Uniformisation d'un système de traces : mise en place des outils Kernel pour les traces et les log

Frédéric Poupard

le cnam

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS ou CENTRE REGIONAL ASSOCIE DE BRETAGNE

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : Ingénieur en informatique

OPTION : Architecture et ingénierie des systèmes et des logiciels

par

Frédéric POUPARD

Uniformisation d'un système de traces : Mise en place des outils Kernel pour les traces et les log.

Soutenu le 19 juin 2013

JURY

PRESIDENT : **M. Yann POLLET Professeur titulaire de la chaire d'Intégration des Systèmes, Conservatoire National des Arts et Métiers**

MEMBRES : **M. Charles PREAUX Professeur des Universités Associées à l' UBS Directeur de la formation cyber défense de l' ENSIBS**

M. Luc PAUGAM Ingénieur software architecte Technicolor R&D France

M. Jean-Claude RIBIERE R&D Manager Deputy Technicolor R&D France

Remerciements

En préambule de ce mémoire, je souhaite adresser tous mes remerciements aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire.

En premier lieu, je tiens à remercier M. PREAUX pour ses conseils et son aide dans la finalisation de ce mémoire ainsi que le Pr. POLLET d'avoir accepté de présider le jury.

Je remercie aussi M. FEUILLATRE et M. RIBIERE de m'avoir proposé ce sujet et de m'avoir accueilli au sein de son service pour le réaliser.

Un grand merci à Luc PAUGAM, mon maître de stage, qui m'a guidé dans l'univers Linux, mais aussi transmis sa culture et son savoir sur les logiciels open source, et soutenu tout au long de mon travail.

Merci aussi au service des ressources humaines, en particulier Véronique BREMONT et Sonia HOURICHI, pour leur accompagnement et leur aide durant tout mon cursus au CNAM.

Je tiens également à remercier mes collègues de bureau, François-Xavier et Cédric, pour leur convivialité, ainsi que leur disponibilité pour répondre à mes nombreuses questions.

Merci également à mes différents relecteurs, mon épouse Katalina, mes sœurs Marie-Laure et Christelle ainsi que mon collègue Pascal MAETZ, pour leurs regards extérieurs et leurs conseils.

Enfin, je remercie mon entourage pour leur soutien et leurs encouragements.

Liste des abréviations

OS : Operating System (système d'exploitation)

DLNA : Digital Living Network Alliance

CPU : Central Processing Unit

FIFO : First In First Out

MMU : Memory Management Unit (Unité de gestion de mémoire)

API : Application Programming Interface (interface de programmation)

LTTng : Linux Trace Toolkit Next Generation

USB : Universal Serial Bus

GNU : Gnu is Not Unix

GPL : General Public Licence

IDE : Integrated Development Environment (Environnement de Développement Intégré)

SSH : Secure SHell

Glossaire

Middleware : Logiciel qui doit permettre à différentes applications d'échanger et d'inter-opérer.

Trace : En informatique, c'est le moyen de suivre et de contrôler toutes les activités relatives au programme qui est en train de s'exécuter.

Journalisation : La journalisation (logging) désigne l'enregistrement séquentiel dans un fichier ou une base de données de tous les événements affectant un processus particulier (application, activité d'un réseau informatique...).

Débogage : Signifie le fait d'enlever les erreurs d'un programme, en suivant le déroulement de celui-ci afin d'en repérer et corriger les dysfonctionnements.

Profilage : Le profilage est l'action qui consiste à enregistrer et traiter un ensemble de données permettant de caractériser le comportement d'une application.

Linux : C'est un système d'exploitation libre fonctionnant avec le noyau Linux. C'est une implémentation libre du système UNIX respectant les spécifications POSIX. Il existe de nombreuses versions de Linux adaptées à différents types de matériels. (ordinateur, téléphones portables, set-top boxes, boîtiers ADSL, lecteurs multimédia, etc).

LTTng : C'est un ensemble d'outils permettant l'enregistrement des événements du noyau Linux puis la visualisation (graphique ou non) des résultats. Le fonctionnement de LTTng nécessite de modifier le noyau Linux, ce qui peut poser des problèmes de compatibilité puisqu'il n'est pas intégré au noyau officiel.

Daemon : Désigne un type de programme informatique, un processus qui s'exécute en arrière-plan plutôt que sous le contrôle direct d'un utilisateur.

Shell : Le *Shell* est une interface texte qui permet à l'utilisateur de communiquer avec l'ordinateur. Le plus connu est celui de DOS (qui est encore émulé sous Windows). Mais bien d'autres existent sous d'autres systèmes tels que Bourn Shell, Bash...

Table des matières

Introduction

I Le contexte

En 1999, après l'obtention de mon BTS en électronique, j'ai intégré le service Hardware plateforme de Technicolor. Au fur et à mesure des années et de ma formation au CNAM, mon travail a consisté à concevoir, développer et maintenir des logiciels de test bas niveau pour la caractérisation des plateformes.

Puis pour la réalisation de mon mémoire, j'ai entrepris, avec l'accord de ma hiérarchie et des ressources humaines, d'intégrer un service plus en adéquation avec le diplôme visé.

Après avoir rencontré différents managers dans le cadre de ma recherche d'un sujet de mémoire, j'ai opté pour le sujet proposé par le service «Set Top Box Middleware».

J'ai donc intégré début septembre un service qui travaille depuis deux ans sur le middleware « Revolution-S ».

II La problématique

Durant ces deux années de développement, l'analyse du temps passé à résoudre les bug était assez important. Le constat étant que les techniques pour tracer le comportement du middleware et le débuguer étaient très disparates dans le service. Les deux principales raisons étant :

- l'hétérogénéité des équipes. La plupart des développeurs ont migré du système propriétaire précédent et n'ont pas la culture Linux nécessaire pour mettre en œuvre facilement les outils de traces adaptés à cet OS.
- L'absence de directives fortes sur les outils et techniques à utiliser.

La mise à disposition d'un système de traces sécurisé dans le middleware serait une bonne stratégie pour améliorer l'efficacité du débogage du logiciel.

De plus, nos clients opérateurs pourraient être intéressés de récupérer et analyser les informations comportementales de la plateforme sous forme de traces, leur permettant ainsi

de diagnostiquer l'origine d'un défaut chez leurs abonnés et ainsi éviter des retours au service après-vente inutiles.

Une autre utilité pourrait être l'enregistrement de certaines traces, sous forme de log, leur permettant d'appréhender les modes de consommation de leurs abonnés, afin de leur proposer des services personnalisés.

Au final, la mise à disposition d'un système de traces est un besoin réel tant pour améliorer la méthode et les outils de travail sur Revolution-S, que pour améliorer les services rendus proprement dits du middleware

III Présentation du Plan

Cette année Technicolor va fournir à Telecom Italia sa toute dernière génération de plateforme alliant télévision payante et serveur personnel [1]. Celle-ci intégrera le middleware « Revolution-S » récompensant ainsi deux ans d'efforts et d'investissements dans ce projet.

En effet, en 2010, Technicolor a décidé de développer son propre middleware pour les décodeurs, tablettes et passerelles internet (gateway). Ce logiciel, qui doit permettre à différentes applications d'échanger et d'inter-opérer, avait pour objectif d'avoir une architecture unifiée et modulaire afin que chaque composant qui le constitue puisse être réutilisé et partagé entre les différentes plateformes. « Revolution-S » s'est alors appuyé sur la solution open source linux.

L'une des difficultés rencontrée durant le développement a été la mise au point du logiciel. Il existe de nombreux outils hétérogènes sur ce domaine dans le monde Linux. Aucune uniformisation n'ayant été mise en place sur Revolution-S, chaque développeur a utilisé sa propre solution. Outre le fait que certaines mauvaises pratiques ont été utilisées, le manque l'homogénéité et l'absence d'une solution adaptée a augmenté le cout du débogage du logiciel au fur et à mesure que sa complexité augmentait avec les fonctions introduites.

[1] http://www.technicolor.com/en/hi/about-technicolor/press-center/2012/technicolor-is-telecom-italia-s-partner-to-bridge-pay-tv-over-the-top-services-and-personal-media-for-its-next-generation-cubovision-service le 10 octobre 2012

C’est dans le cadre de ce mémoire que l’on m’a demandé de mettre en place une méthodologie de traces et d’enregistrement de logs, capable de devenir la règle parce qu’offrant une solution uniforme et commune à tous les développeurs.

Après avoir défini la problématique du sujet, les objectifs attendus et la démarche pour les atteindre, je présenterai la société Technicolor, j’exposerai le travail réalisé dans notre segment de la distribution numérique et celui réalisé par le service « Set Top Box » que j’ai intégré dans le cadre de ce mémoire.

Je décrirai alors les différentes méthodes d’investigation pour corriger et améliorer un logiciel middleware embarqué dans un décodeur, ainsi que les outils disponibles dans un contexte d’exécution Linux sur lequel repose notre middleware.

Ensuite, je présenterai l’étude de l’état de l’art des pratiques au sein du projet «Revolution-S», l’analyse des nouveaux besoins fonctionnels internes et externes à Technicolor qui permettront de définir l’architecture cible du nouveau système de traces et de log.

Enfin, je présenterai le développement de ma solution ainsi que les différents scénarios de test et de validation mis en place.

IV Les objectifs

Le service souhaite donc mettre en place un système de traces & log commun à l’ensemble des développeurs du middleware. Il doit aussi pouvoir répondre à certains besoins clients. On m’a donc défini un certain nombre d’objectifs :

- Etudier les différents systèmes de traces existants dans le domaine open source Linux
- Analyser le besoin des développeurs
- Analyser le besoin des clients
- Faire évoluer le système existant pour répondre aux nouveaux besoins.

Pour réaliser ce projet, je serai l’unique développeur. Il faudra donc prioriser les besoins en accord avec mon maître de stage.

Une fois la solution développée, il sera primordiale de former les différents développeurs à l'utilisation de ce nouveau système de traces et log.

Les aspects de sécurité devront aussi être abordés.

Si à la fin du stage, tous les besoins retenus ne sont pas développés, la conclusion du mémoire devra permettre à mon manager de décider des suites à donner à ce projet.

V La réalisation du sujet

Afin de répondre à la problématique posée et aux objectifs fixés, je vais définir les principales étapes qui vont me guider à la réalisation de ce sujet. Ce découpage va ainsi me permettre d'estimer l'effort ainsi que l'enchaînement des étapes afin de réaliser le planning prévisionnel. Enfin je listerai les ressources nécessaires à mettre en œuvre pour atteindre les objectifs.

V.1 Estimation de l'effort pour les principales étapes

Voici les principales étapes que j'ai identifiées et qui vont me permettre d'atteindre les différents objectifs fixés précédemment.

Pour chacune des étapes, j'ai estimé l'effort en semaines, afin de déterminer le coût de développement de mon sujet.

Définition du contexte :

- Définition des objectifs et dans quel contexte.
- Effort estimé à 1 homme/semaine.

Etude des outils de traces & log Linux :

- Définition des mots clés
- Compréhension du système d'exploitation Linux.
- Recherche des différents outils de trace et log existant dans le domaine open source.
- Effort estimé à 8 hommes/semaine.

Etude de l'existant :

- Analyse du système de traces déjà existant mais très peu utilisé.
- Effort estimé à 2 hommes/semaine.

Etude des besoins :

- Analyse du besoin du client.
- Analyse du besoin des développeurs.
- Effort estimé à 2 hommes/semaine.

Architecture de la solution :

- Architecture représentant la solution à mettre en œuvre pour répondre aux différents besoins.
- Effort estimé à 3 hommes/semaine.

Développement :

- Effort estimé à 9 hommes/semaine.

Test & Validation :

- Effort estimé à 3 hommes/semaine.

Formation des développeurs :

- Formation de l'ensemble des développeurs du service afin qu'ils puissent utiliser cette nouvelle solution.
- Effort estimé à 1 homme/semaine.

Ce qui représente donc un effort global de 29 hommes/semaine soit 7 hommes/mois.

V.2 Les ressources nécessaires

Pour mener à bien ce projet, j'ai listé les ressources qui me seront nécessaires

- Matériels

Décodeur numérique : Mise à disposition d'une plateforme de développement de type décodeur numérique pour y intégrer la solution et la tester.

Ordinateur de développement

- Logiciels

Accès à l'environnement de développement du middleware « Revolution-S »

- Financières

Le matériel nécessaire étant fourni par le service, je n'ai besoin d'aucune ressource financière.

- Humaines

Aucune ressources humaines, à part moi, ne sera affectée à ce développement.

- Formation

Demande de formation sur le développement embarqué sous Linux. Cette demande a été acceptée par mon manager et aura lieu fin janvier.

V.3 Les risques

Afin de permettre au projet de s'exécuter selon les prévisions, l'identification des risques est une étape importante. Selon le guide PMBOK(1), elle se divise en 4 parties.

- Identification des risques.
- Quantification des risques : poids, probabilité d'apparition, criticité.
- Actions correctrices à mettre en place.
- Suivi des actions correctrices.

Le tableau ci-dessous liste les différents risques identifiés et indique leur impact sur le projet. Je leur affecte un poids et une probabilité d'apparition entre 1 et 4. J'évalue alors leur criticité en multipliant le poids avec la probabilité. En fonction de celle-ci, j'indique les actions à mettre en place pour minimiser le risque.

Tableau I : Tableau d'analyse des risques du projet.

Identification		Quantification			Actions Correctrices
Risque	Impact	Poids	Probabilité	Criticité	Couverture Prévue
Mauvaise compréhension du besoin	Développement de fonctionnalité inutile	2	1	2	Validation de chaque besoin avec le maître de stage
Sous-estimation des fonctionnalités à développer	Planning de développement	3	1	3	Hiérarchisation des fonctionnalités allant de indispensables à optionnelles et ceci en accord avec le maître de stage.
Sous-estimation de l'effort pour la partie développement	Développement non terminé à la date prévue	3	2	6	Suite à la hiérarchisation, réalisation d'un micro planning de chaque fonctionnalité à développer. (Date au plus tôt et au plus tard). Puis un suivi hebdomadaire sur l'avancement du développement devra être fait.

V.4 Planning prévisionnel

Suite à l'estimation de l'effort et des risques identifiés, je peux donc élaborer le planning prévisionnel suivant.

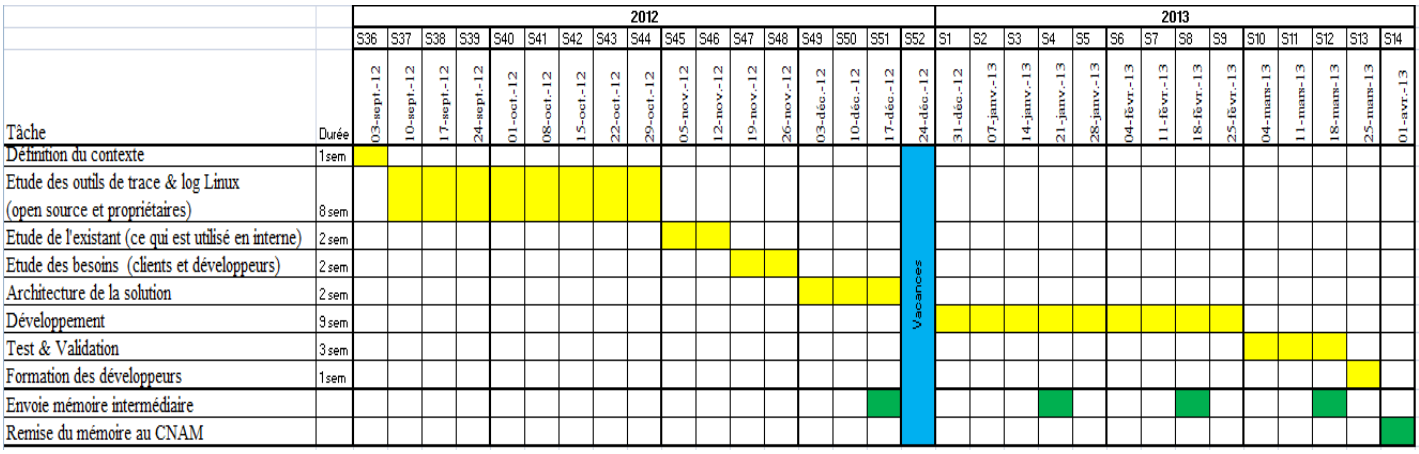


Figure 1 : Planning prévisionnel des principales tâches et de leurs efforts.

Suite à ce planning et avant de vous exposer mon étude sur les différentes méthodes d'investigation pour corriger et améliorer un logiciel middleware, je vais vous présenter un peu plus en détail la société Technicolor, ses activités, le centre de R&D de Rennes et le service « Set Top Box » que j'ai intégré dans le cadre de ce mémoire

PARTIE 1 : TECHNICOLOR

I Son activité

Technicolor [2], leader technologique mondial dans le secteur du Media & Entertainment, est à la pointe de l'innovation numérique. Grâce à ses laboratoires de recherche et d'innovation, il occupe des positions clés sur le marché, au travers de la fourniture de services vidéo avancés pour les créateurs et les distributeurs de contenu. Il bénéficie également d'un riche portefeuille de propriétés intellectuelles, centré sur les technologies de l'image et du son, et reposant sur une activité de Licences performante.

Sa mission est d'enrichir l'expérience de consommation de contenu sur tous les écrans, que ce soit dans les salles de cinéma, à domicile ou en déplacement. Et ceci en fournissant des solutions avancées pour les créateurs et les distributeurs de contenu.

Sa stratégie est de devenir un leader de l'innovation en matière de solutions de monétisation du contenu.

Pour ce faire, Technicolor peut compter sur plus de 16 000 personnes, réparties dans 28 pays.

Figure 2 : Les sites Technicolor dans le monde

[2] http://www.technicolor.com/uploads/about/at-a-glance-fr-def-light-v4.pdf le 10 octobre 2012

II Ses segments

Ses activités se décomposent en 3 segments :

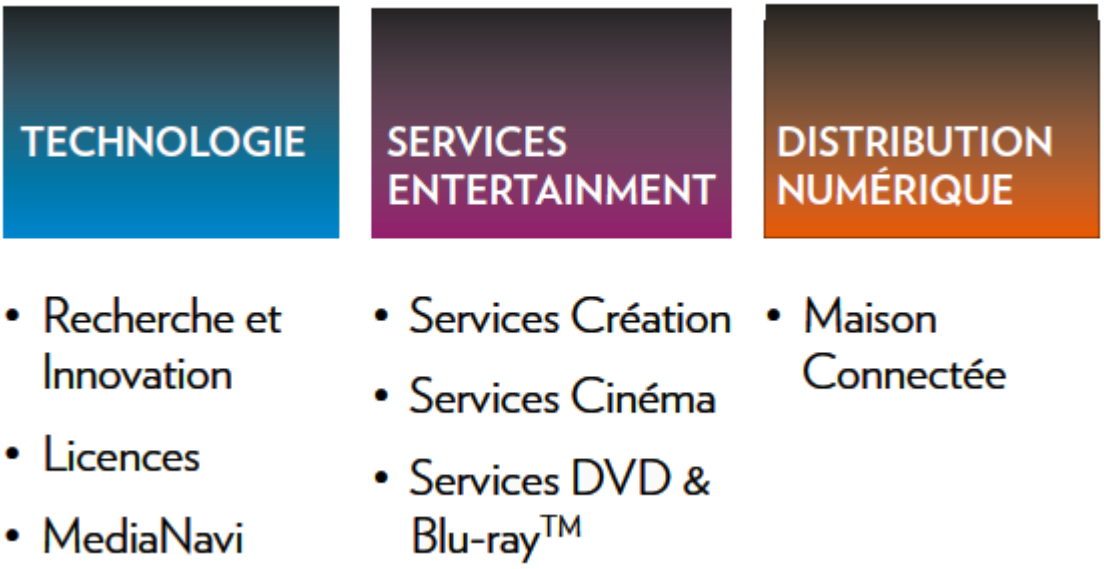


Figure 3 : Les 3 segments de Technicolor

II.1 Technologie

Le segment Technologie comprend les activités Recherche & Innovation, Licences et MediaNavi.

La division Recherche & Innovation invente, développe et transfère des technologies.

La division Licences a pour vocation de protéger et monétiser la propriété intellectuelle du Groupe et génère une grande partie du chiffre d'affaires du segment.

La division MediaNavi comprend les plateformes et applications du Groupe visant à simplifier et enrichir l'expérience de l'utilisateur final en matière de découverte, de consommation et de partage de contenu. Elle comprend en particulier M-GO.

II.2 Services Entertainment

Le segment Services Entertainment développe et commercialise des technologies et des services audiovisuels auprès de l'industrie du Media & Entertainment.

Ce segment offre des services liés à la production, la préparation et la création de contenus vidéos, à leur distribution sous forme de supports physiques et de médias numériques. Il offre également des services de préparation et de gestion du contenu.

II.3 Distribution Numérique

Le segment Distribution Numérique développe et commercialise des solutions matérielles et logicielles pour les industries des Télécommunications et du Media & Entertainment. Son expertise porte notamment sur les plateformes d'accès et de fourniture de services, permettant ainsi à ses clients d'offrir une nouvelle expérience de divertissement aux utilisateurs.

C'est dans ce dernier segment que mon stage se déroule.

III Le site de Rennes

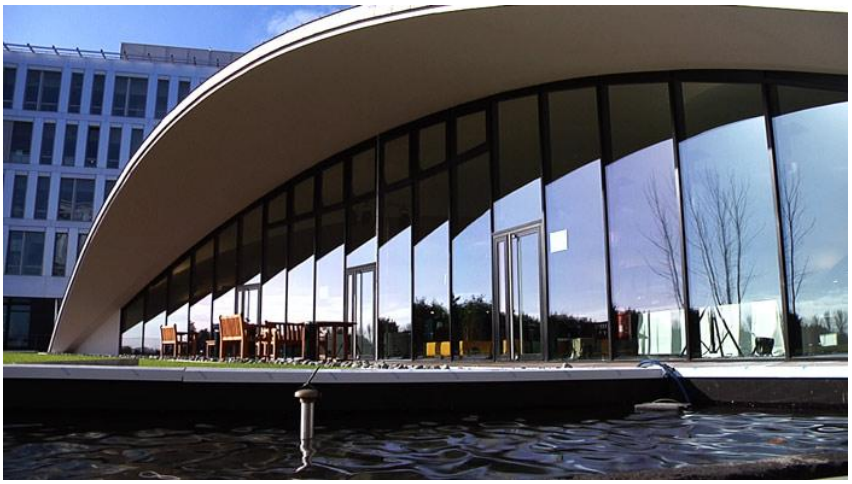


Figure 4 : Le nouveau site Technicolor à Rennes

Technicolor Rennes est le plus grand centre technologique du Groupe Technicolor. Pôle d'excellence sur l'ensemble de la chaîne de l'image, le centre de Rennes conçoit et développe des solutions innovantes pour les industries « Média, Communication & Divertissement». Technicolor Rennes étudie les nouvelles technologies de compression, de protection, de transmission, de production, de gestion des contenus en haute définition et prépare les futures générations d'équipements numériques et de services associés – notamment décodeurs numériques, modems d'accès large bande, modules pour la télévision interactive – pour des clients professionnels et grand public des industries «Média, Communication & Divertissement».

A fin 2011, 529 personnes y travaillent dont 87% sont des ingénieurs et cadres.

Les axes de Recherche et Développement qui soutiennent la vision stratégique du Groupe sont :

<u>dans le segment Technologie</u>

• Compression/décompression avec traitement audio/vidéo

• Acquisition et traitement du signal et de l'image

• Technologies sans fil et large bande

• Sécurité des contenus tout au long de la chaîne de l'image

• 3D

• Brevets

dans le segment Distribution Numérique

• Plates-formes de décodeurs numériques satellite, câble et IP intégrant les dernières technologies avec disque dur, codage MPEG4, protocole Internet, haute définition…

IV Connected Home (Maison Connectée)

IV.1 L'organigramme

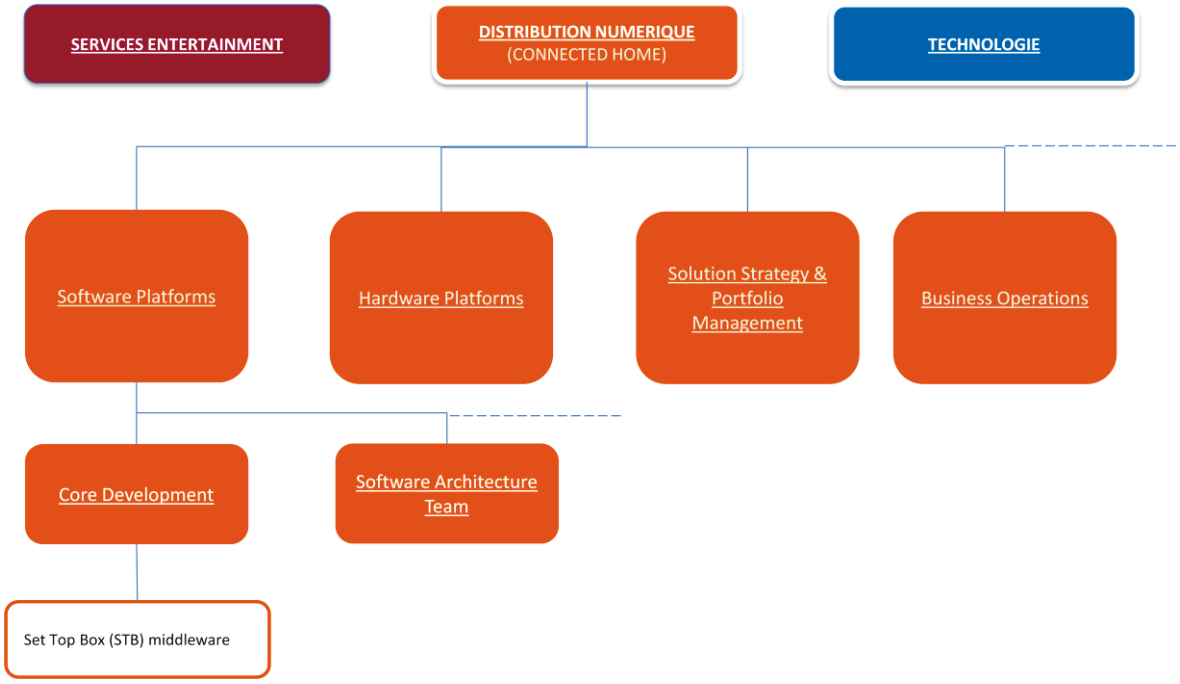


Figure 5 : Organigramme situant le service « Set Top Box » dans Technicolor

IV.2 Ses développements

La division Connected Home (Maison Connectée), qui fait partie du segment Distribution Numérique emploie 2400 personnes dans le monde dont 240 à Rennes. On y conçoit et développe différents types de plateformes qui se déclinent sur tout une gamme pour offrir des fonctions et des services divers et évolués .

Les décodeurs numériques de salon qui permettent de regarder la télévision par différents modes de réception (satellite, câble, terrestre, IP), qui peuvent aussi enregistrer sur un disque dur ou rediffuser le contenu dans la maison par le biais d'un serveur DLNA.

Les passerelles d'accès numériques qui sont les points d'entrées de la maison au réseau internet. Offrant des services avancés comme le triple play (internet, téléphonie et télévision) ou le partage de contenu.

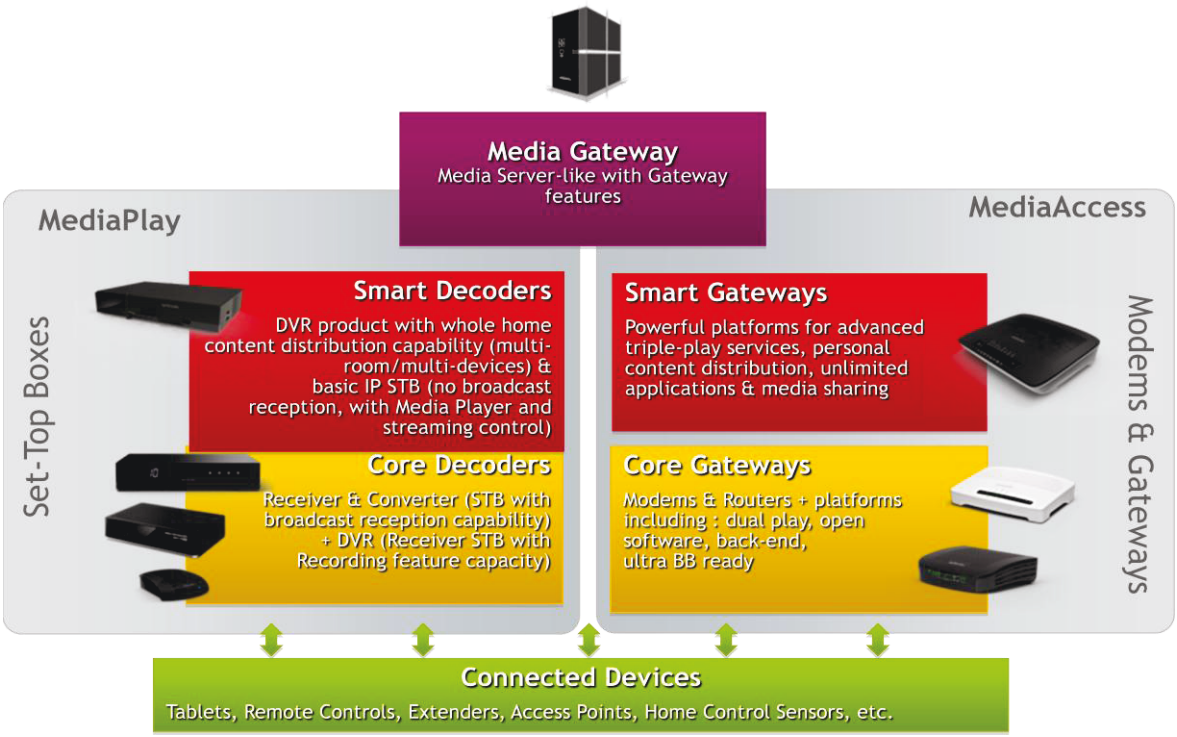


Figure 6 : L'offre de Connected Home

Ces développements ciblent uniquement le marché des opérateurs de télévisons payantes et de contenu.

Figure 7 : Opérateurs de la zone EMEA[3] utilisant des plateformes numériques en 2011

[3] On parle de « zone EMEA » pour désigner l'Europe, le Moyen-Orient (Middle East) et l'Afrique

IV.3 A Rennes

Le site de Rennes travaille essentiellement sur les décodeurs numériques, cela va de la conception matérielle (carte électronique et mécanique) au développement du logiciel middleware.

Les principaux objectifs de notre middleware sont d'offrir à nos clients la possibilité d'exécuter plusieurs applications dans de multiples environnements et ce sur le même décodeur, le tout avec un maximum de sécurité.

Figure 8 : Exemples d'applications supportées avec un maximum de sécurité

Mais aussi de leur permettre de déployer leurs services sur chaque écran connecté, géré ou non, afin de renforcer leur présence à la maison.

Figure 9 : Diffusion du contenu sur tous les types de supports

IV.4 Le service « Set Top Box »

Ce service est en charge du développement du middleware du nom de « Révolution S » pour les décodeurs numériques. Depuis deux ans, une centaine de personnes travaille à la réalisation de ce middleware.

Différents métiers y sont représentés, tel que des architectes systèmes, des développeurs, des intégrateurs, des testeurs, …

C'est dans ce service que j'ai réalisé ce mémoire.

IV.5 Le middleware « Revolution-S »

« Revolution-S » a pour objectif d'offrir à nos clients opérateurs un environnement de développement (SDK) pour leurs applications. Le tout en y intégrant des fonctionnalités spécifiques aux décodeurs numériques (Décodage vidéo, enregistrement numérique, DLNA, …).

Cet environnement de développement est découpé en trois gammes en adéquation avec la déclinaison faite sur les décodeurs numériques (voir figure 5)

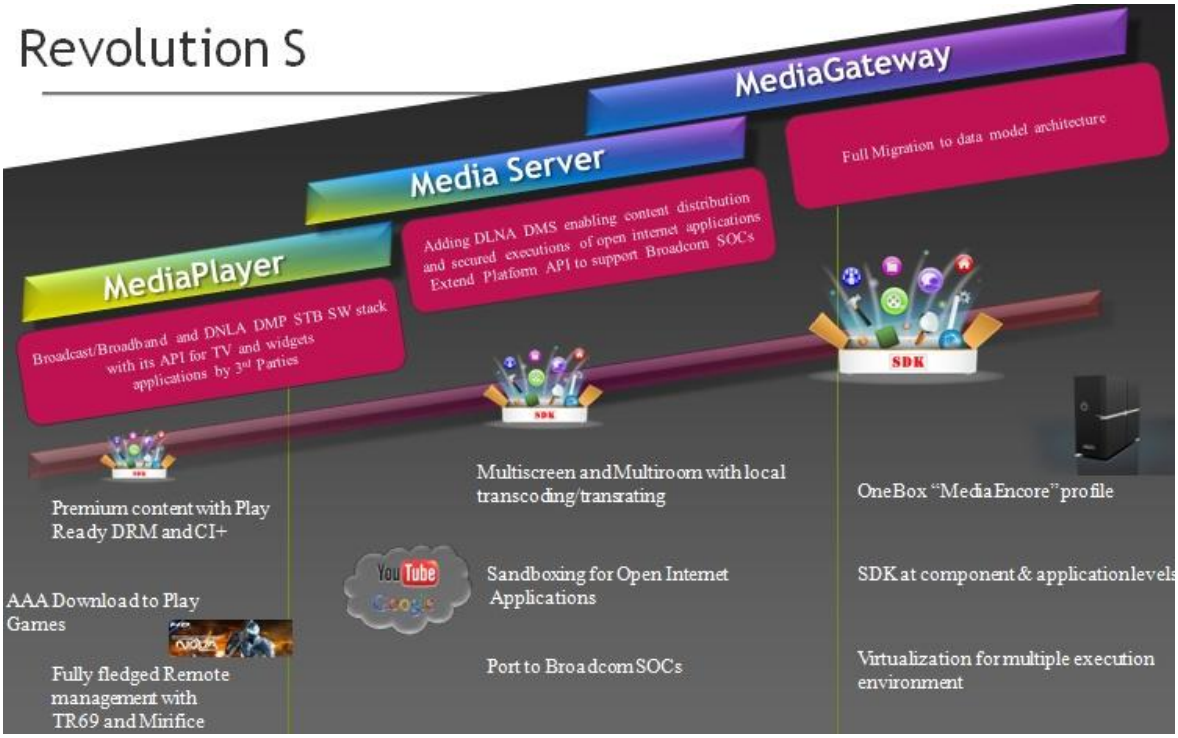


Figure 10 : Les déclinaisons de Revolution-S

Le premier kit « Media Player » offre la possibilité de décoder de la vidéo depuis n'importe quelle interface d'entrée (tuner, IP, USB, …) ainsi que la possibilité d'enregistrement sur disque dur.

Le kit « Media Server » ajoute les possibilités de serveur vidéo dans la maison à travers du DLNA et des fonctions de transcodage.

Enfin le kit « Media Gateway » ajoute les possibilités des passerelles numériques (triple play) et de virtualisation.

Le middleware peut donc être porté sur différents types de plateforme, ce qui implique de multiples architectures matérielles et donc des processeurs pouvant provenir de différents constructeurs. Il doit donc pouvoir s'affranchir de la plateforme.

Pour y parvenir, la solution est basée sur le système d'exploitation LINUX.

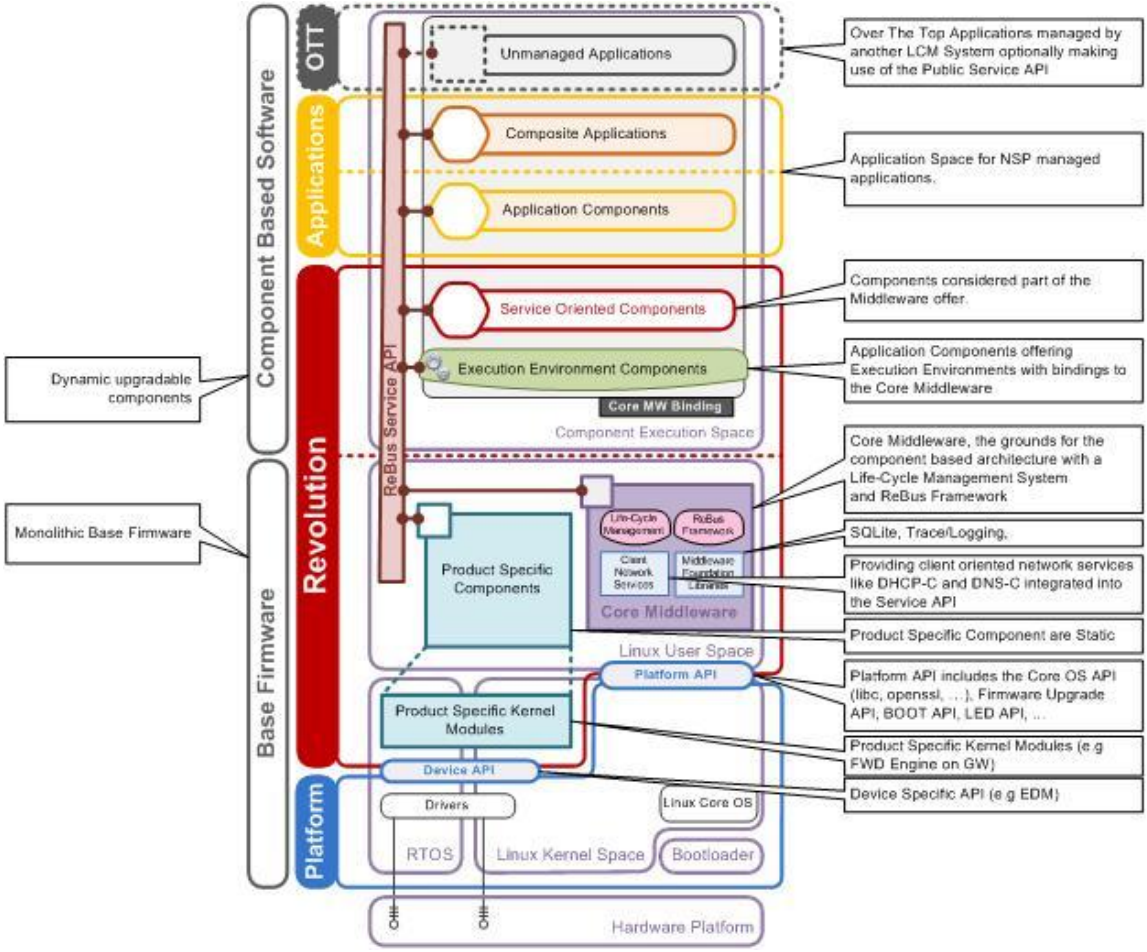


Figure 11 : Représentation de haut niveau du middleware « Revolution-S »

Suite à cette présentation générale sur « Revolution-S » et avant de vous détailler l'analyse des nouveaux besoins fonctionnels internes et externes pour l'uniformisation du

système de trace et log, je vais vous exposer mon étude sur les différentes méthodes d'investigation pour corriger et améliorer un logiciel middleware embarqué dans un décodeur, ainsi que les outils disponibles dans un contexte d'exécution Linux sur lequel repose notre middleware.

PARTIE 2 : L'ETUDE

Pour pouvoir répondre à la problématique posée, il m'a fallu, comprendre le contexte du middleware « Revolution-S », qui repose sur le système d'exploitation Linux, et comprendre les spécificités de celui-ci dûes au décodeur numérique. Je vais ainsi commencer par présenter d'une façon synthétique ce contexte pour ensuite définir les méthodes associées aux traces. Enfin je présenterai les différents outils existants pouvant répondre à la problématique posée et quelle solution je pense retenir par la suite.

I Le système d'exploitation LINUX

Comme je l'ai écrit précédemment, le middleware « Revolution-S » repose sur le système d'exploitation Linux. Il est donc important de comprendre les grandes notions de cet OS ainsi que les spécificités liées aux systèmes embarqués.

I.1 Historique

I.1.1 Le Projet GNU

En 1984(2), un jeune chercheur du laboratoire d'Intelligence Artificielle du MIT décide de lancer un projet de substitut libre pour le système Unix, où tous les outils seraient disponibles gratuitement et sous forme de code source. Richard Stallman fonde à cette occasion le projet GNU (*Gnu is Not Unix*) et surtout lui donne un cadre formel et juridique important : la GPL (*General Public Licence*) qui permet de garantir la liberté de modification et de redistribution, ainsi que la pérennité de l'accès aux sources d'un logiciel.

En 1991, le système GNU s'est enrichi d'un grand nombre d'utilitaires simples (ls, cp, mv, etc.) et complexes (éditeur Emacs, shell Bash, compilateur gcc, etc.) mais il lui manque encore le noyau, le cœur du système d'exploitation, chargé de réguler les accès aux ressources des différentes applications. Pour réaliser ce travail, un projet nommé HURD démarre ; toutefois, plutôt que de « copier » le comportement d'un système existant (comme Linux avec Unix), un véritable travail de conception indépendant est entrepris. Malheureusement, en raison de l'ampleur de la tâche, le développement du noyau est long, et c'est dans cet intervalle que vient se glisser le noyau Linux.

I.1.2 Linux

A cette époque, un étudiant finlandais, Linus Torvalds, commence à développer un noyau et demande aux personnes intéressées d'y contribuer. La licence GPL a été

publiée à la même époque et Linus Torvalds s'est laissé persuader de placer son noyau sous cette dernière.

- **UNIX Family Tree:**

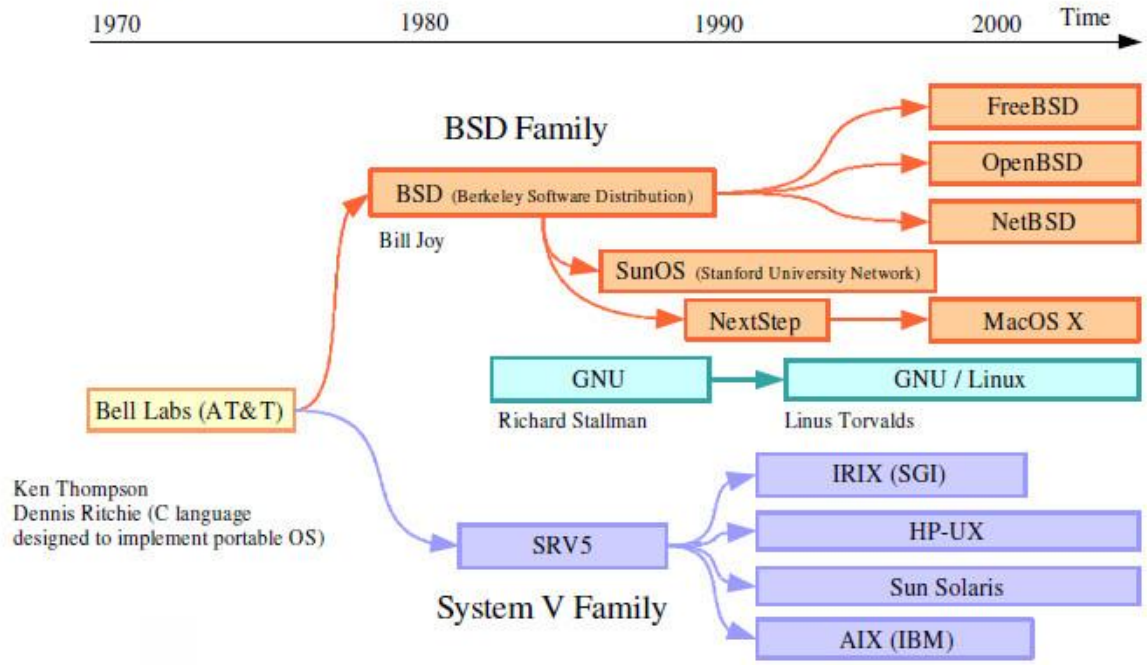


Figure 12 : Ligne temporelle des différents OS à base d'UNIX

Le développement du noyau Linux a progressé assez lentement au départ, puisque ce n'est qu'en mars 1994 que Linus Torvalds le juge suffisamment complet et stable pour être doté d'un numéro de version 1.0. Ce noyau est assez limité, mais contient déjà les éléments essentiels d'une station de travail Unix (réseau, gestion de la mémoire, IPC, divers systèmes de fichiers, pilotes de périphériques divers…)

Au bout de quelques semaines, Linus Torvalds initie une nouvelle branche de développement avec un noyau 1.1.0. À cette occasion, il fut décidé de laisser simultanément en circulation deux versions du noyau : une version stable sur laquelle ne seront apportées que les corrections de bogues éventuels, et une version de développement sur laquelle les nouveaux algorithmes, pilotes, fonctionnalités, etc. seront testés.

Pour les différencier, une numérotation explicite des versions a été mise en place. Le numéro d'un noyau sera ainsi composé de trois chiffres successifs :

- Un numéro majeur de version
- Un numéro mineur de version
- Un numéro de mise à jour.

La règle établie précise que les noyaux stables, ceux que l'on peut se permettre d'utiliser pour des applications de production, ont un numéro mineur pair : 1.0.x, 1.2.x, 2.0.x, 2.2.x et 2.4.x. Les noyaux ayant un numéro mineur impair sont uniquement utilisés à des fins de test et de développement. Certaines mises à jour ne sont même pas compilables pour toutes les options disponibles. Pour un système en production, on n'utilisera donc qu'un noyau stable.

Au moment où je rédige ces lignes, le dernier noyau stable est le 3.6.x figé le 30 septembre 2012.

Le système d'exploitation actuellement connu est donc un assemblage des outils GNU fonctionnant sur un noyau Linux. On a l'habitude de dire que Linux est un système d'exploitation, or il représente uniquement le cœur du système.

Linux est donc un noyau, GNU est un ensemble de programmes utilitaires et GNU/Linux est le système d'exploitation.

I.1.3 Les distributions

À l'origine(3), pour installer un système opérationnel fondé sur le noyau Linux, il fallait être un expert capable de trouver les logiciels nécessaires, et de les installer un à un de manière à former un système cohérent.

Rapidement, des ensembles de logiciels formant un système complet prêt à l'usage ont été disponibles : ce sont les premières distributions Linux.

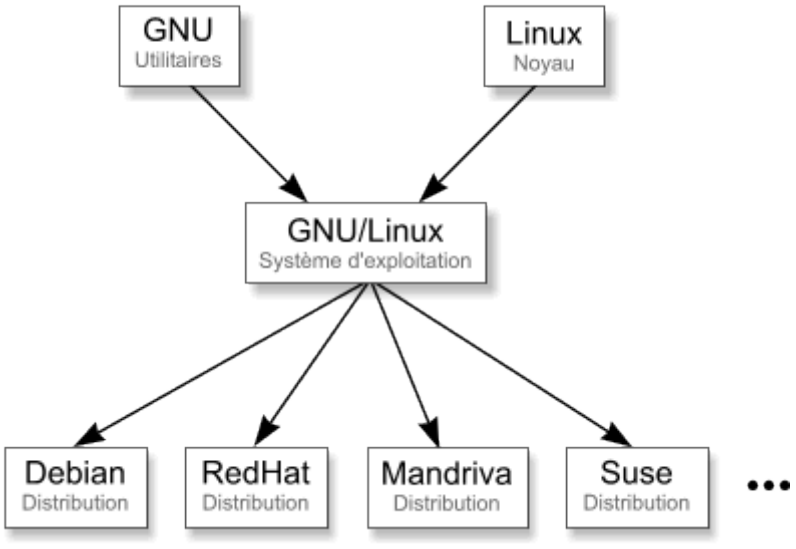


Figure 13 : Le concept des distributions sous Linux

Les distributions GNU/Linux sont très nombreuses, d'autant plus qu'il est facile, à partir d'une distribution, d'en créer une nouvelle. Certaines distributions proposent ainsi des variantes pour des usages différents : commerciales (spécial entreprise, ...), grand public, pour l'éducation, adaptées à un travail spécifique (pour la musique par exemple), d'environnements différents (de la simple différence finale graphique à l'interface utilisateur différente), adaptées pour des raisons éthiques et ainsi de suite.

Le middleware « Revolution-S » peut ainsi être comparé à une distribution pour Linux. Il s'appuie sur le même système d'exploitation mais il est par contre spécifique aux décodeurs numériques que Technicolor développe.

I.2 Présentation générale de l'architecture LINUX

Lorsque l'on regarde Linux dans son ensemble (4), on constate qu'il se décompose en deux couches, appelées « espace noyau» et « espace utilisateur ».

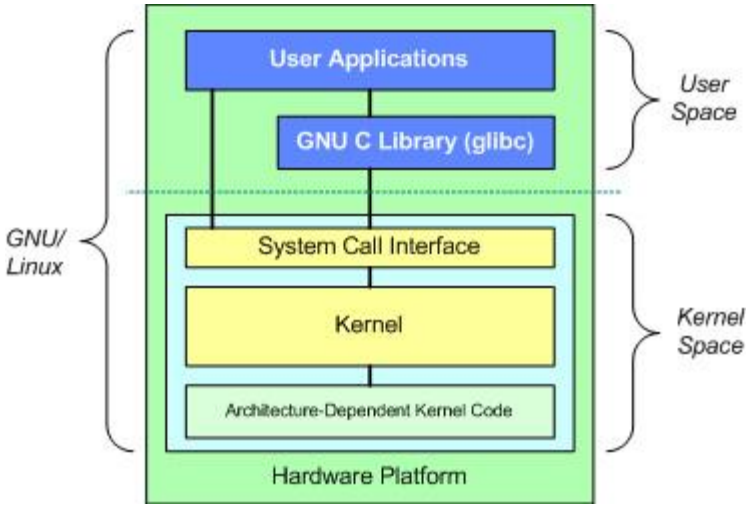


Figure 14 : Vue haut niveau de GNU/Linux

Au sommet se trouve l'espace utilisateur, c'est là que les applications sont exécutées. Lorsqu'une application a besoin d'accéder à un périphérique, elle doit le faire en passant par un appel système, car elle ne peut y accéder directement. C'est l'espace noyau situé en dessous qui, suite à cet appel, va vérifier qu'il s'agit d'une demande valable, exécuter la requête et retourner le résultat à l'espace utilisateur. Le middleware « Revolution-S » s'exécute donc dans l'espace utilisateur.

La plupart des applications s'appuient sur la librairie C, fourni avec les outils GNU, pour accéder aux ressources matérielles.

L'espace noyau peut être divisé en trois niveaux. En haut se trouve l'interface d'appel système. Au milieu se situe le code du noyau qui est générique et commun à toutes les architectures de processeurs supportés par Linux. En dessous, se trouve le code noyau spécifique qui est dépendant de l'architecture matérielle de la plateforme.

Cette décomposition en deux couches permet de cloisonner physiquement l'espace utilisateur et l'espace noyau dans deux zones mémoires distinctes. Ainsi une application en espace utilisateur ne pourra jamais adresser la zone mémoire où s'exécute le noyau et rendre ainsi le système instable. Cette architecture permet aussi de cloisonner les applications entre elles. Ainsi si une application est défectueuse, elle ne pourra pas perturber les autres en venant par exemple écrire dans leur zone mémoire.

Il me semble important de présenter un peu plus en détail ces deux espaces car cela permettra de mieux appréhender les méthodes et outils que je présenterai après.

I.3 L'espace Noyau (Kernel)

Le noyau est le cœur du système d'exploitation Linux et son architecture est vaste et complexe. Il implémente un nombre important de fonctions. Mais comme le montre la figure ci-dessous, il est bien organisé en terme de sous systèmes et de couches.

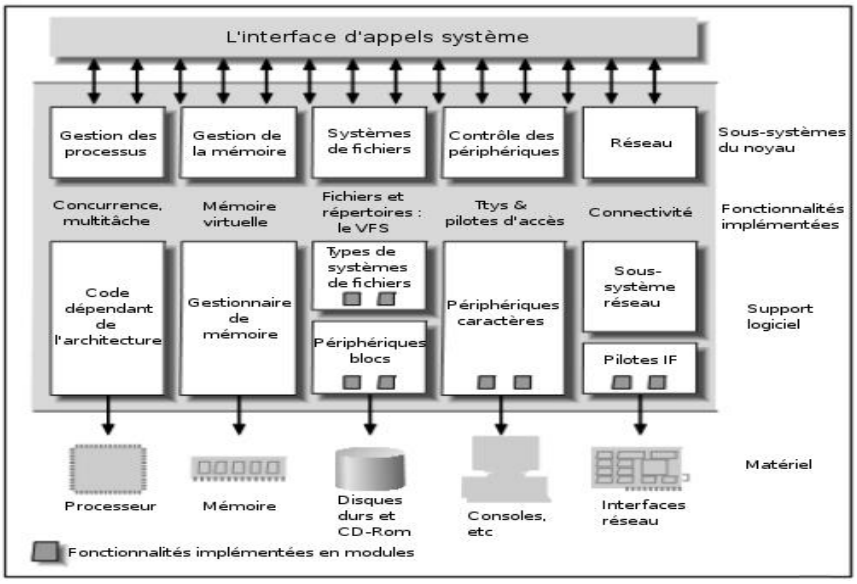


Figure 15 : Décomposition du noyau Linux en couches et sous-systèmes.

Chaque sous système se découpe en couches logicielles successives, les couches supérieures s'appuyant sur les services des couches inférieures. Ainsi, des points de synchronisation peuvent être définis et à partir de ceux-ci le système peut reprendre un

fonctionnement normal après une défaillance. Ces points de synchronisation permettent donc d'assurer la viabilité du système, ou du moins des couches inférieures, même en cas d'erreurs inopinées dans une couche de plus haut niveau.

Ces découpes en couches et en sous systèmes permettent aussi de rendre le noyau modulable. Ainsi les parties fondamentales du système sont regroupées dans un bloc de code unique (monolithique). Les autres fonctions, comme les pilotes matériels, sont regroupées en différents modules qui peuvent être séparés tant du point de vue du code que du point de vue binaire. On parle alors de Noyau monolithique modulaire.

Les parties fondamentales constituent alors une base relativement petite et très bien testée, à laquelle beaucoup de pilotes de périphériques sont ajoutés. La complexité du noyau reste donc limitée, car les pilotes de périphériques ne sont bien entendu pas tous utilisés en même temps sur une même machine. Seuls les pilotes correspondant au matériel effectivement présent dans la machine sont chargés dans le noyau au démarrage.

Je vais donc vous présenter succinctement les cinq sous systèmes du noyau (5) comme le montre la figure ci-dessous.

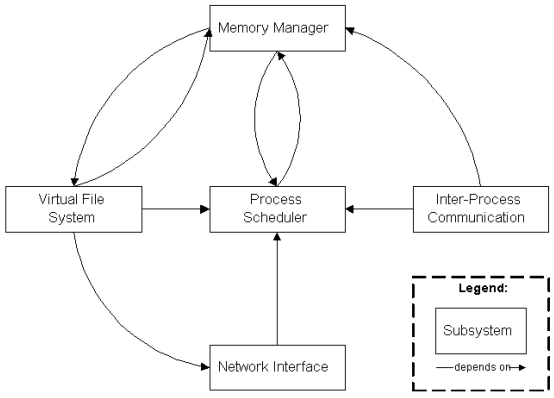


Figure 16 : Vue d'ensemble des sous-systèmes du noyau Linux.

I.3.1 L'ordonnanceur

L'ordonnanceur est le sous système le plus important dans le noyau. Son but est d' allouer les ressources CPU aux différents processus provenant de l'espace utilisateur ou de l'espace noyau. Il est conçu de telle sorte que les processus aient un accès équitable au CPU. Il est donc l'élément central du Noyau et tous les autres sous-systèmes communiquent avec lui.

I.3.2 Le gestionnaire de mémoire

Le sous-système gestionnaire de mémoire est chargé de contrôler l'accès des processus aux ressources mémoires du matériel. Ceci est accompli par l'unité de gestion de la mémoire (*MMU* pour *Memory Management Unit*) qui fournit la correspondance entre la mémoire virtuelle adressée par un processus et la mémoire physique. Il conserve ainsi une cartographie de chaque processus de sorte que deux processus qui ont la même adresse virtuelle sont en fait localisés dans des emplacements physiques différents.

I.3.3 Le système de fichier virtuel

Le système de fichier virtuel est conçu pour présenter une vue cohérente des données stockées sur les périphériques matériels. La plupart des périphériques utilisent une interface de drivers génériques, mais le système de fichier virtuel va plus loin en permettant à l'administrateur de représenter n'importe quel périphérique comme un système de fichier logique. Il permet aussi d'abstraire à la fois les détails du fichier logique et du périphérique à l'espace utilisateur. Ainsi les processus peuvent accéder à un périphérique comme s'il accédait à un fichier, ce qui permet d'avoir une interface commune.

I.3.4 L'interface réseau

L'interface réseau du noyau permet au système de se connecter à d'autres systèmes sur un réseau. Un grand nombre de périphériques sont supportés ainsi qu'un certain nombre de protocoles réseaux qui peuvent être utilisés. L'interface réseau permet l'abstraction de ces deux détails d'implémentation afin que les processus utilisateurs et les autres sous-systèmes du noyau puissent accéder au réseau sans nécessairement savoir quels sont les périphériques physiques ou protocoles utilisés.

I.3.5 Les communications inter processus

Le module de communication inter processus permet aux processus utilisateurs de communiquer entre eux et avec le noyau, pour se synchroniser. La synchronisation permet de réglementer les accès concurrents à une ressource partagée, qu'elle soit matérielle ou logicielle. Sous certaines formes, elles permettent aussi d'échanger de l'information.

C'est par ce sous système que l'espace utilisateur accède le plus régulièrement au noyau.

I.4 L’espace Utilisateur

Le middleware Révolution s’exécute dans l’espace utilisateur. Il est composé d’un certain nombre de processus et threads qui utilisent les mécanismes Linux pour communiquer et se synchroniser. Ces mécanismes sont accessibles à travers la librairie Glibc. Je vais me limiter à une présentation des informations les plus utiles sur cet espace qui permettra de mieux appréhender la partie réalisation de ce mémoire.

L’espace utilisateur (*user space*) correspond à l’espace où sont développées les différentes applications comme par exemple : les environnements de bureau, les logiciels de bureautique ou les middlewares comme « Revolution-S ».

Dans le système Linux, l’espace utilisateur permet de cloisonner les différentes applications entre elles, mais aussi avec l’espace noyau. Celles-ci ne sont donc pas autorisées à effectuer certaines opérations dites ``privilégiées''. Tout particulièrement, elles ne sont pas autorisées à modifier leur espace d'adressage pour accéder à l'espace d'adressage d'une autre application. Seul le noyau est autorisé à prendre en charge ces opérations.

Par conséquent le processeur doit supporter deux modes d'exécution des instructions : le mode privilégié (*root*) et le mode non privilégié (*user*). Les applications utilisateurs fonctionnent en mode utilisateur et font appel aux services du noyau au travers d'un changement de mode parfaitement contrôlé et qui est l’appel système. Ce qui correspond à un appel de fonctions avec changement de privilèges.

Les applications dans l’espace utilisateur disposent donc de pouvoirs restreints sur la mémoire, ils doivent demander au noyau de la mémoire. Le noyau fait appel à son gestionnaire de mémoire pour allouer (ou non) la mémoire au processus qui la demande. Si un processus tente d’utiliser des zones de mémoire ne lui appartenant pas, il est évincé automatiquement. Le mécanisme d’éviction repose sur un mécanisme du processeur, nommément une unité de gestion de la mémoire (*MMU*), qui signale au noyau l’existence d’un accès fautif. C’est le noyau lui-même qui prend la décision de suspendre ou détruire immédiatement le processus fautif.

Une application se compose d’un ou plusieurs processus qui sont eux-mêmes composés d’un ou plusieurs threads.

I.4.1 Les processus et les threads

Le fonctionnement multitâche s'exprime traditionnellement sur les systèmes Linux sous forme de processus. Pourtant, l'ordonnancement n'est que très faiblement concerné par les processus et repose en réalité sur la notion de thread. Un processus correspond plus exactement à un espace indépendant de mémoire virtuelle, dans lequel un ou plusieurs threads s'exécutent. La notion de thread, notamment leur normalisation par le standard Posix, est relativement récente.

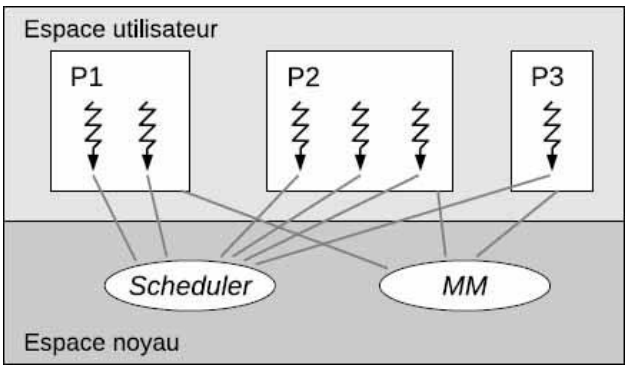


Figure 17 : Représentation des processus et threads Linux

Sur la figure ci-dessus, nous voyons que la commutation entre les threads est gérée par l'ordonnanceur (*scheduler*) tandis que la gestion des espaces de mémoire virtuelle des processus est assurée par le sous-système MM (*Memory Management*) du noyau. Nous pouvons remarquer que le processus P3 ne comporte qu'un seul thread, c'est le cas de la plupart des applications classiques sous Linux.

Un thread peut donc endommager les données d'autres threads du même processus car les threads partagent le même espace mémoire et leurs ressources. Par contre un processus corrompu ne peut pas agir de cette façon, car chaque processus dispose de sa propre copie de l'espace mémoire du programme.

Les threads ne sont intéressants que dans les applications assurant plusieurs tâches en parallèle. Si chacune des opérations effectuées par un logiciel doit attendre la fin d'une opération précédente avant de pouvoir démarrer, il est totalement inutile d'essayer une approche multithreads.

Le partage de données entre des threads est trivial car ceux-ci partagent le même espace mémoire. Le partage de données entre des processus nécessite l'utilisation de mécanismes de communication inter-processus (*IPC*).

I.4.2 Les communications inter-processus

Les communications inter-processus regroupent un ensemble de mécanismes permettant à des processus concurrents de communiquer. Ces mécanismes permettent l'échange de données et/ou la synchronisation entre processus.

Pour l'échange de données, il existe deux mécanismes assez proche, la mémoire partagée et la mémoire mappée.

- La mémoire partagée

Elle permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient appelé *malloc()* et avaient obtenu des pointeurs vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.

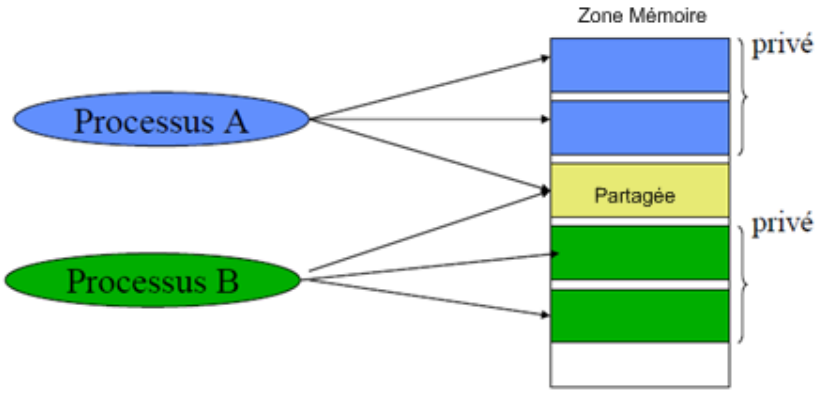


Figure 18 : Principe de la mémoire partagée

- La mémoire mappée

Elle permet à différents processus de communiquer via un fichier partagé. Elle peut être utilisée pour la communication inter-processus ou comme un moyen pratique d'accéder au contenu d'un fichier.

Les segments de mémoire partagée permettent une communication bidirectionnelle rapide entre n'importe quel nombre de processus. Chaque utilisateur peut à la fois lire et écrire, mais un programme doit définir et suivre un protocole pour éviter les conditions de concurrence critique, comme par exemple écraser des informations avant qu'elles ne soient lues. Pour cela il faut utiliser le mécanisme de synchronisation sémaphore.

- Les sémaphores

Ils permettent de coordonner les processus qui doivent accéder à de la mémoire partagée. Un sémaphore peut être vu comme un distributeur de jetons, le jeton étant un droit à poursuivre son exécution. Il préserve ainsi les accès concurrents entre processus.

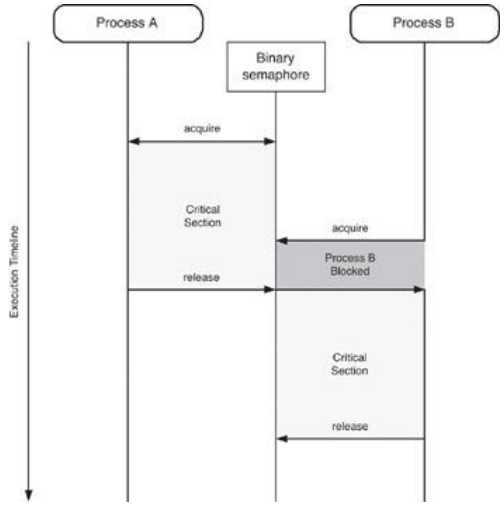


Figure 19 : Principe du sémaphore

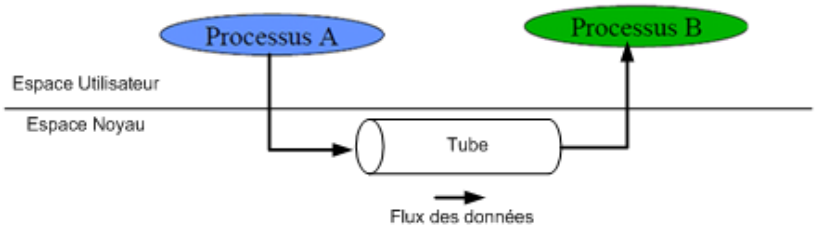
- Les signaux

Le principe est à priori simple : un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même). Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement une mesure spécifique. Le destinataire peut soit ignorer le signal, soit le capturer, c'est-à-dire dérouter provisoirement son exécution vers une routine particulière que l'on nomme gestionnaire de signal. Mais le destinataire peut également laisser le système traiter le signal avec un comportement par défaut.

La plupart des signaux ne sont pas émis par des processus applicatifs, mais directement par le noyau en réponse à des conditions logicielles ou matérielles particulières.

- Les tubes

C’est un dispositif de communication qui permet une communication à sens unique. Les données écrites sur l'« extrémité d'écriture » du tube sont lues depuis l'« extrémité de lecture ». Les tubes sont des dispositifs séquentiels ; les données sont toujours lues dans l'ordre où elles ont été écrites. Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus père et fils.



20 : Principe des tubes Linux

La capacité d'un tube est limitée. Si le processus écrivain écrit plus vite que la vitesse à laquelle le processus lecteur consomme les données, et si le tube ne peut pas contenir de données supplémentaires, le processus écrivain est bloqué jusqu'à ce qu'il y ait à nouveau de la place dans le tube. Si le lecteur essaie de lire mais qu'il n'y a plus de données disponibles, il est bloqué jusqu'à ce que ce ne soit plus le cas. Ainsi, le tube synchronise automatiquement les deux processus.

- File FIFO

Une file premier entré, premier sorti (*First-In, First-Out, FIFO*) est un tube qui dispose d'un nom dans le système de fichiers. Tout processus peut ouvrir ou fermer la FIFO ; les processus raccordés aux extrémités du tube n'ont pas à avoir de lien de parenté. Les FIFO sont également appelées « tubes nommés ».

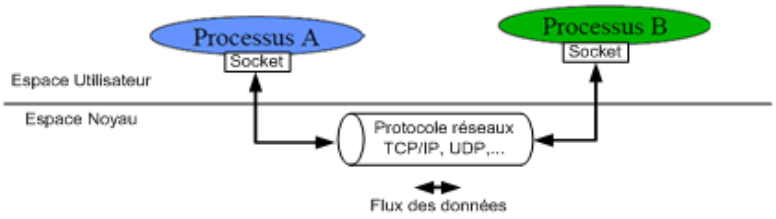
- Les files de messages

Il s'agit d'une implémentation du concept de boîte à lettres. Le but est d'échanger, de façon asynchrone, des listes chaînées. Les files de messages sont gérées par le noyau et contiennent des données organisées sous forme d'un type suivi d'un bloc de message proprement dit. Cette représentation complique un peu la manipulation des messages, mais permet grâce au type transmis, de hiérarchiser par priorité ou d'obtenir un multiplexage, en distinguant plusieurs processus destinataires différents qui lisent sur la même file de messages.

- Les sockets

Une socket est un dispositif de communication bidirectionnel pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus

s'exécutant sur d'autres machines. Il s'agit approximativement d'une extension de la portée des tubes nommés.



21 : Principe des sockets Linux

On peut donc écrire des données dans une socket après l'avoir associée à un protocole de communication. Les couches réseau des deux stations s'arrangeront pour que les données ressortent à l'autre extrémité. La seule complication introduite par rapport aux tubes classiques est la phase d'initialisation, car il faut indiquer l'adresse et le numéro de port du correspondant. Une fois que la liaison est établie, le comportement ne sera pas très différent d'un tube.

Tous ces mécanismes de communication sont gérés par le noyau. Pour pouvoir les utiliser il faut s'appuyer sur les différentes librairies mises à disposition dans Linux comme la Glibc.

I.4.3 Les librairies

Une librairie est un sous-ensemble cohérent de fonctions et de programmes "correctement" validés et testés, regroupés au sein d'une même archive. Il existe deux formes de librairies sous Linux : statiques (.a) et dynamiques (.so).

- Avec une librairie statique

Le contenu de la librairie est copié dans le binaire que l'on compile. Ce dernier est donc plus volumineux que si on l'avait compilé avec une librairie dynamique, mais il est également indépendant des librairies installées sur le système. Enfin, lors de la mise à jour d'une librairie statique, tous les programmes faisant appel à celle-ci devront être recompilés afin que les modifications soient prises en compte.

- Avec une librairie dynamique

Le binaire compilé est lié à cette librairie. C'est une bonne approche pour générer un binaire peu coûteux en espace disque. Toutefois, si la librairie est absente, le programme

ne peut pas se lancer. Enfin, lors d'une mise à jour, seule la librairie dynamique est à recompiler.

Les librairies regroupent des fonctionnalités complémentaires de celles qui sont assurées par le noyau comme par exemple toutes les fonctions mathématiques (le noyau n'utilise jamais les nombres réels). La librairie C permet aussi d'encapsuler les appels-système dans des routines de plus haut niveau et qui sont donc plus aisément portables d'une machine à l'autre.

Il existe, sous Linux, plusieurs implémentations libres de la bibliothèque C standard. La plus connue est « Gnu GlibC » qui est très complète. Mais elle utilise (≈20 Mo) pour un système embarqué.

I.5 Linux dans un système embarqué

I.5.1 Qu'est ce qu'un système embarqué ?

On peut le définir comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise (6). Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte).

Cette définition est très générale car elle recouvre des systèmes de types très différents mais généralement, un système embarqué se résume à :

- un microprocesseur
- OS
- des applications spécifiques
- une limitation de taille mémoire
- une limitation des ressources disponibles
- une interaction limitée avec l'utilisateur
- des fonctionnalités spécialisées

I.5.2 Le décodeur numérique

Le décodeur numérique fait partie des systèmes embarqués assez similaire aux PC mais avec quelques spécificités. L'architecture des processeurs peut être de nature différente (ARM, MIPS, PowerPC ou x86). Le stockage se fait sur de la mémoire flash, de type NOR ou NAND, et de capacité souvent relativement réduite (quelques dizaines de Mo). La capacité de la mémoire vive (RAM) est elle aussi réduite (quelques centaines de Mo). Les CPU intègrent de nombreux périphériques peu courants sur les PC (I2C, SPI, SSP, CAN, etc). Le décodeur numérique fait partie des systèmes embarqués assez similaire aux PC mais avec des ressources mémoires limitées.

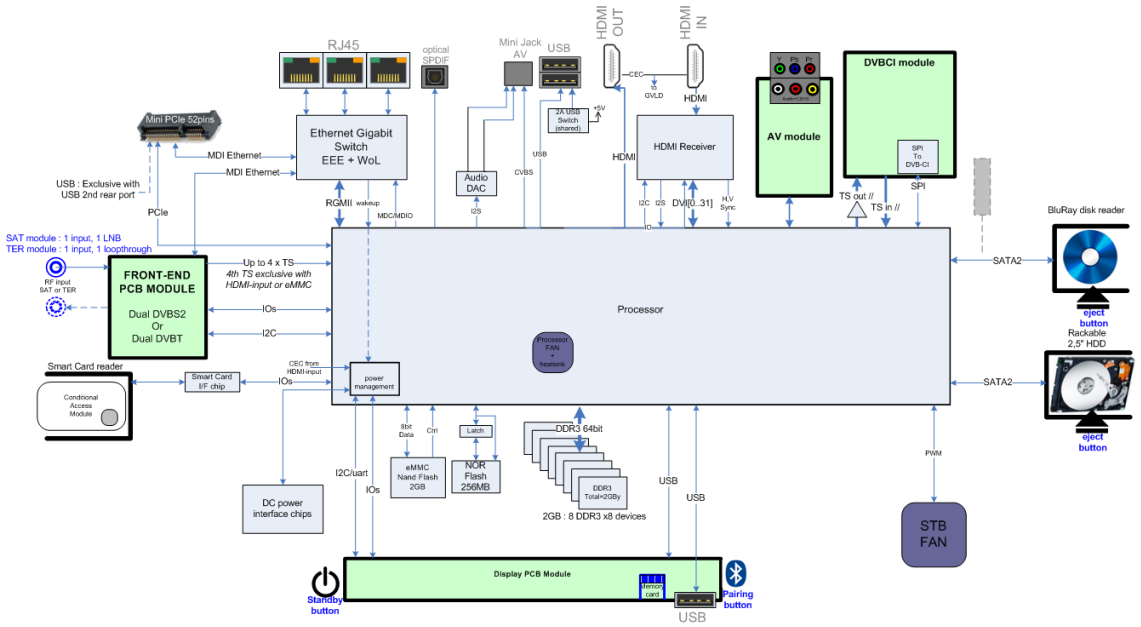


Figure 22 : Architecture matérielle d'un décodeur numérique.

I.5.3 Les spécificités d'un développement Linux pour un décodeur numérique

A la différence de l'informatique traditionnelle, où les tâches réalisées sont généralistes, le logiciel embarqué a un domaine d'actions qui est « limité » aux fonctions pour lesquelles il a été créé.

La durée de vie des produits est longue, car des obligations légales nous obligent à la maintenir pendant une dizaine d'années. Il est donc indispensable que le logiciel embarqué soit « maintenable » durant toute la durée de vie du produit, en cas de découverte d'un problème de fonctionnement, ou lorsque l'ajout de fonctionnalités s'impose. De plus le logiciel nécessite une grande « fiabilité », car il est destiné à un fonctionnement totalement autonome.

Dans l'informatique traditionnelle, un système d'exploitation peut fonctionner sur différents ordinateurs et celui-ci peut faire fonctionner différents systèmes d'exploitation. On peut donc dissocier le logiciel du matériel. Hors, pour les raisons évoquées plus haut, cette distinction ne peut se faire dans un système embarqué. Le système Linux doit donc être configuré au décodeur numérique. Cette adaptation se fait facilement car le système Linux sera installé sur une configuration matérielle connue et par nature très peu évolutive. Permettant ainsi d'installer uniquement le driver nécessaire et obtenir ainsi un système peu encombrant d'un point de vue mémoire.

La difficulté est la mise au point et la correction des erreurs du logiciel. Ceci est en partie dû à l'environnement de développement .

I.5.3.1 L'environnement de développement

Comme pour la réalisation de logiciels ordinaires, nos développeurs ont besoin de compilateur, d'interpréteur, de linker, d'IDE, et d'autres outils de développement. Ils utilisent des distributions Linux classiques car celles-ci intègrent les outils nécessaires à la mise en place de la chaîne croisée.

En effet, comme je l'ai écrit plus haut, le système d'exploitation doit être adapté au décodeur et donc à son processeur. Or celui-ci est de nature différente à ceux utilisés dans les ordinateurs. On utilise alors un certain nombre d'outils qui constituent la « chaîne de compilation croisée » (ou *cross toolchain*). Ceci comprend l'ensemble des outils qui vont permettre de transformer un code source en fichier exécutable spécifique au processeur du décodeur.

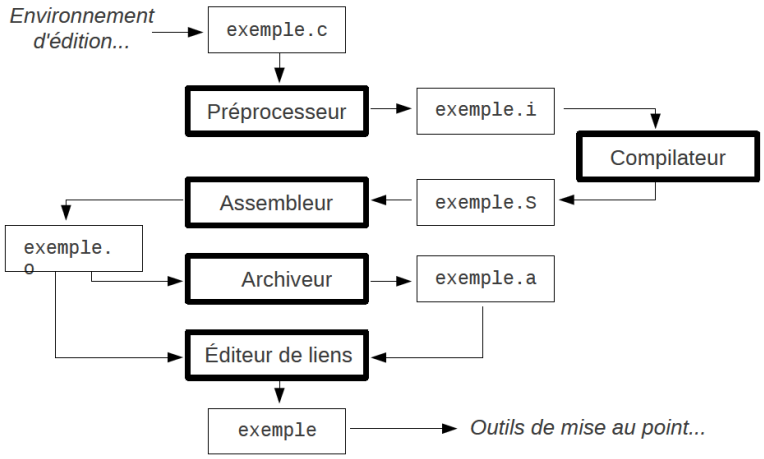


Figure 23 : La chaine de compilation.

Mais aussi les outils de mise au point du logiciel, comme gdb et gdbserver qui permettront de mettre en place la communication entre l'ordinateur et le décodeur. Certains IDE, comme Eclipse qui intègre des interfaces de débogage, s'appuieront alors sur ces outils.

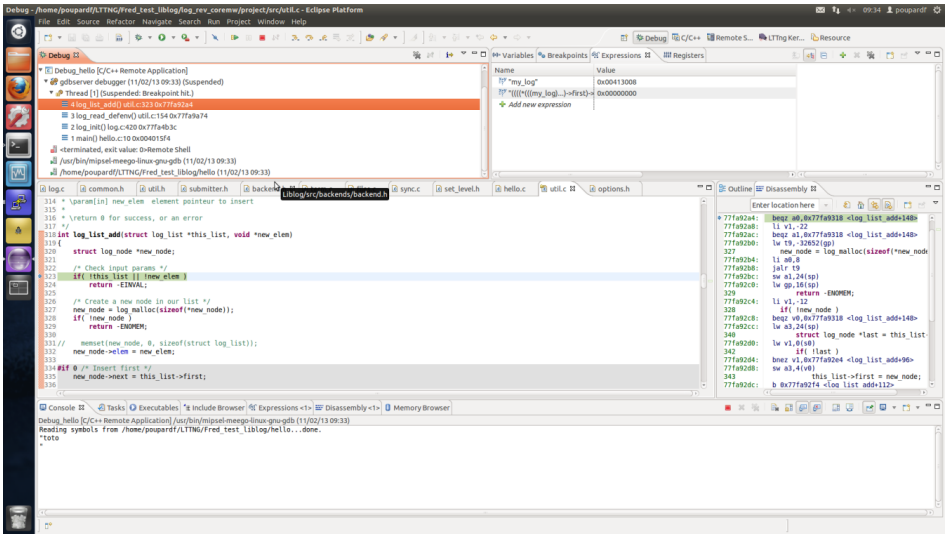


Figure 24 : Eclipse IDE en perspective débogueur.

Enfin certains outils de communication seront nécessaires comme un émulateur de terminal pour se connecter avec un port série (RS232) ou l'interface réseau par un protocole sécurisé comme SSH.

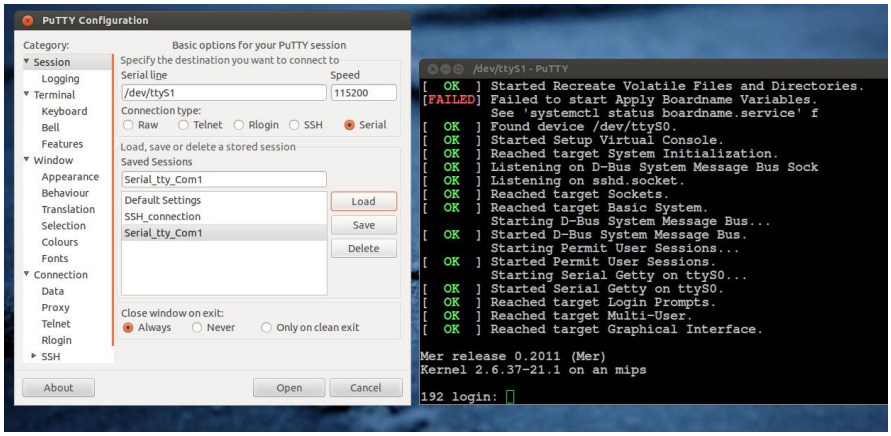


Figure 25 : PuTTY : Emulateur de terminal série

Un système Linux dispose systématiquement d'une console qui est l'interface de communication par défaut. Dans la plupart des systèmes embarqués, la console est un port série, car les processeurs utilisés l'intègrent par défaut. C'est, comme nous allons le voir, un des moyens de communication entre le décodeur et l'ordinateur.

I.5.3.2 La communication avec le décodeur

Si, pour l'utilisateur final, la communication avec le décodeur se fera par le biais des menus affichés sur le téléviseur et de la télécommande, le développeur utilisera généralement ces trois types d'interfaces.

- Le JTAG

La connexion JTAG (*Join Test Action Group*) permet d'accéder en direct au silicium du processeur. C'est une sonde très bas niveau qui permet d'accéder au registre interne mais aussi aux périphériques et aux bus mémoire. Notre utilisation se limite en général à la programmation et au débogage du bootloader que l'on stocke dans la mémoire flash.

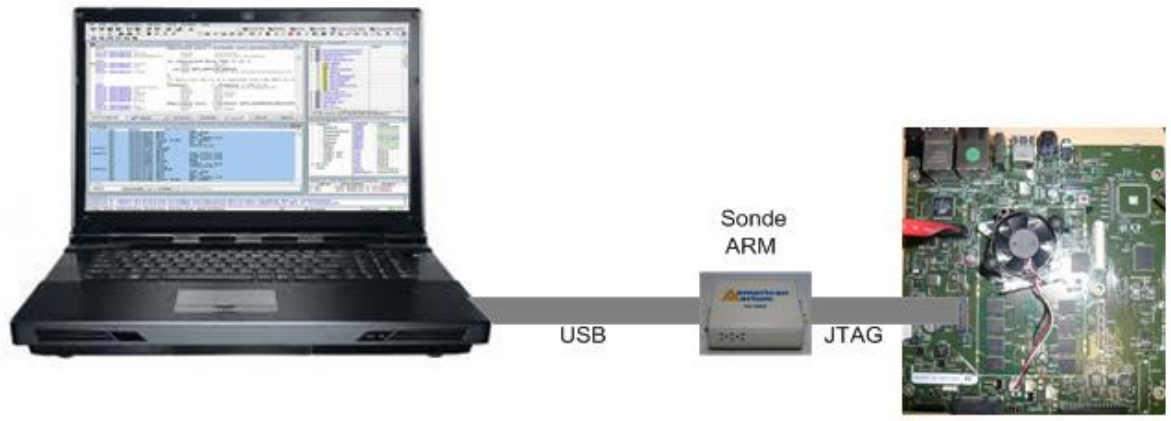


Figure 26 : Sonde JTAG ARM connectée à un décodeur

- La liaison série

La connexion série est l'interface la plus répandue dans le monde de l'embarqué. Car d'un point de vue électrique et logiciel, il est très simple de la mettre en œuvre dans un processeur. Elle sert de port par défaut où sont envoyées les traces générées par le noyau du décodeur. A la fin du démarrage, si un interpréteur de commande a été chargé par le noyau, on pourra alors se connecter au décodeur par un login et un mot de passe pour accéder à l'invite de commande *Shell*.

Figure 27 : Liaison série (type RS232) connectée à un décodeur

Malheureusement, la liaison série, même programmée à 115,2 kb/s, est trop lente pour servir de moyen de téléchargement de fichiers de plusieurs Mo. Cela peut prendre de quelques minutes à quelques dizaines de minutes. On préfère donc utiliser une liaison Ethernet bien plus rapide.

- La liaison Ethernet

La connexion Ethernet est plus complexe à mettre en place qu'une liaison série, aussi bien d'un point de vue électrique que logiciel. Elle permet en revanche des transferts beaucoup plus rapides pour les mises à jour et sert pour le débogage à distance avec la commande gdbserver. Enfin cette liaison permet de mettre en place une connexion sécurisée à l'interpréteur de commande par le protocole SSH.

Figure 28 : Liaison Ethernet connectée à un décodeur

Ces liaisons sont très utilisées durant la phase de développement car elles permettent d'avoir des retours d'informations du logiciel sur son fonctionnement dans le décodeur. Et c'est à travers ces connexions qu'on aura les moyens de corriger et d'améliorer notre middleware.

II Méthodes pour corriger et améliorer un middleware

La mise au point est un aspect fondamental du développement de logiciels industriels. En effet, les contraintes de qualité de fonctionnement y sont bien plus importantes que dans le cas des logiciels généralistes. Si un outil de bureautique est utilisé quelques heures par jour, le logiciel embarqué doit, lui fonctionner 24 heures sur 24, et le redémarrage du système est peu fréquent. La moindre fuite de mémoire, tolérable sur un logiciel généraliste, devient très vite problématique dans le cas d'un logiciel embarqué.

De plus, les problèmes apparaissent beaucoup plus rapidement, car les logiciels embarqués fonctionnent sur du matériel disposant de ressources beaucoup plus limitées. Le facteur de redistribution en masse augmente considérablement les coûts lorsqu'on doit ajouter quelques mégaoctets de mémoire, alors que ce coût est négligeable lorsqu'il s'agit d'un poste de travail classique.

Avant de décrire les différents outils utilisables pour mettre au point un programme sous Linux dans un environnement embarqué, je vais définir les principales techniques qui permettront de classifier ces outils et définir lesquelles pourront être utilisées dans la solution finale.

II.1 Le débogueur

Le débogueur est un outil précieux pour étudier le fonctionnement structurel d'un programme. Il permet au développeur de le stopper selon diverses conditions, d'en inspecter des variables, voire de modifier dynamiquement son état. Cependant, il ne fournit pas d'information sur l'aspect temporel de son exécution. Son utilité est donc très limitée dans le contexte de problèmes de performance, dont la nature même est temporelle.

II.2 La trace

La notion de trace correspond à l'affichage de messages d'informations au cours de l'exécution du programme. La mise en place de traces n'est pas la meilleure méthode, car elle est intrusive par définition. Utilisée avec rigueur, elle constitue cependant un apport indispensable à la mise au point et à la maintenance d'un système. Notons qu'il est possible de tracer aussi bien dans l'espace utilisateur que dans l'espace noyau.

S’il choisit correctement les moments et les informations à imprimer, le développeur peut, en relisant cette trace d’exécution, comprendre où se situe le bogue. En imprimant également l’heure à laquelle chaque message est généré, et ce avec un bon niveau de précision, il obtient la capacité d’isoler des problèmes de performances simples.

Il existe plusieurs méthodes pour aborder les traces. Le développeur peu scrupuleux aura tendance à parsemer son code d’appels sauvages aux fonctions printf ou fprintf, ce qui a quelques conséquences fâcheuses. Le code source perd en clarté, donc en possibilité de maintenance et on trouvera fréquemment des lignes comme printf ("ici\n"). L’accumulation de traces peut aggraver sévèrement les performances s’il n’existe pas de système permettant de limiter l’affichage (configuration du niveau de trace). Le côté intrusif des traces peut entraîner des comportements douteux, comme l’apparition d’un problème si l’on supprime la trace.

Enfin, cette technique comporte des limites. Si la cause du problème de performance n’est pas liée à des lignes de code précises du programme, mais à des activités asynchrones, il sera difficile de l’isoler avec cette technique. De plus, l’analyse doit être faite manuellement par des humains, ce qui contraint sévèrement la taille de traces pouvant être analysée.

II.3 L’enregistrement (Log)

L’enregistrement consiste à garder en mémoire l’historique des traces de façon chronologique. Un peu à la manière d’une boîte noire d’avion qui contient les données des capteurs et les discussions du cockpit. En cas de crash, elle permet de savoir si c’est une erreur humaine ou un problème matériel.

L’enregistrement a d'une certaine façon le même but. Lorsqu'un programme s'exécute et plante, il est utile de savoir ce qu'il s'est passé après coup. Cette technique permet de s’affranchir de la connexion permanente à un terminal d’ordinateur. On utilise souvent l’enregistrement lorsque l’apparition du défaut est aléatoire et a une faible occurrence. Une attention particulière devra être apportée au type de traces à enregistrer. Seules les traces importantes devront être gardées car les systèmes embarqués ont peu de mémoire de stockage et un fichier de traces peut vite devenir volumineux.

II.4 Le profilage (Profiling)

Le profilage est différent de la mise au point puisqu'il ne s'agit pas de traquer un bogue, mais plutôt d'estimer la consommation d'une application afin d'en optimiser le fonctionnement. Il permet ainsi de savoir quels sont les processus qui consomment beaucoup de temps CPU. Mais aussi dans quelles fonctions, voire dans quelles lignes de code, le programme a passé le plus de temps.

Il est également utile de savoir quelles sont les lignes de code, à partir desquelles une fonction ayant consommée beaucoup de temps processeur, ont été le plus appelées.

Le profilage consiste donc à enregistrer des événements durant l'exécution d'un programme. Le développeur examine ensuite ces événements qui donne une information sur l'état actuel du système. Ils sont accompagnés de l'heure à laquelle ils sont survenus ainsi que, parfois, d'informations supplémentaires.

Le profilage ressemble à la journalisation. Cependant, ce terme réfère davantage à des approches optimisées pour soutenir en continu de très hauts débits d'événements, souvent plusieurs milliers par seconde ou plus. La haute performance de certains outils permet d'enregistrer des événements de très bas niveau (ex : l'entrée dans un gestionnaire d'interruption), ce qui constitue un avantage par rapport aux journaux. Le profilage génère cependant une quantité telle de données qu'il est habituellement moins facile de consulter une trace qu'un journal.

Contrairement au débogage, qui agit directement sur l'exécution du programme en pouvant l'arrêter à tout moment et même changer les valeurs de variables, le traçage, la journalisation et le profilage permettent de récupérer un maximum d'informations avec un minimum d'impacts sur une application en cours d'exécution et ceci sans l'interrompre.

De ces quatre techniques, la trace sur le terminal d'un ordinateur est la plus couramment utilisée chez Technicolor. Une des raisons est qu'il existe une multitude d'outils dans le monde de l'open source mais aucun choix clair n'a été fait. La plupart des développeurs ont alors utilisé la solution de facilité qui est le printf vers la console du terminal.

III Les outils disponibles dans le monde LINUX

Il existe donc une multitude d'outils traitant de la mise au point sous Linux, mais ceux-ci ne sont pas toujours bien documentés ni adaptés aux besoins précis de l'utilisateur. Il n'existe pas forcément, comme dans le monde propriétaire, un outil prêt à l'emploi. Je vais donc présenter ma recherche sur ces différents outils et donner une idée des problèmes qu'ils aident à résoudre et quelles techniques ils utilisent.

III.1 Propriétaire

III.1.1 DTrace

- Usage sur la cible : Débogage, traçage, profilage

DTrace(7) est un traceur dynamique développé par Sun Microsystems et intégré aux versions récentes de Solaris et de Mac OS. Il est à la fois un fournisseur et collecteur de données. L'utilisateur de DTrace peut coder, à l'aide du langage D de DTrace, des analyses simples qui ont lieu de façon synchrone pendant l'enregistrement de la trace.

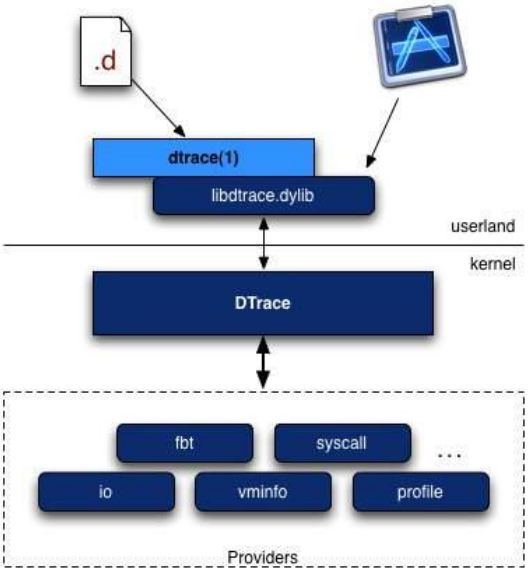


Figure 29 : Dtrace, vue globale

L'instrumentation de DTrace s'appuie sur des fournisseurs qui offrent une interface standard d'accès à de l'instrumentation de nature variée. DTrace permet donc d'activer des points d'instrumentation dans le noyau et dans les processus durant l'exécution. Dans certains cas, il s'agit d'une instrumentation statique qui est alors activée. Dans d'autres cas, comme les entrées et sorties de fonction, l'activation d'un point d'instrumentation se fait

de façon dynamique. En effet, ce n'est qu'au moment de l'activation du point d'instrumentation que DTrace modifie le code pour y ajouter un point de trace.

Lorsqu'un événement se déclenche, DTrace exécute les actions spécifiées par l'utilisateur dans un script en langage D. On peut filtrer l'événement en fonction de la valeur de variables instrumentées, mettre à jour des variables d'état ou imprimer des informations. Le script n'a cependant accès qu'en lecture à la mémoire du noyau afin d'en garantir la stabilité.

Il existe des interfaces graphiques comme Shark et Instruments qui permettent de générer les scripts directement pour Dtrace et de visualiser graphiquement les différents événements.

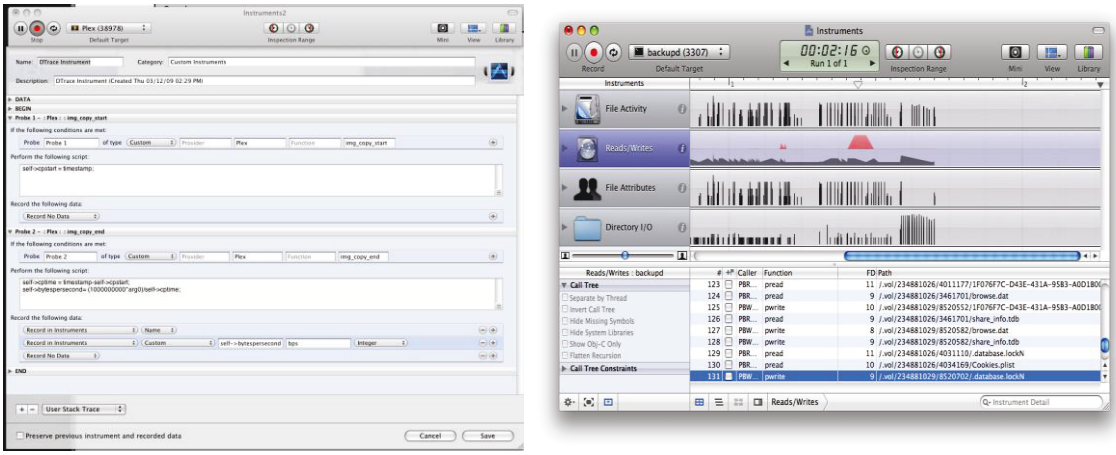


Figure 30 : Logiciel « Instruments » pour Dtrace

- Mes remarques :

Dtrace a été développé par la société Solaris et ne peut être actuellement utilisé avec un noyau Linux pour des raisons de licence.

III.2 Open source

III.2.1 SystemTap

- Usage sur la cible : Débogage, traçage, profilage

SystemTap(8) est un outil de traçage similaire à DTrace permettant la collecte d'information à partir d'un système Linux en cours d'exécution. Cet outil permet l'insertion de points de traces dynamiques ainsi que la collecte de données à partir des points de traces statiques. La trace collectée peut être affichée sur la console au fur et à mesure qu'elle est

produite. Sinon, le mode flight recorder peut enregistrer la trace soit en mémoire soit dans un fichier temporaire pour qu'elle soit analysée plus tard.

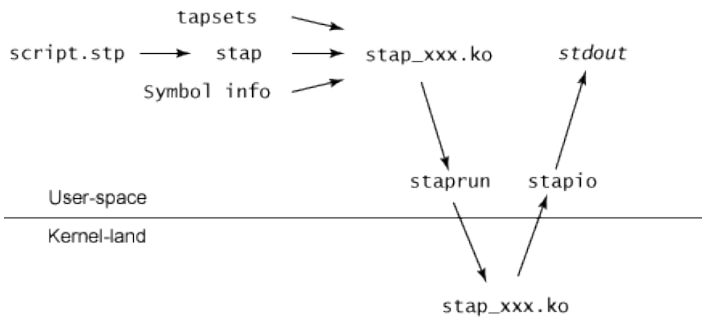


Figure 31 : Le processus de SystemTap

Tout comme DTrace, le code d'instrumentation de SystemTap est écrit sous forme de scripts. Le langage utilisé a une syntaxe très proche de celle de C. Il supporte toutes les opérations de ANSI C. Les types de données supportés sont seulement les entiers et les chaînes de caractères.

Les scripts SystemTap sont convertis en code C, qui sont compilés après sous forme de module noyau. Ce module est alors inséré et communique avec SystemTap pour le traçage.

Les points de traces dynamiques peuvent être insérés dans la quasi-totalité du code du noyau. SystemTap offre une multitude d'événements auxquels on peut associer des points d'instrumentation. On peut citer les suivants :

- les points d'entrée et de sortie de toutes les fonctions traçables du noyau.

- un certain emplacement dans le code du noyau.

- une certaine adresse dans le binaire.

- Mes remarques :

SystemTap a beaucoup de dépendance avec des outils non intégrés dans la version officielle du noyau. De plus, il est assez dur d'utilisation et nécessite de connaître le fonctionnement du code du noyau pour obtenir des données précises et pertinentes.

III.2.2 LTTng

- Usage sur la cible : Débogage, traçage, profilage

Le traceur LTTng(9) est le premier traceur à avoir pris en considération l'importance de l'impact sur le système. Depuis ses débuts en 2006, son développement a

toujours progressé à l'extérieur des sources du noyau officiel, mais il est en cours d'intégration dans la version officielle du noyau de Linux.

LTTng est un ensemble d'outils en espace noyau et utilisateur permettant de générer, contrôler et analyser des traces. Avec la version 2.0, tout le cœur du traceur fonctionne sous la forme de modules. Les traces générées dans la version 2.0 sont au format Common Trace Format

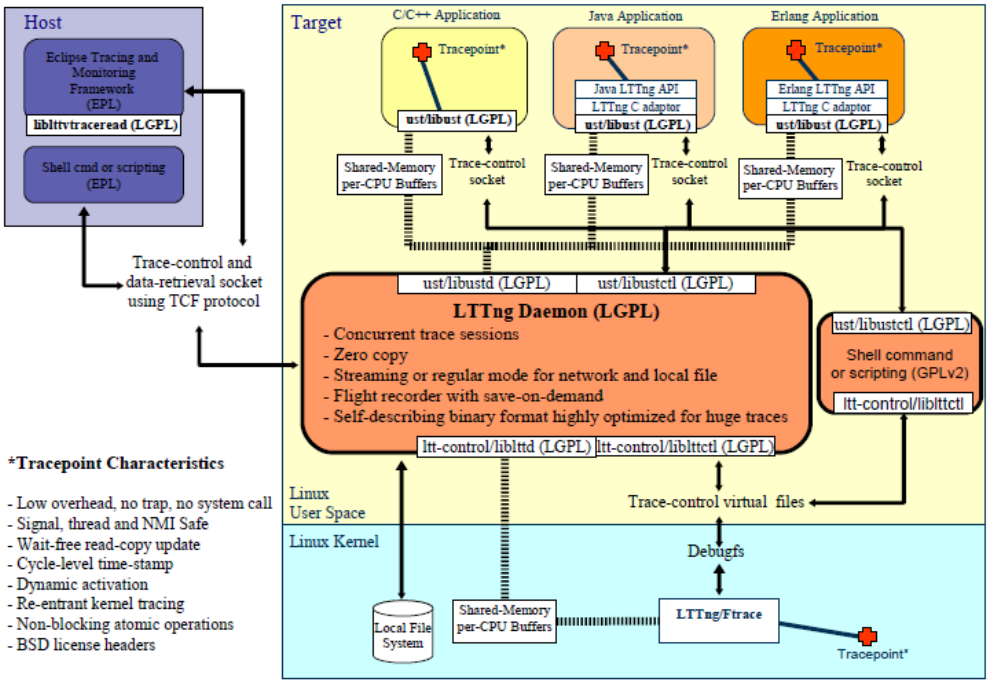


Figure 32 : LTTng vue globale

Il permet d'effectuer des observations multi-niveaux, depuis la plate-forme matérielle jusqu'à l'espace utilisateur. L'outil définit des fonctions de traçages génériques afin de fournir une interface commune d'observation et de rester portable entre les différentes plates-formes. Cependant, les fonctions de traçage nécessitent des mécanismes d'estampillage spécifiques à chaque plate-forme pour être plus précises et moins intrusives.

LTTng permet de tracer principalement des évènements du système d'exploitation (interruptions, changements de contexte, appels aux fonctions du système, etc.). Ils sont appelés points d'instrumentation. Les utilisateurs de LTTng peuvent activer les points d'instrumentation de leurs choix. Le développeur peut aussi implémenter des messages dans son code source qui seront envoyés à l'outil. Il sera alors possible de voir le contexte du noyau au moment de l'apparition du message.

LTTng est souvent utilisé dans le monde de l'embarqué car l'environnement de développement Eclipse intègre un plugin spécifique pour cet outil.

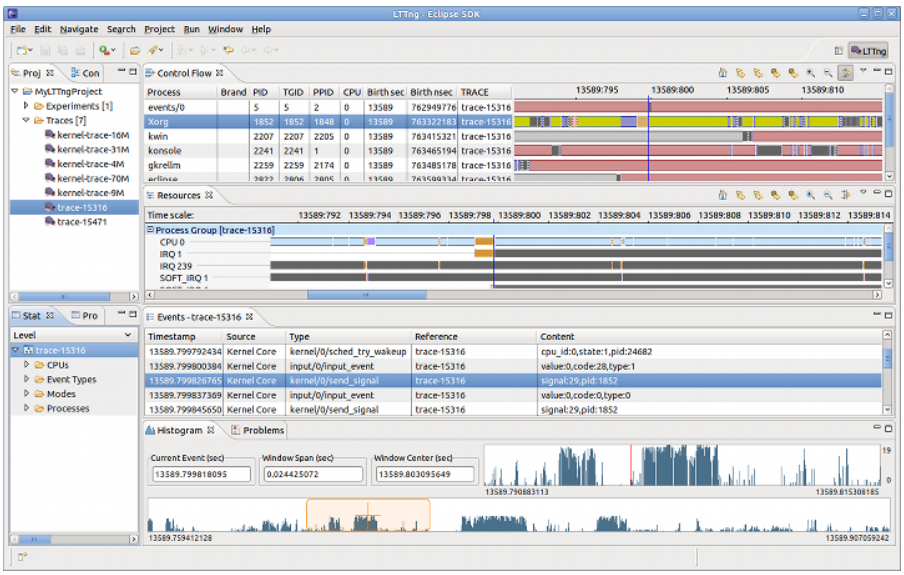


Figure 33 : Eclipse, interface graphique pour LTTng

- Mes remarques :

LTTng n'est pas inclus dans la livraison officielle du noyau mais s'appuie uniquement sur des outils qui le sont. La possibilité de rajouter un plugin à eclipse pour le contrôler et visualiser le résultat est très intéressante.

III.2.3 Utrace

- Usage sur la cible : traçage

Utrace(10) est un ensemble de codes non intégré au noyau, mais fournit par des distributions comme Fedora et Red Hat. Il remplace de manière transparente les fonctionnalités fournies par ptrace, tout en rajoutant une composante de traçage en espace utilisateur. Un point intéressant de ce traceur est que l'interface de programmation est standardisée et compatible avec les fichiers d'en-tête de Dtrace.

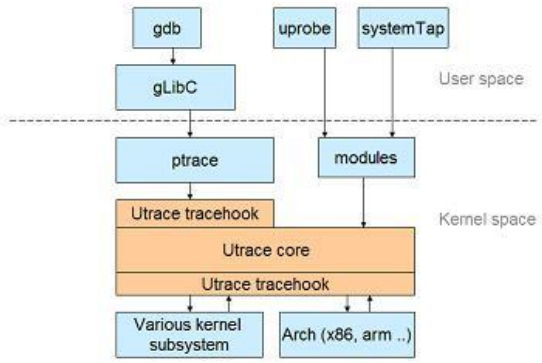


Figure 34 : Utrace, vue globale

Utrace est une infrastructure qui monitor individuellement les threads utilisateurs au niveau du noyau. Typiquement, chaque client de Utrace établit un « moteur » (unité de contrôle dans Utrace) pour chaque thread qu'il souhaite tracer . Utrace propose trois fonctionnalités pour ces unités de contrôle.

- Information d'évènement

L'utilisateur peut être informé d'un appel système qui l'intéresse lorsque son thread le rencontre : ceci inclut les appels d'entrée/sortie, signals, exec, clone, exit, etc.

- Contrôle du thread

L'utilisateur peut demander que son thread soit mis en pause dans l'espace utilisateur, et faire du single-step, block-step, etc.

- Accès à l'état machine du thread

Lors d'un appel système, l'utilisateur peut consulter et/ou modifier l'état machine du thread et de ses registres.

Utrace fonctionne en plaçant des points de traces à des points stratégiques dans le code du noyau. Cela génère alors un appel système lors de chaque instruction tracée, ce qui implique un impact majeur dans les performances de l'application tracée.

- Mes remarques :

Utrace a un impact non négligeable sur les performances du système tracé ce qui est peu tolérable dans un système embarqué temps réel.

III.2.4 OProfile

- Usage sur la cible : traçage

Oprofile(11) est un outil qui se sert de l'échantillonnage pour lire les compteurs de performance du processeur et corréler les changements avec les adresses de code où elles ont eu lieu. Il est donc possible d'obtenir, pour chaque processus, fonction ou ligne de code, un profilage du temps qui y a été passé, mais également du nombre de fautes de cache de données, de fautes de cache d'instructions, de fautes de TLB, etc. qui y sont survenues.

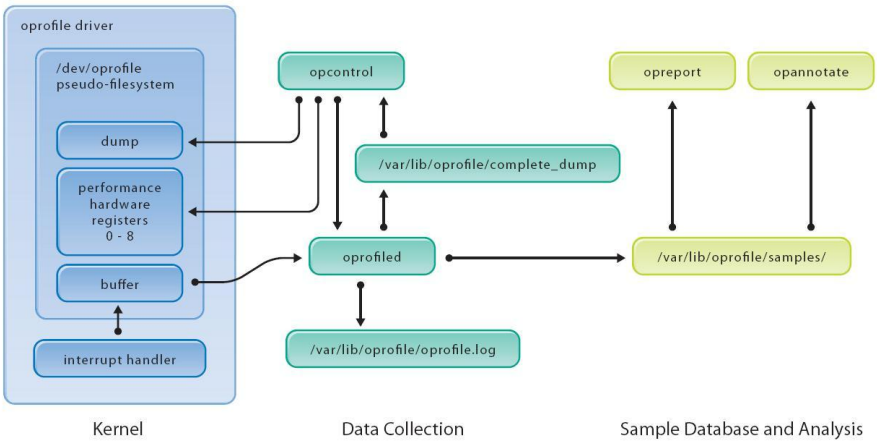


Figure 35 : Oprofile, vue globale

Oprofile est un outil très utilisé pour l'optimisation de très bas niveau (dépendant de l'architecture). Il recueille des informations sur le comportement du logiciel pendant son exécution. Les blocages sont donc hors de sa portée.

- Mes remarques :

Oprofile est trop bas niveau pour que l'on puisse faire la corrélation avec le middleware.

III.2.5 Kprobes

- Usage sur la cible : Débogage, traçage

Kprobes(12) est un mécanisme de débogage développé par IBM. Il faisait partie d'un outil de traçage de plus haut niveau appelé Dprobes. Kprobes a été intégré dans le noyau Linux depuis la version 2.6.9. Il s'agit d'une interface permettant d'insérer dynamiquement des points d'instrumentation dans le noyau au moment de l'exécution.

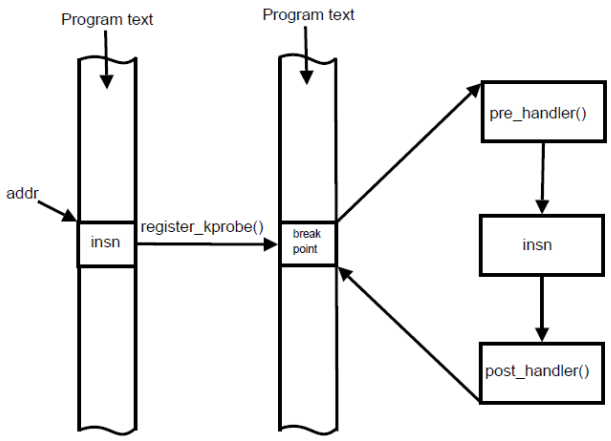


Figure 36 : Processus de Kprobes

Le principe de Kprobes est simple. Il s'agit d'associer un événement à une adresse dans le code du noyau. À chaque fois que l'exécution arrive à cette adresse, l'événement est lancé. Kprobes est capable de s'insérer dans la quasi-totalité du code du noyau et permet d'associer une fonction à exécuter à chaque événement.

Pour chaque événement, on y associe deux fonctions. La première, appelée pre_handler, est exécutée avant l'instruction se situant à l'adresse du point d'instrumentation. La deuxième, appelée post_handler est exécutée après. Quand le pre_handler et le post_handler sont appelés, une copie de l'état de tous les registres leur est passée dans une structure. Cette dernière possède d'autres champs comme celui de l'adresse de la fonction à appeler en cas d'erreur. La forme structure change de contenu d'une architecture à l'autre et contient entre autres les registres à usage général du processeur.

Les Kprobes sont utilisés dans la plupart des outils de traçage noyau récents tels que Ftrace, LTTng et SystemTap.

- Mes remarques :

Les Kprobes sont très orientés noyau et on ne peut pas implémenter de sonde dynamique dans l'espace utilisateur.

III.2.6 Tracepoints

- Usage sur la cible : Débogage, traçage

Tracepoints(13) sont des endroits spécifiés dans le code du Noyau. Ce sont les développeurs du Noyau qui ont estimé les endroits où mettre les tracepoints. L'activation

d'un Tracepoints se fait en appelant la macro TRACE_EVENT(). Cette macro prend 6 paramètres :

- name : correspond au nom du point de trace.
- proto : le prototype des fonctions qu'on pourra connecter au point de trace.
- args : les arguments qui correspondent au prototype.
- struct : la structure qui servira à extraire les données à collecter à partir des arguments passés au callback.
- assign : les assignations nécessaires pour remplir la structure.
- print : la façon dont les données extraites dans la structure seront affichées.

Chaque paramètre est défini à son tour à l'aide d'une macro. Ceci permet de fournir plus qu'un élément par paramètre et aussi de rendre ce mécanisme assez flexible pour qu'il soit utilisé par les traceurs.

Comme les Kprobes, les tracepoints sont utilisés dans la plupart des outils de traçage noyau récents tels que Ftrace, LTTng et SystemTap.

- Mes remarques :

Les tracepoints sont nécessaires dans le cadre d'un développement d'outils de profilage.

III.2.7 Ftrace

- Usage sur la cible : Débogage, traçage, profilage

Ftrace(14) est un mécanisme de traçage pour le noyau de Linux. Il inclut un mécanisme de fournisseur de données, de collecteur de données et des analyses simples. Le formatage des données sous forme de texte et l'analyse ont lieu dans le noyau. La trace en format texte peut être lue directement d'un fichier du système de fichiers DebugFS

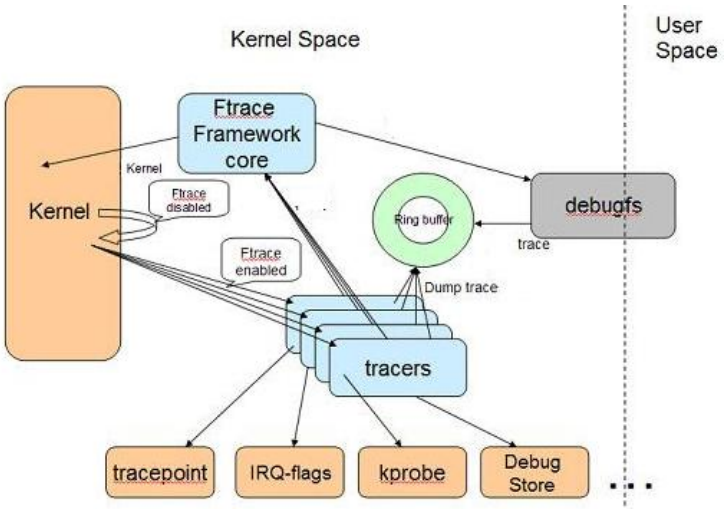


Figure 37 : Ftrace, vue globale

Ftrace est constitué de divers « traceurs » conçus chacun pour un type de problème donné. Voici une liste des événements qu'ils tracent et les principales informations qui les accompagnent.

- Le changement de contexte

Les informations sont : le temps, les processus impliqués, le mode du processus qui se fait réveiller, le numéro du processus.

- Les zones de désactivation des interruptions ou de la préemption

Les informations sont : processus en cours d'exécution, durée de désactivation, fonction où la désactivation et la réactivation ont eu lieu.

- La latence entre le réveil et l'ordonnancement de la tâche temps-réel la plus prioritaire

Les informations sont : tâche, durée, fonction du réveil, fonction du changement de contexte.

- L'exécution de fonctions

Les informations sont : processus, numéro de processeur, temps, fonction, fonction appelante.

L'approche de Ftrace est de fournir un système de traçage précis pour une tâche précise. Chacun de ces traceurs est donc simple et efficace pour le débogage d'un problème

précis pour lequel il est conçu. Son utilité est cependant limitée dans un contexte général où le bogue fait intervenir plusieurs modules du noyau, voire une ou plusieurs applications. Enfin Ftrace n'est pas conçu pour être facilement modifiable afin de tracer d'autres événements.

- Mes remarques :

L'inconvénient de cet outil est l'absence d'une interface graphique comme pour LTTng ou Dtrace

III.2.8 Perf_events

- Usage sur la cible : Débogage, profilage

Le traceur perf_event est un autre traceur intégré au noyau Linux. Celui-ci a été conçu à l'origine pour fournir une interface facile d'accès vers les compteurs de performance présents dans les processeurs. Depuis, il a évolué pour également s'interfacer avec la macro TRACE EVENT et fournir le même type de données. Ainsi, il est possible de l'utiliser pour produire rapidement des statistiques sur le nombre de fois qu'un point d'instrumentation a été exécuté, ou le nombre d'événements enregistrés sur un processeur pendant une période donnée. Le traceur se contrôle par l'outil perfs situé dans le répertoire tools/ des sources du noyau.

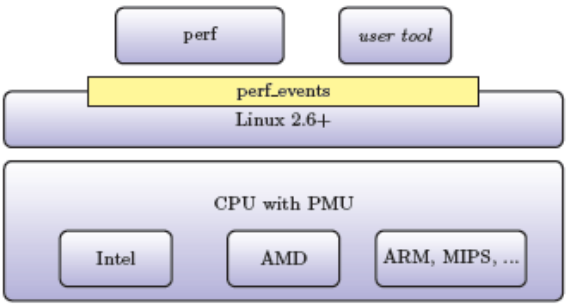


Figure 38 : Perf_envents, vue globale

Perf_events supporte les compteurs de performance hardware, les tracepoints, les sondes dynamiques telles que Kprobes, etc. l'outil perfs permet d'utiliser plusieurs sous commandes.

- Stat : mesure le nombre total d'événement d'un programme ou du système pendant un temps donnée.

- Top : affiche une vue dynamique des fonctions les plus utilisatrices de ressources CPU.
- Record : mesure et enregistre les données d'échantillonnage pour un programme donné.
- Report : Analyse le fichier généré par la commande record. L'affichage peut être à plat ou de manière graphique
- Sched : trace/ mesure les actions de l'ordonnanceur et les latences
- List : affiche la liste des évènements disponibles dans le noyau.

Un usage basique est de réaliser un enregistrement d'un programme, perf va alors collectionner les données dans un fichier. la commande perf-report permettra de le visualiser.

- Mes remarques :

L'inconvénient de cet outil est l'absence d'une interface graphique comme pour LTTng ou Dtrace pour afficher le fichier de rapport. De plus la documentation est assez pauvre.

III.3 En résumé

La liste que je viens de présenter n'est pas exhaustive, j'aurai pu parler de gprof, gcov, ptrace, strace, ltrace, uprobe, valgrind, callgrind, et bien d'autres encore. Mais comme le montre la figure ci-dessous tous ces outils interagissent entre eux.

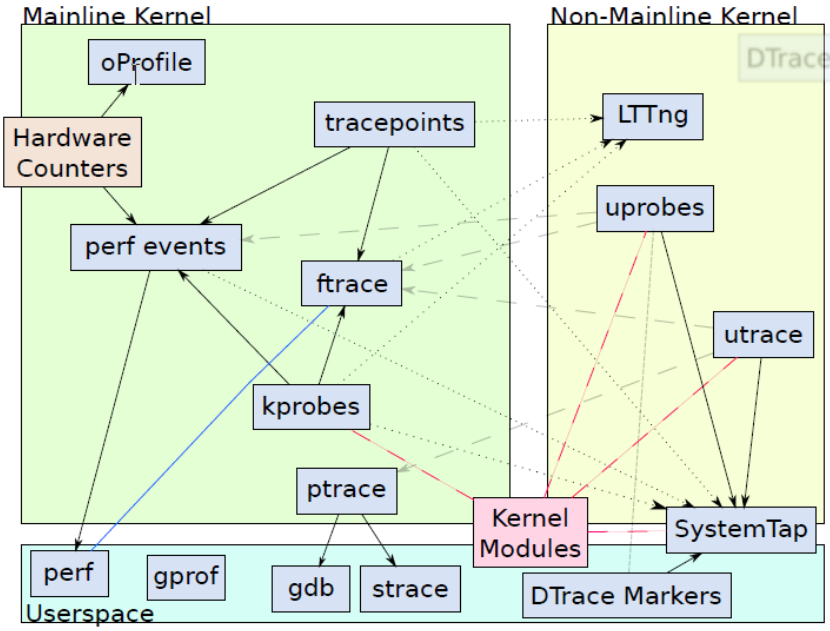


Figure 39 : Interactions entre les outils d'instrumentation Linux(15)

Cette multitude d'outils est due à l'essence même de l'open source. N'importe quel développeur peut récupérer les sources d'un tiers pour l'adapter à son besoin à condition qu'il redonne à son tour l'accès à son code source. Ces outils sont généralement prévus pour un besoin précis. L'expression du besoin permettra de définir si l'un d'entre eux pourra nous convenir.

IV L'outil disponible dans le middleware « Revolution-S »

Il existe dans le middleware « Revolution-S » une librairie de traçage et d'enregistrement appelée « Trace & Logging ». Cette librairie s'appuie sur le protocole Syslog pour créer et transmettre son information.

IV.1 Le protocole Syslog

Le protocole Syslog est un protocole très simple et très largement utilisé dans le monde Linux. Son but est de transporter par le réseau les messages de journalisation générés par une application vers un serveur hébergeant un serveur Syslog.

Le protocole Syslog définit la notion de périphérique, de relais et de collecteur dans une architecture Syslog.

- Un périphérique est une machine ou une application qui génère des messages Syslog.

- Un relais est une machine ou une application qui reçoit des messages Syslog et les retransmet à une autre machine.
- Un collecteur est une machine ou une application qui reçoit des messages Syslog mais qui ne les retransmet pas.

Tout périphérique ou relais sera vu comme un émetteur lorsqu'il envoie un message Syslog et tout relais ou collecteur sera vu comme un récepteur lorsqu'il reçoit un message Syslog.

Le protocole Syslog définit les notions de fonctionnalité, de sévérité et de priorité d'un message Syslog.

- La fonctionnalité d'un message Syslog correspond au type d'application générant le message Syslog. Les 24 fonctionnalités existantes sont définies par la RFC 3164(16)
- La sévérité d'un message Syslog correspond au degré d'urgence du message. Les 8 sévérités existantes sont définies par les RFC
- La priorité d'un message Syslog est définie par sa fonctionnalité et sa sévérité. Cette priorité est un nombre qui est le résultat de la multiplication de la fonctionnalité par 8 auquel est ajoutée la sévérité.

Le protocole Syslog est un protocole en mode "texte", c'est-à-dire qu'il utilise uniquement les caractères du code ASCII. Une trame de protocole Syslog est composée de 3 parties :

- La partie PRI est un nombre qui représente la priorité (en base 10) du message
- La partie HEADER contient le champs TIMESTAMP pour l'horodatage.
- La partie MSG qui comprend le message texte à transférer.

IV.2 L’objectif de « Trace & Logging »

« Trace & Logging » est une librairie que peut utiliser les différents composants du middleware. L’ objectif étant d’offrir aux développeurs la possibilité de garder une trace des événements qui les aideront dans le cadre du développement. Pour ne pas perdre, la valeur des traces importantes ou celle d’une application en cours de débogage dans un flot de traces. deux concepts principaux seront utilisés :

- Un niveau de trace afin de classer le message en fonction de sa gravité ou de sa portée

- Un espace de nommage afin de créer des journaux distincts pour chaque application logicielle.

IV.3 L'architecture de « Trace & Logging »

Il y a deux aspects complètement orthogonaux dans un système de trace. Comment les émettre et que deviennent-elles ? Ces deux aspects ont généralement un intérêt pour des publics différents et dont « Trace & Logging » dissocie entièrement.

Pour les développeurs, la librairie offre une API légère qui leur permet d'émettre des messages d'une manière cohérente sans se soucier de leur destination. Pour les développeurs, les intégrateurs et les testeurs, la librairie propose une série de sorties « back-end » qui détermine ce qui se passe sur les messages, en ce qui concerne :

- leur transport.
- leur stockage.
- leur présentation et leur mise en forme.

Toutes ces sorties « back-end » peuvent être activées et configurées séparément. Certaines sorties attireront davantage les développeurs pour un retour rapide sur leur application en cours d'exécution. Tandis que d'autres sorties intéresseront plus les testeurs pour la journalisation des messages importants provenant des différentes applications.

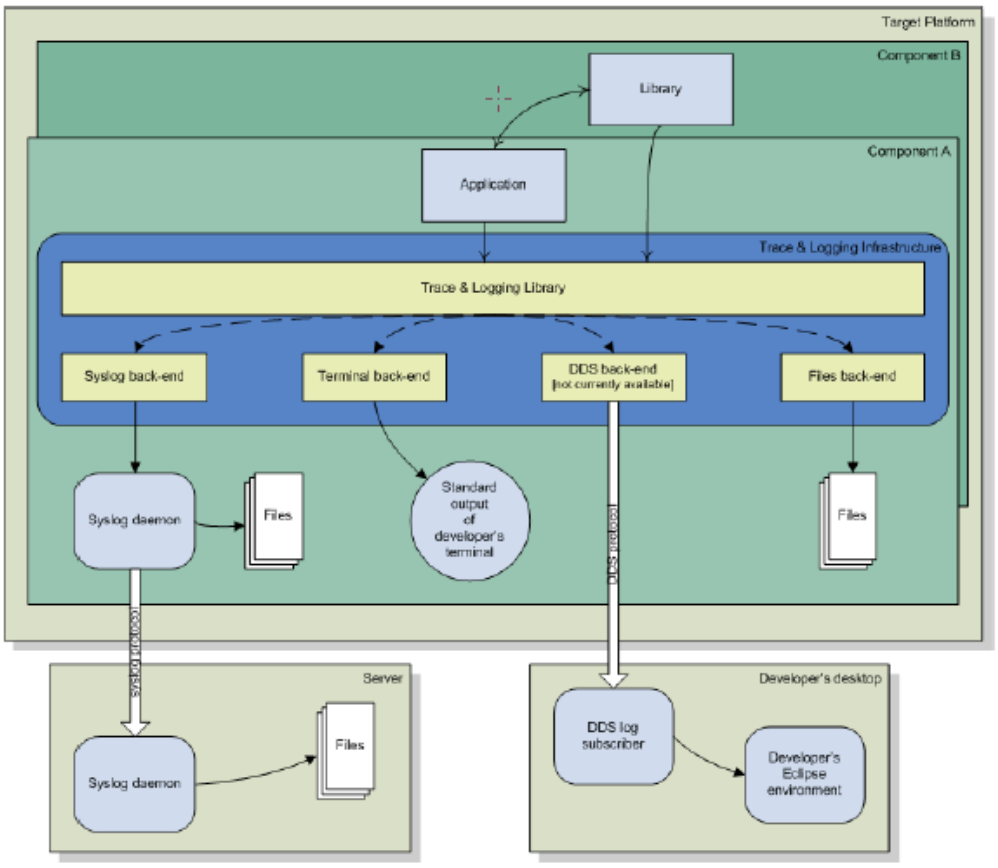


Figure 40 : Architecture de « Trace & Logging »

Enfin, cette architecture offre la possibilité d'intégrer de nouvelle sortie « back-end ».

IV.4 La conception de « Trace & Logging »

IV.4.1 Les niveaux de traces

Les niveaux de traces, avec leurs descriptions et leurs cas d'utilisation, sont répertoriés ci-dessous par ordre décroissant de gravité :

- Critical: Une erreur irrémédiable s'est produite et qui compromet la stabilité du middleware ou de l'application.
- Error: Echec d'exécution ou d'autres conditions inattendues
- Warning: Attirer l'attention sur un message avec un problème potentiel.
- Info: Une information intéressante au sujet d'un événement d'exécution normale.
- Debug : Un message ne servant qu'au débogage.

IV.4.2 L’espace de noms

L’espace de noms permet une différenciation entre les différentes applications. Ainsi il sera possible de faire un traitement distinct sur les noms, comme du filtrage ou du re-routage de message. Les niveaux de traces sont régits par des configurations sur l’espace de noms, ceci permet d’afficher uniquement la trace « debug » pour une application spécifique. Chaque application utilisant la librairie devra déclarer un nom lors de la compilation.

Un espace de noms secondaires peut éventuellement être spécifié. Cela permet plus de finesse pour les sous-modules d'un composant important. Par exemple, un composant fournissant un service réseau peut être subdivisé en plusieurs parties spécialisées: "réseau", "configuration", "stockage", "parser", etc. Avoir un espace de noms secondaires pour ces sous-éléments permet de les analyser séparément.

IV.4.3 La configuration « Trace & Logging »

La configuration de la librairie « Trace & Logging » dans une application se fait à l’initialisation. La librairie cherchera d'abord sa configuration dans les variables d'environnement du middleware. Dans le cas où aucune des variables d'environnement n’est trouvée, la librairie recherchera dans le fichier de configuration à l'adresse par défaut s’il n’a pas été spécifié. Enfin, si le fichier de configuration ne peut être trouvé, on utilisera des valeurs codées en dur. Ces variables permettent de configurer ces différents paramètres :

- Les back-end : Les messages peuvent être envoyés vers un ou plusieurs back-end
- Le filtrage sur les espaces de noms : permet d’afficher uniquement les espaces de noms choisis et inversement.
- Le format du message : permet de personnaliser les messages affichés comme, limiter la longueur du message affiché, afficher ou non le PID du processus, afficher le message d’une certaine couleur en fonction de son niveau de trace, etc.
- Le niveau de trace : permet de définir quel niveau de trace on souhaite afficher.

IV.5 En conclusion

Comme on a pu le voir, il existe de nombreuses solutions, qu'elles soient open sources ou propriétaires, pour corriger et améliorer le middleware « Revolution-S ». Le choix de l'open source permet d'avoir le support de la communauté pour corriger les bugs ou améliorer les fonctionnalités de l'outil. Mais il pose aussi la contrainte de la licence libre. Certains types de licence peuvent obliger à donner accès à tous son code source par effet de contamination. Il est donc préférable de partir d'un code déjà utilisé en interne comme « Trace & Logging » et le compléter avec un outil de profilage.

De tous les outils de profilage présentés, Ftrace et LTTng sont les plus couramment utilisés. Mon choix s'est porté sur le deuxième, parce qu'il existe un plugin à l'IDE Eclipse (utilisé par nos développeurs) qui permet de visualiser graphiquement le résultat contrairement à Ftrace. De plus, sa licence est LGPL V2.1, ce qui permet de l'utiliser en tant que librairie sans que cela nous oblige à rendre libre d'accès notre code source.

La question qui se pose : Est-ce que cette solution peut satisfaire les besoins des utilisateurs et du client ? Pour y répondre, je vais réaliser une analyse des pratiques au sein du projet « Revolution-S » ainsi qu'un recueil des nouveaux besoins fonctionnels internes et externes à Technicolor qui me permettront de confirmer ces choix et définir l'architecture cible du nouveau système de traces et de log.

PARTIE 3 : LES BESOINS

Afin de répondre à la problématique posée, qui est l'uniformisation du système de traces, je suis allé à la rencontre des personnes travaillant sur le projet « Revolution-S ». Je souhaitais connaître leurs pratiques pour déboguer le middleware et quelles utilisations ils faisaient de la librairie « Trace & Logging » que je compte réutiliser. Il me faut donc discerner leurs besoins et celui des clients pour que le nouveau système de traces réponde à leurs attentes. Grâce à cet échange, et en accord avec mon maître de stage, je pourrai définir la priorité de ces différents besoins qui m'aidera dans la partie réalisation et ce jusqu'à la conception du planning de développement.

I L'analyse des pratiques

L'objectif de cette analyse est de connaître les pratiques actuelles de traces et de débogage des différents intervenants sur le projet « Revolution-S ». Je suis allé à la rencontre d'une quarantaine de personnes afin de savoir quels étaient leurs fonctions et quels outils ils utilisaient pour corriger et améliorer le middleware.

I.1 Les métiers concernés

Sur le projet « Revolution-S », on peut distinguer trois types de métiers, les développeurs, les intégrateurs et les testeurs. Il est important de comprendre la fonction de ces postes car ils n'utilisent pas forcément les mêmes outils, les mêmes techniques et n'ont pas les mêmes attentes.

- Développeur

Les développeurs conçoivent et mettent à jour la roadmap du middleware « Revolution-S ». Ils sont répartis dans différentes équipes, chacune ayant en charge un ou plusieurs composants du middleware. Ils sont aussi garants des tests unitaires du composant qu'ils conçoivent et sont les principaux producteurs de traces.

- Intégrateur

Les intégrateurs ont pour rôle d'assembler les différents composants des équipes afin de fournir à intervalle régulier un middleware fini et stable aux testeurs. Ils interviennent rarement dans le code source et sont généralement consommateurs de traces.

- Testeur

Les testeurs sont les garants de la qualité du middleware. Ils détectent les anomalies liées aux fonctionnements techniques. Ils repèrent les erreurs de fabrication et sollicitent les développeurs pour une mise à jour. Comme les intégrateurs, ils interviennent rarement dans le code source et sont généralement consommateurs de traces.

Comme on peut le voir, ces différents métiers n'auront pas obligatoirement les mêmes attentes d'un outil de traces. Par la suite, je pourrai être amené à classer certains résultats ou besoins en fonction de ces métiers.

Sur les quarante personnes rencontrées, 24 étaient des développeurs, 12 des intégrateurs et 4 des testeurs.

I.2 Les techniques et outils utilisés

Sur les différentes techniques décrites dans la partie 2, la trace et la journalisation de celle-ci dans un fichier sont les plus utilisées. Le débogeur est très peu utilisé car il impacte trop le processus temps réel. Enfin le profilage est inexistant.

Le tableau ci-dessous résume les différents outils utilisés par les développeurs.

Tableau II : Tableau d'analyse des techniques et outils utilisés par les équipes projet.

Outils	Nombre d'utilisateurs	Commentaires
« Printf »	13	La plupart des développeurs utilisent le printf directement dans les sources du code lorsqu'ils souhaitent déboguer.
Gstreamer	7	Un des composants du middleware s'appuie sur le Framework multimédia « Gstreamer » et celui-ci intègre son propre système de traces.
Dbus moniteur	6	Dbus est un logiciel de communication interprocessus. Il offre une fonction de monitorage du bus.
GDB	4	Quelques développeurs utilisent GDB, le débogueur GNU.
Solution personnelle	1	Un développeur a conçu sa propre librairie de traces.
autres	1	Des outils comme, Valgrin, Gcov, Syslog, CoreDump, Mtrace, Strace, Smap, Pmap, Logrotate, m'ont été cités. Chacun étant utilisé par des personnes différentes et pour une utilisation occasionnelle.

Le premier constat que l'on peut faire est que les développeurs utilisent fréquemment le « printf » pour déboguer. On utilise souvent cette fonction pour afficher une valeur ou connaître le séquencement d'un programme. Il permet donc rapidement au développeur qui l'utilise, de mieux comprendre la logique mise en place dans le code source.

On pourrait imaginer que cette trace puisse resservir aux autres développeurs pour la même raison, mais les développeurs retirent les « printf » une fois la correction apportée. Un nombre trop important d'appels à cette fonction ralentirait le fonctionnement du décodeur à cause du temps pris à afficher. De plus le nombre de traces serait tel qu'il serait difficile pour un développeur de retrouver facilement l'information pertinente qu'il recherche. Il n'y a donc pas de capitalisation sur les traces de debug alors même que l'outil « Trace & Logging » répond à ce besoin.

Le deuxième constat est que des développeurs utilisent un système de trace spécifique à un Framework. La raison étant que les traces générées via « Trace & Logging » n'étaient synchrones qu'avec les événements générés par le Framework.

Enfin, le dernier constat, est le manque d'un outil de traces avec le noyau. Tous ces outils cités ont pour but d'informer le développeur du déroulement de son application dans l'espace noyau. Le choix de ces outils s'est fait selon la culture du développeur et pour un besoin occasionnel. Dans ce cas précis, l'outil « Trace & Logging » ne peut répondre à ce besoin.

Au vu de cette analyse, on remarque que malgré la disponibilité de l'outil « Trace & Logging », développé pour les aider, les développeurs utilisent quand même des solutions tierces hétéroclites. Je me suis donc intéressé à l'utilisation qu'ils faisaient de l'outil « Trace & Logging ».

II L'outil « Trace & Logging »

Une fois leurs pratiques connues, je me suis attaché à savoir s'ils connaissaient la librairie « Trace & Logging », comment ils utilisaient cette librairie et s'ils avaient des remarques sur cette solution.

II.1 Son utilisation

Si tous les développeurs connaissent cette librairie, un quart (6 sur 24) m'ont dit ne pas l'utiliser. La raison est qu'ils travaillent avec le Framework « Gstreamer » qui intègre son propre système de traces comme je l'ai expliqué plus haut.

Ils ne sont qu'un quart (3 sur 12) des intégrateurs à utiliser « Trace & Logging ». Et concernant les testeurs, aucun ne connaissaient l'outil. En réalité, les intégrateurs et les testeurs utilisent sans le savoir les traces générées par cette librairie. Mais ils ne connaissent pas les mécanismes de filtrage sur les noms ou les niveaux de traces proposés par l'outil.

Pour la génération de traces, les développeurs utilisent la librairie pour les messages de type « error », « warning » et « critical ». Seulement quelques un l'utilisent pour les messages « info » et « debug ».

Pour l'affichage des traces, la librairie offre différentes possibilités (choix du niveau de traces, choix du back-end, filtre, …), et j'ai voulu savoir comment les développeurs et intégrateurs la configurait. Le tableau ci-dessous résume donc l'utilisation faite des différentes fonctionnalités de l'outil « Trace & Logging » par les différents intervenants.

Tableau III : Tableau d'analyse de l'utilisation de la librairie « Trace & Logging ».

		Développeur	Intégrateur	Testeur
	nombre d'utilisateurs	21/26 (81%)	3/12 (25%)	0/4 (0%)
Niveau de traces générées	error	21 (100%)		
	critical	21 (100%)		
	warning	15 (71%)		
	info	5 (24%)		
	debug	3 (14%)		
Sélections des Back-end	sur console terminal	19 (90%)	2 (66%)	0 (0%)
	dans un fichier	15 (71%)	3 (100%)	0 (0%)
	vers le composant syslog	2 (10%)	0 (0%)	0 (0%)
	mode silence (aucune sortie)	1 (5%)	0 (0%)	0 (0%)
Configuration	du filtre sur l'espace des noms	2 (10%)	0 (0%)	0 (0%)
	du niveau de traces en sortie	10 (48%)	1 (33%)	0 (0%)
	du format des traces en sortie	3 (14%)	1 (33%)	0 (0%)

Ce que l'on remarque, c'est que les développeurs n'utilisent qu'une petite partie des possibilités offertes par cette librairie. La principale raison est qu'ils sont peu, à peine 20%, à avoir lu le document d'interface de « Trace & Logging ». Les autres n'avaient pas connaissance de celui-ci.

II.2 Le constat

J'ai souhaité connaître l'avis des utilisateurs sur les avantages et inconvénients de cette solution, ainsi que les possibles remarques des non utilisateurs après avoir lu le document d'interface.

Le principal inconvénient que m'ont remonté les utilisateurs est la configuration faite uniquement au démarrage de l'application et qui ne peut-être modifié pendant l'exécution. L'autre remarque porte sur le document d'interface, celui-ci n'étant pas clair, la configuration leur semblait alors complexe.

Il y a aussi des aspects positifs à cette librairie. La première remarque que l'on m'a faite porte sur les différents niveaux de traces possibles (error, warning, critical, …) pour la génération de message. La seconde est la possibilité du choix du back-end. Puis à la lecture du document, certains ont apprécié les possibilités de filtrage et le fait qu'en fonction du niveau de traces, le texte affiché sur la console puisse changer de couleur (en rouge quand c'est un « error »). De plus, le fait que cet outil est été développé en interne, permettant ainsi des améliorations, est un aspect qui a plu aux différents intervenants.

III Les nouveaux besoins

Pour permettre une uniformisation du système de traces, je me suis penché sur les nouveaux besoins des différents intervenants, y compris ceux des clients opérateurs. Puis j'ai classé, dans le tableau, les réponses en fonction des différents intervenants et si cela pouvait être une amélioration de l'outil « Trace & Logging » ou pas.

Tableau IV : Tableau des nouveaux besoins selon les métiers.

	Amélioration de « Trace & Logging »	Autres
Développeur	• Remonter l'information des traces activées et des filtres mis en place • Lier les traces avec le système de Gstreamer • Pouvoir afficher le fichier, la fonction et la ligne de code qui génère la trace • Créer un Constructeur / destructeur pour le langage C++ • Limiter la taille des fichiers gérés par le back-end « files ».	• Intégrer un outil de profilage
Intégrateur	• Possibilité d'avoir des notifications sur des traces spécifiques ou de s'abonner à un type/niveau de traces	
Testeur	• Gestion dynamique de l'activation des traces par le biais de l'interpréteur de commande *Shell*.	
Client	• Pouvoir transmettre les messages dans les journaux de façon synchrone ou asynchrone. • Gestion dynamique de l'activation des traces par le biais d'une interface web	

Suite à cette expression des besoins, il faut déterminer lesquels sont les plus importants.

IV La hiérarchisation des besoins

Un des risques identifiés au début du projet était la sous estimation des fonctionnalités à développer pouvant avoir un impact sur le planning prévisionnel. Comme on peut le constater, les besoins exprimés sont nombreux et une hiérarchisation de ceux-ci est nécessaire. En accord avec mon maître de stage, j'ai classé les différents besoins par ordre de priorité comme suit :

Priorité 1

- Intégrer un outil de profilage
- Remonter l'information des traces activées et des filtres mis en place
- Gestion dynamique de l'activation des traces par le biais de l'interpréteur de commande *Shell*.
- Limiter la taille des fichiers gérés par le back-end « files ».
- Afficher les identifiants de Processus et de Thread

Priorité 2

- Lier les traces avec le système de Gstreamer
- Lier les traces avec l'outil de profilage
- Possibilité d'avoir des notifications sur des traces spécifiques ou de s'abonner à un type/niveau de traces

Priorité 3

- Améliorer la granularité dans chaque niveau de traces.
- Pouvoir afficher le fichier, la fonction et la ligne de code qui génère la trace
- Créer un Constructeur / Destructeur pour le langage C++

Priorité 4

- Pouvoir transmettre les messages dans les journaux de façon synchrone ou asynchrone.

Priorité 5

- Gestion dynamique de l'activation des traces par le biais d'une interface web

Cette étude de besoins a permis de mettre en lumière le manque d'uniformité dans les outils de traces. Et ce malgré la présence d'un outil, spécialement développé en interne, pour tracer le comportement du middleware. Il y a pour moi deux raisons à l'échec d'uniformisation de celui-ci.

La première est l'absence de communication sur l'outil. Les personnes qui l'utilisent sont principalement ceux qui l'ont développé. Et ils sont les seuls à avoir connaissance du document d'interface.

La deuxième raison, est que la conception de l'outil a été limitée aux besoins des développeurs ayant participé à sa réalisation. Le fait que les intégrateurs et les testeurs n'aient pas connaissance de l'outil, montre qu'ils n'ont pas été impliqués dans sa conception.

Mais cet outil a aussi ses avantages. Le premier est la distinction entre la génération des traces et la consommation de celles-ci par le choix de différents back-end. Il est ainsi facile d'implémenter de nouveaux back-end sans retoucher à toute la librairie. Le deuxième est que cette librairie a été développée en interne, les risques liés à la contamination par une licence libre sont donc écartés. Enfin, il est implémenté dans tous les composants du middleware. Il existe donc un historique non négligeable dont il faudra tenir compte.

C'est pour ces raisons que je vais repartir de cette librairie pour définir l'architecture cible dans laquelle je vais intégrer l'outil de profilage LTTng et que je vais présenter dans la prochaine partie.

PARTIE 4 : LA REALISATION

Dans cette partie, je vais présenter l'architecture cible qui répondra aux besoins exprimés. Grâce à celle-ci et à la hiérarchisation des besoins, j'élaborerai un nouveau planning de développement afin de gérer le risque de sous-estimation de l'effort identifié préalablement. Puis je présenterai la réalisation de mes différents développements.

V Présentation de l'architecture cible

Comme je l'ai présenté dans la partie étude, le middleware « Revolution-S » intègre déjà une solution de traces. Mais le manque de communication sur cet outil et une utilisation qui ne se limite qu'aux développeurs fait qu'il est peu utilisé. L'analyse du besoin a montré qu'une évolution de cet outil pouvait convenir. Cela permettrait ainsi de conserver l'historique des traces générées par cet outil. Je vais donc dans un premier temps présenter l'architecture et les aspects fondamentaux de cet outil pour ensuite expliquer l'architecture cible que je vais mettre en place afin de répondre aux différents besoins.

V.1 Architecture actuelle

Pour comprendre le fonctionnement de l'outil « Trace & Logging », j'ai du faire du reverse engineering sur le code source car je n'avais que le document d'interface qui expliquait comme utiliser l'outil. Je vais donc tout d'abord expliquer comment l'outil s'utilise pour après décrire son fonctionnement interne.

V.1.1 Son utilisation

Le document d'interface décrit les deux principes suivants. La génération de traces et la configuration de l'outil pour les afficher.

L'insertion de traces est assez simple à mettre en œuvre pour le développeur. Dans son code source, le développeur doit inclure la librairie *<log/log.h>* et faire un *#define* des variables *LOG_NAME* et *LOG_SUB_NAME*. Puis dans la fonction *main(),* il doit insérer l'appel à *log_init()* au début et *log_clean()* à la fin.

Ensuite, il lui suffit d'instrumenter son code, et selon le degré de gravité de traces à transmettre, d'appeler les fonctions suivantes.

- *log_error(...)*
- *log_critical(...)*

- *log_warning(...)*
- *log_info(...)*
- *log_debug(...)*

Ces fonctions utilisent le même principe d'implémentation que le *printf()*.

Une fois le code instrumenté, le développeur peut selon sa convenance choisir une ou plusieurs solutions d'affichage des traces, mais aussi le formatage et le niveau de traces qu'il souhaite voir apparaître. Il doit pour cela définir plusieurs variables d'environnement dans un fichier de configuration. Il en existe un par défaut qui est situé dans */etc/log.conf*.

Les principales variables à définir sont : *LOG_OUTPUT* pour le choix des backends, *LOG_FORMAT* pour le formatage des données et *LOG_MASK_PRIO* pour le niveau de traces à afficher.

V.1.2 Son fonctionnement

Pour comprendre le fonctionnement de cette librairie, il a fallu que je procède à du reverse engineering sur le code source. Car le seul aspect d'architecture que j'ai pu trouver dans le document d'interface est cette figure.

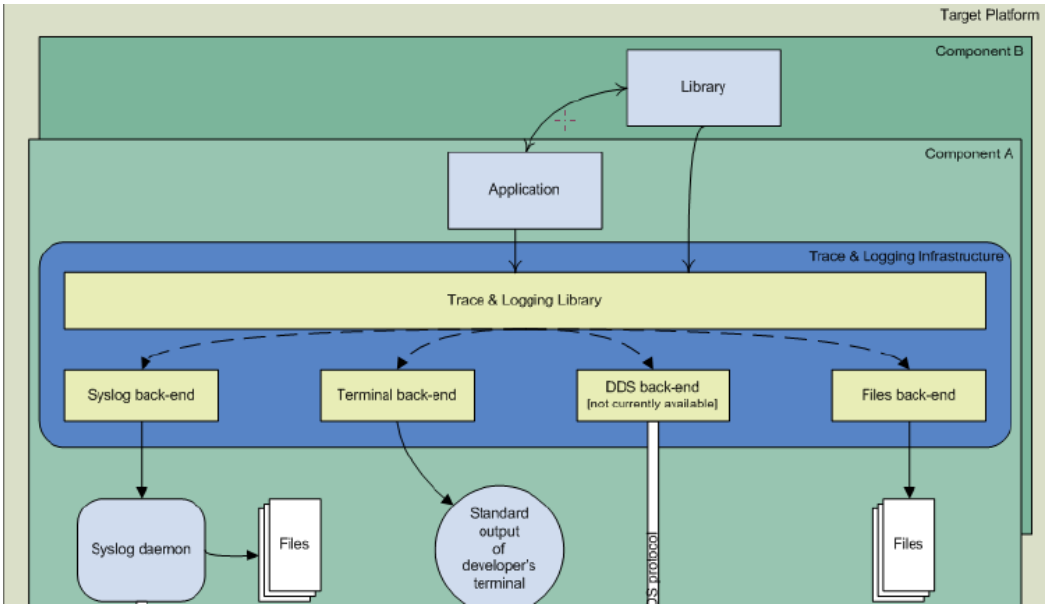


Figure 41 : Vue globale de « Trace & Logging »

La première chose que j'ai analysée est le *makefile* qui compile le code source, il montre que l'outil « Trace & Logging » se compose de cinq librairies. Liblog.so est la

librairie principale et liblog_backend_files.so, liblog_backend_silent.so, liblog_backend _syslog.so , liblog_backend_term.so, sont spécifiques pour chaque back-end.

Ce sont des librairies dynamiques qui sont chargées dans la zone mémoire du processus qui les utilise. Les données que ces librairies manipulent sont donc spécifiques au processus dans lequel elles s'exécutent.

J'ai ensuite analysé le code source. J'ai créé une représentation en une vue statique des principaux éléments qui composent l'outil « Trace & Logging ».

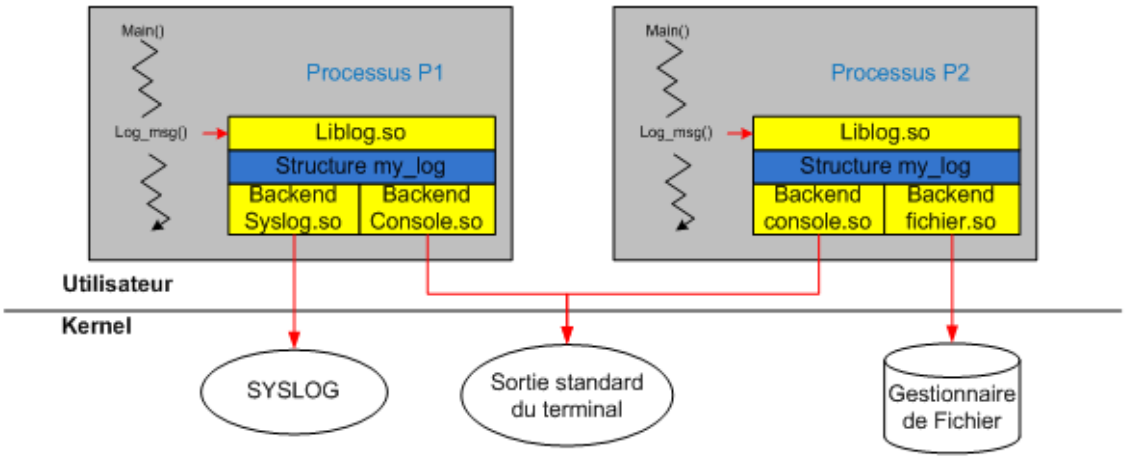


Figure 42 : Vue statique sur l'implémentation de « Trace & Logging »

L'élément important de l'outil est la structure de données *my_log*. Celle-ci est créée par la librairie *liblog.so* et contient toutes les variables d'environnement que le développeur a spécifié dans le fichier de configuration et son code source. Elle sert aussi de passerelle entre la librairie principale et les différents back-end. Chaque processus utilisant « Trace & Logging » aura sa propre structure indépendante des autres.

Le constat que l'on peut faire est qu'après l'installation de la librairie par le processus, il n'y a aucun moyen de venir vérifier sa configuration, ni même de modifier celle-ci.

Pour le confirmer, j'ai réalisé des diagrammes de séquencement lors de l'initialisation de la librairie et pour l'envoi de traces à travers celle-ci.

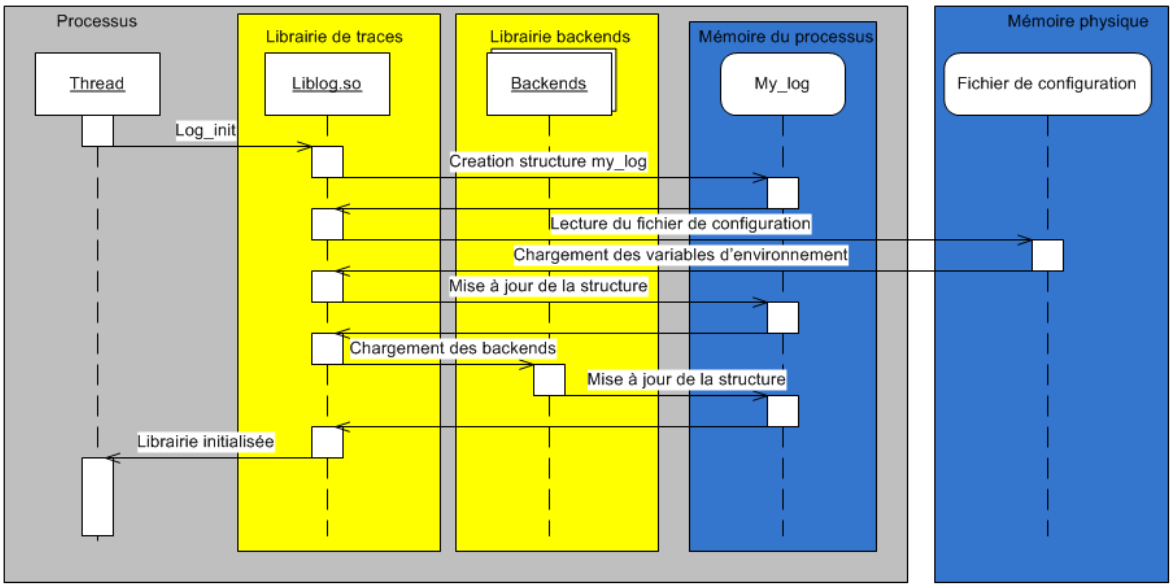


Figure 43 : Diagramme de séquencement de l'initialisation de « Trace & Logging »

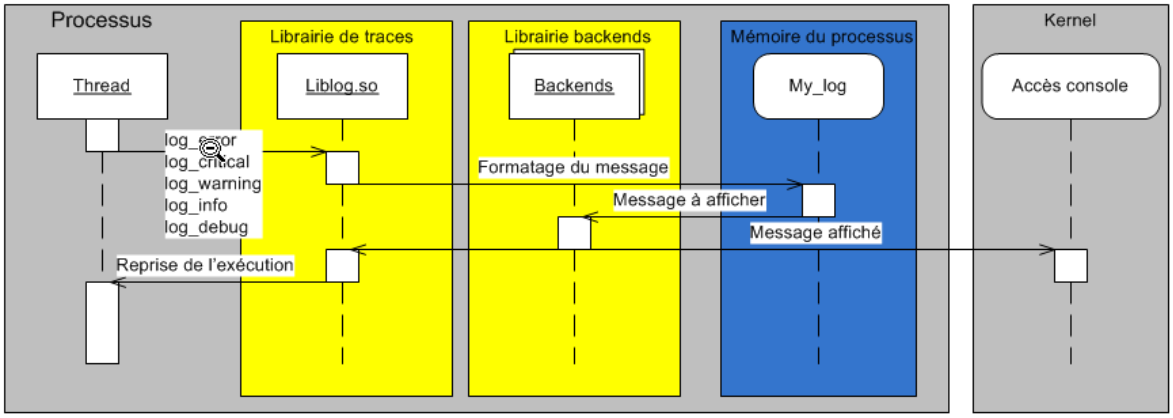


Figure 44 : Diagramme de séquencement pour l'envoi de traces

Comme le montre ces diagrammes, la structure *my_log* est configurée uniquement à l'initialisation. On remarque aussi quel est le lien entre la librairie *liblog.so* et les librairies back-end pour l'envoi d'une trace. Il y a deux raisons à cela.

La première est que *my_log* sert à stocker les différents paramètres d'environnement (filtrage, niveau de traces, back-end charger en mémoire, …) qui serviront au formatage et l'envoi de la trace.

La deuxième raison est qu'elle sert aussi de pointeur de fonction vers la librairie du back-end. Les pointeurs de fonction sont comme des pointeurs de variables, mais qui pointent sur l'adresse d'une fonction(17). Pour ne pas surcharger le processus en intégrant toutes les librairies back-end dans *liblog.so*, celle-ci charge dynamiquement les back-end selon la configuration définie par le développeur. Puis stocke dans la structure *my_log* les

adresses mémoires des différentes fonctions qu'incluent une librairie back-end. La figure ci-dessous représente la structure *my_log* dans son ensemble.

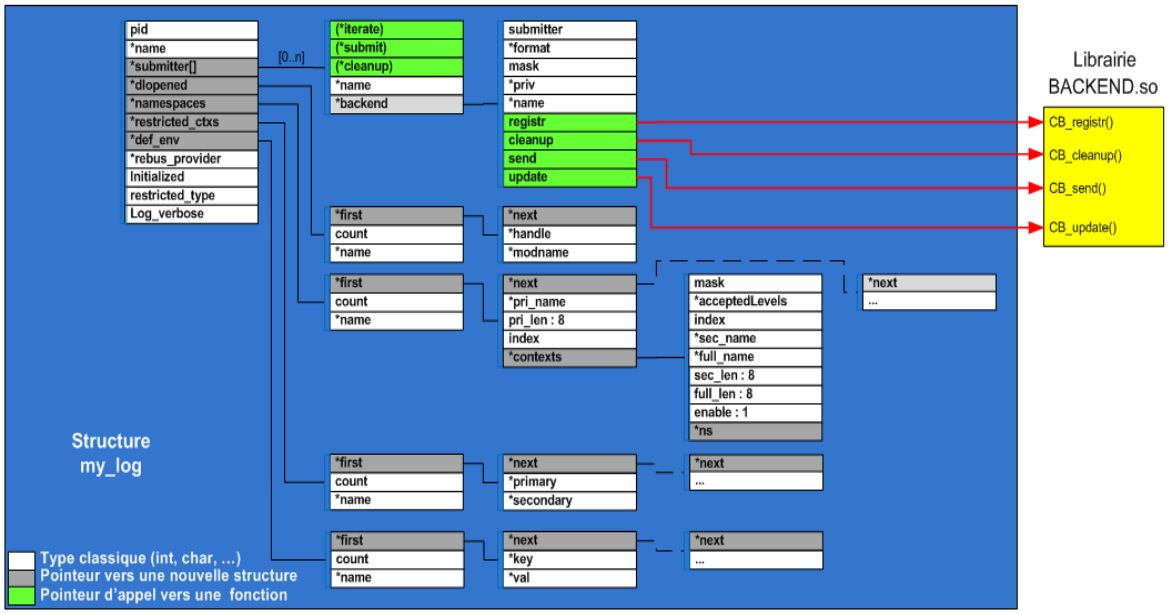


Figure 45 : vue globale de la structure *my_log*

Le reverse engineering que j'ai réalisé sur « Trace & Logging » m'a permis de mieux comprendre son fonctionnement, mais aussi ses limites et les remarques faites sur son utilisation. Ces différents diagrammes vont donc me permettre de présenter l'évolution de « Trace & Logging » pour répondre aux besoins exprimés.

V.2 Architecture cible

Avant de présenter l'utilisation et le fonctionnement de l'architecture cible, je vais décrire la réponse que je vais apporter à chaque besoin identifié.

V.2.1 Les réponses aux différents besoins

Le tableau ci-après reprend la liste des besoins exprimés et comment je vais y répondre.

Tableau V : Tableau des réponses aux différents besoins.

Priorité	Besoins	Réponses
1	Intégrer un outil de profilage	L'outil choisit sera LTTng. La raison étant la possibilité de pouvoir le gérer à travers Eclipse.
1	Remonter l'information des traces activées et des filtres mis en place	Développement d'une commande accessible par une interface *Shell*. Celle-ci permettra d'afficher les informations demandées. Mais aussi de gérer dynamiquement l'activation/désactivation des traces par processus.
1	Gestion dynamique de l'activation des traces par le biais de l'interpréteur de commande *Shell*.	
1	Limiter la taille des fichiers gérés par le back-end « files ».	Amélioration de la librairie *liblog.so*
1	Afficher les processus id et Thread id	Amélioration de la librairie *liblog.so*
2	Lier les traces avec le système de Gstreamer	Développement d'un nouveau back-end qui puisse transmettre les traces au système Gsreamer
2	Lier les traces avec l'outil de profilage	Développement d'un nouveau back-end qui puisse transmettre les traces au système LTTng
2	Possibilité d'avoir des notifications sur des traces spécifiques ou de s'abonner à un type/niveau de traces	Développement d'un nouveau back-end qui permette de s'abonner ou d'être notifié.
3	Améliorer la granularité dans chaque niveau de traces.	Amélioration de la librairie *liblog.so*
3	Pouvoir afficher le fichier, la fonction et la ligne de code qui génère la trace	Amélioration de la librairie *liblog.so*
3	Créer un Constructeur / Destructeur pour le langage C++	Amélioration de la librairie *liblog.so*
4	Pouvoir transmettre les messages dans les journaux de façon synchrone ou asynchrone.	Amélioration de la librairie *liblog.so*
5	Gestion dynamique de l'activation des traces par le biais d'une interface web	Développement d'une commande qui puisse générer une page web et qui puisse faire la même chose que la commande accessible par une interface *Shell*.

La plupart des besoins portent sur l'amélioration même de la librairie *liblog.*so. Je ne représenterai pas ces améliorations dans l'architecture cible car elles sont trop bas niveau par rapport à la vue globale présentée.

Trois besoins nécessitent le développement de nouveaux back-end, dont un pour s'interfacer avec l'outil de profilage LTTng que je devrai intégrer.

Enfin pour la gestion dynamique je devrai développer une interface accessible par l'interface de commande *Shell* mais aussi par le biais d'une page web. Je vais commencer par présenter l'utilisation de cette interface avant de parler du fonctionnement global de la nouvelle architecture..

V.2.2 Son utilisation

L'insertion de traces qui était assez simple à mettre en œuvre reste inchangée dans la nouvelle architecture. Mais une nouvelle commande pour la gestion dynamique est accessible par une interface de console *Shell*. Cette commande s'appelle *liblog_console* dont voici quelques exemples d'utilisation.

Liblog_console –h ou –help : affiche l'aide d'utilisation de la commande

Liblog_console –s ou –status: affiche la configuration de traces de tous les processus ayant la librairie liblog d'initialisé.

Liblog_console –pid num_pid –on/off: permet d'activer ou de désactiver les traces d'un processus en indiquant son numéro de PID.

Liblog_console –web : lance l'interface web.

L'ensemble des commandes seront détaillées dans les chapîtres suivants.

V.2.3 Son fonctionnement

Afin de mettre en place cette commande, il faut que je sache quels processus utilisent « Trace & Logging » et qu'ils m'informent sur leurs configurations. Pour cela, je vais mettre en place un *daemon* (un processus qui est lancé au démarrage en tâche de fond) qui sera chargé de récupérer la configuration de tous les processus qui utiliseront *liblog.so*. La commande viendra alors s'appuyer sur ce processus *daemon*, pour afficher les informations demandées par l'utilisateur, et ainsi pouvoir modifier dynamiquement la

configuration de traces d'un processus. La figure ci-dessous représente la vue statique de l'interaction entre les différents éléments de l'outil « Trace & Logging ».

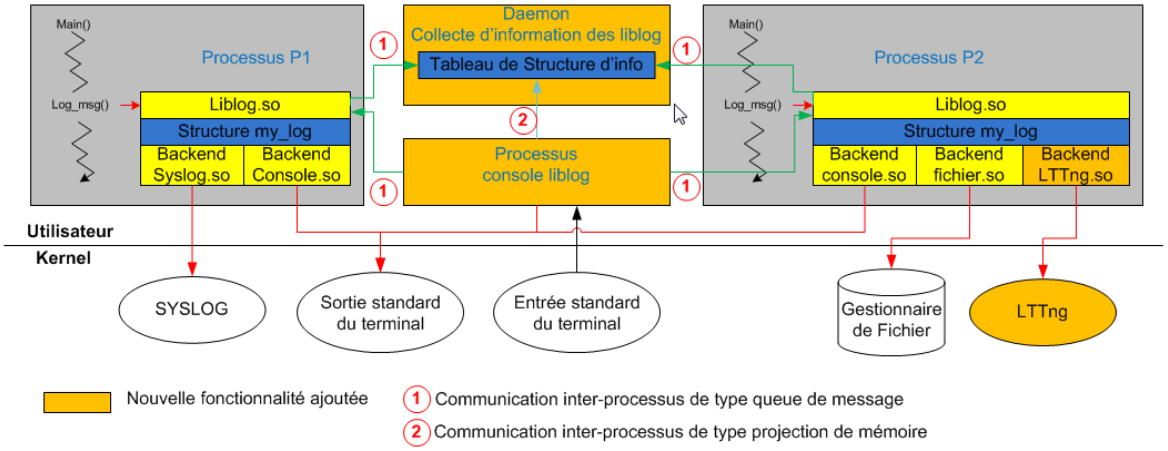


Figure 46 : Vue statique sur l'implémentation cible de « Trace & Logging »

Pour que ces processus puissent communiquer entre eux, je dois utiliser un des moyens de communication que j'ai présenté dans la partie 2 chapitre I.4.2.

J'ai choisi de communiquer par queue de message entre mes développements et les différents processus intégrant liblog.so. Ce moyen est de type asynchrone, c'est-à-dire que celui qui envoie le message n'attend pas de savoir si le destinataire l'a reçu pour continuer son exécution. On peut comparer cette communication à de la correspondance par courrier, le *noyau* jouant alors le rôle de facteur.

Mais entre mon *daemon* et ma commande de console, j'ai choisi d'utiliser la projection mémoire comme moyen de communication. Ceci évite d'alourdir le système par des appels systèmes et empêche une recopie de l'information dans le processus de commande. La zone mémoire projetée sera en lecture seule, ainsi seul le daemon pourra la modifier. De plus un sémaphore sera mis en place pour éviter que le processus de commande ne vienne lire des informations que le *daemon* est en train de modifier.

J'ai aussi représenté dans cette vue statique un nouveau back-end qui est LTTng, Je n'ai pas représenté les autres pour des raisons de lisibilité, mais le principe serait le même, c'est-à-dire la nouvelle librairie back-end viendrait s'intégrer dans le processus puis enverrait la trace vers la sortie définie.

Les figures suivantes sont les diagrammes de séquencement mis à jour pour l'architecture cible. Ils représentent ici un exemple d'utilisation qui commence par l'initialisation de l'outil.

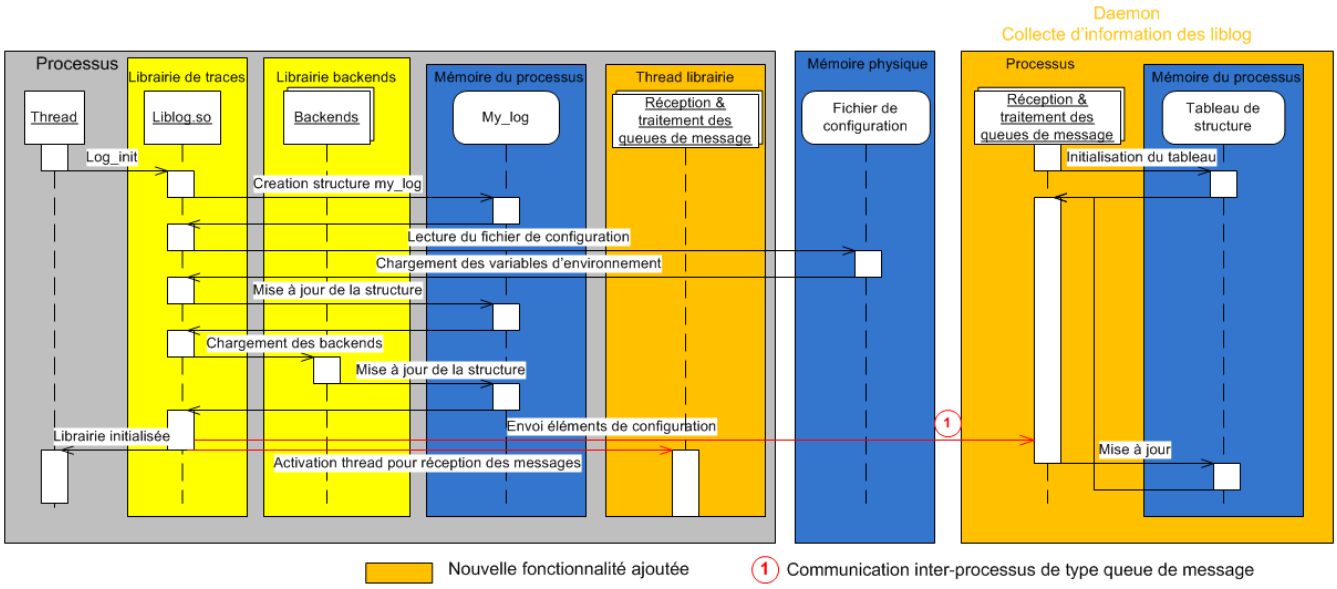


Figure 47 : Diagramme de séquencement de l'initialisation cible de « Trace & Logging »

Sur ce diagramme, on peut voir qu'à la fin de l'initialisation de *liblog.so*, on envoie par queue de message la configuration de la librairie. Le *daemon* récupère ce message et met à jour son tableau de structure. *Liblog.so* lance aussi un nouveau *thread* qui est chargé de réceptionner les messages envoyés par l'utilisateur afin de mettre à jour la structure *my_log*.

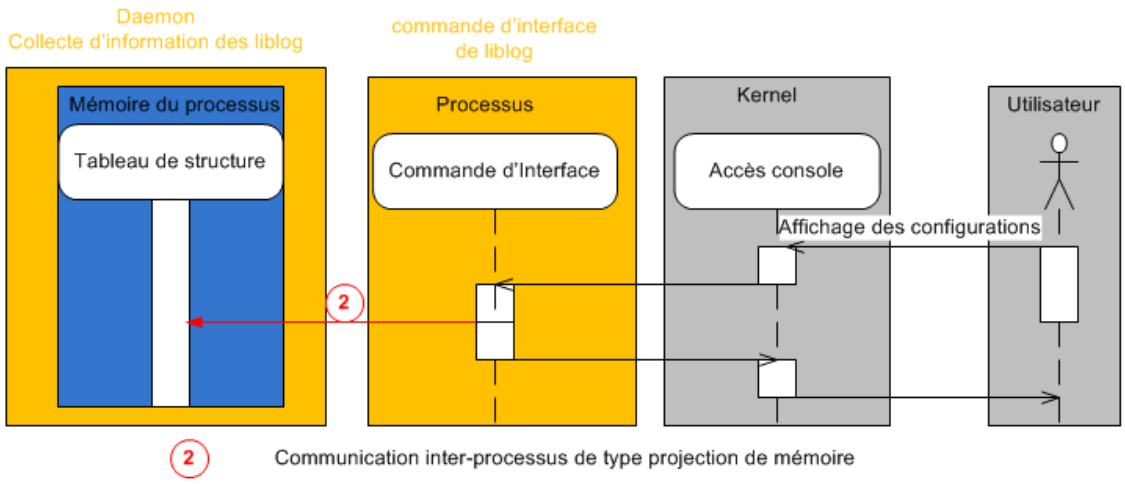


Figure 48 : Diagramme de séquencement pour l'affichage des configurations des traces

Lorsque l'utilisateur veut connaître la configuration des différentes *liblog.so* initialisées, il exécute, à travers la console d'interface, la commande *Liblog_console –s*. Celle-ci va s'appuyer sur le tableau de structure du *daemon* pour récupérer les informations à afficher.

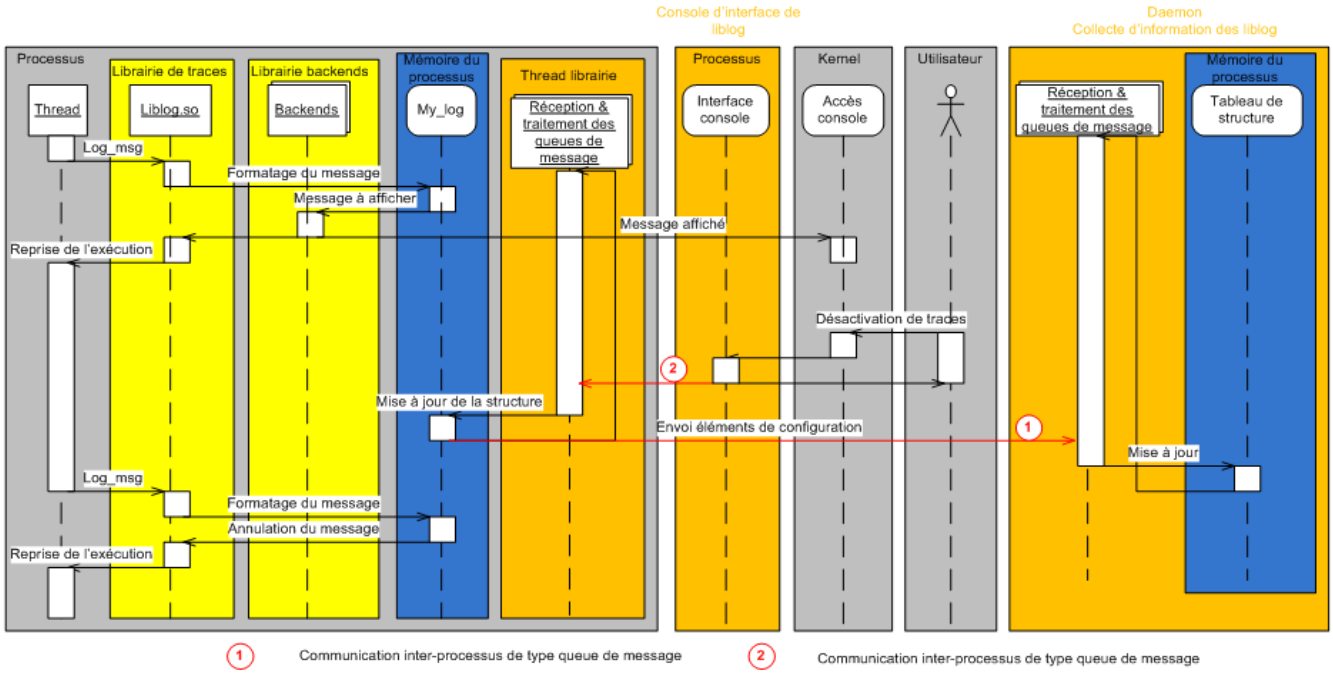


Figure 49 : Diagramme de séquencement pour la désactivation de traces d'un processus

Grâce à ces informations, l'utilisateur peut décider de désactiver les traces d'un processus. Il exécute alors la commande *Liblog_console –pid num_pid –off*. Celle-ci envoie par queue de message l'ordre de désactiver les traces du processus. Le *Thread* récupère cette information et met à jour la structure *my_log*. Les traces du processus ne seront alors plus affichées.

Grâce à la description d'utilisation et à travers ces différents diagrammes décrivant le fonctionnement, je vais pouvoir déterminer les étapes et tâches à réaliser pour définir le planning prévisionnel de développement.

VI Planning prévisionnel de développement

Ce nouveau planning de développement a pour but d'estimer l'effort qu'il faudra pour répondre aux différents besoins. Il me servira aussi pour le suivi hebdomadaire de l'avancement du développement comme je l'ai identifié dans la gestion des risques.

VI.1 Les étapes du développement

Pour réaliser le planning, je vais découper le développement en tâches à réaliser. Puis réaliser l'enchaînement logique de celles-ci selon leurs dépendances et estimer l'effort pour chacune d'entre elles.

VI.1.1 Découpe en tâches et estimation de l'effort

Après avoir réalisé le tableau qui résumait les réponses aux différents besoins, j'ai distingué trois grandes catégories, les améliorations à apporter à la librairie *liblog*.so, les nouveaux back-end à développer et l'interface de commande pour piloter dynamiquement l'activation des traces. Je vais pour chaque catégorie décrire les tâches en leur affectant une lettre et l'effort estimé.

Tableau VI : Tableau des tâches à réaliser

	Tâche	Effort jour	Tâche antérieur	Priorité
Amélioration de *liblog.so*				
A	Limiter la taille des fichiers générés par le back-end « files »	3/5j		1
B	Afficher les numéros PID et thread id qui génère la trace	2/3j		1
C	Afficher le nom du fichier, la fonction et le numéro de ligne qui génère la trace	2/3j		3
D	Ajouter granularité pour chaque niveau de traces	3/4j		3
E	Développer un constructeur/destructeur en C++	3/5j		3
F	Transmettre les traces de façon synchrone/asynchrone	5/7j		4
Intégration de nouveaux back-end				
G	Intégrer de l'outil LTTng	5/7j		1
H	Développer un back-end pour LTTng	3/4j	G	2
I	Développer un back-end pour Gstreamer	5/7j		2
J	Développer un back-end pour abonnement/notification.	5/7j		2
Interface de commande				
K	Envoi depuis *liblog.so* de queue de message vers *daemon*	2/3j		1
L	Réception de queue de message depuis un *daemon*.	2/3j	K	1
M	Création depuis le daemon d'une zone mémoire partagée	2/3j	L	1
N	Affichage information zone mémoire depuis commande en ligne.	2/3j	M	1
O	Affichage de l'aide depuis commande en ligne	2/3j	N	1
P	Envoi depuis commande en ligne de queue de message vers processus	2/3j	O	1
Q	Réception et traitement queue de message dans *liblog.so*	2/3j	P	1
R	Ajout d'une interface web	5/8j	O	5

VI.1.2 Séquencement des tâches

La figure suivant représente l'enchainement logique des tâches.

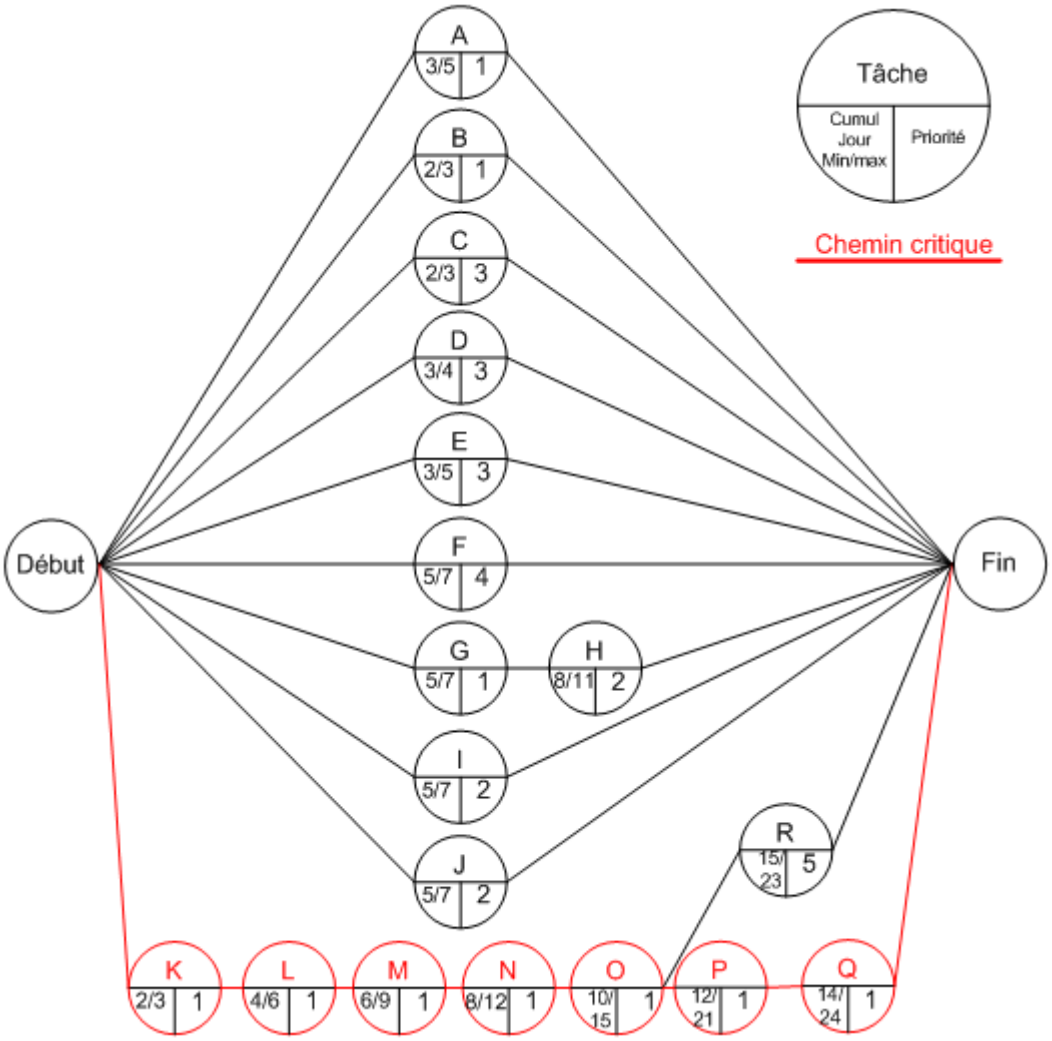


Figure 50 : Diagramme de séquencement des tâches à réaliser

Le diagramme de séquencement montre que le développement de la commande d'interface dynamique, de par sa durée et sa priorité, est sur le chemin critique. Il sera donc traité en premier. L'intégration de l'outil LTTng et le développement de son back-end seront ensuite réalisés, car les autres tâches sont indépendantes les unes des autres. En effet leut développement est de plus courte durée ou de plus faible priorité. Elles seront ensuite prises par ordre de priorité, comme le montre le planning de développement.

VI.2 Planning de développement

Ce planning a deux objectifs, le premier est de représenter la durée que va prendre ce développement et vérifier si les étapes définies pour le planning prévisionnel sont toujours d'actualité. Le deuxième est de mettre en place un suivi hebdomadaire du développement à réaliser.

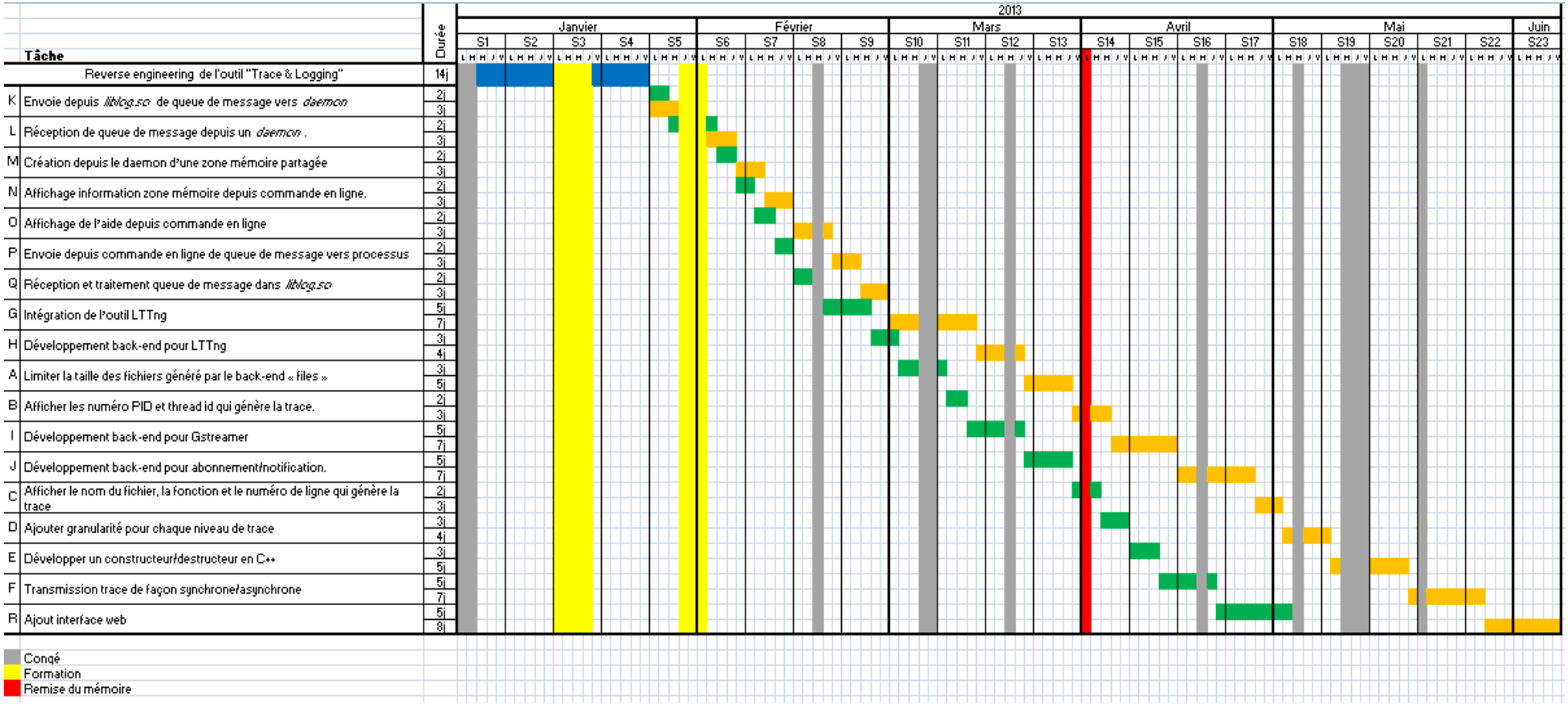


Figure 51 : Planning de développement

Par rapport au planning prévisionnel, je n'avais pas identifié le reverse engineering à pratiquer sur l'outil « Trace & Logging », ainsi qu'une deuxième formation. Ceci a eu pour impact de décaler le début du développement à fin janvier.

Sur la durée de développement, j'avais planifié neuf semaines. Or le développement durera dans le meilleur des cas treize semaines et dans le pire dix-neuf. Ceci est dû à la sous-estimation du besoin au début du projet, mais ce risque avait été identifié et sa gestion détaillée au prochain paragraphe.

Pour la partie test et validation, un test unitaire simple sera réalisé pour chacune des tâches développées et un test de validation globale sera réalisé avec un décodeur intégrant le middleware « revolution-S ». La durée reste estimée à trois semaines après la fin du développement.

VI.3 Gestion du risque.

Un des risques que j'avais identifié au début du projet était la sous-estimation de l'effort de développement. Suite à l'analyse du besoin, je devais réaliser un nouveau planning de développement pour quantifier ce risque et ainsi mettre les actions correctives en place.

Le premier constat est que le développement de l'ensemble des fonctionnalités dépassera la date de dépôt de ce mémoire, or celle-ci devait marquer la fin de la réalisation de ce projet. La première action sera de définir avec mon manager les suites à donner au projet après cette date et la conclusion de ce mémoire pourra aider à la décision. L'objectif fixé par mon maître de stage est de finir la réalisation de l'interface de commande et l'intégration du back-end LTTng avant cette date.

La deuxième action est de faire un suivi hebdomadaire des tâches réalisées durant la phase de développement. L'objectif est de détecter si le cadre que je me suis donné en termes d'effort n'est plus tenu et alors alerter d'un nouveau retard sur le planning. Pour chacune des réalisations que je présenterai, j'indiquerai le temps de développement effectué et s'il a fallu alerter.

VII Mes réalisations

VII.1 La console d'interface dynamique

Ma première réalisation a été de développer la console d'interface qui permet de piloter dynamiquement la configuration des traces. Les objectifs sont :

- Afficher la configuration des processus ayant initialisé la librairie *liblog*.so.
- Activer ou Désactiver dynamiquement les traces d'un processus.
- Changer le niveau de traces d'un des processus de façon dynamique.
- Mettre en place des filtres sur les noms.

Comme je l'ai expliqué dans le paragraphe V.2.3, il m'a fallu adapter l'architecture de l'outil « Traces & Logging » pour mettre en place un daemon qui collecte les informations des différents processus utilisant *liblog*. Mais aussi développer une interface de commande pour modifier dynamiquement la configuration des traces.

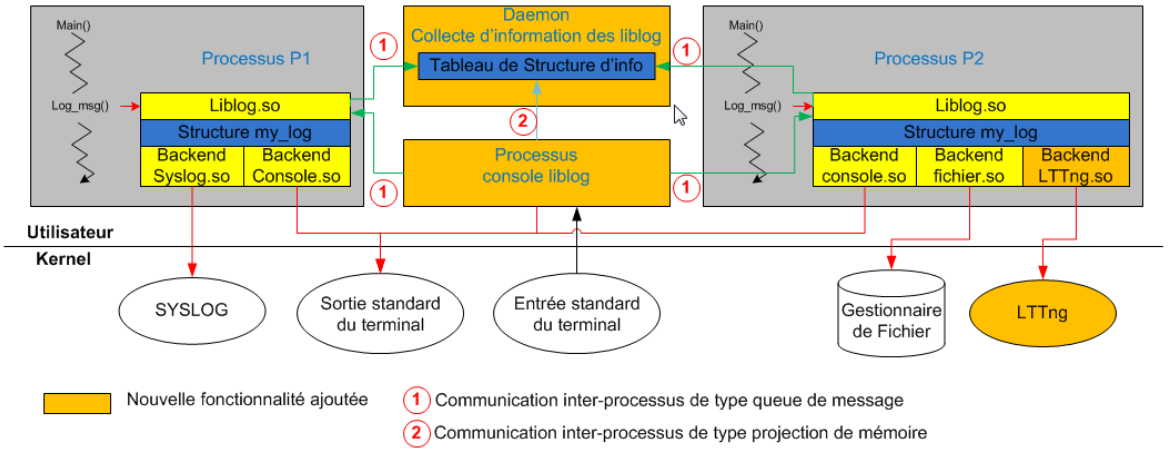


Figure 52 : Vue statique sur l'implémentation cible de « Traces & Logging »

Je vais donc présenter cette réalisation en cinq étapes.

VII.1.1 L'envoi d'informations

VII.1.1.1 Présentation

La première étape a été de définir les informations que les instances des librairies *liblog* initialisées devaient envoyer au *daemon*. Celles-ci doivent permettre aux utilisateurs de connaître les configurations qu'ils peuvent changer. Ce sont les PID, le nom des

processus, les back-end utilisés, les niveaux de traces appliqués, les filtres mis en place et si les traces sont activées ou pas.

La deuxième étape a été de définir le formalisme des trames à envoyer. Celles-ci serviront à remplir et mettre à jour un tableau dans le *daemon*.

Le premier paramètre peut prendre trois valeurs et a pour objectif d'informer le daemon si c'est une initialisation, une mise à jour des informations ou la fin d'utilisation de la librairie. En cas d'initialisation ou de fin d'utilisation, seul le PID est transmis, sinon les informations décrites dans l'étape une sont envoyées.

Voici donc le formalisme d'une trame émise pour une mise à jour :

Info_trame **:** *PID du processus* **:** *trace on/off* **:** *nom du processus* **:** *nom backend 1* **:** *mask backend 1* **:** *...* **:** *nom backend 8* **:** *mask backend 8* **:** *configuration du filtre* **:** *noms filtrés*

La dernière étape a été d'initialiser l'interface de communication interprocessus entre les librairies et le daemon. Elle sera de type queue de message car les données à transférer ne sont pas volumineuses et ne nécessite pas d'avoir un acquittement.

VII.1.1.2 Développement

Pour chaque présentation de développement, je vais mettre en parallèle du code source, un diagramme d'activité ou de séquencement pour clarifier sa compréhension.

Voici le diagramme d'activité et le code source qui consiste à la mise en forme du message à envoyer au *daemon* de la console.

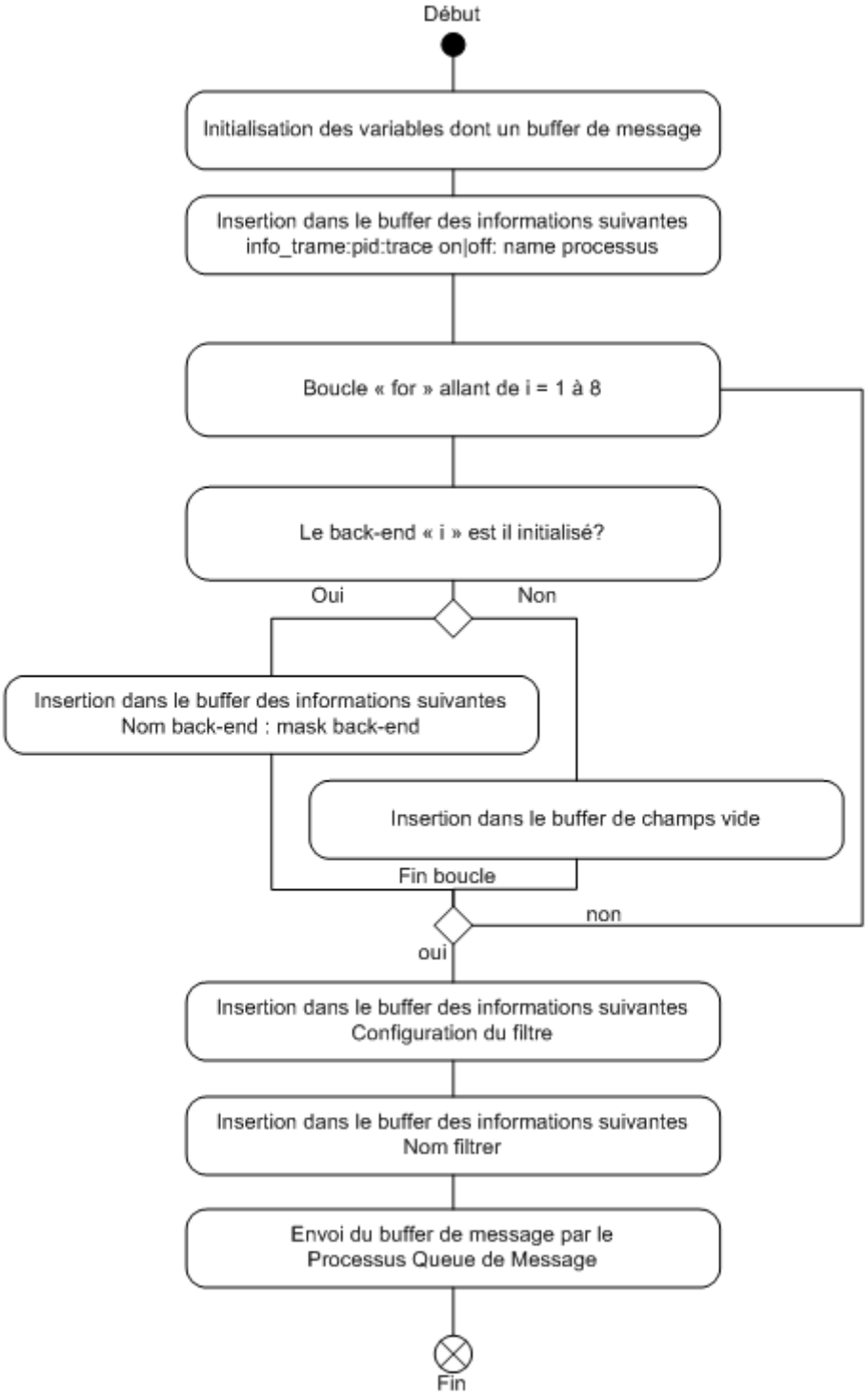


Figure 53 : Diagramme d'activité pour l'envoi d'informations

```c
/**
 * send info structure to daemon of console interface
 */
void structure_info_send(pid_t pid)
{
        /* Local variable*/
        struct log_restricted_namespaces {
                char *primary;
                char *secondary;
        };
        struct log_node *node;

        char message[1500];
        char message_tampon [128];
        int i;

        /* insertion in message buffer "info_trame:pid:trace on|off: name processus*/
        sprintf(message, "2:%d:%d:%s", pid, my_log->p_on_off, my_log->name);

        /* insertion in message buffer "name : mask" of each backend*/
        for(i=0; i < NUMBER_OF_BACKEND; i++)
        {
                if( (i < LOG_SUBMIT_MAX) && my_log->submitter[i])
                {
                        struct log_backend *info_backend = my_log->submitter[i]->backends->first->elem;
                        sprintf(message_tampon, ":%s:%d", info_backend->name, info_backend->mask);
                        strncat(message, message_tampon, sizeof(message_tampon));
                }
                else{
                        strcat(message, "::");
                }
        }

        /* insertion in message buffer "filtre configuration"*/
        sprintf(message_tampon, ":%d:", my_log->restricted_type);
        strncat(message, message_tampon, sizeof(message_tampon));

        /* insertion in message buffer "name filter"*/
        node = my_log->restricted_ctxs->first;
        while( node ) {
                struct log_restricted_namespaces *filtre_elem = node->elem;
                if (filtre_elem->secondary != NULL)
                {
                        sprintf(message_tampon, "%s,%s ", filtre_elem->primary, filtre_elem->secondary);
                        strncat(message, message_tampon, sizeof(message_tampon));
                }else{
                        sprintf(message_tampon, "%s ", filtre_elem->primary);
                        strncat(message, message_tampon, sizeof(message_tampon));
                }
                node = node->next;
        }

        /* send message buffer to daemon of console throught Message Queue process*/
        MQD_lib_send(message);
}
```

Figure 54 : code source de l'envoi d'informations

VII.1.1.3 Test unitaire

Pour chaque test unitaire, j'utiliserai un programme basic de type « hello world » qui utilisera la librairie *liblog* pour générer les traces.

Pour ce test, j'ai développé un soft qui récupère les informations envoyées par la queue de messages et les affiche directement à la console. Le dialogue avec le décodeur se fait à travers une liaison série pour le programme basic et une liaison ethernet (via une connexion SSH) pour le programme de test.

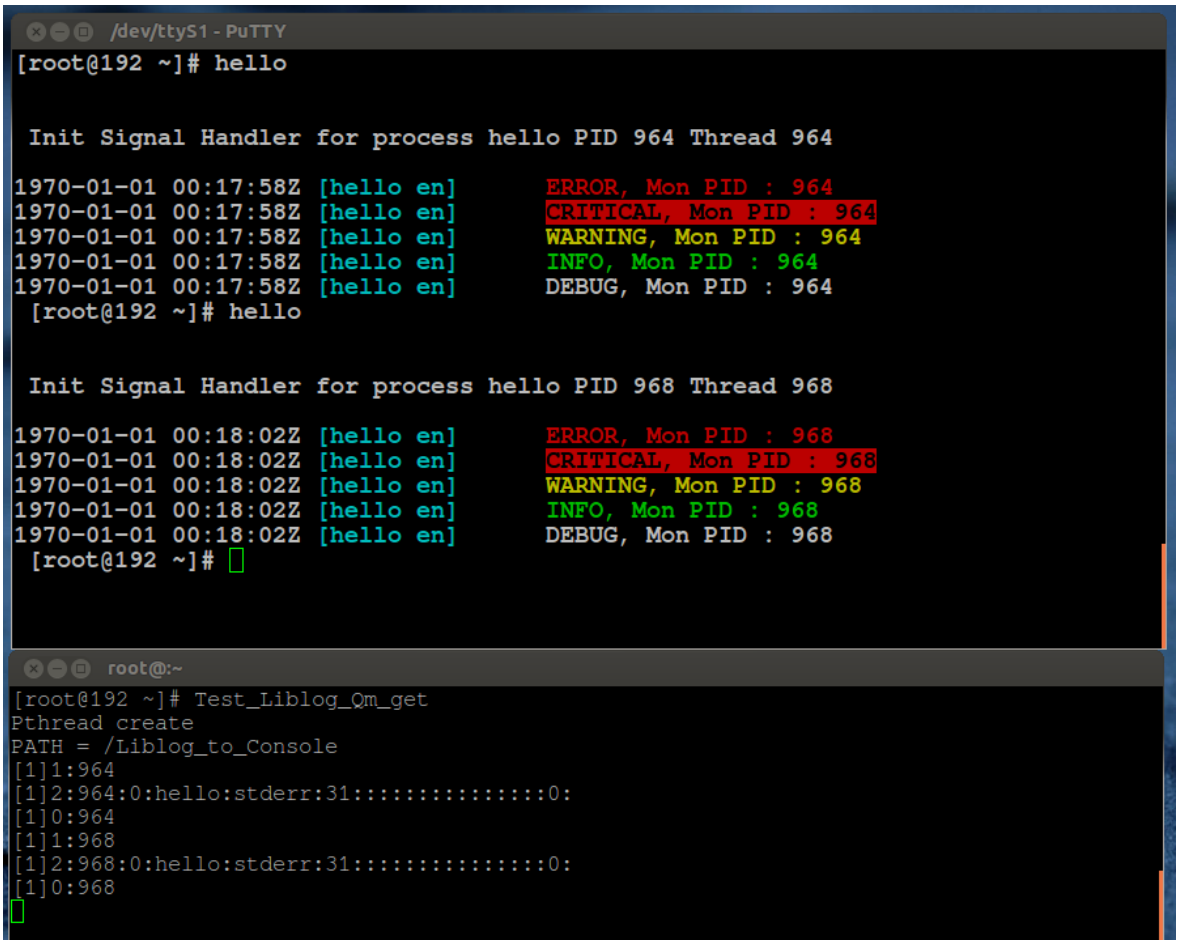


Figure 55 Test unitaire de l'envoi d'informations

Dans la fenêtre terminale du haut, j'exécute deux fois le programme *hello*, ces deux programmes envoient leur configuration par queue de messages à l'adresse « Liblog_to_Console ». Comme on peut le constater le programme de test *Test_Liblog_Qm_get* [4] reçoit bien les trois messages de chaque programme *hello*. L'initialisation *(1 :N° PID)*, La mise à jour *(2 :N° PID :information)* et la fin d'utilisation de liblog *(0 :N° PID)*.

[4] Code source disponible en annexe1

VII.1.2 La collecte d'informations

VII.1.2.1 Présentation

La collecte d'informations se fait par un *daemon*, c'est un programme qui se lance au démarrage du décodeur et qui tourne en tâche de fond. Son rôle est de collecter les informations des différents processus utilisant *liblog*. De stocker ces informations dans un tableau qui sera accessible à l'aide d'une projection de mémoire.

La première étape est d'initialiser le *daemon*, c'est-à-dire vérifier qu'il n'est pas déjà lancé. Puis initialiser le tableau d'informations, la projection de mémoire et la réception des queues de messages.

Le deuxième étape est de gérer le tableau d'informations à chaque message reçu afin de le mettre à jour.

VII.1.2.2 Développement

J'ai d'abord défini la structure qui me servira à stocker les informations reçues par une librairie.

Int	pid
Int	processus_On/Off
Char[32]	name
Char[16]	backend[x]->name
Int	backend[x]->mask
Int	filtre_config
Char[256]	filtre

```
22
23 struct log_backend_info {
24         unsigned int pid;                    /**< processus pid */
25         unsigned int on_off;                 /**< processus trace off : 1, on : 0 */
26         char name[32];                       /**< Processus Name */
27         struct {
28                 char name[16];               /**< Backend : format of the message */
29                 unsigned int mask;           /**< Backend : priority level mask */
30         } backend_info[8];
31         unsigned int filtre_config;          /**< Localisation of Environement configuration file*/
32         char filtre[256];
33 };
34
```

Figure 56 : structure et code source pour la collecte d'informations

Voici les diagrammes d'activités et le code source du *daemon* chargé de la collecte d'informations.

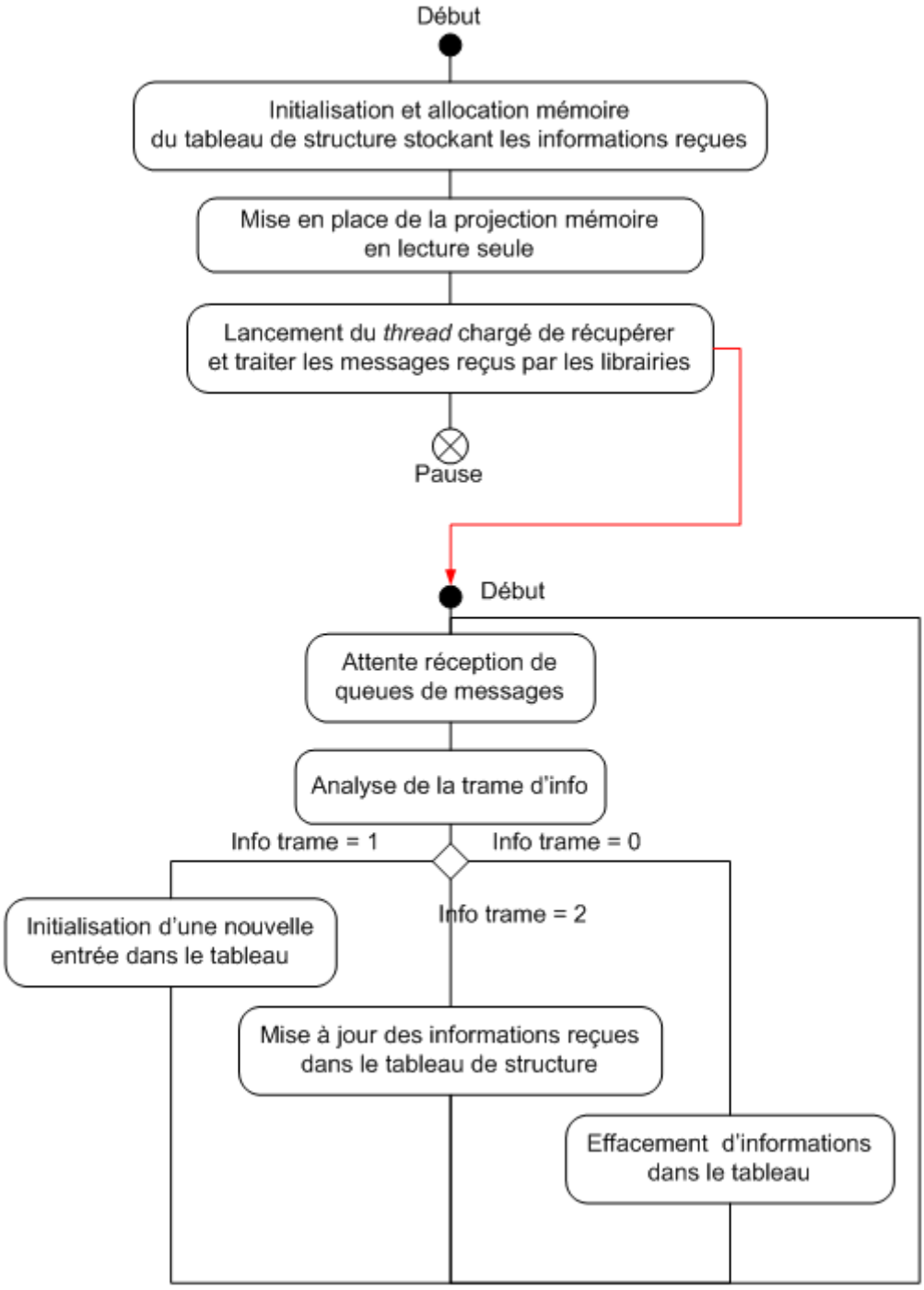


Figure 57 : diagrammes d'activités pour la collecte d'informations

```c
int main(int argc, char * argv[])
{
        /*variable local */
        int fd;
        pthread_t threadId;

        /*memory allocation for information table*/
        tableau = malloc (128 * sizeof(*tableau));
        if( tableau==NULL ) {
                fprintf(stderr, "ERROR: cannot allocate tableau\n");
                exit (-1);
        }
        memset(tableau, 0, 128 * sizeof(*tableau));

        /* memory share*/
        shm_unlink("/liblog_info");
        fd = shm_open("/liblog_info", O_RDWR | O_CREAT , 0400);
        ftruncate(fd,  128 * sizeof(*tableau));
        tableau = mmap(NULL, 128 * sizeof(*tableau), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (tableau == MAP_FAILED) {
                fprintf(stderr,"erreur mmap\n");
                perror("mmap");
                exit(1);
        }

        /* thread launch for message Queue reception*/
        pthread_create(&threadId, NULL, &inotify_event_handler, NULL);

        pause();

 return 0;
}
void tableau_maj()
{
char buf[8192], *str = buf;
char *value;
int check=0;
int pid=0;

        /*recopy of message buffer in buf temporay*/
        strncpy(buf, Buffer, sizeof(buf));

        /*value get string (frame_info) before ":" */
        value = strsep(&str, ":");
        check = atoi(value);
        /*value get string (PID) before ":" */
        value = strsep(&str, ":");
        pid = atoi(value);

        /*check frame info value and call appropriate functions*/
        switch (check)
        {
        case 0 :
                tableau_maj_sup(pid);
        break;
        case 1 :
                tableau_maj_ajout(pid);
        break;
        default :
                tableau_maj_info(pid);
        break;
        }


}

static void * inotify_event_handler(void *data)
{
        MQD_console_get_init();
        while (1)
        {
                Buffer = MQD_console_get();
                tableau_maj();
        }

 printf("inotify_event_handler exit");

 return NULL;
}
```

Figure 58 : codes sources de la collecte d'informations

VII.1.2.3 Test unitaire

Je vais utiliser le même programme basique *hello* et un programme de test unitaire qui vient lire la mémoire partagée du *daemon « dLiblog_console »*.

Tout d'abord, je vais vérifier que celui-ci est bien lancé en tâche de fond grâce à la commande *ps,* qui affiche tous les processus en cours d'exécution.

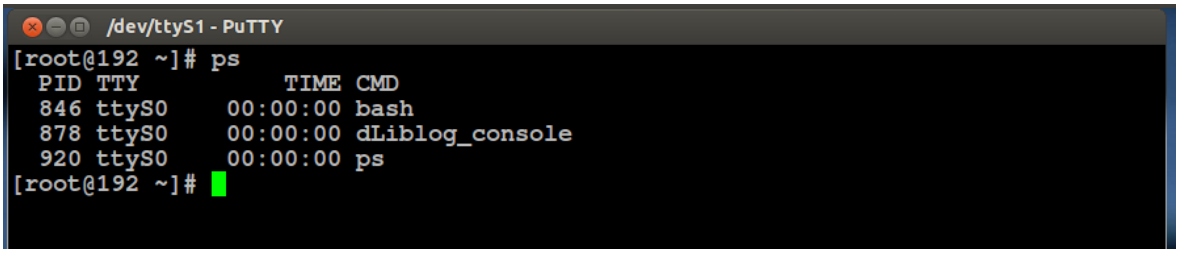


Figure 59 : Vérification de la présence du *daemon « dLiblog_console »*

Puis je lance le programme *hello* et exécute le programme de test *Test_dconsole_Mmap_ge*[5]*t* pour visualiser les informations enregistrées par le *daemon.*

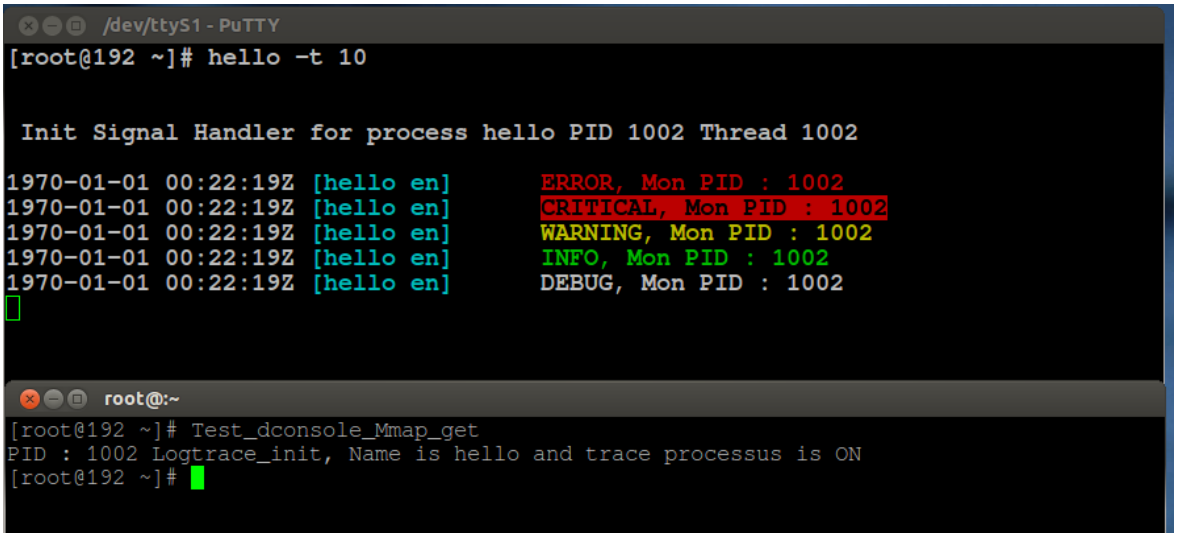


Figure 60 : Test unitaire de la collecte d'informations

Test_dconsole_Mmap_get m'affiche bien que le PID 1002, qui correspond à mon programme *hello,* a initialisé sa librairie *liblog* et que les traces sont activées.

[5] Code source disponible en annexe 2

VII.1.3 L'interface de commande

VII.1.3.1 Présentation

L'interface de commande doit pouvoir afficher les informations stockées par le *daemon*, ainsi les utilisateurs pourront ensuite modifier dynamiquement la configuration des traces.

La première étape est de définir l'utilisation de la commande *liblog_console*. Elle peut soit afficher des informations, soit permettre la mise à jour de la configuration des traces dans les différentes librairies *liblog* initialisées. Pour cela, il faut passer des paramètres à la console, comme par exemple *liblog_console –t off –p 1024.* Ce qui signifie que l'on désactive les traces du processus numéro 1024.

Voici la liste des paramètres possibles et leurs actions.

- –s

affiche les informations stockées dans le *daemon* sur les différentes configurations de libraires initialisées.

- –h

Affiche l'aide sur l'utilisation de la commande *liblog_console.*

- –p *numéro_PID*

Seuls les PID renseignés seront pris en compte par la commande. Si –p n'est pas renseigné, alors tous les PID seront impactés.

- –n *primary* ou *primary:secondary* ou *primary:secondary,…,secondary*

Renseigne les noms primaires et secondaires sur lesquels s'appliqueront la règle de filtrage définie ci-dessous.

- –f *off* ou *enable* ou *disable*

Permet de (*off*) désactiver les filtres sur les noms, ou de (*enable*) filtrer uniquement les noms renseignés par l'option –n ou de (*disable*) ne pas afficher les traces comportant les noms indiqués. Si aucun nom n'est renseigné, le filtre ne s'applique pas.

- –m *error* ou *critical* ou *warning* ou *info* ou *debug*

  Permet de reconfigurer le niveau de traces en sortie des *back-end*

- –t *on* ou *off*

Permet l'activation (*on*) ou la désactivation (*off*) des traces pour les PID sélectionnés. Si aucun PID n'est renseigné dans la commande, alors tout les PID référencés dans le *daemon* seront impactés.

La deuxième étape est d'envoyer vers les différentes librairies la mise à jour à effectuer. Voici donc le formalisme d'une trame émise pour une mise à jour, où *Info_trame* est équivalent au paramètre passé à la commande

Info_trame **:** *Valeur*

VII.1.3.2 Développement

Voici le diagramme d'activité et le code source de l'interface de commande qui permet à l'utilisateur de modifier dynamiquement les traces.

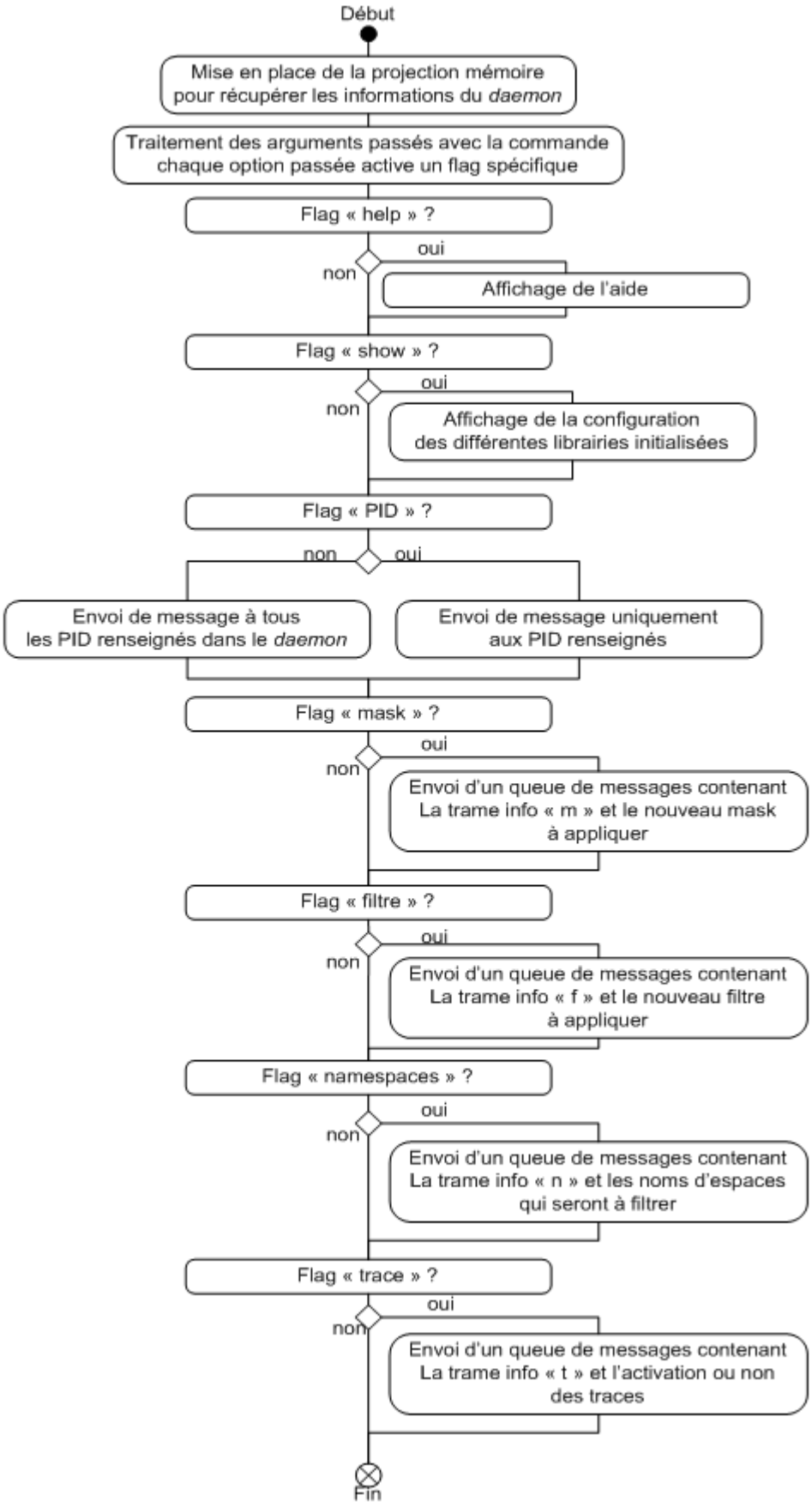


Figure 61 : diagramme d'activité pour l'interface de commande

```
        /*memory sharing with daemon dLiblog_console*/
        fd = shm_open("/liblog_info", O_RDWR, 0600);
        tableau = mmap(NULL, 128 * sizeof(*tableau), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (tableau == MAP_FAILED) {
                fprintf(stderr,"erreur mmap\n");
                perror("mmap");
                exit(1);
        }

        /*start analyse of argument pass with command */
        if (argc == 1)
        {
                log_print_help(operande_absent);
                return 0;
        }

        while ((opt = getopt(argc,argv, "n:p:t:f:m:hs")) != -1)
        {
                switch (opt) {
                case 't' :                      // active flag "trace"
                        if (strcmp(optarg,"off")==0)
                                trace=1;
                        if (strcmp(optarg,"on")==0)
                                trace=0;
                break;
                case 's' :                      // active flag "show"
                        show = 1;
                break;
                case 'h' :
                        help = 1;
                break;
                case 'f' :                      // active flag "filtre"
                        if (strcmp(optarg,"off")==0)
                                filtre = 0;
                        if (strcmp(optarg,"enable")==0)
                                filtre = 1;
                        if (strcmp(optarg,"disable")==0)
                                filtre = 2;
                break;
                case 'n' :                      // active flag "namespace"
                        for (j = optind-1 ; j <argc; j++){
                                if (strncmp(argv[j],"-",1)==0)
                                        break;

                                sprintf(message_tampon, "%s ", argv[j]);
                                strncat(tab_filtre, message_tampon, sizeof(message_tampon));
                                ns_filtre = 1;
                        }
                break;
                case 'm' :                      // active flag "mask"
                        mask = 0;
                        if (strcmp(optarg,"all")==0)
                                mask = 0x1F;
                        for (j = optind-1 ; j <argc; j++)
                        {
                        if (strncmp(argv[j],"-",1)==0)
                                break;
                        if ((strcmp(argv[j],"error")==0)||(strcmp(argv[j],"ERROR")==0))
                                mask = mask | 0x10;
                        if ((strcmp(argv[j],"critical")==0)||(strcmp(argv[j],"CRITICAL")==0))
                                mask = mask | 0x08;
                        if ((strcmp(argv[j],"warning")==0)||(strcmp(argv[j],"WARNING")==0))
                                mask = mask | 0x04;
                        if ((strcmp(argv[j],"info")==0)||(strcmp(argv[j],"INFO")==0))
                                mask = mask | 0x02;
                        if ((strcmp(argv[j],"debug")==0)||(strcmp(argv[j],"DEBUG")==0))
                                mask = mask | 0x01;
                        }
                break;
                case 'p' :                      // active flag "PID"
                        nbr_elem_tab=1;
                        for (j = optind-1 ; j <argc; j++)
                        {
                                if (strncmp(argv[j],"-",1)==0)
                                        break;

                                tab_pid[nbr_elem_tab] = atoi(argv[j]);
                                tab_pid[0] = nbr_elem_tab;
                                nbr_elem_tab++;
                                N_pid = 1;
                        }
                break;
                default :                       // if argument is not correct, display help_commande
                        log_print_help(operande_absent);
                break;
                }

        }

        /*start of action due to argument set with command */
        if (show==1)                            // if flag "show" display status of libary store in deamon
                affichage();

        if (help==1)                            // if flag "help" display help_commande
                log_print_help(help_commande);

        if (N_pid==1)                           // if flag "PID" message queue send only to PID pass in argument
        {
                for (j=1; j<=tab_pid[0]; j++)
                {
                        MQD_console_set_init(tab_pid[j]);

                        sprintf(message_tampon, "%d", trace);
                        if (trace!=-1)          // if flag "trace" message queue send with info_trame = t
                                MQD_console_send(tab_pid[j],"t",message_tampon);

                        sprintf(message_tampon, "%d", filtre);
                        if (filtre!=-1)         // if flag "filtre" message queue send with info_trame = f
                                MQD_console_send(tab_pid[j],"f",message_tampon);

                        if (ns_filtre==1)       // if flag "namespace" message queue send with info_trame = n
                                MQD_console_send(tab_pid[j],"n",tab_filtre);

                        sprintf(message_tampon, "%d", mask);
                        if (mask!=-1)           // if flag "mask" message queue send with info_trame = m
                                MQD_console_send(tab_pid[j],"m",message_tampon);

                        MQD_console_clean(tab_pid[j]);
                }

        }else                                           // if NO flag "PID" message queue send to all PID store in daemon
        {
                for (j=0; j<128;j++){
                        if (tableau[j].pid != 0)
                        {
                                MQD_console_set_init(tableau[j].pid);

                                sprintf(message_tampon, "%d", trace);
                                if (trace!=-1)          // if flag "trace" message queue send with info_trame = t
                                        MQD_console_send(tableau[j].pid,"t",message_tampon);

                                sprintf(message_tampon, "%d", filtre);
                                if (filtre!=-1)         // if flag "filtre" message queue send with info_trame = f
                                        MQD_console_send(tableau[j].pid,"f",message_tampon);

                                if (ns_filtre==1)       // if flag "namespace" message queue send with info_trame = n
                                        MQD_console_send(tableau[j].pid,"n",tab_filtre);

                                sprintf(message_tampon, "%d", mask);
                                if (mask!=-1)           // if flag "mask" message queue send with info_trame = m
                                        MQD_console_send(tableau[j].pid,"m",message_tampon);

                                MQD_console_clean(tableau[j].pid);
                        }
                }
        }
```

Figure 62 : code source de l'interface de commande

VII.1.3.3 Test unitaire

Je vais tout d'abord tester l'affichage à l'aide de la commande *Liblog_console,* puis l'affichage d'informations contenues dans la mémoire du *daemon.* Enfin j'utiliserai un programme de test unitaire qui affichera l'envoi des queues de messages faites par *Liblog_console* afin de vérifier son formalisme.

L'affichage de l'aide pour savoir comment utiliser la commande *Liblog_console* se fait en ajoutant l'option *–h* après celle-ci.

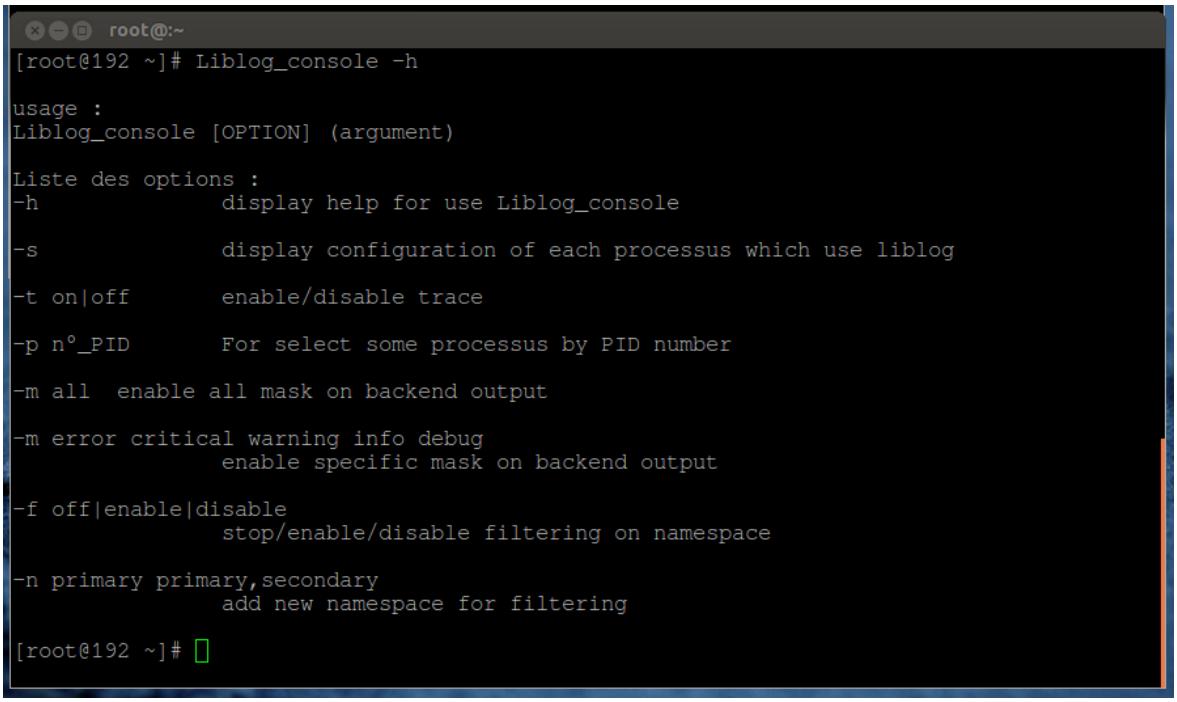


Figure 63 : Test pour afficher l'aide de l'interface de commande

La commande *Liblog_console* permet aussi de connaître tous les processus qui ont la librairie *liblog* initialisée. Ceci en affichant les informations contenues dans la mémoire du *daemon « dLiblog_console ».* Je vais donc exécuter mon programme *hello* puis lancer la commande *Liblog_console –s* pour afficher la configuration de la libraire de ce programme.

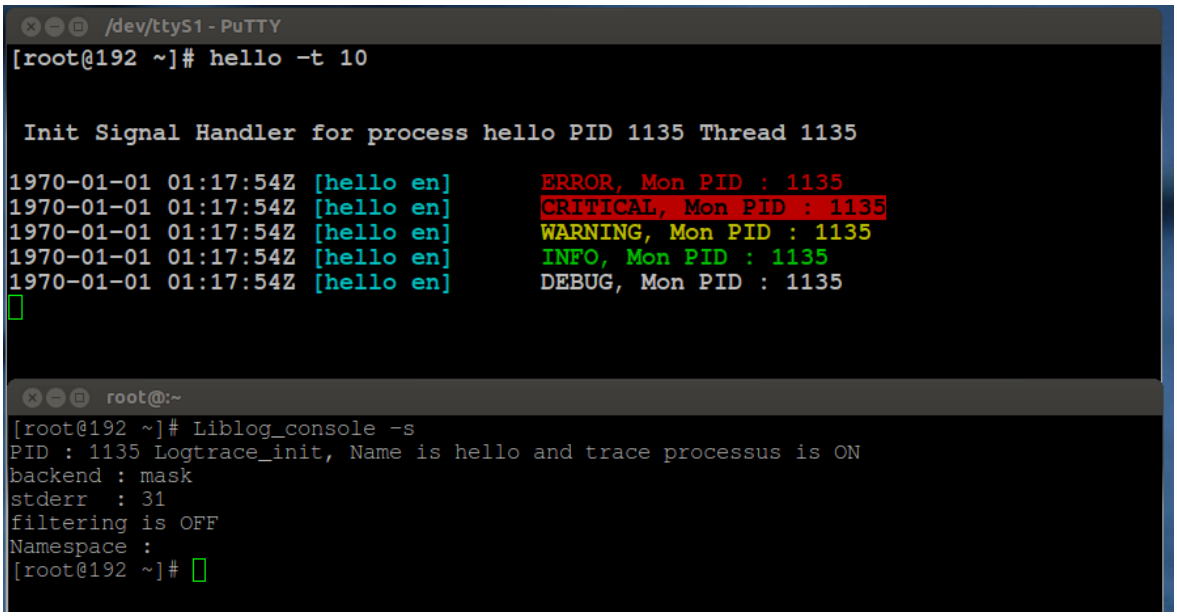


Figure 64 : Test pour afficher le statut de l'interface de commande

Enfin je vais tester le formalisme des trames envoyées par la commande *Liblog_console* pour la modification de la configuration des traces à émettre. Le test unitaire *Test_console_Qm_get*[6] *862* va afficher les queues de messages émises par la commande. La valeur 862 représente le PID vers lequel la commande envoie les trames.

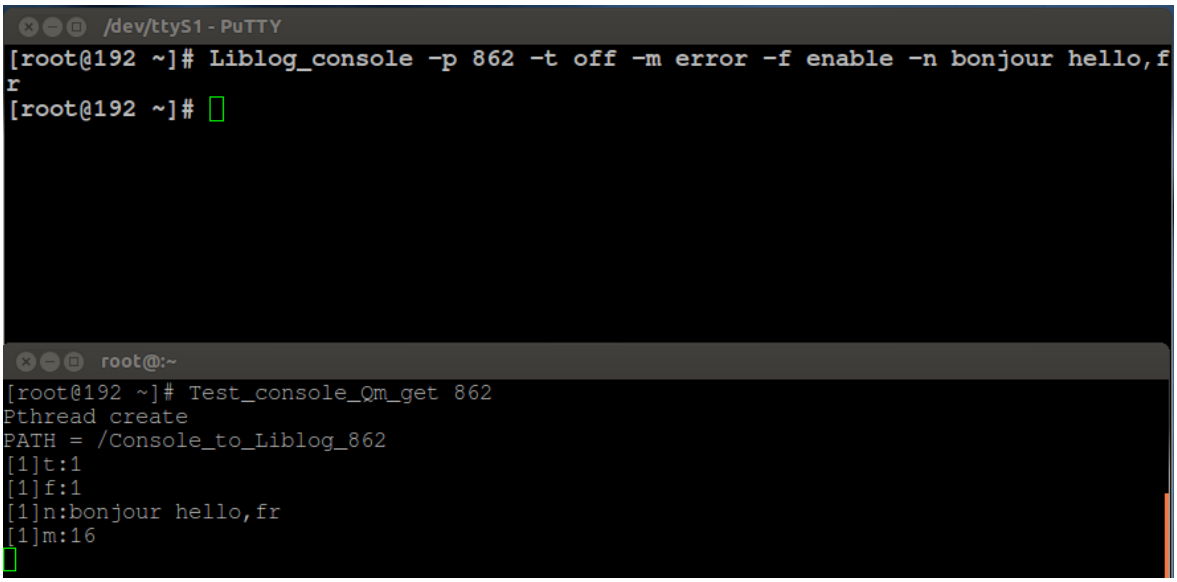


Figure 65 : Test unitaire de l'interface de commande

[6] Code source disponible en annexe 3

VII.1.4 La modification dynamique des traces

VII.1.4.1 Présentation

La modification dynamique est le dernier élément à mettre au point pour finaliser l'interface de commande. Son rôle est de reconfigurer la structure *my_log* de la librairie *liblog* selon les paramètres envoyés par l'utilisateur à travers l'interface de commande.

La première étape consiste à mettre en place un *thread* qui sera chargé de récupérer les queues de messages envoyées par l'interface de commande.

La deuxième étape est de traiter l'information reçue afin de mettre à jour la librairie. La mise à jour peut concerner :

- L'activation ou la désactivation des traces du processus.
- La mise en place du filtrage sur les noms d'espace.
- Le changement de niveau de traces en sortie de la librairie.

La troisième et dernière étape est l'envoi de la mise à jour de la structure *my_log* au *daemon.*

VII.1.4.2 Développement

Voici le diagramme d'activité et le code source qui permet de modifier dynamiquement ses traces dans une librairie.

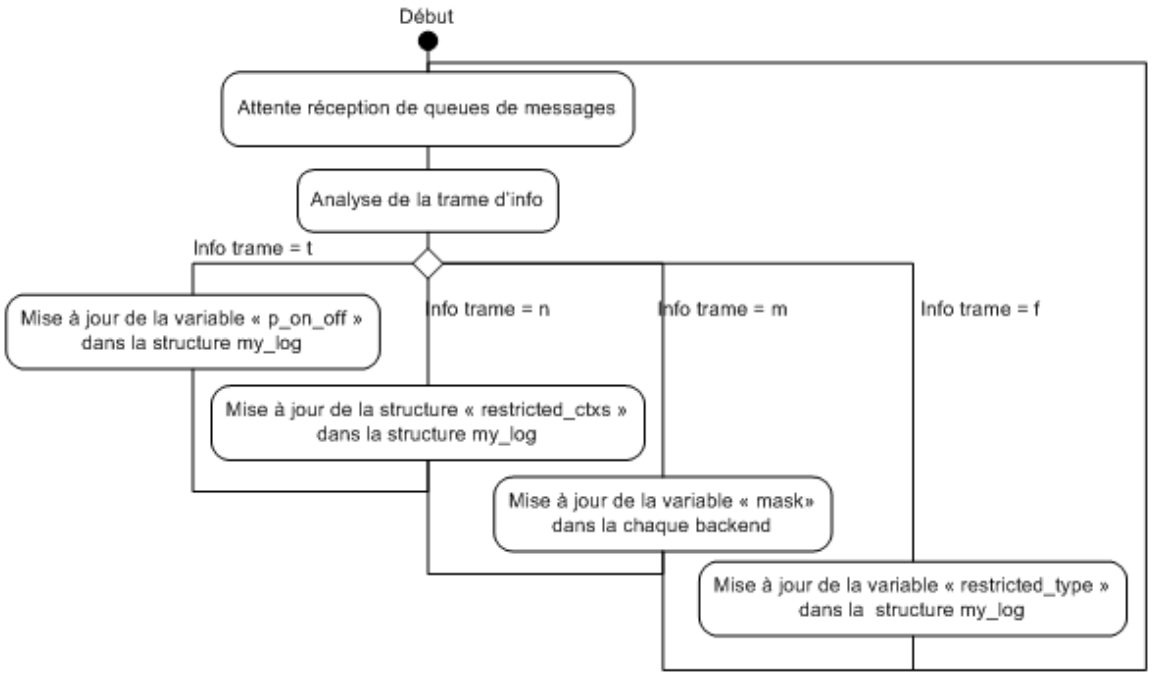


Figure 66 : diagramme d'activité pour la modification dynamique des traces

```
117         while (1)
118         {
119                 /* wait message from Liblog_console*/
120                 Buffer_recu = MQD_lib_get();
121
122                 /* analyse of message for separate info_trame of value */
123                 tmp = strchr(Buffer_recu, ':');
124                 tmp[0] = '\0';
125                 tmp++;
126
127                 info_MQ.val = tmp;
128                 info_MQ.key = Buffer_recu;
129
130
131                 PTHREAD_MUTEX_LOCK( mutex_change);
132                 /*analyse of info_trame*/
133                 switch (info_MQ.key[0]) {
134                 case 't' :                      //info_trame "t" -> update trace on/off in my_log structure
135                         my_log->p_on_off = atoi(info_MQ.val);
136                 break;
137                 case 'n' :                      //info_trame "n" -> update namespace in my_log structure
138                         strncpy(buf, info_MQ.val, sizeof(buf));
139
140                         /* analyse of value for separate primary and secondary namespace*/
141                         while ( (primary = strsep(&str, " ")) ) {
142                                 struct log_restricted_namespaces *dn ;
143                                 int secondary_found = 0;
144
145                                 if (!*primary)
146                                         continue;
147
148                                 secondaries = primary;
149                                 strsep(&secondaries, ",");
150                                 while( (secondary = strsep(&secondaries, ",")) ) {
151                                         if( !*secondary )
152                                                 continue;
153
154                                         secondary_found = 1;
155                                         dn = log_malloc(sizeof(*dn));
156                                         if( !dn ) {
157                                                 log_print_error("%s ERROR: cannot allocate memory", __FUNCTION__);
158                                                 return NULL;
159                                         }
160                                         dn->primary   = log_strdup(primary);
161                                         dn->secondary = log_strdup(secondary);
162                                         log_list_add(my_log->restricted_ctxs, dn);
163                                 }
164                                 if( !secondary_found){
165                                         dn = log_malloc(sizeof(*dn));
166                                         if( !dn ) {
167                                                 log_print_error("%s ERROR: cannot allocate memory", __FUNCTION__);
168                                                 return NULL;
169                                         }
170                                         dn->primary   = log_strdup(primary);
171                                         dn->secondary = NULL;
172                                         log_list_add(my_log->restricted_ctxs, dn);
173                                 }
174                         }
175                         log_maj_id++;
176                 break;
177                 case 'm' :                      //info_trame "m" -> update mask in user context
178                         log_mask_console = atoi(info_MQ.val);
179                         log_maj_id++;
180                 break;
181                 case 'f' :                      //info_trame "f" -> update mask in my_log structure
182                         my_log->restricted_type = atoi(info_MQ.val);
183                         log_maj_id++;
184                 break;
185                 }
186
187                 PTHREAD_MUTEX_UNLOCK(mutex_change);
188                 /*send update of my_log structure to daemon*/
189                 structure_info_send(my_log->pid);
190         }
191
192     return NULL;
193 }
```

Figure 67 : Code source de la modification dynamique des traces

VII.1.4.3 Test unitaire

Je vais utiliser deux programmes basiques, *hello* et *bonjour,* pour qu'ils envoient des traces toutes les dix secondes. Avec l'interface de commande, je vais tester les différents cas de modifications dynamiques sur ce programme.

Dans ce premier test, je vais désactiver les traces d'un des programmes en lançant la commande *Liblog_console –p n° pid –t off*. Puis je vais les réactiver avec la commande *Liblog_console –p n° pid –t on*.

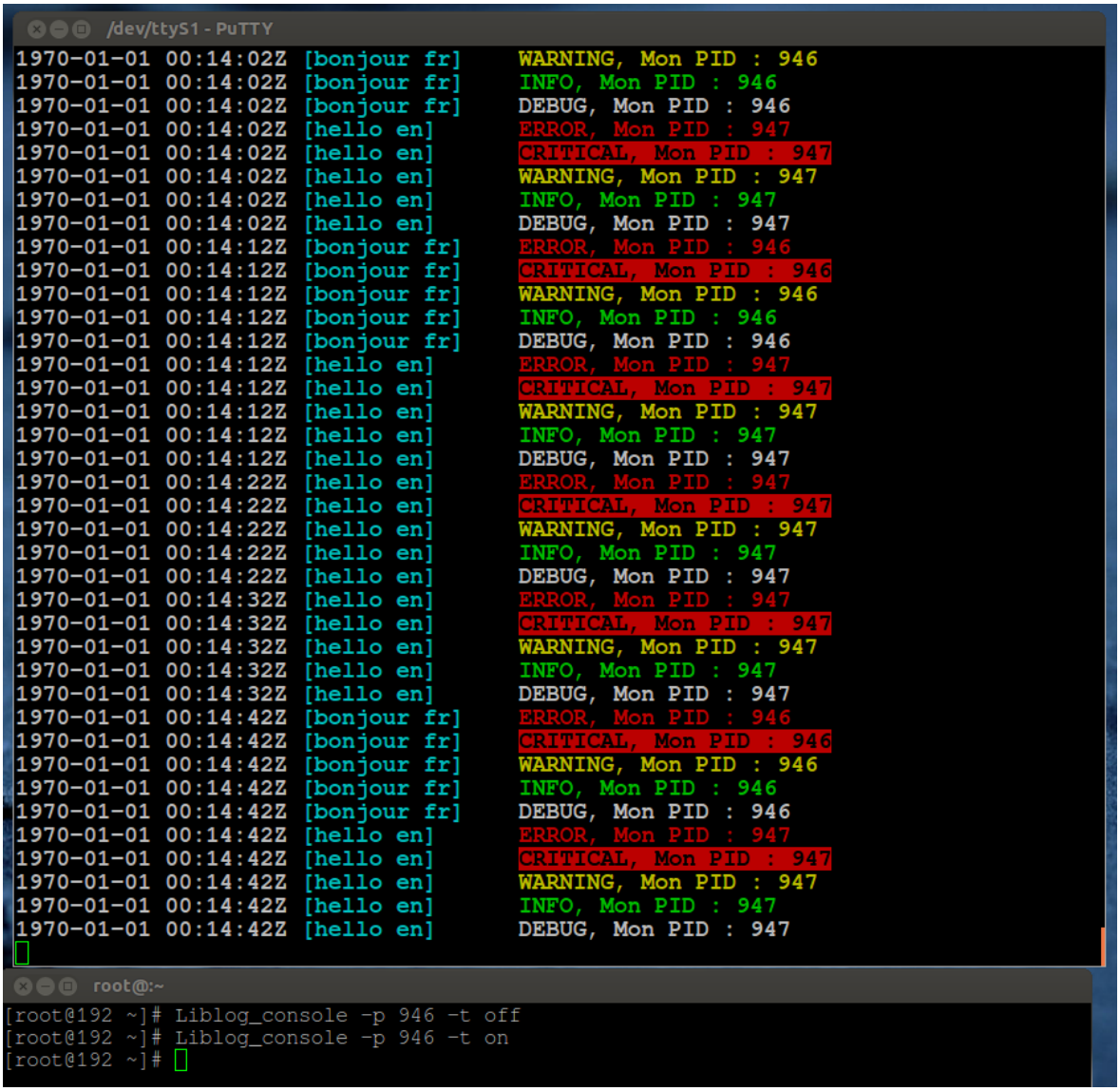


Figure 68 : Test unitaire de la désactivation de traces

On remarque que le programme *bonjour* n'émet plus de traces pendant quelques secondes.

Dans ce deuxième test, je vais désactiver les traces d'un des programmes en mettant en place un filtrage. Je vais exécuter la commande *Liblog_console –n bonjour –f enable*. Puis je vais désactiver le filtre avec la commande *Liblog_console –f off*.

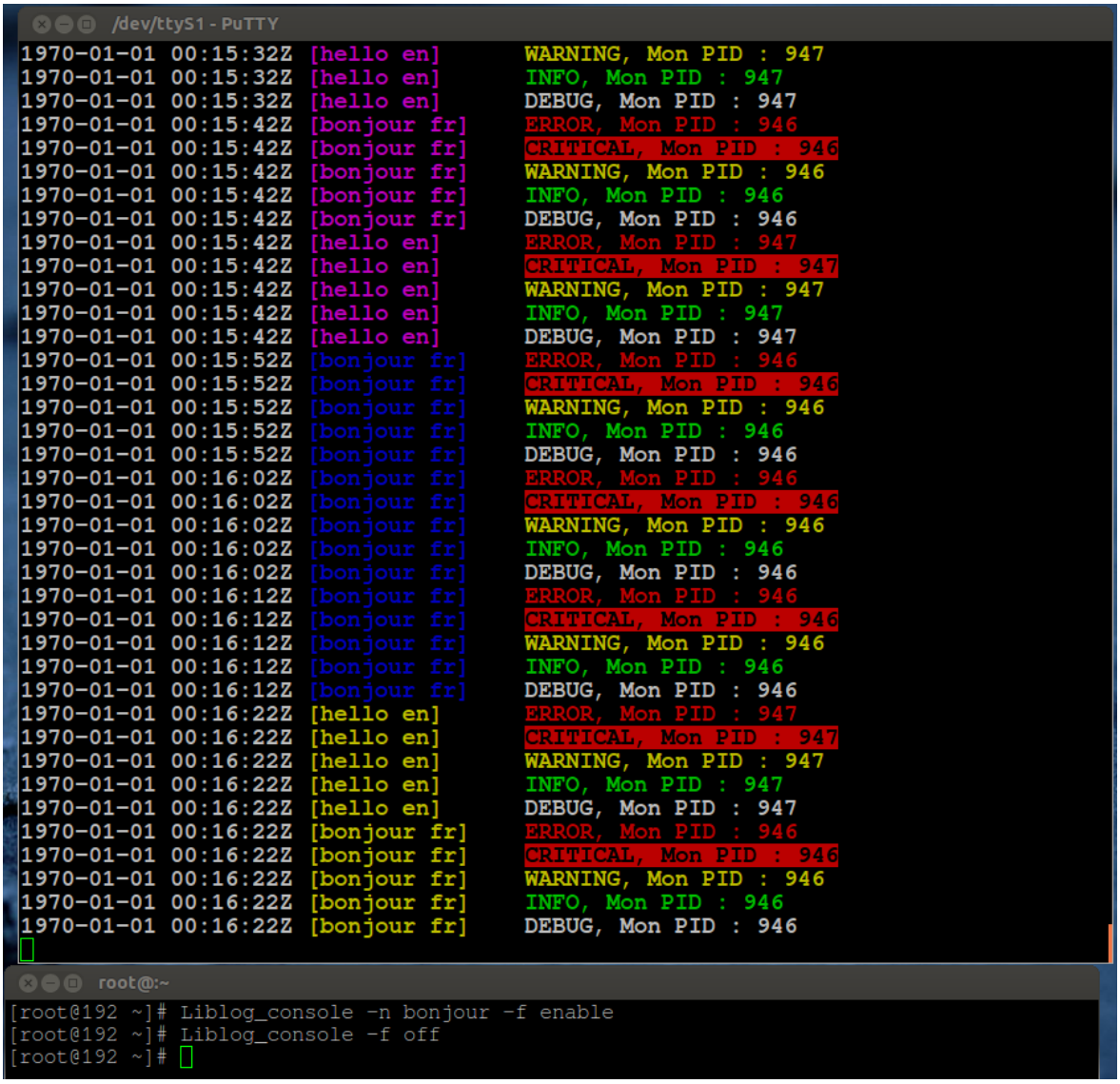


Figure 69 : Test unitaire de la mise en place de filtres

On constate que le programme *bonjour* est le seul à émettre des traces pendant quelques secondes.

Dans ce dernier test, je vais modifier le niveau de traces en sortie en lançant la commande *Liblog_console –m warning*. Puis je vais remettre le niveau de traces par défaut avec la commande *Liblog_console –m all*.

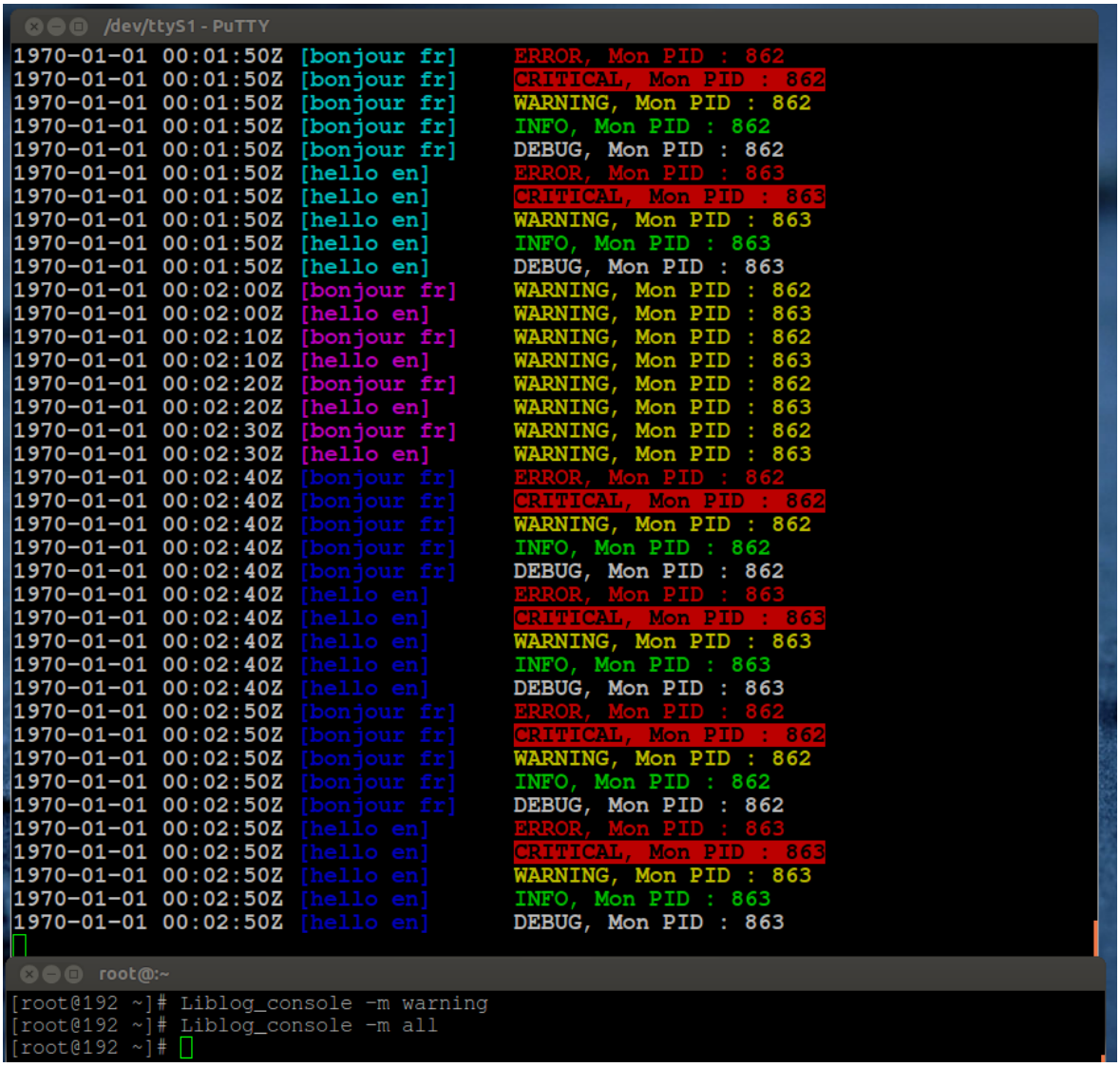


Figure 70 : Test unitaire du changement de niveau de traces

On remarque que les programmes *bonjour* et *hello* n'affichent que les messages de type *warning* pendant quelques secondes.

VII.1.5 Suivi du planning

La réalisation de la console d'interface dynamique est représentée par les lettres allant de K à Q sur le planning de développement. Le temps estimé était compris entre 14 et 21 jours ouvrés, il m'a fallu 24 jours car j'ai intégré les tests unitaires. La raison est que cette interface est une nouveauté de l'outil et qu'il était important de le valider correctement avant de débuter d'autres tâches.

VII.2 L'outil LTTng

VII.2.1 L'intégration de LTTng dans notre noyau

VII.2.1.1 Présentation

Comme je l'ai présenté dans le paragraphe III.2.2, l'outil LTTng a toujours progressé à l'extérieur des sources du noyau officiel. Avant de pouvoir développer son back-end, il me faut donc intégrer à notre noyau ces cinq paquets :

- lttng-tools : Ce paquet fournit les outils nécessaires pour contrôler LTTng depuis l'espace utilisateur.
- lttng-modules : Ce paquet contient les modules *kernel* LTTng 2.0 nécessaires à l'instrumentation du noyau.
- lttng-traces : Ce paquet contient la librairie "UST runtime" nécessaire à l'exécution des applications utilisant la librairie lltng-ust.
- lttng-ust : Cette bibliothèque peut être utilisée par les applications de l'espace utilisateur pour générer des points de traces dans l'outil LTTng.
- userpace-rcu : Cette bibliothèque permet d'accéder, en lecture, aux données générées par LTTng.

Ces cinq paquets sont donc à intégrer à notre noyau. Pour cela, je doit récupérer les codes sources disponibles sur internet, vérifier si les options qui lui sont nécessaires sont activées dans le *noyau,* si besoin recompiler le noyau avant de compiler les différents codes sources, puis intégrer les différents modules *kernel* dans le *noyau*. Enfin j'intégrerai le front-end de LTTng dans notre environnement de développement Eclipse afin de réaliser une capture de l'outil qui permettra de valider son intégration.

VII.2.1.2 Intégration

Après avoir recompilé le noyau en ayant activé les options *« Prompt for development and/or incomplete code/drivers »* et *« Activate markers »,* j'ai compilé les cinq paquets en suivant la procédure décrite dans chaque fichier README, qui consiste à faire un *configure* pour paramétrer la compilation, un *make* pour compiler et un *make install* pour installer les modules kernel générés par la compilation.

Pour tester l'outil LTTng, j'ai utilisé le front-end qui a été développé pour l'environnement de développement Eclipse.

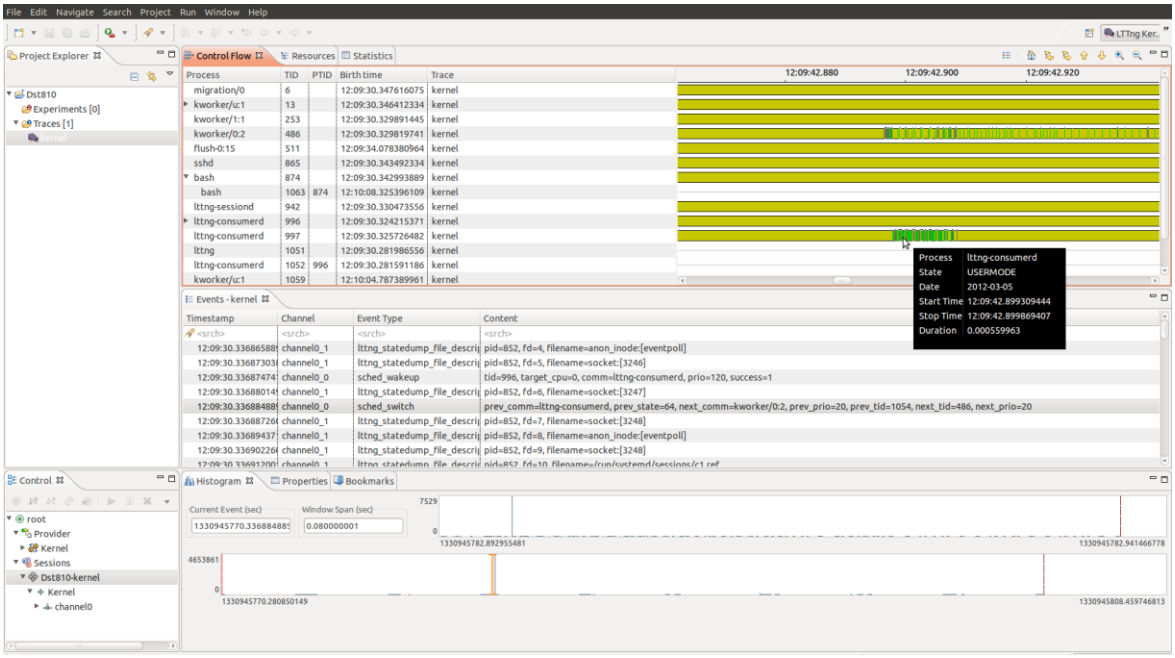


Figure 71 : Intégration de l'outil LTTng

VII.2.2 Développement du back-end LTTng

VII.2.2.1 Présentation

Les back-end ont pour rôle de formater et transmettre la trace vers un front-end d'affichage (le terminal) ou de stockage (un fichier d'enregistrement). Ils sont chargés dynamiquement par la librairie *liblog*. Grace à ce principe, il est possible d'ajouter de nouveaux back-end.

La librairie *liblog* ne sait pas par avance quelles librairies seront chargées, on doit par conséquent respecter le même formalisme dans les noms de fonctions de chaque back-end. Celui-ci doit comporter les fonctions suivantes :

- backend_init : cette fonction est appelée lors de l'initialisation du backend par la librairie *liblog.*

- cb_send : cette fonction est appelée pour transmettre une trace au backend
- cb_cleanup: cette fonction est appelée pour désinstaller un back-end et libérer la mémoire.

Pour transmettre les traces vers l'outil LTTng, je dois définir un TRACEPOINT_EVENT, tel que défini dans la page manuel de *lttng_ust,* et qui sera appelé dans la fonction *cb_send.*

VII.2.2.2 Développement

J'ai repris le code source du back-end « syslog » que j'ai adapté pour LTTng. Ci-après le code source du back-end LTTng, ainsi que la définition de TRACEPOINT_EVENT.

```c
#define DEFAULT_FORMAT "~f ~m"
#define TRACEPOINT_DEFINE
#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
#include "backend_lttng.h"


/* ---------------------------------------------------------------------------
 *   callbacks
 */

/**
 * Sending callback
 *
 * \param[in] msg  Message data
 *
 * \return  number of bytes or 0 if message send, error otherwise
 */
static int
cb_send(struct log_backend *self, struct log_msg *msg)
{
        char buf[MAX_LOG_MESSAGE];

        /* check if the backend has been initialized */
        if( !self )
                return -EACCES;

        log_format_message(buf, sizeof(buf)-1, self->format, msg);
        /*send message to LTTng tools*/
        tracepoint(liblog_component, message, buf);
        return 0;
}


/**
 * Cleanup callback
 */
static void
cb_cleanup(struct log_backend *self)
{
        /* check if the backend has been initialized */
        if( !self )
                return;

        log_free(self->format);
        log_free(self->name);
        log_free(self);

        lib_backend_cleanup(NULL);
}


/* ---------------------------------------------------------------------------
 *   entry point
 */

/**
 * Backend initinialization
 *
 * \param[in] name    Backend name
 * \param[in] opts    Backend options
 *
 * \return  backend  Structure initialized
 */
struct log_backend *backend_init(const char *name,
              const struct log_backend_options *opts)
{
        struct log_backend *backend;

        /* First initialize the lib */
        if( lib_backend_init(NULL) < 0 )
                return NULL;

        /* Allocate backend */
        backend = log_malloc(sizeof(*backend));
        if( !backend ) {
                lib_backend_cleanup(NULL);
                return NULL;
        }
        memset((void *)backend, 0, sizeof(*backend));

        /* Set callbacks */
        backend->send    = cb_send;
        backend->cleanup = cb_cleanup;
        backend->registr = NULL;
        backend->update  = NULL;

        backend->submitter = opts->submitter;
        backend->mask      = opts->mask;
        backend->name = log_strdup(name);
        if(opts->format)
                backend->format = log_strdup(opts->format);
        else
                backend->format = log_strdup(DEFAULT_FORMAT);

        return backend; /* ok */
}


/* ---------------------------------------------------------------------------
 *   backend properties
 */

struct log_backend_properties backend_properties = {
        .default_submitter  = LOG_SUBMIT_SYNC,
        .opt_handlers       = NULL,
};
```

Figure 72 : Code source du back-end LTTng

VII.2.2.3 Test unitaire

Je vais utiliser deux programmes basiques, *hello* et *bonjour,* pour qu'ils envoient des traces toutes les secondes. Avec le front-end Eclipse je vais réaliser une capture de quelque secondes pour visualiser le comportement de mes deux programmes dans le noyau.

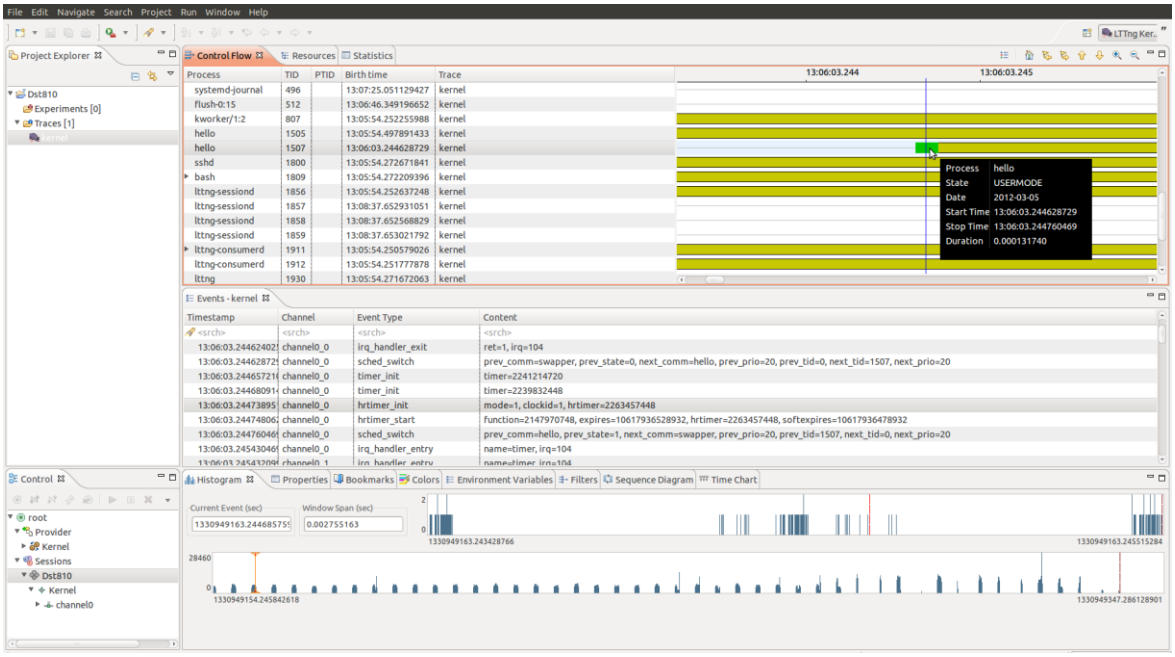


Figure 73 : test unitaire du back-end LTTng

VII.2.3 Suivi du planning

L'intégration de l'outil LTTng et le développement de son back-end sont représentés par les lettres G et H sur le planning de développement. Le temps estimé était compris entre 8 et 11 jours ouvrés, il m'a fallu 9 jours pour y parvenir. A ce stade, j'ai une journée de retard sur le planning. Ici se termine la présentation de mes réalisations.

VII.3 Conclusion sur mes réalisations

L'objectif est de faire un état de l'avancement des différentes réalisations, afin de définir des suites à donner à ce projet. Le tableau qui suit récapitule les tâches réalisées.

Tableau VII : Tableau des tâches réalisées

	Tâches
Amélioration de *liblog.so*	
B	Afficher les numéros PID et thread id qui génère la trace
Intégration de nouveaux back-end	
G	Intégrer de l'outil LTTng
H	Développer un back-end pour LTTng
Interface de commande	
K	Envoi depuis *liblog.so* de queues de messages vers *daemon*
L	Réception de queues de messages depuis un *daemon*.
M	Création depuis le daemon d'une zone mémoire partagée
N	Affichage information zone mémoire depuis commande en ligne.
O	Affichage de l'aide depuis commande en ligne
P	Envoi depuis commande en ligne de queues de messages vers processus
Q	Réception et traitement de queues de messages dans *liblog.so*
R	Ajout d'une interface web

Comme je viens de le présenter, j'ai réalisé le développement de l'interface de commande ainsi que l'intégration du nouveau back-end LTTng.

J'ai aussi effectué l'amélioration de l'affichage du PID et du Thread ID, mais je n'ai pas eu le temps de l'intégrer à ce rapport. Le tableau suivant montre les tâches restantes à développer.

Tableau VIII : Tableau des tâches restantes

	Tâches
Amélioration de *liblog.so*	
A	Limiter la taille des fichiers générés par le back-end « files »
C	Afficher le nom du fichier, la fonction et le numéro de ligne qui génère la trace
D	Ajouter granularité pour chaque niveau de traces
E	Développer un constructeur/destructeur en C++
F	Transmettre les traces de façon synchrone/asynchrone
Intégration de nouveaux back-end	
I	Développer un back-end pour Gstreamer
J	Développer un back-end pour abonnement/notification.

Comme je l'avais identifié lors de l'élaboration du nouveau planning de développement, tous les développements n'ont pu être réalisés à la date butoir fixée à début avril. Mais les objectifs principaux fixés par mon maître de stage, qui étaient le développement de l'interface de commande et l'intégration de l'outil LTTng, ont été menés à bien.

Pour connaître les suites à donner à ce développement, j'ai réalisé un entretien individuel avec mon manager. Il m'a ainsi fixé comme objectifs de :

- Livrer une version stable de l'outil aux différentes équipes à partir de la mi-mai.
- Assurer des séances de formations sur son utilisation.
- Continuer le développement des tâches restantes jusqu'à la soutenance du mémoire.

Conclusion

L'objectif de ce mémoire était de répondre à la problématique de l'uniformisation d'un système de traces au sein des équipes de développement du middleware «Revolution-S». Pour l'atteindre, il m'a fallu définir le contexte et sujet, puis analyser les besoins des différents intervenants pour ensuite proposer une solution qui y réponde.

Une des difficultés a été d'acquérir les connaissances nécessaires sur le monde Linux et la culture open source, car le middleware « Revolution-S » repose sur le noyau Linux de Linus Torvalds.

Après avoir défini les termes sur les notions de traces et de débogage, j'ai étudié les différents outils existants dans le monde de l'open source qui pourraient aider à tracer le comportement de notre middleware. On constate d'après cette recherche qu'il existe de nombreuses solutions, mais la plupart sont mal documentées ou ont des versions de licences incompatibles avec notre middleware. Au final, c'est l'outil LTTng, qui permet de profiler le comportement d'un logiciel, qui a retenu mon attention. Pour savoir si celui-ci allait répondre à la problématique, il m'a fallu analyser les pratiques faites au sein des équipes pour tracer et déboguer le middleware.

Cette observation m'a permis de découvrir l'existence d'un outil de traçage, appelé liblog, propriétaire à TECHNICOLOR et déjà intégré à notre middleware. Cela m'a permis aussi de faire deux constats. Le premier, c'est que très peu de personnes utilisent cet outil par manque de communication sur son existence. Le deuxième, c'est l'hétérogénéité des outils utilisés et leur utilisation sporadique. Pour pouvoir uniformiser les outils, j'ai analysé le besoin des différents intervenants. Les conclusions montrent qu'une amélioration de l'outil existant semble être le meilleur choix. D'une part, parce que l'on garde l'historique des traces que l'outil génère dans le code, mais aussi par le fait que son architecture permet de s'interfacer avec d'autres outils comme LTTng.

J'ai ainsi défini la nouvelle architecture de l'outil liblog pour rendre dynamique la gestion d'affichage des traces. J'ai aussi intégré l'outil LTTng et développé le back-end d'interface dans liblog. LTTng va synchroniser les traces du middleware avec l'activité du noyau ce qui permettra d'analyser le comportement global du logiciel dans son environnement.

Ce nouveau système de traces devrait permettre d'avoir une solution uniformisée de traces au sein des équipes de développement. Mais cela devra passer par une phase de

communication et de formation sur l'outil, afin que chaque intervenant se l'approprie. Et une fois l'outil déployé, une maintenance devra être effectuée pour corriger les possibles bugs rencontrés et répondre aux interrogations des utilisateurs.

En l'état actuel, ce système de traces ne peut être laissé accessible dans nos produits finaux. Il ne peut être utilisé que durant les phases de développement puis retiré de la version livrée aux clients. La raison est d'éviter les risques de piratage de nos produits qui seraient préjudiciables pour nos clients opérateurs. Pourtant, il existe des besoins liés aux traces chez ces derniers et une perspective d'évolution pourrait porter sur la sécurisation des traces ou logs générés par notre outil.

Bibliographie

1. **PMI.** *A Guide to the Project Management Body of Knowledge.* s.l. : IEEE Guide, 1998.
2. **Blaess, Christophe.** *blaess.* [En ligne] 2003. [Citation : 5 Janvier 2013.] http://www.blaess.fr/christophe/articles/linux-histoire-dun-noyau/.
3. **Sauvage, Sébastien.** *s e b s a u v a g e* . [En ligne] [Citation : 5 Janvier 2013.] http://sebsauvage.net/comprendre/linux/.
4. **Jones, Tim.** *IBM.* [En ligne] 2007. [Citation : 18 Octobre 2012.] http://www.ibm.com/developerworks/linux/library/l-linux-kernel/.
5. **Bowman, Ivan.** *Computer Science & Information Systems.* [Online] 1998. [Cited: Janvier 15, 2013.] http://csis.bits-pilani.ac.in/faculty/sundarb/courses/old/spr04/ssg653/refers/lecs/linux-kernel.html.
6. **Ficheux, Pierre.** *Linux embarqué.* [éd.] Eyrolles. 4ème édition. s.l. : Blanche, 2012.
7. **Miller, Greg.** *MacTech Magazineweb site.* [Online] 2007. [Cited: Février 2013, 2013.] http://www.mactech.com/articles/mactech/Vol.23/23.11/ExploringLeopardwithDTrace/index.html.
8. **Jones, M. Tim.** *IBM developerWorks web site.* [Online] 2009. [Cited: Janvier 2013, 10.] http://www.ibm.com/developerworks/linux/library/l-systemtap/index.html.
9. **DESNOYERS, Mathieu.** *LTTNG web site.* [Online] 2009. [Cited: December 1, 2013.] http://lttng.org/files/thesis/desnoyers-dissertation-2009-12-v27.pdf.
10. **Keniston, Jim.** *Kernel Web site.* [Online] 2007. [Cited: December 1, 2012.] http://kernel.org/doc/ols/2007/ols2007v1-pages-215-224.pdf.
11. **Levon, John.** *Oprofile Web site.* [Online] 2005. [Cited: december 12, 2012.] http://oprofile.sourceforge.net/doc/index.html.
12. **Ananth, Mavinakayanahalli, Prasanna, Panchamukhi and Jim, Keniston.** *Kernel Web site.* [Online] 2006. [Cited: December 12, 2012.] http://www.kernel.org/doc/ols/2006/ols2006v2-pages-109-124.pdf.
13. **Baron, Jason.** *Kernel Web site.* [En ligne] 2010. [Citation : 2 Décembre 2012.] http://kernel.org/doc/htmldocs/tracepoint.html.
14. **Rostedt, Steven.** *Linux Weekly News Web site.* [En ligne] 2008. [Citation : 27 Novembre 2012.] http://lwn.net/Articles/290277/.
15. **Munsie, Ian.** *Slide Share web site.* [En ligne] 24 June 2010. [Citation : 14 November 2012.] http://fr.slideshare.net/DarkStarSword/instrumentation.
16. **C, Lonvick.** *The Internet Engineering Task Force Web site.* [En ligne] 2001. [Citation : 14 12 2012.] http://www.ietf.org/rfc/rfc3164.txt.
17. **Haende, Lars.** newty. [En ligne] Janvier 2002. [Citation : 24 Janvier 2013.] http://www.newty.de/fpt/zip/f_fpt.pdf.
18. **Hautrive, Patrick.** *hautrive.* [En ligne] [Citation : 5 Janvier 2013.] http://hautrive.free.fr/ordinateur/systemes-exploitation.html.
19. **Melo, Arnaldo Carvalho de.** *kernel Web site.* [En ligne] 2010. [Citation : 15 Janvier 2013.] http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf.
20. **Desnoyers, Mathieu.** *LTTNG web site.* [Online] 2009. [Cited: December 1, 2013.] http://lttng.org/files/thesis/desnoyers-dissertation-2009-12-v27.pdf.

Table des annexes

Annexe 1
Code source du test unitaire *Test_Liblog_Qm_get*

```
17  #include "../../message_IPC.h"
18
19
20  static void * inotify_event_handler(void *data)
21  {
22          mqd_t mqd;
23          char buffer[8192];
24          int n;
25          unsigned int prio;
26          const char * path = "/Liblog_to_Console";
27          printf ("PATH = %s \n",path);
28          mqd = mq_open(path, O_RDONLY | O_CREAT, 0666, NULL);
29          if (mqd == -1)
30          {
31                  perror(path);
32                  printf("ERREUR\n");
33                  exit(2);
34          }
35          while (1)
36          {
37                  n = mq_receive(mqd, buffer, 8192, & prio);
38                  if (n > 0)
39                  {
40                          buffer[n] = '\0';
41                          fprintf(stderr,"[%d]%s\n", prio, buffer);
42                  }
43          }
44
45    printf("inotify_event_handler exit");
46
47    return NULL;
48  }
49
50
51  int main(int argc, char * argv[])
52  {
53          pthread_t threadId;
54
55          pthread_create(&threadId, NULL, &inotify_event_handler, NULL);
56          fprintf(stderr,"Pthread create\n");
57
58  while (1);
59
60   return 0;
61  }
```

Annexe 2
Code source du test unitaire *Test_dconsole_Mmap_get*

```c
#include "../../message_IPC.h"
#include "../../info_console.h"



struct log_backend_info * tableau;

void affichage ()
{
int i;

        for (i=0; i<128;i++){
                if (tableau[i].pid != 0)
                {

                        fprintf(stderr,"PID : %d Logtrace_init, Name is %s and trace processus is %s \n",
                                tableau[i].pid , tableau[i].name, (!tableau[i].on_off)?"ON":"OFF" );
                }
        }
}


int main(int argc, char * argv[])
{
        int fd;

        fd = shm_open("/liblog_info", O_RDWR, 0600);
        tableau = mmap(NULL, 128 * sizeof(*tableau), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (tableau == MAP_FAILED) {
                fprintf(stderr,"erreur mmap\n");
                perror("mmap");
                exit(1);
        }

        affichage ();


 return 0;
}
```

Annexe 3
Code source du test unitaire *Test_console_Qm_get*

```
19  #include "../../message_IPC.h"
20  char * path = "/Console_to_Liblog";
21  int pid ;
22
23  static void * inotify_event_handler(void *data)
24  {
25          mqd_t mqd;
26          char buffer[8192];
27          int n;
28          unsigned int prio;
29
30          asprintf(&path, "%s_%d", path, pid);
31          printf ("PATH = %s \n",path);
32          mqd = mq_open(path, O_RDONLY | O_CREAT, 0666, NULL);
33          if (mqd == -1)
34          {
35                  perror(path);
36                  printf("ERREUR\n");
37                  exit(2);
38          }
39
40          while (1)
41          {
42                  n = mq_receive(mqd, buffer, 8192, & prio);
43                  if (n > 0)
44                  {
45                          buffer[n] = '\0';
46                          fprintf(stderr,"[%d]%s\n", prio, buffer);
47                  }
48          }
49
50    printf("inotify_event_handler exit");
51
52    return NULL;
53  }
54
55
56  int main(int argc, char * argv[])
57  {
58          pthread_t threadId;
59
60          if (argc<2)
61                  exit(-1);
62
63          pid = atoi(argv[1]);
64
65          pthread_create(&threadId, NULL, &inotify_event_handler, NULL);
66          fprintf(stderr,"Pthread create\n");
67
68  while (1);
69
70   return 0;
71  }
```

Liste des figures

Liste des tableaux

RESUME

Après deux ans, le management a constaté que les techniques pour tracer le comportement de notre middleware et le débuguer étaient très disparates dans le service. Il a fallu mettre en place une solution de traces et d'enregistrement, uniforme et commune à tous les développeurs.

Après avoir analysé les différents outils de traces existants dans le monde de l'open source Linux, j'ai étudié les différentes pratiques au sein du service. Puis j'ai réalisé une analyse des nouveaux besoins et fait évoluer le système de traces existant.

Mon étude sur les outils du monde Linux m'a montré qu'il existait une multitude de solutions. Mais elles n'étaient pas toujours bien documentées ni adaptées à notre besoin.

L'analyse des pratiques a révélé l'utilisation de nombreux outils, alors même qu'un système traces avait été développé en interne au début du projet. Mais malheureusement le manque de communication sur celui-ci a fait qu'il était peu utilisé.

Je suis reparti de cette solution et j'ai adapté son architecture pour répondre au nouveaux besoins. J'y ai intégré l'outil de profilage LTTng qui permet de synchroniser nos traces avec l'activité du noyau.

Mots clés : Trace, Profilage, Journalisation, Linux, Middleware, Débogage, LTTng

SUMMARY

After two years, service management found technique to trace behavior and debugging our middleware very disparate. They also decided to have a same trace and logging solution for all programmers.

First, I analyzed different current trace tools in Linux open source library. Then I studied all our internal techniques performing trace and debug. Finally I made a new analysis of our needs and updated the actual trace system to be compliant with the requirements.

Results of my study on current trace tools in Linux world show that there is lot of solutions available. But generally, there is no documentation or there is no answer that fulfills the requirements.

Inside our labs, programmers usually run different tools. Although there is an official in-house developed trace tool . There is few use due to lack of communication and maintenance.

I have reused and adapted this tool to the software architecture according to the new requirements, and then integrated LTTng profiling tool in order to synchronize our trace with the Linux core activity.

Key words : Trace, Profiling, Log, Linux, Middleware, Debug, LTTng