomalie. La détection est la découverte d'une anomalie tandis que la localisation est tout le travail d'analyse afin d'en trouver la source.

Dans les trois articles, le mot *réseau* fait surtout référence au processus de transmission de paquets et à tout le matériel (routeurs, etc.) nécessaire à un réseau de communication. Il y a très peu de références à un parc de serveurs et aux systèmes que des serveurs peuvent héberger dans ces réseaux.

1.2 Problématiques

À la lecture de l'état de l'art, on peut identifier plusieurs problématiques. Même si un grand nombre d'articles toucheront de près ou de loin chacune d'elles, on peut les rassembler en trois catégories que nous aborderons aux points suivants.

1.2.1 La collecte d'information

Avant de pouvoir diagnostiquer un réseau et signaler un problème, il faut avoir la connaissance du réseau en place. Il faut aussi savoir ce qu'est une situation normale par rapport à une situation anormale dans le réseau. Cette information ne vient pas toute seule. La recherche va dans plusieurs sens sur cette problématique.

Les articles [5, 6] présentent des systèmes experts dans lesquels l'information doit être manuellement entrée dans une base de connaissance. Bien que la détection des équipements réseau soit automatique, les règles qui forment la connaissance du domaine sont faites manuellement par un expert du domaine. L'article suivant [7] utilise le *case-based resoning*, qui donne la possibilité au système d'évoluer à mesure que la base de connaissance grossit. Ici, ce n'est pas la détection des composantes réseau qui est faite automatiquement, mais plutôt celle des règles qui définissent la connaissance des étapes à suivre lors d'un diagnostic qui peuvent évoluer dans le temps lorsque la base de connaissance prend du volume. Les deux approches peuvent aussi être combinées [8].

D'autres approches permettent, également, de prendre automatiquement connaissance du réseau. Par exemple, [9] fait la collecte d'information sur les équipements réseau au moyen du protocole SNMP [1] [10]. Cependant, il y a très peu d'efforts mis sur un diagnostic en cas de problème, on se limite plutôt au monitoring des équipements. NSDMiner [1] fait la découverte des services en écoute sur le réseau en capturant les paquets par un analyseur de paquet et calcule la probabilité qu'un service soit dépendant d'un autre en analysant les communications réseau.

---

1. Simple Network Management Protocol est un protocole réseau défini par l'IETF (Internet Engineering Task Force) utilisé pour la collecte d'information sur des équipements réseau.

L'intuition est que si un service S_1 dépend d'un service S_2, alors, lorsqu'un client C_1 accède à S_1, le service S_1 devrait communiquer avec S_2 et recevoir une réponse avant de lui-même répondre à C_1. Leur recherche démontre que NSDMiner est supérieur à deux autres produits commerciaux, mais leurs résultats laissent voir aussi un très haut taux de faux positifs (une dépendance qui est détectée, mais qui n'en est pas réellement une). Même constat pour TE-CyberSNAP [11], qui utilise la même technique de capture par un analyseur de paquet, mais emploie plutôt le *transfert d'entropie* comme méthode statistique afin de déterminer les dépendances entre les services. Cet outil semble avoir de meilleurs résultats que NSDMiner sans que l'article n'indique de mesures quantitatives sur le nombre de vrai positifs ou de faux positifs. Cependant, ils avouent eux-mêmes qu'il faudrait avoir une approche statistique plus poussée afin de réduire le nombre de faux positifs.

Cette approche d'utiliser un analyseur de paquet afin de pouvoir détecter les dépendances des services du réseau a été analysée et critiquée dans [12]. Les auteurs démontrent que l'analyse de paquet est peu fiable pour la détection de dépendances puisque cette méthode a tendance à donner beaucoup de faux positifs. Cela s'explique, entre autres, par le fait que ces méthodes ne prennent pas en compte les différents types de cache qui peuvent être utilisés à l'intérieur de plusieurs systèmes. De plus, une simple latence due à un accroissement du trafic sur le réseau peut aussi aider à fausser les données.

Outre les méthodes utilisant des captures de paquets, il y a des méthodes un peu plus intrusives telles que [13] qui installera un agent sur chaque serveur afin de monitorer les interactions locales pour ensuite échanger cette information avec les autres agents et, à la fin, construire un modèle statistique pour découvrir les

dépendances. L'article [14] est un peu plus astucieux et créera volontairement un délai dans les transmissions de paquets sur le réseau dans le but de découvrir quel service sera affecté, et ainsi établir quel service dépend de quel autre.

1.2.2 La modélisation de la connaissance du réseau

Dès que la collecte d'information à propos du réseau est complétée, il reste le défi de tout modéliser cette information dans une structure utilisable par des algorithmes. La recherche a plusieurs propositions sur ce sujet, quoique la plupart des travaux modéliseront un réseau en utilisant un graphe.

Plusieurs recherches utilisent les graphes comme représentation du réseau [15, 16, 17, 18]. Par exemple, l'article [19] nous propose une série de façons de modéliser un réseau ainsi que toutes ses composantes et ses dépendances sous forme de graphes. Comme l'information n'est pas toujours exacte, plusieurs méthodes sont aussi proposées pour pallier l'incertitude et extrapoler l'information manquante, s'il y a lieu. On y voit aussi des méthodes afin de pouvoir modéliser l'évolution d'un réseau dans le temps avec des séquences de graphes.

Il existe d'autres façons de modéliser la représentation d'un réseau. Par exemple, [20] utilise un modèle d'entité relation afin de faire une représentation interne du réseau. Des règles sont ensuite générées à partir des états-transitions du modèle. Le système reçoit alors les alarmes venant du réseau et utilise les règles générées afin d'en faire une corrélation et, ultimement, trouver l'origine de la faute.

L'article [21] a une approche totalement différente. Cette recherche propose une méthode où toutes les composantes ont un état fonctionnel/non fonctionnel

et tout le processus de diagnostic est réduit au problème SAT [2], où chaque variable représente l'état d'une composante.

1.2.3 La détection et l'identification de la source d'une anomalie

Avec un réseau modélisé sous une forme donnée, l'utilisation de plusieurs algorithmes devient possible afin de détecter et isoler la source d'une anomalie. Par exemple, [21] vu plus haut modélisera le problème sous la forme d'une formule SAT. Par conséquent, les algorithmes de résolution SAT qui ont connus beaucoup de succès dans les dernières décennies pourront être utilisés afin de déterminer la source d'une anomalie. Dans [22], on utilisera un réseau bayésien afin de trouver les anomalies d'un réseau.

Comme la plupart des recherches utilisent le graphe afin de modéliser un réseau, plusieurs algorithmes utilisant les graphes sont présentés dans les articles scientifiques. Dans [19], on peut voir plusieurs algorithmes de calcul de distance d'édition de graphes et de séquences de graphes utilisés dans le but de trouver des anomalies dans le réseau.

Plusieurs recherches détecteront les anomalies dans un réseau en appliquant des transformations sur des séquences de graphes qui représentent l'évolution d'un réseau dans le temps [16, 23, 24]. Dans [16], la représentation sera faite de graphes où les nœuds représenteront des équipements réseau et les liens seront des dépendances, tandis que dans [23, 24] les liens seront des communications réseau. Dans [25], les auteurs mettront des attributs *temps* à chaque lien des graphes qui représentent une communication réseau. En regroupant les liens des graphes, qui

---

2. Le problème de satisfiabilité d'une formule propositionnelle.

représentent une communication d'un certain intervalle de temps donné, on peut voir un patron émerger pour un certain système. Les anomalies sont détectées lorsque les liens des graphes, et donc les communications, dérogent aux patrons qui ont été identifiés. Dans [26], on va fusionnera des parties de graphe dans le but de détecter les anomalies. Dans [27] et [28], on observera les changements dans un graphe qui évolue dans le temps afin de détecter les anomalies.

1.3 Objectifs

Presque toutes les recherches mentionnées s'adressent aux problèmes de réseaux de communication de tout genre. Dans cette recherche, nous allons cibler des problèmes de réseaux locaux au lieu de problèmes de routage, de transmissions TCP/IP ou de fluctuations réseau. Plus précisément, notre méthode s'adresse aux parcs de serveurs contenant des serveurs Windows. Nous ne nous pencherons que sur les serveurs Windows afin de limiter le temps de développement, mais notre méthode est assez générale et pourrait s'appliquer aux serveurs Linux avec plus de temps de développement.

Nous allons développer, analyser et évaluer une méthode qui s'attaquera aux trois problématiques vues plus haut. Notre méthode fera donc la collecte d'information des services installés sur tous les serveurs Windows dans un réseau donné. La méthode de détection utilisée est nouvelle et sera une contribution de cette recherche. Nous allons ensuite modéliser l'information du réseau en utilisant des graphes de façon similaire à ce qu'on voit dans la littérature. Pour finir, nous allons effectuer la détection et l'identification d'anomalies en appliquant des algorithmes similaires à ce qu'on voit dans la littérature, mais appliqués à notre modèle.

En conséquence, il y aura deux volets à la méthode que nous allons élaborer. Le premier sera la détection des systèmes en place, la découverte de leurs dépendances et la modélisation des interdépendances du réseau. Le second volet portera sur l'élaboration d'une méthode algorithmique pouvant détecter les anomalies du réseau à l'intérieur du modèle.

L'évaluation de la modélisation sera faite en comparant notre modèle et nos résultats avec ceux de NSDMiner [1] qui est l'outil de détection de dépendances qui se rapproche le plus de notre système. Ensuite, nous évaluerons la capacité de notre méthode à détecter certains types d'anomalies qu'il pourrait y avoir dans le réseau.

Bien entendu, un système complet devrait détecter toutes les composantes de tous les systèmes sur le réseau, diagnostiquer automatiquement tous les problèmes et appliquer des correctifs en ordre de priorité, mais dans le cadre de cette maîtrise nous nous concentrerons sur la détection des systèmes et le diagnostic de problèmes.

CHAPITRE II

NOTIONS PRÉLIMINAIRES

Dans ce chapitre, nous aborderons des notions importantes à la bonne compréhension des chapitres suivants.

2.1 Architecture d'un parc de serveurs

En général, dans le réseau informatique d'une entreprise, nous verrons plusieurs utilisateurs travailler sur leur poste de travail et accéder à des ressources ou des applications hébergées sur des serveurs qui, eux, hébergeront plusieurs applications dans le but de servir les usagers.

2.1.1 Les serveurs

Qu'il s'agisse de Microsoft Windows [29] ou de Linux [30], les installations des serveurs sont très similaires aux installations des postes de travail qu'on utilise tous les jours. Cependant, les serveurs sont installés et configurés dans le but d'héberger une ou plusieurs applications. Ces applications peuvent prendre différentes formes telles que des applications web, des applications de base de données, des partages de fichiers, des files d'impression, etc. Les systèmes d'exploitation serveur ont essentiellement besoin des mêmes ressources matérielles qu'un poste de

travail. Ils nécessitent de l'espace disque et un accès réseau qui constituent en soi des composantes dont le serveur dépend pour son bon fonctionnement. L'aspect logiciels des serveurs est installé la plupart du temps sous forme de service.

2.1.2 Les services

Les services Windows ou Linux (bien que la nomenclature soit différente dans une plateforme Linux) représentent un ou plusieurs processus qui travaillent ensemble pour accomplir une fonction précise. Ainsi, un serveur peut être considéré comme un ensemble de services.

2.1.3 Les dépendances des services

Afin d'accomplir leurs fonctions, plusieurs services ont besoin d'avoir accès à d'autres services qui soient aussi fonctionnels. Il y a donc des dépendances de services vers d'autres services. Cela prend la forme de dépendance parent-enfant où un service X(enfant) dépend d'un service Y(parent) pour bien fonctionner. Certaines de ces dépendances peuvent être des services installés localement (sur le même serveur) et d'autres peuvent être installés sur un autre serveur et accessibles par le réseau.

2.1.4 Un réseau informatique en tant qu'ensemble de dépendances

Comme un serveur peut être vu comme un ensemble de services, un réseau qui regroupe plusieurs serveurs peut être vu comme un ensemble d'ensembles de services où les services sont potentiellement dépendant de services installés localement à un serveur donné ou ailleurs dans le réseau. Chacune de ces dépendances peut aussi être dépendante de matériels spécifiques, tels qu'un espace disque ou une

pièce d'équipement spécifique, etc.

Un réseau, avec sa panoplie de dépendances, peut être représenté comme un graphe orienté où chaque nœud représente un serveur, un service, un disque, un accès réseau ou tout autre élément de configuration ou matériel qu'un service peut avoir comme dépendance et les dépendances sont représentées par les arêtes entre les nœuds dans une relation parent-enfant. Cette recherche s'est limitée uniquement aux dépendances entre services, cependant toutes les formes de dépendance pourraient être modélisées. Les serveurs ont aussi été inclus dans les modèles parce que chaque service a une dépendance directe avec le serveur sur lequel il est installé.

2.1.5 Les particularités et problèmes possibles dans un ensemble de services

Il existe plusieurs problèmes qu'on peut rencontrer dans un réseau informatique, notamment un bris matériel, un manque de ressources ou un problème de configuration de logiciels. Cela fait partie des tâches des administrateurs de les diagnostiquer et de les corriger. Bien sûr, il serait impossible de lister tous les problèmes qui peuvent subvenir dans une salle de serveur. Cependant, peu importe le problème, il y aura très souvent une corrélation. En effet, si le service A dont le service B dépend est en problème, le service B aura de fortes possibilités d'être en problème aussi. Il y a par contre quelques particularités. Tous les changements d'état ou de dépendance ne vont pas nécessairement signifier un problème, comme nous le verrons aux points suivants.

2.1.5.1 L'importance des dépendances entre les services

Un service peut avoir plusieurs dépendances. Cependant, chaque dépendance peut avoir une importance différente. Le service A peut dépendre du service B, et si celui-ci est en problème, le service A peut ne plus fonctionner du tout. Mais il est possible également qu'il puisse continuer à fonctionner sans être optimal. Afin de limiter le cadre de ce projet de recherche, nous ne traiterons pas de l'importance entre les dépendances de services, mais cela pourrait constituer une autre recherche en soi. Pour cette recherche, nous ferons abstraction de l'importance d'une dépendance.

2.1.5.2 Les services aux états intermittents

Quelques services s'exécuteront de façon intermittente. Un exemple est le service BITS (Background Intelligent Transfer Service) [31] qui s'exécutera au besoin sur un serveur. Les services de ce type ne sont pas nécessairement en problème lorsqu'ils sont éteints. C'est pourquoi vérifier l'état d'un service semblable n'est pas viable pour un diagnostic puisque le service peut être dans tous les états au cours de son fonctionnement normal. La subtilité ici est que, bien que certains services puissent être dépendants d'un service à état intermittent, aucun n'a besoin qu'il soit toujours en exécution. Donc, dans le but de limiter la recherche et surtout de faire un diagnostic en partant du principe qu'un service doit avoir ses dépendances fonctionnelles pour fonctionner, les nœuds représentant des services intermittents seront éliminés. Ils ne sont pas intéressants pour un diagnostic basé sur leur statut d'exécution si leur statut d'exécution est intermittent dans le cadre normale de leurs fonctions.

2.1.5.3 Les dépendances locales par rapport aux dépendances distantes

Un service peut dépendre d'un service installé localement sur un serveur. Autrement dit, un service peut dépendre d'un autre service installé sur le même serveur. Cependant, il peut aussi dépendre d'un service installé sur un autre serveur ailleurs dans le réseau. Une dépendance distante prend la forme d'une connexion réseau qu'un service A peut faire au service B dont il dépend, installé ailleurs dans le réseau. Or, ici, la dépendance distante elle-même est dépendante de l'état du réseau. Nous ne tenons pas compte des dépendances de ce type dans cette recherche, elles pourraient faire l'objet d'une autre recherche. Cependant, un service X qui reçoit d'habitude des connexions réseau de plusieurs services de partout dans le réseau et qui subitement n'en reçoit plus aucune peut signaler un problème de réseau, et cette situation sera évaluée dans nos scénarios.

2.1.5.4 La redondance, les grappes et les migrations

Si le service A dépend du service B_1 et que B_1 est en problème, il est possible que le service A bascule simplement vers le service B_2. Il est courant dans un parc de serveurs que certains services critiques au bon fonctionnement du réseau soient installés en redondance (ou en grappe) de façon à ce que, si un service tombe en panne, les requêtes soient simplement redirigées vers un service redondant installé sur un autre serveur. On peut voir aussi le même comportement si le service B_1 qui était installé sur le serveur X est maintenant installé sur le serveur Y simplement parce que l'administrateur a migré le service de serveur. Ce sont là des scénarios où il y a un changement dans les dépendances d'un service sans pour autant signifier qu'il y a un problème.

2.1.6 Conclusion

Les serveurs Windows ou Linux peuvent être vus comme un ensemble de services, dont certains peuvent avoir une ou plusieurs dépendances, et un parc de serveurs peut être vu comme un ensemble de services et de dépendances. Il existe plusieurs particularités telles que les services intermittents et ceux qui sont en redondance ou en grappe. La structure mathématique la plus appropriée pour modéliser toute cette information, c'est le graphe, et nous élaborerons notre représentation en détail un peu plus loin dans le document.

2.2 Outil de collecte d'information

Un des objectifs de ce travail de recherche est de pouvoir modéliser de façon automatique un réseau sous forme de graphe. Bien entendu, pour permettre cette modélisation, nous devons nous servir d'outils capables d'aller chercher l'information dont nous avons besoin directement sur les serveurs. Dans cette section, nous ferons une description sommaire de ces outils.

2.2.1 WMI

WMI (Windows Management Instrumentation) est un outil de gestion des systèmes d'exploitation Windows. C'est l'implémentation Microsoft de la norme Web-Based Enterprise Management (WBEM) [32], qui est une initiative de l'industrie pour développer des normes sur la gestion des systèmes dans un réseau d'entreprise. WMI est aussi basé sur le standard CIM (Common Information Model) [33], qui est un modèle de représentation de réseaux informations. WBEM [32] et CIM [33] sont tous les deux développés par le DMFT (Distributed Management Task Force).

WMI permet de lancer des commandes distantes à un serveur Windows avec une syntaxe très similaire à celle d'une requête SQL. Par exemple, la commande *SELECT * FROM Win32_service* retournera la liste de tous les services d'un serveur ou d'une station Windows. La liste retournée n'est pas simplement du texte, mais une liste d'objets où chacun d'eux contient une panoplie d'information tels le nom et l'état du service qu'il représente. De cette façon, on peut rapidement avoir accès à l'état des services de tout un parc de serveurs Windows.

Il y a différentes requêtes possibles sur toutes les composantes d'un serveur Windows, toujours en respectant la syntaxe d'une requête SQL. Un autre exemple serait *SELECT * FROM Win32_LogicalDisk WHERE DriveType=3* qui permet de retourner la liste de toutes les partitions logiques d'un serveur en spécifiant *DriveType=3* qui représente uniquement les disques durs et exclut par conséquent les partitions venant de clés USB ou de DVD-ROM.

2.2.2 Netstat

Netstat [34] est un outil de ligne de commande qui affiche les connexions TCP actives. Il permet de lister autant les connexions entrantes que sortantes ainsi que les ports TCP qui sont en écoute sur le serveur. Il permet également de trouver l'identifiant d'un processus en cours (PID) qui utilise une connexion réseau donnée. On peut voir à la figure 2.1 un exemple de Netstat sur Windows. Cette commande est disponible autant sur Windows que dans les distributions Linux.

2.2.3 Ping

Ping [35] est un outil qui permet de tester si une adresse IP est accessible ou non dans un réseau TCP/IP. Il tient son nom des sonars sous-marins, lorsqu'on envoie

```
PS C:\> netstat -ano

Active Connections

  Proto  Local Address          Foreign Address        State           PID
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       768
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
  TCP    0.0.0.0:623            0.0.0.0:0              LISTENING       7812
  TCP    0.0.0.0:3389           0.0.0.0:0              LISTENING       1248
  TCP    0.0.0.0:8782           0.0.0.0:0              LISTENING       3508
  TCP    10.12.13.75:139        0.0.0.0:0              LISTENING       4
  TCP    10.12.13.75:5040       0.0.0.0:0              LISTENING       6320
  TCP    10.190.64.250:5040     0.0.0.0:0              LISTENING       6320
  TCP    10.190.64.250:49707    13.89.187.212:443      ESTABLISHED     3860
  TCP    10.190.64.250:52455    40.101.128.18:443      ESTABLISHED     5600
  TCP    10.190.64.250:52467    40.100.162.194:443     ESTABLISHED     5600
```

Figure 2.1 La commande Netstat

une pulsation auditive sous l'eau pour ensuite écouter les échos reçus. Il fonctionne en envoyant sur un réseau un paquet ICMP (*echo-request*) et attend ensuite la réponse qui, elle aussi, est un paquet ICMP (*echo-reply*). Avec plusieurs requêtes et réponses de suite, la commande Ping calculera également quelques statistiques, comme le temps aller-retour, qui peuvent être intéressantes. La figure 2.2 est un exemple de la commande Ping sous Windows, mais tout comme Netstat elle est disponible autant sous Windows que Linux.

2.2.4 Tasklist

Tasklist [36] est un outil qui permet de lister les processus en cours d'exécution. Il est spécifique à Windows, mais il est similaire à la commande ps sous Linux. En plus des processus en cours d'exécution, Tasklist permet aussi de lister pour chaque processus son PID et les services reliés à celui-ci, s'il y a lieu. À la figure 2.3, on voit un exemple de la commande Tasklist.

```
PS C:\> ping google.ca

Pinging google.ca [172.217.1.163] with 32 bytes of data:
Reply from 172.217.1.163: bytes=32 time=22ms TTL=53
Reply from 172.217.1.163: bytes=32 time=22ms TTL=53
Reply from 172.217.1.163: bytes=32 time=22ms TTL=53
Reply from 172.217.1.163: bytes=32 time=22ms TTL=53

Ping statistics for 172.217.1.163:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 22ms, Maximum = 22ms, Average = 22ms
PS C:\>
```

Figure 2.2 La commande Ping

2.2.5 WinRM et WinRS

WinRM (Windows Remote Management) et WinRS (Windows Remote Shell) [37] sont utilisés afin d'ouvrir des sessions distantes sur des serveurs Windows. WinRM est le serveur qui reçoit les connexions de WinRS qui, lui, est le client. WinRS permet d'exécuter des commandes à distance sur un serveur Windows de la même façon que les sessions SSH sous Linux. Les sessions distantes ouvertes prennent la forme d'une console texte similaire à une console DOS ou bash.

2.2.6 Conclusion

Tous les outils présentés plus haut sont utilisés dans ce projet dans le but de faire la collecte d'information sur les serveurs. Ils sont tous manipulés pour rechercher de l'information bien précise. Nous verrons comment et à quel moment nous y aurons recours dans le chapitre 3.

```
PS C:\> tasklist /svc

Image Name                     PID Services
========================= ======== ============================================
System Idle Process              0 N/A
System                           4 N/A
smss.exe                       472 N/A
csrss.exe                      732 N/A
wininit.exe                    844 N/A
csrss.exe                      852 N/A
services.exe                   920 N/A
lsass.exe                      940 EFS, KeyIso, Netlogon, SamSs, VaultSvc
svchost.exe                    300 PlugPlay
svchost.exe                    436 BrokerInfrastructure, DcomLaunch, Power,
                                   SystemEventsBroker
WUDFHost.exe                   552 N/A
fontdrvhost.exe                648 N/A
svchost.exe                    768 RpcEptMapper, RpcSs
svchost.exe                   1008 LSM
winlogon.exe                  1084 N/A
fontdrvhost.exe               1172 N/A
svchost.exe                   1248 TermService
svchost.exe                   1264 nsi
svchost.exe                   1272 lmhosts
svchost.exe                   1280 W32Time
svchost.exe                   1428 Dnscache
```

Figure 2.3 La commande Tasklist qui affiche les PID et les services

2.3 Théorie des graphes

Le but de ce travail est de pouvoir modéliser un bon nombre de serveurs, les services qui les composent et leurs dépendances. Comme nous l'avons mentionné plus haut, la structure mathématique appropriée pour cette modélisation est le graphe. Dans cette section, nous aborderons les notions sur la théorie des graphes nécessaires à la compréhension des chapitres suivants.

2.3.1 Notion de base

Il y a plusieurs types de graphes. Dans notre cas, les nœuds représenteront des serveurs ou des services et les arêtes représenteront les dépendances entre les serveurs ou les services. La relation de dépendance entre deux nœuds est toujours que l'un dépend de l'autre dans une relation asymétrique. Donc, pour modéliser cette

propriété, nous utiliserons un graphe orienté $G = (V, A)$, où V est un ensemble fini de *nœuds* et A est un ensemble de *couples ordonnés appelés arcs.*

2.3.2 Graphes avec nœuds à nom unique

En plus d'utiliser des graphes orientés, nous utiliserons aussi une classe spéciale de graphe, soit les graphes avec noms uniques liés aux nœuds. Cette classe de graphes est bien utile afin de représenter les services et les serveurs d'un réseau qui sont des instances uniques. Bien entendu, chaque nœuds d'un graphe est toujours unique, cependant lorsque l'on veut comparer un graphe avec un autre, il nous faut une façon d'identifier les nœuds qui représentent la même chose. C'est ici que les noms des nœuds deviennent utiles.

Les graphes à noms uniques ont aussi pour effet de simplifier plusieurs problèmes algorithmiques. Par exemple, déterminer si deux graphes sont isomorphes pourrait être intractable [38] en général alors que vérifier si deux graphes avec nœuds à noms uniques sont isomorphes a une complexité algorithmique de $O(n^2)$ [38].

2.3.3 Parcours de graphes

En représentant les serveurs, les services et leurs dépendances sous forme de graphe, on peut savoir quels services dépendent de quels autres sur une longue chaîne de dépendances. On peut aussi utiliser les algorithmes existants afin de parcourir le graphe. Si une anomalie est trouvée sur un service donné qui est formalisé sous forme de nœud dans un graphe, un parcours de graphe permet d'identifier les services qui peuvent être affectés par cette anomalie. Un parcours en largueur est plus approprié qu'un parcours en longueur. La raison est qu'il détermine tous

les services affectés par l'anomalie en commençant par ceux qui sont adjacents.

2.3.4 Distance d'édition

Dans cette recherche, nous modéliserons l'évolution des services d'un réseau au cours du temps. Nous nous retrouverons donc avec une suite de graphes. Plusieurs algorithmes existent afin de comparer des graphes entre eux [19]. Un de ces algorithmes est le calcul de la distance entre deux graphes basé sur la topologie et, plus spécifiquement, la distance d'édition. Le principe est simple : plus le premier graphe est dissimilaire au deuxième, plus la distance d'édition est grande. En observant la distance d'édition de plusieurs graphes représentant un réseau informatique, on peut voir une tendance sur l'évolution du réseau dans le temps.

La distance d'édition (Graph Edit Distance ou GED) de deux graphes calcule la séquence d'opérations nécessaires pour modifier le premier graphe afin d'obtenir un graphe isomorphe au second. Le nombre d'éditions nécessaire ainsi que le type d'édition sont calculés, tels que l'ajout et la suppression de nœud ou d'arête. La substitution de nom peut aussi être considérée, ainsi que toutes les propriétés qu'on pourrait définir sur les nœuds ou les arêtes. De façon générale, les algorithmes de calcul de la distance d'édition incluent un coût pour chaque type d'édition possible et utilisent un algorithme de recherche afin d'identifier la séquence d'édition ayant le moindre coût [19]. Le coût total d'édition est la distance d'édition des deux graphes d'entrées de l'algorithme, c'est-à-dire la somme des coûts des opérations nécessaires pour transformer le premier graphe en l'autre. En général, il n'y a pas de séquence d'édition unique pour transformer un graphe en un autre, mais il y aura toujours un coût minimal que l'algorithme va trouver.

La méthode de calcul de la distance d'édition de graphes est définie dans [19], mais elle a été plus formellement expliquée à l'origine dans [39], qui est basé sur le calcul de la distance d'édition de chaîne de caractères dans [40]. Ce dernier calcul étant à mon avis plus facile à comprendre, nous l'expliquerons au prochain point avant d'expliquer le calcul de la distance d'édition entre graphes.

2.3.4.1 Calcul de la distance d'édition de chaîne de caractères

On considère deux chaînes de caractères a et b et on veut la mesure de $distance(a, b)$, c'est-à-dire le coût en opération d'édition pour transformer la chaîne a en chaîne b. La première étape est de définir l'ensemble des opérations d'édition possibles sur les chaînes. Généralement, les opérations sur une chaîne de caractères sont l'insertion, la suppression et la substitution d'un symbole à l'intérieur de la chaîne de caractères. Plus formellement, on peut définir les opérations comme suit :

a) $s_1 \rightarrow \epsilon$: la suppression d'un symbole s_1.
b) $\epsilon \rightarrow s_1$: l'insertion d'un symbole s_1.
c) $s_1 \rightarrow s_2$: la substitution d'un symbole s_1 par s_2.

Un coût est assigné à chaque opération d'édition et il est généralement dépendant de l'application. Dans tous les cas, le coût sera un nombre positif et la substitution d'un symbole par le même symbole aura un coût de 0. Le but est de trouver une séquence d'opérations d'édition qui transformera a en b. Il existe toujours plusieurs solutions. Par exemple, on peut supprimer tous les symboles de a et ensuite ajouter tous les symboles de b. L'objectif est néanmoins de trouver la séquence d'opérations d'édition qui aura le coût minimum.

Formellement, on définit $\sigma = e_1, e_2, ..., e_n$ comme étant une séquence d'opération d'édition où chaque opération a un coût $cost(e_i)$ et on définit

$$distance(a, b) = \sum_{i=1}^{n} min(cost(e_i)) \tag{2.1}$$

Donc, la distance d'édition est la somme des coûts individuels de chaque opération d'édition nécessaire pour transformer a en b. L'algorithme usuel pour calculer la $distance(a, b)$ est basé sur la programmation dynamique. L'algorithme utilise un tableau à plusieurs dimensions. Pour l'exemple d'une chaîne de caractères, on utilise un tableau à deux dimensions $C(n, m)$ de dimensions $(|a|+1) \times (|b|+1)$ pour calculer les coûts en opération d'édition. Par conséquent, il y a une rangée pour chaque symbole de a plus une rangée supplémentaire et une colonne pour chaque symbole de b plus une colonne supplémentaire. La première cellule $C(0, 0)$ est initialisée à 0. Les cellules de la première rangée sont initialisées de la façon suivante $C(i, 0) = C(i-1, 0) + cost(a_i \rightarrow \epsilon)$ et représentent le coût de suppression de tous les symboles de a jusqu'à a_i. Les cellules de la première colonne sont initialisées avec $C(0, j) = C(0, j-1) + cost(\epsilon \rightarrow b_j)$ qui similairement représentent l'ajout des symboles de b. De plus, l'algorithme poursuit pour chaque cellule $C(i, j)$ du tableau en assignant la valeur minimum des trois termes suivants à $C(i, j)$:

a) $C(i-1, j-1) + \text{cost}(a_i \rightarrow b_j)$
b) $C(i-1, j) + \text{cost}(a_i \rightarrow \epsilon)$
c) $C(i, j-1) + \text{cost}(\epsilon \rightarrow b_j)$

La dernière cellule $C(n, m)$ contiendra la valeur du coût minimum. L'article suivant [40] contient la preuve que cet algorithme est correct. Pour avoir la séquence d'opération d'édition pour transformer a en b, on parcourt le chemin inverse dans

le tableau en partant de la dernière cellule $C(n, m)$. Cela permet de construire le chemin optimal dans les opérations possibles pour transformer a en b. Il faut noter qu'il peut y avoir plusieurs chemins possibles, ce qui veut dire qu'il y a plusieurs coûts possibles. C'est le pourquoi de la fonction minimum à l'équation 2.1, car c'est un chemin avec un coût minimum qui nous intéresse qui représente un chemin optimal.

2.3.4.2 Calcul de la distance d'édition de graphes

Le calcul de la distance d'édition de deux graphes se fait en utilisant les mêmes principes qu'avec les chaînes de caractères. La première étape consiste toujours à lister toutes les opérations d'édition possibles sur les nœuds qui sont l'ajout, la suppression et la substitution de nœud. Ensuite, on liste toutes les opérations possibles sur les arêtes entre les nœuds qui sont l'ajout, la suppression et la substitution d'arête. Finalement, on détermine le coût d'édition de chaque opération. Du point de vue algorithmique, c'est exactement la même chose que la distance d'édition de chaîne de caractère vue à la section 2.3.4.1.

Prenons l'exemple de la figure 2.4 où l'on peut voir les opérations d'éditions nécessaires pour transformer le graphe A en graphe B. On commence par éliminer le nœud e. On continue en substituant le nœud d par le nœud f. Ensuite, on ajoute l'arête entre les nœuds a et c, et on termine en éliminant l'arête entre les nœuds b et f. Pour cette exemple, chaque opération d'édition a le même coût et la distance d'édition des deux graphes est de 4 puisque la transformation se fait en 4 étapes.

En utilisant un algorithme similaire à celui présenté à la section 2.3.4.1, on peut calculer la distance d'édition entre deux graphes. L'algorithme utilise une approche

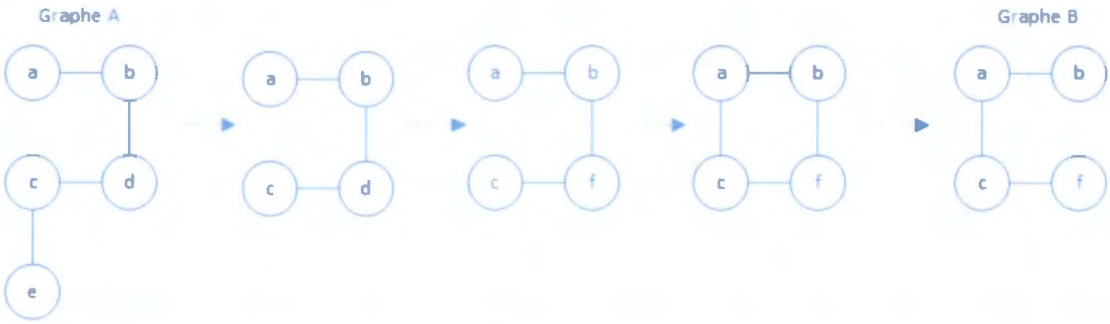

Figure 2.4 Exemple de transformation du graphe A en graphe B.

par programmation dynamique, mais pour le projet de recherche présenté dans cette maîtrise l'algorithme a été beaucoup modifié et utilise une approche naïve. La raison principale est que les graphes utilisés dans notre recherche sont des graphes orientés avec nœuds à noms uniques, et les nœuds ont aussi des attributs qui représentent des états. Par exemple, les nœuds qui représentent des services auront un attribut qui représente leur état d'exécution. De plus, certaines opérations d'édition ont quelques subtilités. La figure 2.5 fait un résumé des changements à l'algorithme de calcul d'édition des graphes.

2.3.5 Conclusion

On a vu que le calcul de la distance d'édition de graphes s'effectue d'une façon similaire au calcul de la distance d'édition de chaîne de caractères. L'algorithme a été modifié de l'algorithme vu en 2.3.4.2 et les changements sont résumés dans la figure 2.5. L'algorithme derrière ces changements sera vu au chapitre 4.

Changements	Explications
Ajout et suppression d'attributs	Ces opérations non pas lieu. Il n'y a pas de possibilité d'enlever un nom ou un attribut à un service ou à un serveur, ils ont toujours leurs attributs.
Substitution de dépendances	Cette opération n'a pas lieu. Il n'y a pas d'intérêt à changer une dépendance pour une autre entre deux mêmes services.
Ajout et suppression de dépendances	Il peut y avoir un coût différent entre une dépendance locale et une dépendance distante au niveau de l'édition. La raison est qu'il peut y avoir des problèmes différents à traiter lorsqu'on a affaire à un problème local ou distant. Il est important de voir cette distinction dans le calcul de la distance d'édition. Nous aborderons ces différences au chapitre 4.
Substitution de nœuds	Étant donné que les nœuds ont tous obligatoirement un nom unique, cette opération est fusionnée avec la substitution de nom (sans inclure de coût additionnel), car les noms et les noeuds sont indissociables.

Figure 2.5 Changements au calcul d'édition des graphes

CHAPITRE III

CONSTRUCTION DU MODÈLE

Dans ce chapitre, nous détaillerons comment le système fait la collecte d'information sur le réseau et ferons ensuite la construction du modèle utilisé dans le but de diagnostiquer les possibles problèmes réseau.

3.1 Les données d'entrées

Les données que le système doit minimalement avoir sont les plages IP que le système utilisera pour sa collecte d'information. Avec cette information, le système découvrira le réseau par lui-même. Par exemple, la plage d'adresses IP utilisée pour nos scénarios a été 10.190.64.0/24 qui est celle utilisée par les serveurs de Rheinmetall Canada Inc. qui ont été utilisés pour nos tests. Le reste de l'information, le système le découvrira automatiquement.

3.2 Les étapes de la construction du modèle

La construction du modèle se fait en plusieurs étapes qui doivent successivement se suivre. On lance des commandes à l'interface du système, qui lance ensuite en parallèle une série d'opérations réseau dans le but de découvrir le parc de serveurs et de construire un graphe de dépendance des services qu'ils servent.

3.2.1 Étape 1 - La détection des nœuds réseau

La première étape est le scan du réseau et la découverte des nœuds IP qui s'y trouvent. En utilisant les plages IP déjà connues, le système lance une série de Ping (la requête ICMP *echo-request*) sur toutes les adresses IP contenues dans les plages IP. Le système garde en liste l'adresse IP de tous les nœuds réseau qui répondent au Ping (la réponse ICMP *echo-reply*).

3.2.2 Étape 2 - La détermination des types de nœuds réseau

Comme cette recherche se limite aux serveurs Windows, le but est de filtrer les nœuds IP qui sont des serveurs Windows et de laisser tomber les autres nœuds IP des serveurs Linux, des équipements réseau, des imprimantes, etc. Donc, en partant de la liste d'adresses IP qui ont répondu à l'étape précédente, on lance une requête WMI à chacun de ces nœuds. Seuls les nœuds réseau qui sont des serveurs Windows répondront à ces requêtes puisque WMI est propriétaire à Microsoft et spécifiquement fait pour Windows. Les requêtes retournent les informations de base telles que la version de Windows et le nom du serveur. À la fin de cette étape, nous avons une liste de noms de serveurs et leurs adresses IP identifiées comme étant des serveurs Windows. Les adresses IP dont les requêtes n'auront rien retourné seront écartées de la liste. À cette étape, on fusionne aussi les entrées qui sont redondantes. Par exemple, un serveur peut avoir deux adresses IP et sera par conséquent interrogé deux fois. Comme les deux interrogations retourneront les mêmes noms, le système détectera qu'il s'agit du même serveur avec deux adresses IP et fusionnera les entrées. La requête WMI suivante est utilisée pour identifier les serveurs Windows : *SELECT * FROM Win32_OperatingSystem*. Elle retourne l'information du système d'exploitation et le nom du serveur, la version et le nom du système d'exploitation seront retenus pour le modèle.

3.2.3 Étape 3 - La détection des composantes des serveurs Windows

La détection des composantes des serveurs Windows se fait par plusieurs autres requêtes WMI séparées. Ces requêtes sont envoyées à tous ceux qui ont été identifiés comme étant des serveurs Windows. La requête *SELECT * FROM Win32_Service* demande la liste de toutes les informations concernant les services Windows tels que l'identifiant du service, son nom, sa description et son statut. Le statut est particulièrement important, car il s'agit de l'état du service. Si le service est en exécution ou non, c'est ici que l'information sera fournie.

3.2.4 Étape 4 - La détection des dépendances internes aux serveurs Windows

La détection des dépendances des services internes aux serveurs se fait par une autre requête WMI. Pour chaque service de chaque serveur Windows, une requête est faite afin de lister les dépendances locales de ce service. La réponse est une liste contenant tous les services Windows dont ce service dépend. Un lien de dépendance parent-enfant (vu au point 2.1.3) entre le service en question et tous les autres services listés dans la réponse est ajouté dans le modèle. La requête suivante est utilisée pour chaque service d'un serveur : *Associators Of Win32_Service.Name = 'NOM_DU_SERVICE' Where AssocClass = Win32_DependentService ResultClass = Win32_Service Role = Dependent.*

3.2.5 Étape 5 - La détection des dépendances externes aux serveurs Windows

La détection des dépendances entre les services de différents serveurs Windows se fait en plusieurs étapes.

3.2.5.1 Étape 5.1 - Exécution de Netstat et Tasklist

Le système ouvre une session à distance sur chacun des serveurs en utilisant WinRM, et sur chacun, il exécute les commandes « netstat –ano » et « tasklist /svc » et capture les résultats. Le résultat de Netstat retourne la liste de toutes les connexions actives ainsi que les ports en écoute. Pour chaque entrée de Netstat, le PID associé est indiqué. Tasklist, quant à lui, retourne une liste de toutes les tâches en cours d'exécution avec les services associés et leur PID respectif.

```
Active Connections

Proto  Local Address          Foreign Address        State           PID
TCP    0.0.0.0:135            0.0.0.0:0              LISTENING       864
TCP    0.0.0.0:445            0.0.0.0:0              LISTENING       4
TCP    0.0.0.0:1433           0.0.0.0:0              LISTENING       2788
TCP    0.0.0.0:3389           0.0.0.0:0              LISTENING       216
TCP    10.190.64.21:445       10.190.64.250:52457    ESTABLISHED     4
TCP    10.190.64.21:1433      10.190.64.34:49203     ESTABLISHED     2788
TCP    10.190.64.21:1433      10.190.64.34:49215     ESTABLISHED     2788
TCP    10.190.64.21:1433      10.190.64.34:49326     ESTABLISHED     2788
TCP    10.190.64.21:1433      10.190.64.34:49338     ESTABLISHED     2788
TCP    10.190.64.21:1433      10.190.64.34:49440     ESTABLISHED     2788
TCP    10.190.64.21:58801     65.52.108.200:443      ESTABLISHED     2788
TCP    10.190.64.21:6244      10.190.64.200:45291    TIME\_WAIT       0
...
```

Figure 3.1 Exemple de la commande netstat –ano

3.2.5.2 Étape 5.2 - Filtre des informations

Les entrées de Netstat et Tasklist sont filtrées. Pour Netstat, les entrées en TIME_WAIT sont gardées par l'OS dans le but d'accélérer la reconnexion, s'il y a lieu. Cependant, elles ne sont associées à aucun PID, et donc inutiles pour le travail. C'est pourquoi elles sont retirées. Ensuite, les entrées dont les adresses IP ne sont pas dans les adresses IP des serveurs Windows qui nous concernent sont inutiles aussi et, donc, elles sont retirées également. Pour l'instant, comme le réseau fonctionne

```
Image Name                     PID Services
========================= ======== ============================================
System Idle Process              0 N/A
System                           4 N/A
svchost.exe                    384 CDPSvc, EventSystem, FontCache, netprofm,
nsi, RemoteRegistry, W32Time,
WinHttpAutoProxySvc
svchost.exe                    372 Dhcp, EventLog, lmhosts, TimeBrokerSvc
spoolsv.exe                   1764 Spooler
sqlceip.exe                   2744 SQLTELEMETRY
sqlservr.exe                  2788 MSSQLSERVER
WmiPrvSE.exe                  3076 N/A
macompatsvc.exe               3328 McAfeeFramework
rdpclip.exe                  10588 N/A
RuntimeBroker.exe             6372 N/A
tasklist.exe                 10664 N/A
...
```

Figure 3.2 Exemple de la commande tasklist /svc

en IPv4, toutes les adresses IPv6 sont retirées. Toutes les entrées en LISTENING et celles où l'IP est 127.0.0.1 vers lui-même sont conservées. Les entrées de Tasklist non associées à un service ne sont pas intéressantes. Elles sont donc retirées. À la fin de cette partie, les entrées de Netstat sont associées aux entrées de Tasklist au moyen du PID pour former des entrées réseau. Avec ces entrées réseau, on peut dire quels services ont ouvert quelles connexions réseau et quels services sont en écoute sur le réseau.

La figure 3.3 montre six entrées Netstat liées à l'entrée Tasklist (au bas de la même figure) puisqu'elles ont le même PID. Elles forment donc six entrées réseau du processus MSSQLSERVER.

3.2.5.3 Étape 5.3 - Séparation des types d'entrées réseau

Ensuite, il faut séparer les entrées réseau selon leur type et les identifier. Il y a trois types à considérer. Le premier type concerne les services qui sont en écoute. Ils sont identifiables par l'état LISTENING des Netstat-Tasklist. Le port local de ces en-

```
Netstat :
 TCP   0.0.0.0:1433          0.0.0.0:0              LISTENING   2788
 TCP   10.190.64.21:1433     10.190.64.34:49203     ESTABLISHED 2788
 TCP   10.190.64.21:1433     10.190.64.34:49215     ESTABLISHED 2788
 TCP   10.190.64.21:1433     10.190.64.34:49326     ESTABLISHED 2788
 TCP   10.190.64.21:1433     10.190.64.34:49338     ESTABLISHED 2788
 TCP   10.190.64.21:58801    65.52.108.200:443      ESTABLISHED 2788

Tasklist :
 sqlservr.exe  2788  MSSQLSERVER
```

Figure 3.3 Les entrées réseau du service MSSQLSERVER

trées est le port utilisé pour l'écoute sur le réseau. Le deuxième type concerne les connexions entrantes. Elles sont identifiables parce que leurs ports locaux sont aussi dans celles ayant l'état LISTENING. Étant donné que les connexions TCP n'utilisent jamais les ports en écoute comme ports locaux pour initier une connexion sortante, toutes les entrées ayant les mêmes ports locaux que ceux en écoute sont obligatoirement des connexions entrantes. Toutes les autres entrées sont forcément des connexions sortantes, qui représentent le troisième type.

Si on reprend la figure 3.3, on voit que dans les six entrées réseau la première représente le service qui est en écoute sur le port tcp/1433. Les quatre entrées suivantes représentent des connexions entrantes puisque le port local est le même que celui en écoute. La dernière entrée est une connexion sortante.

3.2.5.4 Étape 5.4 - Intégrer la liste des ports en écoute dans le modèle

Maintenant que toutes les entrées Netstat sont liées aux entrées Tasklist et que les types d'entrées sont bien identifiés, et ce, pour tous les serveurs Windows, on peut commencer à intégrer au modèle les informations requises. À cette étape, on associe les ports en écoute au bon service dans le modèle. Cela est faisable parce que les entrées réseau ont la liste des services associés aux ports d'écoute (figure

3.3). Ainsi, à la fin de cette étape, chaque service dans le modèle qui écoute sur le réseau contient l'information du port sur lequel il écoute.

3.2.5.5 Étape 5.5 - Intégrer les interdépendances de services

Pour chaque connexion sortante, on doit associer le service qui fait la connexion au service du serveur distant. Cette association se fait en deux étapes. La première étape est de trouver le service local qui a lancé la connexion distante. On retrace les services locaux dans les entrées réseau, car les services qui ont lancé la connexion y sont listés. La deuxième étape est de retrouver les bons serveurs distants en utilisant l'adresse IP du serveur distant dans les entrées réseau. Ensuite, sur ce serveur, on retrace quel service écoute sur le port destination qui a été utilisé dans l'entrée réseau. Avec ces deux informations, on peut donc faire une dépendance entre les deux services distants. Celui qui initie la connexion est dépendant de celui qui la reçoit.

La figure 3.4 montre un exemple d'une dépendance distante. Avec l'information du serveur X (de Netstat et Tasklist) qui représente une connexion sortante, on peut retracer le serveur Y avec l'adresse IP et le bon service avec le port destination. Ainsi, on peut conclure que le service MCAFEETOMCATSRV530 sur le serveur X est dépendant du service MSSQLSERVER sur le serveur Y.

```
Serveur X
TCP  10.190.64.34:49215   10.190.64.21:1433  ESTABLISHED  1504
tomcat7.exe  1504  MCAFEETOMCATSRV530

Serveur Y
TCP  10.190.64.21:1433   10.190.64.34:49215  ESTABLISHED  2788
sqlservr.exe  2788  MSSQLSERVER
```

Figure 3.4 Les entrées réseau du service MSSQLSERVER

3.2.5.6 Étape 6.0 - Le graphe de dépendance

Au final, on obtient un graphe contenant tous les services de tous les serveurs ainsi que les dépendances entre ces services. La construction se fait en plusieurs étapes. D'abord, on commence par créer un nœud dans le graphe pour chaque serveur Windows. Ensuite, pour chaque serveur, un autre nœud est créé pour chaque service local à ce serveur. Un lien de dépendance est créé entre ces services, de sorte qu'un service est toujours dépendant du serveur sur lequel il est installé. À cette étape, nous avons maintenant une série de graphes simples où chaque service est dépendant de leur serveur. Enfin, pour chaque service de chaque serveur, on crée une dépendance pour chaque service dont il dépend. Ces nouvelles dépendances seront autant des services internes au serveur que des services sur d'autres serveurs si le service en question avait une interaction avec le réseau. À cette étape, la construction du graphe est terminée.

Il faut noter que les nœuds du graphe sont tous nommés. Les nœuds serveur seront nommés avec le nom du serveur (sans le nom de domaine). Par exemple, le serveur *serveur.domaine.ca* sera nommé *serveur*. Chaque nœud de service sera nommé selon le nom du serveur qui héberge le service en question avec le nom du service. Par exemple, le service *service1* sur le serveur *serveur.domaine.ca* sera nommé *service1-serveur*. Cela a pour but d'avoir des noms uniques, car certains services peuvent avoir le même nom sur plusieurs serveurs.

Les nœuds sont tous structurés de la même manière. Sur chaque nœud, on peut accéder à ses attributs comme son identifiant, son nom et son type (Serveur, Service, etc.). Aussi, pour chaque nœud, on peut accéder à deux séries de dépendances : les nœuds dont il dépend lui-même et ceux qui dépendent de lui.

Ces dépendances représentent les nœuds qui doivent bien fonctionner pour qu'un nœud donné fonctionne bien, mais aussi les nœuds qui vont mal fonctionner si un nœud donné fonctionne mal. Il faut noter qu'un nœud Service est toujours dépendant d'un nœud Serveur. Si le serveur ne répond plus, tous les services dépendants du Serveur ne seront plus fonctionnels. La même chose sera vraie pour les services distants qui dépendent des services sur ce serveur.

3.3 L'implémentation

Le système a été construit pour être une interface de lignes de commande où on peut lancer la découverte du réseau et construire le modèle pour l'ensuite interroger avec différentes commandes. Le système a été développé en C# et en .Net 4.0 avec Visual Studio 2010. La raison d'utiliser C# plutôt qu'un autre langage est que C# contient tous les objets et API nécessaires pour interfacer avec tous les produits Microsoft. Comme la recherche a été concentrée sur la plateforme Windows uniquement, ce choix avait du sens.

3.4 Conclusion

Dans ce chapitre, on a vu que le modèle se crée en plusieurs étapes. Le système fait la découverte du réseau de tous les nœuds IP avec l'aide d'un Ping pour ensuite filtrer uniquement les serveurs Windows avec une requête WMI. Ensuite, il détecte les composantes et services de chaque serveur avec l'aide d'autres requêtes WMI. Il ajoute au modèle, avec d'autres requêtes WMI, les dépendances entre les services qui sont internes aux serveurs. Il découvre les interdépendances entre les services distants. La détection des interdépendances se fait avec l'aide de Netstat et Tasklist lancés au moyen de WinRM sur chaque serveur. Les captures de Netstat et Tasklist sont par la suite filtrées et liées ensemble par le PID, qui est un attribut

commun aux captures de Netstat et Tasklist. Enfin, avec l'information des adresses IP et des ports locaux et distants, on est capable de déterminer quel service local à un serveur utilise quel port d'écoute et quel service d'un serveur se connectera sur quel service d'un serveur distant. Toute cette information est intégrée au modèle. Un graphe de dépendance est ensuite créé avec toute cette information.

CHAPITRE IV

RECHERCHE D'ANOMALIE

Dans ce chapitre, nous présenterons une méthode de recherche d'anomalies à partir d'une série de plusieurs captures de l'état du réseau.

L'idée générale est qu'on fait plusieurs captures périodiques du réseau. Ces captures serviront à bâtir les modèles internes du réseau, qui eux serviront à faire la détection d'anomalies. Les modèles internes seront deux représentations du réseau : une représentation dans un état normal et une représentation dans un état à diagnostiquer. La détection d'anomalies se fera en comparant les deux représentations.

Les captures périodiques du réseau sont représentées sous forme de graphes, comme nous l'avons vu au chapitre précédent, et elles représentent les données d'entrée du système. Ces captures sont séparées en deux séries. La première série de graphes représentera le réseau dans un état normal et l'autre représentera le réseau dans un état à diagnostiquer. Les graphes de la première série sont fusionnés ensemble dans le but d'obtenir une représentation plus fidèle du réseau dans un état normal. La fusion peut être considérée comme l'union des graphes avec quelques différences, comme nous le verrons dans ce chapitre. Les graphes de la deuxième série sont, pour leur part, fusionnés ensemble dans le but d'avoir une

représentation du réseau dans un état à diagnostiquer. Les fusions des séries de graphes nous donnent deux graphes qui sont les représentations du réseau dans un état normal et dans un état à diagnostiquer.

Ensuite, l'algorithme de recherche d'anomalie est lancé et compare les deux graphes. La recherche se fait en calculant la différence entre les deux graphes, mais l'algorithme est fait de façon à pouvoir avoir un coût différent pour chaque type d'opération d'édition. Les opérations d'édition représentent des changements possibles dans le réseau. Par exemple, un serveur disparu, un nouveau serveur, un service disparu, un nouveau service, des dépendances ajoutées ou disparues, un service en exécution qui ne l'est plus, etc. Tous ces cas de figures sont représentés par différentes opérations d'édition dans le calcul de distance d'édition.

Bien que l'algorithme dans son ensemble soit inspiré de la distance d'édition, on ne s'intéresse pas à la séquence d'opérations d'édition pour transformer un graphe en un autre. On s'intéresse plutôt aux coûts associés aux opérations d'édition. Comme ces opérations représentent des changements dans le réseau, avoir des coûts différents pour chaque opération et ajuster ces coûts selon les scénarios permet de détecter certains types de problèmes réseau bien spécifiques.

Cette façon de faire permet donc d'avoir un algorithme générique en utilisant plusieurs coûts différents, d'être flexible et de cibler des problèmes spécifiques. Nous verrons dans les sections suivantes chaque étape de l'algorithme plus en détail.

4.1 Les captures du réseau

Les captures du réseau constituent les données d'entrée du système. Chaque capture est ensuite modélisée sous forme de graphe, tel que nous l'avons vu au chapitre 3. Cela constitue une représentation du réseau en un temps précis. Les captures peuvent cependant être incomplètes, car bien que l'utilisation de Netstat [34] et de Tasklist [36] lors de la capture nous dit quels services utilisent quelles connexions TCP en cours, il se peut toujours que certaines dépendances ne soient pas en cours d'utilisation durant la capture. Afin d'avoir une représentation plus complète du réseau qui évolue dans le temps, plusieurs captures sont faites après un intervalle de temps choisi arbitrairement.

4.1.1 Le réseau dans un état normal

Le réseau dans un *état normal* veut simplement dire que le réseau ne présente aucune anomalie et que les services sur les serveurs fonctionnent normalement. Les premières captures du réseau sont faites afin de représenter le réseau dans un état normal. Pour avoir de bonnes représentations du réseau, plusieurs captures sont prises de façon chronologique et représentent l'évolution du réseau dans le temps. L'intervalle de temps entre les captures et le nombre de captures peut être arbitraire. Dans cette recherche, nous avons décidé de tester la solution avec dix captures à intervalle d'environ une heure afin de représenter le réseau. Ce choix a été fait en se basant sur une expérimentation que nous verrons au point 5.2.4.

4.1.2 Le réseau dans un état à diagnostiquer

Le réseau dans un *état à diagnostiquer* veut dire que l'état du réseau n'est pas connu. On ne sait pas s'il y a une ou des anomalies dans le réseau. Les dernières

captures du réseau représentent le réseau dans un état indéterminé et qu'on veut diagnostiquer. Dans cette recherche, trois captures ont été faites afin de représenter le réseau qu'on veut diagnostiquer. Le choix du nombre de captures représentant l'état du réseau à diagnostiquer a été fait en expérimentant sur le nombre de captures.

4.1.3 Explication du choix du nombre de captures

Le choix d'avoir dix captures pour la représentation de l'état normal du réseau et trois pour la représentation du réseau à diagnostiquer a été basé sur l'expérimentation élaborée au point 5.2.4. L'intuition derrière ce choix est de pouvoir éventuellement avoir un système qui capture l'état du réseau de façon chronologique à temps régulier en utilisant toujours les dernières captures pour le diagnostic et de pouvoir rapidement détecter les anomalies. Si le réseau n'a pas d'anomalie, la première capture du réseau normal est éliminée, la première capture du réseau à diagnostiqué est envoyée dans le réseau normal et la capture suivante sera la dernière de la série diagnostic. Les captures seraient maintenues de cette façon telle une file. C'est pourquoi le système a été fait pour contenir un certain nombre de captures du réseau pour la représentation du réseau à l'état normal et moins de captures pour le diagnostic. En principe, plus il y a de captures et meilleure serait la représentation, et s'il n'y en a pas assez, il pourrait y avoir plus de faux négatifs lors d'un diagnostic. Cela pourrait constituer une autre recherche en soi, mais pour la grosseur du réseau utilisé durant l'expérimentation les valeurs de dix et trois captures ont été suffisantes.

4.2 La fusion des graphes

On fusionne les graphes des deux séries respectives de façon à n'avoir qu'un seul graphe qui représente chaque série. Le processus de fusion utilise la même méthode pour les deux séries. La méthode, décrite à l'algorithme 1, est inspirée de ce qu'on peut voir dans [19] sur le recouvrement d'information manquante dans une série de graphes.

L'algorithme 1 fait la fusion des graphes. Il prend en entrée une série de graphes avec nœuds à noms uniques qu'on veut fusionner et commence par déclarer un graphe *fusion* qui servira de graphe fusionné en sortie. Le graphe *fusion* est lui aussi avec nœud à nom unique. Ensuite, on boucle sur tous les graphes de la série en entrée. Pour chaque graphe, on ajoute les nœuds du graphe dans *fusion* en filtrant les doublons. Si le nœud qu'on veut ajouter est déjà inclus dans *fusion*, on vérifie qu'il a le même état. Autrement dit, on vérifie si le nœud représente un service, ce service a le même état, c'est-à-dire qu'il en exécute ou pas. Dans le cas de nœud qui n'a pas le même état, nous sommes en présence d'un service qui a un état intermittent, et le nœud en question est marqué comme tel.

Ensuite, on boucle sur chaque dépendance de chaque nœud du graphe en cours de traitement. On ajoute à *fusion* chaque dépendance une à une en n'insérant pas de doublons. Après cette étape, *fusion* représente la fusion de tous les graphes.

4.2.1 Les nœuds avec des états intermittents

L'étape suivante est d'identifier les nœuds de *fusion* qui ont des états intermittents (voir le point 2.1.5.2) et de les filtrer ainsi que toutes leurs dépendances. Ils sont donc représentés par des nœuds de graphes ayant des états différents dans une

Algorithm 1: Fusion des graphes

Input: graphes : une séquence de graphes
Output: fusion : un graphe représentant les graphes fusionnés

```
let fusion : un graphe;
foreach graphe in graphes do
    foreach noeud in graphe do
        let noeudFus = fusion.contient(noeud) /*recherche s'il y a un nœud de
          même nom*/;
        if noeudFus != null and noeudFus.execution != noeud.execution then
            noeudFus.intermittent = true;
        end
        fusion.ajoutNoeud(noeud) /*n'ajoute pas de doublons*/;
    end
    foreach noeud in graphe do
        foreach noeud2 dans noeud.dependances do
            fusion.ajoutLien(noeud, noeud2) /*n'ajoute pas de doublons*/;
        end
    end
end
foreach lien in fusion.liens do
    if lien.enfant.intermittent or lien.parent.intermittent then
        fusion.enleveLien(lien);
    end
end
foreach noeud in fusion do
    if noeud.intermittent then
        fusion.enleveNoeud(noeud);
    end
end
```

même série. Si on reprend l'exemple vu au point 2.1.5.2, le service BITS [31] peut s'exécuter sporadiquement sur un serveur et avoir par conséquent toutes les chances d'avoir des états différents sur plusieurs captures du réseau. Donc, lorsqu'un même nœud a des exécutions différentes sur deux graphes différents, il est marqué comme étant intermittent. Ensuite, on enlève les nœuds du graphe *fusion* ainsi que toutes leurs dépendances parents et enfants. Cette étape est représentée par les deux dernières boucles dans l'algorithme 1. À la fin, on a un graphe fusionné sans les nœuds avec un état intermittent, car ils ne sont pas viables pour un diagnostic (voir le point 2.1.5.2).

4.3 L'algorithme de recherche d'anomalie

L'algorithme 2 représente les boucles principales de la recherche d'anomalie. Il prend en entrée les deux graphes représentant le réseau normal et celui à diagnostiquer et calcule ensuite la distance d'édition. Il commence par déclarer les variables qui lui serviront à faire le diagnostic. La variable *cout* sera utilisée afin de calculer la distance d'édition des deux graphes. Les deux listes *listNouveauServeur* et *listServeurDisparu* seront ensuite utilisées afin de garder une trace des nœuds qui représentent des serveurs qui ont disparu et ceux qui représentent de nouveaux serveurs. Les serveurs disparus sont ceux qui étaient dans le premier graphe, mais ne sont plus dans le deuxième, et les nouveaux serveurs sont ceux qui n'étaient pas dans le premier graphe, mais sont présents dans le deuxième. Ensuite, on boucle sur tous les nœuds des deux graphes. Les paires de nœuds ayant le même nom complet (c'est-à-dire le même serveur ou le même service installé sur un même serveur) sont traitées afin de calculer la distance d'édition avec les trois fonctions *calculerCoutServiceArrete* (algorithme 3), *calculeCoutConnexionsDistantes* (algorithme 5) et *verifierDependances* (algorithme 6). Si un nœud représentant un serveur est présent dans le graphe représentant le réseau normal et n'est pas

présent dans l'autre, c'est qu'il s'agit d'un serveur qui a disparu. On ajoute donc le serveur dans la liste appropriée. Lorsque les nœuds ont tous été comparés, on calcule la distance d'édition reliée aux serveurs disparus ou remplacés avec la fonction *calculeServeursDisparuOuRemplacer* (algorithme 4).

Algorithm 2: Recherche d'anomalie

Input: gNormal : graphe représentant le réseau normal
Input: gDiag : graphe représentant le réseau à diagnostiquer
Output: fusion : liste d'anomalies

```
soit cout = 0;
soit listeNouveauServeur = liste vide;
soit listeServeurDisparu = liste vide;
foreach noeudN dans gNormal do
    soit noeudNTrouver = false;
    foreach noeudD dans gDiag do
        if noeudN.nomComplet == noeudD.nomComplet then
            noeudNTrouver = true;
            cout += calculerCoutServiceArrete(noeudN, noeudD);
            cout += calculeCoutConnexionsDistantes(noeudN, noeudD);
            cout += calculerCoutDependances(noeudN, noeudD,
             listeNouveauServeur);
        end
    end
    if !noeudNTrouver and noeudN.type == SERVEUR then
        listeServeurDisparu.Add(noeudN);
    end
end
cout += calculeServeursDisparuOuRemplacer(listeServeurDisparu,
 listeNouveauServeur)
```

4.3.1 Calculer la distance d'édition d'un service arrêté

L'algorithme 3 représente une fonction qui retourne la distance d'édition de deux nœuds passés en paramètre. Bien que la fonction soit très simple, elle demande

une certaine explication pour bien comprendre son utilité. Cette fonction est appelée dans l'algorithme 2 après la confirmation que les deux nœuds comparés représentent la même chose dans les représentations du réseau. La fonction de l'algorithme 3 traitera donc deux nœuds qui représentent soit le même serveur, soit le même service.

S'il s'agit de serveurs, la fonction retourne un coût de 0, car on ne peut pas avoir de serveurs éteints dans les graphes représentant le réseau. Les serveurs éteints n'apparaissent pas dans les graphes puisqu'ils ne peuvent pas faire partie d'aucune capture réseau. Pour repérer les serveurs éteints, il faut utiliser une autre méthode que nous verrons plus loin.

S'il s'agit de services, la fonction retourne la distance d'édition d'un service éteint seulement si le service était en exécution dans le graphe du réseau normal, mais pas dans le graphe de diagnostic. Cette situation représente un service qui était démarré durant le temps où le réseau était fonctionnel et ce même service est éteint durant le temps où le réseau est dans un état incertain. La situation inverse retourne une distance de 0. Un service qui est éteint durant le fonctionnement normal du réseau et qui est en cours d'exécution durant le temps où le réseau est dans un état incertain peut notamment représenter un changement de configuration ou une installation d'un service quelconque, mais ne sera pas qualifié d'anomalie. Un service dans le même état dans les deux graphes représente une situation où il n'y a aucun changement, et donc la fonction retournera une distance de 0.

Algorithm 3: Calculer coût service éteint

```
Function calculeCoutService(noeudN, noeudD) : int
    if noeudN.type == SERVICE and noeudN.running and not noeudD.running
     then
        return COUT-SERVICE-ETEINT;
    end
    return 0;
```

4.3.2 Calculer la distance d'édition d'un serveur disparu ou remplacé

L'algorithme 4 est la fonction qui s'occupe de calculer le coût en termes de distance d'édition d'un serveur disparu ou remplacé. Un serveur disparu est un nœud de type serveur qui est dans le graphe du réseau normal qui ne sera plus présent dans le graphe diagnostic. Un serveur remplacé sera un serveur disparu en premier lieu et dont les services nécessaires à d'autres services qui étaient sur le serveur d'origine ont été transférés sur un autre serveur. On peut voir ce changement en remarquant que les dépendances d'un service X qui étaient sur un serveur Y sont maintenant sur un serveur Z.

L'algorithme 4 a pour entrée la liste des serveurs disparus et la liste des nouveaux serveurs (voir le point 4.3). La liste des nouveaux serveurs contient aussi pour chaque serveur l'ancien nom du serveur lorsqu'il en remplace un (nous verrons comment il trouve l'ancien nom au point 4.3.4). Alors l'algorithme fait le parcours de la liste des serveurs disparus et, pour chaque serveur disparu, il vérifie si son nom fait partie de ceux que les nouveaux serveurs remplacent.

4.3.3 Calculer le coût des connexions distantes

L'algorithme 5 est la fonction qui s'occupe de calculer le coût en termes de distance d'édition d'un service qui avait des connexions distantes entrantes dans le premier

Algorithm 4: Gestion des serveurs disparus ou remplacés

```
Function calculeServeursDisparuOuRemplacer(listeServeurDisparu,
 listeNouveauServeur) : int
    let cout = 0;
    foreach serveur in listeServeurDisparu do
        let remplacer = false;
        foreach item dans listeNouveauServeur do
            if serveur.nom == item.ancienNom then
                remplacer = true;
            end
        end
        if remplacer then
            cout += COUT-NODE-REPLACED;
        else
            cout += COUT-NODE-DOWN;
        end
    end
    return cout;
```

graphe et qui n'en a plus dans le second. Cela représente les cas où pour une raison quelconque le service ne répond plus sur le réseau. La fonction boucle sur toutes les dépendances qui représentent des connexions réseau entrantes dans *nœudN* et compte le nombre de connexions. On fait la même chose avec *nœudD*. Les connexions réseau entrantes sont des dépendances. Pour trouver ces connexions distantes entrantes, on vérifie que les noms des serveurs de deux services ayant une dépendance soient différents. Ensuite, c'est le sens de la dépendance qui détermine qui a la connexion entrante ou sortante. Le nœud qui est nécessaire à l'autre est celui qui reçoit la connexion entrante. La fonction retourne un coût uniquement s'il y a une connexion distante présente dans le graphe du réseau normal et aucune dans le graphe de diagnostic.

Algorithm 5: calculeCoutConnexionsDistantes

```
Function calculeCoutConnexionsDistantes(noeudN, noeudD) : int
    let cout = 0;
    let connN = 0;
    let connD = 0;
    foreach segment in noeudN.dependancesEnfant do
        if noeudN.nomServeur != segment.enfant.nomServeur then
            connN++;
        end
    end
    foreach segment in noeudD.dependancesEnfant do
        if noeudD.nomServeur != segment.enfant.nomServeur then
            connD++;
        end
    end
    if connN > 0 and connD == 0 then
        connD++;
        cout += COUT-NOEUD-DOWN;
    end
    return cout;
```

4.3.4 Calculer le coût des dépendances

L'algorithme 6 est la fonction qui s'occupe de calculer le coût en termes de distance d'édition des dépendances entre deux nœuds qui étaient présentes dans l'état normal du réseau et qui ne le sont plus dans l'état à diagnostiquer. En d'autres mots, les dépendances ont disparu. La fonction boucle sur les dépendances des deux nœuds envoyés en entrée dans la fonction et vérifie que ces dépendances ont le même *nom complet*. Le *nom complet* est la composition du nom du serveur avec le nom du service. Par exemple, le serveur X et le service Y auront le *nom complet* de SERVEURX-SERVICEY. Si deux nœuds comparés ont le même *nom complet*, c'est qu'il s'agit du même service installé sur le même serveur dans les deux cas, et donc, il n'y aura pas de coût au niveau de la distance d'édition. Si le *nom complet* n'est pas le même, on vérifie que le nom du service uniquement est le même. Si oui, cela veut dire qu'on a la même dépendance au niveau du service, mais ce service est installé sur un autre serveur. Le service a été déplacé de serveur et on prend note de ce changement dans la liste *listeNouveauServeur*. Si le nom du service n'est pas le même, on passe au service suivant dans la vérification. Si, à la fin de la boucle, une dépendance n'a pas été trouvée, cela veut dire que la dépendance a disparu. Dans ce cas-ci, on vérifie si cette dépendance était locale au serveur. Pour savoir si deux nœuds sont locaux au même serveur, on valide que le nom du serveur des deux nœuds est le même. Si une dépendance disparue était locale, on ajoute le coût d'une dépendance locale. Sinon, on ajoute le coût d'une dépendance distante.

4.4 Conclusion

Au final, l'algorithme de recherche d'anomalie fusionne les graphes qui représentent les captures des états du réseau en deux graphes où le premier représente

Algorithm 6: Calcul du coût des dépendances de nœuds

```
Function calculerCoutDependances(noeudN, noeudD, listeNouveauServeur) :
 int
    let cout = 0;
    foreach depN in noeudN.dependancesParent do
        let dependanceTrouve = false;
        foreach depD in noeudD.dependancesParent do
            if depN.nomComplet == depD.nomComplet then
                dependanceTrouve = true;
                break;
            end
            if depN.nomService == depD.nomService then
                dependanceTrouve = true;
                listeNouveauServeur.add(depN.nomServeur, depD.nomServeur);
                break;
            end
        end
        if !dependanceTrouve then
            if noeudN.nomServeur != depN.nomServeur then
                cout += COUT-DEPENDANCE-DISTANTE;
            else
                cout += COUT-DEPENDANCE-LOCALE;
            end
        end
    end
    return cost;
```

un réseau dans un état normal et le deuxième représente le même réseau dans un état qu'on veut diagnostiquer. Ensuite, l'algorithme 2 calcule la distance d'édition entre ces mêmes deux graphes en utilisant les trois fonctions *calculerCoutServiceArrete* (algorithme 3), *calculeCoutConnexionsDistantes* (algorithme 5) et *verifierDependances* (algorithme 6). L'algorithme traitera aussi les cas de serveurs disparus ou, si un nœud représentant un serveur est présent dans le graphe représentant le réseau normal et n'est pas présent dans l'autre, de serveurs disparus ou remplacés avec la fonction *calculeServeursDisparuOuRemplacer* (algorithme 4). Les coûts utilisés par les fonctions sont arbitraires et peuvent changer selon les scénarios qu'on veut tester, comme nous le verrons au prochain chapitre.

Il faut noter que l'algorithme dans son ensemble est inspiré du calcul de distance d'édition entre deux graphes, mais il n'est pas tout à fait la même chose. Aussi, l'algorithme utilise une approche naïve et non une approche par programmation dynamique.

CHAPITRE V

SCÉNARIOS ET RÉSULTATS OBTENUS

Dans ce chapitre, nous élaborerons sur la détection et la modélisation des dépendances, les différents scénarios de diagnostic qui ont été testés et les résultats obtenus. La qualité de la détection et de la modélisation des dépendances a été comparée avec NSDMiner [1].

5.1 L'environnement de test

Afin d'avoir de vraies données à traiter, le réseau qui a été utilisé pour tester le système est l'environnement de production de Rheinmetall Canada inc.[41] situé à Saint-Jean-sur-Richelieu. Le réseau est composé de plusieurs serveurs de différentes plateformes Windows et Linux et chacun héberge une panoplie d'applications et de bases de données de toutes sortes. Comme le système a été fait pour travailler uniquement avec la plateforme Windows, seul l'environnement Windows de la salle de serveur a été utilisé et les autres plateformes ont été exclues. Donc, 23 serveurs Windows de différentes versions ont été utilisés pour la détection et la modélisation des dépendances réseau. Les modèles qui ont été générés ont ensuite été utilisés pour les scénarios de test de diagnostic. Cependant, les tests de diagnostic ont ciblé des cas bien précis sur un ou deux serveurs spécifiques. Afin d'éviter de s'éparpiller trop, toutes les dépendances à des serveurs Linux ou des

dépendances externes au réseau ont été exclues. Autrement dit, seules les dépendances d'un serveur Windows vers un autre serveur Windows ont été utilisées.

Les captures du réseau modélisées sous forme de graphe et l'exécution des tests de diagnostic ont toutes été faites sur la même station de travail qui a été installée dans la salle de serveur et qui avait un accès direct à tous les serveurs monitorés. Pour le test avec NSDMiner, qu'on introduira au point suivant, l'installation a été faite sur un serveur Linux prévu à cet effet dans la salle de serveur. Le test a été fait avec la version 0.2 de NSDMiner [1] que l'on peut retrouver sur SourceForge.net.

5.2 Détection des dépendances

Dans cette partie, nous détaillerons les tests et résultats obtenus durant les captures de réseau et la détection des dépendances en utilisant notre système et aussi celui de NSDMiner. Nous comparerons également les résultats obtenus avec les deux méthodes.

5.2.1 L'outil NSDMiner

L'outil NSDMiner est un outil de détection de dépendances entre plusieurs services distants sur un réseau. Il est très différent de notre approche. Les premières grandes différences sont qu'il utilise un analyseur de paquet et qu'il est non intrusif. Au lieu de se connecter sur les serveurs dans le but de s'informer des connexions réseau en cours, il capturera les paquets de tout le réseau au moyen d'un analyseur de paquet pour ensuite essayer de déterminer les dépendances des services en écoute sur les serveurs de façon probabilistique.

Comme il est complètement non intrusif, il est incapable de déterminer les dépendances de services qui sont locales à un serveur et les dépendances de services qui ne sont pas en écoute sur le réseau, et il n'a pas de fonctionnalités de détection d'anomalie. Il faut comprendre que NSDMiner [1] a une conception différente d'un service que notre méthode. Pour lui, un service est un port TCP en écoute sur le réseau sur un serveur et une dépendance est un autre port TCP en écoute sur un autre serveur. Il fonctionne de façon que si un service S_1 dépend d'un service S_2, alors lorsqu'un client C_1 accède à S_1, le service S_1 devrait communiquer avec S_2 et recevoir une réponse de lui avant de l'envoyer à C_1. Cette méthode peut générer une quantité énorme de faux positifs (une dépendance qui est détectée, mais n'en est pas une réellement). Nous verrons plus loin que c'est effectivement le cas. Par exemple, cette technique peut facilement mélanger les dépendances de deux services différents hébergés sur le même serveur si les requêtes à ces services et leurs dépendances respectives se déroulent durant les mêmes intervalles de temps.

Notre comparaison est donc entre une méthode qui cherche directement la liste des services et connexions réseau en cours sur chaque serveurs et une méthode qui capture les paquets sur le réseau et essaie d'identifier les dépendances entre les services du réseau. Malgré les différences énoncées plus haut, NSDMiner est l'outil qui se rapproche le plus de ce qu'on a fait, car il permet la détection de services sur des serveurs et de leurs dépendances distantes. La recherche de NSDMiner [1] a été citée 25 fois sur Scopus.

5.2.2 Les vérités établies

Identifier les dépendances possibles des services sur plusieurs serveurs Windows est une tâche importante. Cependant, les identifier toute est irréaliste, car il y a plus d'une centaine de services différents par serveur et chacun peut avoir une ou

plusieurs dépendances. Dans le but d'avoir une méthode afin de pouvoir vérifier la validité des dépendances trouvées par les systèmes comparés, une liste de *vérités établies* a été dressée. Elle a été faite d'une façon similaire à celle dans NSDMiner [1], c'est-à-dire qu'on prend un sous-ensemble de dépendances connues et on vérifie si les systèmes les ont détectées convenablement. La liste de *vérités établies* est un outil utilisé de façon similaire au principe d'un sondage où on prend un sous-ensemble d'une population afin de voir comment le tout se comporte globalement.

Les points 5.2.2.1 à 5.2.2.7 représentent la liste de *vérités établies*. Il s'agit d'une liste de services et leurs dépendances qui sont connues. Le but est de pouvoir vérifier si les systèmes les détectent. Certaines dépendances décrites dans les *vérités établies* demandent une compréhension de l'architecture interne d'un domaine Windows [29] ou de certains produits Microsoft. Bien entendu, on pourrait écrire plusieurs pages d'explications sur chacune des dépendances décrites dans les *vérités établies*, mais le but n'est pas d'évoluer trop sur ce sujet. Il y a tout de même une description sommaire de l'utilité de ces services et le nombre d'instances du service qui est monitoré.

5.2.2.1 Netlogon

Dans l'architecture Windows, le service Netlogon est utilisé pour l'authentification des usagers. Lorsqu'une station Windows est installée en domaine, le service Netlogon est dépendant du service NTDS qui, lui, est installé sur les contrôleurs de domaine. Il s'agit en quelque sorte d'une base de données centrale d'usagers qui s'occupe de faire l'authentification des usagers pour tout le domaine. Les services Netlogon de 19 serveurs différents ont été monitorés, ils sont tous dépendants du service NTDS de deux contrôleurs de domaine. Dans les résultats obtenus par NSDMiner et notre système, chaque serveur a été vérifié de façon à voir si le ser-

vice Netlogon avait bel et bien une dépendance distance avec un service NTDS sur l'un des contrôleurs de domaine.

5.2.2.2 DFSR

Dans l'architecture d'un domaine Windows, le service DFSR (Distributed File Service Replication) sur un contrôleur de domaine est dépendant du service DFSR des autres contrôleurs de domaine. Ce service s'occupe de faire de la réplication de fichiers. Ici, le service DFSR des deux contrôleurs de domaine sont interdépendant. Donc la vérification a été faite afin de voir si les deux contrôleurs de domaine avaient bel et bien leurs services DFSR dépendants des autres.

5.2.2.3 MSExchangeAB

Microsoft Exchange est le serveur de courriel dans la gamme des produits pour serveur de Microsoft. Il est constitué de plusieurs services, dont MSExchangeAB (Microsoft Exchange Address Book) qui s'occupe de la gestion des listes de contacts disponibles à tous les usagers par l'intermédiaire du client Outlook. Ce service est dépendant du service RPCSS des contrôleurs de domaine, car il en a besoin afin d'avoir accès aux listes d'usagers ayant un courriel dans le réseau. Nous avons deux instances du service MSExchangeAB sur deux serveurs différents qui sont dépendants du service RPCSS des deux contrôleurs de domaine. Donc la vérification a été faite que les deux services MSExchangeAB présents dans le réseau ont une dépendance distante avec le service RPCSS sur un des contrôleurs de domaine.

5.2.2.4 MSExchangeMailboxAssistants

Dans l'architecture Microsoft Exchange, le service MSExchangeMailboxAssistants fait la maintenance en arrière-plan des boîtes aux lettres contenues dans les bases de données d'Exchange. Une tâche typique de ce service est d'effectuer la défragmentation des bases de données d'Exchange. Ce service dépend du service MSExchangeIS (Microsoft Exchange Information Store), qui est le service de base de données dans Exchange. Dans notre test, nous avions deux instances du service MSExchangeMailboxAssistants qui dépendaient de deux instances du service MSExchangeIS. La vérification a été faite dans le modèle que les deux services MSExchangeMailboxAssistants sont dépendants *des deux* services MSExchangeIS présents dans le réseau.

5.2.2.5 AuthenticationManagerService

Un produit de la compagnie Ricoh est installé sur un serveur nommé RICOHSRV et fait la gestion de la sécurité des imprimantes. Ce produit est constitué de plusieurs services et l'un d'entre eux (AuthenticationManagerService) est dépendant du service MSSQLSERVER qui est installé localement au même serveur. Ce test est intéressant puisque le service AuthenticationManagerService utilise une connexion réseau à l'IP local du serveur 127.0.0.1 dans le but de faire une connexion au serveur SQL. Donc ici on vérifie que le service AuthenticationManagerService est dépendant du service MSSQLSERVER installé sur le même serveur.

5.2.2.6 Applications diverses SQL

Il y a un serveur Microsoft SQL Serveur qui est installé sur un serveur nommé ASTEROID, qui sert de serveur de base de données à plusieurs applications ins-

tallées sur plusieurs serveurs différents. On a compté onze applications différentes et donc onze services sur onze serveurs différents qui ont une dépendance distante vers MSSQLSERVER installé sur le serveur ASTEROID. Chacun des onze services a été vérifié qu'il avait bel et bien une dépendance distance vers MSSQLSERVER sur ASTEROID.

5.2.2.7 Applications diverses LDAP

Les deux contrôleurs de domaine sont aussi des serveurs LDAP. Plusieurs applications utilisent des requêtes LDAP vers ces deux serveurs. On a monitoré cinq services différents sur cinq serveurs différents qui avaient une dépendance vers le service NTDS, qui, lui s'occupe de répondre aux requêtes LDAP. Chacune des cinq applications a été vérifiée qu'elle avait bel et bien une dépendance vers le service NTDS sur un contrôleur de domaine.

5.2.3 Les captures du réseau

Les captures du réseau pour les deux systèmes ont été faites le même jour de 9h à 17h. Les deux systèmes utilisent des méthodes différentes qui seront expliquées aux points suivants.

5.2.4 Les captures du réseau avec notre système

Une capture du réseau est le processus défini au chapitre 3, soit le processus où le réseau est détecté, les serveurs Windows sont identifiés et interrogés, et le graphe représentant les dépendances des services est construit. Dix captures ont été faites afin de représenter le réseau dans un état normal. Pendant huit heures, les captures du réseau ont été faites avec un intervalle d'un peu moins d'une heure. Les

scénarios de détection d'anomalie, que nous verrons plus loin dans le chapitre, ont aussi eu besoin de trois captures afin de représenter le réseau dans un état à diagnostiquer. Pour la comparaison entre NSDMiner et notre système sur leurs capacités à bien détecter et représenter le réseau, seules les dix captures qui représentent le réseau dans un état normal ont été considérées. Les trois autres captures qui constituent le réseau dans un état à diagnostiquer sont utiles uniquement pour la portion diagnostic de la recherche, et NSDMiner ne fait pas de diagnostic.

Le choix du nombre de captures a été fait en expérimentant sur le nombre. Étant donné que les captures se transforment en graphes pour être ensuite fusionnées en un seul graphe et que les dépendances sur les services intermittents sont enlevées, le graphe résultant de la fusion de tous les graphes peut changer beaucoup lorsqu'il y a peu de graphes à fusionner. La raison est qu'avec peu de captures on a moins de chance de pouvoir détecter les services qui s'exécutent de façon intermittente. Les résultats de l'expérimentation sur le nombre de captures vue à la figure 5.1 montrent que le graphe résultant de la fusion des graphes se stabilise rapidement quand le nombre de graphes fusionnés augmente.

5.2.5 Les captures du réseau avec NSDMiner

NSDMiner [1] fonctionne en utilisant un analyseur de paquet. Dans ce cas-ci, on a utilisé l'utilitaire Linux tcpdump [42]. Avec cet outil, on a pu capturer tous les paquets réseau qui ont transigé entre les 23 serveurs Windows monitorés durant l'expérience. À la fin de la capture, on a sauvegardé dans un fichier .pcap tous les paquets qui ont cheminé entre les serveurs monitorés pendant les huit heures qu'a durés l'expérience.

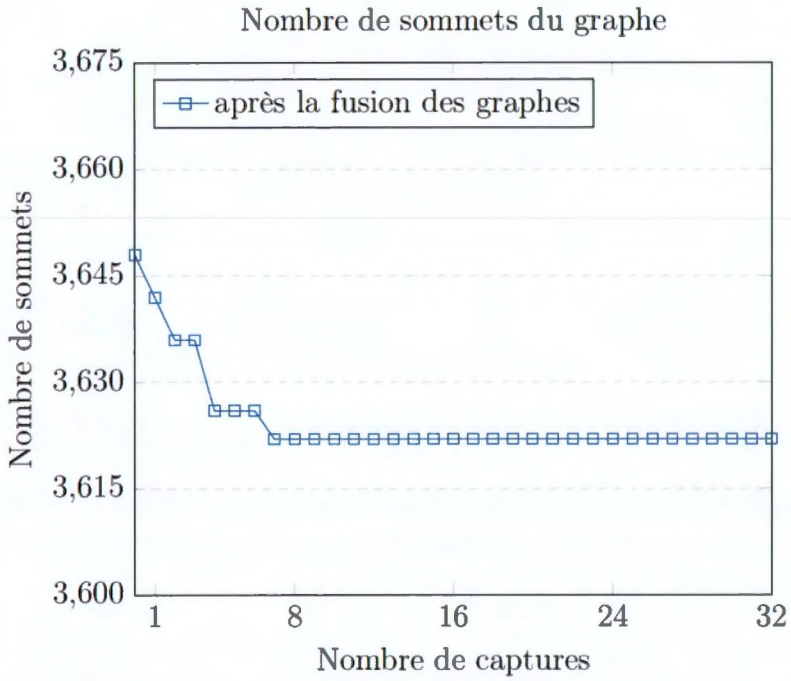

Figure 5.1 Graphe résultant de la fusion des graphes

Le fichier de capture réseau est par la suite envoyé dans une version modifiée de softflowd [43] (modifiée par les auteurs de NSDMiner) qui prend le fichier .pcap qui contient des paquets et le traduit en un fichier .fl qui contient la liste de toutes les communications réseau entre un serveur et un autre. Par exemple, pour une communication TCP, le fichier .pcap contiendra trois paquets pour l'initiation de la connexion TCP pour ensuite compter plusieurs paquets pour l'échange des données, ainsi que les paquets de confirmation TCP et les paquets de terminaisons de connexion TCP, tandis que le fichier .fl ne contiendra qu'une seule ligne représentant la communication au complet.

NSDMiner [1] prendra ensuite le fichier .fl et analysera les communications entre les serveurs dans le but de découvrir les dépendances. Le résultat a la forme d'une

liste où un serveur et un port d'écoute sont listés avec toutes ses dépendances. L'exécution de NSDMiner a été faite avec les paramètres par défaut, c'est-à-dire le nombre de transmissions requises afin de déterminer s'il y a une dépendance ou non sur un service en écoute donné. Le défaut est à 50. En conséquence, le système considérera qu'une dépendance existe s'il voit une communication se répéter 50 fois.

5.2.6 L'analyse des résultats et des deux méthodes

Dans cette partie, nous analyserons les résultats obtenus à partir des deux systèmes et comparerons les deux méthodes utilisées par les deux systèmes.

	#Instance	notre Systeme				NSDMiner			
		VP	VN	FP	FN	VP	VN	FP	FN
Netlogon	19	13	n/a	0	6	0	n/a	0*	19
DFSR	2	2	n/a	0	0	0	n/a	0*	2
MSExchangeAB	2	2	n/a	0	0	0	n/a	0*	2
MSExchangeMailboxAssistants	2	1	n/a	0	1	0	n/a	0*	2
AuthenticationManagerService	1	1	n/a	0	0	0	n/a	0*	1
Applications diverses SQL	11	11	n/a	0	0	0	n/a	0*	11
Applications diverses LDAP	5	4	n/a	0	1	0	n/a	0*	5
Total	42	34	n/a	0	8	0	n/a	0*	42

Tableau 5.1 Résultats obtenus lors de la détection des dépendances

5.2.6.1 L'analyse des résultats

Les résultats obtenus sont présentés dans le tableau 5.1. La première colonne représente les *vérités établies* vues du point 5.2.2.1 au point 5.2.2.7. La colonne

#Instance représente le nombre d'instances du service qui a été testé dans le réseau. Par exemple, le service Netlogon a été testé sur 19 serveurs différents dans le réseau. Les colonnes VP, VN, FP et FN représentent les vrai positif, vrai négatif, faux positif et faux négatif, respectivement. Pour chaque groupe de service testé, NSDMiner n'a pas réussi à découvrir les dépendances tandis que notre système a fait bonne figure. La colonne VN a été incluse simplement pour la forme, mais il faut noter que les vrais négatifs s'appliquent très mal dans le contexte de détection de dépendances, car ils représentent tout ce qui n'a pas été détecté comme étant une dépendance, et que c'était bien que ça n'ait pas été détecté comme tel.

Notre système n'est pas parfait, par contre, car certaines dépendances dans les tests de Netlogon et de MSExchangeMailboxAssistants ont donné des faux négatifs. On peut assumer que les faux négatifs de notre système étaient dus au fait que les captures réseau ont été prises à un moment où ces services n'étaient pas utilisés, et donc aucune entrée n'a pu être détectée avec Netstat pour pouvoir identifier une dépendance. Avec beaucoup plus de captures, ces dépendances auraient peut-être été détectées. Pour les autres, notre système a détecté les dépendances listées dans les *vérités établies*. L'autre aspect intéressant des résultats obtenus est l'absence de faux positifs. En fait, il n'y a pas de possibilité d'avoir de faux positif avec notre système. Si une dépendance a été vue, c'est qu'il y avait vraiment une connexion réseau établie ou une dépendance trouvée par l'intermédiaire des requêtes WMI.

D'un autre côté, NSDMiner [1] a été décevant. Il faut reconnaître qu'il n'était pas utilisé dans un contexte qui lui était favorable. Certaines dépendances n'auraient tout simplement pas pu être détectées, telles que l'AuthenticationManagerService vu au point 5.2.2.5, car ce service fait une connexion TCP/IP localement

sans envoyer de paquet sur le réseau. Ce test a été inclus parce qu'il était intéressant de voir si notre système pouvait détecter une dépendance réseau local à un serveur, mais il représente un cas impossible pour NSDMiner. Cependant, pour les Applications diverses SQL et LDAP, NSDMiner aurait dû faire beaucoup mieux, car il est reconnu pour détecter les dépendances de ce type d'application. Un autre aspect à considérer est le temps de capture. Dans la recherche de NSDMiner [1], ils ont capturé les paquets de leur réseau pendant 46 jours, tandis que nous avons testé notre système pendant seulement 8 heures. Autant de journées de capture nous semblait extrême, mais il est possible que NSDMiner [1] aurait mieux fait avec plus de temps de capture.

Concernant les faux positifs, bien que NSDMiner n'en ait pas détecté dans les services testés de la liste des *vérités établies*, il en a en réalité détecté une quantité énorme sur d'autres services qui n'étaient pas dans les *vérités établies*. La quantité était si significative que ça valait la peine de le mentionner dans ce document. Les tests ont été faits avec le paramètre par défaut, mais on aurait pu changer ce paramètre pour mettre une plus petite valeur. Dans ce cas, il aurait peut-être eu plus de vrais positifs, mais sûrement aussi beaucoup plus de faux positifs. Dans tous les cas, les résultats confirment les résultats obtenus des auteurs de NSDMiner [1] et dans la critique [12] que la détection de dépendances en utilisant un analyseur de paquet peut être très décevante.

5.2.6.2 L'analyse des méthodes

Notre système et NSDMiner utilisent des méthodes bien différentes. Chacune des méthodes a ses pour et ses contre. Les comparaisons se sont faites sur trois catégories que nous verrons au point suivant. Les comparaisons n'ont pas été basées uniquement sur les validations des *vérités établies*, mais aussi sur l'ensemble des

observations faites tout au long de la recherche.

5.2.6.3 Les dépendances locales

La méthode utilisée par notre système est concluante au niveau de la détection des dépendances locales. La raison est que l'information est directement tirée des requêtes WMI et des connexions réseau locales faites d'un service X à un service Y à l'intérieur d'un même serveur. Les vérifications sont donc faciles à faire et il y a peu de place à l'erreur. Dans les *vérités établies*, le test AuthenticationManager-Service est celui qui représentait la détection de dépendance locale au moyen d'une connexion réseau locale. Malheureusement, il n'y avait qu'une seule instance, mais la méthode a tout de même réussie. Des tests sur des dépendances locales basées sur l'information des requêtes WMI n'étaient pas vraiment d'intérêt. L'algorithme aurait à tout coup eu 100%.

Si on compare notre méthode avec celle de NSDMiner [1], on a une amélioration assez claire car NSDMiner [1] ne détecte aucune dépendance locale à un serveur. La raison est qu'il ne détecte les dépendances uniquement qu'avec des captures de paquets réseau faites par des analyseurs de paquets. Il est non intrusif et n'ouvre par conséquent pas de session sur un serveur dans le but d'aller y voir les connexions locales.

5.2.6.4 Les dépendances distantes

Les résultats ont montré que les dépendances distantes peuvent être bien représentées lorsque les connexions TCP sont persistantes. De la même manière que NSDMiner [1], la fidélité de la représentation est proportionnelle à l'activité réseau qu'on peut capturer. Plus on fait de captures et plus la représentation sera fidèle.

La différence sera au niveau de la précision du modèle. Par exemple, NSDMiner [1] pourra détecter que le Serveur A a communiqué avec le Serveur B sur le port P, et donc qu'un Service S_1 du serveur A doit dépendre d'un Servive S_2 qui écoute sur le port P du serveur B. Ici, les deux services S_1 et S_2 sont inconnus et doivent être extrapolés ou trouvés manuellement. Avec notre méthode, la précision sera meilleure. Par exemple, l'algorithme détectera que le Service S_1 sur le Serveur A communique avec le Service S_2 sur le Serveur B, et donc que le Service S_1 doit dépendre du Service S_2. Les services S_1 et S_2 sont connus. Notre méthode constitue une avancée à comparer à celle de NSDMiner [1] concernant les connexions TCP actives puisque nous avons un ajout au niveau de la précision de l'information.

Un autre aspect concernant les dépendances distantes est la viabilité de l'information. Si un système X n'est pas utilisé pendant la capture réseau, les algorithmes n'ont aucune manière de déterminer les dépendances distantes d'un service ou d'un serveur donné. Il y a donc la possibilité d'avoir des faux négatifs. Par contre, avec NSDMiner [1], comme on l'a vu au point 5.2.1, on peut avoir des situations où des paquets sont capturés du réseau d'une manière à penser qu'il y a une dépendance entre certains services sur deux serveurs alors qu'il n'y en a pas en vérité. Alors NSDMiner [1] a la possibilité d'avoir beaucoup de faux positifs. C'est pourquoi ce système comptera le nombre de répétitions de certaines communications entre deux serveurs avant de statuer qu'une dépendance existe afin de limiter les faux positifs. Cependant, avec notre algorithme, il n'y a pas de faux positif possible. Si l'algorithme voit une communication active, c'est qu'il y a bel et bien eu une dépendance entre les deux services.

Un aspect négatif de notre méthode est que nous ne pouvons pas détecter les dépendances réseau sur des services qui utilisent des paquets UDP. La raison est

que le concept de connexion persistante UDP n'existe pas et on n'a aucune façon de le détecter. Sur ce point, NSDMiner [1] est meilleur, car il peut détecter tous les types de paquets. Cependant, les dépendances UDP détectées auront les mêmes limitations au niveau de l'information que celles vues plus haut.

5.2.6.5 Le temps de la prise des captures

Le temps de la collecte d'information est différent dans les deux algorithmes. NSDMiner [1] capturera les paquets du réseau en temps continu pendant des heures afin d'avoir une bonne représentation du réseau. Avec notre méthode, une capture du réseau peut se faire en 3 minutes. Cependant, pour avoir une représentation du réseau la plus fidèle possible, on doit faire plusieurs captures du réseau qui peuvent se répéter dans le temps, ce qui en bout de ligne peut être aussi long. Cependant, sur un aspect plus pratique, la quantité d'espace disque qu'il a fallu pour capturer les paquets de 8 heures pour 23 serveurs Windows (en excluant tous les paquets externes à ces 23 serveurs) a été de 35 Go. Il est facile de concevoir qu'un réseau d'une plus grosse envergure ou plus occupé pourrait demander plus de ressources, tandis que les besoins en ressources pour notre système sont minimes (une sauvegarde des captures réseau sur disque prend 25 Mo et le système utilise 100 Mo de mémoire vive).

5.2.7 Conclusion

On voit que la méthode utilisée dans cette recherche et NSDMiner [1] ont chacun leurs avantages et désavantages. NSDMiner [1] aura besoin d'un analyseur de paquet et d'une capture complète du réseau. Cependant, il ne détectera les dépendances uniquement qu'entre les serveurs tandis que notre méthode aura besoin d'ouvrir une session sur tous les serveurs monitorés et de voir toutes les

connexions TCP actives. La détection de notre système n'inclura pas les transmissions en UDP ni les connexions TCP qui ne sont pas persistantes dans le temps. Il détectera les dépendances entre les services des serveurs et les dépendances locales aux serveurs. De plus, les dépendances sont plus précises, car on peut déterminer quels services sur quels serveurs sont dépendants de quels services sur d'autres serveurs. NSDMiner ne fait aucun diagnostic. Cependant, notre système peut servir à diagnostiquer des problèmes reliés à des dépendances entre plusieurs services. Nous verrons cela à la prochaine section.

5.3 Tests de diagnostic

Dans cette section, nous élaborerons sur les différents tests de diagnostic qui ont été faits. Les scénarios représentent différents problèmes qu'on pourrait vivre à l'intérieur d'une salle de serveur et détecter en utilisant les modèles qui ont été générés à partir du réseau. Chaque test utilise deux représentations du réseau, une pour le réseau normal et l'autre pour le diagnostic. Le but de ces tests est de pouvoir valider la capacité de notre système à détecter des anomalies en comparant l'état du réseau avant et après. Chaque représentation est la fusion des graphes de captures du réseau selon l'algorithme 1. Chaque test a son propre graphe de diagnostic fait spécifiquement pour le test et utilise ensuite l'algorithme de recherche d'anomalie vue au chapitre 4 afin de calculer la distance d'édition afin de confirmer la présence d'anomalie dans le réseau.

Pour chaque test, les poids utilisés dans les différentes parties de l'algorithme ont été spécifiques. L'idée est d'avoir des poids spécifiques afin de déterminer un type de problème uniquement. Il faudrait effectuer d'autres recherches afin d'avoir un système de poids générique pouvant être utilisé dans tous les scénarios. Pour l'instant, seulement des poids de 0 et 1 ont été utilisés. Les explications des choix

des poids sont mentionnées dans chaque scénario.

5.3.1 Aucune anomalie

Pour ce test (figure 5.2), les deux séries de captures du réseau ont été faites l'une à la suite de l'autre rapidement pour volontairement avoir deux représentations du réseau qui sont pareilles. Au niveau des poids, on donne un poids de 1 au calcul de distance d'édition à tous les changements possibles au cours de l'algorithme. Ces poids ont été choisis pour pouvoir détecter n'importe quel changement possible dans le réseau. Comme nous l'avions anticipé, les tests ont donné une distance de 0. On peut donc supposer qu'il n'y a aucune anomalie.

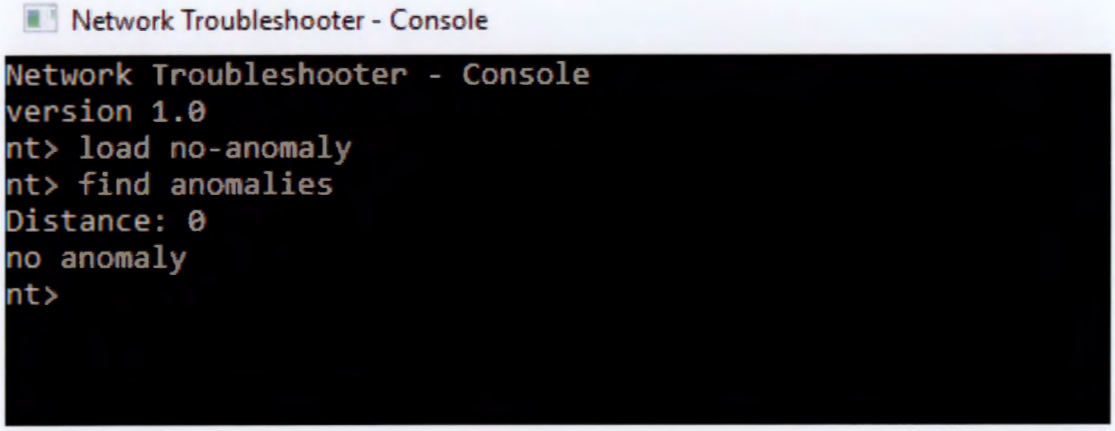

Figure 5.2 Aucune anomalie

5.3.2 Anomalie avec un problème local

Pour ce test (figure 5.3), un serveur installé avec Veritas Backup Exec [44] et Microsoft SQL Server [45] a été utilisé. Ce système a été choisi, car il est composé de plusieurs services qui sont interdépendants sur un même serveur. Ici, un poids de 1 au calcul de distance d'édition à tous les changements possibles permet de

voir les différences entre les deux représentations du réseau. En fait, n'importe quel poids positif ferait le même travail, mais comme on cible une situation bien précise, un poids de 1 est suffisant. Pour le test, nous avons volontairement éteint le service de SQL Server [45] sur la représentation diagnostique. La distance d'édition est de 1 dans ce cas-ci, étant donné qu'il n'y a qu'un seul changement dans le réseau. Tous les services qui dépendent du serveur SQL sont restés en exécution et donc l'algorithme ne détecte pas de changement pour eux, malgré qu'ils soient tous impactés. Avec un parcours de graphe en largeur à partir de la source de l'anomalie, on peut voir quels services sont affectés par cette anomalie, en l'occurrence tous les services de Backup Exec [44]. Le parcours en largeur permet de trouver les services affectés qui sont immédiats au service qui est éteint plus rapidement que le parcours en profondeur.

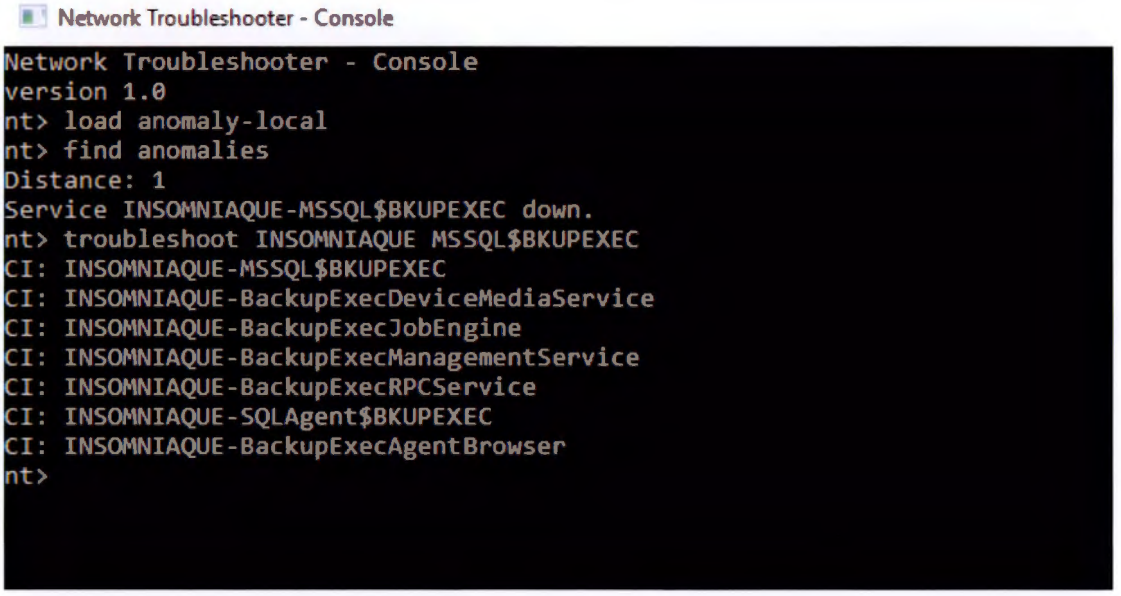

Figure 5.3 Anomalie avec un problème local

5.3.3 Anomalie avec un problème distant

Dans ce test (figure 5.4), toutes les connexions réseau ont volontairement été bloquées sur un serveur de base de données dédié à Microsoft SQL Server [45] utilisé par plusieurs applications différentes installées sur plusieurs serveurs différents. Donc ici le serveur de base de données reçoit habituellement en entrée plusieurs connexions réseau de plusieurs sources différentes avec un débit constant, et il est très anormal pour ce service de ne pas recevoir de connexion réseau. Pour pouvoir repérer une anomalie de ce type, on utilise un poids de 0 sur tous les types de changement, sauf celle des connexions distantes utilisées dans l'algorithme 5 qui, a un poids de 1. Alors le service de Microsoft SQL Server [45] reçoit des connexions de toute part dans le graphe normal et n'en reçoit pas du tout dans le graphe de diagnostic. L'algorithme retourne donc une distance de 1 et conclut qu'il y a effectivement une anomalie.

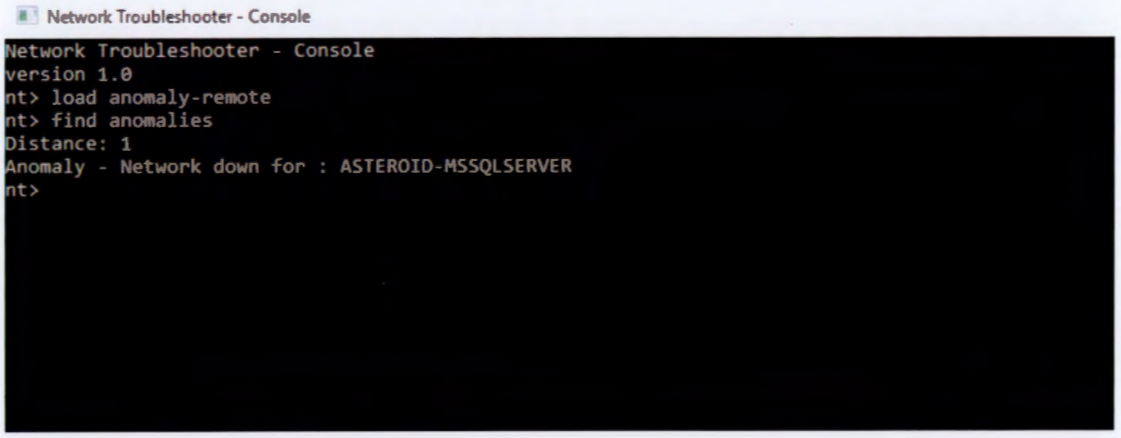

Figure 5.4 Anomalie avec un problème distant

5.3.4 Aucune anomalie sur une migration

Dans ce test (figure 5.5), on utilise un serveur dédié à Microsoft SQL Server [45] qui se nomme ASTEROID. Cependant, on change volontairement le nom du serveur pour NOUVEAU et son IP dans le réseau diagnostique. On donne ainsi l'impression qu'un serveur a été enlevé et un autre, ajouté. Ainsi, toutes les applications qui étaient connectées sur le serveur ASTEROID dans le réseau normal le seront sur le serveur NOUVEAU dans le réseau diagnostic. Le but ici est que l'algorithme puisse déterminer qu'il n'y a aucune anomalie malgré tout puisque les dépendances « disparues » sont remplacées. Par contre, les serveurs qui n'ont pas de remplacement doivent être détectés comme étant en problème (serveur éteint). C'est pourquoi on utilise un poids de 1 pour tous les changements possibles, sauf pour le coût d'un serveur remplacé, qui est de 0. Ici, l'algorithme donne une distance de 0 puisque le seul changement est un remplacement qui vaut 0 et, surtout, l'absence des autres changements possibles qui donnerait un poids positif. On peut donc conclure qu'il n'y a pas d'anomalie. L'algorithme est aussi en mesure de donner une explication sur le remplacement ainsi que la distance de 0 en utilisant les deux listes mentionnées au point 4.3.

5.3.5 Conclusion

Pour conclure cette partie, on voit bien qu'en ajustant les poids utilisés dans le calcul de la distance d'édition on peut utiliser la distance d'édition afin de trouver différents types d'anomalies à l'intérieur des graphes des représentations réseau. Les calculs de distance d'édition doivent être très précis afin de cibler le type d'anomalie qu'on recherche. Plusieurs tests et recherches seraient encore nécessaires afin de pouvoir diagnostiquer plusieurs anomalies différentes à l'intérieur d'un même graphe en utilisant un algorithme générique.

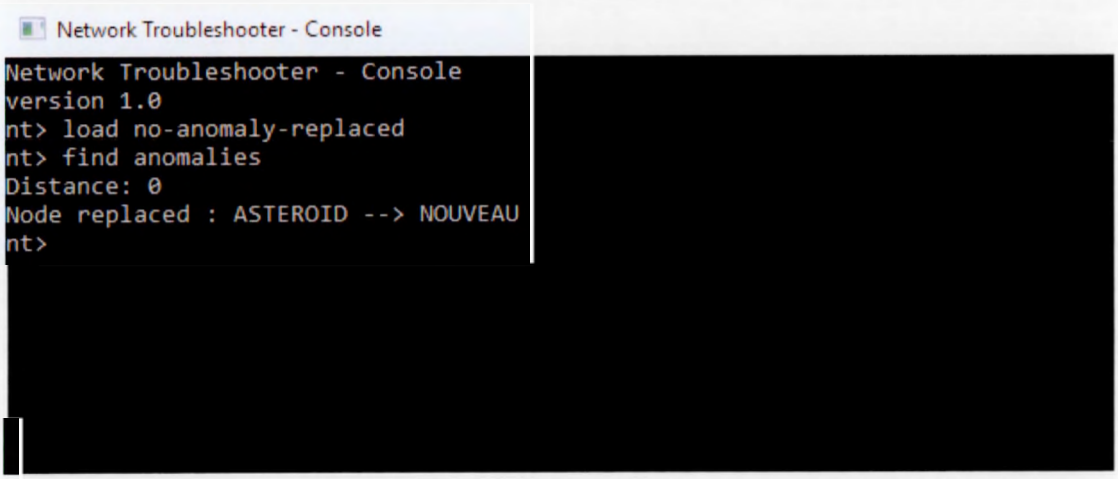

Figure 5.5 Aucune anomalie avec un remplacement

CONCLUSION

Il y a une critique dans [12] qui vaut la peine d'être mentionnée : la majorité des recherches dans ce domaine sont faites dans des réseaux et des contextes très précis qui ne peuvent être répliqués. Notre recherche ne fait pas exception. Même répliquer l'expérience dans le même réseau qu'on a utilisé donnerait des résultats différents, car depuis le temps de l'expérimentation le réseau a déjà changé de façon significative.

Malgré tout, notre méthode détecte automatiquement les serveurs Windows pour ensuite collecter l'information de chacun, détecter les dépendances réseau et modéliser le tout sous forme de séquences de graphes. La fusion des séquences de graphes permet ensuite d'avoir un modèle plus juste du réseau et d'utiliser les algorithmes de graphe. Notre méthode, dans le cas précis de notre environnement de test, a été nettement supérieure à NSDMiner. On pourrait aussi penser qu'elle serait meilleure que les méthodes qui utilisent un analyseur de paquet.

Nous sommes loin du moment où les systèmes pourront se diagnostiquer et se corriger tout seuls. Il faudra nécessairement plusieurs autres recherches. Des suggestions de recherches futures dans ce domaine seraient d'étendre notre méthode à la plateforme Linux. On pourrait aussi ajouter d'autres formes de dépendance telles que les composantes matérielles. On pourrait également voir comment se comporterait le système dans le cas où plusieurs anomalies seraient détectées en même temps. L'évolution dans le temps serait aussi un sujet de recherche, faire en

sorte que le système puisse faire automatiquement une capture du réseau toutes les cinq minutes, garder un bon volume de captures, toujours conserver la capture la plus récente et supprimer la capture la plus vieille. On pourrait voir comment la représentation d'un réseau peut évoluer sur plusieurs jours ou semaines et cerner combien et quel type d'anomalies il peut trouver. Enfin, il y aurait du travail pour plusieurs autres maîtrises.

RÉFÉRENCES

[1] A. Natarajan, P. Ning, Y. Liu, S. Jajodia, and S. E. Hutchinson, "NSDMiner : Automated discovery of network service dependencies," in *Proceedings - IEEE INFOCOM*, pp. 2507–2515, 2012.

[2] Z. Yang, Y. Wang, and J. Lv, "Survey of modern fault diagnosis methods in networks," in *2012 International Conference on Systems and Informatics, ICSAI 2012*, no. Icsai, pp. 1640–1643, 2012.

[3] L. Feng, L. Xiang, and W. Xiu-qing, "A survey of intelligent network fault diagnosis technology," *2013 25th Chinese Control and Decision Conference (CCDC)*, no. 60974063, pp. 4874–4879, 2013.

[4] M. Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of Computer Programming*, vol. 53, no. 2 SPEC. ISS., pp. 165–194, 2004.

[5] M. Al-Emran and H. Al Chalabi, "Developing an IT Help Desk Trouble-shooter Expert System for diagnosing and solving IT Problems," in *BCS International IT Conference 2014*, pp. 1–5, 2015.

[6] T. Sugawara, "A Cooperative LAN Diagnostic and Observation Expert System," in *Computers and Communications, 1990. Conference Proceedings., Ninth Annual International Phoenix Conference on*, pp. 0–7, 1990.

[7] H. R. Berenji and Y. Wang, "Case-based reasoning for fault diagnosis and prognosis," in *Fuzzy Systems, 2006 IEEE International Conference on*, pp. 1316–1321, 2006.

[8] J. Prentzas and I. Hatzilygeroudis, "Categorizing approaches combining rule-based and case-based reasoning," *Expert Systems*, vol. 24, no. 2, pp. 97–122, 2007.

[9] W. Z. W. Zeng and Y. W. Y. Wang, "Design and Implementation of Server Monitoring System Based on SNMP," *2009 International Joint Conference on Artificial Intelligence*, vol. 1, pp. 2–4, 2009.

[10] D. R. Mauro and K. J. Schmidt, *Essential SNMP, Second Edition.* O'Reilly Media, Inc., 2005.

[11] K. ElDefrawy, T. Kim, and P. Sylla, "Automated Inference of Dependencies of Network Services and Applications via Transfer Entropy," in *Proceedings - International Computer Software and Applications Conference*, vol. 2, pp. 32–37, 2016.

[12] T. E. Carroll, S. Chikkagoudar, and K. Arthur-Durett, "Impact of network activity levels on the performance of passive network service dependency discovery," in *Proceedings - IEEE Military Communications Conference MILCOM*, vol. 2015-12, pp. 1341–1347, 2015.

[13] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma, "Constellation : automated discovery of service and host dependencies in networked systems," *Microsoft Research, Msr-Tr-2008-67*, pp. 1–14, 2008.

[14] A. Zand, G. Vigna, R. Kemmerer, and C. Kruegel, "Rippler : Delay injection for service dependency detection," in *Proceedings - IEEE INFOCOM*, pp. 2157–2165, 2014.

[15] M. Iliofotou, "Exploring Graph-based Network Traffic Monitoring," in *IEEE INFOCOM Workshops 2009*, pp. 1–2, 2009.

[16] T. IDÉ and H. KASHIMA, "Eigenspace-based anomaly detection in computer systems," in *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '04*, p. 440, 2004.

[17] M. Iliofotou, M. Mitzenmacher, and G. Varghese, "Network Monitoring using Traffic Dispersion Graphs (TDGs)," in *IMC '07 Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pp. 315–320, 2007.

[18] I. Katzela and M. Schwartz, "Schemas for fault identification in communication networks," *IEEE Transactions on Networking*, vol. 3, no. 6, pp. 753–764, 1995.

[19] H. Bunke, P. J. Dickinson, M. Kraetzl, and W. D. Wallis, *A Graph-Theoretic Approach to Enterprise Network Dynamics*. Birkhäuser Basel, 2007.

[20] C. Lu, Q. Xue-song, and Z. Nan, "A service-oriented alarm correlation method for IT service fault localization," in *First International Conference on Communications and Networking in China, ChinaCom '06*, pp. 1–5, 2007.

[21] D. Baker, M. Nodine, R. Chadha, C. J. Chiang, Y. Gottlieb, C. F. Hsu, R. Jaeger, G. Levin, and Y. Ling, "Computing diagnostic explanations of network faults from monitoring data," in *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pp. 1–7, 2008.

[22] M. Steinder and A. S. Sethi, "Probabilistic fault localization in communication systems using belief networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 809–822, 2004.

[23] V. R. Syrotiuk, K. Shaukat, Y. J. Kwon, M. Kraetzl, and J. Arnold, "Application of a network dynamics analysis tool to mobile ad hoc networks," *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems - MSWiM '06*, p. 36, 2006.

[24] P. J. Dickinson and M. Kraetzl, "Novel Approaches in Modelling Dynamics of Networked Surveillance Environment," in *Proceedings of the 6th International Conference of Information Fusion.*, pp. 302–309, 2003.

[25] J. Abello, T. Eliassi-Rad, and N. Devanur, "Detecting novel discrepancies in communication networks," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 8–17, 2010.

[26] B. Pincombe, "Anomaly detection in time series of graphs using Fusion of Graph Invariants," *IEEE Journal of Selected Topics in Signal Processing*, vol. 24, no. 4, p. 2, 2005.

[27] K. Sricharan and K. Das, "Localizing anomalous changes in time-evolving graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*, pp. 1347–1358, 2014.

[28] T. Wang, C. Fang, D. Lin, and S. F. Wu, "Localizing Temporal Anomalies in Large Evolving Graphs," in *Proceedings of the 2015 SIAM International Conference on Data Mining*, pp. 927–935, 2015.

[29] Wikipedia, "https ://en.wikipedia.org/wiki/Microsoft_Windows," *Windows [Accessed 2018-12-07].*

[30] Wikipedia, "https ://en.wikipedia.org/wiki/Linux," *Linux [Accessed 2018-12-07].*

[31] Wikipedia, "https ://en.wikipedia.org/wiki/Background_Intelligent_Transfer_Service," *Background Intelligent Transfer Service [Accessed 2018-12-07].*

[32] DMTF, *WBEM Server Profile.* DMTF, 2013.

[33] DMTF, *Common Information Model (CIM) Metamodel.* DMTF, 2012.

[34] Wikipedia, "https ://en.wikipedia.org/wiki/Netstat," *Netstat [Accessed 2018-12-07].*

[35] Wikipedia, "https ://en.wikipedia.org/wiki/Ping_(networking_utility)," *Ping [Accessed 2018-12-07].*

[36] Wikipedia, "https ://en.wikipedia.org/wiki/Tasklist," *Tasklist [Accessed 2018-12-07].*

[37] Microsoft, "https ://msdn.microsoft.com/en-us/library/dd163506.aspx," *WinRS [Accessed 2018-12-07].*

[38] U. Schöning, "Graph isomorphism is in the low hierarchy," *J. Comput. Syst. Sci.*, vol. 37, no. Dec. 1988, pp. 312—-323, 1988.

[39] B. T. Messmer and H. Bunke, "A New Algorithm for Error Tolerant Subgraph Isomorphism Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 493–504, 1998.

[40] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *J. ACM*, vol. 21, pp. 168–173, jan 1974.

[41] Rheinmetall Canada inc., "https ://www.rheinmetall.ca/en/rheinmetall_canada/index.php," *Rheinmetall Canada inc. [Accessed 2018-12-07].*

[42] Wikipedia, "https ://en.wikipedia.org/wiki/Tcpdump," *tcpdump [Accessed 2018-12-07].*

[43] Softflowd, "https ://www.mindrot.org/projects/softflowd/," *Softflowd [Accessed 2018-12-07].*

[44] Veritas, "https ://www.veritas.com/en/ca/product/backup-and-recovery/backup-exec," *Backup Exec [Accessed 2018-12-07].*

[45] Microsoft, "https ://www.microsoft.com/en-ca/sql-server/sql-server-2017," *SQL Server [Accessed 2018-12-07].*