UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*
Par

« Sihem Cherrared »

« **Gestion des fautes dans les réseaux multi-tenants et programmables** »

Thèse présentée et soutenue à , le 26 Juin 2020
Unité de recherche : ORANGE Labs Netwoks & INRIA

Rapporteurs avant soutenance :
Hind Castel, Professeur Télécom Sud Paris
Stefano Secci, Professeur, CNAM Paris

Composition du jury :

Hind Castel, Professeur, Télécom Sud Paris. *Rapporteur*
Eric Fabre, Directeur de recherche, INRIA Rennes. *Directeur de thèse*
Yacine Ghamri-Doudane, Professeur, Université La Rochelle. *Examinateur*
Gregor Gössler, Chargé de recherche, INRIA Grenoble. *Examinateur*
Sofiane Imadali, Ingénieur de recherche, Orange Labs.*Membre invité*
Stefano Secci, Professeur, CNAM Paris. *Rapporteur*
Sandrine Vaton, Professeur, IMT Brest. *Examinateur*

Dir. de thèse : Eric Fabre, Directeur de recherche, INRIA.
Co-dir. de thèse : Gregor Gössler, Chargé de recherche, INRIA.

ACKNOWLEDGEMENT

Je tiens tout d'abord à remercier Dieu de m'avoir donné la force et la patience d'accomplir ce modeste travail.

Cette thèse n'aurait pas été possible sans l'intervention, consciente, d'un grand nombre de personnes. Je souhaite ici les en remercier.

La confiance, la patience et les conseils de mes encadrants de thèse ont constitué un apport considérable sans lequel ce travail n'aurait pu être mené à bon port. Mais aussi, l'intérêt qu'ils ont porté à ce sujet de thèse et leurs contributions dans le domaine.

Je tiens à remercier, mon directeur de thèse Eric Fabre d'avoir cru en moi dès le début pour accomplir ce sujet de thèse et d'avoir enrichi mon travail avec son savoir.

Je remercie mon encadrant Sofiane Imadali de m'avoir soutenue, de m'avoir toujours encouragée à valoriser mon travail et à aller plus loin.

Je remercie mon encadrant Gregor Goessler pour la qualité de son suivi et de son orientation pour enrichir et mener à bien mes travaux de thèse.

Mes vifs remerciements vont également aux membres du jury pour l'intérêt qu'ils ont porté à mon travail en acceptant de l'examiner et de l'enrichir par leurs propositions. Je remercie les rapporteurs, pour avoir pris le temps de relire mon manuscrit et pour leurs commentaires pertinents.

Je n'oublie pas mes chers parents, mes frères, ma sœur qui ont toujours été là pour me soutenir à affronter les difficultés à leur façon au cours de cette thèse. Mais aussi, mon neveu et mes nièces qui m'ont apporté amour et joie dans ma vie.

Bien évidemment, je tiens à remercier mes collègues de l'équipe Orange et Sumo pour les moments agréables passés ensemble durant ces années de thèse. Enfin, j'adresse mes plus sincères remerciements à tous mes proches et amis, qui m'ont toujours soutenue et encouragée au cours de la réalisation de ce travail.

DEDICATION

To my beloved Mother,
To my father,
To my brothers and sister.

RÉSUMÉ EN FRANÇAIS

Contexte et problématiques

La nouvelle génération mobile 5G s'accompagne de l'adoption de nouvelles technologies du monde du cloud-computing, avec notamment l'introduction de fonctions réseau virtualisées [79, 151], c'est-à-dire réalisées par du logiciel plutôt que par des machines spécialisées. Par ailleurs, des milliards d'objets avec différents niveaux d'exigence seront à connecter. Ce nouvel environnement offre des services plus dynamiques et flexibles, tout en réduisant les coûts de mise en œuvre et de maintenance. En outre et afin de bénéficier davantage de la virtualisation, les opérateurs partagent leurs ressources non occupées avec les clients. Ces clients proviennent de différents secteurs avec des services de niveaux de priorité distincts (par exemple un réseau hospitalier déployé sur l'infrastructure opérateur). Cela est rendu possible via le concept de "multi-tenancy" qui permet la coexistence de plusieurs clients dans une même infrastructure.

La fiabilité des services dans ce type d'environnement devient un enjeu primordial. Une simple panne peut potentiellement engendrer des pertes humaines et monétaires conséquentes. D'autant plus que le secteur touché est sensible tels que les véhicules connectés autonomes, les processus industriels ou les robots chirurgicaux. Il est donc nécessaire de détecter et localiser rapidement les pannes, tout en progressant vers une auto-réparabilité.

Pour un opérateur réseau et cloud comme Orange, offrir des services dotés d'une forte résilience et une garantie de recouvrement après panne, tout en optimisant l'usage des ressources, sera un argument clé envers ses futurs clients. En effet, une panne d'un service peut engendrer des pertes monétaires, incluant les coûts de réparation et de mobilisation humaine. D'un autre côté, ces pannes peuvent toucher à l'image de qualité de services perçue par ses clients. Un exemple d'une telle perte a été apportée en 2013 par IBM, où l'un de ses serveurs est tombé en panne pendant une semaine causant l'arrêt d'un site web australien et une perte effective estimée à 31 millions de dollars [13]. Il est donc primordial de gérer les pannes, d'autant plus que la virtualisation des services est un aspect émergeant dans le monde des télécommunications.

Alors que la gestion de pannes est une discipline déjà connue par les opérateurs et un problème adressé plusieurs fois par le passé, l'introduction de la virtualisation apporte un ensemble de défis et de problèmes qui rendent obsolètes certaines approches de gestion et localisation de pannes. Parmi ces défis, on peut notamment citer la topologie dynamique. En effet, le changement de topologie constant des entités réseaux complique la localisation de la racine

de la panne et peut engendrer de faux résultats.

L'objectif de cette thèse est de garantir la disponibilité des services de réseaux de télécommunications virtualisés en développant des mécanismes de suivi de causalité de fautes. Ces mécanismes permettent la découverte des composants fautifs et ainsi d'atteindre l'isolation des ressources et d'éviter la propagation de fautes à l'ensemble du système. Le travail de ma thèse décrit dans ce manuscrit est structuré comme suit:

- Dans le chapitre 2, nous présentons le contexte générale de la thèse en expliquant la composition et les caractéristiques des architectures de télécommunications virtualisées. Nous décrivons le cas d'usage du "Virtual IP Multimédia Subsystem (vIMS)" [19] qui est la version virtualisée du standard de communication multimédia: IMS.

- Le chapitre 3 présente notre première contribution qui est un état de l'art sur les approches classiques de gestions de pannes et les limitations de ces approches face aux défis de l'environnement virtuel multi-tenant. Par la suite, nous décrivons des techniques de gestion de pannes plus récentes proposées pour les réseaux virtuels et nous positionnons notre travail de thèse par rapport a l'état de l'art. Ce chapitre 2 à fait l'objet d'une publications dans journal IEEE Transaction [16].

- Dans le chapitre 4, nous proposons une plateforme de gestion de pannes globale qui représente les différentes étapes de gestion de pannes: détection, localisation et réparation de pannes, ainsi que l'ensemble des outils open source utilisés dans chaque étape.

- Le chapitre 5 décrit notre proposition d'auto-modélisation des réseaux virtuels. Le modèle proposé est un graphe de dépendances logique avec les nœuds qui représentent des variables booléennes. Cette proposition a été appliquée sur le use-case vIMS présenté dans le chapitre 2. Nous utilisons la technique de gestion de pannes pour valider, corriger et étendre le modèle. Une algorithmique d'auto-modélisation a été proposée pour permettre de modéliser la topologie actuelle du réseau. Nous proposons a la fin des tests de performance appliqués a l'algorithme d'auto-modélisation.

- Le chapitre 6 propose une procédure de diagnostic de pannes actif qui valide notre approche d'auto-modélisation. Cette procédure prend en considération le graphe de dépendances produit par l'algorithme d'auto-modélisation et les observations (symptômes de pannes) initiales pour retrouver les causes primaires des pannes. La procédure de diagnostic de pannes introduit le concept de tests qui permet de rajouter des observations au fur et à mesure que le diagnostic avance. Pour valider la procédure de diagnostic, nous avons appliqué cette procédure sur des scénarios de pannes réelles.

Dans ce qui suit nous allons présenter un résumé des résultats de la thèse.

Limitation des approches de gestion de pannes face à la virtualisation des réseaux

Le diagnostic de panne est le processus de détection, localisation et résolution des pannes et défaillances du réseau. La figure 1 illustre les différentes étapes et approches de gestion de pannes. La première phase dans le diagnostic de pannes consiste a détecter que le système est en état de défaillance. L'étape de détection de panne détermine si le système fonctionne dans des conditions normales ou si une panne (ou faute) s'est produite. Une *faute* est la *cause primaire* qui peut conduire le système à un état d'erreur. Un échec se produit lorsqu'une erreur provoque un *dysfonctionnement* des périphériques réseau ou des logiciels, ce qui entraîne des symptômes. Les symptômes sont des manifestations externes d'échecs, ils peuvent être observés comme des alarmes, c'est-à-dire des notifications d'échecs potentiels [133].

Pour la détection de pannes, deux types de données sont collectées: les métriques et les alarmes. Les métriques représentent un moyen quantitatif de vérifier les aptitudes souhaitées et de mesurer les dégradations. Ils mesurent l'activité et l'état de fonctionnement de toutes les couches du réseau. Les métriques du réseau comprennent: la gigue, le débit, l'utilisation du réseau, la latence et les pertes de paquets. les alarmes représentent des manifestations externes d'échecs. Ces notifications peuvent provenir d'agents de gestion comme le trap SNMP [127], ou au format de journaux système générés par le protocole syslog (ou d'autres protocoles) [42].

La localisation des pannes (également appelée isolation des pannes) représente la procédure permettant de déduire la cause racine exacte d'une panne. Dans le cadre de localisation de pannes un certain nombre d'approches et méthodologies ont été développés [16]. Ces approches apportent une automatisation de la procédure de gestion de pannes et sont appliquées pour la prédiction de pannes, localisation des composants fautifs ou encore l'analyse de propagation de fautes. Une grande partie de ses approches sont des approches Machine Learning (ML) avec une capacité d'inférer des résultats à partir des données ou modèles. On peut classer ces approches en deux grandes catégories: boite blanche et boite noire.

Les boites blanches, telles que les réseaux bayésiens, sont des approches qui se basent sur un modèle pour le diagnostic. Un modèle représente un ensemble de nœuds et dépendances. Les nœuds représentent des composants réseau, évènements ou fautes. Les dépendances entre nœuds représentent des liens causals, logiques ou probabilistes. Les approches du type boite blanche offrent une meilleure explication du déroulement de la panne en se basant sur le modèle. Alors que les boites noires, telles que les réseaux de neurones, sont des techniques qui apprennent sur un jeu de données un modèle sans vraiment apporter des explications à ce qui a été appris. Un exemple d'application des approches d'apprentissage, serait de prédire si notre système se dirige vers un état de panne ou non en utilisant un modèle de

classe binaire en sortie.

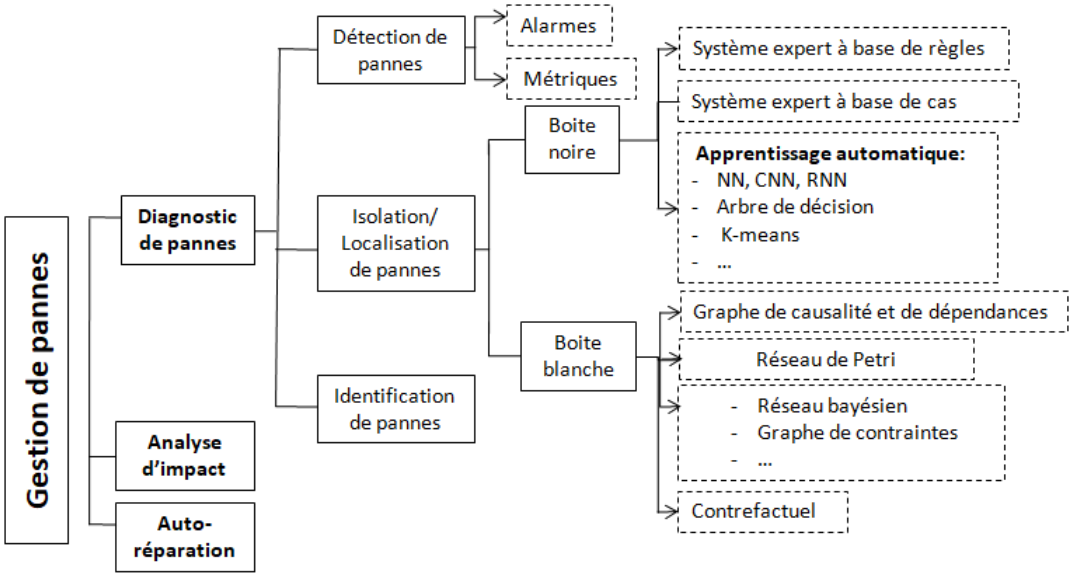


Figure 1 – Les étapes et approches de gestion de pannes.

Les problématiques classiques dans le monde des réseaux se heurtent à de nouveaux défis apportés par la virtualisation, notamment en termes de fiabilité et de disponibilité.:

- **Topologie de réseau dynamique**: la 5G permettra le déploiement de services en temps réel adaptés aux demandes des clients. cette reconfiguration en permanence rend l'évolution de la topologie du réseau et des dépendances des entités sur le réseau imprévisibles. Le système de gestion doit prendre en compte les modifications de la topologie en temps réel pour identifier les composants fautifs et éviter les faux positifs.
- **Manque de visibilité du réseau**: La distribution des fonctions virtuelles sur plusieurs sites offre plus de robustesse face aux pannes de services. Cependant, cela entraîne une nouvelle préoccupation de gestion qui est: la distribution ou centralisation des logs. En effet, les logs d'un même service peuvent être distribués dans plusieurs sites ce qui implique un manque de visibilité et complique le diagnostic de pannes. Par conséquent, pour gérer les services, une vue globale du réseau est nécessaire.
- **Isolation de pannes**: le partage de l'infrastructure entre plusieurs clients, nécessite une isolation des fautes et une notification rapide des clients affectés.
- **Croissance, ambiguïté et incohérence des alarmes**: le nombre croissant de services dans les réseaux virtuels implique davantage d'entités à gérer avec un nombre croissant d'alarmes. Ces alarmes sont produites par diffèrents agents comme Syslog et dans les

différentes couches du réseaux. De plus, une panne peut se propager à travers les couches et les sites, augmentant le nombre d'alarmes. La provenance de ces alarmes de différents sites et entités engendre une ambiguïté et une incohérence des alarmes. Dans ce cas, la gestion des d'alarmes nécessite des techniques de stockage et de filtrage modernes avec des notifications précoces d'échecs graves [45].

Dans le travail de ma thèse, nous optons pour une approche basée modèle. Ce type d'approche offre une meilleure explication du diagnostic, mais souffre d'une limitation majeure qui est la difficulté de définir le modèle appliqué pour le diagnostic. D'autant plus, que les défis de la virtualisation cités précédemment compliquent la dérivation de ce modèle. En effet, le modèle défini doit représenter les différentes couches du réseau virtuel (physique, virtuel, application et service). le modèle doit aussi représenter les granularités les plus fines (par exemple, les processus d'application). Ce modèle doit aussi s'adapter au changement de topologie du réseau.

LUMEN: plateforme de gestion de pannes des réseaux virtuels

Afin de relever les défis liés à la virtualisation, nous proposons LUMEN: une plateforme globale de gestion des pannes [17]. *LUMEN* est une architecture en quatre étapes, notamment : Source, Puits, Extraction et Décision. Chacune des étapes récapitule les méthodes qui doivent être déployées pour répondre aux différents défis de la virtualisation qui sont: l'isolation de pannes clients, l'ambiguïté et la croissance des logs et le manque de visibilité du réseau.

La figure 2 illustre les phases de *LUMEN*. Les phases source, et puits représentent la partie collecte, filtrage et stockage de données telles que les logs ou la description de topologie. Dans cette phase les logs des différents clients sont centralisés pour résoudre le manque de visibilité réseau. Ces logs sont aussi identifiés pour chaque client afin de les isoler. La phase extraction permet d'extraire les données nécessaire selon les besoins de l'approche de décision. Par exemple, seuls les logs qui indiquent un mal-fonctionnement du réseau sont récupérés pour le diagnostic.

LUMEN s'appuie sur un ensemble d'outils open source, notamment Elastic Stack [31], pour la partie stockage et filtrage des données. La contribution majeure de *LUMEN* est la préparation et extraction des données nécessaires pour les approches de diagnostic. Par exemple, dans la partie collecte d'alarmes, un identifiant de client unique est rajouté pour classer les évènements par client et ainsi répondre au besoin d'isolation de pannes clients.

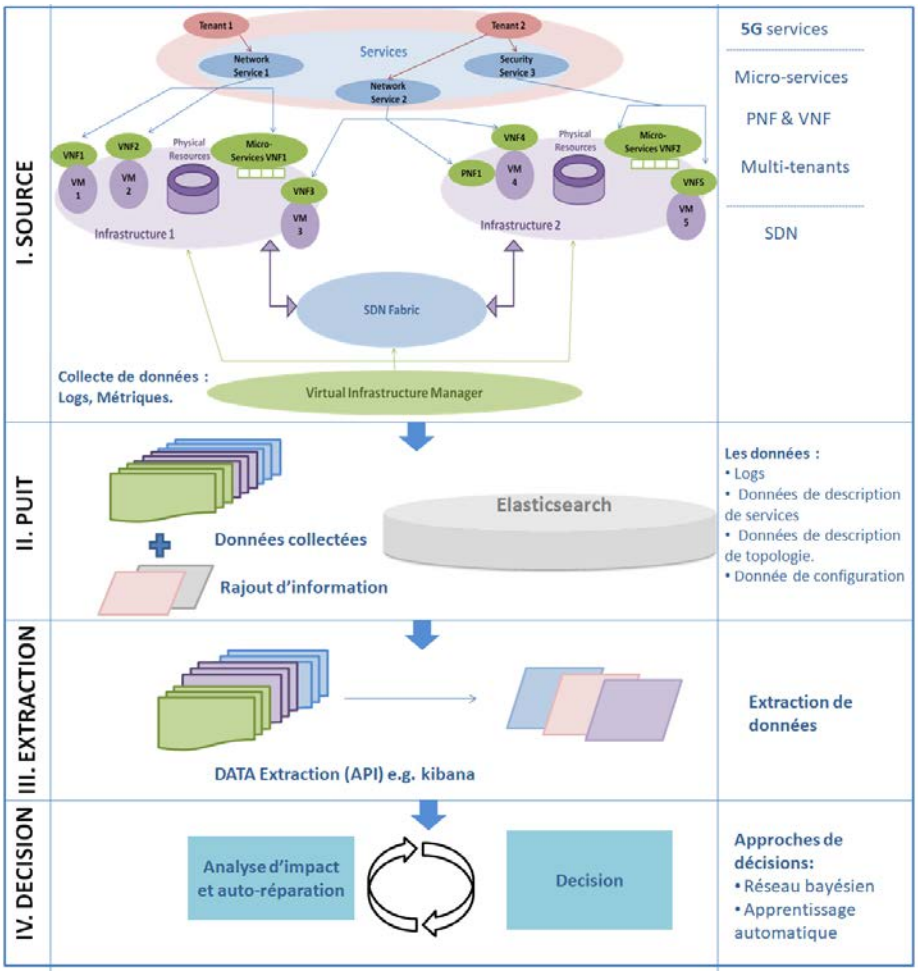


Figure 2 – La platforme *LUMEN*.

Approche d'auto-modélisation et diagnostic actif

Après avoir défini les étapes de gestion de pannes à travers la plateforme *LUMEN*, nous avons opté par la suite pour une approche boite blanche intégrant le mécanisme de suivi de causalité de faute, étant donné que ces approches apportent des explications plus détaillées sur les pannes et nous permettent ainsi d'effectuer une réparation ciblée. Notre motivation pour ce choix malgré la difficulté de la définition du modèle, est l'existence de deux types de connaissances dans les réseaux virtuels : connaissances *acquises* et connaissances *apprises*. Les connaissances acquises peuvent être récupérées des fichiers de description de réseau, des protocoles ou même des experts. Dans un de nos papiers [18], nous avons présenté les différents types de données acquises des réseaux virtuels. D'un autre côté, les connaissances apprises sont récupérées par un mécanisme d'injection de fautes qui consiste à injecter des

pannes telles que la rupture d'un lien de connexion et la collecte d'évènements associés. Ce mécanisme a été introduit par Netflix avec le projet ChaosMonkey pour tester la résilience de leur plateforme [90]. Dans notre cas ce mécanisme a été utilisé pour apprendre les dépendances du modèle.

Pour modéliser les réseaux virtuels nous avons défini un ensemble de "*templates*". Les *template* définissent les composants du réseaux dans chaque couche du réseau virtuel. Une *template* est un graphe orienté acyclique $G = (V, E)$ avec V l'ensemble des nœuds qui représentent des variables booléennes et E l'ensemble des liens logiques. Nous avons défini quatre types de liens logiques entre un nœud et ses prédécesseurs: $dv = \{ET, OU, \neg A \implies \neg B, A \implies B\}$, où A et B sont deux nœuds du graphe.

L'approche de modélisation proposée se base sur le principe de la programmation orientée objet. Les *templates* peuvent être considérés comme des "classes d'objets" dans le paradigme orienté objet. Les règles de modélisation utilisées réassemblent ensuite un diagramme de classes. Le but des règles de modélisation est de construire un graphe de variables booléennes et de dépendances logiques décrivant tous les composants du réseau à l'aide des *templates* définis (un diagramme d'objet, suivant la métaphore de programmation orientée objet). Le graphe résultant représente des instances assemblées de *templates*. Les composants de même nature ont le même modèle avec des instances distinctes de ce modèle. L'instanciation de ces *templates* est opérée par l'algorithme d'auto-modélisation. L'algorithme d'auto-modélisation prend en considération les *templates* définis et la topologie actuelle du réseau décrite dans un ficher YAML[1]. Les *templates* sont définis d'une manière un peu générale qui répond a tous les use cases de réseaux de télécommunications virtuels. Ces définitions intègrent plusieurs aspect commun entre ces use-case tels que le principe d'élasticité des fonctions réseaux virtuelles et les mécanismes d'auto-réparation. Nous avons par la suite appliqué cette approche d'auto-modélisation sur le use-case vIMS avec le projet open source *Clearwater* en version Docker [120]. Docker étant une technologie de virtualisation plus performante et légère (moins d'overhead système) que les machines virtuelles [115]. Nous utilisons une procédure d'injection de fautes pour valider, corriger et étendre les *templates* définies. Le principe et de comparer la propagation de fautes dans un déploiement réel contre la propagation dans le modèle.

La figure 3 illustre un exemple de propagation d'une faute dans un vrai déploiement contre sa propagation dans le modèle de dépendances correspondant à l'architecture. L'architecture de déploiement *Clearwater* vIMS est composée de cinq composants essentiels pour permettre l'authentification des clients. Ces composant sont: Bono, Sprout, Homestead, Cassandre et ETCD. Bono représente le premier point de contact des clients. Sprout permet de router les

[1] YAML est un format de représentation de données qui peut être utilisé pour la description et la configuration du réseau [101].

messages du type Session Initiation Protocol (SIP)[2]. Homestead est responsable de récupérer et stocker les informations des clients tels que les mots de passe des clients, ces informations sont stocker dans la base de données Cassandra. Finalement, ETCD permet le partage des informations de configuration entre les fonctions de *Clearwater*. Dans cet exemple le Docker Sprout est répliqué deux fois. Pour s'authentifier le client envoie une requête via Bono1. Dans notre exemple le Docker Bono1 a été arrêté, ce qui implique une erreur au niveau de la requête SIP avec un code "408" qui signifie un délai de demande écoulé sans réponse pour cette requête. En plus de cette erreur, un autre type d'alarmes est notifié au niveau de Sprout11 et Sprout12 qui n'arrivent pas a rejoindre Bono1.

Cette architecture est décrite dans un ficher YAML que l'algorithme d'auto-modélisation prend en entrée pour générer le graphe de dépendances illustré dans la figure 3-droite.

En injectant la même faute dans ce graphe de dépendances (c'est à dire que l'état du Docker Bono1 est défaillant $val(DC_Bono1_status) = Faux$), nous remarquons les mêmes symptômes de propagation de cette faute dans le graphe. C'est à dire que les états du service d'authentification $Register$ et de la connexion entre Bono1 et le cluster de Sprout1 $C_Bono1_Sprout1$ sont faux. Le modèle est par la suite utilisé comme entrée pour une procé-

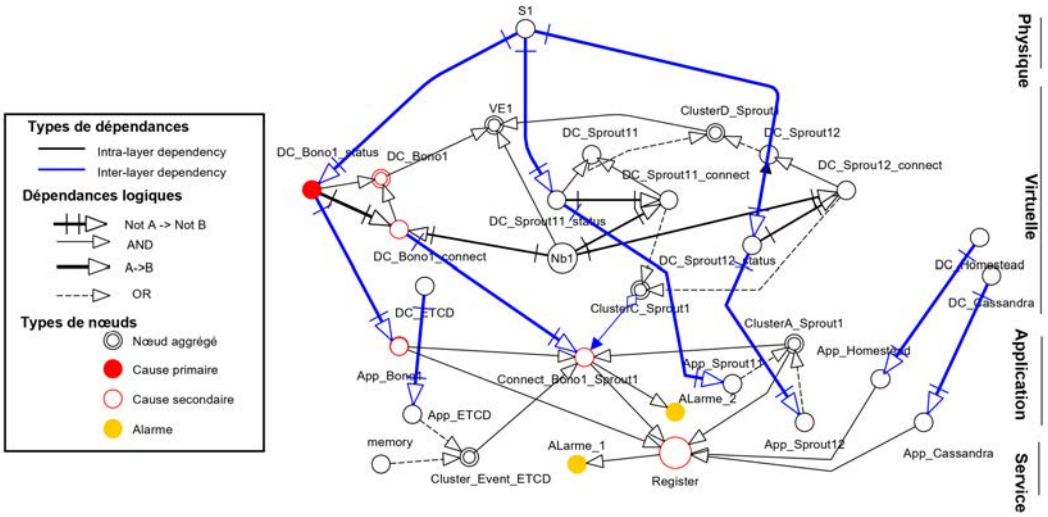


Figure 3 – Exemple de propagation de fautes.

dure de diagnostic qui se base sur le $solveur$ logique de Microsoft Z3 [83][3]. Une fois que la faute est détectée avec les premières observations (ou alarmes), les étapes du diagnostic actif sont les suivants:

1. Modéliser la topologie actuelle en graphe de dépendances global G.

[2]Session Initiation Protocol (SIP) est un protocole standard ouvert de gestion de sessions utilisé dans les communications multimédia.

[3]est un solveur **SAT**(problème de satisfaisabilité booléenne) développé par Microsoft

2. Extraire un sous graphe SG graphe de dépendance global G qui consiste en les nœuds connues des observations et les prédécesseurs directes (parents) des nœuds qui ont un état $Faux$.

3. Le graphe SG est par la suite traduit en un ensemble de contraintes logiques qui décrivent les noeuds et leurs dépendances.

4. Ces contraintes avec les observations sont par la suite proposées au $solveur$ pour obtenir une solution. La solution obtenue contient le minimum nombre de $Faux$ nœuds qui expliquent la faute.

5. Ces $Faux$ nœuds passent par la suite par des tests si ils sont testables. Selon la valeur obtenue ils sont classés en trois catégories :

   - *Innocent*: si le nœud est testable et fonctionne ($val(noeud) = Vrai$).
   - *Suspect*: si le nœud est non testable, ou testable et sa valeur est inconnue.
   - *Défectueux*: si si le nœud est testable et sa valeur est fausse.
   - *Coupable*: si en plus d'être défectueux le nœud coupable est une cause primaire de la panne.

6. Dans le cas innocent et suspect les valeurs de ces nœuds sont rajoutées aux observations.

7. Dans le cas défectueux pour que ce nœud soit une cause primaire il faut qu'il soit spontané (c'est à dire peut tomber en panne tout seul) et sans prédécesseurs.

8. Si la cause primaire de la faute n'est toujours pas retrouvée, le processus de diagnostic refait les mêmes étapes (2 à 7) avec une extension du graphe SG avec les nouvelles observations et les prédécesseurs directes des fausses observations. L'administrateur peut changer les valeurs des noeuds "Suspects" pour les rendre "innocents" ou "coupables".

Résultats expérimentaux:
Pour tester l'algorithme d'auto-modélisation et diagnostic, nous avons implémenté les deux algorithmes dans un projet hébergé dans le dépôt GitHub suivant: [119]. Pour cela, nous revenons a l'exemple de la Figure 3, les premières observations sont $O = \{Register, C_Bono1_Sprout1\}$. Le processus de diagnostic commence par extraire le sous graphe SG de ces nœuds et leurs prédécesseurs. Les solutions proposées par l'algorithme de diagnostic qui sont présentées dans la Figure 4, sont les suivants:

Scénario de faute: Arrêter le docker Bono1, $DcBono1S$:

---

1. Première solution: $Appbono1$
2. Solutions et tests suivants:
   - Test de l'application Bono1 ($val(Appbono1) = Inconnue$).
   - Test de la connexion du Docker ETCD1 ($val(DCE1C) = Vrai$).
   - Test de connexion du Docker Bono1 ($val(DCbono1C) = Faux$).
3. Résultats: la connexion Docker Bono1 ($DCbono1C$) est une cause possible. Continuer?: oui.
4. Solutions et tests suivants:
   - Test du mécanisme de connexion local de Docker: "Network bridge" ($val(NB1) = Vrai$).
   - Test de l'état du Docker Bono1 ($val(DCbono1S) = Faux$).
   - Résultats: L'état du Docker Bono1 ($DCbono1S$) est une cause possible. Continuer?: Non

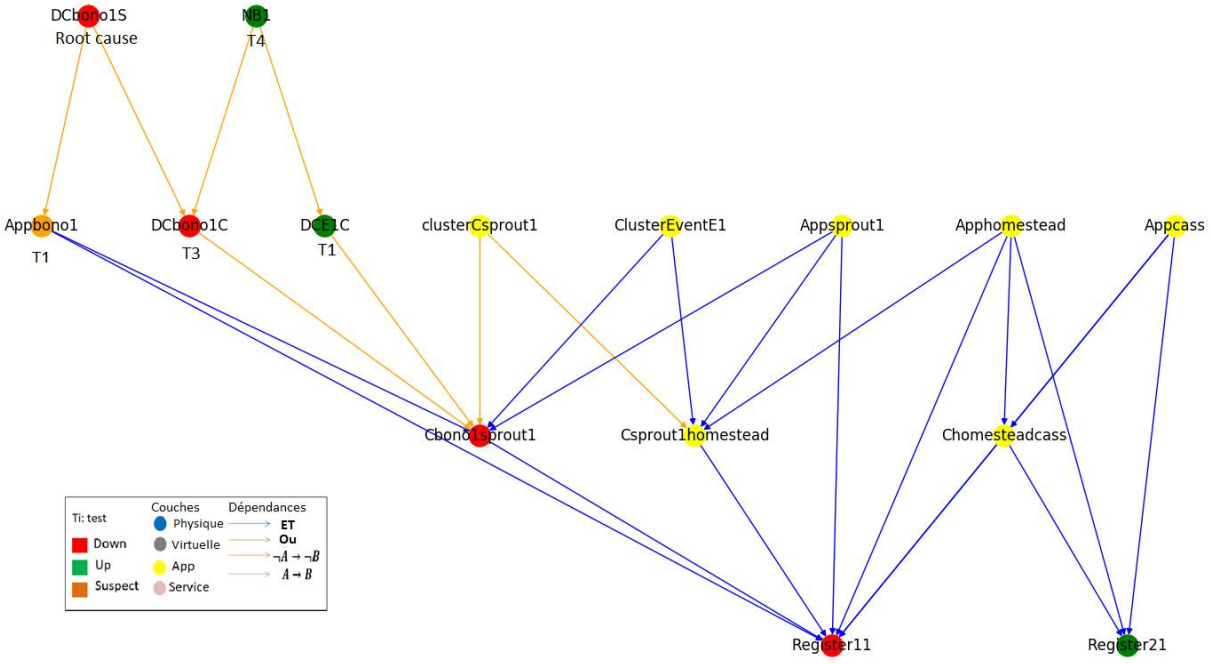


Figure 4 – . Résultat du diagnostic.

En passant par plusieurs tests, l'algorithme de diagnostic détermine la cause primaire $DCbono1S$ et retourne comme résultat le dernier sous-graphe avec les résultat des tests et la cause primaire et secondaire.

Conclusion et perspectives

La virtualisation des réseaux de télécommunications offre des services plus dynamiques et flexibles, tout en réduisant les coûts de mise en œuvre et de maintenance. En outre, et afin de bénéficier d'avantage de la virtualisation, les opérateurs partagent leurs ressources non occupées avec leurs clients. Néanmoins, ce paradigme de virtualisation engendre un certain nombre de défis, notamment en termes de fiabilité et de disponibilité, tels que: la topologie de réseau dynamique, hétérogénéité des composants du réseaux, manque de visibilité, le problème d'isolation de pannes des différents clients partageant une même infrastructure. Ces défis affectent les méthodes et approches classiques de gestion de pannes.

Pour répondre a ces défis, nous avons opté pour une approche de gestion de pannes basée modèle qui permet de donner des explications aux pannes pour une auto-réparation ciblée. Pour cela, nous avons d'abord proposé *LUMEN*, une plateforme qui se base sur des outils open source qui permet de centraliser, filtrer et récupérer les logs nécessaires pour l'approche de diagnostic. La centralisation et filtrage de logs permet de répondre au problème de manque de visibilité et d'isolation des logs clients. Nous avons par la suite défini une approche d'auto-modélisation pour les réseaux virtuels appliquée au use-case vIMS. Le modèle défini a été ensuite validé par des scénarios de propagations de pannes et testé sur un processus de diagnostic actif. Le processus de diagnostic se base sur des tests pour proposer des résultats sous forme d'un graphe de dépendances avec les nœuds innocents, suspects et fautifs. Le résultat proposé permet à l'administrateur de modifier certaines suppositions pour aller plus loin dans le diagnostic. Les résultats de la thèse ont ouvert plusieurs perspectives possibles.

Perspectives :

La plupart des étapes composant les procédures d'auto-modélisation et de diagnostic ont été automatisées. Cependant, d'autres étapes méritent d'être automatisées ou développées, tels que:

- L'auto-apprentissage de "templates": l'extension des "templates" ont été réalisés par un expert humain. Une perspective possible est de développer un algorithme d'auto-apprentissage capable de déduire les dépendances apprises et de corriger ou d'étendre ces "templates". Ces algorithmes appliqueront des scénarios d'injection de fautes à la fois dans le déploiement réel et dans le modèle défini. L'algorithme d'auto-apprentissage comparera ensuite les résultats de la propagation des défauts dans les deux cas.

- La création automatique du fichier YAML: dans notre travail, nous écrivons manuellement le fichier YAML, nous n'avons pas développé la procédure de génération de ce fichier à partir des différents déploiements. Cette procédure est facilement réalisable en interrogeant les orchestrateurs de réseau déployés tels que OpenStack ou Docker dans notre cas.

- l'automatisation de tests: Pour améliorer l'efficacité du processus de diagnostic proposé, une perspective possible est que le processus de diagnostic actif effectue les tests nécessaires sans interroger l'administrateur du réseau. Pour ce faire, l'algorithme de diagnostic actif doit être fourni avec les lignes de commande de test et les informations nécessaires sur les composants du réseau afin d'effectuer ces tests. Par exemple, pour un test de connexion, l'algorithme doit connaître l'adresse IP du nœud à tester.

En plus de ces perspectives, le graphe de dépendances logiques proposé peut être étendu en un graphe probabiliste pour rependre à d'autre types de fautes qui dégradent le bon fonctionnement du réseau, telle que la surcharge CPU d'un serveur hébergeant des fonctions virtuelles. Une autre amélioration possible pour le processus de diagnostic est de capturer le changement de topologie une fois le diagnostic lancé. Pour cela, l'algorithme d'auto-modélisation doit a chaque fois mettre à jour le graphe de dépendances selon les changements de topologie. Une étude que nous avons effectuée sur l'algorithme d'auto-modélisation a prouvé qu'il est plus efficace de modéliser seulement les changements enregistrés sur le ficher YAML au lieu de refaire tout le graphe. En plus de considérer les changements de topologie lors du diagnostic, le processus de diagnostic peut rajouter des actions d'auto-réparation pour avancer ses résultats.

Finalement, nous avons prouvé l'efficacité de notre approche d'auto-modélisation et diagnostic sur le use-case *Clearwter* vIMS, une perspective est de tester notre approche du d'autres use-cases en apprenant les dépendances spécifiques pour chaque use-case avec de l'injection de pannes.

TABLE OF CONTENTS

Résumé en Français **IV**

List of Figures **xxiii**

List of Tables **xxiv**

List of Templates **xxv**

1 Thesis Introduction **1**
- 1.1 Context . . . 1
- 1.2 Thesis objectives . . . 4
- 1.3 Thesis methodology and scientific contribution . . . 4
- 1.4 Thesis results . . . 6
  - 1.4.1 Publications . . . 6
  - 1.4.2 Proof of concepts: . . . 7

2 Programmable virtual networks **8**
- 2.1 Introduction . . . 8
- 2.2 Network virtualization ecosystems . . . 10
  - 2.2.1 Software Defined Networking (SDN) . . . 13
    - 2.2.1.1 SDN and Network Functions Virtualization (NFV) coexistence . . . 15
  - 2.2.2 Multi-tenancy and slicing . . . 16
  - 2.2.3 Virtual Hosting Environment . . . 17
  - 2.2.4 NFV Management and Orchestration (MANO) . . . 18
    - 2.2.4.1 Virtual Infrastructure Manager (VIM) . . . 20
    - 2.2.4.2 NFV networking . . . 21
    - 2.2.4.3 NFV fault and performance management . . . 22
- 2.3 A Virtual Network Function (VNF) chain: *Clearwater vIMS* . . . 24
  - 2.3.1 IP Multimedia Subsystem (IMS) . . . 25
  - 2.3.2 *Clearwater* Virtual IP Multimedia Subsystem (vIMS) . . . 26
- 2.4 Virtual Networks features and challenges . . . 29
  - 2.4.1 Virtual networks dependability . . . 30
  - 2.4.2 Physical and virtual coexistence . . . 31

2.4.3 Dynamicity of the network topology . . . 31
2.5 Conclusion . . . 31

3 A proposal for a comprehensive fault management survey 33
3.1 Introduction . . . 33
3.2 Classical approaches to fault management overview . . . 34
3.2.1 Fault detection and observation collection (logging) . . . 34
3.2.1.1 Data Mining for detection . . . 35
3.2.2 Fault localization and identification . . . 37
3.3 Black-box approaches . . . 40
3.3.1 Rule-based and case-based reasoning . . . 40
3.3.2 Decision trees . . . 43
3.3.3 Neural Network (NN) . . . 46
3.4 White-box approaches . . . 50
3.4.1 Causality/dependency graphs . . . 50
3.4.2 Constraint graphs . . . 55
3.4.3 Bayesian networks . . . 61
3.4.4 Petri nets . . . 66
3.5 Limitations of classical fault management approaches . . . 71
3.6 Novel fault management techniques . . . 74
3.6.1 Self-modeling . . . 74
3.6.2 Self-healing . . . 76
3.6.3 Fault injection . . . 80
3.6.4 Discussion . . . 81
3.7 Positioning of the thesis work . . . 82

4 LUMEN: a global fault management framework 84
4.1 Introduction . . . 84
4.2 LUMEN global fault management framework . . . 84
4.2.1 LUMEN Planes . . . 86
4.3 Virtual Networks data . . . 89
4.4 Application of *LUMEN* to the vIMS use case . . . 90
4.4.1 Docker *Clearwater* vIMS deployment . . . 90
4.4.2 Traffic generation . . . 90
4.4.3 Fault injection procedure . . . 90
4.4.4 Logs collection and filtering . . . 92
4.5 Conclusion . . . 92

5 Self-modeling **93**
5.1 Introduction . . . 93
5.2 Self-modeling approach overview . . . 93
5.3 Modeling rules derivation . . . 94
5.4 Real world application: *Clearwater* vIMS self-modeling . . . 104
5.4.1 *Clearwater* vIMS templates description . . . 105
5.5 Self-modeling algorithm . . . 111
5.5.1 Procedure for listing registration services . . . 115
5.5.2 The dynamicity of topologies . . . 117
5.6 Validation of the vIMS model . . . 118
5.6.1 Templates validation . . . 120
5.6.2 Validation of the model through a fault propagation use-case . . . 122
5.7 Experimental evaluation of the self-modeling algorithm . . . 124
5.7.1 Evaluation of the dependency graph scalability . . . 124
5.7.2 Evaluation of the self-modeling algorithm performance . . . 126
5.8 Summary and conclusion . . . 128

6 Active Fault Localization **129**
6.1 Introduction . . . 129
6.2 Definitions and notations . . . 129
6.2.1 Global dependency graph and sub-graph . . . 129
6.2.2 Diagnosis engines . . . 131
6.3 Diagnosis in a sub-graph . . . 134
6.4 Active diagnosis process . . . 136
6.5 Discussion . . . 140
6.6 Experimental results . . . 142
6.6.1 Diagnosis of faults with identical symptoms . . . 143
6.6.2 Summary . . . 147
6.6.3 Diagnosis of multiple faults . . . 148
6.6.4 Summary . . . 151
6.7 A qualitative comparison with related works . . . 153
6.7.1 A qualitative comparison with classical telecommunications monitoring routines . . . 153
6.7.2 A qualitative comparison with black-box approaches . . . 153
6.7.3 A qualitative comparison with model-based approaches . . . 154
6.8 Conclusion . . . 154

7 Conclusions and future work **156**
7.1 Results obtained during the thesis 156
7.2 Perspectives 158
7.2.1 Automation of other steps composing the self-modeling and diagnosis procedures 158
7.2.2 Probabilistic dependency graph 159
7.2.3 Dynamic diagnosis 160
7.2.4 Application of the self-modeling approach to other use-cases 160

Appendix A **161**

Appendix B **164**

Appendix C **170**

List of Acronyms **171**

Bibliography **177**

LIST OF FIGURES

1 Les étapes et approches de gestion de pannes. VII
2 La platforme *LUMEN*. IX
3 Exemple de propagation de fautes. XI
4 . Résultat du diagnostic. XIII

2.1 Network Virtualization Ecosystems [151][51] 10
2.2 Network Function Virtualization vision [151][51] 11
2.3 Network Functions Virtualization 13
2.4 SDN layered architecture. 14
2.5 An example of NFV and SDN coexistence. The routing of VNFs traffic is enabled by SDN. The SDN virtual switches and the controller are hosted in virtual hosts. 15
2.6 Multi-tenant cases for NFV Software as a service (SaaS). In case 1 tenant-A and tenant-B have different VNFs, while in case 2, they are sharing the same VNFs with two different slices. 16
2.7 A simplified view of VNF deployment on virtual machines, containers, and unikernels [89]. Containers are more lightweight for hosting VNFs than unikernels and Virtual Machines (VMs). 18
2.8 European Telecommunications Standards Institute (ETSI) NFV Architectural Framework [34]. 19
2.9 (a): Three containers sharing the same host and communicating through a network bridge "Docker0", (b): Two hosts communicating through an overlay network. 22
2.10 Fault Correlation schemes in the NFV Architecture, as represented by the ETSI Group Specification (GS) [62]. This architecture illustrates four local fault correlators and three external fault correlators. 23
2.11 Core IMS architecture. 25
2.12 *clearwater* vIMS architecture [19]. 27
2.13 Virtual network Components. 30

3.1 The Fault Management Process. 33
3.2 Fault localization approaches. 39
3.3 Black-box vs Model-based Machine Learning (ML) methods. 40
3.4 (a) Rule-based [23] and (b) case-based reasoning [1]. 41
3.5 *Vitrage* switch alarm use case. 42

3.6 a decision tree sample composed of class "1" and class "0". 44
3.7 Deep Neural Network (DNN) with two hidden layers. 47
3.8 An artificial neuron composition. 47
3.9 Relation between the network data matrix X and the fault label Y [153]. 49
3.10 Causal dependency graph components [76]. 51
3.11 An example of a client IMS Public User Identities (IMPU) verification causal graph after stabilization of nodes states [76, 77]. 53
3.12 An example of a logical causality graph. [76, 77]. 54
3.13 Different ways to represent constraint graphs [26]. 57
3.14 The join-tree clustering procedure. 59
3.15 A constraint network. 60
3.16 Example of an overlay network model [44]. 61
3.17 Belief Propagation (BP) on a tree [70]. 62
3.18 A Bayesian representation of the active redundancy scheme (left) and the Conditional Probability Table (CPT) of node S (right) [71]. 63
3.19 An example of a Generic Bayesian Network (GBN) pattern (left) instantiated three times (right) [60]. 64
3.20 Two users Bayesian Network (BN) instances for IP configuration procedure sharing resources [60]. 65
3.21 An example of a controller failure BN dependency graph [147]. 66
3.22 An example of a Petri net with four transitions. 67
3.23 Spontaneous and persistent faults modeled by Petri nets [10]. 68
3.24 A signal loss failure propagation through port P of SMT-1 [10]. 69
3.25 A signal loss failure propagation through port P of SMT-1 [10]. 69
3.26 Petri nets modeling propagation of faults resulting from a Loss Of Signal (LOS) and the alarms that results from these faults [10]. 69
3.27 Connectivity failure Petri net model [145]. 70
3.28 Multi-tenant's Fault notification ML methods. 72
3.29 Top left: tree topology (D=2, F=2) and the sub-tree D composed of the switch (Sw2) and hosts (H1, H2). Top right: The sub-tree D network (Nt) and (Lt) descriptor. Down the corresponding dependency graph [147]. 76
3.30 *Vitrage* architecture [146]. 77
3.31 The procedure of adding a VM instance-3 before (1) and after (2) [146]. 79
3.32 Open Platform NFV (OPNFV) Server crash use case. 80

4.1 *LUMEN* framework composed of four layers (source, sink, extraction and decision). 85
4.2 Elastic Stack Logs transformation 88

5.1 Model construction and self-modeling Process . . . 94
5.2 Example of assembling templates instances. . . . 96
5.3 Possible dependencies and their associated assumptions . . . 97
5.4 Virtual networks global architecture. . . . 98
5.5 Network elements templates. . . . 99
5.6 Network connections templates. . . . 100
5.7 Aggregated nodes template. . . . 101
5.8 Auto-recovery ($T.AR$) and elasticity ($T.EL$) templates. . . . 102
5.9 Inter-layer dependencies. . . . 103
5.10 vIMS two sites architecture use-case . . . 104
5.11 *Clearwater* vIMS Network elements templates. . . . 105
5.12 BONO and Sprout application templates ($T.APP_Bono$, $T.APP_Sprout$). . . . 106
5.13 The first Session Initiation Protocol (SIP) registration trial to the *Clearwater* vIMS [19]. . . . 107
5.14 The second SIP registration trial to the *Clearwater* vIMS [19]. . . . 107
5.15 vIMS SIP register service modeling. . . . 108
5.16 *clearwater* vIMS network connections templates. . . . 109
5.17 *Clearwater* vIMS network elasticity modeling. . . . 110
5.18 *Clearwater* vIMS aggregated nodes templates ($T.An_S, T.An_{Vi}$). . . . 110
5.19 Inter-layer dependencies. . . . 111
5.20 Inter-layer dependencies with elasticity. . . . 111
5.21 The Yet Another Markup Language (YAML) file for the architecture depicted in Figure 5.10. . . . 112
5.22 a vIMS architecture use-case composed of three sites. . . . 116
5.23 The Network Connectivity Topology (NCT) graph for the vIMS architecture of Figure 5.22 . . . 116
5.24 The register services paths in the NCT graph . . . 117
5.25 Modeling of register services and their dependencies . . . 117
5.26 Topology network update captured in the YAML file: (left) before and (right) after the update. . . . 118
5.27 The dependency graph of the YAML file in Figure 5.26 (before (a) and after (b) the update). . . . 119
5.28 The correction of the distant logical connectivity template. . . . 122
5.29 *Clearwater* vIMS fault injection and propagation use-case. . . . 123
5.30 The self-modeling resulted dependency graph that illustrates the fault propagation of the fault injection use-case in Figure 5.29. . . . 123
5.31 Access and control sites. . . . 125

5.32 Evolution of the Number of vertices according to the number of the deployed sites and VNFs. . . . 126
5.33 Evolution of the Number of dependencies according to the number of the deployed sites and VNFs. . . . 127
5.34 Evolution of the execution time according to the number of the deployed sites and VNFs. . . . 127

6.1 An example of a sub-graph representing the nodes V-1 and V-2 failures and their predecessors . . . 135
6.2 Active Diagnosis process flowchart . . . 139
6.3 An example of a dependency graph with 10 nodes and two initial observations $obs(v_1) = False$, $obs(v_5) = True$ and the node v_2 stated as suspect. . . . 140
6.4 The first method applied to the example of Figure 6.3. . . . 141
6.5 The second method applied to the example of Figure 6.3. . . . 142
6.6 Self-modeling and RCA Framework . . . 143
6.7 Experimental vIMS architecture . . . 144
6.8 . The initial observations sub-graph for Scenarios I, II and III. . . . 145
6.9 . Result sub-graph of fault scenario I . . . 146
6.10 . Result sub-graph of fault scenario II . . . 147
6.11 . Result sub-graph of fault scenario III . . . 148
6.12 . Initial observations generated sub-graph of multiple fault scenario in Bono1 and Homestead virtual connections. . . . 149
6.13 . Results of execution I. . . . 150
6.14 . Results of execution II, detecting the first root cause $DCbono1C$. . . . 151
6.15 . Results of execution II, detecting the second root cause $DChomesteadC$. . . . 152

1 The procedure of data modification in a data base. . . . 162
2 Log $\vec{tr}$ and the corresponding specifications [47]. . . . 163
3 The counterfactual scenarios to the failure of client (a) and journal (b) [47]. . . . 163
4 A deployment of Docker*Clearwater* vIMS. . . . 164
5 The Ellis dashboard screenshot where two clients were created. . . . 166
6 A connected Jitsi client to the "example.com" base trying to call the client with the SIP number 6505550223. . . . 167
7 Three tests generated from the *Clearwater-live-test* applied to the *Clearwater* deployment. The green "Passed" means that the test is successful. . . . 167
8 The command line for stopping the Sprout Docker. . . . 168
9 The *Weave Scope* screenshot after stopping the Sprout Docker. . . . 168

LIST OF TABLES

2.1 ETSI and IETF Notations For virtual functions [80]. 11
2.2 Traditional vs virtual network architecture features. 29

3.1 Example of a Join relation $\phi_1 \bowtie \phi_2$. 56
3.2 Advantages, disadvantages and limitations of classical fault localization approaches to face virtual networks challenges. 73
3.3 Taxonomy of faults in NVE 78

4.1 Acquired and learned knowledge 89
4.2 Overview of the fault injection model in each layer: (1) Physical, (2) virtual, (3) application and (4) service. 91

5.1 Template extension and correction 121
5.2 Number of VNFs in each topology scheme. 125

6.1 A qualitative comparison of *SAKURA* with black-box approaches. 153
6.2 A qualitative comparison of *SAKURA* with model-based approaches 154

1 Two log lines generated by the *Clearwater* components. The first part of the log represents the time and date of the alarm . The second part is the type of alarm (i.e. "Status" and "Warning") with a specific identification defined in the alarming program of *Clearwater* such as "pluginloader.cpp:150" in the first log line. The last part of the log line is a message describing the alarm. 169

LIST OF TEMPLATES

Network Elements Templates:

$T.site$	Physical site
$T.VH$	Virtual Host
$T.APP$	Application
$T.service$	Service

Network Connections Templates

$T.PC$	Physical Connectivity
$T.LVC$	Local Virtual Connectivity
$T.DVC$	Distant Virtual Connectivity
$T.LLC$	Local logical Connectivity
$T.DLC$	Distant logical Connectivity

Aggregated Templates

$T.An_S$	Aggregated node for sites
$T.An_{V_i}$	Aggregated node for virtual hosts

Auto-recovery and Elasticity Templates

$T.AR$	Auto-recovery
$T.EL$	Elasticity

Inter-layer Template:

$T.Inter$	Inter-layer dependencies

CHAPTER 1

THESIS INTRODUCTION

1.1 Context

With the advent of 5G, present day Mobile Network Operators (MNOs) are occupied with a large and growing range of proprietary hardware appliances and complex communication protocols and architectures that need to transition to a software model for network functions design [55]. Consequently, adding a new network service often brings the need to deploy new hardware servers, maintaining them and eventually replacing the malfunctioning servers when they ultimately fail in order to keep the services running correctly.

Moreover, being one of the fastest growing sectors, the telecommunication industry is facing increased competition as new players are entering with emerging software technologies and open source projects. One such project is the recent *Telecom Infra Project* [137], initiated by Facebook and oriented towards an open source and general purpose hardware for a new generation of MNOs. This approach to infrastructure is radically different from what the telecommunication industry is used to. This has lead MNOs to invest in and adopt new technologies such as Software Defined Networking (SDN), Network Functions Virtualization (NFV), cloud infrastructures, analytic and live monitoring technologies. Therefore, operators are investing a lot of energy in order to virtualize their hardware functions. For instance, Orange and AT&T are joining open source software communities hosted by the Linux Foundation such as OpenStack for infrastructure [105], OpenDayLight for SDN controllers [100], and Open Network Automation Platform (ONAP) for network automation [97].

Software-based innovative technologies should be integrated in the design of the upcoming 5G networks [66, 5]. 5G Infrastructure Public Private Partnership (5GPP) projects such as *norma* [95], advocate for API-driven architectural openness, fueling economic growth through over-the-top innovation. These projects propose adaptive decomposition and allocation of mobile network functions, which flexibly decomposes the mobile network functions and places the resulting functions in the most appropriate location. This comes down to defining a service-oriented architecture for the 5G networks, which is disruptive in comparison to how the Long Term Evolution (LTE) network has been designed and implemented. In this software service-oriented paradigm, each service is implemented as a Virtual Network Function (VNF) defined at ETSI [4] or as a Service Function (SF) defined at the Internet Engineering Task Force (IETF)

[40]. These VNFs are chained or connected to each other to achieve an end-to-end 5G service.

Virtual IP Multimedia Subsystem (vIMS) is a representative example of the virtualization of network functions. IMS is the standard architecture that has been adopted by telecommunication operators for the IP-based voice, video and messaging services, replacing legacy circuit-switched systems [21]. The virtualization of IMS brings a number of benefits inherited by the virtualization of network functions: the flexibility and the programmability of services and the optimization of deployment cost and time. Thanks to the virtualization of IMS, operators will be able to propose customized vIMS solutions for their clients. The clients will be able to deploy a vIMS network in just few seconds. Moreover, NFV offers the flexibility to scale up and down the services or reconfigure the network in response to the traffic demand and network state.

In addition to the software-oriented paradigm of the 5G architecture, MNOs are seeking new revenue sources and models. One way proposed by the Next Generation Mobile Networks (NGMN) forum [116], is to break the traditional business model of a single network infrastructure ownership and introduce network-sharing or multi-tenancy. This approach has the potential to recover up one fifth of the estimated operational costs for MNOs [124]. For instance, in 2010 Chinese telecommunication operators saved an investment of more than 12 billion dollars by sharing their network infrastructure [75].

Network slicing is another alternative for sharing MNO Network Services (NSs). A slice represents an isolated end-to-end tunnel tailored to achieve the requirements requested by a particular service. Each client slice has specific requirements such as the network bandwidth.

In order to ensure the high availability and reliability of network services, one more important aspect that MNOs cannot neglect is the management of Network Virtualization Ecosystems (NVEs). The management procedures are often summarized as Fault, Configuration, Accounting, Performance, and Security (FCAPS) management. Fault management represents one of the most important X-management axes since a failure can cause important losses and impact the business image and credibility. One example of recent cloud failures has been registered in 2013 within IBM. The monetary loss was estimated to 31 million Australian dollars and it was due to IBM servers breakdown that brought down the website of major Australian retailer "Myer" for one week during the Christmas period [13].

Fault management is divided into four major steps, namely: fault detection, identification, localization and recovery. Fault detection is the step where the network is stated to be in a faulty behavior. Fault identification identifies the type of faults and the affected components, fault localization or Root Cause Analysis (RCA) is the procedure of pinpointing the root cause i.e. the node (or nodes) responsible for the fault and finally recovery or healing is the procedure of resorting normal working operations on the network.

In telecommunication, fault management is a well known field with many deployed techniques often involving the Simple Network Management Protocol (SNMP) and software tools

such as Nagios [87]. However, virtual networks brings new challenges and issues that change the way fault management is performed. The NVE features, such as the dynamic network topology, highlight the need to rethink the management procedures and adapt them to these environments.

The **main challenge (Ch0)** introduced by the virtualization of telecommunication networks is the automation of the fault management steps. In fact, the primary goal of network virtualization is to automate network operations to remove the need for humans to manually deploy, reconfigure and heal network services. The automation of recovery brings the necessity to accurate Root Cause Analysis (RCA) or fault localization so that the healing actions target the correct erroneous components.

A taxonomy of fault localization approaches and techniques were proposed for the fault management of networks. Model-based techniques are fault localization approaches that provide explanations about faults and alarms propagation represented in an explicit model. They use a model defined from the network components dependencies for diagnosis. However, building a model in virtual networks face a number of challenges and issues. The **first challenge (Ch1)** is the **dynamic** network topology. The flexibility of virtual networks and the dynamicity of VNFs in the network engender continuous changes of the network topology. The self-modeling approach should adapt to the network topology changes to provide correct localization results.

A **second challenge (Ch2)** is the **multi-layered** aspect of virtual networks. The NFV architecture is divided into four layers: the physical, the virtual, the application and the service layer. The *physical layer* includes the physical servers, the *virtual layer* contains virtual hosts such as VMs, the *application layer* is composed of the software running on the virtual hosts and finally the *service layer* with the corresponding protocols. The multi-layered aspect of virtual networks introduced the necessity of a multi-level management process. In fact the defined diagnosis model should consider each network level and the dependencies between layers (i.e. inter-layer dependencies).

The **third challenge (Ch3)** is learning the novel dependencies from virtual networks that are not described in the specification of telecommunication networks services such as the virtual and physical dependency. This kind of dependencies should be included to the defined model.

The **fourth challenge (Ch4)** is the granularity level of the RCA process. In fact, virtual networks introduced a heterogeneity of components including the virtual and the physical hosts and links, and a more smaller granularity such as the host Central Processing Unit (CPU), storage and network. The challenge is to chose the correct granularity level for the diagnosis process and to refine the granularity if necessary.

The **fifth challenge (Ch5)** is the ambiguity and consistency of data. In fact, the scalability and diversity of services in virtual networks will enable enormous types of data including logs and metrics, service description files and topology information. Moreover, each service rep-

resents a chain of connected VNFs hosted in distinct locations. Therefore, logs of the same service will be distributed in distinct locations. This will create another challenge **(Ch6)** which is the lack of visibility. In fact, the logs of the same service should be centralized to enable a global view of faults propagation and their alarms. Furthermore, VNFs of each MNO may be shared between different clients or tenants. This will create the necessity to separate the tenants logs to isolate tenants faults and inform them **(Ch7)**.

1.2 Thesis objectives

The aim of the thesis is to propose an accurate autonomic fault diagnosis procedure that pinpoints root cause(s) and provides explanation of faults propagation to ensure a targeted self-healing actions. The challenges presented above brought a number of requirements for the diagnosis system, that represents the objectives of the thesis :

1. **Challenges (Ch5, 6 and 7)** implies the necessity to propose a procedure to prepare data for the diagnosis process. This procedure aims at collecting, filtering and centralizing data. In addition to tenants faults isolation.

2. When applying a model-based approach for diagnosis, the defined model should adapt to the dynamicity of network topology. We need to establish a self-modeling methodology that tracks the dynamic network topology **(challenge (Ch1))**. The self-modeling approach should include most of the virtual network dependencies, we should find a way to acquire the model knowledge **(challenge (Ch3))**. The self-modeling approach should also consider the multi-layer aspect of virtual networks and the different granularity levels **(challenges (Ch2 and Ch4))**.

3. Once the self-modeling approach is defined, we should apply it to a real world use-case and prove it efficiency through a diagnosis process.

4. Finally, the automation of each procedure is necessary to keep the flexibility and programmability aspects of virtual networks **(challenge (Ch0))**.

1.3 Thesis methodology and scientific contribution

To answer to the thesis objectives, the adopted methodology was the following: first we address the state of the art in both virtual networks ecosystem and fault management techniques to identify the challenges and requirements of fault management in virtual networks. After identifying the main challenges that we wanted to respond to (i.e.**Challenges CH0 to Ch7**), the

second step was to choose an appropriate use case that includes all the features of a telecommunication virtual network. After that we proposed a experimental test-bed composed of open source tools for collecting and filtering data to learn about the chosen use case. The last task was to define an appropriate self-modeling and diagnosis approach. The main contributions of the thesis are presented as follows:

The ***first contribution*** of the thesis is a comprehensive survey of the state-of-the-art fault management techniques and approaches and the impact of the virtualization of network functions on fault management. In the first contribution, we also address the academic and open source solutions for the fault management of virtual networks. We propose a new classification of the current efforts that address fault localization challenges, and compare their major contributions and shortcomings.

In the ***second contribution***, we propose a global fault management framework *LUMEN*. The *LUMEN* framework proposes the automation of the data collection and filtering of virtual networks. *LUMEN* also provides tools to separate between tenants data. The ***second contribution*** answers to challenges **Ch0** and **Ch5, 6 and 7**.

In the ***third contribution***, we proposed an experimental test-bed to learn about the chosen use case (i.e. vIMS). We find it interesting to consider vIMS as a case study since it shows many aspects of future virtual telecommunication architectures such as the multiple layers, the elasticity and the dynamic topology. To define our model, we use two kind of knowledge: the acquired knowledge that we get from service description files and the learned knowledge. The learned knowledge is acquired by the fault injection procedure. In this procedure, we inject faults (e.g. stop a VM) and try to learn the dependencies between network components that are not easy to extract from the acquired knowledge. In the experimental test-bed traffic and fault injectors tools are applied. The ***third contribution***, answers the to the third challenge **Ch3**.

The ***fourth contribution*** consists on proposing a self-modeling approach for an end-to-end virtual service chain. The proposed model is a dependency graph constituted of Boolean variables with logical dependencies. The model is a multi-layer, multi-resolution graph that represents the network from the physical to the application level. The model can be refined to a smaller granularity level such as application's processes. This contribution addresses the second objective of the thesis that responds to challenges **Ch2 and Ch4**.

The ***fifth contribution*** of the thesis consists on a self-modeling algorithm to face the dynamic network topology challenge **Ch1**. The self-modeling algorithm instantiates a dependency graph using the defined modeling rules that fits to the current topology using a description file.

Finally, in the ***sixth and last contribution***, we propose an active diagnosis process to pinpoint the faulty network component(s). The active diagnosis process considers the logical

assumptions defined in the model to infer the root cause(s). The diagnosis process is oriented through the additional observations provided from tests. The result is a dependency graph that indicates the status of components and the faults propagation.

These contributions are presented as follows:

- Chapter 2 entitled *“programmable virtual networks"*, presents the virtual network ecosystem and the NFV concepts and features. In this Chapter 2, we also depict the *Clearwter* vIMS use case.
- Chapter 3 (**first contribution**) entitled *“fault management"*, discusses the classical fault localization approaches and their limitation. This Chapter 3 also addresses the recent fault management techniques applied to virtual networks and positions the thesis work with regards to the related work.
- Chapter 4 (**second contribution**) entitled *“LUMEN global fault management framework"* depicts the global fault management framework *LUMEN* and the applied open source tools in our test-bed.
- Chapter 5 (**third, fourth and fifth contributions**) entitled *“self-modeling"*, presents the self-modeling approach overview and the application of the self-modeling approach on the vIMS use case. This chapter also presents the procedure of validating and extending the model and depicts the self-modeling algorithm.
- Chapter 6 (**sixth contribution**) entitled *“active fault localization"*, describes the active diagnosis process and the results of the application of this process on real world faults.
- Chapter 7 *“conclusions and future work”*, outlines directions for future work, and concludes the dissertation.

1.4 Thesis results

This work has produced the following publications so far:

1.4.1 Publications

International conferences:

- **S. Cherrared**, S. Imadali, E. Fabre and G. Goessler, "LUMEN: A global fault management framework for network virtualization environments," 2018 21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), Paris, 2018, pp. 1-8.

- **Cherrared S**, Imadali S, Fabre E, Goessler G. “Data transformation model for the fault management of multi-tenant network”. 27th European Conference on Networks and Communications (EUCNC), Ljubljana, 2018.
- **S. Cherrared**, S. Imadali, E. Fabre, and G. Gössler. 2019. SAKURA a Model Based Root Cause Analysis Framework for vIMS (poster). In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys ’19). ACM, New York, NY, USA, 594-595.

International Journals:

- S. Cherrared, S. Imadali, E. Fabre, G. Gössler and I. G. B. Yahia, "A Survey of Fault Management in Network Virtualization Environments: Challenges and Solutions," in IEEE Transactions on Network and Service Management, vol. 16, no. 4, pp. 1537-1551, Dec. 2019.

1.4.2 Proof of concepts:

- Monitoring the *Clearwater* vIMS docker logs and metrics with the Elastic stack [85]: this project hosts the proposed *LUMEN* framework applied to *Clearwater* vIMS.
- A Model Based Root Cause Analysis Framework for vIMS [119]: an experimental framework that contains the self-modeling and the active diagnosis process proposed in the thesis.

CHAPTER 2

PROGRAMMABLE VIRTUAL NETWORKS

2.1 Introduction

Nowadays, most traditional network infrastructures are composed of various proprietary hardware appliances such as routers, load balancers and firewalls. Consequently, adding a new network service often brings the need to deploy new devices, maintaining them and eventually replacing the malfunctioning servers when they ultimately fail in order to keep the services running correctly. This procedure often associated with the network function life cycle, has a high cost and is energy consuming. Moreover, buying new servers to replace the failed ones may take a long time (i.e. months) before the demand is satisfied. Another often underestimated part of the MNOs work, is the time spent in different standardization bodies, such as *Telecommunication Standardization Sector of the International Telecommunications Union (IUT-T)*, *European Telecommunications Standards Institute (ETSI)*, *IETF*, or recently *NGMN*, to ensure compliance among the proprietary hardware devices and communication protocols.

The Network Functions Virtualization (NFV) concept was proposed by the ETSI aiming at addressing the drawbacks of using physical appliances in network service provisioning [51]. *Virtualization* or *softwarization* was first introduced by IBM with the concept of virtual machines in the 1960s [88]. In telecommunications, NFV intends to decouple software components of network functions from their respective dedicated hardware, resulting in Virtual Network Functions (VNFs) [151]. Software-based innovative technologies bring a number of advantages to MNOs [51]:

- **Optimization of the deployment costs and time**: NFV reduces time-to-market by minimising the classical network operator cycle of deploying new network services. Moreover, energy and monetary economies are made regarding the investments in hardware-based functionalities that are no longer applicable for software-based networks. Time of deploying new functions is reduced.
- **Increasing the scalability**: NFV allows MNOs to dynamically manage the life cycle of virtual functions aiming at increasing the scalability. VNFs are scaled according to the clients' demands at run-time.

- **Programmability and flexibility of services**: VNFs being simple programs enable flexible reconfiguration in response to changes in the network state. NFV allows immediate technology adoption and the flexibility on the design of new service features. Having the ability to set up and reconfigure network services on the fly enables continuous deployment and integration.

- **Resource sharing**: Sharing their non-occupied physical resources with clients is highly profitable for MNOs and allows for better resource utilization. Moreover, MNOs enable clients to share the same VNFs with isolated traffic which maximizes the flexibility of 5G networks, optimizing both the utilization of the infrastructure and the allocation of resources. This sharing procedure will also enable greater energy and cost efficiencies compared to earlier mobile networks.

To illustrate the benefits made by the virtualization of networks, suppose that a telecommunication service provider wants to virtualize a number of its physical functions. The procedure is to deploy these functions as software hosted on a virtual host in physical servers. The services definitions are done in a description file to make the deployment flexible and reproducible. If the number of client demands increase, instead of deploying a new hardware appliance across the network, the service provider can deploy the same service using the service description on any server already in the network, provided that the underlying hardware has the capacity to support the additional workload. Moreover, the service provider can share its non-occupied hardware servers and create additional revenues. At any time, the provider can instantiate, replicate, migrate or terminate a service according to the network traffic demand. In addition, the service provider can propose specific Service Level Agreement (SLA) to the client's service, e.g., if the client is looking for more CPU capacity.

Figure 2.1 represents a high-level overview of the Network Virtualization Ecosystems (NVEs). It illustrates the vertical and horizontal views of the related entities in a composed per-tenant 5G service. The multi-layered architecture in Figure 2.1 includes a number of coexistent ecosystems and concepts, namely: Software Defined Networking (SDN), NFV, multi-tenancy and slicing. These virtual technologies are enabled by the virtualization layer including the Virtual Infrastructure Manager (VIM) that manages virtual hosts. In this chapter, aiming at providing a clear understanding of the challenges inherent to the fault management in NFV, we explain in more details the entities composing the ecosystem of virtual networks. We depict the composition of an NFV chain supporting network services, and the composition of the infrastructure enabling network function virtualization. We discuss the NFV features and scientific challenges of virtual networks.

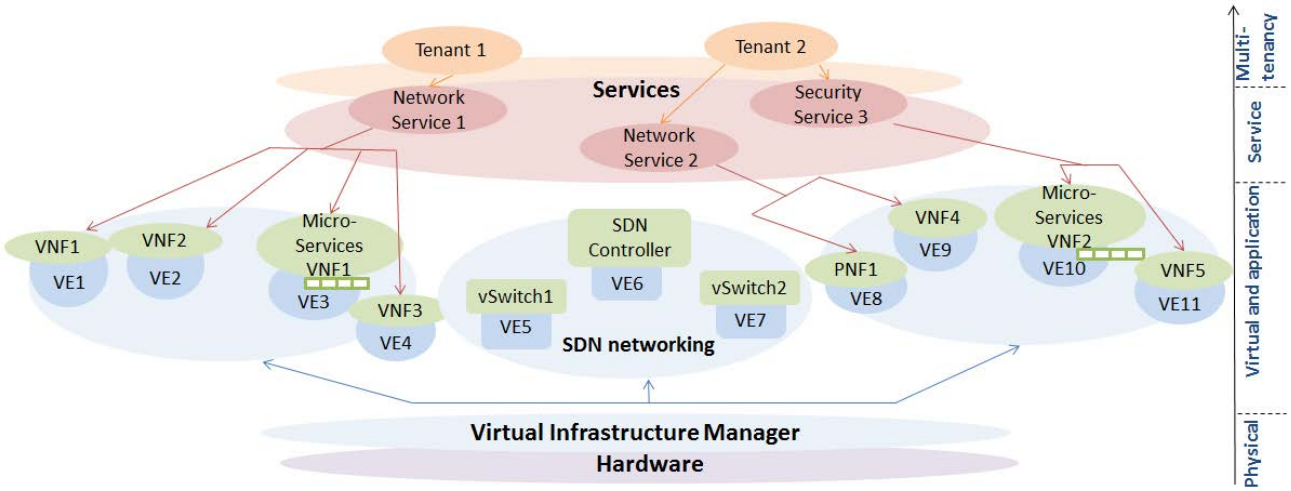


Figure 2.1 – Network Virtualization Ecosystems [151][51]

2.2 Network virtualization ecosystems

The softwarization of networks enabled several network virtualization testbeds and diverse network architectures to coexist in a single infrastructure without affecting production services [32]. NFV is the concept of transforming pure hardware appliances hosting Network Functions (NFs), (e.g. Network Address Translation (NAT), firewall, intrusion detection and Domain Name System (DNS)), into software functions hosted on hardware servers. These functions are decoupled from the underlying hardware and known as Virtual Network Functions (VNFs). Figure 2.2, based on [**ETSI-NFV-start**], presents examples of network functions transition from dedicated hardware to logical functions. For instance, the firewall network function is decoupled from its dedicated physical server, to be run as a separate virtual appliance hosted on a virtual host. Compared to the traditional physical firewall, the virtualization brings the ease of migration. A virtual firewall may fit smaller network architectures with flexible upgrade and maintenance actions.

The ETSI and IETF standard organization bodies of the telecommunication industry created two working groups: the ETSI-NFV and IETF-Service Function Chain (SFC), respectively. The ETSI-NFV group is devoted to high-level standardization of interfaces among components and sub-systems, while the IETF-SFC focuses on routing and data-plane protocol specification (e.g., NSH). The ETSI-NFV and IETF-SFC working groups defined network services as a VNF Forwarding Graph (VNF-FG) or Service Function Chain (SFC), respectively [35, 73, 53]. Each of these groups has its own terminologies but there is a certain similarity between their NFV architectures. Table 2.1 depicts the correspondence between terminologies to define NFV services [80]. Note that most of the applied terminologies in this thesis are those of the ETSI-NFV

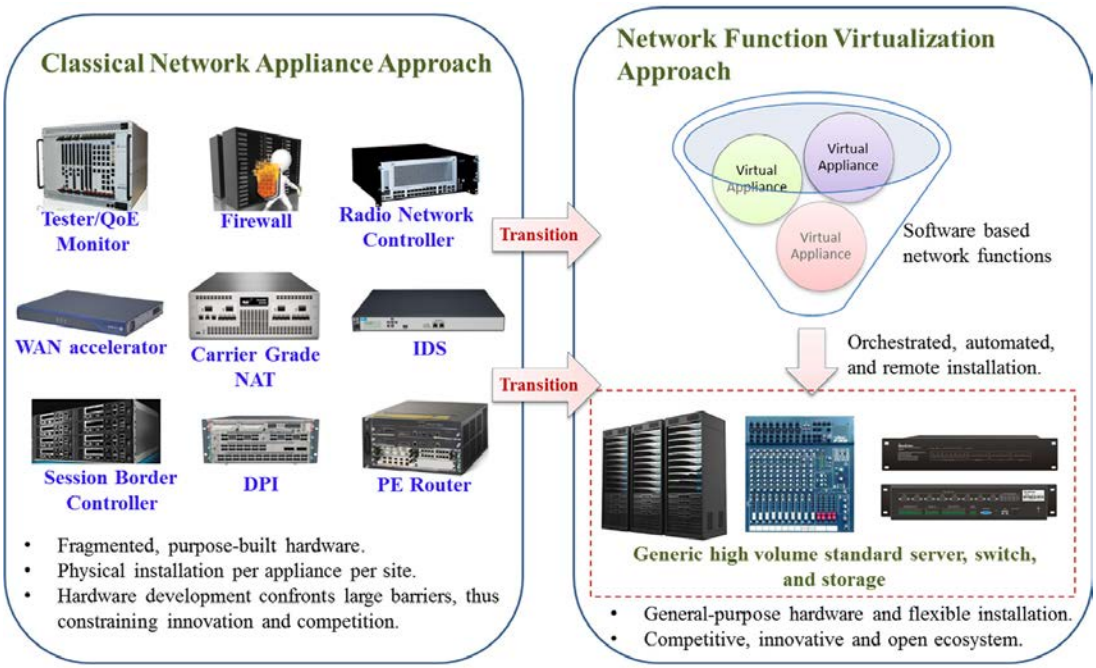


Figure 2.2 – Network Function Virtualization vision [151][51]

organization.

ETSI-NFV		IETF	
Virtualized Network Function	VNF	Service Function	SF
Connection Point	CP	Service Function Forwarder	SFF
Virtual Link	VL	Virtual Link	VL
VNF Forwarding Graph	VNF-FG	Service Function Chain	SFC
Network Forwarding Path	NFP	Service Function Path	SFP

Table 2.1 – ETSI and IETF Notations For virtual functions [80].

As presented in Table 2.1, the network virtualization ecosystem is composed of a number of terminologies introduced by NFV that we define in the following:

- ***Physical Network Function (PNF)***: is the name given to the traditional physical appliance, since it is closely coupled with the appliance.
- ***NFV***: is the concept of applying virtualization to network functions.
- ***VNF***: is the software for a network function application. This software is hosted in a virtual host. VNF is enabled by NFV.
- ***Virtual Host***: is the virtualization technology hosting the network applications. Virtual

hosts are of distinct types, with different requirements. VMs and containers are examples of virtual hosts.

- **NFVI Point of Presence (NFVI-PoP)**: represents the physical resources (i.e. memory, CPU and network). Each NFVI-PoP may host one or more VNFs, whereas a PNF is coupled to one dedicated NFVI-PoP. The NFVI-PoP may also be named: physical host, machine or server.

- ***Virtual Link (VL)***: represents the link between VNFs.

- ***Physical Link (PL)***: represents the link between the NFVI-PoPs. Note that two connected VNFs hosted on a distinct NFVI-PoP communicate through the PL between the NFVI-PoPs.

- ***Connection Point (CP)***: represents the interface that offers the network connections between network functions and links. CP could be of virtual or physical type. Some examples of CPs interfaces are: virtual or physical port and a virtual or physical Network Interface Controller (NIC) address.

- ***Network Function Virtualization Infrastructure (NFVI)***: represents the infrastructure hosting the VNFs. NFVI interconnects the computing and storage resources contained in an NFVI-PoP. The NFVI-PoPs are connected through PLs.

- ***NCT***: specifies the network topology among VNFs i.e. the connection between VNFs and PNFs nodes that compose the global virtual network. In the NCT each VL connects two VNFs through CPs that represent the VNF interfaces.

- ***VNF-FG***: represents a sequence of VNFs interconnected to provide a complex network service with specific functionalities.

- ***Network Forwarding Path (NFP)***: are subsets of the processing flows through the VNF-FG. In fact, one VNF-FG can have multiple forwarding paths. For instance in Figure 2.3, the VNF-FG-1 has two flows, traffic flow through NFP-1 and control flow through NFP-2.

- ***NS***: represents a composition of one or more VNF-FGs. The NS is a complete solution offering network services to clients such as Voice over IP (VoIP). The NS solution includes a number of functions, such as security and monitoring. Each of these functions represent a VNF-FG of connected VNFs. For instance, a video streaming service that applies a security NFV chain composed of a Deep Packet Inspection (DPI) and a Firewall.

- ***VIM***: represents the orchestrator for the deployment of VNFs.

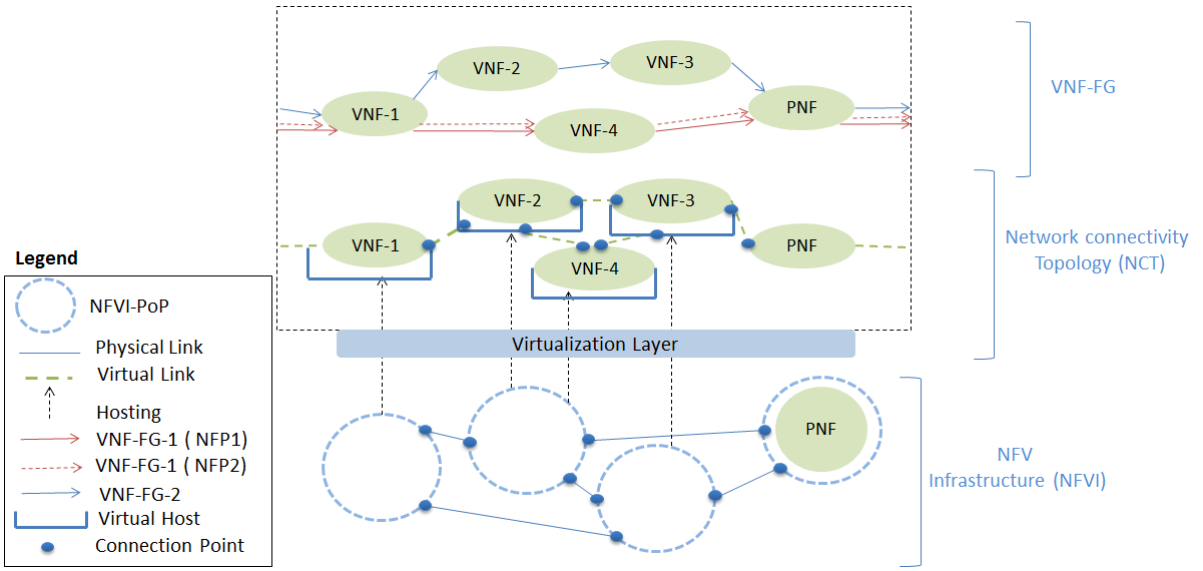


Figure 2.3 – Network Functions Virtualization

- ***Virtualization Layer***: express the separation between the physical layer and the hosted VNFs. The virtualization layer may include the VIM and the virtual hosts.

Figure 2.3, based on the use-case presented in [34], depicts an end-to-end network service chain composed of four VNFs and a PNF. The VNFs are connected with virtual links defined in the NCT. Each service traffic is routed through the VNF-FG. Moreover, the coexistence between VNFs and PNF is likely to happen in production infrastructures since some functions might still not be virtualized. For instance, a web Hypertext Transfer Protocol (HTTP) authentication request composed of a virtual webserver hosted in VNF-1, a virtual database hosted in VNF-4 and a physical firewall hosted in the PNF. In addition to the NFV concept, a number of ecosystems are included in the network virtualization architecture presented in Figure 2.1. We depict these ecosystems in the following Sections 2.2.1, 2.2.2, 2.2.3.

2.2.1 Software Defined Networking (SDN)

SDN is an emerging network architecture that decouples the network control, management, and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services i.e. the virtualization of network functions such as the virtual switches [69]. The abstraction consists of separating data and control plane. The separation between control and data planes means that the control plane, which contains the SDN controller, decides on behalf of the data plane resources.

In the SDN architecture the controller defines the control plane functions that include the system configuration, management, and exchange of routing table information. The network switches become then simple forwarding devices that are super efficient in blindly applying forwarding rules and the control logic is fully delegated to the controller. This simplifies policy enforcement and network reconfiguration, using OpenFlow protocol for example. The construction of SDN enables the controller to directly interact with the forwarding plane of network devices such as switches and routers.

The SDN architecture is generally divided into three layers: the infrastructure (or data plane), the control plane, and the application layer (cf. Figure 2.4). The infrastructure layer also known as data plane represents the layer holding the OpenFlow switches, the hosts/servers acting as traffic sources and sinks and all the control and data links seen in the infrastructure. The application layer hosts the NFs and communicates with the SDN control plane through the northbound Application Programming Interface (API) regarding the status of the network and its particular requirements. The controllers, situated in the control plane, dictate forwarding rules to the data forwarding devices through the southbound API.

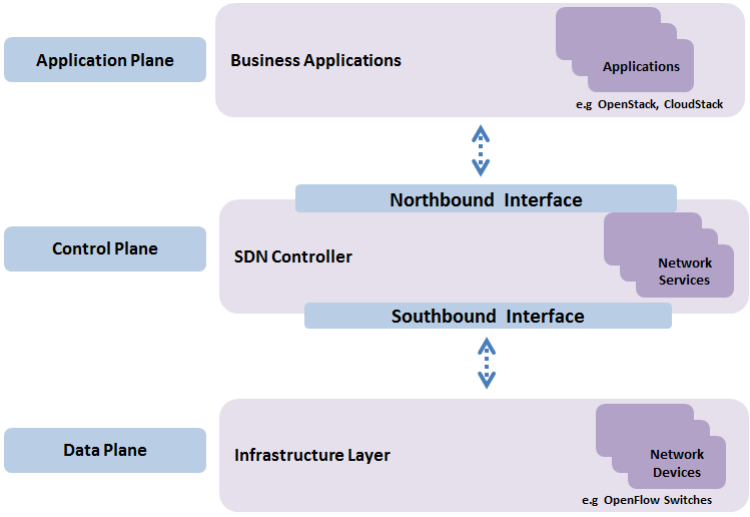


Figure 2.4 – SDN layered architecture.

Recently, open source communities proposed various SDN controllers with varying degrees of maturity and functionalities. OpenDaylight [100] is an open source controller hosted by the Linux foundation. OpenDaylight renamed recently as an OpenDaylight platform which is a project enhancing the SDN community for the OpenDaylight Controller. The OpenDaylight platform enables users to build an SDN controller to fit their specific needs. This platform is multi-protocol by the mean that users can select multiple protocols (e.g., OpenFlow, NETCONF and SNMP). Open Network Operating System (ONOS) [96], is another SDN controller project

hosted by the Linux foundation. This project focuses more on the performance and the clustering aspects to increase the availability and scalability. Finally, Opencontrail [99], is a Jupiter Networks project and an open cloud network automation product that uses both SDN and NFV technologies to orchestrate the creation of virtual networks.

2.2.1.1 SDN and NFV coexistence

SDN and NFV are two network architectures that might be used interchangeably in the literature, they are however independent concepts and arguably complementary. At their essence, NFs can be virtualized and deployed without SDN technologies, and non-virtualized functions can be controlled by an SDN. Meanwhile, both technologies can be complementary and mutually beneficial; NFV is able to support SDN by providing the infrastructure upon which the SDN software can run. For instance, we can consider the software of the SDN controller or the function of the forwarding devices used in the SDN infrastructure plane as a VNF. Furthermore, an SDN infrastructure can be used for the data forwarding between VNFs. From the Open Networking Foundation (ONF) viewpoint, a VNF for an SDN controller is just another resource, a node function in a network graph with known connectivity points [155].

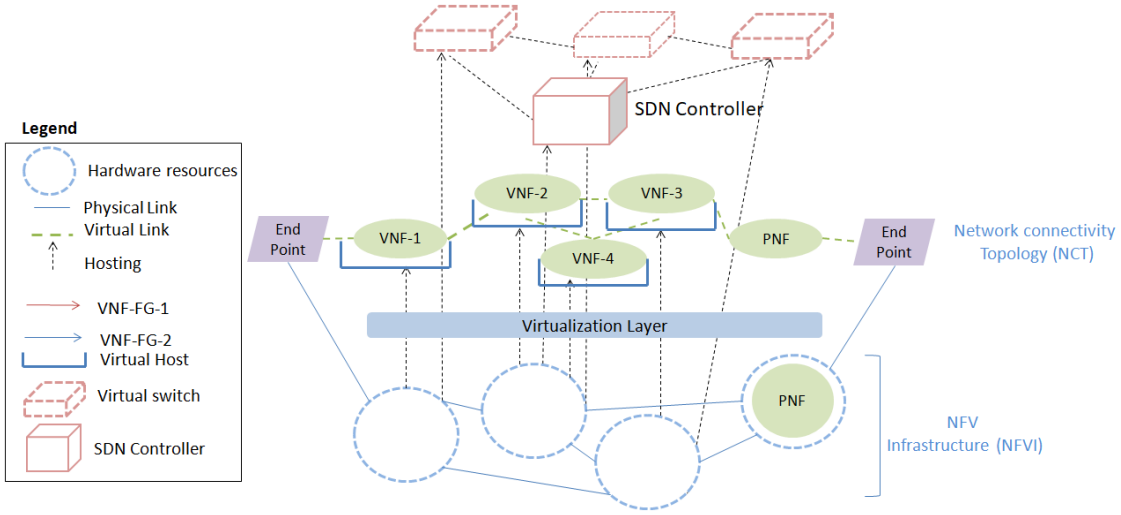


Figure 2.5 – An example of NFV and SDN coexistence. The routing of VNFs traffic is enabled by SDN. The SDN virtual switches and the controller are hosted in virtual hosts.

Combining NFV with SDN in one infrastructure makes sense from a network operator's standpoint, to reduce the costs of NF deployment and management. Moreover, the SDN scalability and elasticity allows for a dynamic deployment of NFV that suits the on-demand NFV chain placement and the communication requirements for both virtual and physical networking infrastructures [155]. In addition,

To illustrate the relationship between SDN and NFV, we added in Figure 2.5 an SDN cluster to the example presented in Figure 2.3. The cluster is composed of three virtual switches and one virtual controller. The traffic between the VNFs composing the NCT is routed through the virtual switches with the instructions of the SDN controller. In this case, the network functions are included in the SDN switching rules.

However, deploying SDN controllers in the NFV architecture remains an option. Some NFV use-cases have their own ways to route the NS packets. For instance, the OpenStack open source infrastructure project provides a networking module called Neutron [103]. Neutron is an OpenStack module that provides networking services for OpenStack VMs. Further examples of networking including Neutron are depicted in Section 2.2.4.2.

2.2.2 Multi-tenancy and slicing

To make a better use of virtualization, operators may share their non-occupied resources with customers, who become tenants of the same infrastructure. The definition of multi-tenancy differs between vendors and entities. Generally speaking, it means that multiple tenants or clients are sharing the same virtual compute, storage and network resources [124]. For instance, in Figure 2.1, two tenants are sharing NSs in the same infrastructure. Tenant-1 owns the NS-1 and tenant-2 the NS-2 and NS-3. These NSs are running on the same physical infrastructure.

Multi-tenancy is enabled by the notion of slicing, which allows the traffic of multiple tenants to be compartmentalized through the same infrastructure, with a tunnel representing the path for data [2]. The network slice is composed of an independent set of software network functions that support the requirements of a particular use-case. Slices are completely isolated so that no slice can interfere with the traffic in another slice. Slices enables to deploy only the functions necessary to support particular customers with a particular SLA. For instance, an autonomous Vehicle to anything (V2X) car communication service requires low latency but not a high throughput. While a streaming video in the car requires a high throughput and is susceptible to latency. Both services could be delivered over the same shared physical infrastructure but on different virtual network slices implementing different SLA.

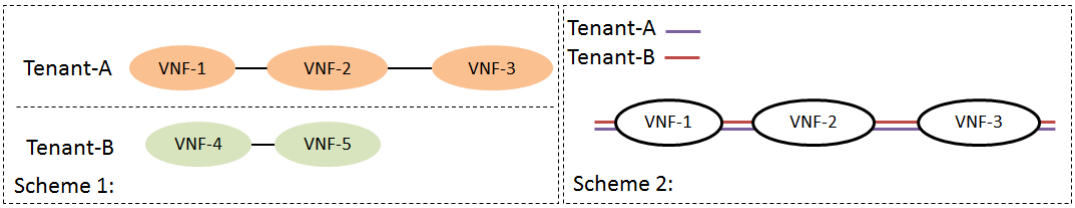


Figure 2.6 – Multi-tenant cases for NFV SaaS. In case 1 tenant-A and tenant-B have different VNFs, while in case 2, they are sharing the same VNFs with two different slices.

MNOs open their infrastructure to clients with different levels of restrictions: they provide software with no client control (SaaS), or software and data client control (Platform as a service (PaaS)) or provide virtualization, storage, and processing and delegate the control of applications to clients (Infrastructure as a service (IaaS)). In the IaaS scheme, the client has access to the whole shared infrastructure, while in the PaaS the infrastructure is managed by the MNO and the client can deploy any service or application. In the SaaS scheme, MNOs can share the whole VNF-FG representing a NS for a client or only a slice through the VNF-FG. In Figure 2.6, we depict both SaaS sheme two cases. In the first case, tenant-1 owns (VNF-1, VNF-2 and VNF-3), while the second tenant-2 owns different VNFs: VNF-4 and VNF-5. In the second scheme both of the tenants share the same VNFs: VNF-1, VNF-2 and VNF3. However, their traffic is routed in two different network slices.

2.2.3 Virtual Hosting Environment

Virtual functions are only software applications and operators should provide an appropriate running environment. Virtual Machines (VMs), containers and unikernels provide virtual compute and storage resources that are crucial to network function virtualization. Containers emerged as a way of running applications in a more flexible and agile way. Containers existed since 2006, but they became more popular with the arrival of Docker containers in 2013 [29]. Containers enable running lightweight applications directly within Linux Operating System (OS), whereas each VM runs an independent OS. The VMs are logically isolated from one another.

A Virtual Machine (VM) hosts an operating system that runs on top of the hosting machine's OS. The hypervisor enables a single physical machine to run multiple VMs with different operating systems. For example *VMware*, *Xen* and *KVM* represent one of the most popular hypervisors. Unikernels are a lightweight alternative to VMs that packages the VNFs with their required libraries; unlike VMs that provide an entire guest OS. Unikernels use a *"library operating system"* implementing only the kernel features compiled with the unikernel image code. This makes the sizes of unikernel images similar to those of containers. The VMs and containers offer novel ways of virtualization discussed in [151]. One possible configuration is running containers inside the same VM.

The NFV Research Group Internet Research Task Force (NFVRG-IRTF) [91], a research branch of the IETF, focuses on longer term research issues related to NFV, presented in [89] a comparison of VMs, containers and unikernels hosts for NFV. In terms of service agility that represents the ability to migrate and spin up and down VNFs, container and unikernels come first, since they are lightweight compared to VMs. Same goes for the memory consumption, containers and not far behind unikernels and are less memory consuming. In terms of security and isolation, VMs provide a better isolation compared to containers since VMs run on isolated OSs. Finally, VMs and containers are more compatible with open source frameworks compared

to unikernels that are less present in open source communities [89].

However, the choice between the technologies depends on the network policy and requirements. It depends on how much value MNOs place on requirements such as strong isolation eligibility, performance and compatibility with applications and management platforms [89]. In fact, containers are used when the service provided need to be lightweight with a rapid execution time like microservices. VMs are usually applied to host containers or to deploy over different OSs.

Figure 2.7, illustrates a comparison between the implementation of a VNF in a VM, a container and a unikernel machine. VMs run a different OS on top of the physical server OS, while LibOS or unikernels only select, from a modular stack, the minimal set of libraries which correspond to the OS constructs required for their application to run. This makes unikernels ligher than VMs. In the case of containers, the container engine uses the Linux host machine resources through *cgroups*[1] and *namespace*[2].

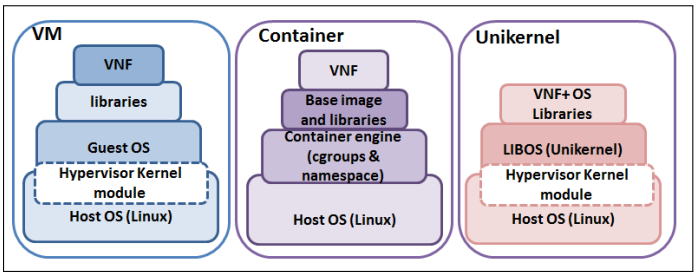


Figure 2.7 – A simplified view of VNF deployment on virtual machines, containers, and unikernels [89]. Containers are more lightweight for hosting VNFs than unikernels and VMs.

2.2.4 NFV Management and Orchestration (MANO)

The **management** and **orchestration** of virtual networks represent an essential part of the NFV architecture to ensure a complete VNF life cycle starting from the deployment to the administration, maintenance and provisioning, **Orchestration** is the process that enables autonomic deployment, provisioning and configuration of network services. While the **management** process considers the availability of VNFs and services by managing the life cycle of VNFs (i.e. instantiate, scale, update and/or upgrade, and terminate VNFs). Management also includes the traditional FCAPS functions.

[1] *Cgroups* provides a mechanism for partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior.

[2] *Namespace* abstracts the system resources to make it appear to the processes that they have their own isolated instance of the global resource.

A lot of standardization work has been done to define the most efficient NFV architectures. The ETSI-NFV working group divides the NFV architectures into major components including VNFs, NFV MANO, and NFVI on top of the traditional network components like Operation Support System (OSS) and Business Support System (BSS) [36]. The later includes the collection of systems and management applications that a service provider uses to operate its business such as VoIP. Figure 2.8 illustrates the NFV architectural framework of ETSI depicting the main functional blocks and reference points. The reference points represent the connections between the functional blocks as defined by ETSI. The execution connection point represents the relation between the NFVI and the instantiated VNF. The NFVI includes both the hardware and software environments in which the VNFs can be deployed, hosted and managed.

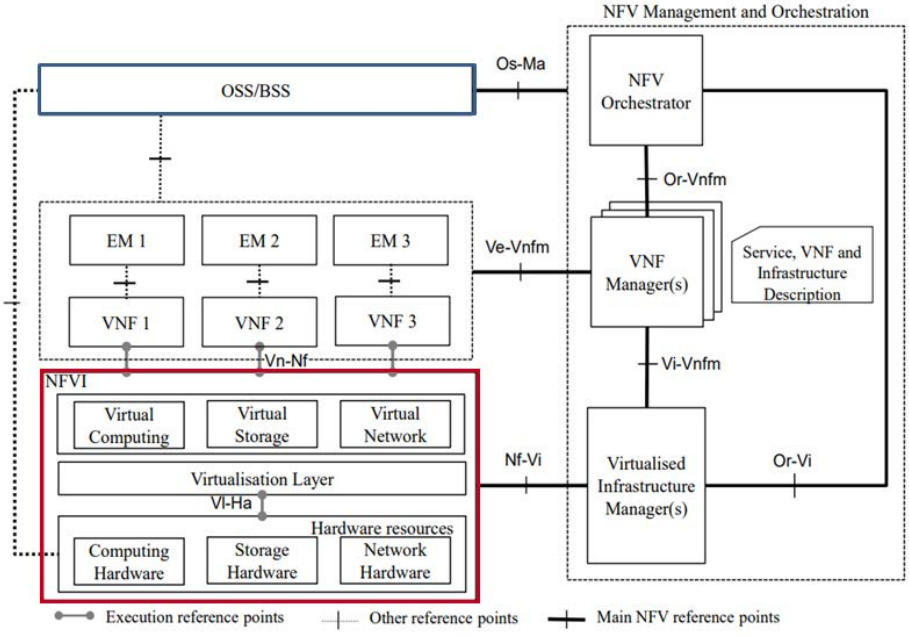


Figure 2.8 – ETSI NFV Architectural Framework [34].

The MANO manages the NFVI and orchestrates the allocation of resources needed by the NSs and VNFs. The NFV-MANO is composed of three main blocks:

- The VNF Manager (VNFM): responsible for the life cycle management of the VNFs. The VNFM life cycle operations for VNFs include: instantiating, scaling, updating and/or upgrading and terminating VNFs.

- The NFV Orchestration (NFVO): manages and coordinates the life cycle of: NSs, VNFM and NFVI devices, and ensures an optimized allocation of the necessary resources (i.e.

computing, storage and network). The NFVO can interact directly with the NFVI resources. It can either coordinate, authorize, release, and engage the resources without/or by engaging a VIMs through the northbound API (i.e. the "Or-Vi" reference point shown in Figure 2.8).

- The VIM: controls and manages the NFVI compute, storage and network resources, i.e. the NFVI-PoPs.

In order to instantiate a network service in the NFV architecture, first the OSS/BSS sends a service order to the NFVO described in a file such as OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [142][3]. The NFVO translates the order into resources sent as output to the NFVI infrastructure to allocate the necessary resources (i.e. Compute, storage and network) to instantiate that network service. The VNFM will then instantiate the VNFs and the VIM allocates the necessary VNFs in the NFVI for each virtual link. Element Managements (EMs) are responsible for the management of the functional behavior of VNFs, such as a signalling issue in mobile core.

2.2.4.1 Virtual Infrastructure Manager (VIM)

The VIM represents an important part of the NFV-MANO architecture, and is critical to reach the full benefits enabled by the NFV architecture. It coordinates physical resources to deliver network services as defined in the virtualization layer. In the NFV market, many vendors propose VIMs solutions: *OpenVIM*, *OpenStack* and *Kubernetes*.

- ***OpenVIM***: is a lightweight implementation of an NFV VIM supporting some of the Enhanced Platform Awareness (EPA) features such as the support of memory huge pages [107]. EPA aims at facilitating decision making related to VM placement and improvement for cloud clients. OpenVIM was created to represent an open source project providing a practical implementation of the whole ETSI MANO architecture. In this project "OpenVIM" is the project representing the VIM. OpenVIM is maintained nowadays by the opensource MANO (OSM) project [37].
- ***Openstack***: is an open-source software platform for cloud computing. It aims at running a cluster of the devices executing different kinds of hypervisors and to manage the required storage facilities and virtual network infrastructures. The OpenStack architecture is composed of multiple modules responsible of the compute, storage, and networking of resources throughout a data center. It also provides a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

[3]TOSCA is a description language for cloud and network services developed by the Organization for the Advancement of Structured Information Standards (OASIS)

A typical OpenStack cluster includes a controller node, a network node hosting the cloud networking services, compute nodes (VMs) and storage nodes for data and VMs [12]. The OpenStack Heat module is responsible for creation, modification, rebuild and deletion of the entire stack of VMs.

OpenStack offers a telemetry service, namely Ceilometer, for collecting measurements of the utilization of physical and virtual resources [104]. Ceilometer can collect a number of metrics across multiple OpenStack components, it can watch for variations and trigger alarms based upon the collected data. In more recent OpenStack versions, containers are considered to be deployed on top of OpenStack VMs.

- ***Kubernetes***: is an open source container management and orchestration engine originally developed by Google [72]. Kubernetes aims at the automation of deployment, scaling, and management of containerized applications. Kubernetes follows a master/slave model, where a master node is deployed to manage Docker containers across multiple Kubernetes nodes.

Another way to orchestrate containers without deploying a fully Kubernetes orchestrator is to use the native Docker compose [30]. Docker compose is integrated to the Docker project. It enables to run multi-container Docker applications. It also offers tools to scale and manage the life cycle of containers.

2.2.4.2 NFV networking

Networking between VMs or containers in most of the VIMs presented in Section 2.2.4.1 could be managed by an external SDN controller. However, each VIM project has its own default integrated networking module. To manage its network, OpenVIM interfaces with a Openflow controller such as OpenDaylight. The OpenStack VIM proposes the Neutron module for networking [103]. Neutron is a software module specifically dedicated to network service management. A centralized Neutron server stores all network-related information. It provides an API that enables operators to define network connectivity and addressing in the cloud. Neutron also provides a variety of network services such as NAT, load balancing and virtual private networks. In the case of Docker, the Dockers running on the same physical host communicate through a network called a "*bridge*" network. A *bridge* is a private, internal default Docker network on the host. All containers are attached to this network by default with an internal Internet Protocol (IP) address (Generally in the range of "**170.17.x.x**").

For example, suppose that we have a physical host (Host1) with a physical network interface **eth0** (Cf. Figure 2.9-a). Attached to that is a bridge network **Docker0**, and attached to that is a virtual network interface **veth0**. In this example **Docker0** is assigned **172.17.0.1** and

is the default gateway for **veth0**, **veth1** and **veth2**, with the following IP addresses: **172.17.0.2**, **172.17.0.3** and **172.17.0.4**, respectively. The processes inside each Docker see only the corresponding virtual Docker interface **"veth"**, and communicate with the outside world through **Docker0** and **eth0**. Container1, container2 and container3 share the same logical network as the bridge **Docker0**, so they can communicate through the bridge as long as they can discover each others IP address. Note that a Docker can be isolated from the other Dockers on the network with a "**None**" network type.

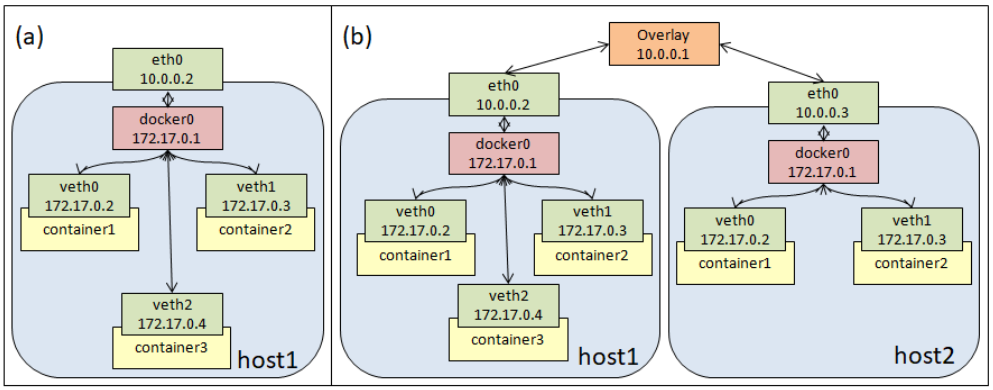


Figure 2.9 – (a): Three containers sharing the same host and communicating through a network bridge "Docker0", (b): Two hosts communicating through an overlay network.

Now suppose that another physical host (Host2) is running two other containers (Cf. Figure 2.9-b). Each Docker host has its own internal private network in the "172.17.x.x range" allowing the containers running on each host to communicate with each other. However, containers across the host have no way to communicate with each other unless you create the "**overlay**" network. An overlay network creates an internal private network that connects all the containers composing the same swarm cluster (i.e., the containers on the same swarm cluster are authorized to communicate with each other).

Kubernetes networking is similar to Docker networking. Kubernetes defines communications between pods. Pods are the basic kubernetes application or service unit. A pod consists of one or more containers that are hosted on the same physical host. Communication inside a pod is enabled by the network "bridge" and the communication between pods is enabled by the "overlay" network or an external router defined by Kubernetes.

2.2.4.3 NFV fault and performance management

The traditional FCAPS management tasks are outside the scope of the traditional MANO [94] architecture, that focuses on development of configuration and the life cycle management.

However, the ETSI GS for NFV takes into consideration the fault and performance management of the VNFs life cycle. Additional components to the existent MANO blocks will ensure the VNFs service availability and continuity with respect to the defined SLAs. One example of the actions defined by the ETSI GS to ensure VNF availability is the distributed fault correlation processing. Its scope is to avoid propagating large number of failure notifications to a centralized entity by sending locally correlated reports [62].

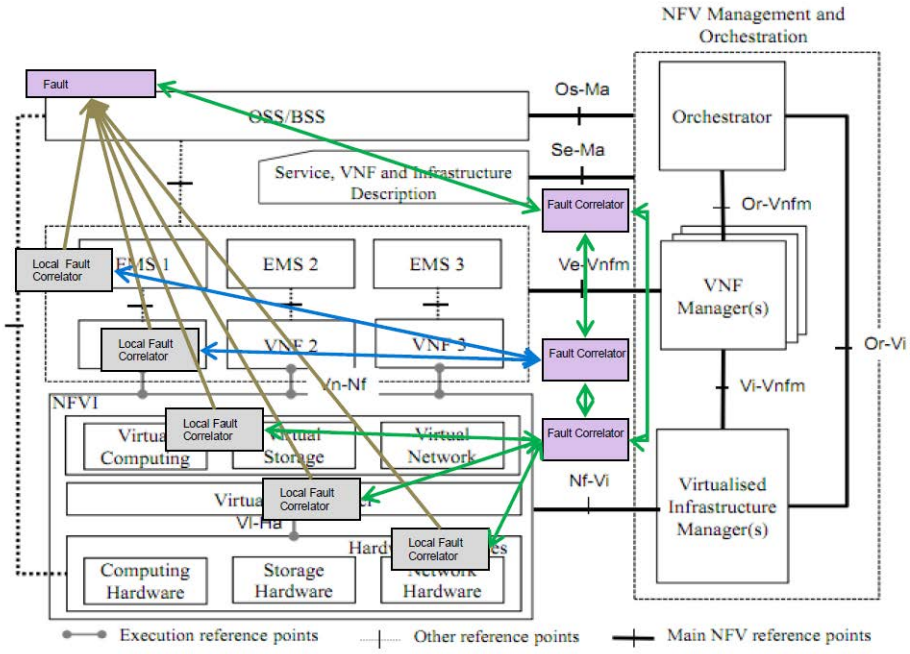


Figure 2.10 – Fault Correlation schemes in the NFV Architecture, as represented by the ETSI GS [62]. This architecture illustrates four local fault correlators and three external fault correlators.

This example is depicted in Figure 2.10. In this example, potential placements of fault correlators are illustrated. The fault correlators could be placed locally in each layer (i.e. hardware, virtualization layer, VNFs and EMs) or externally. Local fault correlators collect failure alarms within their layer. Each local fault correlator defines correlation rules to select the possible root cause candidates that are responsible for the reported errors in their managed layer. The external fault correlators collect correlated reports from local correlators and/or the reports from other external fault correlators then apply correlation rules based on a layered graph [62].

Other initiatives to build a unified management and orchestration platform were presented in other working groups. For example the OPNFV[4] project [98], created the **Doctor** project [109], to build fault management and maintenance rules for the high availability of the NSs. This

[4]OPNFV aims at facilitating the development and evolution of NFV components across various open-source

project focuses on immediate notification of unavailability of the virtual resources, to process recovery actions on the affected VNFs.

2.3 A VNF chain: *Clearwater vIMS*

The Virtualisation trend poses to MNOs the challenge to be able to rapidly deploy and offer new services. In essence, a number of use-cases were presented in the white papers [35, 73, 53], that represent the principal inspiration for most industries when deploying or illustrating NFV solutions. Other industrial products and solutions of VNFs are summarized in the NFV Survey [151]. The Virtual IP Multimedia Subsystem (vIMS)[5] represents one example of VNF solutions that enables MNOs not only to be more efficient and flexible, but also to offer more advanced and reliable voice communication services. vIMS is the virtual solution of the classical IMS, which is an IETF and 3GPP standard for Voice over IP (VoIP) for 4G and 5G, and an architectural framework for delivering IP multimedia services such as voice, video calling, and messaging applications. These multimedia services are delivered through SIP. SIP is the standard protocol for the telecommunication multimedia services signaling [58].

IMS provides a number of benefits for MNO including the efficient use of spectrum, eliminating the need to separate voice and data in two different networks and the interoperability of multimedia services across operators such as video calling [21]. The virtualization of IMS brings additional benefits to MNOs, inherited from NFV: scalability, programmability and flexibility of services.

A number of open source organization proposed vIMS solutions: Oracle IMS Session Delivery [110], open source IMS [102] and *Clearwater* vIMS [19]. In this section we present the *Clearwater* vIMS use-case. *Clearwater* is an open source virtual implementation of the IMS developed by Metaswitch [82]. Two versions of *Clearwater* vIMS are available: the Docker version [138] and the VM OpenStack version [140]. The *Clearwater* vIMS implements the principal standardized interfaces and functions of an IMS. It adapts the established design patterns for building and deploying massively expandable virtual applications in cloud to meet the constraints of IMS, which enables industries to easily deploy, integrate and scale IMS functions. Figure 2.12 presents the global architecture of the *clearwater* vIMS components and protocols. To clarify this architecture we will first depict the traditional IMS architecture.

ecosystems. Its objective is to create a unified reference NFV platform to accelerate the transformation of enterprise and service provider networks. This is enabled by combining multiple open source projects (e.g. *OpenStack*, *OpenDaylight* and *ONOS*) to create a common ecosystem for NFV, through system level integration, deployment and testing.

[5] vIMS stands for "virtual Ip Multimedia Subsytem" and not "Virtual Infrastructure Manager"!

2.3.1 IP Multimedia Subsystem (IMS)

IP Multimedia Subsystem (IMS) vision is to integrate mobile/fixed voice communications and internet technologies together. It enables a variety of IP-based Multimedia Telephony Service (MMTel) services like voice, video, multiparty conferencing and instant messaging. The architecture of IMS illustrated in Figure 2.11, provides a number of mechanisms for managing, controlling and routing sessions, in addition to the authentication, authorization and accounting controls. The Call Session Control Function (CSCF) is the heart of the IMS architecture responsible for regulating communications flows and controlling sessions between clients User Equipments (UEs) (or terminals) and applications.

To enable the home network to control its communication in the case of roaming. The CSCF function was divided into three main functions: Proxy-CSCF (P-CSCF), Serving-CSCF (S-CSCF) and Interrogating-CSCF (I-CSCF) [141]. In fact, the P-CSCF, which is the client's initial contact point inside a local or visited operator IMS network, will either serve the client request internally or forward it to other servers in the case of a client visiting another operator network (i.e. roaming). The P-CSCF also forwards SIP messages from the CSCF functions to the UE.

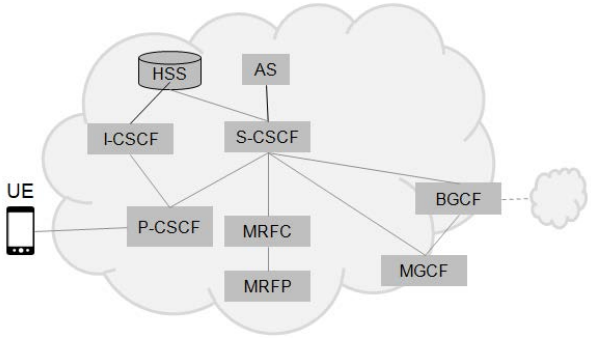


Figure 2.11 – Core IMS architecture.

The I-CSCF within an operator's network is the contact point for all the requests destined to a user of that network operator or a roaming user currently located within that network operator service area. The I-CSCF is responsible for assigning the right S-CSCF to the UE performing a SIP registration. To choose the appropriate S-CSCF to route the user's request, the I-CSCF obtains the S-CSCF address from the Home Subscriber Server (HSS). This address is then used to route all the client's requests in the same session from the P-CSCF to the chosen S-CSCF directly. The S-CSCF is in charge of managing both client's registration and service assignment. The S-CSCF, validates the client's authentication information from the profile in the HSS. HSS is the master database that maintains all user profile information used to authenticate and authorize subscribers. The HSS holds both static and dynamic client's data. The static informa-

tion concerns the client's fixed information such as the IMS Public User Identities (IMPU). The IMPUs are public identifiers that identify the client for originating and terminating multimedia sessions. The dynamic data is the data that only last for one SIP session such as the S-CSCF address assigned to a specific client's UE.

All the MMTels such as voice and video calling, requested by a user are provided by the IMS Application Server (AS). The Multimedia Resource Function (MRF) enhances multimedia application provided by the AS. The MRF functionality is to control the media streams and provide new functionalities to process it. MRF integrates advanced video conferencing features and supports new audio and videos CODECs for conferencing and streaming. The MRF is divided into a Multimedia Resource Function Controller (MRFC) and a Multimedia Resource Function Processor (MRFP). The MRFP is responsible for managing the media streams, while the MRFC controls the MRFP and forwards the S-CSCF requests addressed by the AS to the be proceeded by the MRFP.

The IMS architecture allows different charging capabilities to be used, particularly, off-line (or postpaid) and on-line (or prepaid) charging. The Charging Trigger Function (CTF) generates charging events based on the observation of network resource usage and sends these charging events towards the Charging Data Function (CDF) server via the interface called Reference Point (Rf). It then uses the information contained in the charging events to produce the Charging Data Records (CDRs) billing reports [14].

To enable interoperability with Public Switched Telephone Network (PSTN), the IMS core interconnects to the PSTN through the Breakout Gateway Control Function (BGCF) and Media Gateway Control Function (MGCF). The BGCF chooses the network where PSTN breakout happens. The BGCF receives request from S-CSCF to select appropriate PSTN break out point for the session. If the breakout is to occur in the same network as the BGCF then the BGCF selects a MGCF which will be responsible for the interworking with the PSTN. For destinations in peer IMS networks, the BGCF selects the appropriate Interconnection Border Control Function (IBCF) to handle the interconnection to the peer IMS domain. The MGCF ensures to convert SIP messages to ISDN User Part (ISUP) PSTN signaling.

2.3.2 *Clearwater* vIMS

The cloud-native *Clearwater* vIMS follows the classical IMS architectural specifications. It offers a diversity of MMTels such voice and video calling services, including basic calling features such as Call Forwarding. *Clearwater* offers the possibility to host other rich telephony services using an external Telephony Application Server (TAS) like MetaSphere Multimedia [19]. MetaSphere Multimedia is a Metaswitch solution that offers a sophisticated multi-service audio features [81].

As depicted in Figure 2.12, the main functions of IMS presented in Section 2.3.1 can be virtualized. Each box represented in the architecture of Figure 2.12, implements one or more

classical IMS functions. For instance the *Sprout* box contains the I-CSCF, S-CSCF, BGCF and TAS. The defined architecture in Figure 2.12 only represents the main VNFs of IMS. The deployment of the *Clearwater* vIMS project includes others functions such as *ETCD*, *Astaire* and *Chronos*. The *Cassandra* database, represented in Figure 2.12, is deployed in a separate virtual host than *Homestead*.

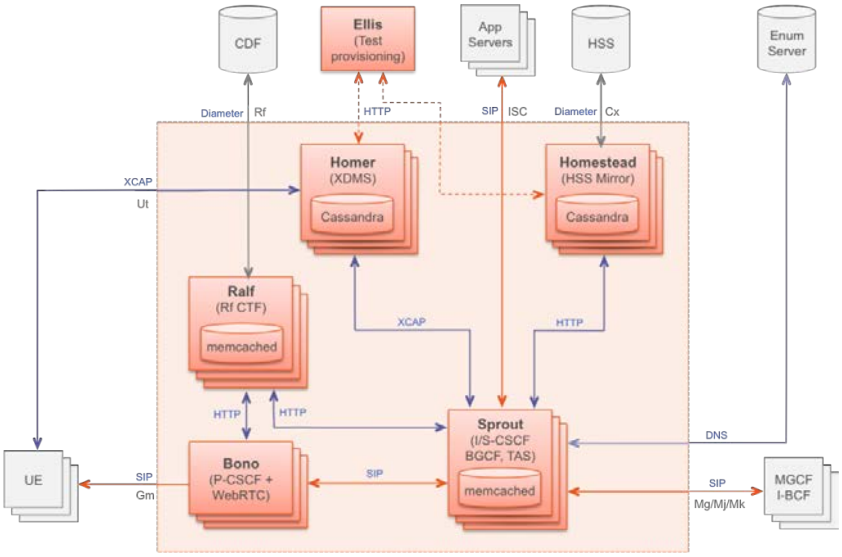


Figure 2.12 – *clearwater* vIMS architecture [19].

The main vIMS *clearwater* functions represented in Figure 2.12 are depicted as follows:

- ***Bono*** is the SIP proxy implementing the P-CSCF. Bono nodes provide the entry point for the client's UE connection to the *clearwater* system. Bono supports any client using the SIP protocol or SIP over a Web Real-Time Communication (WebRTC) web socket. The client is then attached to a particular Bono node for the duration of its registration, but can switch to another Bono node if the connection fails. The client request is then routed to the connected Sprout.

- ***Sprout*** implements both the S-CSCF and I-CSCF functions. Sprout nodes send requests to Homestead and Homer interfaces to retrieve client configurations such as authentication data and MMTel settings that are stored in Cassandra. Sprout node hosts the MMTel AS. In sprout, we found the TAS telephony service such as call waiting, call transfer and call blocking services.

- ***Homestead*** represents an HTTP RESTful server that allows Sprout to retrieve authentication credentials and user profile information. It either stores users data in a Cassandra database or requests data from an external HSS.

- ***Homer*** is a standard XML Document Management Server (XDMS) used to store MMTel settings documents for each client. Documents are manipulated using a standard XML Configuration Access Protocol (XCAP) interface.

- ***Cassandra*** is a distributed No Structured Query Language (NoSQL) open source database used by Homestead to store authentication credentials and profile information, and used by Homer to store MMTel service settings [6]. Cassandra represents the HSS.

- ***Ralf*** enables both Bono and Sprout to report billable events. An external CDF to *clearwater* could be deployed to connect to Ralf via the Rf interface.

- **Ellis** is a dashboard that simplifies the client's MMTel services provisioning, providing line management and control. *Ellis* enables the *Clearwater*'s administrator to create SIP profile users containing their authentication password and SIP numbers that enables them to register to a *Bono* node for any vIMS MMTel service. The administrator can also manage the client's MMTel service setting such as the allocated MMTel services.

There exist *Clearwater* functions that are not mentioned in the architecture presented in Figure 2.12 and are present in the deployment of *Clearwater*:

- ***ETCD***: is a distributed reliable key-value store [33]. In *Clearwater* vIMS, ETCD cluster is used to share clustering and configuration information between nodes. ETCD is a crucial node that enables sharing the networking configuration.

- ***Chronos***: is a distributed timer function developed by Metaswitch. It is applied by *Sprout* and *Ralf*, via an HTTP API, for example for the SIP Registration expiry.

- ***Memcached*** / ***Astaire*** / ***Rogers***: This *Memcached* cluster is used by *Sprout*, *Homestead* and *Ralf* for storing registration and session state. The *Memcached* cluster is synchronized by Astaire and fronted by *Rogers*. *Astaire* enables a rapid scale up and down of *Memcached* clusters. *Rogers* provides data replication between the *Memcached* nodes.

Other external functions could be added to *Clearwater* such as the *ENUM* for mapping PSTN numbers to SIP Uniform Resource Identifier (URI) using DNS, the CDF for client charging and the MGCF to convert the SIP messages. In the more recent version of *Clearwater*, the *Ralf* and *Homestead* nodes are hosted in a component called the diameter gateway *Dime* and the distributed databases *Cassandra* and *Memcached* are stored in *vellum* [120]. In particular *Clearwater* enables to scale services. Interfaces between *Clearwater*'s components use statistical recycling of connections to ensure load is spread uniformly as components are added and removed from the network.

To illustrate the role of each *Clearwater* component, suppose a client wants to call another user using a UE supporting SIP applications. The UE should first register or subscribe to the nearest *Bono* node. To register, a node should provide its authentication information. The *Bono* node sends the client request to the Sprout node. Sprout requests verification from Homestead. Homestead retrieves the authentication information from *Cassandra* and compares with the ones in the client request. If the authentication credentials are correct the node will be registered. The clients can then access to SIP application provided by the AS. The client's billing information is managed by *Chronos* and *Ralf*.

2.4 Virtual Networks features and challenges

Network Virtualization Ecosystems (NVEs) enable distinct network architectures to coexist in a single infrastructure. This architecture brings new features that are in the same time benefits and challenges to the existing fault management procedures. In fact, virtualization offers a number of benefits to MNOs including scalability, the reduction of management and deployment costs, and the programmability and flexibility of services. One more important benefit expected from NVE is the high availability and reparability of services. Virtualization offers the option to heal service failures through automated reconfiguration in the case of software failures and moving VNFs in the case of hardware failures or even moving traffic to new VNFs in the case of VNF traffic load [144]. Moreover, the healing procedure could be autonomic with self-healing that consolidates the necessity of diagnosis to enable targeted self-healing actions.

Traditional Networks	Virtual Networks
Fixed topology	Dynamic topology
One ownership	Multi-tenancy
Physical server hosting one application	Multiple VNFs in one server
Entities: dedicated servers connected with PLs	Entities: VNFs, VLs, PLs, servers, Virtual hosts
One management level	Multi-layered management

Table 2.2 – Traditional vs virtual network architecture features.

The virtualization of networks made a radical change in the composition of the traditional network components, topologies and architectures. Table 2.2 depicts the main NVE features when compared to traditional networks. Most of these features are due to the architectural aspects of virtual networks. In the following we describe main NVE features with regards to the management of networks and in particular fault management.

2.4.1 Virtual networks dependability

The virtualization of networks introduced new types of entities to be managed in the network architecture. The managed entities have different granularity levels. Each layer of virtual networks hosts a number of components. These components can be classified in different ways with regards to their functional type.

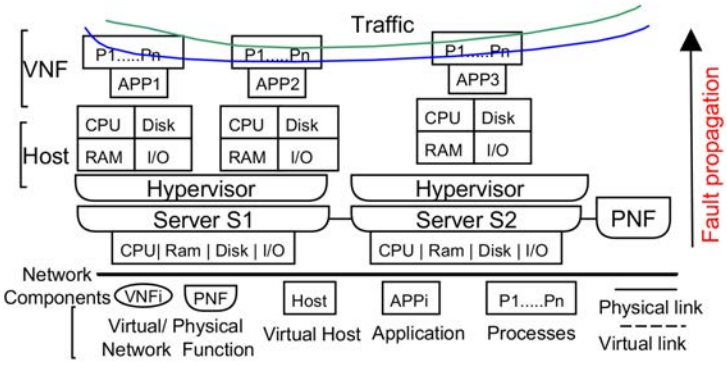


Figure 2.13 – Virtual network Components.

As depicted in Figure 2.13, NVEs entities are divided in two types, *virtual* or *physical*. The managed target could be a *node*, a *link*, a *network* or a *service*. A *node* represents a PNF, a VNF, a virtual host (e.g. container, VM or Unikernel), a site or a link. The *network*, as defined in [32], monitors a set of connected virtual and physical nodes. A *service* is a network of VNFs and PNFs offering a specific application such as VoIP. In the following we depict the different entities of each network layer:

- **Physical layer**: contains the physical hosts. In virtual networks, servers are seen as a set of physical resources: CPU, storage and network. Considering, each resource separately is more efficient when smaller granularity is needed. For instance, in the case of CPU load, considering the CPU entity separately when doing fault management provides more accurate results. PNFs, Physical links and ports are also entities of this layer.
- **Virtual layer**: hosts two main entities: the virtual host and the VIM. For instance, a VM virtual host and an OpenStack VIM entity. The hypervisors could also be considered as a separated entity. Virtual links and ports are classified in this layer. We can also consider a smaller granularity for the virtual host with the virtual computing, storage and network
- **Application layer**: contains the VNFs that represent applications. Each application includes a number of processes.
- **Service layer**: contains the NS that represents a chain of VNFs. Traffic slices and protocols are also contained in this layer.

2.4.2 Physical and virtual coexistence

In the foreseeable future, the most common deployment in telecommunication networks includes mixed VNFs and PNFs. This coexistence define a new kind of dependencies that should be considered when managing virtual networks. A physical server can host a number of virtual functions connected with virtual links. This implies the necessity to consider the management of each VNF hosted in the server separately. Different types of physical and virtual coexistence are possible: a physical server hosting a number of VNFs, a VNF-FG composed of VNFs and PNF, virtual links on top of a physical link and virtual ports on top of a physical port.

2.4.3 Dynamicity of the network topology

The flexibility and programmability of NFV causes an excessive network topology changes. The topological changes are due to network updates and re-configurations. The most common changes are: migration of a VNF, replication of a VNF, instantiation and termination of a VNF. In addition to the dynamicity of the virtual network topology, the network infrastructure is multi-layered. As depicted in Figure 2.13, the NVE architecture is divided into four layers: physical, virtual, application and service layer. Moreover, multi-tenancy implies different kind of membership in one infrastructure. A client can own a server, a VNF-FG or only slices of network services. Which specifies an additional layer to be considered for network management in the case where the network is shared. Figure 2.1 in Section 2.1 illustrates the multi-tenant layer. In this example, two tenants owns different NSs in the same infrastructure. Each NS represents a number of connected VNFs hosted in the physical server. The tenant's NS availability depends on the components composing the NS from the application layer to the physical layer.

2.5 Conclusion

The NVEs consists in a variety of architectures: SDN, NFV, and multi-tenant environments. MNOs benefit from virtualization with the reduction of management and deployment costs, and the programmability and flexibility of services. Moreover, virtualization enables the migration of VNFs between physical servers with the same configuration and without interrupting services. This facilitates the servers management and failure recovery, bringing more resilience to the network. Thanks to virtualization, realistic implementations of hardware-based telecommunication deployments become possible. One example is the *clearwater* vIMS project that enables to run locally an end-to-end service. This provides more flexibility to MNOs, especially if they want to test the system resilience before deployment. However, the NFV architecture is characterized with a number of features such as the dynamic network topology. These features open up new fault management challenges and issues where classical fault management methods

are limited. In the next chapter 3, we propose a comprehensive fault management survey and discuss the issues engendered by the NVE architecture features.

CHAPTER 3

A PROPOSAL FOR A COMPREHENSIVE FAULT MANAGEMENT SURVEY

3.1 Introduction

Fault Management (FM) is the process of locating, analyzing, fixing and reporting network problems such as link failures and network overloads, which in turn makes the network more efficient and productive. Figure 3.1 depicts the FM steps from detection to healing. *Fault diagnosis* aims at achieving three complementary tasks: fault detection, localization and identification [152].

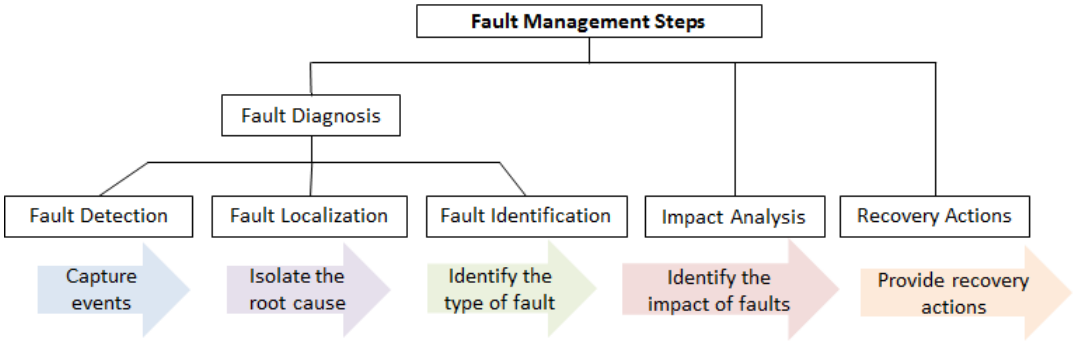


Figure 3.1 – The Fault Management Process.

The introduction of network softwarization enables more services to be deployed with different features and owners, which contributes significantly to the size and complexity of today's networks. In fact, softwarization introduced a number of features such as the network programmability and the dynamicity of network topology addressed in Chapter 2, Section 5.4. Therefore, fault management is becoming increasingly challenging, requiring to early detect faults, inform clients and adopt the necessary healing actions. This chapter addresses our first contribution that represents a comprehensive survey of "traditional" approaches and "novel" techniques to FM (Journal survey [16]). Section 3.2 covers the first part of state of the art, related to the different steps of FM as applied to classical telecommunication networks. Section 3.5 describes the FM issues and discusses the impact of virtualization on the classical

techniques for fault management. Section 3.6 presents recent approaches to FM for virtual networks. The last Section 3.7 positions the thesis work with respect to the presented state of art.

3.2 Classical approaches to fault management overview

This section defines the FM steps. In each step we depict the techniques applied in traditional telecommunication networks.

3.2.1 Fault detection and observation collection (logging)

The fault detection step determines whether the system works in normal conditions or whether a fault has occurred. A *fault* is the *root cause* that may lead the system to an error state. A failure occurs when an error causes a *malfunctioning* of network devices or software leading to symptoms. *Symptoms* are external manifestations of failures. They can be observed as alarms, i.e. notifications of a potential failures [133]. Alarms are notifications send by the system or external agents to express a violation of a metric threshold or an event such as loss of signal. Normal and faulty behaviors may depend on the SLA and the performance requirements of each network service provider.

Faults can be classified in different types depending on their behavior and cause. Faults can be permanent, transient for a short period or repetitive. They can be accidental faults made by humans that are operating or maintaining a system or originate from configuration bugs, or malicious intrusion [45]. Moreover, a fault could range from soft to harsh. *Harsh* faults lead the system to a complete crash while the *soft* faults provoke degradation in one component or more, in one layer or multiple layers. Network administrators use two kinds of data to determine the state of the network: *metrics* and *alarms*.

1. **Metrics**: one way to detect a faulty behavior is by collecting network performance metrics. Metrics represent a quantitative way to verify desired aptitudes and to measure degradations. They measure the activity and health of all the network layers. Network metrics include: delay, jitter, throughput, network utilization, latency and packet losses. For instance, jitter is the inter-packet delay variation, and network utilization is a measure of how much of the capacity is currently in use. In the fault detection process, system level metrics are continuously collected and compared to an acceptable quality level. If the metric measures degradation or violation of SLA a notification is raised. Metrics are present in each layer of the network such as the CPU load in the physical layer, packet loss in the service layer, the connected users, the failed requests, data-base average

response time, etc. Metrics collected in a specific layer may be due to a fault in lower layers. Therefore, metrics of the different layers should be considered jointly in the detection process.

2. **Alarms**: are external manifestations of failures. These notifications may originate from management agents like SNMP traps [127], or in the format of system log files generated by the syslog protocol (or alternative protocols) [42]. The network components auto-monitor their activity by systematically logging their events. Alarms are mostly generated due to the metrics or SLA violations such as latency average passes threshold, packet loss, timeout. Syslog allows devices to send event notification messages over networks to any predefined collector [42]. The information carried within syslog messages may include: the identity of the IP address of the object that generated the alarm, a timestamp, an alarm identifier, a measure of severity of the failure condition and an additional textual description of the failure. Syslog messages are generally stored in logs. These messages contain important information about the health and operation of the system, some are just informative and others present urgent notifications. However, even if alarms provide precious clues about the root cause and the type of fault, some faults may be partially observable, unobservable or hidden. Observable faults are faults that provide notifications when they occur. Unobservable faults may be faults that are not notified due to the lack of management functionality necessary to provide indications of their existence, such as a crash of a physical server that is unable to notify the fault. Unobservable faults might also be faults corrected through an auto-recovery mechanism that removes evidence of this fault occurrence [133]. Hidden faults are masked faults due to other faults that provide more notifications, for example a bug in the application hosted on a VM that is suffering from high CPU load. Since the VM is slow then the packets are not treated by the VM, which hides the fact that the application is erroneous.

Summary:
Fault detection is both an elementary and a difficult task. Metrics may be straightforward and provide directly information about the malfunction. However, alarms are difficult to interpret due to fault propagation. For instance, the root cause may be in an unsupervised part of the network (e.g., lower layers). Moreover, maintenance operations, reboots, upgrades and misconfigurations happen all the time, adding noise to logs, which makes it hard to detect failures that require intervention.

3.2.1.1 Data Mining for detection

Log analysis was first performed by human experts, but due to the growing number of connected elements that implies a large range of monitoring information, automation of log analyt-

ics became crucial. Recently, with the growing capacity of servers and the distribution of data in clouds, storing huge amounts of data — known as big data — became possible. The amount of available data triggered data mining approaches to fault analysis. Data mining refers to the action of extracting pertinent information from large data sets. In the data mining process, the first three steps — data integration, cleaning and selection — are used to prepare the data to apply data mining techniques.

First the data are collected from multiple distributed sources and combined into one data set. **Then, data are cleaned** from noise and only useful data are stored for the mining procedure. For example, Syslog messages that provide debugging information are neglected in the case of fault diagnosis.

A number of efforts [131, 154, 143, 128] proposed ways to unify and exploit syslog logs. For instance the work done in [143] and [128], proposes an organized and generalized way to store syslog messages so that the information collected are used efficiently for the fault localization process. Authors of [128] proposed to generalize the format of the received syslog messages. They defined a number of agents that collect logs and realize the structure of the log collection to finally store it.

Other papers [118, 117] proposed to improve the timeliness and reliability when transporting syslog messages by using the Stream Control Transmission Protocol (SCTP) [93] instead of User Datagram Protocol (UDP) in the earlier version of syslog [42] or Transmission Control Protocol (TCP) proposed in RFC 3195 [93].

Once the data have been prepared, they can be transformed. This stage in the data mining process involves transforming the selected data into appropriate formats (e.g. patterns, features, Key Performance Indicators (KPIs), numerical values) for the mining procedure [122]. **As a final step, data mining techniques like clustering and association analysis are deployed** to discover the interesting patterns and evaluate them to make final decisions. Several efforts [150, 8] used this approach to extract interesting patterns for the anomaly detection in virtualized environments. *Yamanishi et al.* [150] proposed a dynamic syslog mining methodology in order to detect failures among computer devices. The solution took syslog logs as input and used Hidden Markov Models (HMMs) to analyze the logs. HMM is a statistical Markov model[1] in which the system being modeled is assumed to be a Markov process with hidden states. Before applying an HMM modeling, the collected syslog messages were divided into sessions, where each session is a subsequence of events forming a time series. An event is a syslog log line composed of an “Event Severity” that indicates the severity level of the message, two attributes “Att1” and “Att2” that are fields for processes that generated the message, and a “Message” that provides detailed information of the event [150]:

[1] A Markov model is a stochastic model used to model randomly changing systems, where future states depend only on the current state.

ID:	Timestamp:	Event Severity:	Att1	Att2	Message
##:	Nov 13 10:15:00:	WARN:	INTR:	ether2atm:	Eth Slot2L/1 Lock-Up

The syslog sessions are represented by an HMM mixture. An HMM mixture is a linear combination of HMMs where each HMM component corresponds to a syslog behavior pattern and a mixture of K components that represent the different patterns.

Baseman et al. [8], presented a method for anomaly detection based on syslog data collected from VMs. They extract the Infomap clusters on textual data and relational features on numeric data, and combine their features with keyword counts to create a single data set. Infomap represents a hierarchical clustering algorithm based on the probability of a random walker (presented in [121]) to transition between communities in the graph as well as the probability to stay within a community. Each message is assigned to a cluster according to the percentage of its textual tokens contained in each Infomap cluster.

In the open source community the elastic stack [31] is a good reference for logs storing and information extraction. The stack is composed of three main components: Logstash or Beats for data collection, the Elasticsearch engine for data storing and Kibana for extraction and visualization.

3.2.2 Fault localization and identification

After a faulty state has been detected due to an alarm notification or degradation in the system performance, much ambiguity remains to be resolved. Log management in complex networks raises several issues:

- A single failure may generate multiple alarms due to the connectivity and dependency of devices, for instance, a high CPU load in a database generates a number of alarms from all the hosts that try to reach the database. Operators drowned in alarms tend to ignore them until the problem is reported by other measures such as a client complaining;
- False positives in the case of alarms generated during reconfiguration or maintenance operations or caused by the network device state that takes a long time to stabilize (e.g. after equipment reboot);
- Loss, delay, or different time formats of alarms that do not always respect the *ISO8601* standard [63];
- The problem of clock synchronization in distributed systems that affects the order of the received alarms. Causal observation is not always guaranteed (one may observe consequences before causes);

- Ambiguity due to the same alarms or symptoms stemming from two different failures.
- Repetitive alarms due to intermittent faults. This kind of alarms is mostly present in the case of a fixed metric threshold. Since the flows vary in time, alarms are generated each time the metric crosses the threshold.
- Alarm logs contain a lot of noise such as informative alarms and maintenance faults alarms [2].

As the detection process provides only few indications, fault evidence may be inconclusive, inconsistent and incomplete (see survey [133]). Fault localization (also named fault isolation or RCA) represents the procedure of deducing the exact root cause of a failure from a set of alarms, notifications and indications. Fault localization addresses several challenges and issues caused by the ambiguity and inconsistency of alarms and errors propagation [133]. Moreover, fault localization should provide the human operator with clear explanations about the (most likely) root causes and type of faults and their secondary effects (impact), whence the name "alarm correlation and filtering", and should take into account the network topology, the running services, the configuration, and ongoing maintenance operations.

White-box vs black-box approaches:

In Figure 3.2, we provide a new classification of network fault localization techniques. The proposed classification complements the methods presented in survey [133]. Fault localization techniques are classified into white-box and black-box techniques. Figure 3.2 depicts the fault localization methods that have proven to be relevant for RCA of networks. As a definition, white-box techniques use an explicit model of supervised network or system, its topology, service description, etc. The white-box techniques' model can also be fitted to features extracted from data such as fault likelihoods. In black-box techniques, on the other hand, an implicit representation is constructed through a learning process. White-box approaches give an explanation of failures and aim at recovering the propagation of faults by modeling the relationships between nodes, events, alarms and faults. Black-box methods learn a "*function*" or a "*model*"[3] associating observed symptoms (e.g. alarms) to failures from examples, but the learned model is often difficult to understand and explain. Some techniques, such as BNs could be used in both approaches since a learning process might be applied to learn the BN parameters or even the stochastic dependencies from data.

Artificial Intelligence (AI) and particularly the ML branch is getting much attention in the world of telecommunication management since it provides solutions to automatize some of the management tasks. ML uses mathematical models trained on huge sets of labeled data to

[2] Maintenance faults alarms: false positives due to maintenance operations such as reloading a server.

[3] Note that the word "*model*" in black-box is different from white-box approaches. In white-box approaches the model is an explicit structure or graph that express dependencies, while in black-box approaches the model is generally a statistical function learned from data.

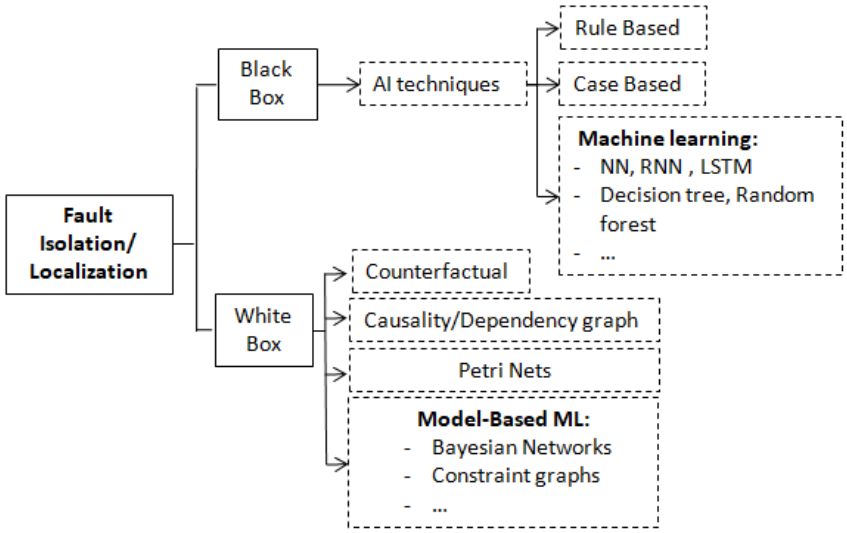


Figure 3.2 – Fault localization approaches.

make decisions without being explicitly programmed to perform this decision task [123]. Black-box ML methods address four main types of problems: clustering alarm patterns, classification, regression and rule extraction [11].

- Classification aims at matching of large vectors of features (the "observations", e.g. alarms and metrics) to a set of discrete output values (the "decision", e.g. fault nature and location). A classification algorithm learns the function that allows classifying new observed data into a set of classes. The input of a classification algorithm is the data features (e.g. severity of a syslog message), the output is a specific class (e.g. faulty state).
- In clustering problems the goal is to gather data into groups for some measure of similarity. A clustering algorithm partitions the input "observations", composed of a number of features, into different clusters. In the clustering algorithm the resulting groups have no classes (or are not labeled), in contrast to classification given to experts for interpretation.
- Rule extraction derives symbolic decision rules from data. This kind of approaches tries to formalize the learned function into decision rules.
- Regression algorithms learn a function that predicts a real-valued variable from a set of training data also under the form of numerical data for example, past values of this variable. The output of regression algorithms is indeed a real number rather than a class such as a linear function.

Methodologies for learning:

Figure 3.3 presents the two different ML methodologies, i.e. black-box and model-based methods. Model-based techniques such as BNs seek to construct an explicit model for the inference

process, whereas black-box ML methods (e.g. NNs) learn the model from training data. The data could be totally, partially, or not labelled, for supervised, semi-supervised and unsupervised learning, respectively. In white-box techniques a model is an explicit representation of the network dependencies that is used for the localization of faults. In the case of black-box techniques it represents the association learned from examples.

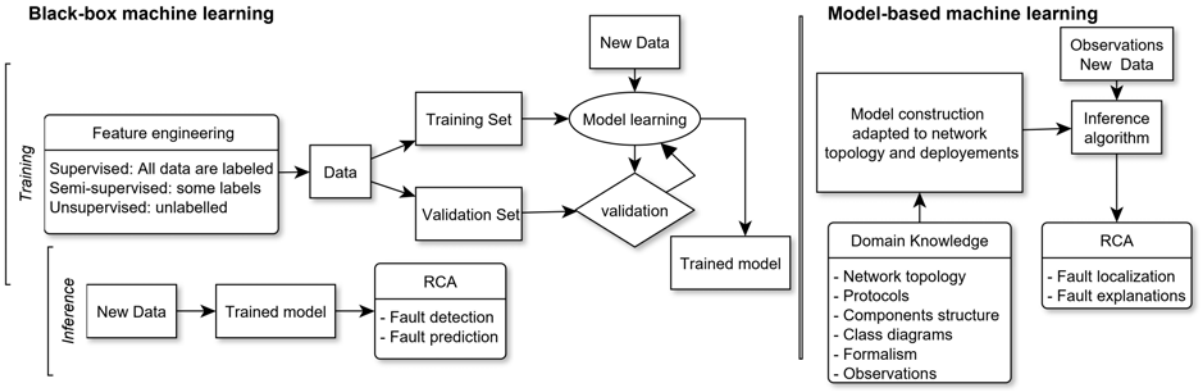


Figure 3.3 – Black-box vs Model-based ML methods.

Techniques in both models (i.e. white-box and black-box) are classified into **fixed** and **dynamic** approaches or both of them. **Static** or **fixed** approaches consider a fixed snapshot of the network state and reason on the state of network as described in the snapshot so the time is assumed frozen, by construct dynamic approaches reason on sequences within a period of time and consider all the events happening during that time. Dynamic methods thus model network behaviors, state changes, or trajectories with stochastic/non stochastic, concurrent or sequential automata. For static models, on the other hand, all consequences of a fault are supposed to have occurred when a snapshot of the state takes place. For dynamic models one may observe fault propagation as it progresses.

3.3 Black-box approaches

Black-box techniques are methods that do not require an explicit system model of the system behavior. Rather, only the inputs (e.g. symptoms) and outputs (e.g. faults) are observed. In the following rule-based and case-based reasoning, decision trees and NN are depicted.

3.3.1 Rule-based and case-based reasoning

Two of the first AI approaches to fault management were rule-based and case-based expert systems. An expert system is a software that enables resolving problems of a specific domain

that is usually performed by human experts. Expert systems consist of three main parts: the working memory with the current input data, the control or inference engine, and a knowledge base that stores the rules and cases. The rule-based techniques describe a number of rules in form of conditions: if <symptoms> then <root cause> [23]. The basic architecture of a rule-based expert system is shown in Figure 3.4-a [23].

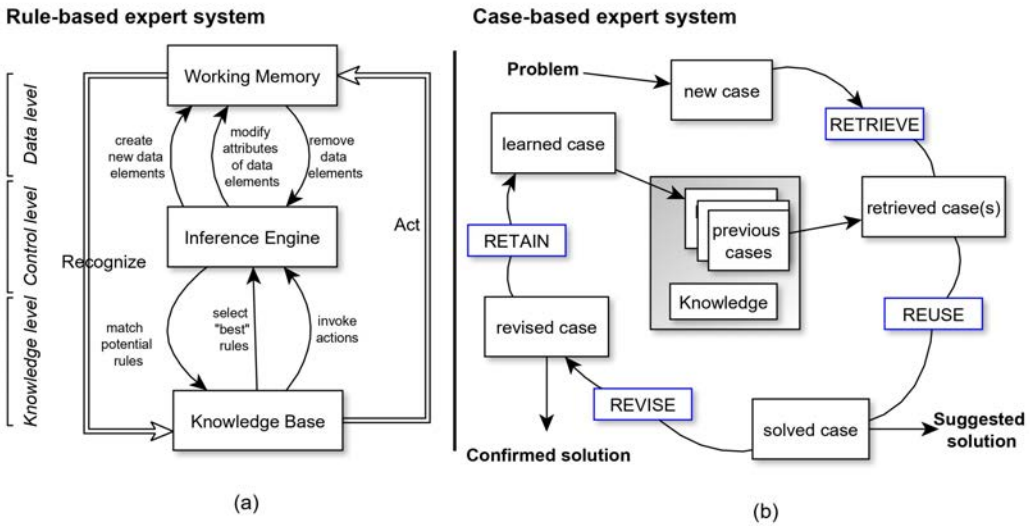


Figure 3.4 – (a) Rule-based [23] and (b) case-based reasoning [1].

Rule-based expert systems perform reasoning using a set of rules derived from knowledge and experience of the domain to be diagnosed. The expert knowledge is formalized as rules stored in a knowledge base. The inference engine determines the best rules to apply considering the current symptoms described in the working memory. The inference engine first finds all the rules that are satisfied by the current contents of working memory, then, determines what are the best rules to apply. This cycle is repeated until no more matches are found.

Another type of expert systems are case based techniques. The case-based approach consists on learning from past experience and past situations. Each time a problem case is solved, the case and its solution are stored on the knowledge base for future use. As illustrated in Figure 3.4-b, the new case solutions pairs are stored in a knowledge base and retrieved to solve the new cases.

Application of rule-based reasoning in the diagnosis of telecommunication networks: The Nokia *Vitrage* project [146] is an OpenStack RCA tool that applies a rule-based approach to deduce faults alarms propagation. *Vitrage* applies a number of rules in a graph that represents the network topology. This graph is constructed through a the *Vitrage* graph building module.

Nodes of this graph represent: the servers or hosts, the VM instances contained in the hosts and the switches attached to the hosts. The edges are expressions with distinct meanings: "On", "Causes", "Contains" and "attached". The graph is updated through the Nova OpenStack module responsible for provisioning and creating the OpenStack VMs.

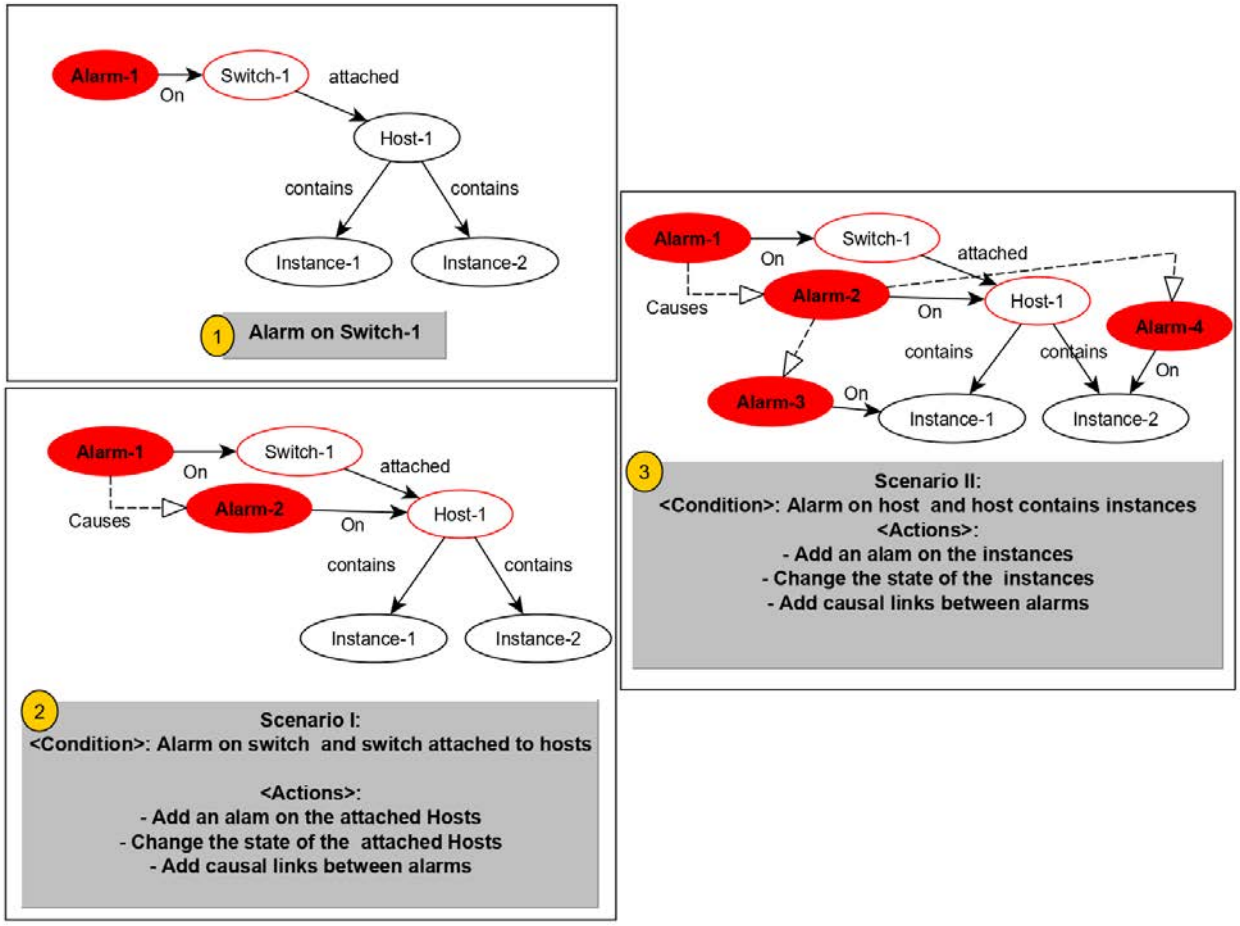


Figure 3.5 – *Vitrage* switch alarm use case.

Figure 3.5 illustrates an example of the topology built by the *Vitrage* graph module. The rules applied by *Vitrage* are designed by network experts. An example of *Vitrage* rules is depicted in Figure 3.5. In this example, an alarm about a switch down is received. The rules are represented as scenarios, each scenario is composed of a <condition> and <actions> to apply for each condition. For instance, if an alarm is raised in "Switch-1" then an alarm is added to all the hosts attached to "Switch-1" (scenario-I). The second action of scenario-I is to change the status of "Host-1" to down. The third action is to add a causality link between the alarms.

In the second scenario (i.e. scenario-II), because Host-1 is down, then all the instances

contained in Host-1 are down. In this scenario-II alarms are attached to instances and linked with a causality relationship. The aim of *Vitrage* is to deduce alarms and rapidly notify faults.

Summary:

Rule-based and case-based techniques are capable of reproducing the cognitive mechanisms of a human expert under the form of facts and rules with a clear separation between the data and the control. However, in the case of a very large domain, the acquisition of the knowledge necessary to constitute the facts and formulating rules can be very difficult and time consuming. Moreover, the number of rules may increase drastically and it becomes very difficult to maintain. Rule-based techniques are very suitable for fixed domains. However, if the domain is very dynamic, certain rules can quickly become obsolete and the rule-based system will become unable to solve certain problems. For instance, the application of rules to monitor virtual networks ([146]), which are dynamic networks is not suitable, since the solution is enable to solve novel types of faults.

In the case, of the case-based techniques the Knowledge acquisition is based on past situations, which enables to learn new cases and adapts the solution based on past experiences. However, case-based techniques still need human expert in the learning loop, which reduces the time efficiency of the process, specially in the case of real-time alarm correlation situations.

3.3.2 Decision trees

A decision tree is a supervised machine learning algorithm used for both classification and regression problems. A decision tree uses a tree-like graph to model decisions and their consequences. The decision tree structure is composed of nodes that denote a test on an attribute of the element to classify, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label. The paths from root to leaf represent classification rules. The Decision graph is composed of:

- **Root node**: represents the entire population or sample.
- **Decision node**: the resulted node from splitting root node or sub-nodes.
- **Leaf or terminal Node**: are nodes that represent the decision.
- **Branch or sub-tree**: a subsection of the entire tree is called branch or sub-tree.

The construction of a decision tree is based on the idea of splitting the set of all possible attributes values into subsets based on a membership test. In the beginning, all the attributes of the source set are considered as root node. Suppose that we have the following binary set (Cf. Figure 3.6):

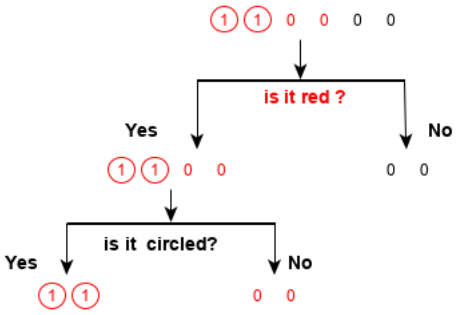


Figure 3.6 – a decision tree sample composed of class "1" and class "0".

The set is composed of two classes of values: class "1" and class "0". We want to separate the class "0" from the class "1" using their attributes: circled or not, red or black. The features are color (red vs black) and whether the observation is circled or not. Asking the question "is it red ?" separates the set into two subsets, or two branches of the tree.

However, one of the decision nodes remains ambiguous since we don't know the class yet. Therefore, the question "is it circled?" in this case separates the two classes. We notice that asking the question "is it circled ?" from the start provides quick results. Therefore, the primary challenge in decision trees is to identify which are the best questions or attributes that we need to consider to split nodes. This can be based on information theory, considering as first splitting attribute the one that maximizes its mutual information with the class value Decision trees use multiple algorithms for node splitting such as the ID3 algorithm.

The ID3 algorithm builds decision trees using a top-down greedy search approach through the space of possible branches with no backtracking. A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. In each step of the ID3 algorithm the attribute applied to split the set S is selected through the value of "the Entropy and Information gain" [4] of this attribute. It then selects the attribute which has the largest Information gain on the class value to guess.

The construction of decision tree classifier does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. However, this method is prone to overfitting, which is the phenomenon in which the learning system tightly fits the given training data so much that it

[4]The entropy is a measure of uncertainty of a random variable. Shannon's entropy is a mathematical function, intuitively corresponding to the amount of information contained or delivered by an information source. This source can be a text written in a given language, an electrical signal or another arbitrary computer file (collection of bytes). From the perspective of a receiver, the more different the source emits information, the more entropy (or uncertainty about what the source emits) is large, and vice versa. The more we receive information on the transmitted message, the more the entropy (uncertainty) with respect to this message decreases, and the information gain increase.

would be inaccurate in predicting the outcomes of the untrained data.

To prevent overfitting, randomized decision trees or random forest are applied. A random forest is an ensemble learning method for both classification and regression that operates by constructing a multitude of decision trees of short depth, operating on a subset of attributes, at training time. In random forest, a random subset of the features are applied in each decision tree. Each decision tree gives a vote for the prediction of target variable. Random forest chooses the prediction that gets the most votes.

Application of decision trees in the diagnosis of telecommunication networks: *Sauvanaud et al.* [126], applied a random forest algorithm to predict the anomalous VMs. *Sauvanaud et al.* [126] proposed an approach to detect SLA violations and preliminary symptoms to identify the anomalous VM at the origin of the detected SLA violation. The data set was collected through fault injection applied on the *Clearwater* vIMS use case. Around 16500 observations for each VM (i.e. Bono, Sprout, and Homestead) are collected. Before the creation of models, the validation data set is shuffled and split into 60% of training data and 40% of testing data. The VM observations are associated with 5 classes (normal behavior, heavy workload, injections in Bono, injections in sprout, injections in homestead). The random forest algorithm is used in order to classify the VNF behaviors. The algorithm is configured with ten decision trees. Moreover, trained random forest models the probabilities of class membership for an input observation. For example, a random forest can output that some feature vector has a probability of 0.6 for it to be in the class "heavy behavior". Given the resulting probabilities, it is then easy to set a threshold defining the limit probability above which an observation corresponds to an "anomalous behavior".

Gonzalez et al. [46], used random forest for the automatic identification of dependencies between system events in virtual networks. To do so, they used as inputs a data set that was obtained from an industrial network composed of 21 devices. Each event is composed of an event identifier, the date when this event is registered, the agent generating the event, its type, description, and severity categorized into: “critical", “major", “minor", and “blank". Critical and major events are more important in the analysis process, since they are the sign of the system malfunctioning. The data set was transformed using a data windowing technique [15]. This technique divides events into an "observation window" containing the observations before the event happens and "prediction window" containing the event. Each moment of time when an event happened is examined as follows: first an observation window is created for it, and the event in question is assigned to the prediction window.

The sequence of events are stored in a table data set, where each row contained the input variables (i.e. how many times each different event had happened on the observation window) and the output variable, a Boolean value indicating whether the objective event had happened in the prediction window. The output classes represent the appearance or absence of the

objective event on the prediction window. For instance, one of the learned behavior between events, is that the virtual switches seem to have a strong influence on themselves and on several virtual machines. This can be explained by the network topology where the failed switches affect the switches and VMs they are linked to.

Summary:

Decision trees are intuitive and straightforward graphical models. They can handle both continuous and categorical attributes and perform well on large data sets. However, decision trees are subject to overfitting. Overfitting means that the model is learning the noise from the data set and its ability to generalize the results is very low, which implies a high variance of errors for new examples (tests). Variance is the variability of model prediction for a given data. Model with high variance perform well on training data and does not generalize on the test data. Random forests reduce the overfitting by using a random subset of features. However, random forests are less interpretable than an individual decision tree since they represent a large number of trees. Moreover, training a large number of deep trees can be costly in terms of CPU and memory. Furthermore, both decision trees and random forests may not fit adapt to changing systems, since the data set is exposed to constant changes.

3.3.3 Neural Network (NN)

Neural Networks (NNs) are composed of more than one layer of neurons which takes in an input and provides an output (Cf. Figure 3.7). Inputs and outputs are variables (e.g. Boolean variable with two values). For instance, inputs could represent symptoms and the outputs are the type of faults that generate this symptoms. NN is composed of one input and one output layer, and a number of hidden layers. The number of hidden layers depends on the complexity of the problem to be solved. A NN having more than one hidden layer is called a DNN. DNNs is defined by (Cf. Figure 3.7):

- L layers, including one input layer and one output layer,
- Each layer has a number of nodes $n \in N$,
- A relation $R \subseteq N \rightarrow N$ for each connected nodes, R represent one of the activation functions described above.
- A weight $w(n_i, n_{i+1})$ between nodes n_i of layer (i) and n_{i+1} of layer $(i+1)$,
- For an activation function $f : R^n \rightarrow R$, the value of node n_{i+1} is $v(n_{i+1}) = \sum_i W_{(n_i,n_{i+1})} v(n_i)$

In NNs, an artificial neuron is the smallest processing unit. It receives one or more inputs. The inputs are weighted and summed with a function called an "activation" or "transfer" function to produce an output. An activation function is a non-linear function usually applied to a layer

output. Its main role is to introduce non-linearity between layers, and thus to avoid factorizing the whole network into a single linear operation. There exist a number of activation functions used in neural networks: step, Rectified Linear Unit (ReLU), tanh, sigmoid and pooling [57]. An example of a neuron is illustrated in Figure 3.8. An artificial neuron considers a linear combinations of the inputs x_i that are transformed using an activation function f.

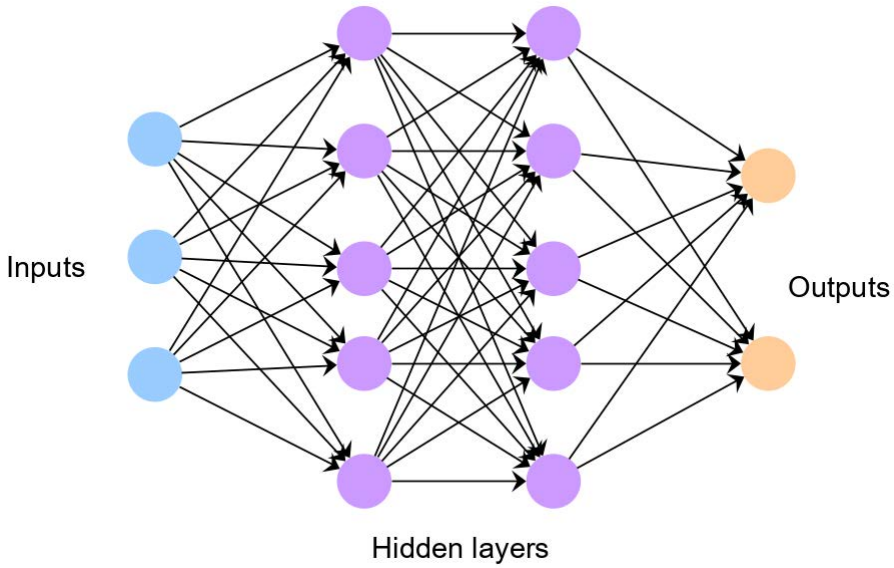


Figure 3.7 – DNN with two hidden layers.

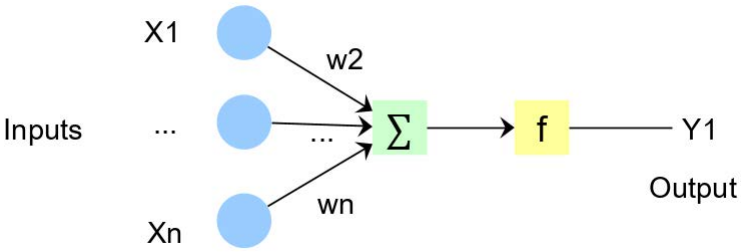


Figure 3.8 – An artificial neuron composition.

NNs learn progressively by considering examples (i.e. inputs and outputs). The learning system represents an association of neurons forming a less or more complex graph. Learning is accomplished by adjusting the connection weights in response to input-output pairs, and training can be done either offline, or online during actual use. The learning process objective is to minimize the loss of a given architecture on the training set. To do so, batches of examples

are processed, the loss function is computed on these examples, and the result gradient error is back propagated throughout the whole architecture to update all weights concurrently. This process is typically split into two main parts: feed forward (i.e. inference), where the output is computed for each input, and back propagation, where the weights are updated. The loss function (also referred to as cost function) evaluates how far the computed output is from the expected one. Common loss functions used to train NNs are mean square error, cross entropy and hinge loss. The trained NNs can then be used to provide outputs for real data (Cf. Figure 3.3).

Recurrent Neural Network (RNN) are a variant of NN designed to handle sequences of feature vectors, for example to predict their next value. Long Short-Term Memory (LSTM) are a special kind of RNN that have the same chain structure as RNN, but instead of having a single neural network layer, there are four interacting layers. LSTMs use gated cells to store information. With these cells, the network can manipulate the information in many ways, including storing information in the cells and reading from them which provides the ability to retain information for a long period of time.

Application of NN in the diagnosis of telecommunication networks:
A number of papers [154, 86, 153], applied NNs and their variants for the diagnosis of faults in telecommunication networks. *Zhaojun et al.* [154], used NNs with the ReLU activation function applied to syslog messages. The syslog message is split into fields such as "Time" and "IP Address" and converted into numerical values to be used as inputs in the learning process. The output layer has two cells corresponding to two output values: y1, y2, which denotes whether a host performs well or not. The outputs are then interpreted with decision-making rules.

Moustapha et al. [86], used RNN to detect faulty nodes in a wireless sensor networks (WSN)[5]. WSN networks connect a wide range of sensors. The defined RNN model nodes correspond to the nodes of WSN topology. The output of the RNN is an estimation of the operation of the WSN that is compared with real WSN behaviors to achieve fault detection. In fact, Sensor nodes can be viewed as dynamic systems with memory that forward information from one node to the next node. *Moustapha et al.* [86], divided the modeling process into two phases the learning phase where the RNN adjusts its weights to healthy or N-faulty models, and the production phase that compares between the output of real WSN and RNN WSN to measure the health status. In the case of new fault type the model is updated with the corresponding parameters.

Zhang et al. [153], apply LSTM to predict faults in virtual networks. The input data represent

[5]WSN are self-configured wireless networks composed of spatially dispersed and dedicated sensors to monitor physical or environmental conditions, such as temperature, sound, vibration and pressure.

a vector of parameters:

$$X(t) = [X^P(t)X^F(t)X^S(t)X^T(t)]$$

$$P : Performance, F : Function, S : Statistical, T : Topology.$$

each of the defined vectors has a number of parameters. For instance, the performance $X^P(t)$ vector is defined as the following:

$$X^P(t) = X^P_B(t), X^P_D(t), X^P_L(t), X^P_S(t),$$

$$X^P_B(t) : bandwidth, X^P_D(t) : network\ delay, X^P_L(t) : packet\ loss\ rate, X^P_S(t) : transmission\ rate.$$

Figure 3.9 illustrates the relation between the network fault data defined as a matrix $X(t)$ and the fault label. At time t, a matrix $X(t)$ which includes the data of time length"l". For history data at time $t-1$, the network fault is $Y(t)$. *Zhang et al.* [153], simulated six types of faults in OPNET network simulator [108]. Results showed that the more serious the faults, the more accurate is the prediction. Prediction time was estimated to 800 s.

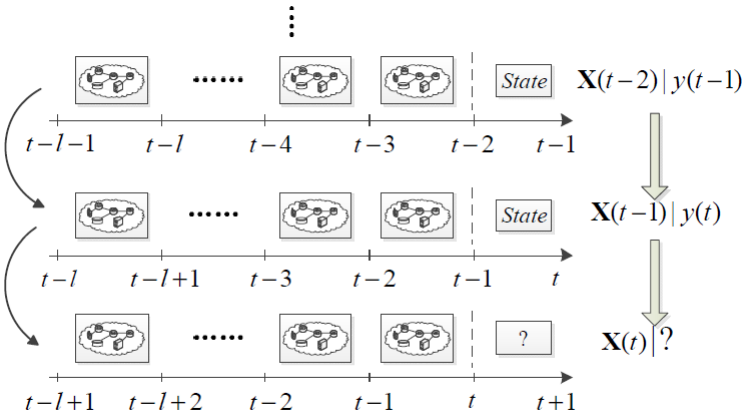


Figure 3.9 – Relation between the network data matrix X and the fault label Y [153].

Summary:

NNs generalize well without requiring a deep understanding of the knowledge domain. They provide a fast and efficient method for patterns matching. However, in the case of RCA NNs can only operate as pattern matching (e.g., match symptoms with fault types) and do not provide explanations.

3.4 White-box approaches

White-box methods can be characterized by four main ingredients, namely the modeling formalism, the modeling procedure, the model and the inference algorithm.

1. **The modeling formalism** represents the mathematical formalism in which the network model will be expressed such as Petri nets, BN and causality graphs.
2. **The modeling procedure** builds one model instance in the appropriate formalism, that matches the current network. It captures numerous aspects such as the network topology, service descriptions, experts knowledge and the formalism applied to the model (e.g. in the case of Bayesian networks we define probabilistic dependencies). The modeling procedure can be extended by an expert, or be fully autonomous algorithm.
3. **The model** is one object in the modeling formalism. For example, a constraint graph. It results from deploying the modeling procedure to a specific network instance. The model relates faults (hidden) to their consequences (secondary faults) and to their symptoms (alarms and metrics).
4. **The inference algorithm** (or solving method) is an event management algorithm used to identify the root cause(s) of a system malfunctions. It uses the model to infer hidden faults and explanations from symptoms.

White-box techniques provide explanations for their decisions and conclusions since each stage in their analysis can be followed and understood. The model can be constructed from different information sources: network resources, dependencies, events, etc. The model can be extended and validated in an interactive way. In fact, explaining faults through an explicit and human readable model enables experts to include observations during the diagnosis process. Nevertheless, the model-based approaches face a number of challenges: the definition of the model construction and its validation, the definition of the modeling algorithm, and the handling of large scale complex networks with different granularity levels.

3.4.1 Causality/dependency graphs

The causality (or dependency) graph is an intuitive representation of the monitored system. It represents the relationship between alarms, intermediate faults and the failures generated, up to visible symptoms. The edges between the nodes represent a causality relation [59]. The network topology and fault propagation knowledge comes from administrators expertise based on hardware and software specifications. Causality dependency graphs are most of the time performed in a static image of the network, also called a snapshot of the topology.

Application of causality/dependency graphs in the diagnosis of telecommunication networks:
Many contributions [76, 56], were based on dependency graphs for the fault localization in networks. The contributions propose variations of causal graphs that differ on the node types and values, edge types and propagation modalities. *Lu et al.* [76], define a causal graph as a Directed Acyclic Graph (DAG) $G = (V; E)$ with five types of nodes, presented in Figure 3.10:

- Primary cause nodes: have no predecessors (root node), are generally called a primary cause and represent the failures one wishes to identify.
- Intermediates cause node: is a node with both predecessors and successors.
- Observations: have no successors (leaf nodes); they represent alarms.
- Test node: they are observations that are not automatically notified and need to be retrieved from the network components.
- Repair nodes: do not take part in the causal relation between causes and effects. Repair actions only serve when the root cause has been localized.

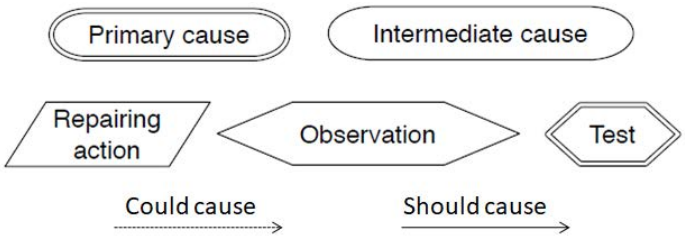


Figure 3.10 – Causal dependency graph components [76].

Edges are causal relations that could be a "should cause" or a "could cause". A "should cause" or a "must cause" is a deterministic cause, for instance if the connection cable between two servers is disabled so the communication between the two servers is broken. A "could cause" or a "may cause" is a possible cause to symptoms, that might happen or not, for instance a high CPU load may lead to a system crash.

Authors of *Lu et al.* [76, 77], start the diagnostic process by involving all the available observations, and determine, on the basis of these observations, the root causes. Each node has a unique state as following: guilty, innocent, suspect and unknown (the default state). A guilty node is a faulty node due to a test result or an active alarm or a proved cause. If the **guilty** node is a root of the causality graph, then this node is the root cause. An **innocent** node is a node working correctly and can be due to a test result. A **suspect** node is a suspected

node to be the root cause (a test to run or an unknown cause to investigate). An **unknown** state is the default state [77].

The guilty and innocent states can be provided by alarms and tests. *Lu et al.* [76, 77], proposes a rule-based approach to infer faults from alarms. Rules are applied until stability of nodes states. The rules are executed in the order of apparition (i.e. from R1, R1,'..,R6). *Lu et al.* [76, 77], defines the propagation rules that follows [77]:
R1 : if a node is guilty, then all its "should cause" successors are guilty.
R1' : if a node is guilty, at least one of its predecessors is guilty.
R2 : if a node is innocent, then all its "should cause" predecessors are innocent.
R2' : if all the predecessors of a node are innocent, the node is innocent.
R3 : if a node is guilty, then all its predecessors become suspect.
R4 : if a node is suspect, then all its "should cause" predecessors are suspect.
R4' : if a node is suspect, then all its "should cause" successors are suspect.
R5 : if a test node is suspect, then the test is performed. Depending of the result, the state becomes guilty or innocent.
R6 : if a repairing action is guilty, then the corresponding repair should take place.

For example, Figure 3.11, illustrates an example of the causal graph of an IMS VoIP service request. In this example, the clients IMPU is checked in the HSS through Subscriber Locator Function (SLF) [6]. In this case, if an unknown client IMPU alarm is received, three root causes are possible: "SLF synchronization failure", "forbidden or baring client's IMPU" or "unknown client's IMPU in the HSS". To find the root cause of the "alarm Unknown IMPU in SLF", the propagation rules are executed as follows:

1. (R1'): As "Alarm unknown IMPU in SLF" is guilty (i.e. alarm present), then at least one of the predecessors "IMPU unknown in SLF" or/and "IMPU barring in SLF" are guilty.
2. (R3): As "Alarm unknown IMPU in SLF" is guilty (alarm present), then "IMPU unknown in SLF" and "IMPU barring in SLF" are suspect.
3. (R4): "IMPU unknown in SLF" is suspect, then its predecessor "Unknown client in SLF (i.e. No account)" is set to suspect.
4. (R4): "Unknown client in SLF (No account)" is suspect, then "Unknown client in HSS (No account)" is set to suspect.
5. (R4): "Unknown client in HSS (No account)" is suspect, then "Test of unknown client" is set to suspect.
6. (R5) "Test of unknown client" is suspect, then the test is performed (in this example the test is the state of "Unknown client in HSS (No account)" is set to guilty).

[6] SLF provides information about the HSS that is associated with a particular user profile.

7. (R1) “Unknown client in HSS (No account)” is guilty, then its "should cause" successors “Unknown client in SLF (No account)” and “Test of unknown client” are guilty.
8. (R1) “Unknown client in SLF (No account)” is guilty, then “IMPU unknown in SLF” is guilty.
9. Achieved stability of node states.

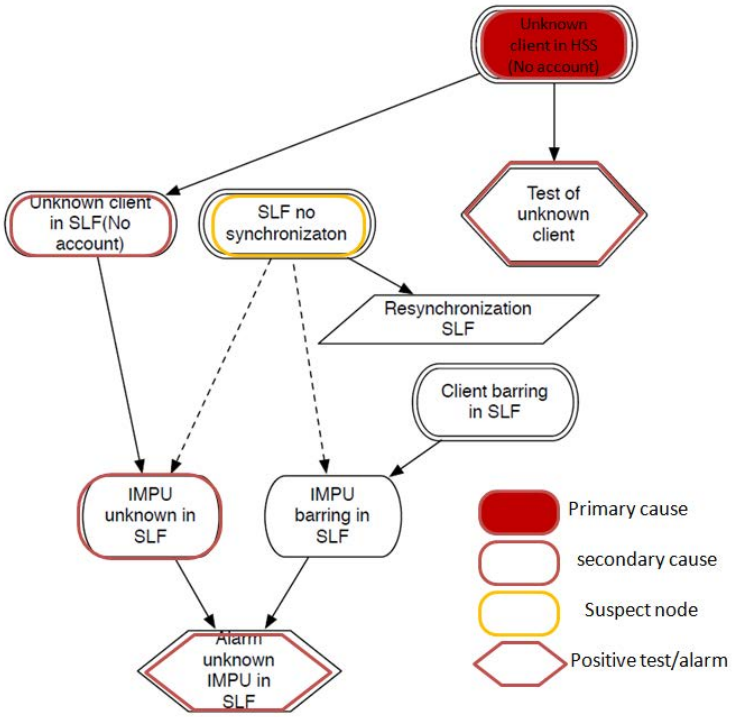


Figure 3.11 – An example of a client IMPU verification causal graph after stabilization of nodes states [76, 77].

Figure 3.11, depicts the last result of the rules propagation. We notice in this example that the failure to synchronize the SLF database and the unknown client failure has the same alarm symptom (i.e. "unknown IMPU"). The distinction between these two causes will be based precisely on the test available.

The separation between these two causes is important to identify the necessary healing actions. For instance, in the case of the SLF synchronization failure a re-synchronization action is needed. In this example, according to rule "R1'", if the alarm "unknown IMPU in SLF" is guilty, then at least one of its predecessors "should cause" is guilty, so either the IMPU is unknown or barring in SLF. If the test about the unknown IMPU is positive, so according to rule "R1" all the “should cause” successors are guilty. In this case, the fault is due to the unknown IMPU, but doesn’t innocent the SLF synchronization node, since there exist a "could cause" between

the "IMPU unknown in SLF" and the "SLF synchronization" node, and according to the rule "R3", if a node is guilty, then all its predecessors become suspect, so in this case, the SLF synchronization remains suspect (Cf. Figure 3.11).

The solution proposed by *Lu et al.* [76, 77], applies an interesting feature in diagnosis by including tests to get more observations. However, the proposed rules return only one possible configuration. Moreover, we notice that, rules R3 and R1' has the same condition and since R3 comes after R1', the decision of R3 is the one considered in the end of the execution (i.e. rule R3 overwrite the results of R1').

Hasan et al. [56] defines the edges between the nodes as logical constraints. For instance, Figure 3.12 illustrates eight nodes and their logical "And" relations. *Hasan et al.* [56] defines causal event C by a set of events E, and a set of equations of the form $e = \phi$ where $e \in E$ and ϕ is an expression constructed using propositional Boolean operations and elements of E as propositional symbols. *Hasan et al.* [56] translates the logical relations relating each node to its successors into a collection of local inference rules that allow to determine the value of a node from that of its successors. For instance, in Figure 3.12, the local inference rule for node "5" is: $5 = 1 \wedge 2 \wedge 3$.

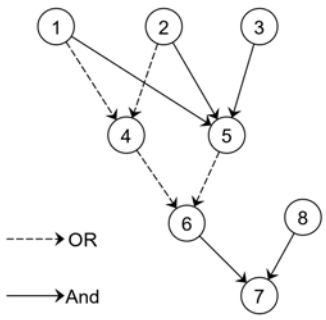


Figure 3.12 – An example of a logical causality graph. [76, 77].

In the work presented by *Lu et al.* [76] and *Hasan et al.* [56], the way of constructing the causality dependency graph from real network topology was not deeply addressed. *Lu et al.* [76], defines causality graphs for IMS, while *Hasan et al.* [56] does not relate its work to any network use case.

Summary:

Dependency graphs are graphical models that enable operators to detect the root causes and provide explanation of what is going on in the network by identifying the consequences of failures, allowing operators to choose best decisions to heal the network. The major difficulty of this approach is the definition of the model for the given domain. Indeed, the construction of the graph requires the acquisition of very precise knowledge about the domain components

dependencies and faults propagation. Moreover, the dependency graph describes a frozen image of the network and can become obsolete if the network is very dynamic.

Another form of reasoning about causation on dependency graphs is *counterfactual reasoning* [7], which is further discussed in the Appendix 7.2.4.

3.4.2 Constraint graphs

A constraint network R is represented as a triple $R = (X, D, \phi)$, with [26]:

- A finite set of variables $X = \{X_1, \ldots, X_n\}$;
- A domain of values for each variable $D = \{D_{X_1}, \ldots, D_{X_n}\}$;
- We denote by $D_v = X_{x_i \in V} D_{X_i}$ for a subset $V \subseteq X$ of variables;
- A set of constraints $\phi = \{\phi_1, \ldots, \phi_t\}$, each ϕ_j operates on subset $V_j \subseteq X$ of variables and specifies which values of D_{V_j} are allowed for the tuple $(X_i)_{X_i \in V_j}$, one has $\phi_j \subseteq D_{V_j}$.

A number of operations can be applied to constraints, we define the "projection" and "join" relations as the following:

- A **projection** of a constraint ϕ operating on V onto a subet $V' \subseteq V$ is obtained by restricting ϕ to variables in V'.
  For $V = V' \bigotimes V'' \subseteq X$, $\pi_{V'}(\phi) = \{v' \in D_{V'} : \exists v'' \in D_{V''}, (v', v'') \in \phi\}$.
- The **join** operator ($\bowtie$) takes two constraints $\phi' \in D_{V'}$ and ϕ" $\in D_{V''}$ and yields a new constraint that consists of tuples of ϕ' and ϕ'' combined on all their common variables in $D_{V' \cup V''}$. We have $\phi' \bowtie \phi'' = \{v \in D_{V' \cup V''} : v_{|V'} \in \phi', v_{|V''} \in \phi''\}$, where $v_{|V'} = \pi_{V'}(v)$.

For instance, given three variables A, B, C with the associated domain values: $D_A = \{a_1, a_2, a_3\}$, $D_B = \{b_1, b_2, b_3\}$ and $D_C = \{c_1, c_2, c_3, c_4\}$. Let $\phi_1 \subseteq D_A \times D_B$ and $\phi_2 \subseteq D_B \times D_C$ defined as:

$$\phi_1 = \{(a_1, b_2), (a_1, b_3), (a_2, b_1), (a_3, b_2)\},$$

$$\phi_2 = \{(b_1, c_1), (b_2, c_3), (b_3, c_1), (b_3, c_2)\}.$$

The projection of ϕ_1 onto $A \in D_A$ consists on the tuples in ϕ_1, with only keeping the values of variable A:

$$\pi_A(\phi_1) = \{a_1, a_2, a_3\} \subseteq D_A.$$

The join of constraints ϕ_1 and ϕ_2 defined as $\phi_3 = \phi_1 \bowtie \phi_2$, with $\phi_3 \in D_A \times D_B \times D_C$, consists of a new constraint ϕ_3 containing tuples that are combinations of pairs of tuples from ϕ_1 and ϕ_2 that share the common variable A.

To do so, we combine each tuple in ϕ_1 with all tuples in ϕ_2 that agree on the values of common variable B :

$$\phi_3 = \phi_1 \bowtie \phi_2 = \{(a_1, b_2, c_1), (a_1, b_3, c_1), (a_2, b_1, c_3), (a_3, b_2, c_2)\}$$

For instance, as illustrated in Table 3.1, tuple $(a_1, b_2) \in \phi_1$ agrees with the tuple $(b_2, c_3) \in \phi_2$ about he the value b_2, resulting the tuple $(a_1, b_2, c_3) \in \phi_3$.

ϕ_1:

A	B
a_1	b_2
a_2	b_3
a_3	b_2

ϕ_2:

B	C
b_1	c_1
b_2	c_3
b_3	c_1
b_3	c_2

$\phi_3 = \phi_1 \bowtie \phi_2$:

A	B	C
a_1	b_2	c_3
a_2	b_3	c_1
a_2	b_3	c_2
a_3	b_2	c_3

Table 3.1 – Example of a Join relation $\phi_1 \bowtie \phi_2$.

A constraint network could be represented in different ways: hypergraphs, primal, dual graphs and trees; as illustrated in Figure 3.13-a, Figure 3.13-b and Figure 3.13-c, respectively. An **hypergraph** is a generalization of a graph in which an edge can join any number of vertices. An edge in an hypergraph is called an "hyperedge". An "hyperedge" in the constraint graph joins the variables linked by a constraint, i.e. for each ϕ_i, V_i defines an hyperedge.

For instance, Figure 3.13-a shows four hyperedges with the following subsets:

$$\{V_1, V_2, V_3, V_4\} = \{\{F, A, E\}, \{A, B, C\}, \{C, D, E\}, \{A, C, D, E\}\}.$$

In the **primal graph** (Cf. Figure 3.13-b), each node represents a variable and the arcs connect all nodes whose variables are included in some constraint scope V_i. So each V_i defines a "clique" i.e. a complete sub-graph.

Dual graphs represent the hyperedges as vertices or clusters, i.e. each hyperedge is replaced by a single node. Nodes are connected if they share variables and the arcs are labeled by the shared variables. For instance, vertices 1 and 2 are linked with the variable "A". **Trees** are acyclic graphs, in which any two vertices are connected by exactly one path (Cf. Figure 3.13-d).

A **Constraint Satisfaction Problem (CSP)** involves finding solutions to a constraint network that satisfy all its constraints. The solutions are assignments of values to the constraint network variables. The resolution of the constraint satisfaction problem, generally holds two steps: a processing phase and a back-tracking procedure. The processing step establishes the local constraints propagation, while the back-tracking procedure produces the appropriate answers [27]. One of the efficient solutions applied to resolve constraint networks is the BP algorithm [113]. In order for this algorithm to converge and work efficiently, it should be applied to tree shaped constraint graphs [27]. Since tree structures or acyclic representations

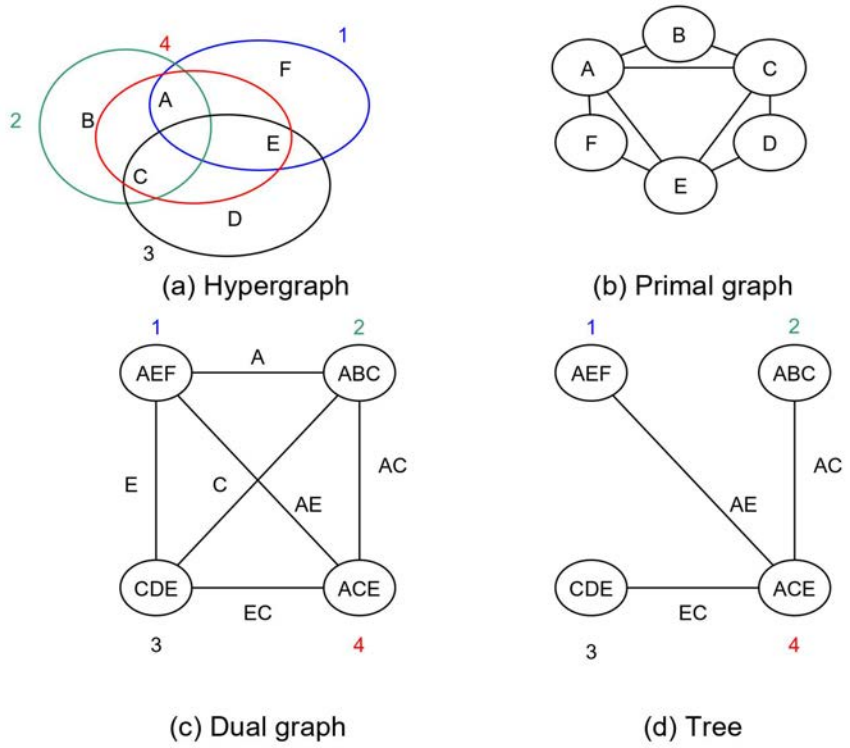


Figure 3.13 – Different ways to represent constraint graphs [26].

of constraints are desirable to execute the Belief propagation algorithm, the first step consists on transforming a constraint network into a tree. This procedure is called Join-Tree Clustering (JTC). The JTC is defined by the following algorithm:

Input: A constraint problem $R = (X, D, \phi)$ and its primal graph $G = (X, E)$.
Output: An equivalent acyclic constraint problem and its join-tree: $T = (X, D, \phi')$
1. Select a variable ordering, $d = (X_1, ..., X_n)$.
2. **Triangulation** (create the induced graph along d and call it $G*$):
For j = n to 1 by -1 do
$E \leftarrow E \cup \{(i, k)|(i, j) \in E, (k, j) \in E\}$
3. Create a join-tree of the induced graph $G*$:

- a. Identify all maximal cliques in the chordal graph (each variable and its parents is a clique). Let $C_1, ..., C_t$ be all such cliques, created going from last variable to first in the ordering.

- b. Create a tree-structure T over the cliques:
  Connect each C_i to a $C_j (j < i)$ with whom it shares largest subset of variables to identify one of its join trees.

4. Place each input constraint in one clique containing its scope, and let ψ_i be the constraint sub-problem associated with the clique C_i.
5. Solve ψ_i to get cliques local solutions.
6. Return the new set of constraints and their join-tree, T.

We explain this algorithm with an example, given a CSP on variables A,B,C,D,E,F, defined by the constraints:

$$\begin{cases} \phi_1 \in D_A \times D_C, \\ \phi_2 \in D_A \times D_D, \\ \phi_3 \in D_B \times D_D, \\ \phi_4 \in D_C \times D_E, \\ \phi_5 \in D_E \times D_D, \\ \phi_6 \in D_C \times D_F. \end{cases}$$

The primal graph of the constraint network is depicted in Figure 3.14, the first step (step-1) of the JTC consists on selecting a random ordering. One possible ordering for the primal graph is illustrated in Figure 3.14-b: d=E,D,C,A,B,F.

The triangulation algorithm [136], consists on choosing a random ordering (step-1) and filling the edges recursively between any two non adjacent nodes that are connected via nodes higher up in the ordering (step-2). In the example of Figure 3.14, the edge between (A,C) and (A,D) fill-in the edge (C,D). The resulted chordal graph is illustrated in Figure 3.14-c. To create the join-tree, we first identify the maximal cliques in the chordal graph (step-3.a) and create the corresponding tree structure.

In the example of Figure 3.14, the maximal cliques are: (E,C,D), (C,D,A), (C,F), (D,B). When connecting the cliques C_i with the ones that shares the maximal variables, we obtain the dual graph in Figure 3.14-d, and the corresponding joint-tree in Figure 3.14-e. The last step of JTC solves the constraints associated to the cliques. For instance, solving the constraint of clique (E, C,D) means finding all the assignments to variables (E, C, D) which are consistent to the input constraint ϕ_4 and ϕ_5 (i.e. solve $\phi_4 \bowtie \phi_5$).

Once a cluster tree has been derived, the inference problem can be solved by the BP algorithm. Let us denote by $(V_1 ... V_n)$ the clusters (organized as tree (Cf. Figure 3.15)), each V_i being the scope of constraint ϕ_i. The objective is to derive the ϕ'_i (reduced constraints) defined by $\phi'_i = \pi_{V_i}(\phi_1 \bowtie \phi_2 \bowtie ... \bowtie \phi_n)$. Each $v_i \in \phi'_i$ is thus the projection (restriction to V_i) of some global solution $x \in D_X$ of the constraint set.

One can easily prove that $\phi'_i = \pi_{V_i}(\phi_i \bowtie \bowtie_{j \in N(i)} \phi'_j)$, where $N(i)$ denotes indexes j such

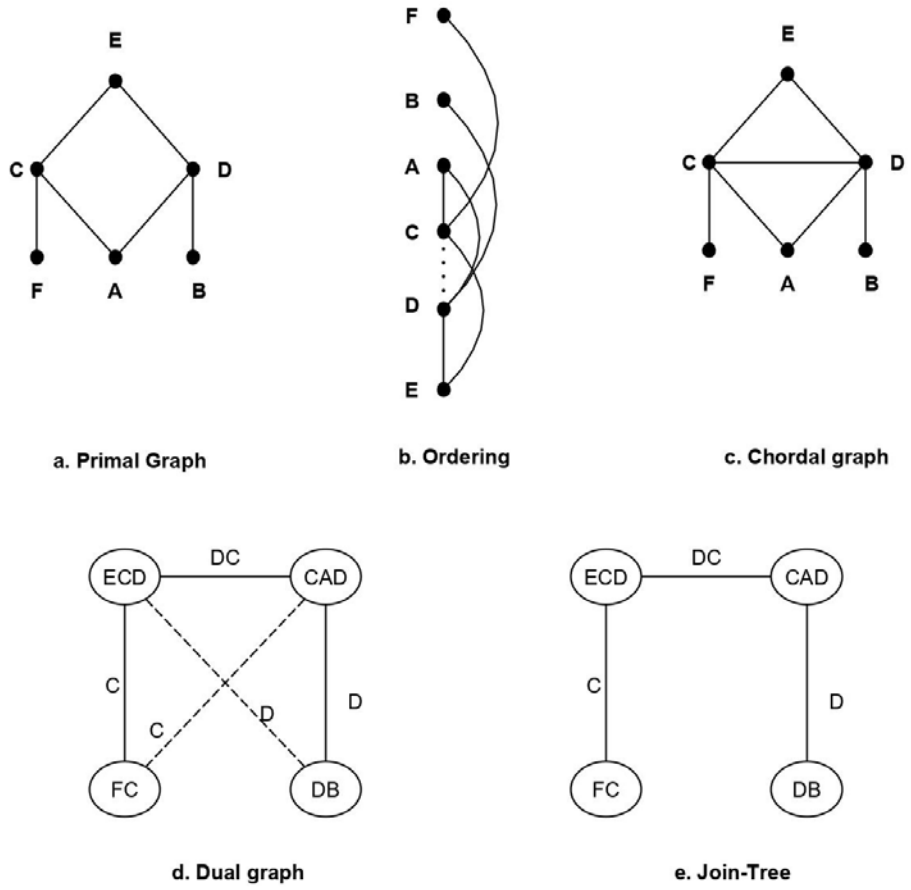


Figure 3.14 – The join-tree clustering procedure.

that V_i and V_j are neighbors on the cluster tree. This fix point equation can be reduced into: $\phi'_i = \phi_i \bowtie \bowtie_{j \in N(i)} \pi_{V_i}(m_{j \to i})$, where $m_{j \to i} \subseteq D_{V_i \cap V_j}$ is a message from cluster j to cluster i summarizing constraints on their shared variables. $m_{j \to i} \triangleq \pi_{V_i \cap V_j}(\bowtie_{k:i-j-k} \phi_k) \subseteq D_{V_i \cap V_j}$ is the "summary" of all the constraints in the clusters that are "behind j" in the tree from the point i.

One can prove that messages satisfy : $m_{j \to i} = \pi_{V_i \cap V_j}(\phi_j \bowtie \bowtie_{k \in N(j)|i} (m_{k \to j})$. This yields an iterative procedure to compute all messages (two per edge, one in each direction) starting from the leaves of the cluster tree.

CSP instances can also be solved by state-of-the-art Boolean Satisfiability Problem (SAT) solvers [78]. A SAT solver is the problem of determining if there exists an interpretation that satisfies a given assumptions (i.e. a set of formula). Efforts [148, 135, 41] proposed transitions from CSP to SAT. One direct transition is to define constraints on variables values as a formula. We associate a propositional variable, zij with each value x_{ij} that can be assigned to the CSP

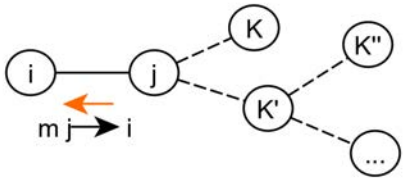


Figure 3.15 – A constraint network.

variable Xi. For instance, if $D_A = \{a_1, a_2\}$ is a CSP variable, we assign to $z_{A1} = a_1$ and $z_{A2} = a_2$. Then, for two varibales $D_A = \{a_1, a_2\}$ and $D_B = \{b_1, b_2\}$ a constraint $\phi_1 = \{(a_1, b_2)\}$, with $z_{A1} = a_1, z_{B1} = b_1$, is defined as follows: $z_{A1} \wedge z_{B1}$. One of the powerful SAT solvers are advanced Satisfiability Modulo Theories (SMT) solvers such as z3 [83] that can solve ten of thousands of constraints and millions of variables [25].

Application of constraint graphs in telecommunication networks:

Few papers applied CSP for the RCA of telecommunication networks. We cite the work of *Gillani et al.* [44]. *Gillani et al.* [44] proposed a fault diagnosis approach based on end-user observations in an overlay network [7]. The end-users share their network performance information which is considered as negative symptoms, and is used to localize performance anomalies and determine the packet loss contribution of each network component.

The problem is formulated as a constraint-satisfaction problem. The assumptions are made of variables defined on overlay components and network paths. An overlay network illustrated in Figure 3.16 [44], is a directed graph composed of overlay components c_j (i.e. routers (r) or overlays (v)) and a path p_k that consists of multiple overlay links. For instance, a path p_1 between n_1 and n_3 is $p_1 = \{v_1, r_1, v_3\}$ (Cf. Figure 3.16). Considering E as negative performance evidences $e_i \in E$ and C as the collection of all components on the network $c_j \in C$. For each negative performance evidence e_i, a *bad* label is assigned to the corresponding overlay path p_k and for each bad overlay path there is at least one anomalous component $c_j \in ei$ along the path dropping packets. *Gillani et al.* [44] defined a model composed of a number of assumptions deduced from the overlay network, we cite the **Evidential Reasoning Invariant** that represents an anomaly scenario. The anomaly scenario explicits a constrained logical relationship between the negative performances E and all its components. The logical relationship implies that there exists at least one bad component in each evidence e_i and can be represented as a Boolean function:

$$\chi(e_i) = \bigvee_{c_j \in e_i} \chi(c_j)$$

[7] An overlay network is a computer network that is built on top of another network. Most of the time, an overlay network is applied to run multiple separate, discrete virtualized network layers on top of the physical network, often providing new applications or security benefits.

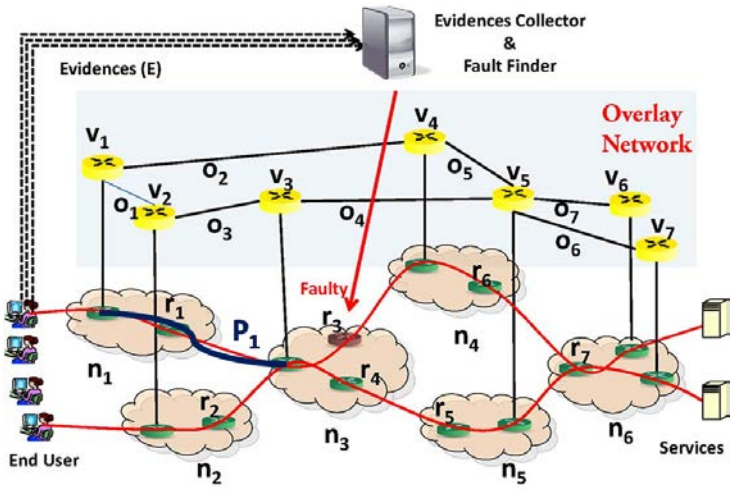


Figure 3.16 – Example of an overlay network model [44].

So E is represented as a Boolean function as follows:

$$\chi(E) = \bigwedge_{e_i \in E} \left(\bigvee_{c_j \in e_i} \chi(c_j) \right)$$

The assumptions are solved by a Z3 SMT solver that generates a single satisfiable solution and forced each time to get more solutions by asserting back the complement of the solution to the model. The solutions represent the set of anomalous scenarios to a given observations.

3.4.3 Bayesian networks

A BN represents a probabilistic extension of constraints graphs, where "hard constraints" are replaced by "soft constraints", weighted by a likelihood. A BN is a DAG that represents cause to effect relationships between (observable or unobservable) events. When a set of symptoms is observable, the most likely causes can be determined [9]. The nodes in the DAG are random variables (e.g. the state of network elements, the occurrence or not of events and faults), while the edges denote existence of statistical relations between the connected variables. The strength of the causality is expressed with conditional probabilities. To construct a BN model a deep human expert knowledge of the cause and effect relationships in the domain is required. This allows humanly understandable RCA explanations compared to black-box techniques [59].

BNs is defined with a set of variables $(X_1 ... X_n)$ and a set of directed edges between variables organized as a DAG; each variable has a finite set of mutually exclusive states; to each variable X_i is attached a conditional probability distribution: $P(X_i|X_{p(i)})$, where $p(i) \subseteq \{1, .., n\}$

denotes indexes of (intermediate) parents of X_i. If $p(i)$ is empty, this conditional probability reduces to the marginal probability $P(X_i)$. The overall joint distribution satisfies: $P(X) \triangleq P(X_1, ..., X_n) = \Pi_{i=1}^{n} P(X_i|X_{p(i)})$.

Some of these variables maybe observable (alarms), others maybe non-observable (faults, state variables). Denoting $X = \{X_1, ..., X_n\}$ the unknown variables and $E = (E_1, ..., E_m)$ the observed ones. Querying the BN consits either in computing posterior marginals $P(X_i|E = e)$, $i = (1, .., n)$, i.e. $P(X_i|E_1 = e_1, ..., E_m = e_m)$ for an observed value $e = (e_1, , .., e_m)$, or better in estimating the most likely value of X for the posterior distribution: $x* = argmax_{x=(x_1,...,x_n)} P(X_1 = x_1, ..., X_n = x_n|E_1 = e_1, .., E_m = e_m)$.

These posteriors $P(X_i|E = e)$ can be derived through a message passing algorithm that extends the one seen for constraint graphs, provided the DAG of the BN forms a tree. Specifically, one has $P(X_i|E = e) \triangleq \pi(X_i).\lambda(X_i))$. Where $\pi(X_i)\, \alpha\, P(X_i|E_{<i} = e_{<i})$, and $\lambda(X_i)\, \alpha\, P(E_{>i} = e_i|X_i)$. $E_{>i}$ denotes evidence variables in $(E_1, ..., E_m)$ that are below i in the DAG, and $E_{<i}$ denotes the complement $E_{<i} = E|E_{>i}$. The λ and π can be computed recursively by message passing. Figure 3.17 illustrates an example of a node A that has two children C and D and a

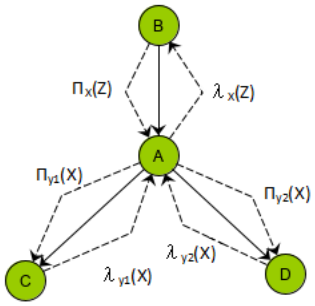


Figure 3.17 – BP on a tree [70].

parent B. In this case:

$$P(e_{>A}|A = a) = \lambda(a) = \sum_{c,d} P(e_{>c}|C = c)P(e_{>d}|D = d)P(c, d|A = a)$$

$$= \sum_{c,d} \lambda(c)\lambda(d)P(c, d|A = a).$$

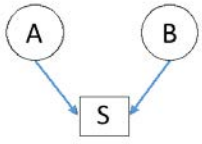
$$P(A = a|e_{<A}) = \pi(a) = \sum_{b} \pi(b)P(a|b).$$

Application of BNs in telecommunication networks:

Some contributions [59, 71, 147] used BN for the RCA of telecommunications networks.

Ktari et al. [71], proposed a Bayesian network modeling for Private Mobile Networks (PMR) networks. PMR networks are designed with a range of protection and recovery capabilities to maintain continuity of voice and data services even under multiple faults and natural disasters such as fire or blackout. PMR networks are composed of different types of redundancy schemes applied to components such as antennas and base station and links. These protection mechanisms cover active redundancy, standby redundancy, load balancing, one for n redundancy, etc. Each of these protection schemes can be modeled simply as a local Bayesian network, relating the probability of each component alone to fail to the probability that the redundant service fails. By assembling the local models of protection mechanisms of all the elements in a PMR, one obtains a larger Bayesian network (BN) model for the whole system. Through classical inference algorithms, one can use this BN both for risk analysis (what is the probability that the overall service fails if a single failure occurs) and for diagnosis (what is the most likely component failure given that this service works and this other one does not).

As an illustration of this approach, one of the redundancy techniques applied in PMR is the active/parallel redundancy. Active redundancy means that two redundant units are operating simultaneously in parallel, rather than having a spare component being switched-on when needed (standby redundancy). Each of the redundant units is capable of handling the full load without sharing with other units. If one of the units fails the other one takes over. *Ktari et al.* [71] models this scheme in the case of two parallel components A and B, with two nodes A and B, and one additional node S that represents the compound service system (Cf. Figure 3.18). The CPT of node S is thus naturally represented as an OR gate, as shown in Figure 3.18.



A	B	S	
		0	1
0	0	1	0
	1	0	1
1	0	0	1
	1	0	1

Figure 3.18 – A Bayesian representation of the active redundancy scheme (left) and the CPT of node S (right) [71].

To localize a faulty component, *Ktari et al.* [71] uses a Bayesian analysis. It is applied to the global Bayesian network (BN) model obtained by assembling all the local protection models for the functions of a PMR. Evidences correspond to the observed state of some functions (working or not), and are propagated through the Bayesian network to resul in probability distributions for each possible initial failure, assuming a single failure occurred. This allows one to deduce the most probable fault. The fault diagnosis was applied on different PMR emergency scenarios

with a high and low Common Cause Failure (CCF) [8]. The results are applied to provide the network administrator a priority checking and maintenance plan for PMR components.

Hounkonnou et al. [59], proposed a self-modeling and diagnostic engine based on the formalism of generic BNs for fault localization in IMS. *Hounkonnou et al.* [59], deal with two main limitations: the generation of the dependency graph and the inference over large dependency graphs. To deal with large dependency graphs the authors define generic Bayesian networks. When a failure occurs in the IMSs service, the algorithm locates the on-line instances of that model in a given network topology and generates the corresponding generic BNs instance. Generic BN (GBN) allow one to build a large BN by assembling smaller (generic) patterns of variables. Figure 3.19 depicts an example of a GBN and three instances sharing one variable "X".

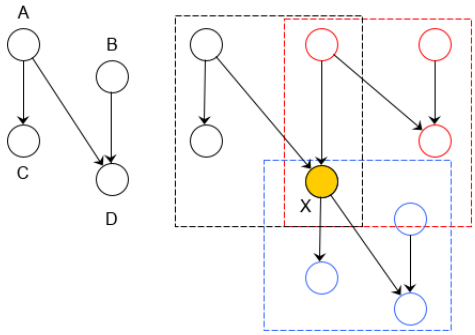


Figure 3.19 – An example of a GBN pattern (left) instantiated three times (right) [60].

For a malfunction of the $user_A$ IMS IP configuration service, illustrated in Figure 3.20. The Dynamic Host Configuration Protocol (DHCP) sever assigns an IP address to a given UE through two main procedures, involving the following entities: UE, Access Relay Function (ARF), Network Access Configuration Function (NACF), and Connectivity Location Function (CLF).

The self-diagnosis steps are defined as the following:

1. Generation of the generic BN describing the resources used in the IP configuration service of $user_A$.

2. Locate BN instances of the same generic model in the IMS network: identify the user's IP configuration service that share the same resources as the affected IP configuration. Figure 3.20, depicts two users "A" and "B" sharing ressources.

[8]Common Cause Failure (CCF), Common Cause Failure, represents the case where a unique initial fault event leads to two or more equipment failures at the same time. The CCF factor is high in the case of a disaster (e.g. fire) where a number of components fail.

3. Perform inference in the current BN instance: The conditional probability of a child given one of its parent is $P = 1 - q$, with $q = 0.1$ is the probability of a node to fail by itself. Given the observations the beliefs are propagated through the network.

4. If uncertainty is high explore and add other patterns: the authors of [60], calculate the entropy of the a posterior distribution to quantify how uncertain the root cause is. If the root cause previously calculated over the current BN is not pinpointed with enough confidence, they propose to extend the BN with other IP configuration instances that share the same resources as the failed instance) to the current BN, in order to collect more evidence (observations, alarms) and solve the case.

5. Repeat the expansion until confidence in the explanation becomes sufficient.

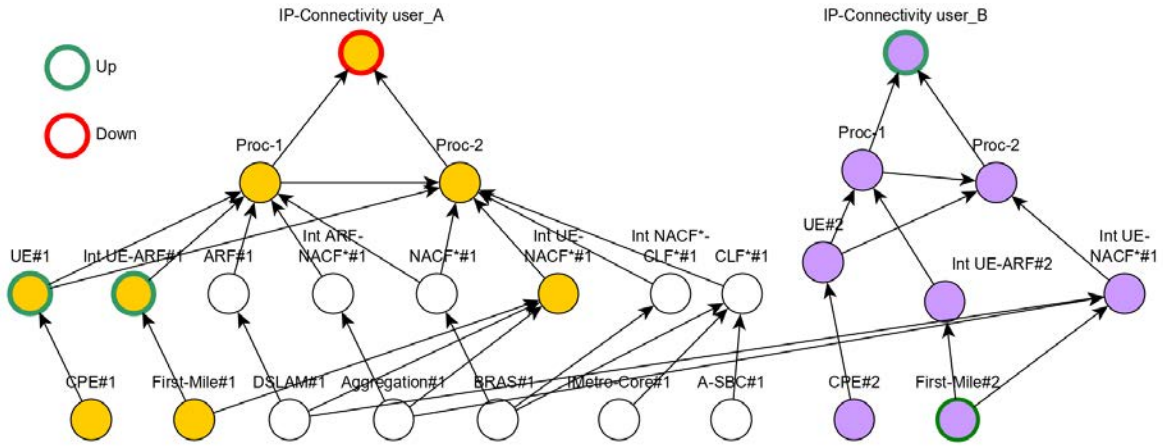


Figure 3.20 – Two users BN instances for IP configuration procedure sharing resources [60].

Sánchez et al. [147], proposed a self-modeling based diagnosis approach using Bayesian Networks for SDN. The proposed BN model defines the physical, logical, and virtual network resources and their sub-components (i.e. CPU, ports, application, and its associated configuration) as binary variables, with state ('down' or 'up') and the edges represent the dependencies among network components. The authors of [147], applies the BP algorithm to propagate evidence or observations on the defined BN model through the graph based on the CPT until it reaches the root vertices. Figure 3.21 depicts an example of a failure in the SDN controller. The SDN controller does not respond to the ping. Thus, the observations about the controller's ports down are added to the BN engine. The BN engine determines that the most probable root cause is the controller with probability 94.2%, with an equal probability (31.4 %) for CPU, the Floodlight SDN application and configuration. Which specifies three possible root causes with no more details or explanations.

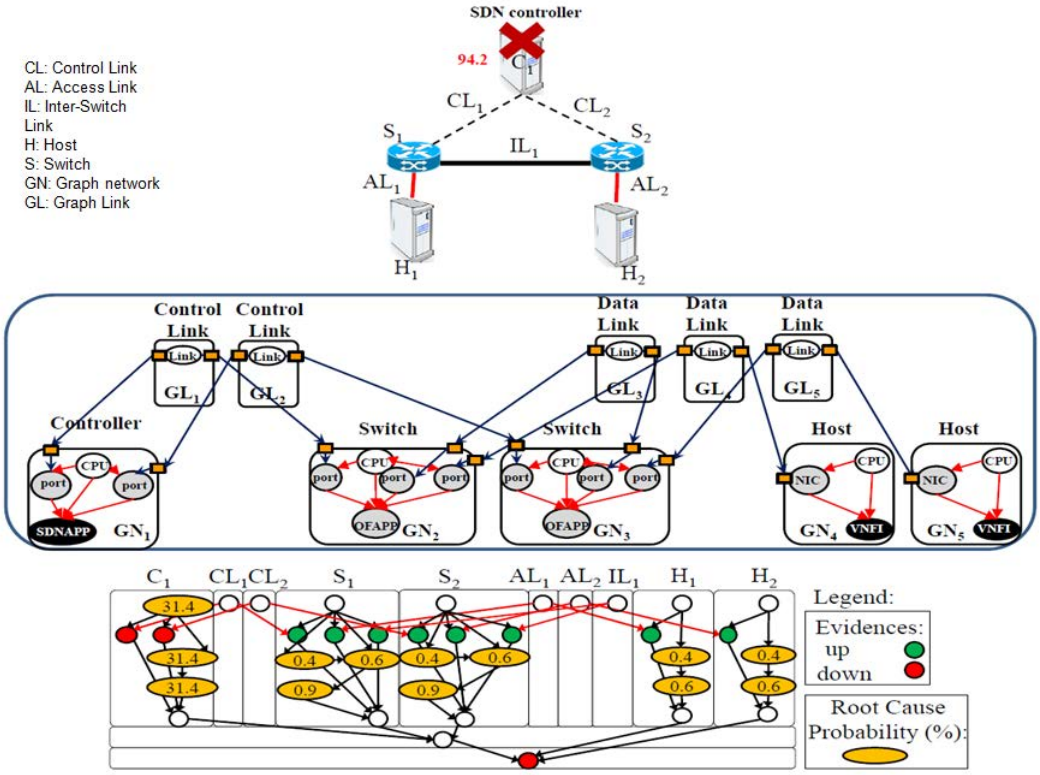


Figure 3.21 – An example of a controller failure BN dependency graph [147].

3.4.4 Petri nets

Petri nets are a basic model of dynamic and distributed systems that describe state changes in a system with transitions and places. Places allow you to represent the states of the system, while transitions represent the set of events that change the system state. An ordinary Petri net is a 4-tuple $N = (P, T, W^-, W^+, m_0)$, where:

1. P is a finite set of places;
2. T is a finite set of transitions;
3. $P \cap T = \varnothing$, $P \cup T \neq \varnothing$;
4. $W^- \in [P \times T \to \mathbb{N}]$, $W^+ \in [T \times P \to \mathbb{N}]$, are the pre and post arc weight mapping;
5. $m_0 \in [P \to \mathbb{N}]$, is a finite multi-set on P representing the initial marking of the Petri net, i.e. the number of tokens that each place initially holds.

Each place p holds a number of tokens that specifies the state of the system. The occurrence of an event corresponds to a transition t. The transitions depend on the availability of

input resources, in places connected to these transitions, transitions consume these ressources and produce others in return.

A marking M is a function $M : P \rightarrow \mathbb{N}$, such as $\forall p \in P$, $M(p)$ is the number of possible tokens into place p. A transition $t \in T$ is enabled (or activated) in marking M, iff $\forall p \in P, M(p) \geq W(p,t)$. Firing t from M yields the new marking M' such as that $\forall p \in P, M'(p) = M(p) - W^-(p,t) + W^+(t,p)$.

The use of a Petri nets is very often associated with its marking graph. The graph of the markings of a Petri net N, is a directed graph whose nodes are the markings of N, and each arc links a marking to another which is immediately accessible by one transition firing. If a transition t_n causes the system to go from state M_n to state M_{n+1}, then an arc is added between the markings M_n and M_{n+1}. However, the construction of the graph of markings requires an exhaustive enumeration of all the possible system states.

Petri nets enable to define a step semantics, which better expresses the concurrent behaviors. In step semantics, one allows a multiset of transitions to fire simultaneously (i.e. a multiset S is enabled in marking M if M contains enough tokens to fire all transitions in S).

Moreover, Petri net markings can be limited to no more than K tokens: let N be a Petri net, N is said to be k**-safe**, if no reachable marking of N can contain more than k tokens in any place ($k \geq 0$).

Figure 3.22 illustrates a successive marking of a Petri net example. In this example, places are represented as circles, and transitions as flat rectangles, while the arrows stand for W^- and W^+. Tokens are represented as black dots places. The initial marking is represented on the left, with a dot/token in both P0 and P3 : transitions t1, t2 can fire simultaneously, since they don't require the same resources. The result is the marking presented in the right. Observe that t3 and t4 are then exclusive as they compete for the token in P3.

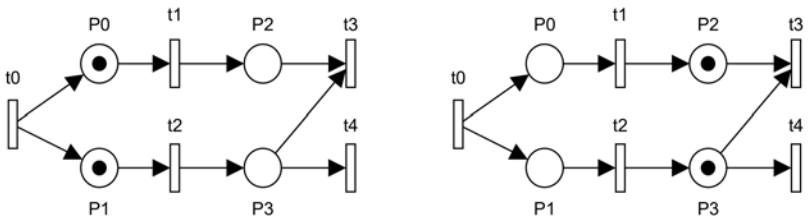


Figure 3.22 – An example of a Petri net with four transitions.

Application of Petri nets in telecommunication networks:

Petri nets were applied in the diagnosis of telecommunication networks. *Boubour et al.* [10], applied 1-safe Petri nets to represent failure propagations in telecommunication networks. In

the model defined by *Boubour et al.* [10], places represent failures occurence and the transition of a failure to a secondary one is represented by a transition with the associated alarm pattern so the nets are acyclic. The capacity of places is one token because of the nature of faults (i.e. present or absent). Figure 3.23 illustrates the spontaneous and persistent faults modeled with 1-safe Peri nets. A spontaneous fault is not the result of a fault propagation phenomenon, so it is represented with the first transition t0 with no input, while a persistent fault is a repeated fault illustrated by a cycle, where P1 is the failure and t1 represents the transition of failure P1 to the same failure (i.e cycle) [10]. *Boubour et al.* [10], applied Petri nets to Synchronous Data

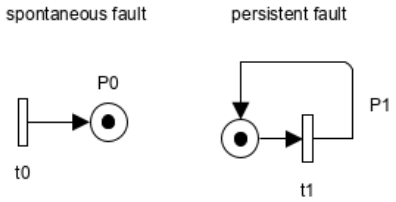


Figure 3.23 – Spontaneous and persistent faults modeled by Petri nets [10].

Hierarchy (SDH)[9]. An SDH signal is constructed from Synchronous Transport Module (STM)-1 frames. Figure 3.24 illustrates a part of an SDH data network and its associated management network. The network is composed of three sensors (s_1, s_2, s_3) associated to the SDH sites. The network elements (i.e. STM-16, DG) are connected via bi-directional connections. Each of these elements contains STM-1 ports. A number of components are associated to each port: Synchronous Physical Interface (SPI), Multiplexage Section (MS), Administrative Unit (AU), etc. In this example, the STM-1 port P in site DG is disabled. In this architecture, alarms go from network components (SPI, MS, AU) through the sensors s_i to the supervisor "S".

Figure 3.25 shows the partial order propagation of faults and alarms due to the occurrence of a LOSs on a port "P" of the STM-1 module, the corespondent Petri net is illustrated in Figure 3.26.

The detection of fault will then consists in matching alarm patterns to a given observation. A diagnostic could be provided in the form of a backward tracing of the fault net according to the alarm pattern.

Varga et al. [145], used Petri nets to detect network faults. Petri nets are applied in a different way than *Boubour et al.* [10]. *Varga et al.* [145], defines Petri nets as action plans to find the root cause of an alarm. Each alarm has its associated Petri net. An RCA scheduler is applied to choose the appropriate Petri net model for scheduling the analysis based on

[9]SDH is a standardized protocol that transfers multiple digital bit streams synchronously over optical fiber using lasers or highly coherent light networks

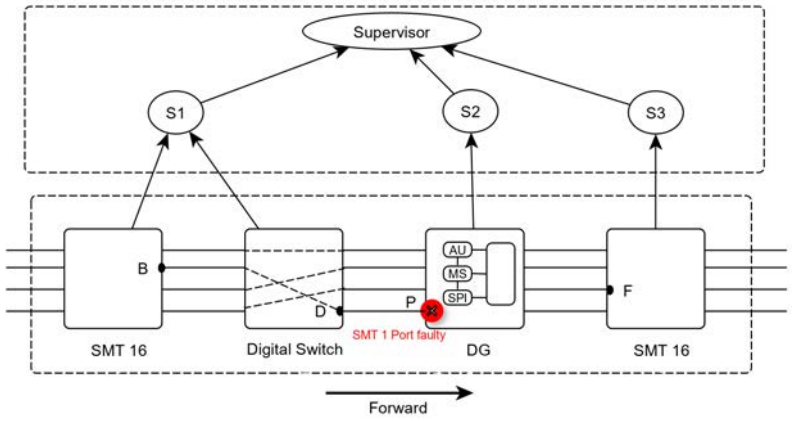


Figure 3.24 – A signal loss failure propagation through port P of SMT-1 [10].

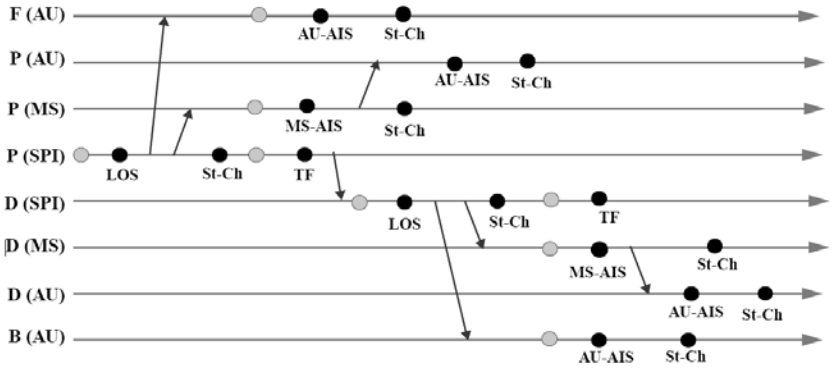


Figure 3.25 – A signal loss failure propagation through port P of SMT-1 [10].

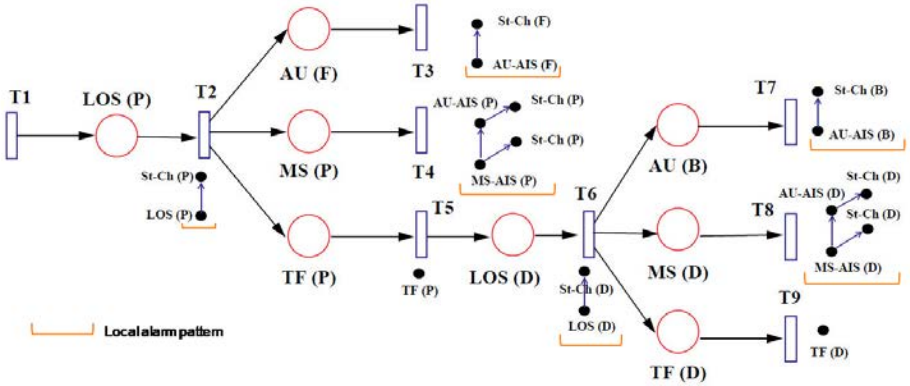


Figure 3.26 – Petri nets modeling propagation of faults resulting from a LOS and the alarms that results from these faults [10].

the alarm type. In the defined Petri net model, the transitions are elementary investigation checks (e.g. a port state check), while the places represent the input and output results of an elementary check.

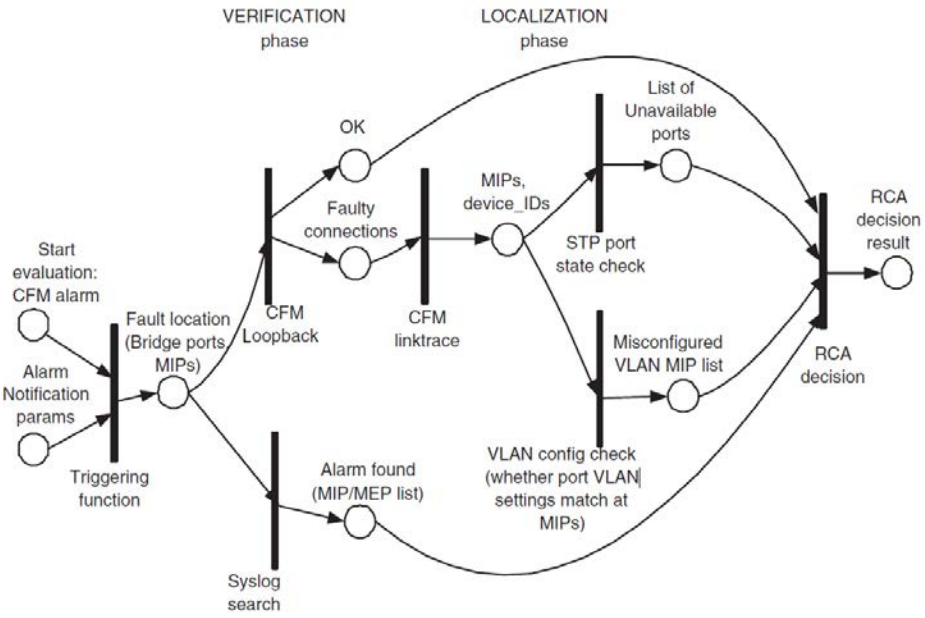


Figure 3.27 – Connectivity failure Petri net model [145].

When an alarm is raised (e.g. a connectivity fault), the scheduler starts running the associated Petri net by adding a token marking in the appropriate place. For instance, Figure 3.27, depicts the Petri net model for an alarm indicating a loss of connectivity between two devices. The scheduler starts running this Petri nets by adding a token in both "Alarm Notification params" and "Start evaluation CFM alarm". In the upper branch of verification the CFM or IEEE 802.1ag Loopback (i.e. ping) and linktrace (i.e. traceroot) functions are used as RCA elementary checks, while in the lower branch a search for a Syslog event such as ”interface down" errors is also launched. In this example, each port has Maintenance End Points (MEPs) and Maintenance Intermediate Points (MIPs)[10]. The RCA decision result depends on the results obtained from the transitions through "RCA decision". Once the Petri net in Figure 3.27 is terminated the expert will get a number of checks such as the "list of unavailable ports", according to this checks the expert will make decisions.

Variations of Petri nets were defined in previous works: *Guerraz et al.* [50], proposes timed Petri nets. In timed Petri nets the form of partially ordered transition is related to time constraints, while *Fabre et al.* [38], proposes a distribution of the diagnosis procedures by modeling

[10]MIP and MEP - are software (or Hardware) entities applied for connectivity fault management

the fault propagations as distributed Petri nets. This work is complementary to the work done in *Boubour et al.* [10], with a distribution of the diagnosis of the alarms collected on different supervisors that are communicating asynchronously. *Aghasaryan et al.* [3] proposed probabilistic Petri nets to complement the work of *Boubour et al.* [10]. The use of probabilities enables to deal with uncertainties due to poor knowledge such as loss of alarms and to provide robustness in the diagnostic process based on an extension of the algorithm of Viterbi to calculate the most likely path in the probabilistic Petri net.

Petri nets are dynamic systems that can only encode sequences of events. They have the the advantage of being able to handle multiple operations to represent dynamic systems. However, the application of Petri nets to model alarm propagation in networks suffer from alarm loss. Moreover, the architecture of Petri nets is difficult to scale and generalize in the case of large dynamic systems.

3.5 Limitations of classical fault management approaches

NVEs offer a number of benefits to MNOs including the reduction of management and deployment costs and the programmability and flexibility of services. However, NFVs will inherit a number of vulnerabilities that limited the efficiency of the existing fault management techniques. The requirements of fault management is remarkably changing, the management system should take into consideration the various granularity including the logical and virtual resources. But also the dynamic network topology, the distribution of the tenant resources in different locations, and the hardware and software dependencies. In this section we describe a number of NVE issues that we addressed in our survey [16].

1. Fault detection issues:
   Fault detection facing alarm storms: the growing number of services in NFV implies more entities to manage with a growing number of faults alarms. The alarms may originate from different network layers and resource types. For instance, these alarms can be traditional OS Syslog messages or other specific alarms defined for each NFV service. The distribution of VNFs over sites provides more robustness to the network. However, this brings a new management concern: whether to distribute or centralize the logs and the management system. Furthermore, in NFV a failure may propagate through layers and sites, increasing the number of alarms. Dealing with alarm storms requires modern storing and filtering techniques with early notifications of severe failures [45].

   Distribution and consistency of logs: in the detection and localization process, events in the form of logs or metrics are collected to identify the failures. These events contain

important information about the health and operations of the system. However, the collected data originate from distinct sources with different formats and are most of the time ambiguous and full of insignificant information for the diagnosis of faults. In addition to this complexity, in NVE the distribution of VNFs of the same service causes a lack of visibility and implies the centralization of logs generated from the different infrastructures where the VNFs are hosted. Moreover, sharing the MNOs architecture with clients arises challenging questions such as the separation of clients alarms, notification of tenants, lack of visibility due to the clients' access restriction to the owner infrastructure, and sharing the management task between infrastructure owners and their clients [28]. Figure 3.28 illustrates the multi-tenant fault isolation issue. In this example, a hardware server crash affects two VNFs (i.e. VNF1 and VNF2) of tenant T1 and T3 respectively. In this case, a rapid isolation of faults and notification of the two tenants is crucial to provide necessary recovery actions.

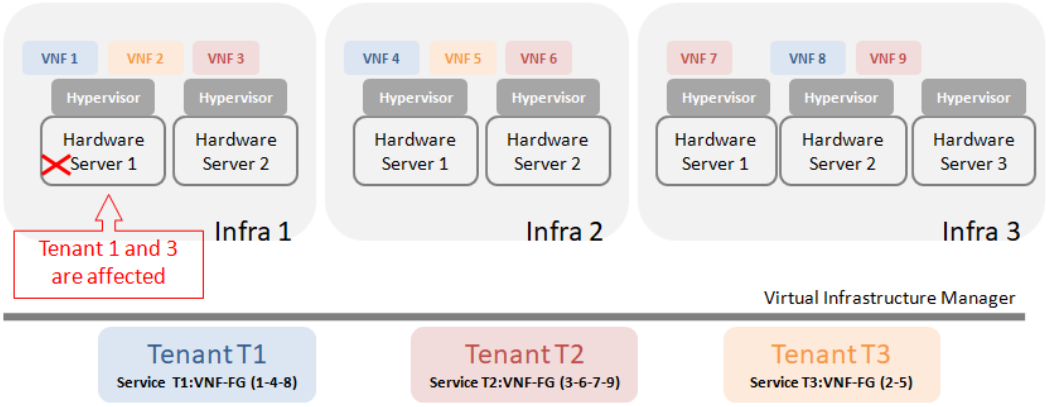


Figure 3.28 – Multi-tenant's Fault notification ML methods.

2. Fault localization issues:
   Table 3.2, depict the advantages, the disadvantages and the limitation of the application of black-box and white-box approaches to virtual networks fault management. In the following, we describe these limitations.

   Fault localization white-box approaches issues: Model-based techniques use models that fit the network topology, over accurate and wide range reasoning techniques to provide explanation about failures. These approaches have two weaknesses: deriving an accurate model and dealing with huge models [60]. This two weakness became greater when applied to virtual networks. In fact, the flexibility of NFV enables constant topology changes and components relocation in the network. Considering the actual network topology when a fault occurs is crucial to avoid false positives, i.e. pinpoint a faulty net-

	Approaches	Advantages	disadvantages and limitations to virtual networks fault localization
Black-box	Rule-based	- Rules are easy to extract for known problems - Clear separation between the data and the control	- System brittleness (no explanation for novel virtual networks faults) - Suitable only for fixed domain - Rules or knowledge acquisition bottleneck
	Case-based	- Learn from experience - Address novel problems	- No automation of learning loop - Real-time alarm correlation
	Decision trees/ Random Forest	- Interpretable but less explanations than model-based - Handle both continuous and categorical attributes - Perform well on large data sets	- Intensive training - Overfitting - Outdated data for virtual networks dynamic topologies
	NN/ RNN/ LSTM	- Fast and efficient - Handle incomplete, ambiguous and imperfect data	- Intensive training - No explanation for the solution provided - Outdated data for virtual networks dynamic topologies - Learn known dependencies
White-box	Causality/ dependency graphs	- Capture the dependencies between resources - Explain what happened as consequences of these failures	- Deterministic relations - Additive effects of failures - Difficult to model a complex multi-layer architecture -Fit the dynamic topology
	Constraint graphs / Bayesian networks	- Provide explanation about their solutions - Classic inference algorithms - Large scale models - Possibility to learn and extend the model	- Difficult to Model a complex multi-layer architecture - Fit the dynamic topology
	Petri nets	- Handle multiple operations to represent dynamic systems	-Difficult to scale and generalize in the case of large dynamic systems

Table 3.2 – Advantages, disadvantages and limitations of classical fault localization approaches to face virtual networks challenges.

work component that is not running anymore in the network. Moreover, the coexistence of physical and virtual entities in the NVE architecture entails new types of dependencies that should be considered when building the model. In addition, in virtual networks the topology is multi-layered. Therefore, while building the model, one should take into consideration the different management levels and consider the dependencies between layers. Moreover, for a good accuracy of fault localization, one should consider a finer granularity (e.g. application processes) while defining the modeling rules.

Fault localization black-box approaches issues:

Black-box approaches learn fault-symptoms associations from data. For a good accuracy of the learning process, the training data set must be consistent and contain most of the system features. However, in virtual network, the dynamicity of the network topology may obsolete the data set applied in the learning process. For instance, when applying random forest to classify the states (e.g. normal or faulty state), suppose that the data set used for training consider a snapshot of the network topology, while in virtual networks the components change constantly a replication of a network component may cause false positives. Moreover, black-box techniques pinpoint faulty components without providing explanations which are crucial for self-healing actions. For instance, a NN that pinpoints a VM as faulty while only a process of the application hosted in this VM is in the "stopped" status. In this situation a simple "run" of this process will solve the problem.

3.6 Novel fault management techniques

In this section we discuss current efforts to tackle the NVEs issues. The recent efforts in fault management involve automatic actions in their solutions, introducing new concepts and reviving old ones that became feasible and useful in NVEs.

3.6.1 Self-modeling

To perform fault detection in most of the model-based techniques building the model is an important step toward fault diagnosis and healing. In earlier networks the construction of the model was much easier; the network was much smaller with a fixed topology, only standards and expert knowledge were sufficient to build the model. However, with virtualization, the topology becomes dynamic which complicates the definition of a model that fits the network topology. The defined model should be generic in a way that it can be instantiated automatically to match the topological changes.

In NVE, few efforts [134, 146] proposed to model virtual networks. *Sánchez et al.* [134] proposed the use of the topology description extracted from the SDN controller. The defined model is a dependency model that is applied for diagnosis using Bayesian networks (Cf. Section 3.4.3). To model SDN components, *Sánchez et al.* [134], defines two types of templates: the network node template (Nt) and link template (Lt). These templates represent the relationships between virtual and physical sub-components (e.g. CPU and ports) inside each network element.

Sánchez et al. [134], propose to build the network dependency graph with four steps: the network topology interpreter, the dependency sub-graph instantiation algorithm, the topological sorting algorithm and the edge addition algorithm. In the first step, the **network topology interpreter** extracts the network topology from the SDN controller and generates descriptors that classify the network nodes in the following types: controllers, switches, hosts, control links, access links and inter switch links. The **dependency sub-graph instantiation algorithm** is then used to model the network according to the network descriptor. The algorithm first Identifies the type of network element (node or link) and then instantiates its corresponding template according to the type of element. The dependency sub-graph of the instantiated network element is added to the global network dependency graph. The last two steps (i.e. the topological sorting algorithm and the edge addition algorithm) are applied to sort the dependency graph to avoid repeated vertices between the connected sub-graphs and to add the missing edges between sub-graphs.

Figure 3.29 depicts the different steps of generating a dependency graph. In Figure 3.29, the Nt and Lt descriptors are modelled into a dependency graph.

Vitrage (addressed in Section 3.3.1) uses a rule-based technique applied on a topology graph that represents the OpenStack resources. The topology graph is build by an entity graph module as illustrated in Figure 3.30. In this topology graph (Figure 3.31), the VMs are connected to the hosts with a "contains" relationship and switches to hosts with an "attached" relationship. The information about the resources is extracted directly from the OpenStack module *Nova*. *Vitrage* also represents the alarms and deduced alarms in the model. The notification about an alarm is received from open source monitoring tools (Nagios and Zabbix). The deduced alarms results from the rules defined by *Vitrage* templates and evaluated by the *Evaluator* module in Figure 3.30. The alarms are attached with a link "On" to the failed resources and "causes" link type to indicate the propagation of faults.

Suppose that a new VM is instantiated. The procedure of adding this instance to the graph is described as follows (Cf. Figure 3.31): first the *Nova* data source driver gets a message bus notification about a new *Nova* instance. It then sends the corresponding event to the *Entity Graph*. The *Entity Graph* returns a Vertex with an edge to the host Vertex in the graph and notifies *Nova* through the *Notifier* module.

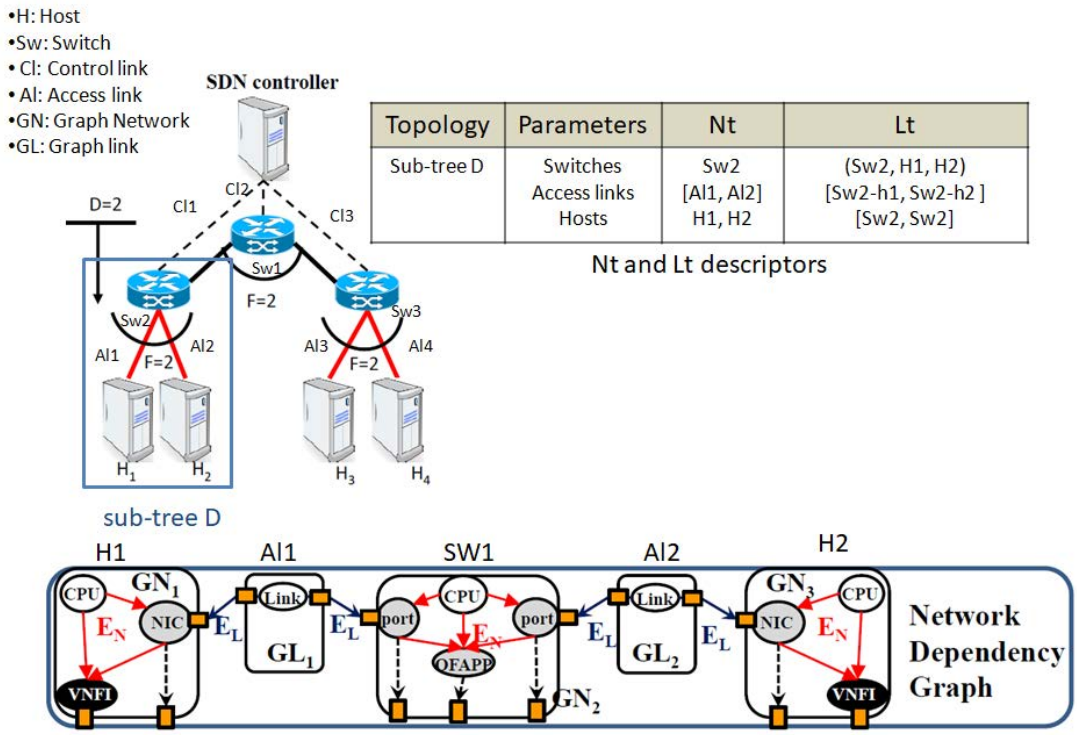


Figure 3.29 – Top left: tree topology (D=2, F=2) and the sub-tree D composed of the switch (Sw2) and hosts (H1, H2). Top right: The sub-tree D network (Nt) and (Lt) descriptor. Down the corresponding dependency graph [147].

3.6.2 Self-healing

Self-healing represents the autonomic ability for a system to recover from failures. In NVE, self-recovery reduces the time of healing and troubleshooting can occur while keeping the services running. To achieve self-recovery in NVE, several mechanisms have been developed and improved. Table 3.3 represents a taxonomy of possible self-healing actions according to the type of faults. Faults can generate from the distinct virtual network layers. Faults could be due to a server maintenance procedure. In this case a simple notification and migration of VNFs hosted in the server under maintenance to another hosting server resolves the problem. Other faults may be due to a network element crash or saturation, or due to a service mis-configuration. Thanks to the NVE programmability, self-healing became possible. For instance, in the case of a server crash, a simple VNFs migration resolves the problem. The OPNFV DOCTOR use case in Figure 3.32, presents the different steps that should be followed for a rapid fault detection and notification of the affected tenants [109]. In this use case, one of the servers hosting the Openstack VMs is down. The fault propagates trough the virtualization layer and affects the running VMs. Once the VIM has identified which VMs are affected, the

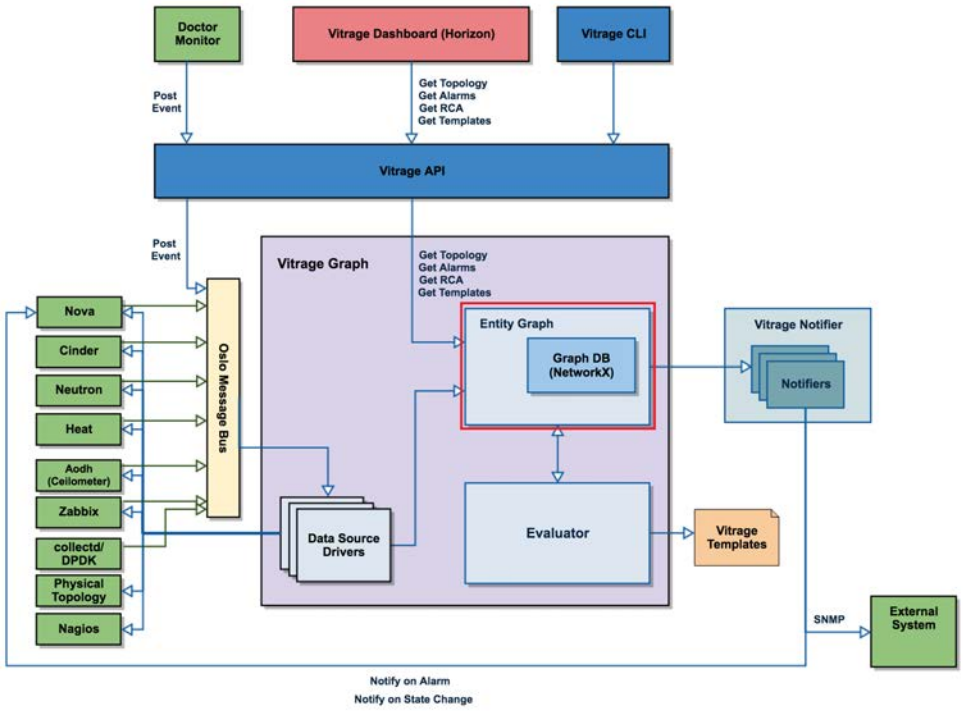


Figure 3.30 – *Vitrage* architecture [146].

concerned client(s) will be informed to get recovery instructions. In this case, client-1 switches to the standby VM-1.

In the following, important self-healing mechanisms and associated efforts are depicted:

Opensource recovery tools: OpenStack VIM offers some basic recovery actions when a VM is down using the Heat module that tries to rebuild the non-responsive VMs. However, it does not support any failover or redundancy mechanisms. Several solutions fill this gap by adding OpenStack failover plug-ins like Pacemaker [111]. These failover plug-ins instantiate redundant VMs to take over the failled ones. *Moghaddam et al.* [84], proposed to use a multi-agents system to monitor the VMs in OpenStack. The idea is to apply both the functionalities offered by PaceMaker and OpenStack modules. When a VM is down a new VM with the same configurations and services runs instead while the agents recover the failed VM. This process reduces considerably the recovery time to face the failover problem.

SDN controller failover: redundancy is also a way to secure the fault tolerance of the SDN controllers. Indeed, OpenFlow, does not support the control plane restoration mechanism. Moreover, to provide effective failover mechanisms, we should answer arising questions such as: "which controller is the best replacement for the failed one(s)?", "how many redundant controllers are needed?", or "how many updates are needed to keep the redundant controller

Layers	Fault types	SDN/NFV faults	Self-Healing actions
Physical	Maintenance	Maintenance of servers	1- Send Maintenance notification to the clients, 2- Turn the status of the server to a maintenance state.
	Physical Crash	Failure in a switch physical port.	Transfer the logical interfaces into another physical port
		SDN Controller failure.	Balancing to a secondary controller.
		Link down	Look for alternative path
		Crash of a physical machine (PM).	1- Change the state of (PM), 2-Transfer the VNFs running on the PM, 3- Empty the PM,for maintenance.
Virtual	Crash/ Saturation	VM crash or VM high cpu load.	Instantiate other VM, or extend the memory and CPU of the VM.
		Saturation of a vSwitch port.	Instantiate another port or vSwitch.
	Configuration	Switch ignores how to reach clients.	Reconfiguration of the controller or switch
		Bridge misconfigured.	Bridge reconfiguration.
		SFC Misconfiguration.	SFC reconfiguration.
Application/ Service	Crash/ Saturation	Too many requests to a VNF service.	1- Replicate the VNF, 2- Increase memory and CPU of VNF host.
		Crash of end-to-end service.	1- Reinitiate involved VNFs, 2- Restart end-to-end service.
		VNF application crash.	Restart or Instantiate the VNF host.

Table 3.3 – Taxonomy of faults in NVE

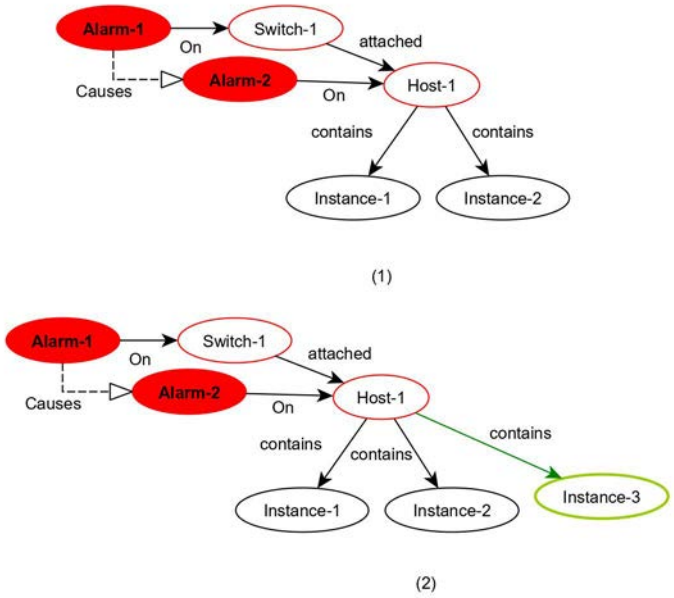


Figure 3.31 – The procedure of adding a VM instance-3 before (1) and after (2) [146].

aware of the current network topology ?". However, redundancy or multi-controller architectures implies dealing with controllers synchronization and network updates overhead [39]. *Guo et al.* [52], propose to conduct effective state synchronizations among controllers only when the load of a specific server exceeds a certain threshold to reduce the synchronization overhead between controllers.

Channel or Link failover: in SDN, the OpenFlow-Switch can identify a failed link but have to wait for the controller to establish alternative routes. Efforts [61, 43, 68], discussed the possible link failure recovery schemes in SDN using protection [11] and restoration[12] models. *Hwang and Tang* [61], presented a failover solution for both the control and data channels. For the data channel, both the restoration and protection mechanisms are used. In the restoration process, the SDN controller computes backup paths based on the complete bipartite graphs of the network topology, whereas, in the protection mechanism the controller configure the flow entries of SDN switches. However, in restoration failover time increases proportionally with the number of switches along the path, while protection has a weakness that it cannot deal with dynamic changes of network states and may cause congestion due to links reservation. To address the drawbacks, *Ko et al.* [68], combined both approaches in a dynamic network hypervisor based framework. The proposed method consists in the calculation of backup paths

[11] Protection approach: the routing links are reserved and computed before a failure occurs.
[12] Restoration: the new links allocation happens after the failures.

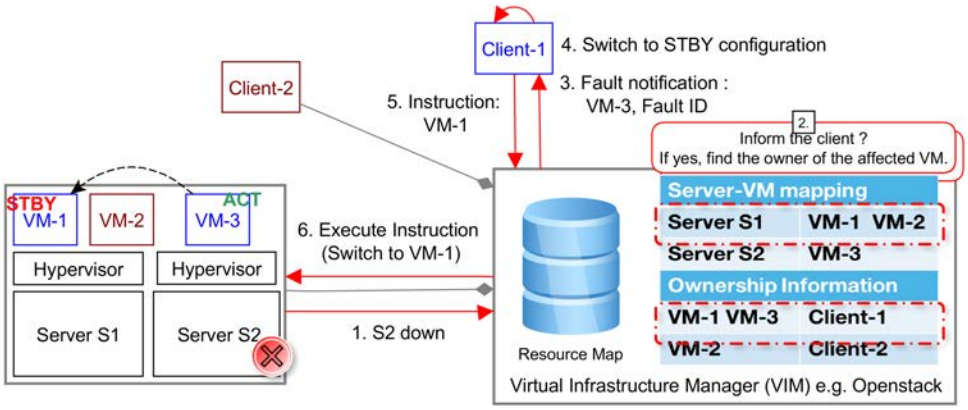


Figure 3.32 – OPNFV Server crash use case.

in a finer time frame (e.g. 5 seconds), and the recovery flow rules are configured only if a real physical network failure occurs. *Soualah et al.* [132], proposed to minimize penalties induced by service interruptions due to physical link failures. A decision tree approach is used to select reliable paths to prevent link failures and reactively reorient impacted virtual links in safer physical paths once an outage occurs.

NFV chain service failover: redundancy is also applied to Service Function Chains (SFCs). When a Service Function (SF) fails, the SFC control plane provides an alternative SF. However, this procedure may delay the failure when waiting for the alternative SF. *Lee and Shin* [74], addressed this problem by shifting the traffic in a case of a SF failure to another running function stored in the Service Function Forwarding (SFF) list.

Rollback: one of the well known techniques to provide fault tolerance is the rollback approach. Rollback is a way to recover a system transparently using snapshots [24]. The snapshots represent the correct running state of a VM. This state is periodically recorded into a stable storage to restore it when a failure occurs rather than restarting from the initial state. This significantly reduces the amount of lost computation. However, rollback augments the service down time comparing to normal system initialization. This is due to the huge size of the snapshot which is proportional to the VM size. To reduce the rollback latency problem, *Cui et al.* [24], proposed a rollback system which takes advantage of the similarity among VMs. The proposed solution leverages multicast to transfer the identical pages across VMs to disperse hosts with a single copy, rather than deliver them individually and independently.

3.6.3 Fault injection

The fault injection process consists on a number of defects injected on the network. Different fault injection scenarios are possible depending on the injection target and the fault type. The target represents the network element where the fault is injected. It could be a CPU usage

of a VM, a process or a network link. The faults could be applied to stress the network (e.g. CPU overload) or to completely disable the target (e.g. stop a physical server). Usually, the fault injection process is applied to test the network robustness such as the chaos monkey techniques used by Netflix [90]. The chaos monkey technique aims at collecting network performance under faults and evaluating the network ability to provide and maintain an acceptable SLA. *Cotroneo et al.* [22], also proposed to apply fault injection to evaluate virtual network use-cases such as vIMS. The authors proposed a taxonomy of faults injected to these use-cases. The aim is to provide a dependability benchmark to support NFV providers at making informed decisions about which virtualization, management, and application solutions can achieve the best dependability. Fault injection is also applied to provide the learning and validation data set for ML methods. In fact, acquiring this data could take several days, for example *Gonzalez et al.* [46] waited 206 days for around 21,442 network events. Efforts [126, 114], proposed the use of fault injection models to shorten this time. *Sauvanaud et al.* [126], triggered two methods of faults injection applied to vIMS: local fault injection applied in single VMs (increase of CPU consumption, memory leaks, etc.) and global fault injection applied to the global vIMS network (heavy workload). *Pham et al.* [114], applied faults injection to learn a set of fault-symptoms combinations in OpenStack. The aim is to use this fault-symptoms combinations to find the nearest fault type in the data-set that describes symptoms in the case of a network malfunction.

3.6.4 Discussion

Most of the addressed efforts tend to use automation for the fault management steps. Thanks to virtualization, a lot of efforts [84, 112, 61, 43, 68, 132, 74, 24, 129] apply self-healing and failover mechanisms such as redundancy and rollback. In the fault localization process, black-box ML approaches perform very well when it comes to extracting features from data and predicting the future behavior of the network state [11]. However, the learned features should provide a non-acquired knowledge (i.e. network components dependencies) and avoid learning knowledge that is easily extracted from data such as the dependencies due to network topology. For instance, the dependency between a VM and the hosting server learned in the work [46], is easily extracted from the network topology description. Furthermore, white-box approaches provide better fault explanations and are able to consider smaller granularity since sub-components such as processes can be considered in the model definition. However, due to the virtualization challenges such as the dynamic network topology, few efforts [147, 146] proposed white-box techniques in virtual networks. Moreover, fault localization proposals such as [146, 46, 126], are focusing on the management of the virtual layer. Faults in application or process levels that generate service alarms are not addressed.

3.7 Positioning of the thesis work

To progress towards the more ambitious objectives mentioned above, one must adopt a model-based approach, which provides explanations for their decisions and conclusions since each stage in the analysis can be followed and understood. However, virtualization introduced a number of issues and challenges for network fault management that we addressed in Section 3.5. Particularity, we refer to the challenges of model-based approaches: the lack of visibility on a global scale, the complexity and heterogeneity of virtual networks, and the topology constant changes and components relocation in the network. To face this issues, we propose three steps described in Chapters 4, 5 and 6, respectively. In the first step (Chapter 4), we propose a fault management framework to collect, centralize and filter logs for the fault localization process. In the second and third step (Chapters 5 and 6), we propose a self-modeling and an active diagnosis process. The position and contributions of our work compared to the existent fault localization solutions in virtual networks are the following:

- We propose a self-modeling approach to provide explanation about the faults rather than only pinpointing the faulty component such as in the case of [114]. The proposed self-modeling process with the diagnosis approach enables to explain novel virtual networks faults considering all the network layers (i.e. physical, virtual, application and service). In fact, applying a model-based approach allows to detect novel faults and not only the ones described in the diagnosis engine and rules such as the case of rule-based techniques (*vitrage* [146]);

- We propose to use the existent languages and definitions to model the network and to learn only the behaviors that are not easy to extract or deduce instead of learning already known dependencies from data. For instance, learning dependencies that are due to network topology links between components such as the case of [46]. We apply fault injection in order to learn the dependencies that are not described by experts or service description files and to extend and validate the defined modeling rules;

- The proposed model is a dependency graph (Section 3.4.1) with Boolean nodes and logical dependencies extendable to a probabilistic graph (i.e. BNs Section 3.4.3). When a fault occurs, the model can be instantiated to fit the current network topology using a self-modeling algorithm and the topology description file. We consider alarms and logs from service level. Till now, approaches [46, 146, 147, 126, 125], only consider virtual infrastructure logs and alarms;

- The self-modeling approach gathers the main features of virtual network use-cases (i.e. multi-layered, elasticity and auto-recovery). In our work, we apply this approach on the real world *Clearwater* vIMS use-case;

- The proposed diagnosis process is provided with observations through tests. We applied a constraints solver (Section 3.4.2) in the dependency graph to pinpoint the root cause(s) even when the root cause(s) has a smaller granularity such as application processes.

CHAPTER 4

LUMEN: A GLOBAL FAULT MANAGEMENT FRAMEWORK

4.1 Introduction

Virtualization of networks introduced a number of issues to fault management including: lack of network understanding due to complex topologies of interconnected components, lack of consistency and ambiguity of data and the dynamic network topology addressed in Chapter 3. To accomplish our ambitious self-modeling approach of virtual networks, we worked on virtual networks' data types and virtual network components behaviors. This phase is a preliminary step to the self-modeling procedure. In this phase, we defined a global fault management framework, namely *LUMEN*, that defines the canonical steps of fault management [17]. We highlighted the necessity to provide automation to the canonical fault management steps. *LUMEN* summarizes the fault management steps in four planes: source, sink, extraction and decision. The first three planes compose the detection step. In these three phases, the data is remodeled to prepare the decision plane where deduction methods can be deployed. The word "data" is used to describe all the type of collected information in the network. It could be: logs, components status, service and topology description files. These data are collected from network components, orchestrators (e.g. OpenStack) and monitoring agents. In this chapter, we will depict the *LUMEN* framework in Section 4.2 and illustrate its application to the Docker-based *Clearwater* vIMS use case. This chapter also addresses the virtual networks data types in Section 4.3 and the taxonomy of open source projects applied in our testbed to apply a self-modeling approach for virtual networks.

4.2 LUMEN global fault management framework

LUMEN is a Global Fault Management Framework for NVE. Figure 4.1 illustrates the four step architecture where each plane summarizes the methods to address the different NVEs challenges. *LUMEN* addresses the challenge of **lack of network visibility** by centralizing the collected data (i.e. logs and metrics) in one source plane. As depicted in Chapter 3 Section

3.5, the distribution of VNFs composing a same network service in distinct locations causes a lack of visibility, especially if each VNF is managed locally (i.e. logs of each location are treated separately). Therefore, centralizing the logs of VNFs composing a same service is necessarily to fault detection. Moreover, in *LUMEN*, we face the **tenants faults isolation** challenge by filtering and separating each tenant logs.

Figure 4.1 illustrates the four steps of the *LUMEN* framework. In the "source" plane, we showcase the different data types (logs, metrics, and network topology) that can be collected from distinct virtual environments (e.g. SDN and NFV). In the "sink" plane this data is filtered, organized and stored. The "extraction" plane depicts the procedure of data extraction according to the formalism applied in the decision plane.

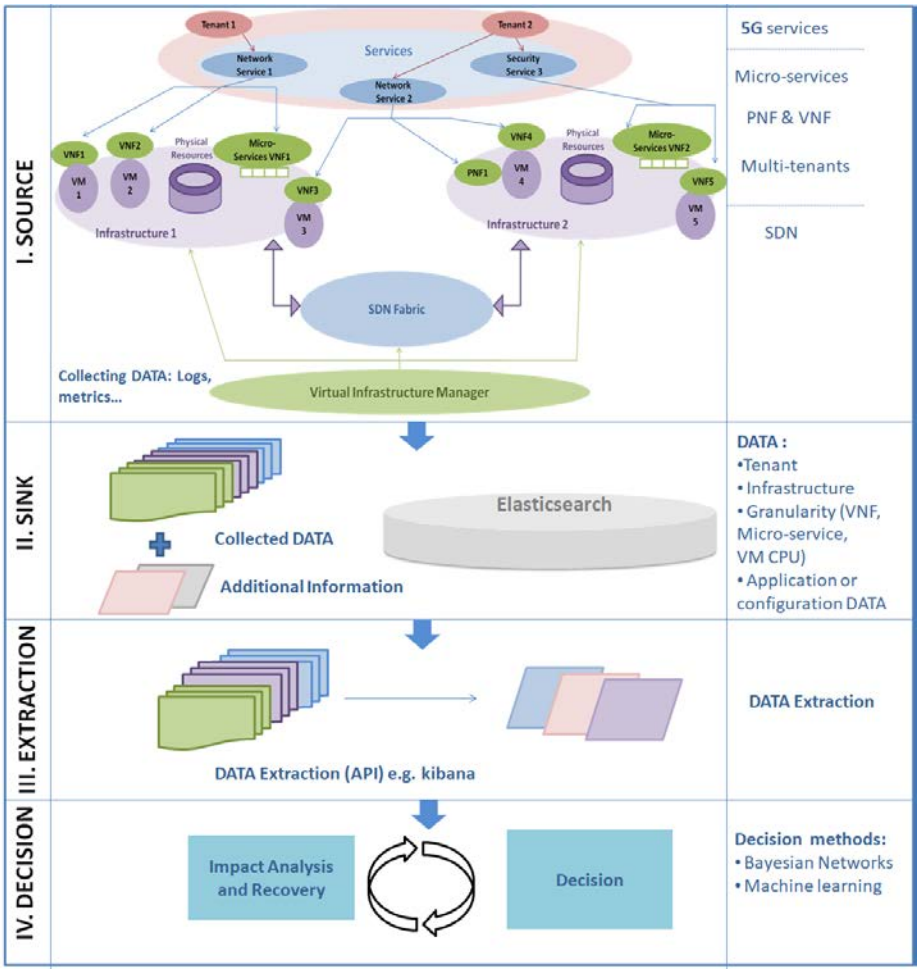


Figure 4.1 – *LUMEN* framework composed of four layers (source, sink, extraction and decision).

The objective of the *LUMEN* framework is to structure the different procedures applied

to organize and prepare data (i.e. Logs, metrics and service descriptors) for the deductions methods. The *LUMEN* framework leverages open source tools, namely the Elastic Stack [31]. The *LUMEN* framework can include different tools at each layer if necessary. The Elastic Stack ensures a real time data collection, storage, search, analysis, and visualization. The Elastic Stack is generally composed of four tools:

- Beats: are lightweight data shippers that can be installed as dedicated agents on managed entities to send specific types of operational data.
- Logstash: allows larger data collection and enables filtering, enriching, and transforming data from a variety of sources.
- Elasticsearch: is a search engine and analytics NoSQL database designed for storing efficiently the gathered data.
- Kibana: is used for the data extraction and visualization.

Information about the network health and the topology changes are collected in the *LUMEN* framework. However, this data is frequently noisy for the fault diagnosis process. For instance, only syslog messages that have severity level from 0 to 4 (i.e. 0: emergency, 1: alert, 2: critical, 3: error and 4: warning) are considered for the RCA process, informative syslog messages are not important in the case of a fault. The different *LUMEN* planes are depicted in the following Section 4.2.1.

4.2.1 LUMEN Planes

LUMEN enables the autonomic collection and organization of data to prepare the fault management decision process. The first three planes represent the detection step in fault management. The fourth plane represents the localization and recovery step. The functions of the *LUMEN* framework planes are described in the following.

I. Source Plane: the first step of our framework consists in gathering logs and metrics from distinct entities and distributed locations. For example, the Beats agents of the Elastic Stack can be deployed in every network entity to send real time alarms and metrics to Elasticsearch. The gathered data depends on the deduction method and the process that will be deployed in the decision plane. For instance, one way to answer the problem of dynamic topology for the model-based technique is to model the network entities dependencies using the real time network topology information extracted from files describing the deployed entities and their connections. The network topology information can be found in SDN controllers or networking modules like OpenStack Neutron [103]. Monitoring data such as logs and alarms are also an important source of inputs. They generally contain relevant hidden information about the network state, faults and root causes.

In virtual networks, we can collect different types and formats of data that may originate from distinct management levels:

- **The type of data**: if the collected data is syslog logs, metrics or topology information. The types of data are further discussed in Section 4.3.
- **The management entity**: if the collected data concern a specific tenant or the whole infrastructure.
- **The granularity**: if the collected logs and metrics concern a virtual host (e.g. a VM CPU load) or a whole service (e.g. the number of packets).

When proceeding to collect information about virtual networks, we may face the following questions: what is the type of data that we want to collect? at which level of granularity? and where? The answer of these questions depends on the type of management and decision procedures. For instance, to monitor the performance of VMs in the virtual layer, we can proceed by collecting VMs metrics such as CPU, memory, and disk utilization.

II. Sink Plane: Since the collected data originates from different sources, entities and applied technologies, the data will have different formats (e.g. syslog or JavaScript Object Notation (JSON) format for logs). Therefore, a unification of the data formats and the organization of data is important to facilitate the next steps of fault management. Moreover, the data should be filtered and the insignificant messages should be dropped. Filtering logs is the procedure of keeping only significant logs for the decision process. Significant logs are the logs that are useful for the decision procedure. For instance, if one wishes to monitor only logs of a specific tenant-A that is sharing resources with other tenants in the same infrastructure, filtering logs in this case consists in isolating the logs of tenant-A.

Figure 4.2-D presents an example of a controller syslog message structured by Logstash in a JSON format. JSON is a lightweight file description format that uses human-readable text to transmit data objects [64]. Logstash unifies messages from disparate sources and normalizes the structure of the message before storing it in the sink. Figure 4.2-A and 4.2-B exposes how Filebeat configuration files enables message filtering and dropping irrelevant information, respectively. In Figure 4.2-A, a filter was added "include-lines" to consider the log line that has the "ERR" for error and "WARN" for warning tags. In Figure 4.2-B, the log lines that has the word "DBG" for debug messages are dropped. This way of filtering logs is necessary in fault diagnosis, to keep only the critical messages that notify a fault in the network.

Adding additional indications before the storing process is also crucial to localize the tenants and the network slices in further fault management steps. Filebeat transforms the data by adding new fields (Figure 4.2-C). In fact, one way to separate the logs collected from distinct tenants is to add a tenant identification field in the stored logs. For instance, in Figure 4.2-C an additional field called "slice_id" was added to identify the log owner.

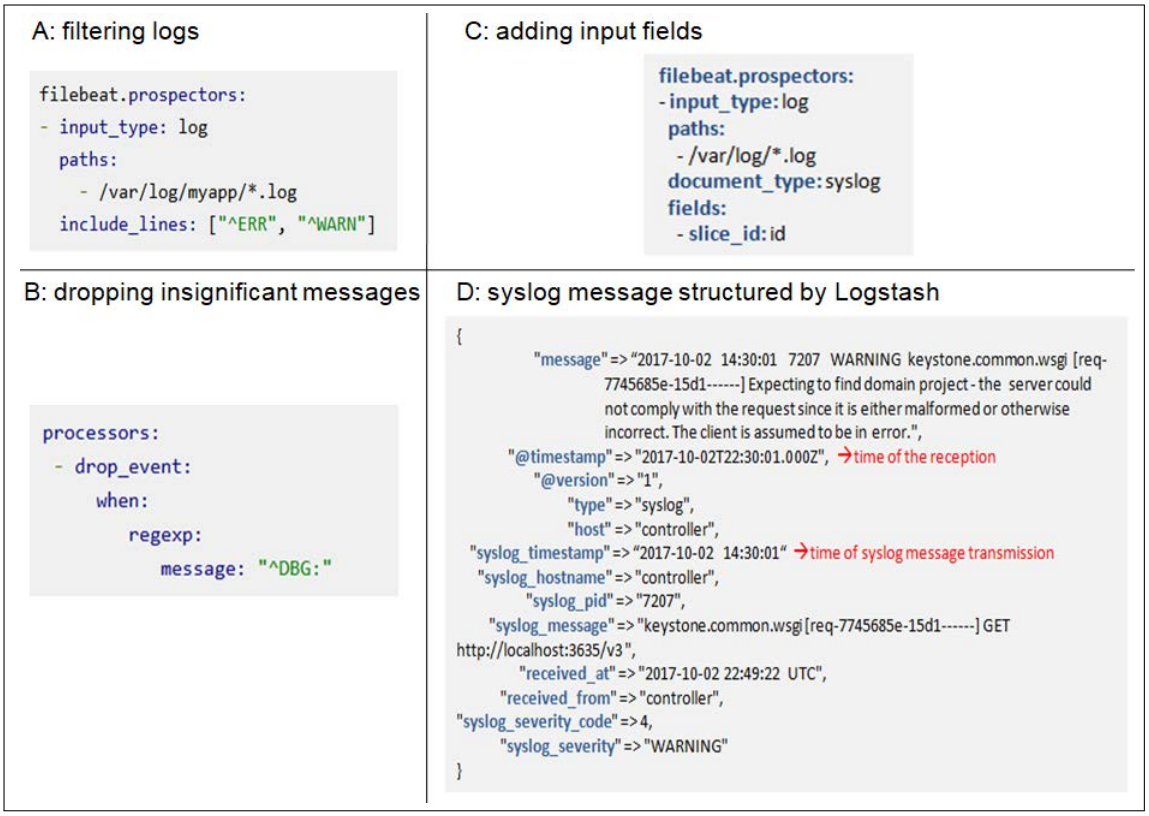


Figure 4.2 – Elastic Stack Logs transformation

After the data transformation process is completed, the collected logs can be stored in a sink, Elasticsearch for instance [31]. Elasticsearch is a robust search engine and NOSQL database, that centralizes data and enables efficient extraction of features.

III. Extraction Plane: This step is highly dependent of the decision process. In fact, the extraction of data depends on the inference engine used by the decision methods. For instance, some KPIs such as VM CPU and disk load can be mined to calculate VNF SLA violations. APIs of the sink plane can be used for the data extraction process.

IV. Decision Plane: In this step we create an educated guess through one or more deduction methods and diagnosis approaches. This step is enabled by the first three planes that provide the necessary data in a unified and organized way for an efficient decision process. Note that this is where the diagnosis algorithm that we will depict later in Chapter 6 is instantiated. The visualization of logs and metrics is not depicted in the *LUMEN* framework. However, a number of open source tools can be applied to visualize logs or metrics: Elastic Stack Kibana [67], Grafana [49], etc.

4.3 Virtual Networks data

The main observable facts/observation families in networks are: metrics, logs, traces, and alerts. However, in virtual networks other types of data are necessary to apply a model-based approach (e.g. the network topology and service specifications). The virtual network data (or knowledge) are of two types: *acquired* and *learned* data (Cf Table4.1). The *acquired knowledge* is the available knowledge collected from the network description files or logs. This knowledge is applied to learn the building rules of the model, to define the current topology or define the status of some network components. We call this data "knowledge" since it provides information about the network dependencies, component status, network topology, services description, etc. In virtual networks the *acquired knowledge* could be:

- **Service specifications**: the knowledge about the service component requirements and capabilities that we can obtain from description languages and standards like TOSCA [142], expert operators or network protocols.

- **Observations**: available in a running environment, observations represent the status of the deployed services, the virtual and physical hosting environment, the network functions' connections and components. This knowledge can be collected from orchestrators such as the Docker engine and OpenStack modules or open source tools (e.g. Weavescope a monitoring tool for Docker and Kubernetes [149]).

- **Network topology data**: can be extracted from SDN controllers or VIMs such as OpenStack or the Docker engine. This type of data will provide a real time vision about the location of the virtual network entities in physical servers and infrastructures.

On the other hand the *learned knowledge* is acquired by tests and fault injections. The tests are performed to obtain information about the health of network components. While, fault injection is conducted to learn about the components dependencies.

Table 4.1 – Acquired and learned knowledge

Acquired knowledge	Learned knowledge
- The network service model including the components requirements and capabilities (e.g TOSCA). - The network topology, physical and virtual dependencies.	- Tests (e.g ping) - Fault injection (e.g link severing)

4.4 Application of *LUMEN* to the vIMS use case

In our work, we applied the *LUMEN* framework for collecting and filtering the vIMS logs. We also opensourced the code in our project stored in the GitHub repository [85]). The use case applied in the *LUMEN* source plane is the Docker-based *Clearwater* vIMS use case [138]. The *Clearwater* project defined its own logging procedure. Each *Clearwater* component (e.g. Bono, Sprout) generates logs and stores them internally (i.e. inside the Docker volume attached to the component that is running). So we proceeded to centralize the logs of all the *Clearwater* components in one location using the open source Elastic stack. The procedures: vIMS deployment, traffic generation, fault injection and log collection and filtering are depicted in the following sections. These procedures represent the first three planes of *LUMEN* that prepare the decision plane. The decision plane consists of the self-modeling procedure and active diagnosis algorithm that we will address in the next chapters. More technical details and examples about the procedures depicted in the following are provided in the Appendix C 7.2.4.

4.4.1 Docker *Clearwater* vIMS deployment

The deployed vIMS is the open source project *Clearwater*. We were interested in the Docker version of this project, since Dockers are lightweight and generate less overhead compared to VMs. The original project is hosted in the GitHub repository [139]. This project builds the Docker images in each execution. We modified the file containing the functions into a Docker-compose to consider the already built images stored in the public Docker-hub, for a faster instantiating of Docker images. The modified project is hosted in the GitHub repository: [138]. When deploying *Clearwater*, we discovered a number of features that are crucial when constructing the modeling rules. These features are depicted in the Appendix C 7.2.4.

4.4.2 Traffic generation

A number of SIP traffic generators exists: client SIP services (e.g Jitsi [65]) and Open source test tools (e.g SIPp [130]). These solutions enables to generate SIP traffic through a bunch of SIP clients. The *Clearwater* dashboard Ellis allows to create and modify the identity of the SIP clients.

4.4.3 Fault injection procedure

Usually, fault injection is applied to test network robustness in order to increase its resilience, such as the chaos monkey techniques used by Netflix [90], (Cf. Chapter 3, Section 3.6.3). However, in our case, fault injection is applied to learn network dependencies in order to verify and expand our knowledge about vIMS. This knowledge will be applied to define the modeling

rules in the next chapter. These technique also provides a way to check the validity of the diagnosis results.

Network layers		Fault injection	Reverse actions: healing
(1)	Physical network	Disconnect physical servers.	Connect physical servers.
	Physical hardware	Disable physical servers.	Migrate network functions.
(2)	Local virtual network	Disable the shared network bridge.	Enable the shared network bridge.
	Distant virtual networks	Disable the overlay network.	Enable the overlay network.
	Virtual hosts	- Kill /Stop Dockers (functions /recovery/ config). - Disconnect Dockers from the network bridge and overlay networks.	- Start new Dockers / Restart the old Dockers. - Reconnect the same Dockers or kill the old Dockers and connect new Dockers.
(3)	Application processes	Kill / Stop processes.	Start new / Restart the processes if auto-recovery disabled.
	auto-recovery processes	Kill/Stop auto-recovery processes (e.g. Monit)	Start new/Restart the auto-recovery processes e.g. Monit.
	Configuration management processes	Kill/Stop ETCD Client processes.	Start new /Restart ETCD processes.
(4)	Services	Increase the number of clients registration requests.	Decrease the number of clients or add resources (e.g Bono).

Table 4.2 – Overview of the fault injection model in each layer: (1) Physical, (2) virtual, (3) application and (4) service.

To inject faults, two methods are possible. The first method consists in stressing the network by injecting more traffic. The second method consists in crashing network components, processes and links. In our testbed we mostly apply the second type of faults while trying each time to target a different network layer. Table 4.2 summarizes a number of faults injected in the vIMS *Clearwater* environment and the reverse actions that represent the healing of each fault injection to regain the network health. Note that theses faults are injected separately and manually to the deployed vIMS. Moreover, the "kill" and " Stop" fault are different. The "Kill" fault means that the target where this fault is injected die and can not return to the "run" state, while a "Stop" fault only suspends the target and can be restarted.

4.4.4 Logs collection and filtering

In this step we deployed a number of monitoring tools to collect observations about the network before and after fault injection. The observations consist of:

- The alarms and logs of the *Clearwater* components that we collect from each network function and filter and store in the Elasticsearch sink.
- The real-time status of components from *Weave Scope* and Docker engine.
- The network health reports collected from the *Clearwater*-live-test.

4.5 Conclusion

This chapter 4 addresses the first challenges of virtual networks: lack of network visibility, isolation of tenants faults and the consistency and ambiguity of data. This allows us to prepare the fault localization step. In the next Chapter 5, we will depict the self-modeling procedure and illustrate how each of the steps addressed in this chapter are useful when defining the modeling rules and validating them.

CHAPTER 5

SELF-MODELING

5.1 Introduction

Model-based approaches are knowledge-based systems which reason about a system from an explicit representation of its structure and functional behavior. Model-based techniques solve novel diagnosis problems and *provide explanations for their decisions*. However, current Model-based methods suffer some limitations, since virtual networks bring new challenges such as their complex multi-layer nature, the coupling of physical and virtual behaviors, the lack of network visibility due to the distribution of VNFs in distinct sites, the dynamic topology and the elasticity of services. Those limitations raise the difficulty of obtaining a reliable and up-to-date model.

To face those limitations, we propose a self-modeling approach and illustrate it on the vIMS use-case. The vIMS use-case includes most of the infrastructure and service specifications that we can identify in a virtual telecommunication environment. The proposed model is a multi-resolution directed acyclic graph that represents the intra-layer and inter-layer logical dependencies with fine grain and coarse grain variables.

5.2 Self-modeling approach overview

The main weaknesses of model-based diagnosis methods is the derivation and the validation of a suited model for the diagnosis process, in particular if one wishes to capture several network layers and segments with different granularity. Another difficulty is that the defined model should follow the network updates. Our goal is to define a generic modeling methodology and an online reconfigurable model. We understand by "generic model", the ability to instantiate the model no matter the elasticity of services and dynamicity of the network topology; and by an online reconfigurable model the fact that our model captures the topology changes such as the migration, elasticity and scalability of VNFs. However, we assume that the topology changes are slow compared to the faults and alarms propagation, so the diagnosis will be performed on a fixed and supposedly up to date model. Moreover, the model should represent the network components from the physical level up to the service layer.

Our main motivation in deriving such a model regarding the number of challenges introduced by the virtual networks is the existence of a double nature of knowledge (Table 4.1): *acquired* and *learned* knowledge, presented in Chapter 4, Section 4.3. We understand by "knowledge" an expertise about the network that enables us to build the model. This knowledge is usually extracted from experts or network description files (i.e. acquired knowledge). However, this knowledge can also be directly inferred from the network using fault injections (i.e. learned knowledge), such as described in the previous Chapter 4, Section 4.3.

Figure 5.1 illustrates the two steps of the global self-modeling approach: the construction of the building rules and the self-modeling procedure. The first step consists in defining the building rules for the use-case we want to monitor using the two types of knowledge (i.e. acquired and learned knowledge). These rules are then validated and stored in a knowledge base. The second step represents the self-modeling procedure that builds a model instance fitting the current network topology and following the building rules. The first step is described in Section 5.3 that describes the modeling or building rules and in Section 5.6 that showcases the validation of the defined building rules. The self-modeling algorithm, depicted in the bottom part of Figure 5.1, is defined in Section 5.5.

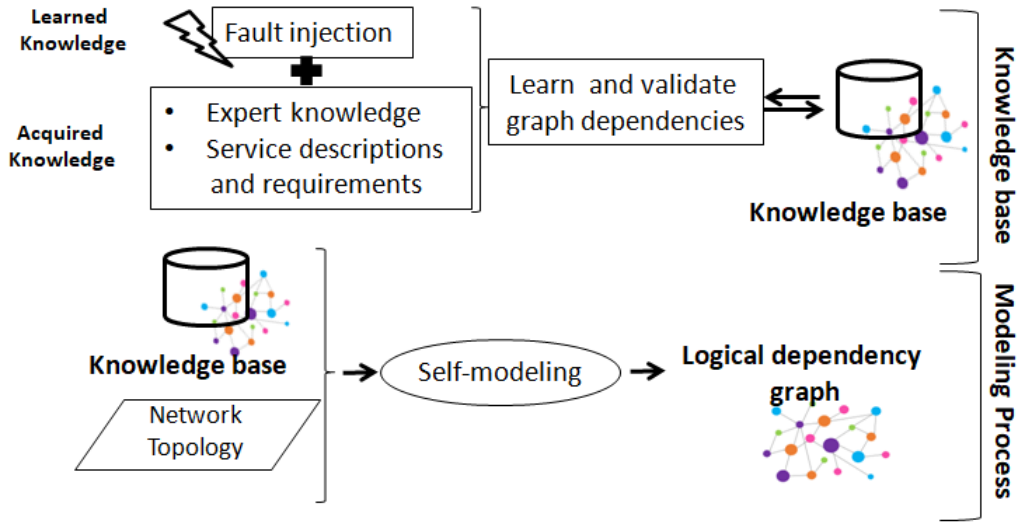


Figure 5.1 – Model construction and self-modeling Process

5.3 Modeling rules derivation

To define the modeling rules, we process with the first step represented in Figure 5.1 (top). Our modeling rules represent a number of generic network components and their possible

dependencies represented as generic templates. A network component could be a VNF, a connection, or a service. For each network layer a number of templates are defined for components. For instance, servers or sites in the physical layer, the Dockers or VMs in the virtual layer, the VNFs and their processes in the application layer and finally the services running on top of the network under the form of requests sent between VNFs. Templates can be thought of as "object classes" in the object-oriented paradigm. Used modeling rules will then reassemble a class diagram. The aim of the modeling rules is to construct a graph of Boolean variables and logical dependencies describing all the network components using the defined component templates (i.e. an object diagram, following the object oriented programming metaphor). The resulting graph represents assembled instances of templates describing the network components. Components of the same nature have the same template with distinct instances of this template. The instantiation of these templates is operated by the self-modeling algorithm that we will define in Section 5.5.

A template represents a generic description of a network component defined as a Directed Acyclic Graph (DAG): $G = (V, E)$, with a set of nodes V and a set of edges $E \subseteq V \times V$. The nodes represent Boolean state variables for this component type and the edges are logical relations. Each template (class) can have multiple instances (objects) in the network model. Each node is "labeled" with a unique name in each template instance. The nodes' unique labeling enables the assembling of the different template instances to construct the global model. Figure 5.2-a illustrates a model example based on three distinct templates. Each of the templates $Template_1$ and $Template_2$ has two instances. $Template_3$ has only one instance. These instances are assembled through the labels of nodes such as depicted in Figure 5.2-b.

Nodes V are Boolean variables with value $True$ for an "Up" status and $False$ for a "Down" status for the component state it represents. We define for each node $v \in V$ a number of attributes represented as a tuple $\ell(v) = (l_v, t_v, SF_v)$ where l_v is the network layer to which the node belongs:

$$l_v = \begin{cases} 0 & \text{if physical layer} \\ 1 & \text{if virtual layer} \\ 2 & \text{if application layer} \\ 3 & \text{if service layer} \end{cases}$$

$t_v \in \{\textit{true}, \textit{false}\}$ indicates whether the state of node v is directly testable from the network ($t_v = true$, means testable), and SF_v indicates if the nodes can represent spontaneous fault or not (i.e. the node can be "Down" regardless of the value of its predecessors in the graph G). SF_v can take three different values:

$$SF_v = \begin{cases} 0 & \text{if non spontaneous node} \\ 1 & \text{if spontaneous node} \end{cases}$$

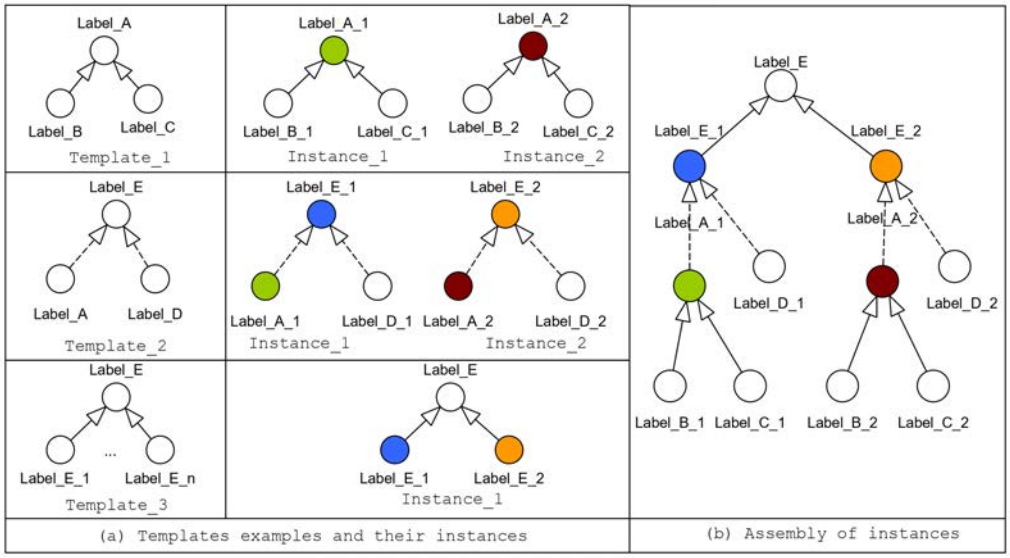


Figure 5.2 – Example of assembling templates instances.

Relation between a node and its predecessors:

We label each edge $E(v', v)$ with a type of the logical relationship defined as: $d_{(v',v)} \in \{AND, OR, \Rightarrow, \Leftarrow\}$. The $\{\Leftarrow, \Rightarrow\}$ are an implicit way to represent the logical edge types $\neg A \implies \neg B$ and $\{B \implies B\}$, respectively.

We define a "valuation" on G as a function $val : V \to \{true, false\}$ be a function that indicates, at a given time, the state of each node $v \in V$, where for nodes representing an entity of the network, $val(v)$ tells us whether node v is up.

op	constraint on val
AND	$val(v) = \bigwedge_{v':d_{(v',v)} \in \{AND\}} val(v')$
OR	$val(v) = \bigvee_{v':d_{(v',v)} \in \{OR\}} val(v')$
$\Rightarrow$	$\exists! v' : d_{(v',v) \in \{\Rightarrow\}} \wedge (val(v') \implies val(v))$
$\Leftarrow$	$\exists! v' : d_{(v',v) \in \{\Leftarrow\}} (\neg val(v') \implies \neg val(v))$

Intuitively, nodes v' that have $d_{(v',v)} = \{AND\}$ (resp. $d_{(v',v)} = \{OR\}$) means that node v is up if and only if nodes v' are up (resp. at least one of the nodes v' is up). The dependency types $\Rightarrow$ and $\Leftarrow$ means that node v is up at least when its predecessor v' is up (resp. down when its predecessor is down).

Figure 5.3 shows the five possible dependencies configurations in our current model. In the first four cases, the dependencies between a node v and its parents $v' \in (v_1, .., v_m)$ are of the same type $d_{(v',v)} =$ " $\Rightarrow$ ", $d_{(v',v)} =$ " $\Leftarrow$ ", $d_{(v',v)} =$ "*OR*", $d_{(v',v)} =$ "*AND*". Notice that the case

$d_{(v',v)} =$ " $\Rightarrow$ " is equivalent to the case $d_{(v,v')} =$ " $\Leftarrow$ ". However, we represent both cases to express the causality relation between nodes that is necessary to the diagnosis process that we will present in Chapter 6. The only case where we have two different types is represented in the last configuration. These case is present generally due to the inter-layer dependencies of type $\Leftarrow$. In this case, the node v_k is in a the upper layer than v (i.e. an inter-layer dependency). This dependency is translated separately (Figure 5.3 in blue).

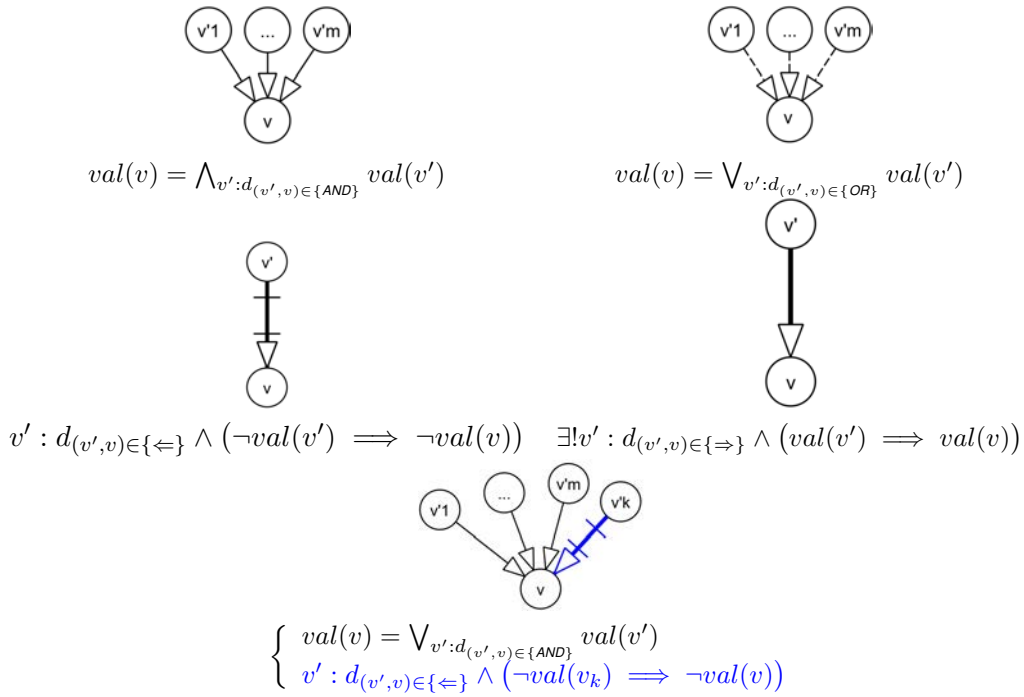


Figure 5.3 – Possible dependencies and their associated assumptions

To construct this templates we leverage both types of knowledge:

1. The acquired knowledge: this type of knowledge is applied to construct a first representation of the relations constituting the templates such as the relation deduced from the fact that if a Docker is down, the VNF hosted on the affected Docker is down.

2. The learned knowledge: this type of knowledge provides dependencies that are difficult to obtain from the network environment and that can only be discovered experimentally. We learn these dependencies by injecting a failure in a node, a process or a link; and observe the behaviors of the connected components. For instance, while injecting faults, we may notice the existence of an auto-recovery procedure that should be considered in the modeling rules. This part is further depicted in Section 5.6.

Deployed virtual network services may differ in the type of applied virtualization technologies (i.e. Docker or VMs OpenStack.), the deployed VNFs and protocols. However, a number of

common features and components between these virtual network services exist and can be incorporated to the defined templates.

Figure 5.4, illustrates a generic structure of virtual networks. Virtual networks are structured into multiple layers (i.e. physical, virtual, application and service layer). In each layer a number of components are present (e.g. servers and physical links in the physical layer). Other aspects such as elasticity (or replication) of VNFs and auto-recovery mechanisms are possible. Each of these features are defined in "templates" described in the following. Note that these is a general description of possible templates in virtual networks. These templates can be extended with other nodes and dependencies that are specific to the use-case we want to model. To represent

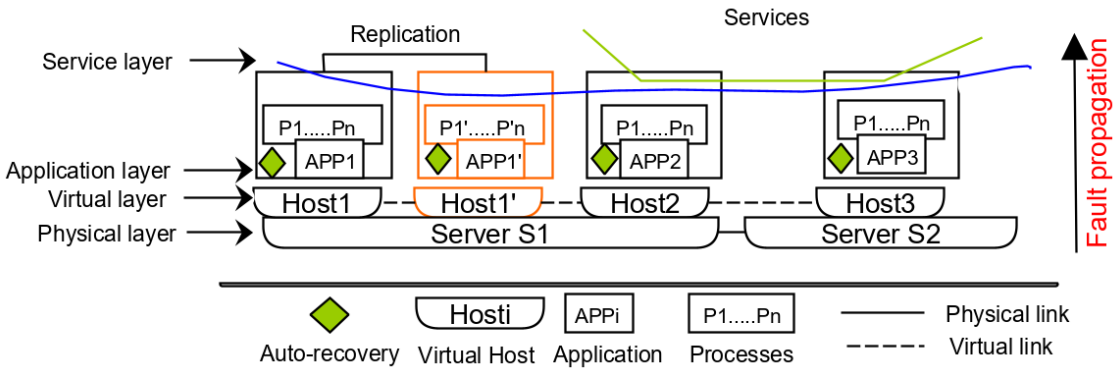


Figure 5.4 – Virtual networks global architecture.

the features of virtual networks, we divide the templates into five groups:

- **Network elements templates**: gather the templates that define the network components composing each layer. For instance, sites for the physical layer. The network connections are not described in this group.
- **Network connections templates**: group the templates that describe the network connections in each layer.
- **Aggregated node templates**: describe two templates for aggregated nodes in the physical and virtual layer.
- **Auto-recovery and elasticity templates**: gather the templates that define two features of virtual network: the auto-recovery mechanism and the elasticity of VNFs.
- **Inter-layer templates**: describe the templates that link the four layers of virtual networks.

Network elements templates:

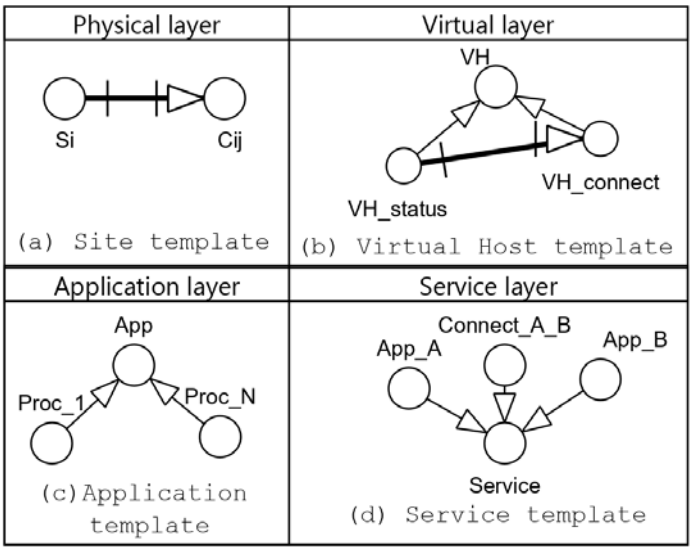


Figure 5.5 – Network elements templates.

- `Site template` ($T.site$), (Figure 5.5-a): defines a physical site. Each physical site is a sub-graph composed of two nodes: a status S_i and a connectivity node C_{ij}. The status node S_i is a Boolean variable that expresses the status of the physical server (i.e. Up or down). The C_{ij} node represents the status of the physical connectivity between $site_i$ and $site_j$. Nodes S_i and C_{ij} are linked with a relationship type "$A \implies B$". If a site is isolated from the other sites, then the physical connectivity node is omitted.

- `Virtual host template` ($T.VH$), (Figure 5.5-b): describes the virtual hosting nodes (i.e. Dockers or VMs). Each VH has a status node VH_S and a connectivity node VH_C, linked with a relationship type "$A \implies B$". The node VH is related with VH_S and VH_C with an "*AND*" relationship.

- `VNF application template` ($T.APP$), (Figure 5.5-c): defines the global VNF application template. VNFs are software programs with a number of processes. A general view of the VNF template is illustrated in Figure 5.5-c. The application depends on process status. Thus the relationship "And", between the application and its processes. However, each VNF has its own template composition depending on the code of the application. We will depict further examples of VNFs composing the vIMS use-case in Section 5.4.

- `Service template` ($T.service$), (Figure 5.5-d): defines the service dependencies. A service is composed of a set of connected VNFs. Therefore, we define the service template with an "*AND*" relationship between a set of VNFs nodes and the logical connection between them.

Network connections templates:

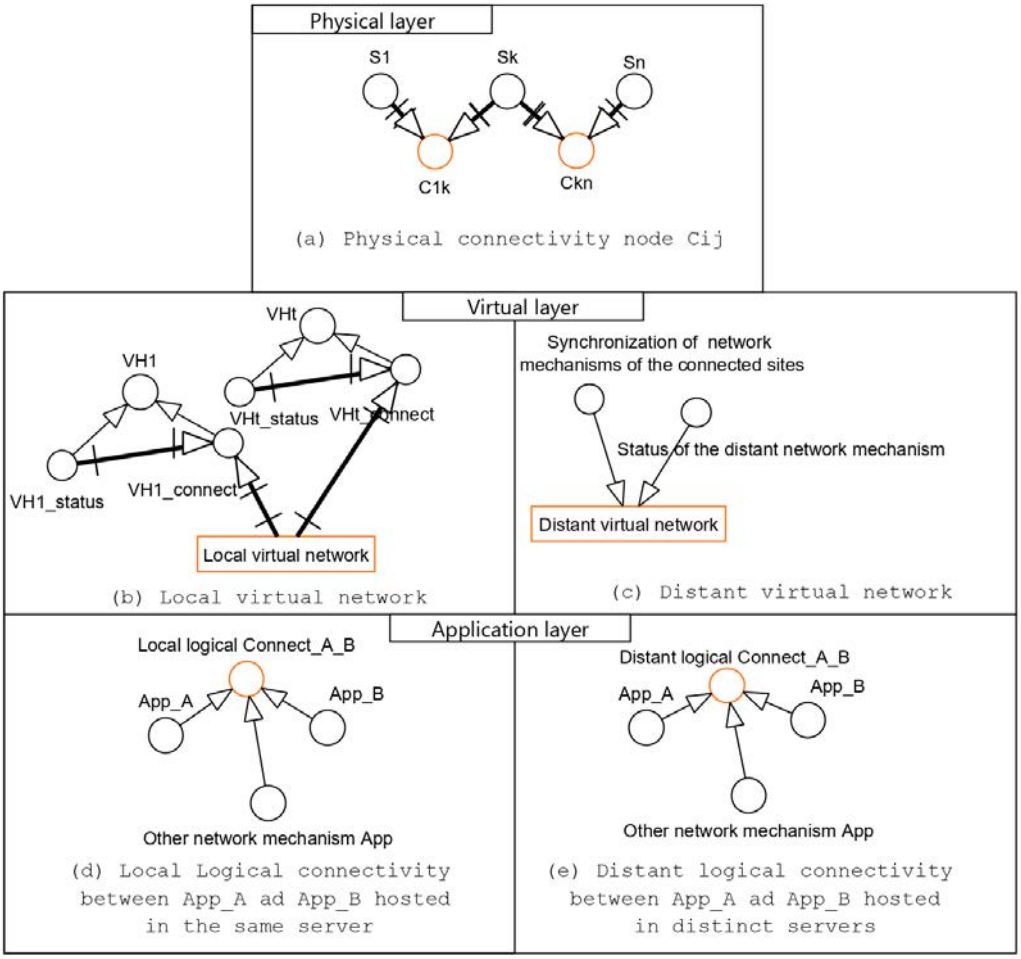


Figure 5.6 – Network connections templates.

- `Physical Connectivity template` ($T.PC$): is represented by the variables C_{ij} between $Site_i$ and $Site_j$ defined in the site template as depicted in Figure 5.6-a.
- `Local Virtual Connectivity template` ($T.LVC$) description differs from one virtualization technology to another. It could be an SDN controller and switches or a Docker bridge network. We give a general view of the local virtual network variable and its dependencies in Figure 5.6-b. A local virtual connectivity is represented as a virtual network node linked with an "$A \implies B$" relationship with the virtual hosts connectivity nodes VH_C. In other words, if a virtual network node is "down" than all the virtual hosts connected to that network are disconnected to the local network so their connections status are "down".

- `Distant Virtual Connectivity template` ($T.DVC$): represents a virtual connection between two distinct sites (e.g. an overlay network in Docker). It generally depends on its status and the synchronization of the virtual network mechanism of the two sites (Cf. Figure 5.6-c).

- `Local Logical Connectivity` ($T.LLC$) and `Distant Logical Connectivity` ($T.DLC$): represent the status of the connection between two applications. In fact, the connection between two applications depends generally on the status of the two applications. However, some connectivity mechanisms such as DNS may also be related to these templates. The logical connectivity can be between two local applications (i.e. both applications are hosted by the same servers), or distant (hosted by different servers). The local logical Connectivity ($T.LLC$) and distant logical connectivity ($T.DLC$) are illustrated in Figure 5.6-d and Figure 5.6-e, respectively. The two kinds of logical connectivity are defined in the same way, the difference appears in the inter-layer templates.

Aggregated nodes templates: (towards a multi-resolution model)

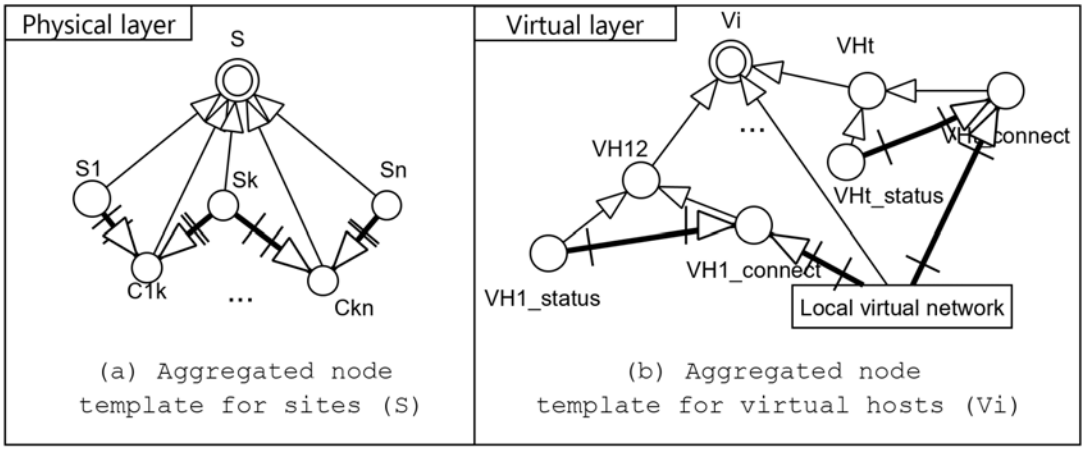


Figure 5.7 – Aggregated nodes template.

Aggregated nodes are introduced to abstractly represent a set of atomic nodes in order to enable diagnosis on multiple levels of abstraction. For instance, if we want to suppose that the virtual and physical layer are "up", we only have to set the aggregated variables of sites and virtual layers to "up". In fact, all the nodes of templates that have successors (i.e. not leaves) are considered as aggregated nodes. Moreover, two global aggregated node templates have been defined for sites and virtual environments. These nodes enable to separate the management with the application and service layers when necessary.

- `Aggregated node template for sites` ($T.An_S$), (Figure 5.7-a): describes an aggregated node S for sites. It aggregates all the status and connection site variables to express the server availability [1].

- `Aggregated node template for virtual hosts` ($T.An_{V_i}$), (Figure 5.7-b): describes an aggregated node V_i for the virtual environment. It aggregates all the virtual hosts variables contained in the same site S_i and the local virtual network that connects the virtual nodes. The $T.An_{V_i}$ is only instantiated when the number of virtual environment of the same site are greater than 1.

Auto-recovery and elasticity templates:
Virtual network service use-cases such as vIMS, have two specific features: **auto-recovery** and **elasticity**. Auto-recovery are mechanisms defined to auto-monitor the system. They are responsible for checking the status of the monitored components and restart these components in the case of their failure. For instance, an auto-recovery mechanism to check the status of VMs. In the other hand, the elasticity aspect of network components means that the same

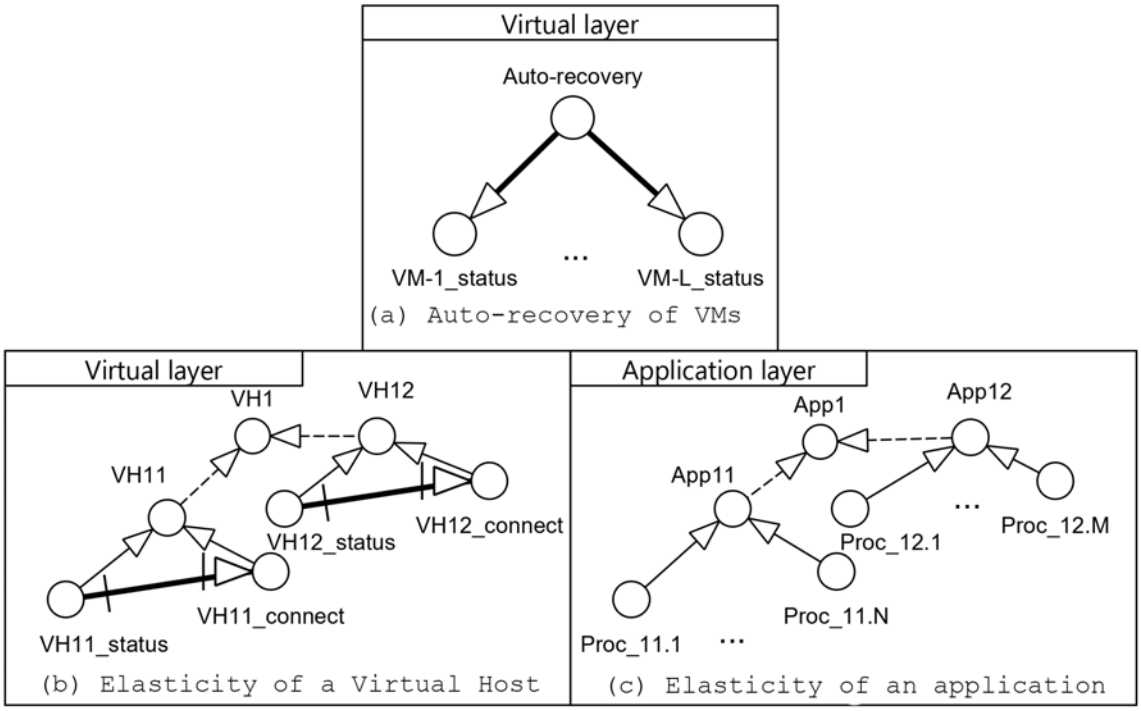


Figure 5.8 – Auto-recovery ($T.AR$) and elasticity ($T.EL$) templates.

component is replicated to ensure a coverage in the case of a failure or huge service request

[1] Note that the aggregated node for sites is only instantiated when the number of sites that we want to model is greater than 1

load. The auto-recovery and elasticity templates are illustrated in Figure 5.8 and defined as follows:

- `Auto-recovery template` ($T.AR$): the auto-recovery node is linked with a "$A \implies B$" relationship with the components that are monitored by this node. Figure 5.8-a showcases an example of an auto-recovery node template applied to a set of VMs. These node recovers the VMs in the case of a failure.
- `Elasticity template` ($T.EL$): this template defines an aggregated node that gathers the replicated nodes with an "*OR*" relationship. It means that all the replicated nodes must be "down" for the replicated function to be "down".

  Figure 5.8-b illustrates a replication of a virtual host offering the same application. In this example $VH1$ status depends on the status of ($VH11$ OR $VH12$). Note that if a virtual host is replicated, its application template is also updated with an aggregated node that represents the elasticity in the application layer. This case is illustrated in Figure 5.8-c.

Inter-layer dependencies:
The inter-layer dependencies ($T.inter$) represent the relationships between the different layers nodes and are of two types: inter-layer dependencies on components status and inter-layer dependencies on connectivity as illustrated in Figure 5.9.

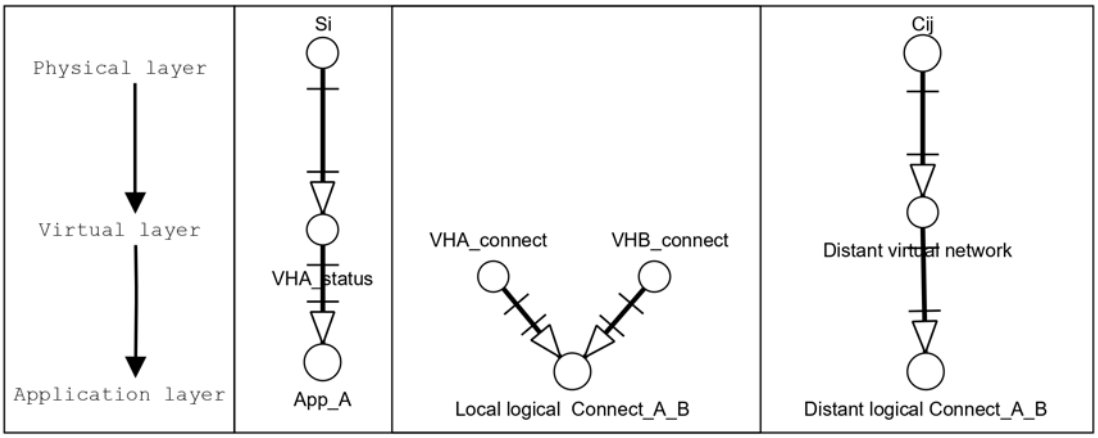


Figure 5.9 – Inter-layer dependencies.

In fact the status of an application depends on the hosting environment that depends on its turn on the server status. The inter-layer connectivity relationships differ in the case where the logical connectivity node is between two local applications (i.e. the applications are running in the same server) or distant ones. In the first case the connectivity depends on the local network and the virtual hosts. In the second case it depends on the distant virtual connectivity that in turn depends on the physical connectivity node C_{ij}.

5.4 Real world application: *Clearwater* vIMS self-modeling

In this section, we present an application of our building rules in the Docker *Clearwater* vIMS use-case. As mentioned in the definition of the templates in Section 5.3, the model of virtual networks captures four layers: physical, virtual, application and service layer. This structuring also allows one to model each layer by progressive refinements, thus selecting the finest granularity to manage the network.

In the self-modeling procedure of *Clearwater*, we only consider the central *Clearwater* vIMS functions namely: Bono, Sprout, Homestead, Homer and Cassandra. We omit Ralf and Chronos that are responsible for billing the clients communications and Ellis that represents the *Clearwater* dashboard. The deployed functions are hosted in Docker containers. Figure 5.10 illustrates an example of a deployment use-case of the *Clearwater* vIMS. In this architecture, the IMS functions Sprout and Bono are hosted in a separate distinct physical server from the Homestead, Cassandra and Homer functions.

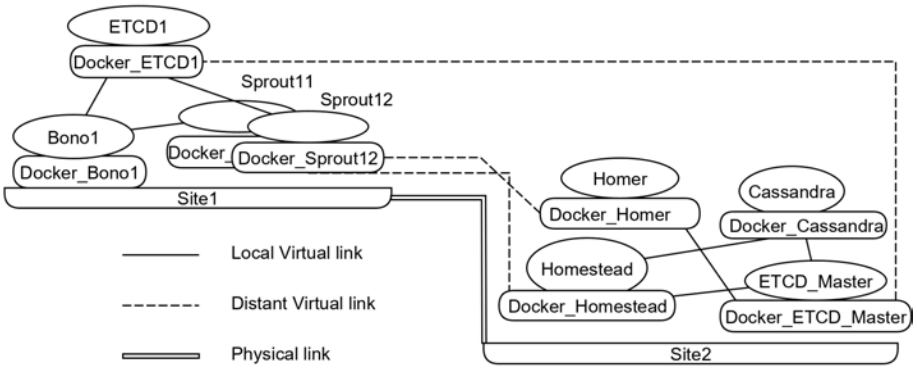


Figure 5.10 – vIMS two sites architecture use-case

In order to model the dependencies between the *Clearwater* vIMS components, we started by defining the associated templates. Most of the dependencies can be deduced from the expert knowledge and the IMS standard descriptions. These acquired knowledge enables us to define the first guesses about the definition of templates. However, some dependencies composing these templates are not easy to infer from the network and should be learned. One way to discover them is the fault injection process. The aim of the fault injection process is to verify that the described modeling rules or templates give correct explanation about failures injected in a real deployment. Otherwise, other hidden dependencies should be deduced. We will further discuss the learning of templates dependencies in Section 5.6.

5.4.1 *Clearwater* vIMS templates description

In this section, we describe the knowledge base that stores the defined templates for the Docker version of the *Clearwater* vIMS use-case. These templates have been deduced from the acquired knowledge extracted from description files and deployment (Cf. Chapter 4, Section 4.4), and learned knowledge provided from faults injection that we will depict in Section 5.6. These templates complement the global templates for virtual networks defined in Section 5.3 with additional dependencies that are specific to the *Clearwater* use-case. The *Clearwater* vIMS templates are described in the following. Each time, we showcase the additional dependencies that are specific to *Clearwater*.

Network elements templates description:

Figure 5.11 illustrates the distinct Network elements templates for the *Clearwater* vIMS use-case. The **site** ($T.site$) and **virtual host** ($T.VH$) defined for the *Clearwater* use-case are similar to the global templates of Section 5.2.

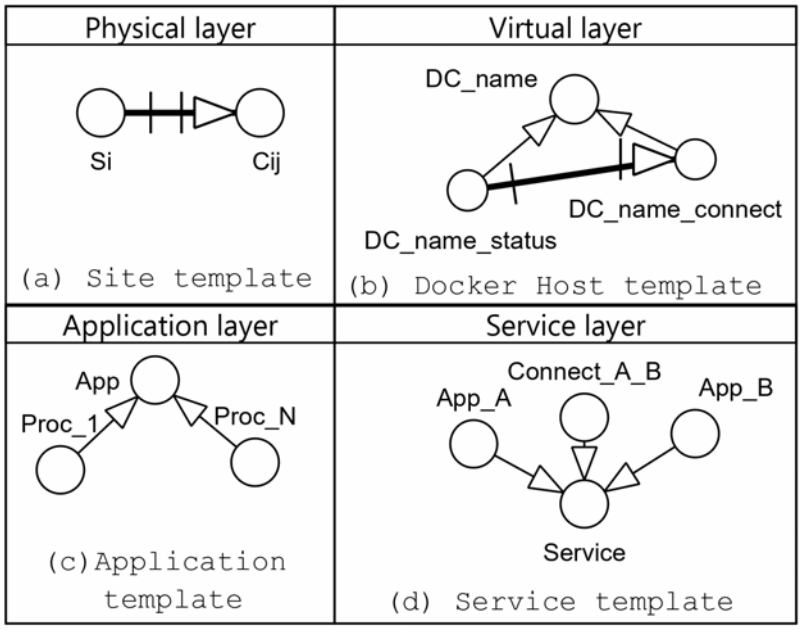


Figure 5.11 – *Clearwater* vIMS Network elements templates.

VNF application template ($T.APP$): differs between the *Clearwater* functions. Each function (i.e. Bono or Sprout) has its own definition that corresponds to the processes of each application. The general dependencies are similar to the one presented in Figure 5.11-c. The difference is in the type of processes. In Figure 5.12, we present the Bono and Sprout application templates.The Bono application template ($T.APP_Bono$) has two main process a process called Bono ($Proc_Bono$) and a process ETCD client ($Proc_Bono_ETCD_Client$) that enables the exchange with the master ETCD node. Whereas, Sprout has a Sprout process

($Proc_Sprout$) and the same type of process for ETCD client ($Proc_Sprout_ETCD_Client$).

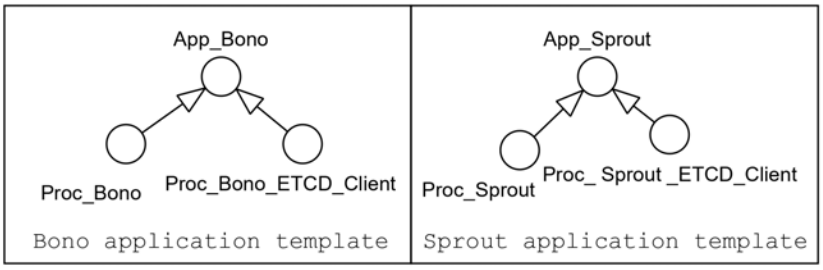


Figure 5.12 – BONO and Sprout application templates ($T.APP_Bono$, $T.APP_Sprout$).

Service template ($T.service$): defines the service dependencies. To model vIMS services, we should first learn and define the service dependencies. *Clearwater* provides a number of services such as registration of clients, texting, calls and video calls. In the following we depict the modeling procedure of a registration service.

Registration service modeling:

The client registration service represents an important and basic IMS service. The SIP registration service is a regular procedure for each client's terminal before initiating or receiving any other SIP signaling. The IMS registration procedure allows the IMS network to authenticate the client and authorize the establishment of sessions. The procedures required to meet the above prerequisites are defined in the SIP protocol and the *Clearwater* architecture [58].

The SIP registration process goes through two steps as depicted in Figure 5.13 and Figure 5.14. In the first step the subscriber's UE tries to make a first contact with the corresponding P-CSCF/Bono node. In this case, the UE sends a SIP register to the attached P-CSCF/Bono node. In this first step, the SIP register message doesn't contain any subscriber authentication information. The P-CSCF/Bono forwards the message to the Sprout node. Sprout node forwards the message to the Homestead node that verifies from the HSS/Cassandra database the subscriber credentials. Since the client doesn't send the authentication information, Sprout refuses the requests with a SIP code 400 (i.e. Unauthorized) through Bono. In the second step the subscriber's UE sends the SIP register message with authentication information. This time the request will be accepted if authentication credentials are correct with a SIP code "200".

As depicted in Figure 5.15, the registration procedure depends on the performance of each of: Bono, Sprout, Homestead and Cassandra applications and the connections between these applications. Note that each of the variables depicted in Figure 5.15, are represented in the application level (i.e. the layer just above the service layer), these variables have other inter-layer dependencies with the virtual layer that are not showed in Figure 5.15.

Network connections templates description:

In *Clearwater* Docker, the **physical connectivity template ($T.PC$)** illustrated in Figure 5.16-a

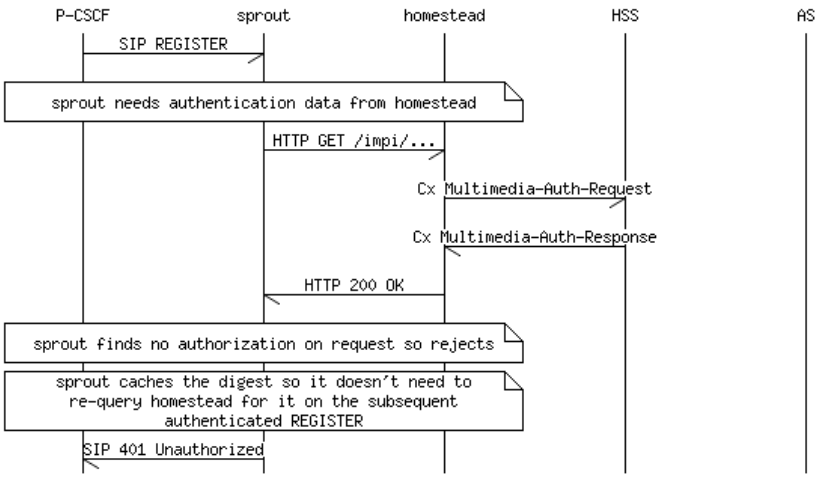


Figure 5.13 – The first SIP registration trial to the *Clearwater* vIMS [19].

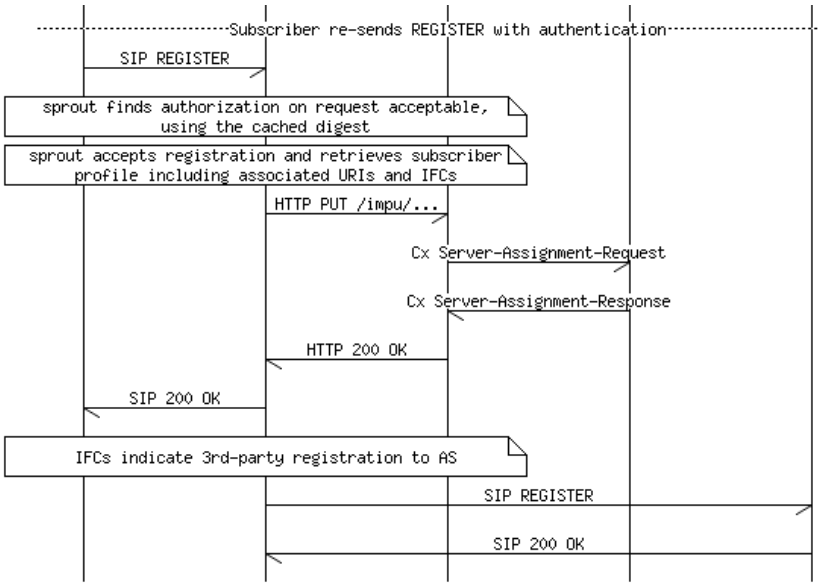


Figure 5.14 – The second SIP registration trial to the *Clearwater* vIMS [19].

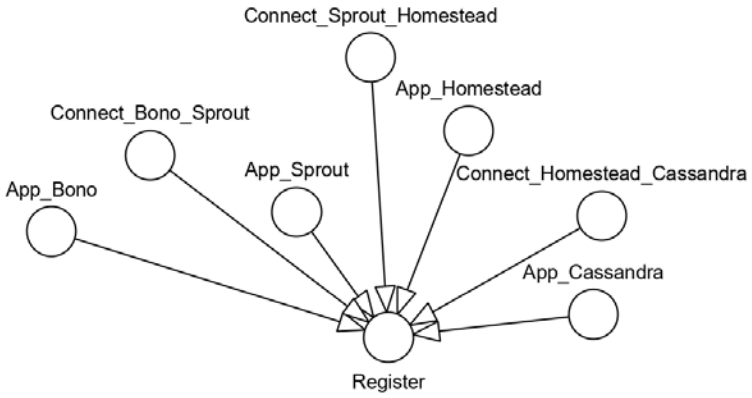


Figure 5.15 – vIMS SIP register service modeling.

is similar to the global template defined in Section 5.2. In the **local virtual connectivity template (**$T.LVC$**)** (Figure 5.16-b), the local network node is replaced with the network bridge (Nb). In fact, Docker engine connects the local Dockers with a shared network bridge. This network bridge has a status defined in the Boolean node Nb_i for each aggregated virtual environment V_i. The Nb_i node affects the Dockers connectivity $DC - Name_connect$ with a dependency type "$\neg A \implies \neg B$".

The **distant virtual connectivity template (**$T.DVC$**)** is described in Figure 5.16-c. In this template, the distant network node between $site_i$ and $site_j$ is replaced with the overlay network aggregated node (OV_ij) that is related with an "*AND*" logical relation to a node representing its status (OV_{status}), the Docker ETCD of each sites (DC_ETCD_i, DC_ETCD_j) and the convergence between the ETCD Dockers of both sites. In the Docker technology the overlay network is defined to connect distant Dockers. *Clearwater* defines a mechanism of sharing the network identities between the different components (i.e. IP addresses) through ETCD. The aim is to enable the different *Clearwater* functions to connect between each other even if they are distributed in distant servers. ETCD is a key-value store that forms a cluster with the *Clearwater* functions to share the network configuration files.

The **local logical connectivity template (**$T.LLC$**)** and **distant logical connectivity template (**$T.DLC$**)** are described the same way as in the global templates definitions. Since the network mechanism is ETCD in the case of *Clearwater*, node App_ETCD is added to the templates. In the case where the applications are hosted in distinct sites, the ETCD applications of both sites are added to the $T.DLC$ template.

Auto-recovery and elasticity:

Clearwater functions have the ability to scale. Each function such as Bono can be replicated. These functions share the SIP requests equally and if one replicate node is down, the traffic sent to this replica will be forwarded to the other replicas.

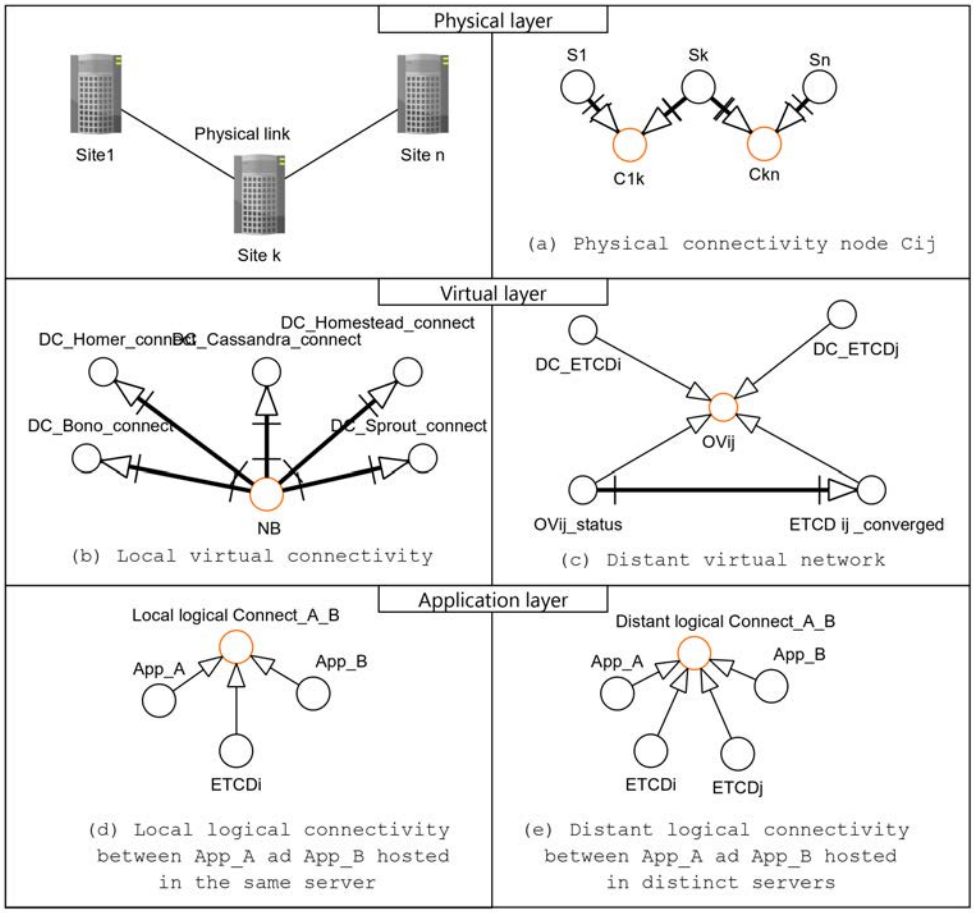


Figure 5.16 – *clearwater* vIMS network connections templates.

Figure 5.17 describes the elasticity template ($T.El$) of a *Clearwater* function in the virtual and application layer. The replicas in the case of virtual layer are Dockers: DC_name_1 and DC_name_2. They are linked with an "*OR*" relation to an aggregated node $ClusterD_name$.

Same procedure for the elasticity in the application layer, illustrated in Figure 5.17-(right), the aggregated node is called $ClusterA_name$. In this Figure 5.17, we illustrate the templates of two replicas of a Docker VNF in the virtual and application layer.

The auto-recovery mechanism in *Clearwater* was discovered while injecting faults to the network. We will depict this auto-recovery template in Section 5.6.

Aggregated nodes templates:

Aggregated nodes for sites ($T.An_S$) and virtual environment ($T.An_{V_i}$) are similar to the global templates definition. We depict these templates in Figure 5.18. In the aggregated template for virtual environment ($T.An_{V_i}$), the local virtual network is the network bridge node NB.

Inter-layer dependencies:

Figure 5.19 shows the links between the network layers. The application status is related to the

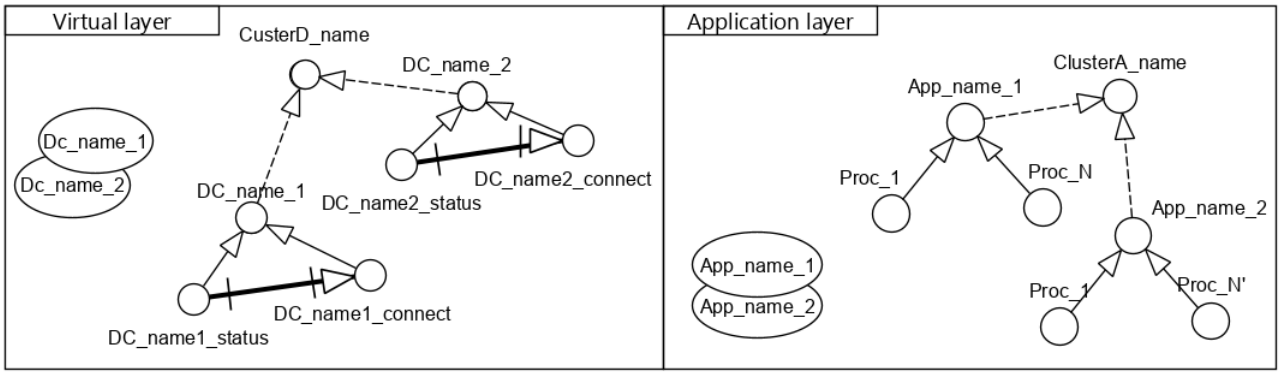


Figure 5.17 – *Clearwater* vIMS network elasticity modeling.

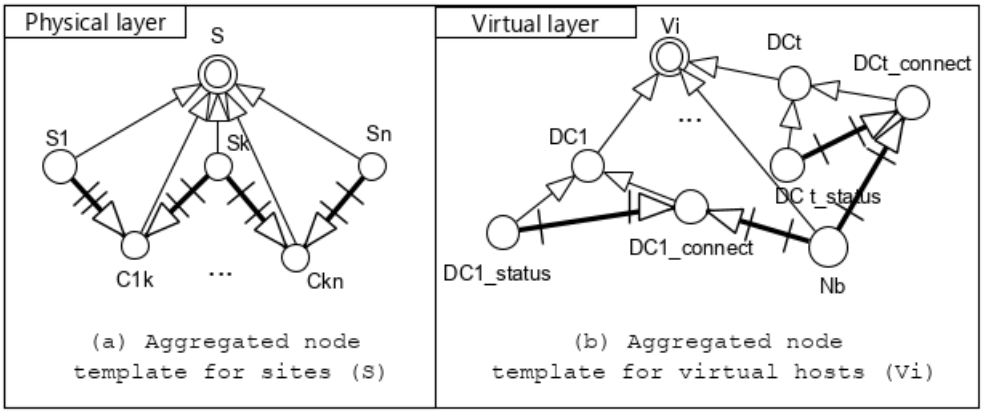


Figure 5.18 – *Clearwater* vIMS aggregated nodes templates ($T.An_S, T.An_{Vi}$).

Docker status which is in turn related yo the site or server status. This dependency is represented by the "$\neg A \implies \neg B$" type. In the case of elasticity of Dockers, each Docker replicas status affects the associated application status node such as illustrated in Figure 5.20. The local logical connectivity ($connect_AppA_AppB$) between two applications "A" and "B" hosted in the same site is related to the virtual connectivity of the Docker hosting the application "A" (i.e. $DC_A_connect$) and the one hosting the application "B" (i.e. $DC_B_connect$). For applications from distant sites, their logical connectivity depends on the overlay node that depends on its turn to the physical connections between the sites. In the case of elasticity of Dockers, an aggregated node $ClusterC$ is instantiated to aggregated the Dockers virtual connections. Figure 5.20 depicts this case with the elasticity of the application named "A".

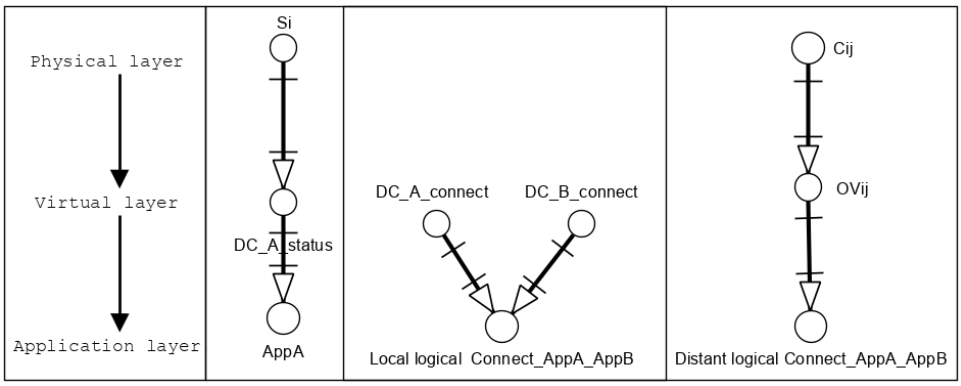


Figure 5.19 – Inter-layer dependencies.

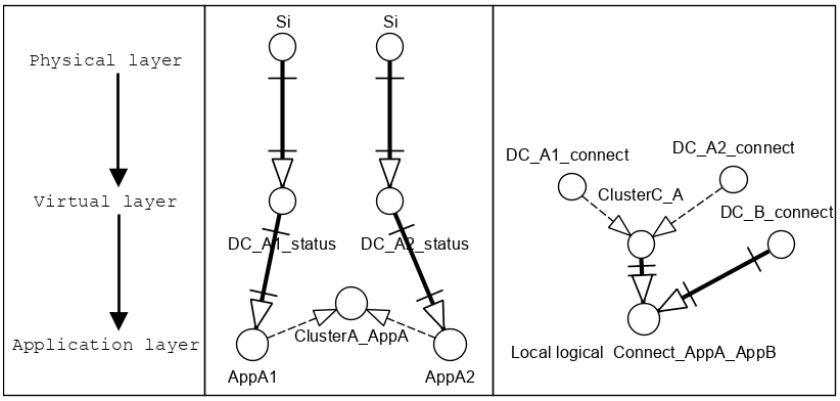


Figure 5.20 – Inter-layer dependencies with elasticity.

5.5 Self-modeling algorithm

Self-modeling represents the procedure of instantiating the defined templates in order to fit the current network topology. The dependency graph is constructed based on both the topology description and the knowledge base of templates (building rules). The self-modeling procedure is called in each failure detection to create the model that corresponds to the current network topology. The network topology could be retrieved by interrogating the orchestration platforms (e.g. Docker daemon) or through open source monitoring tools. We defined an YAML[2] file to describe the network topology. For instance, the YAML file for the architecture illustrated in Figure 5.10, is described in Figure 5.21. In this file we try to capture the network components and elasticity in order to instantiate them correctly. We have defined for each physical $site_i$ two fields: "$VNFs$" and "$Networks$". The "$VNFs$" field represents the VNFs hosted in each physical $site_i$ and are described as follows:

[2]YAML is a human-readable language for description and configuration files [101]

```
Sites:
  site1:
    VNFs:
        - VNF:
          - Name: bono1
          - nb: 1
          - monit: True
          - nb_VLcon: 2
          - nb_VDcon: 0
          - VLcon: sprout1
          - VLcon: E1
          - Type: Docker
        - VNF:
          - Name: sprout1
          - nb: 2
          - monit: True
          - nb_VLcon: 1
          - nb_VDcon: 2
          - VLcon: E1
          - VDcon: homer
          - VDcon: homestead
          - Type: Docker
        - VNF:
          - Name: E1
          - nb: 1
          - monit: True
          - nb_VLcon: 0
          - nb_VDcon: 1
          - VDcon: EM
          - Type: Docker
    Networks:
      - Network_bridge: NB1
      - etcd: E1     #None for nothing
      - nb_Pcon: 1
      - Pcon: 2     # 2 for site2
```

```
  site2:
    VNFs:
        - VNF:
          - Name: homer
          - nb: 1
          - monit: True
          - nb_VLcon: 2
          - nb_VDcon: 0
          - VLcon: EM
          - VLcon: cass
          - Type: Docker
        - VNF:
          - Name: homestead
          - nb: 1
          - monit: True
          - nb_VLcon: 2
          - nb_VDcon: 0
          - VLcon: EM
          - VLcon: cass
          - Type: Docker

        - VNF:
          - Name: cass
          - nb: 1
          - monit: True
          - nb_VLcon: 1
          - nb_VDcon: 0
          - VLcon: EM
          - Type: Docker
        - VNF:
          - Name: EM
          - nb: 1
          - monit: True
          - nb_VLcon: 0
          - nb_VDcon: 0
          - Type: Docker
    Networks:
      - Network_bridge: NB2
      - etcd: EM   #None for nothing
      - nb_Pcon: 0
```

Figure 5.21 – The YAML file for the architecture depicted in Figure 5.10.

- "$Name$": the name of the *Clearwater* VNF that enables to choose the appropriated templates for each VNF. Each VNF has a unique name that refers to its function (e.g. the first Bono in the network is $bono1$).
- "nb": represents the number of elasticity of the VNF.
- "$monit$": to define the presence of the auto-recovery node, since this mechanism can be disabled.
- "nb_VDcon" and "nb_VLcon": represent the number of the distant and the local virtual connections, respectively.
- "$VDcon$" and "$VLcon$": represent the distant and local virtual connections, respectively.

- "$Type$": defines the type of the virtualization technology (e.g. Docker).

The "$Networks$" filed is composed of the following attributes:

- "$Network_bridge$": defines the name of the local virtual network mechanism applied in the site. We suppose that we have one $Network_bridge$ in one site. Otherwise, each $Network_bridge$ will be represented separately with the correspondent VNFs connected to this network.

- "$etcd$": defines the name of the ETCD function in this site or "None" for other types of network configuration sharing.

- nb_Pcon: number of physical connections.

- $Pcon$: the physical connection.

Note that the defined YAML file is a unified way to describe the current network topology. This file can be constructed from the Docker engine or monitoring projects such as *Weave Scope* in the case of Docker. In our work, we supposed that we have this file. The modeling process takes as entry the YAML topology file and defines the corresponding dependency graph, as represented in the modeling process Algorithm 1. Note that $Pcon$, $Network_bridge$ and $etcd$ are lists of all the physical connections, network bridges and etcd present in the site, respectively. $VDcon$ and $VLcon$ fields are a list of all the distant and local connections of this VNF to the other VNFs in the network.

Algorithm 1: Self-modeling

Input: Topology: { $Sites$: list of sites
$Site_i$ = [$Network_bridge, etcd, nb_Pcon, Pcon$]
$nb.Sites$: number of sites
$VNFs(i)$: list of VNFs in a $site_i$
VNF = [$name, type, nb, nb_VLcon, nb_VDcon, VLcon, VDcon, monit$]
$nb.VNFs(i)$: Number of VNFs in $site_i$ }

Input: $Register_list$: list of registration nodes
Input: Knowledge base templates:{$T.site, T.VH, T.App, T.An_S, T.An_{V_i}$, ;
$T.service, T.EL, T.PC, T.LVC, T.DVC, T.LLC, T.DLC, T.Inter$}
Output: Global Dependency Graph $G = (V, E)$

1 **for** *each* $site_i$ *in* $Sites$ **do**
2 → Instantiating $T.site(i)$;
3 → Instantiating $T.PC(nb_Pcon, Pcon)$;
4 → Instantiating $T.DVC(site_i, Pcon)$;
5 **for** *each* VNF *in* $site_i$ **do**
6 → Instantiating $T.VH(name, type)$;
7 → Instantiating $T.App(name)$;
8 → Instantiating $T.LLC(nb_VLcon, VLcon)$;
9 → Instantiating $T.DLC(nb_VDcon, VDcon)$
10 **if** $nb > 1$ **then**
11 → Instantiating $T.EL(name, nb)$
12 **if** *(*$nb.VNFs(i) > 1$*) OR (*$nb.VNFs(i) == 1$ *AND* VNF*[nb] > 1)* **then**
13 → Instantiating $T.LVC(VNFs(i), Network_bridge)$;
14 → Instantiating $T.An_{V_i}(VNFs(i))$
15 → Instantiating $T.Inter(site_i, VNFs(i))$
16 **if** $nb.Site > 1$ **then**
17 → Instantiating $T.An_S(nb.Sites)$
18 **for** *each* $Register_i$ *in* $Register_list$ **do**
19 → Instantiating $T.service(Register_i)$

The self-modeling algorithm starts by instantiating for each $site_i$: the site template $T.site(i)$, the physical connectivity template $T.PC(nb_Pcon, Pcon)$ that generates the instances according to the list of physical connections to the $site_i$: "$Pcon$" with a length of nb_Pcon , and the distant virtual connectivity template $T.DVC(site_i, Pcon)$ that models the overlay network in the case of docker from $site_i$ to the sites of the list of $Pcon$ with the corresponding $etcd$ list if it is not empty.

In the second for-loop, the algorithm instantiates for each VNF_j in $site_i$: the virtual host template $T.VH(name, type)$ that takes as inputs the name of the VNF and the type (i.e. Docker in our case), the application template $T.App(name)$, the local $T.LLC(nb_VLcon, VLcon)$, and distant $T.DLC(nb_VDcon, VDcon)$ connectivity templates that takes as inputs the list of of all the local and distant connections of VNF_j and their lengths. The elasticity template $T.EL(name, nb)$ of VNFs is instantiated when the number of replication of the same VNF exceeds one. The local

virtual connectivity template $T.LVC(VNFs(i), Network_bridge)$ and the aggregated template for virtual hosts $T.An_{V_i}$ are instantiated when there is at least two distinct or replicated VNFs. If there is only one VNF in one site, there is no need to model the local virtual connectivity or aggregate the virtual hosts.

The $T.LVC$ template takes as input the name of the $network_bridge$ and the connected $VNFs(i)$ to this bridge. Once all the VNFs of the same sites has been instantiated the inter-layer template $T.Inter(site_i, VNFs(i))$ is instantiated to model the inter-layer dependencies. The same procedure in the case where the number of sites exceeds one, the aggregated template for sites $T.An_S(nb.Sites)$ is instantiated. In the end of the algorithm, the services are instantiated according to the list of register services that exist in the topology. This list is extracted from the network topology. We explicit this procedure in the next Section 5.5.1.

5.5.1 Procedure for listing registration services

For each vIMS network architecture there exists a number of possible registration services depending on the number of deployed vIMS functions. To define all the registration services and instantiate them in the model, we proposed the procedure depicted in Algorithm-2.

Algorithm 2: Defining the Registration service list

Input: Topology YAML file $TOPO$
Input: List of registration services $Register_list$
→ Get VNF_list , $VNF_i_Connections$, $List_Bono$, $List_Cassandra$ from $TOPO$;
→ Create the **NCT** graph from (VNF_list, $VNF_i_connections$);
→ Remove("$ETCD$", "$Homer$") from **NCT for** *each* $bono_j$ *in* $List_Bono$ **do**
    for *each* $cassandra_k$ *in* $List_Cassandra$ **do**
        → Get $path$ from ($bono_j$ to $cassandra_k$);
        if $path$ *exists* **then**
            → Add $path$ to $Register_list$

The procedure to define the registration list that contains all the possible registration services, consists in first creating the Network Connectivity Topology (NCT) graph that corresponds to the topology described in the current YAML topology file. This graph is constructed through the $VNF_i \in VNF_list$ and the local and distant connections of each VNF_i. The second step is to remove from this graph the functions that are not related to the registration service (i.e. $Homer$ and $ETCD$). The third step consists in retrieving all the paths from each root $bono_j \in List_Bono$ to each leaf $cassandra_k \in List_Cassandra$. Each path represents a register service. Each register service is stored in the $Register_list$ and will be modeled through the $T.service$ template such as depicted in Algorithm I. To illustrate this procedure, we

suppose that we have the architecture, in Figure 5.22. In this architecture we have two Bono proxy nodes "Bono-1" in "Site-1" and "Bono-2" in "site-2".

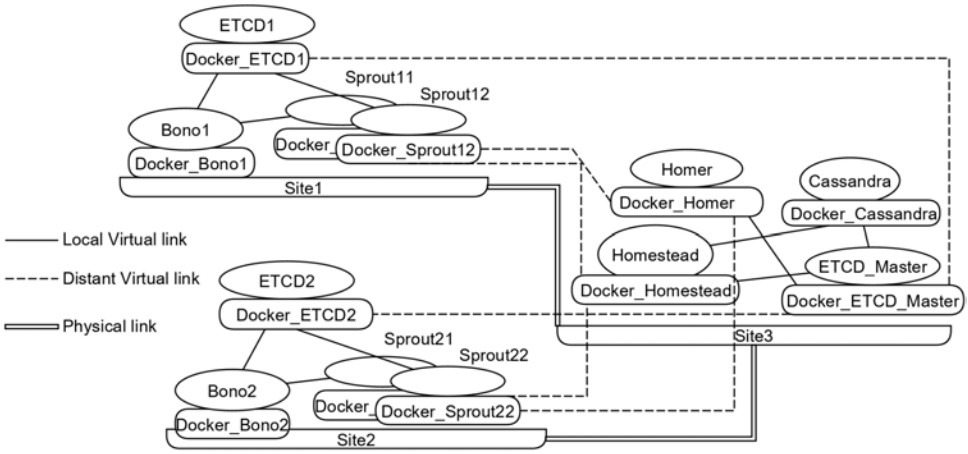


Figure 5.22 – a vIMS architecture use-case composed of three sites.

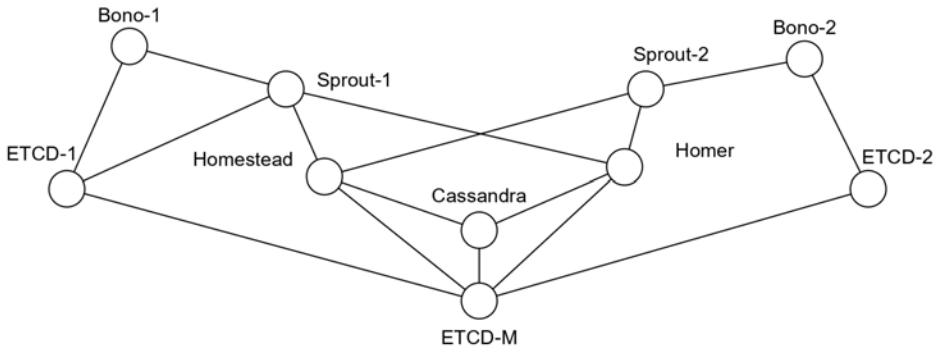


Figure 5.23 – The NCT graph for the vIMS architecture of Figure 5.22

The correspondent NCT graph is represented in Figure 5.24. The graph is represented following the VNFs and the connections described in the topology YAML file. After removing the "ETCD" and "Homer" nodes from the NCT, we extract each path from $bono$ to $cassandra$ that contains the following combination of nodes (Bono-Sprout-Homestead-Cassandra) (Cf. Figure 5.25). Each extracted path is a register service composed of a number of nodes that would be represented with the $T.service$ template as illustrated in Figure 5.25.

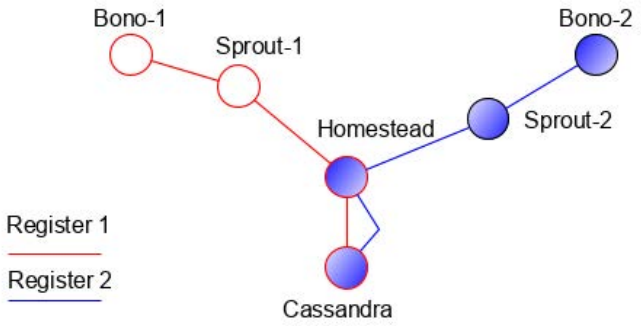


Figure 5.24 – The register services paths in the NCT graph

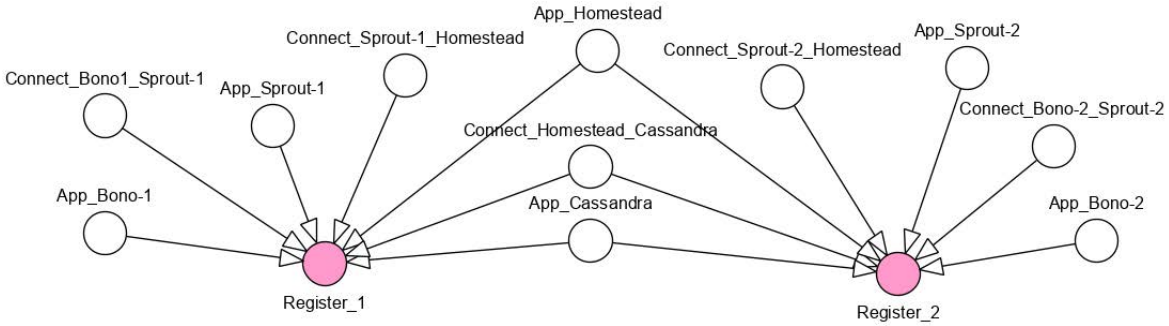


Figure 5.25 – Modeling of register services and their dependencies

5.5.2 The dynamicity of topologies

In order to illustrate how the self-modeling algorithm captures the network updates using the YAML file, we propose the following network update scenario:

To reduce the size of the dependency graph, we consider an example of two deployed VNFs (i.e. Bono and Sprout) hosted in one site. The YAML file that describes this example is depicted in Figure 5.26. In this YAML file only one site is defined. The site contains two VNFs (i.e. Bono1 and Sprout1). The local virtual network is "$NB1$". The physical connectivity $Pcon$ and the $etcd$ attributes are set to "$None$". Now suppose that due to traffic congestion, we scale the $Sprout1$ function to split the traffic through the Sprout replicas. At this moment, the new YAML file describing the current topology will change in the value of the attribute nb of the Sprout VNF that will be equal to "two": $(nb : 2)$ (C.F. Figure 5.26).

The dependency graph generated before and after the update is illustrated in Figure 5.27-a and 5.27-b, respectively. The changes are emphasized with in "orange". The elasticity of Sprout changes the dependency graph in both the virtual and the application layer. In addition, the self-modeling algorithm instantiates the cluster aggregated nodes: $ClusterD_Sprout1$,



Figure 5.26 – Topology network update captured in the YAML file: (left) before and (right) after the update.

$ClusterC_Sprout1$ and $ClusterA_Sprout1$.

Summary:

Capturing the network topology in the YAML file enables us to model the current network elements efficiently. This will help us consider the actual network elements in the RCA process and avoid false positives when pinpointing the root cause. However, once the RCA process is launched we consider that the network topology is the one captured when the failure to be explained by the process was detected.

5.6 Validation of the vIMS model

The defined templates stored in the knowledge base should be validated. To do so, fault injection scenarios are deployed. A fault injection scenario consists in a number of actions applied to the network. The actions represent a list of injected faults and healing actions. While building the templates of vIMS, we started by constructing a first representation of the templates with the knowledge we had from the description files of *Clearwater*. We then added new dependencies to these templates by learning (i.e. extending) and correcting the defined ones with fault injection.

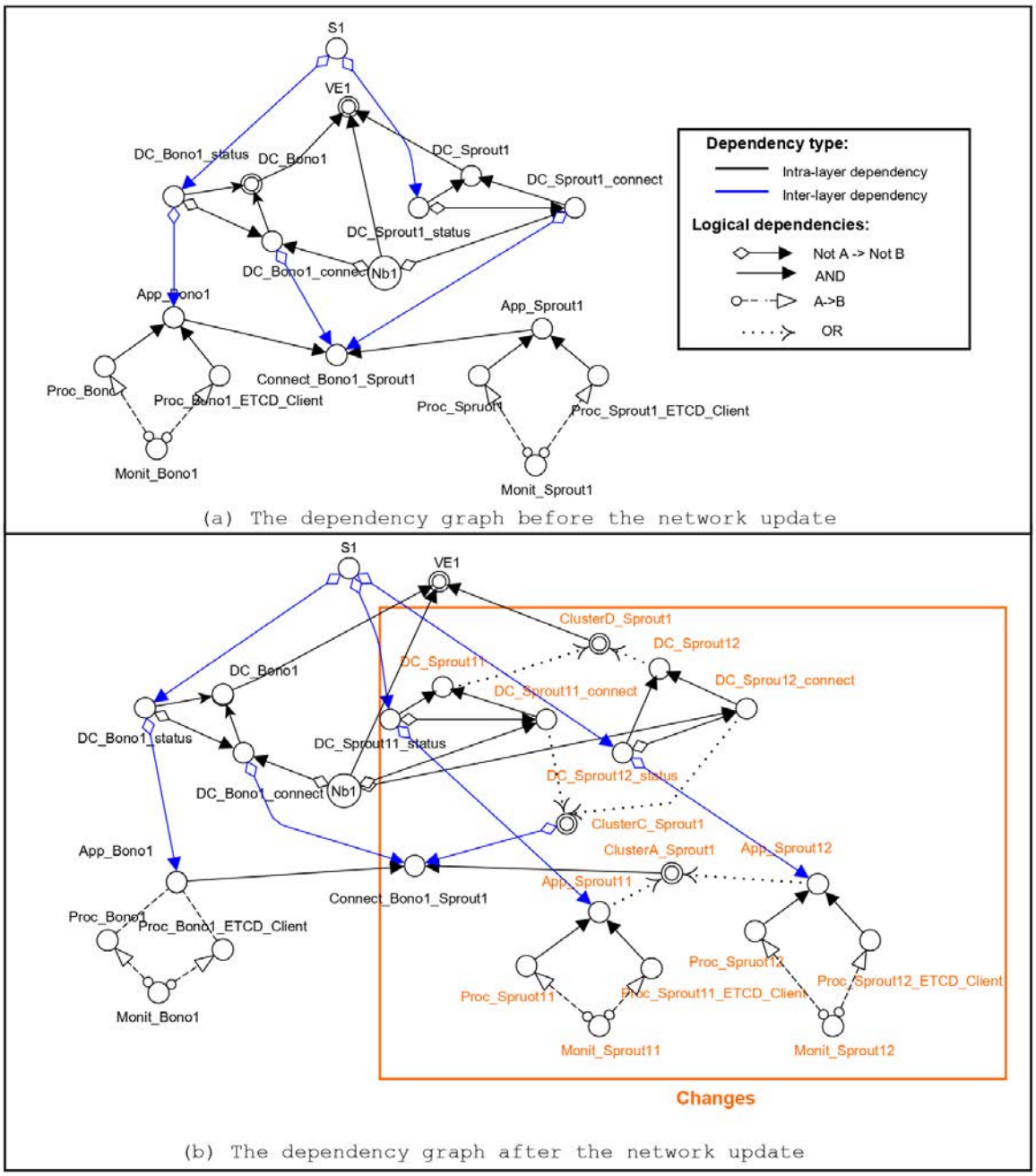


Figure 5.27 – The dependency graph of the YAML file in Figure 5.26 (before (a) and after (b) the update).

5.6.1 Templates validation

Table 5.1 depicts two examples of templates validation. The first fault injection scenario enabled us to learn new dependencies and the second fault injection scenarios allowed us to correct presumed dependencies. In the first scenario, we learned a new dependency for the application template. After stopping the Sprout process ($Proc_Sprout$) inside the Sprout Docker and tested the call and register services, both services was successful. We deduced from this fault injection, that an auto-recovery mechanism existed. This auto-recovery mechanism is a process called *Monit* that is present in every *Clearwater* Docker node.

Monit is responsible for restarting every defected process in the application level. Therefore, in the case of the presence of *Monit* (i.e. $Monit - Status = True$) the processes constituting the application must all be up. This relation is represented with the logical link type "$A \implies B$".

In the second scenario, we showcase a correction of the local logical connection template ($T.LLC$). In our first representation of the local logical connection template ($T.LLC$), the connection between the two applications depended on the presence of ETCD that shares the network configuration to the *Clearwater* functions.

However, in this scenario when we stopped the ETCD node, the call and register services was still functioning, even when we stopped the Sprout Docker and restarted it. The only case, where the services failed was when we killed the Sprout Docker and started a new one with a new network IP address. We discovered that, this is due to the presence of a memory cache where the network IP addresses are stored in each *Clearwater* function. If the network IP addresses of *Clearwater* Dockers don't change, the Dockers can still communicate even if ETCD is down. This fact is not represented in the old $T.LLC$ template. To correct the template, we define a $memory$ node. $memory$ is a Boolean variable that represents the network connection updates. For instance, in the case of a node "A" connected to "B", the Boolean "$memory$" variable between the nodes "A" and "B" is defined as the following:

$$val(memory) = \begin{cases} True & \text{if no network connection update} \\ False & \text{if network connection update} \end{cases}$$

Where "network connection update" between a node "A" and "B" happens when one of the nodes changes its IP address due to a migration or re-instantiation of the node. The new $T.LLC$ template illustrated in Table 5.1 is defined as the following:

$$val(connect_A_B)) = App_A \wedge App_B \wedge cluster_Event_ETCD.$$

$$val(cluster_Event_ETCD) = App_ETCD \vee memory.$$

Where $cluster_Event_ETCD$ is an aggregated node that links both the App_ETCD and $memory$ nodes with an "*OR*" logical relation. The "*OR*" relation represents the fact that the con-

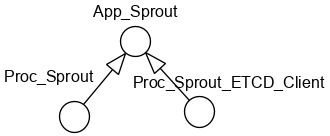
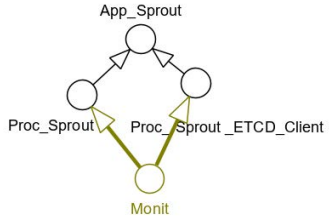
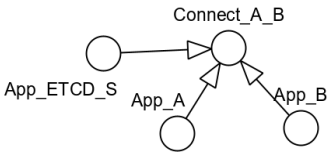
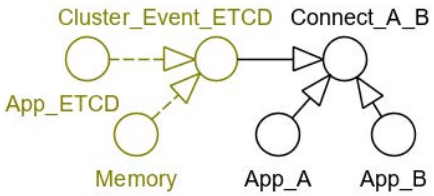
Fault Scenarios	Observations	Old templates	Correction and extension
1.Stop Sprout process in the Sprout application	Action(1): Call and register test works.	 Existence of an auto-recovery process *Monit*	Extension: 
Actions: 1- Docker Sprout stop 2- Docker ETCD stop 3- Docker Sprout restart 4- Docker Sprout Kill and run new Sprout	Actions (1, 2): - Alarm in Bono (unable to reach Sprout) - Alarm in service register and call (SIP code 408 Timeout) - Alarms in all nodes connected to ETCD (unable to reach ETCD) Action(3): - Call and register test works Action(4): - Alarm in Bono (unable to reach Sprout) - Alarm in service register and call (SIP code 408 Timeout)	Existent Knowledge:  Learned Knowledge: Existence of memory	Correction: 

Table 5.1 – Template extension and correction

nection between nodes A and B is "down" if both ETCD is down (i.e. $val(App_ETCD) = False$) tan the IP address of A or B changes (i.e. $val(memory) = False$). Given the Boolean value of "False" or "down status" is "0" and he Boolean value of "True" or "up status" is "1", the logical values for the aggregated node $cluster_Event_ETCD$ are defined in the following table:

	memory = 0	memory = 1
App_ETCD=0	0	1
App_ETCD=1	1	1

Note that the dependencies added in the local logical connectivity template $T.LLC$ is also applicable in the case of the distant logical connectivity $T.DLC$. We illustrate this in Figure 5.28.

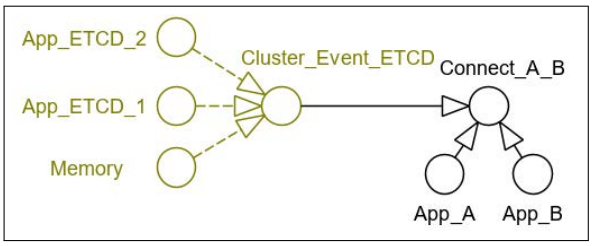


Figure 5.28 – The correction of the distant logical connectivity template.

Another, interesting result about this scenario is that the proposed self-healing action in the case where a vIMS Docker (e.g. Bono) and the ETCD connected to this Docker are down, would be rather restarting the old Docker to keep the same IP address than starting new one.

5.6.2 Validation of the model through a fault propagation use-case

In this section, we will showcase a validation of the global dependency graph representing the four layers of the model and not only a validation of a template. To do so, we propose the fault propagation use-case illustrated in Figure 5.29.

To capture a real fault propagation use-case, we followed these steps:

1. We deployed the *Clearwater* vIMS topology depicted in Figure 5.29.
2. We injected in this use-case a fault in the virtual layer by stopping the Bono1 Docker.
3. We injected a SIP register traffic through Bono1.
4. We collected two types of alarms: the SIP protocol code error 408 that indicates a request timeout and an alarm in the Docker Sprout1 that indicates that Bono1 is unreachable.

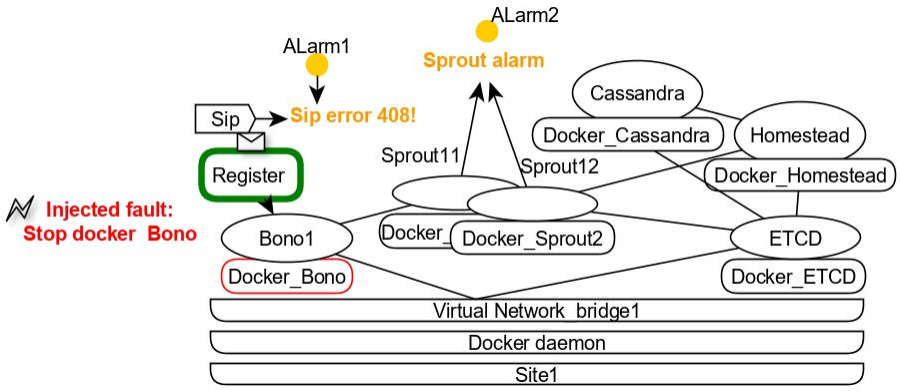


Figure 5.29 – *Clearwater* vIMS fault injection and propagation use-case.

The collected alarms indicate that the injected fault in the virtual layer affected the logical connection between Bono1 and Spout1 and the service register through Bono1. Which means that the fault propagated from the virtual layer to the application and service layer. Figure 5.30 depicts the dependency graph that represents the topology of Figure 5.29.

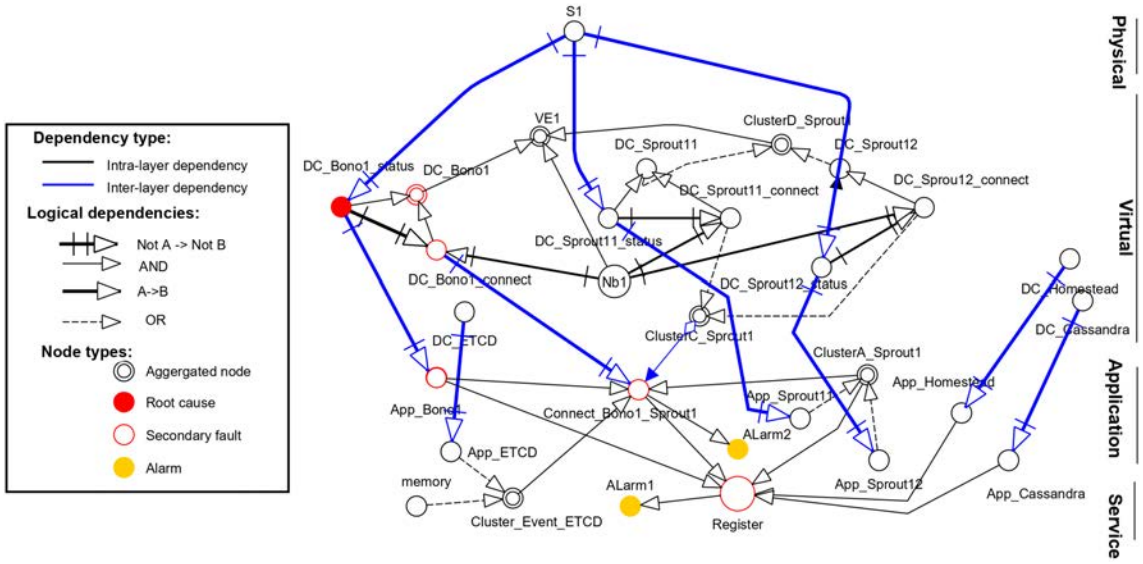


Figure 5.30 – The self-modeling resulted dependency graph that illustrates the fault propagation of the fault injection use-case in Figure 5.29.

In this dependency graph, we represent the injected fault in the DC_Bono1_status variable with a "False" status. The fault then propagated to other non-faulty components via inter-layers and intra-layer interactions.

We notice that the collected alarms matches the propagation in the defined graph. The

Docker Sprout1 alarm indicates that the status of the logical connection between Bono1 and Sprout1 is down (i.e. node $connect_Bono1_Sprout1$ is $False$), and the SIP register alarm indicates that the register service failed (i.e. node $Register$ is $False$). The dependency graph also illustrates the propagation of this fault from the Bono1 Docker status to the Bono1 application.
Summary: The defined dependency graph represents the multiple layers of virtual networks with the intra-layer and inter-layers dependencies. This graph enables as to provide an explicit presentation of a real world fault propagation use-cases. However, the main goal of the self-modeling algorithm is to use the derived dependency graph to pinpoint the root cause(s) of a network fault and to provide explanations about faults. This will be addressed in Chapter 6.

5.7 Experimental evaluation of the self-modeling algorithm

In this section, we evaluate the performance of the self-modeling algorithm. The first performance test studies the scalability of the generated dependency graph compared to the number of sites and VNFs in the network topology. The second performance test evaluates the self-modeling algorithm execution time. To evaluate the self-modeling algorithm, we implemented this algorithm in a python code using *Networkx Python* project [92]. The NetworkX library is applied to define and visualize the resulted dependency graph. The templates are defined in a knowledge base python file. The self-modeling algorithm and the knowledge base are stored in the following Github repository: [119]. The self-modeling algorithm evaluation tests was performed in a Intel Core i7-6500U CPU computer, with 8 GB of RAM.

5.7.1 Evaluation of the dependency graph scalability

In this evaluation, we study the behavior of the defined dependency graph with regards to the scalability of network topology. A dependency graph represents a number of vertices connected with a number of dependencies.

To study the evolution of the number of vertices and dependencies and the performance of the self-modeling algorithm when increasing the number of sites and VNFs, we propose the following topology configurations. We define two types of sites: "access" and "control" site. The "access" site contains the functions that enable a client to connect and to route the vIMS network, namely, Bono and Sprout. The "control" site contains the storing Cassandra database and the controlling homestead and homer functions. In both sites an ETCD node is deployed. These sites are depicted in Figure 5.31.

we have defined five examples of topology configurations by increasing the number of sites each time:

- Topo **A**: composed of two sites: one "access" and one "control".

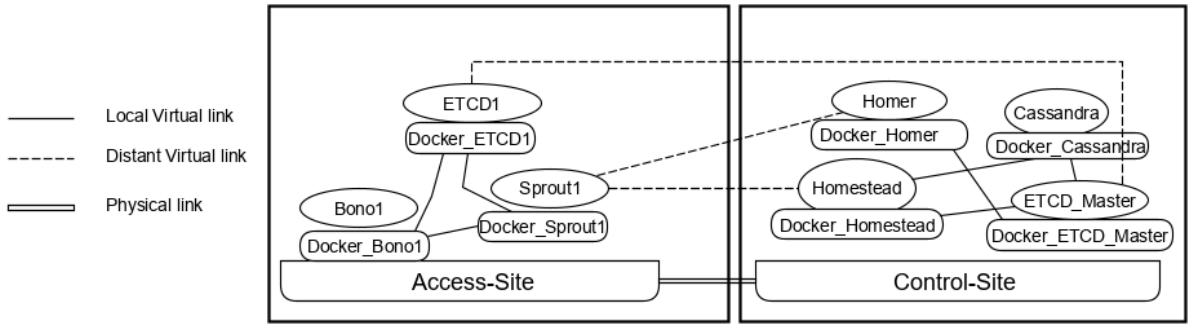


Figure 5.31 – Access and control sites.

- Topo **B**: composed of three sites: two "access" and one "control".
- Topo **C**: composed of four sites: three "access" and one "control".
- Topo **D**: composed of five sites: four "access" and one "control".
- Topo **E**: composed of ten sites: eight "access" and two "control".

Table 5.2 depicts the number of VNFs in each topology configuration. We increase the number of VNFs in each topology scheme addressed above with two different ways. The first manner is to replicate each *Clearwater* function. For instance "Topo*5" means that each function in the topology file is replicated five times and the five functions run in a redundant way. While "Topo**5" means that for each function there is five VNFs that are separated from each other and might be allocated to distinct tenants.

Topo/nb-VNF	A	B	C	D	E	F	G
Topo*1	7	10	13	16	32	48	64
Topo*5 / Topo**5	35	50	65	80	160	240	320
Topo*10 / Topo**10	70	100	130	160	320	480	640

Table 5.2 – Number of VNFs in each topology scheme.

The topology configurations described above have been implemented in separated YAML files. We then executed the self-modeling algorithm for each topology configuration defined in YAML files. Each time, we captured the number of vertices and dependencies using the Networkx library and the time of execution of self-modeling algorithm. The results are depicted in Figures 5.32, 5.33 and 5.34.

Figure 5.32 and 5.33 illustrate how the number of vertices and dependencies in the dependency graph increases when the numbers of deployed sites and VNFs augments. For instance,

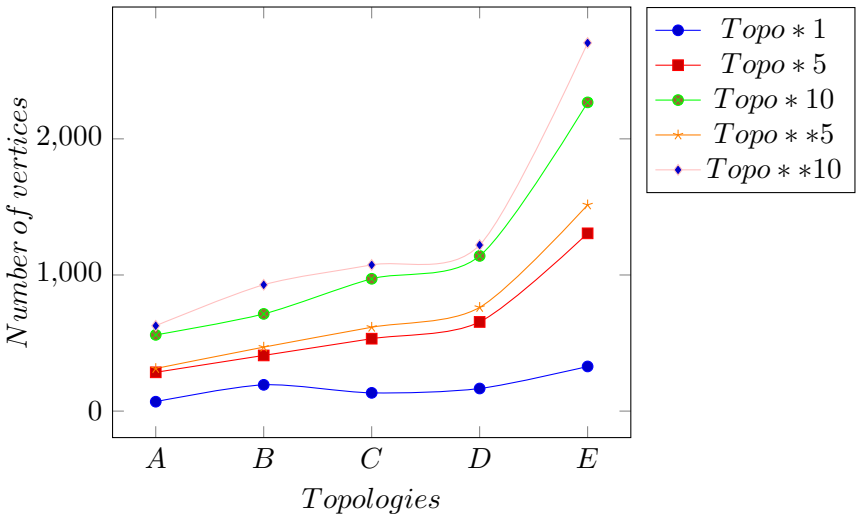


Figure 5.32 – Evolution of the Number of vertices according to the number of the deployed sites and VNFs.

for the topology "E" with 10 sites and 10 times the number of each VNF (i.e. TOPO**10) the number of vertices is around 3000 and the number of dependencies is around 8000.

We notice that even if topologies of type "Topo*i" and "Topo**i" with the same number of sites has the same number of VNFs, the number of vertices and dependencies is a little bit higher in the case where the multiplied VNFs are separated functions (i.e. "Topo**i") and not replicas (i.e. "Topo*i"). This is due to the additional vertices and dependencies in the "Topo**i" case. These dependencies are the logical connections between the different applications. In the case of replication only one connection node between the replicated application is considered.

5.7.2 Evaluation of the self-modeling algorithm performance

Figure 5.34 shows the evolution of the execution time of the self-modeling algorithm with regards to the different topology configurations. We notice that the execution time is less than 1 second whatever the number of sites we had in the case of replications (i.e.Topo*$_i$). However, it increases due to the multiplied VNFs in the Topo**$_i$ configuration.

From the previous results in Figure 5.33 and 5.32, we noticed a small difference between the number of vertices and dependencies between the two configurations Topo*$_i$ and Topo**$_i$. However, this small difference doesn't explain the considerable difference between the execution time of these two configurations.

In fact, this is due to the time that the self-modeling algorithm spend on reading the YAML file. The topology YAML file is much longer in the case where we define multiple VNFs. In the

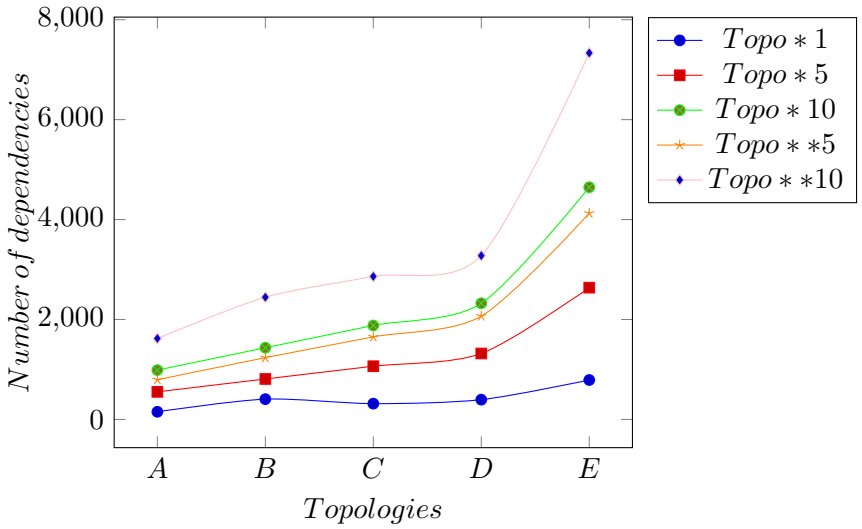


Figure 5.33 – Evolution of the Number of dependencies according to the number of the deployed sites and VNFs.

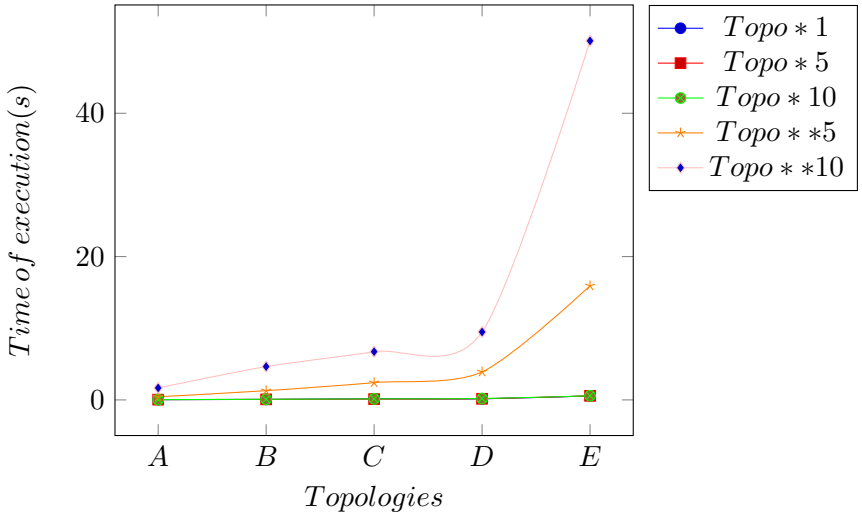


Figure 5.34 – Evolution of the execution time according to the number of the deployed sites and VNFs.

case of replicas, we only change the nb attribute of a VNF that represents its elasticity which explains the reduced YAML lines in this case.

5.8 Summary and conclusion

The evaluation of the performance of the self-modeling algorithm shows that the time for modeling a network topology described in the YAML is not negligible in the case when the network scales up. Scalability may cause the YAML file lines to grow which increases the self-modeling execution time while reading this file in order to model the network. This result showcases the importance of considering only the network updates to re-model the dependency graph.

However, in the diagnosis process that we will depict in Chapter 6, we assume that the topology changes are slow compared to faults and alarms propagation, so we don't change the YAML file, once the diagnosis process is launched. We only keep the last updated file. Therefore, this result is to be considered in the thesis perspectives, if one wishes to update the network during the diagnosis process, it would be more efficient to only model the changes.

Moreover, the size of the dependency graph might also affect the execution time of the diagnosis process due to the huge number of dependencies. To do so, in the next Chapter 6, we propose an active RCA process that reasons in on small portions of the dependency graph.

CHAPTER 6

ACTIVE FAULT LOCALIZATION

6.1 Introduction

In the previous chapter, we proposed multi-resolution and multi-layer logical templates for *Clearwater* vIMS. We defined the self-modeling process to enable the autonomic modeling and tracking of components in the network topology. However, to validate and showcase the value of the proposed modeling rules, we define in this chapter an active Root Cause Analysis (RCA) or diagnosis process.

The proposed diagnosis process is "*active*" and "*interactive*". *Active* because the diagnosis can incorporate new observations resulting from test results and "*interactive*" because the diagnosis process provides results under the form of a sub-graph that explains the propagation of the faults and the associated alarms. These propagation graphs enable the administrators to propose changes on the values of nodes and to introduce new observations in order to get more accurate results. The active diagnosis process takes into consideration as starting point the dependency graph resulting from the self-modeling process, and the initial observations. At each step/stage the result of the diagnosis algorithm is verified with tests. These tests orient the diagnosis process to the root cause. The diagnosis process returns an explicit dependency graph to the administrators with possible root causes. The diagnosis process reasons on portions of the global dependency graph to reduce the complexity of the constraints it resolves.

6.2 Definitions and notations

In the following, we describe the global definitions and notations applied in the active diagnosis process.

6.2.1 Global dependency graph and sub-graph

In Chapter 5 Section 5.3, we defined the different templates that compose the self-modeling approach. Each template is a directed graph composed of Boolean nodes and a set of directed edges. When a failure occurs, the self-modeling algorithm assembles instances of these templates according to the current network topology defined in the YAML description file, to form a

graph representing the network resources and their logical dependency relations. We call this graph, the **global dependency graph** $G = (V, E)$, with V as set of nodes V and $E \subseteq V \times V$ as the set of directed edges. The nodes represent Boolean (state) variables and the edges establish logical relations, as we recall below. We labeled each node $v \in V$ with several features $\ell(v) = (l_v, t_v, SF_v)$ where $l_v \in \{0, 1, 2, 3, 4\}$ is the network layer to which the node belongs: (0: physical, 1: virtual, 2: application, and 3: service), $t_v \in \{True, False\}$ indicates whether the state of node v is directly testable from the network ($t_v = true$, means testable), and SF_v indicates if the nodes can represent spontaneous fault or not. ($SF_v = 0$) for non-spontaneous node, ($SF_v = 1$) for a spontaneous node. We labeled each arc (v', v) from v' to v with a logical type $d_{(v',v)}$ satisfying $d_{(v',v)} \in \{\textit{AND}, \textit{OR}, \Rightarrow, \Leftarrow\}$. The arc types on graph G must satisfy local patterns, which are guaranteed by the self-modeling algorithm. Only five possible dependency configurations are allowed between node v and its predecessors, as illustrated in Figure 5.3 of Chapter 5. For example, the *OR* (resp. *AND*) type expresses that the logical variable v is the *OR* (resp. *AND*) of all its predecessors v'. Similarly, $d_{(v',v)} =$ " $\Rightarrow$ " stands for $v' \Rightarrow v$ (and in that case v' is the unique predecessor of v), and $d_{(v',v)} =$ " $\Leftarrow$ " stands for $\neg v \Rightarrow \neg v'$ (and in that case v can have several predecessors). Further, for a node v that represents a possible spontaneous fault ($SF_v = 1$), the above local logical dependencies must account for this extra degree of freedom (under the form of an "*AND* no spontaneous fault on v").

Given a dependency graph $G(V, E)$ for each node $v \in V$, we define:

- $S(v)$: the set of children of node v, i.e. its immediate successors in G,
- $P(v)$: the set of parents (immediate predecessors) of node v,
- $\hat{S}(v)$: the set of all successors of node v, so $S(v) \subseteq \hat{S}(v)$,
- $\hat{P}(v)$ the set of all predecessors of node v, so $P(v)] \subseteq \hat{P}(v)$.

We defined a **"valuation"** on G as a function $val : V \to \{False, True\}$ that assigns a logical value to each node $v \in V$ of graph G, and that satisfies all logical relations expressed by G. For node v representing some entity in the network modeled by G, $val(v) = True$ (resp $False$) expresses that the network entity modeled by v is up (resp down).

Later our diagnosis process will focus on a subset $V' \subseteq V$ of variables of G, defining a sub-graph SG of the global dependency graph G as follows: the **extracted dependency sub-graph** SG associated to nodes $V' \subseteq V$ is defined as $SG = (V', E')$ where the edge set $E' \subseteq V' \times V'$ is the restriction of E to variables V', i.e. $E' = E \cap (V' \times V')$. The choice of nodes V' will be explained later.

In our diagnosis process, when a spontaneous fault node v is suspected to be faulty, we will start by checking if that node is actually down (i.e. test the value of the node v). In the case

where it is confirmed that the fault is due to a propagation from lower layers in the graph, the predecessors of v will be investigated.

In the active diagnosis process a node v may be stated as:

- **Innocent**: if its value is known to be $True$ from the initial observations or tests (i.e. $val(v) = True$).
- **Faulty**: if its value is known to be $False$ from the initial observations or tests (i.e. $val(v) = False$).
- **Suspect**: if the node is suspected to be faulty (value $False$ is compatible with initial observations and tests performed so far) but the node is not testable ($t_v = False$) or a test over its value was not conclusive and returned "unknown". The suspect nodes indicate a possible root cause and should be tested first by the administrator.
- **Guilty/root cause**: if its value is known to be $False$ from the initial observations or tests (i.e. $val(v) = False$) and the node is a spontaneous fault node ($SF_v = 1$) with no predecessor or with predecessors and this node is confirmed by the administrator to be the cause of the failure.

6.2.2 Diagnosis engines

In this section, we present the different diagnosis engines that we will apply in the active diagnosis process. Given a dependency graph $G = (V, E)$, the nodes V are partitioned into observed and unobserved nodes: $V = O \uplus U$. An observation is a partial valuation over V that assigns a value to nodes of O only, i.e. $obs : O \to \{False, True\}$. We say that a failure occurs when at least one of these observed variables takes value $False$. We are interested in deriving all valuations $val : V \to \{False, True\}$ that are consistent with $G = (V, E)$ and that match/extend this observation, i.e. $val_{|O} = obs$. To discover the value of the unobserved nodes U, we will proceed progressively, by considering only relevant subsets of U. Specifically, we are going to focus on a subset of nodes $V' \subseteq V$, such that $O \subseteq V'$ (all observations are considered), and on partial valuations $val : V' \to \{False, True\}$, matching obs and consistent with the sub-graph $G' : G_{|V'} = (V', E' = E_{|V' \times V'})$.

The method consists in propagating/solving the logical constraints defined by G', starting from the value over observed nodes O given by obs. In practice, constraint propagation/solving determines uniquely the value of some unobserved nodes, for example when the edge $(v, v') \in O \times U$ is of "$\Rightarrow$" type, and $obs(v) = True$. Other unobserved nodes may not assume a unique value, i.e. there may exist two valuations val and val' that both match observation obs, and for some $v' \in U$ yield $val(v') = True$ and $val'(v') = False$.

Formally, performing a diagnosis would consist in exploring valuations val over V that both match observation obs and have a minimal number of root causes, that is nodes able of spontaneous failure, and then checking whether these faults are actually present in the network, in order to further eliminate possible explanations, until the true fault propagation pattern is discovered. To do so, one can focus on a subset of variables V', namely the predecessors $\hat{P}(O)$ of the observed nodes O. But unless if one has access to parsimonious SAT solvers that would minimize the number of faulty nodes, numerous partial valuations over V' might be returned. So we rather propose below an active and recursive procedure that will progressively explore G towards the predecessors (causes) of faulty nodes, to reveal and check possible causes, in order to keep under control the number of possible explanations to handle. In other words, we will not introduce right away in V' all variables of $\hat{P}(O)$, but will increase V' progressively.

In practice, to extend an observation $obs : O \rightarrow \{False, True\}$ into valuation(s) $val : V' \rightarrow \{False, True\}$, we will rely on a number of computation engines (sub-routines). These engines, structure the diagnosis process and their algorithms are defined later. To illustrate the use of these engines, we depict in the following the reasoning of the active diagnosis procedure that will be detailed in Section 6.4.

When a failure occurs, the active diagnosis procedure will follow the principles below.

1. Starting from an initial set O_0 of observed variables, with values defined by $obs_0 : O_0 \rightarrow \{False, True\}$, we extract a sub-graph SG of G (where again G results from the self-modeling algorithm, implemented in the $Self - modeler$ engine, fed by the current topology $Topo$). To do so, we select as variables in V' the observed nodes in O_0 plus the parents of faulty nodes in O_0. So $SG = G_{|V'}$ with $V' = O_0 \cup P(obs_0^{-1}(False))$. This procedure is performed by the $Extractor$ engine.

2. The $Diagnoser$ engine is then applied on the sub-graph SG. $Diagnoser$ relies on two main sub-functions, the $Translator$ and the $Solver$, and returns one valuation $s : V' \rightarrow \{True, False\}$ (among possibly many) that matches the observation obs_0. To do so, the $Translator$ engine first translates the sub-graph SG into a set of constraints (logical formulae). The result is fed to $Solver$, which performs logical inference and returns a first possible "solution" s that explains the observed failures. The $Diagnoser$ then extracts from this solution the possible root cause(s) by $R_0 = s^{-1}(False) \cap (V' \setminus O_0) \subseteq U$.

3. The possible root causes in R_0, i.e. nodes suggested as down by solution s, are then tested (provided they are testable). Then these nodes are incorporated in the set of observed variables through $O_1 = O_0 \cup R_0$, and the observed values on R_0 (either $False$ or $True$) is positioned to form the extended observation $obs_1 : O_1 \rightarrow \{False, True\}$. The whole process can start again (at point 1 above), with this larger observation set O_1 replacing O_0, in order to explore further (backwards) the propagation path of failures.

4. When checking the actual value of nodes in R_0, one may find some node v that either is not testable, or the test on its value returns "unknown". In such a situation, node v is tagged as suspect. The $Diagnoser$ keeps looking for larger solutions s until a spontaneous fault node is tested to be down. The actual value of node v may change in the next (extended) solutions found by $Solver$, or node v may remain suspect. Alternatively, one may also ask the administrator for an assumption about the value of this node v, as it is a possible root-cause or an intermediate fault propagation node.

5. The diagnosis process stops when the administrator is satisfied with the proposed root cause or when no more solution is proposed by the $Solver$. The result of the diagnosis process is a sub-graph with the last observations and the nodes stated as innocent, suspect, faulty and guilty.

In summary, our algorithm uses the following functions.

- $Self-modeler : (Topo, Templates) \rightarrow G$: takes as inputs the templates and the topology YAML file "$Topo$", and outputs a dependency graph $G = (V, E)$.
- $Extractor : (G, obs, O) \rightarrow SG$: takes as inputs the global dependency graph G and the observed nodes O and return a sub-graph $SG = (V', E')$, with $V' = O \cup P(obs^{-1}(False))$. $P(obs^{-1}(False))$ are the immediate predecessors (parents) of nodes assuming value $False$.
- $Translator : (SG, obs, O) \rightarrow \mathbf{A}$: takes as inputs the sub-graph SG and the observed nodes O, and generates a set of logical relations $\mathbf{A}(v)$ translated from the SG dependencies and observations. For eacn node $v \in V'$ of graph SG, we build a logical formula $\mathbf{A}(v)$ connecting the value of node v to the value of its predecessors.
- $Solver : \mathbf{A} \rightarrow solution$: takes as inputs all the logical relations and provides one possible solution s, i.e. a valuation over V' that satisfies all logical relations $\mathbf{A}$.
- $Tester : V \rightarrow \{True, False, Unknown\}$: provides the value of a node $v \in V'$ through testing the network (e.g. a Ping test). The test is possible only if the node is labeled as testable $t_v = True$. The selection of nodes to test is defined in the active diagnosis process.
- $Diagnoser : (SG, obs, O) \rightarrow s$: is the procedure that includes both the $Translator(SG, O)$ and the $Solver$ engines. It takes as inputs the sub-graph SG and the observed nodes O and outputs a partial valuation $s : V' \rightarrow \{False, True\}$, that represents a partial explanation (or “solution") to the observation of faulty nodes in obs. $Diagnoser$ is the central engine of the diagnosis process, and is recursively called to extend/refine the valuation s until the root cause of the observed failure is correctly identified.

6.3 Diagnosis in a sub-graph

In this section, we depict the $Diagnoser$ engine. The $Diagnoser$ provides a valuation $s : V' \rightarrow \{False, True\}$ that satisfies the observations. The node(s) proposed to be $False$ in this valuation (i.e. $s^{-1}(False) \cap U$) are possible root cause(s) and need to be checked. This engine will be applied in the active diagnosis process detailed in Section 6.4. The algorithm describing the $Diagnoser$ is defined as follows:

Algorithm 3: Diagnoser

Input: Dependency sub-graph $SG = (V', E')$
Input: a set O of observed variables through valuation $obs : O \rightarrow \{False, True\}$
Output: a valuation $s : V' \rightarrow \{False, True\}$

```
1 begin
2     A = Translator(SG, obs) ;
3     s = Solver(A) ;
4     return s
```

To translate the sub-graph and the observations into logical relations the $Translator(SG, O)$ proceeds as follows: each dependency sub-graph $SG = (V', E')$ consists of a set V' of nodes and a set $E' \subseteq V' \times V'$ of edges. The V' nodes are Boolean variables and the typed edges E' define logical dependencies. The $Translator$ defines for each node $v \in V'$ a logical relation using the type of link $d_{(v',v)}$ relating v to each of its predecessor v'. The values of the observed nodes are then added to these logical relations. The logical relations are stored in a file called the SMT file. The $Translator$ engine is detailed in Algorithm 4.

To illustrate the operation of the $Diagnoser$ engine, we assume that we have the sub-graph $SG = (V', E')$ of Figure 6.1. In this sub-graph, variables v'_1 and v'_2 are observed as faulty. The status of this nodes v'_1 and v'_2 depends on the status of their parents $\hat{P}(v'_1)$. In this example, either the failure of v'_1 is spontaneous, or it results from a fault propagation pattern from its predecessors $\hat{P}(v'_1)$. The objective is thus to estimate a valuation $val : V' \rightarrow \{False, True\}$. In order to estimate this valuation, the first step consists in defining the correspondent logical relations to the dependencies presented in Figure 6.1. The $Translator$ algorithm 4 starts by adding the initial observations:

$$\begin{cases} v'_1 = False \\ v'_2 = False \end{cases}$$

The $Translator$ algorithm 4 then adds the logical relations for the nodes that have at least one predecessor. In our example, nodes v'_1 and v'_2 have more than one predecessor with a link type AND and OR, respectively. The logical relations $\mathbf{A}$ of the example in Figure 6.1 are the following:

Algorithm 4: Translator

Input: Dependency sub-graph $SG = (V', E')$
Input: a set of $O \subseteq V'$ with a partial valuation $obs : O \to \{False, True\}$
Input: a global valuation : $val : V' \to \{False, True\}$
Output: Logical relations $\mathbf{A}$

begin
  // Initializing the constraints $\mathbf{A}$ with the observations ;
  $\mathbf{A} := obs$;
  // Adding the sub-graph dependencies to the file $\mathbf{A}$;
  for $v \in V'$ **do**
    $\mathbb{P}_{AND} := \{v' \in P(v); d_{(v',v)=AND}\}$;
    $\mathbb{P}_{OR} := \{v' \in P(v); d_{(v',v)=OR}\}$;
    $\mathbb{P}_{\Leftarrow} := \{v' \in P(v); d_{(v',v)="\Leftarrow"}\}$;
    $\mathbb{P}_{\Rightarrow} := \{v' \in P(v); d_{(v',v)="\Rightarrow"}\}$;
    if $\mathbb{P}_{AND} \neq \emptyset$ **then**
      $\mathbf{A} := \mathbf{A} \cup \{val(v) = \bigwedge_{v' \in \mathbb{P}_{AND}} val(v')\}$
    if $\mathbb{P}_{OR} \neq \emptyset$ **then**
      $\mathbf{A} := \mathbf{A} \cup \{val(v) = \bigvee_{v' \in \mathbb{P}_{OR}} val(v')\}$
    if $\mathbb{P}_{\Rightarrow} \neq \emptyset$ **then**
      $\mathbf{A} := \mathbf{A} \cup \{val(v') \implies val(v), v' \in \mathbb{P}_{\Rightarrow}\}$
    if $\mathbb{P}_{\Leftarrow} \neq \emptyset$ **then**
      $\mathbf{A} := \mathbf{A} \cup \{\neg val(v') \implies \neg val(v), v' \in \mathbb{P}_{\Leftarrow}\}$
  return A

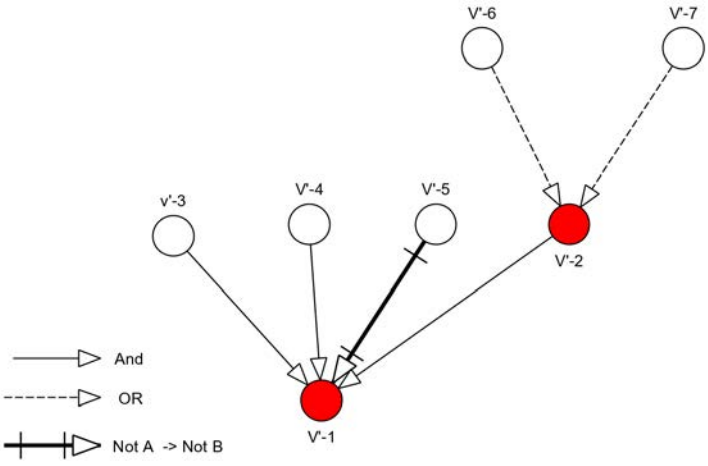


Figure 6.1 – An example of a sub-graph representing the nodes V-1 and V-2 failures and their predecessors

$$\mathbf{A}: \begin{cases} v'_1 = v'_3 \wedge v'_4 \wedge v'_2 \\ Not(v'_5) \implies Not(v'_1) \\ v'_2 = v'_6 \vee v'_7 \end{cases}$$

The next step of the $Diagnoser$ engine is to solve the logical relations $\mathbf{A}$. The solver should return a solution (a valuation) with a minimum of $False$ nodes. To do so, we obtain all the possible solutions of a configuration of assumptions and choose the solution with the minimum number of $False$ nodes. The $Solver$ takes as inputs the following logical relations $\mathbf{A}$:

$$\mathbf{A}: \begin{cases} V'_1 = False \\ V'_2 = False \\ V'_1 = V'_3 \wedge V'_4 \wedge V'_2 \\ Not(V'_5) \implies Not(V'_1) \\ V'_2 = V'_6 \vee V'_7 \end{cases}$$

The following $s : V' \rightarrow \{False, True\}$ represents one possible valuation that solves the logical relations $\mathbf{A}$:

$$s: \begin{cases} v'_1 = False \\ v'_2 = False \\ v'_3 = False \\ v'_4 = True \\ v'_5 = True \\ v'_6 = False \\ v'_7 = False \end{cases}$$

In this result, the $False$ unobserved nodes are $R_0 = s^{-1}(False) \cap (V' \setminus O_0) \subseteq U = \{v'_6, v'_7\}$. So both v'_6 and v'_7 are suspected to be the root cause(s) of the fault. The active diagnosis process that we will present in Section 6.4, operates a number of tests to check these nodes and extends the graph SG if necessary.

6.4 Active diagnosis process

The active diagnosis process starts after an anomaly has been detected. The network failure generates a number of alarms that generally state a failure of a network component, connection or service. These events or observations are considered as inputs to the diagnosis process. For instance, an alarm in the Sprout application that states that the Bono Docker is unreachable means that the logical connection between Sprout and Bono is down (i.e. "$Connect_Bono_Sprout = False$"), and is added to the observations O.

When the failure is detected in the network, the observations and the current network topology YAML file are the first inputs of the diagnosis process. The diagnosis process starts by generating the global dependency graph from the YAML file using the self-modeling algorithm ($Self - modeler$). The second step consists in extracting a sub-graph from the global dependency graph that represents the observations and the parents of the observations O. This step is operated by the $Extractor$ engine defined as follows:

Algorithm 5: Extractor

Input: Global dependency graph $G = (V, E)$; set O of observations, with a valuation obs
Output: Dependency sub-graph SG with valuations val

begin
  $val := obs$;
  $V' := O \cup P(\{v \in O \mid obs(v) = False\})$;
  $E' := (V' \times V') \cap E$;
  return (V', E')

In the third step, the $Diagnoser$ engine is applied. In this step, the sub-graph with the observations are translated into constraints (logical relations). The constraints are then resolved by the $Solver$. The $Solver$ provides one possible valuation with nodes stated to be faulty (i.e. False value). The active diagnosis process then checks their values and extend the sub-graph if necessary, each time, by considering the $Diagnoser$ solution and the additional observations obtained by tests until one gets a sufficient explanation to the detected failure.

The different steps of the active diagnosis process with the extension of the graph are illustrated in the flowchart of Figure 6.2. In the active diagnosis process a node v may be stated as: innocent, suspect, faulty or guilty. Note that the faulty and guilty nodes both have a down status and the guilty node is responsible for the failure.

The steps of the flowchart in Figure 6.2 are detailed below:

- `Step 0`: model the global dependency graph G using the $Topo$ file and the templates $Self - modeler(Topo, Templates)$.
- `Step 1`: extract the sub-graph SG from the global dependency graph that represents the observed nodes and the predecessors of $False$ nodes $Extractor(G, O)$.
- `Step2` $Diagnoser$:
  - `Step 2.1`: define the SMT file that contains the logical constrains $\mathbf{A}$ of the sub-graph SG with the values in the observations $Translator(SG, O)$.
  - `Step 2.2`: apply the SMT $Solver(\mathbf{A})$ to get a valuation $s : V' \rightarrow \{False, True\}, S \subseteq U$. With $R_i = s^{-1}(False) \cap (V' \setminus O_i) \subseteq U$. If no more solutions are possible then step 5.

- `Step 3`: For each node $v* \in R_i$ do:

  - `Step 3.1`: if the proposed root cause $v*$ is not testable (i.e. $t_{v*} = False$), or testable (i.e. $t_{v*} = True$) with an $unknown$ value, then the node $v*$ will be marked as suspect.
  - `Step 3.2`: if the proposed root cause is testable with a $True$ value, then the node $v*$ will be marked as innocent ($val(v*) = True$) and added to the set of observations $v* \in O, s(v*) = True$.
  - `Step 3.3`: if the proposed root cause is testable with a $False$ value:
    - If the node $v*$ is not a spontaneous fault (i.e. $SF_{v*} = 0$), then the node $v*$ will be marked as faulty and added to the set of observations $v* \in O, s(v*)^{-1} = False$.
    - If the node $v*$ is a spontaneous fault (i.e. $SF_{v*} = 1$) and has no predecessors (i.e. $P(v*) = \emptyset$), then the node $v*$ is a root cause (guilty).
    - If the node $v*$ is a spontaneous fault ($SF_{v*} = 1$) and has at least one predecessor (i.e. $P(v*) \geq 1$), then the node $v*$ is a possible root cause. The administrator can order to continue (step 1) or to exit (step 4).
  - If no more nodes in $s^{-1}(False)$ go back to step 2 to check another valuation.

- `Step 4`: exit with a sub-graph with the nodes marked as "guilty", "suspect", "faulty" and "innocent".

If we go back to the example of Figure 6.1, the results obtained from the $Diagnoser$ is the set $s^{-1}(False) = \{v'_6, v'_7\}$. The next step following the diagnosis flowchart in Figure 6.2 applied in this example are resumed as follows :

- If the nodes v'_6 and v'_7 are testable and spontaneous fault so if they are roots of the graph (i.e. number of predecessors equals to 0) and the test is "Down or $False$", so both of nodes v'_6 and v'_7 are root causes, if they are not roots or not spontaneous fault, we extend the graph to the predecessors of node v'_6 and v'_7 and repeat the same procedure.

- If they are not testable or testable and the test is "unknown", so both of them are suspects. The same reasoning is applicable in the case where the values of T_{v*} and SF_{v*} for nodes v'_6 and v'_7 are different. For instance, node v'_6 is tested "$False$" and spontaneous fault and node v'_7 is tested "$False$" and not spontaneous fault, so node v'_6 is a root cause if it has no predecessors and the fault in v'_7 is due to its predecessors.

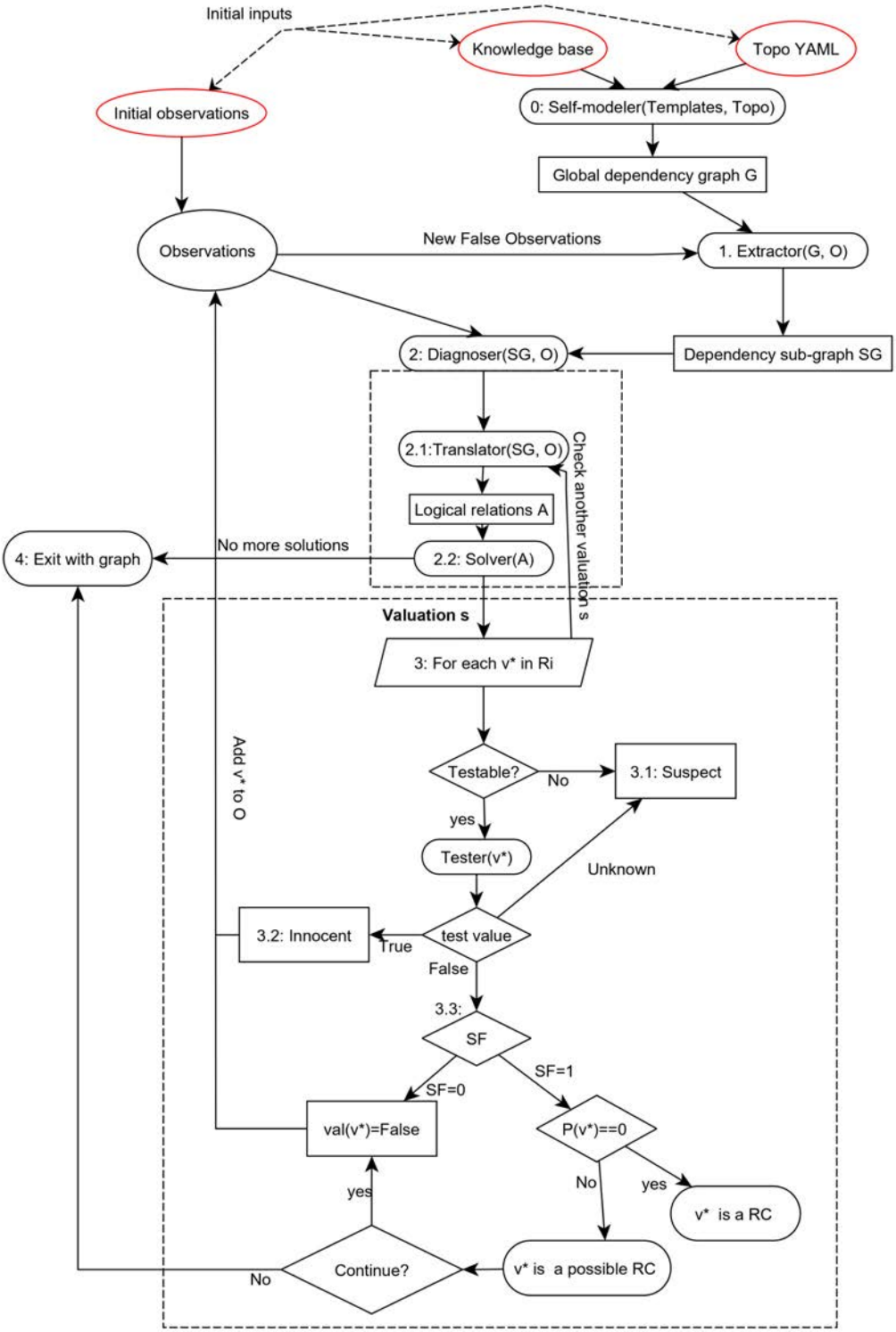


Figure 6.2 – Active Diagnosis process flowchart

6.5 Discussion

In our diagnosis process, we opted for a method that reasons on a portion of the dependency graph composed of the observations and of the parents of nodes observed to $False$. The aim is to reduce the number of unknown nodes to the immediate possible causes of a malfunction, in order to minimize the number of solutions proposed by the $Solver$, that must then be validated through testing. However, limiting the extension of the graph to only the parents of $False$ nodes has some drawbacks, as some useful logical constraints could be ignored. In particular, some observations assuming the value $True$ might exonerate a suspected node through a long causality chain. Our choice thus possibly increases the number of suspects.

Figure 6.3, illustrates an example of a dependency graph composed of nodes $\{v_1, ..., v_{10}\}$ with nodes $\{v_1, v_5\}$ are the first observations. The extracted sub-graph SG with the initial observations is composed of nodes $\{v_1, v_2, v_3, v_5\}$ (circled in blue in Figure 6.3). These nodes are the initial observations nodes $O_o = \{v_1, v_5\}$ and the parents of $False$ nodes $P(obs^{-1}(False)) = P(v_1) = \{v_2, v_3\}$. In one of the solutions proposed by the $Solver$ given the sub-graph SG, v_2 is proposed to be $False$. In the case where v_2 is not testable or the test is "Unknown", this node will be stated as suspect.

However, from the successors of node v_2, we notice that node $v_5 \in \hat{S}(v_2)$ is $True$. This node will innocent v_2 since the logical dependencies are of type ("*AND*" and implication). Since this kind of situation is not captured in the proposed sub-graph, we propose two possible improvements to the diagnosis process.

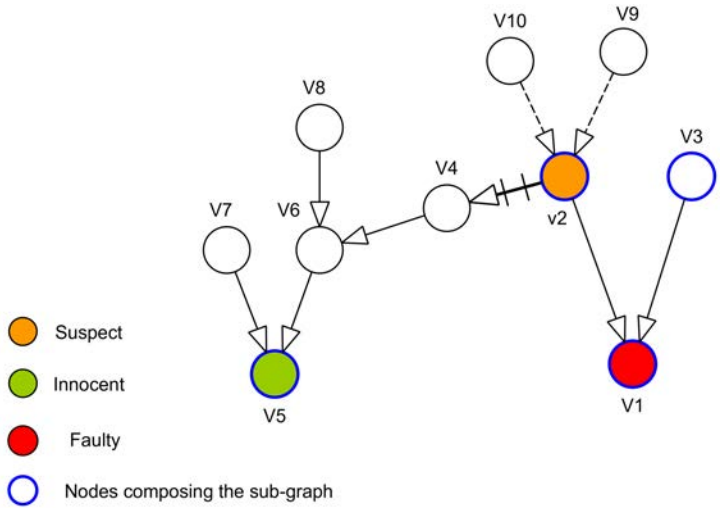


Figure 6.3 – An example of a dependency graph with 10 nodes and two initial observations $obs(v_1) = False$, $obs(v_5) = True$ and the node v_2 stated as suspect.

1. The first method consists in a preliminary procedure that deduces all the $True$ nodes from the $True$ observations $obs^{-1}(True)$. To do so, we look in the predecessors of $True$ nodes $\hat{P}(obs^{-1}(True))$ if there are nodes that can be deduced as $True$ from the dependencies (i.e. *AND* and implication types). The deduced nodes are then added to the observations before launching the $Diagnoser$ and thus included into the sub-graph SG. This enables to reduce the number of suspected nodes and minimises the number of solutions proposed by the $Solver$.

   We illustrate this method in Figure 6.4 with the same previous example. With the preliminary procedure the node v_5 will innocent nodes v_7, v_6, the node v_6 will innocent in its turn nodes v_4, v_8, and finally node v_4 will innocent node v_2. All the innocent nodes will be included to the observations $O_1 = O_0 \cup \{v_2, v_4, v_6, v_7, v_8\}$ and $obs(v) = True, v \in \{v_2, v_4, v_6, v_7, v_8\}$. Once the nodes are added to the observations, the extracted sub-graph will include this nodes. In this case, node v_2 is stated as innocent and will not appear in the suspected nodes. This method will increase the sub-graph with the new observed nodes and decrease the number of suspects.

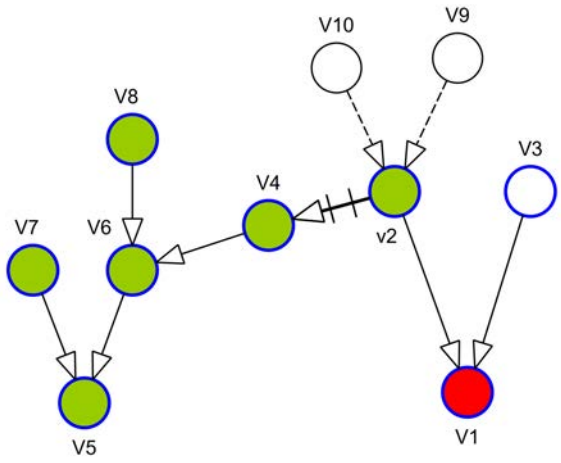


Figure 6.4 – The first method applied to the example of Figure 6.3.

2. The second method proceeds in the reverse way. Instead of introducing extra observations to SG beforce computations start, SG remains unchanged and one rather validates a posteriori the explanations proposed by $Solver$. This consists in checking the successors of a suspected node v before stating that this node is indeed suspect. Specifically, if node v has one or more of its successors displaying a $True$ value, this might exonerate node v.

   We illustrate this method in Figure 6.5 with the same previous example. In this case,

when the node v_2 is suspected the next step is to check the successors of node v_2: $\hat{S}(v_2) = \{v_4, v_5, v_6\}$. From the successors of node v_2, node $v_5 \in \hat{S}(v_2)$ is $True$. This will innocent nodes v_6, v_4 and v_2. In this case, node v_2, will be stated as innocent and the new sub-graph will only include the successors of node v_2 that are $True$: $O_1 = O_0 \cup \{v_2, v_4, v_6\}$ and $obs(v) = True, v \in \{v_2, v_4, v_6\}$.

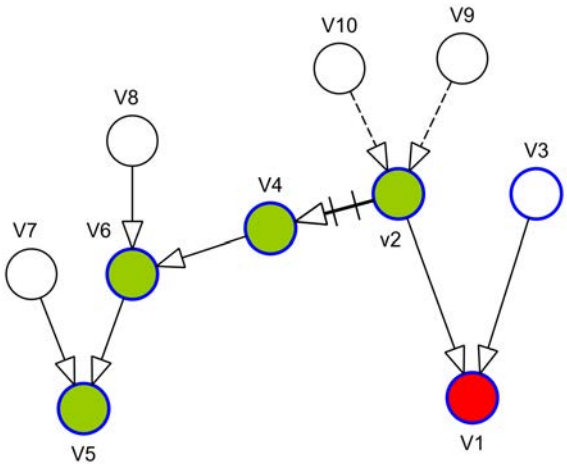


Figure 6.5 – The second method applied to the example of Figure 6.3.

6.6 Experimental results

To validate the proposed diagnosis methodology, we implemented the knowledge base, the self-modeling module and the active diagnosis process in a *Python3* environment. The proposed RCA framework depicted in Figure 6.6 is hosted on the GitHub project: [119]. The applied SMT $Solver$ is the Z3 logical modeling and solving project developed by Microsoft Research [83]. Z3 resolves a number of logical constraints or assumptions defined in a file called the SMT file.

To showcase the validation of our fault localization process, we propose three types of faults with different granularity levels.

1. **Docker disconnect**: in this fault scenario, the Docker is disconnected. The root cause is the Dc_name_C.
2. **Docker status stop**: in this fault scenario, the Docker is stopped. The root cause is the Dc_name_S.
3. **VNF process stop**: in this fault scenario, One important process of the VNF application is stopped. The root cause is the $Pname$.

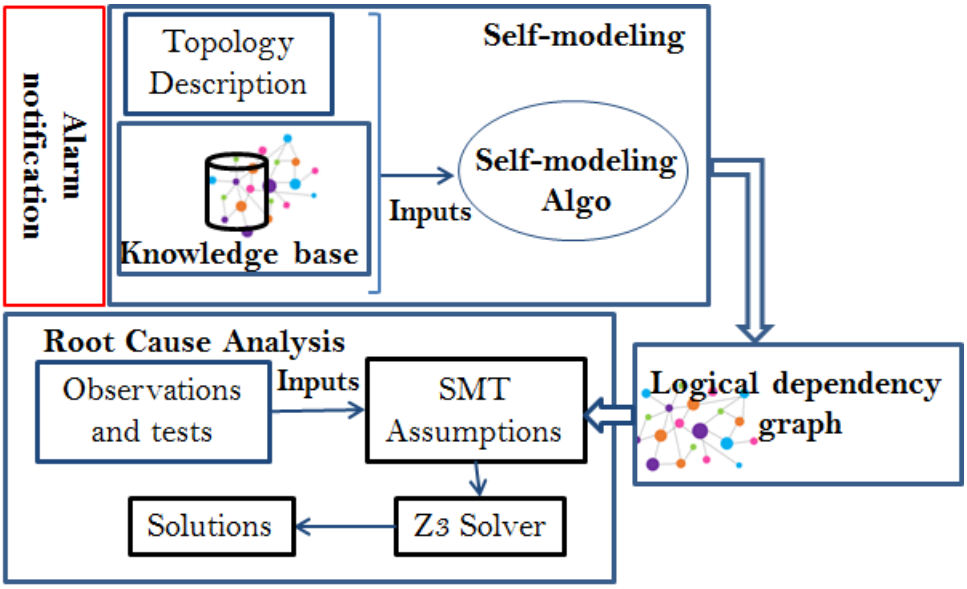


Figure 6.6 – Self-modeling and RCA Framework

vIMS test architecture: We apply the proposed fault scenarios in the architecture depicted in Figure 6.7. This architecture contains three sites. Site-3 hosts common functions between site 1 and site-2 i.e. Homestead, Homer and Cassandra. Site-1 and site-2 hosts the proxy Bono and the router Sprout. A client can register from the Bono-1 of site-1 or Bono-2 of site-2. This enables two Register services. one through site-1 and the second through site-2. The self-modeling algorithm takes as inputs the YAML file that describes this topology and generates the global dependency graph G.

6.6.1 Diagnosis of faults with identical symptoms

In this Section, we depict the results obtained from the application of the active diagnosis process on the different fault scenarios. The three different fault scenarios was injected on the Bono1 Docker (i.e. disconnect Bono1, stop Docker Bono1 and stop process Bono1). These faults when injected to the *Clearwater* deployment provided similar alarms (or symptoms) described in the following:

- Sprout1 could not reach Bono1: $obs(Cbono1sprout1) = False$.
- The Register service through Bono1 returns SIP code error 408 (i.e. request timeout): $obs(Register11) = False$.
- The Register service through Bono2 is correct: $obs(Register21) = True$.

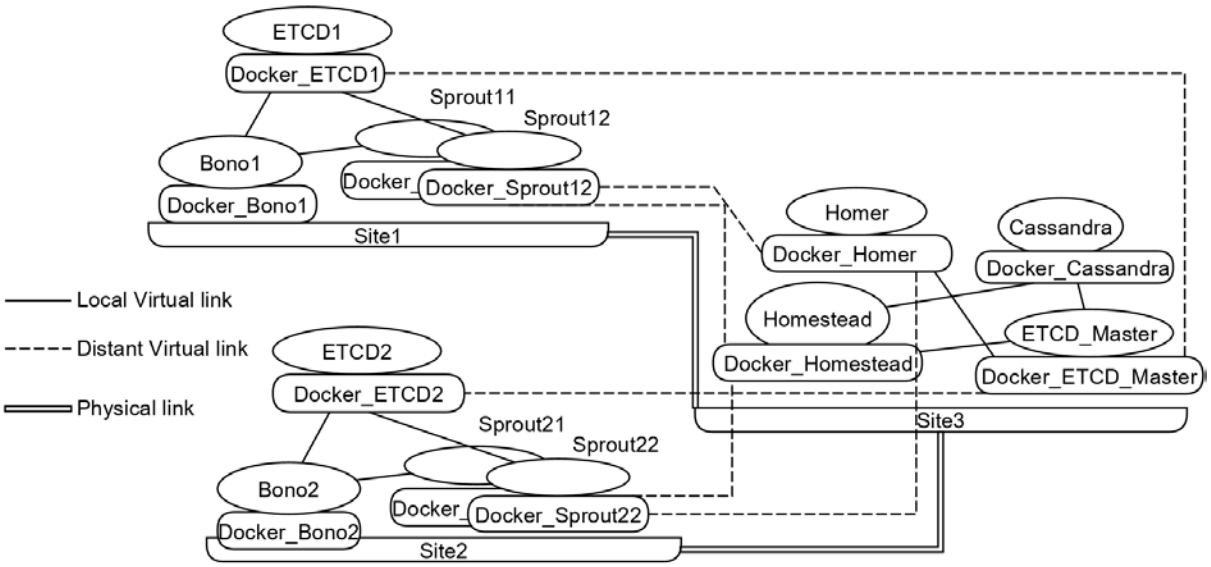


Figure 6.7 – Experimental vIMS architecture

These alarms are considered as initial observations $O_0 = \{Cbono1sprout1, Register11, Register21\}$. Figure 6.8 illustrates the initial observations nodes and their predecessors in the first sub-graph generated from $Extractor(G, obs, O)$. The active diagnosis process takes as inputs the global dependency graph G describing the topology of Figure 6.7 and the initial observations and generates the sub-graph depicted in Figure 6.8. Since the three scenarios have same symptoms, the first solution provided by the $Solver(\mathbf{A})$ is similar between the fault scenarios. With $\mathbf{A}$ representing the assumptions translated from the generated sub-graph in Figure 6.8.

Scenarios I, II and III provides the results depicted in Figures 6.9, 6.10 and 6.11, respectively. Note that these Figures are snapshot of the resulted dependency graph modeled in our python code with the NetworkX library. While labeling the templates nodes, we tried to match the theoretical definition in Chapter 5. However, some nodes labels might be resumed in our implementation for better readability of graph nodes labels. We explicit the defined templates nodes' labels in the Appendix 7.2.4. The results of the three scenarios are depicted in the following:

Scenario I: disconnect Bono1, $DcBonoC$:

---

1. Initial Observations: (Cf. Figure 6.8)

2. First proposed root cause: $Appbono1$

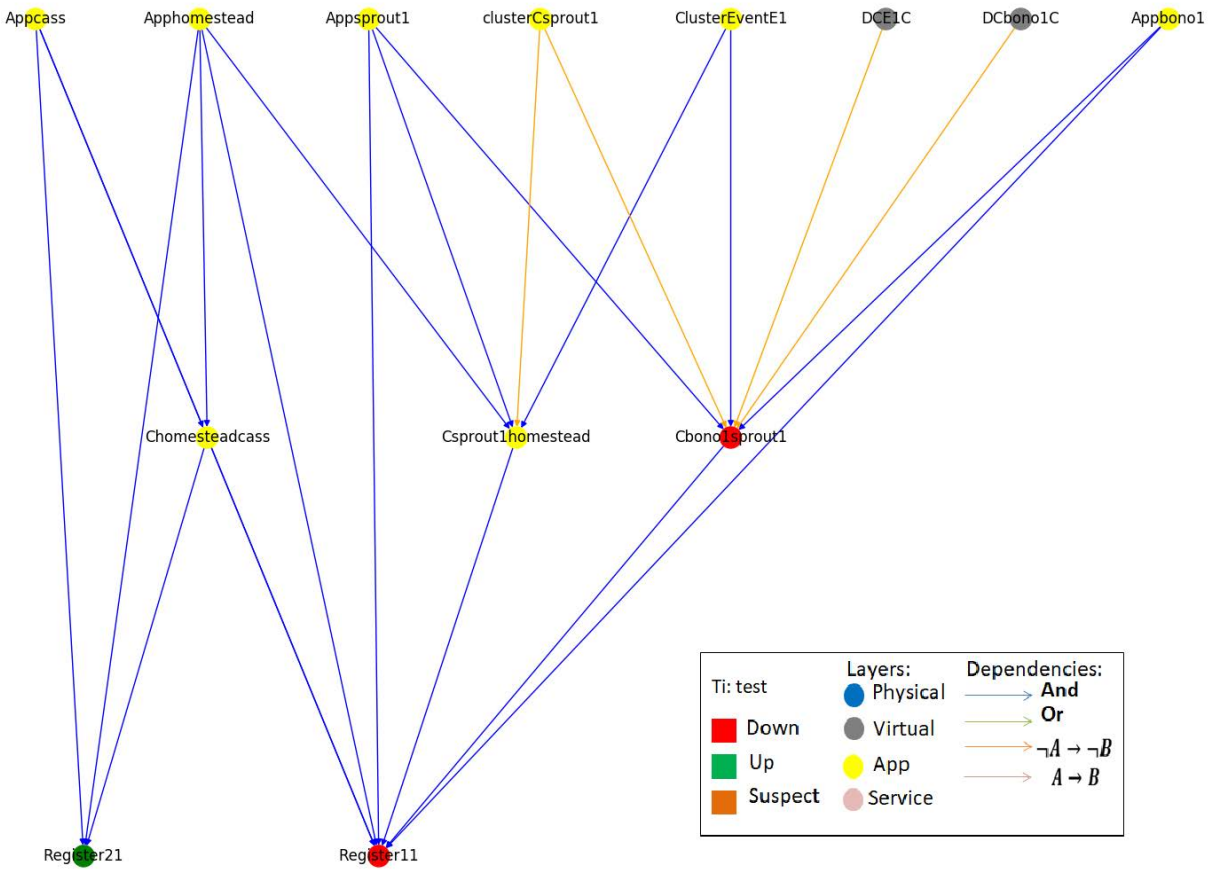


Figure 6.8 – . The initial observations sub-graph for Scenarios I, II and III.

3. Next proposed root cause(s) and Tests results:

   - Test of the Bono1 application ($val(Appbono1) = Unknown$.
   - Test of the ETCD1 Docker connectivity ($val(DCE1C) = True$.
   - Test of the Bono1 connectivity ($val(DCbono1C) = False$.

4. Results:

   Bono1 Docker connectivity ($DCbono1C$) is a possible root cause. Continue?: No, (Cf. Figure 6.9).

Scenario II: stop Docker Bono1, $DcBono1S$:

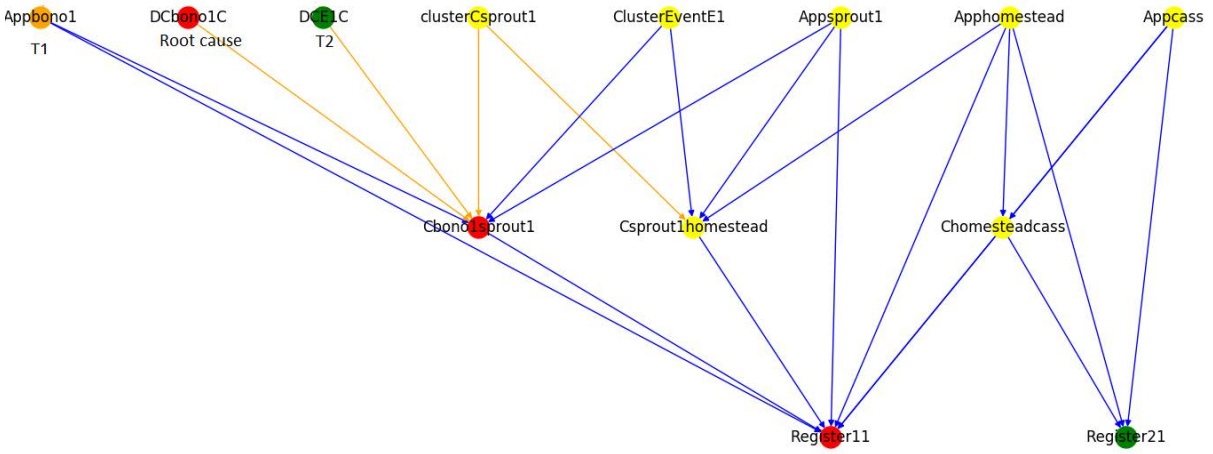


Figure 6.9 – . Result sub-graph of fault scenario I

1. Initial Observations: (Cf. Figure 6.8)
2. First proposed root cause: $Appbono1$
3. Next proposed root cause(s) and Tests results:
   - Test of the Bono1 application ($val(Appbono1) = Unknown$).
   - Test of the ETCD1 Docker connectivity ($val(DCE1C) = True$).
   - Test of the Bono1 Docker connectivity ($val(DCbono1C) = False$).
4. Results: Bono1 Docker connectivity ($DCbono1C$) is a possible root cause. Continue?: Yes.
5. Next solutions and Tests results:
   - Test of the network bridge status ($val(NB1) = True$).
   - Test of the Bono1 Docker status ($val(DCbono1S) = False$).
   - Results: Bono1 Docker status ($DCbono1S$) is a possible root cause. Continue?: No (Cf. Figure 6.10).

Scenario III: stop Bono1 process, $Pbono1$:

---

1. Initial Observations: (Cf. Figure 6.8)
2. First proposed root cause: $AppBono1$

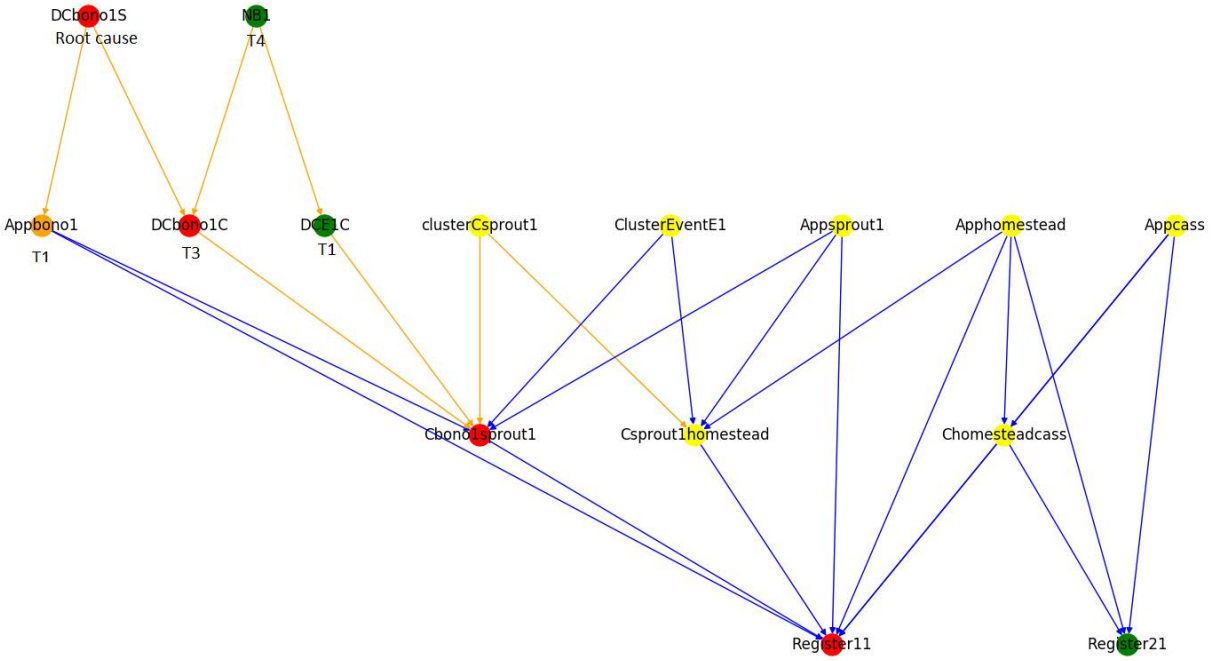


Figure 6.10 – . Result sub-graph of fault scenario II

3. Next proposed root cause(s) and Tests results:
   - Test of the Bono1 application ($val(Appbono1) = False$).
   - Test of the Bono1 ETCD connectivity process ($val(Pbono1EC) = True$).
   - Test of the Bono1 ETCD connectivity process ($val(Pbono1) = False$).
4. Results: Process Bono1 ($Pbono1$) is the root cause, please check the auto-recovery node (i.e. Monit). (Cf. Figure 6.11)

6.6.2 Summary

As illustrated in the proposed fault scenarios, the diagnosis process starts with an initial sub-graph of the first observations (Cf. Figure 6.8) and collects observations through tests. We notice that the three fault scenarios have the same initial observations. In each scenario, the diagnosis process is oriented differently with the results of tests. Our diagnosis process is able to detect the root cause and explain the fault even with few initial observations and is able to separate faults with identical symptoms.

Moreover, the diagnosis process detects the faults that occur at a finer granularity level such as the case of the fault scenario III, where the fault was within the process of the Bono1

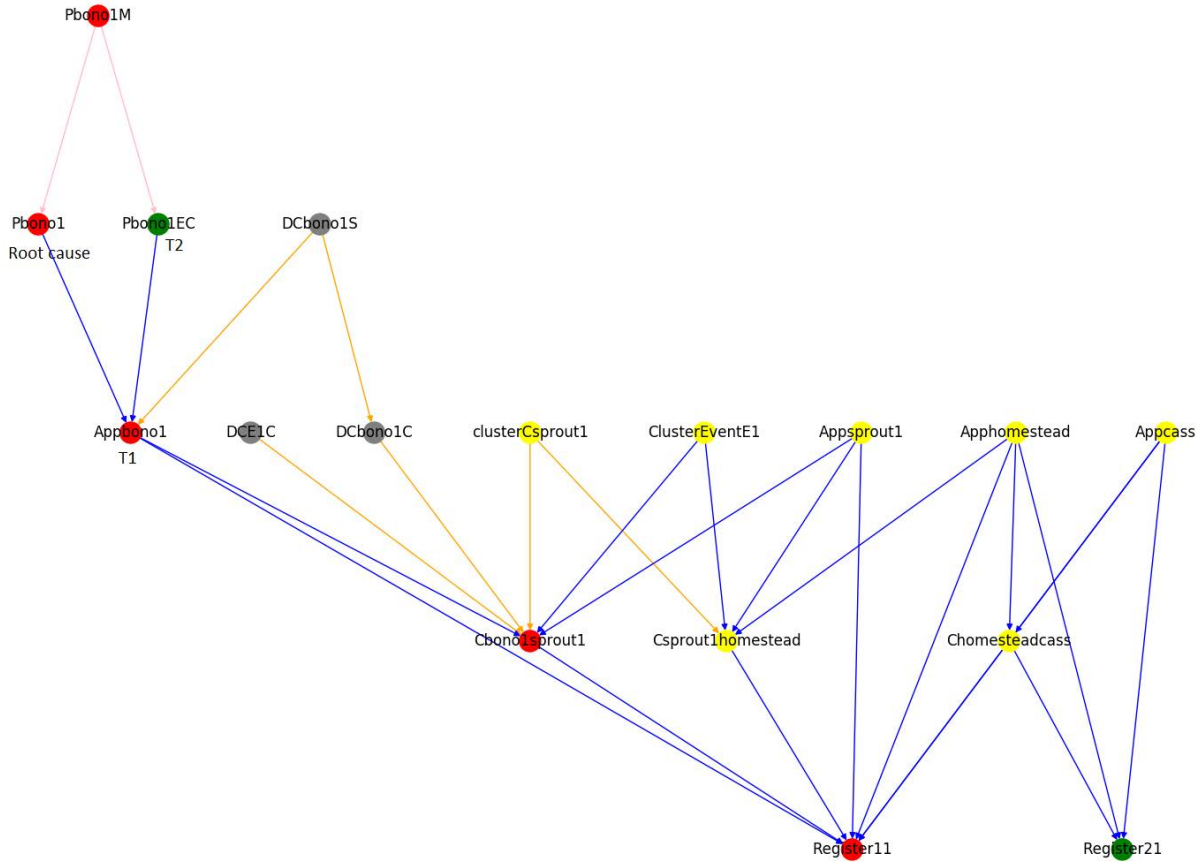


Figure 6.11 – . Result sub-graph of fault scenario III

application. This enables providing effective self-healing actions. For instance, in the case of the fault scenario III, the self-healing action consists in restarting the Bono1 process and the auto-recovery nodes instead of restarting the same Docker or starting a new Docker, which might be time consuming and might either not solve the problem.

6.6.3 Diagnosis of multiple faults

To evaluate the performance of the active diagnosis process in the case of multiple faults, we propose a fault scenario with two faults (f_1 and f_2). f_1 is a fault in the virtual connection of the Bono1 Docker in site1 (disconnect Docker Bono1) and the f_2 is a fault in the virtual connection of the Homestead Docker in site3 (disconnect Docker Homestead). The faults are injected in the same deployed three sites architecture of Figure 6.7. The first alarms considered as initial observations are depicted as follows:

- Sprout1 could not reach Bono1: $obs(Cbono1sprout1) = False$.

- Sprout1 could not reach Homestead: $obs(Csprout2homestead) = False$.
- The Register service through Bono1 returns SIP code error 408 (i.e. request timeout): $obs(Register11) = False$.

The first sub-graph generated by the $Extractor$ and the initial observations is illustrated in Figure 6.12.

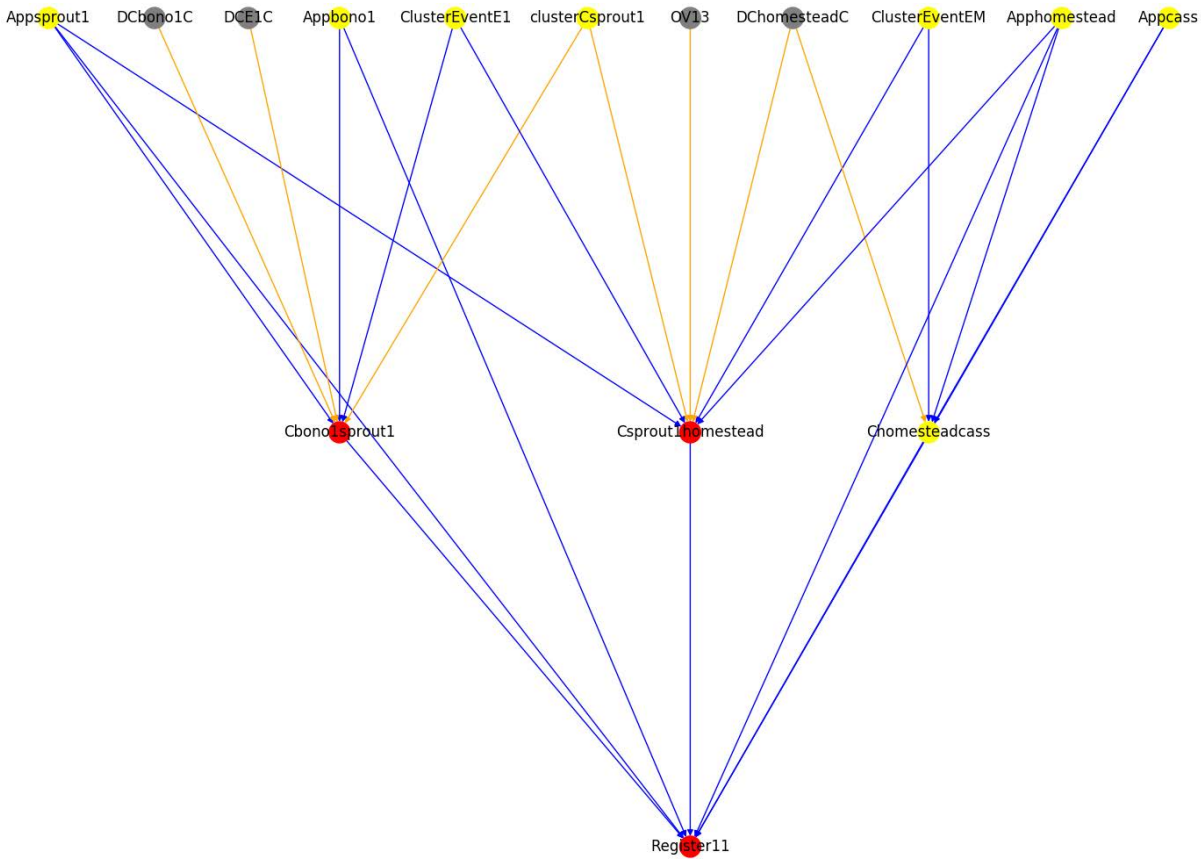


Figure 6.12 – . Initial observations generated sub-graph of multiple fault scenario in Bono1 and Homestead virtual connections.

The active diagnosis process may suggest different tests each time we launch it. In fact, it depends on the proposed solutions by the Z3 solver. However, in the end, the diagnosis process is able to pinpoint the correct root causes. We will illustrate in the following two different executions that differ on the proposed tests.

Execution I:

---

1. Initial Observations: (Cf. Figure 6.12)
2. Tests and results:
   - Node $ClusterCsprout1$ suspect.
   - Node $ClusterEventE1$ suspect.
   - Test of the Bono1 application ($val(Appbono1) = True$) and Node $OV13$ suspect.
   - Test of the Docker ETCD1 virtual connectivity ($DCE1C = True$) and test of the logical connectivity between Homestead and Cassandra ($Chomesteadcass = False$) and $ClusterEventEM$ suspect.
   - Test of the Bono1 virtual connectivity ($DCbono1C = False$) and test of the Homestead virtual connectivity ($DChomesteadC = False$).
3. Results: $DCbono1C$ and $DChomesteadC$ possible root causes continue? No, (Cf. Figure 6.13).

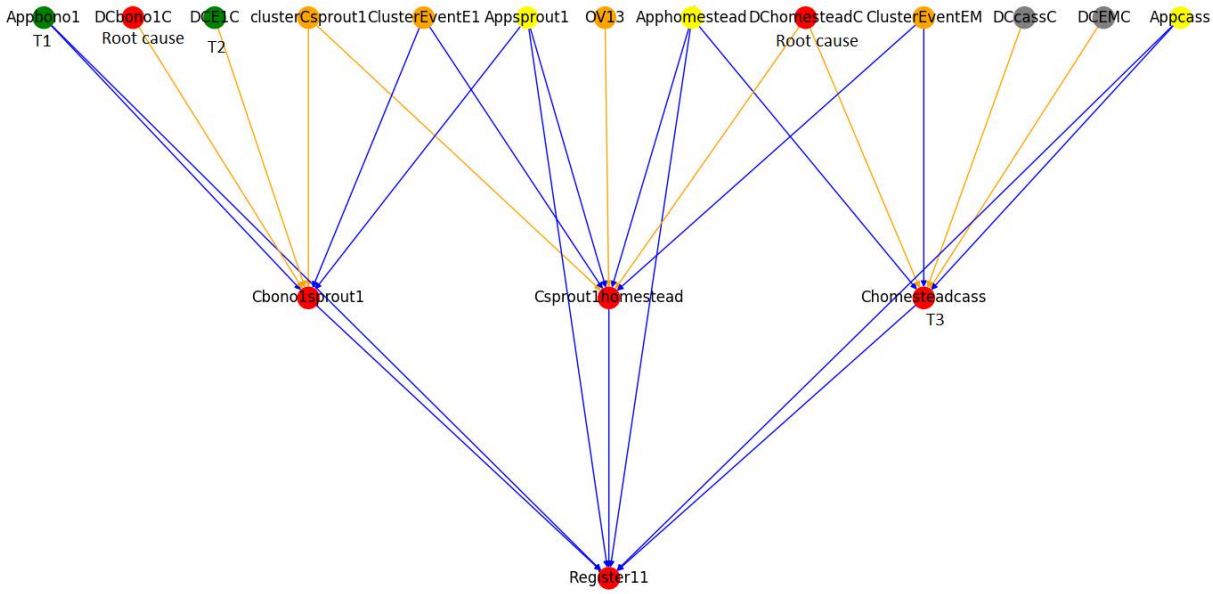


Figure 6.13 – . Results of execution I.

Execution II:

1. Initial Observations: (Cf. Figure 6.12)
2. Tests and results:

- Node $ClusterCsprout1$ suspect.
- Node $ClusterEventE1$ suspect.
- Test of the sprout1 apllication status "unknown": $val(Appsprout1) = True$.
- Test of the Bono1 application ($val(Appbono1) = True$) and Node $OV13$ suspect.
- Test of Homestead application ($Apphomestead == True$), test of the logical connectivity between Homestead and Cassandra ($Chomesteadcass = False$) and test of the Bono1 virtual connectivity ($DCbono1C = False$).

3. Results: $DCbono1C$ is a possible root causes continue? Yes, (Cf. Figure 6.14).
4. Next tests:
   - Test of the Homestead virtual connectivity ($DChomesteadC = False$) and test of the Docker Bono1 status ($DCbono1S = False$).
5. Results: $DChomesteadC$ is a possible root causes continue? No, (Cf. Figure 6.15).

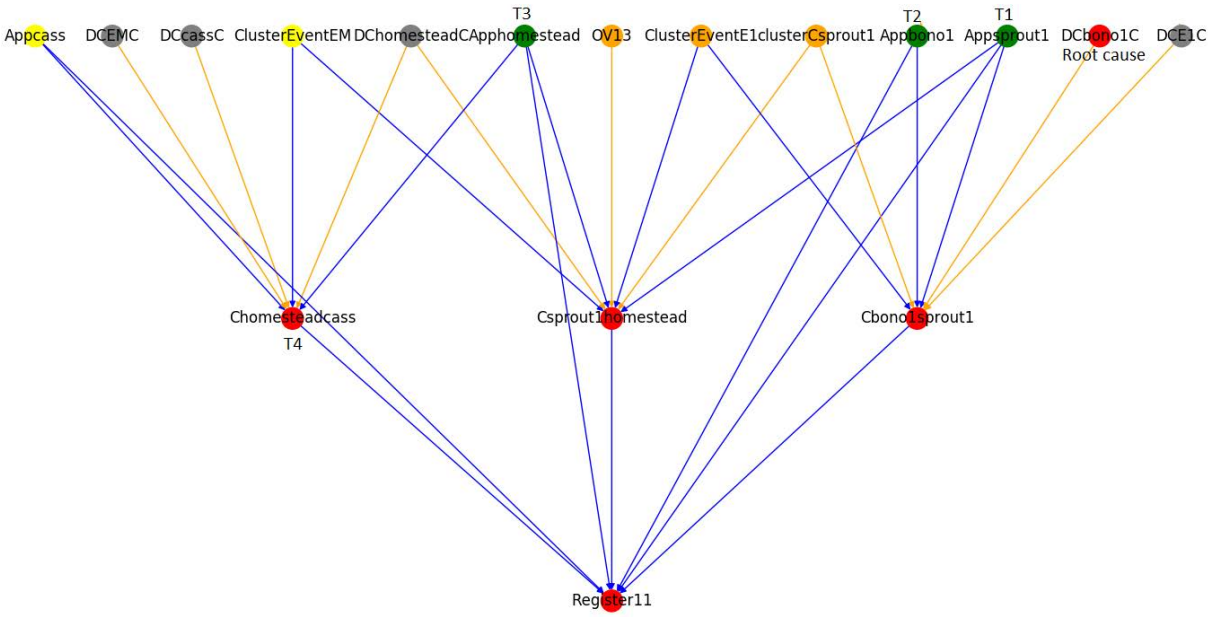


Figure 6.14 – . Results of execution II, detecting the first root cause $DCbono1C$.

6.6.4 Summary

When diagnosing multiple faults such as the example discussed above and illustrated in Figure 6.12, we notice that the active diagnosis process tries each time to satisfy both faults by pinpointing parents of $False$ nodes in the observations that explain the symptoms. For instance,

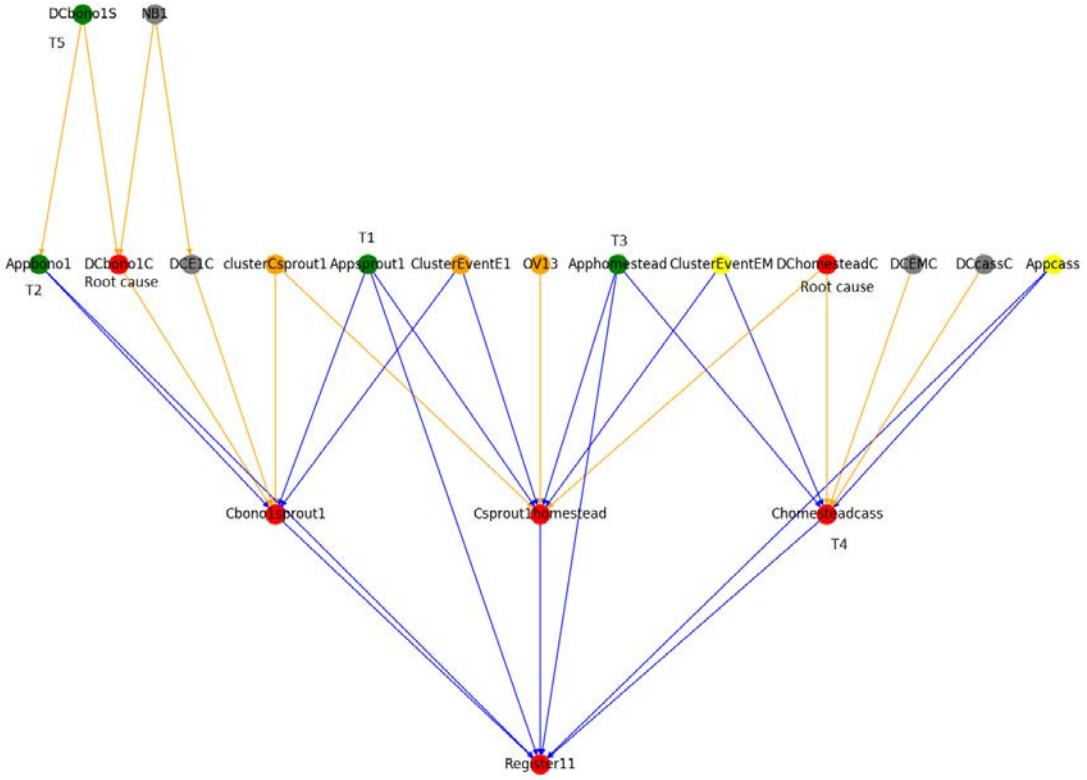


Figure 6.15 – . Results of execution II, detecting the second root cause $DChomesteadC$.

in the first solution proposed by the $Solver$, node $ClusterCsprout1$ is proposed as a root cause (i.e. $False$). This node is a common predecessor between all the $False$ observed nodes $\{Cbono1sprout1, Csprout1homestead, Register11\}$. We presented two distinct executions of the $Solver$. In the first execution, the $Solver$ proposed in the end both of the root causes, so the diagnosis process was able to pinpoint both faults at the same time. In the second execution, the last proposition was $DCbono1C$, $Apphomestead$ and $Chomesteadcass$ that explains all the symptoms. $DCbono1C$ explains the $Cbono1sprout1$ and $Register11$, $Apphomestead$ explains the $Csprout1homestead$ and $Register11$, and $Chomesteadcass$ explains the $Register11$. However, only $DCbono1C$ was a real root cause proved by a test. The result was proposed to the administrator that decided to continue the diagnosis with the same values of known nodes, since healing $DCbono1C$ didn't resolve the failure. The diagnosis provides then the $DChomesteadC$ and $DCbono1S$ as possible root cause(s). The $DChomesteadC$ is then detected as a second root cause. Note that the value of $DCbono1S$ was provided because the administrator left the $DCbono1C = False$, so this node was pinpointed to explain the fault in $DCbono1C$.

6.7 A qualitative comparison with related works

In this Section 6.7, we present a qualitative comparison of the proposed self-modeling and active diagnosis algorithm (*SAKURA* framework) with existent efforts and techniques applied to virtual networks.

6.7.1 A qualitative comparison with classical telecommunications monitoring routines

our active diagnosis compared to classical routines of monitoring in telecommunication networks such as syslog and performance monitoring, provides automation, fault explanation and accuracy. In fact, most of the performance and syslog monitoring techniques (e.g. Nagios) rises alarms to the administrator if a warning or a critical syslog message is received or a degradation of the network performance is noticed. These, alarms provide few indications about the fault and enable the administrator to guess the fault without any automation or by applying rules. These monitoring tools might detect some faults such as a docker down. However in the case of faults with a smaller granularity, multiple faults or faults with similar symptoms, where more explanations are needed these techniques are limited. Our diagnosis process goes beyond providing only alarms to faults. It provides explanation to the administrator with an explicit graph.

6.7.2 A qualitative comparison with black-box approaches

As presented in Chapter 3, black box approaches are provide less explanations about their solutions, and in particular about the propagation of faults in case of multiple faults.

	Approach	Explanation of solutions	Dependencies	Layers	Granularity	Adaptability to new use cases
Gonzalez et al. [46]	Random Forest	No	Learn from data known/ unknown dependencies (e.g. topology relations)	Physical and Virtual layers	VM is the smallest granurality	No (need to learn from new data)
Sauvanaud et al. [126]	Random Forest	No	Learn from data known/unknwon dependencies	Physical and Virtual layers	VM is the smallest granurality	No (need to learn from new data)
SAKURA	Dependency Graph	Yes	Learn only unknown dependencies	From physical to service layer	Peocesses are the smallest granularity	Yes (extention of the model)

Table 6.1 – A qualitative comparison of *SAKURA* with black-box approaches.

Moreover, most of the learned knowledge is already acquired knowledge as in the case of [46, 126] (Table 6.1), where most of the learned dependencies could be extracted from the network topology. Furthermore, our model captures smaller granularity (such as application processes) which enables to provide targeted self-healing actions. In addition, the defined

model can be extended to other virtual networks use cases by keeping the common dependencies and learning the unknown ones. While in black-box approaches such as in efforts [46] and [126], the diagnosis of a novel use case requires learning from new data all the dependencies of this new use-case.

6.7.3 A qualitative comparison with model-based approaches

In addition to the localization and explanation of faults, our diagnosis framework *SAKURA* has a number of advantages compared to the existent model-based efforts [106, 146], applied to virtual networks (Table 6.2). The defined model includes the application and service layers in addition to the layers addressed in [106, 146]. Moreover, the proposed model includes a number of dependencies not addressed by other efforts such as the auto-recovery and elasticity mechanisms. In addition to these dependencies, the model can be extended and validated through fault injection learning.

	Dynamic topology	Layers	Auto-recovery /elasticity mechanisms	Model learning /validation	Additional observations (tests) and interactive diagnosis	Use case
Sánchez et al. [134]	yes	virtual layer	no	no	no	SDN
Vitrage [146]	yes	virtual layer	elasticity	no	no	Open Stack VMs
SAKURA	yes	service layer	yes	yes	yes	SFC (vIMS) possible extension to other use cases

Table 6.2 – A qualitative comparison of *SAKURA* with model-based approaches

6.8 Conclusion

We proposed an active diagnosis process to validate our self-modeling approach. The diagnosis process receives as inputs the model defined by the self-modeling algorithm and the observations. The observations are nodes status collected from alarms and logs. The diagnosis process reasons on small parts of the global dependency graph to reduce the number of assumptions introduced to the $Solver$, and thus the number of provided explanations. The defined diagnosis process proposes a number of tests to be performed on the real deployment in order to get new observations and progressively pinpoint the true fault propagation pattern. The result is a dependency sub-graph that reveals the innocent, faulty and suspected nodes. The advantage of presenting results as a dependency graphs is first to explain the results of inferences to the network administrator, and mostly to enable an interactive reasoning incorporating human knowledge: the operator is then able to correct or add new observations about

known, unknown or suspected nodes to get more elaborate or alternative explanations. We finally showcased the efficiency of the diagnosis process on real vIMS fault scenarios. We proved that our diagnosis process can handle single faults scenarios with identical symptoms, as well as multiple faults scenarios. It is also able to detect faults with finer granularity in the network such as application processes.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Results obtained during the thesis

Virtualization of networks is a considerable driving force behind the design process of 5G architectures for both computing and networking. Specifically, virtualization enabled the flexibility of telecommunication network services by facilitating the dynamic VNFs creation, adaptation and termination. Moreover, virtualization introduced new features to the telco world such as the dynamic network topology, the distribution of VNFs through distinct sites and the coexistence between physical and virtual entities. In such a context, management of virtual networks is a core feature to be addressed at an early stage to enable applications and services to take full benefits of the virtualization. The fault management process includes three main tasks: detection, localization (RCA or diagnosis), and healing. A taxonomy of approaches were applied in each fault management step. However, the introduction of virtualization brought new issues that limited the existent state of the art fault management approaches and techniques.

To study the classical fault management techniques and their limitation when applied to virtual networks, we proposed a comprehensive survey in Chapter 3, that reviews the canonical fault management steps and efforts applied to classical telecommunication networks. We highlighted significant issues to be considered in the fault management of virtual networks and described the recent research achievements to face these limitations.

In the taxonomy of approaches discussed in Chapter 3, we focused on white-box or model-based techniques that are able to solve novel problems and provide explanations for the diagnosis decisions and conclusions through an explicit representation of the network dependencies. This kind of techniques are the key to an autonomous management of network malfunctions. They are adaptable to network components, over accurate and range reasoning methods and can go as far as suggesting the best self-healing actions.

However, to realize this ambitious objective, we had to face the challenges of virtual networks: the multi-layered architecture and complexity of virtual networks and the dynamic network topology. Our motivation to define this model was the existence of two types of knowledge in the network: the learned and the acquired knowledge. The acquired knowledge is the available knowledge collected from the network description files, expert knowledge, logs and network data. This knowledge is applied to learn the building rules of the model, define the current

topology or define the network element status to be considered as observations for the diagnosis process. On the other hand, the learned knowledge is provided through targeted faults injected to the network. To collect these data such as logs and network topology description, we proposed *LUMEN*, the global fault management framework in Chapter 4. The aim of this framework is to prepare the data for the self-modeling and diagnosis process. The proposed framework solved the lack of network visibility by centralizing the logs. Moreover, a number of open source tools were applied to generate traffic, inject faults to collect, filter and store logs.

In Chapter 5, we proposed our self-modeling approach that solves the burden of model construction and boils it down to the design of generic templates of the model, exploiting two types of knowledge: the acquired and learned knowledge. To face the dynamic topology issue, we proposed to model the real topology through a YAML file. The defined model includes the virtual networks granularity and the multi-layers aspect. The proposed self-modeling approach was applied to the vIMS use-case. One important aspect of the proposed self-modeling approach is that the defined templates can be validated and extended through fault injections in a real platform. Basing the modeling on generic templates allows one to address a wide range of use-cases, as templates can be combined in numerous ways. The performance of out self-modeling algorithm was then evaluated through two tests. The first performance test studies the scalability of the generated dependency graph compared to the number of sites and VNFs in the network topology. The second performance test evaluates the self-modeling algorithm execution time. The results showed the importance of remodeling only the network topology changes to avoid the considerable time spent on reading the YAML file.

To validate the efficiency of the self-modeling approach, the best way is to prove that the model obtained with this approach is able to provide correct and accurate explanations about faults through the diagnosis process. Therefore, we proposed in Chapter 6, an active diagnosis process that reasons on portions of the global dependency graph. This graph is derived from the self-modeling algorithm of the monitored network topology. The active diagnosis process applies a $Translator$ to translate the graph into logical dependency relations between network entities. These relations are then "solved" through a logical $Solver$ to discover the value of unobserved variables. The active diagnosis process is combined with online test operations, and provides explanations as sub-graph of relevant network resources with the identification of guilty, faulty, innocent and suspect nodes, that make explicit the fault propagation scenario. The network administrator has the ability to change or correct the values of some nodes at any time to get new results and confirm/reject his own interpretations. To demonstrate the relevance of the proposed active diagnosis approach, we applied it to a real world Docker-based *Clearwater* vIMS architecture subject to real fault injections.

7.2 Perspectives

This thesis opens up several research directions worth exploring in the future. We explain each direction in the following.

7.2.1 Automation of other steps composing the self-modeling and diagnosis procedures

We propose self-modeling and diagnosis procedures that respond to the challenges of virtual networks. These procedures enable an autonomous management of network malfunctions. Most of the steps composing the self-modeling and diagnosis procedures were automated. However, other steps are worth to be automated or further developed:

- **Autonomic creation of the YAML topology file**: we proposed in Chapter 5 Section 5.5, a self-modeling algorithm that takes as inputs an YAML topology file. We defined this YAML file to unify the description of the deployed network topology. It contains the information necessary to instantiate the defined templates. This file captures the current network topology to enable instantiating an up to date model. However, in our work, we manually write this YAML, we didn't develop the procedure for generating these models from different deployments. This procedure is easily doable by interrogating the deployed network orchestrators such as OpenStack or Docker daemons in our case.

- **Automation of tests in the active diagnosis process**: the active diagnosis process presented in Chapter 6 identifies a number of tests to perform in order to progressively pinpoint the cause of a malfunction. To improve the efficiency of this procedure, an obvious direction is to automate the necessary tests, to releave the (human) network administrator. To do so, the active diagnosis algorithm should be provided with the test command lines and the necessary information about the network components in order to perform these tests. Connectivity tests, for example, are easy to automate as soon as the algorithm should know the IP address of the node to ping.

- In addition to automation, a number of other improvements to the diagnosis process remain possible. For example, the use of a Solver minimizing the number of nodes assigned to "False" would simplify the recursive tests. One can also imagine the selection of the most informative test to perform, or the choice of their ordering, in order to progress more quickly to the true root cause, or also considering a specific granularity of the model to run the diagnosis process and refine this granularity if necessary, as the diagnosis progresses.

- **Self-learning algorithm:** we proposed in Chapter 5 Section 5.6 a procedure for the validation and refinement of our templates using fault injections. This procedure was applied using fault injection scripts. However the comparison between real world observations and predictions provided by our templates, and the derivation of necessary changes in our templates to match real life behaviors, were all done by a human expert. An ambitious perspective would be to automate this task through self-learning techniques, with the objective to identify component dependencies, propose templates, design validation procedures and correct/extend the templates. This algorithm would apply fault injection scenarios in both a real deployment and its corresponding model. The self-learning algorithm would then compare the results of the fault propagation of both sides. The result of this algorithm in case a mismatch is detected could be: in a template "y" there might be a node "x" linked with nodes of template "y" with a dependency type d_v. After that, it is up to the human expert to identify the correspondence of the added node X in the real network.

7.2.2 Probabilistic dependency graph

We proposed in this thesis a self-modeling approach and validated it with the active diagnosis process. The resulting model is a logical dependency graph with Boolean variables and logical dependencies. To relax this model a direct extension is to keep the Boolean variables and relax the logical (hard) dependencies into probabilistic (soft) dependencies. So instead of hard and certain propagation patterns, we allow for randomness. For example, a direct way to do so is to replace the OR by a noisy OR, the AND by a noisy AND, etc. This enables the introduction of likelihoods for spontaneous faults, some being more frequent. This flexibility also allows one to improve the robustness of models to modeling errors, as more valuations become possible compared to logical dependencies, but may simply become less likely. Naturally, this model extension requires to replace logical solvers by Bayesian inference algorithms. But, again, the resource is abundant on this side. The main difficulty of model-based approaches lies in the derivation of an accurate model.

Pushing further the modeling capabilities, while we have focused on hard failures, resulting in Boolean variables in our models, one can imagine addressing softer failures. For example system degradations like a high CPU load, a low bandwidth, a high latency, or at service levels higher disconnection rates, long connection procedures, etc. It is likely that part of the structural/logical dependencies we have identified can carry over to such softer degradations, which could make our approach extensible to capturing a wider range of malfunctions.

7.2.3 Dynamic diagnosis

In the present work, we have assumed that the diagnosis was performed on a static model, although the model may change from one call of the diagnoser to the next. In other words, we reasoned on a snapshot, or a picture, of the network, assuming a fault is fully deployed in all its consequences. A first extension would be to model dynamic behaviours to network element, in order to observe and diagnose a malfunction as it is propagating (providing time scales are relevant). That would mean performing diagnosis on a movie of a dynamic systems. This is a classical topic, that has been already addressed for Bayesian inference, including for network diagnosis (see Dynamic Bayesian Networks for example). However, so far, this was never mixed with the fact that the model changes! As we are fully equipped for capturing dynamic changes in a network topology, one can wonder whether it would be meaningful to design diagnosis algorithms for systems that change structure over time. This would allow one, for example, to incorporate in the inference engines the self-healing mechanisms of our models, or the scaling up/down mechanisms, or VM migration phenomena, etc.

7.2.4 Application of the self-modeling approach to other use-cases

We proposed in Chapter 5 Section 5.3, a self-modeling approach that could be applied to a wide range of virtual network use-cases. This approach captures the most common features of virtual networks such as the multi-layered, the elasticity and auto-recovery aspects. Moreover, the learning of templates dependencies with fault injections could be applied to learn specific dependencies of each use-case. We proved the correctness of the defined templates by the application of this model to a real world use-case: Docker-based *Clearwater* vIMS VNF chain. This approach could be applied to other telecommunication virtual network use-cases. The definition of the templates could be extended by fault injection to fit the corresponding use-cases. The YAML file could also be adjusted to the use-case if necessary.

APPENDIX A

Counterfactual analysis

Many variants of causality have been proposed in the literature and used in different disciplines. It is questionable that one single definition of causality could fit all purposes. It may be useful to ask different questions, such as: "could event A have occurred in some cases if B had not occurred?" (1) [47]. Recent definitions of actual causality [54] and fault ascription [48] use counterfactual analysis in order to pinpoint the events or components responsible for the violation of a safety propriety, for instance, defined by VNF requirements.

Two definition of causality relationships namely *necessary* and *sufficient* causality have been formalised in [47] in order to identify the component responsible for a failure of the system, or, the occurrence of a given event. The input to the problem is:

- A set L of logs L_i;
- A sequence of events observed for each component C_i;
- A number of specifications S_i for each component C_i;
- A global property P such that $\wedge_{i\in[1,n]}S_i \implies P_i$;
- A set of assumptions BH_i on the behaviour of C_i;
- The set of logs L is assumed to be faulty and not consistent with the required property P.

Given the availability of the correct model for the component C_i, what would have been the course of events if C_i had behaved correctly?. To answer this question, [47] proposed to perform the following causality analysis and then check whether the hypothetical logs L'_i meet the property P:

$$\begin{array}{ll} Observed logs L_i \rightarrow & Potential behaviours Bh_i \\ & \downarrow \\ Hypothetical logs L'_i \rightarrow & Hypothetical behaviours Bh'_i \end{array}$$

With $Bh_i \in BH_i$ are the behaviours of the components that are consistent with the observed logs L_i, and the $Bh'_i \in BH_i$ are modifications of behaviours Bh_i in which a number of erroneous

behaviours are replaced by correct behaviors. The Bh'_i behaviours produce the hypothetical logs L'_i.

According to [47], for this type of causality analysis eight possible forms of causality can be noted. Among the proposed choices, the "$Necessary^{\forall,\forall}$" choice corresponds to the following definition:

> Considering the evidence provided by the set of logs L, Component C_i is a $Necessary^{\forall,\forall}$ cause for the failure of the system if for all potential behaviors Bh of the system consistent with L, all behaviors Bh' similar to Bh except for the behavior of C_i which is made correct, lead to correct execution logs.

This reasoning was applied on an example composed of a database system consisting of three components communicating by message passing with First In First Out (FIFO) buffers. The components are: a client C1, a database server C2, a journaling system C3. A client C1 that wants to modify data in the database C2, starts by locking the data and then modifies the data, the database sends a "Journ" to save the modification in the journal C3 and inform a client with the modification (i.e. ACK OK). The correct flow of messages represents three specifications (S_1, S_2, S_3) defined in Figure 1. Figure 2 represents a Log $\vec{tr} = (tr_1, tr_2, tr_3)$, where: "!" stands for the emission messages and "?" for received messages, a,b are events. "x" is an event that appears when a lock message is missing. In the log, tr_1 violates S_1 at event "a" and tr_3 violates S_3 at "b".

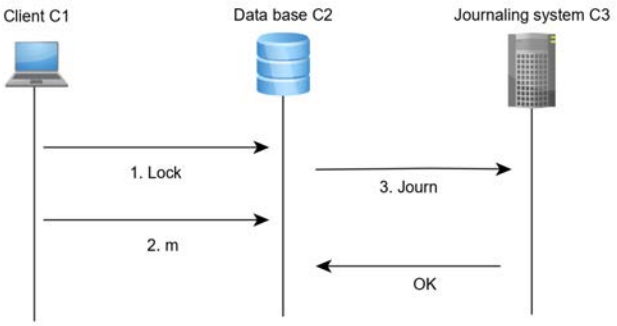


Figure 1 – The procedure of data modification in a data base.

In order to analyze which component(s) caused the violation of the property P that models the absence of a conflict event "x", using a counterfactual approach, we obtain: If C1 had worked correctly, it would have produced the trace $tr'_1 = lock!.m!$. This gives us the counterfactual scenario consisting of the traces $tr'_1 = (tr'_1, tr_2, tr_3)$. However, this scenario is not consistent as C1 now emits **lock**, which is not received by C2 in tr_2, and since in the FIFO

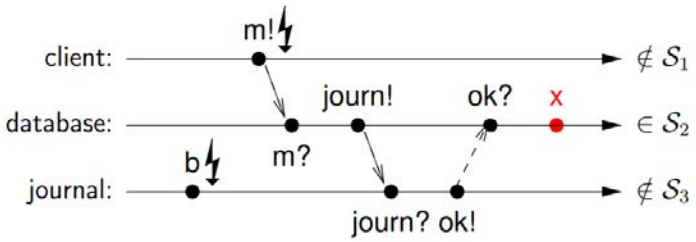


Figure 2 – Log $\vec{tr}$ and the corresponding specifications [47].

buffers there is no message losses. To correct the global traces a projection of the counterfactual traces is illustrated in Figure 3. The projection on the unaffected components by the failures. The set of counterfactuals is the set of system-level traces whose projections on the components extend the unaffected prefixes with correct behaviors. The blue prefixes in Figure 3-a and Figure 3-b represent the unaffected components of the failure of the client and Journal, respectively.

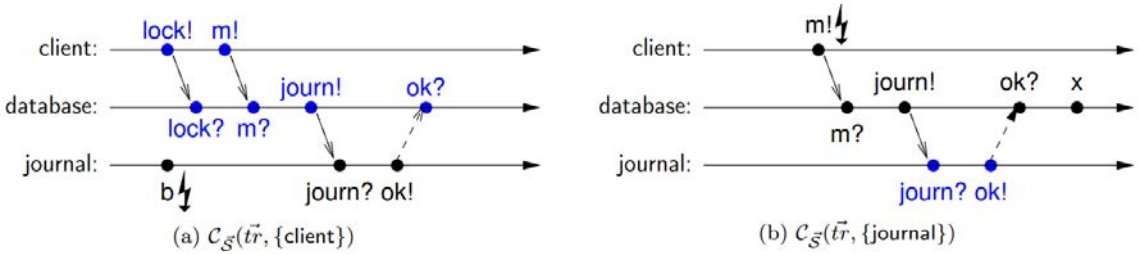


Figure 3 – The counterfactual scenarios to the failure of client (a) and journal (b) [47].

APPENDIX B

Docker *Clearwater* vIMS deployment features

The deployed *Clearwater* vIMS differs in many ways from the architecture presented in Chapter 2, Section 2.3.2. Figure 4 represents a *Weave Scope* screenshot of Docker *Clearwater* deployment. *Weave Scope* is an open sources project that enables to detect automatically the running Dockers and processes in the server where it is deployed [149]. In this Figure 4, the *Clearwater* functions are deployed in separated Dockers running in the same server. Each Docker holds an IP address and a *Clearwater* function name (e.g. Bono). The links illustrated in Figure 4 represent the communications between *Clearwater* Dockers.

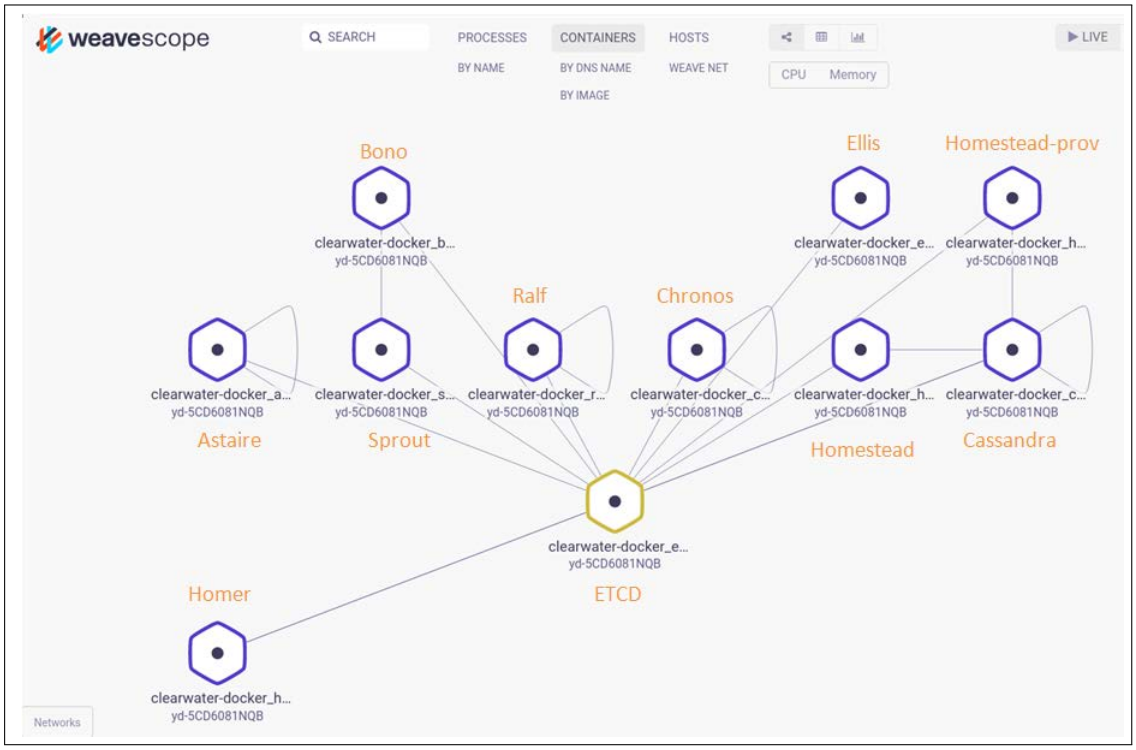


Figure 4 – A deployment of Docker*Clearwater* vIMS.

In Figure 4, we can notice that *Clearwater* has a number of features that we should consider when building the model. These features are deduced from acquired knowledge extracted from description files [138] and from the deployment.

- Each *Clearwater* application (e.g. Bono or Sprout) is deployed in a separate Virtual Docker host.
- All *Clearwater* components are horizontally scalable using simple, stateless load-balancing.
- *Clearwater* vIMS allows its Dockers to automatically connect to the correct Dockers and share configuration with each other. This feature uses Master ETCD as a decentralized data store and ETCD clients installed in each vIMS *Clearwater* node.
- Databases such as Cassandra are implemented in a separate Virtual host.
- *Clearwater* enables online elastic scaling of its functions. Functions could be replicated, without disrupting calls or losing data. The traffic is distributed equally between the scaled functions. In fact, the interfaces between the different vIMS Dockers use connection pooling with statistical recycling of connections to ensure load is spread evenly as *Clearwater* Dockers are scaled and removed from each layer.
- The existence of the homestead-prov Docker that provides a programmable interface to create new clients. It does the same work as the Ellis dashboard but it uses a programmable interface.
- Failure alarms and logs can be collected from the *Clearwater* components and from the SIP test application.

Traffic generation

A number of SIP traffic generators exists: client SIP services (e.g Jitsi [65]) and Open source test tools (e.g SIPp [130]). These solutions enables to generate SIP traffic through a bunch of SIP clients. The *Clearwater* dashboard Ellis allows to create and modify the identity of the SIP clients. Figure 5 illustrates an example of two SIP clients. Each SIP client has a SIP number (e.g. 6505550760) and a password that enables him to register to the *Clearwater* SIP base, in this case its: example.com.

Jitsi is an open source SIP application that enables clients to connect to the base using their SIP number and password and benefit from the vIMS multimedia services [65]. Figure 6, showcases a connected client through the SIP identifier 6505550760@example.com that tries to call the first client in the Ellis dashboard (Figure 5). Jitsi enables to generate traffic between clients. However, it is difficult to increase the number of connected clients since each client should be connected to its own application running on a distinct server with a distinct SIP identifier. Therefore, we opted for the *Clearwater-live-test* project that includes a number of

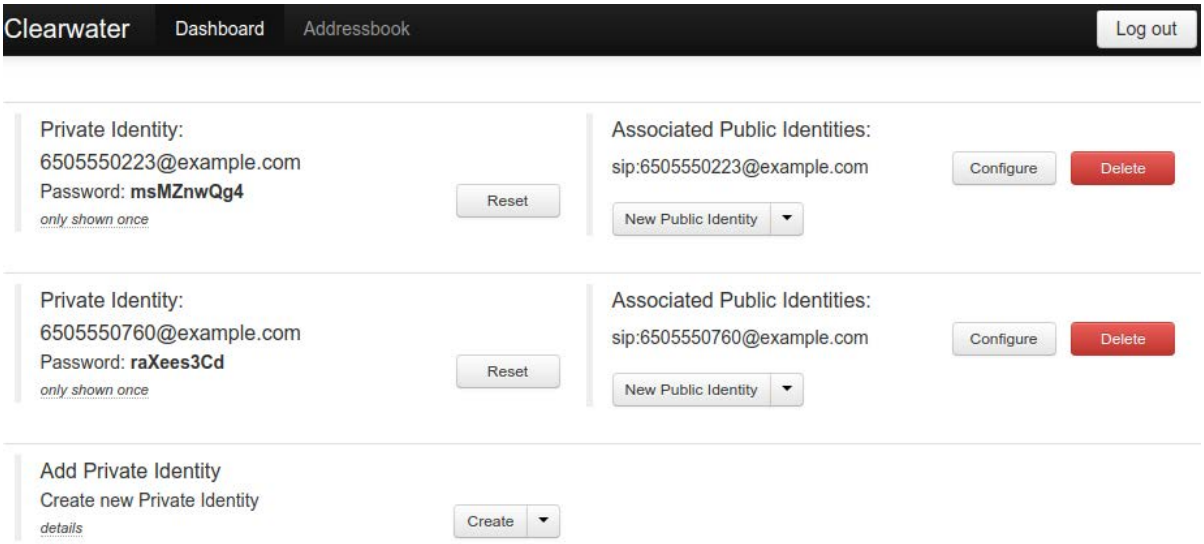


Figure 5 – The Ellis dashboard screenshot where two clients were created.

SIP traffic scenarios and reports the network health in each test [20]. The *Clearwater-live-test* represents a suite of live tests that can be run over a deployment to confirm if the high level network functions are working correctly. Traffic scenarios or tests in the framework are scripted using short Ruby programs that can be extended. These programs use the Quaff library for SIP calls, and the rest-client library to communicate with Ellis for provisioning. For instance, one example of the performed tests is to create two SIP clients, create a SIP call between the two clients, and report the received SIP response codes. Figure 7, illustrates three possible *Clearwater-live-test* tests described as follows:

- Basic call (TCP)- (number1,number2): generates a basic SIP call using TCP in the transport layer between the two clients with the associated SIP numbers: number1 and number2.

- Basic call- unknown number (TCP)- (number1, number2): test if the *Clearwater*, (i.e. the Homestead node that checks the numbers in the Cassandra data base), recognizes the case where a client with "number1" tries to connect to an unknown number "number2" that is not defined in any base.

- Basic Registration (TCP)-(number1): test a SIP authentication of a client with number1 to the *Clearwater* vIMS.



Figure 6 – A connected Jitsi client to the "example.com" base trying to call the client with the SIP number 6505550223.

```
** Invoke test (first_time)
** Execute test
Basic Call - Mainline (TCP) - (6505550501, 6505550009) Passed
Basic Call - Unknown number (TCP) - (6505550042, 6505550370) Passed
Basic Registration (TCP) - (6505550761) Passed
```

Figure 7 – Three tests generated from the *Clearwater-live-test* applied to the *Clearwater* deployment. The green "Passed" means that the test is successful.

Fault injection procedure

Figure 8 and Figure 9 illustrate an example of a fault injection in the virtual layer. In this example, we stopped the Sprout Docker with the command line in Figure 8, the result of this command is stopping the Docker Sprout that disappeared from the *Weave Scope* dashboard illustrated in Figure 9. Note that with this command line the Docker Sprout is not destroyed and can be restarted.

Logs collection and filtering

The logs generated by the *Clearwater* components are structured as the following:

Filtering the logs enabled us to keep only significant log lines. To do so, we drop the lines of logs that are only informative such as the first log in Table 1. We only keep the "warning"

```
sihem@yd-5CD6081NQB:~$ docker stop e2ca17f3c95e
e2ca17f3c95e
                                    Docker Sprout Id
```

Figure 8 – The command line for stopping the Sprout Docker.

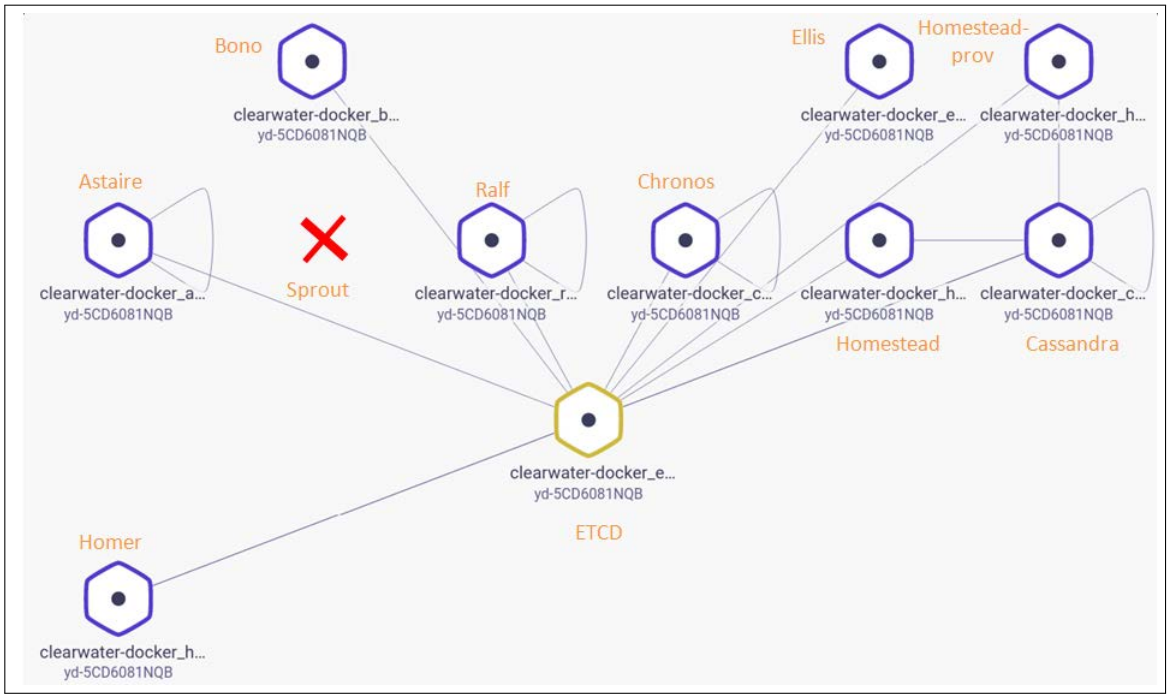


Figure 9 – The *Weave Scope* screenshot after stopping the Sprout Docker.

log lines. The "warning" log lines inform us about the type of alarms. For instance, the second log line in Table 1, represents a loss of connection between the component raising the alarm and the component with the IP address 172.18.0.4. This log line inform us about a connectivity failure. The generated alarms are then considered as initial observations to the diagnosis algorithm in the decision plane.

Timestamp	Type and identification	Descriptive message
11-01-2020 15:51:32.510	UTC Status pluginloader.cpp:150	Finished loading plug-ins
10-01-2020 02:48:10.062	UTC Warning pjsip: tcpc0x7fa8c405	Unable to connect to 172.18.0.4:5052

Table 1 – Two log lines generated by the *Clearwater* components. The first part of the log represents the time and date of the alarm . The second part is the type of alarm (i.e. "Status" and "Warning") with a specific identification defined in the alarming program of *Clearwater* such as "pluginloader.cpp:150" in the first log line. The last part of the log line is a message describing the alarm.

APPENDIX C

List of vIMS implemented Templates Nodes

Physical layer

S	aggregated node for sites
S_i	status of $site_i$
C_{ij}	physical connectivity between $site_i$ and $site_j$

Virtual layer

V_i	aggregated node for the virtual environment
DC_name	aggregated node for a Docker status and connectivity
DC_name_C	the Docker connectivity
DC_name_S	the Docker status
$ClusterC_name$	the virtual connectivity of the cluster of Dockers "name"
$ClusterS_name$	the status of the cluster of Dockers "name"
$ClusterEvent_(ETCD_name)$	the status of the cluster of ETCD and Memory
NB_i	the network bridge of $site_i$
OV_{ij}	aggregated node for overlay network
OV_S_{ij}	status of the overlay network
OV_C_{ij}	convergence of $ETCD_i$ of $site_i$ and $ETCD_j$ of $site_j$

Application layer

App_name	aggregated node for an application status
$C_name_1_name_2$	logical connection between App_name_1 and App_name_2
P_name	a process status
P_name_M	the auto-recovery Monit process
P_name_EC	the status of the ETCD client process

Service layer

$Register$	the register service status

ACRONYMS

AI Artificial Intelligence 38
API Application Programming Interface 14
ARF Access Relay Function 64
AS Application Server 26
AU Administrative Unit 68
BGCF Breakout Gateway Control Function 26
BSS Business Support System 19
BN Bayesian Network XXI
BP Belief Propagation XXI
CCF Common Cause Failure 64
CDF Charging Data Function 26
CDR Charging Data Record 26
CLF Connectivity Location Function 64
CP Connection Point 12
CPT Conditional Probability Table XXI
CPU Central Processing Unit 3
CSCF Call Session Control Function 25
CSP Constraint Satisfaction Problem 56
CTF Charging Trigger Function 26
DAG Directed Acyclic Graph 51
DHCP Dynamic Host Configuration Protocol 64
DNN Deep Neural Network XXI
DNS Domain Name System 10
DPI Deep Packet Inspection 12
EM Element Management 20
EPA Enhanced Platform Awareness 20

ETSI European Telecommunications Standards Institute XX
FCAPS Fault, Configuration, Accounting, Performance, and Security 2
FIFO First In First Out 162
FM Fault Management 33
GBN Generic Bayesian Network XXI
GS Group Specification XX
HMM Hidden Markov Model 36
HSS Home Subscriber Server 25
HTTP Hypertext Transfer Protocol 13
IaaS Infrastructure as a service 17
IBCF Interconnection Border Control Function 26
I-CSCF Interrogating-CSCF 25
IETF Internet Engineering Task Force 1
IMPU IMS Public User Identities XXI
IMS IP Multimedia Subsystem XVI
IP Internet Protocol 21
ISUP ISDN User Part 26
IUT-T Telecommunication Standardization Sector of the International Telecommunications Union 8
JSON JavaScript Object Notation 87
JTC Join-Tree Clustering 57
KPI Key Performance Indicator 36
LOS Loss Of Signal XXI
LSTM Long Short-Term Memory 48
LTE Long Term Evolution 1
MANO Management and Orchestration XVI
MEP Maintenance End Point 70
MGCF Media Gateway Control Function 26
MIP Maintenance Intermediate Point 70
ML Machine Learning XX

MMTel Multimedia Telephony Service . . . 25
MNO Mobile Network Operator . . . 1
MRF Multimedia Resource Function . . . 26
MRFC Multimedia Resource Function Controller . . . 26
MRFP Multimedia Resource Function Processor . . . 26
MS Multiplexage Section . . . 68
NACF Network Access Configuration Function . . . 64
NAT Network Address Translation . . . 10
NCT Network Connectivity Topology . . . XXII
NF Network Function . . . 10
NFP Network Forwarding Path . . . 12
NFV Network Functions Virtualization . . . XVI
NFVI Network Function Virtualization Infrastructure . . . 12
NFVO NFV Orchestration . . . 19
NFVI-PoP NFVI Point of Presence . . . 12
NFVRG-IRTF NFV Research Group Internet Research Task Force . . . 17
NGMN Next Generation Mobile Networks . . . 2
NIC Network Interface Controller . . . 12
NN Neural Network . . . XVII
NoSQL No Structured Query Language . . . 28
NS Network Service . . . 2
NVE Network Virtualization Ecosystem . . . 2
OASIS Organization for the Advancement of Structured Information Standards . . . 20
ONAP Open Network Automation Platform . . . 1
ONF Open Networking Foundation . . . 15
ONOS Open Network Operating System . . . 14
OPNFV Open Platform NFV . . . XXI
OS Operating System . . . 17
OSM opensource MANO . . . 20
OSS Operation Support System . . . 19

PaaS Platform as a service 17
P-CSCF Proxy-CSCF 25
PL Physical Link 12
PMR Private Mobile Networks 63
PNF Physical Network Function 11
PSTN Public Switched Telephone Network 26
RCA Root Cause Analysis 2
ReLU Rectified Linear Unit 47
Rf Reference Point 26
RNN Recurrent Neural Network 48
SaaS Software as a service XX
SAT Boolean Satisfiability Problem 59
SCTP Stream Control Transmission Protocol 36
SDH Synchronous Data Hierarchy 68
SDN Software Defined Networking XVI
S-CSCF Serving-CSCF 25
SF Service Function 1
SFC Service Function Chain 10
SFF Service Function Forwarding 80
SIP Session Initiation Protocol XXII
SLA Service Level Agreement 9
SLF Subscriber Locator Function 52
SMT Satisfiability Modulo Theories 60
SNMP Simple Network Management Protocol 2
SPI Synchronous Physical Interface 68
STM Synchronous Transport Module 68
TAS Telephony Application Server 26
TCP Transmission Control Protocol 36
TOSCA OASIS Topology and Orchestration Specification for Cloud Applications 20
UDP User Datagram Protocol 36

UE User Equipment 25
URI Uniform Resource Identifier 28
VIM Virtual Infrastructure Manager XVI
vIMS Virtual IP Multimedia Subsystem XVI
VL Virtual Link 12
VM Virtual Machine XX
VNF Virtual Network Function XVI
VNF-FG VNF Forwarding Graph 10
VNFM VNF Manager 19
VoIP Voice over IP 12
V2X Vehicle to anything 16
WebRTC Web Real-Time Communication 27
WSN wireless sensor networks 48
XCAP XML Configuration Access Protocol 28
XDMS XML Document Management Server 28
YAML Yet Another Markup Language XXII
5GPP 5G Infrastructure Public Private Partnership 1

BIBLIOGRAPHY

[1] Agnar Aamodt and Enric Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches", *in*: *AI communications* 7.*1* (1994), pp. 39–59.

[2] I. Afolabi, T. Taleb, K. Samdanis, et al., "Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions", *in*: *IEEE Communications Surveys Tutorials* (2018), ISSN: 1553-877X, DOI: 10.1109/COMST.2018.2815638.

[3] Armen Aghasaryan et al., "Fault detection and diagnosis in distributed systems: an approach by partially stochastic Petri nets", *in*: *Discrete event dynamic systems* 8.*2* (1998), pp. 203–231.

[4] M. Agiwal, A. Roy, and N. Saxena, "Next Generation 5G Wireless Networks: A Comprehensive Survey", *in*: *IEEE Communications Surveys Tutorials* 18.*3* (2016), pp. 1617–1655, ISSN: 1553-877X, DOI: 10.1109/COMST.2016.2532458.

[5] J. G. Andrews et al., "What Will 5G Be?", *in*: *IEEE Journal on Selected Areas in Communications* 32.*6* (2014), pp. 1065–1082, ISSN: 0733-8716, DOI: 10.1109/JSAC.2014.2328098.

[6] *Apache Cassandra*, URL: http://cassandra.apache.org/.

[7] Alexander Balke and Judea Pearl, "Counterfactual Probabilities: Computational Methods, Bounds and Applications", *in*: *CoRR* abs/1302.6784 (2013), URL: http://arxiv.org/abs/1302.6784.

[8] E. Baseman et al., "Relational Synthesis of Text and Numeric Data for Anomaly Detection on Computing System Logs", *in*: *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2016, pp. 882–885, DOI: 10.1109/ICMLA.2016.0158.

[9] Irad Ben-Gal et al., *Bayesian networks, encyclopedia of statistics in quality and reliability*, 2007.

[10] Renée Boubour et al., "A Petri net approach to fault detection and diagnosis in distributed systems. I. Application to telecommunication networks, motivations, and modelling", *in*: *Proceedings of the 36th IEEE Conference on Decision and Control*, vol. 1, IEEE, 1997, pp. 720–725.

[11] Raouf Boutaba et al., “A comprehensive survey on machine learning for networking: evolution, applications and research opportunities”, *in*: *Journal of Internet Services and Applications* 9.*1* (2018), p. 16, ISSN: 1869-0238, DOI: 10.1186/s13174-018-0087-2, URL: https://doi.org/10.1186/s13174-018-0087-2.

[12] F. Callegati et al., “Performance of multi-tenant virtual networks in OpenStack-based cloud infrastructures”, *in*: *2014 IEEE Globecom Workshops (GC Wkshps)*, 2014, pp. 81–85, DOI: 10.1109/GLOCOMW.2014.7063390.

[13] Christophe Cérin et al., “Downtime statistics of current cloud solutions”, *in*: *International Working Group on Cloud Computing Resiliency, Tech. Rep* (2013).

[14] *Charging architecture and principles*, tech. rep. TS 32.240 V16.0.0, 3rd Generation Partnership Project, 3GPP, URL: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1896.

[15] Nitesh V Chawla et al., “SMOTE: synthetic minority over-sampling technique”, *in*: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[16] S. Cherrared et al., “A Survey of Fault Management in Network Virtualization Environments: Challenges and Solutions”, *in*: *IEEE Transactions on Network and Service Management* (2019), pp. 1–1, DOI: 10.1109/TNSM.2019.2948420.

[17] S. Cherrared et al., “LUMEN: A global fault management framework for network virtualization environments”, *in*: *21st Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2018, DOI: 10.1109/ICIN.2018.8401622.

[18] Sihem Cherrared, Sofiane Imadali Eric Fabre, and Gregor Goessler, “Data Transformation Model For The Fault Management of Multi-tenant Networks”, *in*: ().

[19] *Clearwater*, URL: https://www.metaswitch.com/products/core-network/clearwater-ims-core.

[20] *Clearwater-live-test*, URL: https://github.com/Metaswitch/clearwater-live-test.

[21] *Cloud-Native IMS: Critical for Mobile Operators White paper*, 2019, URL: https://mavenir.com/resources/library/white-paper/cloud-native-ims.

[22] D. Cotroneo, L. De Simone, and R. Natella, “NFV-Bench: A Dependability Benchmark for Network Function Virtualization Systems”, *in*: *IEEE Transactions on Network and Service Management* (2017), ISSN: 1932-4537, DOI: 10.1109/TNSM.2017.2733042.

[23] R. N. Cronk, P. H. Callahan, and L. Bernstein, “Rule-based expert systems for network management and operations: an introduction”, *in*: *IEEE Network* 2.*5* (1988), pp. 7–21, ISSN: 0890-8044, DOI: 10.1109/65.17975.

[24] Lei Cui et al., “Piccolo: A Fast and Efficient Rollback System for Virtual Machine Clusters”, *in*: *IEEE Transactions on Parallel and Distributed Systems* 28.*8* (2017), pp. 2328–2341.

[25] Leonardo De Moura and Nikolaj Bjørner, “Satisfiability modulo theories: introduction and applications”, *in*: *Communications of the ACM* 54.*9* (2011), pp. 69–77.

[26] Rina Dechter, *Constraint Processing*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ISBN: 9780080502953.

[27] Rina Dechter and Judea Pearl, “Tree clustering for constraint networks”, *in*: *Artificial Intelligence* 38.*3* (1989), pp. 353–366.

[28] V. Del Piccolo et al., “A Survey of Network Isolation Solutions for Multi-Tenant Data Centers”, *in*: *IEEE Communications Surveys Tutorials* 18.*4* (2016), pp. 2787–2821, DOI: 10.1109/COMST.2016.2556979.

[29] *Docker*, URL: https://www.docker.com/.

[30] *Docker Compose*, URL: https://docs.docker.com/compose/.

[31] *Elastic Stack*, URL: https://www.elastic.co/products/elasticsearch.

[32] R. P. Esteves et al., “On the management of virtual networks”, *in*: *IEEE Communications Magazine* 51.*7* (2013), pp. 80–88.

[33] *ETCD*, https://github.com/etcd-io/etcd.

[34] GSNFV ETSI, “Network functions virtualisation (nfv): Architectural framework”, *in*: *ETsI Gs NFV* 2.*2* (2013), p. V1.

[35] GSNFV ETSI, “Network functions virtualisation (NFV); use cases”, *in*: *V1* 1 (2013), pp. 2013–10.

[36] NFVGS ETSI, “Network Functions Virtualisation (NFV); Management and Orchestration”, *in*: *NFV-MAN* 1 (2014), p. v0.

[37] *ETSI Opensource MANO*, URL: https://osm.etsi.org/.

[38] Eric Fabre et al., “Algorithms for distributed fault management in telecommunications networks”, *in*: *International Conference on Telecommunications*, Springer, 2004, pp. 820–825.

[39] K. Foerster, S. Schmid, and S. Vissicchio, “Survey of Consistent Software-Defined Network Updates”, *in*: *IEEE Communications Surveys Tutorials* 21.*2* (2019), pp. 1435–1461, ISSN: 1553-877X, DOI: 10.1109/COMST.2018.2876749.

[40] *From Webscale to Telco,the Cloud Native Journey*, tech. rep., July 2018.

[41] Marco Gavanelli, “The log-support encoding of CSP into SAT”, *in*: *International Conference on Principles and Practice of Constraint Programming*, Springer, 2007, pp. 815–822.

[42] Rainer Gerhards, *The Syslog Protocol*, RFC 5424, Mar. 2009, DOI: 10.17487/RFC5424, URL: https://rfc-editor.org/rfc/rfc5424.txt.

[43] Dimitris Giatsios et al., “SDN implementation of slicing and fast failover in 5G transport networks”, *in*: *Networks and Communications (EuCNC), 2017 European Conference on*, IEEE, 2017, pp. 1–6.

[44] S. F. Gillani et al., “Problem Localization and Quantification Using Formal Evidential Reasoning for Virtual Networks”, *in*: *IEEE Transactions on Network and Service Management* 11.*3* (2014), pp. 307–320, ISSN: 2373-7379, DOI: 10.1109/TNSM.2014.2326297.

[45] A. J. Gonzalez et al., “Dependability of the NFV Orchestrator: State of the Art and Research Challenges”, *in*: *IEEE Communications Surveys Tutorials* 20.*4* (2018), pp. 3307–3329.

[46] J. M. N. Gonzalez et al., “Root Cause Analysis of Network Failures Using Machine Learning and Summarization Techniques”, *in*: *IEEE Communications Magazine* (2017).

[47] Gregor Gössler and Daniel Le Métayer, “A general framework for blaming in component-based systems”, *in*: *Science of Computer Programming* 113 (2015), pp. 223–235.

[48] Gregor Gössler and Jean-Bernard Stefani, “Fault ascription in concurrent systems”, *in*: *International Symposium on Trustworthy Global Computing*, Springer, 2015, pp. 79–94.

[49] *Grafana the open observability platform*, URL: https://grafana.com/.

[50] Bruno Guerraz and Christophe Dousson, “Chronicles construction starting from the fault model of the system to diagnose”, *in*: *International Workshop on Principles of Diagnosis (DX04)*, Citeseer, 2004, pp. 51–56.

[51] R Guerzoni et al., “Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper”, *in*: *SDN and OpenFlow World Congress*, vol. 1, 2012, pp. 5–7.

[52] Zehua Guo et al., “Improving the Performance of Load Balancing in Software-Defined Networks through Load Variance-based Synchronization”, *in*: *Computer Networks* 68 (Aug. 2014), pp. 95–109, DOI: 10.1016/j.comnet.2013.12.004.

[53] W Haeffner et al., “Service function chaining use cases in mobile networks”, *in*: *Internet Engineering Task Force* (2015).

[54] Joseph Y. Halpern, “A Modification of the Halpern-Pearl Definition of Causality”, *in*: *CoRR* abs/1505.00162 (2015), URL: http://arxiv.org/abs/1505.00162.

[55] B. Han et al., “Network function virtualization: Challenges and opportunities for innovations”, *in*: *IEEE Communications Magazine* 53.*2* (2015), pp. 90–97, ISSN: 0163-6804, DOI: 10.1109/MCOM.2015.7045396.

[56] Masum Hasan, Binay Sugla, and Ramesh Viswanathan, “A conceptual framework for network management event correlation and filtering systems”, *in*: *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, IEEE, 1999, pp. 233–246.

[57] NL Hjort, *Pattern recognition and neural networks*, Cambridge university press, 1996.

[58] C Homberg, E Burger, and H Kaplan, “Request for Comments 6086 (“RFC 6086”)”, *in*: *Session Initiation Protocol (SIP) INFO Method and Package Framework, Internet Engineering Task Force* 36 (2011).

[59] C. Hounkonnou and E. Fabre, “Empowering self-diagnosis with self-modeling”, *in*: *8th international conference on network and service management (cnsm)*, 2012, pp. 364–370.

[60] Carole Hounkonnou, “Active self-diagnosis in telecommunication networks”, PhD thesis, Universit de Rennes 1, 2013.

[61] Ren-Hung Hwang and Yu-Chi Tang, “Fast Failover Mechanism for SDN-Enabled Data Centers”, *in*: *International Computer Symposium (ICS)*, IEEE, 2016.

[62] NFV ISG, “Network Function Virtualisation (NFV)-Resiliency Requirements,”” *in*: *ETSI GS NFV-REL* 1 (2014), p. v1.

[63] ISO, *ISO 8601:2000. Data elements and interchange formats — Information interchange — Representation of dates and times*, 2000, p. 29, URL: http://www.iso.ch/cate/d26780.html.

[64] *JavaScript Object Notation (JSON)*, URL: https://www.json.org/json-fr.html.

[65] *Jitsi*, URL: https://jitsi.org/.

[66] K. Katsalis et al., “5G Architectural Design Patterns”, *in*: *IEEE International Conference on Communications Workshops (ICC)*, 2016.

[67] *Kibana*, URL: https://www.elastic.co/fr/kibana.

[68] Kyungchan Ko et al., “Dynamic failover for SDN-based virtual networks”, *in*: *Network Softwarization (NetSoft), 2017 IEEE Conference on*, IEEE, 2017, pp. 1–5.

[69] D. Kreutz et al., “Software-Defined Networking: A Comprehensive Survey”, *in*: *Proceedings of the IEEE* 103.*1* (2015), pp. 14–76, ISSN: 0018-9219, DOI: 10.1109/JPROC.2014.2371999.

[70] Mark L Krieg, *A tutorial on Bayesian belief networks*, tech. rep., DEFENCE SCIENCE and TECHNOLOGY ORGANISATION SALISBURY (AUSTRALIA . . ., 2001.

[71] Salma Ktari, Stefano Secci, and Damien Lavaux, "Bayesian diagnosis and reliability analysis of Private Mobile Radio networks", *in*: *Computers and Communications (ISCC), 2017 IEEE Symposium on*, Heraklion, Greece, July 2017, DOI: 10.1109/ISCC.2017.8024695, URL: https://hal.sorbonne-universite.fr/hal-01589722.

[72] *kubernetes orchestration*, URL: https://kubernetes.io/.

[73] S Kumar et al., *Service function chaining use cases in data centers*, tech. rep., 2015.

[74] Seung-Ik Lee and Myung-Ki Shin, "A self-recovery scheme for service function chaining", *in*: *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*, IEEE, 2015, pp. 108–112.

[75] T. Li and L. Bai, "Model of Wireless Telecommunications Network Infrastructure Sharing amp; Benefit-Cost Analysis", *in*: *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*, vol. 2, 2011, pp. 102–105, DOI: 10.1109/ICIII.2011.171.

[76] Jingxian Lu, Christophe Dousson, and Francine Krief, "A self-diagnosis algorithm based on causal graphs", *in*: *The Seventh International Conference on Autonomic and Autonomous Systems, ICAS*, vol. 2011, 2011.

[77] Jingxian Lu et al., "Towards an autonomic network architecture for self-healing in telecommunications networks", *in*: *IFIP International Conference on Autonomous Infrastructure, Management and Security*, Springer, 2010, pp. 110–113.

[78] Inês Lynce and Joao Marques-Silva, "Building state-of-the-art sat solvers", *in*: *ECAI*, 2002, pp. 166–170.

[79] Antonio Manzalini et al., "Towards 5g software-defined ecosystems", *in*: (2016).

[80] M. Mechtri, C. Ghribi, and D. Zeghlache, "A Scalable Algorithm for the Placement of Service Function Chains", *in*: *IEEE Transactions on Network and Service Management* 13.*3* (2016), pp. 533–546, ISSN: 1932-4537, DOI: 10.1109/TNSM.2016.2598068.

[81] *Metasphere*, URL: https://www.metaswitch.com/products/applications/metasphere-mtas.

[82] *Metaswitch*, URL: https://www.metaswitch.com/.

[83] *Microsoft Z3 Solver*, URL: https://github.com/Z3Prover/z3.

[84] Fereydoun Farrahi Moghaddam, Abdelouahed Gherbi, and Yves Lemieux, “Self-healing redundancy for openstack applications through fault-tolerant multi-agent task scheduling”, *in*: *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, IEEE, 2016, pp. 572–577.

[85] *Monitoring Clearwater vIMS with the Elastic Stack*, URL: https://github.com/cherrared/Monitoring-Clearwater-vIMS-with-the-Elastic-Stack.

[86] A. I. Moustapha and R. R. Selmic, “Wireless Sensor Network Modeling Using Modified Recurrent Neural Networks: Application to Fault Detection”, *in*: *2007 IEEE International Conference on Networking, Sensing and Control*, 2007, pp. 313–318, DOI: 10.1109/ICNSC.2007.372797.

[87] *Nagios*, URL: https://www.nagios.com/.

[88] Susanta Nanda and tzi-cker Chiueh, “A Survey on Virtualization Technologies”, *in*: (Jan. 2005).

[89] S Natarajan et al., *An analysis of container-based platforms for nfv*, tech. rep., nfvrg-IRTF, 2016, URL: https://www.ietf.org/proceedings/94/slides/slides-94-nfvrg-11.pdf.

[90] *Netflix Chaos Monkey*, URL: https://netflix.github.io/chaosmonkey/.

[91] *Network Function Virtualization Research Group (NFVRG)-IRTF*, URL: https://trac.ietf.org/trac/irtf/wiki/nfvrg.

[92] *NetworkX*, URL: https://networkx.github.io/.

[93] Darren New and Marshall T Rose, “RFC 3195: Reliable Delivery for syslog”, *in*: (2001).

[94] ETSI GR NFV-REL, “Network Function Virtualisation (NFV); Reliability; Report on the resilience of NFV-MANO critical capabilities”, *in*: *ETSI Standard GS NFV-REL* 007 (2017), p. V1.1.2.

[95] *NORMA*, URL: https://5g-ppp.eu/5g-norma/.

[96] *ONOS*, URL: https://onosproject.org/.

[97] *Open Network Automation Framework (ONAP)*, URL: https://www.onap.org/.

[98] *Open Platform for NFV (OPNFV)*, URL: https://www.opnfv.org/.

[99] *OpenContrail*, URL: https://www.opendaylight.org/.

[100] *OpenDaylight*, URL: https://www.opendaylight.org/.

[101] *OpenFlow Switch Specification Version 1.5.1*, URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[102] *OpenSourceIMS*, URL: http://openimscore.sourceforge.net/.

[103] *OpenStack Neutron:* URL: https://wiki.openstack.org/wiki/Neutron#OpenStack_Networking_.28.22Neutron.22.29.

[104] *OpenStack Telemetry (Ceilometer)*, URL: https://wiki.openstack.org/wiki/Ceilometer.

[105] *OpenStack Telemetry tools:* URL: https://www.openstack.org/.

[106] *Openstack USER SURVEY*, URL: https://www.openstack.org/assets/survey/April2017SurveyReport.pdf..

[107] *OpenVIM*, URL: https://github.com/nfvlabs/openvim/.

[108] *OPNET*, URL: http://opnetprojects.com/opnet-network-simulator/.

[109] *OPNFV Doctor project:* URL: https://wiki.opnfv.org/display/doctor/Doctor+Home.

[110] *Oracle, agile IMS network infrastructure for session delivery*, URL: oracle.com/us/industries/communications/agile-ims-network-ds-2062279.pdf.

[111] *PaceMaker, Cluster Labs*, URL: http://clusterlabs.org/.

[112] Vasily Pashkov et al., "Controller failover for SDN enterprise networks", *in*: *Modern Networking Technologies(MoNeTeC)*, IEEE, 2014.

[113] Judea Pearl, "Fusion, propagation, and structuring in belief networks", *in*: *Artificial intelligence* 29.*3* (1986), pp. 241–288.

[114] C. Pham et al., "Failure Diagnosis for Distributed Systems Using Targeted Fault Injection", *in*: *IEEE Transactions on Parallel and Distributed Systems* 28.*2* (2017), pp. 503–516, DOI: 10.1109/TPDS.2016.2575829.

[115] Preeth E N et al., "Evaluation of Docker containers based on hardware utilization", *in*: *2015 International Conference on Control Communication Computing India (ICCC)*, 2015, pp. 697–700.

[116] El Hattachi Rachid et al., *NGMN Alliance 5G White Paper; Reliability; Report on Models and Features for End-to-End Reliability*, 2015.

[117] M. Rajiullah et al., "Syslog performance: Data modeling and transport", *in*: *2011 Third International Workshop on Security and Communication Networks (IWSCN)*, 2011, pp. 31–37, DOI: 10.1109/IWSCN.2011.6827714.

[118] Mohammad Rajiullah, Anna Brunstrom, and Stefan Lindskog, "Priority Based Delivery of PR-SCTP Messages in a Syslog Context", *in*: *Access Networks: 5th International ICST Conference on Access Networks, AccessNets 2010 and First ICST International Workshop on Autonomic Networking and Self-Management in Access Networks, SELF-MAGICNETS 2010, Budapest, Hungary, November 3-5, 2010, Revised Selected Papers*, ed. by Róbert Szabó et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 299–310, ISBN: 978-3-642-20931-4, DOI: 10.1007/978-3-642-20931-4_23, URL: https://doi.org/10.1007/978-3-642-20931-4_23.

[119] *RCA Framework*, URL: https://github.com/cherrared/FaultGraphExplorer..

[120] *Recent Clearwater vIMS architecture*, URL: https://www.projectclearwater.org/technical/clearwater-architecture/.

[121] Martin Rosvall and Carl T Bergstrom, "Maps of random walks on complex networks reveal community structure", *in*: *Proceedings of the National Academy of Sciences* 105.*4* (2008), pp. 1118–1123.

[122] Eleni Rozaki, "Network Fault Diagnosis Using Data Mining Classifiers", *in*: *Eleni Rozaki International Journal of Data Mining & Knowledge Management Process* (2015).

[123] Stuart Russell and Peter Norvig, "A modern approach", *in*: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), p. 27.

[124] K. Samdanis, X. Costa-Perez, and V. Sciancalepore, "From network sharing to multi-tenancy: The 5G network slice broker", *in*: *IEEE Communications Magazine* 54.*7* (2016), pp. 32–39.

[125] José Manuel Sanchez Vilchez et al., "Softwarized 5G Networks Resiliency with Self-Healing", *in*: *5GU - 1st International Conference on 5G for Ubiquitous Connectivity*, Levi, Finland, Finland, Nov. 2014, DOI: 10.4108/icst.5gu.2014.258123, URL: https://hal.archives-ouvertes.fr/hal-01165601.

[126] C. Sauvanaud et al., "Anomaly Detection and Root Cause Localization in Virtual Network Functions", *in*: *27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, DOI: 10.1109/ISSRE.2016.32.

[127] JD Case M Fedor ML Schoffstall and C Davin, "RFC 1157: Simple network management protocol (SNMP)", *in*: *IETF, April* (1990).

[128] Shi Shengyan et al., "Research on System Logs Collection and Analysis Model of the Network and Information Security System by Using Multi-agent Technology", *in*: *Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking and Security*, MINES '12, Washington, DC, USA: IEEE Computer Society, 2012,

pp. 23–26, ISBN: 978-0-7695-4852-4, DOI: 10.1109/MINES.2012.181, URL: http://dx.doi.org/10.1109/MINES.2012.181.

[129] Bin Shi et al., "Mercurial: A Traffic-Saving Roll Back System for Virtual Machine Cluster", *in*: *Proceedings of the IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, Washington, DC, USA, 2014, ISBN: 978-1-4799-7881-6, DOI: 10.1109/UCC.2014.143, URL: http://dx.doi.org/10.1109/UCC.2014.143.

[130] *SIPp*, URL: http://sipp.sourceforge.net/.

[131] K. Slavicek et al., "Mathematical Processing of Syslog Messages from Routers and Switches", *in*: *2008 4th International Conference on Information and Automation for Sustainability*, 2008, pp. 463–468, DOI: 10.1109/ICIAFS.2008.4783957.

[132] Oussama Soualah et al., "A link failure recovery algorithm for Virtual Network Function chaining", *in*: *IFIP/IEEE Symposium on Integrated Network and Service Management*, IEEE, 2017.

[133] Ma Igorzata Steinder and Adarshpal S. Sethi, "A survey of fault localization techniques in computer networks", *in*: *Science of Computer Programming* 53.*2* (2004), pp. 165 – 194, ISSN: 0167-6423, DOI: http://dx.doi.org/10.1016/j.scico.2004.01.010, URL: http://www.sciencedirect.com/science/ARTICLE/pii/S0167642304000772.

[134] J. M. Sánchez et al., "Self-modeling based diagnosis of Software-Defined Networks", *in*: *Network Softwarization (NetSoft)*, IEEE, 2015, DOI: 10.1109/NETSOFT.2015.7116174.

[135] Naoyuki Tamura and Mutsunori Banbara, "Sugar: A CSP to SAT translator based on order encoding", *in*: *Proceedings of the Second International CSP Solver Competition* (2008), pp. 65–69.

[136] Robert E Tarjan and Mihalis Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs", *in*: *SIAM Journal on computing* 13.*3* (1984), pp. 566–579.

[137] *Telecom Infra Project*, URL: https://telecominfraproject.com.

[138] *The docker deployment of the Clearwater vIMS*, URL: https://github.com/cherrared/clearwater-docker.

[139] *The docker deployment of the Clearwater vIMS*, URL: https://github.com/Metaswitch/clearwater-docker.

[140] *The Openstack deployment of the Clearwater vIMS*, URL: https://github.com/Metaswitch/clearwater-heat.

[141] ETSI TISPAN, "Universal Mobile Telecommunications System (UMTS) Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN)-IP Multimedia Subsystem (IMS)-Functional architecture", *in*: *ETSI TS* (), pp. 123–517.

[142] *TOSCA*, URL: https://docs.docker.com/compose/.

[143] H. Tsunoda and G. M. Keeni, "Managing syslog", *in*: *The 16th Asia-Pacific Network Operations and Management Symposium*, 2014, pp. 1–4, DOI: 10.1109/APNOMS.2014.6996575.

[144] ETSI GS NFV-EVE 011 V3.1.1, "Network functions virtualisation (nfv): Specification of the Classification of Cloud Native VNF implementations", *in*: (2018).

[145] P. Varga and I. Moldovan, "Integration of service-level monitoring with fault management for end-to-end multi-provider ethernet services", *in*: *IEEE Transactions on Network and Service Management* (2007), ISSN: 1932-4537, DOI: 10.1109/TNSM.2007.030103.

[146] *Vitrage*, URL: https://wiki.openstack.org/wiki/Vitrage.

[147] José Manuel Sánchez Vílchez, "Cross-layer self-diagnosis for services over programmable networks", PhD thesis, Institut National des Télécommunications, 2016.

[148] Toby Walsh, "Sat v csp", *in*: *International Conference on Principles and Practice of Constraint Programming*, Springer, 2000, pp. 441–456.

[149] *WeaveScope*, URL: https://www.weave.works/oss/scope/.

[150] Kenji Yamanishi and Yuko Maruyama, "Dynamic Syslog Mining for Network Failure Monitoring", *in*: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, Chicago, Illinois, USA: ACM, 2005, pp. 499–508, ISBN: 1-59593-135-X, DOI: 10.1145/1081870.1081927, URL: http://doi.acm.org/10.1145/1081870.1081927.

[151] Bo Yi et al., "A comprehensive survey of network function virtualization", *in*: *Computer Networks* (2018).

[152] J. Zaytoon and S. Lafortune, "Overview of fault diagnosis methods for Discrete Event Systems", *in*: *Annual Reviews in Control* (2013), ISSN: 1367-5788, DOI: https://doi.org/10.1016/j.arcontrol.2013.09.009, URL: http://www.sciencedirect.com/science/ARTICLE/pii/S1367578813000552.

[153] Lei Zhang et al., "A Novel Virtual Network Fault Diagnosis Method Based on Long Short-Term Memory Neural Networks", *in*: *86th Vehicular Technology Conference (VTC-Fall)*, IEEE, 2017.

[154] G. Zhaojun and W. Chao, "Statistic and Analysis for Host-Based Syslog", *in*: *2010 Second International Workshop on Education Technology and Computer Science*, vol. 2, 2010, pp. 277–280, DOI: 10.1109/ETCS.2010.128.

[155] M Zimmerman, D Allan, M Cohn, et al., "Openflow-enabled sdn and network functions virtualization", *in*: *Solution Brief, ONF, Solution Brief sbsdn-nvf-solution. pdf* 1 (2014).

UNIVERSITÉ DE RENNES 1

Titre: Gestion des fautes dans les réseaux multi-tenants et programmables

Mot clés : Gestion de pannes, réseaux virtuels, Approches basées modèles, auto-modelisation, diagnostic actif.

Resumé : La virtualisation est une tentative prometteuse pour résoudre certains défis de la 5G. La virtualisation consiste à exécuter des fonctions réseaux en tant que logiciels sur une infrastructure physique partagée. Cela optimise les coûts de déploiement et simplifie la gestion, mais il introduit de nouveaux défis tels que la topologie de réseau dynamique et le manque de visibilité. Dans cette thèse, nous proposons un algorithme d'auto-modélisation et un processus de diagnostic actif pour relever ces défis. Nous apprenons et validons le modèle défini par injection de pannes. Nous appliquons notre approche au use-case "virtual Ip Multimedia (vIMS)".

Title:Fault management of programmable multi-tenant networks

Keywords : Fault management, Network Functions Virtualization, Model-Based approaches, self-modeling, active diagnosis.

Abstract : Network Functions Virtualization (NFV) is one promising attempt at solving some of the 5G challenges. NFV is about running network functions as virtualized workloads on commodity hardware. This may optimize deployment costs and simplify the lifecycle management, but it introduces new challenges such as the dynamic network topology and the lack of visibility. In this thesis, we propose a self-modeling algorithm and an active diagnosis process to face these challenges. We define a dependency model learned from faults injection. The self-modeling and the active diagnosis approach was applied to the real-world virtual Ip Multimedia Subsystem (vIMS) use-case.