N° d'ordre : 2688

THÈSE

présentée

Devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1

Mention INFORMATIQUE

Par

Zakia Marrakchi

Équipe d'accueil

SUPELEC, Campus de Rennes, Équipe SSIR (Sécurité des Systèmes d'Information et Réseaux)

École doctorale

Mathématiques, Informatique, Signale et Télécommunications (MATISSE)

Composante universitaire

IFSIC

Détection d'intrusions comportementale dans les systèmes à objets répartis : modélisation des séquences de requêtes et de la répartition de leurs paramètres

Soutenue le 31 Mai 2002 devant la commission d'Examen

Composition du Jury :

M.	Jean-Marc Jézéquel	Président
M.	Guy Bernard	Rapporteur
M.	Baudouin Le Charlier	Rapporteur
M.	Gérardo Rubino	Directeur de thèse
M.	Ludovic Mé	Co-Directeur de thèse
M.	Eric Malville	Examinateur

Table des matières

Introduction

L'apparition des réseaux et l'explosion d'internet ont conduit, depuis les années 70, à la répartition des systèmes, des données et des programmes.

Un système réparti est un ensemble d'entités matérielles et logicielles distribuées qui coopèrent à la réalisation d'un objectif commun [54]. Ces entités peuvent résider sur des machines distribuées, être écrites dans des langages de programmation différents et communiquer via des réseaux de communications hétérogènes. Les entités communiquent selon un protocole client/serveur : les clients sont les applications destinées aux utilisateurs finaux et invoquant les services du système, les serveurs sont les programmes implantant ces services et gérant les ressources partagées par les entités du système.

L'objectif d'un environnement de communication répartie (appelé aussi *middleware*) est de permettre l'échange d'informations entre les différentes entités tout en garantissant [6] :

- la disponibilité des informations partagées,
- la fiabilité des services offerts,
- la gestion de la diversité technologique (machines, langages de programmation et réseaux de communication).

Par ailleurs, le paradigme «objet» a introduit une nouvelle approche de conception d'applications informatiques et a été largement utilisé dans les méthodes de conception [8] (OMT [53], OOA [11], etc.), les langages de programmation (Smalltalk [24], Eiffel [47], C++ [57], Java [49], etc.), les bases de données [15] et, plus tard, la conception des systèmes répartis.
La conception orientée objet consiste à modéliser un système par un ensemble d'entités indépendantes, appelées *objets*. Nous ne détaillons pas tous les concepts de cette approche, mais nous nous limitons à certaines propriétés de base [1], notamment :

- **L'encapsulation** des traitements et des données dans une même struc-

1. Pour plus de détails sur l'approche orientée objet, nous renvoyons le lecteur vers [46].

ture appelée *objet*. L'objet regroupe les propriétés (*attributs*) et les traitements (*méthodes*) de l'entité qu'il représente. Chaque objet a un identifiant qui le caractérise, un ensemble d'attributs et un ensemble de méthodes qui permettent de manipuler ses attributs.

- **L'abstraction** : les objets sont en réalité les instances de structures abstraites appelées *classes*. En effet, c'est au niveau d'une classe que sont définis les attributs et les méthodes des objets qu'elle représente. Créer une *instance* d'une classe consiste à créer un objet, c'est-à-dire une occurence des attributs de classe.
- **L'invocation de méthodes** : la communication entre les objets passe par l'invocation de leurs méthodes. L'implémentation des méthodes de chaque objet est totalement transparente aux autres objets. En revanche, les *signatures* (nom et paramètres) des méthodes sont visibles. L'exécution d'une méthode se fait donc par un appel conforme à sa signature. L'invocation d'une méthode met en relation deux objets, un objet invocant la méthode, appelé objet *client* et un objet invoqué, appelé objet *serveur*. Pour répondre au besoin de l'objet client, l'objet serveur peut lui même faire appel à une méthode d'un autre objet (délégation), il joue alors le rôle de *serveur/client*.

La vision «objet» repose sur des concepts qui s'adaptent bien à la représentation de systèmes répartis. En effet, les entités peuvent être vues comme des objets ayant une existence propre au sein du système et la communication entre les objets se fait par échanges de messages.

L'intégration de l'approche orientée objet dans la conception de systèmes répartis a donné lieu à des systèmes dits *à objets répartis*.

Les caractéristiques d'un système à objets répartis sont donc les suivantes :

- **Invocation de méthodes à distance**
  Ce mécanisme est la généralisation du mécanisme connu d'invocation de procédure à distance ou RPC (*Remote Procedure Call*) aux objets répartis. Un système à objets répartis est donc responsable de fournir les mécanismes de base d'invocation de méthodes d'un objet distant et de réception de réponses de manière transparente aux objets.
- **Communication par envoi de messages**
  L'invocation de méthodes à distance se fait par envoi de messages à un objet. Lorsqu'un client invoque le service d'un objet, l'appel à la méthode invoquée et ses arguments passent par la plate-forme de communication du système qui les encapsule dans un message et les achemine via le réseau jusqu'à l'objet serveur. Le système est en particulier responsable du codage portable des requêtes, des réponses et de leurs paramètres (*marshalling/unmarshalling*).

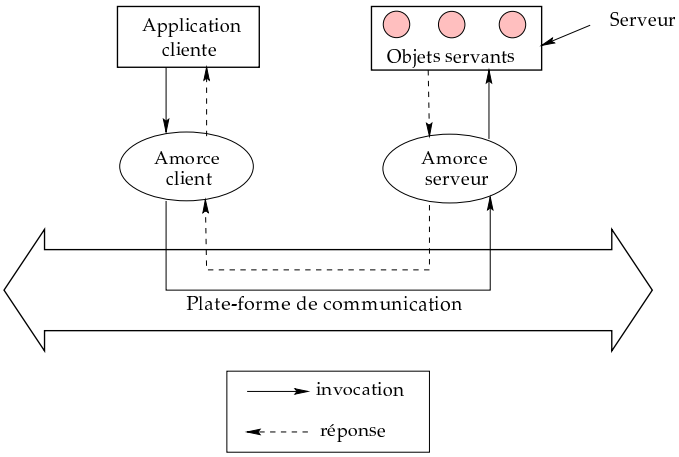

FIG. 1 – *Architecture d'un système réparti*

– **Délégation**
Pour répondre à la requête d'un client, un objet serveur peut à son tour demander le service d'un autre objet. Dans ce cas, il y a délégation et l'objet serveur doit se servir des droits du client pour pouvoir invoquer en son nom la méthode d'un autre objet.

La figure 1 présente les principaux composants d'un système réparti :

– L'application cliente qui invoque les services du système en envoyant des requêtes au serveur.
– Le serveur qui abrite plusieurs objets *servants* implantant les services rendus par le système.
– Les amorces[2] côté client et côté serveur. Les amorces côté client (appelées aussi *stubs*) jouent le rôle de proxy local du service proposé par le serveur. Elles sont aussi responsables de coder l'invocation du client dans un message transportable via la plate-forme de communication et de décoder la réponse du serveur. Côté serveur, les amorces (appelées aussi *skeleton*) jouent le rôle d'interfaces entre la plate-forme de communication et l'implantation. Elles permettent de décoder les messages d'invocation et de coder les réponses envoyées au client.
– La plate-forme de communication qui est responsable entre autres de l'envoi de messages entre l'application cliente et le serveur, de la localisation du serveur et de l'instanciation des amorces.

2. Les amorces sont aussi appelées souches ou talons.

Parmi les environnements répartis les plus connus, on peut citer :

- Le mécanisme Java/RMI (*Remote Method Invocation*) de Sun [58]. Ce mécanisme est basé sur les technologies de sérialisation d'objets en Java et utilise le protocole JRMP (*Java Remote Method Protocol*) afin de permettre la communication entre objets distants à travers le réseau. Plus tard, l'implantation de RMI sur IIOP a donné lieu au protocole RMI/IIOP [13]. Chaque objet serveur RMI spécifie une interface contenant toutes les méthodes (ou services) qu'il propose. Les amorces (*stubs*) permettent au client d'accéder aux services du serveur distant, désigné par une référence. Grâce au stub, l'appel à l'objet serveur distant se fait comme un appel à un objet local sur la même machine virtuelle Java ou JVM (*Java Virtual Machine*) du client. Le mécanisme Java/RMI présente l'avantage de pouvoir être utilisé sur n'importe quelle plateforme qui supporte une JVM. Il est notamment utilisé par les EJB (Enterprise JavaBeans) dans l'architecture J2EE de Sun. En revanche, il reste spécifique aux seuls objets Java, ce qui présente une limite majeure à la prise en compte de la diversité des objets dans les systèmes à objets répartis.
- L'environnement DCOM (*Distributed Component Object Model*) de Microsoft [48]. DCOM est l'extension de COM[3] (*Component Object Model*) aux objets répartis. DCOM est basé sur le protocole ORPC (*Object Remote Procedure Call*) et permet la communication entre objets distants à travers le réseau. Chaque objet DCOM peut définir autant d'interfaces que de services proposés. Pour invoquer le service d'un objet serveur, un objet client a besoin d'un pointeur sur l'une des interfaces de l'objet serveur qui implémente le service demandé. Ce pointeur permet à l'objet client d'invoquer le service d'un objet distant comme une invocation à un objet local. Un objet DCOM peut être écrit dans n'importe quel langage sur des plate-formes supportant le mécanisme COM, ce qui reste assez spécifique aux environnements *Windows* de Microsoft. Cette restriction constitue la principale faiblesse de DCOM, puisqu'elle limite l'ouverture à d'autres systèmes[4].
- L'environnement CORBA (*Common Object Request Broker Architecture*) proposé par l'OMG (*Object Management Group*) [26]. CORBA est une architecture de base permettant la communication et la coopération entre des objets écrits dans des langages différents, sur des systèmes

3. COM est l'environnement de base de Microsoft permettant l'échange de messages entre objets résidants sur une même machine.

4. Il existe néanmoins de récentes adaptations de DCOM aux environnements Unix et Linux [1].

d'exploitation hétérogènes et via des réseaux de communication différents. La communication entre objets se fait via un bus logiciel, appelé ORB (*Object Request Broker*), en utilisant le protocole GIOP (*General Inter-ORB Protocol*).
Chaque objet serveur CORBA définit une interface contenant la liste de ses méthodes. Pour invoquer les méthodes d'un objet serveur, un objet client a besoin de la référence de l'objet serveur à invoquer. Cette référence est communiquée à l'ORB qui est responsable de la localisation de l'objet serveur, de la transmission à cet objet de l'appel à la méthode et du retour du résultat à l'objet client. Les objets clients et serveurs coopèrent avec l'ORB via des interfaces de communication générées à partir d'une description des services offerts écrite dans un langage commun, appelé IDL (*Interface Definition Language*).

On peut se référer à [50] et [22] pour de plus amples détails concernant les environnements répartis dont la plupart proposent des mécanismes basés sur l'invocation de méthodes à distance afin de permettre la communication entre les applications réparties et hétérogènes. Dans le cadre de notre étude, nous avons porté notre intérêt sur l'architecture CORBA car elle permet, en outre, de répondre aux problèmes d'interopérabilité et d'hétérogénéité en spécifiant une vraie plate-forme standardisée (*middleware*) qu'est l'ORB. Cette plate-forme permet :

- la **coopération** entre des applications hétérogènes par le biais d'interfaces de communication standardisées (interfaces IDL),
- le **partage** des services offerts par l'ORB entre les applications réparties (localisation, communication, sécurité, etc.),
- la **portabilité** par la construction d'applications indépendantes du système et de l'environnement d'exécution.

La communication au sein d'un système à objets répartis suppose un «climat de confiance» entre les différents objets communiquant. Néanmoins, l'envoi de messages via le réseau, l'invocation à distance de méthodes et la délégation exposent les objets à certains risques :

1. *Atteinte à la confidentialité* : un objet peut écouter les communications et accéder ainsi aux données confidentielles auxquelles il n'a pas droit d'accès.
2. *Atteinte à l'intégrité des messages* : un objet peut interférer dans la communication entre deux objets et altérer l'information contenue dans le message par une modification, une insertion, une suppression.

3. *Usurpation d'identité*: un objet peut utiliser l'identité d'un autre objet afin d'invoquer des services auxquels il n'a pas droit. Ce risque augmente surtout en cas de délégation de droits d'accès entre objets.
4. *Déni de service*: un objet peut saturer par de fausses requêtes un service pour empêcher les vraies demandes d'être servies. C'est une technique d'attaque visant à rendre le système non disponible pour rendre ses services aux objets autorisés.

L'un des soucis majeurs pour les systèmes à objets répartis est donc de garantir la sécurité des échanges de messages, des données et des services fournis. Il s'agit, en d'autres termes, de définir une *politique de sécurité* qui établit l'ensemble des règles garantissant la *confidentialité*, l'*intégrité* et la *disponibilité* des données et des services. La mise en œuvre d'une telle politique consiste à mettre en place les services de sécurité permettant de protéger les objets de ces attaques [25]. Les cinq services de sécurité de base sont [32]:

- **L'authentification** qui assure la garantie de l'identité d'une entité du système.
- **L'autorisation** ou le **contrôle d'accès** qui attribue des droits d'accès aux entités.
- **La confidentialité** qui garantit le fait que seule l'entité autorisée peut prendre connaissance de l'information.
- **L'intégrité** qui garantit le fait que seule l'entité autorisée peut modifier l'information.
- **La non répudiation** qui garantit le fait que toute entité ayant réalisé une action sur le système ne peut pas nier l'avoir fait.

Les mécanismes qui implantent ces services sont basés esssentiellement sur des techniques cryptographiques (pour l'authentification, la confidentialité et l'intégrité) et sur les pare-feux (pour le contrôle d'accès) [27].

Ces techniques de sécurité ne sont pas, pour autant, incontournables et n'empêchent pas de manière définitive l'exploitation des failles présentes dans les systèmes à objets répartis. Le risque d'attaques est d'autant plus grand que les objets sont distants et communiquent via des réseaux hétérogènes. D'après un rapport du CERT [9], les principaux facteurs qui augmentent le risque d'attaques sur les systèmes répartis sont:

- L'explosion de la technologie répartie qui a été profitable aux attaquants pour mettre en place de nouvelles attaques *réparties* (les plus

connues sont des attaques de type DDoS (*Distributed Denial of Service*) [5] ;
- La diversité et la multiplicité des outils d'écoute et de collecte d'informations au sein des systèmes répartis ; ces outils sont facilement accessibles aux attaquants, même non spécialistes, d'où la facilité de mettre en œuvre ce type d'attaques ;
- La vulnérabilité des mécanismes de sécurité (les algorithmes cryptographiques peuvent être cryptanalysés, les mots de passe peuvent être volés, etc.).

Il n'est donc pas suffisant d'agir préventivement, c'est-à-dire de définir et mettre en œuvre une politique de sécurité. Il est également nécessaire de disposer de moyens permettant de détecter toute tentative de violation de la politique de sécurité, c'est-à-dire toute *intrusion*. La détection d'intrusions pourrait être réalisée par l'analyse manuelle des informations collectées sur l'activité d'un système. Néanmoins, face au volume important des données collectées, l'analyse manuelle n'est pas envisageable ; elle laisse place à des outils de détection automatique d'intrusions, appelés IDS (*Intrusion Detection System*). La mise en place d'un IDS implique donc une surveillance permanente des échanges entre les objets du système afin de s'assurer de leur légitimité. Cette surveillance est réalisée par le biais du mécanisme d'*audit de sécurité* qui collecte des informations sur les actions réalisées sur un système. Les informations collectées sont regoupées dans des fichiers appelés *traces d'audit*. L'analyse de ces traces d'audit permet la *détection d'intrusions*.

Deux approches sont utilisées : l'approche par scénario (*misuse detection*) et l'approche comportementale (*anomaly detection*). L'approche par scénario analyse des données d'audit à la recherche de scénarios d'attaques prédéfinis (dans une *base d'attaque*). Le principe consiste à considérer que tout ce qui est décrit dans la base d'attaque est reconnu comme intrusif ; le reste est considéré normal. En revanche, l'approche comportementale analyse les traces d'audit afin d'y détecter toute déviation par rapport à un comportement *normal* préalablement défini (dans une *base de comportements*). Tout ce qui est décrit dans la base est considéré normal ; le reste ne l'est pas.

Dans les environnements répartis, en particulier pour les applications basées sur CORBA, il n'est pas aisé de constituer *a priori* la base d'attaques sur l'ORB et sur l'application. En outre, une telle base serait très difficile à maintenir. Aussi, a-t-on retenu une approche comportementale consistant à

5. http://www.cert.org/tech_tips/denial_of_service.html

modéliser les comportements des objets CORBA impliqués dans une communication, afin de détecter ensuite toute déviation par rapport à ces comportements types considérés comme «normaux». C'est cette thèse qui est développée dans ce mémoire.

Notre approche est basée sur la mesure de déviations entre un comportement observé et le comportement de référence. Pour la modélisation du comportement de référence, nous pensons que les messages échangés entre les clients et les serveurs constituent les données discrimantes pour la définition d'un comportement normal. Ainsi, nous considérons, pendant une phase d'apprentissage, la suite de requêtes invoquées entre chaque couple client/serveur durant chaque connexion d'un client. Cela nous permet de construire le modèle de comportements de référence sous la forme d'un ensemble de chemins de taille variable au sein d'une structure arborescente. Le modèle tient compte, d'une part de l'enchaînement des requêtes, d'autre part des valeurs de leurs paramètres.

Nous proposons, dans une première phase, de mettre en œuvre notre modèle côté client et côté serveur. Côté client, l'approche consiste à modéliser le comportement de référence de chaque client observé vis-à-vis des objets répartis invoqués. Côté serveur, la modélisation consiste à construire le modèle de comportement des objets vis-à-vis des requêtes qu'ils reçoivent de leurs clients. Cette première mise en œuvre a pour objectif de mettre en avant l'apport d'une représentation arborescente avec des chemins de taille variable dans la modélisation d'un comportement normal. La prise en compte des paramètres d'une requête consiste, dans cette première phase, à modéliser les variations des valeurs de paramètres simplement en proposant des *intervalles de tolérance* pour chaque paramètre. Ces intervalles permettent la mesure de la déviation des valeurs observées par rapport aux valeurs de référence apprises[6].

Dans une seconde phase, nous étudions plus spécifiquement les paramètres des requêtes. Nous proposons d'une part, d'étudier statistiquement la dispersion des valeurs de paramètres et d'autre part, de prendre en compte les éventuelles corrélations[7] entre les paramètres d'une même requête qu'il est important de considérer dans la définition d'un comportement normal.

6. Cette étape a fait l'objet d'une publication à RAID'00 [42].

7. Par exemple, les corrélations possibles entre l'heure et le montant de retrait d'une somme d'argent, entre le revenu et le rendement d'une personne, etc.

Le mémoire est organisé de la manière suivante. Le chapitre 1 décrit l'état de l'art sur l'architecture CORBA et la détection d'intrusions et comprend deux parties. La première présente les aspects de l'architecture CORBA ayant trait à notre approche, mettant en avant les services de sécurité qui doivent y être intégrés. La deuxième présente les deux approches de détection d'intrusions et les outils les mettant en œuvre. Des travaux en détection d'intrusions en environnements répartis de type CORBA ont été menés par M.Stillerman et al. [56]. Nous les présentons en vue de positionner notre travail par rapport à ces recherches.

Le chapitre 2 présente notre approche de modélisation du comportement des clients. La première partie détaille la phase d'apprentissage : collecte des informations sur les interactions, construction de la base de comportement et des intervalles de tolérance. La deuxième partie présente la phase de détection : mesure du degré de déviation du comportement observé par rapport au comportement appris et génération d'alertes en fonction de cette déviation. Nous avons effectué deux types de tests pour cette approche. Les premiers portent sur une maquette CORBA que nous avons réalisée. Les seconds portent sur des données réelles d'une application en environnement à objets répartis. Ces données nous ont été rendues disponibles grâce à un partenariat avec FT R&D. La troisième partie du chapitre est dédiée à la présentation des résultats obtenus.

Le chapitre 3 présente les modifications apportées à notre modèle initial afin de modéliser le comportement des objets serveurs vis-à-vis des requêtes clients. Les modifications introduites au modèle initial portent sur les points suivants :

- Nous proposons une seule représentation arborescente qui modélise le comportement des objets invoqués suite aux requêtes de leurs clients.
- Nous considérons, en plus de la variation des valeurs de paramètres des requêtes, les délais de temps entre deux requêtes successives. Nous pensons que cette prise en compte est essentielle dans la définition d'un comportement normal.
- Enfin, nous avons présenté les vulnérabilités des architectures supportant des délégations. Nous proposons donc de prendre en compte la délégation de requêtes entre objets serveurs dans la modélisation de leur comportement.

La première partie du chapitre 3 présente la nouvelle architecture de base pour la modélisation du comportement des objets. Les deuxième et troisième

parties détaillent respectivement la phase d'apprentissage et la phase de détection. La dernière partie présente les résultats expérimentaux obtenus sur les données acquises à partir de l'application de FT R&D.

Le chapitre 4 présente la deuxième phase de nos travaux dans laquelle nous proposons un modèle qui rend compte, d'une part de la dispersion des valeurs de paramètres, d'autre part des éventuelles corrélations entre les paramètres d'une même requête. La première partie du chapitre présente les hypothèses émises pour le choix d'un modèle de représentation des valeurs apprises. La deuxième partie détaille l'algorithme d'apprentissage. La troisième partie donne l'algorithme de détection permettant de mesurer la déviation des valeurs observées par rapport aux valeurs apprises au vu des corrélations entre leurs paramètres. La dernière partie présente les résultats obtenus et nos conclusions sur cette dernière étape de notre travail.

Enfin, notre conclusion constitue une récapitulation générale du travail accompli : approches proposées, résultats obtenus et enseignements de nos recherches. Nous terminons par quelques perspectives pour les suites à donner à ces travaux.

Chapitre 1

Systèmes à objets répartis et détection d'intrusions

Nous présentons dans ce chapitre un aspect de l'état de l'art concernant l'ensemble des éléments sur lesquels s'appuie notre travail.

Nous commençons par une présentation de l'architecture de base utilisée pour notre approche de la détection d'intrusions dans les systèmes à objets répartis. Dans le paragraphe 1.1, nous présentons les composants de l'architecture CORBA : le bus CORBA et ses interfaces de communication. Dans le paragraphe 1.2 nous présentons les mécanismes de sécurité mis en place dans CORBA. Dans le paragraphe 1.3, nous étudions l'audit de sécurité, mécanisme à la base de la détection d'intrusions et nous présentons ce mécanisme dans le cas de l'architecture CORBA. Le paragraphe 1.4 est dédié plus spécifiquement aux approches et mécanismes de détection d'intrusions. Nous présentons respectivement dans le paragraphe 1.4.1 et 1.4.2 l'approche par scénario et l'approche comportementale. Nous donnons dans le paragraphe 1.4.3 une classification des outils de détection d'intrusions mettant en œuvre ces approches. Le paragraphe 1.5 est une présentation des travaux similaires aux nôtres tout en positionnant notre approche par rapport à ces travaux.

1.1 L'architecture CORBA

L'OMG (*Object Management Group*) est un consortium international créé en 1989 regroupant plus de 800 membres composés d'industriels, d'universitaires et de gouvernementaux (AT&T, Alcatel, Ericcson, HP, Iona, Oracle, Sun, NSA, etc.) qui travaillent à la mise en œuvre d'outils et d'environnements pour la communication entre systèmes répartis, basés sur des technologies orientées objet (UML, etc.).

Face à l'hétérogénéité et la diversité technologique dans les systèmes à objets répartis, l'OMG a proposé une architecture de base, appelée OMA (*Object Management Architecture*) [55], permettant à des objets hétérogènes de communiquer et d'interopérer de manière indépendante :

- de leur localisation,
- des langages de programmation dans lesquels ils sont écrits,
- du matériel et des systèmes d'exploitation sur lesquels ils sont exécutés,
- des réseaux de communication utilisés.

L'architecture OMA repose sur deux modèles : un modèle objet (*Object Model*) et un modèle de référence (*Reference Model*).

Le modèle objet permet de décrire les objets servants répartis en se basant sur les concepts «orientés objet». Ces concepts se résument dans :

- l'encapsulation des objets,
- l'accès aux objets par leurs interfaces,
- la transparence de la localisation et des implantations des objets,
- la communication par envoi de messages.

Le modèle de référence définit les services de base permettant les interactions entre les clients et les objets servants. Au cœur de ce modèle, un bus logiciel, appelé ORB (*Object Request Broker*), permet aux clients et aux différents objets servants de communiquer par un mécanisme d'échange de messages. Afin de permettre aux différents objets de communiquer avec l'ORB, l'OMG prévoit des interfaces de communication entre l'ORB et les objets.

L'ensemble des spécifications de ces services constitue la norme CORBA (*Common Object Request Broker Architecture*) dont les spécifications ont été définies en 1992 et reprises dans la version 2 [26] en 1995. La dernière

version de CORBA (3.0) date de 1998. Plusieurs ORBs ont été proposés par les industriels et dans le domaine public. Nous donnons quelques exemples d'ORBs respectant la norme CORBA v2.3 :

- du domaine privé : Visibroker de Borland/Inprise, Orbix et ORBacus de Iona, DAIS de Peerlogic, etc.
- du domaine public : MICO, ORBit proposé dans l'IHM Gnome de Linux, JavaORB proposé dans le JDK1.4, OmniORB2 de AT&T, etc.

Le paragraphe 1.1.1 détaille les différents composants de l'architecture CORBA, à savoir l'ORB et ses interfaces de communication avec les clients et les objets servants : stubs et squelettes. Nous limitons notre présentation aux seuls composants de l'architecture CORBA utiles à la compréhension du mécanisme d'échange de messages via l'ORB[1]. Le paragraphe 1.1.2 présente le protocole de communication prévu par l'OMG afin de permettre l'interopérabilité entre les ORBs. Le paragraphe 1.1.3 récapitule le mécanisme d'échange de messages via l'ORB en mettant en évidence le rôle de chaque composant présenté.

1.1.1 L'ORB et les interfaces de communication

L'ORB joue un rôle essentiel dans la communication par messages entre les clients et les objets servants. Son rôle consiste à :

- recevoir l'appel à la méthode lancé par le client,
- trouver l'objet servant et le code de la méthode invoquée,
- passer à l'objet servant les paramètres de la méthode invoquée,
- transmettre le résultat au client.

L'ORB est responsable de l'acheminement transparent des messages entre les clients et les objets servants. En effet, un client n'a aucune information sur :

- La localisation de l'objet servant invoqué : cet objet peut résider sur la même machine que le client, comme il peut en être distant. Du point de vue du client, l'appel à l'objet servant se fait comme un appel local.
- Le langage d'implantation de l'objet servant qui peut être différent de celui du client.
- Le système d'exploitation supportant l'objet servant.

1. Une présentation détaillée de l'architecture CORBA est donnée dans [22].

- L'état courant de l'objet servant au moment de l'invocation. En effet, c'est l'ORB qui se charge de l'activation de l'objet servant invoqué avant de lui envoyer la requête du client.
- Le protocole de communication utilisé par l'ORB pour l'acheminement de la requête jusqu'à l'objet servant.

Cette transparence est assurée grâce aux stubs, aux squelettes (détaillés dans le paragraphe 1.1.1.1 pour le côté client et dans le paragraphe 1.1.1.2 pour le côté serveur) et à l'adaptateur d'objets (BOA ou POA).

1.1.1.1 Les interfaces côté client

Les applications clientes communiquent avec l'ORB via deux principales interfaces (voir figure 1.1) :

- Les *interfaces d'invocation statique* ou SII (*Static Invocation Interface*), appelées aussi *Stubs Clients*. Ces interfaces jouent le rôle d'intermédiaires entre le client et l'ORB. Chaque objet servant spécifie dans une interface, écrite en IDL, les services offerts par ses méthodes. A partir de ces spécifications, un compilateur CORBA IDL génère les *stubs* dans le même langage que celui utilisé pour écrire le client. Il existe donc autant de stubs côté client que d'interfaces à des objets servants. Lors d'une invocation d'un objet servant, il suffit donc pour le client de faire appel au *stub* spécifique à l'objet invoqué.

- L'*interface d'invocation dynamique* ou DII (*Dynamic Invocation Interface*) permet la construction dynamique d'invocations d'objets. Contrairement aux interfaces statiques, le client n'utilise pas de *stub* pour invoquer une méthode d'un objet servant mais lance une requête directement au servant (via l'ORB) en spécifiant l'objet invoqué, la méthode à exécuter ainsi que ses paramètres.

1.1.1.2 Les interfaces côté serveur

Les objets servants sont les programmes qui implémentent les méthodes invoquées par les clients. Nous retrouvons, de manière analogue au côté client, des interfaces permettant aux objets servants de dialoguer avec l'ORB (voir figure 1.1). Il existe trois principales interfaces côté serveur :

- Les *interfaces statiques* ou SSI (*Static Skeleton Interface*) sont des interfaces pour chaque objet servant. Les *Skeleton* sont générés suivant le même principe que les *stubs*, par le compilateur IDL approprié, dans le

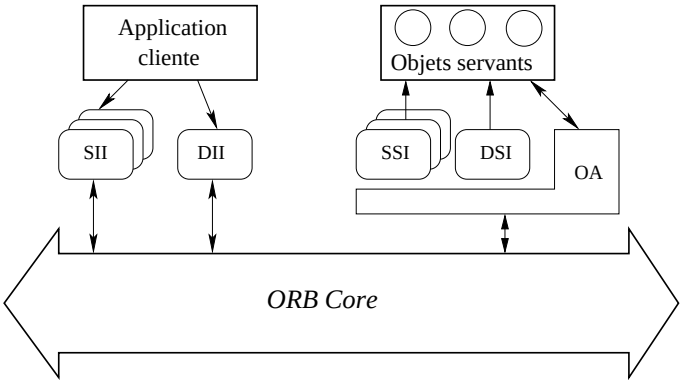

FIG. 1.1 – *Architecture de l'ORB*

même langage que celui utilisé pour écrire le code du servant. Il existe autant d'interfaces SSI que d'objets servants.

- L'interface DSI (*Dynamic Skeleton Interface*) est l'interface équivalente à l'interface *DII* côté client. Elle reçoit les invocations dynamiques du serveur et permet un accès direct à la méthode invoquée et à ses paramètres.
- L'adaptateur d'objets ou OA (*Object Adaptor*) joue un rôle important dans la communication via l'ORB. Il n'existe que du côté serveur puisqu'il est responsable de la gestion des différents objets servants d'un serveur. Il offre ainsi les services suivants :
  - *Activation des objets servants et de leurs implémentations.* Pour qu'un objet servant soit accessible par les clients, il faut que son implémentation soit activée après enregistrement auprès de l'adaptateur d'objets.
  - *Désactivation des objets servants et de leurs implémentations.* Lorsque le client n'a plus besoin d'un objet servant, l'adaptateur d'objets se charge de désactiver l'implémentation associée à cet objet. La désactivation permet de libérer la mémoire allouée à cet objet côté serveur.
  - *Génération et interprétation des références d'objets.* L'adaptateur génère les références des objets servants qu'il gère[2]. Il est aussi responsable d'interpréter les références contenues dans une requête adressée au serveur afin de localiser et d'activer l'objet invoqué

2. Les références d'objets ou IOR (*Interoperable Object Reference*) sont des données spécifiques à chaque objet invoqué contenant des informations utiles à sa localisation. Le contenu de ces références est détaillé dans le paragraphe 1.1.2.

correspondant.

- *Invocation de méthodes.* L'adaptateur d'objets prépare les implémentations à recevoir les invocations de méthodes en les activant. Une fois ces implantations activées, elles sont prêtes à traiter les invocations reçues. L'accès aux implantations se fait alors à ce niveau par les interfaces correspondantes (*SSI* ou *DSI*).
- *Sécurité des interactions.* L'adaptateur peut collaborer avec le service de sécurité afin de transmettre au serveur les informations permettant d'identifier le client invoquant le service de cet objet. Ces informations sont nécessaires pour le contrôle d'accès aux implémentations des objets servants par les clients.

Dans les premières spécifications de CORBA [26], l'OMG a défini un adaptateur d'objets standard commun à tous les ORBs appelé BOA (*Basic Object Adaptor*). Dans les spécifications de CORBA 2.2 [28], une version plus évoluée et portable de l'adaptateur d'objets a été proposée, appelée POA (*Portable Object Adaptor*).

1.1.2 Le protocole GIOP/IIOP

Afin de permettre l'interopérabilité entre les ORBs, l'OMG a défini un protocole unique de communication, appelé **GIOP** (*General Inter-ORB Protocol*). Ce protocole spécifie :

- Un format unique de représentation des données transmises appelé CDR (*Common Data Representation*). Ce format permet de représenter les requêtes du client et les réponses du serveur dans un message générique transportable sur le réseau. Ce format permet d'encoder et de décoder tous les types de données utilisées dans CORBA.
- Le format des messsages transportant les requêtes. Tous les messages GIOP sont composés d'une en-tête et d'un corps. Le contenu des messages GIOP diffère selon qu'il s'agit d'une requête ou d'une réponse. Le contenu de ces deux types de messages est détaillé dans le paragraphe 1.1.3.

L'OMG a aussi défini le protocole IIOP (*Internet Inter-ORB Protocol*) qui est l'implantation de GIOP sur la base de TCP/IP, la plus utilisée par les ORBs. Le rôle du protocole IIOP est d'acheminer les messages GIOP grâce au protocole TCP/IP. L'IIOP précise dans le message GIOP la référence à l'objet invoqué permettant de le localiser. Le message obtenu au niveau IIOP s'appelle un *Transport Message*. Les références d'objets interopérables

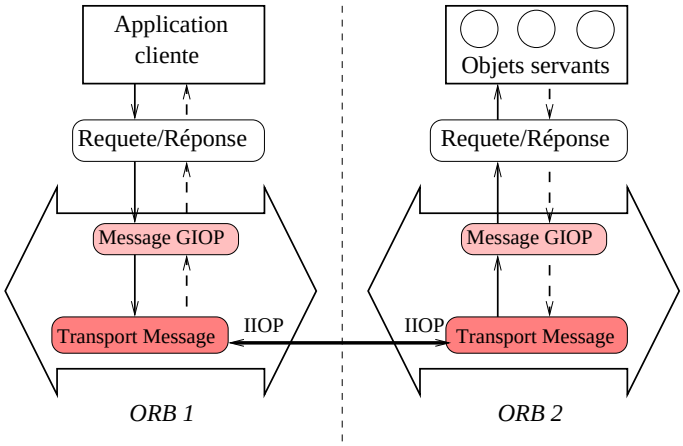

FIG. 1.2 – *Echange de messages entre objets*

ou IOR (*Interoperable Object Reference*) contiennent les informations utiles à leur localisation par l'ORB lors d'une invocation d'un objet client. Les principales informations contenues dans l'IOR[3] sont :

- l'adresse IP de la machine du serveur,
- le numéro de port utilisé par le serveur contenant l'objet invoqué,
- une clé contenant l'identificateur du POA et de l'objet servant permettant de le localiser dans le serveur.

1.1.3 Invocation d'un objet servant par un client

Nous passons en revue tous les éléments impliqués dans la communication entre les clients et les objets servants. Lorsqu'un client veut invoquer une méthode d'un objet servant, l'invocation passe par différentes étapes (voir figure 1.2). La méthode invoquée passe d'abord par le *stub* qui fait appel à l'ORB pour créer un message GIOP correspondant à l'appel de méthode. Le message requête GIOP obtenu comprend au niveau de l'en-tête les informations suivantes :

- un numéro unique de requête (contenant la méthode invoquée par le client),
- le nom de la méthode,
- la clé pour identifier l'objet cible.

3. Les références IOR données ci-après sont spécifiques au protocole IIOP.

Le corps du message contient toutes les informations relatives aux paramètres de la méthode invoquée. Le message GIOP est ensuite passé au niveau IIOP de l'ORB. L'IIOP encapsule le message dans un *Transport Message* qui contient l'adresse de l'objet servant dont la méthode a été invoquée. Le message est ensuite envoyé à l'ORB sur lequel tourne l'objet servant. Lorsque la requête du client a été traitée par l'objet servant, le mécanisme inverse se produit. Un message GIOP encapsule la réponse du servant. L'entête du message réponse comprend :

- le numéro de la requête à laquelle s'adresse la réponse,
- une information sur le type de réponse (normale, exception, etc.).

Le corps du message renvoie la valeur du résultat rendu par la méthode invoquée après traitement. Le message GIOP est envoyé au niveau IIOP et transmis, dans un *Transport Message*, à l'ORB sur lequel tourne le client. L'ORB retourne, par l'intermédiaire du stub, la réponse à la requête du client.

1.2 La sécurité dans CORBA

Les sytèmes à objets répartis, de part leur architecture, sont plus vulnérables que les systèmes classiques. En effet, la répartition des données et des traitements augmente, d'une part le nombre de points d'attaques sur le système, d'autre part le nombre d'interactions entre des objets hétérogènes, ce qui augmente le risque d'attaque [29].

L'OMG a identifié les risques pour l'architecture CORBA et a décrit les différentes vulnérabilités dans les spécifications des services de sécurité [29] :

- Abus de droits d'accès. Un objet autorisé peut tenter d'accèder à une information à laquelle il n'a pas droit.
- Possibilité pour un objet d'utiliser l'identité d'un objet autorisé pour accéder au système. Il s'attribue ainsi tous ses droits d'accès. Ce risque augmente lorsque l'objet autorisé délègue ses droits d'accès à d'autres objets pouvant ainsi exécuter des opérations en son nom.
- Possibilité pour un objet d'interférer dans une communication et avoir ainsi accès à des informations confidentielles.
- Altération de l'information contenue dans une communication entre objets (par ajout, modification ou supression).

Face à ces risques, l'OMG a défini un modèle de sécurité appelé SRM (*Security Reference Model*) [29] visant à protéger les objets CORBA ainsi que les messages échangés via l'ORB. Néanmoins, les applications basées

sur CORBA restent vulnérables. Nous présentons dans le paragraphe 1.2.1 le modèle SRM et dans le paragraphe 1.2.2 les limites de la sécurité dans CORBA.

1.2.1 Le modèle SRM (*Security Reference Model*)

Le modèle SRM propose les services de sécurité suivants :

- L'**identification** et l'**authentification** d'un «*principal*». L'OMG a défini un «*principal*» comme étant un utilisateur ou un objet ayant des attributs (appelés aussi *credentials*) au sein du système. Dans le modèle SRM, un principal a deux types d'attributs :
  - Une *identité* (parfois plusieurs selon la politique adoptée) utilisée pour accéder aux objets invoqués, pour l'audit de sécurité, pour la signature de messages, etc.
  - des *privilèges* déterminés par la politique de contrôle d'accès précisant les droits d'accès et les restrictions du principal.

  L'authentification de l'attribut d'identité permet de vérifier si le principal est bien celui qu'il prétend être pour accèder au système.
- Le **contrôle d'accès** permet de vérifier si le principal a le droit d'accéder à un objet étant donnés les privilèges du principal et les différents attributs de contrôle de l'objet. Dans certains cas, pour répondre à la requête du principal (appelé aussi «*initiateur*») l'objet invoqué fait appel au service d'autres objets (délégation). Le problème qui se pose alors est de décider des droits d'accès utilisés par l'objet invoqué (appelé aussi «*intermédiaire*») pour faire appel aux services d'un autre objet (qui peut être lui-même un intermédiaire ou bien la «*cible finale*») pour le compte du principal [4].
- La **délégation**. Il s'avère utile, dans certains cas, de déléguer tout ou partie des droits d'accès du principal. Néanmoins, la délégation de droits d'accès peut conduire, surtout en cas de délégation multiple, à des abus (usurpation de l'identité d'un objet, etc.) et à la création de points d'attaques au niveau des différents intermédiaires. Il est donc nécessaire d'établir une *politique de délégation* de droits d'accès qui définit les différentes stratégies de délégation autorisées au sein d'une application basée sur CORBA. L'OMG a donc défini un modèle de délégation qui propose cinq stratégies de délégation : *pas de*

4. Nous reprenons ici les termes *initiateur*, *intermédiaire* et *cible finale* utilisés dans les spécifications de l'OMG pour désigner les différents types d'objets impliqués dans une chaîne de délégation.

délégation, *délégation simple*, *délégation combinée*, *délégation composite* et *délégation tracée*. Ces différentes stratégies sont détaillées dans [29].

- L'**audit de sécurité** permet de garder une trace des actions réalisées sur le système. L'analyse de ces actions permet de détecter toute violation de la politique de sécurité du système. Il est important de décider des informations pertinentes à collecter, lesquelles dépendent du système et de la nature de l'application à auditer. L'OMG a défini une politique d'audit de sécurité permettant de définir les informations utiles à collecter. Deux principales politiques d'audit sont proposées :
  - *audit système* qui consiste à tracer les actions réalisées sur l'ORB et sur le service de sécurité CORBA mis en place (fichiers de logs),
  - *audit applicatif* qui consiste à tracer toutes les actions réalisées sur les objets servants CORBA (intercepteurs[5]).
- La **sécurité des communications** est basée sur l'entente mutuelle entre les clients et les objets servants sur le type et le degré de confidentialité des messages échangés. Elle est donc garantie par :
  - l'identification et l'autorisation des clients par les objets servants et vice versa,
  - la protection de l'intégrité et de la confidentialité des messages échangés.
- La **non-répudiation** qui fournit les preuves[6] irréfutables que des actions ont été réalisées afin d'éviter des tentatives de déni d'envoi ou de réception de données entre objets.
- L'**administration** des informations de sécurité, notamment l'administration de la politique de sécurité mise en place.

L'OMG a défini trois niveaux de sécurité dans CORBA :

- le niveau 1 comporte les services de sécurité mis en place au niveau de l'ORB et qui sont transparents pour l'application basée sur l'ORB (on parle de *unaware application level security*). Ce niveau comporte les services d'authentification, de contrôle d'accès, de confidentialité et d'audit.
- Le niveau 2 (appelé aussi *aware application level security*) comporte les services de sécurité mis en place au niveau de l'application même. Ce niveau ajoute la protection contre le rejeu et surtout l'accès à une interface d'administration de la sécurité.
- La non répudiation est considérée comme un service optionnel.

5. Les intercepteurs sont présentés dans le paragraphe 1.3.3.

6. Par exemple, la preuve de l'origine des données transmises à un objet servant, la preuve de réception des données pour le client, etc.

1.2.2 Limites de la sécurité dans CORBA

L'OMG propose une architecture de base permettant l'implantation des services de sécurité requis au sein d'une application basée sur CORBA. Selon l'OMG s'il y a des vulnérabilités, elles ne sont pas liées aux spécifications CORBA mais plutôt aux différentes implantations des ORBs.

Des travaux ont été alors menés afin d'identifier ces vulnérabilités dans les ORBs ([51], [38], [3]). Tous ces travaux mettent en avant les mêmes vulnérabilités. Les plus importantes concernent :

- Les références d'objets (IOR). Elles sont utilisées dans les invocations d'objets et véhiculent souvent des informations confidentielles sur les objets (les clés permettant de localiser les objets CORBA, les noms des serveurs internes, etc.). Ces références présentent ainsi une cible d'attaques pouvant générer des exceptions côté serveur (de type «objet non identifié» ou «accès refusé») que l'ORB n'est pas toujours capable de gérer.
- Le modèle d'authentification et d'autorisation. Les attributs prédéfinis d'un principal présentent des faiblesses et ne suffisent pas toujours à bien identifier un principal. Les mêmes faiblesses sont aussi constatées au niveau des droits d'accès (ou privilèges) accordés aux objets qui sont limités face aux différents scénarios possibles d'accès aux objets au sein d'une application CORBA. Le résultat de toutes ces insuffisances est le développement de modèles de contrôle d'accès propriétaires à chaque industriel, ce qui présente une limite à l'interopérabilité.

En résumé, les problèmes constatés sont de trois niveaux : niveau conceptuel (spécifications CORBA incomplètes), niveau implémentation (implémentations incomplètes et pas toujours portables des spécifications) et niveau exploitation (failles au niveau des modèles de sécurité). On ne peut donc pas considérer que la sécurité des applications sur CORBA ait atteint sa maturité aujourd'hui. Entre temps, des attaques exploitant les failles de ces systèmes et contournant leurs mécanismes de sécurité sont encore possibles. L'OMG déclare qu'il n'a aucune information sur des attaques liées à l'architecture CORBA et que les attaques possibles sur une application basée sur CORBA proviennent du réseau et concernent les protocoles et mécanismes cryptographiques implantés. Il s'avère donc nécessaire de surveiller les échanges de messages au sein d'une application CORBA afin de s'assurer de leur légitimité. Cette surveillance est réalisée par la mise en place de mécanismes d'audit de sécurité permettant de collecter les informations nécessaires à la détection d'intrusion.

1.3 Audit de sécurité

Dans le cadre qui nous préoccupe, auditer un système informatique consiste à enregistrer tout ou partie des actions réalisées sur le système afin de les analyser. Les informations collectées sont regoupées dans des fichiers appelés *traces d'audit*. L'analyse de ces traces peut servir à réaliser des statistiques sur l'utilisation des données et ressources d'un système. Lorsque l'analyse des traces d'audit sert à détecter d'éventuelles violations de la politique de sécurité d'un système, on parle d'*audit de sécurité*.

Il existe plusieurs sources de données d'audit dans un système. Chacune peut générer un nombre important de données dont la collecte et l'analyse constituent ce que l'on appelle la «détection d'intrusions». Pour une détection de qualité, il est essentiel de décider des sources de données à retenir et des types d'informations utiles à analyser. Nous détaillons dans le paragraphe 1.3.1 la sélection des sources de données. Nous présentons dans le paragraphe 1.3.2 les efforts de standardisation des formats de fichiers d'audit. Le paragraphe 1.3.3 présente enfin les mécanismes d'audit applicatif prévu dans CORBA : les intercepteurs.

1.3.1 Type d'informations à collecter

Les informations collectées sur l'activité d'un système peuvent provenir de diverses sources de niveau *système*, de niveau *réseau* ou de niveau *applicatif*.

Sources de données système

Les sources de données système permettent de collecter des informations sur l'activité du système d'exploitation. Les données collectées sont constituées essentiellement de l'historique des commandes systèmes passées par les utilisateurs et de l'usage des ressources partagées. Les informations collectées sont stockées dans des *fichiers d'audit*, constitués d'enregistrements d'audit. Chaque enregistrement décrit une action réalisée par un processus de la machine. Les IDS utilisant ces sources de données sont appelés *Host Based Intrusion Detection Systems* ou *HIDS*.

Sources de données réseau

Avec l'ouverture d'internet et l'interconnexion des systèmes informatiques, de nouvelles attaques véhiculées par le réseau sont apparues (Dénis de Service, IP Spoofing, ...). Les sources d'audit locales ne sont pas toujours les plus pertinentes pour détecter ce type d'attaques. Des outils (*sniffers*) ont

été alors développés afin de capturer le trafic réseau. Les IDS utilisant ces sources de données sont appelés *Network Based Intrusion Detection Systems* ou *NIDS*.

Sources de données applicatives

Les applications constituent aussi des sources de données importantes dans la sécurité d'un système. Des sondes installées au niveau d'une application permettent de remonter des informations de «haut niveau» sur l'utilisation faite de l'application et de ses ressources. Dans le cas des systèmes répartis ces sondes sont greffées dans l'environnement d'exécution des objets répartis et le mécanisme d'audit devient transparent pour l'application répartie. Les informations collectées permettent la détection d'anomalies dans le fonctionnement observé.

Nous avons utilisé ces sources de données (captées au niveau réseau) afin de collecter des informations sur les actions réalisées au sein d'une application basée sur CORBA.

1.3.2 Format des traces d'audit

Plusieurs travaux de standardisation des traces d'audit ont été menés afin de permettre l'échange de données et leur analyse par les différents systèmes d'analyse de traces d'audit. Parmi ces travaux nous citons :

- Bishop [7] qui a défini en 1995 un format standard pour les enregistrements d'audit. Dans ce format, il spécifie le codage de chaque champ contenu dans un même enregistrement sans fixer le nombre de champs dans un enregistrement afin de faciliter l'extension des fichiers.
- Le format NADF (*Normalized Audit Data Format*) a été proposé par les concepteurs du système d'analyse de traces d'audit ASAX [30]. Ce format propose de constituer chaque enregistrement comme une séquence de structures de données. Chaque structure est composée de trois champs : un *identifiant*, une *taille* et une *valeur* qui marquent respectivement le type, la taille et la valeur de la donnée représentée dans le fichier d'audit. Ce format, dit TLV (Type, Longueur, Valeur), a été proposé dans l'optique d'une indépendance totale des systèmes d'exploitation.
- Le groupe IDWG (*Intrusion Detection Working Group*) de l'IETF travaille sur la définition de formats standards et de procédures d'échange pour permettre le partage des données entre les IDS. Le groupe travaille

sur trois documents :

1. un document décrivant les besoins généraux pour permettre la communication entre les IDS [61],
2. un document décrivant le format des alertes répondant aux besoins décrits dans le premier document : le format IDMEF (*Intrusion Detection Message Exchange Format*) [12],
3. un document décrivant le protocole d'échange d'alertes spécifiées dans le deuxième document : le protocole IDXP (*Intrusion Detection Exchange Protocol*) [20].

– Les travaux du CIDF (*Common Intrusion Detection Framework*)
L'objectif de ce projet, financé par la DARPA, est de développer des protocoles et des API facilitant la communication et l'échange de données et ressources entre les IDS. Ce projet a aussi développé un langage CISL (*Common Intrusion Specification Language*) qui définit un standard de représentation des données [19].

L'objectif de tous ces travaux est de contribuer à homogénéiser les traces d'audit pour une meilleure interopérabilité entre les systèmes d'analyse de traces d'audit. Néanmoins, il n'existe pas aujourd'hui de format unique qui se soit imposé. Dans le cadre de notre étude, nous utilisons un format [7] simplifié répondant aux besoins de notre analyse du comportement et surtout adapté aux types de données collectées par les *intercepteurs* dans CORBA.

1.3.3 Audit de sécurité dans CORBA : les intercepteurs

Les intercepteurs sont des mécanismes interposés dans le chemin d'invocation et de réponse entre un objet client et un objet serveur. Ils sont capables de récupérer des informations passant par l'ORB (requêtes et leurs paramètres) et de les modifier (par exemple pour chiffrer les paramètres d'une opération). Ils sont utilisés pour exécuter des services de l'ORB, comme le contrôle d'accès, le chiffrement, la replication d'objets, etc. Il sont aussi utilisés pour la collecte d'informations sur tout ce qui transite par l'ORB. C'est cette dernière fonctionnalité qui nous intéresse pour faire de l'audit de sécurité dans CORBA.

Il existe deux types d'intercepteurs (voir Figure 1.3) : les intercepteurs de niveau requête (ou *Request Level Interceptors*) et des intercepteurs de niveau message (ou *Message Level Interceptors*) qui peuvent être implantés

7. Nous détaillons le format de nos traces d'audit dans le chapitre 2.

aussi bien côté client que côté serveur. Nous détaillons dans les paragraphes suivants ces deux types d'intercepteurs.

1.3.3.1 Les intercepteurs de niveau requête

Ces intercepteurs sont utilisés pour tous les services qui nécessitent la manipulation des requêtes (le contrôle d'accès, la réplication d'objets, etc.). Par exemple, l'intercepteur de contrôle d'accès se charge de récupérer les informations sur l'identité et les privilèges du principal ainsi que sa requête afin de vérifier ses droits d'accès à l'objet invoqué.

Lorsque l'ORB fait appel à l'intercepteur requête côté serveur, celui-ci commence par localiser l'objet serveur invoqué, le nom de la méthode, ses paramètres et le contexte d'exécution et transmet à l'objet serveur la requête qu'il vient de recevoir. Lorsque la requête a été traitée par l'objet serveur, l'intercepteur récupère (toujours à la demande de l'ORB) le résultat de la requête et le transmet à l'ORB qui fait appel à l'intercepteur requête côté client afin de lui transmettre le résultat.

1.3.3.2 Les intercepteurs de niveau message

Ces intercepteurs sont utilisés dans le cas d'une invocation à distance afin d'encapsuler la requête du client dans un message transportable sur le réseau. Ils sont aussi utilisés pour les services qui nécessitent la manipulation de messages (chiffrement).

Un intercepteur message peut être appelé par l'ORB afin d'envoyer ou de recevoir (et éventuellement modifier) les messages provenant des requêtes et des réponses. Lorsqu'un intercepteur message côté client est appelé pour envoyer un message à l'ORB, il tranforme le message reçu et l'envoie à l'ORB. Du côté serveur, l'ORB fait appel à l'intercepteur message afin de recevoir le message.

1.3.3.3 Informations collectées par les intercepteurs

Les intercepteurs sont capables de collecter des informations sur :

- l'identité du serveur (nom de la machine sur laquelle il tourne, le nom du serveur et le login sous lequel le serveur a été lancé),
- l'identité du principal,
- le nom de l'objet cible invoqué,
- le nom de la méthode invoquée et ses paramètres,

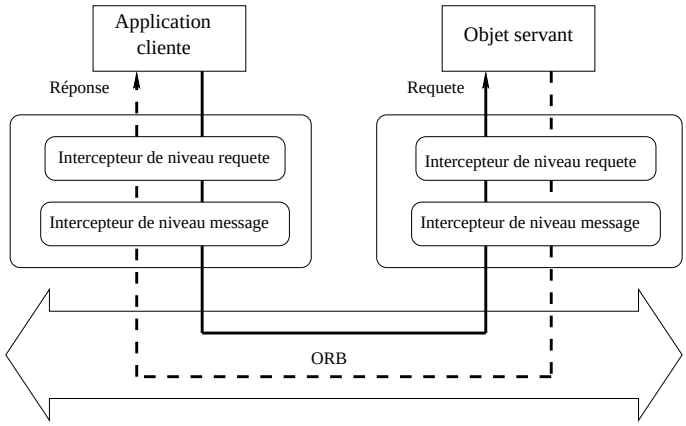

FIG. 1.3 – *Les intercepteurs*

- le statut de la réponse. S'il n'y a pas d'exception : `NO_EXCEPTION` ; s'il y a une exception, le type d'exception : `USER_EXCEPTION` ou `SYSTEM_EXCEPTION`.
- l'horodatage des actions réalisées.

1.3.3.4 Choix des intercepteurs

L'OMG présente les intercepteurs comme base des services nécessitant l'observation ou la transformation des requêtes et réponses, sans autre impact sur le noyau de l'ORB que l'appel d'une interface standard. C'est pourquoi nous avons choisi d'utiliser les intercepteurs pour acquérir de l'information sur les interactions entre clients et serveurs.

Un intercepteur de requête en reçoit les paramètres, peut les modifier et invoquer d'autres objets puis fait suivre la requête. Nous n'utilisons que ce type d'intercepteur. En effet, le niveau message qui permet l'accès aux fragments véhiculant les requêtes et les réponses est sans intérêt pour nous. En outre, Visibroker (l'ORB sur lequel notre maquette est construite) n'offre que le niveau requête.

1.4 Méthodes et outils de détection d'intrusions

La détection d'intrusions a été introduite en 1980 par J.P Anderson qui a été le premier à montrer l'importance de l'audit de sécurité [4] dans la détection d'éventuelles violations de la politique de sécurité d'un système.

Anderson définit une violation de la politique de sécurité comme une tentative délibérée :

- d'accéder de manière non autorisée à l'information. L'accès non autorisé peut être effectué par une personne extérieure au système (n'ayant aucun droit d'accès) ou bien par une personne interne (ayant des droits d'accès limités ; dans ce cas il y a violation de droit d'accès). Dans les deux cas on parle de violation de la contrainte de confidentialité des données.
- de modifier l'information de manière non autorisée. Il s'agit, dans ce cas, d'une atteinte à l'intégrité des données.
- de détériorer tout ou partie des données et ressources d'un système afin de le rendre inutilisable ou non fiable. C'est le cas de violation de la contrainte de disponibilité des données et des ressources d'un système.

La première approche possible en détection d'intrusions est appelée «**approche comportementale**». Elle cherche à détecter toute déviation par rapport à un comportement normal préalablement défini, généralement par apprentissage, et stocké dans une base de comportements.

Cette approche a été utilisée par Anderson qui a proposé de décrire statistiquement le comportement usuel d'un utilisateur, afin de détecter toute action inhabituelle de cet utilisateur (horaires de connexions anormaux, volume important d'actions sur le système, etc.).

Une autre approche consiste à modéliser non plus des comportements normaux, mais des comportements interdits sous forme de scénarios d'attaques. Il s'agit de l'«**approche par scénario**». Contrairement à l'approche comportementale, il n'est pas possible, dans cette approche, d'attribuer un profil à l'attaquant du fait que les informations collectées sur les attaquants par les systèmes d'audit ne sont généralement pas suffisantes à l'élaboration d'un modèle statistique permettant de modéliser l'attaquant. En revanche, les systèmes d'audit permettent de collecter un nombre important d'informations sur les actions réalisées, ce qui permet de construire des signatures d'attaques et de les stocker dans une base de signatures. La détection d'intrusions suivant cette approche consiste à rechercher, dans les traces d'audit, les signatures de la base.

Chacune de ces approches peut conduire à des faux positifs (détection d'attaque(s) en absence d'attaque) ou à des faux négatifs (absence de détection en présence d'attaque(s)) :

- Un outil basé sur l'approche comportementale génère une alarme dès qu'il détecte un comportement non appris. Or, la déviation du compor-

tement observé peut être due à une évolution naturelle de l'environnement et du système : c'est un faux positif. En outre, l'attaquant (utilisateur interne malicieux) peut modifier lentement son comportement afin de parvenir à un comportement intrusif qui, ayant été progressivement appris, ne sera pas détecté : c'est un faux négatif.

- Le risque de faux positifs est moindre avec l'approche par scénario car l'activité litigieuse est décrite dans la base d'attaque. Cependant, la qualité de la signature est importante : si elle n'est pas assez précise, elle peut également conduire à de nombreux faux positifs. Enfin, bien évidemment, si la signature de l'attaque n'est pas dans la base (comme c'est le cas pour les nouvelles attaques), l'attaque en question ne sera pas détectée (c'est là un problème similaire à ce que l'on connait avec les bases de signatures de virus).

Les outils mettant en œuvre ces deux approches sont appelés des systèmes de détection d'intrusion ou IDS (*Intrusion Detection Systems*).

1.4.1 Mise en œuvre de l'approche par scénario

L'approche par scénario consiste à localiser, dans les traces d'audit, des scénarios d'attaque prédéfinis (dans une base d'attaque). Cette base est composée de suites de signatures décrivant les scénarios. Plusieurs approches ont été proposées afin de définir les signatures d'attaques : les systèmes à base de règles [30], l'analyse de signatures d'attaques [36], les algorithmes génétiques [44], etc. Les deux premières ont été les plus utilisées ; nous les décrivons ci-après.

1.4.1.1 Systèmes à base de règles

Un système à base de règles est composé d'une base de connaissances et d'un moteur d'inférences. La base de connaissances est elle-même composée d'une base de règles décrivant les attaques et d'une base de faits contenant les événements d'audit relatifs à ces attaques. Le moteur d'inférences est capable, en phase de détection, de raisonner à partir des informations contenues dans la base de connaissances et de décider de la présence d'une attaque dans les traces d'audit. Cette méthode a été utilisée dans [39], mais elle est très peu utilisée par les outils récents. L'inconvénient de cette approche réside dans la difficulté de mise à jour de la base des règles. En effet, l'ajout ou la suppression d'une règle dans la base doit tenir compte des interactions avec les autres règles, ce qui constitue une démarche difficile à mettre en pratique lorsque le volume de données est important.

1.4.1.2 Analyse de signatures (*Pattern matching*)

Cette approche consiste à représenter les scénarios d'attaque sous forme de suites d'événements appelées signatures. En phase de détection, ces signatures sont comparées aux traces d'audit afin de reconnaître un scénario d'attaque [37]. Cette méthode est utilisée par de très nombreux IDS du commerce (RealSecure, eTrust, etc.) et du domaine public (SNORT, etc.). Elle donne de bons résultats de détection avec des temps de traitement acceptables lorsque les signatures sont très simples mais présente l'inconvénient de détecter seulement les scénarios d'attaque connus du système.

1.4.1.3 Avantages et inconvénients de l'approche par scénario

L'approche par scénario permet de détecter les attaques exploitant les vulnérabilités connues du système. Le principal avantage de cette méthode est le faible taux de faux négatifs : tant que l'attaquant se comporte de la même manière que celle décrite dans la base, le risque d'erreur est très faible. En revanche, de nouvelles attaques ne sont pas systématiquement reconnues à cause de la difficulté de mise à jour de la base d'attaque.

1.4.2 Mise en œuvre de l'approche comportementale

L'approche comportementale consiste, dans une première phase, à surveiller l'activité de l'utilisateur ou du système à protéger afin de construire son profil, appelé aussi *comportement normal*. Dans une seconde phase, elle mesure la déviation par rapport au comportement appris ; toute déviation du comportement peut révéler une intrusion.

Plusieurs méthodes de modélisation du comportement ont été proposées pour définir ce qui est *normal* : des outils statistiques [17] ; des systèmes experts [33] ; des méthodes plus récentes se sont inspirées de l'immunologie [18]. Ces techniques sont présentées dans les paragraphes suivants.

1.4.2.1 Méthodes statistiques

Le profil est calculé à partir de variables considérées comme aléatoires et échantillonnées à intervalles réguliers. Dans un environnement informatique classique (réseau de machines UNIX et NT), ces variables peuvent être le temps processeur utilisé, la durée et l'heure des connexions, etc. Un modèle statistique est utilisé pour construire la distribution de chaque variable et pour mesurer, par le biais d'une grandeur synthétique, le taux de déviation entre un comportement courant et le comportement passé. L'avantage de ces méthodes est l'apprentissage continu du comportement habituel, ce qui

permet une représentation fidèle du comportement usuel. Néanmoins le principal inconvénient de l'apprentissage continu est le risque d'apprendre des comportements intrusifs, progressivement insérés dans le comportement habituel. Cette méthode a été proposée par Anderson en 1980 [4], puis reprise dans plusieurs outils, dont NIDES [33], jusqu'au milieu des années 90.

1.4.2.2 Les systèmes à base de règles

Cette approche est aussi utilisée pour modéliser le comportement normal d'un utilisateur [59]. La base de règles décrit le profil de l'utilisateur au vu de ses précédentes activités. En phase de détection, son comportement courant est comparé aux règles, en vue de détecter une éventuelle anomalie. Cette approche implique une mise à jour régulière de la base de règles. Cette méthode, de performances décevantes, est, à notre connaissance, aujourd'hui abandonnée.

1.4.2.3 Approche immunologique

Cette approche, proposée en 1996 par Forrest [18] est inspirée du système immunitaire pour appliquer la même stratégie de défense aux systèmes informatiques. Cette analogie avec l'immunologie biologique consiste à construire un modèle de comportement normal des services (et non des utilisateurs) au travers de courtes séquences d'appels systèmes considérées suffisamment représentatives de l'exécution normale des services considérés. La phase d'apprentissage consiste à observer un service pendant un certain temps afin de construire une base des séquences d'appels normales. En phase de détection, toute séquence étrangère à cet ensemble est considérée comme une potentielle exploitation d'une faille de sécurité du service.

1.4.2.4 Approche bayésienne

Les réseaux Bayésiens permettent de modéliser des situations dans lesquelles la causalité joue un rôle, mais où la connaissance de l'ensemble des relations entre les phénomènes est incomplète, de telle sorte qu'il est nécessaire de les décrire de manière probabiliste [45]. Les indications obtenues progressivement sur l'état du système modélisé influent sur la confiance que l'on accorde à une proposition donnée. Quelques travaux sont basés sur cette approche [5] [60].

1.4.2.5 Avantages et inconvénients de l'approche comportementale

L'approche comportementale consiste à identifier dans les traces d'audit le comportement normal, préalablement défini. Toute déviation de ce comportement peut révéler une intrusion. Le principal avantage de cette approche est la capacité à détecter de nouvelles attaques. Néanmoins, elle présente quelques inconvénients :

- un taux de faux positifs assez élevé en cas de modification brutale de l'environnement (nouvelle fonctionnalité ou application, nouveaux utilisateurs, etc.).
- des faux négatifs en cas de modification lente du comportement d'un utilisateur dans l'intention d'habituer le système à apprendre un comportement intrusif.

1.4.3 Classification des outils de détection d'intrusions

Le milieu des années 80 a vu se développer de nombreux travaux inspirés du modèle d'Anderson et plusieurs IDS le mettant en œuvre ont été commercialisés. De 1984 à 1986, Denning et al. [17] ont travaillé à la conception d'un système de détection d'intrusions basé sur des méthodes statistiques et des systèmes experts. Cet outil, appelé IDES (*Intrusion Detection Expert System*) est le premier outil hybride regroupant l'approche comportementale et par scénario. Le prototype d'IDES a été ensuite développé et amélioré pour aboutir en 1993 à NIDES. NIDES ainsi que d'autres outils développés durant la même période ont montré la possibilité de distinguer un comportement normal d'un comportement anormal sur une machine UNIX, en utilisant des informations sur les commandes passées sur cette machine. Cependant, si le comportement de l'utilisateur est trop riche, sa modélisation devient difficile et l'approche comportementale atteint sa limite. A partir du milieu des années 90, c'est donc l'approche par scénario qui est principalement implantée dans les outils. Aujourd'hui, la plupart des IDS commerciaux reposent sur ce principe même si l'on assiste, depuis quelques mois, à un regain d'intérêt pour l'approche comportementale. Du côté de la recherche, l'intérêt porte toujours sur l'implantation des deux approches et même vers le développement d'outils les combinant. La tendance va ainsi de plus en plus vers la coopération entre IDS [45] pour tirer partie des avantages de chacun.

Il existe de nombreux IDS du domaine public et industriel qu'il n'est pas possible de présenter de manière exhaustive. En revanche, nous proposons une classification inspirée de [31] (voir figure 1.4) selon les critères suivants :

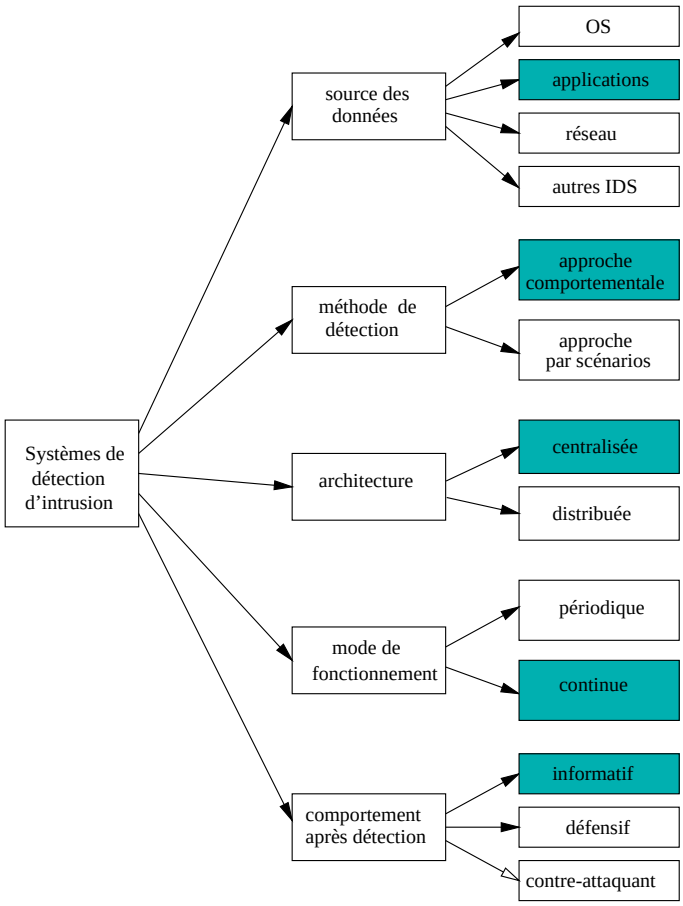

FIG. 1.4 – *Classification des IDS (les choix faits dans le cadre de cette thèse sont grisés)*

- **Source de données**
  Les IDS peuvent utiliser diverses sources de données : source de données *système*, source de données *réseau*, source de données *applicative* ou encore des *alertes* (dans le cas des IDS basés sur une approche par scénario).
- **Méthode de détection**
  Les IDS peuvent être classés selon la méthode de détection implantée : *comportementale* ou *par scénario*.
- **L'architecture**
  L'architecture des IDS peut être centralisée ou distribuée. Distribuer le traitement des événements permet de diminuer considérablement le transfert de données des points d'audit locaux vers un point d'analyse

central, mais ne permet pas d'avoir une vision globale du système. L'avantage d'une architecture centralisée est de regrouper des événements qui, localement peuvent être considérés comme normaux, mais dont la corrélation peut révéler une attaque.

- **Mode de fonctionnement**
  Les IDS peuvent être aussi classés selon leur mode de fonctionnement. Les IDS peuvent surveiller continuellement les événements observés ou bien analyser le journal d'audit de manière périodique.
- **Comportement après détection**
  Deux types de comportements après détection peuvent être envisagés par les IDS. Un comportement passif ou informatif consiste à générer une alarme (message à l'administrateur, etc.). Un comportement actif ou défensif consiste à réagir à l'attaque (reconfiguration du firewall, rupture de connexion, blacklist, etc.).

L'approche que nous proposons dans cette thèse pour la détection d'intrusion dans CORBA se situe dans cette classification avec les spécifications suivantes (voir figure 1.4) :

- Les sources de données sont applicatives. Ces données sont collectées par les mécanismes d'intercepteurs dans CORBA.
- Nous avons exposé dans l'introduction les difficultés de disposer d'une base d'attaque pour CORBA. Nous avons donc opté pour une approche comportementale.
- La collecte des données se fait de manière répartie sur tous les objets observés, mais l'analyse des traces d'audit se fait de manière centralisée.
- L'utilisation de l'IDS est continue. Les alertes sont générées au fur et à mesure de l'utilisation de l'application CORBA.
- Le comportement après détection se limite à la génération d'alerte.

1.5 Détection d'intrusions dans CORBA : état de l'art

A notre connaissance, les seuls travaux portant sur la détection d'intrusions en environnement CORBA ont été menés par M.Stillerman et al. [56, 41]. Ces travaux sont inspirés de l'approche immunologique proposée par Forrest [18] afin de l'appliquer à des systèmes répartis. L'analogie avec le système de défense immunitaire faite par Forrest permet de considérer une application CORBA comme étant l'organisme à protéger dont les cellules sont les clients de l'application. Le système de défense immunitaire est basé

sur le principe de reconnaissance des cellules étrangères dans les cellules du corps. De manière analogue, l'IDS développé par M.Stillerman et al., appelé *CORBA Immune System*, compare le comportement observé au comportement passé et décide, en fonction de la variation observée, de la présence d'anomalie. L'approche suppose que les messages échangés entre client et serveur constituent les données discriminantes pour la définition d'un comportement normal. Seules les signatures des requêtes clients interceptées sont considérées ; les paramètres de ces requêtes ne sont pas pris en compte. L'algorithme utilisé pour la détection permet, lors de l'observation du comportement courant d'un objet, de déterminer si ce comportement est conforme au passé (algorithme dit *sliding window algorithm* [56]). L'algorithme consiste à répertorier, en phase d'apprentissage, toutes les suites de signatures de requêtes d'un client au serveur et créer à partir de ces suites des séquences de taille fixe qui constituent la base de comportements. Par exemple, considérons la suite de signatures suivantes (chaque signature est réduite à une lettre) : $ABCDBCDBABC$. Si la largeur choisie pour la fenêtre est de 4 par exemple, nous obtenons les séquences (appelées *N-grams*) suivantes : $ABCD$, $BCDB$, $CDBC$, $DBCD$, $BCDB$, etc. Ces séquences sont répertoriées et données en entrée à un automate d'états finis qui permet de prévoir à partir d'une séquence et de la nouvelle signature de requête, la séquence qui doit suivre. En phase de détection, les séquences de taille N (N étant préalablement fixée durant l'apprentissage) sont créées et comparées à celles de la base de comportements. Si la séquence observée n'est pas dans la base, il y a anomalie dans le comportement mais il n'y a pas pour autant une alerte déclenchée. Le nombre de séquences anormales observées et le temps écoulé entre ces anomalies détermine le déclenchement d'une alerte.

Nous appuyons notre approche sur le travail de M.Stillerman et al. : nous adoptons une approche comportementale et nous nous intéressons aux clients et aux objets d'une application dans un environnement à objets répartis, en prenant l'exemple d'une application CORBA.

Cependant, nous enrichissons leur approche en prenant en compte les éléments suivants :

- Dans la définition d'un comportement normal, nous prenons en compte, outre les signatures de méthodes, les valeurs de leurs paramètres. En effet, une requête peut être considérée normale si elle est toujours invoquée avec la même signature mais elle peut révéler une anomalie si elle est invoquée avec des valeurs inhabituelles de ses paramètres. Il est donc important d'affiner l'analyse d'une requête et de considérer les

paramètres comme une donnée discriminante dans la définition d'un comportement normal.

- Nous avons opté pour une représentation arborescente du comportement normal. Un chemin de l'arbre représente une suite d'invocations du client entre chaque connexion (racine de l'arbre) et déconnexion (feuille de l'arbre). Nous obtenons ainsi un arbre avec des chemins de tailles variables correspondants au nombre d'invocations durant chaque connexion. Cette représentation permet une meilleure prise en compte du comportement habituel observé. Debar, Dacier et Wespi [14] ont montré les avantages des séquences de taille variable par rapport aux séquences de taille fixe. C.Marceau a également proposé récemment une évolution du travail de M.Stillerman et al. avec des patterns de tailles variables [40].
- Nous prenons également en compte dans notre modèle de comportement le temps écoulé entre deux invocations. Il y a lieu de penser, en effet, qu'un écart trop faible ou trop élevé entre deux invocations successives peut révéler une anomalie dans le comportement.

Notre travail est organisé en trois étapes. Dans la première étape, nous nous intéressons au comportement des clients d'une application CORBA que nous modélisons suivant l'approche présentée ci dessus. Dans une deuxième phase, nous proposons de modéliser le comportement des objets répartis au vu des requêtes reçues de leurs clients. Nous gardons toujours la même approche utilisée moyennant quelques modifications dans le modèle. Enfin, nous proposons de tenir compte d'éventuelles corrélations entre les valeurs des paramètres d'une même requête en modélisant leur répartition.

Chapitre 2

Modélisation du comportement du client

2.1 Introduction

La première étape de notre travail a consisté à modéliser le comportement des clients d'objets répartis, en prenant l'exemple d'une application CORBA. Nous nous intéressons au comportement des clients en considérant, pendant une phase d'apprentissage, la suite de requêtes invoquées entre chaque couple client-serveur. Nous prenons en compte les requêtes invoquées entre chaque connexion et déconnexion du client. Cela nous permet de construire le modèle de comportements de chaque client sous la forme d'un ensemble de chemins de taille variable au sein d'une structure arborescente. Le modèle tient compte, d'une part de l'enchaînement des requêtes, d'autre part des valeurs de leurs paramètres. Pour rendre compte de la dispersion des valeurs des paramètres, nous construisons, à l'issue de la phase d'apprentissage des intervalles de tolérance pour les paramètres numériques. Ces intervalles permettent, en phase de détection, la mesure de la déviation des valeurs observées par rapport aux valeurs de référence apprises.

Durant la phase de détection, l'arbre de chaque client est parcouru à partir de la racine en calculant un degré de similarité, en fonction des intervalles de tolérance construits. Le comportement observé est considéré valide si on atteint une feuille lors de la déconnexion avec un degré de similarité acceptable. Dans le cas contraire, on déclenche une alerte.

La mise en œuvre de notre approche s'est déroulée sur deux phases. Initialement, ne disposant pas d'application réelle, nous avons développé une maquette d'une application bancaire très simplifiée permettant de tester l'ap-

proche proposée. La maquette ne constitue pas une application CORBA réelle avec de vrais utilisateurs, mais elle nous a permis de mettre en œuvre l'ensemble de notre approche (interception, apprentissage, détection). Dans une deuxième phase, nous avons pu disposer d'un fichier d'audit traçant l'activité réelle d'une application en environnement réparti. Ce fichier a été le point d'entrée pour une nouvelle série de tests.

Ce chapitre est organisé comme suit :

- Le paragraphe 2.2 détaille la phase d'apprentissage, la construction de la base de comportements et des intervalles de tolérance.
- Le paragraphe 2.3 présente l'algorithme de détection. Ces deux premiers paragraphes présentent les concepts généraux applicables à un système à objets répartis.
- Le paragraphe 2.4 détaille l'application de notre approche sur la maquette CORBA développée : architecture de la maquette, collecte d'informations, format des informations collectées et résultats de la détection.
- Le paragraphe 2.5 présente les résultats obtenus par l'application de notre approche sur des données d'audit réelles.

2.2 Phase d'apprentissage

La phase d'apprentissage constitue la période d'activité pendant laquelle les clients des objets servants sont observés afin de construire leurs modèles de comportements (voir figure 2.1). Durant cette phase, totalement transparente pour les clients, les comportements appris sont considérés exempts d'attaque.

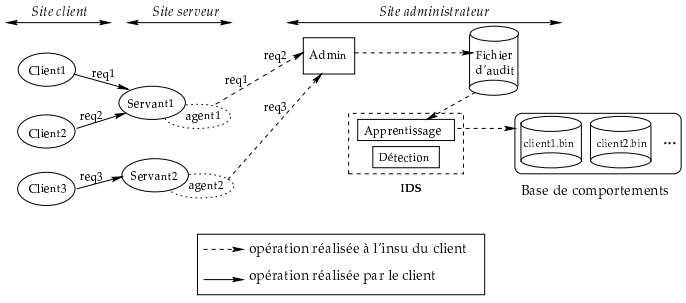

FIG. 2.1 – *Phase d'apprentissage*

L'architecture présentée dans la figure 2.1 met en évidence les aspects suivants :

- La collecte d'informations se fait de manière transparente pour les clients des objets servants.

- La collecte d'informations se fait de manière répartie sur plusieurs agents, installés sur chaque objet servant, mais leur analyse se fait de manière centralisée sur le site de l'administrateur.
- A la demande de l'administrateur, l'IDS reçoit une copie du fichier d'audit et construit autant de modèles de comportements qu'il y a d'utilisateurs dans le fichier d'audit.

Pour la modélisation du comportement, nous prenons en compte la suite des requêtes invoquées entre chaque connexion et déconnexion et, pour chaque requête, les valeurs prises par leurs paramètres. Nous pensons que cette prise en compte est essentielle pour définir un modèle réaliste du comportement normal. Nous construisons, dans un second temps, des intervalles de tolérance autour des valeurs observées afin d'offrir aux clients une certaine liberté dans la déviation par rapport à leurs comportements de référence. Nous détaillons dans le paragraphe 2.2.1 la construction de la base de comportements et dans le paragraphe 2.2.2 la construction des intervalles de tolérance.

2.2.1 Construction de la base de comportements

Nous modélisons les comportements des clients par une structure d'arbre. La racine de l'arbre marque une connexion, une feuille marque une déconnexion. Un nœud correspond à une requête et à ses paramètres. Il est composé des éléments suivants (illustrés dans la figure 2.2) :

- le nom de l'opération invoquée,
- la liste des paramètres associés à l'opération. Chaque paramètre de la liste est composé d'un nom, d'un type, d'une valeur et d'un nombre d'occurences (c'est le nombre de fois où cette valeur a été observée).
- la liste des intervalles de tolérance associés aux paramètres numériques de l'opération. Pour simplifier la mise en œuvre, nous associons à chaque paramètre numérique un seul intervalle de tolérance. La construction de ces intervalles est détaillée dans le paragraphe 2.2.2.
- le nombre d'occurences de l'opération : c'est le nombre de fois où l'opération a été invoquée.

Un chemin de l'arbre décrit donc un comportement observé entre une connexion et une déconnexion (voir Figure 2.3). L'arbre ainsi construit comprend des chemins de tailles variables. L'ensemble des arbres constitue la base de comportements.
La construction de l'arbre d'un client se fait par ajout de chemins. Si l'on observe N comportements successifs déjà modélisés, on estime que l'arbre a

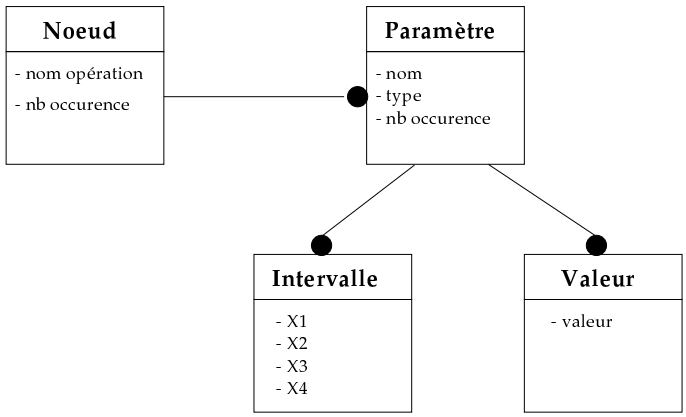

FIG. 2.2 – *Structure des nœuds*

atteint un état stable. La durée de la phase d'apprentissage et la valeur de N dépendent de la complexité de l'application : plus l'application est riche, plus l'apprentissage est long. Pour une application réelle plusieurs semaines sont vraisemblablement nécessaires.

Nous donnons dans la figure 2.4 l'algorithme principal d'apprentissage pour les clients de l'application. A partir du fichier d'audit **auditfile** (traçant l'activité de l'ensemble des clients), l'algorithme construit pour chaque client (**user**) un modèle de comportement de référence appelé **profil**. L'ensemble des profils sont sauvegardés dans des fichiers binaires **login_user.bin**. Le principe de l'algorithme d'apprentissage est le suivant :

- lire ligne par ligne le fichier d'audit contenant les requêtes des clients aux objets servants durant la période d'apprentissage. Créer pour chaque ligne un nœud et l'ajouter à l'arbre du client correspondant. La racine de l'arbre marque une connexion, une feuille marque une déconnexion. Un nœud correspond à une requête et à ses paramètres.
- Lors de l'ajout d'un nouveau nœud dans l'arbre, deux cas se présentent (voir algorithme de la figure 2.5) :
  1. Le nouveau nœud existe déjà dans l'arbre, c'est-à-dire que le nouveau nœud porte le même nom d'opération que le nœud courant. Il s'agit dans ce cas de «fusionner» les deux nœuds en intégrant les informations portées par le nouveau nœud dans celles du nœud courant. L'intégration se fait au niveau des paramètres observés ; deux cas se présentent pour chaque paramètre de l'opération :
     (a) si la valeur observée du paramètre est la même dans le nouveau nœud et dans le nœud courant, on incrémente juste le

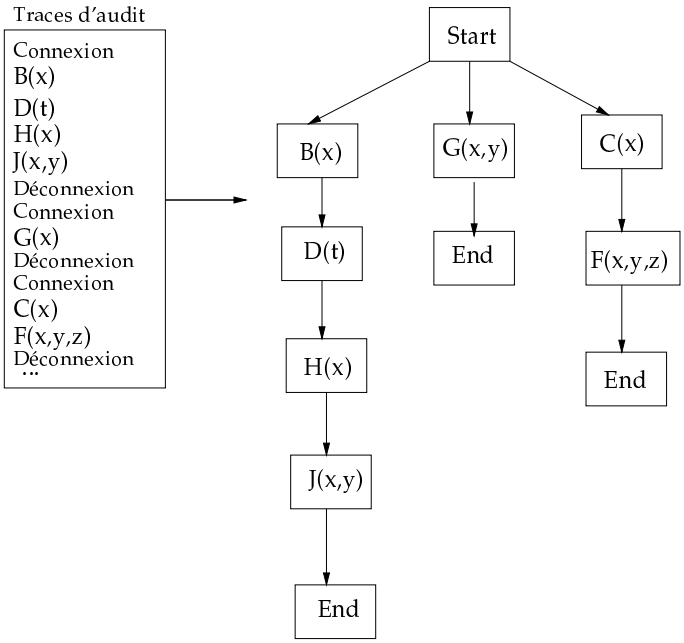

FIG. 2.3 – *Construction de la base de comportements*

nombre d'occurences du paramètre dans le nœud courant ;

(b) si la valeur observée du paramètre est différente de celles dans le nœud courant, elle est ajoutée à la liste des valeurs déjà observée : il s'agit dans ce cas d'apprendre une nouvelle valeur pour ce paramètre.

2. Le nouveau nœud n'existe pas, c'est-à-dire qu'il ne désigne pas la même opération que le nœud courant, il est alors ajouté comme «*frère*» du nœud courant, au même niveau dans l'arbre[1].

2.2.2 Construction des intervalles de tolérance

Comme déjà mentionné, dans la modélisation du comportement, nous prenons en compte les paramètres des requêtes afin d'affiner la définition des comportements normaux. En effet, un écart important de la valeur d'un paramètre peut révéler une anomalie. Pour autant, une répartition raisonnable autour de la valeur apprise doit être acceptée. A cet effet, nous construisons des intervalles de tolérance pour exprimer ces variations. Leur intérêt est

1. Par exemple, dans la figure 2.3 les nœuds portant les opérations B(x), G(x,y) et C(x) sont des «*frères*».

```
Learn_users(auditfile : in)
BEGIN
open(auditfile)
read(auditfile : in, auditline : out)
user ← get_user_name(auditline : in)
action ← get_user_action(auditline : in)
while (user ≠ null) do
  if (user ∉ users) then
    profil ← create_profil(user : in)
  end if
  addUserAction(profil : in, action : in, users : inout)
  read(auditfile : in, auditline : out)
  user ← get_user_name(auditline : in)
  action ← get_user_action(auditline : in)
end while
i ← 0
for all users[i] do
  nom ← get_user_name(users[i] : in)
  profil ← get_user_profil(users[i] : in)
  affecte_intervalle(nom : in, profil : inout)
  profilName ← nom.bin
  save_profil(profil : in, profilName : in)
end for
close(auditfile)

END
```

FIG. 2.4 – *Algorithme d'apprentissage du comportements des clients. La fonction* `addUserAction` *est détaillée dans la figure 2.5.*

de permettre à l'IDS d'accepter des valeurs proches de celles apprises, bien qu'elles n'aient jamais été observées. Les intervalles sont construits à la fin de la création des arbres. Ils sont construits uniquement pour les paramètres numériques des requêtes dont les valeurs peuvent dévier de celles apprises avec un certain degré d'acceptabilité[2]. Pour les paramètres symboliques tel que le nom du client connecté, on n'accepte que les valeurs qui ont déjà été observées : en effet, accepter des variations sur des données symboliques n'a pas de sens[3]. L'analyse consiste d'abord à répertorier les valeurs apprises d'un même paramètre et leurs nombres d'occurences respectifs et les représenter par des nuages de points (éventuellement réduits à des valeurs isolées). Nous associons ensuite à chaque nuage un intervalle de tolérance comme le montre

2. Il revient à l'administrateur de l'IDS de désigner les paramètres n'admettant aucune déviation. Par exemple : le numéro de compte bancaire (par opposition au montant d'un retrait).

3. L'administrateur peut marquer certains paramètres symboliques à ne pas considérer dans le modèle (exemple : texte libre, etc.).

```
addUserAction(profil: in, action: in, users: inout)
BEGIN
trouve ← false
tree ← get_current_node(profil : in)
fils ← get_children(tree)
inti ← 0
taille ← get_length(fils)
while (¬trouve and i < taille) do
  actionName ← get_action_name(action : in)
  actionParams ← get_action_Params(action : in)
  filsName ← get_action_name(fils[i] : in)
  filsParams ← get_action_Params(fils[i] : in)
  if (filsName == actionName) then
    trouve ← true
    filsParams ← merge(filsParams : inout, actionParams : in)
    tree ← fils[i]
  end if
end while
if (¬trouve) then
  tree ← addAction(tree : inout, action : in)
  tree ← action
end if
if (action == disconnect) then
  tree ← root
end if
END
```

FIG. 2.5 – *Fonction* `addUserAction` *: ajout d'un noœud à l'arbre*

la figure 2.6.

La difficulté dans cette étape est de déterminer un seuil à partir duquel la valeur observée d'un paramètre doit être considérée comme trop différente de celles apprises. Le problème revient à définir une mesure de l'écart entre ce qui est appris et ce qui est observé. Les intervalles de tolérance permettent d'exprimer l'écart autorisé autour des valeurs apprises en fonction de l'étendue et de la densité des nuages de points correspondants. Ainsi on autorise un écart plus grand pour les valeurs qui ont été souvent observées et un écart moins important pour les valeurs rares. L'écart dépend aussi de la sensibilité du paramètre vis-à-vis de la sécurité de l'application CORBA.

Construire un intervalle de tolérance revient donc à trouver les valeurs du quadruplet $[x_1,x_2,x_3,x_4]$ qui expriment l'écart autorisé dans l'intervalle. Faute d'application réelle, nous avons proposé un algorithme simpliste de construction des intervalles de tolérance qui tient compte seulement de l'étendue et de

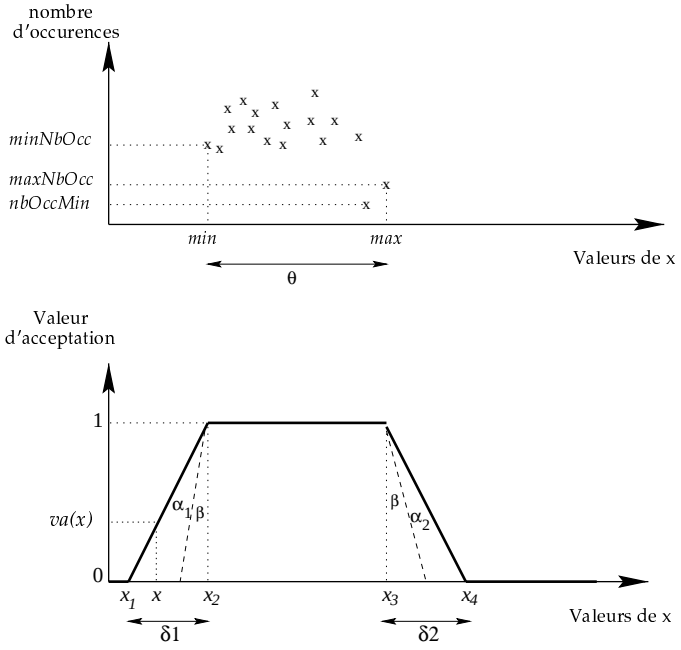

FIG. 2.6 – *Construction de l'intervalle de tolérance. Les points x_2 et x_3 délimitent le nuage des valeurs observées. Les points x_1 et x_4 expriment l'écart autorisé autour de x_2 et x_3. $va(x)$ est la fonction d'acceptation trapézoïdale sur l'intervalle de tolérance.*

la densité du nuage. Nous proposons dans le chapitre 4 une amélioration de cet algorithme. Nous donnons dans la figure 2.7 les différentes étapes de l'algorithme de construction des intervalles de tolérance pour chaque paramètre numérique considéré dans une requête. L'algorithme permet de calculer les écarts δ_1 et δ_2 respectivement autorisés autour des valeurs minimale x_2 et maximale x_3 observées.

Un intervalle de tolérance s'exprime par une fonction trapézoïdale va qui associe à chaque valeur de x une valeur d'acceptation $va(x)$ comprise entre 0 et 1 qui mesure le degré de similarité entre ce qui a été appris et ce qui est observé en phase de détection. Le principe consiste à considérer toutes les valeurs apprises (comprises entre x_2 et x_3) et à leur associer une valeur d'acceptation égale à 1 (voir Figure 2.6). Les valeurs observées autour des valeurs normales ($[x_1,x_2[$ et $]x_3,x_4]$) dans l'intervalle de tolérance sont acceptées mais ont une valeur d'acceptation comprise entre 0 et 1. Cette valeur augmente en s'approchant des limites d'un trapèze (x_2 et x_3) et diminuent en s'éloignant de ces valeurs. Les valeurs observées en dehors de ces intervalles

sont considérées anormales et leur valeur d'acceptation est nulle.

La fonction va est définie comme suit :

$$va(x) = \begin{cases} 0 & \text{pour } x < x_1 \text{ ou } x > x_4 \\ 1 & \text{pour } x \in [x_2,x_3] \\ \frac{x-x_1}{x_2-x_1} & \text{pour } x \in [x_1,x_2[\\ \frac{x_4-x}{x_4-x_3} & \text{pour } x \in]x_3,x_4] \end{cases} \tag{2.1}$$

La figure 2.6 donne une représentation de l'intervalle de tolérance. Cette représentation est utilisée lors de la détection afin de calculer le degré de similarité de chaque paramètre numérique. L'intervalle ainsi construit permet, comme nous venons de l'expliquer, de donner une marge au client en lui autorisant un certain écart par rapport à son comportement appris. De ce fait, il permet de limiter le taux de fausses alertes durant la détection grâce au calcul d'un degré de similarité. Cette notion est définie dans le paragraphe 2.3.2.

affecte_intervalle(nom: in, profil: inout)
BEGIN
$min \leftarrow$ valeur minimale observée pour un paramètre durant l'apprentissage
$minNbOcc \leftarrow$ nombre d'occurences de min
$max \leftarrow$ valeur maximale observée pour un paramètre durant l'apprentissage
$maxNbOcc \leftarrow$ nombre d'occurences de max
$nbOccMin \leftarrow$ nombre minimum d'occurences observé sur toutes les valeurs apprises
$\theta \leftarrow max - min$
$\alpha_1 \leftarrow$ angle mesurant l'écart autorisé autour de min
$\alpha_2 \leftarrow$ angle mesurant l'écart autorisé autour de max
$\beta \leftarrow$ angle mesurant l'écart autorisé autour de min et max (en fonction de θ)
if $(minNbOcc == nbOccMin)$ **then**
  $\alpha_1 = \Pi/4$
else
  $\alpha_1 = 3\Pi/8$
end if
if $(maxNbOcc == nbOccMin)$ **then**
  $\alpha_2 = \Pi/4$
else
  $\alpha_2 = 3\Pi/8$
end if
if $(\theta > 3 * x_2)$ **then**
  $\beta = \Pi/16$
else
  $\beta = 0$
end if
$x_2 = min$
$x_3 = max$
$\delta_1 = \tan(\alpha_1 + \beta)$
$\delta 2 = \tan(\alpha_2 + \beta)$
$x_1 = x_2 - \delta_1$
$x_4 = x_3 + \delta_2$
END

FIG. 2.7 – *Algorithme de construction des intervalles de tolérance*

2.3 Phase de détection

Durant la phase de détection, le client est observé entre chaque connexion et déconnexion. L'objectif n'est plus d'apprendre son comportement mais de décider de la légitimité de celui-ci, en calculant un degré de similarité d_i entre ce qui est observé durant une connexion i et ce qui a été appris. L'architecture de détection est illustrée dans la figure 2.8. Nous donnons dans le paragraphe 2.3.1 le principe de l'algorithme de détection. Le paragraphe 2.3.3 présente les deux types d'alertes générées par l'algorithme. Nous détaillons dans le paragraphe 2.3.2 le calcul du degré de similarité.

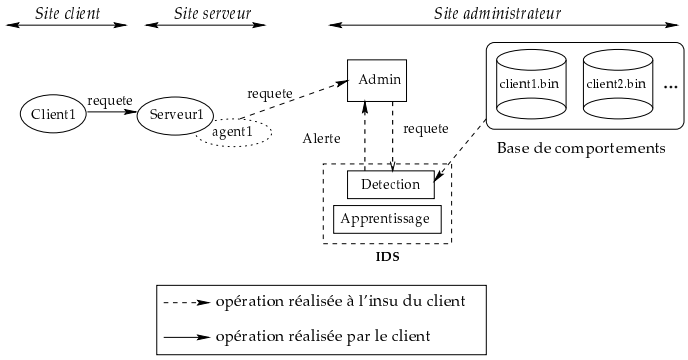

FIG. 2.8 – *Phase de détection*

2.3.1 Algorithme de détection

L'algorithme de détection permet de détecter en temps réel toute déviation par rapport au comportement appris de l'utilisateur. Nous considérons un comportement normal durant une connexion comme une suite de requêtes invoquées avec des valeurs apprises de leurs paramètres, jugées normales. L'algorithme consiste à parcourir l'arbre en ajustant, pour chaque nœud exploré et jusqu'à parvenir à une feuille, le degré de similarité. Le degré ainsi calculé permet, dans une première phase, de déduire si le comportement observé à l'instant t pour la connexion i du client est normal ou intrusif. Si le degré est inférieur à un premier seuil d'acceptabilité, appelé *seuil instantané* s, une *alerte instantanée* est déclenchée (voir Figure 2.9).

Il est aussi intéressant d'étudier le comportement du client durant une période plus longue regroupant plusieurs connexions. Ainsi, au cours de plusieurs connexions consécutives, on observe les différents d_i obtenus afin de fournir une deuxième alerte, appelée *alerte composée*. Cette alerte met en

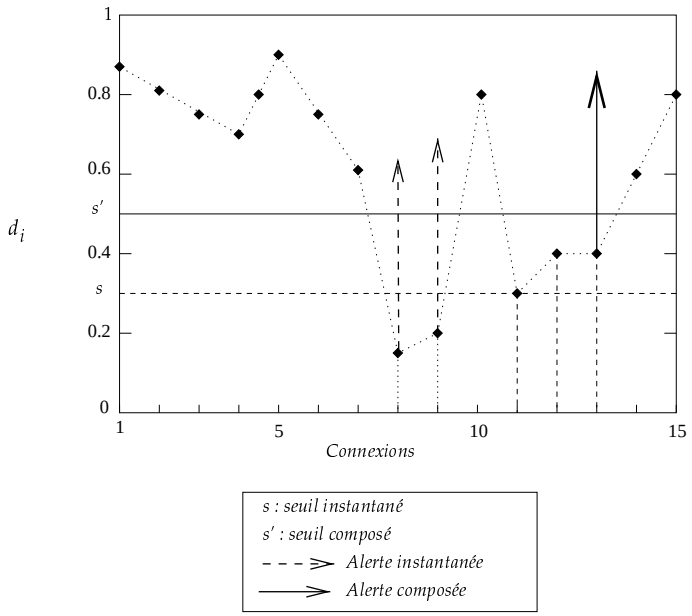

FIG. 2.9 – *Principes de la génération d'alertes*

évidence les corrélations possibles entre plusieurs connexions pouvant constituer une attaque. Elle est déclenchée lorsque d_i reste inférieur à un deuxième seuil, appelé *seuil composé* s' pendant un nombre de fois $nbComp$ supérieur à une limite S'.

L'algorithme de détection est donné dans la figure 2.10. Le principe de l'algorithme est le suivant :

- l'IDS reçoit chaque ligne d'audit qui lui est envoyée en temps réel par l'administrateur,
- Pour chaque ligne, l'IDS cherche l'utilisateur invoquant la requête et charge en mémoire son arbre. Trois cas sont possibles:
  1. Le client n'existe pas dans la base de comportements (il n'existe pas de fichier portant son login). Dans ce cas, un message est transmis à l'administrateur de l'application pour l'aviser.
  2. Le client vient de se connecter. L'IDS charge alors son arbre en mémoire (à partir de son fichier `toto.bin`) et se positionne à la racine de l'arbre. On commence à la racine par une valeur du degré de similarité égale à 1 qui est mise à jour à chaque nœud visité.
  3. Le client s'est déjà connecté, il suffit de se positionner sur la dernière opération réalisée par le client (qui correspond au nœud

courant dans l'arbre du client).

- Une fois le client localisé, un nouveau nœud est créé à partir de la ligne d'audit reçue pour le comparer aux fils du nœud courant.
- L'IDS parcourt tous les nœuds fils du nœud courant de l'arbre (c'est-à-dire les nœud potentiellement prévus) à la recherche d'un nœud portant le même nom d'opération que le nouveau nœud. Cette étape est décrite dans l'algorithme donné en figure 2.11. A chaque fils visité, deux cas sont possibles :
  1. Si le nouveau nœud et le nœud fils visité portent le même nom d'opération, on considère que le client a invoqué une opération prévue dans le chemin de l'arbre. On calcule dans ce cas la pénalité accordée à ce nœud (en utilisant les intervalles de tolérance correspondants aux paramètres de cette opération) et on met à jour la valeur du degré de similarité : une valeur d'acceptation nulle exprime un comportement intrusif donc une pénalité importante : la valeur du degré de similarité diminue considérablement. Une valeur d'acceptation proche de 1 exprime un comportement jugé normal d'où une pénalité faible et un degré de similarité faiblement affecté.
  2. Si le nouveau nœud ne correspond à nœud fils prévu, on considère que le comportement est intrusif et on diminue le degré de similarité de δ.
- en fonction de la valeur obtenue pour le degré de similarité et des seuils d'acceptabilité, l'algorithme de détection décide du déclenchement d'une alerte.

2.3.2 Calcul du degré de similarité

Le comportement observé est jugé anormal si son degré de similarité (un degré faible traduit une correspondance faible entre ce qui observé et ce qui a été appris) est inférieur à un certain seuil, appelé *seuil instantané «s»*. Plus le seuil est faible, plus on accepte des comportements déviants. En augmentant le seuil, on limite le nombre de comportements admis, mais on augmente le taux de fausses alertes. Le problème reste toujours de trouver un bon compromis entre une «bonne» détection et un taux de fausses alertes acceptable. Ce compromis est trouvé expérimentalement (voir paragraphe 2.4.5).

Le calcul du degré de similarité d_i pour la connexion i commence à la racine de l'arbre avec $d_i^0 = 1$. Puis, le degré est ajusté à chaque nœud exploré

par application d'une éventuelle pénalité fonction des n valeurs d'acceptation va_j (cf. définition au paragraphe 2.2.2) des paramètres de l'opération associée au nœud. L'expression de la valeur de d_i au nœud k (d_i^k), en fonction de sa valeur au nœud précédent (d_i^{k-1}), est la suivante :

$$d_i^k = d_i^{k-1} - P(va_1..va_n) \tag{2.2}$$

$P(va_1..va_n)$ exprime la pénalité globale affectée au nœud, c'est-à-dire à l'opération en considérant ses n paramètres. Elle est issue de la conjonction de pénalités élémentaires $p_j(va_j)$ calculées pour chaque paramètre de l'opération. Cela permet à toutes les valeurs élémentaires de contribuer au calcul de la pénalité globale P de l'opération. Il faut donc choisir un opérateur de conjonction parmi les trois les plus courants : le *min*, le *max* et la *moyenne*. Le *min* calcule le minimum des $p_j(va_j)$. L'inconvénient majeur du *min* est sa forte sensibilité aux valeurs nulles, ce qui se traduit par une pénalité globale P nulle au niveau de l'opération. Il permet ainsi d'accepter à tort certaines opérations. Par opposition au *min*, l'opérateur *max* applique à l'opération la plus grande pénalité observée dans les paramètres en prenant le maximum de toutes les p_j, ce qui est assez restrictif. Nous avons donc opté pour l'application de l'opérateur *moyenne*. Contrairement aux deux autres, cet opérateur a un effet de compensation d'où une faible sensibilité aux valeurs nulles. La pénalité affectée à l'opération est donc exprimée en fonction des p_j comme suit :

$$P(va_1..va_n) = \frac{\sum_{j=1}^{n} p(va_j)}{n} \tag{2.3}$$

La pénalité élémentaire p_j du paramètre j est fonction de la valeur d'acceptation va_j de ce paramètre. Pour les paramètres symboliques, une simple comparaison de la valeur observée du paramètre avec celles apprises nous donne une pénalité nulle ou maximale suivant que les valeurs sont identiques ou différentes. Pour les paramètres numériques, la pénalité élémentaire s'exprime par la formule suivante :

$$p(va) = (1 - va)^o \tag{2.4}$$

Cette pénalité doit décroître lorsque va_j augmente, partant du point (va_j,p_j) = (0,1) pour arriver au point (1,0) comme le montre la Figure 2.12. Si la valeur observée d'un paramètre numérique n'appartient pas à l'intervalle de valeurs préalablement défini, sa valeur d'acceptation va_j est faible. Par conséquent, la pénalité élémentaire associée p_j est grande et le degré d_i diminue considérablement. Plus la valeur d'acceptabilité augmente, moins la pénalité est importante et moins le degré de similarité est diminué.

Cette fonction permet de contrôler la pente de la courbe en variant le degré o. Un degré o élevé exprime l'acceptation d'une plus large variation du paramètre associé. A titre d'illustration, prenons l'exemple présenté dans la figure 2.13. Considérons une requête $F(x)$ invoquée par le client avec $x = 140$. Sa valeur d'acceptation est alors égale à 0.4 (voir figure 2.13). Le tableau 2.1 montre la variation du degré de similarité en fonction de la pénalité. Nos premiers tests sont basés sur une fonction de pénalité quadratique ($o = 2$) pour toutes les opérations invoquées. Nous proposons, pour nos travaux futurs, de faire varier le degré o pour chaque paramètre en fonction de sa sensibilité par rapport à l'application.

o	$p(va_j) = (1 - 0.4)^o$	$d^1 = d^0 - P(va_1..va_n)$
2	0.36	0.64
4	0.1296	0.8704
8	0.0167	0.9832

TAB. 2.1 – *Variation de d_i en fonction de $p_j(va_j)$*

2.3.3 Génération d'alertes

L'algorithme de détection nous fournit deux types d'alertes.

Tout d'abord, au cours de l'exploration d'un chemin de l'arbre, le degré d_i peut diminuer car le client a cumulé un certain nombre de requêtes suspectes (imprévues ou prévues avec des valeurs anormales) provoquant la chute de ce degré. Lorsque le degré diminue considérablement en dessous d'un seuil d'acceptabilité s, une *alerte instantanée* est alors émise.

Par ailleurs, à chaque déconnexion d'un client, on effectue un test qui peut aussi déclencher une *alerte instantanée*. Le test consiste à vérifier si on est bien arrivé à une feuille de l'arbre de comportements. Dans ce cas, si la valeur de d_i calculée au cours de cette connexion i est supérieure au seuil s, on n'émet pas d'alerte. Dans le cas contraire, on émet une *alerte instantanée*. Dans le cas où la déconnexion du client ne passe pas par une feuille, on considère que ce comportement est intrusif et on déclenche également une *alerte instantanée*.

Dans tous les cas, on continue à observer le comportement du client au cours des connexions suivantes. L'objectif est de pouvoir détecter des suites de connexions anormales pouvant révéler une anomalie engendrée par de petites déviations consécutives et étalées dans le temps. Ainsi, à chaque déconnexion du client, si le le degré d_i obtenu pour la connexion i est inférieur à un deuxième seuil s', on incrémente le nombre de connexions suspectes *nbComp*.

Une *alerte composée* est déclenchée si le degré de similarité d_i reste inférieur au seuil composé s' pendant un nombre de connexions suspectes $nbComp$ supérieur au nombre maximum autorisé de suites de connexions suspectes S'.
Nous résumons ci-après la décroissance du degré de similarité et la génération d'alertes en fonction de sa valeur obtenue :

1. Initialiser pour chaque connexion i le degré de similarité à 1 à la racine de l'arbre : $d_i^0 = 1$
2. Pour chaque nœud parcouru :
   - Si l'opération invoquée par le client est imprévue dans le chemin déjà entamé dans l'arbre, alors le degré de similarité est affecté au nœud k : $d_i^k = d_i^{k-1} - \delta$ (δ étant préalablement fixée)
   - Sinon ajuster la valeur du degré de similarité en fonction des pénalités associées à chaque paramètre de l'opération $d_i^k = d_i^{k-1} - P(va_1..va_n)$
3. Si $d_k < s$ alors générer une alerte instantanée
4. Si déconnexion :
   - Si $d_k < s'$ alors $nbComp = nbComp + 1$
   - Si $nbComp > S'$ alors générer une alerte composée

La Figure 2.9 montre l'évolution de d_i au cours de plusieurs connexions consécutives et met en évidence les deux types d'alerte. Pour cet exemple, nous supposons que $s = 0.3$, $s' = 0.5$ et $S' = 3$. On remarque que pour les connexions 8 et 9 le degré d_i est inférieur à s, ce qui déclenche à chaque déconnexion une *alerte instantanée* mais pas d'*alerte composée* puisque d_i est resté 2 fois inférieur à s'. Pour les connexions 11,12,13 le degré d_i reste inférieur à s' ce qui déclenche une *alerte composée* puisque le nombre maximum autorisé est $S' = 3$. En revanche, pour ces mêmes connexions, il n'y a pas d'alerte instantanée puisque d_i reste supérieur à s, ce qui illustre bien le fait que de petites variations dans le comportement d'un client peuvent être instantanément considérées acceptables (d'où l'absence d'alerte instantanée) mais peuvent révéler au bout d'un certain temps une anomalie.

Faute d'application réelle, aucune stratégie de réaction n'est prévue face à cette situation pour le moment. Il faut en effet étudier dans le futur, pour une vraie application CORBA, les différentes possibilités de réactions face à un comportement intrusif détecté.

```
Detect_users(ProfilBase: in, SERVERPORT: in)
S ← 0.3
S' ← 0.5
maxAlertes ← 2
nbAlertesComp ← 0
intrus ← true
result ← false
BEGIN
action ← read_action(SERVERPORT : in)
user ← get_user(action_user : in)
if (user ∈ ProfilBase) then
  user_profil ← get_Profil(ProfilBase : in, user : in)
else
  Print ( «Client inconnu» )
end if
while ((action_user ≠ null) and (result ≠ intrus)) do
  result ← detect(action_user, profil)
  degre ← get_user_degre(user_profil : in)
  if (degre < S) then
    result ← true
    nbAlertesComp + +
  end if
  if (action_user ≠ "DISCONNECT") then
    if (degre ≥ S) then
      Print ( «Comportement normal» )
    end if
    if (result == intrus) then
      Print ( «Alerte instantanée : Comportement Anormal ! » )
      gotoRoot
    end if
  else if (nbAlertesComp ≥ maxAlertes) then
    Print ( «Alerte composée !!» )
  end if
  action ← read_action(SERVERPORT : in)
  user ← get_user(action_user : in)
  if (user ∈ ProfilBase) then
    user_profil ← get_Profil(ProfilBase : in, user : in)
  end if
end while
END
```

FIG. 2.10 – *Algorithme de détection. La fonction* `detect` *est détaillée dans la figure 2.11.*

```
detect(action_user: in, profil: in)
trouve ← false
intrus ← false
degra ← 1
λ ← penalité associé au degré de similarité si l'opération n'est pas trouvée
p ← 0 (pénalité affectée à l'opération)
while ((¬trouve) and (¬intrus)) do
  fils ← get_next_fils(profil)
  if (action_user = current) then
    trouve ← true
    p ← get_penalite(action_user : in, fils : in)
    degre ← degre − p
    current ← fils
  end if
end while
if (¬trouve) then
  intrus ← true
  degre ← degre − λ
end if
return(intrus)
END
```

FIG. 2.11 – *Algorithme de parcours de l'arbre de comportement et détection d'anomalies*

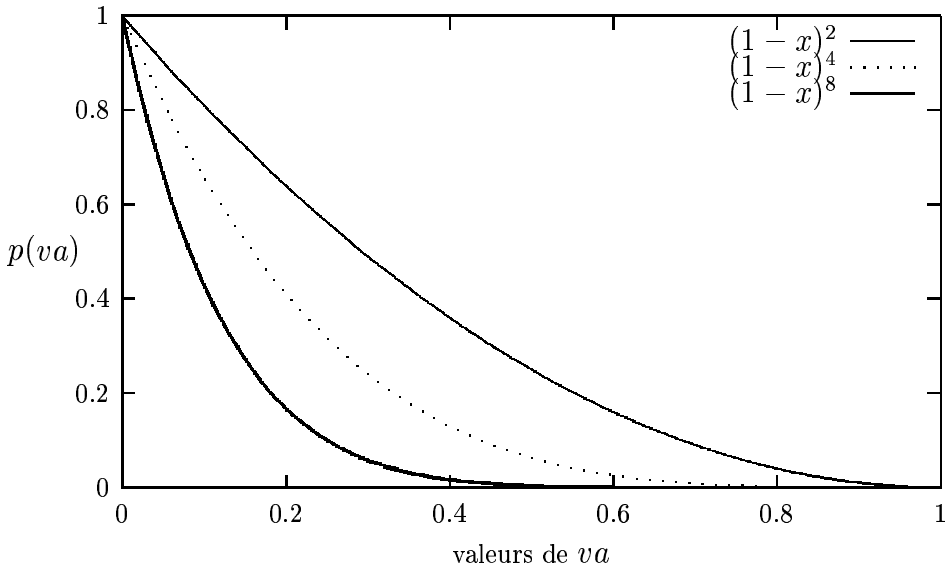

FIG. 2.12 – *Fonction de pénalité pour la valeur d'acceptation*

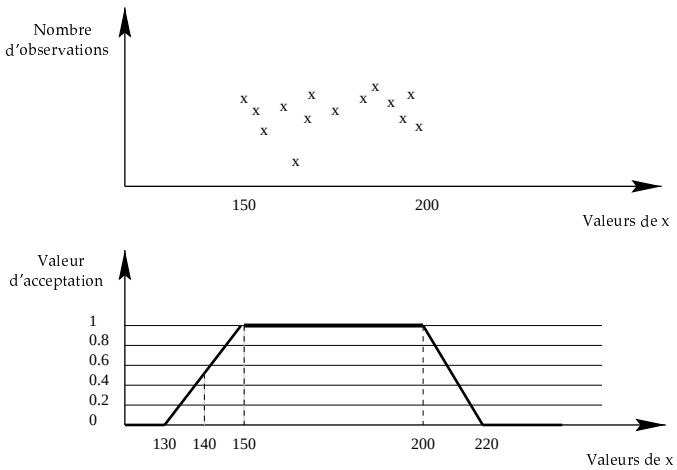

FIG. 2.13 – *Représentation des intervalles de tolérance pour les nuages de points de x*

On s'intéresse, par exemple, aux valeurs prises par le paramètre x de la requête $F(x)$. Les valeurs observées pour x durant la phase d'apprentissage sont comprises entre 150 et 200. En phase de détection, une valeur observée égale à 148 ne doit pas être considérée anormale vue qu'elle est très proche des valeurs habituellement observées ([150,200]).

2.4 Mise en œuvre de l'approche

La première étape de réalisation a consisté à mettre en œuvre notre approche sur une maquette CORBA simplifiée modélisant une application bancaire. L'architecture retenue pour la maquette comprend quatre principaux acteurs (illustrés dans la Figure 2.14) :

- un *client* bancaire ;
- un *serveur* : dans notre cas il s'agit du serveur d'une application bancaire triviale, mais on peut lui substituer n'importe quel autre serveur ;
- un *agent* d'interception se greffant automatiquement sur un serveur pour espionner ses communications avec les clients ;
- un *Administrateur* qui attend les inscriptions des agents, les télécommande et recueille leur traces. L'administrateur et les agents communiquent par l'ORB et peuvent donc résider dans des machines différentes.

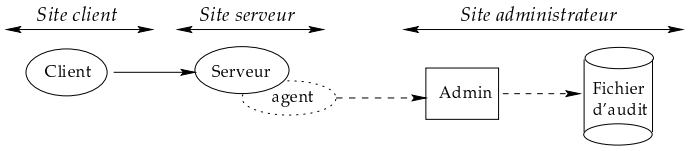

FIG. 2.14 – *Architecture de base*

Cette architecture a été mise en place en utilisant l'ORB Visibroker (Version 3.3 pour Java)[4]. Nous avons choisi cet ORB car il propose, en plus des intercepteurs, d'autres mécanismes de collecte d'informations permettant d'enrichir notre modèle de comportement. Néanmoins, ces mécanismes (appelés *EventHandler*) sont spécifiques à l'ORB Visibroker. Nous détaillons dans le paragraphe 2.4.1 ces deux types de mécanismes d'interception. Nous donnons dans le paragraphe 2.4.2 notre choix de données discriminantes pour la définition du comportement normal. Enfin, le paragraphe 2.4.3 détaille le format choisi pour représenter les données dans le fichier d'audit.

2.4.1 Collecte d'informations sur les interactions clients-serveurs

Basée sur l'architecture décrite dans la Figure 2.14, la mise en place de la surveillance d'un client se fait de la manière suivante (voir Figure 2.15) :

1. lancement de l'administrateur qui se met à l'écoute des inscriptions des

4. http://www.borland.com/bes/visibroker

agents ;

2. lancement du serveur qui initialise l'ORB et déclenche, à son insu, le chargement de l'agent par une option de la ligne de commande ;
3. l'agent initialise les mécanismes d'interception et s'enregistre auprès de l'administrateur ;
4. tant que l'administrateur n'a pas demandé de trace, les opérations du serveur ne donnent lieu à aucune capture ;
5. lorsque l'administrateur demande l'interception, l'ORB communique des informations à l'agent qui les transmet à l'administrateur.

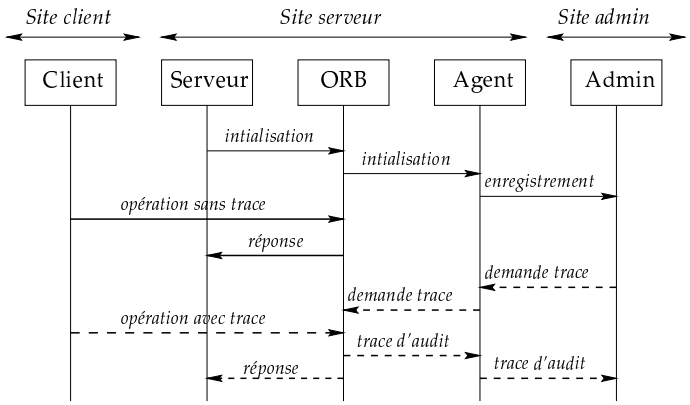

FIG. 2.15 – *Architecture et principe des échanges*

L'agent de capture utilise deux types de mécanismes d'interception : les intercepteurs standards de CORBA et les *EventHandler* spécifiques à Visibroker. Nous avons utilisé ces deux mécanismes que nous détaillons dans le paragraphe 2.4.1.1 et 2.4.1.2.

2.4.1.1 Les intercepteurs

Nous rappelons que notre approche vise à protéger les objets d'une application CORBA des comportements potentiellement malicieux de ses clients. Nous souhaitons observer les clients, de manière transparente à ceux-ci. Nous proposons donc d'implanter des intercepteurs, de niveau requête, côté serveur uniquement afin de récupérer les requêtes émises par les clients aux objets servants. Visibroker propose deux types d'intercepteurs de niveau requête :

– `ServerInterceptors` : implantés côté serveur, ils sont capables de récupérer des informations sur le serveur, en plus de l'interception proprement dite.

- `ClientInterceptors` : implantés côté client, ils sont capables de récupérer des informations sur le client, en plus de l'interception.

Nous avons utilisé le premier type d'intercepteurs qui nous a permis de collecter des informations sur :

- le nom de la machine sur laquelle tourne le serveur,
- le nom du serveur,
- le login sous lequel le serveur a été lancé,
- le principal. Dans notre maquette, il s'agit du login sous lequel le client est connecté. Contrairement aux autres informations, le login du client est une information volontairement donnée par le client grâce à une requête à l'ORB.
- le nom de l'objet cible invoqué,
- le nom de l'opération invoquée,
- les paramètres de la requête,
- le statut de la réponse.

Le principal avantage des intercepteurs est leur capacité à tracer et modifier les paramètres des opérations. Néanmoins, ils ne donnent pas d'information sur l'adresse réseau du client.

2.4.1.2 Les *EventHandler*

Les *EventHandler* sont aussi des mécanismes de collecte d'informations mais qui sont propriétaires à Visibroker. Visibroker en propose deux types : `ClientEventHandler` et `ImplEventHandler` implantés respectivement côté client et côté serveur.

Nous avons utilisé les `ImplEventHandler` qui permettent d'informer le serveur sur certains événements de communication : connexion et déconnexion du client, connexion avortée, pre-opération et post-opération, exception sur l'exécution d'une opération, etc. Les informations supplémentaires collectées par les *eventHandler* sont les suivantes :

- l'adresse IP de la machine du client,
- le numéro de port utilisé par le client,
- le nom de l'exception.

Le principal avantage des *eventHandler* est leur capacité à connaître l'adresse et le numéro de port du client ainsi que le nom des exceptions levées par les opérations. De plus, ils indiquent l'arrêt brutal des clients, ce

qui permet la destruction des contextes liés aux connexions sans avoir recours à des temporisations. Ils permettent aussi de refuser une connexion du client. Néanmoins, contrairement aux intercepteurs, ils ne sont pas portables et ne permettent pas de tracer et de modifier les paramètres.

2.4.2 Choix des données discriminantes

Chacun des deux mécanismes permet de collecter un certain type d'informations. Le choix des données nécessaires à la définition d'un comportement normal est déterminant dans le choix du mécanisme d'audit les collectant. Pour notre maquette, nous avons utilisé ces deux mécanismes d'interception et nous avons prévu les fonctionnalités suivantes :

- Utiliser les intercepteurs ou les *eventHandler* ou les deux en même temps. Dans ce choix, il est important de considérer la surchage du temps d'exécution des requêtes induite par l'interception. Nous donnons à titre indicatif dans le tableau 2.2 le temps d'exécution d'une opération avec et sans interception.

Type d'interception	Temps moyen d'une opération de versement (msec)
Serveur lancé sans interception	5.1
Serveur lancé avec intercepteur mais aucune trace n'est demandée par l'administrateur	5.8
Intercepteur en fonction (traces demandées par l'administrateur)	7.1 - 18.2
EventHandler en fonction (traces demandées par l'administrateur)	6.6 - 17.8
Intercepteur et EventHandler en fonction	8 - 30.2

TAB. 2.2 – *Surcharge induite par l'interception*

- Filtrer les informations collectées. Le filtrage permet de décider pour quel client, quel objet et quelle opération nous souhaitons recueillir des informations. Dans le cadre des tests effectués sur notre maquette, nous avons choisi d'observer toutes les requêtes de tous les clients de la maquette développée ; aucun filtre n'a été alors utilisé.

- Poser un masque sur les données. Cette option permet d'afficher ou de masquer les données sélectionnées à l'administrateur. Dans les deux cas, les informations recueillies par les intercepteurs sont sauvegardées dans un fichier d'audit.

En utilisant les intercepteurs et les *eventhandler*, nous avons pu collecter toutes les informations suivantes, considérées discriminantes pour la définition d'un modèle de comportement normal :

- *principal* : constitué du login du client, il est nécessaire à l'identification du client initiateur de la requête,
- *nom de l'objet invoqué* : permet de vérifier si le client a le droit d'invoquer le service de cet objet,
- *nom de l'opération invoquée* : permet de vérifier si le client a le droit d'invoquer cette opération sur cet objet,
- *paramètres de l'opération invoquée* : permettent de considérer les valeurs observées pour les paramètres comme des données discriminantes,
- *date et heure d'invocation* : permettent d'établir le contexte d'invocation d'une requête. Elles sont utiles plus tard dans le calcul de l'écart de temps entre deux invocations successives.

Les requêtes invoquées par les clients et leurs paramètres sont stockées dans un fichier d'audit. Nous détaillons dans le paragraphe suivant le format choisi pour représenter les traces d'audit.

2.4.3 Format des traces d'audit

Pour la représentation des données collectées, nous avons proposé un format simple permettant notamment une exploitation rapide du fichier d'audit en phase de détection. La nature des données collectées étant elle-même simple, nous avons proposé de les représenter dans un fichier texte, en respectant une syntaxe particulière. Nous donnons ci-après le modèle de représentation de chaque ligne dans le fichier d'audit :

```
<server "date heure" "nom_machine" "nom_application"
"login_client" "num_agent" "type_intercept">

<operation "nom_operation" "user type nom_user"
"nom_param type_param val_param" ...
"nom_param type_param val_param">
```

Chaque ligne décrit un événement et contient deux items :

- <server...> décrit la source de l'événement.

  Exemple :
  <server "9/29/00 08:10:47" "machine" "BanqueLeon" "appli"
  "1" "IS">

  - le mot clé server indique le début de l'item,
  - chaque item est compris entre guillemets. Les items sont séparés par un simple espace. Le premier item contient la date et l'heure, séparées par un espace,
  - nom de la machine,
  - nom de l'application (par défaut BanqueLeon),
  - login sous lequel tourne le serveur (par défaut appli),
  - numéro historique de l'agent d'interception num_agent. Nous avons considéré dans notre maquette un seul objet serveur, il y a donc un seul agent d'interception (son numéro est par défaut égal à 1). Dans une application réelle avec plusieurs objets CORBA, il est possible de créer plusieurs agents associés aux objets.
  - nature du mécanisme d'audit (IS pour Intercepteur de Serveur et H pour eventHandler).

- <operation...> décrit l'opération tracée.

  Exemple :
  <operation "connexion" "user string[0] 'zakia' ">

  - le mot clé operation marque le début de l'item,
  - nom de l'opération,
  - liste de paramètres. Chaque paramètre est composé d'une chaîne de caractères résultant de la concaténation du nom du paramètre nom_param, de son type type_param et de sa valeur val_param, entre simples quotes.

A titre d'exemple, nous donnons une ligne extraite du fichier d'audit de notre maquette :

```
<server "9/29/2000 08:10:47" "machine" "BanqueLeon"
"appli" "1" "IS">
<operation "Connexion" "user string[0] 'zakia'"
"Valider int '7'" "ctrl_password\$o string[0]
'EMMO4554'" "ctrl_nom\$o string[0] 'EMMO4554'">
```

Le fichier d'audit constitue le point d'entrée pour les phases d'apprentissage et de détection.

2.4.4 Implémentation

Nous avons utilisé le langage Java (JDK1.2) pour l'implémentation de notre IDS. Nous donnons dans le tableau 2.3 le nombre de lignes de code pour le programme d'apprentissage et de détection.

	nombre de lignes de code
Programme d'apprentissage	1400
Programme de détection	440

TAB. 2.3 – *Nombre de lignes de code pour l'IDS en utilisant les intervalles de tolérance*

L'apprentissage du comportement de chaque client se fait par la commande suivante :

```
java Learn [login_user_auditFile]
```

Le programme `Learn` crée pour chaque client un profil, sauvegardé dans un fichier binaire `login_user.bin`. La détection du comportement se fait par la commande Java suivante :

```
java Detect [port_number]
```

2.4.5 Résultats des tests

Nous avons testé notre approche sur l'application bancaire développée et obtenu des résultats encourageants. Le comportement appris du client est constitué d'une suite d'opérations de base sur un compte bancaire (créer, verser, retirer, consulter le solde). Une période d'apprentissage de trois semaines a permis de couvrir l'ensemble des comportements de notre client observé. Pour la détection, nous avons implanté dans la maquette les règles de détection énoncées dans le paragraphe 2.3.1.

Une première série de tests consiste à tester la pertinence des intervalles de tolérance construits. Deux types de tests sont réalisés :

1. Le client testé invoque des opérations «légales» mais avec des valeurs de paramètres différentes de celles apprises. Ces tests montrent bien qu'un

écart trop important par rapport aux valeurs apprises, diminue sensiblement le degré se similarité et augmente ainsi le nombre d'alertes.

2. Le client testé refait les mêmes opérations avec les mêmes valeurs de paramètres que dans le premier test, mais dans ce cas les écarts autorisés (δ_1 et δ_2) pour les paramètres numériques sont modifiés. Lorsque les écarts sont diminués, le degré de similarité de chaque opération est rapidement diminué, il en résulte un nombre plus important d'alertes générées pour le même comportement observé (voir première courbe de la figure 2.16). Lorsque les écarts sont importants, le degré de similarité est faiblement affecté, le comportement déviant est accepté sans alerte (voir deuxième courbe de la figure 2.16).

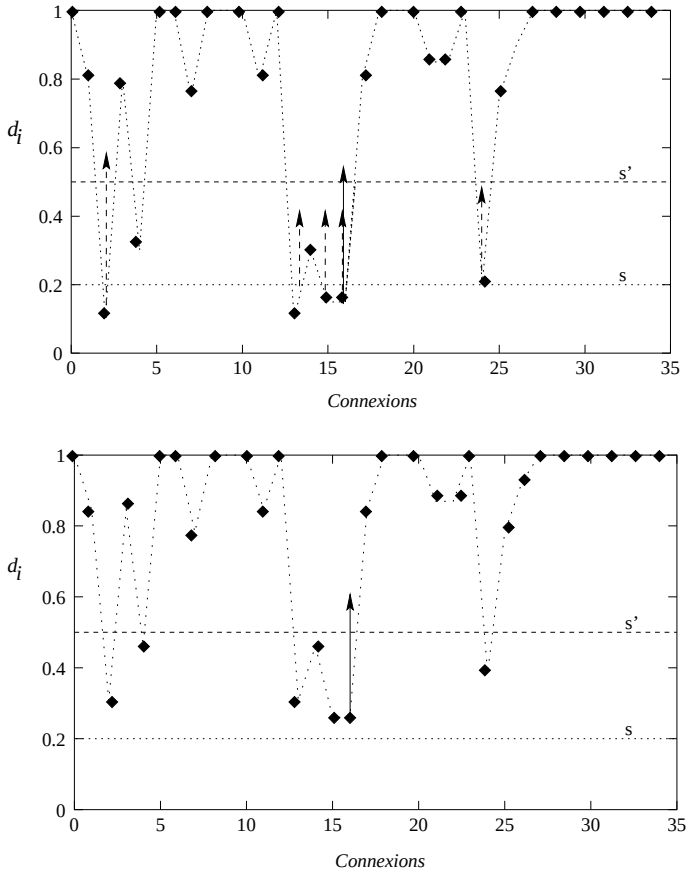

FIG. 2.16 – *Variation du nombre d'alertes générées en fonction des intervalles de tolérance*

Une fois les écarts δ_1 et δ_2 fixés, on effectue une deuxième série de tests afin d'adapter le seuil s en fonction de ce qui a été appris. Le nombre d'alertes

dépend de ce seuil. Le seuil d'acceptabilité doit donc varier suivant le contexte applicatif et la richesse du comportement appris du client. La figure 2.17 montre la variation du nombre d'alertes générées en fonction des valeurs des seuils d'acceptabilité. La figure 2.17 présente les alertes générées pour les couples (s,s') : (0.3,0.5) et (0.45,0.5).

Ces tests montrent bien que le nombre d'alertes dépend d'un part, des intervalles de tolérance et des écarts autorisés autour des valeurs apprises, et d'autres part, des seuils d'acceptabilité s et s'. La suite de l'étude des seuils ne peut être envisagée que si l'on dispose d'une application CORBA réelle.

2.5 Test de l'approche sur des données d'audit réelles

Grâce au partenariat avec FT R&D, nous avons pu disposer du fichier d'audit traçant l'activité d'une application réelle dans un environnement à objets répartis. Néanmoins, n'ayant pas eu accès à l'application même, les tests se sont fait hors ligne. La figure 2.18 donne l'architecture d'apprentissage et de détection pour le test avec ces données. Nous donnons dans le tableau 2.4 des informations sur les données réelles testées.

Nombre de clients	6
Taille des fichiers d'audit	entre 6Ko et 150 Ko par client
Nombre de nœuds dans un arbre	entre 30 et 150 nœuds

TAB. 2.4 – *Caractéristiques de la Base de Comportements*

Pour tester la détection sur la base des arbres construits lors de l'apprentissage, nous avons généré plusieurs séquences de traces à partir du journal initial. Nous avons effectué deux types de tests sur cette nouvelle base de comportements :

1. Pour Détecter toute déviation dans le comportement d'un même utilisateur, nous avons modifié le comportement normal des utilisateurs à tester. Nous donnons deux exemples de tests de comportements :
   - Le comportement testé est le même que le comportement appris. Dans ce cas l'IDS reconnaît bien un comportement normal, aucune alerte n'est générée.
   - Nous avons modifié le comportement d'un utilisateur en changeant les valeurs de certains paramètres d'opérations. L'objectif étant de

faire décroître le degré de similarité afin de tester la pertinence des intervalles de tolérance.

Les résultats obtenus sont les mêmes qu'avec la maquette réalisée. Nous détectons toute déviation dans le comportement normal portant sur :

- l'ordre des invocations d'opérations,
- les valeurs prises par leurs paramètres, compte tenu des intervalles de tolérance correspondants,

2. Pour tester le comportement normal de certains utilisateurs avec le profil (comportement appris) d'autres utilisateurs, nous avons testé les comportements normaux de certains utilisateurs avec les profils d'autres utilisateurs. L'IDS émet évidemment des alertes puisqu'il ne reconnaît pas le profil de l'utilisateur testé. Néanmoins, il arrive parfois que certains utilisateurs se comportent de la même manière, c'est-à-dire qu'ils ont la même utilisation des services d'une application ; dans ce cas, l'IDS ne détecte aucune anomalie puisque les profils des utilisateurs testés sont considérés identiques.

Nous avons essayé à partir de ces tests de valider notre approche sur des données réelles. Nous avons essayé de varier les tests en proposant de tester le comportement d'un utilisateur avec le profil d'un autre. Les résultats montrent que l'algorithme détecte bien toute déviation selon les critères définis au départ.

Il serait maintenant plus intéressant de faire des tests en environnement et temps réels pour obtenir des résultats plus tangibles.

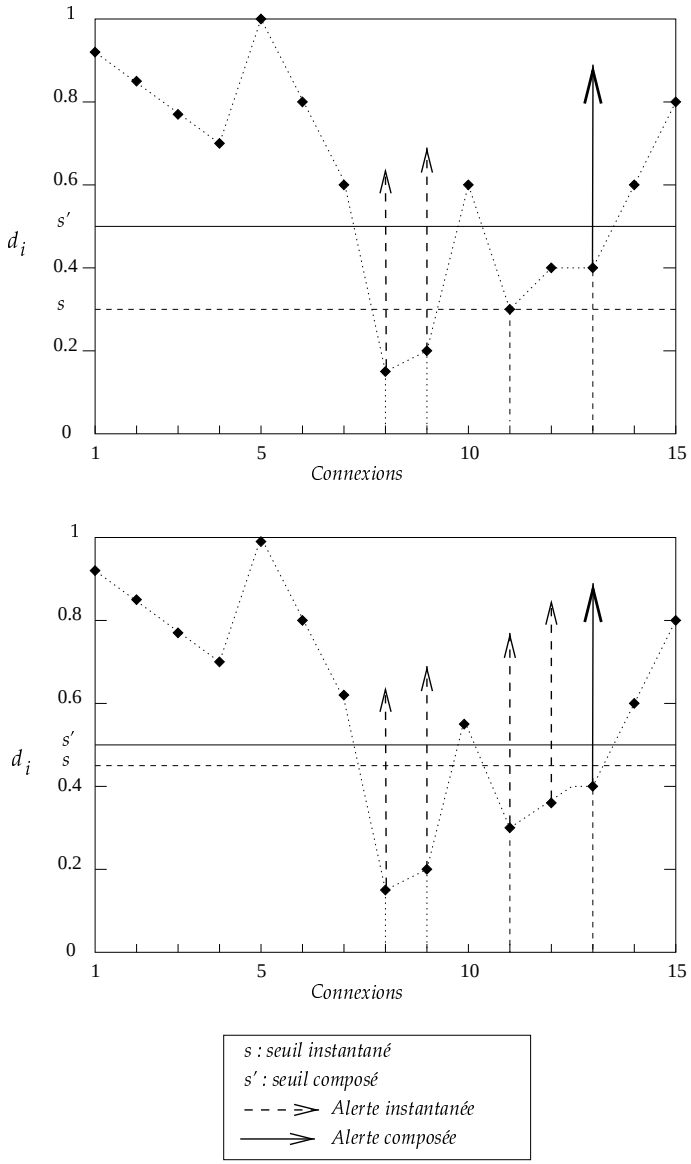

FIG. 2.17 – *Variation du nombre d'alerte générées en fonction des seuils d'acceptabilité*

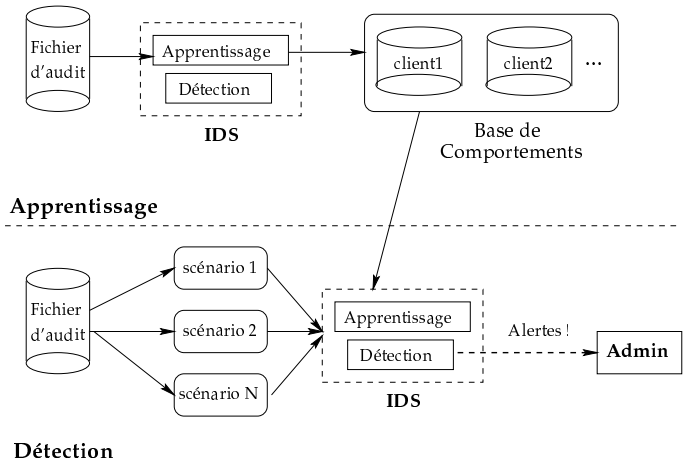

FIG. 2.18 – *Architecture de Test*

2.6 Conclusion

Nous avons proposé dans cette première étape les principes de notre approche de détection d'intrusions basée sur une modélisation arborescente des comportements des clients observés. Cette approche a été appliquée dans un environnement CORBA. L'algorithme de détection proposé est basé sur la mesure de la déviation du comportement observé par rapport au comportement appris d'un client. L'algorithme prend en compte l'écart autorisé pour chacun des paramètres des requêtes invoquées, afin de calculer un degré de similarité. Ce degré est calculé durant chaque connexion et permet de décider du déclenchement d'une alerte en fonction d'un seuil préalablement fixé. Le client est en outre observé durant plusieurs connexions, afin de détecter des suites suspectes de petites déviations pouvant constituer une attaque.

Les tests expérimentaux montrent que l'algorithme détecte bien les comportements anormaux. Lorsque le comportement est anormal, l'algorithme nous fournit le degré de similarité de la connexion correspondante. Il reste à décider de la pertinence de l'alerte, compte tenue de la valeur de ce degré. Pour l'instant, nous ne disposons pas d'une application réelle permettant de conclure sur la présence d'une attaque. Le taux de fausses alertes reste donc à étudier.

De plus, nous avons constaté que le comportement de chaque client peut être assez riche et donc assez long à modéliser. Il serait alors intéressant d'étudier le comportement d'une population de clients d'une même application. Ce comportement serait plus rapide à stabiliser, ce qui diminuerait considérablement la durée de la phase d'apprentissage, et par conséquent le risque d'attaque durant cette période. Nous proposons ainsi, dans la deuxième phase de notre travail d'étudier, côté serveur, le comportement des objets servants invoqués par l'ensemble des clients de l'application. Ce travail fait l'objet du chapitre 3.

Chapitre 3

Modélisation du comportement des objets servants

3.1 Introduction

Nous présentons dans ce chapitre une approche de détection d'anomalies basée sur la modélisation, non plus du comportement des clients vis-à-vis d'une application, mais des objets servants de l'application vis-à-vis des requêtes qu'ils reçoivent. En effet, il s'avère parfois difficile de modéliser le comportement d'un utilisateur lorsque celui-ci est très riche. En revanche, il est plus aisé de modéliser celui des objets servants d'une application. Apprendre le comportement des objets revient donc à construire une représentation de leur fonctionnement. Nous proposons de modéliser le comportement des objets servants par un arbre décrivant les suites de requêtes passées entre les objets de l'application au vue des invocations des clients. La modélisation du comportement des objets est un peu plus complexe que celle d'un utilisateur puisqu'elle prend en compte les points suivants :

- Tous les objets servants de l'application, émetteurs et cibles de requêtes, sont considérés et modélisés dans une seule et même structure arborescente.
- Nous considérons un comportement normal de l'ensemble des objets servants d'une application comme une suite d'invocations/réponses jugées normales entre les objets durant une connexion d'un client.
- Les clients sont considérés dans leur globalité comme les initiateurs des comportements des objets qu'ils invoquent.
- Nous considérons, en plus des valeurs de paramètres des requêtes, les délais de temps entre deux requêtes successives. En effet, l'intervalle de temps séparant deux invocations successives peut être révélateur

d'anomalie ; il est donc important de le considérer dans la définition d'un comportement normal.
- Enfin, nous avons présenté dans l'introduction générale, les vulnérabilités des architectures supportant la délégation ; nous proposons donc de prendre en compte la délégation des requêtes qui met en évidence l'interaction entre les objets servants suites aux invocations des clients.

La phase de détection permet de déceler toute déviation dans le fonctionnement appris des objets responsables des réponses aux requêtes des utilisateurs de l'application. Cette modélisation ne donne plus une vision du comportement des utilisateurs mais plutôt une vision globale permettant de détecter les effets d'une anomalie sur le comportement des objets servants.

Le paragraphe 3.2 donne la nouvelle architecture de base pour la modélisation du comportement des objets servants. Le paragraphe 3.3 présente la phase d'apprentissage mettant en avant les modifications effectuées sur le modèle de comportement proposé au chapitre 2 pour la modélisation des clients. Le paragraphe 3.4 détaille la phase de détection. Le paragraphe 3.5 détaille la mise en œuvre de cette approche sur une maquette CORBA. Le paragraphe 3.6 détaille les tests réalisés sur des données réelles. Enfin, le paragraphe 3.7 donne nos conclusions pour cette modélisation.

3.2 Architecture générale

Pour la mise en œuvre de ce modèle, nous proposons une architecture de base pour la collecte d'informations, la modélisation du comportement et la détection d'anomalies. Cette architecture est composée des entités suivantes (voir figure 3.1) :

- Un ensemble d'utilisateurs d'une application à objets répartis. Dans cette architecture tous les utilisateurs dialoguent avec un même objet client auquel ils adressent leurs requêtes : c'est le point d'entrée vers l'application.
- Les objets de l'application : ces objets reçoivent les requêtes des utilisateurs et sont responsables des réponses. Ce sont donc des objets servants mais ils peuvent éventuellement se comporter comme des clients d'autres objets servants en cas de délégation. Face aux requêtes des utilisateurs, les objets servants réagissent de trois manières différentes :

  1. si l'invocation d'un client est légale et que l'objet invoqué est en mesure d'y répondre directement alors l'objet exécute la requête et retourne une réponse au client ;

2. si l'invocation d'un client est illégale, elle génère une exception qui est retournée au client par l'objet servant ;
3. si l'invocation est légale mais ne peut être exécutée directement par l'objet servant, celui-ci fait appel aux services d'un ou plusieurs autres objets servants afin d'accomplir la requête. Il s'agit dans ce cas d'une délégation de requêtes entre les objets servants.

– Contrairement à l'architecture utilisée dans le chapitre 2, les agents d'interception sont installés sur les objets servants et clients. Ces agents sont capables, à la demande de l'administrateur de l'application, de collecter des informations sur les requêtes émises ou reçues par les objets.
– L'administrateur qui télécommande les agents d'interception, reçoit les traces collectées et les sauvegarde dans un fichier d'audit.
– L'IDS qui reste composé de deux modules : le module d'apprentissage qui génère un modèle de comportement pour l'ensemble des objets servants à partir du fichier d'audit et le module de détection qui mesure le degré de déviation du comportement observé par rapport au comportement modélisé et déclenche des alertes suivant les critères de détection (voir paragraphe 3.4).

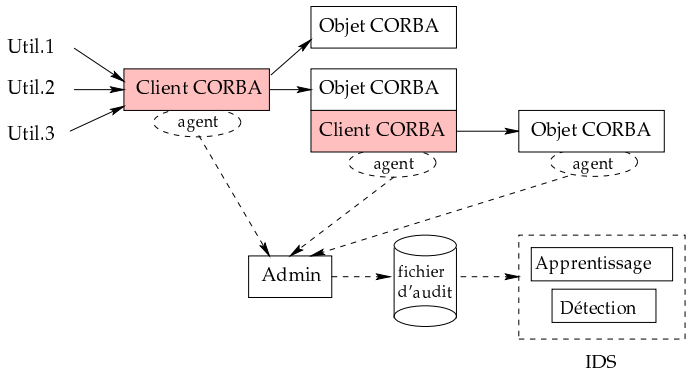

FIG. 3.1 – *Architecture d'audit*

3.3 Phase d'apprentissage

L'apprentissage consiste à générer, à partir du journal d'audit, un modèle de comportement pour les objets servants de l'application. Le comportement d'un objet est défini par la suite des requêtes reçues (par les utilisateurs ou les objets) et émises vers les autres objets. Le paragraphe 3.3.1 détaille

la construction de l'arbre de comportement pour les objets de l'application. Le paragraphe 3.3.2 donne l'algorithme d'apprentissage. Le paragraphe 3.3.3 détaille la construction des intervalles de tolérance pour les valeurs de paramètres numériques ainsi que pour les délais.

3.3.1 Construction du modèle de comportement

Pour modéliser le comportement de l'application, nous proposons une architecture d'arbre qui traduit les critères décrits dans le paragraphe 3.2. L'arbre de comportement proposé est constitué :

- d'un ensemble de nœuds : chaque nœud représente une invocation d'un objet sur un autre de l'application ;
- d'un ensemble de branches : on fait porter à chaque branche reliant deux nœuds consécutifs des informations sur le temps écoulé entre les requêtes successives portées par ces deux nœuds.

Nous illustrons la construction de l'arbre de comportement par un exemple basé sur l'architecture présentée dans la figure 3.1. Nous proposons le scénario suivant, illustré par la figure 3.2 :

1. Utilisateur `U1` :
   - Connexion de `U1`
   - A la demande de `U1`, l'objet client `appli` exécute l'opération `f` sur l'objet `B` d'où une délégation :
     - A la demande de `U1` et de `appli`, l'objet `B` exécute `f` sur l'objet `C`
     - A la demande de `U1` et de `appli`, l'objet `B` exécute `f` sur l'objet `D`
   - A la demande de `U1`, l'objet client `appli` exécute l'opération `f` sur l'objet `A`
2. Utilisateur `U2` :
   - Connexion de `U2`
   - A la demande de `U2`, l'objet client `appli` exécute l'opération `f` sur l'objet `B` d'où une délégation :
     - A la demande de `U2` et de `appli`, l'objet `B` exécute `f` sur `C`
     - A la demande de `U2` et de `appli`, l'objet `B` exécute `h` sur `D`
   - A la demande de `U2`, l'objet client `appli` exécute `g` sur l'objet `A`

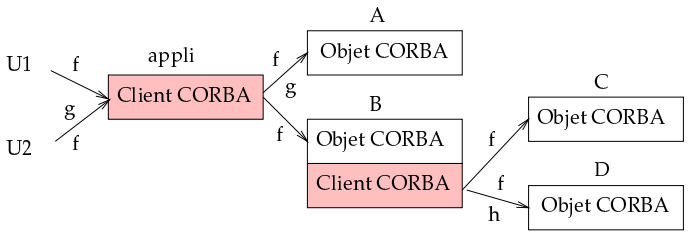

FIG. 3.2 – *Exemple de scénario d'invocation*

La figure 3.3 présente l'arbre de comportement obtenu pour ce scénario. Cette architecture d'arbre permet, en outre, de représenter les opérations issues d'une délégation. Les nœuds en pointillé désignent les nœuds cibles de délégation. Ils sont construits de la même manière que les autres nœuds mais sont distingués par leur liste d'émetteurs. Les branches en pointillé relient les nœuds sources de délégation aux nœuds cibles de délégation. L'ensemble des nœuds de délégation constitue un sous-arbre de délégation (s-a-d).

Nous donnons dans la figure 3.4 le diagramme de classes illustrant la structure des nœuds et des branches de l'arbre. Nous présentons dans le paragraphe 3.3.1.1 le shéma de délégation choisi pour notre architecture. Nous détaillons ensuite la structure des nœuds ainsi que celle des branches respectivement dans le paragraphe 3.3.1.2 et le paragraphe 3.3.1.3.

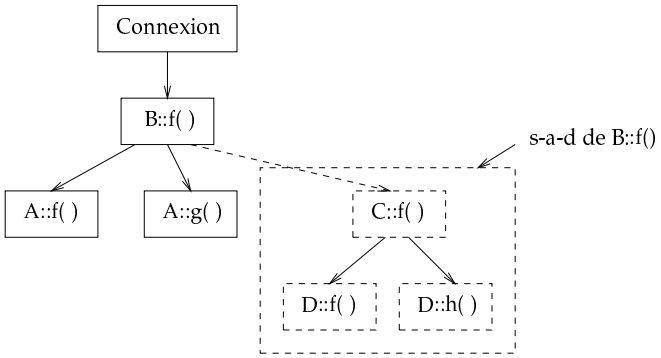

FIG. 3.3 – *Arbre de comportement*

3.3.1.1 Délégation

Dans l'étude du comportement des objets servants vis-à-vis des requêtes des utilisateurs, nous nous intéressons aux interactions entre les objets servants. Il est dans ce cas important de prendre en compte la délégation étant

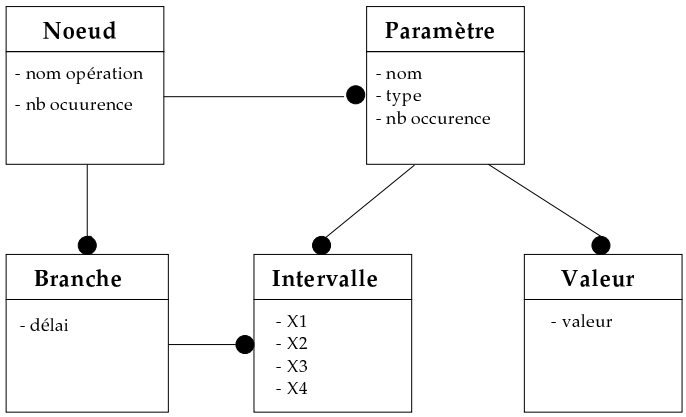

FIG. 3.4 – *structure des nœuds et des branches*

donné qu'à chaque invocation, un objet servant (intermédiaire ou cible finale) a la responsabilité d'accepter (et éventuellement de transmettre) la requête qu'il reçoit. Un problème essentiel lors de la délégation est de récupérer les identités des objets intermédiaires jusqu'à l'initiateur principal d'une requête, afin de pouvoir les retrouver en phase de détection. Deux cas sont envisageables pour ce problème :

1. L'objet servant intermédiaire utilise ses propres privilèges pour invoquer un autre objet servant. Dans ce cas aucune délégation n'est définie (voir Figure 3.5).

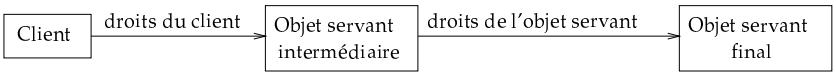

FIG. 3.5 – *pas de délégation*

2. L'objet servant décide de la légitimité de la requête reçue compte tenu des privilèges du principal et de l'ensemble des objets intermédiaires. Dans ce cas, plusieurs schémas de délégation sont prévus :

   délégation simple : Seuls les privilèges du principal sont délégués. Le dernier objet servant reçoit uniquement les droits du client mais pas ceux des objets intermédiaires (voir Figure 3.6).

   délégation composite : Dans ce cas, les privilèges du principal et de chaque objet intermédiaire sont transmis à l'objet servant sous deux formes (Figure 3.7) :

   – Les privilèges du principal sont séparés de ceux des objets in-

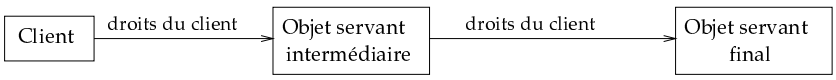

FIG. 3.6 – *délégation simple*

termédiaires. Ce type de délégation permet de connaître quel client a été à l'origine de quelle requête.

– Les privilèges du principal sont combinés avec ceux des objets intermédiaires dans un seul ensemble de privilèges.

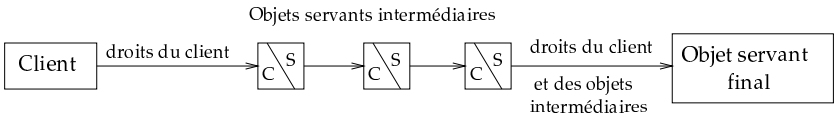

FIG. 3.7 – *délégation composite*

Ces schémas de délégation sont décrits dans les spécifications des services de sécurité de CORBA 2.3 [29].

Nous avons choisi le dernier schéma de délégation composite : à chaque étape de la délégation, les droits des objets intermédiaires sont cumulés. Pour la mise en œuvre de notre approche dans un environnement CORBA, nous avons utilisé l'ORB Visibroker 4.0 qui ne prévoit aucun schéma de délégation de privilèges spécifiés par l'OMG. Nous avons donc appliqué ce schéma pour la délégation non plus des privilèges, mais de l'identité des émetteurs intermédiaires, en cas de délégation de requêtes. En effet, nous souhaitons, lors de la détection, pouvoir retrouver la trace des émetteurs intermédiaires d'une requête afin de détecter les abus.

Par ailleurs, dans les spécifications de l'OMG, il est clairement indiqué que le client principal est impliqué dans le processus de délégation. Il nous est donc impossible de remonter à l'identité du client (principal) à moins de l'inclure explicitement dans le chemin d'invocation d'une requête. Nous avons décidé d'ajouter explicitement l'identité du client et des objets servants intermédiaires à chaque étape de la délégation d'une requête.

3.3.1.2 Structure des nœuds

L'arbre de comportement de l'application a pour rôle de représenter l'ensemble des invocations entre les objets de l'application issues d'une requête d'un client de l'application. Un nœud de l'arbre porte donc l'ensemble des

informations relatives à une opération invoquée par un objet à un autre au sein de l'application. Un nœud comprend :

- le nom de l'opération invoquée et sa liste de paramètres,
- l'objet émetteur de la requête (ou la liste des émetteurs en cas de délégation),
- l'objet cible de la requête,
- les branches issues de ce nœud. Chaque branche relie un nœud de l'arbre à un de ses fils et porte le délai de temps écoulé entre les deux opérations portées par le père et le fils. Un nœud fils dans l'arbre de comportement désigne une opération succédant à celle du nœud père. Un nœud fils dans le sous-arbre de délégation désigne une opération cible d'une délégation déclenchée par le nœud père.
- la liste des intervalles de tolérance pour les paramètres numériques de l'opération. Cette liste associe à chaque paramètre numérique un ensemble d'intervalles de tolérance, représenté par un ensemble de trapèzes. La construction des intervalles de tolérance est détaillée dans le paragraphe 3.3.3.

Nous donnons ici un exemple du contenu d'un nœud construit à partir d'une ligne du fichier d'audit qui correspond à l'opération `f` exécutée sur l'objet `::applic::s1::` par l'objet `bv1@Appli` le `09 Novembre 2001` à `14h19mn08` :

```
01/11/09:14:19:08:0012 {bv1@Appli} {2}::applic::s1::A::f
{i long 23}
```

Le nœud correspondant contient les informations suivantes :

- nom opération : `f`
- paramètre : `(i long 23)`
- objet émetteur : `(bv1@Appli)`
- objet cible : `(::applic::s1::A)`

3.3.1.3 Structure des branches

Une branche relie un nœud de l'arbre à un nœud fils. Il existe autant de branches issues d'un nœud que de fils de ce nœud. Une branche fait partie des composants d'un nœud et porte les informations suivantes :

- le nom du nœud fils. Le nœud fils peut être un nœud de l'arbre ou bien un nœud du sous-arbre de délégation du nœud père.

- le nom de l'objet invoqué par le nœud fils.
- la liste des émetteurs. C'est la liste des émetteurs qui permet de différencier les nœuds issus d'une délégation des autres nœuds. Cette mémorisation d'information est importante pour la fusion et la détection (voir paragraphe 3.4). En effet, si deux nœuds fils portent le même nom d'opération invoquée sur le même objet cible par les mêmes émetteurs ils sont fusionnés, sinon deux branches différentes sont créées.
- le délai de temps écoulé entre l'opération portée par le nœud père et celle portée par le nœud fils. Une fois la branche créée, on mémorise à chaque passage par cette branche tous les délais observés entre ces deux nœuds.
- l'intervalle de tolérance construit autour des délais observés durant l'apprentissage entre les deux nœuds liés par la branche. L'intervalle de tolérance exprime dans ce cas l'écart autorisé dans les délais entre deux opérations consécutives.

3.3.2 Algorithme d'apprentissage

L'apprentissage consiste à construire, à partir du fichier d'audit, un arbre global pour l'application par ajout de chemins. Un chemin est une suite d'opérations invoquées entre les objets de l'application durant une connexion du client (entre début et fin de session). Les étapes suivies sont les suivantes :

- lire le fichier d'audit ligne par ligne et pour chacune créer un nouveau nœud. Une fois le nœud créé, il est ajouté à l'arbre. En observant les numéros historiques et la liste des émetteurs d'une opération plusieurs cas se présentent :
  - Si le nouveau nœud a la même liste d'émetteurs que le nœud courant dans l'arbre, le nouveau nœud est ajouté comme un fils au nœud courant. Le nouveau nœud devient le nœud courant dans l'arbre.
  - Si les deux nœuds n'ont pas la même liste d'émetteurs, plusieurs cas se présentent aussi. On recherche d'abord si le nouveau nœud existe déjà dans la liste des fils du nœud courant. On considère qu'un nœud existe déjà dans la liste des fils si on trouve un fils qui porte le même nom d'opération invoquée sur le même objet cible par les mêmes émetteurs que le nouveau nœud. Suite à cette recherche, deux cas sont possible :
    1. Le nouveau nœud existe déjà dans la liste des fils. Il s'agit donc de fusionner le contenu du nouveau nœud avec celui du

fils déjà existant. La branche entre le nœud père et le nœud fils existe déjà. Il suffit d'ajouter le nouveau délai observé à la liste des délais de cette branche.

2. Le nouveau nœud n'existe pas dans la liste des fils du nœud courant. Nous testons alors si le nouveau nœud appartient déjà au sous-arbre de délégation du nœud courant. Deux cas sont alors possible :
   (a) Si le père apparaît comme avant dernier émetteur dans la liste des émetteurs du nouveau nœud : il s'agit d'une opération de délégation. Le nouveau nœud est ajouté au sous-arbre de délégation du nœud père et devient le nœud courant dans le sous-arbre de délégation.
   (b) Sinon, on entame une recherche du nœud dans le sous-arbre de délégation courant. Si le nœud courant dans le sous-arbre de délégation apparaît comme dernier émetteur dans la liste des émetteurs du nouveau nœud, il s'agit aussi d'une opération de délégation dans le sous-arbre de délégation. Le nouveau nœud est ajouté au sous-arbre de délégation et devient le nœud courant dans le sous-arbre de délégation.

   Dans tous les cas de non fusion, on ajoute une nouvelle branche entre le nœud père et le nouveau nœud en mémorisant le délai de temps écoulé entre les deux opérations.

- Une fois l'arbre et ses sous-arbres de délégation construits, on procède à la création des intervalles de tolérance représentés par des trapèzes. Ces trapèzes sont construits uniquement pour les délais de temps séparant deux invocations successives et pour les paramètres numériques des requêtes dont les valeurs peuvent dévier de celles apprises avec un certain seuil d'acceptabilité s.
- L'arbre est enfin sauvegardé afin d'être exploité en phase de détection.

3.3.3 Construction des intervalles de tolérance

La construction des intervalles de tolérance pour les paramètres numériques est la même que celle proposée dans le chapitre 2 (cf. paragraphe 2.2.2).

Pour les délais de temps, nous proposons de laisser l'administrateur paramétrer le module de construction des intervalles de tolérance en décidant des écarts autorisés pour les délais. La création des trapèzes est une procédure indépendante de la construction de l'arbre, ce qui facilite son paramétrage. En

effet, il est possible pour l'administrateur de l'application de poser des conditions sur certains paramètres en fonction de la sémantique associée et de leur degré de sensibilité dans l'application. Considérons par exemple le paramètre *numéro de téléphone d'un client*. Même si ce paramètre est numérique, aucune déviation par rapport à sa valeur n'est autorisée. Il en est de même pour les numéros de compte, etc. De plus, certains paramètres peuvent être plus «sensibles» que d'autres. L'administrateur de l'application peut donc autoriser un écart autour des valeurs apprises pour ces paramètres mais à des degrés différents. Il suffit dans ce cas de pondérer les valeurs d'acceptation associées aux valeurs observées par des coefficients et exprimer ainsi la sensibilité ou l'importance de certains paramètres par rapport à d'autres. L'ajustement de ces paramètres peut être effectué de manière indépendante du programme d'apprentissage et sans affecter le contenu des nœuds.

Le module de construction des intervalles prend donc en paramètre les écarts autorisés (δ_1 et δ_2) autour des valeurs apprises pour les délais de temps durant la phase d'apprentissage. Les valeurs apprises pour les délais séparant deux opérations successives sont sauvegardées au niveau de chaque branche liant les deux opérations. La construction des intervalles de tolérance consiste à parcourir toutes les branches et à créer pour chacune un trapèze pour les délais observés sur la branche. La création d'un trapèze passe d'abord par la recherche des valeurs minimale x_2 et maximale x_3 des délais observés. Les écarts δ_1 et δ_2 (donnés en paramètres) permettent de déduire les valeurs de x_1 et x_4. Le trapèze correspond donc au quadruplet $[x_1,x_2,x_3,x_4]$.

Un intervalle de tolérance s'exprime toujours par la fonction trapézoïdale $va(x)$ (cf. équation 2.1).

3.4 Phase de détection

La phase de détection consiste à mesurer la déviation du comportement des objets observés par rapport au comportement appris. Dans la mesure des déviations, nous prenons en compte les critères de détection suivants :

- l'ordre d'invocation des opérations,
- les valeurs prises par leurs paramètres,
- l'émetteur (ou la liste des émetteurs en cas de délégation) d'une opération,
- le délai de temps écoulé entre deux opérations consécutives.

Le comportement observé est jugé normal s'il est constitué d'une suite d'opérations connues avec des valeurs de paramètres jugées normales et des

délais acceptables. Nous détaillons dans le paragraphe 3.4.1 le calcul du degré de similarité. Nous donnons, dans le paragraphe 3.4.2 l'algorithme de détection.

3.4.1 Calcul du degré de similarité

Le calcul du degré de similarité d_i pour la connexion i commence toujours à la racine de l'arbre avec $d_i^0 = 1$. Puis, le degré est ajusté à chaque nœud exploré par application d'une éventuelle pénalité fonction des n valeurs d'acceptation va_j (cf. définition au paragraphe 2.2.2) des paramètres de l'opération associée au nœud. L'expression de la valeur de d_i au nœud k (d_i^k), en fonction de sa valeur au nœud précédent (d_i^{k-1}), est toujours la même :

$$d_i^k = d_i^{k-1} - P(va_j..va_n) \tag{3.1}$$

Dans la modélisation du client (voir chapitre 2), la pénalité $P(va_j..va_n)$ affectée à un nœud est obtenue par l'aggrégation des pénalités élémentaires $p(va)$ affectées à chaque paramètre. Pour les paramètres symboliques, une simple comparaison entre la valeur observée et la valeur apprise nous donne une pénalité maximale ou nulle selon que ces valeurs sont identiques ou différentes. Pour un paramètre numérique, la pénalité élémentaire dépend de la valeur d'acceptation du paramètre selon la formule suivante :

$$p(va) = (1 - va)^2 \tag{3.2}$$

La pénalité globale est obtenue par le calcul de la moyenne des pénalités élémentaires comme indiqué dans la formule suivante :

$$P(va_1..va_n) = \frac{\sum_{j=1}^{n} p_j(va_j)}{n} \tag{3.3}$$

Dans la modélisation des objets, nous considérons, en plus des paramètres de l'opération, le délai de temps écoulé entre deux opérations successives. La fonction de calcul de la pénalité prend alors en deuxième paramètre la valeur d'acceptation associée au délai de temps et l'intègre dans le calcul de la pénalité globale associée au nœud. La formule utilisée devient alors la suivante :

$$P(va_1..va_n,va_{delai}) = \frac{\sum_{j=1}^{n} p_j(va_j) + p(va_{delai})}{n+1} \tag{3.4}$$

Cette formule permet de considérer le délai comme un paramètre supplémentaire dans la requête. Cette écriture donne la même importance à tous les paramètres de la requête et au délai ; elle permet d'atténuer l'effet d'un

petit ou un trop grand écart dans le délai de temps. Nous pouvons envisager de pondérer les pénalités élémentaires de ces paramètres par des coefficients suivant l'importance accordée à chaque paramètre et au délai.

3.4.2 Algorithme de détection

La détection consiste à observer le comportement courant de l'ensemble des objets surveillés afin de détecter toute anomalie. Toute déviation par rapport au comportement normal modélisé des objets peut affecter le degré de similarité si :

1. le client a invoqué une opération imprévue dans le chemin déjà entamé dans l'arbre,
2. l'opération invoquée est prévue mais les valeurs de paramètres sont inhabituelles,
3. l'opération invoquée est prévue mais l'émetteur n'est pas reconnu. Dans le cas d'une délégation, c'est la liste des émetteurs qui est non reconnue.
4. l'opération invoquée est prévue mais le temps écoulé depuis l'opération précédente dépasse les délais autorisés.

L'algorithme de détection est le même que pour le chapitre 2 ; nous en rappelons les principales étapes :

- L'IDS reçoit à chaque requête invoquée une ligne d'audit envoyée en temps réel par l'administrateur. A la réception d'une ligne d'audit, un nouveau nœud est créé.
- Si on est au sommet de l'arbre, c'est-à-dire à la connexion d'un client, le degré de similarité associé à ce nœud est égal à 1. Sinon, il est mis à jour en appliquant une éventuelle pénalité au nouveau nœud fonction des règles de détection décrites plus haut.
- En fonction de la valeur du degré de similarité obtenue pour la connexion i et de celle des seuils d'acceptabilité s et s', l'algorithme génère des alertes.

Nous avons mis en œuvre notre approche sur une maquette CORBA avec la délégation de requêtes entre plusieurs objets. La maquette est présentée dans le paragraphe 3.5. Nous avons pu disposer d'un fichier d'audit traçant l'activité des utilisateurs d'une application dans un environnement réparti. Ce fichier a été le point d'entrée pour des tests dont les résultats sont donnés dans le paragraphe 3.6.

3.5 Mise en œuvre de l'approche dans un environnement CORBA

Pour la mise en œuvre de cette approche, nous proposons une nouvelle maquette pour la collecte d'informations, l'apprentissage et la détection. Nous présentons dans le paragraphe 3.5.1 l'architecture retenue pour cette maquette. Nous donnons dans le paragraphe 3.5.2 les mécanismes d'interception mis en place pour la capture d'informations. Nous décrivons, dans le paragraphe 3.5.3, les données discriminantes pour la définition du comportement normal des objets répartis ainsi que le nouveau format d'audit permettant la collecte de ces données au sein de l'architecture retenue.

3.5.1 Architecture de Test

L'architecture retenue pour la maquette, décrite dans la figure 3.8, comprend les entités suivantes :

1. Les utilisateurs de l'application CORBA. Ils invoquent les services de l'application en adressant leur requêtes à un objet client CORBA qui est le point d'entrée de l'application. Ces utilisateurs sont identifiés par leur UID (exemple : $U1$, $U2$). Lors d'une invocation, leurs identifiants sont concaténés au nom de l'objet client CORBA avec lequel ils communiquent pour transmettre les requêtes aux objets servants CORBA.
2. L'objet client CORBA (exemple : *appli*). C'est le point d'entrée de l'application. Il reçoit les requêtes des utilisateurs et les transmet aux objets servants invoqués. La liste des émetteurs d'une requête, émise par l'objet client CORBA vers un objet servant, au nom d'un utilisateur, comprend l'UID de l'utilisateur concaténé au nom du client CORBA. Les identifiants des utilisateurs sont mémorisés lors de l'apprentissage dans l'arbre de comportement. Durant la détection, ces identifiants sont utilisés pour retrouver l'utilisateur initiateur d'une requête.
3. Les objets CORBA servants (cibles), éventuellement clients (si délégation). Ces objets peuvent éventuellement résider sur des machines différentes. Chaque objet CORBA est désigné par un identifiant résultant de la concaténation de son nom et de celui du serveur auquel il appartient (exemple : $A@S1$). La maquette est constituée de deux serveurs : $S1$ contient les objets A et B et $S2$ contient l'objet C.
4. Les agents d'interception, responsables de la collecte des traces d'audit sur les interactions et de la transmission des identifiants des émetteurs de requêtes. Ces agents sont greffés sur les objets CORBA (servants et

clients). Chaque agent greffé sur un objet met en place un intercepteur client (pour les requêtes reçues par l'objet) et un intercepteur serveur (pour les requêtes émises par ce même objet). Ces agents sont capables à la demande de l'administrateur de collecter les informations nécessaires pour le système de détection d'intrusions (apprentissage et détection).

5. L'administrateur de l'application. Son rôle consiste à :
   - télécommander les agents d'interception : création des agents sur chaque objet CORBA, autorisation d'interception et récupération des traces d'audit. L'administrateur et les agents communiquent via des interfaces CORBA. Les interfaces côté «agent» permettent à l'administrateur de paramétrer l'audit (filtrage des informations collectées, lancement/arrêt de l'audit). Côté «administrateur», les interfaces permettent aux agents de s'inscrire auprès de l'administrateur et de lui envoyer leurs traces.
   - sauvegarder les traces d'audit reçues par les agents dans un fichier d'audit.
   - lancer les modules d'apprentissage et de détection de l'IDS.
6. L'IDS qui comprend deux modules : le module d'apprentissage qui génère un modèle de comportement pour les objets à partir du fichier d'audit et le programme de détection qui mesure la déviation entre le comportement appris et le comportement observé et génère des alertes en fonction des critères de détection.

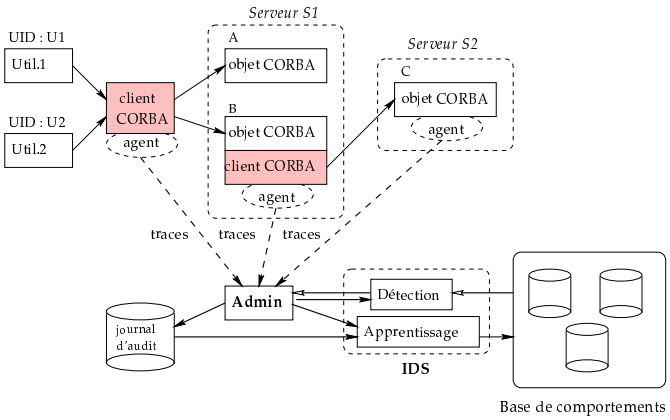

FIG. 3.8 – *Architecture de la maquette*

3.5.2 Collecte d'informations

Dans la collecte des informations sur les interactions entre les différents objets CORBA, quatre contraintes sont posées :

1. L'interception se fait côté serveur, contrairement à la première maquette. En effet, nous modélisons le comportement des objets CORBA et non plus ceux des utilisateurs de l'application.
2. Tous les objets CORBA emetteurs et cibles de requêtes sont observés par les agents d'interception.
3. L'interception se fait de manière transparente aux utilisateurs et sans aucun impact sur le code des objets CORBA, sauf pour capturer l'identifiant de l'utilisateur.
4. Les traces d'audit et la transmission des identifiants des émetteurs permettent de reconstituer les délégations entre les objets.

Pour respecter ces contraintes, nous avons mis en place deux types d'intercepteurs proposés dans l'ORB Visibroker 4.0 : des intercepteurs client (appelés `ClientInterceptors`) et des intercepteurs serveur (appelés `ServerInterceptors`) implantés uniquement côté serveur [1].

Chaque agent d'interception greffé sur un objet CORBA met en place un `ClientInterceptor` et un `ServerInterceptor`. Les `ClientInterceptors` sont actifs pour les objets clients, c'est-à-dire émetteurs (intermédiaires) de requêtes. Ces intercepteurs récupèrent le nom de l'opération invoquée par l'objet émetteur et ajoutent à son nom à la liste des émetteurs.

Les `ServerInterceptor` sont actifs pour les objets cibles de requêtes (intermédiaires ou finaux) de requêtes. Ces intercepteurs récupèrent l'opération reçue par l'objet cible et la liste de ses émetteurs.

Après traitement, tous ces intercepteurs envoient leurs traces à l'administrateur qui se charge de les sauvegarder dans un fichier d'audit.

3.5.3 Nature et format des données d'audit

Pour la modélisation du comportement des objets, nous proposons un nouveau format de traces d'audit qui prend en compte les nouvelles données nécessaires à la construction du profil des objets. Nous donnons ci-après le format de représentation d'une ligne dans le fichier d'audit :

1. Nous rappelons que pour la modélisation du client, nous avons utilisé uniquement des `ServerInterceptor` implantés côté serveur.

```
date:heure objet_cible::nom_operation {liste_demandeurs}
{liste_num_hist} {nom_param type_param val_param} ...
{nom_param type_param val_param}
```

- Tous les items sont séparés par de simples espaces.
- Le premier item donne la date et l'heure d'invocation.
- Le deuxième item est consacré à la cible. Il est obtenu par la concaténation du nom de l'opération invoquée et du nom de l'objet cible. Cette prise en compte est importante lors de la construction du modèle de comportement : une même opération invoquée sur deux objets différents est considérée comme deux invocations distinctes puisqu'elle invoque deux objets distincts.
- La liste des demandeurs nous renseigne sur les émetteurs d'une requête. C'est cette liste qui permet de reconstituer les délégations entre les objets. La liste se réduit à un seul élément lorsque la requête n'est pas issue d'une délégation. Dans le cas contraire, la liste contient le nom de tous les émetteurs intermédiaires dans la chaîne de délégation.
- La liste des numéros historiques est utile en cas de non respect de l'ordre temporel dans le fichier d'audit. Dans ce cas, il est important de réordonner les invocations afin de pouvoir construire la suite d'invocations entre chaque connexion et déconnexion. Pour les requêtes simples, la liste se réduit à un seul numéro. En cas de délégation, la liste comprend autant d'éléments que d'émetteurs de la requête.
- La liste des paramètres de l'opération. Chaque paramètre est un triplet contenant le nom, le type et valeur du paramètre. Chaque triplet est délimité par deux accolades { } et séparé des autres paramètres par un simple espace.

A titre d'exemple, nous considérons l'opération `f(somme float 12.0)` invoquée sur l'objet `appli` le `02/02/02` à `14h15`. La ligne d'audit correspondante est :

```
02/02/02:14:15:00 appli::f {appli} {1} {somme float 12.0}
```

3.6 Test de l'approche sur des données réelles

L'architecture proposée ci-dessus est destinée à être mise en place pour une application réelle de FT R&D. Néanmoins, nous n'avons pas eu accès

à cette application, mais uniquement au fichier d'audit traçant l'activité de l'application durant quatre mois. De ce fait, aucun test n'a pu être réalisé en temps réel.

Nous donnons dans le tableau 3.1 les caractéristiques des données d'audit testées ainsi que celles de la base de comportements construite sur deux périodes de 2 et 3 mois.

Période d'apprentissage	2 mois	3 mois
Taille des fichiers d'audit	2Mo 200Ko	2Mo 400Ko
Nombre de requêtes	9000	10200
Nombre de nœuds	8500	9500

TAB. 3.1 – *Caractéristiques de la Base de Comportements*

Une première série de tests a consisté à modéliser le comportement des objets durant une période d'apprentissage de 2 mois. Le comportement testé en détection porte sur une période de 2 mois. La figure 3.9 donne les degrés de similarités obtenus pour chaque connexion durant la phase de détection (qui porte sur 24 connexions).

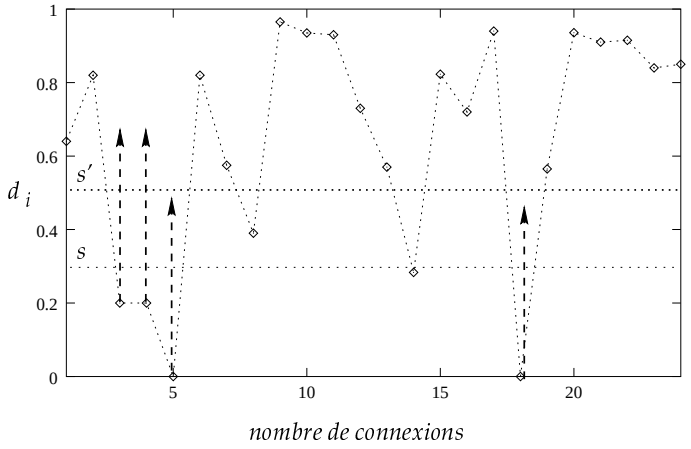

FIG. 3.9 – *Apprentissage 2 mois, Détection 2 mois : 4 alertes instantanées et pas d'alerte composée*

Les figures 3.10 et 3.11 montrent les deux évolutions du degré de similarité au cours d'une connexion. La figure 3.10 illustre une connexion normale (exemple de la connexion 17 de la figure 3.9). La figure 3.11 illustre une connexion anormale (exemple de la connexion 18 de la figure 3.9).

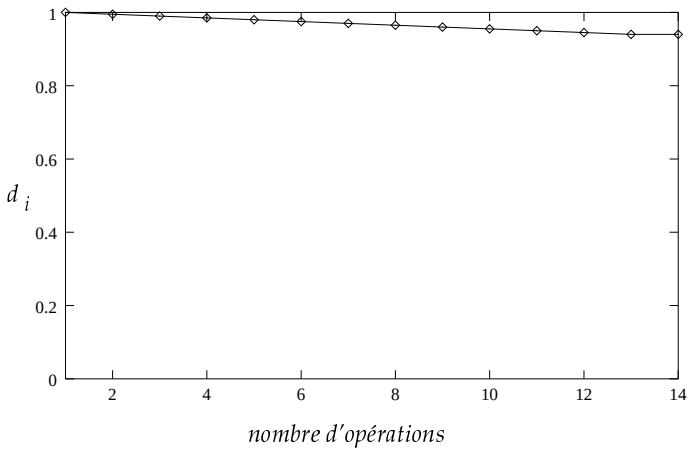

FIG. 3.10 – *Connexion 17 : comportement normal ($d_{final} = 0.94$)*

Le degré de similarité diminue faiblement durant la connexion. En effet, le comportement observé correspond bien à une connexion type déjà apprise, avec de faibles variations des valeurs de paramètres et des délais séparant deux opérations successives.

La figure 3.11 montre l'évolution du degré de similarité pour la connexion 18. Durant cette connexion, le degré de similarité est élevé pour les premières opérations. Ensuite un écart important dans les valeurs observées pour les opérations 4 et 6 conduit à l'annulation du degré de similarité dès la sixième opération. En observant les traces d'audit, nous avons remarqué que pour les opérations 4 et 6, les valeurs observées se situent à l'extérieur des intervalles de tolérance construits pour les paramètres correspondants.

Nous constatons que pour les opérations avec un nombre faible de paramètres, un écart important de l'un des paramètres (traduit par une valeur d'acceptation $va = 0$ et une pénalité $p(va) = 1$) affecte considérablement la pénalité globale de l'oprétaion P et ainsi le degré de similarité. En revanche, pour une opération avec un nombre important de paramètres, le même écart est «dissout» et affecte faiblement le degré de similarité. Il serait donc intéressant d'étudier d'autres opérateurs de conjonction que la moyenne, qui semble être restrictif pour les opérations avec un nombre faible de paramètres.

Nous proposons dans la deuxième série de tests de considérer une période de trois mois pour l'apprentissage et d'un mois pour la détection. Les résultats obtenus sont illustrés dans la figure 3.12. Nous remarquons qu'il n'y a eu

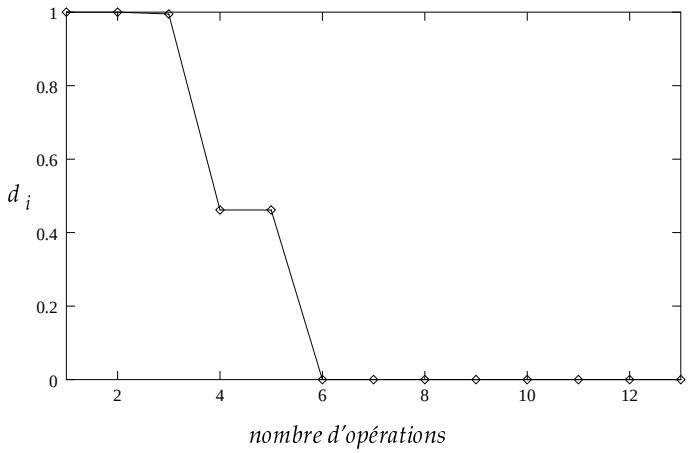

FIG. 3.11 – *Connexion* 18 *: comportement anormal* ($d_{final} = 0.0$)

aucune alerte, ce qui indique que le comportement des objets se stabilise. Néanmois, le mois de test ne comporte que 5 connexions, ce qui ne permet pas de conclure de manière définitive.

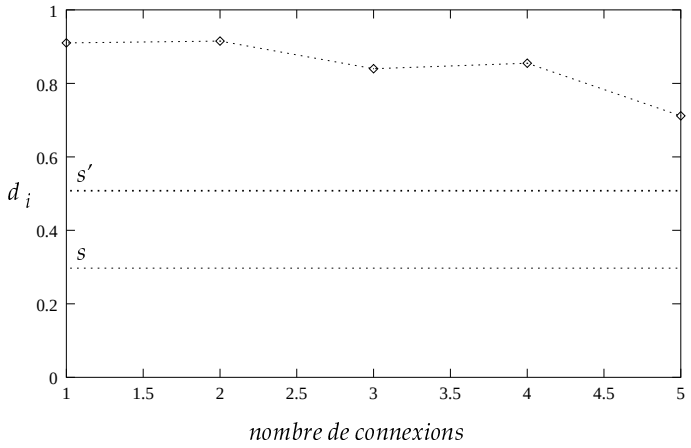

FIG. 3.12 – *Apprentissage 3 mois, Détection 1 mois : pas d'alerte instantanée ni d'alerte composée*

Nous donnons, à titre d'exemple, le détail de la connexion 1 et de la connexion 5, respectivement dans la figure 3.13 et la figure 3.14.

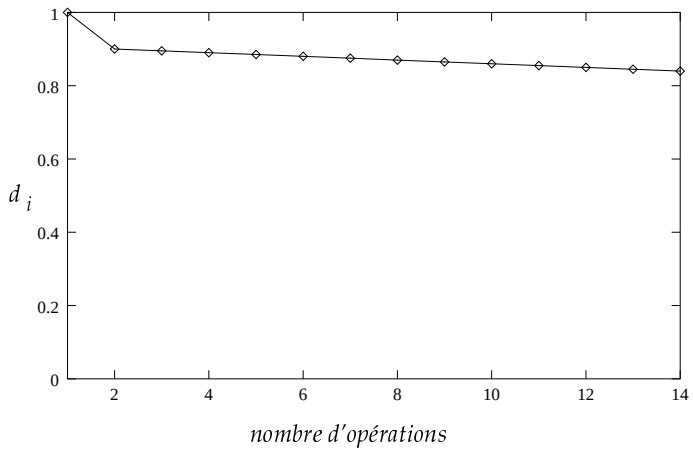

FIG. 3.13 – *Connexion 1 : comportement normal* ($d_{final} = 0.8399$)

3.7 Conclusion

Nous avons proposé dans ce chapitre une modélisation côté serveur du comportement des objets servants d'une application répartie. Le modèle de comportement tient compte, en plus de l'ordre d'invocation des requêtes et des valeurs de leurs paramètres, des délais de temps séparant deux invocations successives et de la délégation de requêtes.

Cette modélisation a été mise en œuvre dans un environnement CORBA et testée sur des données réelles. L'apprentissage a permis de constater que la stabilité de la base de comportement de l'ensemble des objets modélisés est *a priori* obtenue plus rapidement que pour la modélisation des clients.

L'algorithme de détection a donné de bons résultats. En revanche, il est nécessaire de tester l'approche en environnement réel afin d'étudier les points suivants :

- le paramétrage des intervalles de tolérance en fonction de la sensibilité des requêtes et de leurs paramètres vis--vis de la sécurité de l'application à protéger,
- la durée de la phase d'apprentissage,
- la délégation de requêtes et leur incidence sur le comportement des objets,
- la pertinence des alertes générées.

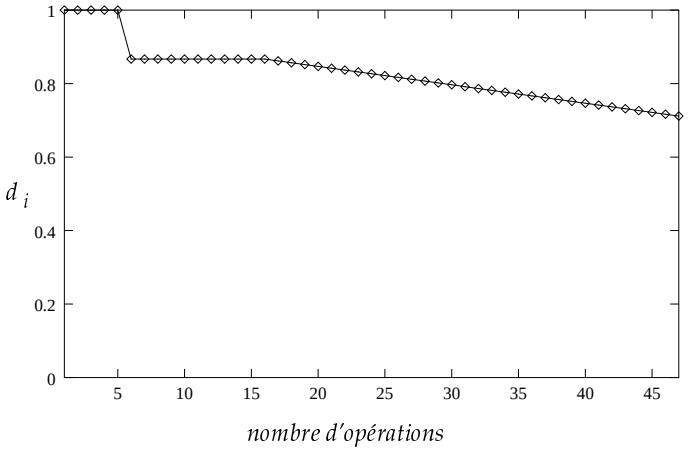

FIG. 3.14 – *Connexion 5 : comportement normal* ($d_{final} = 0.71$)

Chapitre 4

Modélisation statistique du comportement

4.1 Introduction

Nous avons présenté dans les chapitres précédents une modélisation du comportement des clients et des objets servants basée sur une représentation arborescente regroupant des chemins de taille variable. La détection d'anomalie est basée sur la mesure de la déviation du comportement observé par rapport au comportement habituel appris. Dans la mesure de l'écart observé, le modèle autorise de petites variations autour du comportement appris par la construction d'intervalles de tolérance. Nous reprenons dans la figure 4.1 les nuages de points exprimant les fréquences d'observation des valeurs apprises pour un paramètre x considéré ainsi que l'intervalle de tolérance correspondant.

Pour simplifier le modèle, chaque paramètre étudié était représenté, jusqu'ici, par un seul nuage de points auquel correspond un seul intervalle de tolérance construit autour des valeurs minimales et maximales observées durant l'apprentissage. Cette modélisation n'est pas toujours pertinente du fait qu'elle suppose une dispersion homogène des valeurs apprises entre les valeurs minimales et maximales, ce qui n'est pas toujours le cas. Comme le montre la figure 4.1, dans certains cas les points observés sont répartis entre plusieurs nuages, ce qui exprime une particularité dans le comportement modélisé qu'il est important de considérer. Nous proposons donc, dans cette étape de notre étude, d'utiliser une méthode de partitionnement afin de générer automatiquement des nuages de points à partir d'un ensemble d'observations.

Par ailleurs, nous avons évoqué dans l'introduction de ce mémoire le

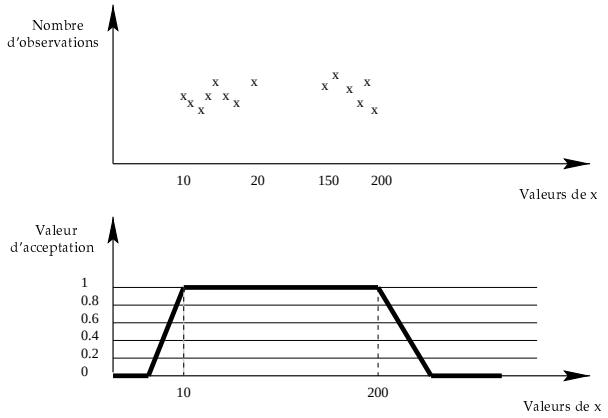

FIG. 4.1 – *Représentation en trapèze du nuage de valeurs d'un paramètre x*

problème des corrélations possibles entre les paramètres d'une même requête. Pour apporter une solution à ce problème, nous proposons simplement de prendre en compte dans une même représentation tous les paramètres numériques d'une même requête. Les nuages de points que nous traitons ne sont donc plus liés à un paramètre[1], mais à une requête.

Une fois les nuages identifiés, nous proposons d'approximer les valeurs observées dans chaque nuage par une loi statistique. En effet, pour des applications réelles avec des volumes importants de données collectées, il est intéressant d'étudier statistiquement la variation des valeurs des paramètres lorsque leur nombre d'observations devient assez important. Pour étudier statistiquement la variation des valeurs, il faut d'abord définir une variable aléatoire à étudier. Nous proposons donc d'associer à chaque requête un vecteur aléatoire *multidimensionnel* de dimension n (n étant le nombre de paramètres numériques à considérer). Les p réalisations du vecteur aléatoire sont construites à partir des p valeurs observées pour les n paramètres numériques de la requête durant la phase d'apprentissage. Ces valeurs sont représentées par un ensemble de p points à n dimensions.

Plusieurs modèles statistiques sont possibles pour décrire un comportement aléatoire (variation des valeurs observées) [34]. Pour étudier statistiquement un vecteur aléatoire, dont les réalisations sont représentées en plusieurs nuages de points, nous utilisons un modèle statistique de mélange de distributions, appelé aussi *mixture model*, permettant de modéliser la distri-

1. Nous rappelons que jusqu'ici les nuages de points étaient considérés séparément pour chaque paramètre et aggrégés à la fin pour le calcul du degré de similarité pour la requête.

bution d'un vecteur aléatoire par la combinaison linéaire des distributions élémentaires. Il s'agit, en d'autres termes, de trouver une approximation de la fonction de densité de probabilité, appelée aussi fonction *pdf* (*probability density function*), de chaque vecteur aléatoire représentant chaque requête.

Nous utilisons un modèle de mélange de distributions général qui est largement utilisé pour modéliser de nombreux phénomènes aléatoires [43]. Ce modèle peut s'exprimer comme suit :

- soit $\bar{y} = [y_1,y_2,...,y_n]$ le vecteur[2] aléatoire à étudier de dimension n,
- soit $\bar{y}_i = [y_{1i},y_{2i},...,y_{ni}]$ la i^{eme} réalisation des n paramètres y_n du vecteur $\bar{y}$,
- soit $\bar{Y} = [\bar{y}_1,\bar{y}_2,...,\bar{y}_p]^T$ le vecteur[3] des p réalisations du vecteur $\bar{y}$,

Le modèle de mélange de distributions pour le vecteur aléatoire $\bar{y}$ est alors exprimé par la combinaison linéaire des *pdf* représentant les distributions élémentaires comme suit [2] :

$$f(\bar{y},\bar{\Psi}) = \sum_{k=1}^{K} \omega_k g(\bar{y},\bar{\theta}_k) \tag{4.1}$$

avec :
g : la *pdf* d'une distribution élémentaire
ω_k : le coefficient de pondération pour la k^{eme} fonction g
$\bar{\theta}_k$: l'ensemble des paramètres pour la k^{eme} fonction g
K : le nombre de fonctions considérées dans le modèle
$\bar{\Psi} = [\omega_1,\omega_2,...,\omega_K,\bar{\theta}_1,\bar{\theta}_2,...,\bar{\theta}_K]$ représente le vecteur des inconnues du modèle : les coefficients de pondération ω_i ainsi que les paramètres de chaque fonction $\bar{\theta}_k$. Ce sont ces paramètres que nous recherchons par ajustement afin d'approximer la fonction *pdf* de $\bar{y}$ par $f(\bar{y},\bar{\Psi})$.

Le vecteur aléatoire $\bar{y}$ de densité de propabilité $f(\bar{y},\bar{\Psi})$ est connu analytiquement mais dont les paramètres $\bar{\Psi}$ sont inconnus (numériquement).

La méthode du maximum de vraisemblance permet de chercher la valeur de $\bar{\Psi}$ qui rend le plus probable les réalisations obtenues pour le vecteur $\bar{y}$.

Un algorithme itératif pour l'optimisation du vecteur inconnu $\bar{\Psi}$ a été proposé [16]. Cet algorithme, appelé algorithme *EM* (*Expectation-Maximization*), utilise le critère de maximum de vraisemblance, appelé aussi *ML*

2. Nous utlisons le symbol $\bar{x}$ pour tout x désignant un vecteur ou une matrice.
3. T désigne la transposée

(*Maximum Likelihood*). Nous avons choisi cet algorithme car, d'une part, il donne de bons résultats d'optimisation [16, 43] et, d'autre part, il est bien adapté aux données incomplètes, ce qui est une propriété intéressante pour les IDS basés sur une approche comportementale. En effet, la pertinence et la stabilité de la base de comportement dépend de la durée de la phase d'apprentissage et de la richesse du comportement à modéliser ; on ne garantit donc pas toujours une description complète du comportement habituel. Il est donc souhaitable que la méthode d'optimisation utilisée soit adaptée pour la généralisation du comportement observé.

Etant donné le modèle de mélange de distributions défini pour l'étude d'un vecteur aléatoire, il reste à partitionner l'ensemble des points observés en nuages de points.

Le modèle de mélange de distributions, défini plus haut, peut être facilement appliqué pour le partitionnement des données observées en considérant un *modèle paramétrique de mélange de distributions* [21]. Dans ce modèle chaque nuage est représenté par une seule distribution élémentaire, ce qui implique une courbe pour chaque nuage. Les distributions élémentaires ne sont pas forcément les mêmes pour tous les nuages : elles peuvent représenter une loi de *Gauss*, une loi de *Student*, etc. Dans ce modèle, chaque distribution élémentaire est fixe mais ses paramètres peuvent changer (moyenne, covariance, etc.).

Nous avons opté pour ce modèle, vu que nos données se limitent à des paramètres numériques simples ; nous souhaitons donc associer à chaque nuage une seule distribution élémentaire dans le modèle de mélange de distributions. Le modèle paramétrique nous semble le plus simple et le plus adapté à notre problème.

Par ailleurs, dans le cas où la distribution de chaque nuage est plus complexe (c'est-à-dire qu'elle ne peut pas être représentée par une seule distribution élémentaire), il existe d'autres approches de partitionnement plus générales permettant de modéliser aussi la distribution de chaque nuage par un modèle de mélange de distributions : l'approche *semi-paramétrique* et l'approche *non paramétrique* [52]. Dans le cas d'une représentation semi-paramétrique, la distribution de chaque nuage est représentée par une combinaison finie de distributions élémentaires. Pour une représentation non-paramétrique, chaque réalisation est utilisée comme le prototype d'une seule distribution élémentaire (le modèle de mélange de distributions a donc le même nombre de distributions élémentaires que de réalisations considérées lors de l'optimisation des paramètres de distribution).

Pour le choix de la loi statistique, nous avons observé plusieurs paramètres

numériques extraits de deux fichiers d'audit contenant des événements collectés sur des périodes de quatre et deux mois. Nous avons ainsi construit pour chaque paramètre le nuage de points correspondant aux fréquences d'observations de ses valeurs durant deux mois et durant quatre mois (voir figure 4.2). Une simple observation des points obtenus nous permet de constater qu'au fur et à mesure que le nombre d'observations augmente, la courbe liant les points prend progressivement l'allure d'une courbe en cloche, qui correspond bien à l'expression d'une loi de *Gauss*. Nous proposons d'approximer les valeurs observées dans un nuage de points par une loi normale, ce qui nous semble le choix le plus approprié pour l'étude d'un comportement aléatoire. En effet, la loi normale offre des propriétés bien adaptées à la représentation de plusieurs comportements aléatoires et au calcul de la variation par rapport à ces comportements.

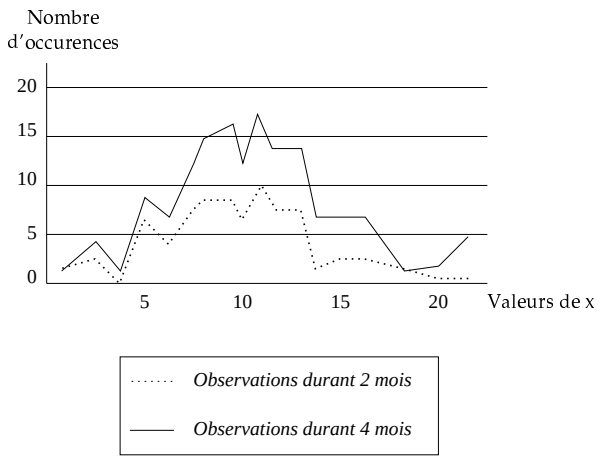

FIG. 4.2 – *Variation des valeurs observées pour le paramètre x*

En résumé, nous proposons dans ce chapitre d'étudier tous les paramètres d'une même requête dans une seule représentation afin de pouvoir représenter et étudier les éventuelles corrélations entre eux. Cette représentation permet d'associer à chaque requête un vecteur aléatoire multidimensionnel. Nous avons choisi de modéliser le comportement de chaque vecteur par une combinaison de lois normales. Dans cette nouvelle modélisation, nous nous fixons les objectifs suivants :

- Représenter chaque requête par un vecteur aléatoire à n dimensions (n étant le nombre de paramètres à considérer). Les observations du vecteur aléatoire durant la phase d'apprentissage sont les valeurs apprises pour les n paramètres de la requête.

- Trouver le nombre optimal de nuages à partir de l'ensemble de points initial.
- Une fois les nuages identifiés, nous proposons d'associer à chaque nuage identifié, une Gaussienne.

Dans la phase de détection, les courbes ainsi obtenues sont utilisées pour mesurer la déviation entre le comportement appris et le comportement observé.

Ce chapitre est organisé comme suit : le paragraphe 4.2 présente la phase d'apprentissage basée sur ce nouveau modèle de représentation des paramètres numériques. Le paragraphe 4.3 présente l'algorithme de détection. Le paragraphe 4.4 donne les résultats de tests réalisés avec ce modèle. Enfin, le paragraphe 4.5 donne nos conclusions pour cette dernière étape de notre travail.

4.2 Apprentissage et algorithme *EM*

Dans cette étape de notre travail, nous nous sommes basés sur le modèle de comportement présenté dans le chapitre 3. La construction de l'arbre de comportement est la même : le comportement habituel est toujours considéré durant une connexion. Nous avons remplacé la construction des intervalles de tolérance par celle de modèles de mélange de gaussiennes en utilisant l'algorithme *EM*.

La première étape consiste à associer à chaque événement observé dans les traces d'audit un vecteur aléatoire $\bar{y}$ multidimensionnel. Nous considérons toujours les paramètres numériques dans une requête lancée par un objet. C'est le nombre de paramètres numériques dans la requête qui définit la dimension du vecteur aléatoire qui lui est associé.

Nous présentons dans le paragraphe 4.2.1 le modèle de mélange de gaussiennes utilisé pour modéliser les paramètres de chaque requête.

Nous détaillons dans le paragraphe 4.2.2 l'algorithme *EM*. Nous donnons dans le paragraphe 4.2.3 les contraintes d'application de l'algorithme *EM* et enfin, dans le paragraphe 4.2.4, nous détaillons la méthode de minimisation d'entropie permettant de trouver le nombre optimal de K, c'est-à-dire le nombre de clusters correspondant au nombre de fonctions élémentaires dans la fonction *pdf*.

4.2.1 Le modèle de mélange de Gaussiennes : *GMM (Gaussian Mixture Model)*

Nous modélisons le comportement de chacun des vecteurs par une fonction *pdf* suivant le modèle de mélange de distributions. Pour la loi statistique nous avons choisi d'utiliser la loi normale (voir paragraphe 4.1). La fonction élémentaire $g(\bar{y},\bar{\theta}_k)$, exprimée dans l'équation 4.1, est alors remplacée par la fonction de distribution d'une loi normale $\phi(\bar{y},\bar{\mu_k},\overline{\Sigma}_k)$. L'expression de la fonction *pdf* du vecteur aléatoire $\bar{y}$ devient alors :

$$f(\bar{y},\bar{\Psi}) = \sum_{k=1}^{K} \omega_k \phi(\bar{y},\bar{\mu_k},\overline{\Sigma}_k) \tag{4.2}$$

avec :
$\phi(\bar{y},\bar{\mu_k},\overline{\Sigma}_k)$: la fonction de distribution d'une loi normale
$\bar{\mu_k}$: représente la moyenne
$\overline{\Sigma}_k$: représente la matrice de covariance
K : le nombre de nuages
$\bar{\Psi} = [\omega_1,\omega_2,...,\omega_K,\bar{\mu_1},\bar{\mu_2},...,\bar{\mu_K},\overline{\Sigma}_1,\overline{\Sigma}_2,...,\overline{\Sigma}_K]$: est le vecteur d'inconnus

Lorsque toutes les fonctions *pdf* élémentaires décrivent une distribution d'une loi normale, le modèle général de mélange de distributions est appelé : *modèle de mélange de gaussiennes* ou *GMM* (pour *Gaussian Mixture Model*).

4.2.2 Algorithme *EM*

L'algorithme *EM*, utilisé pour optimiser le modèle de mélange de distributions, est basé sur l'approche du maximum de vraisemblance afin d'approximer la fonction *pdf* d'ordre K d'un vecteur aléatoire $\bar{y}$. Il suppose connu K, le nombre de fonctions élémentaires pour la fonction *pdf*. L'algorithme *EM* est détaillé dans [16]. Nous détaillons les étapes *E-step* et *M-step* de chaque itération de l'algorithme *EM*.

Le but de la méthode du maximum de vraisemblance (ML) est de trouver une estimation $\bar{\Psi}^*$ pour $\bar{\Psi}$ qui maximise la vraisemblance du vecteur aléatoire $\bar{y}$ pour un ensemble d'observations $\bar{Y} = [\bar{y}_1,\bar{y}_2,...,\bar{y}_p]^T$. Nous supposons que les différentes réalisations $\bar{y}_1$, $\bar{y}_2$, ...,$\bar{y}_p$ du vecteur $\bar{y}$ sont indépendantes[4].

La fonction de vraisemblance logarithmique pour $\bar{\Psi}$ est la suivante :

4. Nous avons fait cette hypothèse afin de faciliter les calcul. Néanmoins, cette hypothèse n'est pas toujours vérifiée en pratique ; nous discutons ce point dans la conclusion de ce chapitre.

$$LogL(\bar{\Psi}) = \sum_{j=1}^{p} Log\left(\sum_{k=1}^{K} \omega_k \phi(\bar{y}_j, \bar{\mu}_k, \overline{\Sigma}_k)\right) \tag{4.3}$$

Une estimation de $\bar{\Psi}$ selon le critère du maximum de vraisemblance est obtenue en trouvant une solution à l'équation suivante (4.4) qui correspond à un maximum local de (4.3) :

$$\frac{\delta LogL(\bar{\Psi})}{\delta \bar{\Psi}} = 0 \tag{4.4}$$

Sachant qu'il est difficile d'optimiser $\bar{\Psi}$ directement, des variables cachées z_{jk} (non observées) sont alors introduites dans l'ensemble des observations tel que z_{jk} est égale à 1 si $\bar{y}_j$ est issu du k^{eme} composant du modèle de mélange de gaussiennes, z_{jk} est égale à 0 dans le cas contraire ($j = 1,...p$; $k = 1,...K$).

Soit $\bar{z}_j = [z_{j1}, z_{j2}, ..., z_{jK}]^T$ le vecteur des variables cachées pour l'observation $\bar{y}_j$. Nous définissons le vecteur de données complètes $\bar{X}$ (non observé) par (4.5) :

$$\bar{X} = [\bar{x}_1^T, \bar{x}_2^T, ..., \bar{x}_K^T] \tag{4.5}$$

tel que :

- $\bar{x}_1 = (\bar{y}_1^T, \bar{z}_1^T)^T, ..., \bar{x}_K = (\bar{y}_K^T, \bar{z}_K^T)^T$ sont des réalisations indépendantes et équitablement distribuées.
- $\bar{z}_1, \bar{z}_2, ..., \bar{z}_p$ sont des réalisations indépendantes issues d'une distribution multinomiale qui consiste à faire un tirage parmi les K cas possibles avec les probabilités respectives $\omega_1, ..., \omega_K$.

Pour cette spécification, la vraisemblance logarithmique pour le vecteur de données complètes est donnée par (4.6) :

$$LogL_c(\bar{\Psi}) = \sum_{k=1}^{K} \sum_{j=1}^{p} z_{jk} log(\omega_k \phi(\bar{y}_j, \bar{\mu}_k, \overline{\Sigma}_k)) \tag{4.6}$$

L'algorithme *EM*, défini dans [16], s'adapte bien lorsque la maximisation de la vraisemblance des données complètes (4.6) est plus facile que la maximisation de la vraisemblance des données incomplètes (4.3). L'algorithme *EM* propose donc de trouver itérativement une estimation de $\bar{\Psi}$ qui donne un maximum local (s'il existe) pour l'équation (4.6). Il consiste à fournir une estimation pour la distribution de $\bar{x} = (\bar{y}^T, \bar{z}^T)^T$ qui permet indirectement d'obtenir une estimation de $\bar{\Psi}$. En d'autres termes, maximiser l'équation (4.3) revient à maximiser l'équation (4.6) (lorsque les observations

sont indépendantes).

L'algorithme propose deux étapes dans chaque itération : l'étape *E-step* (*E* pour *Expectation*) et l'étape *M-step* (*M* pour *Maximization*).

Si on considère $\bar{\Psi}^i$ l'estimation de $\bar{\Psi}$ à la i^{eme} itération, l'étape *E-step* de l'itération $(i+1)$ consiste à calculer (4.7) :

$$Q(\bar{\Psi},\bar{\Psi}^i) = E_{\bar{\Psi}^i}(logL_c(\bar{\Psi})|\bar{Y}) \tag{4.7}$$

où $Q(\bar{\Psi},\bar{\Psi}^i)$ est l'espérance de la vraisemblance logarithmique des données complètes ($logL_c(\bar{\Psi})$) définie dans (4.6), étant données les observations $\bar{Y}$ et l'estimation courante $\bar{\Psi}^i$ de $\bar{\Psi}$.

Etant donné que $logL_c(\bar{\Psi})$ est une fonction linéaire des variables z_{jk} non observables, l'étape *E* est exécutée en remplaçant chaque z_{jk} par son espérance étant données l'observation $\bar{y}_j$ et l'estimation courante $\bar{\Psi}^i$ de $\bar{\Psi}$. L'équation (4.7) est remplacé comme suit par l'équation (4.8) :

$$\tau_k(\bar{y}_j,\bar{\Psi}^i) = E_{\bar{\Psi}^i}(z_{jk}|\bar{y}_j) = \frac{\omega_k^i \phi(\bar{y}_j,\bar{\mu}_k^i,\overline{\sum}_k^i)}{\sum_{k'=1}^K \omega_{k'}^i \phi(\bar{y}_j,\bar{\mu}_{k'}^i,\overline{\sum}_{k'}^i)} \tag{4.8}$$

Nous constatons que $\tau_k(\bar{y}_j,\bar{\Psi}^i)$ exprime l'estimation courante de la probabilité *a priori* que la j^{eme} réalisation $\bar{y}_j$ provienne du k^{eme} nuage ($p(k|\bar{y}_j)$). L'équation (4.8) peut être donc réécrite sachant que :

$$p(k|\bar{y}_j) = \frac{\omega_k^i \phi(\bar{y}_j,\bar{\mu}_k^i,\overline{\sum}_k^i)}{\sum_{k'=1}^K \omega_{k'}^i \phi(\bar{y}_j,\bar{\mu}_{k'}^i,\overline{\sum}_{k'}^i)} = \frac{\omega_k^i p(\bar{y}_j|k)}{p(\bar{y}_j)} = \frac{p(k)p(\bar{y}_j|k)}{p(\bar{y}_j)} \tag{4.9}$$

L'équation (4.9) n'est autre que l'expression du théorème de Bayes tel que l'estimation de la probabilité de chaque composant $p(k)$ est obtenue par l'estimation courante du coefficient de pondération $\bar{\omega}_k^i$.

En combinant l'équation (4.7) et l'équation (4.8), nous obtenons l'expression de l'étape *E-step* suivante :

$$Q(\bar{\Psi},\bar{\Psi}^i) = \frac{\sum_{j=1}^p log \sum_{k=1}^K \omega_k^i \phi(\bar{y}_j,\bar{\mu}_k^i,\overline{\sum}_k^i)}{p} \tag{4.10}$$

équivalente à :

$$L(\Psi^i) = Q(\bar{\Psi},\bar{\Psi}^i) = \frac{\sum_{j=1}^{p} log \sum_{k=1}^{K} \omega_k^i p(\bar{y}_j|k)}{p} = \frac{\sum_{j=1}^{p} log(p(\bar{y}_j))}{p} \quad (4.11)$$

L'objectif de l'étape *M-step* de l'itération $(i+1)$ est de choisir la valeur $\bar{\Psi}^{i+1}$ de $\bar{\Psi}$ qui maximise $Q(\bar{\Psi},\bar{\Psi}^i)$. Il s'agit, en d'autres termes, d'ajuster les valeurs des coefficients de pondération (ω_k^{i+1}), des moyennes $(\bar{\mu}_k^{i+1})$ et des matrices de covariance $(\overline{\sum}_k^{i+1})$ comme suit :

$$\omega_k^{i+1} = \frac{\sum_{j=1}^{p} \tau_k(\bar{y}_j,\bar{\Psi}^i)}{p} \quad (4.12)$$

$$\bar{\mu}_k^{i+1} = \frac{\sum_{j=1}^{p} \tau_k(\bar{y}_j,\bar{\Psi}^i)\bar{y}_j}{\sum_{j=1}^{p} \tau_k(\bar{y}_j,\bar{\Psi}^i)} \quad (4.13)$$

$$\overline{\sum}_k^{i+1} = \frac{\sum_{j=1}^{p} \tau_k(\bar{y}_j,\bar{\Psi}^i)(\bar{y}_j - \bar{\mu}_k^{i+1})(\bar{y}_j - \bar{\mu}_k^{i+1})^T}{\sum_{j=1}^{p} \tau_k(\bar{y}_j,\bar{\Psi}^i)} \quad (4.14)$$

équivalente à :

$$\omega_k^{i+1} = \frac{\sum_{j=1}^{p} p(k|\bar{y}_j)}{p} \quad (4.15)$$

$$\bar{\mu}_k^{i+1} = \frac{\sum_{j=1}^{p} p(k|\bar{y}_j)\bar{y}_j}{\sum_{j=1}^{p} p(k|\bar{y}_j)} \quad (4.16)$$

$$\overline{\sum}_k^{i+1} = \frac{\sum_{j=1}^{p} p(k|\bar{y}_j)(\bar{y}_j - \bar{\mu}_k^{i+1})(\bar{y}_j - \bar{\mu}_k^{i+1})^T}{\sum_{j=1}^{p} p(k|\bar{y}_j)} \quad (4.17)$$

Une caractéristique intéressante de l'algorithme *EM* est de garantir le fait que la vraisemblance logarithmique du modèle de mélange de gaussiennes $L(\bar{\Psi})$ ne peut jamais baisser après chaque itération de l'*EM*. Nous en déduisons que $L(\bar{\Psi}^{i+1}) \geq L(\bar{\Psi}^i)$ ce qui implique la convergence de $L(\bar{\Psi})$ vers L^*, si $L(\bar{\Psi})$ est borné.

Les étapes *E-step* et *M-step* sont répétées plusieurs fois jusqu'à arriver à une variation de l'estimation de la vraisemblance logarithmique (ou des estimations des paramètres) assez petite indiquant la convergence[5] vers L^*. Nous donnons les différentes étapes de l'algorithme :

1. Initialiser $\bar{\Psi}^0$ avec des valeurs aléatoires pour $\omega_1^0,\omega_2^0,...,\omega_K^0,\bar{\mu}_1^0,\bar{\mu}_2^0,...,\bar{\mu}_K^0$, $\overline{\sum}_1^0,\overline{\sum}_2^0,...,\overline{\sum}_K^0$

5. En effet, la diminution de l'écart au cours des itérations est un signe de convergence de l'algorithme.

2. Initialiser $i = 0$ et calculer $L(\Psi^i)$ en évaluant l'équation (4.11), ce qui correspond à l'exécution d'une étape *E-step*
3. Pour $i = i + 1$, calculer la nouvelle valeur de $\bar{\Psi}^i$ en exécutant l'étape *M-step* (équations 4.15, 4.16, 4.17) et de $L(\Psi^i)$ en exécutant l'étape *E-step* (équation 4.11)
4. Si $L(\Psi^i) - L(\Psi^{i-1}) > \delta$ (pas encore convergé), répéter l'étape (3)
5. Mettre à jour les paramètres $\bar{\Psi}^*$ du modèle courant avec ceux obtenus pour le modèle convergé : $\bar{\Psi}^* = \bar{\Psi}^i$

où δ est une constante de convergence.

4.2.3 Contraintes d'utilisation de l'algorithme *EM*

La première difficulté dans l'application de l'algorithme *EM* se situe au niveau de l'équation (4.6) qui admet plusieurs maxima locaux pour un modèle de mélange de gaussiennes. En effet, différentes initialisations peuvent conduire à différents modèles correspondant aux différents maxima obtenus pour la fonction de maximisation de vraisemblance. Le risque lors de l'initialisation de l'algorithme avec des valeurs aléatoires est d'obtenir un maximum local qui ne correspond pas forcément à une solution optimale. Face à ce problème d'initialisation, plusieurs solutions ont été apportées à l'algorithme *EM* [43, 21, 10]. Parmi ces solutions, nous en retenons une qui consiste à exécuter l'algorithme *EM* plusieurs fois et, à chaque exécution, à faire une initialisation aléatoire. Au bout de C_{max} (fixé) exécutions de l'algorithme, nous retenons les paramètres du modèle qui donne la valeur maximale obtenue pour l'étape *E-step* conduisant à la convergence de l'algorithme à chaque exécution.

De plus, dans le cas d'une matrice de covariance non restrictive, la fonction de maximisation de vraisemblance peut tendre vers l'infini (pas de maximum global). C'est le cas, en pratique, des covariances très faibles (proches de zéro mais non nulles) ; les maxima locaux prennent alors des valeurs assez élevées. Il est donc nécessaire de vérifier les coefficients de pondération ω_k et les covariance $\sum_k$ afin d'identifier ces cas particuliers pour les maxima locaux.

Compte tenu de ces contraintes, nous proposons une adaptation de l'algorithme *EM* initialement défini dans le paragraphe 4.2.2. Le nouvel algorithme *EM* adapté est le suivant :

Algorithme *EM* adapté

1. Initialiser le compteur $c = 0$

2. $c = c + 1$
3. Initialiser $\bar{\Psi}^0$ avec des valeurs aléatoires pour ω_1^0,ω_2^0,...,ω_K^0,$\bar{\mu}_1^0$,$\bar{\mu}_2^0$,...,$\bar{\mu}_K^0$, $\overline{\sum}_1^0$,$\overline{\sum}_2^0$,...,$\overline{\sum}_K^0$
4. Initialiser $i = 0$ et calculer $L(\Psi^i)$ en évaluant l'équation 4.11, ce qui correspond à l'exécution d'une étape *E-step*
5. Pour $i = i + 1$, calculer la nouvelle valeur de $\bar{\Psi}^i$ en exécutant l'étape *M-step* (équations 4.15, 4.16, 4.17) et de $L(\Psi^i)$ en exécutant l'étape *E-step* (équation 4.11)
6. Si $L(\Psi^i) - L(\Psi^{i-1}) > \delta$ (pas encore convergé), répéter l'étape (5)
7. si $((|det(\overline{\sum}_1^i)| < \epsilon)||(|det(\overline{\sum}_2^i)| < \epsilon)||...||(|det(\overline{\sum}_K^i)| < \epsilon))$ alors répéter l'étape (2)
8. Si $(c = 1)||(L(\Psi^i) > L_{opt})$ alors
   $L_{opt} = L(\Psi^i)$
   $\bar{\Psi}_{opt} = \bar{\Psi}^i$
9. Si $c \leq C_{max}$ alors répéter l'étape (2)
10. Mettre à jour les paramètres $\bar{\Psi}^*$ du modèle courant avec les paramètres obtenus pour le modèle convergé : $\bar{\Psi}^* = \bar{\Psi}_{opt}$

où δ est une constante de convergence.

4.2.4 Critère de minimisation d'entropie

Pour l'exécution de l'algorithme *EM*, le modèle d'ordre K doit être fourni (K correspond au nombre de clusters dans le cas d'un modèle paramétrique de mélange de distributions pour le partitionnement des données). Etant donné que le nombre de clusters n'est pas connu à l'avance, il faut donc chercher le nombre optimal de clusters.

Notre objectif est de donner une estimation «idéale» du partitionnement des points et de trouver le nombre K optimal des clusters ($k = 1,...K$). Un partitionnement idéal donne, pour chaque réalisation $\bar{y_j}$, une probabilité $p(k|\bar{y_j})$ proche de 1 pour une seule valeur de k (c'est-à-dire pour un seul cluster) et proche de 0 pour toutes les autres (voir équation 4.9).

Le partitionnement idéal pour les données observées peut être obtenu en minimisant l'entropie de *Shannon* [52]. L'entropie est évaluée, pour chaque observation en calculant une mesure H_K comme suit[6] :

$$H_K = -\sum_{k=1}^{K} p(k|\bar{y_j}) log(p(k|\bar{y_j})) \tag{4.18}$$

6. Il est facile de vérifier ce critère par une simple observation de l'expression de l'entropie, donnée en équation 4.19. Nous renvoyons le lecteur vers [52] pour une explication formelle de ce critère.

La valeur espérée de l'entropie est obtenue par le calcul de la moyenne des H_k sur l'ensemble $\bar{Y}$ de toutes les observations comme suit :

$$\widehat{E}(H_K) = -\frac{\sum_{j=1}^{p}\sum_{k=1}^{K} p(k|\bar{y_j})log(p(k|\bar{y_j}))}{p} \tag{4.19}$$

où : $\widehat{E}$ représente un estimateur d'espérance et H_K une mesure de l'entropie.

Pour trouver le nombre K optimal, nous essayons K_{max} modèles avec les différents K possibles (K = 1,2,...,K_{max}) et on évalue pour chaque modèle l'entropie (4.19). Le modèle qui fournit la valeur minimale de H_K est celui qui donne le nombre K optimal de clusters.

Nous donnons ci-après l'algorithme complet permettant d'obtenir le modèle le plus adapté aux données en utilisant l'algorithme *EM* avec une recherche automatique du nombre optimal de K selon le critère de minimisation d'entropie :

Algorithme *EM* adapté pour l'optimisation du nombre de clusters

1. Initialiser : $K = 0$, $K_{opt} = 1$
2. $K = K + 1$
3. Appliquer le modèle d'ordre K aux valeurs apprises en utilisant l'algorithme *EM*
4. Calculer l'espérance de H_K (comme le montre l'équation 4.19)
5. Si $(K == 1)||(H_K < H_{opt})$:
   $H_{opt} = H_K$
   $K_{opt} = K$
   $\bar{\Psi}_{opt} = \bar{\Psi}^*$
6. Si $K < K_{max}$ répéter l'étape (2)
7. Mettre à jour le modèle d'ordre K avec le modèle optimal d'ordre $K = K_{opt}$
8. Mettre à jour les paramètres $\bar{\Psi}^*$ avec ceux du modèle optimal $\bar{\Psi}^* = \bar{\Psi}_{opt}$

K_{max} est le nombre maximal (fixé) de K

4.2.5 Test de l'algorithme *EM*

Plusieurs implémentations ont été proposées pour l'algorithme *EM*. Ces implémentations sont accessibles sur l'internet dans les librairies scienti-

fiques[7]. Parmi ces implémentations, nous citons : Autoclass (en Fortran et C) [10], EMMIX (en Fortran) [43], MCLUST (en environnement S-Plus) [21], etc.

A notre connaissance, il n'existe pas d'implémentation disponible de l'algorithme *EM* en Java (le langage utilisé pour implémenter notre systèmes de détection d'intrusion) pour des modèles de mélange de gausiennes multidimensionnels. Cependant, Akaho a proposé une implémentation en Java de l'algorithme *EM* pour les modèles de mélange de gausiennes bi-dimensionnels [2], qui ne peut être adaptée à des modèles de mélange de gaussiennes sans impact sur la structure du code. Nous avons donc adapté cette implémentation afin de l'appliquer aux modèles de mélange de gaussiennes multidimensionnels.

Nous donnons dans le tableau 4.1 le nombre de lignes de code pour le programme d'apprentissage et de détection en utilisant l'algorithme *EM*.

	nombre de lignes de code
Programme d'apprentissage	2200
Programme de détection	540

TAB. 4.1 – *Nombre de lignes de code pour l'IDS en utilisant le modèle de mélange de gaussiennes*

Deux séries de tests ont été réalisés afin de tester l'algorithme *EM*. La première série a pour objectif de valider l'algorithme proposé sur des données gaussiennes artificielles. Nous illustrons respectivement dans le paragraphe 4.2.5.1 et le paragraphe 4.2.5.2 les résultats des tests obtenus pour des données unidimensionnelles bidimensionnelles.

La deuxième série a pour objectif de tester, d'une part l'algorithme sur des données réelles, et d'autre part le nouvel algorithme de détection proposé pour le modèle de mélange de gaussiennes. Ces tests sont présentés dans le paragraphe 4.4.2, après présentation de l'algorithme de détection.

4.2.5.1 Données uni-dimensionnelles

Nous avons généré des données gaussiennes réparties en 3 clusters. Pour chaque cluster, nous avons généré 50 points. Les caractéristiques de ces trois clusters sont données respectivement dans les tableaux 4.2, 4.3, 4.4.

7. Par exemple : statlib (`http://lib.stat.cmu.edu`)

TAB. 4.2 – *Données uni-dimensionnelles : k = 1 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3293
$\mu[1]$	0.0	0.0066
$\sum[0][0]$	1.0	1.9088

TAB. 4.3 – *Données uni-dimensionnelles : k = 2 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3221
$\mu[1]$	-5.0	-4.9024
$\sum[0][0]$	3.0	1.8993

L'algorithme *EM* a été exécuté avec un nombre maximum de clusters à tester $K_{max} = 4$. Nous donnons dans la figure 4.4 la variation de l'entropie H_K pour chaque nombre de clusters K testé. Nous remarquons que l'entropie est minimale pour $K = 3$, qui est le nombre optimal de clusters.

Les résultats, donnés dans les tableaux 4.2, 4.3, 4.4, montrent que l'algorithme donne une bonne estimation des paramètres de chaque cluster. La figure 4.3 illustre bien la répartition des données initiales en 3 clusters, avec un petit chevauchement entre les clusters.

Nous représentons dans les figures 4.5, 4.6, 4.7, pour chaque cluster k (k, $k = 1..3$), les probabilité $p(k|y_j)$ pour chaque observations y_j. Chaque figure représente sur l'axe des abscisses le nombre d'observations j ($j = 1..p$) et pour chaque j^{eme} observation, la probabilité $p(k|y_j)$ correspondante sur l'axe des ordonnées. $p(k|y_j)$ étant la probabilité pour l'observation y_j d'appartenir au cluster k. Nous remarquons que pour chaque observation y_j, sa probabilité $p(k|y_j)$ est proche de 1 pour le cluster qui donne la meilleure estimation de sa valeur et cette même probabilité est proche de 0 pour les autres clusters.

TAB. 4.4 – *Données uni-dimensionnelles : k = 3 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3456
$\mu[1]$	5.0	5.0596
$\sum[0][0]$	2.0	1.9088

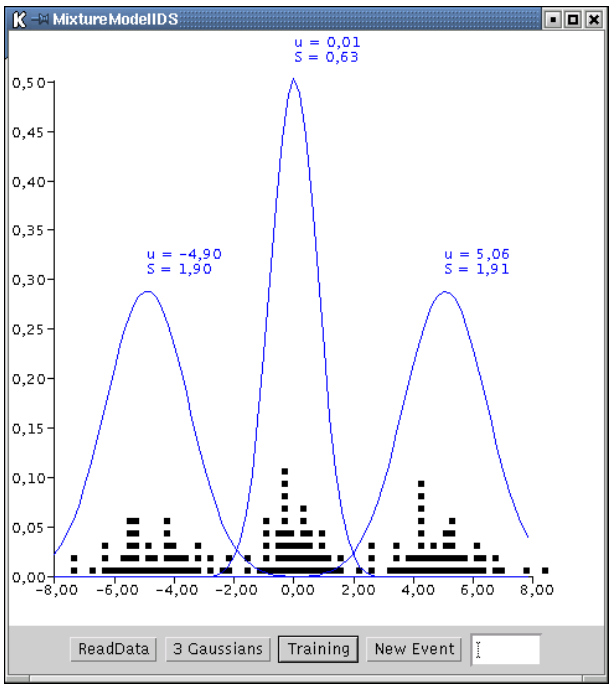

FIG. 4.3 – *Identification du nombre optimal de clusters pour des données uni-dimensionnelle. Nous représentons en abscisse, les observations* $\bar{y}_j$*, et en ordonnées la pdf de chaque cluster* $\phi(y,\mu,\sum)$.

4.2.5.2 Données bidimensionnelles

Pour réaliser ce test, nous avons généré des données gaussiennes bi-dimensionnelles réparties en 3 clusters aussi (comme le montre la figure 4.8). Pour chaque cluster, nous avons généré 50 points. Les caractéristiques de ces trois clusters sont données respectivement dans les tableaux 4.5, 4.6, 4.7.

Les tests ont été effectués pour un $K_{max} = 4$. Nous donnons les valeurs respectives obtenues de H_K pour chaque K ($K = 1...4$) dans la figure 4.9. Nous remarquons que la valeur minimale de H_K correspond à $K = 3$ qui est le nombre optimal de clusters. Dans ce cas aussi, l'algorithme *EM* donne une bonne estimation des paramètres de chaque cluster, comme le montrent les résultats donnés dans les tableaux 4.5, 4.6, 4.7.

Nous vérifions dans les figures 4.10, 4.11, 4.12 que le partitionnement obtenu est optimal avec $K = 3$.

TAB. 4.5 – *Données bidimensionnelles : k = 1 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3208
$\mu[1]$	0.0	0.0416
$\mu[2]$	0.0	-0.0546
$\sum[0][0]$	1.0	0.5272
$\sum[0][1] = \sum[1][0]$	0.5	0.3355
$\sum[1][1]$	2.0	2.2792

TAB. 4.6 – *Données bidimensionnelles : k = 2 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3356
$\mu[1]$	-5.0	-4.7767
$\mu[2]$	-5.0	-5.1376
$\sum[0][0]$	3.0	2.2433
$\sum[0][1] = \sum[1][0]$	0.0	0.1410
$\sum[1][1]$	4.0	4.7055

TAB. 4.7 – *Données bidimensionnelles : k = 3 (50 points)*

Paramètres	Valeur réelle	Estimation
w	0.3333	0.3434
$\mu[1]$	5.0	5.0847
$\mu[2]$	5.0	4.8056
$\sum[0][0]$	2.0	1.5874
$\sum[0][1] = \sum[1][0]$	0.5	0.6866
$\sum[1][1]$	2.0	2.7738

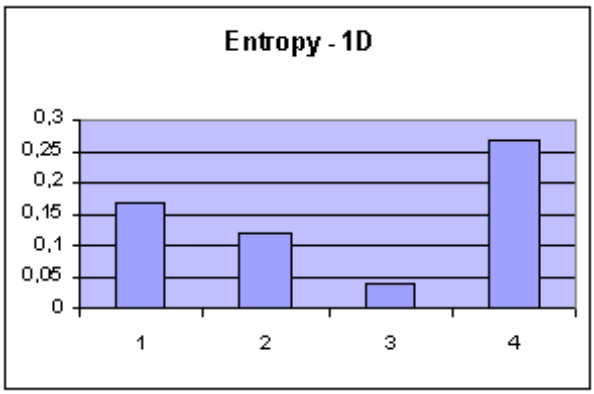

FIG. 4.4 – *Evolution de la valeur de l'entropie : pour* $K = 1$ *:* $H = 0.1681$ *; pour* $K = 2$ *:* $H = 0.1186$ *; pour* $K = 3$ *:* $H = 0.0406$ *; pour* $K = 4$ *:* $H = 0.2677$ *;* $K_{opt} = 3$

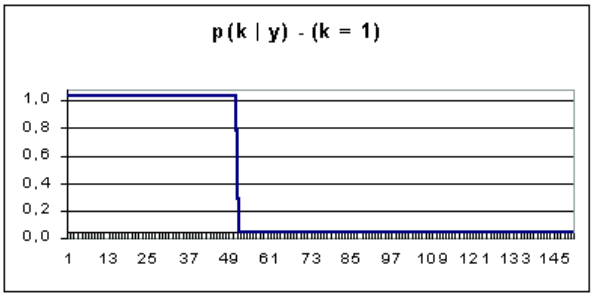

FIG. 4.5 – *Probabilités* $p(k|y_j)$ *pour le premier cluster* $(k = 1)$
Les 50 *premières observations ont des probabilités très proches de* 1*, toutes les autres observations ont des probabilités nulles.*

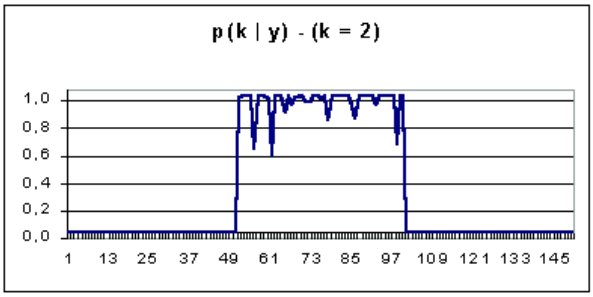

FIG. 4.6 – *Probabilités* $p(k|y_j)$ *pour le deuxième cluster* $(k = 2)$
Entre 50 *et* 100 *observations les probabilités sont proches de* 1*, toutes les autres observations ont des probabilités nulles.*

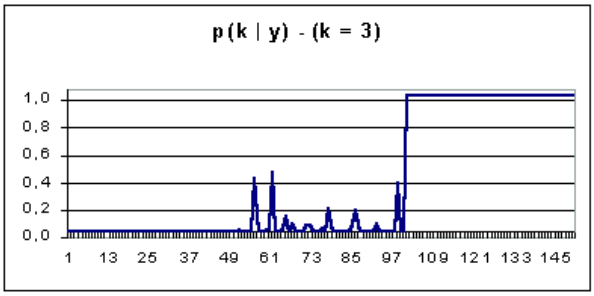

FIG. 4.7 – *Probabilités $p(k|y_j)$ pour le troisième cluster ($k = 3$)*
Les 50 premières observations ont des probabilités nulles, entre 50 et 100 observations les probabilités sont faibles (proches de 0), au delà de la 100^{eme} observation les probabilités sont proches de 1.

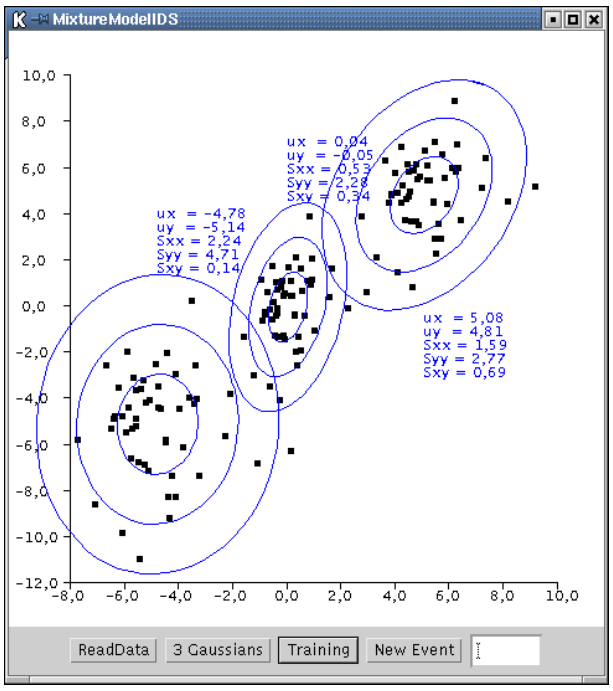

FIG. 4.8 – *Identification du nombre optimal de clusters pour des données bidimensionnelles. Nous représentons en abscisse, les valeurs de la première variable aléatoire $\bar{y}_1$, et dans l'axe des ordonnées les valeurs de la deuxième variable aléatoire $\bar{y}_2$.*

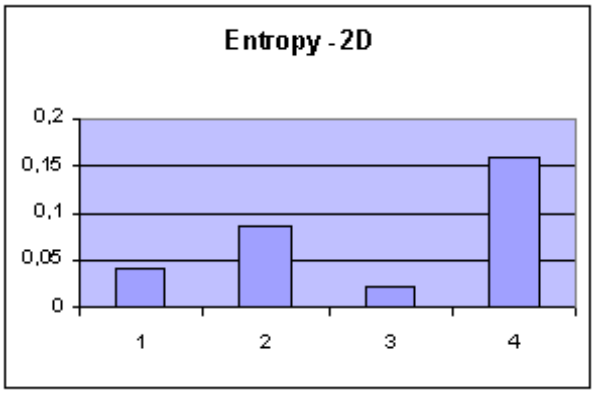

FIG. 4.9 – *Evolution de la valeur de l'entropie : pour* $K = 1$ *:* $H = 0.0414$ *; pour* $K = 2$ *:* $H = 0.0885$ *; pour* $K = 3$ *:* $H = 0.0235$ *; pour* $K = 4$ *:* $H = 0.1602$ *;* $K_{opt} = 3$

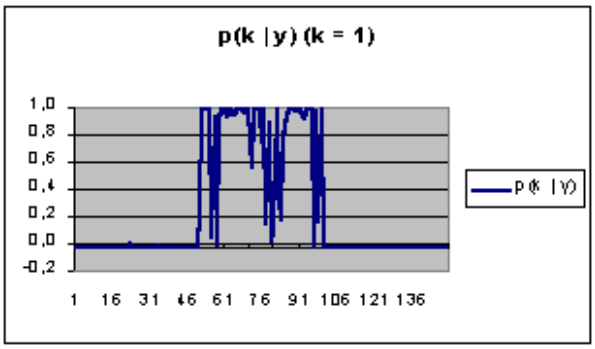

FIG. 4.10 – *Probabilités* $p(k|\bar{y}_j)$ *pour le premier cluster* $(k = 1)$
Les 50 *premières observations ont des probabilités nulles, entre* 50 *et* 100 *observations les probabilités sont proches de* 1*, au delà de la* 100^{eme} *observation les probabilités sont nulles.*

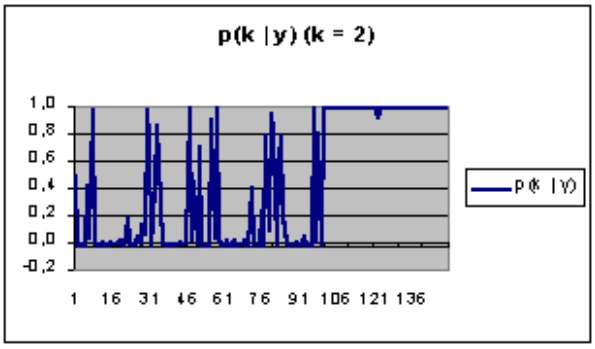

FIG. 4.11 – *Probabilités $p(k|\bar{y}_j)$ pour le deuxième cluster ($k = 2$)*
Les 100 *premières observations ont des probabilités moyennes (entre* 0 *et* 1*), au delà de la* 100^{eme} *observation les probabilités sont nulles.*

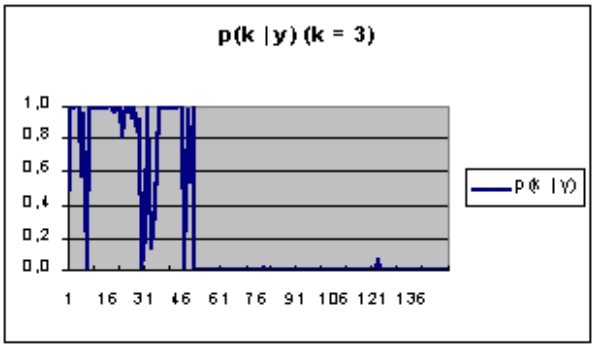

FIG. 4.12 – *Probabilités $p(k|\bar{y}_j)$ pour le troisième cluster ($k = 3$)*
Les 50 *premières observations ont des probabilités nulles, au delà de la* 50^{eme} *observation les probabilités sont moyennes (entre* 0 *et* 1*).*

4.3 Détection et algorithme EM

L'objectif de cette étape est de détecter toute déviation dans le comportement observé par rapport au comportement appris. Le principe de l'algorithme de détection est le même : nous calculons un degré de similarité qui mesure l'écart entre les opérations apprises et les opérations observées durant chaque connexion d'un client. En fonction de la valeur obtenue du degré de similarité et de celles des seuils d'acceptabilité, l'algorithme décide de la génération d'alertes.

Les modifications apportées à l'algorithme proposé dans le chapitre 3 portent sur le calcul du degré de similarité. En effet, nous proposons un nouvel algorithme qui tient compte du modèle de mélange de gaussiennes associé. Nous donnons dans le paragraphe 4.3.1 l'algorithme de détection et nous détaillons dans le paragraphe 4.3.2 le calcul du degré de similarité.

4.3.1 Algorithme de détection

L'algorithme de détection est basé sur les mêmes critères de détection définis dans le chapitre 3. Le comportement est considéré normal s'il correspond à une suite d'opérations normales entre une connexion et une déconnexion du client. Une opération est considérée normale si elle a été invoquée avec des valeurs de paramètres jugées normales. Le comportement est considéré anormal si le client invoque une opération imprévue dans le chemin déjà entamé dans l'arbre de comportement. Dans ce cas, le client est pénalisé en diminuant le degré de similarité. L'algorithme de détection est donné dans la figure 4.13. Nous résumons les principales étapes de l'algorithme comme suit :

- soit d le degré de similarité mis à jour à chaque opération,
- soit α la pénalité associée au degré en cas d'insertion d'opérations,
- soit s le seuil instantané,
- soit s' le seuil composé,
- soit $nbAlerteComp$ le nombre d'alertes composées,
- soit $nbAlerteMax$ le nombre maximum d'alertes composées. Si le degré d reste inférieur au seuil s' $nbAlerteMax$ fois, l'algorithme déclenche une alerte composée.

1. Initialiser $d = 1$ au sommet de l'arbre, $i = 1$
2. Lire l'opération invoquée,
   Si l'opération existe déjà alors :
   - calculer le degré de similarité λ_i pour l'opération courante i (voir

paragraphe 4.3.2)
- calculer la pénalité associée à l'opération $p_i = (1 - \lambda_i)^2$
- mettre à jour le degré de similarité pour l'opération $d_i = d_{i-1} - p_i$
Sinon : $d = d - \alpha$ (pénalisation d'une opération inconnue)
- Si $d_i < s \rightarrow$ alerte instantanée
- Si $d_i < s' \rightarrow nbAlerteComp = nbAlerteComp + 1$
- Si $nbAlerteComp > nbAlerteMax \rightarrow$ alerte composée
- $i = i + 1$

3. tant que (client toujours connecté) répéter l'étape 2

4.3.2 Calcul du degré de similarité

Le degré de similarité est mis à jour au niveau de chaque nœud visité de l'arbre. Nous rappelons que jusqu'ici, la valeur du degré de similarité était mise à jour pour chaque opération par l'application d'une éventuelle pénalité globale affectée à l'opération qui prend en compte des intervalles de tolérance construits pour chaque paramètre numérique.

Nous proposons dans cette étape, un nouvel algorithme pour le calcul du degré de similarité pour chaque opération qui tient compte du modèle de mélange de gaussiennes associé. Le modèle de mélange de gaussiennes tient compte de tous les paramètres numériques de l'opération et rend compte des éventuelles corrélations entre eux. Le calcul du degré se similarité est réparti en deux étapes.

La première étape consiste à calculer, pour chaque observation $\bar{y_j}$, les différentes probabilités $p(k|\bar{y_j})$ que l'observation appartienne à l'un des k clusters ($k = 1,...K$) (en appliquant l'équation 4.9).

La deuxième étape consiste à calculer le degré de similarité de l'observation $\bar{y_j}$ par rapport à chaque cluster (en calculant les probabilités $p(\bar{y_j}|k)$ pour tous les k). Etant donné que les modèles gaussiens utilisés sont de nature continue, il n'est pas pertinent de calculer $p(\bar{y_j}|k)$ par une simple évaluation de la valeur de la fonction *pdf* du cluster ($p(\bar{y_j}|k) = \phi(\bar{y_j},\bar{\mu_k},\overline{\sum_k})$). Une manière plus réaliste pour les données continues est de calculer la probabilité qu'une nouvelle valeur observée appartienne à un sous-espace de valeurs (noté Π_k) défini en fonction des paramètres de la fonction *pdf* du cluster k ($\bar{\mu_k}$ et $\overline{\sum_k}$). Cette probabilité s'exprime comme suit :

$$P(\Pi_k|k) = \int_{\Pi} \phi(\bar{y},\bar{\mu_k},\overline{\sum_k})d\Pi \tag{4.20}$$

Cette probabilité peut être considérée comme la fonction de distribution cumulative *cdf* si l'on défini le sous-espace Π_k comme suit :

$$\Pi_k = \{\bar{y} \in R^n | \frac{\| (\bar{y} - \bar{\mu}_k) \|^2}{\| \overline{\sum}_k \|} \geq \gamma^2\} \tag{4.21}$$

tel que :

- $\frac{\|(\bar{y}-\bar{\mu}_k)\|}{\|\overline{\sum}_k\|}$ représente la distance normalisée entre l'observation et la moyenne.
- γ est une constante qui dépend de chaque observation $\bar{y}_j$

Nous rappelons que dans notre modèle la matrice de covariance pour chaque cluster est non restrictive. De ce fait, les composants inconnus de l'équation 4.21 doivent être bien choisis afin de construire de manière pertinente le sous-espace Π_k. Π_k est construit comme étant l'espace complémentaire de l'isodensité ellipsoïdale (dans l'espace R^n) centrée en $\bar{\mu}_k$ et contenant la valeur $\bar{y}_j$ sur les frontières.

En d'autres termes, le sous-espace Π_k a comme frontière interne une hyper surface ellipsoïdale de dimension n formée par tous les points $\bar{y}$ ayant la même densité que $\bar{y}_j$ ($\phi(\bar{y},\bar{\mu}_k,\overline{\sum}_k) = \phi(\bar{y}_j,\bar{\mu}_k,\overline{\sum}_k)$).

L'équation 4.21 peut être donc réecrite comme suit :

$$\Pi_k = \{\bar{y} \in R^n | \sum_{\alpha\beta} (y_\alpha - \mu_k)_\alpha [\overline{\sum}_k^{-1}]_{\alpha\beta} (y_\beta - \mu_k)_\beta \geq \gamma^2\} \tag{4.22}$$

tel que :

- $\bar{y} = (y_1,y_2,...,y_n)^T$,
- $[\overline{\sum}_k^{-1}]_{\alpha\beta}$ est l'élément de la matrice de covariance inverse correspondant à l'intersection de la ligne α et de la colonne β.
- γ^2 est définie comme suit :

$$\gamma^2 = \sum_{\alpha\beta} (y_{j\alpha} - \mu_k)_\alpha |\overline{\sum}_k^{-1}|_{\alpha\beta} (y_{j\beta} - \mu_k)_\beta \tag{4.23}$$

La figure 4.14 illustre la construction du sous-espace Π dans un espace uni-dimensionnel.

La figure 4.15 illustre la construction du sous-espace Π dans un espace bidimensionnel pour une distribution de gausiennes bidimensionnelle sans corrélation entre les deux paramètres (matrice de covariance diagonale).

Pour une distribution de gaussiennes avec une matrice de covariance non restrictive (paramètres corrélés), il faut procéder à une rotation qui consiste

à trouver la matrice de transformation, formée à partir des vecteurs propres de la matrice de covariance. Cette matrice permet de trouver les directions principales (y'_1 et y'_2) donnant une distribution transformée équivalente à la première distribution mais en supprimant la corrélation entre les paramètres. En effet, la figure 4.16 montre bien que dans le nouveau repère (y'_1,y'_2), la matrice de covariance est restrictive. Les valeurs propres de la matrice de covariance correspondent aux variances de la distribution transformée (σ'_{y1y1} et σ'_{y2y2}) comme le montre la figure 4.16 dans un espace bidimensionnel. Cette procédure de transformation, connue sous le nom d'*Analyse en composantes principales* ou *ACP* [35], s'applique aux distributions gaussiennes multidimensionnelles avec des matrices de covariance non restrictives afin de donner une distribution équivalente non-corrélée [52].

Une fois le sous-espace Π_k trouvé, il faut évaluer la probabilité qu'une nouvelle observation appartienne à cet espace, ce qui revient à évaluer l'équation 4.20. Etant donné que cette équation ne peut être évaluée analytiquement, elle est généralement évaluée numériquement. Néanmoins, dans la pratique, l'évaluation numérique n'est pas aisée surtout dans des espaces multidimensionnels. De plus, elle implique un temps d'exécution trop élevé pour envisager une détection en temps réel [23].

Dans notre cas, nous proposons de contourner ce problème en utilisant une table de probabilité pour une loi normale réduite ($\mu = 0$ et $\sigma = 1$) afin d'évaluer les valeurs approximées des probabilités dans l'équation 4.20. Cette table (donnant les valeurs de la fonction *cdf* complémentaire pour une distribution normalisée unidimensionnelle) est utilisée avec une variable généralisée dans la procédure de recherche des valeurs de probabilités :

$$\gamma_{opt} = \frac{\gamma}{\sqrt{n}} \tag{4.24}$$

Si $\Phi_{(0,1)}(y)$ représente la fonction *cdf* d'une distribution gaussienne unidimensionnelle avec une moyenne $\mu = 0$ et une variance $\sigma = 1$, la procédure de recherche dans la table est la suivante :

Procédure de recherche

1. Pour chaque nouvelle réalisation $\bar{y}_j$, calculer γ_{opt} par application des équations (4.23) et (4.24).
2. Chercher dans la table (donnée en Annexe 4.5) la valeur γ' correspondant à la colonne γ_{opt} la plus proche de γ_{opt} calculé dans l'étape 1.

3. Chercher à la même ligne correspondant à l'étape 2, la colonne $1 - \Phi_{(0,1)}(\gamma) = \Phi(-\gamma)$ pour la valeur de $\Phi_{(0,1)}(\gamma)$.
4. L'approximation de $P(\Pi_k|k)$ est donnée par : $1 - \Phi(\gamma') + \Phi(-\gamma')$

Nous constatons que les probabilités données en appliquant cette procédure sont très faibles même pour $\gamma_{opt} = 1$, ce qui peut être restrictif. En effet, si la procédure donne des valeurs très faibles même pour les valeurs apprises, il est difficile de décider de la normalité d'un comportement observé. Pour ce problème, nous proposons d'utiliser non plus la valeur de $P(\Pi_k|k)$ mais celle $P'_k(\gamma_{opt})$ obtenue comme suit :

$$P'_k(\gamma_{opt}) = \begin{cases} 1 & si \gamma_{opt} \leq \gamma_1 \\ \frac{\Phi(-\gamma_{opt}) - \Phi(\gamma_{opt}) + \Phi(\gamma_2) - \Phi(-\gamma_2)}{\Phi(-\gamma_1) - \Phi(\gamma_1) + \Phi(\gamma_2) - \Phi(-\gamma_2)} & si \gamma_1 < \gamma_{opt} < \gamma_2 \\ 0 & si \gamma_{opt} \geq \gamma_2 \end{cases} \tag{4.25}$$

Le degré de similarité peut être finalement calculé comme suit :

$$\lambda(\bar{y}_j) = \sum_{k=1}^{K} p(k|\bar{y}_j) P'_k(\gamma_{opt}) \tag{4.26}$$

Nous résumons les différentes étapes de l'algorithme de calcul du degré de similarité (donné dans la figure 4.17) comme suit :

1. $k = 0$, $\lambda = 0$
2. $k = k + 1$
3. Evaluer $p(k|\bar{y}_j)$ en utilisant l'équation 4.11 (première étape)
4. Evaluer $P'_k(\gamma_{opt})$ comme dans l'équation 4.25 en utilisant la table de probabilité afin d'obtenir la meilleure approximation pour $\Phi_{(0,1)}(\gamma')$
5. $\lambda += p(k|\bar{y}_j) P'_k(\gamma_{opt})$
6. Si $k < Kmax$ alors répéter l'étape 2
7. Retourner la valeur de λ à l'algorithme de détection.

4.4 Tests de détection

Nous avons effectué deux séries de tests : la première série de tests porte sur des données gaussiennes artificielles, générérées aléatoirement et la deuxième série porte sur des données réelles. Nous les représentons successivement dans le paragraphe 4.4.1 et dans le paragraphe 4.4.2.

4.4.1 Test sur des données artificielles

Les tests de détection ont été effectués sur les données gaussiennes bi-dimensionnelles générées pour tester l'algorithme d'apprentissage (voir figure 4.8).

L'algorithme donne de bons résultats de détection : pour les valeurs proches de la moyenne et appartenant à la première courbe de niveau des clusters, le degré de similarité est proche de 1. Le degré diminue en s'éloignant de la moyenne (compris entre 0 et 1 à l'intérieur des courbes de niveau) et s'annule en dehors des courbes de niveau pour chaque cluster.

Nous illustrons ces résultats dans les figures 4.18, 4.19, 4.20. Dans chacune de ces figures, nous représentons sur l'axe des abscisses, les valeurs observées pour la première variable aléatoire $\bar{y}_1$ et sur l'axe des ordonnées, les valeurs de la deuxième variable aléatoire $\bar{y}_2$. Chaque point de la figure est une observation de dimension 2 correspondant à une observation (avec 2 paramètres) durant la phase d'apprentissage. L'observation testée durant la détection est marquée par un petit carré grisé.

Nous donnons les résultats de détection illustrés dans chacune des figures :

- La figure 4.18 illustre un comportement parfaitement normal pour l'observation $\bar{y}_j = (-4.0, -5.0)$ appartenant à la première courbe de niveau avec un degré de similarité $\lambda = 1.0000$.
- La figure 4.19 illustre un comportement anormal pour l'observation $\bar{y}_j = (4.0, -6.0)$ qui se situe à l'extérieur des courbes de niveau des clusters. Le degré de similarité est $\lambda = 0.0000$.
- La figure 4.20 illustre un comportement jugé normal pour l'observation $\bar{y}_j = (-1.0, -2.0)$ qui se situe au niveau de la 3^{eme} courbe de niveau du 1^{er} cluster mais aussi dans la 2^{eme} courbe de niveau du 2^{eme} cluster, ce qui donne un degré de similarité $\lambda = 0.7873$.

4.4.2 Test sur des données réelles

Les tests de l'approche ont été effectués pour une période d'apprentissage de 2 mois et une période de détection de 2 mois. Pour illustrer les résultats de l'algorithme *EM*, nous proposons de comparer les résultats obtenus pour une connexion jugée normale suivant l'algorithme présenté en chapitre 3 et ceux obtenus avec l'algorithme *EM* pour la même connexion.

La connexion testée (*connexion 1*) est composée d'une suite de 14 opérations dont 8 contiennent des paramètres numériques. Les résultats de détection obtenus pour cette connexion dans le chapitre 3, donnent un degré de

similarité final de 0.8399 (voir figure 3.13). Nous rappelons que la décroissance du degré de similarité de 1 jusqu'à 0.8399 pour cette connexion est due soit à une opération imprévue, soit à une valeur observée d'un paramètre éloignée de celles apprises ; sa valeur d'acceptation *va* est donc faible selon les intervalles de tolérance correspondant au paramètre, ce qui diminue le degré de similarité. Nous constatons, que dans cette connexion, toutes les opérations ont au plus deux paramètres numériques. En d'autres termes, avec un nombre faible de paramètres, une faible valeur d'acceptation de l'un des paramètres affecte considérablement la pénalité associée à l'opération (en calculant la moyenne des valeurs d'acceptations) ce qui implique une faible valeur du degré de similarité. Dans le cas de cette connexion, les valeurs d'acceptation pour chaque paramètre, sont considérées comme assez élevées, ce qui affecte faiblement le degré de similarité.

Nous étudions l'évolution du degré de similarité pour cette même connexion, avec les mêmes critères de détection, mais en considérant, non plus les intervalles de tolérances pour les paramètres numériques, mais les modèles de mélange de gaussiennes pour chaque opération.

Nous donnons, à titre d'exemple, les modèles gaussiens obtenus à l'issue de la phase d'apprentissage ainsi que les observations en phase de détection pour quelques opérations de la *connexion 1*. Les valeurs obtenues pour λ à chaque opération sont très élevées, ce qui indique un comportement observé normal, avec prise en compte des corrélations entre les paramètres.

Nous donnons dans la figure 4.25 la variation du degré de similarité au cours de cette connexion. Le degré de similarité final obtenu est de 0.7186, ce qui est inférieur au degré de similarité obtenu en considérant les intervalles de tolérance, ce qui met en évidence que des valeurs prises isolément peuvent être considérées normales (avec des valeurs d'acceptation très élevées), mais que la corrélation avec d'autres valeurs peut indiquer un écart dans le comportement appris.

Cet exemple illustre bien que même si la connexion est considérée normale en appliquant les deux algorithmes, le degré de similarité obtenu n'est pas le même. En prenant en compte les corrélations entre les paramètres, l'analyse effectuée est plus fine et peut révéler une anomalie qui n'aurait pas été révélée sans l'utilisation des modèles de mélange de gaussiennes.

```
Detect_users(ProfilBase: in, SERVERPORT: in)
S ← 0.3
S' ← 0.5
d ← 1
i ← 0
α ← 0.01
nbAlertesMax ← 2
nbAlertesComp ← 0
BEGIN
action ← read_action(SERVERPORT : in)
user ← get_user(action_user : in)
if (user ∈ ProfilBase) then
  user_profil ← get_Profil(ProfilBase : in, user : in)
  while (action_user ≠ "DISCONNECT") do
    if action_user ∈ user_profil then
      λ_i ← calcul_degre(λ_i : out,action_user, profil)
      p_i ← (1 − λ_i)^2
      d_i ← d_{i−1} − p_i
    else
      d_i ← d_{i−1} − α
    end if
    if (d_i < S) then
      Print ( «Alerte instantanée : Comportement Anormal ! » )
    end if
    if (d_i < S') then
      nbAlertesComp + +
    end if
    action ← read_action(SERVERPORT : in)
    i ← i + 1
    user ← get_user(action_user : in)
    if (user ∈ ProfilBase) then
      user_profil ← get_Profil(ProfilBase : in, user : in)
    end if
  end while
  if (degre ≥ S) then
    Print ( «Comportement normal» )
  end if
  if (nbAlertesComp ≥ nbAlertesMax) then
    Print ( «Alerte composée !!» )
  end if
end if

END
```

FIG. 4.13 – *Algorithme de détection. La fonction* `calcul_degre` *est détaillée dans la figure 4.17.*

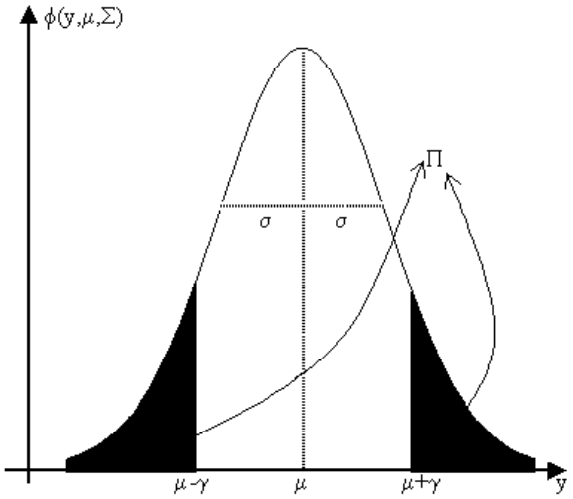

FIG. 4.14 – Π *pour un cluster avec un modèle gaussien uni-dimensionnel*

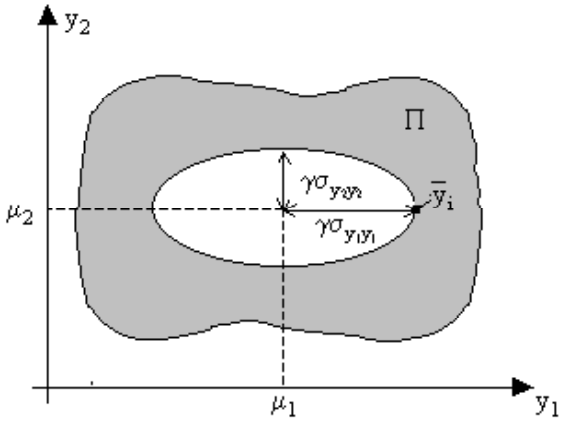

FIG. 4.15 – Π *pour un cluster avec un modèle gaussien bidimensionnel sans corrélation*

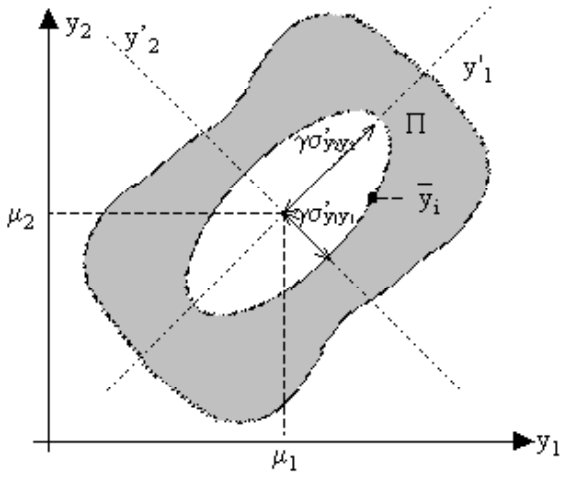

FIG. 4.16 – Π *pour un cluster avec un modèle gaussien bidimensionnel avec corrélation*

calcul_degre ($\lambda_i : out$, $action\_user : in$, $profil : in$)
$k \leftarrow 0$
$\lambda \leftarrow 0$
BEGIN
while $k < K_{max}$ **do**
  $k \leftarrow k + 1$
  calcul_p($k|\bar{y}_j$)
  calcul_$P'_k(\gamma_{opt})$
  $\lambda \leftarrow \lambda + p(k|\bar{y}_j).P'_k(\gamma_{opt})$
end while
return (λ)
END

FIG. 4.17 – *Calcul du degré de similarité associé à l'opération courante.*

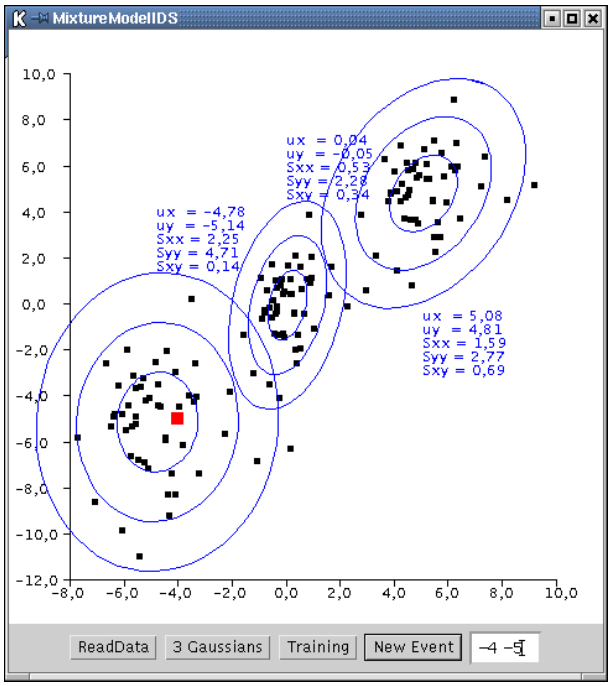

FIG. 4.18 – *Comportement normal :* $\lambda = 1.0000$

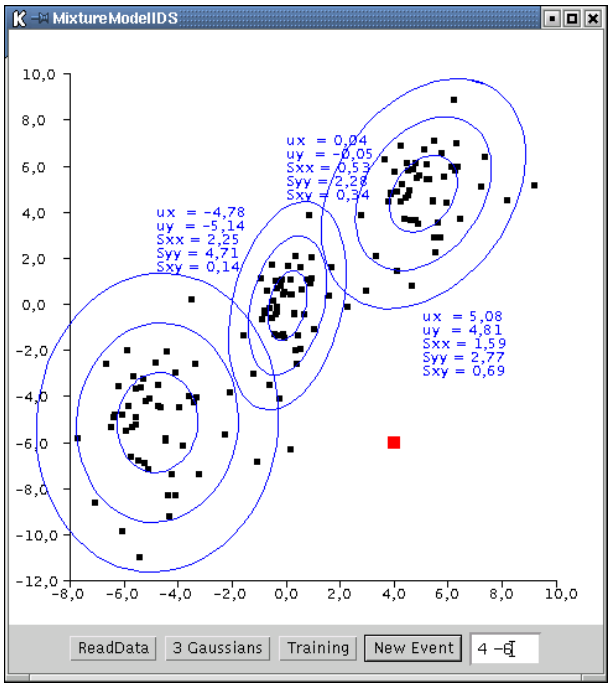

FIG. 4.19 – *Comportement anormal :* $\lambda = 0.0000$

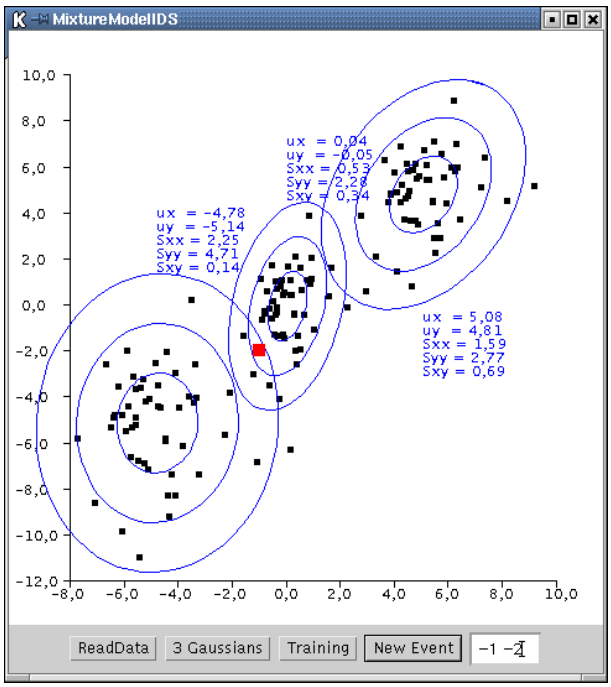

FIG. 4.20 – *Comportement normal :* $\lambda = 0.7873$

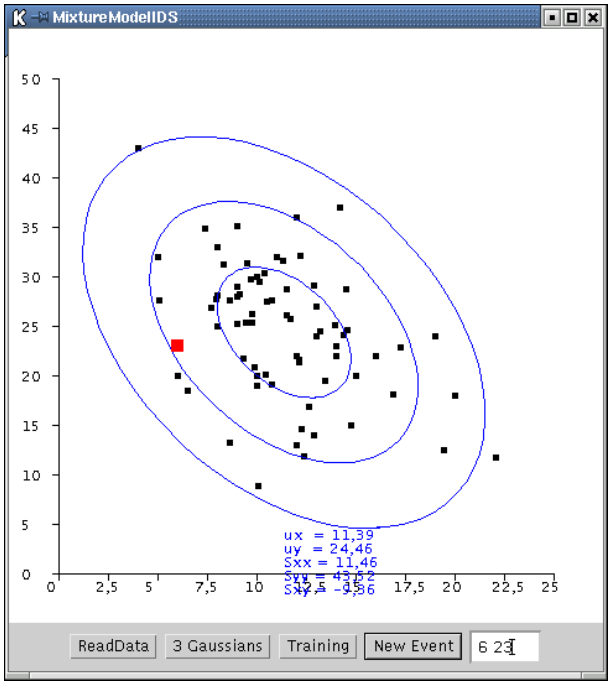

FIG. 4.21 – *Gaussienne bidimensionnelle pour l'opération 1*
Pour une observation de valeur (6,23)*, le degré de similarité* λ *vaut* 0.6448.

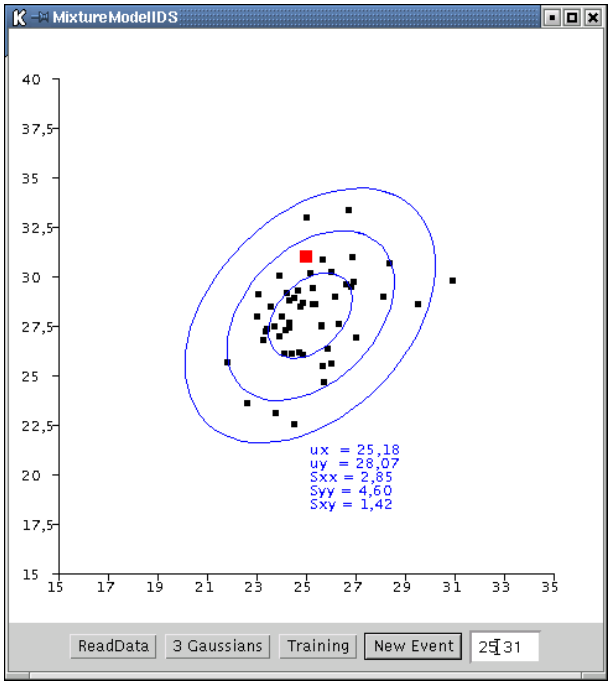

FIG. 4.22 – *Gaussienne bidimensionnelle pour l'opération 6*
Pour une observation de valeur (25,31)*, le degré de similarité* λ *vaut* 0.9647

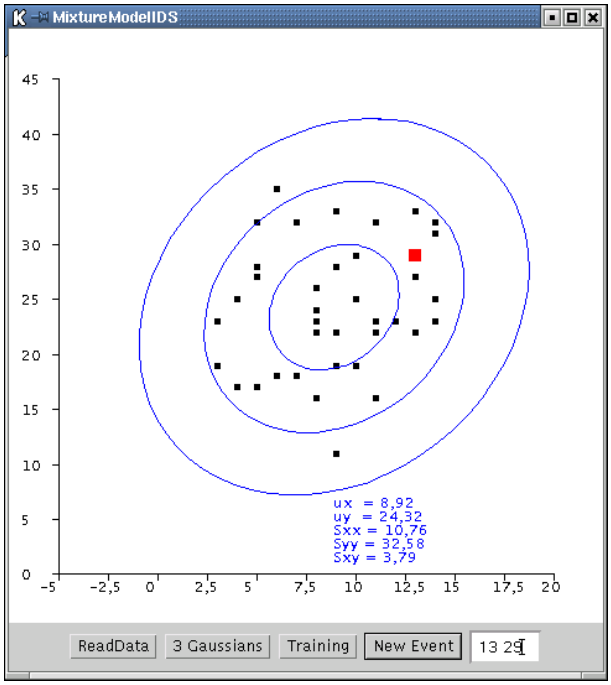

FIG. 4.23 – *Gaussienne bidimensionnelle pour l'opération 8*
Pour une observation de valeur (13,29)*, le degré de similarité* λ *vaut* 1.0

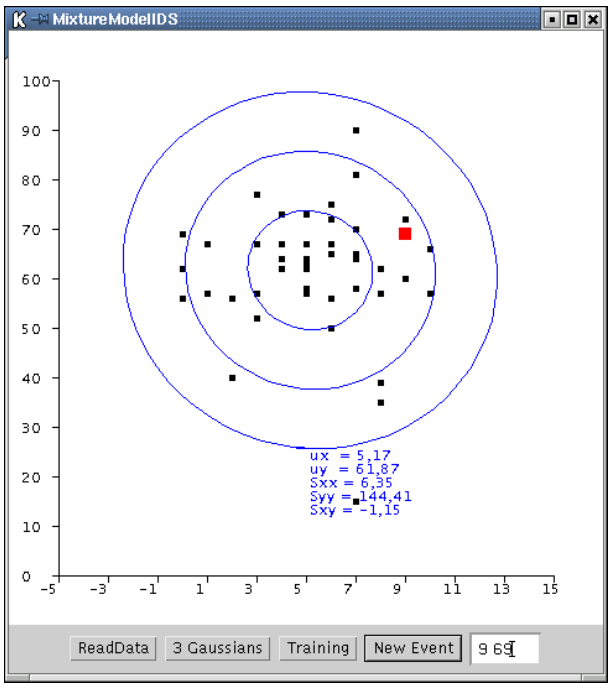

FIG. 4.24 – *Gaussienne bidimensionnelle pour l'opération 10*
Pour une observation de valeur (6,69)*, le degré de similarité* λ *vaut* 0.8392

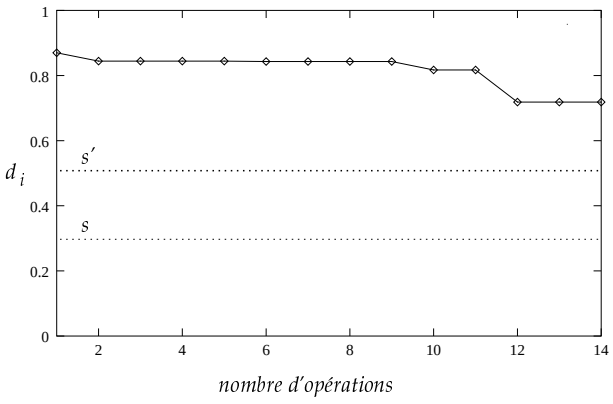

FIG. 4.25 – *Variation du degré de similarité* λ

4.5 Conclusion

Nous avons proposé dans ce chapitre un modèle statistique de mélange de distributions afin d'étudier la dispersion des valeurs apprises pour les paramètres numériques des requêtes des clients d'une application à objets répartis. Ce modèle a été mis en œuvre en utilisant l'algorithme *EM*. Afin d'appliquer l'algorithme *EM* à notre problème de détection d'intrusions, nous avons fixé certaines hypothèses liées à la nature de l'architecture de détection visée, ainsi qu'aux contraintes d'adaptation de l'algorithme *EM*. Cet algorithme, connu par ses performances dans plusieurs domaines (traitement d'images, etc.), a donné de très bons résultats dans le cadre de notre système de détection d'intrusions.

Néanmoins, nous proposons certaines perspectives pour ce travail.

- Nous nous sommes intéressés dans cette étude aux paramètres numériques des requêtes émises par les clients (variables quatitatives). Il serait intéressant d'intégrer dans le modèle gaussien les paramètres symboliques (variables qualitatives), moyennant une normalisation des données.
- Nous avons supposé dans cette modélisation que les différentes réalisations pour chaque vecteur aléatoire sont indépendantes, ce qui revient à supposer que les différentes observations des valeurs des paramètres d'une même requête sont indépendantes. Dans la pratique, rien ne garantit que cette hypothèse soit vérifiée. Une généralisation du modèle consisterait à prendre en compte les corrélations entre les différentes observations d'une même requête en appliquant un modèle de Markov, par exemple.

Conclusion

L'objectif de cette thèse est la détection des comportements malveillants des clients d'une application à objets répartis. A cette fin, nous avons proposé une approche de type comportementale, qui consiste à modéliser le comportement normal des clients, observés durant une première phase, dite d'apprentissage, puis à mesurer la déviation entre le comportement appris et le comportement observé, durant une deuxième phase, dite de détection.

Cette approche présente les avantages suivants :

- La modélisation, basée sur des séquences de tailles variables des invocations réalisées par un client, donne une représentation significative du comportement du client. Cette représentation, proposée dans le chapitre 2, donne de meilleurs résultats de détection que les méthodes basées sur des séquences de taille fixe, comme l'ont montré [14, 40].
- La modélisation du comportement des objets servants d'une application, proposée dans le chapitre 3, permet de rendre compte des effets d'un comportement anormal des clients sur celui des objets invoqués, notamment par la prise en compte de la délégation des requêtes entre objets.
- L'application du modèle de mélange de gaussiennes utilisant l'algorithme *EM*, présenté dans le chapitre 4, montre que les données testées sont de nature gaussiennes, ce qui confirme notre hypothèse de départ. Le modèle montre, en outre, que ces données sont bien réparties en plusieurs clusters, ce qui confirme l'hypothèse de comportement variable d'un client pour une même requête. Enfin, ce modèle a permis de rendre compte des corrélations entre certains paramètres d'une même requête, ce qui permet d'affiner le diagnostic de comportement anormal.
- L'algorithme de détection proposé exige peu de temps de calcul (complexité en $O(n)$). Il est donc envisageable de l'utiliser en temps et environnement réels.

En revanche, l'approche que nous proposons recèle quelques insuffisances :

- Nous avons audité les requêtes invoquées par les clients de manière distribuée, mais l'analyse est réalisée de manière centralisée, ce qui implique un flux de données important pour une application réelle[8]. Pour contourner cette difficulté, les outils de détection d'intrusion vont de plus en plus vers un filtrage des traces d'audit à la source, une décentralisation de leur analyse et une coopération des résultats d'analyse à un plus haut niveau. Ce n'est pas l'approche que nous avons suivie.
- La modélisation du comportement des clients trouve ses limites pour des applications très riches avec des comportements de clients très diversifiés ou évolutifs. C'est une faiblesse inhérente à l'approche comportementale.
- Faute de données réelles suffisantes, les tests de l'approche ont été limités à une détection *offline*.

Les travaux présentés dans ce mémoire ont été réalisés en 3 étapes. Dans un premier temps, nous avons modélisé le comportement des clients d'une application en considérant les requêtes invoquées par ces clients comme étant les données discriminantes pour la définition d'un comportement normal. Ce travail fait l'objet du chapitre 2. Les requêtes sont considérées entre chaque connexion et déconnexion d'un client permettant de construire un modèle de comportement de référence sous la forme d'un ensemble de chemins de taille variable au sein d'une structure arborescente. Dans ce modèle, nous prenons en compte, en plus des signatures des requêtes invoquées, les valeurs prises par leurs paramètres. Nous autorisons ainsi des déviations autour de ces valeurs ce qui permet, d'une part d'offrir une certaine liberté au client de dévier du comportement appris et, d'autre part, de limiter les fausses alertes. Nous définissons une limite à partir de laquelle la valeur observée est jugée trop éloignée des valeurs apprises. Cette limite dépend essentiellement du niveau de sécurité associé au paramètre. Ce niveau doit être fourni par l'administrateur.

L'algorithme de détection proposé pour tester ce modèle consiste à calculer une mesure, appelée *degré de similarité* qui exprime la déviation entre les valeurs observées et celles apprises. En fonction de la valeur du degré de similarité et des seuils d'acceptabilité, l'algorithme de détection décide du déclenchement d'alertes.

L'algorithme a montré de bons résultats sur des données réelles, mais la pertinence des alertes générées reste à étudier en exploitation réelle. Par

8. Nous signalons qu'il s'agit dans notre cas d'un audit applicatif, *a priori* moins volumineux qu'un audit de requêtes systèmes.

ailleurs, pour des applications réelles, les comportements des clients deviennent trop riches et le modèle par client trouve ses limites. Nous nous sommes donc orienté, dans une deuxième étape, vers une modélisation des objets de l'application à protéger.

La deuxième étape de notre travail, exposée au chapitre 3, consiste donc à modéliser le comportement des objets servants d'une application vis-à-vis des requêtes reçues d'un ensemble de clients. Cette modélisation fait abstraction du comportement de chaque client, mais prend en compte l'ensemble des invocations des clients pour modéliser l'activité des objets servants. Dans cette étape, nous enrichissons la modélisation proposée dans le chapitre 2 en introduisant dans la définition d'un comportement normal le délai de temps séparant deux invocations successives. Comme pour les valeurs des paramètres numériques, nous autorisons une déviation par rapport aux délais entre deux invocations successives. L'algorithme de détection prend donc en compte les critères suivants :

- les signatures des requêtes invoquées,
- pour chaque requête, les valeurs prises pour ses paramètres,
- l'ordre d'invocation des requêtes entre chaque connexion et déconnexion,
- le délai de temps séparant deux invocations successives.

Nous proposons dans la dernière étape de notre travail d'automatiser la génération de sous-nuages à partir d'un ensemble de valeurs observées et d'étudier les éventuelles corrélations entre les paramètres d'une reuqête. La dernière étape de notre travail a pour objectif d'étudier statistiquement la dispersion des valeurs de chaque paramètre, en appliquant un modèle de mélange de gaussiennes qui s'est avéré bien adapté à la nature des données testées. Nous avons opté pour une représentation statistique qui considère l'ensemble des paramètres d'une même requête, afin de rendre compte des éventuelles corrélations entre eux. Nous avons alors utilisé un modèle de mélange de gaussiennes multidimentionnel de dimension n (n étant le nombre de paramètres numériques considérés dans une requête). Dans ce modèle, les valeurs apprises pour les n paramètres peuvent se répartir en plusieurs classes. Nous avons mis en œuvre ce modèle en utilisant l'algorithme *EM*, permettant de faire une classification des données observées et d'associer à chaque classe identifiée un modèle gaussien multidimentionnel permettant de donner la meilleure estimation pour le sous-ensemble de données. Cet algorithme, bien connu pour ses performances en classification et en caractérisation des comportements avec des distributions complexes, a donné de bons résultats dans plusieurs domaines (traitement d'images, etc.). Il n'avait, à notre connaissance, jamais été utilisé en détection d'intrusion. Il donne là aussi de très bons résultats.

Notre travail ouvre des perspectives donnant lieu à des études complémentaires dans les directions suivantes :

- Le paramétrage des niveaux de sécurité pour les opérations et les paramètres.
- Alors que notre travail a été orienté vers l'étude des paramètres numériques des requêtes émises par les clients, il serait intéressant d'intégrer dans le modèle gaussien les paramètres symboliques, moyennant une normalisation des données.
- Notre hypothèse pour les modélisations repose sur le caractère indépendant des différentes réalisations pour chaque variable aléatoire. Celà revient à supposer que les différentes observations des valeurs des paramètres d'une même requête sont indépendantes. Il est possible, dans la pratique, que cette hypothèse ne soit pas toujours vérifiée. Une généralisation du modèle consisterait à prendre en compte les corrélations entre les différentes observations d'une même requête en appliquant par exemple le modèle de Markov qui suppose que la réalisation courante dépend de la réalisation précédente. Il serait, en outre, intéressant d'appliquer ce modèle à l'algorithme *EM* dans un système de détection d'intrusion.
- Pour une application en environnement réel, nous proposons d'approfondir les points suivants :
  - L'étude de la durée de la phase d'apprentissage et de la stabilité de la base de comportements.
  - L'étude de la pertinence des alertes générées par notre algorithme de détection.
  - La possibilité de faire un apprentissage continue, sachant qu'il permet de prendre en compte de nouveaux comportements sans pour autant générer d'alerte mais au risque d'apprendre un comportement intrusif.
- Nous proposons enfin d'enrichir notre système par une stratégie de réaction après détection. En effet, un système sécurisé ne se contente pas de déclencher des alertes, mais doit être capable d'identifier les intrusions et de réagir en conséquence. Il y a donc lieu d'envisager des réactions de type : *black-list*, coupure de connexion, leurre du client, contre-attaques, etc.

Pour conclure, l'apport de notre travail se situe au niveau de la modélisation du comportement normal. La modélisation proposée prend en compte des séquences de requêtes de taille variable et la répartition de

leurs paramètres. Elle vise ainsi à traduire le plus fidèlement possible le comportement normal observé.
Une telle modélisation, proposée pour les environnements à objets répartis, peut être mise à profit pour des environnements de types différents.

Bibliographie

[1] Software AG. EntireX DCOM under UNIX. http://www.Software AG.com/entirex, 2000.

[2] S. Akaho. Mixture model for image understanding and the EM algorithm. Technical Report TR-95-13, National Institute of Advanced Industrial Science and Technology (AIST), 1995.

[3] A. Alireza, U. Lang, M. Padelis, R. Schreiner, and M. Schumacher. The Challenges of CORBA Security. In *Proceedings of The GI Workshop Sicherheit in Mediendaten*, September 2000.

[4] J.P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980.

[5] D. Barbara, N. Wu, and S. Jajodia. Detecting Novel Network Intrusions Using Bayes Estimators. In *Proceedings of the first SIAM International Conference on Data Mining (SDM 2001)*, 2001.

[6] K.P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[7] M. Bishop. A standard audit trail format. In *Proceedings of the Eighteenth National Information Systems Security*, pages 136–145, 1995.

[8] G. Booch. *Conception orientée objets et applications.* Addison-Wesley, 1992.

[9] CERT. Results of the Distributed-Systems Intruder Tools Workshop. Technical report, CERT coordination Center, November 1999.

[10] P. Cheeseman and J. Stutz. Bayesian Classification (AutoClass): Theory and Results. In U.M. Fayyad, G. Piatetsky-Shapiro, P.Smith, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180, 1996.

[11] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon press computing series, 1991.

[12] D.A. Curry and H. Debar. Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Docu-

ment Type Definition. http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-06.txt, June 2002.

[13] D. Curtis. RMI, IIOP et EJB. http://www.borland.com, Avril 1999.

[14] H. Debar, M. Dacier, M. Nassehi, and A. Wespi. Fixed vs. Variable-Length Patterns for Detecting Suspicious Process. In *Proceedings of the 1998 ESORICS Conference*, number 1485 in LNCS, pages 1–16, september 1998.

[15] C. Delobel, C. Lecluse, and P. Richard. *Bases de données, des systèmes relationnels aux systèmes à objets.* InterEditions, 1991.

[16] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of Royal Statistical Society*, B(39):1–38, 1977.

[17] D.E. Denning. An Intrusion-Detection Model. *IEEE transaction on Software Engineering*, 13(2):222–232, 1987.

[18] P. D'haeseleer, S. Forrest, and P. Helman. An Immunological Approach to Change Detection: Algorithms, Analysis and Implications. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy.* IEEE Computer Society, IEEE Computer Society Press, 1996.

[19] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A Common Intrusion Specification Language (CISL). Common Intrusion Detection Framework (CIDF), http://www.isi.edu/brian/cidf/drafts/language.txt, Mars 2000.

[20] B.S. Feinstein, G.A. Matthews, and J.C.C. White. The Intrusion Detection Exchange Protocol (IDXP). http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-04.txt, January 2002.

[21] C. Fraley and A.E. Raftery. MCLUST: Software for Model-Based Cluster and Discriminant Analysis. Technical Report 342, Department of Statistics, University of Washington, 1998.

[22] J.M. Geib, C. Gransart, and P. Merle. *CORBA des concepts la pratique.* Dunod, 1999.

[23] A. Genz. Numerical Computation of Multivariate Normal Probabilities. *Journal of Computer Graph Statistics IEEE*, 1:141–149, 1992.

[24] A. Goldberg and D. Robson. *SMALLTALK-80, the language and its implementation.* Addison-Wesley, 1983.

[25] OMG (Object Management Group). Security Service Specification. http://www.omg.org, Novembre 1995.

[26] OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification, 2.0. http://www.omg.org/technology/documents/corba_spec_catalog.htm, Juillet 1995.

[27] OMG (Object Management Group). CORBA Firewall Security. ftp://ftp.omg.org/pub/docs/orbos/98-07-03.pdf, Mai 1998.

[28] OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification, 2.2. http://www.omg.org, Février 1998.

[29] OMG (Object Management Group). Security Service Specification, 1.7. http://www.omg.org/technology/documents/formal/security_service.htm, Mars 2001.

[30] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule- Based Language for Universal Audit Trail Analysis. In *Proceedings of the Second European Symposium on Research in Computer Security (ESORICS'92)*, pages 435–450, 1992.

[31] H.Debar, M.Dacier, and A.Wespi. A Revised Taxonomy for Intrusion-Detection Systems. *Annales des Télécommunications*, 55(7-8):361–378, 2000.

[32] ISO. Information Processing Systems – OSI Reference Model – Part 2: Security Architecture, International Standards Organization, 1989.

[33] H.S. Javitz, A. Valdes, T.F. Lunt, A. Tamaru, M. Tyson, and J. Lowrance. Next Generation Intrusion Detection Expert System (NIDES). Technical Report A016-Rationales, SRI, 1993.

[34] R.A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, 1998.

[35] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 2nd edition, 2002.

[36] S. Kumar and E.H. Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report CSD-TR-94-013, Purdue University, June 1994.

[37] J. Kuri, G. Navarro, L. Mé, and L. Heye. A Pattern Matching Based Filter for Audit Reduction and Fast Detection of Potential Intrusions. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 17–27, October 2000.

[38] U. Lang, D. Gollmann, and R. Schreiner. Security Attributes in CORBA. In *Proceedings of The IEEE Symposium in Security and Privacy*, 2001.

[39] T.F. Lunt and R. Jagannathan. A Prototype Real-Time Intrusion-Detection Expert System. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 59–66, 1988.

[40] C. Marceau. Characterizing the Behavior of a Program Using Multiple-Length N-grams. In *Proceedings of the New Security Paradigms Workshop 2000*, September 2000.

[41] C. Marceau, M. Stillerman, M. Stillman, and S. Forrest. Research in Computer Security and Immunology for ORBs. Technical Report TM-99-0007, Odyssey Research Associates, August 1999.

[42] Z. Marrakchi, L. Mé, B. Vivinis, and B. Morin. Flexible Intrusion Detection Using Variable-Length Behavior Modeling in Distributed Environment: Application to CORBA Objects. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, number 1907 in LNCS, pages 130–144, October 2000.

[43] G.J. McLachlan, D. Peel, K.E. Basford, and P. Adams. THE EMMIX Software for the fitting of mixtures of normal and t-components. *Journal of statistical software*, 4, 1999.

[44] L. Mé. GASSATA, A Genetic Algorithm as an Alternative Tool for Security Audit Trails Analysis. Web proceedings of the First international workshop on the Recent Advances in Intrusion Detection (RAID'98), http://www.raid-symposium.org/raid98, September 1998.

[45] L. Mé, Z. Marrakchi, C. Michel, H. Debar, and F. Cuppens. La détection d'intrusions : les outils doivent coopérer. *Revue de l'Electricité et de l'Electronique*, 5:56–59, May 2001.

[46] B. Meyer. *Conception et programmation par objets.* InterEditions, 1990.

[47] B. Meyer. *EIFFEL, the language.* Prentice Hall Object-Oriented Series, 1992.

[48] Microsoft. DCOM. http://www.microsoft.com/com/tech/DCOM.asp, 1999.

[49] P. Niemeyer, J. Peck, and E. Dumas. *Java par la pratique.* O'reilly, 1996.

[50] R. Orfali and D. Harkey. *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, Inc., 1998.

[51] J. Pan, Y. Huang, R. Gruber, and M.L. Jian. Robustness Testing and Hardening of CORBA ORB Implementations. In *Proceedings of The International Conference on Dependable Systems and Networks*, June 2001.

[52] S.J. Roberts, R. Everson, and I. Rezek. Maximum Certainty Data Partitioning. *Pattern Recognition*, 33(5):833–839, 1999.

[53] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani, and W. Lorensen. *OMT, modélisation et conception orientées objet.* Prentice Hall, 1995.

[54] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice Hall, 1987.

[55] R.M. Soley and C.M. Stone. *Object Management Architecture Guide, 3rd Edition*. John Wiley & Sons, Inc., 1995.

[56] M. Stillerman, C. Marceau, and M. Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, July 1999.

[57] B. Stroustrup. *Le langage C++*. InterEditions, 1989.

[58] SUN. Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html, 1999.

[59] H.S. Vaccaro and G.E. Liepins. Detection of Anomalous Computer Session Activity. In *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, pages 280–289, 1989.

[60] A. Valdes and K. Skinner. Probabilistic Alert Correlation. In *Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, number 2212 in LNCS, pages 54–68, October 2001.

[61] M. Wood and M. Erlinger. Intrusion Detection Message Exchange Requirements. http://www.ietf.org/internet-drafts/draft-ietf-idwg-requirements-06.txt, 2002.

Annexe1

La table 4.8 donne les valeurs de la fonction *cdf* complémentaire ($1 - \Phi(\gamma) = \Phi(-\gamma)$) obtenues pour une loi normale réduite (de moyenne $\mu = 0$ et d'écart-type $\sigma = 1$). La loi normale vérifie que 99.7% des observations appartiennent à l'intervalle $[\mu - 3\sigma, \mu + 3\sigma] = [-3.0,3.0]$ pour une loi normale réduite.

Etant donnée que la fonction *cdf* est symétrique autour de μ, nous considérons sa répartition uniquement sur l'intervalle [0.0,3.0], présentée dans la figure 4.26.

La table suivante permet de déterminer la probabilité que la variable x s'écarte de la moyenne μ de plus de $\alpha * \sigma = \gamma$ dans l'intervalle [0.0,3.0] (avec une précision sur la deuxième décimale, donnée sur les colonnes). Au delà ce cet intervalle, $1 - \Phi(\gamma)$ est nulle.

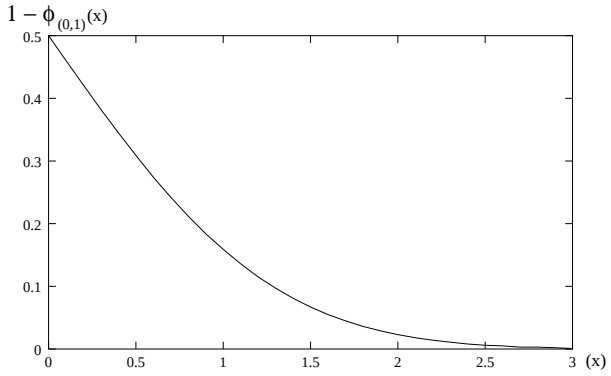

FIG. 4.26 – *Loi normale centrée réduite*

	0	1	2	3	4	5	6	7	8	9
0.0	0.500	0.496	0.492	0.488	0.484	0.480	0.476	0.472	0.468	0.464
0.1	0.460	0.456	0.452	0.448	0.444	0.440	0.436	0.433	0.429	0.425
0.2	0.421	0.417	0.413	0.409	0.405	0.401	0.397	0.394	0.390	0.386
0.3	0.382	0.378	0.374	0.371	0.367	0.363	0.359	0.356	0.352	0.348
0.4	0.345	0.341	0.337	0.334	0.330	0.326	0.323	0.319	0.316	0.312
0.5	0.309	0.305	0.302	0.298	0.295	0.291	0.288	0.284	0.281	0.278
0.6	0.274	0.271	0.268	0.264	0.261	0.258	0.255	0.251	0.248	0.245
0.7	0.242	0.239	0.236	0.233	0.230	0.227	0.224	0.221	0.218	0.215
0.8	0.212	0.209	0.206	0.203	0.200	0.198	0.195	0.192	0.189	0.187
0.9	0.184	0.181	0.179	0.176	0.174	0.171	0.169	0.166	0.164	0.161
1.0	0.159	0.156	0.154	0.152	0.149	0.147	0.145	0.142	0.140	0.138
1.1	0.136	0.133	0.131	0.129	0.127	0.125	0.123	0.121	0.119	0.117
1.2	0.115	0.113	0.111	0.109	0.107	0.106	0.104	0.102	0.100	0.099
1.3	0.097	0.095	0.093	0.092	0.090	0.089	0.087	0.085	0.084	0.082
1.4	0.081	0.079	0.078	0.076	0.075	0.074	0.072	0.071	0.069	0.068
1.5	0.067	0.066	0.064	0.063	0.062	0.061	0.059	0.058	0.057	0.056
1.6	0.055	0.054	0.053	0.052	0.051	0.049	0.048	0.047	0.046	0.046
1.7	0.045	0.044	0.043	0.042	0.041	0.040	0.039	0.038	0.038	0.037
1.8	0.036	0.035	0.034	0.034	0.033	0.032	0.031	0.031	0.030	0.029
1.9	0.029	0.028	0.027	0.027	0.026	0.026	0.025	0.024	0.024	0.023
2.0	0.023	0.022	0.022	0.021	0.021	0.020	0.020	0.019	0.019	0.018
2.1	0.018	0.017	0.017	0.017	0.016	0.016	0.015	0.015	0.015	0.014
2.2	0.014	0.014	0.013	0.013	0.013	0.012	0.012	0.012	0.011	0.011
2.3	0.011	0.010	0.010	0.010	0.010	0.009	0.009	0.009	0.009	0.008
2.4	0.008	0.008	0.008	0.008	0.007	0.007	0.007	0.007	0.007	0.006
2.5	0.006	0.006	0.006	0.006	0.006	0.005	0.005	0.005	0.005	0.005
2.6	0.005	0.005	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
2.7	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
2.8	0.003	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
2.9	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.001	0.001	0.001

TAB. 4.8 – *Fonction de distribution complémentaire cdf*