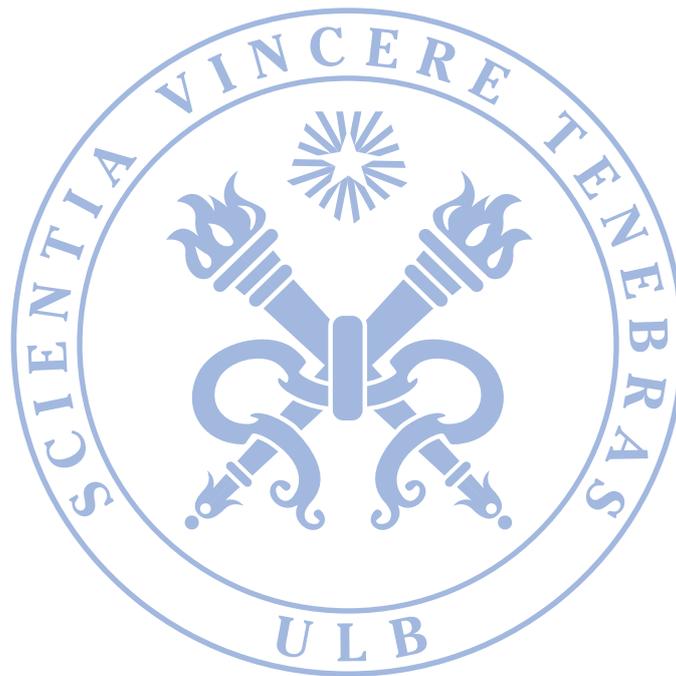


www.Mcours.com

Site N°1 des Cours et Exercices Email: contact@mcours.com

Sécurité dans les systèmes temps réel

Thomas VANDERLINDEN



Mémoire présenté sous la direction des **Professeurs Joël Goossens et Olivier Markowitch**
en vue de l'obtention du grade de Licencié en Informatique
Année académique 2006–2007

Remerciements

Un mémoire est un travail de recherche ardu et long mais intéressant. Malgré que ce soit un travail en solitaire, il aurait été difficile de me passer de l'appui de nombreuses personnes d'un point de vue académique et moral, à qui je tiens à donner toute ma reconnaissance.

Tout d'abord, je souhaiterais remettre mes premiers remerciements au professeur qui m'a suivi toute cette année en organisant des réunions régulièrement, Monsieur Joël Goossens, promoteur de ce mémoire.

Je tiens également à remercier Monsieur Olivier Markowitch, codirecteur de ce travail, qui s'est lui aussi joint à certaines de ces réunions, et sur qui je pouvais compter pour me fournir des conseils dans le domaine de la cryptographie lorsque j'en avais besoin ainsi que Monsieur Bernard Fortz pour m'avoir suggéré certaines améliorations lors de la programmation des méthodes heuristiques présentées dans ce mémoire et le professeur Hakan Aydin pour avoir répondu à mes questions sur l'ordonnancement temps réel par courrier électronique.

Je tiens à donner toute ma gratitude à mon grand-père qui a pris la peine de relire ce travail pour apporter quelques corrections orthographiques. Sur le plan moral, je tiens à remercier mes parents, toute ma famille et mes amis, ils comptent énormément pour moi, ils ne m'ont jamais abandonné durant mes études et m'ont toujours encouragé durant l'élaboration de ce mémoire.

Soyez tous sincèrement remerciés,

Thomas.

Table des matières

1	Introduction	5
1.1	Le problème	5
1.2	Organisation du mémoire	6
2	Pré-requis	7
2.1	Systèmes temps réel	7
2.1.1	Introduction	7
2.1.2	Les travaux	7
2.1.3	Les tâches	8
2.1.4	Le système	9
2.1.5	L'ordonnancement	10
2.2	Cryptologie	13
2.2.1	Introduction	13
2.2.2	Confidentialité	14
2.2.3	Intégrité et authentification	15
3	Etat de l'art	16
3.1	Introduction	16
3.2	Qualité de service et longueur des clés	17
3.2.1	Introduction	17
3.2.2	Modélisation du système	18
3.2.3	L'algorithme d'ajustement de la qualité de service	19
3.2.4	Conclusion	22
3.3	Ordonnancement dynamique basé sur des niveaux de sécurité	23
3.3.1	Conclusion	25
3.4	Méthode SASES	25
3.4.1	Modélisation	26
3.4.2	Différents services	28
3.4.3	Calcul du bénéfice au système	34
3.4.4	L'algorithme SASES	35
3.4.5	Conclusion	36

3.5	Méthode en graphe de services	37
3.5.1	Modélisation	37
3.5.2	Recherche dans le graphe	38
3.5.3	Conclusion : Comparaison avec SASES	40
3.6	Conclusion	41
4	Récompense et sécurité	42
4.1	Introduction	42
4.2	Reward-Based Scheduling	43
4.2.1	Introduction	43
4.2.2	Une fonction de récompense	44
4.2.3	Modélisation du système	47
4.2.4	Calcul de la récompense moyenne du système	47
4.2.5	Approche Mandatory-First (parties obligatoires d'abord)	48
4.2.6	Optimalité	53
4.2.7	Optimalité avec des fonctions linéaires de récompense	57
4.2.8	Optimalité avec des fonctions convexes de récompense	60
4.3	Conclusion	62
5	Simulations	64
5.1	Introduction	64
5.2	Génération d'un système	65
5.2.1	Génération des périodes	65
5.2.2	Génération de l'ensemble de tâches	66
5.3	L'algorithme orienté évènement	68
5.3.1	La gestion des évènements	69
5.3.2	L'algorithme	70
5.4	Rappel du problème d'optimisation des temps optionnels	72
5.5	Notations utilisées	73
5.6	Le recuit simulé	74
5.7	La recherche tabou	77
5.8	Obtention d'une solution voisine	79
5.9	Expérimentation	81
5.9.1	Limites des algorithmes	81
5.9.2	Analyse avec des fonctions linéaires de récompense	83
5.9.3	Simulation avec des fonctions convexes de récompense	83
5.10	Conclusion	87
6	Conclusion	89
A	Complément à la démonstration de H. Aydin	93

B	Code source de la simulation	95
B.1	Système	95
	B.1.1 Systeme.h	95
	B.1.2 Systeme.cpp	97
B.2	Tâche	107
	B.2.1 Tache.h	107
	B.2.2 Tache.cpp	108
B.3	Travail	109
	B.3.1 Travail.h	109
	B.3.2 Travail.cpp	110
B.4	Générateur	111
	B.4.1 Generateur.h	111
	B.4.2 Generateur.cpp	111
B.5	Divers	113
	B.5.1 Divers.h	113
	B.5.2 Divers.cpp	113
B.6	Event	114
	B.6.1 Event.h	114
	B.6.2 Event.cpp	115
B.7	Solution	115
	B.7.1 Solution.h	116
	B.7.2 Solution.cpp	117
B.8	Main	122
	B.8.1 Main.cpp	123

Chapitre 1

Introduction

1.1 Le problème

Les systèmes temps réel prennent une place de plus en plus importante dans notre société, ils servent à contrôler, réguler en « temps réel » des dispositifs électroniques grâce à des capteurs, embarqués dans des robots, des véhicules spatiaux, etc. Ces systèmes temps réel embarqués sont souvent utilisés par le public dans la vie de tous les jours sans même qu'on ne s'en rende compte, par exemple dans les systèmes de freinage d'une voiture, le contrôle de vol d'un avion,...

Pour contrôler des systèmes critiques comme des centrales nucléaires, des réseaux électriques, etc. . . , le réseau public Internet est de plus en plus utilisé car cela a deux avantages majeurs :

- la maintenance du système est facilitée, on peut se connecter au système à partir de tout endroit où une connexion Internet est disponible ;
- les coûts sont réduits étant donné qu'il ne faut pas mettre en place un réseau privé.

Mais il en découle également comme inconvénient l'accroissement des besoins en sécurité. Le problème qui se pose est donc d'arriver à maximiser la sécurité des échanges d'informations sans que cela nuise au respect des échéances des tâches du système temps réel. Nous voulons dire par là qu'il faut dans les limites du possible, qu'un système soit toujours ordonnançable avec la sécurité mise en place afin de garantir un fonctionnement sûr du système. En effet, sécuriser au maximum le système pourrait nous amener à un système qui dépasse sa capacité de traitement. Il faut donc limiter la sécurité de chaque tâche

selon son importance par rapport aux autres afin de ne rater aucune échéance. Ce mémoire fournit des solutions pour y parvenir.

Nous nous limiterons à étudier dans ce mémoire les systèmes « hard real-time », c.-à-d. que les échéances devront toutes être respectées malgré l'augmentation du temps d'exécution des différents travaux introduite par la mise en place de la sécurité dans le système en posant comme hypothèse que les tâches sont périodiques.

1.2 Organisation du mémoire

Clairement, ce mémoire liera donc deux disciplines en informatique, l'*ordonnancement de systèmes temps réel* et la *sécurité de systèmes informatiques*.

Nous ferons d'abord un état des techniques utilisées dans la littérature pour résoudre ce problème. Nous nous pencherons sur les algorithmes itératifs permettant de chiffrer les données (ex : AES, RC6,...), sur l'authentification de l'information et sur le contrôle d'intégrité (ex : méthodes HMAC, CBC-MAC,...).

Nous analyserons ensuite une méthode, le « Reward Based Scheduling » fournissant une récompense en fonction du temps ajouté pour la mise en place dans notre cas de la sécurité.

Pour clôturer, nous étudierons des heuristiques qui sur base de cette méthode tentent de trouver les temps se rapprochant le plus possible des temps optimaux pour établir la sécurité tout en validant le système, c.-à-d. en vérifiant par simulation que les contraintes d'échéance soient respectées.

Chapitre 2

Pré-requis

2.1 Systèmes temps réel

2.1.1 Introduction

Nous apporterons dans cette section quelques définitions largement inspirées de [7] et notations qui pourront servir à la compréhension de la suite de ce mémoire.

Un *système temps réel* est constitué d'un ensemble de *tâches* et chacune d'elles doit effectuer une quantité de calculs. Les résultats produits par ces différentes tâches doivent être d'une certaine qualité mais aussi être fournis avant un instant donné, que nous appellerons l'*échéance* de cette tâche.

Ces systèmes sont utilisés principalement dans des applications critiques qui ne peuvent pas se permettre de fournir un résultat erroné ou incomplet au moment de l'échéance, comme pour le contrôle d'une centrale nucléaire, le contrôle aérien, ...

2.1.2 Les travaux

Toute tâche n° i (que nous noterons T_i) prise dans un ensemble de n tâches ($i \in [1, n]$) composant le système, partage la quantité de calculs qu'elle doit effectuer en un certain nombre de *travaux*. Ces travaux obtiennent le processeur à intervalle régulier (périodique) ou non. Chaque *travail* (aussi appelé *instance*

ou *job* en anglais) obtient le processeur à un instant donné et peut s'exécuter sur une certaine durée.

Définition Travail : Un travail τ_{ij} , soit le j^{e} travail de la tâche T_i sera caractérisé par le tuple (a, e, d) . Un instant d'arrivée a_{ij} , un temps d'exécution e_{ij} et une échéance d_{ij} . Le travail τ_{ij} devra donc recevoir e_{ij} unités d'exécution sur l'intervalle $[a_{ij}, d_{ij})$

Il existe différentes classes de niveaux de contraintes temporelles, on parle d'échéances *strictes* (hard), *fermes* (firm) ou *souples* (soft) [1].

- Un système temps réel est dit à échéances strictes (hard deadlines) lorsqu' on ne peut rater aucune échéance, les conséquences en seraient catastrophiques ;
- Pour les échéances fermes (firm deadlines), les conséquences sont moins sévères, mais la valeur de l'exécution d'un travail après son échéance est nulle et donc d'aucune utilité.
- Un travail possédant une échéance souple (soft deadline) peut terminer son exécution après son échéance, son exécution après l'échéance a toujours une certaine valeur, bien que celle-ci décroisse avec le temps suivant l'échéance.

2.1.3 Les tâches

Classiquement dans un système temps réel, les calculs (l'exécution de ces travaux) sont récurrents, les tâches peuvent être *périodiques* ou *sporadiques*. Nous nous focaliserons dans ce mémoire sur les systèmes à tâches périodiques.

Définition Tâche périodique : Une tâche périodique T_i est caractérisée par le tuple (A_i, P_i, D_i, C_i)

- A_i est la date d'arrivée du premier travail de la tâche T_i , ce qui ne signifie pas que celui-ci s'exécutera à cette date.
- P_i est la période, c.-à-d. la durée qui sépare deux arrivées successives de travaux. Le deuxième travail d'une tâche T_i ne pourra donc pas s'exécuter avant l'instant $A_i + P_i$.
- D_i est l'échéance de la tâche T_i , qui dénote la limite supérieure de temps entre l'arrivée d'un travail et la fin de son exécution.

- C_i est le temps d'exécution de chaque travail de la tâche T_i , nous parlons aussi de $WCET_i$ (Worst Case Execution Time) pour dénoter la limite supérieure du temps d'exécution de chaque travail de la tâche T_i .

M_i, O_i : Dans la partie sur le « Reward-based Scheduling », ces notations dénotent respectivement la partie obligatoire (Mandatory) et optionnelle (Optional) d'une tâche T_i . Les notations m_i et o_i , dénotent respectivement la limite supérieure du temps d'exécution de la partie obligatoire M_i et la limite supérieure sur le temps d'exécution de la partie optionnelle O_i . Nous y reviendrons par la suite.

Chaque tâche génère un travail à chaque instant $A_i + k \cdot P_i$ avec une échéance à l'instant $A_i + k \cdot P_i + D_i$ pour tout entier $k \geq 0$.

Les tâches *sporadiques* sont similaires aux tâches périodiques à l'exception faite que P_i correspond à la durée minimum qui sépare deux arrivées de travaux de la tâche T_i .

Mais il existe également des tâches non récurrentes dites *apériodiques*, elles sont utilisées en général pour traiter des alarmes et des états d'exception, l'instant de réveil n'est pas connu au départ et il n'y a donc pas de période.

Chaque tâche a une certaine proportion d'exécution par rapport à sa période, nous parlerons d'*utilisation* que l'on définit mathématiquement comme : $U(T_i) = C_i/P_i$. L'utilisation d'un système S est la somme des utilisations des tâches qu'il contient, autrement dit : $U(S) = \sum_{T_i \in S} U(T_i)$.

Dans la partie sur le « Reward-based Scheduling », nous parlerons d'utilisation par les parties obligatoires d'un système composé de n tâches. Nous noterons ce facteur d'utilisation U_m que nous définissons par $U_m = \sum_{i=1}^n m_i/P_i$, c.-à-d. le rapport sur le temps d'exécution des parties obligatoires par rapport au temps total disponible. U quand à lui, désignera le facteur d'utilisation total (parties obligatoires et optionnelles confondues).

2.1.4 Le système

Il existe plusieurs expressions qualifiant un système constitué de n tâches, on parle de systèmes :

- à *échéance contrainte* : signifie que l'échéance de toute tâche du système ne peut pas être supérieure à la période de celle-ci ($D_i \leq P_i \forall i \in [1, n]$).

- à *échéance sur requête* : signifie que l'échéance de toute tâche du système coïncide avec la période de celle-ci ($D_i = P_i \forall i \in [1, n]$), chaque travail d'une tâche doit se terminer avant l'arrivée du travail suivant de cette même tâche. C'est donc un cas particulier de système à échéance contrainte.
- à *échéance arbitraire* : aucune contrainte n'est imposée entre l'échéance et la période.
- à *départ simultané* : l'instant d'arrivée du premier travail de chaque tâche coïncide avec les autres, on peut donc poser $A_i = 0 \forall i \in [1, n]$ sans nuire à la généralité.

Nous nous intéresserons plus particulièrement dans nos analyses aux systèmes à échéance sur requête et à départ simultané.

Le processeur

Il existe deux types d'environnement possibles pour un même système de tâches :

- *monoprocesseur* ;
- *multiprocesseur*.

Un environnement multiprocesseur constitué de k processeurs peut exécuter k unités de temps d'exécution simultanément contrairement à un environnement monoprocesseur qui est limité à une exécution à la fois. Nous resterons dans le cadre de ce mémoire sur des environnements monoprocesseur.

Définition *oisif* : Nous reprenons la traduction de la terminologie anglaise : « idle » (*inoccupé*) en parlant du processeur ou d'instant pour dire de ceux-ci qu'ils sont inutilisés, autrement dit qu'il n'y a pas de travail en cours d'exécution.

Nous parlerons également d'unités de temps oisives, pour désigner les moments où le processeur n'exécute aucun travail.

2.1.5 L'ordonnancement

L'*ordonnancement* est le mécanisme permettant de choisir la tâche qui va être exécutée par le processeur à un instant donné. L'algorithme qui va effectuer ce choix est appelé l'*ordonnanceur* (le *scheduler*).

Il existe deux manières d'appeler cet ordonnanceur :

- appels à intervalle régulier (par exemple à chaque unité de temps) ;
- appels basés sur des événements comme l'arrivée, la fin d'exécution ou l'échéance d'un travail.

Nous expliquerons par la suite pourquoi nous avons préféré utiliser une méthode basée sur des événements dans nos simulations.

A chaque fois qu'il est appelé, l'ordonnanceur va choisir le travail qui doit être exécuté au moment de l'appel si celui-ci existe bien entendu, car il est probable qu'aucun travail ne soit disponible à cet instant, le processeur restera alors oisif. Un ordonnancement dit « préemptif » signifie qu'une tâche peut être interrompue par une autre plus prioritaire. Si le processus choisi, c.-à-d. le programme en cours d'exécution, est différent de celui exécuté actuellement, l'ordonnanceur effectue alors *un changement de contexte* qui consiste en une sauvegarde des données de la tâche actuelle et une restauration des données de la nouvelle tâche si cela est nécessaire, on appelle ce changement de contexte une « *préemption* » et nous dirons du travail interrompu qu'il est « *préempté* ».

Dans un ordonnanceur temps réel, si le système arrive à ordonner toutes ses tâches en respectant leurs échéances, nous dirons de celui-ci qu'il est *ordonnançable*.

L'ordonnanceur va donc devoir assigner des priorités aux différentes tâches ou aux travaux de ces tâches. Il existe donc deux types d'assignations des priorités :

1. *priorité fixe* : également appelé assignation à *priorité statique*, chaque tâche reçoit une priorité par rapport aux autres à l'initialisation du système et chaque travail lancé hérite de la priorité de sa tâche ;
2. *priorité dynamique* : la priorité est donnée aux travaux des tâches, non plus aux tâches. La priorité peut passer à un travail d'une autre tâche pendant l'exécution du système, selon la règle d'assignation utilisée, qui peut se baser par exemple sur le temps restant avant la prochaine échéance ou le temps d'exécution déjà écoulé pour ce travail.

Définition *faisable - ordonnançable*[18] : On dit d'un ordonnancement qu'il est *faisable* si chaque tâche se termine avant son échéance dans cet ordonnancement. Un système est *ordonnançable* s'il existe un ordonnancement *faisable* pour ce système.

Définition *optimalité* : Une règle d'assignation de priorité *R* désigne la façon dont la priorité de sélectionner un travail par rapport à un autre est définie. On dit que *R* est *optimale* si, lorsqu'un système est *faisable*, il est *ordonnançable* étant donné la règle *R*.

Pour tester l'ordonnançabilité d'un système en posant l'hypothèse que celui-ci est constitué de n tâches à échéance contrainte et à départ simultané, il suffit de tester celui-ci jusqu'à l'instant P que nous appellerons l'*hyper-période* et qui est égale au plus petit commun multiple des périodes des différentes tâches composant le système.

$$P = \text{ppcm} \{P_1, P_2, \dots, P_n\}$$

En effet, à cet instant et sous ces hypothèses, nous nous retrouvons dans la même position qu'à l'instant initial : toutes les tâches envoient un nouveau travail.

« Earliest Deadline First »

Nous allons présenter maintenant la règle d'assignation que nous utiliserons dans notre étude, EDF (« Earliest Deadline First », plus proche échéance d'abord). EDF est une règle d'assignation à priorité dynamique. Lorsqu'un événement survient : comme la fin de l'exécution d'un travail ou l'arrivée d'un nouveau travail dans le système, l'ordonnanceur va sélectionner parmi tous les travaux prêts à être exécutés celui dont l'échéance est la plus proche. Ce travail sera alors exécuté sur le processeur. Autrement dit, plus le travail doit être exécuté rapidement, plus il aura de chance d'être exécuté.

Cette règle d'assignation est optimale, car EDF trouve toujours un ordonnancement faisable s'il en existe un. Pour un système S à échéance sur requête, EDF a une borne supérieure de 100% sur l'utilisation, cela signifie que si $U(S) \leq 1$ alors il est garanti qu'EDF trouvera un ordonnancement faisable contrairement à une règle d'assignation à priorité fixe comme RM (« Rate Monotonic ») qui donne une priorité plus élevée aux tâches ayant les périodes les plus courtes. La borne sur l'utilisation pour laquelle il est garanti de trouver un ordonnancement faisable n'est que de $\ln 2 \cong 69,3\%$. Cette borne peut-être augmentée à 83% et 78% si nous nous limitons respectivement à 2 ou 3 tâches, d'après la démonstration développée dans [13], mais reste toutefois inférieure à EDF.

Donnons maintenant un exemple de système afin d'illustrer le principe du fonctionnement d'EDF.

Prenons un système à échéance sur requête composé de 2 tâches $T_1(P_1 = D_1 = 5, C_1 = 3)$ et $T_2(P_2 = D_2 = 3, C_2 = 1)$, voici comment EDF va ordonnancer les différents travaux de ces tâches dont le schéma de l'ordonnancement est représenté sur la figure 2.1.

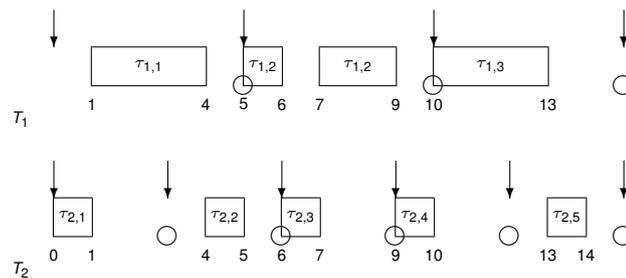


FIG. 2.1 – Exemple d'ordonnancement de 2 tâches avec la règle d'assignation EDF

Les deux tâches sont à départ simultané et envoient chacune leur premier travail à l'instant 0 (autrement dit $O_1 = O_2 = 0$). EDF sélectionne le travail $\tau_{2,1}$ pour être exécuté car l'échéance $d_{2,1} < d_{1,1}$. A l'instant 1, $\tau_{1,1}$ se termine et EDF doit choisir à nouveau un travail, il ne reste que $\tau_{1,1}$ qui continue son exécution à l'instant 3 lors de l'arrivée d'un nouveau travail de T_2 car son échéance est plus proche. Nous remarquons, qu'à l'instant 6, $\tau_{1,2}$ est préempté lors de l'arrivée de $\tau_{2,3}$ car celui-ci possède une échéance $d_{2,3} < d_{1,2}$.

Nous stoppons notre exemple à l'hyperpériode $P = 15 = \text{ppcm}(5, 3)$ car à cet instant nous nous retrouvons exactement dans la même situation qu'à l'instant 0 et l'ordonnancement peut recommencer de la même manière indéfiniment.

Lorsque les échéances sont égales, le choix entre les deux travaux se fait arbitrairement en donnant priorité au travail en cours d'exécution, comme à l'instant 12 dans notre exemple où $\tau_{1,3}$ poursuit son exécution.

2.2 Cryptologie

2.2.1 Introduction

La cryptologie est la science du secret, elle se divise en deux branches :

La cryptographie : qui étudie les différentes possibilités de cacher, protéger ou contrôler l'authenticité d'une information ;

La cryptanalyse : qui étudie les moyens de retrouver cette information à partir du texte chiffré (de l'information cachée) sans connaître les clés ayant servi à protéger celle-ci, c'est en quelque sorte l'analyse des méthodes cryptographiques.

La cryptographie doit garantir certains principes qualifiant la bonne sécurité d'un système :

1. **Confidentialité** : grâce à un chiffrement pour que les données soient illisibles par une personne tierce et ainsi garantir que l'information est restée secrète de bout en bout.
2. **Authentification** : afin que personne ne puisse se faire passer pour la source de l'information, que la provenance des données soit garantie.
3. **Intégrité** : pour que les données ne puissent pas être modifiées sans qu'on ne s'en rende compte.

2.2.2 Confidentialité

La confidentialité est historiquement le premier but des études en cryptographie : rendre secrètes des informations.

Cela se réalise par un *chiffrement* mathématique des données qui utilise comme paramètre une *clé*. Le chiffrement consiste à appliquer une suite d'opérations sur un *texte clair*, pour obtenir un texte chiffré, aussi appelé *cryptogramme*, ne pouvant être déchiffré que par l'entité qui possède la clé adéquate.

Il existe deux catégories de chiffrements :

Le chiffrement symétrique également appelé le chiffrement à clé secrète :

Le principe est de chiffrer le texte clair avec une clé et de le déchiffrer avec la même clé ou une clé dérivée de celle-ci. La clé n'est connue que par les deux entités s'échangeant des informations.

Le chiffrement asymétrique également appelé le chiffrement à clé publique :

Les clés utilisées pour le chiffrement et le déchiffrement sont différentes et ne peuvent être déduites l'une de l'autre par un observateur extérieur sans la connaissance des informations nécessaires. Une des clés peut être connue de tous tandis que l'autre doit rester secrète. Le but étant que tout le monde puisse à l'aide d'une clé publique chiffrer des données que seule l'entité possédant la clé secrète puisse déchiffrer. La vision d'un ensemble de textes chiffrés ne doit apporter aucune information sur le texte clair, c'est ce qu'on appelle la notion de *sécurité sémantique* (propre au chiffrement asymétrique).

Dans le cadre de la sécurité dans les systèmes temps réel qui doivent respecter des contraintes de temps, nous nous intéresserons plus particulièrement au chiffrement symétrique qu'il est préférable d'utiliser car le chiffrement asymétrique a été montré comme plus complexe au niveau du temps de calcul [16].

2.2.3 Intégrité et authentification

Le réseau reliant les entités n'étant pas toujours sûr, il est important de contrôler la provenance des informations et de s'assurer qu'elles n'ont pas été modifiées en chemin, ce sont respectivement les principes d'authentification et d'intégrité des données.

Afin de garantir ces deux principes, on utilise des codes d'authentification de message (MAC : Message Authentication Code). Un MAC est un code envoyé avec le message, il est aussi appelé *hachage* ou *empreinte* du message. Le hachage correspond au message et permet de garantir la validité de celui-ci. Il est obtenu à partir d'un algorithme MAC qui prend deux paramètres en entrée : le message dont on désire garantir l'intégrité et une clé secrète connue des deux entités s'échangeant ce message.

La probabilité que des données différentes possèdent le même hachage est très faible, c'est ce qu'on appelle une « collision ». La probabilité de trouver une collision et que le message possédant le même hachage soit compréhensible par le récepteur est quasi nulle. Une modification même très légère des données provoque un changement radical au niveau du hachage obtenu. Si une modification a lieu entre la source et la destination, le hachage ne correspondra plus aux données et celles-ci seront rejetées par le récepteur.

Le fait d'utiliser une clé secrète partagée entre les deux entités en plus de la fonction de hachage garantit l'authenticité des données étant donné qu'un attaquant ne connaissant pas la clé ne peut envoyer des informations accompagnées d'un hachage correct de celles-ci.

Différentes méthodes existent pour créer une empreinte du message :

- une fonction de hachage utilisant une clé en paramètre en plus du message (ex : HMAC) ;
- un chiffrement par blocs (comme les méthodes CBC-MAC).

Chapitre 3

Etat de l'art

3.1 Introduction

Comme nous l'avons mentionné dans le chapitre 1, dans les systèmes embarqués qui utilisent un réseau, nous devons à la fois respecter les contraintes d'échéances, tout en assurant la confidentialité des données, l'intégrité et l'authentification. Mais malheureusement, l'utilisation des algorithmes de cryptographie amène souvent au non-respect des contraintes d'échéances, surtout pour des systèmes embarqués qui sont limités en ressources pour des raisons de coût et de consommation.

Les travaux sur la sécurité et sur les systèmes temps réel ont été étudiés la plupart du temps séparément. Il n'existe que très peu de travaux étudiant le problème de la sécurité sous des contraintes d'échéance. En étudiant les travaux et articles rédigés à propos de la sécurité dans les systèmes temps réel, nous remarquons que ceux-ci se basent généralement sur les concepts de qualité de service (QoS), qui ont déjà été plus largement analysés dans la littérature. Le rapport de recherche [14] reprend différents mécanismes de gestion de la qualité de service dans les systèmes temps réel.

Souvent, différents « niveaux de service » sont appliqués afin de respecter au mieux les échéances tout en sécurisant le plus possible le système. Une certaine valeur est associée à chaque niveau selon son importance, l'objectif étant de maximiser la valeur totale du système.

Ce chapitre présente des méthodes que nous allons analyser et éventuellement critiquer.

3.2 Qualité de service et longueur des clés

3.2.1 Introduction

En cryptographie, il est bien connu qu'en augmentant le nombre de bits dont est constitué une clé qui servira à chiffrer les données, un attaquant mettra plus de temps à casser le chiffrement [17]. La complexité pour casser le chiffrement est donc exponentielle en fonction de la longueur de la clé.

Il est donc nécessaire d'utiliser une clé de taille suffisante. Une clé formée de ℓ bits appartient à un espace de clés constitué de 2^ℓ clés possibles. Un attaquant qui est capable de tester m clés par seconde, mettrait donc en moyenne $2^{\ell-1}/m$ secondes pour trouver la bonne clé par une attaque de type *brute-force* ($2^\ell/m$ secondes au pire des cas).

Brute-force est une recherche exhaustive sur tout l'espace des clés. Il est donc indispensable que le temps mis pour casser la sécurité par ce type d'attaque soit trop important par rapport à la valeur des données et que le système de chiffrement soit robuste, c'est à dire que le seul moyen d'attaquer le système soit d'appliquer une attaque *brute-force*.

AES, par exemple, utilise des clés de 128, 192, ou 256 bits [21] donc la quantité de clés possibles avec ces dimensions est relativement importante. De plus, AES est un algorithme de chiffrement robuste puisqu' aucune attaque sur les versions standards d'AES n'a été découverte à ce jour¹, il pourrait donc tout à fait être utilisé dans ce cadre.

Cependant, employer toujours la plus longue clé provoque une augmentation du temps de calcul utilisé par l'algorithme de chiffrement et donc il en résulte un besoin de dégrader la qualité de service. Cette diminution de la qualité de service permet de diminuer le temps d'exécution de manière à ce que les échéances soient encore toutes respectées.

Kyoung Don Kang et Sang H. Son dans leur étude [9] que nous allons présenter dans cette section, essayent de maximiser la qualité de service tout en garantissant que le système soit suffisamment sécurisé. Les auteurs, présentent leur méthode avec DES, mais ce n'est pas un algorithme de chiffrement robuste, DES a été cassé en 22 heures et 15 minutes lors du 3^e « DES Challenge » en janvier 1999 [19]. Nous préférons donc présenter cette méthode en nous basant sur l'algorithme de chiffrement AES.

¹ Il existe néanmoins des attaques sur AES en diminuant le nombre de tours à 5, 6 ou 7 [22].

3.2.2 Modélisation du système

Le modèle sur lequel nous nous basons ici est constitué d'un ou de plusieurs systèmes embarqués temps réel (SETR), par exemple des capteurs, comme représenté sur la figure 3.1. Ce SETR doit échanger ses informations avec un centre de contrôle (CC).

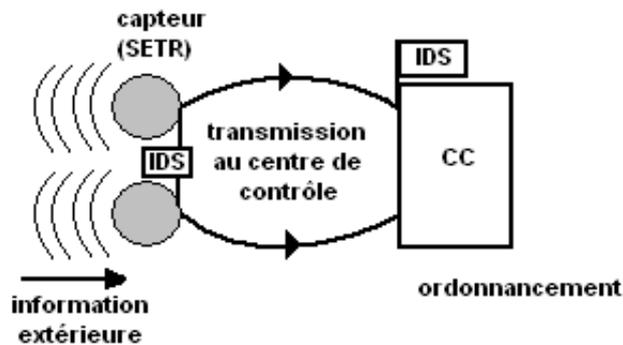


FIG. 3.1 – Schéma représentant le modèle SETR→CC

Le SETR est constitué de n tâches T_1, T_2, \dots, T_n .

Le pire temps d'exécution C_i d'une tâche T_i tient compte :

1. du temps d'exécution normal de la tâche : $C_{i,c}$
2. du temps utilisé par le chiffrement avec des clés de ℓ bits : $C_{i,e(\ell)}$

$$C_i = C_{i,c} + C_{i,e(\ell)} \tag{3.1}$$

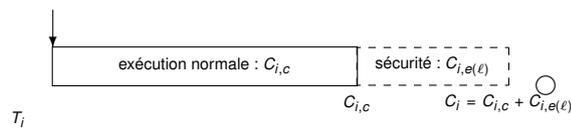


FIG. 3.2 – Représentation du temps d'exécution total C_i

La longueur de la clé ℓ est augmentée en fonction du risque encouru. Ce risque est transmis par un détecteur d'intrusion (IDS : « Intrusion Detection System ») installé sur le SETR et sur le CC. Lorsqu'une intrusion est détectée, le niveau de risque est augmenté. Le SETR va devoir alors augmenter la taille de la clé avec laquelle il chiffre ses données. Cela entraînera une diminution de la qualité de service de certaines tâches. La notion de qualité de service utilisée ici n'est pas définie dans [9] mais nous pouvons supposer qu'il s'agit de la fréquence des prises d'informations par les capteurs. Cette diminution correspond

donc à une diminution du facteur d'utilisation de ces tâches. En effet, la qualité de service étant plus faible, moins de traitements sont nécessaires et C_i est diminué. Cela permet d'éviter de rater des échéances à cause du coût en temps trop élevé dû à la mise en place de la sécurité.

Trois niveaux de risques sont définis dans [9] :
faible : $R = 1$, moyen : $R = 2$, élevé : $R = 3$.

Le niveau de défense $S(\ell)$ se calcule en fonction de la longueur de la clé ℓ . En nous basant sur les possibilités offertes par AES par rapport à la taille des clés, nous posons donc :

$$S(128) = 1, S(192) = 2, S(256) = 3.$$

Le but est de maximiser la qualité de service globale du système (QoS) qui augmente en fonction de la qualité de service $Q(i)$ des différentes tâches.

$$\text{Maximiser } QoS = S \frac{\sum_{i=1}^n Q(i)}{\sum_{i=1}^n \max Q(i)} \leq 1 \quad (3.2)$$

où $Q(i)$ et $\max Q(i)$ sont respectivement la qualité de service courante et maximum pour la tâche T_i . Nous définissons la variable S par 3.3

$$S = \begin{cases} 1 & \text{si } S(\ell) \geq R \\ 0 & \text{sinon} \end{cases} \quad (3.3)$$

Ce qui signifie (voir équation 3.2) que si le niveau de défense ($S(\ell)$) est inférieur au niveau de risque (R) envoyé par l'IDS, la qualité de service du système est nulle.

Nous calculons l'utilisation de la tâche T_i en tenant compte de $C_{i,e(\ell)}$, l'utilisation du système, ne pouvant pas (pour rappel) dépasser une certaine borne d'utilisation (B), par exemple 100% pour EDF (Earliest Deadline First) sinon le système n'est plus ordonnançable.

3.2.3 L'algorithme d'ajustement de la qualité de service

Un algorithme hors-ligne (algorithme 1) calcule premièrement la qualité de service optimale pour chaque tâche T_i à utiliser en fonction de la longueur de la clé ℓ et d'une fonction g_i .

En effet, chaque tâche T_i possède une fonction $g_i : u_i \rightarrow q_i$ qui associe à un facteur d'utilisation u_i un certain niveau de qualité de service q_i , le but de

l'algorithme étant de choisir le niveau de qualité de service adéquat en tenant compte du facteur d'utilisation imposé par la mise en place de la sécurité.

Nous allons détailler dans cette section cet algorithme repris de [9].

Description de l'algorithme

L'algorithme retourne donc un tableau $Q_List[\ell][i]$ qui représente la QoS optimale à choisir pour la tâche T_i lorsque la clé ℓ est utilisée. Les étapes qui suivent sont exécutées pour chaque longueur de clé de manière à fournir les différentes lignes $Q_List[\ell]$ du tableau.

Etape 1 : Pour chaque tâche, on calcule à l'aide de g_i et selon la longueur de la clé, l'enveloppe convexe des différentes paires $\langle Utilisation, QoS \rangle$ pour tous les niveaux discrets de QoS (voir figure 3.3). Le but étant de sélectionner les points sur g_i garantissant une qualité de service maximum avec un facteur d'utilisation minimum.

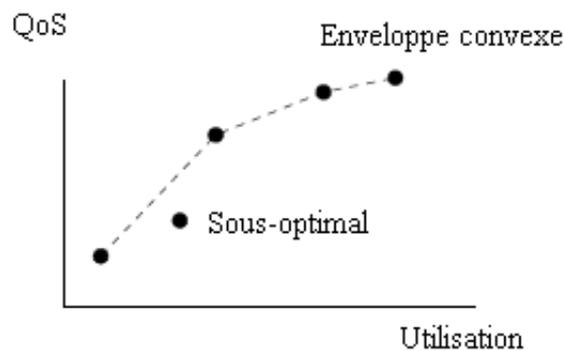


FIG. 3.3 – Exemple de fonction g_i et enveloppe convexe

Cela permet d'enlever les paires $\langle u_{ij}, q_{ij} \rangle$ qui ne sont clairement pas optimales, où j est le niveau de QoS et u_{ij} est le facteur d'utilisation requis pour la tâche T_i avec ce niveau j de QoS et avec la clé ℓ . Il nous reste alors par exemple (figure 3.3) un ensemble de paires $H_i = \{ \langle u_{i1}, q_{i1} \rangle, \langle u_{i3}, q_{i3} \rangle, \dots \}$

Etape 2 : Les ensembles $H_i \forall T_i$ sont fusionnés et triés par ordre croissant d'utilisation excepté les paires $\langle u_{i1}, q_{i1} \rangle$ qui correspondent à la QoS minimum qui devra être garantie à l'étape 5. Le résultat de ce tri/fusion est la liste H_List .

Etape 3 : On initialise $Q_List[\ell]$ à l'ensemble vide.

Entrée : $g_i : u_i \rightarrow q_i$ pour chaque tâche T_i

Sortie : $Q_List[\ell][i]$

Pour chaque clé l (128, 192, 256 bits) **faire**

1. **Pour** i de 1 à n **faire**

 Pour la tâche T_i , calculer l'enveloppe convexe

$H_i = \{ \langle u_{i1}, q_{i1} \rangle, \dots, \langle u_{ij}, q_{ij} \rangle \mid j \leq L \text{ (# niveaux de QoS)}$

Fin Pour

/*Fusion triée des enveloppes convexes de toutes les tâches par ordre non-décroissant sur l'utilisation*/

2. $H_List \leftarrow Fusion(H_1 - \{ \langle u_{11}, q_{11} \rangle \}, H_2 - \{ \langle u_{21}, q_{21} \rangle \}, \dots, H_N - \{ \langle u_{n1}, q_{n1} \rangle \})$

3. $Q_List[\ell] \leftarrow \emptyset$

4. $U^r \leftarrow B$

5. **Pour** i de 1 à n **faire**

Si ($U^r < u_{i1}$) **Alors**

FIN, ERREUR

Fin Si

$Q_List[\ell][i] \leftarrow q_{i1}$

$U^r \leftarrow U^r - u_{i1}$

$U_i \leftarrow u_{i1}$

Fin Pour

6. **Pour** i de 1 à $|H_List|$ **faire**

$id \leftarrow H_List[i].id$

$\delta U \leftarrow H_List[i].u - U_{id}$

Si ($\delta U > U^r$) **Alors FIN Fin Si**

$Q_List[\ell][id] \leftarrow H_List[i].q$

$U^r \leftarrow U^r - \delta U$

$U_{id} \leftarrow H_List[i].u$

Fin Pour

Fin Pour

Algorithme 1: Ajustement optimal de la qualité de service en fonction de la longueur de la clé

Etape 4 : U^r représente tout au long de l'exécution de l'algorithme l'utilisa-

tion disponible restante, et est initialisé à la borne d'utilisation maximale B de la politique d'ordonnancement utilisée.

Etape 5 : On va maintenant initialiser $Q_List[\ell]$ pour que le système supporte au moins le niveau minimum de QoS pour chaque tâche avec une clé de ℓ bits. Si l'utilisation restante U^r est suffisante pour garantir ce niveau de service, on fixe $Q_List[\ell][i] = q_{i1}$ et on soustrait u_{i1} à U^r , autrement l'algorithme se termine par une erreur puisqu'on dépasse la borne d'utilisation B même avec la QoS minimum.

Etape 6 : Et pour terminer, on procède à l'augmentation au maximum de la QoS de chaque tâche. Pour effectuer cela, on se base sur l'ensemble H_List obtenu à l'étape 2. On sélectionne la première tâche $T_{id} = H_List[i].id$, c.-à-d. celle qui possède la plus faible utilisation. On calcule le facteur d'utilisation supplémentaire δU nécessaire pour augmenter le niveau de QoS de T_{id} . Si celui-ci ne dépasse pas U^r , on augmente le niveau de QoS à la valeur $H_List[i].q$ et on soustrait δU à l'utilisation restante U^r . On passe à l'élément suivant dans H_List jusqu'à ce que U^r soit insuffisant pour augmenter encore la QoS d'une tâche.

La longueur des clés et le niveau de qualité de service sont ajustés en ligne à l'aide du tableau $Q_List[\ell][i]$ fourni par l'algorithme exécuté hors-ligne.

L'IDS envoie périodiquement le niveau de risque courant R , si $R > S(l)$, on passe à une clé $\ell' > \ell$ de façon à ce que $S(\ell') = R$, ensuite la qualité de service des tâches T_i est diminuée selon $Q_List[\ell'][i]$ de manière à ne rater aucune échéance.

3.2.4 Conclusion

Nous remarquons qu'un grand intérêt de cet algorithme est que la modification en ligne de la qualité de service pour ajuster le temps d'exécution se fait avec une complexité constante ($O(1)$) grâce au précalcul effectué hors-ligne par l'algorithme 1.

La complexité de l'algorithme 1 se calcule par rapport au nombre des différentes tailles de clés possibles (K , ici $K = 3$), au nombre de niveau de QoS (L) et au nombre de tâches (n). On exécute K fois le tri-fusion de $L \cdot n$ paires $\langle Utilisation, QoS \rangle$. La complexité du tri-fusion étant de $O(n \log n)$, la complexité de l'algorithme hors-ligne est donc de $O(K \cdot Ln \cdot \log Ln)$ qui peut se réduire à $O(n \log n)$ étant donné que K est constant (128, 192 ou 256 bits) tout

comme le nombre de niveaux de qualité de service (L).

Nous allons maintenant analyser la façon dont d'autres auteurs voient le problème, ne se basant plus sur la longueur des clés mais en utilisant différents niveaux de sécurité, qu'ils déterminent soit par différents algorithmes cryptographiques, soit en rajoutant différents services selon le temps disponible et l'importance des tâches.

3.3 Ordonnancement dynamique basé sur des niveaux de sécurité

Tao Xie, dans son étude [27] expose une technique qui établit certains « niveaux de sécurité ».

Les niveaux de sécurité sont définis comme étant une combinaison du transport de l'information sur le réseau, des garanties sur le chiffrement de cette dernière et de l'authentification du client.

Les niveaux choisis dans [27] qui se basent sur différents modes fournis par le protocole de sécurité SSL (« Secure Sockets Layers ») sont :

1. Routage uniquement
2. Routage + chiffrement
3. Routage + SSL
4. Routage + SSL (avec chiffrement de l'information)
5. Routage + SSL (avec authentification du client)
6. Routage + SSL (avec chiffrement de l'information + authentification du client)

Plus le niveau de sécurité augmente, plus le temps supplémentaire nécessaire pour établir la sécurité est important.

Chaque tâche possède comme paramètre, en plus de son échéance, un domaine de sécurité, c.-à-d. une borne inférieure et supérieure, qui sont les valeurs minimales et maximales admises comme niveau de sécurité de cette tâche.

A leur arrivée, les tâches sont prises en charge par un contrôleur d'admission, qui fixe le niveau de sécurité et décide si oui ou non la tâche peut être exécutée.

Le contrôleur de sécurité entre alors en jeu. Il augmente le plus possible le niveau de sécurité de chaque tâche dans la file des tâches acceptées en respectant deux contraintes, augmenter le niveau de sécurité ne doit pas :

1. empêcher la tâche de respecter son échéance ;
2. ni provoquer l'éjection d'une tâche ayant déjà été acceptée.

Il choisira de façon optimale le niveau de sécurité parmi les valeurs dans le domaine de sécurité de la tâche.

EDF ordonnancera ensuite les tâches, en prenant la tâche au sommet de la queue, c.-à-d. la tâche ayant l'échéance la plus proche. Le schéma de fonctionnement de cette méthode et les structures utilisées sont représentés sur la figure 3.4.

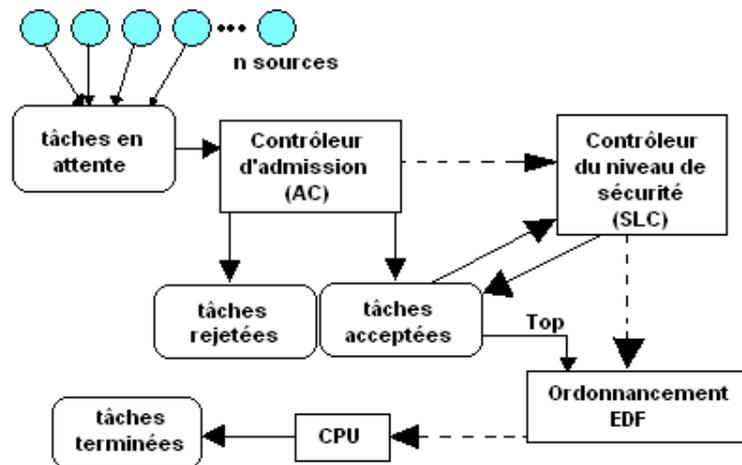


FIG. 3.4 – Schéma représentant l'architecture et les structures utilisées

La notion de « Success Ratio » (SR) est établie. SR est défini comme étant le rapport entre le nombre de tâches acceptées (N_a) par le contrôleur d'admission et le nombre de tâches reçues (N_r).

$$SR = \frac{N_a}{N_r} \quad (3.4)$$

Le but étant de maximiser SR tout en maximisant la valeur en sécurité du système, il compare 4 algorithmes utilisant tous EDF comme politique d'ordonnancement :

- EDF_MINS : le plus petit niveau de sécurité spécifié par la tâche est automatiquement sélectionné par le contrôleur d'admission. Ce qui nous per-

met donc d'obtenir la plus importante proportion de tâches respectant leur échéance (SR) mais il néglige la sécurité.

- EDF_MAXS : Le contrôleur d'admission choisit le plus haut niveau de sécurité spécifié dans les paramètres de la tâche (S_i_{max}). Ce qui garantit au contraire, un niveau de sécurité maximum au détriment du SR.
- EDF_RNDS : qui est une autre façon de voir le problème, en prenant le niveau de sécurité au hasard entre les deux bornes. Ce qui implique un SR et un niveau de sécurité global se situant entre celui de EDF_MINS et EDF_MAXS.
- EDF_OPTS : a été mis au point pour pallier le manque d'optimalité de EDF_RNDS. Il trouve un équilibre entre le niveau de sécurité et le SR. Ce dernier est de peu inférieur au SR utilisé avec EDF_MINS mais la qualité de la sécurité en est fortement améliorée.

3.3.1 Conclusion

Malheureusement, cette technique autorise certaines tâches à ne pas être acceptées à cause d'une augmentation de la sécurité sur d'autres tâches, c'est ce qu'on appelle des systèmes temps réel souples (soft real-time). Nous souhaitons plutôt nous intéresser ici à des systèmes qui tentent d'atteindre un niveau maximum de sécurité sans rejeter de tâches, c'est pourquoi nous ne rentrons pas dans le détail de l'optimisation apportée par l'algorithme EDF_OPTS, le lecteur intéressé par cette méthode peut consulter [27]. Cette méthode nous permet d'approcher la notion de niveau de sécurité que l'on retrouve dans les différents travaux analysés dans ce chapitre. La section suivante établit les niveaux de sécurité en fonction de l'algorithme utilisé, nous analyserons cette méthode et nous donnerons notre opinion sur la pertinence d'un tel choix.

3.4 Méthode SASES

Nous allons donc maintenant présenter une technique étudiée dans [26], appelée SASES (Security-Aware Scheduling for Embedded Systems).

Cet algorithme tente de fournir une bonne sécurité au système tout en tenant compte cette fois de l'importance du respect de toutes les échéances (système temps réel strict).

3.4.1 Modélisation

Nous nous concentrerons sur l'étude de tâches temps réel périodiques qui sont indépendantes l'une de l'autre et nous supposerons que les tâches sont à échéance sur requête et à départ simultané. Donc le système sera ordonnançable par une politique d'ordonnancement EDF si la contrainte $U \leq 1$ est satisfaite (voir chapitre 2).

Nous utilisons ici trois services de sécurité :

1. L'authentification ;
2. La confidentialité ;
3. L'intégrité.

nous utiliserons pour désigner ces services respectivement les notations a,c,g. Le modèle permet de rajouter d'autres services.

Les tâches

Chaque tâche est composée d'un ensemble de paramètres, $T_i = (C_i, P_i, L_i, S_i, W_i)$:

C_i : son pire temps d'exécution (worst-case execution time), sans tenir compte de la sécurité ;

P_i : sa période ;

L_i : la taille des données à protéger pour chaque travail, exprimée en Ko ;

S_i : un ensemble de bornes définissant le niveau de sécurité minimum et maximum à fournir à T_i pour les différents services de sécurité.

W_i : un vecteur $[w_i^a, w_i^c, w_i^g]$ contenant les poids des différents services k les uns par rapport aux autres. Ces poids sont attribués par l'utilisateur selon l'importance de la sécurité de ces services pour la tâche.

$$\sum_{k \in \{a,c,i\}} w_i^k = 1 \text{ et } w_i^k \in [0, 1]$$

La taille des données

Il n'est pas précisé dans [26] comment L_i est fixé mais nous supposons que l'utilisateur fournit le débit de traitement des données qu'il souhaite imposer pour chaque tâche T_i . Nous noterons par Deb_i ce débit exprimé en Ko/s. Il est

alors possible par la formule 3.5 de calculer la taille des données à traiter pour chaque travail de la tâche T_i . La quantité de données que nous noterons L_i exprimée en Ko se calcule en fonction de la période P_i exprimée en nombre de tics d'horloge du processeur.

$$L_i = Deb_i \cdot \frac{P_i}{\#tics/s} \quad (3.5)$$

Niveaux de sécurité

Chaque service possède une borne inférieure et supérieure pour le niveau de sécurité.

$$S_i = (S_i^a, S_i^c, S_i^g) = ([s_i^a_{min}, s_i^a_{max}], [s_i^c_{min}, s_i^c_{max}], [s_i^g_{min}, s_i^g_{max}]) \quad (3.6)$$

où S_i^a , S_i^c et S_i^g dénotent respectivement pour T_i les bornes pour le service d'authentification, de confidentialité et d'intégrité. Le niveau de sécurité peut s'étendre de 0 à 1. Le but de l'ordonnanceur sera donc de choisir le niveau de sécurité s_i le plus approprié dans le domaine S_i , c.-à-d. $s_i = (s_i^a, s_i^c, s_i^g)$ où $s_i^k \in S_i^k \quad \forall k \in \{a, c, g\}$.

Temps de calcul de la sécurité

Posons sec_i , le temps d'exécution utilisé pour la sécurisation d'un travail de la tâche T_i .

Nous calculons sec_i comme exprimé dans 3.7.

$$sec_i = \sum_{k \in \{a, c, g\}} sec_i^k(s_i^k) \quad (3.7)$$

C'est donc la somme des $sec_i^k(s_i^k)$, c.-à-d. la somme des temps d'exécution nécessaires pour l'application de chaque service de sécurité k en fonction du niveau de sécurité choisi par l'ordonnanceur pour chacun d'eux.

La méthode pour calculer les temps de calcul $sec_i^k(s_i^k)$ pour chaque service de sécurité est peu définie dans [26]. Nous avons donc analysé et critiqué la méthode d'un autre article [25] des mêmes auteurs.

Dans [25], chaque niveau de sécurité correspond à un algorithme cryptographique (algorithme de chiffrement, fonction de hachage ou méthode d'authentification). Selon ce même travail, plus le niveau de sécurité est important, plus le débit de traitement de l'algorithme associé à ce niveau est faible et donc le temps dépensé pour appliquer la sécurité sera plus important. Par la suite, nous expliquerons pourquoi nous ne sommes pas entièrement du même avis que l'auteur de [25].

A l'aide des tests de performance fournis dans la librairie crypto++ [4], nous avons calculé le débit de différents algorithmes sur notre processeur, Intel Pentium M Processor 735 - 1.7 GHz. Pour chaque service nous avons choisi une liste d'algorithmes de manière à répartir correctement la valeur des niveaux de sécurité en nous basant sur le choix fait par Tao Xie et Xiao Qin dans [25].

Nous allons maintenant montrer comment est calculé le temps d'exécution nécessaire pour les différents services.

3.4.2 Différents services

La confidentialité

Imaginons qu'un système embarqué temps réel, que nous noterons SETR, veuille transmettre une information (P) à un serveur, que nous noterons CC (centre de commande).

Le service de confidentialité est indispensable pour rendre les données circulant entre ce SETR et le CC inintelligibles.

Le SETR va chiffrer (voir 3.8) cette information grâce à un algorithme de chiffrement (E), en utilisant la clé de chiffrement (K_e) lui ayant été fournie par le CC pour l'algorithme E . Afin de réduire le coût en temps, nous pouvons supposer qu'il n'y a pas de négociation des clés en ligne et que l'enregistrement des clés se fait lors de la maintenance du système.

Le chiffrement est symétrique, ce qui signifie que chaque entité (source et destination) possède la même clé secrète que ce soit pour chiffrer ou déchiffrer les données. Un chiffrement symétrique est utilisé car il est plus rapide qu'un chiffrement asymétrique [16], ce qui est utile dans les systèmes temps réel pour diminuer le temps d'exécution requis par la mise en place de la sécurité.

Le cryptogramme C est alors obtenu par 3.8 :

$$C = E(P)_{\{K_e, Cpt\}} \quad (3.8)$$

Un compteur Cpt est inclus dans le message à chiffrer, il permet d'éviter une attaque par rejeu, c'est à dire une attaque où l'attaquant renvoie une copie d'un ancien message. Il est incrémenté à chaque message P envoyé par le SETR, s'il ne l'est pas, le CC va rejeter P .

Il existe plusieurs manières de chiffrer un message. Chaque algorithme de chiffrement correspond à un niveau de sécurité s_i^c et possède un certain débit de chiffrement $\mu(s_i^c)$, c.-à-d. la quantité de données que l'algorithme à ce niveau peut traiter par unité de temps. Ce débit dépend de la rapidité du processeur et de l'algorithme utilisé, autrement dit du niveau de sécurité choisi.

Selon la quantité de données L_i à traiter dans chaque travail de la tâche T_i , on peut calculer la durée du chiffrement sec_i^c pour un travail de T_i grâce à l'équation 3.9.

$$sec_i^c = \frac{L_i}{\sigma(s_i^c)} \quad (3.9)$$

où $\sigma(s_i^c)$ est une fonction qui donne le débit $\mu(s_i^c)$ de l'algorithme dont le niveau de sécurité est le plus proche et $\leq s_i^c$.

Nous avons testé onze algorithmes de chiffrement différents. En fonction des performances de ces différents algorithmes nous établissons un niveau de sécurité s_i^c pour chacun d'eux qui se situe entre 0,16 et 1, i étant le numéro de l'algorithme dans la liste 3.1.

La valeur 1 est attribuée à IDEA, l'algorithme le plus « fort » selon [25], mais possédant aussi le débit le plus faible (= 4,44 Ko/ms) et donc le temps d'exécution le plus important pour une même quantité de données. Le niveau de sécurité des autres algorithmes est déduit de celui-ci, par la formule 3.10.

$$s_i^c = \frac{4,44}{\mu_i^c}, 1 \leq i \leq 11 \quad (3.10)$$

Nous fournissons dans le tableau 3.1, la liste des algorithmes de chiffrement que nous avons obtenus grâce aux tests de performance, triés en ordre croissant par rapport au niveau de sécurité avec leurs performances respectives (le débit) et le niveau de sécurité qui en découle. Ils seront utilisés pour garantir la confidentialité des données .

Algorithmes de chiffrement	Niveau de sécurité s_l^c	Débit μ_l^c (en Ko/ms)
Rijndael (128 bits)	0,16	27,04
SEAL-3.0-BE	0,22	19,72
Salsa20	0,38	11,52
Blowfish	0,49	9,09
RC6	0,53	8,44
Twofish	0,65	6,87
DES	0,75	5,93
Camellia (256 bits)	0,87	5,11
Shacal-2 (128 bits)	0,90	4,93
IDEA	1,00	4,44

TAB. 3.1 – Confidentialité - Niveaux de sécurité et débits des algorithmes

Nous remarquons que la façon dont les algorithmes sont classés est étonnante. En effet, Xie et Qin évaluent les performances d'un algorithme cryptographique en fonction du débit de celui-ci. Mais d'après nos tests, nous observons par exemple, que le débit de l'algorithme Rijndael-128 bits (AES) est plus important que celui de DES ce qui rejoint les résultats obtenus dans [25]. AES utilisera donc moins de temps pour le chiffrement et recevra un niveau de sécurité inférieur.

Or, la valeur en sécurité d'un algorithme ne dépend pas uniquement du débit de celui-ci mais de sa complexité et de la longueur des clés utilisées (comme nous l'avons mentionné à la section 3.2.1). Le chiffrement DES, à cause de la faible longueur de la clé (56 bits), a été cassé en 1999 en moins d'un jour. Rijndael quant à lui, utilise des clés de minimum 128 bits, ce qui est usuellement admis comme étant la limite inférieure acceptable pour la taille des clés d'un algorithme cryptographique permettant d'atteindre une sécurité satisfaisante (du moins jusqu'à ce jour) [2].

Il n'est donc pas logique de poser pour l'algorithme DES, un niveau de sécurité supérieur à AES. Lorsqu'on dispose de plus de temps lors de l'ordonnement, on passerait à un algorithme moins performant aussi bien en terme de débit que de résistance aux attaques.

Nous allons néanmoins continuer à expliquer l'idée exposée dans cette étude. Elle pourrait être intéressante à appliquer en définissant autrement les niveaux de sécurité. Il est possible de définir le niveau de sécurité par rapport à la longueur des clés en utilisant le même algorithme pour les différents niveaux de sécurité, ce qui réduirait la quantité de ressources nécessaires pour implémenter ces différents algorithmes dans le système embarqué. Citons par exemple,

l'idée exposée dans [10], qui est d'utiliser les variantes 128, 192, 256 bits d'AES.

L'intégrité

Le service d'intégrité sert à garantir que les données n'ont pas été modifiées entre le SETR et le CC.

Le temps d'exécution sec_i^g nécessaire pour ajouter le contrôle d'intégrité au temps normal d'exécution est calculé dans [25] à partir d'un tableau fournissant les débits de différentes fonctions de hachage de la même manière que pour le service de confidentialité (par l'équation 3.11).

$$sec_i^g = \frac{L_i}{\sigma(s_i^g)} \quad (3.11)$$

Nous avons calculé le débit de neuf fonctions de hachage différentes (voir tableau 3.2) et avons ensuite établi un niveau de sécurité pour chacune de celles-ci par rapport à leurs performances dans la mesure où plus le hachage est long, plus la sécurité est renforcée mais le débit de traitement est plus faible.

Le calcul du niveau de sécurité des différentes fonctions s'effectue de manière analogue au service de confidentialité, nous comparons grâce à la formule 3.12 les différentes fonctions de hachage par rapport aux performances de SHA-256 (possédant le plus faible débit : 8,98 Ko/ms) dont le niveau de sécurité est fixé à 1. Le niveau de sécurité s_i^g varie entre 0,05 et 1.

$$s_i^g = \frac{8,98}{\mu_i^g}, 1 \leq i \leq 9 \quad (3.12)$$

A nouveau, l'auteur suppose qu'une fonction de hachage dont le temps de calcul est plus important (plus faible débit), sera de meilleure qualité, ce qui est probable mais n'est pas prouvé. Nous n'avons pas trouvé de contre-exemple comme nous l'avons fait avec AES par rapport à DES dans le cadre de la confidentialité.

Remarque: On ne peut garantir l'intégrité des données en appliquant uniquement sur celles-ci une simple fonction de hachage à sens unique sans clé comme une des fonctions du tableau 3.2. Il est important de chiffrer le message dont on désire garantir l'intégrité avec une clé secrète connue uniquement par le

Fonctions de hachage	Niveau de sécurité s_i^g	Débit μ_i^g (en Ko/ms)
Adler-32	0,05	170,00
CRC-32	0,08	117,98
MD5	0,15	58,99
RIPE-MD128	0,35	25,36
Tiger	0,39	22,69
SHA-1	0,43	20,71
RIPE-MD160	0,56	16,05
HAVAL-5	0,63	14,36
SHA-256	1,00	8,98

TAB. 3.2 – Intégrité - Niveaux de sécurité et débits des fonctions de hachage

SETR et le CC, car autrement un attaquant pourrait modifier le message et hacher à nouveau celui-ci. Etant donné que le message correspondra au résultat de la fonction de hachage, il sera accepté alors que celui-ci a été modifié.

On peut supposer ici que la fonction de hachage est appliquée et ensuite chiffrée par le service de confidentialité. Mais si seule l'intégrité doit être garantie, cette méthode ne convient pas, il est préférable d'utiliser des MAC (Message Authentication Code) qui garantissent à la fois l'authentification et l'intégrité des données.

L'authentification

Il est indispensable de vérifier l'authenticité des données circulant dans le système, c.-à-d. qu'il faut garantir que c'est bien la source concernée qui envoie les données. Dans ce but, nous utilisons des codes d'authentification de message (MAC).

Le service d'authentification est divisé dans [25] en 3 classes qui utilisent chacune une méthode d'authentification différente :

1. *faible*, nous utiliserons pour ce niveau HMAC-MD5 ;
2. *acceptable*, avec HMAC-SHA-1 ;
3. *forte*, en utilisant CBC-MAC-AES.

Chaque classe a besoin d'un certain temps de calcul. Ces temps sont listés dans le tableau 3.3 issu de [25], la différence de niveau est calculée à partir de ces temps de calcul.

Méthodes d'authentification	Niveau de sécurité s_j^a	Temps de calcul μ_j^a (en ms)
HMAC-MD5	0,55	90
HMAC-SHA-1	0,91	148
CBC-MAC/AES	1,00	163

TAB. 3.3 – Authentification - Niveaux de sécurité et temps de calcul des différentes méthodes

On obtient $sec_i^a = \mu_j^a(s_i^a)$ comme étant le temps nécessaire pour garantir l'authenticité des données au niveau s_i^a (faible, acceptable, fort).

Les services d'intégrité et d'authentification sont séparés, or un MAC garantit à la fois l'intégrité et l'authentification des données. Il est donc, selon nous, inutile d'utiliser le service d'intégrité comme il est décrit dans [25]. Il serait préférable de lier ces deux services en un seul.

Voici comment appliquer un service qui garantit l'authentification et l'intégrité par une méthode de type HMAC (keyed-hash message authentication code) utilisée avec une fonction de hachage à sens unique comme SHA-1 ou MD5 :

Après avoir chiffré l'information afin d'en garantir la confidentialité, il nous faut garantir que cette information n'a pas été modifiée et provient effectivement du SETR. Dans ce but, un code d'authentification du message (MAC que nous noterons ici M) sera calculé et rajouté à la suite du message :

$$M = H(S|D|C|Cpt)_{K_m} \quad (3.13)$$

Une méthode d'authentification (H) du tableau 3.3 est appliquée avec la clé K_m sur la concaténation de l'adresse de la source (S : le SETR), de la destination (D : le CC), du message chiffré et du compteur Cpt . Nous tenons à préciser que la clé K_m doit être choisie comme différente de K_e (la clé utilisée pour le chiffrement). En effet, pour des raisons de sécurité [17], il est préférable d'utiliser des clés différentes pour l'algorithme de chiffrement et la fonction de hachage. Ensuite le SETR envoie le message crypté Msg et le résultat du hachage du message M vers le CC, ce transfert est donc constitué de :

$$\text{Message } S \rightarrow D : Msg, M \quad (3.14)$$

$$Msg = S, D, C, Cpt \quad (3.15)$$

Le code M garantit en effet l'**intégrité** des informations transmises.

Si un opposant, que nous appellerons Oscar, intercepte et modifie le message, le CC s'en rendra compte, à moins qu'Oscar n'arrive à modifier le résultat

M produit par la fonction de hachage pour que celui-ci corresponde à la modification effectuée sur Msg , ce qui voudrait dire qu'Oscar possède la clé K_m .

L'**authentification** est également garantie par la connaissance de K_m étant donné que l'adresse de la source et de la destination sont intégrées au code, Oscar ne peut modifier ces adresses sans que le code M ne soit erroné si celui-ci ne dispose pas de K_m .

Le récepteur du message, ici le CC calcule le code $M' = H(Msg)_{K_m}$, si $M' \neq M$ le CC va rejeter le message. Si $M' = M$, cela veut dire que le message n'a pas été altéré et il va donc extraire C et le déchiffrer. Notons, que le même algorithme peut être utilisé pour le chiffrement et la fonction de hachage sans affecter la sûreté du système [17], ce qui peut à nouveau être intéressant pour limiter la quantité de ressources à installer sur le système embarqué.

3.4.3 Calcul du bénéfice au système

SASES utilise une fonction pour calculer le bénéfice en sécurité accru par une tâche T_i :

$$SL_i = b_i \sum_{k \in \{a,c,i\}} w_i^k s_i^k \quad (3.16)$$

où $b_i = \frac{P}{P_i}$, c.-à-d. le nombre de travaux de cette tâche que l'on peut ordonnancer jusqu'à l'hyperpériode P , w_i^k est le poids spécifié par l'utilisateur pour le service k de la tâche T_i et s_i^k est le niveau de sécurité du k^e service qui est identique pour tous les travaux de la tâche.

L'ordonnancement est faisable si :

- les niveaux de sécurité respectent les bornes 3.18 ;
- les échéances $d_{ij} = j \cdot P_i$ sont toutes respectées. Cela est le cas si les tâches sont planifiées avec une politique d'ordonnancement EDF et que le facteur d'utilisation $U \leq 1$ (cf. équation 3.19), suivant l'hypothèse selon laquelle les tâches sont périodiques et à échéance sur requête. Cela revient à dire que la somme des rapports entre le temps d'exécution total (temps nécessaire pour la sécurisation sec_i ajouté au temps d'exécution normal C_i) et la période P_i de chaque tâche n'est pas supérieur à 100%.

Le problème peut donc s'écrire comme une maximisation d'un niveau de sécurité global (SV), étant la somme des niveaux de sécurité des n tâches formant

le système :

$$\text{maximisation de } SV = \sum_{i=1}^n SL_i = \sum_{i=1}^n b_i \sum_{k \in \{a,c,i\}} w_i^k s_i^k \quad (3.17)$$

$$\text{à condition que } \min(S_i^k) \leq s_i^k \leq \max(S_i^k) \quad (3.18)$$

$$\text{et que } U = \sum_{i=1}^n \frac{[C_i + \sum_{k \in \{a,c,i\}} \text{sec}_i^k(s_i^k)]}{P_i} \leq 1. \quad (3.19)$$

3.4.4 L'algorithme SASES

Nous allons montrer maintenant comment, à partir des caractéristiques des tâches (C_i, P_i, L_i, S_i) , le niveau de sécurité est augmenté le plus possible en tenant compte du poids w_i^k de la sécurité de chaque service de celles-ci. Nous avons repris l'idée de l'algorithme présenté dans [26] en le modifiant quelque peu pour le rendre plus compréhensible (voir algorithme 2).

Description de l'algorithme

1. Premièrement les niveaux de sécurité de tous les services pour chaque tâche sont initialisés à leurs valeurs minimales.
2. Une structure *EnsembleServices* contient tous les services qui doivent encore être traités, chaque service est représenté par le numéro de la tâche et le service de sécurité concerné ($a = \text{authentification}$, $c = \text{confidentialité}$, $g = \text{intégrité}$).
3. Si l'utilisation (définie en 3.19) est supérieure à 1 le système n'est pas ordonnançable même avec le minimum de sécurité.
4. On sélectionne le service de sécurité k' de la tâche i' qui possède le poids $w_{i'}^{k'}$ le plus important et qui consomme le moins de temps en l'augmentant d'un niveau de sécurité ($\Delta s_{i'}^{k'}$).
5. Si le fait d'augmenter le niveau de sécurité de ce service de $\Delta s_{i'}^{k'}$ ne lui fait pas dépasser son niveau maximal de sécurité et que le système reste ordonnançable avec cette augmentation ($U \leq 1$), on augmente le niveau de celui-ci, autrement on retire $s_{i'}^{k'}$ de *EnsembleServices* de façon à ce qu'il ne soit plus choisi à l'étape 4.
6. L'algorithme prend fin lorsque l'ensemble des services est vide, c.-à-d. lorsque tous les services de chaque tâche auront été traités.

```

EnsembleServices ← ∅
Pour i de 1 à n faire
  Pour chaque k ∈ {a, c, g} faire
    sik ← min {Sik}
    EnsembleServices ← EnsembleServices ∪ {i, k}
  Fin Pour
Fin Pour
Si (∑i=1n  $\frac{[C_i + \sum_{k \in \{a,c,g\}} sec_i^k(s_i^k)]}{P_i} \leq 1$ ) Alors
  Tant que (EnsembleServices ≠ ∅) faire
    • Choisir {i', k'} dans EnsembleServices tel que :
       $w_{i'}^k \Delta s_{i'}^{k'} / [sec_{i'}^{k'}(s_{i'}^{k'} + \Delta s_{i'}^{k'}) - sec_{i'}^{k'}(s_{i'}^{k'})]$ 
      = maxi ∈ [1, n], k ∈ {a, c, g} {  $w_i^k \Delta s_i^k / [sec_i^k(s_i^k + \Delta s_i^k) - sec_i^k(s_i^k)]$  }
      Si (si'k' + Δsi'k' ≤ max {Si'k'}
      ET ∑i=1n  $\frac{[C_i + \sum_{k \in \{a,c,i\}} sec_i^k(s_i^k)]}{P_i} \leq 1$  avec si'k' = si'k' + Δsi'k') Alors
        //On augmente le niveau de sécurité
        si'k' ← si'k' + Δsi'k'
      Sinon
        EnsembleServices ← EnsembleServices \ {i', k'}
        //de façon à ne plus en tenir compte dans le choix •
      Fin Si
    Fait
  Sinon
    //pas ordonnançable avec le niveau de sécurité le plus bas
  Fin Si

```

Algorithme 2: Algorithme SASES

3.4.5 Conclusion

Nous avons présenté dans cette section l'algorithme SASES permettant de sélectionner un ensemble de moyens cryptographiques pour sécuriser les données de chaque tâche tout en respectant leurs échéances. L'algorithme que nous exposerons dans la section suivante est une autre façon de choisir les services les plus appropriés de manière plus efficace que le parcours effectué par l'algorithme SASES.

3.5 Méthode en graphe de services

3.5.1 Modélisation

Man Lin et Laurence T. Yang dans leur étude [12], représentent chaque principe de sécurité par un groupe de services. Par exemple, il y aura un groupe pour les différentes méthodes d'authentification, un autre pour les algorithmes de chiffrement, . . . Ces différentes méthodes seront ici appelées « services ».

De la même manière que dans l'algorithme SASES, chaque groupe i possède un certain poids W_i par rapport aux autres, et dans chacun des Ng groupes, les services ont une certaine qualité, autrement dit une valeur qui leur est propre. Nous noterons Q_{ij} , la qualité du service j dans le groupe i .

Les services sont représentés comme étant les nœuds d'un graphe, le nœud $\langle i, j \rangle$ représentant le service j du groupe i . Bien entendu, un seul service maximum dans chaque groupe peut-être sélectionné à la fois de manière à former un ensemble de services qui seront utilisés pour sécuriser le système, nous appellerons cet ensemble un chemin. Dans la figure 3.5, un exemple de chemin $P = \{ \langle 1, 1 \rangle ; \langle 2, 2 \rangle ; \langle 3, 2 \rangle \}$ est représenté en gris.

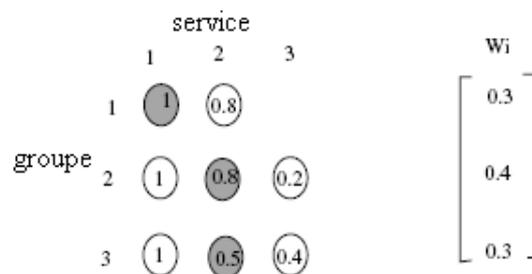


FIG. 3.5 – Représentation du chemin P

Les services sont classés par ordre décroissant par rapport à leur qualité (*par exemple* : $Q_{1,1} > Q_{1,2}$). Nous noterons sec_{ik} le temps nécessaire à la sécurisation de la tâche T_i par le service sélectionné dans le groupe k . Lorsque la qualité d'un groupe est diminuée, sec_{ik} sera moins important également, ce qui permet donc d'ajuster la qualité de service pour respecter les échéances.

La vérification du respect des échéances se fait sur base de la condition d'ordonnancement nécessaire et suffisante imposée par EDF, selon laquelle la somme des facteurs d'utilisation des n tâches dont est composé le système doit

être inférieur à 100% (voir équation 3.20).

$$\sum_{i=1}^n \frac{C_i + \sum_{k=1}^{Ng} sec_{ik}}{P_i} \leq 1 \quad (3.20)$$

Nous tenons compte dans le calcul du facteur d'utilisation, du temps d'exécution normal C_i et de la mise en place de la sécurité pour chaque groupe.

Le but est de sélectionner dans chaque groupe, le service qui permettra de maximiser la qualité globale $Q(P)$ du chemin P tout en respectant les échéances.

$$Q(P) = \sum_{i=1}^{Ng} Q_i \quad (3.21)$$

De plus, la qualité Q_i (la valeur inscrite dans le noeud) d'un groupe doit être supérieure à une borne inférieure $MinQ_i$, la qualité minimale acceptée pour chaque groupe de services.

$$Q_i > MinQ_i \quad (3.22)$$

3.5.2 Recherche dans le graphe

Il est proposé dans [12], une méthode de recherche par un parcours de graphe en profondeur (« depth-first ») qui permet de choisir les meilleurs services permettant de garantir que le système soit toujours ordonnançable en utilisant ceux-ci. La recherche est améliorée afin de ne pas devoir tester toutes les possibilités de manière exhaustive. Nous allons présenter maintenant ces améliorations du parcours du graphe.

En effet, il existe trois moyens d'élaguer le graphe afin de restreindre l'espace de recherche :

1. Premièrement, sont supprimés dans chaque groupe i , les services qui ne respectent pas la contrainte de qualité minimale acceptable, soit les noeuds $\langle i, j \rangle$ tels que $Q_{ij} < MinQ_i$ (voir figure 3.6).
2. En parcourant ensuite le graphe par un parcours en profondeur, si un chemin n'est pas faisable (voir figure 3.7[b]), c.-à-d. s'il ne respecte pas la contrainte d'ordonnançabilité 3.20, on sait que l'ordonnançabilité ne sera

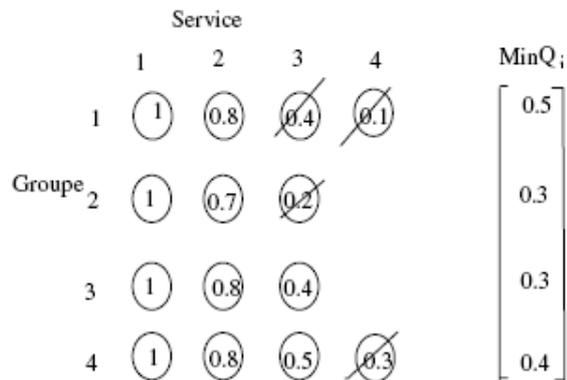


FIG. 3.6 – Elagage par rapport à la valeur minimale de qualité de service

pas non plus respectée avec un chemin plus long formé par le même préfixe, on peut donc élaguer les chemins ([d], [e] et [f]). On continue alors le parcours en profondeur comme s'il n'y avait pas de nœuds fils (voir [c]), la qualité de service du groupe courant est donc baissée, autrement dit on passe directement au nœud suivant dans le groupe, sans être descendu dans le graphe.

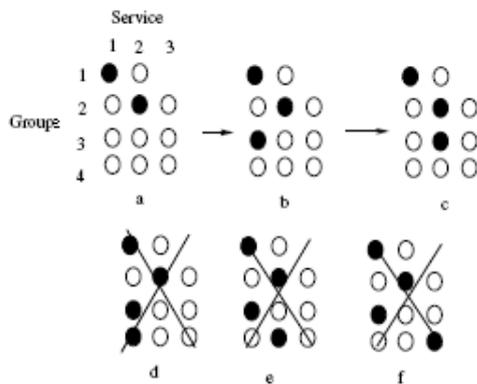


FIG. 3.7 – Elagage en fonction du test de faisabilité

3. Il est encore possible de supprimer des branches en évitant d'analyser des chemins qui ne donneront pas une meilleure qualité de service. Soit un chemin P de longueur Ng faisable, avec une qualité de service $Q(P)$ accrue par celui-ci.
 - On remarque que diminuer la qualité du dernier groupe Ng sur P ne mènera pas à une augmentation de $Q(P)$ (dans l'exemple présenté à la

figure 3.8 $Ng = 4$). On poursuit alors le parcours en profondeur en remontant au niveau (groupe) $Ng - 1$ et en diminuant la qualité de service à ce niveau (voir figure 3.8[a']) pour tester à nouveau le niveau Ng par après.

- Lorsque la qualité Q_i de tous les niveaux i entre L et Ng est au maximum et que le chemin est faisable, on peut directement remonter au niveau $L - 2$ et continuer le parcours pour essayer de trouver un meilleur chemin.

Dans la figure 3.8, sont représentés 3 exemples de chemins faisables ([a], [b] et [c]) et les chemins analysés respectivement juste après ceux-ci ([a'], [b'], [c']) dans l'ordre du parcours en profondeur du graphe.

Pour [c], la recherche se termine car il n'y a plus moyen d'augmenter la qualité de service.

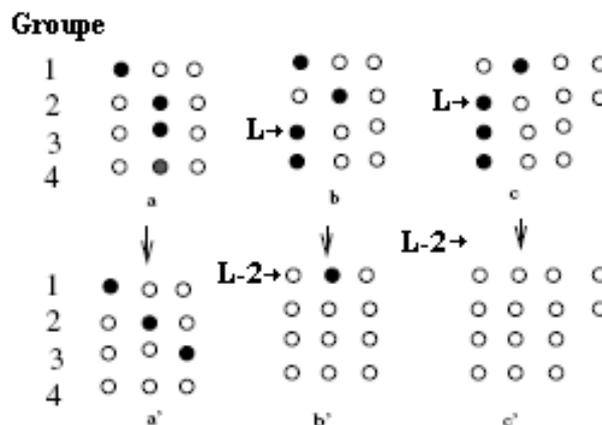


FIG. 3.8 – Elagage en fonction de la qualité de service

3.5.3 Conclusion : Comparaison avec SASES

Remarquons que contrairement à l'algorithme SASES, cette méthode attribue le même algorithme cryptographique (service) dans un groupe pour toutes les tâches. Cela rend le calcul plus rapide mais est selon nous moins performant au niveau de la valeur en sécurité du système étant donné que lorsque l'algorithme augmente la sécurité de toutes les tâches en même temps, il est probable que le système ne soit plus ordonnable et donc qu'on doive baisser le niveau de sécurité de toute les tâches de manière à ne rater aucune

échéance. Cela n'aurait sans doute pas été le cas en modifiant la qualité de service tâche par tâche. Malgré sa complexité supérieure, SASES permet une granularité plus importante au niveau de la sécurité et est donc selon nous plus approprié à ce problème d'optimisation de la sécurité.

3.6 Conclusion

Nous avons dans ce chapitre développé plusieurs méthodes existantes dans la littérature permettant de choisir le meilleur moyen de sécuriser le système, nous avons décrit les algorithmes utilisés, les points forts et les points faibles de ces différentes techniques.

Comme nous l'avons fait remarquer dans ce chapitre, le fait d'établir le niveau de sécurité par rapport au débit des algorithmes n'est pas adapté, il serait préférable de garder le même algorithme et de faire varier la taille de la clé ou le nombre de tours de l'algorithme cryptographique selon le temps disponible de manière à représenter la valeur en sécurité comme une évolution de la sécurité en fonction du temps et d'attribuer une récompense à cette évolution.

Nous nous sommes donc intéressés plus particulièrement aux travaux de H. Aydin [3] qui a développé une méthode qui attribue une récompense selon un supplément de temps d'exécution de manière à profiter au maximum du temps disponible pour exécuter une partie optionnelle qui sera dans notre cas destinée à la sécurité.

Le reste de ce mémoire sera consacré à cette méthode (chapitre 4) et à la façon dont nous avons implémenté celle-ci (chapitre 5) de manière à en évaluer les performances.

Chapitre 4

Récompense et sécurité

4.1 Introduction

Comme nous l'avons défini au chapitre 2, chaque tâche dans un système temps réel doit se terminer avant une certaine échéance en produisant le résultat attendu. Si le système n'est pas ordonnançable, il faudra alors choisir des parties qui resteront non exécutées de façon à ce que toutes les tâches respectent leur échéance. Le système peut ne plus être ordonnançable suite à une demande de précision trop fine du résultat, par exemple, lors du raffinement de la qualité d'une image ou pour citer notre cas, si le niveau de sécurité attribué aux différentes tâches est trop important.

La communauté de recherche dans ce domaine, a mis au point plusieurs techniques. Nous pouvons par exemple « sauter » [11] entièrement certains travaux d'une tâche donnée suivant un facteur de saut qui détermine combien de travaux sur un certain nombre de travaux de cette tâche peuvent être abandonnés par l'algorithme d'ordonnancement.

Mais nous supposons dans ce cas, que les données produites par une tâche qui n'a pas été exécutée entièrement ne sont d'aucune utilité. Or, dans de nombreux secteurs comme le traitement d'images, de la voix, le contrôle de robot, et bien d'autres encore, une solution partielle est souvent suffisante. Pour résoudre notre problème de sécurité, nous pourrions également utiliser de telles techniques qui peuvent nous apporter en un temps acceptable, une sécurité partielle mais suffisante.

4.2 Reward-Based Scheduling

4.2.1 Introduction

Ce chapitre est basé sur les travaux de H. Aydin [3]. Nous n'envisagerons dans cette section que le cas des tâches périodiques avec un système temps réel à échéance sur requête.

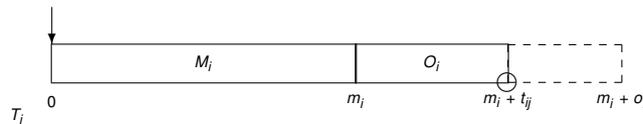


FIG. 4.1 – Partie obligatoire et optionnelle d'un travail τ_{ij} de la tâche T_i

Comme son nom l'indique, le principe de cet ordonnancement est d'associer une récompense à l'exécution d'une tâche.

Chaque tâche T_i peut être décomposée en une sous-tâche (partie) obligatoire (M_i) et une sous-tâche optionnelle (O_i) (voir figure 4.1).

La longueur de la partie obligatoire de chaque travail de T_i sera notée m_i et o_i désignera la longueur maximum que peut prendre la partie optionnelle d'un travail de T_i . Il est primordial que la partie obligatoire se termine avant l'échéance, et nous attribuerons à l'exécution de la partie optionnelle une fonction de récompense non-décroissante.

Ce type de technique est basé sur les modèles IRIS [5] (« Increasing Reward with Increasing Service »), ce qui signifie en reprenant notre cas, que plus un travail s'exécutera longtemps, plus la qualité de la sécurité s'en verra améliorée et donc plus la récompense sera importante.

Dans le modèle IRIS, la distinction entre partie obligatoire et optionnelle n'est pas faite, on permet uniquement aux tâches d'obtenir une augmentation de la récompense en fonction de l'allongement du temps d'exécution sans imposer de borne supérieure pour celui-ci exceptée bien entendu l'échéance de la tâche.

De manière à ne pas continuer à sécuriser une tâche au-delà d'un certain niveau de sécurité maximum, ce qui risquerait de nous empêcher de sécuriser les autres tâches efficacement, nous remarquons qu'il serait préférable dans le cadre de la sécurité, d'utiliser également une borne supérieure pour le temps d'exécution de la partie optionnelle, ce qui est possible avec cette version « Reward Based Scheduling ».

Nous allons donner priorité à la partie obligatoire, la partie optionnelle sera ensuite exécutée en utilisant le temps restant avant l'échéance si la partie obligatoire courante d'aucune tâche n'est encore présente dans le système.

Conditions d'ordonnançabilité et d'optimalité

Un niveau minimum de sécurité sera imposé et sera pris en compte dans la partie obligatoire.

Une condition *nécessaire* pour l'ordonnançabilité du système est que les parties obligatoires et optionnelles se terminent toutes avant l'échéance de manière à fournir une sortie minimale acceptable.

Mais il faudra également essayer de maximiser le plus possible la moyenne pondérée des récompenses obtenues par les différentes tâches afin de rendre cet ordonnancement *optimal*.

Malheureusement, ces deux objectifs sont la plupart du temps antagonistes étant donné que si l'on désire respecter les échéances, il faut généralement sacrifier en partie la sous-tâche optionnelle et donc la récompense s'en verra diminuée.

4.2.2 Une fonction de récompense

Chaque tâche se voit attribuer une fonction de récompense qui sera utilisée pour tous ses travaux. L'abscisse représente le temps d'exécution utilisé par la partie optionnelle du travail, l'ordonnée retourne la récompense obtenue avec ce temps. Plusieurs types de fonctions de récompense sont modélisables selon le type d'application qui nous intéresse.

Types de fonctions :

Convexe :

Définition *fonction convexe [23]* : Une fonction f d'un intervalle I vers \mathbb{R} est dite convexe si, $\forall x$ et $y \in I$ et tout t dans $[0, 1]$ on a :

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y).$$

De manière géométrique, cela correspond à un graphe dont la partie « bombée » est tournée vers le bas (ex : $f(x) = x^2$). Il en découle que,

plus le traitement est important, plus la façon dont la récompense augmente sera importante.

Concave :

Définition fonction concave [23] : On dit que la fonction f est concave si la fonction opposée est convexe. Cela équivaut à : $\forall x$ et $y \in I$ et tout t dans $[0, 1]$ on a :

$$f(tx + (1 - t)y) \geq tf(x) + (1 - t)f(y).$$

De manière géométrique, la partie bombée est tournée vers le haut, donc la partie la plus importante est située au début de l'exécution, c'est à ce moment que la récompense augmentera le plus rapidement, ce qui est par exemple surtout très utile dans le traitement d'images, où l'amélioration de la qualité est importante au début et faible par la suite.

Linéaire : (qui sont des cas particuliers de fonctions concaves et convexes) la récompense augmente uniformément pendant l'exécution de la partie optionnelle.

Choix du type de fonction

FEISTEL propose en 1973 [15], les paramètres et principes de base pour construire un bon algorithme de chiffrement. Parmi ceux-ci, on retrouve la taille des blocs, la taille de la clé, le nombre de tours dans l'algorithme, la complexité d'un tour et la complexité de génération des sous-clés pour chaque tour. Augmenter ces paramètres élève la sécurité mais diminue la vitesse de chiffrement. Il pourrait donc être intéressant d'augmenter ceux-ci lorsque l'ordonnancement le permet.

Nous choisirons ici d'utiliser des fonctions convexes qui se révèlent être intéressantes pour modéliser la valeur en sécurité d'un système en fonction du temps dépensé pour mettre celle-ci en place.

En effet, comme nous l'avons vu au chapitre 3, la valeur en sécurité d'un système dépend de la difficulté à « casser » cette sécurité. Plus la longueur de la clé utilisée pour le chiffrement est importante, plus une recherche exhaustive sur l'ensemble des clés consommera du temps avant de trouver la clé correcte. En rajoutant 1 bit à la clé l , l'espace de toutes les clés possibles et donc le temps mis par une attaque exhaustive sont multipliés par deux.

Soit une récompense de 2 pour une clé de 1 bit, la récompense vaudra donc 2^l pour une clé de l bits, où la récompense représente le nombre de clés

à tester au pire cas. Il reste donc à calculer le temps de chiffrement en fonction de la longueur de la clé et de la taille des données à chiffrer. A l'aide de la librairie crypto++ [4], nous avons calculé la durée de chiffrement d'une quantité de données de 1Mo avec les trois variantes officielles d'AES (voir tableau 4.1). Il est malheureusement impossible de modifier la longueur de la clé bit à bit car l'algorithme n'a pas été prévu pour cela à la base : le nombre de tour, la taille des clés et la taille des blocs sont dépendants les uns des autres.

Longueur des clés	Temps de chiffrement (en ms) pour 1Mo de données
Rijndael / 128 bits	38,16
Rijndael / 192 bits	41,93
Rijndael / 256 bits	46,31

TAB. 4.1 – Calcul de la durée de chiffrement selon la longueur de la clé

Sur la figure 4.2 est représentée graphiquement l'évolution de la valeur en sécurité du système en fonction du temps obtenu à partir du tableau 4.1. L'amélioration de la sécurité est si importante que sur le graphique, la différence de récompense n'était plus faite entre la version 128 et 192 bits par rapport à la version 256 bits, nous avons donc opté pour un graphique à l'échelle semi-logarithmique. Nous obtenons bien une fonction convexe exponentielle pour représenter l'évolution de la récompense en fonction du temps.

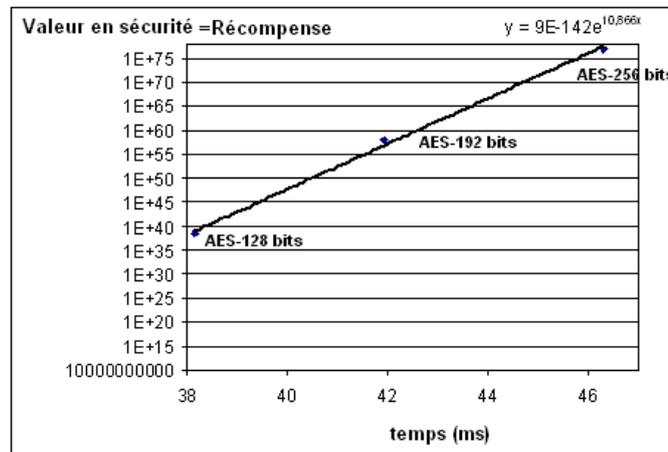


FIG. 4.2 – Valeur en sécurité du système en fonction du temps

4.2.3 Modélisation du système

Nous considérons un ensemble S de n tâches périodiques : T_1, T_2, \dots, T_n de période $P_i \forall i \in [1, n]$ à échéance sur requête et à départ simultané.

L'ordonnancement sera testé jusqu'à l'hyperpériode P (définie dans la chapitre 2), ce qui signifie que sur l'intervalle $[0, P)$, chaque tâche T_i enverra $b_i = \frac{P}{P_i}$ travaux. Dans ce modèle, chaque tâche est donc constituée de b_i travaux, nous nous référons au j^{e} travail de T_i par τ_{ij} . Ici, nous supposons que les tâches sont toutes indépendantes l'une de l'autre et prêtes à l'instant $t = 0$.

Lorsque l'exécution de la partie obligatoire M_i est achevée, la partie optionnelle devient disponible pour l'exécution et s'exécute aussi longtemps que l'ordonnanceur le permet avant l'échéance.

4.2.4 Calcul de la récompense moyenne du système

Comme nous l'avons dit dans la section 4.2.2, nous allons associer une fonction de récompense à la partie optionnelle, que nous noterons $R_i(t_{ij})$ et qui retournera la récompense accrue par le travail τ_{ij} lorsqu'il reçoit t_{ij} unités d'exécution au delà de sa partie obligatoire.

$$R_i(t_{ij}) = \begin{cases} f_i(t_{ij}) & \text{si } 0 \leq t_{ij} \leq o_i \\ f_i(o_i) & \text{sinon,} \end{cases} \quad (4.1)$$

Où f_i est une fonction réelle continue positive non-décroissante et o_i représente la longueur maximale de la partie optionnelle.

$m_i + o_i$ représente dans notre cas le temps maximum nécessaire pour sécuriser complètement les informations. Au delà de ce seuil ($t_{ij} > o_i$), la récompense ne sera plus augmentée (voir équation 4.1) et la priorité sera alors donnée à un travail qui n'a pas encore suffisamment sécurisé ses informations.

La **récompense moyenne** de T_i sera définie comme :

$$REW_i = \frac{\sum_{j=1}^{b_i} R_i(t_{ij})}{b_i}, \quad (4.2)$$

C'est-à-dire, la moyenne des récompenses accrues par chaque travail de la tâche jusqu'à l'hyperpériode ($= P = \text{ppcm}\{P_1, P_2, \dots, P_n\}$), b_i étant le nombre de fois que l'on peut placer la période de la tâche T_i dans cet intervalle ($= b_i = \frac{P}{P_i}$).

Pour calculer la **moyenne pondérée de la récompense** (REW_W), nous additionnons les récompenses REW_i de chaque tâche T_i à laquelle nous associerons un certain poids selon l'importance de la partie optionnelle de la tâche (voir équation 4.3). Nous intégrerons par la suite ce poids directement dans la fonction $f_i(t_{ij})$ sous forme d'un coefficient k_i .

$$REW_W = \sum_{i=1}^n w_i REW_i, \quad (4.3)$$

4.2.5 Approche Mandatory-First (parties obligatoires d'abord)

Dans cette approche, le principe est comme nous l'avons mentionné plus haut, de donner priorité aux parties obligatoires. Nous n'exécuterons les parties optionnelles que si la partie obligatoire de chaque tâche a été complètement exécutée.

Diverses techniques utilisant l'approche Mandatory-First existent. Nous pouvons les classer en approches statiques ou dynamiques.

Utilisation avec une méthode statique

Il existe plusieurs méthodes statiques, nous pouvons citer parmi celles-ci « **Rate Monotonic** » et « **Least Utilization** ».

Soit l'ordonnanceur donne une plus haute priorité :

- aux parties optionnelles de la tâche ayant la plus petite période, cette méthode est appelée « **Rate Monotonic** » (RM),
- soit en utilisant une méthode « **Least Utilization** » (LU) qui assigne les priorités statiquement aux parties optionnelles selon leur facteur d'utilisation (calculé par : $(o_i - m_i)/P_i$). Les tâches possédant un facteur d'utilisation plus faible reçoivent une priorité plus importante [6].

Utilisation avec une méthode dynamique

« Best Incremental Return » (BIR) est une des meilleures méthodes dynamiques car à chaque unité de temps, l'algorithme sélectionne la partie O_i qui augmente le plus la récompense en faisant le calcul $f_i(t_{ij} + \Delta) - f_i(t_{ij})$, où Δ est la

durée minimum que l'ordonnanceur attribuera à une tâche optionnelle. La fonction qui fournit la plus grande différence est élue et la tâche associée à cette fonction peut exécuter sa partie optionnelle.

BIR est la méthode la plus efficace en terme de récompense mais est très gourmande en calcul et n'est donc pas utilisée en pratique excepté comme étalon, pour comparer les performances d'autres algorithmes dans les simulations. De plus, BIR n'est pas optimal étant donné qu'il ne tient pas compte du temps restant avant les échéances.

Nous pouvons citer d'autres méthodes d'ordonnement dynamiques telles « **Earliest Deadline First** » (EDF) qui donne une priorité plus importante à l'exécution des parties optionnelles de la tâche ayant l'échéance la plus proche ou « **Least Laxity First** » (LLF) qui sélectionne un des travaux τ_{ij} prêt à l'instant courant t dont la laxité $\ell_{ij}(t)$ est la plus faible. La laxité se calcule par la formule 4.4.

$$\ell_{ij}(t) = d_{ij}(t) - rem_{ij}(t) \quad (4.4)$$

où $d_{ij}(t)$ est le temps restant avant l'échéance du travail τ_{ij} et $rem_{ij}(t)$ est le temps d'exécution restant de celui-ci calculé à l'instant t . Un travail qui a une échéance proche et qui a encore un temps important d'exécution devant lui est donc plus facilement sélectionné par LLF.

« **Least Attained Time** » (LAT) est une méthode intéressante dans le cas de fonctions concaves car elle donne une plus grande priorité aux parties optionnelles des tâches ayant utilisé le moins de temps processeur (priorité = $1/t_{ij}$), ce qui permet d'équilibrer le temps d'exécution des parties optionnelles.

Résultats de simulations

Dans son étude [3] sur le « Reward-Based Scheduling », H. Aydin a effectué des tests sur ces différentes méthodes et a constaté que :

- Plus l'utilisation du système par les parties obligatoires augmente, plus l'écart s'accroît entre la récompense accrue par l'algorithme optimal et celle accrue par les méthodes évoquées plus haut (statiques ou dynamiques) utilisées avec une approche Mandatory-First. Cela s'explique par le fait que ces méthodes perdent des occasions d'exécuter des parties optionnelles avantageuses au niveau de la récompense en favorisant constamment les parties obligatoires. Nous reviendrons de manière plus détaillée sur l'algorithme optimal par après.

- La récompense accumulée par une méthode LAT se rapproche plus de la méthode BIR lorsque les fonctions de récompense sont concaves. Comme LAT, des fonctions concaves favorisent des temps d'exécution équilibrés pour les parties optionnelles.
- L'écart entre la méthode optimale et les approches Mandatory-First est moins important lorsque les tâches possèdent des fonctions linéaires de récompense. En effet, les approches Mandatory First perdent des occasions d'exécuter des premiers slots valables de parties optionnelles. Or, l'exécution d'un premier slot de partie optionnelle est plus important si la fonction est concave vu que c'est à ce moment que la récompense augmente le plus.

Exemple d'ordonnement

Schématisons un exemple basé sur un cas d'école afin de montrer les limites d'un algorithme Mandatory-First.

Considérons deux tâches où $P_1=4$, $m_1=1$, $o_1=1$, $P_2=8$, $m_2=3$ et $o_2=5$ avec des fonctions de récompenses $f_1(t_1) = k_1 t_1$ pour T_1 et $f_2(t_2) = k_2 t_2$ pour T_2 , k_1 et k_2 étant donc les coefficients dans les fonctions linéaires de récompense de T_1 et T_2 respectivement. Nous supposerons que k_2 est négligeable par rapport à k_1 (c.-à-d. $k_1 \gg k_2$).

Avec une approche Mandatory-First nous obtenons donc l'ordonnement représenté dans la figure 4.3. Nous observons que quelle que soit l'assignation de priorité (statique ou dynamique) utilisée dans cet exemple avec l'approche Mandatory-First, le processeur sera occupé dans cet exemple par les parties obligatoires jusqu'à l'instant $t = 5$. Sur l'intervalle inoccupé $[5, 8]$ l'algorithme choisira de planifier O_1 complètement (soit une unité) étant donné qu'il fournit une plus grande récompense que O_2 . Le temps restant (2 unités) sera consacré à O_2 .

Un algorithme optimal produirait cependant l'ordonnement reproduit sur la figure 4.4. L'exécution de M_2 est retardée d'une unité afin de permettre l'ordonnement de O_1 sur la première période de T_1 , ce qui nous fournirait une plus grande récompense que celle fournie par l'exécution d'une unité supplémentaire de O_2 et les échéances des parties obligatoires sont toujours respectées. Comme $k_1 \gg k_2$, on peut conclure que la récompense accrue par l'algorithme Mandatory-First ne peut valoir qu'environ la moitié de celle accrue par l'algorithme optimal.

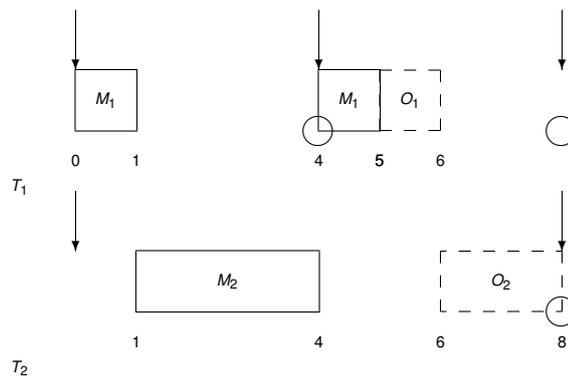


FIG. 4.3 – Ordonnancement généré par un algorithme Mandatory-First

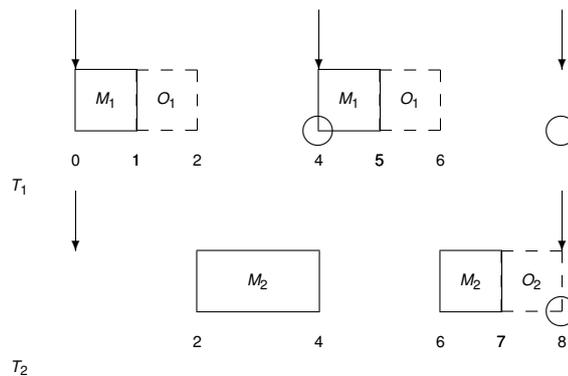


FIG. 4.4 – Ordonnancement généré par un algorithme optimal

Pour poursuivre notre comparaison entre le « meilleur » algorithme Mandatory-First et un algorithme optimal, nous pouvons prouver qu’au pire cas, la proportion de récompense accumulée par l’approche Mandatory-First par rapport à la récompense obtenue par un algorithme optimal peut être arbitrairement proche de zéro si on prend un r arbitrairement grand dans le théorème 1 issu de l’article d’Aydin [3]. Il n’y a donc pas de garantie de performance des méthodes Mandatory-First.

Théorème 1 *Il existe une instance du problème d’ordonnancement périodique basée sur la récompense pour lequel la proportion :*

$$\frac{\text{Récompense du meilleur algorithme Mandatory-First}}{\text{Récompense d’un algorithme optimal}} = \frac{2}{r}.$$

pour n’importe quel entier $r \geq 2$.

Démonstration: *Prenons un système formé de deux tâches : T_1 et T_2 avec*

des fonctions de récompense respectivement : $f_1(t_1) = k_1 \cdot t_1$, $f_2(t_2) = k_2 \cdot t_2$ et posons $r = P_2/P_1$, $k_1/k_2 = r \cdot (r - 1)$, $m_2 = \frac{1}{2} \cdot (r \cdot o_2)$ et

$$P_1 = m_1 + o_1 + \frac{m_2}{r} = m_1 + \frac{m_2}{r-1}$$

ce qui implique que $o_1 = \frac{m_2}{r(r-1)}$ et que P_2 est l'hyperpériode du système.

Suivant ces hypothèses, nous pouvons observer qu'à chaque période de T_1 , l'algorithme peut exécuter en plus de la partie obligatoire M_1 , une partie de O_1 et/ou M_2 .

Avec une approche Mandatory-First, le processeur exécute uniquement les parties obligatoires jusqu'à l'instant $t = P_2 - P_1 + m_1$ (voir figure 4.5). Le temps restant, est utilisé pour ordonnancer tout d'abord O_1 entièrement car $k_1 > k_2$ ($k_1 = r \cdot (r - 1) \cdot k_2$), et ensuite $t_2 = \frac{m_2}{r} = \frac{o_2}{2}$ unités de O_2 .

Nous calculons donc la récompense moyenne obtenue grâce à l'algorithme Mandatory-First :

$$R_{MFA} = REW_1 + REW_2 = \frac{f_1(o_1)}{r} + f_2(t_2),$$

d'après les formules 4.2 et 4.3 en sachant que $P_1/P = P_1/P_2 = 1/r$ et $P_2/P = 1$

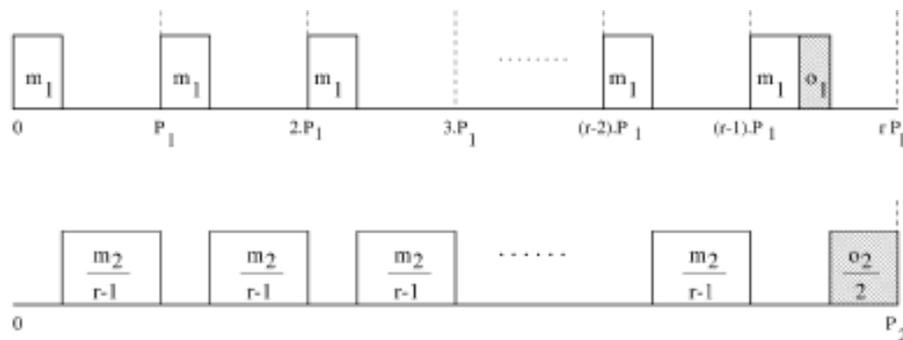


FIG. 4.5 – Un ordonnancement généré par un algorithme « Mandatory-First ».

Un algorithme optimal, aurait choisi quant à lui de retarder l'exécution de M_2 de o_1 unités à chaque période de T_1 . Cela permet d'accroître la récompense fournie par T_1 à chaque travail. L'ordonnancement résultant est représenté sur la figure 4.6 et la récompense moyenne en utilisant un tel algorithme serait :

$$R_{OPT} = REW_1 + REW_2 = REW_1 = \frac{r \cdot f_1(o_1)}{r} = f_1(o_1),$$

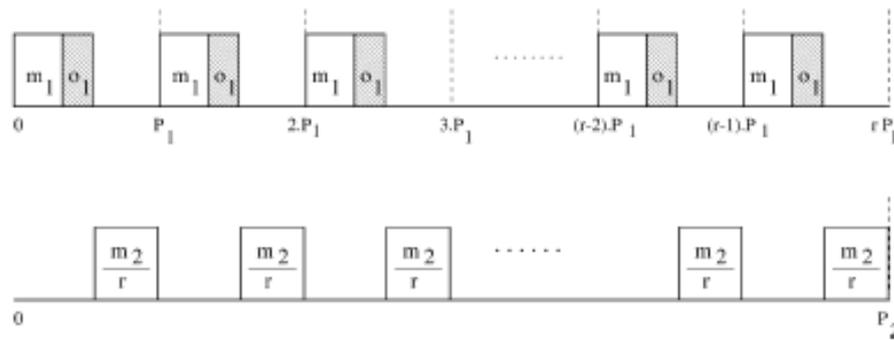


FIG. 4.6 – Un ordonnancement généré par un algorithme optimal.

Nous obtenons donc la proportion sur la récompense entre les deux algorithmes ($\forall r \geq 2$) en remplaçant les :

$$\begin{aligned} \frac{R_{MFA}}{R_{OPT}} &= \frac{1}{r} + \frac{f_2(t_2)}{f_1(o_1)} \\ &= \frac{1}{r} + \frac{k_2 \cdot t_2}{k_1 \cdot o_1} \\ &= \frac{1}{r} + \frac{1}{r(r-1)} \frac{m_2 r(r-1)}{r m_2} \\ &= \frac{2}{r}. \end{aligned}$$

Il suffit ensuite de choisir un système avec une constante r tendant vers ∞ pour que cette proportion tende vers 0.

□

4.2.6 Optimalité

Dans cette section nous allons formaliser le problème de maximisation de la récompense sur un ordonnancement de tâches périodiques. Notre objectif, est de trouver les valeurs optimales t_{ij} dans l'équation 4.5 qui maximiseront cette récompense. Nous verrons par la suite qu'il peut être intéressant de poser $t_{ij} = t_{ik} \quad \forall i, k$ mais nous analyserons également le cas plus général (où $\exists i, j, k$ tel que $t_{ij} \neq t_{ik}$). L'équation 4.5 est construite à partir des équations 4.2 et 4.3, en supposant que le poids w_i de la tâche T_i est intégré dans la fonction de

récompense $R_i(t_{ij})$.

$$\sum_{i=1}^n \frac{P_i}{P} \sum_{j=1}^{b_i} R_i(t_{ij}), \quad (4.5)$$

Il faudra ajouter à cette équation quelques contraintes.

1. Il faut imposer que le temps utilisé par toutes les parties obligatoires et optionnelles de chaque tâche jusqu'à l'hyperpériode (P) ne dépasse pas le temps processeur disponible sur celle-ci. Autrement dit, que

$$\sum_{i=1}^n \sum_{j=1}^{b_i} (m_i + t_{ij}) \leq P.$$

(réécrit d'une autre manière dans l'équation 4.7)

Malheureusement, cette contrainte est nécessaire mais en aucun cas suffisante pour établir la faisabilité du système avec les valeurs des m_i et t_{ij} .

2. Il faut donc exprimer la faisabilité totale avec les valeurs m_i et t_{ij} . Mais à notre connaissance, il n'existe pas dans la littérature, de formule mathématique permettant de tester la faisabilité avec des temps d'exécution différents pour chaque travail d'une même tâche, ce qui peut être ici le cas puisque t_{ij} pourrait varier d'un travail τ_{ij} de T_i à l'autre. Nous effectuerons ce test en simulant l'ordonnancement jusqu'à l'hyperpériode.
3. Un temps négatif n'a pas d'interprétation physique et t_{ij} dépassant o_i n'est d'aucune utilité étant donné que la récompense n'est plus augmentée. Ce qui nous amène à poser $0 \leq t_{ij} \leq o_i$, nous pouvons donc remplacer $R_i()$ par $f_i()$ (voir équation 4.1).

Reprenant toutes ces contraintes, le problème que nous appellerons REW-PER est donc de trouver les valeurs optimales pour les t_{ij} :

$$\text{maximisant} \quad \sum_{i=1}^n \frac{1}{b_i} \sum_{j=1}^{b_i} f_i(t_{ij}) \quad (4.6)$$

$$\text{à condition que} \quad \sum_{i=1}^n b_i m_i + \sum_{i=1}^n \sum_{j=1}^{b_i} t_{ij} \leq P \quad (4.7)$$

$$0 \leq t_{ij} \leq o_i \quad i = 1, \dots, n \quad j = 1, \dots, b_i \quad (4.8)$$

Il existe un *ordonnancement faisable* avec les valeurs $\{m_i\}$ et $\{t_{ij}\}$.
(4.9)

Cette dernière condition 4.9 revient à dire que le système avec les valeurs $\{m_i\}$ et $\{t_{ij}\}$ doit être ordonnançable avec EDF sur l'intervalle $[0, P)$. Nous obtenons donc un ordonnancement faisable avec les valeurs m_i et des valeurs optimales pour t_{ij} .

Notons que si la contrainte 4.7 n'est pas satisfaite, la contrainte 4.9 ne l'est pas non plus. Nous aurions donc pu nous limiter à poser la contrainte 4.9, mais 4.7 reste intéressante pour tester la faisabilité.

En effet, lorsque 4.7 n'est pas satisfait, nous savons que le système n'est alors pas ordonnançable et il n'est donc pas nécessaire d'effectuer la simulation jusqu'à l'hyperpériode pour tester son ordonnançabilité.

H. Aydin nous démontre par le théorème 2 dans son article [3] comment il est possible de supprimer la contrainte 4.9 en utilisant des temps d'exécution des parties optionnelles t_{ij} égaux à chaque appel ($t_{ij} = t_{ik} \quad \forall j, k$). Cette démonstration est applicable uniquement dans le cadre d'une utilisation de fonctions de récompense *concaves* ou linéaires (qui sont des cas particuliers de fonctions concaves). Nous remarquons intuitivement que si les fonctions sont convexes, il est plus intéressant de prolonger l'exécution de t_{ij} pour certains travaux puisque la récompense augmente de manière exponentielle.

Théorème 2 *Soit un exemple donné du problème REW-PER, il existe une solution optimale où les parties optionnelles d'une tâche T_i reçoivent le même temps d'exécution à chaque travail, c'est-à-dire, $t_{ij} = t_{ik} \quad 1 \leq j < k \leq \frac{P}{P_i}$. De plus, n'importe quelle méthode d'ordonnancement périodique temps-réel qui peut utiliser entièrement le processeur (EDF, LLF, RMS-h) peut être utilisée pour obtenir un ordonnancement faisable avec ces assignations.*

Démonstration: *Abandonnons la contrainte 4.9 sur la faisabilité et appelons MAX-REW le problème REW-PER sans la contrainte de faisabilité.*

Assertion 1 *Posons t_{ij} comme une solution optimale de MAX-REW, $1 \leq i \leq n \quad 1 \leq j \leq \frac{P}{P_i} = b_i$. Alors, t'_{ij} , en posant*

$$t'_{i1} = t'_{i2} = \dots = t'_{ib_i} = t'_i = \frac{t_{i1} + t_{i2} + \dots + t_{ib_i}}{b_i}$$

$$1 \leq i \leq n \quad 1 \leq j \leq b_i,$$

est aussi une solution optimale de MAX-REW.

1. t'_{ij} satisfait les contraintes du problème MAX-REW (4.7 et 4.8) si t_{ij} les satisfait déjà.

En effet,

- La contrainte 4.7 est toujours respectée avec les valeurs t'_{ij} car $\sum_{j=1}^{b_i} t_{ij} = \sum_{j=1}^{b_i} t'_{ij} = b_i t'_i$.
- La contrainte 4.8 est également respectée puisque par hypothèse $t_{ij} \leq o_i \forall j$, donc $\max_j \{t_{ij}\} \leq o_i$. D'autre part, $t'_i \leq \max_j \{t_{ij}\}$ autrement $b_i t'_i > \sum_{j=1}^{b_i} t_{ij}$, ce qui contredirait l'hypothèse de l'assertion. Cela implique que $t'_i \leq o_i, i \in [1, n]$.

2. H. Aydin démontre mathématiquement dans [3] que la récompense ne peut pas décroître en utilisant des parties optionnelles identiques, en prouvant que $\sum_{j=1}^{b_i} f_i(t_{ij}) \leq b_i f_i(t'_i)$ si $f_i(t)$ est une fonction concave (incluant le cas des fonctions linéaires). Le lecteur intéressé peut consulter cette démonstration que nous avons reprise dans l'annexe A.

□

Nous pouvons dès lors simplifier le problème MAX-REW et le réécrire comme une :

$$\text{maximisation de } \sum_{i=1}^n f_i(t_i) \quad (4.10)$$

$$\text{à condition que } \sum_{i=1}^n \frac{P}{P_i} t_i + \sum_{i=1}^n \frac{P}{P_i} m_i = \sum_{i=1}^n \frac{P}{P_i} (m_i + t_i) \leq P \quad (4.11)$$

$$0 \leq t_{ij} \leq o_i \quad i = 1, \dots, n. \quad (4.12)$$

La contrainte 4.11 peut encore être simplifiée comme $\sum_{i=1}^n \frac{(m_i+t_i)}{P_i} \leq 1$. Ce qui est équivalent à exprimer la condition $U = \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$ nécessaire et suffisante sur l'utilisation pour qu'une politique d'ordonnancement périodique classique (comme par exemple EDF, LLF, ...) permette d'obtenir un ordonnancement faisable avec n tâches pour lesquelles le temps d'exécution $C_i = m_i + t_i \forall i \in [1, n]$. Cela nous permet de ne pas devoir tenir compte de la contrainte de faisabilité 4.9 du problème REW-PER.

Limites de l'optimalité

Malheureusement l'optimalité avec $t_{ij} = t_{ik} \forall j, k$ ne tient pas dans tous les cas. Il faudra donc poser quelques hypothèses sur le système de tâches pour

que le système fournisse une récompense optimale avec des parties optionnelles ayant des temps d'exécution identiques à chaque période, tout en satisfaisant la contrainte de faisabilité. Ces hypothèses sont démontrées à chaque fois par un contre-exemple dans l'article [3].

Récapitulons ces hypothèses :

1. Toutes les tâches doivent être à échéance sur requête, c.-à-d. que l'échéance doit être égale à la période pour chacune d'elles.
2. L'optimalité ne tient généralement pas si l'algorithme d'ordonnancement utilise une règle d'assignation statique. C'est pourquoi nous utiliserons EDF comme politique d'ordonnancement (dynamique).
3. La fonction de récompense doit être de type concave.

Nous allons maintenant voir comment résoudre le problème MAX-REW sur un système constitué de tâches périodiques à échéance sur requête avec une fonction linéaire de récompense associée à chaque tâche. Les parties optionnelles seront de même taille de manière à garantir l'optimalité (voir théorème 2) et l'ordonnancement sera basé sur une règle d'assignation dynamique des priorités, « *Earliest Deadline First* » (EDF). Nous respecterons donc toutes les hypothèses citées ci-dessus.

Nous nous baserons sur un algorithme présenté dans [3]. Celui-ci est optimal lorsque les fonctions de récompense sont linéaires, il nous assure donc d'obtenir la meilleure récompense possible. Cependant, cet algorithme n'est plus optimal lorsqu'une des fonctions de récompense est convexe. Il sera présenté à la section 4.2.7 et utilisé dans le chapitre suivant comme étalon afin de le comparer à des méthodes heuristiques.

4.2.7 Optimalité avec des fonctions linéaires de récompense

Soit une fonction linéaire de récompense de la forme : $f_i(t_i) = k_i t_i$, en augmentant la valeur t_i de Δ unités de temps, la récompense accrue augmentera de $k_i \Delta$ mais nous utiliserons $b_i \Delta$ unités de la laxité disponible d . Nous noterons par b_i (4.14) le nombre de travaux de la tâche T_i qu'il est possible de placer dans l'hyperpériode. Donc, la valeur apportée au système par la tâche T_i par unité consommée sur la laxité est $w_i = \frac{k_i \Delta}{b_i \Delta} = \frac{k_i}{b_i}$ avec $b_i = \frac{P}{P_i}$.

Définition *laxité* : Nous définissons la laxité d au début de l'exécution de l'algorithme, comme le temps restant disponible pour exécuter des parties option-

nelles jusqu'à l'hyperpériode, soit :

$$d = P - \sum_{i=1}^n m_i \cdot b_i \quad (4.13)$$

$$\text{avec } b_i = \frac{P}{P_i} \quad (4.14)$$

Nous allons donc calculer la valeur apportée w_i pour chaque tâche T_i . Ensuite les tâches seront triées par rapport à cette valeur, les tâches ayant un w_i plus important seront favorisées.

Posons T_1 comme la tâche ayant la plus grande valeur w_i , T_2 la suivante, et ainsi de suite. Si le fait de choisir le temps maximal o_1 pour les parties optionnelles des travaux de la tâche T_1 ne nous fait pas dépasser d ($b_1 \cdot o_1 \leq d$), nous fixons le temps optionnel t_1 à o_1 et la laxité restante est diminuée de $b_1 \cdot o_1$.

Ensuite, on recommence la même procédure avec la tâche T_2, T_3, \dots jusqu'à ce qu'à la i^{e} étape, la laxité restante soit insuffisante pour accepter de prendre o_i comme temps optionnel. Dans ce cas, la laxité restante est divisée par le nombre de travaux (b_i) de T_i jusqu'à l'hyperpériode. Cette valeur est attribuée comme temps optionnel t_i et toute la laxité est alors consommée, l'utilisation du processeur, parties obligatoires et optionnelles confondues (U) vaut donc dans ce cas 1, ce qui permet d'ordonnancer le système avec EDF.

Il est aussi possible que la laxité soit suffisante pour appliquer $t_i \leftarrow o_i$ à toutes les tâches, U est alors inférieur à 1 et donc le système est également ordonnançable avec EDF.

Les dernières tâches de la liste triée, n'ayant pas un poids w_i suffisamment important, reçoivent 0 comme temps optionnel.

Voici l'algorithme 3 exprimé de façon plus formelle, utilisé pour définir la taille optimale des $t_i \forall i \in [1, n]$:

Entrées : Ensemble des n tâches avec leurs caractéristiques (k, b, m, o) et l'hyperpériode P

Sorties : Vecteur des temps optionnels t_i calculés pour chaque tâche T_i

Tri des tâches en fonction des valeurs $w_i = \frac{k_i}{b_i}$;

$i \leftarrow 1$; //Indice des tâches dans la liste triée.

$d \leftarrow P - \sum_{i=1}^n m_i \cdot b_i$;

Tant que $(i \leq n$ **ET** $d > 0)$ **faire**

Si $(b_i o_i < d)$ **Alors**

$t_i \leftarrow o_i$;

$d \leftarrow d - b_i o_i$;

Sinon

$t_i \leftarrow \frac{d}{b_i}$;

$d \leftarrow 0$;

Fin Si

$i \leftarrow i + 1$;

Fait

//les tâches restantes n'exécutent pas leur partie optionnelle

Tant que $(i \leq n)$ **faire**

$t_i \leftarrow 0$;

$i \leftarrow i + 1$;

Fait

Algorithme 3: Recherche des t_i optimaux - Cas linéaire

Après au plus n itérations de la première boucle, la latitude (ou laxité) d est complètement consommée.

En plus de ce parcours qui possède une complexité au pire cas en $O(n)$, nous devons effectuer un tri dont la complexité du meilleur algorithme est en $O(n \log n)$. La complexité de l'algorithme est donc identique à celle du tri, soit $O(n \log n)$.

Deux types de simulation

Nous constatons, que la taille des t_i peut être un nombre rationnel positif. En effet, le résultat de $\frac{d}{b_i}$ n'est pas forcément une valeur entière.

Nous nous sommes confrontés à ce problème en réalisant l'implémenta-

tion¹ de cet algorithme. En effet, le fait que les temps ne soient pas des valeurs entières nous a forcés à utiliser une technique de simulation « *orientée évènement* » (« Event driven ») plutôt qu'une simulation *basée sur une horloge* où le temps progresse de façon linéaire d'une unité à chaque tour de boucle du simulateur.

Explications :

Dans la **simulation orientée évènement**, chaque évènement est rajouté par une insertion triée dans une file à priorité par rapport à l'instant auquel l'évènement doit se déclencher.

Un évènement peut être :

1. l'arrivée d'un travail et l'échéance du travail précédent étant donné que nous travaillons sur des systèmes à échéance sur requête ;
2. la fin d'exécution d'un travail.

Dans ce type de simulation, on fait donc uniquement appel à l'ordonnanceur lorsqu'un évènement se produit, ce qui permet de ne pas *réveiller* l'ordonnanceur continuellement sur des périodes *oisives* et sur les longues périodes durant lesquelles un travail pourrait s'exécuter sans être *préempté*. Nous évitons donc ainsi les tours de boucle inutiles dans lesquels aucun évènement important ne se produit à l'exception de l'augmentation de Δt unités d'exécution du travail en cours d'exécution.

Soit Δt , l'incrémentation du temps à chaque tour de boucle de l'ordonnanceur dans une **simulation basée sur une horloge**, le fait d'utiliser la notion d'évènements a comme avantage majeur qu'il ne faut pas choisir un Δt très petit, par exemple $\Delta t = 0.001$ afin de gérer les décimales ce qui provoquerait également des erreurs de précision.

Nous expliquerons plus en détail la simulation basée sur les évènements dans le chapitre suivant lorsque nous décrirons notre simulation complète.

4.2.8 Optimalité avec des fonctions convexes de récompense

Dans le cas où les fonctions de récompense sont de type convexe, des temps d'exécution optimaux pour les parties optionnelles sont beaucoup plus complexes à obtenir au niveau du temps de calcul. H. Aydin [3] nous démontre

¹Le code C++ de la simulation et de l'algorithme 3 sont fournis dans l'annexe B (méthode linOPT())

que c'est un problème NP-difficile même lorsque les parties optionnelles sont identiques d'une instance à l'autre.

Problème NP-difficile, NP-complet

Les problèmes sont séparés en plusieurs classes de complexité, citons parmi celles-ci les 2 classes les plus connues :

- **Classe P** : Le problème peut être résolu de manière déterministe en un temps polynomial.
- **Classe NP** : Le problème peut être résolu de manière non-déterministe en un temps polynomial.

Le caractère non-déterministe de la classe NP signifie que le problème peut être représenté par un arbre, sur lequel à chaque étape (chaque nœud) il y aurait un choix à faire entre plusieurs actions et la machine devant résoudre ce problème ne pourrait en effectuer qu'une, sans être certain que la branche choisie soit la bonne. Le problème est donc exponentiel selon la taille des données. Si une machine pouvait tester tous les nœuds fils en même temps à chaque étape (ce qui est encore théorique mais non applicable en pratique), un problème de la classe NP pourrait être résolu en un temps polynomial.

Définition Problème NP-difficile [24] : *Un problème est NP-difficile s'il est au moins aussi dur que tous les problèmes dans NP. Soit un problème π et un algorithme permettant de réduire en un temps polynomial (dans P) ce problème à un problème π' , il est donc possible de résoudre π' si l'on sait résoudre π , ce qui rend π aussi difficile à résoudre que π' . π est alors NP-difficile si pour tout problème π' de NP, π' se réduit en un temps polynomial à π .*

Définition Problème NP-complet [24] : *Un problème est NP-complet si il est dans NP et est NP-difficile.*

Pour prouver que notre problème est NP-difficile, tentons de faire l'analogie de celui-ci au problème *subset-sum*, également appelé le problème du sac à dos, bien connu pour être NP-complet. Si cette analogie est correcte, l'hypothèse sera prouvée.

Premièrement, définissons le problème subset-sum.

Définition Problème Subset-sum : *Pour un ensemble S donné $\{s_1, s_2, \dots, s_n\}$ de nombres entiers positifs et le nombre entier M , existe-t-il un ensemble $S_A \subseteq S$ tel que $\sum_{s_i \in S_A} s_i = M$?*

Démonstration: *Construisons un ensemble de n tâches de même période M sans partie obligatoire. Ce qui simplifie la contrainte 4.11 du problème MAX-REW par l'équation 4.16 car $m_i = 0 \forall i$.*

Posons $W = \sum_{i=1}^n s_i$ avec $s_i = o_i$ et choisissons comme fonction de récompense la fonction $f_i(t_i) = t_i^2 + (W - s_i)t_i$ qui est strictement convexe si $t_i \geq 0$. Elle peut être réécrite comme $t_i(t_i - s_i) + Wt_i$.

Etant donné que la période M est identique pour chaque tâche, l'hyperpériode P dans MAX-REW peut être remplacée par M et nous pouvons réécrire le problème MAX-REW ainsi :

$$\text{maximisons} \quad \sum_{i=1}^n t_i(t_i - s_i) + W \sum_{i=1}^n t_i \quad (4.15)$$

$$\text{à condition que} \quad \sum_{i=1}^n t_i \leq M \quad (4.16)$$

$$0 \leq t_i \leq s_i. \quad (4.17)$$

Nous observons que si $0 < t_i < s_i$, alors $t_i(t_i - s_i) < 0$ ce qui ne maximise clairement pas la récompense car si $t_i = 0$ ou $t_i = s_i$ nous obtenons la quantité $t_i(t_i - s_i) = 0$. Et donc, la première partie de l'équation 4.15 peut être supprimée, il reste à maximiser $W \sum_{i=1}^n t_i$. La récompense totale que nous noterons $Rew \leq W \cdot M$ à cause de la contrainte 4.16.

Résoudre la question « Rew est-elle égale à $W \cdot M$? » revient à résoudre la question « Existe-il un ensemble de t_i tel que $\sum_{i=1}^n t_i = M$, chaque t_i étant égal à 0 ou à s_i ? », autrement dit $Rew = W \cdot M$ si et seulement s'il existe un ensemble $S_A \subseteq S$ tel que $\sum_{s_j \in S_A} s_j = M$. L'analogie au problème Subset-Sum est faite, ce qui implique que MAX-REW et donc REW-PER avec des fonctions convexes de récompense est NP-difficile. \square

4.3 Conclusion

Nous avons développé dans ce chapitre, la méthode de H. Aydin : le « Reward-Based Scheduling ». Dans son travail [3], cette méthode est appliquée avec des fonctions concaves de récompense, nous avons montré pourquoi dans le cadre de la sécurité, les fonctions convexes semblent plus appropriées. Il a été démontré que le problème est NP-complet avec de telles fonctions.

Cependant, même s'il n'existe pas encore à ce jour de méthodes déterministes et polynomiales en temps pour résoudre le problème ² avec des fonctions convexes de récompense, nous analyserons dans le chapitre 5 des méthodes heuristiques afin de trouver une solution au problème se rapprochant le plus possible de la solution optimale.

²étant donné qu'à l'heure actuelle, l'égalité $NP = P$ n'a pas encore été démontrée est récompensée par 1 000 000\$ à celui qui la prouvera

Chapitre 5

Simulations

5.1 Introduction

Le but principal de ce chapitre sera d'essayer en se basant sur l'idée développée au chapitre 4 de développer un algorithme permettant d'attribuer des durées plus ou moins longues aux parties optionnelles des différents travaux, lorsque la récompense évolue de manière convexe en fonction de ce temps.

Pour parvenir à ce but, nous avons dû implémenter un simulateur nous permettant de tester si le système est toujours ordonnançable avec les durées choisies. Nous allons donc dans ce chapitre montrer comment nous générons un ensemble de tâches formant ce système. Ensuite nous présenterons deux méthodes heuristiques que nous avons implémentées : le recuit simulé et la recherche tabou. Nous ferons une comparaison des résultats obtenus avec ces méthodes par rapport à l'algorithme présenté dans le chapitre précédent (section 4.2.7) qui est optimal dans le cas où les fonctions de récompense sont linéaires et nous effectuerons ensuite une comparaison avec des fonctions convexes de récompense.

5.2 Génération d'un système

5.2.1 Génération des périodes

Un système est composé de tâches, chaque tâche T_i lance une instance (un travail) à chaque instant $kP_i \forall k \geq 0$. La simulation devant être effectuée sur l'intervalle $[0, P]$, il est indispensable lors de la génération de cet ensemble de tâches, de choisir judicieusement les valeurs des périodes P_i des différentes tâches.

En effet, si P_i est choisi de manière arbitraire entre deux bornes, et si le nombre de tâches est important, cela pourrait nous amener à ce que le plus petit commun multiple de ces valeurs soit énorme. La simulation serait donc beaucoup plus longue à effectuer étant donné qu'il faut simuler l'évolution du système jusqu'à l'hyperpériode $P = \text{ppcm} \{P_i\} \forall T_i$.

Nous avons donc décidé, afin de faciliter nos simulations d'utiliser la technique exposée dans [8] qui consiste à choisir la période comme un multiple des premières puissances de nombres premiers :

$$P_i = \prod_{\{p|p \text{ est premier}\}} p^{u_p}$$

où p est un nombre premier et u_p est la puissance de p sélectionnée au hasard.

Une matrice contient un ensemble de nombres premiers et leurs premières puissances (ex : $2^1, 2^2, \dots$).

De manière à ne pas obtenir des périodes trop importantes, nous avons construit cette matrice avec de petits nombres premiers ($p \mid p \text{ est premier} \in [2, 11]$) et surtout de petites puissances ($u_p \in [1, 2]$), nous avons choisi d'utiliser dans notre simulation la matrice χ :

$$\chi = \begin{pmatrix} 2 & 2 & 4 \\ 3 & 3 & 9 \\ 5 & 5 & 25 \\ 7 & 7 & 7 \\ 11 & 11 & 11 \end{pmatrix}$$

L'algorithme (repris de [8]) qui retourne une période à partir de la matrice χ est décrit ci-dessous :

```

periode ← 1
Pour chaque ligne i de  $\chi$  faire
    |  $p \leftarrow \text{Arrondi}(\text{random}(1, |\chi_i|))$ 
    |  $\text{periode} \leftarrow \text{periode} \times \chi_{i,p}$ 
Fin Pour
Retourner periode

```

Algorithme 4: Détermine la période

Sur chaque ligne, l'algorithme sélectionne une valeur et il multiplie les valeurs sélectionnées entre elles. Avec la matrice χ , la période maximum est donc : $4 \times 9 \times 25 \times 7 \times 11 = 69300$, c.-à-d. le produit des valeurs les plus importantes sur chaque ligne, c'est aussi la taille de la plus grande hyperpériode car n'importe quelle autre période calculée sera automatiquement un diviseur de 69300.

5.2.2 Génération de l'ensemble de tâches

- Nous générons ensuite l'ensemble des tâches T_i avec leurs caractéristiques :
- P_i : La période, générée par l'algorithme 4 ci-dessus ;
 - m_i : le temps d'exécution de la partie obligatoire ;
 - o_i : le temps d'exécution maximal de la partie optionnelle.
 - coeff_i : qui est le coefficient de la fonction de récompense de la tâche T_i , nous le choisissons au hasard entre 1 et 100.

L'instant d'arrivée $A_i = 0 \forall T_i$ et l'échéance $D_i = P_i \forall T_i$, car nous travaillons sur des systèmes à départ simultané et à échéance sur requête.

Il s'avère compliqué de générer au hasard un système composé d'exactly n tâches avec un facteur global d'utilisation U_m , n et U_m étant fournis en entrée à l'algorithme. Soit le nombre de tâches est correct mais l'utilisation n'est pas proche de U_m , soit l'utilisation est proche de U_m mais le nombre de tâches est différent de n .

Nous avons donc écrit deux algorithmes : l'algorithme 5 génère un système d'exactly n tâches indépendamment de l'utilisation, l'algorithme 6 génère un certain nombre de tâches avec un facteur d'utilisation des parties obligatoires se rapprochant le plus possible de U_m passé en paramètre.

```

Entrée :  $n$ 
Sortie : Ensemble de tâches
 $i \leftarrow 0$ 
 $utilisation\_courante \leftarrow 0$ 
Tant que ( $i \neq n$ ) faire
     $P \leftarrow$  période générée par l'algorithme 4
     $m \leftarrow random(0, P)$ 
     $o \leftarrow P - m$ 
     $coeff \leftarrow random(1, 100)$ 
     $taskU \leftarrow \frac{m}{P}$ 
    Si ( $utilisation\_courante + taskU \leq 1$ ) Alors
         $utilisation\_courante \leftarrow utilisation\_courante + taskU$ 
        AjoutTache( $P, m, o, coeff$ )
         $i \leftarrow i + 1$ 
    Fin Si
Fait

```

Algorithme 5: Génération d'un ensemble de n tâches

```

Entrée :  $U_m$ 
Sortie : Ensemble de tâches
 $utilisation\_courante \leftarrow 0$ 
Tant que ( $utilisation\_courante \leq U_m - 0.05$ ) faire
     $P \leftarrow$  période générée par l'algorithme 4
     $m \leftarrow random(0, P)$ 
     $o \leftarrow P - m$ 
     $coeff \leftarrow random(1, 100)$ 
     $taskU \leftarrow \frac{m}{P}$ 
    Si ( $utilisation\_courante + taskU \leq U_m$ ) Alors
         $utilisation\_courante \leftarrow utilisation\_courante + taskU$ 
        AjoutTache( $P, m, o, coeff$ )
    Fin Si
Fait

```

Algorithme 6: Génération d'un ensemble de tâches avec un facteur d'utilisation proche de U_m

L'utilisateur donne en paramètre l'utilisation du système par les parties obligatoires (pour rappel : $U_m = \sum_{i=1}^n \frac{m_i}{P_i}$) dont il voudrait se rapprocher ou le nombre de tâches : n .

Tant que l'on ne dépasse pas le facteur d'utilisation ou le nombre de tâches désiré selon l'algorithme, on génère une nouvelle tâche. La période est calculée à partir de l'algorithme 4, le temps d'exécution de la partie obligatoire est choisi au hasard entre 0 et la période jusqu'à ce que le système ait atteint l'utilisation ou le nombre de tâches désiré.

Nous avons d'autre part décidé de choisir des durées maximales pour les parties optionnelles de mêmes longueurs que le temps restant avant l'échéance laissé par les parties obligatoires, soit pour chaque tâche T_i , $o_i = P_i - m_i$.

Dans l'algorithme 6, si l'utilisation courante en tenant compte de la tâche venant d'être générée ne dépasse pas U_m , nous rajoutons la tâche et augmentons l'utilisation courante. Dans l'algorithme 5, nous vérifions uniquement que les parties obligatoires soient ordonnancables après avoir ajouté la tâche ($utilisation_courante + taskU \leq 1$).

5.3 L'algorithme orienté évènement

Comme nous l'avons expliqué au chapitre précédent, il nous a fallu mettre au point un algorithme d'ordonnancement basé sur la notion d'évènements et non plus un algorithme qui effectue une incrémentation du temps à chaque tour de boucle.

Dans cette section, nous fournirons au lecteur une description de l'algorithme « Event-driven » 7 et de la façon dont nous avons implémenté celui-ci.

Nous utiliserons les termes :

- « *Arrivée* » : pour désigner l'évènement qui, pour une tâche donnée, annonce l'arrivée dans le système d'un nouveau travail et l'échéance du travail précédent de cette tâche par la même occasion ;
- « *FinJob* » : pour l'évènement qui annonce la terminaison d'un travail.

Le système comporte en plus des caractéristiques des différentes tâches, une file d'évènements.

Chaque évènement que l'on peut insérer dans cette file comporte 4 informations :

1. le **temps** : c'est-à-dire l'instant auquel cet évènement doit se déclencher ;

2. un pointeur vers le **travail ou la tâche** à laquelle il se rapporte ;
3. son **type** : qui identifie l'évènement (Arrivée, FinJob), le type est utilisé pour l'insertion dans la file, nous y reviendrons en présentant la gestion des évènements ;
4. et un pointeur vers le **système** : ce qui permet à la gestion d'un évènement d'insérer elle-même un nouvel évènement dans la file du système. Par la suite, pour des raisons d'optimisation nous avons supprimé ce pointeur, et passons par la tâche qui possède un pointeur vers le système.

5.3.1 La gestion des évènements

Comme nous l'avons mentionné plus haut, il y a deux types d'évènements possibles. Lorsqu'un de ceux-ci se déclenche, l'algorithme fait appel à sa routine de gestion.

Voici comment celle-ci procède en fonction du type de l'évènement.

- *Arrivée* : Lorsqu'une arrivée survient pour la tâche T_i , on vérifie premièrement le respect de l'échéance, c.-à-d. qu'on vérifie que le travail précédent soit bien terminé, si tel n'est pas le cas, le système n'est pas ordonnançable et la simulation peut donc s'arrêter. Ensuite, on signale qu'un travail est prêt à être exécuté pour T_i et on insère l'arrivée suivante à l'instant $t + P_i$ dans la file des évènements, où t est l'instant courant.
- *FinJob* : Le travail qui se termine est marqué comme terminé.

L'insertion triée dans la file des évènements se base à la fois sur l'instant de déclenchement, mais aussi sur le type de l'évènement à rajouter. Il est indispensable de gérer les évènements qui se produisent au même instant dans l'ordre suivant :

1. *FinJob* ;
2. *Arrivée*.

La gestion de l'évènement « Arrivée » comprend également la gestion des échéances qui contrôle premièrement l'échéance du travail précédent avant d'en lancer un nouveau. Etant donné que nous travaillons sur des systèmes à échéance sur requête, il est inutile de gérer un évènement « Echéance ».

En effet, si la fin d'un travail survient au même instant que l'échéance de sa tâche, le système est ordonnançable. Il faut donc que le travail soit marqué comme terminé avant de tester s'il reste un travail pour cette tâche. Dans le cas contraire, la gestion de l'échéance nous renverrait qu'il y a encore un travail en cours d'exécution et que le système n'est donc pas ordonnançable, ce qui est

faux. Il ne faut donc pas non plus lancer un travail τ_{ij} avant de tester l'échéance de $\tau_{i,j-1}$, pour les mêmes raisons.

5.3.2 L'algorithme

Basant notre simulation sur des systèmes à départ simultané, au début de l'exécution de l'algorithme nous rajoutons dans la file d'évènements une *Arrivée* à l'instant 0 pour chaque tâche présente dans le système.

1. L'évènement au sommet de la file est sélectionné, c.-à-d. celui qui possède le plus petit instant de déclenchement. Cet évènement est géré de manière différente selon qu'il s'agisse d'une « Arrivée » ou d'une « Fin-Job » et est ensuite supprimé de la file. On gère tous les évènements qui se déclenchent à cet instant.
2. Après, l'algorithme sélectionne le travail le plus prioritaire, prêt à être exécuté. C'est ici qu'intervient la politique d'ordonnancement. Nous avons utilisé EDF comme règle d'assignation des priorités (voir section 2.1.5).
3. Le temps restant avant le prochain évènement (*rem*) qui se trouve au sommet de la file est calculé par rapport à l'instant courant. Si le temps d'exécution restant du travail (*remjob*) que l'on vient de sélectionner est inférieur ou égal au temps restant avant l'évènement suivant, nous insérons une FinJob à l'instant $t = currenttime + remjob$ où *currenttime* est l'instant actuel. Autrement nous savons que l'exécution de ce travail prioritaire ne sera pas terminée avant que le prochain évènement ne se déclenche. Dans ce cas, nous diminuons simplement le temps d'exécution restant du travail que l'on se prépare à lancer, $remjob = remjob - rem$.
4. Nous procédons au changement de contexte, c.-à-d. que le travail qui était exécuté est mis dans l'état « wait », les données de celui-ci sont sauvegardées, les données du travail prioritaire sont restaurées et le travail prioritaire est exécuté.
5. Nous reprenons à l'étape 1 jusqu'à atteindre l'hyperpériode à partir de laquelle nous savons que le système est ordonnançable si toutes les échéances jusqu'à cet instant (compris) ont été respectées.

Voici l'algorithme de la simulation basée sur les évènements en pseudo-code (algorithme 7).

```

currenttime ← 0 ;
Pour Chaque tâche T faire
  | insère Arrivée(T, instant 0)
Fin Pour
Tant que (currenttime ≤ hyperpériode) faire
  | Tant que (currenttime = temps de l'évènement au sommet) faire
  |   | EV ← évènement au sommet de la file
  |   | currenttime ← temps de l'évènement EV
  |   | gérer EV (voir section 5.3.1)
  |   | supprimer EV
  | Fait
  |   | PRIOR ← travail le plus prioritaire
  |   | rem ← temps de l'évènement suivant – currenttime
  |   | Si (remjob ≤ rem) Alors
  |   |   | insère FinJob(PRIOR, currenttime + remjob)
  |   | Sinon
  |   |   | remjob ← remjob – rem
  |   |   | /*remjob est une variable appartenant au travail le plus prioritaire
  |   |   | actuel et initialisée à son temps d'exécution.*/
  |   | Fin Si
  |   | travail courant ← état wait
  |   | PRIOR ← état run
Fait

```

Algorithme 7: Ordonnanceur basé sur la gestion d'évènements

Il serait possible de rendre ce simulateur plus général, de façon à ce qu'il puisse gérer les échéances arbitraires et les départs différés (ce qui ne nuit pas à l'optimalité d'EDF). Il faudrait alors en ajoutant un évènement « Echéance », séparer la gestion des arrivées dans laquelle on vérifiait que l'échéance soit respectée en même temps que la gestion de l'arrivée du travail suivant. L'instant de départ A_i pouvant être différent d'une tâche T_i à l'autre il faudra effectuer le test d'ordonnançabilité sur un intervalle : $[\min \{A_i\}, P + \max \{A_i\}]$. Cependant, ici nous nous focaliserons sur des systèmes à départ simultané et à échéance sur requête, inutile donc de rendre la simulation plus longue en temps de calcul.

5.4 Rappel du problème d'optimisation des temps optionnels

Nous allons maintenant reprendre le problème d'optimisation analysé dans le chapitre précédent.

Problème :

$$\text{maximiser} \quad \sum_{i=1}^n \frac{1}{b_i} \sum_{j=1}^{b_i} f_i(t_{ij}) \quad (5.1)$$

$$\text{à condition que} \quad 0 \leq t_{ij} \leq o_i \quad i = 1, \dots, n \quad j = 1, \dots, b_i \quad (5.2)$$

$$\text{Il existe un ordonnancement faisable avec les valeurs } \{m_i\} \text{ et } \{t_{ij}\}. \quad (5.3)$$

où b_i pour rappel, est le nombre de fois que l'on peut placer la période de la tâche T_i dans l'hyperpériode, soit $b_i = P/P_i$.

La fonction à maximiser représente la somme des moyennes des récompenses pour chaque tâche du système, la récompense d'un travail τ_{ij} étant obtenue par la fonction $f_i(t_{ij})$.

Si le facteur d'utilisation $U = \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$ alors un algorithme utilisant EDF comme politique d'ordonnancement trouverait un ordonnancement faisable. Mais étant donné que les t_{ij} peuvent être de durées différentes, nous ne pouvons pas utiliser cette condition nécessaire et suffisante sur la faisabilité de l'ordonnancement.

En effet, dans notre cas, le temps d'exécution $C_i = m_i + t_{ij}$ peut varier d'un travail à l'autre comme les t_{ij} ne sont pas forcément identiques pour une tâche T_i donnée, le calcul du facteur d'utilisation du système n'a donc plus de sens. Nous continuerons néanmoins à parler d'utilisation en ne prenant en compte que les parties obligatoires des tâches.

D'après nos contacts avec H. Aydin, il n'existe pas d'autres moyens que celui de simuler l'ordonnancement jusqu'à l'hyperpériode P . Il est néanmoins possible de poser une condition nécessaire supplémentaire de manière à ne pas avoir à simuler l'ordonnancement jusqu'à l'hyperpériode lorsque celle-ci n'est pas respectée.

$$\sum_{i=1}^n b_i m_i + \sum_{i=1}^n \sum_{j=1}^{b_i} t_{ij} \leq P \quad (5.4)$$

Si la condition 5.4 n'était pas respectée, cela reviendrait à dire qu'il y aurait plus d'unités d'exécution, parties obligatoires et optionnelles confondues, que le nombre d'unités disponibles entre l'instant 0 et l'hyperpériode. Cette condition n'est malheureusement pas suffisante pour garantir que le système est faisable.

Il est possible de réexprimer l'équation 5.4 de ce problème comme :

$$\sum_{i=1}^n \sum_{j=1}^{b_i} t_{ij} \leq d = P - \sum_{i=1}^n b_i m_i$$

Où d est la latitude maximum disponible après avoir déjà ordonnancé les parties obligatoires jusqu'à l'hyperpériode P . Nous pourrions donc ordonnancer dans ces d unités de temps oisives les parties optionnelles.

Il nous faut trouver les valeurs des t_{ij} qui maximisent l'équation 5.1 tout en respectant les contraintes formulées par les équations 5.2 et 5.3.

Comme nous l'avons montré au chapitre 4 (page 45), l'augmentation de la sécurité en fonction du temps est mieux représentée par une fonction convexe, mais le problème étant NP-COMPLET lorsque les fonctions de récompenses sont convexes, nous utiliserons des méthodes heuristiques pour trouver une solution se rapprochant le plus possible de la solution optimale. Nous citerons et décrirons dans ce chapitre les méthodes heuristiques de base permettant de répondre à ce problème d'optimisation.

5.5 Notations utilisées

Nous utiliserons quelques notations pour présenter ces algorithmes :

- x_n : est la solution courante à l'étape n , dans notre problème elle sera constituée de :
 - un tableau stockant les valeurs des t_{ij} ;
 - la récompense obtenue avec ces valeurs.

Remarque: Notons que nous utilisons ici le terme « solution » pour désigner ce qui n'est en réalité qu'une tentative de solution et non la solution finale au problème. Cette solution courante peut ne pas être ordonnancable et ne sera alors très certainement pas retenue comme la meilleure solution.

- X : est l'ensemble de toutes les solutions valables (qui respectent les contraintes 5.1, 5.2 et 5.3) dans lequel la solution finale sera choisie par l'algorithme utilisé.

- $V(x_n)$: est le voisinage de la solution x_n , c'est donc un ensemble de solutions qui se rapprochent par leurs caractéristiques de la solution x_n . Toute la difficulté étant de trouver une manière de définir ce rapprochement dans le contexte de l'ordonnancement des parties optionnelles. Nous expliquerons dans la section 5.8 comment nous obtenons une solution voisine.
- $F(x_n)$: représente le retour de la fonction objectif, autrement dit la récompense obtenue avec cette solution x_n . Dans notre contexte, cela représente donc la récompense accrue par les t_{ij} de la solution x_n (équation 5.1). Si le système n'est pas ordonnançable après avoir effectué une simulation avec x_n , $F(x_n)$ sera nul.

5.6 Le recuit simulé

Le principe de cet algorithme est basé sur les techniques de recuit utilisé en sidérurgie. Le métal est chauffé de manière à fournir une certaine énergie au système qui permet aux atomes de se déplacer. Les atomes se positionnent de façon à libérer le métal de toutes tensions internes dues aux déformations ayant été appliquées au métal. Ensuite la baisse de température fige lentement les atomes dans une position d'énergie minimale.

L'algorithme du recuit simulé gardera donc ce même principe. Au début de son exécution, l'algorithme autorise des changements importants de la solution courante ensuite la solution se fige peu à peu et le système finit par se geler dans un optimum global.

On part de la solution initiale x_0 , pour laquelle tous les temps optionnels (t_{ij}) valent 0, la récompense est donc minimale étant donné qu'on travaille sur des fonctions non-décroissantes. En toute logique, on pourrait même travailler avec des fonctions de récompense de la forme $f_i(0) = 0 \quad \forall i$ car une récompense n'est pas utile si la partie optionnelle est nulle.

A chaque étape n , une solution voisine x^* de la solution courante x_n (c'est-à-dire $\in V(x_n)$) est choisie au hasard, nous expliquerons à la section 5.8 comment nous sélectionnons la solution voisine.

Si la récompense accrue grâce à cette solution est meilleure qu'avec la précédente, c.-à-d. si $F(x^*)$ est meilleur que $F(x_n)$, x^* devient la nouvelle solution courante. Autrement, on tire au sort selon une certaine probabilité p , le fait de garder la solution x_n ou de choisir x^* comme nouvelle solution pour l'étape $n+1$.

Remarque: Même si ce nouvel ensemble de durées pour les parties optionnelles est moins intéressant au niveau de la récompense, cela pourrait devenir par après plus avantageux car nous pourrions rebondir sur une meilleure solution que x_n par la suite, c'est ce qu'on appelle la phase de diversification.

Nous notons p , la probabilité de prendre x^* comme nouvelle solution malgré le fait que la récompense obtenue avec x^* soit inférieure à celle obtenue avec la solution précédente x_n . La probabilité p diminue lorsque la température T baisse, et ce de manière à figer petit à petit la solution.

p est donc calculé en fonction de la température T mais aussi en fonction de la diminution de récompense ΔF subie en acceptant x^* comme nouvelle solution.

$$\Delta F = F(x_n) - F(x^*)$$

La fonction $p(T, \Delta F)$ a été reprise par analogie au recuit en sidérurgie, il s'avère expérimentalement que la distribution de Boltzmann ci-dessous est un bon choix pour calculer p [20].

$$p(T, \Delta F) = e^{-\frac{\Delta F}{T}}$$

La température T quant à elle est fonction du temps et diminue par palier de manière géométrique, tous les L tours de boucle de l'algorithme. On choisit une température initiale T_0 qui ne diminue pas durant L tours, ensuite la température diminue à $T_1 = \alpha T_0$ après L tours, $T_2 = \alpha^2 T_0$, et ainsi de suite. Après kL tours de boucle la température vaudra $T_k = \alpha^k T_0$.

Les valeurs T_0 , L et α doivent être choisies expérimentalement :

- La température initiale T_0 est calculée par rapport à la probabilité p souhaitée au début de l'exécution de l'algorithme.

Exemple : Si l'on souhaite obtenir une probabilité de 50% d'accepter une solution qui est moins bonne que la solution courante, il faut que $e^{-\frac{\langle \Delta F \rangle}{T_0}} = 50\%$, autrement dit, que $T_0 = \frac{\langle \Delta F \rangle}{\ln 2}$.

$\langle \Delta F \rangle$ est la moyenne des détériorations déduite par simulation. Avant l'exécution à proprement parlé du recuit simulé, nous effectuons 100 tours de boucle afin de calculer cette moyenne et de fixer la valeur T_0 .

- L , dépend de l'importance du voisinage de la solution. En effet, la phase d'exploration devra durer plus longtemps avant de diminuer la température si le nombre de solutions voisines est important. Le nombre de possibilités de solutions voisines dépend ici du nombre de travaux présents jusqu'à

l'hyperpériode, qui est proportionnel au nombre de tâches dont est composé le système. Nous avons donc décidé de donner comme valeur à L ce nombre de tâches.

- Plus α est important, plus le refroidissement sera lent, nous avons choisi $\alpha = 0,95$.

Ci-dessous, nous avons repris un algorithme de base (issu de [20]) en le modifiant légèrement, sans entrer dans les détails du calcul des variables, pour présenter de manière intuitive le principe de fonctionnement de celui-ci au lecteur.

```

 $\hat{F} \leftarrow F(x_0); (x_0 \in X, \text{ solution initiale})$ 
 $n \leftarrow 0;$ 
Tant que ( vrai) faire
   $x^* \leftarrow$  une solution tirée au sort de  $V(x_n);$ 
  Si  $(F(x^*) \geq F(x_n))$  Alors
     $x_{n+1} \leftarrow x^*;$ 
    Si  $(F(x^*) > \hat{F})$  Alors  $\hat{F} \leftarrow F(x^*);$  Fin Si
  Sinon
     $q \leftarrow \text{random}(0, 1);$   $q$  reçoit un nombre au hasard entre 0 et 1
    Si  $(q \leq p)$  Alors
       $x_{n+1} \leftarrow x^*;$ 
    Sinon
       $x_{n+1} \leftarrow x_n;$ 
    Fin Si
  Fin Si
  Si (condition d'arrêt) Alors STOP ; Fin Si
   $n \leftarrow n + 1;$ 
Fait

```

Algorithme 8: Recuit simulé - Squelette de base

où \hat{F} est la sauvegarde du meilleur retour obtenu par la fonction objectif.

La *condition d'arrêt* peut être de différents types :

- Pour reprendre l'analogie avec le recuit en sidérurgie, nous pourrions nous arrêter lorsque la température est descendue en dessous d'une certaine borne ;
- ou lorsque la fonction F n'a pas augmenté de 0,1% pendant 100 paliers

- consécutifs ;
- mais afin de comparer le plus justement possible le recuit simulé avec la recherche tabou (que nous analyserons à la section 5.7), nous stopperons la recherche après un nombre fini d'itérations (1000) afin d'attribuer le même temps processeur aux deux algorithmes.

5.7 La recherche tabou

Le principe de la recherche tabou est d'associer à l'algorithme de base, une mémorisation des solutions visitées. L'algorithme de la recherche tabou ne prend pas en compte le facteur température utilisé dans le recuit simulé. La mémoire est constituée d'une liste « tabou », qui empêche de multiples cycles dans l'algorithme, elle est appelée tabou car une solution dont les caractéristiques correspondent à celles rencontrées dans la mémoire ne sera pas reprise comme solution courante. Nous parlons ici de caractéristiques car il est possible de limiter la taille de la mémoire tabou en ne prenant en compte que quelques caractéristiques au lieu de devoir comparer tout le tableau des valeurs t_{ij} . Nous enregistrerons dans cette liste les permutations effectuées sur les parties optionnelles lors de l'obtention d'une solution voisine (voir section 5.8).

La liste tabou est gérée de manière FIFO (First in / First out), ce qui veut dire que les changements récents remplacent les plus anciens, cela empêche donc de refaire un changement que l'on vient d'effectuer et de cycler sur les permutations. Cependant, il est à noter qu'un cycle plus long que la taille de la mémoire reste possible, mais nous avons remarqué qu'avec une taille suffisante de permutations enregistrées (à partir de 10, valeur que nous avons choisie pour notre implémentation de l'algorithme) les situations de cycles restent rares, le nombre de choix parmi les temps optionnels des travaux à permuter pouvant être important.

A chaque étape n , on extrait du voisinage de la solution courante $V(x_n)$, un sous-voisinage que nous notons V^* . Une fois V^* extrait, toutes les solutions de cet échantillon sont comparées et la meilleure solution (x^*) dont les caractéristiques ne sont pas tabou, est enregistrée dans la mémoire et comme nouvelle solution courante x_{n+1} . Le fait que x_n offre une meilleure récompense n'a pas d'importance. Cela permet de garder le principe d'exploration des solutions et de ne pas rester calé dans un maximum local. Cependant il est possible de rajouter des critères qui pourraient autoriser une solution tabou à être réutilisée. Une augmentation forte de la récompense pourrait être un critère. Nous

réutilisons une solution tabou si toutes les solutions de V^* sont tabou, nous sélectionnons alors la meilleure parmi celles-ci.

Si c'était possible, on prendrait $V^* = V(x_n)$ mais dans notre cas, cela serait fort coûteux en temps de calcul de trouver la meilleure solution de tout le voisinage. Nous préférons donc tirer au sort un échantillon de $V(x_n)$ afin d'obtenir V^* .

Nous nous sommes à nouveau basé sur l'algorithme issu de l'ouvrage [20] que nous avons légèrement modifié pour présenter la structure de base de la recherche tabou que voici :

```

 $\hat{F} \leftarrow F(x_0)$ ; ( $x_0 \in X$ , solution initiale)
Tant que (vrai) faire
   $V^* \leftarrow$  sous-voisinage de  $V(x_n)$ ;
   $x^* \leftarrow$  meilleure solution de  $V^*$ ;
  Tant que ( $x^* =$  solution tabou) faire
     $x^* \leftarrow$  meilleure solution suivante de  $V^*$ ;
  Fait
  Si ( $x^*$  est tabou  $\forall x^* \in V^*$ ) Alors
     $x^* \leftarrow$  meilleure solution de  $V^*$ ;
  Fin Si
   $x_{n+1} \leftarrow x^*$ ;
  mise à jour de la liste tabou;
  Si (condition d'arrêt) Alors STOP; Fin Si
   $n \leftarrow n + 1$ ;
Fait

```

Algorithme 9: Recherche Tabou - Squelette de base

Nous avons effectué des simulations afin de déterminer la taille idéale L du sous-voisinage V^* . Cette taille dépend du nombre de tâches étant donné que chaque tâche possède sa propre fonction de récompense. Nous avons donc choisi de calculer cette taille par la formule 5.5.

$$L = x \cdot nbatches \quad (5.5)$$

où $nbatches$ est le nombre de tâches présentes dans le système et x est un coefficient que nous avons déduit par simulation.

Nous avons effectué 1000 simulations de l'algorithme de la recherche tabou pour plusieurs valeurs de x différentes, sur un même système composé de 6

tâches. Nous avons comparé la récompense moyenne par rapport à l'algorithme optimal, pour chaque valeur de x , voir graphique 5.1. Nous en sommes arrivés à la conclusion que la meilleure valeur pour la taille de V^* est $L = 41 \cdot nb\text{taches}$

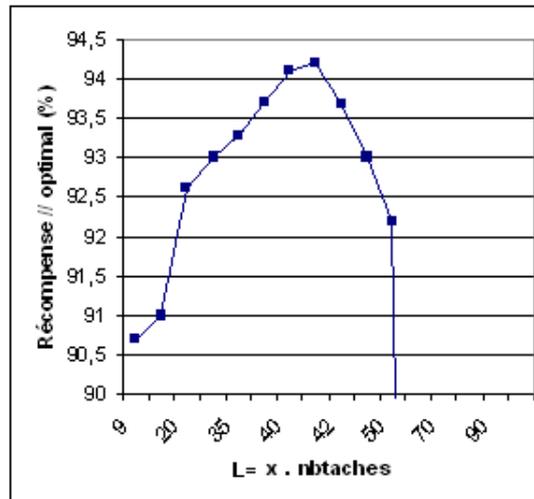


FIG. 5.1 – Recherche de la meilleure taille du sous-voisinage

5.8 Obtention d'une solution voisine

Sur base d'une solution x_n , l'algorithme du recuit simulé doit construire une solution voisine x^* qui se rapproche par ses caractéristiques de la solution x_n , l'algorithme de la recherche tabou doit fournir un voisinage $V(x_n)$, c.-à-d. un ensemble de solutions voisines de x_n .

Après avoir simulé le système avec la solution x_n , nous procédons comme suit pour obtenir une solution voisine :

1. Si le système est **ordonnançable** avec la solution x_n , on sélectionne un t_{ij} au hasard dans le tableau des temps optionnels et nous l'augmentons d'un certain palier sans dépasser la valeur maximale des temps optionnels (o_i) pour T_i . En effet, si $t_{ij} + palier > o_i$ alors $t_{ij} \leftarrow o_i$

2. Si la contrainte nécessaire 5.4 :

$\sum_{i=1}^n b_i m_i + \sum_{i=1}^n \sum_{j=1}^{b_i} t_{ij} \leq P$ a été violée, le système n'est pas ordonnançable, on sélectionne alors un t_{ij} au hasard parmi un ensemble S constitué des travaux optionnels pour lesquels $t_{ij} - palier > 0$ et $t_{ij} \leftarrow t_{ij} - palier$.

En effet, diminuer les temps optionnels est dans ce cas le seul moyen de récupérer un système ordonnançable.

S'il n'existe pas de tels travaux (S est vide), rien n'est diminué mais lorsque le palier aura atteint une valeur suffisamment petite, S ne sera plus vide.

- De plus, nous permutons (avec une probabilité de 50%) deux t_{ij} du tableau afin de diversifier la recherche pour que l'algorithme ne reste pas calé dans un maximum local.

Le palier dépend de la tâche et est sélectionné au hasard entre 0 et $paliermax_i$. $paliermax_i$ est la valeur maximale que peut atteindre le palier de T_i . Le palier est sélectionné entre ces deux bornes de manière à ce que le palier ne soit pas contraint à être uniquement diminué. Nous initialisons $paliermax_i$ à $\min(o_i, P_i - m_i)$, c.-à-d. le temps maximum restant pour la partie optionnelle de T_i .

Nous avons donc décidé d'après nos tests, de diminuer $paliermax_i \forall T_i$ de manière géométrique avec un facteur $diminpalier = 0.97$ ($paliermax_i \leftarrow paliermax_i \cdot diminpalier$) à chaque fois que trois simulations consécutives renvoient que le système n'est pas ordonnançable. Pour déterminer ce facteur, nous avons fixé 3 comme le nombre de fois qu'une tentative de solution obtenue peut ne pas être ordonnançable avant de diminuer le palier. Ensuite, à partir de cette constante fixée, nous avons effectué des tests avec différents facteurs et comparé la récompense obtenue par rapport à l'algorithme optimal (voir figure 5.2).

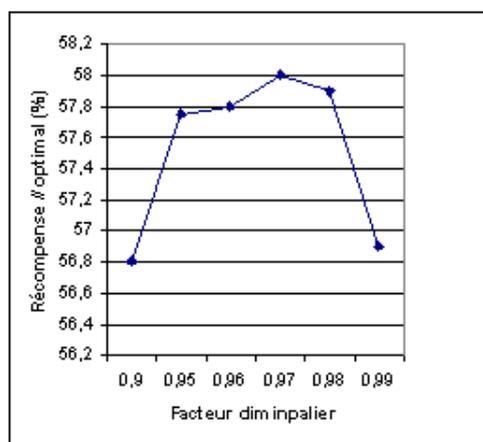


FIG. 5.2 – Ajustement du facteur de diminution du palier maximum

5.9 Expérimentation

Nous avons effectué des tests afin de comparer la récompense obtenue par les deux méthodes heuristiques. Pour cela, nous avons d'abord généré différents fichiers contenant chacun les spécifications de 1000 systèmes tous composés de n tâches.

Les résultats produits par nos tests sur de petits ensembles de 100 systèmes divergeaient d'une simulation à l'autre mais restaient identiques entre eux avec un ensemble de 1000 systèmes et similaires aux résultats produits par un ensemble de 10000 systèmes, nous avons donc estimé qu'il s'agissait d'un bon compromis entre le temps de calcul et la qualité du résultat.

Pour chaque tâche, nous enregistrons la période, le temps d'exécution obligatoire, le temps optionnel maximum et le coefficient utilisé dans la fonction de récompense.

Les tests ont été faits en donnant le même temps processeur aux deux algorithmes¹, le temps processeur dépendant essentiellement du nombre de simulations effectuées avec les différentes solutions. Nous nous limiterons à 1000 simulations par algorithme, soit à 900 itérations de la boucle principale pour le recuit simulé car 100 itérations sont utilisées pour déterminer la température initiale et à $\frac{1000}{L}$ pour la recherche tabou étant donné qu'il faut comparer et donc simuler L (la taille du sous-voisinage V^*) systèmes à chaque tour de boucle.

5.9.1 Limites des algorithmes

Nous remarquons que l'efficacité de l'algorithme dépend principalement du nombre de travaux à simuler jusqu'à l'hyperpériode et ce pour deux raisons, la présence d'un nombre important de travaux :

1. nous oblige à limiter le temps d'exécution des algorithmes, ce qui les rendent moins efficaces ;
2. provoque de mauvais choix effectués au hasard sur ce grand nombre de travaux.

¹implémentations du recuit simulé : `recuit()` et de la recherche tabou : `tabousearch()` dans l'annexe B, section B.1.

Problème de lenteur d'exécution

Nous avons fortement été ralenti par l'optimisation de la simulation et le choix de structures de données adéquates :

- utilisation d'une file à priorité pour gérer les évènements ;
- stockage des évènements et travaux statiquement pour éviter les allocations et désallocations continues de la mémoire ;
- enregistrement des modifications effectuées sur le tableau des temps optionnels afin de ne pas devoir recopier entièrement le tableau à chaque fois que l'on revient à une solution précédente, . . .

A chaque travail correspond un évènement « Arrivée » et « FinJob ». S'il y a 100 travaux à simuler jusqu'à l'hyperpériode, il y aura donc pour chaque simulation avec une solution x_n , 200 évènements à gérer. Comme nous autorisons l'algorithme à faire 1000 simulations, qu'il y a 1000 systèmes différents par fichier et que nous testons nos 2 méthodes sur 12 fichiers différents, chaque fichier se distinguant par le nombre de tâches des systèmes qui composent celui-ci. Un rapide calcul permet de conclure qu'une instruction de $1\mu s$ dans la gestion des évènements équivaut à $10^{-6}s \times 200 \times 1000 \times 1000 \times 2 \times 12 = 4800s = 80$ minutes supplémentaires sur la simulation totale.

Problème du choix des temps optionnels à modifier

Il est bien entendu évident que si le nombre de travaux optionnels est important, le choix de la partie optionnelle dont on va augmenter le temps d'exécution est primordial et cela reste le cas malgré le fait qu'on joue sur la granularité en augmentant fortement les temps optionnels au début de l'exécution de l'algorithme et en diminuant cette augmentation par le système de palier.

Or, le recuit simulé choisi au hasard t_{ij} qu'il va modifier. Si le choix se porte sur un travail τ_{ij} qui n'augmente que de très peu la récompense, à l'étape suivante nous risquons de continuer à travailler sur une solution qui n'est pas très intéressante. Dans la recherche tabou, c'est la solution de V^* qui rapporte le plus au système qui sera sélectionnée avant de passer à l'étape suivante. C'est pourquoi la recherche tabou fournit un bien meilleur résultat lorsque le nombre de tâches est important. En effet, si le nombre de tâches est important, la probabilité d'avoir plus de travaux est beaucoup plus importante également et l'algorithme aura plus de chance de choisir un travail dont la tâche possède un coefficient plus élevé dans la fonction de récompense.

Voilà pourquoi nous avons essayé de réduire le nombre de travaux par la

méthode donnée à la section 5.2 en choisissant la période de manière plus sélective que totalement aléatoirement entre deux bornes.

5.9.2 Analyse avec des fonctions linéaires de récompense

La comparaison est faite par rapport à l'écart de récompense entre le recuit simulé, la recherche tabou et la méthode optimale (présentée au chapitre 4) selon le nombre de tâches. Pour rappel, la méthode optimale n'est effectivement optimale que lorsque les fonctions de récompense sont linéaires. Dans ce cas, les méthodes heuristiques n'ont donc aucun intérêt, mais c'est notre seul moyen de comparaison étant donné que la méthode optimale dans le cas linéaire ne l'est plus avec des fonctions de récompense convexes.

Nous avons donc premièrement effectué un test où chaque tâche possédait une fonction linéaire de récompense de la forme :

$$f(t_{ij}) = k_i \cdot t_{ij} \quad (5.6)$$

Les résultats de nos simulations sont représentés sur la figure 5.3, l'abscisse représente le nombre de tâches dont sont composés les 1000 systèmes et l'ordonnée représente le rapport moyen de la récompense sur ces 1000 systèmes par rapport à l'algorithme optimal (algorithme étalon).

Nous remarquons que le rapport diminue en fonction du nombre de tâches comme nous l'avons expliqué dans la section précédente et que la recherche tabou fournit effectivement de meilleurs résultats que le recuit simulé, malgré que la récompense moyenne totale vaille $\approx 80\%$ de celle obtenue avec l'algorithme optimal sur des systèmes de 12 tâches générés comme présenté à la section 5.2.2. Ce rapport est néanmoins une bonne performance pour une méthode heuristique.

5.9.3 Simulation avec des fonctions convexes de récompense

Ensuite, nous avons trouvé intéressant de comparer l'efficacité des algorithmes lorsque les fonctions de récompense sont de type convexe étant donné que le niveau de la sécurité d'un système et donc la récompense augmente de manière convexe en fonction du temps utilisé pour appliquer celle-ci. Ce serait donc selon nous le type de fonction qui pourrait caractériser au mieux l'augmentation de la valeur en sécurité en fonction du temps. Nous avons choisi pour cela

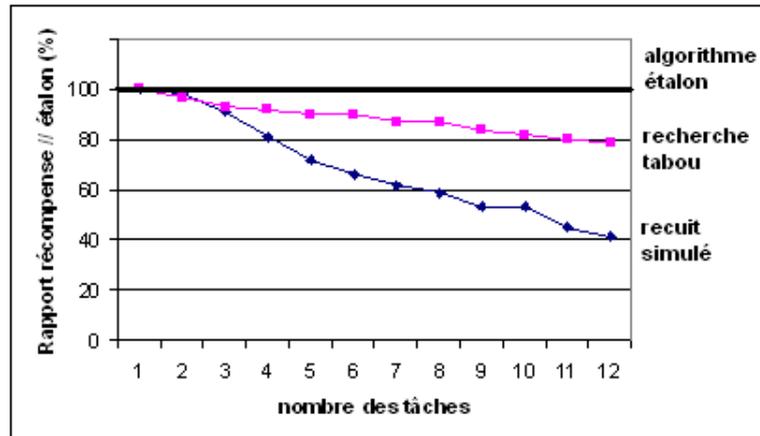


FIG. 5.3 – Cas linéaire — Comparaison entre le recuit simulé, la recherche tabou et l’algorithme optimal

une fonction de la forme :

$$f(t_{ij}) = k_i \cdot t_{ij}^2, \quad (5.7)$$

Nous avons réutilisé les mêmes ensembles de systèmes pour effectuer cette simulation, le coefficient k_i reste donc également identique (sélectionné pour rappel entre 1 et 100).

Nous nous rendons compte que nos méthodes heuristiques ne fournissent un meilleur rapport de récompense par rapport à l’algorithme étalon (l’algorithme optimal avec des fonctions linéaires) que lorsque le nombre de tâches est très faible (2 ou 3 tâches). Ce rapport décroît très vite lorsque le nombre de tâches augmente étant donné que lorsque le nombre de tâches augmente, le nombre de travaux jusqu’à l’hyperpériode deviendra plus important. Une erreur dans le choix des parties optionnelles à modifier est alors beaucoup plus pénalisante que dans le cas des fonctions linéaires de récompense.

Nous avons également comparé nos deux méthodes par rapport à l’algorithme de la descente en cascade². Cet algorithme ressemble à l’algorithme du recuit simulé, excepté qu’on ne tient pas compte de l’évolution de la température. Lorsqu’une solution n’est pas meilleure que la précédente on ne la reprend pas à l’étape suivante, autrement dit la probabilité p (utilisée dans le recuit simulé) de continuer avec une solution fournissant une moindre récompense est nulle.

²voir fonction `desc_cascade()` dans l’annexe B, section B.1.

Nous constatons que l'algorithme de la descente en cascade offre pour ce problème une meilleure récompense moyenne que le recuit simulé. Ceci est dû au fait qu'en acceptant dans le recuit simulé de repartir à l'itération suivante sur une solution fournissant une moins bonne récompense, nous risquons aussi de partir sur une mauvaise voie. Nous remarquons qu'il est donc préférable de n'accepter la solution obtenue que si celle-ci est meilleure que la solution précédente, ce qui nous amène à la conclusion que le recuit simulé n'est pas adapté à notre problème.

Cela dit, il est évident que la récompense moyenne fournie par les méthodes heuristiques est supérieure en comparaison à celle fournie par un algorithme qui choisirait au hasard les valeurs t_{ij} sans aucune forme d'intelligence et qu'on laisserait s'exécuter sur le même temps processeur que les méthodes heuristiques citées plus haut. En effet, le système ne serait pas ordonnançable la plupart du temps et la récompense serait alors nulle.

Les résultats de cette simulation sont représentés sur la figure 5.4.

La recherche tabou surpasse à nouveau largement le recuit simulé pour la même raison que dans le cas linéaire, mais nous ne sommes pas parvenus à dépasser la récompense moyenne obtenue avec l'algorithme étalon.

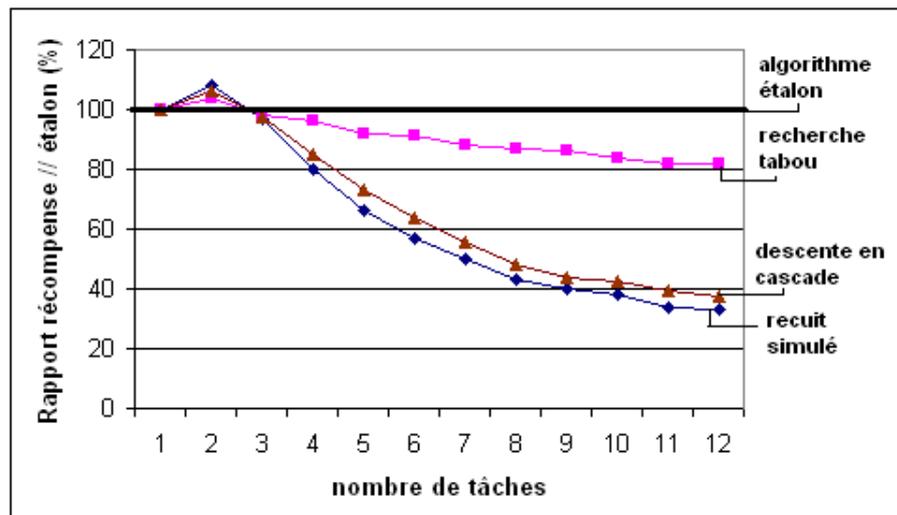


FIG. 5.4 – Cas convexe — Comparaison entre le recuit simulé, la recherche tabou, l'algorithme de la descente en cascade et l'algorithme étalon en fonction du nombre de tâches

Comparaison par rapport à l'utilisation

Nous avons ensuite généré 10 fichiers contenant chacun les spécifications de 1000 systèmes. Les systèmes enregistrés dans un fichier possèdent tous le même facteur d'utilisation U_m (par les parties obligatoires), ces systèmes sont générés grâce à l'algorithme 6 présenté à la section 5.2. U_m varie d'un fichier à l'autre entre 0,1 et 1 par pas de 0,1, le nombre de tâches pouvant ici être différent d'un système à l'autre pour un même fichier.

Si l'utilisation augmente, cela signifie que le rapport entre la durée des parties obligatoires et de la période s'intensifie, il y aura donc moins de temps disponible pour l'exécution des parties optionnelles. Cela implique que la récompense attribuée à l'ordonnancement s'en verra diminuée étant donné que celle-ci n'est attribuée qu'à la partie optionnelle.

Pour montrer cela, nous avons effectué une simulation sur laquelle les tâches possèdent toutes la même fonction de récompense : $f(t_{ij}) = 2^{t_{ij}/300}$. Nous avons choisi une fonction exponentielle de manière à nous rapprocher le plus possible de l'évolution de la valeur en sécurité du système en fonction du temps.

Remarque: L'exposant t_{ij} est divisé par 300 de manière à réduire considérablement la récompense, étant donné que les parties optionnelles peuvent être relativement importantes lorsque l'utilisation par les parties obligatoires est faible. Si cette division n'était pas effectuée, le résultat du calcul de la récompense serait trop important.

Nous présentons les résultats de cette simulation sur la figure 5.5. Sur ce graphe dont l'abscisse est graduée à l'échelle logarithmique, nous constatons que la récompense moyenne régresse effectivement de manière exponentielle en augmentant le facteur d'utilisation U_m .

Lorsque $U_m = 1$, la partie optionnelle est nulle et il en découle donc également une récompense nulle, quelque soit l'algorithme utilisé. Dans ce cas, le rapport est de 100% par rapport à l'algorithme étalon aussi bien pour le recuit simulé que pour la recherche tabou.

Cependant, lorsqu'il y a des parties optionnelles à ordonnancer, les performances des méthodes heuristiques dépendent encore du nombre de tâches comme nous l'avons vu plus haut. Ceci explique l'irrégularité des résultats obtenus sur la figure 5.6 étant donné que pour réussir à obtenir l'utilisation désirée, le nombre de tâches n'est pas fixé et varie d'un système à l'autre, il suffit par exemple qu'un fichier comporte beaucoup de systèmes composés de deux tâches pour que le rapport de récompense entre les heuristiques et l'algorithme

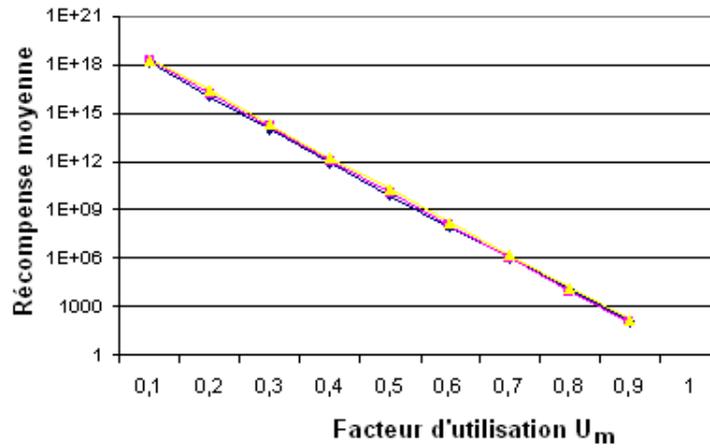


FIG. 5.5 – Récompense en fonction du facteur d'utilisation

étalon soit plus élevé.

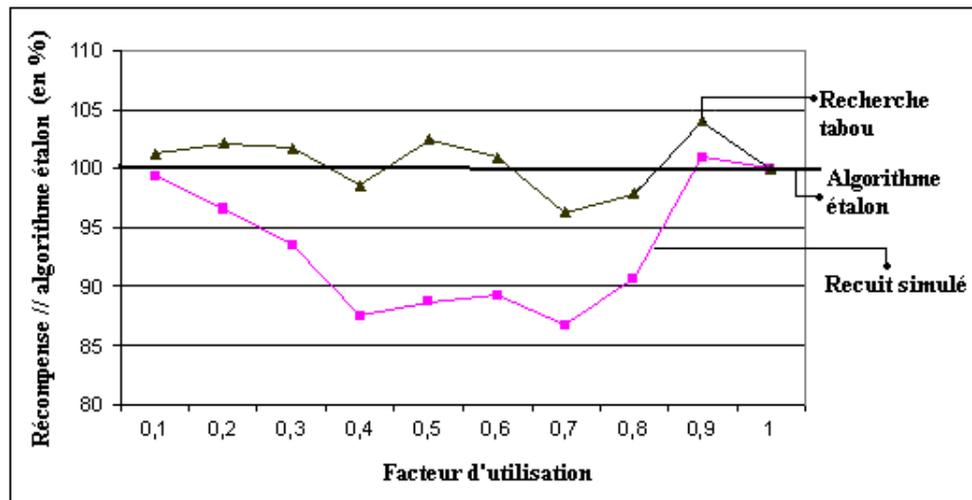


FIG. 5.6 – Cas convexe — Rapport de la récompense en fonction de l'utilisation

5.10 Conclusion

Nous avons expliqué dans ce chapitre comment nous avons effectué nos simulations. Nous avons montré comment les tâches sont générées et avons présenté la méthode basée sur des événements qui fait appel à l'ordonnan-

ceur EDF. Une fois la simulation de l'ordonnancement présentée, nous avons développé deux méthodes heuristiques : le recuit simulé et la recherche tabou qui utilisent notre simulation de manière à résoudre le problème d'optimisation évoqué au chapitre 4 et décrit à nouveau dans ce chapitre en utilisant ici des fonctions de récompense convexes.

D'après les résultats de ces simulations que nous avons exposés à la section 5.9, nous avons fait remarquer que le recuit simulé n'est pas adapté à ce problème. Etant donné la taille du problème, il est préférable d'effectuer à chaque étape des comparaisons entre différentes solutions possibles avant d'en choisir une pour servir de base à l'étape suivante. L'algorithme de la recherche tabou effectue ces comparaisons et offre donc de meilleurs résultats au niveau de la récompense apportée par la sécurité. Nous avons prouvé par nos résultats que la qualité de la récompense dépend du nombre de tâches présentes dans le système et que cette récompense diminue en fonction du facteur d'utilisation des parties obligatoires.

Chapitre 6

Conclusion

Nous avons présenté dans ce mémoire différentes méthodes permettant de sécuriser par des moyens cryptographiques un système temps réel tout en contrôlant le respect des échéances des différentes tâches composant ce système.

Remarquons que la littérature est encore pauvre à ce sujet, nous entendons par là que beaucoup d'articles existent à propos de la cryptographie, et de très nombreux articles se concentrent sur l'ordonnancement temps réel mais le lien entre ces deux disciplines n'est que très rarement établi.

Après avoir analysé et expliqué les points faibles de différentes méthodes existantes dans lesquelles on retrouve la notion de niveaux de sécurité, nous nous sommes basés sur l'idée d'offrir une récompense de manière proportionnelle à la durée allouée pour mettre en place la sécurisation du système. Dans ce but, nous nous sommes appuyés sur une technique nommée « Reward Based Scheduling » [3] dont le principe est de diviser le travail d'une tâche en une partie obligatoire et une autre facultative en offrant une récompense dépendant de la longueur de la partie facultative. La sécurité serait affectée à la partie optionnelle tout en intégrant une sécurité minimum dans la partie obligatoire.

Nous avons montré que l'évolution de la qualité de la sécurité peut se caractériser par une fonction convexe exponentielle lorsque la sécurité évolue selon la taille des clés de chiffrement employées.

Nous nous sommes alors retrouvé devant un problème NP-complet qui est d'allouer une certaine durée pour l'exécution de la partie optionnelle de chaque travail, cette durée pouvant être différente d'un travail à l'autre pour une même tâche et l'ordonnancement devant rester faisable avec ces parties optionnelles.

Il nous a fallu nous pencher sur des méthodes heuristiques qui essayent en augmentant ou en diminuant étape par étape la longueur des parties optionnelles de se rapprocher de la meilleure récompense qui soit. Nous avons eu l'occasion de nous rendre compte que la programmation des méthodes heuristiques est ardue car celles-ci dépendent de nombreuses variables qu'il faut ajuster pour améliorer les performances de l'algorithme. Nous avons montré que la recherche tabou en comparaison au recuit simulé donne de meilleurs résultats dans ce cadre.

Pour conclure ce mémoire, nous avons comparé nos résultats avec ceux de l'algorithme présenté par H. Aydin dans [3], ceux-ci restent supérieurs aux nôtres dans de nombreux cas, mais nous avons néanmoins montré par nos simulations que dans certaines circonstances (par exemple, lorsque le nombre de tâches est faible ou lorsque l'utilisation par les parties obligatoires est importante) il est possible d'obtenir une récompense plus élevée en utilisant des méthodes heuristiques, d'où l'intérêt porté à ces méthodes. Celles-ci restent actuellement le seul moyen de résoudre des problèmes d'optimisation comme celui développé dans ce mémoire.

Bibliographie

- [1] AHMED, Q. N., AND VRBSKY, S. V. Maintaining security in firm real-time database systems. *acsac 00* (1998), 83.
- [2] AVOINE, G. RFID et sécurité font-elles bon ménage ? In *Conférence SS-TIC* (2006).
- [3] AYDIN, H., MELHEM, R., MOSSÉ, D., AND MEJIA-ALVAREZ, P. Optimal reward-based scheduling for periodic real-time tasks. *IEEE TRANSACTIONS ON COMPUTERS* 50, 2 (feb 2001).
- [4] DAI, W. Librairie crypto++ 5.4. <http://www.cryptopp.com/>,.
- [5] DEY, J. K., KUROSE, J., AND TOWSLEY, D. On-line scheduling policies for a class of IRIS (Increasing Reward with Increasing Service) real-time tasks. *IEEE TRANSACTIONS ON COMPUTERS* 45, 7 (jul 1996).
- [6] ENGEN, M. Scheduling periodic tasks in real-time system that allow imprecise results. Norwegian University of Science and Technology, 2006.
- [7] GOOSSENS, J. Techniques avancées de systèmes d'exploitation. Faculté des Sciences, Département Informatique, Université Libre de Bruxelles, 2004-2005.
- [8] GOOSSENS, J., AND MACQ, C. Limitation of the hyper-period in real-time periodic task set generation. In *Proceedings of RTS'2001, Paris, France* (2001), teknea, Ed., pp. 133–148.
- [9] KANG, K. D., AND SON, S. H. Towards security and qos optimization in real time. *ACM SIGBED Review*.
- [10] KANG, K.-D., AND SON, S. H. Systematic security and timeliness tradeoffs in real-time embedded systems. *rtcsa 0* (2006), 183–189.
- [11] KOREN, G., AND SHASHA, D. Skip-over : algorithms and complexity for overloaded systems that allow skips. *rtss 00* (1995), 110.
- [12] LIN, M., AND YANG, L. T. Schedulability driven security optimization in real-time systems. In *ARES '06 : Proceedings of the First International*

- Conference on Availability, Reliability and Security (ARES'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 314–320.
- [13] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (1973), 46–61.
- [14] MARCHAND, A., AND SILLY-CHETTO, M. Simulation et évaluation d'algorithmes d'ordonnancement temps-réel sous des contraintes de QoS. Laboratoire d'Informatique de Nantes-Atlantique, Septembre 2004.
- [15] PENTTONEN, M. Data security 2003. <http://www.cs.uku.fi/~junolain/secu2003/secu2003.html>.
- [16] PERRIG, A., STANKOVIC, J., AND WAGNER, D. Security in wireless sensor networks. *Commun. ACM* 47, 6 (2004), 53–57.
- [17] SCHNEIER, B. *Applied cryptography : protocols, algorithms, and source code in C*, 2nd ed. Wiley, New York, 1996.
- [18] SHIH, W. K., AND LIU, J. W.-S. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Transactions on Computers* 44, 3 (1995), 466–471.
- [19] STURM, G. Encryption standards : Aes vs. des. <http://www.logic.at/lvas/krypto/Sturm/html/index.html>.
- [20] TEGHEM, J., AND PIRLOT, M. *Optimisation approchée en recherche opérationnelle*. Hermes Science, May 2002.
- [21] U.S. DEPARTMENT OF COMMERCE/N.I.S.T. Processing Standards Publication 197 - Advanced Encryption Standard (AES).
- [22] WAGNER, D. The boomerang attack. *Lecture Notes in Computer Science* 1636 (1999), 156–170.
- [23] WIKIPEDIA. Fonction convexe. http://fr.wikipedia.org/wiki/Fonction_convexe.
- [24] WIKIPEDIA. Théorie de la complexité. http://fr.wikipedia.org/wiki/Th%C3%A9orie_de_la_complexit%C3%A9.
- [25] XIE, T., AND QIN, X. Scheduling security-critical real-time applications on clusters. *IEEE Trans. Comput.* 55, 7 (2006), 864–879.
- [26] XIE, T., QIN, X., AND SUNG, A. An approach to satisfying security needs of periodic tasks in high performance embedded systems. *The 12th Annual IEEE International Conference on High Performance Computing (HiPC 2005, poster session)*.
- [27] XIE, T., SUNG, A., AND QIN, X. Dynamic task scheduling with security awareness in real-time systems. *ipdps 16* (2005), 274a.

Annexe A

Complément à la démonstration de H. Aydin

Nous allons présenter ici la preuve que la récompense ne décroît pas lorsqu'une fonction concave de récompense $f_i(t)$ est utilisée avec des temps optionnels de tailles identiques pour tous les travaux de la tâche T_i . Cette preuve est utilisée dans la démonstration du théorème 2 (page 55) et est extraite de l'article [3].

Il faut donc prouver que :

$$\sum_{j=1}^{b_i} f_i(t_{ij}) \leq b_i f_i(t'_i), \quad (\text{A.1})$$

où $t'_i = \frac{t_{i1} + t_{i2} + \dots + t_{ib_i}}{b_i}$ et b_i est le nombre de travaux de la tâche T_i jusqu'à l'hyperpériode.

Démonstration: Si $f_i(t)$ est une fonction linéaire de la forme $f_i(t) = k_i \cdot t$, alors $\sum_{j=1}^{b_i} f_i(t_{ij}) = k_i (t_{i1} + t_{i2} + \dots + t_{ib_i}) = k_i b_i t'_i$, ce qui vérifie l'équation A.1.

Observons à présent le cas plus général, où f_i est concave et pourrait être non-linéaire avec la définition d'une fonction concave :

$$\alpha f_i(x) + (1 - \alpha) f_i(y) \leq f_i(\alpha x + [1 - \alpha] y), \quad \forall x, y \quad 0 \leq \alpha \leq 1 \quad (\text{A.2})$$

La preuve de la validité de A.1 est faite par induction. Si $b_i = 1$, l'inégalité A.1 est trivialement vérifiée. Supposons qu'elle tienne toujours avec $b_i = 2, 3 \dots m - 1$.

Cette hypothèse implique d'après A.1 que :

$$\sum_{j=1}^{m-1} f_i(t_{ij}) \leq (m-1) f_i \left(\frac{t_{i1} + \dots + t_{i(m-1)}}{m-1} \right). \quad (\text{A.3})$$

Choisissons $\alpha = \frac{m-1}{m}$, $x = \frac{t_{i1} + \dots + t_{i(m-1)}}{m-1}$, $y = t_{im}$, en remplaçant dans A.2, nous pouvons écrire :

$$\frac{m-1}{m} f_i \left(\frac{t_{i1} + \dots + t_{i(m-1)}}{m-1} \right) + \frac{1}{m} f_i(t_{im}) \leq f_i \left(\frac{t_{i1} + \dots + t_{i(m)}}{m} \right). \quad (\text{A.4})$$

En combinant les équations A.3 et A.4, nous obtenons :

$$\frac{1}{m} \sum_{j=1}^m f_i(t_{ij}) \leq f_i \left(\frac{t_{i1} + \dots + t_{i(m)}}{m} \right) \quad \forall m. \quad (\text{A.5})$$

Ce qui établit la validité de A.1. □

Annexe B

Code source de la simulation

Nous présentons dans cette annexe le code source C++ de notre simulation ayant servi à obtenir les résultats exposés à la fin du chapitre 5.

B.1 Système

La classe système est la classe principale de notre simulation. Elle contient l'ensemble des tâches stockées dans un vecteur et la file des évènements triée sur base de l'instant de déclenchement de ceux-ci. L'algorithme qui permet de simuler l'ordonnancement du système jusqu'à l'hyperpériode est défini dans cette classe (*simulation(Solution* solution)*) tout comme les différentes méthodes heuristiques : la descente en cascade (*desc_cascade()*), le recuit simulé (*recuit()*) et la recherche tabou (*tabousearch()*) qui ont été présentées au chapitre 5.

Nous avons également implémenté dans cette classe la méthode optimale avec des fonctions linéaires dont nous avons détaillé l'algorithme au chapitre 4 (page 59).

Le calcul de la récompense obtenue par ces différentes méthodes est effectué par la fonction *calcRew()*.

B.1.1 Systeme.h

Listing B.1 – Systeme.h

```
1 #ifndef SYSTEME_H
  #define SYSTEM_H

  #include "Tache.h"
  #include "Event.h"
6 #include "Solution.h"
  #include "Divers.h"

  #include <queue>
  #include <vector>
```

```

11 using namespace std;

typedef priority_queue<Event *, deque<Event *>, EventComparison> QUEUE;

class Tache;
16 class Systeme{
    QUEUE eventQueue;//file qui contient les évènements triés par rapport à
        l' instant de déclenchement
    double temps;//temps écoulé depuis le lancement de la simulation
    int hyperP;//l'hyperpériode (ppcm des périodes)
21 int nbtaches;
    double reward;
    double d;//slack (durée disponible pour les parties optionnelles)

    bool ordo;
26
    void tri (int deb, int fin );
    int partition (int deb, int fin );
public:
    vector<Tache*> task;
31 vector<Tache*> sortedTasks;
    Solution * sol;// solution à simuler

    // constructeur/destructeur
    Systeme();
36 ~Systeme();
    // accesseurs
    inline int getHyperP(){return hyperP;}//hyperpériode
    inline int getNbTaches(){return nbtaches;}
    inline double getRew(){return reward;}
41 // méthodes
    void addEvent(Event* newEvent){eventQueue.push(newEvent);}
    inline void setOrdo(bool o){ordo=o;}
    int calcHyperPer();
    double calcRew();
46 void calcSlack();
    Tache* addTask(Tache* T);
    Travail * getPriorJob();

    // algorithmes
51 double linOPT();//Algorithme étalon optimal dans le cas linéaire
    double recuit();//Recuit simulé
    double desc_cascade();//Descente en cascade
    double tabousearch();//Recherche Tabou

56 int simulation(Solution* solution);
    void clean();

    void displayTasks();//affichage

61 };
    #endif

```

B.1.2 Systeme.cpp

Listing B.2 – Systeme.cpp

```

1  #include "Systeme.h"
   #include "Solution.h"
3  #include "Tache.h"

   #include <stdio.h>
   #include <time.h>
   #include <math.h>
8  #include <limits>
   #include <algorithm>
   #include <vector>
   #include <list>

13 using namespace std;

   Systeme::Systeme()
   {
       hyperP=0;nbtaches=0;temps=0.0;ordo=true;reward=0.0;
18 }

   Systeme::~Systeme()
   {
       sortedTasks.clear();
23   for(unsigned i=0; i< task.size() ; i++)
       {
           delete task[ i ];
       }
       task.clear() ;
28   int taille =(int)eventQueue.size();
       for(int i=0; i < taille ; i++) eventQueue.pop();
   }

   Tache* Systeme::addTask(Tache* T)
33 {
       task.push_back(T);
       T->sys=this;
       nbtaches++;
       return T;
38 }

   int Systeme::calcHyperPer()
   {
       int d=0,i=0,ppcm,max;
43   max=task[0]->getP();
       for ( i=0;i<nbtaches;i++)
       {
           if (task[ i ]->getP() > max) max=task[i]->getP();
       } //max des période est la tâche i
48   for(ppcm=0;d!=nbtaches;)
       {
           ppcm+=max;

```

```

        d=0;
        for( i=0;i<nbtaches;i++)
53      {
            if (ppcm%task[i]->getP()==0)
                d++;
        }
    }
58      if (d==nbtaches)
    {
        hyperP=ppcm;
        return ppcm;
    }
63      else return -1;
    }

void Systeme::calcSlack()
{
68      int sum=0;
        for(int i=0; i<nbtaches;i++)
        {
            task[i]->setb(getHyperP()/task[i]->getP());
            task[i]->setw(task[i]->getk()/(double)(task[i]->getb())); // retour
73      sum+=task[i]->getb()*task[i]->getM();
        }
        sortedTasks=task;
        tri (0,nbtaches-1);
        /* for (int i=0;i<nbtaches;i++)
78      {
            sortedTasks[i]->Affiche();
        } */
        // printf ("_____ \n");
        d=getHyperP()-sum;
83 }

void Systeme::tri(int deb, int fin )
{ //Fonction QuickSort utilisée pour trier les tâches par rapport à leurs poids
    int middle;
88      if (deb < fin)
    {
        middle = partition (deb, fin );
        tri (deb, middle);
        tri (middle+1, fin );
93      }
    }

int Systeme::partition(int deb, int fin )
{
98      double x = sortedTasks[deb]->getw();
        int i = deb-1;
        int j = fin+1;
        Tache* temp;
        do
103      {

```

```

        do
        {
            j--;
        }while (x >sortedTasks[j]->getw());
108     do
        {
            i++;
        }while (x<sortedTasks[i]->getw());
        if (i<j)
113     {
            temp=sortedTasks[i]; // swap des éléments aux positions
            // i et j
            sortedTasks[i]=sortedTasks[j];
            sortedTasks[j]=temp;
        }
118     }while (i<j);
        return j; // retourne l'index du milieu
    }

double Systeme::linOPT()
123 {
    int i, bi, oi, bo;
    double div,slack = d;
    sol = new Solution(this,false);
    for (i=0;i<nbtaches;++i)
128     {
        bi=sortedTasks[i]->getb();
        oi=sortedTasks[i]->getO();
        bo=bi*oi;
        if (bo<slack)//b_i*o_i<slack
133     {
            for(int j=0; j<bi;j++)
            {
                sol->otime[sortedTasks[i]->getid()-1][j]=(double)oi;
                printf ("Temps optionnel[Tache no %d,Job no %d]:
// %f\n",sortedTasks[i]->getid(),j+1,otime[sortedTasks[i]->getid()-1][j]);
138             }
            slack-=bo;
        }
        else
143     {
            div = slack/bi;
            for (int j=0; j<bi;j++)
            {
                sol->otime[sortedTasks[i]->getid()-1][j]=div;
                printf ("Temps optionnel[Tache no %d,Job no %d]:
// %f\n",sortedTasks[i]->getid(),j+1,otime[sortedTasks[i]->getid()-1][j]);
148             }
            break;
        }
    }
    i++;
153     while(i<nbtaches)
    {

```

```

        bi=sortedTasks[i]->getb());
        for(int j=0; j<bi; j++)
        {
158             sol->otime[sortedTasks[i]->getid()-1][j]=0;
//             printf ("Temps optionnel[Tache no %d,Job no %d]:
                %f\n",sortedTasks[i]->getid(),j+1,otime[sortedTasks[i]->getid()-1][j]);
        }
        ++i;
    }
163     simulation(sol);
        sol->printSol();
        reward=sol->rew;
        delete sol;
        return reward;
168 }

double Systeme::recuit()
{
    int sim=1, L=nbtaches, counter=0;
173     double diminPalier=0.97, moydeltaF=0;
        double* paliermax = new double[nbtaches];
        for(int i=0; i<nbtaches; i++)
            paliermax[i]=min(task[i]->getO(),task[i]->getP()-task[i]->getM());

    int cptnonordo=0;
178     double q,p,temper=nbtaches,alpha=0.95,deltaF=0,savrew;

    list <Modif*> modiflist;
    //solution initiale tous les travaux recoivent 0 comme temps optionnels
    Solution tempsol(this,true);
183     Solution bestsol(this, true);
        int cpt=0;
        while(cpt<100)//test pour trouver la moyenne deltaF
        {
            savrew = tempsol.rew;
188             tempsol.getSolVois(sim,&modiflist,paliermax);//mets la solution
                voisine dans tempsol
            sim=simulation(&tempsol);//simulation avec la solution temporaire

            if (sim<1)
            {
193                 cptnonordo++;
                    // si ce n'est pas ordonnancable pendant 3 tours
                    if (cptnonordo%3==0)for(int i=0; i<nbtaches; i++)
                        paliermax[i]*=diminPalier;
            }
            else cptnonordo=0;
198             // printf ("paliermax = %f\n",paliermax);
                // printf ("Recompense: %f\n",reward);
            if (tempsol.rew > savrew)
            {
                //on continue à utiliser la nouvelle solution
203                 if (tempsol > bestsol) bestsol = tempsol;
            }
        }

```

```

else
{
    208     q=hasard(0.0,1.0);
           deltaF=savrew – tempsol.rew;
           moydeltaF+=deltaF;
           counter++;
           p=exp(-deltaF/temper);
           213     if (q>p)tempsol.back(&modiflist);//on revient
           //sinon on continue à utiliser la solution (tempsol)
}
cpt++;
if (cpt%L==0)temper*=alpha;
clean();
218 }
cpt=0;
moydeltaF/=counter;
temper=moydeltaF/log(2.0);
for (int i=0;i<nbtaches;i++)paliermax[i]=task[i]->getP()-task[i]->getM());
223 tempsol.init ();
while(cpt<900)
{
    savrew = tempsol.rew;
    tempsol.getSolVois(sim,&modiflist,paliermax);//mets la solution
    voisine dans tempsol
    228     sim=simulation(&tempsol);//simulation avec la solution temporaire
    if (sim<1)
    {
        cptnonordo++;
        // si ce n'est pas ordonnancable pdt 3 tours
        233     if (cptnonordo%3==0)for(int
           i=0;i<nbtaches;i++)paliermax[i]*=diminPalier;
    }
    else cptnonordo=0;
    // printf ("paliermax = %f\n",paliermax);
    // printf ("Recompense: %f\n",reward);
    238     if (tempsol.rew > savrew)
    {
        //on continue à utiliser la nouvelle solution
        if (tempsol > bestsol)
            bestsol = tempsol;
    243     }
    else
    {
        q=hasard(0.0,1.0);
        deltaF=savrew – tempsol.rew;
        248     p=exp(-deltaF/temper);
        if (q>p)//on revient à la solution précédente
            tempsol.back(&modiflist);
        //sinon on continue à utiliser la solution (tempsol)
    }
    253     cpt++;
    if (cpt%L==0)temper*=alpha;
    clean();
}

```

```

    reward = bestsol.rew;
258     delete[] paliermax;
        bestsol.printSol ();
        return reward;
    }

263 double Systeme::desc_cascade()
    {
        int sim=1;
        double diminPalier=0.97, savrew;
        double* paliermax = new double[nbtaches];
268     for(int i=0;i<nbtaches;i++)
            paliermax[i]=min(task[i]->getO(),task[i]->getP()-task[i]->getM());
        int cptnonordo=0;

        list <Modif*> modiflist;
        //solution initiale tous les travaux recoivent 0 comme temps optionnels
273     Solution tempsol(this,true);
        Solution bestsol(this, true);
        int cpt=0;
        for (int i=0;i<nbtaches;i++)paliermax[i]=task[i]->getP()-task[i]->getM();
        tempsol.init ();
278     while(cpt<1000)
        {
            savrew=tempsol.rew;
            tempsol.getSolVois(sim,&modiflist,paliermax); //mets la solution
                voisine dans tempsol
            sim=simulation(&tempsol); //simulation avec la solution temporaire
283     if (sim<1)
            {
                cptnonordo++;
                // si c'est pas ordonnancable pdt 3 tours
                if (cptnonordo%3==0) for(int
                    i=0;i<nbtaches;i++)paliermax[i]*=diminPalier;
288     }
            else cptnonordo=0;
            // printf ("paliermax = %f\n",paliermax);
            // printf ("Recompense: %f\n",reward);
            if (tempsol.rew > savrew)
293     {
                //on continue à utiliser la nouvelle solution
                if (tempsol > bestsol) bestsol = tempsol;
            }
            else
298     {
                tempsol.back(&modiflist); //on revient
                //sinon on continue à utiliser la solution (tempsol)
            }
            cpt++;
303     clean();
        }
        reward = bestsol.rew;
        delete[] paliermax;
        bestsol.printSol ();
    }

```

```

308     return reward;
    }

    double Systeme::tabousearch()
    {
313         int sim,cpt=0,cptnonordo=0;

        vector<Solution*> V; int tailleV =41*nbtaches;//sous voisinage
        list <Tabou> taboulist; int tabousize=10;
        const int cptmax=1000/tailleV;
318         double diminPalier=0.97;
        double* paliermax=new double[nbtaches];
        for (int i=0; i<nbtaches; i++) paliermax[i] =
            min(task[i]->getO(),task[i]->getP())-task[i]->getM());

        vector<Solution*>::iterator it ;
323         Solution x(this, true);
        //solution initiale tous les travaux recoivent 0 comme temps optionnels
        Solution bestsol(this, true);

        Solution* voisine, *tmpbestsol;
328         sim = simulation(&x);
        clean();// on nettoie la simulation: remise du temps à 0, récompense,...
        while(cpt<cptmax)
        {
            for(int i=0;i< tailleV ; i++)//V=sous- voisinage de x
333             {
                voisine = x.getSolVoisine(sim,paliermax);
                sim=simulation(voisine);
                if (sim<1)
                {
338                     cptnonordo++;
                    // si c'est pas ordonnancable pdt 3 tours
                    if (cptnonordo%3==0)
                        for(int i=0;i<nbtaches;i++)
                            paliermax[i]*=diminPalier;
                    // printf ("paliermax : %f\n",paliermax);
343                 }
                else cptnonordo=0;
                V.push_back(voisine);
                clean();
            }
            sort(V.begin(), V.end(), trisolutions ());
            // tri décroissant par rapport à la récompense
            it =V.begin();
            do
            {
353                 tmpbestsol=*it;
                it ++;
            }
            while(tmpbestsol->tabou.isTabou(&taboulist) && it!=V.end());
            if ( it ==V.end())
358                 x=*(V.begin()); //même s'il est tabou
            else x = *tmpbestsol;

```

```

        if (x > bestsol)bestsol=x;
        taboulist .push_front(x.tabou); //mise à jour de la liste tabou
        if ((int) taboulist .size ()>tabousize)taboulist .pop_back();
363     for( it=V.begin(); it !=V.end(); it++)
            delete *it ;
        V.clear ();
        cpt++;
    }
368     delete[] paliermax;
        taboulist .clear ();
        reward=bestsol.rew;
        bestsol.printSol ();
        return reward;
373 }

void Systeme::clean()
{
    Event* ev;
378     int siz = (int)eventQueue.size();
        for(int i=0; i < siz ; i++)
        {
            ev= eventQueue.top();
            eventQueue.pop();
383     }

        for(int t=0;t<nbtaches;t++)
        {
            task[t]->clean();
388     }
        reward=temps=0.0;
}

int Systeme::simulation(Solution *solution)
393 {
    double sumopt=0, sumtaski=0,remTimeBeforeNextEvent;
    int Mi,bi;
    sol = solution ;
    for(int i=0;i<nbtaches;i++)
398     { //on regarde d'abord si ça vaut la peine de simuler
        Mi=task[i]->getM();
        bi=task[i]->getb();
        for(int j=0;j<bi;j++)
        {
403             sumopt += sol->otime[i][j];
        }
        sumtaski += bi*Mi + sumopt;
        sumopt=0;
    }
408     if (sumtaski > getHyperP() + 0.00001)
    {
        // printf ("\n***** PAS ORDONNANCABLE *****\n");
        // printf (" Il y a plus d'unites (%f) que l'hyperperiode
            (%d)\n",sumtaski,getHyperP());
        sol->rew=0;
    }
}

```

```

413         reward=0;
           ordo=false;
           return -1;
       }

418     Travail* priorjob ;
     Event* nextEvent;
     //displayTasks();
     for (int t=0;t<nbtaches;t++)//au début toutes les tâches envoient un job
       (départ simultané)
     {
423         addEvent(task[t]->arrivee);
     }
     // printf ("*** Hyper-période= %d ***\n",getHyperP());
     do//on exécute la simulation sur l' intervalle d'étude P
     {
428         // printf ("\nInstant %f: \n",eventQueue.top()->temps);
         do
         {
             nextEvent = eventQueue.top();
             temps = nextEvent->temps;
433             ordo = nextEvent->processEvent();
             eventQueue.pop();
         }while(ordo && fabs(temps - eventQueue.top()->temps) <
             0.000001);//tant que l'instant est identique

         if (!ordo)break;
438     priorjob = getPriorJob(); // politique EDF
         if (priorjob !=NULL)//s'il y a au moins un job en attente
         {
             remTimeBeforeNextEvent = eventQueue.top()->temps -
                 temps;
             if (( priorjob->getRem() < remTimeBeforeNextEvent) ||
                 fabs(priorjob->getRem() - remTimeBeforeNextEvent) <
443                 0.000001)
             {
                 priorjob->finJob->temps = temps +
                     priorjob->getRem();
                 addEvent(priorjob->finJob);
             }
             else
448             {
                 priorjob->setRem(priorjob->getRem()-remTimeBeforeNextEvent);
             }
         }
     }while(temps<getHyperP());
453     if (ordo)
     {
         // printf ("\n***** Systeme ordonnancable *****\n");
         calcRew();
         return 1;
458     }
     sol->rew=0;
     reward=0;

```

```

        // printf ("\n***** PAS ORDONNANCABLE *****\n");
        return 0;
463 }

Travail * Systeme::getPriorJob()
{
    double resttime, min=(double)INT_MAX;
468 Travail *job, *priorjob=NULL;
    for (int t=0;t<nbtaches;t++)
    { //EDF on compare chaque tâche et on garde le job actif dont l'échéance est
      la plus proche
        job = task[t]->getJob();
        if (job!=NULL)
473 {
            resttime=(double)job->getP()-fmod(temps,(double)job->getP());//temps
            restant avant la prochaine échéance
            if (resttime<min)
            {
                min=resttime;
478                priorjob=job;
            }
        }
    }
    return priorjob ; //retourne celui qui a le plus petit temps restant avant
    l'échéance
483 }

void Systeme::displayTasks()
{
    for (int i=0;i<nbtaches;i++)
488 {
        task[i]->Affiche();
    }
}

493 double Systeme::calcRew()
{
    double sum1=0, sum2=0;
    int bi;
    for (int i=0;i<nbtaches;i++)
498 {
        bi=task[i]->getb();
        sum1=0;
        for (int j=0;j<bi;j++)
        {
503            sum1 +=task[i]->fonction(sol->otime[i][j]);
        }
        sum2+=sum1/(double)bi;
    }
    sol->rew=reward=sum2;
508 return reward;
}

```

B.2 Tâche

Le système comporte un ensemble de tâches et chaque tâche est définie par certaines caractéristiques : sa période (P), la longueur de la partie obligatoire (M) et la longueur maximum de la partie optionnelle (O). Une instance de la tâche est séparée en deux parties : obligatoire (*jobOblig*) et optionnelle (*jobOpt*), nous enregistrons dans chaque tâche un pointeur vers ces deux parties. Chaque tâche possède une fonction de récompense, les fonctions des différentes tâches se différencient par un certain coefficient k . Un pointeur vers l'évènement *Arrivée* (d'un nouveau travail) est stocké et l'instant de déclenchement de l'arrivée est augmenté de P à chaque instance de manière à gérer l'arrivée suivante.

B.2.1 Tache.h

Listing B.3 – Tache.h

```

1  #ifndef TACHE_H
   #define TACHE_H

   #include "Travail.h"
   #include <math.h>
6  #include <list>
   using namespace std;

   class Travail ;
   class Systeme;
11  class Event;

   class Tache{
       int id;
       int P; //période
16      int M; //longueur de la partie obligatoire
       int O; //longueur maximum de la partie facultative
       int b; //nb de fois que l'on peut placer la tâche dans l' intervalle d'étude
           (hyperpériode)
       Travail * jobOblig;
       Travail * jobOpt;
21      int numopt; //numéro du job actuel
       double k; //coeff de la fonction de récompense
       double w; //k/b: importance de la tâche (proportionnel au coefficient)
   public:
       Tache(int ident, int per, int man, int opt, double coeff);
26      ~Tache();
       Systeme* sys;
       Event* arrivee;
       //accesseurs
31      inline int getid() {return id ;};
       inline int getP(){return P;};
       inline int getM(){return M;};
       inline int getO(){return O;};

```

```

    inline int getnumopt(){return numopt;}
    inline int getb(){return b;};
36  inline double getk(){return k;}
    inline double getw(){return w;}
    inline bool isJobRunning(){return jobOblig->isRunning() ||
        jobOpt->isRunning();}
    //méthodes
    void sendJob();
41  Travail * getJob();
    void clean();
    inline void setb(int x){b=x;};
    inline void setw(double x){w=x;};
    void setArriveeSuivante();
46  void Affiche ();
    inline double fonction(double t){return k* pow((double)2,(double)t/300);}
    void efface( Travail * job );
};
#endif

```

B.2.2 Tache.cpp

Listing B.4 – Tache.cpp

```

1  #include "Tache.h"
    #include "Travail .h"
    #include "Systeme.h"

5  Tache::Tache(int ident, int per, int man, int opt, double coeff)
    {
        id=ident;P=per;M=man;O=opt;k=coeff;
        jobOblig = new Travail(this, true);
        jobOpt = new Travail(this, false);
10  arrivee = new Arrivee(this,0);
        numopt=0;
    }
    Tache::~Tache()
    {
15  delete jobOblig;
        delete jobOpt;
        delete arrivee;
    }
    void Tache::Affiche ()
20  {
        printf ("\nTache_ %d:\nPeriode:_%d\nPartie_obligatoire:_%d\nPartie_
            facultative:_%d\nCoeff_de_recompense:_%%.2f\nRetour_marginal:_%
            %f\n",id,P,M,O,k,w);
    }

    Travail * Tache::getJob()
25  {
        if (jobOblig->isRunning()) return jobOblig;
        if (jobOpt->isRunning())return jobOpt;
        return NULL;
    }

```

```

30 void Tache::sendJob()
    {
        if (M>0.0)
        {
35             jobOblig->setJob();
        }
        if (sys->sol->otime[this->getid()-1][numopt] > 0.000001)
        { //s' il y a un temps optionnel > 0
            jobOpt->setJob();
40         }
        numopt++;
    }

void Tache::clean()
45 {
    jobOblig->stop();
    jobOpt->stop();
    arrivee->temps=0;
    numopt=0;
50 }
void Tache::setArriveeSuivante()
{
    sys->addEvent(this->arrivee);
}

```

B.3 Travail

Dans notre simulation, une tâche lance à chaque période deux travaux, nous identifions par *oblig* le fait que le travail correspond à la partie obligatoire ou optionnelle de la tâche. Ce qui permet de choisir M comme temps d'exécution si la partie est obligatoire ou le temps optionnel (*otime*) déterminé par les méthodes heuristiques si la partie est optionnelle.

B.3.1 Travail.h

Listing B.5 – Travail.h

```

1 #ifndef TRAVAIL_H
  #define TRAVAIL_H

  class Tache;
  class Event;
6
  #include<list>
  using namespace std;

  class Travail
11 {
    private:
        double remtime;//temps d'exécution restant

```

```

    Tache* tache;//tâche à laquelle il appartient
    int P;
16    bool oblig; //partie obligatoire ou optionnelle
    bool running;

    public:
        Event* finJob;
21    //constructeur, destructeur
        Travail (Tache* tache, bool mandatory=true);
        ~Travail ();
        //accesseurs
        inline double getRem(){return remtime;}
26    inline int getP(){return P;}
        inline bool isMand(){return oblig;};
        inline bool isRunning(){return running;};
        //méthodes
        inline void stop(){running=false;}
31    inline void run(){running=true;}
        void setJob();
        inline void setRem(double rem){remtime=rem;};
};

36 #endif

```

B.3.2 Travail.cpp

Listing B.6 – Travail.cpp

```

1  #include "Travail.h"
    #include "Tache.h"
    #include "Systeme.h"
4

    Travail :: Travail (Tache* task, bool mandatory)
    {
9      this->tache=task;
        running=false;
        P=task->getP();
        oblig=mandatory;
        finJob = new FinJob(this,0);
14 }

    void Travail :: setJob()
    {
19     if (oblig)
        {
            remtime=(double)tache->getM();
            // printf ("+ La tache %d envoie un travail obligatoire de %f
                unites\n", tache->getid(),remtime);
        }
        else
24     {
            remtime=tache->sys->sol->otime[tache->getid()-1][tache->getnumopt()];

```

```

        // printf ("+ La tache %d envoie un travail optionnel avec
        t=%fn",tache->getid(),remtime);
    }
    running=true;
29 }
    Travail::~Travail ()
    {
        delete finJob;
    }

```

B.4 Générateur

La classe « Generateur » est utilisée pour créer un ensemble de systèmes. Les systèmes seront encodés dans un fichier de manière à récupérer aisément ceux-ci lors de l'exécution de la simulation.

B.4.1 Generateur.h

Listing B.7 – Generateur.h

```

1 #ifndef GENERATEUR_H
2 #define GENERATEUR_H

    #include "Systeme.h"

    class Generateur
7 {
    public:
        void genStaticT(int nbsys, int n, char fichier []);
        //génère nbsys systèmes de n tâches dans fichier
        void genStaticU(int nbsys, double U, char fichier []);
12 //génère nbsys systèmes avec une utilisation U dans fichier
        Systeme* getNextSystem(long &pos, char fichier[]);
        //retourne le système à la position pos dans le fichier
    };
    #endif

```

B.4.2 Generateur.cpp

Listing B.8 – Generateur.cpp

```

1 #include "Generateur.h"
    #include "Divers.h"
    #include "Tache.h"
4
    void Generateur::genStaticT(int nbsys, int n, char fichier [])
    { //génère nbsys systèmes de n tâches indépendamment de l'utilisation
        FILE * simul;
        int i=0, P, M, O;
9        double currentload = 0,coeff;
        simul = fopen ( fichier , "w");

```

```

    for(int k=0;k<nbsys;k++)
    {
        i=0; currentload=0;
14      while(i!=n)
        {
            P = genPeriode();//Période
            M = hasardint(0,P);
            O = P-M;//partie facultative max
19      coeff = hasard(1,100);// coefficient de la fonction

            if (currentload + (double)M/P <= 1)
            {
                currentload += (double)M/P;
24      fprintf (simul,"%d_%d_%d_%lf\n",P,M,O,coeff);
                i++;
            }
        }
        printf ("U=%lf\n",currentload);
29      fprintf (simul,"*\n");
    }
    fclose(simul);
}

void Generateur::genStaticU(int nbsys, double U, char fichier[])
34 { //génère nbsys systèmes dans le fichier avec une utilisation proche de U
    //indépendamment du nombre de tâches
    FILE * simul;
    int i=0, P, M, O;
    double currentload = 0,coeff;
    simul = fopen ( fichier , "w");
39      for(int k=0;k<nbsys;k++)
    {
        i=0; currentload=0;
        while(currentload<=U-0.0005)
        {
44      P = genPeriode();//Période
            M = hasardint(0,P);
            O = P-M;//partie facultative max
            coeff = hasard(1,100);// coefficient de la fonction

49      if (currentload + (double)M/P <= U)
            {
                currentload += (double)M/P;
                fprintf (simul,"%d_%d_%d_%lf\n",P,M,O,coeff);
            }
54      }
        printf ("U=%lf\n",currentload);
        fprintf (simul,"*\n");
    }
    fclose(simul);
59 }

Systeme* Generateur::getNextSystem(long &pos,char fichier[])
{
    FILE* simul=fopen(fichier, "r");

```

```

64     fseek(simul, pos, SEEK_SET);
        int P, M, O, cpt=0;
        Systeme *S = new Systeme();
        double coeff;
        while(true)
69     { //on crée le système suivant du fichier .
            cpt++;
            fscanf (simul, "%d_%d_%d_%d\n", &P, &M, &O, &coeff);
            S->addTask(new Tache(cpt,P,M,O,1));
            // printf ("%d %d %d %d\n", P, M, O, coeff);
74         if ((char)fgetc(simul)=='*') break;
            fseek(simul, -1, SEEK_CUR);
        }
        S->calcHyperPer();
        S->calcSlack();
79     pos= ftell (simul);
        fclose (simul);
        return S;
    }

```

B.5 Divers

B.5.1 Divers.h

Cette classe contient différentes fonctions de base utilisées par les autres classes, permettant de générer un nombre aléatoire entre deux bornes, de générer une période pour une tâche,...

Listing B.9 – Divers.h

```

1  #ifndef DIVERS_H
    #define DIVERS_H
3
    const int Matrix [5][3]={{2,2,4},{3,3,9},{5,5,25},{7,7,7},{11,11,11}};
    double hasard(double a, double b);
    int hasardint(int min, int max);
    double min(double x, double y);
8  // inline int round(double x){return(x)>=0?(int)((x)+0.5):(int)((x)-0.5);}

    int genPeriode();
    #endif

```

B.5.2 Divers.cpp

Listing B.10 – Divers.cpp

```

1  #include "Divers.h"

    #include<stdlib.h>
4
    double hasard(double a, double b)

```

```

    {
        return (a + (b-a)*((double) rand() / (RAND_MAX)));
    }
9  int hasardint(int min, int max)
    {
        return rand() % (max-min+1) + min;
    }
double min(double x,double y)
14 {
    return x<y?x:y;
}
int genPeriode()
{
19     int period = 1,p;
    for(int i=1;i<5;i++)
        {
            p = hasardint(0,2);
            period*=Matrix[i ][p];
24     }
    return period;
}

```

B.6 Event

La classe Event permet de gérer les évènements lorsque ceux-ci se déclenchent. (voir description section 5.3, page 68)

B.6.1 Event.h

Listing B.11 – Event.h

```

1  #ifndef EVENT_H
    #define EVENT_H

4  #include<math.h>
    #include<stdio.h>

    class Tache;
    class Travail ;

9  class Event {
    public:
        Event(double t, int typ) : temps(t),type(typ){ }
        double temps;
14     int type;
        virtual bool processEvent() = 0;
    };

    struct EventComparison { //comparateur utilisé pour l'insertion triée
19     bool operator () (Event* left , Event* right) const
        {

```

```

        if (fabs( left ->temps - right->temps)<0.000001) return left->type
            > right->type;
        return left ->temps > right->temps;
    }
24 };

class FinJob:public Event{
public:
    FinJob(Travail* trav , double temps) : Event(temps,1), job(trav) {}
29     Travail* job;
    virtual bool processEvent();
};

class Arrivee:public Event{
34 public:
    Arrivee(Tache* tache, double temps): Event(temps,3), task(tache){}
    Tache* task;
    virtual bool processEvent();
};
39 #endif

```

B.6.2 Event.cpp

Listing B.12 – Event.cpp

```

1 #include "Event.h"
  #include "Tache.h"
  #include "Travail.h"

bool Arrivee::processEvent()
6 {
    if (task->isJobRunning()) return false; //test du respect de l'échéance

    task->sendJob(); //on lance un nouveau travail
    temps+=task->getP(); //prochaine arrivee fixée à l'instant courant + période
11 task->setArriveeSuivante(); //repositionnement dans la queue d'évènements
    return true;
}

bool FinJob::processEvent()
16 {
    job->stop(); //le travail se termine
    return true;
}

```

B.7 Solution

Une solution, qui est en fait une tentative de solution et non la solution définitive, comprend un tableau $otime[i][j]$ qui contient la longueur des parties optionnelles pour tous les travaux j de la tâche i . On passe d'une solution à l'autre en choisissant une solution voisine grâce à la fonction `getSolVoisine()` ou `getSolVois()`. La fonction `getSolVoisine()` se différencie par le fait qu'elle enregistre

les permutations dans la liste *tabou*. Une description de la façon dont est sélectionnée une solution voisine est présentée à la section 5.8, page 79. D'autre part, pour l'algorithme du recuit simulé, les modifications effectuées sont enregistrées grâce à la classe *Modif*, de manière à revenir facilement à la solution précédente sans devoir à chaque fois recopier tout le tableau *otime*. Cette solution est ensuite transmise en paramètre à la fonction *simulation()* qui va tester si le système est ordonnançable et si tel est le cas, calculer la récompense accrue avec ces temps optionnels. Cette récompense est enregistrée dans la variable *rew* qui sera comparée avec la récompense des solutions suivantes.

B.7.1 Solution.h

Listing B.13 – Solution.h

```

1  #ifndef SOLUTION_H
   #define SOLUTION_H
   #include<list>
   #include<vector>
   using namespace std;
6  class Systeme;

   class Modif
   {
11     public:
        int i;
        int j;
        double val;
        Modif(int k,int l, double value):i(k),j(l),val(value){}

   };
16  class Tabou
   {
        int a;
        int b;
21     int c;
        int d;
   public:
        Tabou():a(-1),b(-1),c(-1),d(-1){}
26     Tabou(int a1,int b1,int a2,int b2):a(a1),b(b1),c(a2),d(b2){}
        Tabou(const Tabou& tab):a(tab.a),b(tab.b),c(tab.c),d(tab.d){}
        Tabou& operator=(const Tabou& tab);
        inline bool operator==(Tabou& x){return (x.a==this->a &&
            x.b==this->b && x.c==this->c && x.d==this->d)||
            (x.a==this->c &&
            x.b==this->d && x.c==this->a && x.d==this->b);}
        inline void set(int i, int j, int k, int l){a=i;b=j;c=k;d=l;}
31     bool isTabou(list<Tabou>* taboulist);
   };

   class Solution
   {
36     int nbtaches;

```

```

        int* b;
        Systeme* sys;
    public:
        Solution(Systeme* sys, bool init);
41      Solution(const Solution& s);
        ~Solution();

        Solution* getSolVoisine(int sim, double* paliermax);
        void getSolVois(int sim, list <Modif*>* backup, double* paliermax);
46      void back(list <Modif*>* modiflist);
        bool addModif(int i, int j, double val, list <Modif*>* modiflist);
        bool cherchejobO(int &i, int &j, double palier);
        void printSol();
        void init();
51      double ** otime; // Tableau des temps optionnels
        double rew; //récompense obtenue (calculée après la simulation)
        Tabou tabou; //Swap effectué sur la solution

        bool operator>(const Solution& s); //comparaison de la récompense
56      bool operator<(const Solution& s); //idem
        Solution& operator=(const Solution& s); //opérateur d'assignation
};
struct trisolutions
{
61      bool operator()(Solution const* s1, Solution const* s2)
        {
            if (!s1)
                return false;
            if (!s2)
66          return true;
            return s1->rew > s2->rew;
        }
};
#endif

```

B.7.2 Solution.cpp

Listing B.14 – Solution.cpp

```

1  #include "Solution.h"
   #include "Systeme.h"

   #include "Tache.h"
5
   Solution::Solution(Systeme* systeme, bool init)
   {
       rew = 0;
       sys = systeme;
10      nbtaches = sys->getNbTaches();
       b = new int[nbtaches];
       otime = new double*[nbtaches];
       for(int i=0; i<nbtaches; i++)
       {
15          b[i] = sys->task[i]->getb();
       }
   }

```

```

        otime[i] = new double[b[i]];
        if ( init )for(int j=0;j < b[i ]; j++) otime[i ][ j ] = 0;
    }
}
20 Solution::~Solution()
{
    for(int i=0;i<nbtaches;i++)
    {
        delete[] otime[i];
25     }
    delete[] b;
    delete[] otime;
}

30 Solution::Solution(const Solution& s)
{
    rew = s.rew;
    sys = s.sys;
    tabou = s.tabou;
35     nbtaches = s.nbtaches;
    b = new int[nbtaches];
    otime = new double*[nbtaches];
    for(int i=0;i<nbtaches;i++)
    {
40         b[i] = s.b[i];
        otime[i] = new double[b[i]];
        for(int j=0;j < b[i ]; j++) otime[i ][ j ] = s.otime[i ][ j ];
    }
}
45 void Solution::init ()
{
    for(int i=0;i<nbtaches;i++)
    {
        for(int j=0;j < b[i ]; j++) otime[i ][ j ] = 0;
50     }
    rew=0;
}

Solution& Solution::operator=(const Solution& s)
{
55     rew = s.rew;
    sys = s.sys;
    tabou = s.tabou;
    nbtaches = s.nbtaches;
    for(int i=0;i<nbtaches;i++)
60     {
        b[i] = s.b[i];
        for(int j=0;j < b[i ]; j++) otime[i ][ j ] = s.otime[i ][ j ];
    }
    return *this;
65 }

bool Solution::operator>(const Solution& s)
{
    return (this->rew > s.rew);
}

```

```

70 }

bool Solution::operator<(const Solution& s)
{
    return (this->rew < s.rew);
75 }

Solution* Solution::getSolVoisine(int sim, double* paliermax)
{
    int i = hasardint(0,nbtaches-1);
80 double palier=hasard(0.0,paliermax[i]);
    // printf ("palier = %f, paliermax = %f\n", palier ,paliermax[i] );
    int j = hasardint(0,b[i]-1);
    Solution* x = new Solution(*this); //copie
    if (sim==1)//si c'est ordonnançable on augmente un tij de palier ou
        =maxoptional
85     {
        double maxOptional = x->sys->task[i]->getO();
        // printf ("Tache %d, OMAX= %f\n",i+1,maxOptional);
        if (x->otime[i][j]+palier <= maxOptional)
        {
90             x->otime[i][j]+=palier;
            // printf ("On augmente otime[%d][%d] de %f -> =
                %f\n",i,j,palier,x->otime[i][j] );
        }
        else
        {
95             x->otime[i][j]=maxOptional;
            // printf ("On augmente otime[%d][%d] a
                %f\n",i,j,x->otime[i][j] );
        }
    }
    if (hasard(0,1)<0.5)
100 { //permutation de 2 temps optionnels au hasard sans dépasser oi
        int k = hasardint(0,nbtaches-1);
        int l = hasardint(0,b[k]-1);
        double backup = min(x->otime[i][j],x->sys->task[k]->getO());
        x->otime[i][j] = min(x->otime[k][l],x->sys->task[i]->getO());
105 x->otime[k][l] = backup;
        // printf ("On swap t[%d][%d] et t[%d][%d]\n",i,j,k,l);
        x->tabou.set(i,j,k,l);
    }
    if (sim==1)//pas ordonnançable car plus d'unités que l'hyperpériode
110 {
        if (x->otime[i][j] - palier < 0.0001)
            x->cherchejobO(i,j,palier);
        //on cherche un job qui possède un temps optionnel > palier

115
        if (x->otime[i][j]-palier > 0.0001)
        {
            x->otime[i][j]-=palier;
            // printf ("On diminue otime[%d][%d] de %f -> =
                %f\n",i,j,palier,x->otime[i][j] );
        }
    }
}

```

```

120         }
           else
           {
               x->otime[i][j]=0;
               // printf ("On diminue otime[%d][%d] a 0\n",i,j);
125         }

           }
           return x;
       }
130 void Solution::back( list <Modif*>* modiflist)
       {
           for( list <Modif*>::iterator it = modiflist ->begin() ; it != modiflist ->end(); )
           {
               otime[(*it)->i][(*it)->j]=(*it)->val;
135           delete (*it);
               it = modiflist ->erase(it);
           }
       }
140 bool Solution::addModif(int i, int j, double val, list <Modif*>* modiflist)
       {
           for( list <Modif*>::iterator it = modiflist ->begin()
               ; it != modiflist ->end(); it++)
           {
               if ((*it)->i==i && (*it)->j==j)return false; // il y a déjà eu une
                   modification sur ce job
           }
145     modiflist ->push_back(new Modif(i,j,val));
           return true; // on a rajouté une modification
       }
150 void Solution::getSolVois(int sim, list <Modif*>* backup,double* paliermax)
       {
           int i = hasardint(0,nbtaches-1);
           double palier=hasard(0.0,paliermax[i]);
           int j = hasardint(0,b[i]-1);
           if (sim==1)//si c'est ordonnançable
           {
155           double maxOptional = sys->task[i]->getO();
               // printf ("Tache %d, OMAX= %f\n",i,maxOptional);
               if (otime[i][j]+palier <= maxOptional)
               {
                   addModif(i,j,otime[i][j], backup);
160           otime[i][j]+=palier;
                   // printf ("On augmente otime[%d][%d] de %f -> =
                       %f\n",i,j,palier,otime[i][j]);
               }
           else
           {
165           addModif(i,j,otime[i][j], backup);
               otime[i][j]=maxOptional;
                   // printf ("On augmente otime[%d][%d] a
                       %f\n",i,j,otime[i][j]);
           }
       }

```

```

170 //permutation de 2 temps optionnels au hasard sans dépasser oi
    if (hasard(0,1)<0.5)
    {
        int k = hasardint(0,nbtaches-1);
        int l = hasardint(0,b[k]-1);
175 double sav = min(otime[i][j], sys->task[k]->getO());
        addModif(i,j,otime[i][j], backup);
        otime[i][j] = min(otime[k][l], sys->task[i]->getO());
        addModif(k,l,otime[k][l], backup);
        otime[k][l] = sav;
180 // printf ("On swap t[%d][%d] et t[%d][%d]\n",i,j,k,l);
    }
    if (sim==-1)//pas ordonnançable car plus d'unités que l'hyperpériode
    {
        if (otime[i][j] - palier < 0.00001)
185 cherchejobO(i,j, palier);
        //on cherche un job qui possède un temps optionnel > palier
        if (otime[i][j]-palier > 0.00001)
        {
            addModif(i,j,otime[i][j], backup);
190 otime[i][j]=palier;
            // printf ("On diminue otime[%d][%d] de %f -> =
                %f\n",i,j,palier,x->otime[i][j]);
        }
        else
        {
195 addModif(i,j,otime[i][j], backup);
            otime[i][j]=0;
            // printf ("On diminue otime[%d][%d] a 0\n",i,j);
        }
    }
200 }

bool Solution::cherchejobO(int &i, int &j, double palier) //cherche au hasard un job
    > palier que l'on peut diminuer
{ //de manière à ne pas boucler en retombant sur des doublons
205 bool trouve=false;
    list <int*> combi;
    for(int m=0;m<nbtaches;m++)
    {
        for(int n=0;n<b[m];n++)
210 {
            if (otime[m][n]>=palier)
            {
                int* coord = new int[2];
                coord[0]=m;
215 coord[1]=n;
                combi.push_back(coord);
            }
        }
    }
220 int size=(int)combi.size();
    if (size>0)//s'il y a au moins un job>=palier

```

```

    {
        int x = hasardint(0,size-1);//entre 0 et size-1
        list <int*>:: iterator it = combi.begin();
225     for(int h=0;h<x;h++)it++;//se positionner sur le xme élément
        i = (* it) [0];
        j = (* it) [1];
        trouve=true;
    }
230     for( list <int*>:: iterator
        it=combi.begin();it!=combi.end();it++)delete[](* it );
        combi.clear();
        return trouve;
    }

235 void Solution :: printSol ()
    {
        /* printf ("\nRecompense : %lf",rew);
        for( int i=0;i<nbtaches;i++)
240     {
            printf ("\n\nTache %d : ",i+1);
            for( int j=0;j<b[i]; j++) printf ("%lf ,", otime[i][j] );
        }
        printf ("\n");*/
245 //*****

    Tabou& Tabou::operator=(const Tabou& tab)
    {
        a = tab.a;
250     b = tab.b;
        c = tab.c;
        d = tab.d;
        return *this;
    }
255 bool Tabou::isTabou(list<Tabou> *taboulist)
    {
        for( list <Tabou>::iterator it =taboulist->begin();it!= taboulist->end();it++)
        {
            if ((* it) ==*this) return true;
260     }
        return false;
    }
}

```

B.8 Main

Chaque fichier *simulation_i.txt* représente un ensemble de 1000 systèmes ayant soit le même nombre de tâches (comme dans cet exemple), soit le même facteur d'utilisation. Nous appliquons les algorithmes sur tous les systèmes de chaque fichier. Ensuite, pour chaque fichier on compare la récompense entre les différents algorithmes. Le résultat des simulations est enregistré dans le fichier *resultat.txt*.

B.8.1 Main.cpp

Listing B.15 – Main

```

1  #include "Générateur.h"
   #include<stdio.h>
3  #include<time.h>
   #include <stdlib.h>

   using namespace std;

8  int main()
   {
       srand((unsigned)time(NULL)); //initialise le random

       Générateur G;

13      const int TAILLE=1000;//nombre de systèmes à générer par fichier
       clock_t time1= clock();
       FILE* resultat = fopen(" resultat . txt ", "w");
       fprintf ( resultat , "*****\n");
18      fclose ( resultat );
       for(int i=1;i<12;i++)
       {
           char buffer [20];
           sprintf ( buffer , "simulation%d.txt", i );
23          //G.genStaticT(TAILLE,i, buffer) ;// génère un fichier statique de
              1000 systèmes de i tâches
           long pos=0;
           int nbsys,cpt=0;
           double
               rewopt=0,rewrecuit=0,rewtabou=0,rewcascade=0,moyrecuit=0,moytabou=0,
               moycascade=0;
           for(nbsys=1;nbsys<=TAILLE;nbsys++)
28          {
               printf ("***_%d_me_systeme_***\n",nbsys);
               Systeme* S = G.getNextSystem(pos,buffer);
               rewopt=S->linOPT();
               S->clean();
33              rewrecuit=S->recuit();
               S->clean();
               rewtabou=S->tabousearch();
               S->clean();
               rewcascade=S->desc_cascade();
38              delete S;
               //rapport moyen entre les différentes méthodes et
               l'algorithme étalon
               moyrecuit += rewrecuit/rewopt;
               moytabou += rewtabou/rewopt;
               moycascade += rewcascade/rewopt;

43              printf ("****recuit_--_moyenne_par_rapport_optimal:_
                  %lf\n",(moyrecuit)/nbsys);

```

```
        printf ("****tabou_---_moyenne_par_rapport_optimal:_\n",\n                %f\n", (moytabou)/nbsys);\n        printf ("****cascade_---_moyenne_par_rapport_\n",\n                optimal:_%f\n", (moycascade)/nbsys);\n    }\n48     resultat=fopen("resultat . txt ", "a");\n        fprintf ( resultat , "%d_tâches:_recuit=%f,_tabou=%f,_\n",\n                cascade=%f\n", i, (moyrecuit)/TAILLE, (moytabou)/TAILLE,\n                (moycascade)/TAILLE);\n        fclose( resultat );\n    }\n53     clock_t time2=clock();\n        printf ("Temps_de_la_simulation:__%f_secondes\n",\n                (double)(time2-time1)/CLOCKS_PER_SEC);\n        return 0;\n    }
```

