Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux

Nadège Pontisso

▶ **To cite this version:**

Nadège Pontisso. Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux. Autre [cs.OH]. Institut National Polytechnique de Toulouse - INPT, 2009. Français. NNT : 2009INPT065H . tel-00459071v2

HAL Id: tel-00459071

https://theses.hal.science/tel-00459071v2

Submitted on 20 Dec 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *Institut National Polytechnique de Toulouse*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par *Nadège Pontisso*
Le *16 décembre 2009*

Titre : *Association cohérente de données dans les systèmes temps réel à base de composants - Application aux logiciels spatiaux*

JURY
Yamine Aït Ameur (PR) - LISI, Poitiers
Nicole Levy (PR) - PRISM, Versailles
Xavier Olive - Thales Alenia Space, Cannes
Françoise Simonot-Lion (PR) - LORIA, Nancy
Gérard Padiou (PR) - IRIT, Toulouse
Philippe Quéinnec (MdC) - IRIT, Toulouse

Ecole doctorale : *Mathématiques, Informatique et Télécommunications de Toulouse*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse*
Directeur(s) de Thèse : *Gérard Padiou, Philippe Quéinnec*
Rapporteurs : *Yamine Aït Ameur, Françoise Simonot-Lion*

Résumé

Les architectures distribuées des systèmes embarqués sont souvent décrites sous la forme de composants concurrents communiquant entre eux. De tels systèmes sont à la fois orientés flot de données pour leur description, et dirigés par le temps pour leur exécution. Cette thèse s'inscrit dans cette problématique et se concentre sur le contrôle de la compatibilité temporelle d'un ensemble de données interdépendantes utilisées par les composants du système.

L'architecture d'un système modélisé par composants forme un graphe où plusieurs chemins peuvent relier deux composants, avec des caractéristiques temporelles hétérogènes, ce qui induit des temps de parcours disparates. Il est alors important que ces flots d'information soient assemblés de façon cohérente sur le composant destinataire, c'est-à-dire de telle manière que le composant utilise en entrée des données dépendant (directement ou indirectement) du même pas d'exécution du composant à l'origine de ces flots multiples.

Dans un premier temps, ce principe d'association cohérente de données est identifié et formalisé. Une méthodologie est proposée afin de détecter, dans un graphe de composants, les configurations pouvant poser des problèmes d'association de données

Dans un deuxième temps, différentes approches sont détaillées afin de gérer l'association cohérente des données dans des systèmes périodiques sans supposer de propriétés strictes sur l'ordonnancement des composants. Dans les systèmes où les composants partagent la même période et où les communications intra-périodiques sont interdites, l'association des données est gérée par un mécanisme de files permettant de rééquilibrer les temps de parcours des données sur les différents chemins. Dans le cas où les composants sont de périodes diverses, un mécanisme d'estampillage des données est utilisé afin de mémoriser les dépendances entre données. Associé à l'utilisation de files, cet estampillage permet aux composants de sélectionner, à chacune de leurs phases d'activation, des ensembles de données cohérents choisis parmi les données à leur disposition.

La notion d'association cohérente est ensuite relâchée, permettant une utilisation de données approximativement cohérentes. Des files filtrantes, n'enregistrant qu'une donnée sur un certain nombre de données reçues, permettent de réduire la taille des files nécessaires.

Par ailleurs, du fait de la liberté du modèle d'exécution choisi, il existe des situations où il est impossible de garantir la vivacité de l'association cohérente des données. D'autre part, une architecture particulière peut générer des contraintes de cohérence conflictuelles et aboutir à une impossibilité de gestion de la cohérence.

Pour terminer, les résultats de ces travaux sont appliqués sur le logiciel applicatif d'un satellite d'observation terrestre détectant des points chauds.

Abstract

Distributed real time architecture of an embedded system is often described as a set of communicating components. Such a system is both data flow (for its description) and time-triggered (for its execution). This thesis fits in with these problematics and focuses on the control of the time compatibility of a set of interdependent data used by the components of the system.

The architecture of a component-based system forms a graph of communicating components, where more than one path can link two components. These paths may have different timing characteristics, so information which transits on these paths takes various time to reach the final component. However, the flows of information need to be adequately matched, so that the final component uses inputs which all (directly or indirectly) depend on the same production step of the initial component. We call this property consistent data matching.

The data matching property is defined and formalized. A methodology is proposed to detect, in a component graph, the architecture configurations that have to be analyzed.

Several approaches are developed to manage data matching in periodic systems, without considering strict properties on the system scheduling. First, we consider systems composed by components sharing the same period and where intra-periodic communications are forbidden. Data matching is managed using queues that allows to balance the data transit times through the several paths. Then, we study systems where components have independent periods. Queues are also used and data timestamping is added to record data dependencies. Thus, a component is able, for each of its activation steps, to select consistent data sets according to data dependencies among the available input data.

Data matching consistency is relaxed to allow the use of approximately consistent data sets. We use filtering queues which record only one data among a given number they receive. Their use allows to reduce the necessary queue size.

Due to the loose execution model constraints, some situations exist where data matching liveliness is not guaranteed. Moreover, particular system architectures generate conflictual constraints and lead to an impossible data matching management.

The thesis results are applied on the software of an earth observation satellite constellation, Fuego, which detects fires or eruptions.

Remerciements

La rédaction de ce mémoire, ainsi que le travail effectué pendant ces trois dernières années, doit beaucoup à l'ensemble des personnes j'ai pu côtoyer. Je voudrais donc les en remercier à travers cette page.

Je remercie Yamine Aït Ameur et Françoise Simonot, pour avoir accepté d'être les rapporteurs de ma thèse et membres du jury. Merci pour leurs lectures et leurs commentaires. Merci à Xavier Olive qui a également accepté de faire partie du jury et qui, bien que non rapporteur, a effectué une lecture attentive de mon mémoire et fourni de nombreux commentaires. Je suis également reconnaissante à Nicole Levy qui a accepté d'endosser un rôle d'examinateur. Merci à tous les membres pour leurs efforts d'organisation pour assiter à ma soutenance.

Je remercie Gérard Padiou, mon directeur de thèse, ainsi que David Chemouil et Xavier Olive pour avoir été les initiateurs de cette thèse et pour m'avoir donné l'opportunité d'en être l'actrice principale.

Je remercie tout particulièrement Philippe Quéinnec, devenu mon co-directeur en cours de thèse, et dont la présence m'a été très précieuse. Je le remercie pour sa grande disponibilité, sa sympathie et ses relectures attentives. Il est certain que mes travaux ne seraient pas ce qu'ils sont sans nos échanges et je suis consciente de la chance que j'ai eu de l'avoir à mes côtés. Ma ponctuation n'a pas toujours correspondu à ses goûts ; je lui dédis donc ce paragraphe ; le style approximatif me sera probablement pardonné.

Durant cette thèse, j'ai eu l'occasion de réaliser plusieurs séjours au sein de Thales Alenia Space. Je remercie les personnes qui ont su me fournir des indications précieuses. Merci à Guillaume Veran, devenu mon encadrant en cours de route et qui a pris le temps de répondre à mes multiples questions. Merci à Gérald Garcia qui a su se rendre disponible alors que rien ne l'y obligeait et malgré son emploi du temps chargé. Merci également à tous les membres de l'équipe recherche pour la bonne ambiance qui m'a entourée.

J'ai également une pensée pour tous les membres de l'IRIT que j'ai eu l'occasion de côtoyer durant ces trois années. Merci à tous pour votre sympathie et votre bonne humeur. J'ai une pensée particulière pour Nassima Izerrouken et ses collations énergétiques qui ont certainement alimentées mes plus riches idées.

Enfin, je remercie tous les membres de mon entourage qui ont été à mes côtés pendant toute cette période. Merci à ceux qui se sont souciés de ma thèse. Merci également à ceux qui ne s'en sont pas préocupé et merci à ceux qui n'ont pas cherché à comprendre mes travaux. Leur présence a sans nul doute été source d'équilibre. Merci à ma sœur pour avoir été partante pour tout type d'activité. Merci à mes parents, mes grands-parents et à tous ceux qui, sans même qu'aucun d'eux ou moi ne s'en rende compte, m'ont permis d'arriver jusqu'ici.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte et problématique

Dans le domaine des systèmes embarqués, nous assistons au développement de systèmes de plus en plus complexes alors que le degré de criticité de certains de ces systèmes reste constant. Dans le domaine spatial, la tendance grandissante va vers des satellites de plus en plus autonomes et les progrès dans les ressources disponibles à bord permettent le développement d'applications de plus en plus évoluées. Dans le même temps, le développement de tels systèmes se doit d'être optimisé, moins coûteux et l'application réalisée doit toujours être plus sûre. Pour gérer la complexité des systèmes embarqués et réduire les coûts de développement, des méthodes de conception orientées composants ont émergé. L'objectif de ces méthodes est de décomposer une application en composants distincts interagissant via leurs interfaces.

On distingue particulièrement la conception orientée acteur qui décompose un système en composants gérant leurs propres actions et communiquant entre eux via des canaux de communications. Nous nous concentrons sur l'analyse de systèmes décrits sous cette forme. L'avantage de cette approche de conception est d'une part de décomposer un logiciel en éléments pouvant être développés indépendamment et d'autre part de pouvoir réutiliser des éléments déjà conçus afin de les intégrer dans une nouvelle application. Le concepteur dispose ainsi de composants « sur l'étagère » qu'il peut assembler suivant les objectifs de l'application développée. L'assemblage de ces briques élémentaires est facilité par une standardisation des interfaces des composants. Cependant, si les interfaces sont compatibles syntaxiquement entre elles, cela ne garantit pas que l'application globale aura le comportement désirée. En effet, d'une part la sémantique exacte des composants doit être précisée et d'autre part, les communications entre composants ou les modes d'activation des composants peuvent influer sur le comportement du système.

Cette problématique se pose d'autant plus dans un système temps réel. Dans de tels systèmes, les composants et leur système de communication peuvent être activés en fonction de contraintes temporelles provoquant ainsi des réponses différentes du système selon les paramètres temporels appliqués. Les systèmes temps réel peuvent également requérir des contraintes sur le comportement temporel du système, par exemple imposer un temps de réponse borné. Ces contraintes sont alors à prendre en compte pour pouvoir qualifier un système comme répondant à ses objectifs.

La description d'un système sous forme de composants s'échangeant des données permet également de détecter du parallélisme potentiel entre composants. Si l'exécution du système exploite ce parallélisme alors ce mode d'exécution peut également influer sur le comportement du système. Il est alors nécessaire de déterminer cet im-

pact. Par exemple, nous pouvons chercher à déterminer si l'exécution parallèle produit le même résultat qu'une exécution séquentielle déterminée. Si le système est également temps réel, alors nous déterminons l'influence du parallélisme sur les contraintes temporelles.

Nous nous plaçons dans le cadre de systèmes temps réel, distribués (ou plus généralement autorisant du parallélisme) et embarqués. Nous avons affaire à des systèmes critiques, agissant sous contraintes temporelles et avec une exécution potentiellement parallèle. Nous nous intéressons plus particulièrement aux données utilisées par les composants d'un tel système. Les composants étudiés sont activés indépendamment de toutes contraintes sur leurs données d'entrée. Cependant, nous définissons leur cohérence d'exécution comme dépendant des données qu'ils utilisent à chacune de leurs activations.

Dans un système découpé en composants, un composant peut transmettre une donnée vers différents composants destinataires. Cette donnée initiale est alors traitée par plusieurs composants indépendants (ne communiquant pas entre eux) qui eux-mêmes peuvent transmettre leurs sorties à de nouveaux composants indépendants. Sur le graphe de composants, nous distinguons alors plusieurs chemins partant d'un composant (source), se prolongeant via différents chemins de composants et finalement se regroupant sur un même composant (puits). Le puits utilise alors des données d'entrée dépendant indirectement du même composant source initial. Nous dirons qu'une donnée b dépend d'une donnée a si a est utilisée pour calculer b ou si b utilise une donnée qui, elle-même, dépend de a.

Nous souhaitons alors, qu'à chacune des activations du composant puits, les données qu'il utilise dépendent de la même donnée produite par le composant source. Nous dirons que le puits réalise une association cohérente de données ou encore qu'il a un comportement cohérent. Nous étendons cette définition en considérant que le composant source peut avoir plusieurs sorties et donc émettre des données différentes à destination de différents composants à chacune de ses activations. Nous dirons alors que le puits a un comportement cohérent si les données qu'il utilise en entrées dépendent de données produites lors de la même phase d'activation de la source.

1.2 Thèse soutenue

L'objectif de cette thèse est tout d'abord de définir le concept d'association cohérente des données et d'exécution cohérente. Nous définissons une relation dite d'influence pour formaliser ce concept. Le second objectif de la thèse est alors de garantir un comportement cohérent pour tous les composants puits du système dans le cadre d'un ordonnancement avec peu de contraintes. Le but est d'une part de pouvoir réaliser des analyses le plus en amont possible dans le processus de développement et d'autre part de proposer des solutions imposant le moins de contraintes possibles sur l'ordonnancement final. Cette approche nous permet également d'analyser des systèmes composés de boîtes noires qu'il est impossible de contraindre afin qu'elles aient un comportement désiré, par exemple, s'il est impossible de modifier leurs fréquences. Cela est particulièrement vrai lorsqu'il s'agit de réutiliser des composants matériels.

Afin de gérer cette problématique, tout système est d'abord analysé sous forme de graphe orienté. Cette première phase permet d'identifier les composants puits et leurs sources correspondantes en cherchant un type de sous-graphe particulier que nous appelons *fuseau*.

Afin de pouvoir vérifier s'il s'exécute de manière cohérente, un puits doit être capable de connaître de quels composants et de quelles phases d'activation dépendent ses données d'entrée. Pour cela, un mécanisme d'estampillage des données est utilisé

afin de mémoriser les dépendances des données circulant dans le système. Cet estampillage permet à chaque puits de vérifier si ses données d'entrée dépendent bien de la même phase d'activation de sa source.

Nous ajoutons à cette problématique un mode d'activation particulier des composants. Nous considérons des systèmes composés de composants périodiques. Les composants sont donc activés à chacune de leurs périodes mais l'instant de lecture des données d'entrée par rapport au début d'une période peut varier d'une période à l'autre. Les différents chemins reliant une source à son puits peuvent être de longueurs différentes car non composés du même nombre de composants. De plus, le temps entre la production d'une donnée par la source jusqu'à l'arrivée d'une donnée qui en dépend en entrée du puits est également fonction des périodes des composants, des moments de lecture des entrées de chaque composant et des délais de communication. Ce temps peut donc varier suivant les différents chemins utilisés. Il est alors nécessaire d'utiliser des files d'attente en entrée des puits pour stocker les données produites rapidement en attente de celles produites plus lentement et correspondant au même pas d'activation de la source. Elles seront alors utilisées ensemble lors d'un pas d'activation du puits. Dans le cas où le système est composé de composants de même fréquence et où les communication entre composants ne se font que d'une période à l'autre, la cohérence du système peut se gérer en utilisant uniquement des files d'attente dont la taille peut être calculée. L'estampillage des données n'est alors pas nécessaire. Si nous considérons des systèmes dont les composants ont des fréquences diverses et dont la seule contrainte imposée sur les modes de communication est que les temps de communication sont bornés, le problème devient plus complexe. Il est alors nécessaire d'associer un mécanisme d'estampillage à l'utilisation de files d'attente. La taille de ces files reste calculable.

La notion d'association cohérente de données peut être relâchée. Nous considérons alors qu'un puits a un comportement cohérent si les données qu'il utilise en entrées dépendent de données produites, non pas lors de la même phase d'activation de la source, mais lors de phases d'activation comprises dans un intervalle de temps limité. Cette contrainte assouplie permet, dans certains cas, d'utiliser des files d'attente spécifiques appelées files filtrantes. Ces files n'enregistrent qu'une donnée sur un certain nombre de données reçues. Elles permettent ainsi de réduire les tailles des files nécessaires en entrée des puits en exploitant la tolérance de cohérence au niveau de l'association des données.

Pour autant, la liberté laissée au système pour s'exécuter peut rendre difficile la gestion de la cohérence. En effet, du fait de la présence de composants de périodes diverses et sans synchronisation entre eux, il existe des situations où il est impossible de garantir la vivacité de l'association cohérente des données : dans ces situations, le composant puits peut ne pas être capable de composer des ensembles cohérents à une fréquence satisfaisante. Dans le pire des cas, il peut exister des exécutions où le puits est incapable de construire un seul ensemble cohérent. D'autre part, une architecture particulière d'un système peut générer des contraintes contradictoires aboutissant à une impossibilité de gestion de la cohérence.

Le point fort de notre approche est de définir la cohérence de données à l'aide d'une abstraction dans un cadre général pour ensuite l'instancier dans différents types de systèmes périodiques. Par ailleurs, notre point de vue est de considérer le système sous un aspect flot de données et non par l'approche classique flot de contrôle conduisant souvent à des analyses d'ordonnancement.

1.3 Cadre de l'étude

Cette étude a été réalisée en collaboration avec le CNES (Centre National d'Etudes Spatiales) et Thales Alenia Space. Elle a été effectuée au sein du laboratoire IRIT-CNRS et co-financée par le CNES et Thales Alenia Space. Thales Alenia Space a fourni l'expertise des systèmes spatiaux et a apporté le cas d'application.

1.4 Organisation du mémoire

Ce mémoire est structuré en six chapitres. Le premier chapitre est consacré à l'état de l'art. Après des généralités sur les systèmes distribués, répartis et embarqués et la description des spécificités des systèmes spatiaux, la modélisation par composants de tels systèmes est évoquée. Nous poursuivons en décrivant divers modèles orientés flot de données qui, s'ils sont adaptés pour la description de nos systèmes, ne le sont pas en ce qui concerne leurs modes d'exécution. Le mot « cohérence » revêtant de multiples définitions, nous consacrons une partie de ce chapitre a en parcourir les différentes sémantiques dans différents domaines du génie logiciel. Nous constatons que la cohérence telle que nous la définissons et traitons dans nos travaux n'a pas été développée par ailleurs.

Dans le chapitre 2, nous définissons la problématique au cœur de cette thèse. La notion d'association cohérente de données (stricte et relâchée) est définie, d'abord informellement, puis formellement à l'aide d'une relation baptisée relation d'influence. En s'appuyant sur cette relation, nous définissons clairement tous les concepts essentiels à nos travaux. Ce chapitre présente également l'analyse de graphe permettant d'identifier, dans un graphe de composants, les configurations pouvant causer des problèmes d'association cohérente. Enfin, cette partie se termine par la présentation d'un mécanisme d'estampillage permettant à chaque donnée de mémoriser ses dépendances.

Le chapitre 3 résout le problème d'association cohérente de données dans le cadre de systèmes périodiques dans lesquels tous les composants sont de même période. Il apparaît alors que le problème peut être géré en utilisant des files FIFO en entrée des composants puits et qu'il n'est pas nécessaire d'estampiller les données. Le calcul des bornes sur ces files est détaillé dans le cadre de systèmes simples ou plus complexes.

Le chapitre 4 s'intéresse à des systèmes dont les composants sont de fréquences diverses et les communications entre eux peu conditionnées. Le mécanisme d'estampillage présenté chapitre 2 est alors utilisé, associé à des files en entrée des puits. Le comportement de ces files et le calcul de leurs tailles sont présentés. La notion de cohérence relâchée est ensuite exploitée pour réduire les tailles de files nécessaires. Pour cela, les files filtrantes sont présentées puis leur taille nécessaire est calculée. Une autre méthode permettant de gérer la cohérence relâchée est évoquée : elle consiste en l'utilisation de composants retardateurs sur certains chemins de données.

Le chapitre 5 souligne des cas particuliers pour lesquels la gestion de la cohérence des données est difficile voire impossible. Tout d'abord, sont évoquées les problématiques particulières liées à l'utilisation de systèmes multipériodiques et dont les composants s'exécutent librement. La difficulté ou l'impossibilité du maintien d'une cohérence des données forte tout en préservant la vivacité du système est analysé. L'architecture du système peut également avoir un impact sur la gestion de la cohérence. Nous présentons alors les configurations interdisant une gestion correcte des données.

Enfin, nous terminons avec le chapitre 6 présentant l'application des résultats de cette thèse sur système réel avec la mission FUEGO, un satellite d'observation terrestre.

Chapitre 2

État de l'art - contexte

Ce chapitre aborde les concepts généraux et le contexte qui forment le cadre de cette thèse. Dans un premier temps, les caractéristiques générales des systèmes distribués, temps réel et embarqués (DRE systems : Distributed Real time Embedded systems) sont étudiées. Nous mettons l'accent sur l'aspect temps réel de ces systèmes et évoquons les aspects distribués et embarqués. Une section est consacrée à des systèmes DRE particuliers, les systèmes spatiaux. Ces systèmes sont le domaine d'application de nos travaux. Nous présentons ensuite le concept de modélisation par composants, technique classiquement utilisée pour la conception des DRE. Plus particulièrement, nous nous intéressons à la modélisation orientée acteur que nous adoptons dans cette thèse pour modéliser les systèmes à étudier. Nous utilisons plus précisément un modèle orienté acteur mais également flot de données. Nous abordons alors ce dernier aspect dans une section dédiée. Nous identifions le fait que si ces modèles sont adaptés pour spécifier des systèmes, les modèles d'exécution qui leur sont généralement associés divergent du nôtre. Pour terminer, nous abordons la notion de cohérence, terme clé de nos travaux. Nous constatons que ce terme peut avoir de multiples significations et nous passons en revue celles se rapprochant de notre problématique. Cependant, nous constatons que si des similarités existent, aucune des notions évoquées ne correspond totalement à notre définition de la cohérence.

2.1 Les systèmes distribués, temps réel et embarqués

2.1.1 Systèmes temps réel, systèmes dirigés par le temps

Un système est considéré comme temps réel [48] si son bon fonctionnement n'est pas caractérisé uniquement par la correction des valeurs qu'il produit mais également par le respect de contraintes temporelles sur cette production de valeurs. Un système temps réel interagit avec l'environnement, qui lui même évolue avec le temps. Le temps de réponse désiré d'un tel système est donc généralement fixé d'après l'environnement avec lequel il interagit. Un distributeur de billets doit délivrer des billets en moins de dix secondes, un système de freinage ABS doit réagir en moins d'une seconde.

On distingue les systèmes temps réel dur, l'arrivée après échéance d'un événement attendu ne doit jamais se produire (gestion d'erreurs), et les systèmes temps réel mou, l'arrivée exceptionnelle après échéance d'un événement attendu ne mettra pas le système en danger.

Une tâche temps réel se différencie d'une tâche non temps réel essentiellement par l'existence d'une échéance temporelle. Ainsi une tâche temps réel pourra être caractérisée par son échéance (deadline), son éventuelle période ou un délai minimal entre

deux activations (tâche sporadique), son temps d'exécution minimal ou maximal... La problématique essentielle des systèmes temps réel est de garantir qu'une action sera effectuée dans un temps déterminé. Pour cela, des mécanismes d'ordonnancement des tâches, des communications ou des accès à des ressources partagées sont mis en place. La gestion des systèmes temps réel nécessitent une notion fiable du temps global si des garanties temporelles sont souhaitées sur le système global.

Deux principaux modes d'exécution sont appliqués aux systèmes temps réel [40] : exécution dirigée par les évènements (event-triggered) ou exécution dirigée par le temps (time-triggered). Un système dirigé par les évènements suit un principe de réaction à la demande. Le système réagit suivant les évènements qu'il reçoit, généralement par le biais d'interruptions. L'interaction avec l'environnement est relativement imprévisible et l'ordonnancement des tâches doit être calculé de manière dynamique, même si une pré-analyse statique peut permettre de faciliter le déploiement des composants en vue d'un meilleur ordonnancement dynamique [39]. Ce principe amène des problématiques particulières de dépassement de contraintes temporelles (le comportement d'une tâche influençant les autres) et empêche de fournir des spécifications temporelles précises sur les système. Cependant, ce modèle est bien adapté pour un système gérant des actions sporadiques ou cherchant à optimiser ses ressources, par exemple pour un système cherchant à maximiser son occupation processeur ou à réduire sa consommation d'énergie. Ce modèle est également celui qui permet de faire évoluer l'architecture du système le plus facilement.

Les systèmes dirigés par le temps se basent sur la progression globale du temps pour réaliser le contrôle du système. Les composants sont en grande majorité périodiques et notamment ceux en interaction avec l'environnement, permettant ainsi de contrôler l'interaction du système avec ce dernier. Contrairement aux systèmes dirigés par les évènements, l'ordonnancement des tâches ou des communications d'un système dirigé par le temps peut être établi de façon statique. Cela permet de spécifier précisément le comportement temporel du système mais peut amener à une sous-utilisation des ressources ou à de grandes latences. En effet le temps réservé pour chaque tâche doit être au moins aussi long que son temps maximal d'exécution. Du point de vue évolutivité du système, cette approche est plus rigide. L'ajout d'un composant ou la modification de propriétés temporelles de certains impose un nouveau calcul de l'ordonnancement.

Suivant les caractéristiques du système conçu, un choix important devra être fait entre une approche dirigé par le temps ou les évènements, chaque approche présentant ses avantages et ses inconvénients. Des approches proposent également de combiner les deux principes, par exemple en utilisant des sous-systèmes dirigés par les évènements reliés entre eux par des communications dirigées par le temps [20]. Ainsi, le comportement temporel du système au niveau architecture est prédictible tout en permettant que l'ordonnancement et la gestion des ressources des sous-systèmes soient flexibles.

Nous pouvons également distinguer deux approches dans la conception de systèmes temps réel : l'approche synchrone et l'approche asynchrone. Dans l'approche synchrone, nous pouvons considérer que le temps est gelé durant un pas d'activation. Les pas d'activation des éléments du système sont considérés comme étant instantanés et les communications comme étant de durée nulle. Les sorties d'un système sont considérées comme simultanées aux entrées. Un processus synchrone fige les valeurs de ses signaux d'entrée au début de son exécution et pour toute la durée de celle-ci. D'un point de vue interne, le temps est éliminé. Le système peut donc adopter un temps logique discret calé sur le rythme des entrées. Il appartient au compilateur de gérer concrètement l'approche synchrone. Toute approche ne respectant pas ces propriétés sera considérée comme asynchrone.

Lustre[30], Signal[14] et Esterel [15] sont des langages synchrones qui permettent

la description du comportement d'un système ou d'une tâche sous la forme d'un ensemble d'actions à réaliser à chaque instant en réponse aux sollicitations de l'environnement et en fonction des réactions passées. Lustre et Signal utilisent une approche flot de donnée (voir 2.4). Un programme est un réseau d'opérateurs (décrit sous forme d'équations) produisant et consommant des flots de données à chaque cycle d'activation. Esterel adopte une autre approche et décrit un système sous forme procédurale (états, évènements, transitions) avec des automates hiérarchiques et parallèles. Le système est décrit sous forme de flots de contrôle.

Un système temps réel distribué ne donne généralement pas les moyens d'appliquer une approche synchrone au système global. En effet, les composants ne partagent pas toujours une horloge globale, les échanges de données ne sont pas synchronisées... Une approche mêlant les deux concepts est alors utilisée en considérant que l'exécution du système est globalement asynchrone mais localement synchrone dans des sous-systèmes. La communication entre sous-systèmes est une communication asynchrone. On parle alors d'approche GALS (Globally Asynchrononous Locally Synchronous) [19].

Parmi les architecture systèmes entièrement dirigées par le temps, nous pouvons citer l'architecture TTA (Time-Triggered Architecture) [41]. L'architecture est construite à base de composants de calcul et de communication, tous activés périodiquement suivant la progression du temps global. La communication entre les composants est assurée par un protocole TTP (Time-Triggered Protocol). L'accès au bus est généralement contrôlé par la technique TDMA (Time-Division Multiple Access) dans laquelle le temps est partagé entre les différents nœuds (abonnés) du réseau. C'est uniquement pendant ses intervalles de temps alloués qu'un abonné peut transmettre un message. La mise en œuvre d'un système TTA passe par le calcul de l'ordonnancement des accès au bus en fonction des messages à échanger entre composants.

Le modèle TTA permet d'analyser rigoureusement la correction d'un système distribué. Cette architecture permet une implantation sur une architecture matérielle distribuée d'un modèle temps réel périodique synchrone. Cependant il nécessite une gestion du temps global précise qui n'est pas toujours possible, par exemple, si les communications entre composants sont à longues distances ou sans fil.

L'architecture LTTA (Loosely Time-Triggered Architecture) [12, 13] est une forme moins stricte que TTA. Dans une architecture LTTA :

- l'accès au bus est quasi-périodique, utilisant les différentes horloges locales, et non bloquant
- les opérations de lecture et d'écriture sont indépendantes et activées par chaque horloge locale
- le bus se comporte comme une mémoire partagée, les données supportées par le bus sont rafraîchies périodiquement d'après l'horloge locale du bus.

Le terme "quasi-périodique" indique que les horloges activant les écritures, les mises à jour et les lectures sur le bus ne sont pas synchrones. LTTA est un système multi-horloges dans lequel les horloges sont approximativement liées au temps physique et dont leurs dérives relatives sont bornées.

Le principe de cohérence général décrit dans cette thèse peut s'appliquer à des systèmes ayant différents types de système de contrôle. Cependant, les systèmes dans lesquels nous gérons concrètement la cohérence respectent un modèle d'exécution dirigé par le temps. L'exécution des composants est périodique et leur comportement répond à des contraintes temporelles d'échéance et de temps d'exécution minimal et maximal. Il n'est pas nécessaire que les composants connaissent la date globale du système. Cependant, ils doivent respecter leurs périodes et il est donc nécessaire qu'ils partagent tous la même précision de mesure du temps. Dans les satellites, cela pourra être géré grâce au mécanisme PPS (voir 2.2). Ne voulant pas appliquer de trop fortes

contraintes sur l'ordonnancement du système, notre approche est non synchrone (dans le sens du synchronisme Lustre).

2.1.2 Systèmes distribués

Un système distribué (ou réparti) est composé d'éléments reliés par un système de communication. Les éléments possèdent des fonctions de traitement (processeurs), de stockage (mémoire) et de relation avec le monde extérieur (capteurs, actionneurs, communication). Le système de communication met en œuvre une méthode de communication (par exemple, par envoi de messages). Les différents éléments du système ne fonctionnent pas indépendamment mais collaborent à une ou plusieurs tâches communes. En conséquence, une partie au moins de l'état global du système est distribuée entre plusieurs éléments.

Dans un système réparti sans aucune contrainte, les communications et les traitements sont asynchrones. Il n'y a pas de borne supérieure de temps de transition d'un message et pas de bornes sur le rapport relatif des vitesses d'exécution sur deux sites. Si ces temps sont précisés, alors le système est dit réparti synchrone [1].

Dans un système réparti primitif, des messages indépendants unidirectionnels sont envoyés sur un réseau, fiable ou non fiable. Suivant les cas, on pourra considérer les communications comme FIFO (First In First Out).

L'exécution d'un processus est une suite d'événements (local, émission, réception) appelée histoire (ou trace) du processus. Cette suite est ordonnée par une horloge locale au processus. Un système réparti ne dispose généralement pas d'horloge globale. Cependant, il est utile de pouvoir définir un ordre entre évènements appartenant à plusieurs processus. Pour cela, il est possible par exemple de définir une relation globale de précédence, l'une des plus répandues étant la relation de causalité définie par Lamport [43]. Il existe divers mécanismes de codage de ces relations d'ordre (horloges logiques de Lamport, horloges vectorielles et matricielles...).

Nous considérons dans cette thèse des systèmes potentiellement parallèles. En conséquence, il peut s'agir de systèmes réellement répartis ou non. Nous prenons l'hypothèse de communications par envoi de messages, FIFO et fiables. Ces hypothèses, assez fortes pour un cadre réparti général, sont raisonnables dans le monde temps réel et embarqué. Les principes décrits dans cette thèse peuvent s'appliquer aussi bien à des systèmes composés de composants matériels dotés de leur propres ressources de traitement qu'à des composants logiciels connectés par un bus logiciel et projetés sur un unique processeur.

2.1.3 Systèmes embarqués

La notion de systèmes embarqués regroupe une multitude de différents systèmes. Nous pouvons considérer que tout système informatique n'étant pas un ordinateur de bureau peut être qualifié de système embarqué. Nous pouvons cependant distinguer deux types de systèmes embarqués : les systèmes critiques ou non critiques. Des systèmes critiques peuvent mettre en jeu des vies humaines ou des investissements importants. Ces systèmes doivent alors fournir des propriétés strictes de sûreté de fonctionnement.

Les systèmes embarqués sont généralement conçus pour réaliser un seul objectif. Ils partagent généralement la propriété d'avoir des ressources limitées en terme d'espace mémoire, de ressources de calcul ou d'énergie. Il sont dans le même temps

1. Notons les multiples définitions du mot synchrone dans le domaine logiciel. Les paragraphes suivant en feront indirectement une démonstration, évoquant ce terme à plusieurs reprises avec des sens différents.

généralement temps réel, le système devant réagir dans un temps limité, et doivent fonctionner longtemps sans intervention extérieure.

Nous ne gérons pas l'aspect ressource de tels systèmes. Seul l'aspect sûreté de fonctionnement est abordé en établissant un mode de gestion de la cohérence des données garantissant cette cohérence en toute situation. Nous ne cherchons pas à optimiser la gestion de la cohérence en fonction des ressources ou à les prendre en compte afin de produire nos solutions.

2.2 Les systèmes spatiaux

Les logiciels embarqués sur les systèmes spatiaux font partie des systèmes embarqués critiques. En effet, il existe toujours un investissement financier important associé à ces systèmes, il peuvent également représenter des applications stratégiques (satellites militaires) et mettre en jeu des vies humaines (station orbitale). Parmi les systèmes spatiaux, nous traitons plus précisément les systèmes orbitaux (satellites).

Ces systèmes possèdent la caractéristique d'être soumis à d'importantes contraintes environnementales (radiation, vide) et doivent également être autonomes du point de vue énergétique. En conséquence, les ressources matérielles (mémoire, puissance de calcul) disponibles à bord sont généralement réduites.

Un satellite est généralement décomposé en deux parties : la plateforme et la charge utile. La plateforme est souvent standardisée et assure les fonctions de support comme la fourniture d'énergie, la propulsion, le contrôle thermique, le maintien de l'attitude (orientation du satellite) et les communications. La charge utile est conçue spécifiquement pour la mission qu'elle doit remplir. Il peut s'agir par exemple d'instruments photographiant la surface terrestre, d'instruments de télécommunication ou encore d'un télescope spatial. Elle utilise les ressources fournies par la plateforme.

Le satellite est suivi par un centre de contrôle au sol qui envoie des instructions et recueille les données collectées grâce à un réseau de stations terrestres. Pour remplir sa mission, le satellite doit se maintenir sur une orbite de référence en orientant ses instruments de manière précise. Des interventions sont nécessaires à intervalles réguliers pour corriger les perturbations de l'orbite dues à des phénomènes naturels (champ de gravité, influence du soleil, de la lune...).

Même si certaines tâches de maintenance ou liées à la mission sont commandées depuis le sol, les satellites sont en grande partie automatisés. En effet, l'utilisateur ne peut pas commander en permanence le système depuis le sol pour des raisons d'efficacité mais également en raison de passages des satellites dans l'ombre de la Terre. Il s'agit de périodes pendant lesquelles les communications sont impossibles avec le satellite du fait de son positionnement par rapport aux stations sol. De plus, la complexité croissante des projets scientifiques orbitaux amène les instruments à réaliser eux-mêmes des traitements numériques à bord.

Le logiciel contrôlant le satellite est appelé logiciel de vol ou encore logiciel de bord (On Board Software : OBSW). Le développement de ces logiciels se décompose en quatre étapes principales : phase de spécification, conception de l'architecture, conception détaillée et enfin codage. A ces étapes correspondent des phases de tests et de validation suivant la logique d'un cycle en V. Le découpage en tâche et la définition des interfaces sont réalisés durant la phase de conception d'architecture. Simultanément, les contraintes temps réel de ces tâches (périodes, priorités,...) sont déterminées. Ces contraintes sont donc connues tôt dans le processus de développement. Le temps d'exécution des tâches est également estimé au plus tôt en se basant sur les expériences précédentes. Les échéances des tâches sont très souvent égales à la fin de leur période. Une approche par composants (voir section 2.3 suivante) est

généralement adoptée car plusieurs partenaires sont souvent impliqués. Les approches modèles et les méthodes formelles sont encore en cours d'expérimentation et/ou en voie d'adoption dans le spatial. Lorsque des modèles sont utilisés, il sont généralement destinés uniquement à la phase de conception d'architecture. Les simulations et les tests restent encore la principale méthode de vérification des systèmes.

Les logiciels de vol actuels des satellites sont composés de tâches dont les périodes varient principalement entre 10 ms et 100 secondes. La majorité des tâches possèdent une période de l'ordre de la seconde. Leur temps d'exécution est souvent très inférieur à leur période, on constate des rapports de période sur temps d'exécution supérieur à 100. La taille du code d'un logiciel de vol atteint au maximum 100 000 lignes, codées en C ou Ada. La taille des codes devenant importante, l'industrie spatiale recherche de plus en plus des méthodes de validation efficaces en terme de résultats et de coûts.

Dans le modèle actuel avionique 4000 de Thales Alenia Space (consacré aux satellites de télécommunication), un logiciel est composé de 42 tâches dont 30 périodiques. Généralement, les tâches non périodiques sont assignées à des traitements d'alarme ou d'exception. Lors des phases de simulation, ces tâches sont ramenées à des tâches cycliques dont la période correspond à un calcul de cas pire. Le système d'exploitation préemptif Ostrales gère l'ordonnancement de ces tâches. Le logiciel de vol doit être réutilisable sur différentes plateformes. Pour cela, 40 000 variables de configuration sont définies.

L'architecture logicielle retenue pour les satellites actuels par Thales Alenia Space est une architecture à base de composants connectés via un bus logiciel développé en interne. L'architecture matérielle est composée d'équipements reliés par un bus 1553 ou OBDH (On Board Data Handling). Actuellement les systèmes multiprocesseurs ne sont pas utilisés mais un intérêt est manifesté pour cette technologie pour les applications futures.

La majorité des satellites dispose d'un câble spécifique fournissant un signal d'horloge global, le PPS (Pulse Per Second). Ce câble ne permet pas de recevoir la date précise mais le début de chaque seconde. Le recalage des horloges passe par le bus 1553.

2.3 Modélisation par composants des systèmes DRE

2.3.1 Modélisation par composants

La conception de logiciels devenant de plus en plus complexes, un ensemble de méthodes, de modèles et d'outils se développent afin d'en maîtriser la complexité. D'autre part, des soucis d'efficacité imposent de pouvoir réutiliser dans plusieurs applications certains éléments déjà développés pour d'autres projets. Dans ce contexte, le développement par composants [63] est une technique en constante croissance [51] et une grande variété de travaux traitent le sujet, y compris dans le domaine avionique qui nous concerne [5]. Cette technique de développement a d'ailleurs été identifiée comme "technologie clé" dans l'étude [23] commandée par le Ministère de l'économie, des finances et de l'industrie afin d'identifier les technologies porteuses d'avenir en termes d'attractivité et de compétitivité à l'horizon 2010.

D'après ce rapport, un composant est défini comme

> "une brique logicielle réutilisable par plusieurs applications. Conceptuellement, un composant correspond à l'expression logicielle des caractéristiques et des comportements d'un objet, d'un service, d'un aspect ou éventuellement d'une application informatique complète. Les composants lo-

giciels tendent à devenir des agents autonomes, capables d'apprendre, de s'organiser, de découvrir les services offerts par les autres composants."

La notion de composant définie ici est très générique. Nous distinguerons deux principaux types de composants et donc deux différents types d'approches : la conception orientée objet et la conception orientée acteur. Par la suite, nous nous intéresserons uniquement au concept orienté acteur.

Au sein de Thales Alenia Space, le développement par composants a été identifié comme un concept important. Les avantages identifiés sont les suivants :

- le développement multi-équipes est facilité. Les développements des composants peuvent être réalisés par différentes équipes sans besoin de relations fréquentes entre ces équipes. Chaque équipe publie alors une description normalisée de ses composants pour les autres équipes et les équipes intégratrices. Les interfaces des composants sont décrites tôt dans le processus de développement
- une approche de développement incrémental peut être adoptée. Certaines parties du logiciel peuvent être testées alors que d'autres ne sont pas encore réalisées.
- le développement et la validation des différents composants peuvent être isolés.
- la réutilisation des composants est optimisée. La standardisation de l'interface des composants et leur paramétrage permettent de les réutiliser dans différents projets.
- l'assemblage des composants est facilité. La description des interfaces permet de garantir la compatibilité entre les interfaces des composants.
- la documentation peut être produite automatiquement.
- la conception du projet reste cohérente. L'architecture finale correspond bien à celle déterminée en début de développement.
- l'utilisation de différents langages de programmation (Ada et C par exemple) est facilitée

Dans [23], l'intérêt de pouvoir concevoir de nouveaux logiciels à partir de briques préexistantes est souligné. Mais il est également mis en avant que l'intégration de ces différents composants est loin d'être évidente. En effet, la mise en relation des différentes interfaces est facilitée par leur description normalisée mais le comportement global du système reste à contrôler. Des méthodes et outils doivent être développés pour assister l'intégration de ces composants afin de garantir que le comportement du système développé sera conforme à ce que le concepteur attend. La thèse présentée ici trouve sa raison d'être dans le cadre de cette problématique.

2.3.2 Modélisation orientée objet

Dans la conception orientée objet, les objets interagissent principalement par un transfert de contrôle appelé appel de méthode. Les objets sont décrits d'après les concepts réels qu'ils représentent. Les interfaces qu'ils fournissent sont les noms des procédures (les méthodes) qu'ils rendent accessibles. Ces méthodes manipulent l'état de l'objet invoqué. Dans cette méthodologie, nous disposons en quelque sorte de "noms" et de "verbes" qui peuvent être utilisés de façon intuitive. Le langage UML [56] est le principal langage de modélisation orienté objet.

2.3.3 Modélisation orientée acteur

La conception orientée acteur propose une approche différente de celle orientée objet. Dans ce concept, il est considéré que les logiciels peuvent être développés en assemblant des composants préfabriqués. Une logique proche d'un assemblage de composants électroniques ou mécaniques est alors suivie. Cette méthodologie est particulièrement efficace pour la modélisation et la conception au niveau système, car elle

décompose un système du point de vue de ses actions. La méthodologie orientée acteur préconise la décomposition des systèmes en composants interagissants. Dans ce type de modélisation, les acteurs s'exécutent et communiquent avec d'autres acteurs. Contrairement à la méthodologie objet, un acteur définit des activités locales sans se référer implicitement à d'autres acteurs.

Alors que les objets sont relativement passifs, les acteurs maîtrisent leurs activités. Leur environnement, qui peut être composé d'autres composants, leur fournit des données auxquelles ils réagissent et ils produisent éventuellement de nouvelles données pour leur environnement. La différence entre les deux concepts est principalement que les objets interagissent par transfert de contrôle alors que les acteurs interagissent par échanges de données. Potentiellement, les acteurs peuvent s'exécuter en parallèle, entraînant alors le fait que la conception orientée acteur a tendance à être hautement concurrente alors que la conception orientée objet est essentiellement séquentielle.

La notion de modélisation et de conception orientées acteur tire sa source des travaux de Gul Agha [1, 3, 2], ces travaux étant eux même inspirés de Carl Hewitt et d'autres auteurs (voir [4] pour une liste de références). Actuellement, la modélisation par acteurs reprend les idées développées par Agha et Hewitt mais certains aspects comme le fait que chaque acteur dispose de son propre flot de contrôle, ou que les communications entre acteurs sont asynchrones, ne sont pas nécessairement conservés [50].

Un acteur a une interface bien définie. Cette interface différencie l'état interne du composant de son comportement et définit l'interaction entre cet acteur et son environnement. L'interface d'un acteur inclut les ports qui représentent ses points de communication et les paramètres utilisés pour configurer ses opérations par une entité extérieure. Un acteur peut aussi être vu comme une encapsulation des actions effectuées sur des données d'entrée et produisant des données de sortie après son exécution. Les données d'entrée et de sortie sont communiquées par des ports bien définis.

Le contrôle des acteurs est effectué par le modèle qui les englobe à travers un ensemble de méthodes généralement identiques à tous les composants. Typiquement, ces opérations exécutent des fonctions d'initialisation, d'activation et d'arrêt du composant. Le modèle contient des canaux de communication explicites qui véhiculent des données entre les ports des acteurs sous forme de messages. Le contrôle de ces communications peut, par exemple, être géré par un bus logiciel.

De nombreuses communautés utilisent une approche que l'on peut qualifier comme orientée acteur [44]. Dans le cadre des langages synchrones, les composants réagissent aux signaux d'une horloge globale et non lorsque d'autres composants appellent leurs méthodes. Les langages Lustre [30] et Signal [14] (voir [11] pour une liste de références) ajoutent un concept orienté flot de donnée car les composants consomment des entrées et produisent des sorties. Les modèles orientés flot de données asynchrones peuvent également être considérés comme orientés acteur car la réaction des composants dépend de leurs entrées.

La modélisation des composants que nous utiliserons tout au long de cette thèse repose sur un modèle orienté acteur et flot de données. Nous décrirons nos composants comme des boîtes noires produisant des données de sorties en fonction de leurs entrées.

2.3.4 Description d'un composant

2.3.4.A Corba Component Model

Au sein de Thales Alenia Space, le modèle de composant CORBA (Common Object Request Broker Architecture) est utilisé. Le langage de description des com-

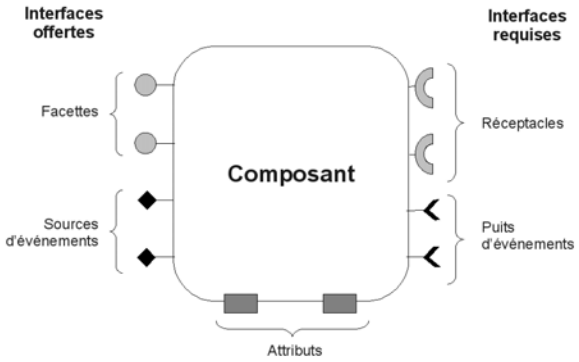


FIGURE 2.1 – Modélisation d'un composant CCM

posants est le langage IDL (Interface Description Langage). IDL est un langage de description d'interfaces totalement indépendant de tout langage de programmation. Le mécanisme de projection permet ensuite de produire automatiquement du code dans un langage de programmation donné.

Thales Alenia Space définit un composant comme un élément logiciel qui :

- fournit zéro, une ou plusieurs interfaces pour d'autres composants ;
- requiert zéro, une ou plusieurs interfaces d'autres composants ;
- a zéro, un ou plusieurs attributs (paramètres de configuration).

Le modèle de composant CORBA (CCM) permet de modéliser des systèmes composés d'éléments hétérogènes. Dans ce modèle [22], un type de composant (figure 2.1) regroupe la définition d'attributs et de ports. Les attributs représentent les propriétés configurables du type de composant. Un port représente une interface soit fournie, soit utilisée, par le type de composant. Quatre types de ports sont définis :

- Une facette est une interface fournie par un type de composant et qui est utilisée par des clients en mode synchrone
- Un réceptacle est une interface utilisée par un type de composant en mode synchrone.
- Une source d'événement est une interface fournie par un type de composant et utilisée par ses clients en mode asynchrone.
- Un puits d'événement est une interface utilisée par un type de composant en mode asynchrone.

Un réceptacle permet à un type de composant d'accepter une référence d'objet (c'est-à-dire une référence de facette ou de composant). Un réceptacle permet donc d'assembler des instances de composants. La composition d'une application permet aux instances de composants de faire de la délégation de traitement.

Le modèle d'événements est du type producteur/consommateur. Pour recevoir des événements, un client doit souscrire un abonnement à ce type d'événements. Ensuite, seul le mode push est utilisé. Les sources d'événements sont de deux sortes, une catégorie n'acceptant qu'un seul consommateur (connexion directe avec le consommateur) et une catégorie en acceptant plusieurs (abonnement du consommateur à un canal de communication). Un puits d'événements permet à un composant de recevoir des événements d'un certain type. Le composant ne contrôle pas ce récepteur, il le rend public pour recevoir des événements. Le puits peut donc recevoir des événements en provenance de plusieurs producteurs.

Dans le modèle CCM, un système sera ainsi modélisé par des instances de composants connectés via leurs différents types de ports.

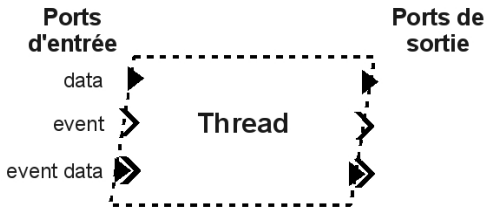


FIGURE 2.2 – Modélisation d'un thread AADL

2.3.4.B Modèle AADL

Un langage de description d'architectures (ADL) est utilisé pour décrire les composants logiciels et/ou matériels d'un système et les interactions entre ces composants. La notion d'architecture est un concept de base pour le développement de systèmes en général. Ces systèmes peuvent être aussi bien des configurations à grande échelle telles que des constellations de satellites, que des configurations locales telles qu'un ordinateur de bord. Les aspects dynamiques (propriétés temporelles, dimensionnement, performances, fiabilité) sont pris en compte par les ADL. Leur principale vocation est d'intégrer ces aspects très tôt dans le processus de développement.

Un ADL se doit de mettre à disposition au moins les trois notions suivantes [52] : composant, connecteur et configuration. On y ajoute également la notion d'interface :

- Un composant représente une entité logicielle de traitement ou de stockage répondant à un besoin fonctionnel ou applicatif. Il peut être atomique ou composé.
- Un connecteur est un composant spécialisé dans l'interaction et ne répondant pas directement à un besoin fonctionnel ou applicatif.
- Une interface est un ensemble de points d'interaction d'un composant ou d'un connecteur avec son environnement. Ces points d'interaction sont généralement orientés.
- Une configuration est un graphe de composants et de connecteurs spécifiant tout ou partie d'une architecture selon un certain point de vue.

Le langage AADL (Architecture Analysis and Design Language) [26, 68] est utilisé pour concevoir et analyser les architectures logicielles et matérielles de systèmes temps réel critiques. Il est employé pour décrire les structures de tels systèmes comme un assemblage de composants logiciels projetés sur une plateforme d'exécution. Il peut être utilisé pour décrire les interfaces fonctionnelles des composants telles que les entrées ou sorties de données, ou les aspects critiques tels que les contraintes de temps. AADL peut aussi être utilisé pour décrire la façon dont les composants interagissent. Nous pouvons par exemple représenter comment les entrées et sorties de données sont connectées ou comment les composants logiciels sont alloués sur les composants de la plateforme d'exécution.

AADL permet de décrire les aspects critiques importants tels que les contraintes de temps, les comportements en cas de fautes ou d'erreurs, le partitionnement du temps ou de l'espace mémoire et les propriétés de sûreté.

Un modèle AADL est composé de composants reliés entre eux par des connexions qui relient leurs interfaces. Chaque composant est défini par un type et une ou plusieurs implémentations. Les composants peuvent contenir des sous-composants. Il est possible d'attribuer des propriétés à chaque élément du modèle (composants, sous-composant, connexion, ports,...). Une description AADL a pour but de modéliser la réalité, pour permettre de produire facilement un système fonctionnel à partir de sa description. Tous les éléments d'une description correspondent à quelque chose de concret.

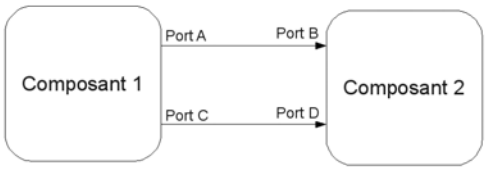


FIGURE 2.3 – Modélisation simplifiée de composants

Parmi les types de composants AADL, nous trouvons des composants logiciels tels que des "threads", représentant des tâches logicielles, ou des composants matériels tels que des "devices", représentant une entité en interface avec l'environnement externe d'une application. Ce type de composant peut interagir avec les composants du logiciel applicatif par l'intermédiaire de leurs ports ou par des sous-programmes. Typiquement, un capteur peut être modélisé par ce type de composant.

Il existe trois catégories de ports :

- *data* : transportent des données, comparables aux états d'un circuit électronique,
- *event* : semblables aux signaux,
- *event data* : signaux transportant des données, comparables aux messages.

L'information envoyée sur un port data écrase la précédente. Par contre, les ports event ou event data peuvent comporter une file d'attente. La figure 2.2, donne la représentation graphique d'un thread AADL avec ses différents types de ports.

Un système AADL est construit à base de composants connectés entre eux via leur ports. Chaque élément du système peut avoir des propriétés spécifiques. Par exemple, le temps d'exécution et la période d'un thread ou la taille de la file d'attente d'un port peuvent être spécifiés.

2.3.4.C Modélisation utilisée

Afin de réaliser l'analyse de systèmes, nous choisissons une modélisation mêlant les concepts de CCM et AADL. Nous conservons les concepts essentiels communs aux deux modèles : les notions de composant, interface et connecteur.

En ce qui concerne les composants, nous choisissons plutôt une approche CCM où les composants sont des unités dont l'implantation finale n'est pas spécifiée. Un composant peut ainsi être aussi bien un composant matériel que logiciel, une tâche au sein d'une application, ou une application elle-même. Nous adoptons cependant la possibilité de définir des propriétés temporelles sur les composants tôt dans le processus de développement telles que proposées par AADL. En effet, l'objectif de nos travaux est de réaliser des analyses temporelles.

Pour les interfaces, un seul type de port est considéré, équivalent aux ports event data d'AADL ou aux puits et sources d'évènements de CCM. Nous conservons la possibilité d'AADL de spécifier la taille des files d'attente en entrée de ses ports.

Plus particulièrement :

- les composants sont représentés par des boîtes noires possédant des ports d'entrée et de sortie,
- les périodes et temps d'exécutions minimaux des composants peuvent être spécifiés,
- les connexions entre ports sont orientées,
- le temps de communication entre ports peut être spécifié,
- un port d'entrée ne peut être connecté qu'à un seul port de sortie,
- un port de sortie peut être connecté à plusieurs ports d'entrées,
- les ports peuvent avoir des files d'attente.

Nous obtenons alors un modèle simple de système pouvant être mis en relation avec des modèles AADL ou CCM. La représentation graphique choisie est illustrée figure 2.3.

2.4 Modèles orientés flot de données

De façon générale, un programme peut être décrit comme un ensemble d'opérations devant être exécutées dans un certain ordre. Deux types d'ordre de ces opérations peuvent alors être distingués [8] : l'ordre orienté flot de contrôle (control flow), fondé sur une séquence temporelle d'opérations et l'ordre orienté flot de données (data flow), fondé sur la disponibilité des données.

Dans un programme orienté flot de contrôle, l'ordre des opérations est prédéterminé par les choix d'ordonnancement du programmeur. Dans un programme dirigé par les données, l'ordre des opérations n'est pas spécifié par le programmeur mais est produit suivant les dépendances et les disponibilités des données du système.

Les modèles orientés flot de données ont fait leur apparition dans les années 70, introduits par Jack Dennis en tant que base de langage de programmation ou d'architecture parallèle et par Gilles Kahn en tant que modèle de concurrence. Ces deux chercheurs ont cependant une approche différente des modèles flot de données. Les flots de données de Kahn sont décrits sous forme de processus continus recevant des flots infinis de données et produisant des séquences de sorties en fonction des séquences d'entrée. L'approche de Dennis décrit les programmes sous forme d'opérations atomiques déclenchées suivant des règles exprimées en fonction des données d'entrées des opérations. Alors que Kahn utilise des lectures bloquantes, Dennis active les composants uniquement lorsque toutes les données sont disponibles. Cependant, des correspondances peuvent être réalisées entre ces deux modèles en considérant le modèles de Dennis comme un cas particulier des réseaux de Kahn [45].

2.4.1 Réseaux de Kahn

Un réseau de Kahn (KPN : Kahn Process Network) [35, 36] est composé d'un ensemble fini de processus qui communiquent entre eux par des canaux FIFO (first in first out) disposant d'une capacité de stockage infinie. Un canal ne peut être connecté qu'à un seul processus en entrée et à un seul processus en sortie. Chaque processus exécute un programme séquentiel pouvant utiliser des instructions de lecture ou d'écriture sur un canal spécifié. La particularité de ces systèmes est que l'écriture est non bloquante alors que la lecture est bloquante. Il n'y a pas d'échange d'information entre nœuds autrement que par l'intermédiaire de canaux de communication.

Ces systèmes sont souvent représentés sous forme de graphes orientés. Les nœuds représentent alors les processus et les arcs, les canaux de communication. Les données circulant dans le réseau sont échantillonnées sous forme de jetons. Un flux de jetons est présenté en entrée du réseau et un flux de jetons est récupéré en sortie. Lorsqu'un jeton est lu, il est consommé.

L'intérêt des réseaux de Kahn en synthèse de systèmes embarqués est qu'ils permettent de décrire des systèmes comportant du parallélisme tout en se prêtant à une analyse des dépendances entre données. Ils conduisent d'autre part à une représentation graphique qui est familière aux spécialistes du traitement du signal et de l'électronique. Ce modèle est en effet très utilisé dans le domaine du traitement du signal car il est naturel pour décrire ces systèmes où des flux infinis de données sont transformés par des processus.

Les réseaux de Kahn possèdent la particularité d'être déterministes si les processus qui le composent le sont également. Cela signifie que l'ordre d'exécution des processus

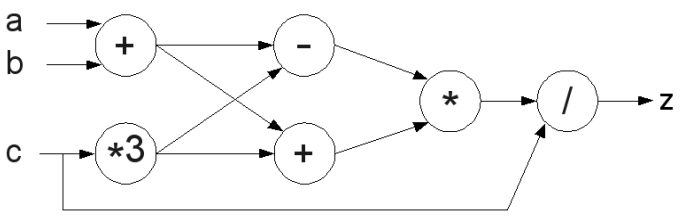


FIGURE 2.4 – Exemple de graphe flot de données

n'affecte pas les valeurs produites en sortie. Seules les valeurs fournies en entrée du réseau influencent les sorties.

Le principal problème à gérer sur les KPN est le calcul des bornes sur les canaux de communication, question conditionnant la faisabilité du système. Le nombre de jetons non consommés présents sur un canal dépend de l'ordre d'exécution des processus. Une source de données spontanée peut produire une accumulation de jetons sur un canal si aucun processus n'est exécuté pour les consommer.

Le bornage des files peut être géré de différentes façons [54]. Tout d'abord, la borne maximale peut être calculée au moment de la conception du système. Cela n'est cependant pas possible pour tout KPN. Une autre voie exploitée est d'agrandir la taille des files à la demande en cours d'exécution. Enfin, les opérations d'écritures peuvent être bloquées lorsque les files sont pleines. Cette approche impose que les tailles des files soient correctement calculées afin de ne pas aboutir à un interblocage artificiel.

2.4.2 Modèle flot de données de Dennis

Le modèle orienté flot de donnée est un modèle proche de celui des réseaux de processus. Les premiers modèles flot de données ont été proposés par Jack B. Dennis [25, 24]. Dans ces modèles, les programmes sont également décrits sous forme de graphes et les arcs, appelés aussi dépendances, représentent des canaux de communications entre les nœuds. Les nœuds, appelés également blocs, représentent des acteurs.

La figure 2.4 est une représentation sous forme de graphe flot de donnée du calcul suivant :

$$x = a + b,\ y = 3c,\ z = \frac{(x-y)(x+y)}{c}$$

Au lieu d'utiliser le mécanisme de lecture bloquante des KPN, les acteurs se déclenchent uniquement lorsqu'ils disposent d'un nombre suffisant de jetons sur leurs entrées. Lorsqu'un acteur est déclenché, il consomme un nombre fini de jetons sur ses entrées et produit un nombre fini de jetons en sortie. Cette réaction est atomique. Les valeurs des jetons en sortie dépendent uniquement des valeurs d'entrée. Il n'y a pas d'interaction par effet de bord entre les nœuds. Nous pouvons considérer que les données s'écoulent de manière asynchrone à travers le réseau, activant des acteurs lorsque tous leurs arguments sont disponibles.

Un processus peut être formé par des déclenchements répétés d'un acteur flot de données auquel on fournit des flots infinis de données. Nous parlerons alors d'un processus flot de données [47]. Un réseau de processus peut ainsi être créé à partir de tels processus.

Un graphe flot de données décrit un ordre partiel entre opérations. L'ordre d'exécution des blocs n'est contraint que par les dépendances entre données. Une propriété

de parallélisme potentiel apparaît : deux opérations sans dépendance de données entre elles pourront s'exécuter en même temps si l'architecture matérielle le permet. Le modèle flot de données fournit un parallélisme implicite, le programmeur n'a pas besoin de spécifier quelles opérations peuvent être exécutées en parallèle. Nous voyons sur l'exemple précédent que le calcul de x peut être réalisé en parallèle avec le calcul de y.

Les résultats des opérations d'un modèle orienté flot de données ne dépendent pas de l'ordre relatif dans lequel des nœuds potentiellement parallèles sont exécutés. On dit alors que l'exécution d'un programme orienté flot de données est un processus commutatif. Cette propriété en implique une deuxième importante : l'exécution d'un modèle flot de données est déterministe.

De plus, l'avantage des graphes flot de données est que les dépendances de données sont clairement exprimées. Les problèmes d'accès concurrents aux données n'existent pas. Dans la pratique, les données produites sont recopiées pour chacun des nœuds les utilisant. Cependant, le partage de données de volume important, tels que des tableaux, impose une gestion particulière abandonnant le modèle flot de données pur que nous n'évoquerons pas ici.

2.4.3 Mécanisme de gestion des jetons, jetons marqués

Du point de vue architecture, le mécanisme de stockage des jetons est une caractéristique importante des systèmes flot de données. Le modèle flot de données théorique nécessite des capacités de stockage infinies sur les arcs du graphe. Pour contourner cette difficulté, deux approches existent :

- approche statique : une capacité limitée de stockage de jetons est définie sur chaque arc. En pratique, un seul emplacement de stockage est utilisé par arc.
- approche dynamique : une allocation dynamique est gérée pour le stockage des jetons dans un espace commun (mémoire centrale de jetons). Chaque jeton a alors un marquage permettant d'indiquer sa position logique sur les arcs (son numéro de génération). On parle alors de jetons marqués (tagged-tokens).

La classe d'architecture flot de données statique exploite le modèle d'exécution de la manière la plus simple. Chaque arc ne peut mémoriser qu'un seul jeton. Chaque opération ne contient qu'un seul emplacement pour recevoir la valeur de son argument et une opération n'émet pas de nouveaux jetons tant que son jeton précédent n'a pas été traité. Avec ce principe, il est alors impossible d'avoir plusieurs instances d'un même processus exécutées en même temps. Ce modèle d'exécution n'exploite que le parallélisme de contrôle.

La classe d'architecture flot de données dynamique [7, 61] exploite le modèle de manière plus élaborée : plus d'un jeton par arc est autorisé lors de l'exécution. Le parallélisme de flux est ainsi exploité en plus du parallélisme de contrôle. En effet, un effet de pipeline s'ajoute au parallélisme du graphe. Il est possible d'envoyer plusieurs générations de données dans le graphe avant même que la première génération ait produit des résultats.

Le stockage des jetons entre les instructions est réalisée grâce à une mémoire centralisée de jetons. A chaque cycle, pour chaque instruction, on cherche dans la mémoire centralisée si tous les arguments sont arrivés. Si c'est le cas, l'instruction est prête. Dès que du temps de calcul est disponible, les jetons contenant les valeurs des arguments sont retirés de la mémoire, l'instruction est exécutée et les jetons résultants de l'opération sont stockés dans la mémoire centralisée de jetons. L'exécution s'arrête lorsque l'ensemble des instructions prêtes devient vide.

Il est alors indispensable d'identifier les jetons afin de pouvoir distinguer les différentes générations des données circulant dans le graphe. Des jetons marqués sont utilisés. Ils contiennent :

– la valeur de la donnée
– le code opération de l'instruction pour laquelle ils sont destinés,
– la position de l'argument dans l'instruction,
– le numéro de génération du jeton.

Le numéro de génération permet d'assembler des jetons provenant de la même génération de données. Ce mécanisme permet d'éviter la gestion de files FIFO sur chaque arc. L'ordre logique d'arrivée des jetons est codé grâce aux numéros de génération.

2.4.4 Computation graph

Le modèle *computation graphs* [10, 28], proposé par Karp et Miller au milieu des années 60 [37, 38], est un des premiers modèles flot de données étudié. Il peut être considéré comme un cas particulier du modèle flot de données de Dennis dans lequel, pour chaque arc entre deux nœuds n_1 et n_2 (n_1 produisant des données utilisées par n_2), il est spécifié :

– le nombre de jeton initialement présent sur l'arc
– le nombre de jeton produits par chaque exécution de n_1
– le nombre de jeton requis sur l'arc pour que n_2 s'exécute
– le nombre de jeton consommés à chaque exécution de n_2

Le nombre de jetons consommé par un nœud doit être inférieur ou égal au nombre de jetons requis pour son déclenchement.

Les *computation graphs* possèdent la propriété d'être déterministes, l'ordre d'exécution des acteurs n'influence pas les résultats produits par le graphe. Karp et Miller ont également établi les conditions selon lesquelles l'exécution d'un graphe pouvait aboutir à un interblocage ou sous quelles conditions les files du système sont bornées.

2.4.5 Modèle orienté flot de données synchrone (SDF)

Les modèles flot de données synchrone (SDF : Synchronous Data Flow) [27, 46] sont des cas particuliers des *computation graphs* dans lesquels, pour tout arc du graphe, le nombre de jetons requis par un nœud pour son déclenchement est égal au nombre de jetons consommés lors de son déclenchement.

Le nombre de jetons produits et consommés étant constant pour tous les déclenchements, il est possible de déterminer la tailles des files nécessaires entre acteurs et de construire statiquement un ordonnancement du système [16]. L'ordonnancement déterminé doit garantir qu'il n'existe pas d'interblocage et que les acteurs ont toujours le nombre requis de jetons en entrée lorsqu'ils sont exécutés. L'ordonnancement déterminé est ensuite répété périodiquement pour chaque génération d'entrée du système. Chaque répétition de l'ordonnancement est appelée une itération. Un graphe SDF n'a pas toujours d'ordonnancement satisfaisant, on dit alors que le graphe SDF n'a pas de solution. Lorsqu'il est possible de déterminer un ordonnancement valide, le graphe SDF est dit cohérent.

Un programme Lustre (voir 2.1.1) peut être décrit sous forme de graphe SDF. Chaque itération du système représente alors un pas du programme Lustre.

Nous pouvons rapprocher le modèle SDF des réseaux de Petri [55]. La notion de place dans un réseau de Petri peut être vue comme équivalente à la notion d'arc dans un graphe SDF et les transitions peuvent être rapprochées des nœuds d'un graphe SDF. Une place possède un nombre de jetons initial. Chaque transition entre des places sources et destinataires est activée lorsqu'un nombre défini de jetons est présent dans les places source de la transition. Lorsque cette transition est tirée, le nombre de jetons correspondant est supprimé des places sources et un nombre de

jetons prédéfini est ajouté dans les places destinataires de la transition. SDF est en réalité dérivé d'une sous-classe de réseau de Petri, les *marked event graph*.

Dans le modèle *marked event graph* [21], une place a exactement une transition en entrée et une en sortie. Une place peut alors être vue comme une file de jetons. Le nombre de jetons consommés et produits par chaque transition (poids) est égal à 1. Cette simplification structurelle permet de prouver des propriétés usuellement indécidables sur les réseaux de Petri généraux, par exemple, le problème de la terminaison. Ces modèles sont déterministes et confluents : tous les ordres d'exécutions des transitions "tirables" finissent par autoriser les mêmes comportements, simplement plus ou moins décalés dans le temps.

Les SDF sont un raffinement des *marked event graphs* dans lequel on associe des poids divers aux transitions. Un graphe SDF est dit homogène et équivalent à un *marked event graph* si, pour toute transition, les poids d'entrée et de sortie sont égaux.

2.4.6 Modèle utilisé

Les systèmes que nous étudions sont décrits graphiquement sous forme de graphes orientés flot de données. Cependant, contrairement aux modèles orientés flot de données, les composants ne sont pas activés en fonction de leurs données d'entrée. Nous choisissons d'étudier des systèmes dont l'exécution n'est pas dirigée par les données mais dirigée par le temps (time-triggered). Ce sont donc des systèmes dont l'exécution est orientée flot de contrôle.

D'autre part, dans un modèle flot de données, tout jeton entrant dans le système parcourt tout le graphe. Dans notre cas, nous acceptons que certaines données ne parcourent pas tout le graphe, notamment à cause de certains rapports de périodes entre composants. Par exemple, dans un système périodique, si un composant producteur transmet des données à un composant destinataire ne lisant qu'une donnée à chacun de ses pas et que ce composant est de période d'activation deux fois plus petite que le producteur, toutes les données émises par le producteur ne pourront pas être traitées par le destinataire.

Le mécanisme de marquage de jetons semble être proche du mécanisme d'association de données que nous utilisons dans cette thèse. Effectivement, nous nous basons sur un principe de marquage pour associer des données. Cependant, la notion de génération utilisée sous-entend l'existence d'une source de données unique ou d'une synchronisation entre sources afin de fournir des valeurs sur toutes les entrées à chaque itération du système. Dans notre modèle, les fréquences d'entrée des données dans le système peuvent être indépendantes. La notion de génération de données globale et d'itération n'est donc pas adaptée. L'association des données que nous réalisons en entrée de certains composants ne se fait pas en fonction d'une source globale mais en fonction de composants spécifiques. De plus, le mécanisme de marquages des jetons dans les systèmes orientés flot de données est réalisé afin de ne pas gérer des files FIFO sur chaque entrée de bloc. Le marquage sert uniquement à mémoriser l'ordre logique d'arrivée des données. L'utilisation de files FIFO rendrait inutile ce marquage.

Le principal objectif des travaux sur les modèles flot de données est de fournir un ordonnancement des systèmes décrits sous forme de graphes afin que tous les jetons soient traités et que les files utilisées soient bornées. Notre optique est totalement différente. Nous cherchons à garantir des associations cohérentes de données tout en prenant en compte et en imposant le minimum de contraintes sur l'ordonnancement du système analysé. Nous ne réalisons pas de calcul d'ordonnancement. De plus, la problématique d'association de données telle que nous la considérons n'est pas abordée dans les modèles orientés flot de données.

De par le modèle d'exécution que nous avons choisi et les problèmes considérés, nous ne pouvons donc pas nous appuyer sur les théories des flots de données pour résoudre notre problématique. Seuls les systèmes monopériodiques peuvent s'y rapporter. Dans la suite, nous conserverons uniquement l'aspect descriptif des graphes orientés flot de données.

2.5 Différentes notions de cohérence

De nombreuses approches consacrées à la vérification de propriétés temporelles des systèmes distribués se basent sur l'étude du comportement temporel des tâches. Nous nous différencions de ce travail en nous concentrant sur les données échangées et leurs valeurs au lieu de nous centrer sur les tâches. Ce point de vue permet d'analyser des systèmes sans connaître leur ordonnancement final ou leur implantation (par exemple, le nombre de processeurs utilisés). Notre approche nous permet également de gérer des systèmes composés de boîtes noires que nous ne pouvons pas contraindre afin qu'elles aient un "bon" comportement. Par exemple, nous ne pouvons pas les contraindre afin qu'elles respectent l'ordre causal des opérations. Leurs fréquences peuvent également être non configurables Cela est souvent vrai lorsqu'il s'agit de la réutilisation de composants matériels.

La cohérence des systèmes distribués est essentiellement gérée d'un point de vue logique, se concentrant par exemple sur la préservation d'un ordre causal ou total entre les opérations du système. Certains travaux ajoutent à cet ordre des contraintes temporelles à respecter par le système.

Des approches se basant sur les données plutôt que sur les tâches qui produisent ces données sont présentes dans le domaine des bases de données. L'essentiel des approches proposées se concentrent sur la fraîcheur des données prises individuellement. Cependant certaines approches proposent également une notion de cohérence mutuelle.

2.5.1 Cohérence dans les systèmes distribués

Dans le domaine des systèmes distribués temps réel, la notion de cohérence a de multiples définitions. Dans [66], les auteurs distinguent deux aspects différents de la cohérence : la cohérence d'ordre et la cohérence temporelle. La cohérence d'ordre impose un ordonnancement des opérations des différents sites du système de façon à ce qu'elles respectent une relation d'ordre donné. La cohérence temporelle ajoute à la cohérence d'ordre des contraintes temporelles et analyse au bout de combien de temps les effets d'une opération sont perçus par l'ensemble du système.

2.5.1.A Cohérence sans contrainte temps réel

Dans un système distribué, l'*histoire globale* H du système est un ensemble partiellement ordonné d'opérations se déroulant sur tous les sites du système. La séquence des opérations se déroulant sur le site i est notée H_i. Si a se déroule avant b dans H_i, nous disons que a précède b dans l'ordre du programme. Quand on ne s'intéresse qu'à la cohérence, les opérations de H sont soit des lectures soit des écritures.

Parmi les modèles de cohérence les plus répandus, nous pouvons citer la *cohérence séquentielle*. Celle-ci impose uniquement que l'histoire globale du système soit sérialisable, les contraintes temps réel ne sont pas prises en compte. Dans ce modèle, défini par Lamport [42], il n'est pas garanti que chaque opération de lecture retourne la valeur la plus récente par rapport au temps réel. Une *sérialisation* S d'un ensemble

d'opérations D est une séquence linéaire contenant toutes les opérations de D, respectant l'ordre partiel de D et telle que chaque opération de lecture d'un objet retourne la valeur écrite par l'opération d'écriture la plus récente sur cet objet dans l'ordre de S. La cohérence impose uniquement que le résultat de toute exécution soit le même que si les opérations de tous les sites étaient exécutées dans un ordre séquentiel pour lequel les opérations de chacun des sites apparaissent dans l'ordre spécifié par son programme.

Dans la *cohérence causale*, un modèle plus faible que la cohérence séquentielle, les opérations liées par un lien de causalité doivent être perçues dans le même ordre par tous les sites du système alors que les opérations concurrentes peuvent être observées dans des ordres différents. La cohérence causale est suffisante pour les applications qui supportent des partages entre les utilisateurs distribués. Elle est exploitée dans les systèmes de transmission de messages et dans les mémoires partagées.

Une opération a précède causalement b ($a \rightarrow b$) si une des conditions suivantes est vérifiée [43] :

- a et b sont exécutés sur le même site et a est exécuté avant b
- b lit la valeur d'un objet écrit par a
- Il existe une opération c telle que : $a \rightarrow c$ et $c \rightarrow b$

Deux opérations sont concurrentes si elles ne vérifient aucune de ces relations. L'ensemble des opérations de H_i ajouté à l'ensemble des opérations d'écriture de H est notée H_{i+w}. L'histoire globale H respecte la cohérence causale si, pour tous les sites i, il existe une sérialisation de H_{i+w} respectant l'ordre causal.

La notion de temps n'est pas intégrée dans la cohérence causale ou séquentielle. Mais des travaux permettent d'intégrer des contraintes temporelles en se basant sur ces types de cohérence.

2.5.1.B Cohérence avec prise en compte de contraintes temps réel

Aux notions de cohérences précédentes, des extensions sont apportées pour prendre en compte des contraintes temps réel. Par exemple, l'histoire globale d'un système satisfait la condition de *linearisabilité* s'il existe une sérialisation de H qui respecte l'ordre induit par les temps effectifs des opérations [32]. La *sérialisabilité stricte* est définie dans le cadre d'histoires composées de transactions. Elle impose l'existence d'une sérialisation qui respecte l'ordre temps réel des transactions [53].

Dans [72], les auteurs utilisent un modèle à évènements discrets. Le système est découpé en composants qui communiquent par évènements discrets portant des estampilles temporelles. Ces estampilles sont utilisées par des ports spécifiques qui doivent recevoir les évènements avant que le temps réel n'excède leur estampille temporelle à une précision près. Lorsque cela est nécessaire, ces estampilles sont liées au temps physique. Les auteurs proposent alors un modèle de programmation appelé PTIDES (Programming Temporal Integrated Distributed Embedded Systems) dont l'objectif est de préserver le déterminisme du système sans imposer une exécution totalement ordonnée.

La méthode proposée se base sur une analyse statique des relations causales entre évènements. Une première notion d'*interface causale* permet d'identifier la dépendance entre évènements d'entrée et de sortie. Un graphe est alors tracé en liant par un arc les ports du système liés par la notion d'interface causale. La notion de *dépendance pertinente* est ensuite définie afin de spécifier les relations entre ports d'entrée d'un composant. Le port p_1 est lié au port p_2 par cette relation si un évènement se produisant sur p_1 affecte un évènement de sortie qui peut également dépendre de p_2. Le graphe précédant est alors transformé en fusionnant les ports liés par dépendance pertinente. Le graphe obtenu permet de définir un *ordre pertinent*. Il s'agit d'un ordre

partiel entre les évènements que le système doit respecter lors de son exécution. La notion d'évènement minimal est défini afin de déterminer, lors d'une exécution, l'évènement qui doit être traité en priorité parmi tous les évènements en attente. Afin de le définir, l'ordre des évènements établi précédemment est utilisé. Les auteurs utilisent également les estampilles temporelles de certains évènements en ajoutant la notion de Δ-minimal : les estampilles temporelles des évènements sont alors utilisées pour détecter, parmi les évènement minimaux, les évènements qui doivent être traités au plus tôt.

Le résultat de cette méthode produit une stratégie d'exécution n'imposant pas un ordre total des évènements sans sacrifier le déterminisme de l'exécution. Un programme PTIDES est défini comme déployable si le système est capable de respecter les contraintes temporelles définies sur les temps de traitement des évènements. Le progrès de ces travaux est de proposer une approche conservatrice plus souple que celle de Chandy et Misra [18] qui impose de plus grands délais de traitement des évènements et qui respecte donc plus difficilement les contraintes temps réel des systèmes.

Dans [66], les auteurs définissent deux types de cohérences temporelles : la *cohérence causale temporisée* et la *cohérence séquentielle temporisée*. Ils ajoutent aux cohérences causale et temporelle une contrainte temporelle imposant que si une opération d'écriture est effectuée au temps t, la valeur écrite devra être visible de tous les sites du système au temps $t + \Delta$, Δ étant un paramètre de l'exécution. L'implémentation de ces principes est réalisée à l'aide d'un protocole basé *durée de vie* [65]. Dans ce cadre, une notion de *cohérence mutuelle* entre deux objets est définie. Deux valeurs sont mutuellement cohérentes si elles coexistent, c'est-à-dire si leurs domaines de validité temporelle se superposent. Cette définition sert à définir comme cohérente la mémoire cache d'un site dans laquelle sont stockées des copies d'objets. Le cache est cohérent si tous les objets sont mutuellement cohérents deux à deux.

Dans les protocoles Δ-causaux, les messages doivent respecter l'ordre causal et possèdent également une durée de validité d'un temps Δ. Après ce délai, les données ne peuvent plus être utilisées. Ce protocole est essentiellement utilisé dans les applications multimédia temps réel. Les recherches sur ce thème [9, 64] se sont concentrées sur les adaptations (ajustement du Δ) et des optimisations des transmission (réduction du seuil de bande passante en minimisant les informations transportées). L'objectif de la Δ-causalité est de favoriser la latence même si la non prise en compte d'un message trop en retard peut amener à la brisure de chaînes causales.

De nombreux travaux adoptent le même type d'approche que les articles cités dans ce paragraphe : une première analyse permet d'établir un ordre de traitement des évènements sur chaque site d'un système distribué afin que les comportements de chaque site soient cohérents entre eux. Une prise en compte de certaines contraintes temporelles s'ajoute alors à cette première analyse. La cohérence traitée est essentiellement un respect de l'ordre causal avec une volonté de prise en compte des contraintes temporelles sur les délais d'exécution des évènements. Dans certains travaux, l'ordre est alors optimisé afin de respecter les contraintes temporelles du système. Dans d'autres travaux, l'optique choisie est d'analyser l'impact de différentes méthodes de gestion de la concurrence sur le respect de ces contraintes temporelles. Pour une liste étendue de références, le lecteur pourra se référer à [66]. Dans nos travaux, nous ne cherchons pas à déterminer un ordonnancement des opérations, que ce soit du point de vue causal ou temporel. Nous ne prenons pas non plus un ordonnancement fixé pour analyser les systèmes, nous souhaitons pouvoir analyser un système sans avoir beaucoup d'information sur son ordonnancement final.

La latence des systèmes est souvent un critère temporel analysé. Dans notre cas, nous recherchons des associations cohérentes de messages propagés par différents chemins. La latence est imposée par le chemin le plus lent et les messages sur les chemins

les plus rapides sont différés pour permettre une association de données cohérente. Plus globalement, nous ne cherchons pas à optimiser des critères temporels de fonctionnement du système.

Dans ces études, lorsque des estampilles temporelles sont utilisées par les données, elles identifient le plus souvent le domaine de validité de ces données. Nous pouvons les distinguer des estampilles gardant la trace des dépendances entre données, telle que celles proposées par Lamport. Nous adopterons pour notre part un mécanisme d'estampillage identifiant les dépendances entre données mais nous définirons un mécanisme codant ces dépendances de façon plus précise, notre problématique nécessitant le respect de contraintes différentes de la causalité.

2.5.2 Cohérence dans les bases de données

2.5.2.A Cohérence individuelle

Dans une base de données temps réel, les objets enregistrés sont des images du monde réel, telles que la position d'un satellite ou la vitesse d'un avion. Ces objets deviennent donc invalides après un certain temps qui dépend de la vitesse d'évolution des objets réels qu'ils représentent. Ainsi, un intervalle de validité temporelle est défini pour chaque objet de la base de données. C'est alors au système gérant la base de données de contrôler la mise à jour des objets de façon à ce qu'ils ne dépassent pas leurs intervalles de validité temporelle et que lorsqu'une transaction valide (réalisation d'un commit), les valeurs utilisées par cette transaction n'aient pas dépassé leurs domaines de validité, c'est-à-dire que les données ne soient pas devenues obsolètes. De nombreuses approches existent dans la littérature pour déterminer un ordonnancement des transactions maintenant la fraîcheur des objets. Cette cohérence des objets par rapport à leur fraîcheur est aussi appelée *cohérence absolue*.

La sémantique des variables et leur domaine temporel de validité sont utilisés dans [70] pour optimiser l'ordonnancement des transactions dans les bases de données. Dans [6], des contraintes OCL (UML Object Constraint Language) sont utilisées pour définir le domaine de validité des variables. Une variante de TCTL (Timed Computation Tree Logic) est utilisée pour vérifier le comportement du système et prévenir l'utilisation d'une variable hors de son domaine de validité.

Dans [60], les auteurs reprennent les principaux concepts de cohérence des bases de données temps réel. Cependant, seule la gestion de la fraîcheur des données est évoquée. L'article présente également différentes méthodes pour gérer cette contrainte en contrôlant l'ordonnancement des transactions et les notions de qualité de service pouvant être associées aux bases de données temps réel. Des applications dans des domaines connexes (réseaux de capteurs, bases mobiles, services web temps réel) des théories présentées sont ensuite évoquées.

De nombreux autres travaux traitent de la cohérence temporelle dans les bases de données. Les auteurs cherchent la plupart du temps à déterminer un ordonnancement des transactions permettant de garantir une fraîcheur déterminée (voir [60] pour une liste de références). L'approche retenue se base souvent sur des transactions périodiques mais les variantes sont multiples (par exemple, [69] se distingue en proposant un modèle d'exécution sporadique).

Ces approches se concentrent uniquement sur le maintien de la fraîcheur de chaque objet pris individuellement sans prendre en compte leur fraîcheur mutuelle. Cela peut alors aboutir à une vue incohérente du système. En effet, plusieurs objets peuvent être liés entre eux. C'est le cas notamment pour des documents mis en ligne, par exemple sur une page donnant des informations accompagnées d'images ou de vidéos. Si l'information est modifiée, il faut s'assurer que les documents l'accompagnant sont

toujours cohérents avec la nouvelle information et pas uniquement qu'ils sont assez frais.

2.5.2.B Cohérence mutuelle

Dans [33], les auteurs prennent conscience de la problématique de la cohérence d'un ensemble de données et soulignent que les objets actualisés par différents capteurs d'un système peuvent être liés les uns aux autres. Il est donc important de préserver la fraîcheur mutuelle des objets en plus de leur fraîcheur individuelle. La prise en compte des relations entre objets aboutit à une notion de *cohérence mutuelle*.

Les auteurs déterminent si un ordonnancement de requêtes et de mises à jour donné maintient la cohérence mutuelle des objets. Les auteurs se placent dans le cadre d'exécutions de requêtes périodiques et non préemptives. Ils proposent tout d'abord des formules donnant la valeur maximale des écarts mutuels parmi un ensemble de lectures de données. Connaissant les paramètres du système, il est alors possible de vérifier si l'exigence de cohérence mutuelle fixée est respectée. Une approche de conception est également présentée afin de choisir des périodes et des échéances relatives permettant de garantir la cohérence mutuelle. Une autre approche permet de déduire si un ensemble de requêtes avec ses échéances relatives et ses périodes peut garantir une cohérence mutuelle. D'une certaine manière, cet article cherche à identifier quelles sont les contraintes que l'ordonnancement des requêtes doit respecter pour garantir une cohérence mutuelle.

Dans cet article, une donnée est dite dépendante d'une autre s'il est nécessaire que leurs instants de mise à jour soient assez proches pour donner un aperçu cohérent du système réel qu'elles transcrivent. Notons que la notion de dépendance que nous utilisons dans cette thèse est différente. Nous considérons qu'une donnée dépend d'une autre si cette autre donnée est utilisée pour calculer la première.

L'objectif du mantien de la cohérence mutuelle décrite est de garantir que les mises à jour des données produisent une vue cohérente du système. Dans nos travaux, la cohérence est définie par le fait d'utiliser des données dont les historiques de dépendances sont cohérents. Notre notion de cohérence ne s'intéresse pas à une cohérence temporelle des capteurs du système mais à une cohérence logique des calculs réalisés.

De plus, l'optique choisie est essentiellement de déterminer les contraintes d'ordonnancement permettant de garantir le maintien de la cohérence mutuelle. De notre côté, nous cherchons à garantir la cohérence des données sans prendre en compte ni produire des contraintes d'ordonnancement.

Dans [62], les auteurs s'intéressent à la cohérence temporelle des données en terme de fraîcheur mais également de cohérence mutuelle. Chaque objet du système est daté. Une notion de dispersion mesurant la différence entre les âges de deux objets est ajoutée. Les auteurs s'intéressent alors à la cohérence temporelle des données en fonction de l'âge et de la dispersion des données. Les objets sont *temporellement cohérents* si leurs âges et leurs dispersions sont suffisamment petits pour correspondre aux besoins de l'application.

Les auteurs distinguent les *objets images* qui sont périodiquement produits par la sortie de capteurs, et les *objets dérivés* qui sont calculés à partir de valeurs d'un ensemble d'objets. Les objets images sont périodiquement mis à jour à partir d'un objet réel. Une image a ainsi de multiples versions d'image créées à chaque mise à jour et datées. L'âge d'une image est l'âge de sa version la plus récente. Les périodes de mise à jour des images dépendent de la vitesse à laquelle la valeur de l'objet réel change. Un objet dérivé est lui calculé à partir d'un ensemble d'objets qui peuvent être des objets images ou des objets dérivés. Il est daté de l'âge de l'objet le plus vieux de l'ensemble utilisé pour le calculer.

Un ensemble d'objets est considéré comme *absolument temporellement cohérent* si tous les objets ont un âge inférieur à un certain seuil. Un ensemble d'objet est *relativement temporellement cohérent* si tout couple d'objets de l'ensemble a une dispersion inférieure à un certain seuil. Les auteurs évaluent alors l'impact que peuvent avoir différents contrôles de la concurrence (pessimiste et optimiste) et différents algorithmes d'ordonnancement préemptif (rate monotonic et earliest deadline first) sur la cohérence des données. Des simulations sont utilisées pour mesurer le pourcentage de transactions temporellement incohérentes, c'est-à-dire lisant des données absolument ou relativement incohérentes. Pour analyser ces résultats, d'autres mesures sont également effectuées telles que la dispersion maximale des données, le taux de préemption...

L'orientation de cet article est différente de notre travail. Les auteurs partent d'un mode d'ordonnancement donné pour analyser son impact sur la cohérence des données. Nous cherchons une méthode permettant de garantir la cohérence des données quel que soit l'ordonnancement choisi. La notion de cohérence, bien que similaire, n'est pas la même. Un objet est daté uniquement à partir de l'âge des objets utilisés pour le calculer et non par la date à laquelle il a été calculé. De plus, sa date ne contient qu'une information : l'âge de l'objet le plus vieux utilisé pour le calculer. Ainsi, un objet ne mémorise pas son historique de dépendance.

Dans une certaine mesure, notre notion de cohérence stricte pourrait être gérée à l'aide de la notion de dispersion en imposant une dispersion égale à zéro. Cependant, cela impose alors de lourdes contraintes inutiles sur tout le système. En effet, cela impose une cohérence par rapport à l'environnement pour tous les composants du système, ce qui ne correspond pas à notre objectif. Si nous considérons notre cohérence relâchée, elle ne peut pas être gérée par la notion de dispersion à cause du choix de datation réalisé. En effet, nous avons besoin de mémoriser l'historique de dépendance de chaque objet afin de gérer des systèmes complexes où des configurations problématiques peuvent être imbriquées. La perte d'information engendrée par la datation des objets ne nous permettrait pas alors de contrôler la cohérence dans ces systèmes.

Remarquons également la problématique que pourrait poser des boucles dans des systèmes gérés par le principe présenté dans cet article. La date d'un objet étant celle du plus vieil objet utilisé, si un objet boucle sur lui même, sa date ne change jamais. Impossible alors de considérer la cohérence d'un objet faisant l'addition de ses entrées successives. Nous verrons également dans nos travaux que la présence de boucles dans le système impose de leur appliquer un traitement particulier.

Dans [29], les auteurs déterminent un algorithme calculant quelles données ont besoin d'être réactualisées en prenant en compte les relations entre données. Des données sont en relation si certaines sont utilisées pour en calculer d'autres (données dérivées). Ainsi, si une donnée est mise à jour, l'algorithme décide si les données dérivées doivent l'être également suivant leur domaine de validité. Le calcul non systématique des données dérivées permet de réduire la charge du système.

La grande majorité des travaux incluant une notion de cohérence mutuelle dans les bases de données temps réel s'orientent vers une analyse de l'impact de certains choix d'ordonnancement ou de gestion de la concurrence sur la cohérence mutuelle ou prennent le point de vue inverse en calculant les contraintes à respecter sur l'ordonnancement pour garantir la cohérence mutuelle. Dans nos travaux, nous considérons peu de contraintes d'ordonnancement et nous n'en produisons pas non plus.

La notion d'objet dérivé peut être rapproché de notre problématique où nous considérons des données produites à partir d'autres données. Cependant, en général, lorsque les approches incluent l'analyse d'objets dérivés, seule la date de l'objet le

plus ancien utilisé pour calculer ces objets dérivés a de l'importance. Dans notre cas, c'est toute la chaîne de dépendance entre données que nous devons mémoriser.

2.5.3 Cohérence dans le domaine internet

Les principes de cohérence étudiés dans les bases de données sont également applicables dans le domaine internet. Une cohérence individuelle des données est utilisée à partir de domaine de validité temporelle mais il est également mis en avant que cette cohérence individuelle n'est pas suffisante, certains objets pouvant être liés entre eux. Dans [67], l'objectif est de garantir une cohérence des réplicats (objets caches sur un proxy) respectant les données d'origine (pages sur un serveur). Les auteurs proposent un algorithme pour déterminer un taux adéquat de mise à jour du serveur afin que les réplicats ne divergent pas trop. L'algorithme est ensuite adapté pour maintenir une cohérence faible d'un ensemble de pages. On retrouve ici un concept proche de la notion de dispersion dans les bases de données.

De nombreux travaux se concentrent sur le maintien de la cohérence entre objets cache. Dans [17], différentes approches sont analysées et la faisabilité d'une cohérence forte est mise en avant. Dans [71] les auteurs analysent l'impact sur la performance du système d'un protocole de mise à jour de cache orienté serveur.

La problématique des applications internet réunit les deux aspects de cohérence traités précédemment. D'une part, le réseau internet peut être considéré comme étant un système distribué dans lequel tous les sites doivent avoir des comportements cohérents entre eux. D'autre part, les applications internet peuvent exploiter des bases de données temps réel afin de fournir des informations, des prévisions, les derniers indices boursiers... Dans ce cas, toutes les problématiques des bases de données temps réel leur sont associées.

2.5.4 Conclusion sur les différentes notions de cohérence

Dans les systèmes distribués et/ou temps réel, la notion de cohérence des données peut prendre diverses définitions. Dans les systèmes distribués, il peut s'agir tout d'abord qu'un certain ordre entre opérations soit respecté sur les différents sites. Des contraintes temporelles peuvent être également ajoutées à ce principe afin de contraindre qu'une opération soit effectuée dans un certain délai. Dans le domaine des bases de données temps réel, la gestion de la cohérence consiste en la gestion de la mise à jour des données afin que les données stockées respectent une certaine fraîcheur. Dans ce cadre, la cohérence mutuelle impose que les données respectent d'une part leur fraîcheur individuelle mais également que leurs âges ne soient pas trop dispersés. Enfin, le domaine internet peut être considéré comme regroupant les problématiques des systèmes distribués et des bases de données.

Bien que certains types de cohérence passés en revue présentent des similitudes avec nos travaux, aucun d'eux ne présente une totale adéquation avec notre propre notion de cohérence. De plus, dans un contexte temps réel, l'objectif principal de ces travaux est soit de produire un ordonnancement garantissant la cohérence des donnée, soit d'analyser l'impact d'un ordonnancement donné sur le maintien de la cohérence ou sur les performances du système. Nous ne pouvons pas nous baser sur ces concepts car le but de notre analyse n'est pas de produire un ordonnancement ni de prendre en compte de lourdes contraintes sur ce dernier.

Chapitre 3

Association cohérente de données

3.1 Définition de la cohérence

Dans un calcul temps réel réparti, il est intéressant d'observer d'où proviennent les données utilisées par les composants. Si l'on se situe dans une approche de développement par composant, cette optique n'est pas évidente. En effet, dans cette approche, l'idéal est de disposer de composants développés indépendamment que l'on assemble en fonction de l'objectif du système final. Le problème est alors de vérifier que le système ainsi conçu à partir de briques indépendantes a un bon comportement global. Dans cette thèse, nous nous concentrerons sur cette problématique d'échanges de données entre composants.

3.1.1 Configurations étudiées

Le type de configuration que nous étudions est le suivant : à chacun de ses pas d'exécution, un composant C produit des données pour plusieurs composants. Ces composants destinataires utilisent les données fournies par C pour en produire de nouvelles et les envoyer à leur tour à d'autres composants non reliés entre eux. De la même manière, ces derniers utilisent les données et en produisent de nouvelles utilisées par d'autres composants. Le chemin des données se poursuit ainsi de proche en proche sur ce principe. Ainsi, les données initiales produites par C parcourent indirectement des chemins indépendants au travers de divers composants.

Enfin, supposons que ces différents chemins se rejoignent sur un composant C'. Ce dernier utilise donc des données qui dépendent indirectement de données fournies par C. Nous disons alors qu'un problème d'association de données au niveau des entrées de C' peut se poser. Nous souhaitons que les données utilisées par C' dépendent de données produites lors du même pas d'exécution de C.

Le figure 3.1 illustre le type de configuration étudié. Nous apportons les précisions suivantes :

- Les chemins reliant C et C' sont au minimum au nombre de 2 mais il n'existe pas de maximum.
- Les chemins n'ont pas nécessairement le même nombre de composants.
- Les composants C et C' peuvent être reliés directement. Ainsi, sur un ou plusieurs chemin, C' peut utiliser directement les données fournies par C.

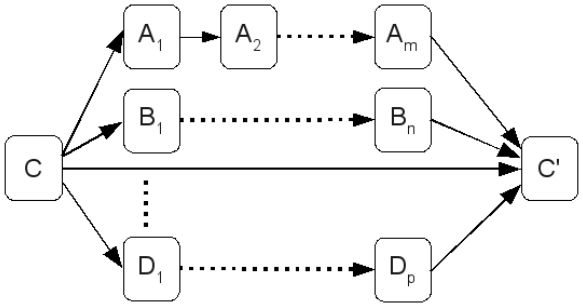


FIGURE 3.1 – Type de configuration étudiée

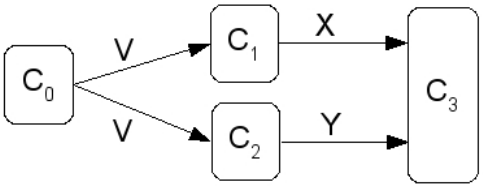


FIGURE 3.2 – Système présentant un problème d'association de données

– C peut avoir une ou plusieurs sorties. Il peut donc envoyer une même donnée sur tous les chemins ou des données différentes.

Afin de décrire les interactions entre données, nous définissons la notion de dépendance entre données et celle de propagation d'une donnée.

Définition 1 (Dépendance entre données) *Une donnée b dépend d'une donnée a si a est utilisée par un composant pour calculer b. Nous dirons également par transitivité que c dépend de a si le calcul de c utilise une donnée b qui elle-même dépend de a.*

Définition 2 (Propagation d'une donnée) *Une donnée d se propage d'un composant C à C' si C produit d et C' utilise une donnée dépendant de d.*

Nous étudions donc des systèmes dans lesquels un composant C' utilise plusieurs entrées dépendant de données produites par un unique composant C et qui se sont propagées de C à C' via différents chemins de composants.

Pour illustrer notre problématique, nous utilisons le système de la figure 3.2. Dans ce système, le composant C_3 utilise des données fournies par C_1 et C_2. Ces deux composants produisent leurs données à partir de données fournies par un même composant C_0. Ainsi, les entrées de C_3 dépendent des données de C_0.

3.1.2 Cohérence stricte des données

L'objectif de cette thèse est de gérer des associations cohérentes de données. Dans le cas où C' utilise des données dépendant de C, nous voulons que ces données dépendent de données produites lors du même pas d'exécution de C. Ce principe permet de garantir la correction de calculs ou de traitements répartis.

La figure 3.3 représente un exemple simple d'application de la cohérence. Le système permet de calculer, pour tout x fourni par C_1, le résultat de $2x + 3x$. Des composants indépendant se chargent de calculer $2x$ pour C_2 et $3x$ pour C_4. Les résultats qu'ils produisent sont ensuite utilisés par le composant C_4 qui en fait l'addition.

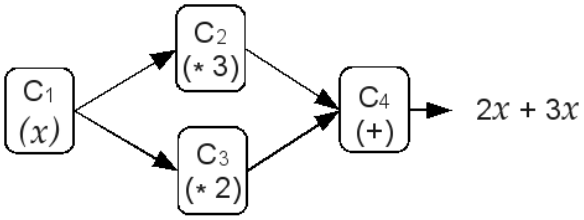


FIGURE 3.3 – Calcul de $2x + 3x$

Nous voyons que pour que les résultats de C_4 aient un sens, C_4 doit utiliser des résultats calculés par C_2 et C_3 à partir du même x, c'est à dire à partir du même pas d'exécution de C_1.

Définition 3 (Données (strictement) cohérentes) *Si un composant C' utilise plusieurs données dont les valeurs dépendent de sorties d'un même composant C, alors ces données d'entrée de C' seront considérées comme strictement cohérentes si elles dépendent de données produites par le* même pas *de calcul de C. Nous simplifierons en disant simplement que les données sont cohérentes.*

Définition 4 (Exécution cohérente, association cohérente) *L'exécution d'un composant est cohérente si le composant utilise des données cohérentes pour réaliser son pas de calcul. Nous disons également que le composant réalise une association de données cohérente. L'exécution d'un système est cohérente si tous les composants s'exécutent de façon cohérente.*

Les chronogrammes des figures 3.4 et 3.5 illustrent deux exécutions différentes du système de la figure 3.2. Sur ces figures, la queue d'une flèche représente le moment où une donnée est produite et sa tête représente le moment où cette donnée est disponible en entrée d'un composant.

Lors de l'exécution illustrée par la figure 3.4, pendant le pas d'exécution S, C_3 utilise la donnée X_b fournie par C_1 et la donnée Y_b fournie par C_2. Ces données sont calculées en utilisant la même donnée V_b produite par C_0 au pas B. Les entrées utilisées par C_3 pour le pas S proviennent du même pas d'exécution de C_0. Elles sont donc cohérentes.

Au contraire, lors de l'exécution illustrée par la figure 3.5, pendant le pas d'exécution S, C_3 utilise la donnée X_b fournie par C_1 et la donnée Y_a fournie par C_2. X_b est calculée à partir de la donnée V_b provenant du pas B alors que Y_a est calculée à partir de V_a provenant du pas A. Les entrées utilisées par C_3 dépendent de pas de calcul différents de C_0. Elles sont donc considérées comme incohérentes.

Notons que la définition de la cohérence peut s'appliquer à des systèmes évoluant dans le temps. Une architecture statique n'est pas nécessaire : au cours d'une exécution, les liens entre composants peuvent changer et des composants peuvent être ajoutés ou retirés. En effet, aucune référence à l'architecture du système n'est requise dans notre définition, seules les dépendances entre données et les pas des composants sont analysés.

3.1.3 Cohérence relâchée des données

Nous relâchons la définition 3 de données cohérentes en permettant que les données ne dépendent pas strictement de données produites par le même pas d'exécution de C. Nous tolérons un écart temporel entre les dates de début d'exécution de C qui

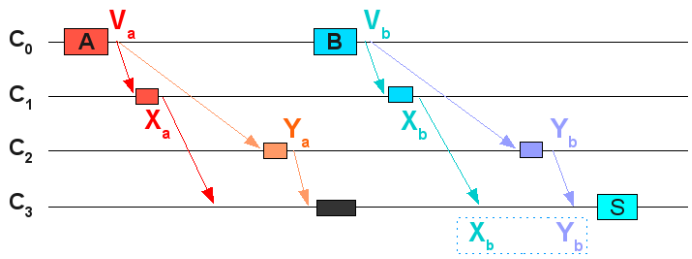


FIGURE 3.4 – Exécution cohérente

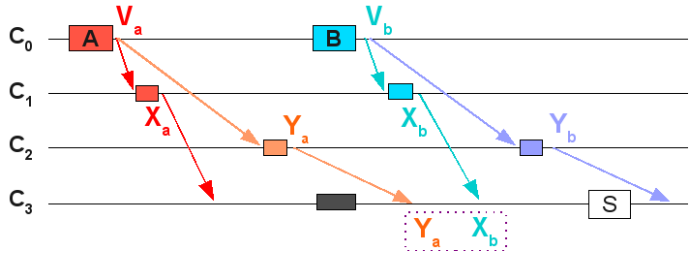


FIGURE 3.5 – Exécution incohérente

ont produit les données dont dépendent les données utilisées par C'. La définition 3 devient donc :

Définition 5 (Données cohérentes avec une tolérance τ) *Si un composant C' utilise plusieurs données dont les valeurs dépendent de sorties d'un même composant C, alors les données d'entrée de C' seront considérées comme cohérentes avec une tolérance τ si l'écart de temps maximal entre les pas de C dont ces données dépendent est égal au plus au temps τ. Nous employons également le terme de données cohérentes τ-relâchées.*

Ce concept de cohérence relâchée peut être utilisé par exemple lorsque le concepteur préfère que le composant C' utilise des données fraîches plutôt que des données strictement cohérentes mais tout en préservant un certain écart de cohérence entre ces données.

D'autre part, ce concept trouve son utilité dans un autre phénomène. Dans une gestion d'association de données strictement cohérente, il faut que C' reçoive sur toutes ses entrées des données dépendant du même pas de C. Si toutes les entrées ne composent pas un ensemble cohérent, il ne peut pas s'exécuter de façon cohérente. Dans ce cas, suivant le choix du concepteur, soit il n'exécutera pas de pas de calcul, soit il utilisera un vieil ensemble cohérent qu'il a pu composer lors d'un pas précédent.

Si l'on se place dans le cadre de composants périodiques, lorsqu'un composant a une fréquence plus lente que son prédécesseur, ce composant ne va pas pouvoir utiliser toutes les données fournies par son prédécesseur. Ce composant agit ainsi comme un filtre ne laissant se propager que certaines données fournies par son prédécesseur. Par exemple, si un composant C de période T émet des données à destination d'un composant C' de période $2T$, C' ne pourra pas prendre en compte toutes les données produites par C. En moyenne, seule une donnée sur deux émise par C sera utilisée par C'. C' agit donc comme un filtre ne laissant passer en moyenne qu'une donnée sur deux émise produite par C. Si ce phénomène se produit sur plusieurs chemins, il peut arriver que les données reçues sur les entrées d'un composant ne dépendent jamais

du même pas de calcul de C. Dans ce cas, le fait de tolérer un écart de cohérence permet de pouvoir construire plus souvent des ensembles de données cohérents. Nous approfondirons ce problème dans le chapitre 6.

3.2 Modélisation des systèmes étudiés

Nous étudions des systèmes découpés en composants. Pour leur représentation graphique, nous avons besoin des notions de composants, de ports d'entrée et de sortie et de lien de communication entre ces ports. Nous avons également besoin de préciser certaines propriétés des composants et des liens de communication.

Afin de ne pas alourdir inutilement certains schémas et parce que ce type de structuration des systèmes est déjà bien établi, nous n'utilisons pas de formalisme précis. Nous représentons les composants par des boîtes et les liens entre eux par des flèches. De plus, ce choix de formalisme nous permet d'analyser directement nos systèmes comme des graphes orientés.

Si un langage de modélisation précis avait dû être retenu, AADL aurait été choisi. En effet, il contient toutes les concepts qui sont nécessaires à notre analyse.

Remarquons que cette thèse trouve particulièrement son intérêt dans un développement par composant. En effet, idéalement, cette approche consiste en l'assemblage de composants développés indépendamment afin d'en faire un système. Il s'agit alors de vérifier si ce système se comporte "correctement".

3.3 Analyse de graphe

Le système est analysé en tant que graphe orienté. Les composants deviennent des nœuds du graphe et les liens de communication entre eux, des arcs orientés. Au sein de ce graphe, nous cherchons à identifier les configurations provoquant des problèmes d'association de données.

3.3.1 Définitions graphiques

Définition 6 (Chemin) *Un chemin est une séquence de nœuds* $(C_1, C_2, ..., C_n)$ *avec* $n > 1$ *où* $\forall i \in [1..n-1]$*, il existe un arc antre* C_i *et* C_{i+1}*.*

S'il existe plusieurs arcs entre deux nœuds, nous pouvons avoir besoin de les distinguer. Nous noterons alors les chemins sous la forme : $(C_1 \xrightarrow{a_1} C_2 ... \xrightarrow{a_z} C_n)$*, où les* a_i *sont des étiquettes d'arc.*

Définition 7 (Chemin simple) *Un chemin simple est un chemin dans lequel tous les nœuds sont distincts. Soit* $P = (C_1, C_2, ..., C_n)$ *un chemin :*

$$P = (C_1, C_2, ..., C_n) \textit{est simple} \triangleq \forall i : \forall j : i \neq j \Rightarrow C_i \neq C_j$$

Remarquons qu'un chemin simple ne peut pas comporter de boucles. Nous ne traiterons pas ici les systèmes comprenant des boucles entre composants. Ce problème sera traité ultérieurement.

Définition 8 (Chemins séparés) *Deux chemins simples sont séparés s'ils n'ont aucun nœud en commun excepté leurs nœuds initiaux et finaux*

Soit $P_1 = (C_1, C_2, ..., C_n)$ *et* $P_2 = (C'_1, C'_2, ..., C'_m)$ *des chemins simples.*

$$P_1, P_2 \textit{ sont séparés} \triangleq$$
$$(\forall i : \forall j : 1 < i < n \wedge 1 < j < m \Rightarrow C_i \neq C'_j) \wedge (C_1 = C'_1) \wedge (C_n = C'_m)$$

3.3.2 Fuseaux : identification des configurations problématiques

Nous avons identifié qu'une configuration pouvant provoquer des problèmes d'association de données se compose d'un composant C' utilisant des données qui dépendent de données produites par un même composant C. Les données produites par C se propagent vers C' en empruntant différents chemins de données. Graphiquement, nous retrouverons donc deux composants, C et C', reliés par plusieurs chemins. L'identification des configurations problématiques repose sur la notion de fuseaux.

Définition 9 (Fuseau) *Un fuseau entre deux nœuds est l'ensemble des chemins reliant ces nœuds tels qu'il existe au moins deux chemins séparés dans cet ensemble.*

Définition 10 (Source) *La source d'un fuseau est le nœud initial des chemins composant ce fuseau.*

Définition 11 (Puits) *Le puits d'un fuseau est le nœud final des chemins composant ce fuseau.*

Un fuseau est donc la représentation graphique de la configuration problématique identifiée dans la section 3.1. En effet, il illustre le fait qu'un composant utilise des données dépendant, via plusieurs chemins, de données fournies par un même composant initial.

Propriété 1 *Un problème d'association de données peut intervenir entre deux composants si et seulement si il existe un fuseau entre eux.*

En effet, si un fuseau est présent entre deux composants, cela signifie que plusieurs chemins séparés lient les deux composants. Des données dépendant de la source parviennent donc au composant puits via plusieurs chemins pour lesquels rien n'indique que les temps de parcours sont égaux. Un problème de cohérence est donc à gérer sur ces données.

Dans le sens inverse, si un problème de cohérence se pose entre un composant C et C' alors cela signifie qu'il existe au moins deux chemins reliant C à C' et que ces deux chemins sont forcément séparés. Si ces chemins n'étaient pas séparés, cela signifierait :

- soit qu'il existe un composant C'' sur lequel les chemins se regroupent avant d'atteindre C'. Cela signifie que la cohérence des données est à gérer sur C'' et non sur C'.
- soit que les chemins partent de C et se séparent uniquement après un composant C''. La cohérence des données est donc à gérer par rapport à C'' et non à C.

Un problème d'association de données n'étant à gérer que dans le cas où il existe au moins deux chemins séparés reliant C et C', il n'existe que lorsqu'un fuseau est présent entre C et C'.

La figure 3.6 illustre ces configurations. Entre C_0 et C_5 nous trouvons bien deux chemins : $(C_0, C_1, C_2, C_4, C_5)$ et $(C_0, C_1, C_3, C_4, C_5)$. Ces deux chemins ne sont pas séparés car en plus de leurs nœuds d'extrémité, ils partagent les composants C_1 et $C4$. Nous voyons bien ici que ce n'est pas C_0 qui émet des données pour plusieurs composants mais C_1. Nous observons également que le problème de cohérence de données ne se pose pas en entrée de C_5 mais bien sur les entrées de C_4 qui doit regrouper plusieurs chemins provenant de C_1. Par contre, nous constatons qu'il existe bien deux chemins séparés entre C_1 et C_4. Cette présence de fuseau nous indique qu'il faut gérer la cohérence des données sur C_4 par rapport à C_1.

Notons qu'un système peut comporter plusieurs fuseaux. Ces fuseaux peuvent également être imbriqués entre eux. Ainsi, le puits d'un fuseau peut être la source

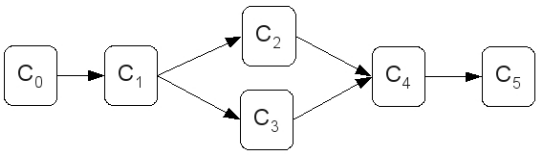


FIGURE 3.6 – Exemple de chemins non séparés

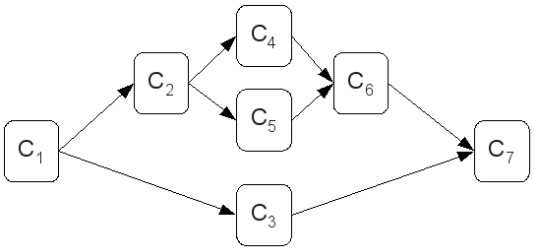


FIGURE 3.7 – Exemple de fuseau

d'un autre, les sous-chemins d'un fuseau peuvent composer un fuseau, le puits d'un fuseau peut être le composant quelconque d'un autre fuseau...

Sur la figure 3.7, l'ensemble des trois chemins simple $\{(C_1, C_2, C_4, C_6, C_7)$, $(C_1, C_2, C_5, C_6, C_7)$, $(C_1, C_3, C_7)\}$ est un fuseau de source C_1 et de puits C_7.

L'ensemble $\{(C_2, C_4, C_6)$, $(C_2, C_5, C_6)\}$ forme également un fuseau de source C_2 et de puits C_6.

3.3.3 Fuseaux imbriqués, différents types de fuseaux

Un système peut comporter plusieurs fuseaux. Ce qui nous importe alors est de pourvoir déterminer si un fuseau influence le comportement d'un autre. Si les fuseaux ne partagent aucun composant, il est évident que leurs comportements ne vont pas s'influencer. Par contre, des difficultés peuvent se présenter dans le cas de fuseaux imbriqués. Nous détaillerons cette problématique dans le chapitre 6 et, dans une moindre mesure, dans le chapitre 4.

Définition 12 (Fuseaux imbriqués) *Deux fuseaux F et F' sont imbriqués si leurs chemins partagent au moins un composant :*

$$\exists P = (C_1, C_2, ..., C_n) \in F \wedge \exists P' = (C'_1, C'_2, ..., C'_m) \in F' :$$

$$\exists i : 1 \leq i \leq n \wedge \exists j : 1 \leq j \leq m : C_i = C'j$$

Nous distinguons également la notion de sous-fuseau qui est un cas particulier de fuseau imbriqué.

Définition 13 (Sous-fuseaux) *Un fuseau F' est un sous-fuseau de F si F' est composé de sous-chemins de F :*

$$\forall P' = (C_i, C_{i+1}, ...C_j) \in F', \exists P = (C_1, C_2, ...C_n) \in F :$$

$$1 \leq i \leq j \leq n \wedge \forall x \in [i..j], C_x \in P$$

Sur la figure 3.7, le fuseau entre C_2 et C_6 est un sous-fuseau du fuseau entre C_1 et C_7.

Définition 14 (Fuseau simple) *Un fuseau simple est un fuseau ne contenant pas de sous-fuseau*

Définition 15 (Fuseau principal) *Un fuseau principal est un fuseau n'étant pas sous-fuseau d'un autre.*

3.4 Formalisation de la cohérence

Nous avons défini informellement ce que nous considérons comme étant une exécution cohérente dans la section 3.1. Nous allons dans ce qui suit proposer deux définitions formelles de cette notion. La première repose sur une relation dite "d'influence", la seconde repose sur une équivalence de graphe. Cependant, dans la suite des travaux, nous nous appuierons uniquement sur la relation d'influence car cette première définition permet plus de souplesse et est plus facile à manier.

3.4.1 Formalisation par la relation d'influence

Nous considérons une exécution d'un système réparti modélisée par trois types d'évènements :

- évènements d'émission (notés e)
- évènements de délivrance (notés d)
- évènements internes (notés i)

Nous notons e_C, d_C ou i_C un évènement ayant lieu sur le composant C.

Nous notons d^C un évènement de délivrance correspondant à la délivrance d'un message venant du composant C. L'évènement noté $d^C_{C'}$ est donc un évènement de délivrance ayant lieu sur C' et correspond à un message envoyé par C.

Les évènements internes correspondent aux pas de calculs et sont considérés de durée nulle. A chaque pas d'exécution d'un composant correspond un et un seul évènement interne. De plus, au sein d'un pas d'exécution, un composant lit tout d'abord tous les messages liés aux évènements de délivrance qui lui sont nécessaires, puis réalise son évènement interne et enfin provoque ses évènements d'émission. Le déclenchement des divers évènements d'émission sur chacun des ports de sortie peut être atomique ou étalé dans le temps. Nous considérons indifféremment des systèmes dynamiques (dont la configuration évolue dans le temps) ou statique.

Nous notons $\prec$ la relation de précédence temporelle entre évènements sur un composant.

3.4.1.A Relation d'influence directe

La relation d'influence directe $\rightarrow$ est définie ainsi :

- pour un message m, l'envoi influe sur la délivrance $e(m) \rightarrow d(m)$.
- un évènement interne influe sur les émissions immédiatement suivantes (jusqu'à l'évènement interne suivant) : si e et i sont sur le même composant, que e est postérieur à i et qu'il n'existe pas de i' entre i et e alors $i \rightarrow e$:

$$i_C \rightarrow e_C \text{ ssi } \forall e_C, i_C : i_C \prec e_C \wedge \nexists i'_C : i_C \prec i'_C \prec e_C$$

- la dernière délivrance venant d'un composant influe sur les évènements internes qui suivent, jusqu'à la prochaine délivrance venant du même composant : si $d^{C'}$ et i sont sur le même composant, que i est postérieur à $d^{C'}$ et qu'il n'existe pas de $D^{C'}$ entre $d^{C'}$ et i, alors $d^{C'} \rightarrow i$:

$$d^{C'}_C \rightarrow i_C \text{ ssi } \forall d^{C'}_C, i_C : d^{C'}_C \prec i_C \wedge \nexists D^{C'}_C : d^{C'}_C \prec D^{C'}_C \prec i_C$$

3.4.1.B Relation d'influence

La relation d'influence, notée $\rightarrow^*$, est construite par clôture transitive de $\rightarrow$:

$$a \rightarrow^* b \triangleq a \rightarrow b \vee \exists c : a \rightarrow^* c \wedge c \rightarrow^* b$$

Notons que $i \rightarrow^* i'$ si et seulement si il existe une séquence respectant le motif suivant : $i \rightarrow s_1 \rightarrow d_1 \rightarrow i_2 \rightarrow s_2 \rightarrow d_2 \rightarrow i_3 \cdots \rightarrow i'$.

Comparaison avec la relation de causalité La relation d'influence est plus forte que la relation de causalité : si a influence b alors a précède causalement b. La réciproque n'est pas nécessairement vraie. En particulier, la séquentialité des évènements (de délivrance ou d'émission) sur un composant n'est pas une influence car seuls les pas de calculs créent une dépendance. Seul le dernier message reçu depuis un destinataire donné compte.

Notons également qu'un évènement interne en influence un autre uniquement s'il existe un échange de message entre eux. Deux évènements internes consécutifs se déroulant sur le même composant ne s'influencent pas directement car nous nous intéressons à la circulation de l'information entre les composants, et non pas au sein d'un composant. Cela signifie qu'un évènement interne est directement influencé par les derniers évènements délivrés par d'autres composants mais pas par le dernier évènement interne.

Les évènements de délivrance sont conservés jusqu'à ce qu'un évènement plus récent soit utilisé. Nous voyons ici que le relation d'influence est plus proche d'une description d'un système distribué par un modèle mémoire que par un modèle message.

3.4.1.C Passé d'influence

On définit le passé d'influence d'un évènement i comme l'ensemble des évènements internes qui influent sur i :

$$\text{passé}(i) \triangleq \{i' \mid i' \rightarrow i\}$$

3.4.1.D Exécution cohérente

D'après la définition 3, si un composant C' utilise plusieurs données dont les valeurs dépendent de sorties d'un même composant C, alors ces données d'entrée de C' seront considérées comme strictement cohérentes si elles dépendent de données produites par le même pas de calcul de C.

En terme de relation d'influence, les données dont dépend une autre donnée d se traduisent par le passé d'influence de l'évènement interne qui a produit d. Nous souhaitons donc que ce passé d'influence ne comporte pas des données provenant de différents pas d'un même composant. Dans notre modèle, les pas sont modélisés par des évènements internes.

Nous notons $S|C$ l'ensemble des évènement de l'ensemble S se déroulant sur le composant C.

Définition 16 *Une exécution est cohérente si tout évènement interne ne contient dans son passé d'influence qu'au plus un évènement interne par composant :*

$$\forall i : \forall C : card(passé(i)|C\}) \leq 1$$

Notons que cette propriété peut être utilisée sur des graphes de composants dynamiques. En effet, nous comparons les évènements internes présents dans les passés d'influence sans avoir besoin de connaître l'architecture du système. Elle peut donc évoluer en cours d'exécution.

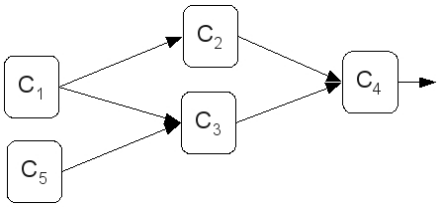


FIGURE 3.8 – Graphe statique des composants

3.4.1.E Formalisation de la cohérence relâchée

La notion de cohérence relâchée définie dans le paragraphe 3.1.3 peut également se formaliser par la relation d'influence.

Nous utilisons une datation temps réel pour dater les évènements se déroulant durant l'exécution du système. Nous notons $date(i)$ la date de l'évènement interne i.

Nous notons $ecart(E)$ l'écart maximal entre les évènements contenus dans E :

$$ecart(E) = \max_{i_1,i_2 \in E}(date(i_1) - date(i_2))$$

Définition 17 *Une exécution cohérente τ-relâchée est telle que :*

$$\forall i \ : \ \forall C \ : \ ecart(passé(i)|C) \leq \tau$$

Une exécution cohérente 0-relâchée est une exécution strictement cohérente. Notons que dans cette définition nous comparons des dates d'évènements se déroulant sur un même composant. Ainsi, une horloge globale n'est jamais requise.

Dans un système réel, il arrive fréquemment que nous ne souhaitions pas la même tolérance de cohérence sur tous les composants d'un système. La définition d'une exécution cohérente d'un système sera alors telle que chaque composant respecte sa tolérance fixée. Si nous notons $\tau_{C'}$ la tolérance de cohérence entre les entrées d'un composant C', nous avons alors la définition suivante d'une exécution cohérente τ-relâchée :

$$\forall C' : \forall i_{C'} \ : \ \forall C \ : \ ecart(\text{passé}(i_{C'})|C) \leq \tau_{C'}$$

Notons que la gestion de la cohérence τ-relâchée peut être encore plus précise. Nous pourrions préciser la tolérance non pas pour un composant mais entre chaque couple d'entrée de ce composant.

3.4.1.F Extension de la notion d'influence : influence entre données

Dans le paragraphe 3.4.1.B, nous avons défini la relation d'influence entre évènements. Pour les chapitres suivants, nous aurons besoin de transposer cette définition dans un modèle orienté données.

Définition 18 (Influence entre données) *Une donnée d produite par un pas d'exécution S influence une donnée d' produite par un pas d'exécution S' si l'évènement interne modélisant le pas S influence l'évènement interne modélisant le pas S'.*

En d'autres termes, d influence d' si d est directement utilisée pour calculer d' ou si une donnée dépendant de d est utilisée pour calculer d'. Voir la définition 1 de la dépendance du paragraphe 3.1.

Nous dirons également qu'un pas S influence une donnée d' si une donnée produite lors du pas S influence la donnée d'.

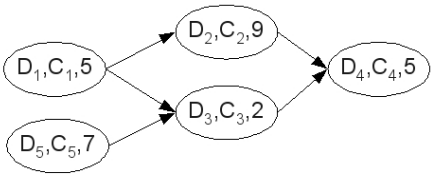


FIGURE 3.9 – Graphe de dépendance

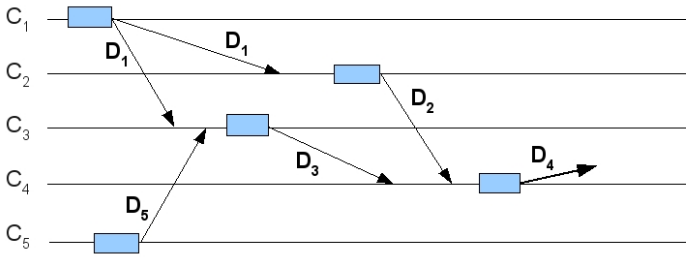


FIGURE 3.10 – Exécution correspondant au graphe 3.9

3.4.2 Formalisation par analyse de graphes

Nous avons formalisé précédemment la notion d'exécution cohérente en nous appuyant sur la relation d'influence. Nous proposons dans ce qui suit une autre formalisation de cette notion en nous appuyant sur une analyse de graphes de dépendance.

3.4.2.A Graphe de dépendances

Nous modélisons les calculs au sein d'un système sous la forme d'un graphe de dépendance dont les nœuds sont les données produites par les composants. Les arcs relient deux nœuds D et D' par la relation suivante :

La donnée D «est utilisée pour le calcul de »D'

Nous supposons que l'exécution des composants peut être découpée en pas indépendants. Ainsi, nous attachons à toute donnée un attribut qui précise le composant et le pas d'exécution qui l'a produite. Un nœud du graphe est ainsi étiqueté par un triplet (D, C, H), dans lequel D est la valeur de la donnée, C le composant qui l'a produite et H l'identité locale au composant du pas d'exécution qui l'a produite. Par exemple, un arc relie un sommet étiqueté $(D_1, C_1, 5)$ à un sommet étiqueté $(D_2, C_2, 9)$ si et seulement si la donnée D_1 produite par le composant C_1 au pas numéro 5 a été utilisée par le composant C_2 pour calculer la donnée D_2 lors de son pas numéro 9.

Chaque graphe de dépendance est construit à partir d'une donnée qui est l'unique nœud final. Pour toutes les données du graphe, nous traçons toutes les données dont elles dépendent. Toutes les dépendances de toutes les données du graphe doivent être représentées.

Du point de vue de l'analyse de la cohérence des données, seuls les sous-graphes ayant pour nœud finaux les données produites par un composant puits sont intéressants. En effet, nous avons vu dans le paragraphe 3.3.2 que la cohérence n'est à vérifier que sur les composant puits.

Nous analysons le système de la figure 3.8. Le graphe 3.9 est un exemple de graphe de dépendance construit à partir d'une donnée produite par C_4. Le graphe

3.10 représente l'exécution qui a donné lieu à ce graphe. Nous voyons que pour calculer la donnée D_4 durant son pas numéro 5, C_4 a utilisé la donnée D_2 produite par C_2 durant son pas numéro 9 et la donnée D_3 produite par C_3 lors de son pas numéro 2. D_2 et D_3 sont calculées en utilisant la donnée D_1 produite par C_1 lors de son pas 5. En suivant les dépendances, nous voyons que D_4 dépend de D_1.

3.4.2.B Exécution cohérente

En analysant un graphe de dépendance, nous pouvons vérifier si les données utilisées pour le calcul d'une donnée constituent un ensemble cohérent. En effet, un nœud du graphe contient l'identité du composant et de son pas d'exécution qui a produit la donnée correspondante au nœud. Nous pouvons donc vérifier si la définition 4 est respectée. Cette définition peut être traduite ainsi :

Définition 19 (Exécution cohérente) *Dans un graphe de dépendance, les pas d'exécution qui produisent les données représentées dans le graphe sont cohérents, si et seulement si, pour tout ensemble de nœuds portant le même identifiant de composant, ces nœuds portent également le même identifiant de pas. L'exécution d'un composant est cohérente s'il ne réalise que des pas cohérents.*

Nous vérifions sur le graphe 3.9 que le pas d'exécution 5 de C_4 est cohérent. En effet, la donnée D_4 ne dépend pas de pas différents d'un même composant.

La figure 3.11 illustre également un pas d'exécution cohérent de C_4. A la différence de la figure 3.9, ici C_1 émet deux données lors de son pas d'exécution numéro 5. La donnée D_4 dépend donc de deux données différentes produites par C_1. Mais ces données provenant du même pas d'exécution, D_4 est bien produite à partir d'un ensemble de données cohérent.

Sur la figure 3.12, nous constatons que les deux données produites par C_1 ne proviennent pas du même pas d'exécution. L'exécution représentée est donc incohérente.

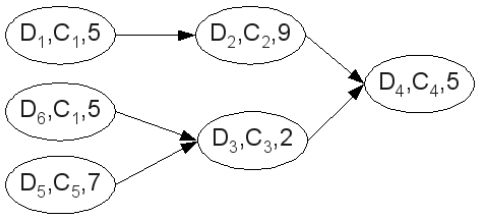


FIGURE 3.11 – Exécution cohérente avec deux sorties sur C_1

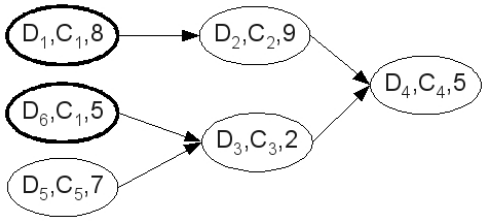


FIGURE 3.12 – Exécution incohérente

3.4.2.C Propriété d'isomorphisme de graphe

La notion de cohérence reposant sur les pas ayant produit les données et non sur les données elles-mêmes, nous définissons une relation d'équivalence entre nœuds du graphe afin de regrouper les données produites lors du même pas d'exécution d'un même composant. Nous définissons que deux nœuds (D_1, C_1, H_1) et (D_2, C_2, H_2) appartiennent à la même classe d'équivalence si $C_1 = C_2$ et $H_1 = H_2$.

Le graphe de dépendance est alors transformé en représentant les nœuds appartenant à une même classe d'équivalence par un sommet unique. Les arcs liés aux nœuds initiaux sont alors reliés au nouveau sommet. Ce sommet prend comme étiquette une des étiquettes des nœuds initiaux.

Dans un système statique (configuration n'évoluant pas au cours du temps), si le graphe transformé est isomorphe à un sous-graphe du graphe de composants du système, alors cela signifie que les données analysées ne dépendent pas de plus d'un pas d'exécution par composant du système. Les données du graphe de dépendance initial sont donc calculées à partir d'ensemble de données cohérent. Nous obtenons alors la propriété suivante :

Propriété 2 *Dans un système statique, si le graphe de dépendance des données transformé par la relation d'équivalence est isomorphe à un sous-graphe du graphe des composants, alors tous les pas d'exécution ayant produit les données du graphe sont cohérents.*

Nous voyons que le graphe 3.11 transformé par la relation d'équivalence est bien isomorphe au graphe de composants 3.8.

3.4.2.D Cohérence relâchée

Nous pouvons également vérifier que les données d'un graphe de dépendance respecte une cohérence relâchée. Cependant, nous réduisons la définition 5 en ne considérant pas un écart de temps entre les pas qui ont produit les données utilisées, mais un écart en nombre de pas. En effet, le graphe de dépendance ne trace que l'identité des pas qui ont produit les données et non leur date de début d'exécution. Le paramètre de cohérence τ exprime ici un nombre de pas. La tolérance τ peut être un paramètre propre à chaque composant. Nous notons alors τ_C la tolérance vis-à-vis du composant C.

Définition 20 (Pas d'exécution cohérent avec une tolérance τ) *Dans un graphe de dépendance, les pas d'exécution qui produisent les données représentées dans le graphe sont cohérents, si et seulement si pour tout ensemble de nœuds portant un même identifiant de composant C, l'écart maximal entre les identifiants de pas de ces nœuds est inférieur ou égal à τ_C.*

Notons que la propriété d'isomorphisme de graphe (propriété 2) est plus problématique à transposer dans le cadre d'une cohérence relâchée. En effet, dans ce cadre, nous sommes alors tenté de nous baser également sur les classes d'équivalence afin de contrôler la cohérence des exécutions. La relation serait alors que deux nœuds $N_1 = (D_1, C_1, H_1)$ et $N_2 = (D_2, C_2, H_2)$ appartiennent à la même classe d'équivalence si $C_1 = C_2$ et $H_1 \in [H_2 - tolerance..H_2 + tolerance]$.

Cependant, le problème qui se pose ici est qu'une relation d'équivalence est transitive. Cette approche n'est donc pas satisfaisante car nous considérons un n-uplet de données cohérent seulement si chaque couple de données de ce n-uplet respecte la condition de cohérence. Ainsi, si nous tolérons un écart d'un pas entre données

et si nous disposons des données $N_1 = (x, C, 1)$, $N_2 = (y, C, 2)$ et $N_3 = (z, C, 3)$, cet ensemble n'est pas cohérent car $H_3 > H_1 + 1$.

Or, avec une relation d'équivalence nous aurions : N_1 équivalent à N_2 et N_2 équivalent à N_3 donc N_1 équivalent à N_3. Tous ces nœuds appartiennent donc à la même classe d'équivalence.

Pour pouvoir utiliser la propriété d'isomorphisme de graphe dans le cadre d'une cohérence relâchée, il faudrait donc définir une relation spécifique entre sommets.

3.4.2.E Comparaison avec la relation d'influence

Dans le cadre de la cohérence stricte, la correspondance entre la définition par la relation d'influence et celle faite par l'analyse de graphe est assez immédiate.

La construction d'un graphe de dépendance dont le composant final est une donnée D produite par un puits est équivalente à la recherche du passé d'influence de l'évènement interne i influant sur l'émission de D. En effet, la présence d'un nœud dans le graphe de dépendance est équivalent à la présence d'un évènement de délivrance dans le passé d'influence de i car toute donnée utilisée par un composant est liée à un évènement de délivrance. Un évènement de délivrance est toujours influencé par un évènement interne. Des évènements de délivrance provenant du même pas de calcul sont donc influencés par un unique évènement interne.

Ainsi, dans le graphe de dépendance, si toutes les données provenant d'un même composant proviennent également du même pas d'exécution, cela est équivalent au fait qu'il n'existe qu'un seul évènement interne de ce composant dans le passé d'influence de i. Nous voyons alors que la définition 19 est équivalente à la définition 16.

3.5 Codage de la relation d'influence

Afin de contrôler la cohérence d'une exécution telle qu'elle est définie dans le paragraphe 3.4.1, il faut que chaque composant soit capable de connaître le passé d'influence de ses évènements internes. S'il a accès à ce passé, le composant peut alors vérifier si son exécution est cohérente. Une solution pour que chaque composant puisse connaître le passé d'influence d'un évènement est que l'évènement lui-même transporte cette information. Pour cela, nous introduisons la notion d'estampille attachée à chaque évènement.

3.5.1 Marquage

Définition 21 *Un marquage est un couple $\langle Composant, valeur \rangle$ où les valeurs sont choisies parmi n'importe quel ensemble infini.*

Cet ensemble n'a pas besoin d'être ordonnée mais nous utilisons un compteur croissant pour générer des valeurs distinctes successives.

Chaque composant possède une horloge logique $H(C)$ qui marque les évènements internes produits par le composant. Cette horloge "compte" le nombre de pas d'exécution réalisés par un composant. Ainsi, chaque marquage correspond à un évènement interne d'un composant. A chaque marquage, noté M_i, correspond exactement un évènement interne i et réciproquement.

3.5.2 Estampille

Définition 22 *Une estampille est un ensemble de marquages. Elle stocke le passé d'influence d'un évènement. L'estampille portée par l'évènement a est notée E_a.*

Chaque évènement de délivrance utilisé pour réaliser un évènement interne i_C est ajouté à l'ensemble $Input(i_C)$.

$$Input(i_c) = \{d_c^{c'} : (d_c^{c'} \prec i_c \wedge \nexists D_c^{c'} : d_c^{c'} \prec D_c^{c'} \prec i_c)\}$$

Règles d'estampillage

- Un évènement de délivrance a pour estampille l'estampille d'émission correspondante.
- Un évènement d'émission a pour estampille l'estampille du plus récent évènement interne qui le précède.
- Un évènement interne i d'un composant C a pour estampille l'ensemble des estampilles des évènements de délivrance qu'il utilise à ce pas de calcul auquel il ajoute son propre marquage :

$$E_i = \bigcup_{d \in Input(i)} E_d \cup \{\langle C, H(C)\rangle\}$$

et $H(C)$ est incrémentée.

Lemme 1 (Estampilles et marquages)

$$E_i = \{M_i\} \cup \{M_j \mid j \rightarrow^* i\}$$

Preuve Nous notons :

$$\begin{array}{lcl} i \overset{1}{\mapsto} i' & \triangleq & \exists s, d : i \rightarrow s \rightarrow d \rightarrow i' \\ i \overset{n}{\mapsto} i' & \triangleq & \exists i'' : i \overset{1}{\mapsto} i'' \wedge i'' \overset{n-1}{\mapsto} i' \end{array}$$

Par les règles d'estampillages :

$$\begin{array}{rcl} E_i & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} E_j \\ & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \left(\{M_j\} \cup \bigcup_{k \mid k \overset{1}{\mapsto} j} E_k\right) \\ & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \{M_j\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \bigcup_{k \mid k \overset{1}{\mapsto} j} E_k \\ & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \{M_j\} \cup \bigcup_{k \mid k \overset{2}{\mapsto} i} E_k \\ & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \{M_j\} \cup \bigcup_{k \mid k \overset{2}{\mapsto} i} \left(\{M_k\} \cup \bigcup_{l \mid l \overset{1}{\mapsto} k} E_l\right) \\ & = & \{M_i\} \cup \bigcup_{j \mid j \overset{1}{\mapsto} i} \{M_j\} \cup \bigcup_{k \mid k \overset{2}{\mapsto} i} \{M_k\} \cup \bigcup_{l \mid l \overset{3}{\mapsto} i} E_l \\ & = & \{M_i\} \cup \bigcup_{1 \leq y \leq n} \bigcup_{j \mid j \overset{y}{\mapsto} i} \{M_j\} \cup \bigcup_{j \mid j \overset{n+1}{\mapsto} i} E_j \end{array}$$

Toutes les chaînes $\mapsto$ sont bornées (évènement initial)

$$\begin{array}{rcl} & = & \{M_i\} \cup \bigcup_{y \geq 1} \bigcup_{j \mid j \overset{y}{\mapsto} i} \{M_j\} \\ & = & \{M_i\} \cup \{M_j \mid j \rightarrow^* i\} \end{array}$$

□

Théorème 1 *L'estampillage est un codage de l'influence :*

$$i \rightarrow i' \Leftrightarrow E_i \subset E_{i'}$$

Preuve L'implication directe est déduite des règles d'estampillage et de la transitivité de $\rightarrow^*$:

$$\begin{gathered}\text{Si } i \rightarrow s \rightarrow d \rightarrow i', \text{ alors par les règles d'estampillage :} \\ \Rightarrow E_{i'} = \{M_{i'}\} \cup E_i \cup X \\ M_{i'} \text{ est unique et provient seulement de } i'. \\ \text{Comme } i' \not\rightarrow^* i \wedge i \neq i', M_{i'} \notin E_i \\ \Rightarrow E_i \subsetneq E_{i'}\end{gathered}$$

$$\begin{gathered}\text{En utilisant la transitivité de } \rightarrow^* : \\ i \rightarrow^* i' \Leftrightarrow i \rightarrow ... \rightarrow i' \Rightarrow E_i \subsetneq E_{i'}\end{gathered}$$

L'implication inverse provient des règles d'estampillage et du lemme 1 :

$$\begin{array}{ll} E_i \subsetneq E_{i'} & \Rightarrow M_i \subsetneq E_{i'} \text{ (lemme 1)} \\ & \Leftrightarrow M_i \subsetneq \{M_{i'}\} \cup \{M_j \mid j \rightarrow^* i'\} \text{ (lemme 1)} \\ & \text{Un marquage est unique et } i \neq i' \Rightarrow M_i \neq M_{i'} \\ & \Leftrightarrow M_i \subsetneq \{M_j \mid j \rightarrow^* i'\} \\ & \Rightarrow \exists j : M_i = M_j \wedge j \rightarrow^* i' \\ & \text{Un marquage est unique :} \\ & \Leftrightarrow i \rightarrow^* i' \end{array}$$

. □

Théorème 2 *Nous notons $E_a|C$, l'ensemble des marquages contenu dans E_a qui sont générés par un composant C. Une exécution est cohérente si et seulement si il n'existe pas plusieurs marquages provenant d'un même composant dans l'estampille de tout évènement interne.*

$$\textit{Exécution cohérente} \triangleq \forall i : \forall C : card(E_i|C) \leq 1$$

Preuve

$$\begin{array}{ll} E_i & = \{M_{i'} \mid i' \rightarrow^* i\} \cup \{M_i\} \\ & = \{M_{i'} \mid i' \rightarrow^* i \vee i' = i\} \\ & = \{M_{i'} \mid i' \in past(i)\} \end{array}$$

Deux évènements distincts ne peuvent pas générer le même marquage, donc le nombre de marquage et le nombre d'évènements sont égaux :

$$\begin{gathered}card(\{M_{i'} \mid i' \in past(i)\}|c) = card(past(i)|c) \\ \text{(avec une restriction sur } C \text{ ou non), et donc} \\ card(E_i|c) = card(past(i)|c)\end{gathered}$$

Cette condition est équivalente à la condition de cohérence définie au paragraphe 3.4.1.

□

La figure 3.13 retrace l'exécution incohérente de la figure 3.5. Nous suivons les évènements se déroulant sur chaque composant du système avec leurs estampilles correspondantes. Nous constatons que l'estampille du dernier évènement interne de C_3 contient deux marquages différents de C_0. Nous détectons ainsi que les messages utilisés par cet évènement interne sont influencés par deux pas d'exécution différents de C_0. L'exécution est donc incohérente.

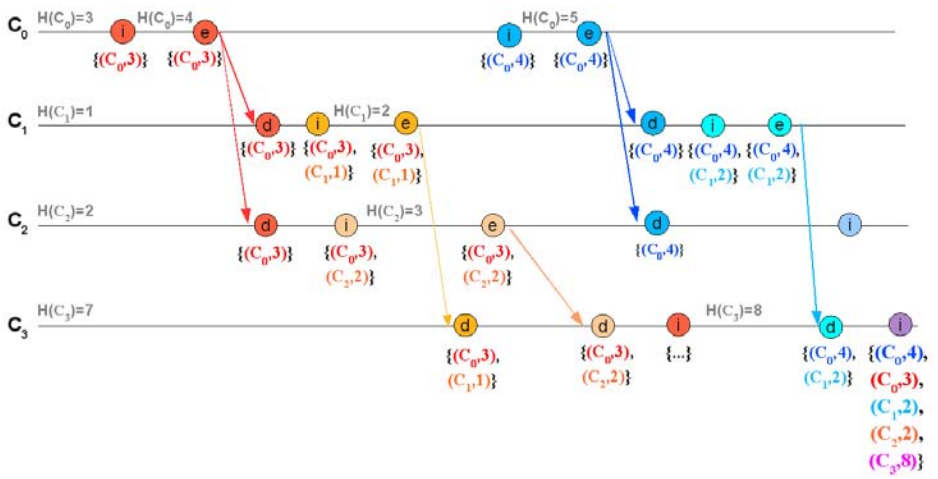


FIGURE 3.13 – Construction d'estampilles : détection d'exécution incohérente

Comparaison avec d'autres types d'encodage Notre travail est dans le même esprit que les travaux classiques de Lamport [43] et Mattern [49] encodant la relation de causalité dans un système distribué. Cependant, notre relation d'influence étant différente de la relation de causalité habituelle, nous utilisons un codage différent. L'horloge locale ne se comporte pas comme une horloge de Lamport (l'horloge n'est pas mise à jour en utilisant les estampilles des messages) et les estampilles associées aux messages ne sont pas des vecteurs d'horloge de Fidge-Mattern. Nous utilisons un ensemble, non pas pour optimiser le codage d'un vecteur creux, mais parce que nous pouvons avoir plus d'un marquage provenant d'un même composant. Cependant, notons que lors d'une exécution cohérente, il n'y aura jamais plus d'un marquage par composant et un vecteur pourrait alors être utilisé à la place d'un ensemble.

3.5.3 Composants marqueurs et contrôleurs

Afin de réduire le nombre de marquages utilisés dans le système, nous identifions les composants qui doivent produire des marquages. Nous identifions également les composants sur lesquels un contrôle de cohérence devra être réalisé. Nous appelons de tels composants, des contrôleurs.

Nous avons vu dans le paragraphe 3.3.2, qu'un problème d'association de données peut intervenir entre deux composants si et seulement si il existe un fuseau entre eux. Ainsi, un contrôle de cohérence de données doit être réalisé sur chaque puits par rapport aux marquages de la source du fuseau. Nous pouvons donc réduire le nombre de marquage en ne produisant des marquages que pour les sources de fuseau.

Propriété 3 *Chaque source de fuseau est un composant marqueur et chaque puits est un composant contrôleur.*

3.5.4 Élimination des marquages inutiles

Nous pouvons également optimiser la propagation des marquages lorsqu'un marquage n'est plus utile. En effet, il n'est pas nécessaire de propager le marquage d'un composant C dans les estampilles des données si aucun contrôle de cohérence sur ce marquage n'est effectué au delà. Le marquage de C n'est utile que pour les puits de fuseaux de source C. Il est donc possible que le puits de ce fuseau ne propage pas

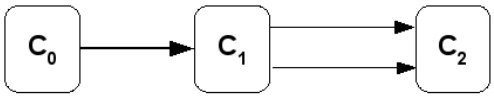


FIGURE 3.14 – Exemple de système

les marquages de C dans l'estampille de ses sorties. Attention toutefois à la présence de fuseaux imbriqués pouvant impliquer qu'un composant soit source de plusieurs fuseaux : dans ce cas, le marquage ne doit être supprimé qu'après le dernier puits traversé par des données correspondant à cette source. Il s'agit des puits des fuseaux principaux de source C.

3.5.5 Variation du modèle d'exécution choisi

Nous avons imposé dans le modèle d'exécution choisi que lors d'un pas d'exécution, l'enchaînement suivant est respecté : évènements de délivrance, puis évènement interne et enfin évènements d'émission. De plus nous imposons qu'il n'y ait qu'un seul évènement interne par pas d'exécution.

Nous pouvons cependant assouplir le modèle en permettant l'entrelacement de ces évènements. Sur un composant ayant deux ports d'entrée (A et B) et deux de sortie (C et D), nous pouvons avoir la séquence suivante : délivrance sur A, évènement interne, émission sur C, délivrance sur B, évènement interne, émission sur D.

Mais ce type de séquence ne serait pas compatible avec les règles d'estampillages que nous avons choisies. En effet, l'horloge d'un composant est incrémentée à l'occasion d'un évènement interne. Ainsi si plusieurs évènements internes se déroule au sein du même pas d'exécution d'un composant C, ils ont alors des marquages différents vis-à-vis de C au sein de leurs estampilles. Les évènements d'émission qui les suivent ont alors également des marquages différents sur C. Nous obtenons ainsi des évènement d'émission provenant du même pas d'exécution de C avec des marquages différents sur C. Nous ne pouvons donc pas contrôler la cohérence grâce aux estampilles puisqu'elles ne permettent pas alors de vérifier que des évènements dépendent du même pas de calcul d'un composant.

Cependant, grâce à une différence d'implémentation mineure, nous pouvons imaginer de tels entrelacements d'évènement. La gestion de l'horloge doit alors être modifiée en n'incrémentant pas l'horloge d'un composant après chaque évènement interne mais par exemple au début du pas du composant ou du moins avant le premier évènement interne. En utilisant ce principe, les estampilles d'évènement d'émission provenant d'un même pas d'exécution d'un composant C portent bien le même marquage sur C. La cohérence par rapport aux pas d'exécution de C peut donc être vérifiée au moyen des estampilles.

Par contre, la tolérance de l'entrelacement permet que les évènement d'émission d'un même pas n'aient pas le même passé d'influence. Ainsi nous ne pouvons plus utiliser le théorème 2.

Nous imaginons un entrelacement possible sur le système de la figure 3.14. L'exécution et la propagation des estampilles est illustrée par la figure 3.15. Nous supposons ici que les composants sont périodiques et nous appliquons le principe de l'incrémentation de l'horloge de chaque composant au début de leur période. Nous constatons sur C_2 que les estampilles des évènements de délivrance portent le même marquage sur C_1 mais pas sur C_0. Si ce type de motif d'exécution se répète, C_2 ne pourra jamais faire le choix d'un ensemble cohérent tel que défini par le théorème 2. Il n'aura jamais un ensemble d'évènement de délivrance tel qu'il n'existe qu'un seul marquage

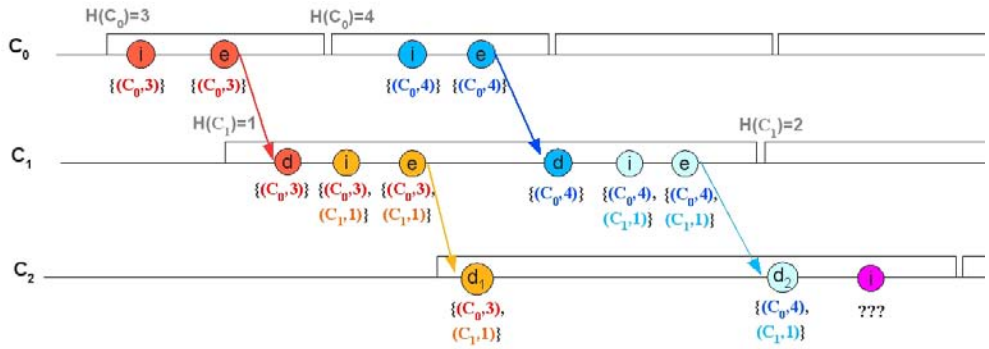


FIGURE 3.15 – Influence de l'entrelacement entre évènements du système 3.14

par composant au sein de cet ensemble car deux données avec le même marquage de C_1 ont toujours un marquage différent de C_0 et vice-versa.

Nous constatons ici que la tolérance de l'entrelacement entre types d'évènements impose, en plus d'une gestion spécifique des horloges, une gestion spécifique de la cohérence. Une solution pour gérer le problème décrit est que chaque composant ait une liste de composants par rapport auxquels il doit assurer une cohérence. Cet ensemble s'obtient facilement : pour chaque composant puits d'un ou plusieurs fuseaux, il s'agit de sa ou ses sources, pour les composants non puits, l'ensemble est vide. Ici, C_2 est uniquement puits d'un fuseau de source C_1, il ne doit donc se préoccuper que des marquages de C_1. Le théorème 2 devient alors :

Théorème 3 *Nous notons $E_a|C$, l'ensemble des marquages contenu dans E_a qui sont générés par un composant C. Nous notons, $V_{C'}$, l'ensemble des composants par rapport auxquels le composant C' doit vérifier la cohérence des données. Une exécution est cohérente si et seulement si, pour chaque composant C', il n'existe pas plusieurs marquages provenant d'un même composant appartenant à $V_{C'}$ dans l'estampille de tout évènement interne.*

$$\textit{Exécution cohérente} \triangleq \forall i : \forall C' : \forall C \in V_{C'} : card(E_{i_{C'}}|C) \leq 1$$

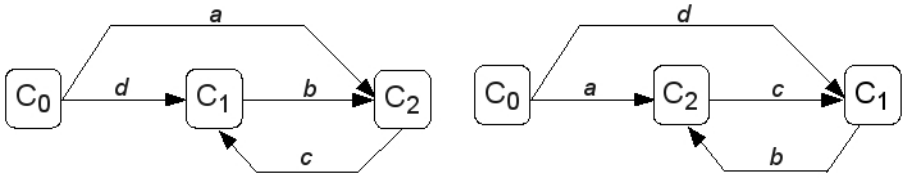
3.6 Systèmes comportant des boucles

Dans la suite de ce mémoire, nous étudierons uniquement des systèmes ne contenant pas de boucle de rétroaction, c'est à dire des graphes ne contenant pas de circuit. Nous allons cependant évoquer cette problématique et la façon dont nous proposons de la gérer dans ce qui suit.

3.6.1 Caractéristiques des boucles

Définition 23 (Circuit) *Un circuit est un chemin dont les extrémités coïncident.*

En terme de composants logiciels, l'existence d'un circuit comportant un composant C signifie que, pour calculer sa nouvelle sortie, le composant C utilise une donnée dépendant d'une donnée produite par ce même composant C dans le passé. Sur la figure 3.16(a), il existe un circuit entre C_1 et C_2. La donnée b produite par C_1 dépend des données d et c. La donnée c est produite par C_2 et dépend de b, elle-même produite par C_1. La donnée c dépend donc indirectement de C_1 tout en étant utilisée par ce dernier.



(a) Graphe1 (b) Graphe2 (équivalent au graphe1)

FIGURE 3.16 – Graphe avec circuit

En terme de gestion de la cohérence des données, une problématique apparaît. Si tous les composants utilisent le mécanisme d'estampillage des données présenté en 3.5, le composant C_1 reçoit, via la donnée c, une estampille contenant au moins un de ses propres marquages. En effet, la donnée c dépend d'une donnée émise par C_1. L'estampille de l'évènement interne de C_1 contient alors au moins deux de ses propres marquages : le nouveau marquage correspondant à l'exécution courante et l'ancien porté par la donnée c utilisée en entrée. D'après le théorème 2, l'exécution est donc considérée comme incohérente puisqu'il existe plusieurs marquages provenant d'un même composant dans l'estampille de l'évènement interne. Au cours de l'exécution du système, les multiples passages dans la boucle vont créer une accumulation de marquages de tous les composants membres de la boucle. Ainsi, l'exécution de tous les composants va être considérée comme incohérente.

Ce phénomène est observé car la présence d'une boucle ramène des marquages du passé. Par définition, une boucle possède cette caractéristique, ce comportement ne doit donc pas être considéré comme incohérent. Les boucles doivent être traitées de façon particulière. Il est alors tout d'abord nécessaire de spécifier leur sémantique.

3.6.2 Identification des arcs de retour de boucle

Lorsqu'une boucle est présente dans un système, il est nécessaire d'attacher une sémantique précise au fonctionnement de cette boucle. Le "début" de la boucle doit être identifié. En d'autres termes, il s'agit d'identifier qui est le producteur de la première valeur. Sur la figure 3.16(a), d'après nos habitudes graphiques, nous considérons généralement que C_1 produit le premier une valeur b en ignorant l'entrée de c ou en ayant une valeur par défaut pour celle-ci. Ensuite, en fonction de la valeur de b qu'il reçoit, C_2 produit c. C_1 pourra alors produire une nouvelle donnée b en fonction de cette donnée c et la boucle sera enclenchée. Cependant, ce fonctionnement correspond uniquement à une intuition. La figure 3.16(b) reprend le même système que la figure 3.16(a). Seule la disposition graphique des composants a été modifiée. Nous voyons alors que le fonctionnement du système peut être inverse. La donnée b peut avoir une valeur par défaut et C_2 peut être le premier composant à produire une donnée c qui sera utilisé par le composant C_1 ensuite. Il est donc important que le concepteur précise quel est le comportement qu'il attend de son système.

Nous proposons une notation graphique afin d'identifier quel arc représente le retour d'une boucle. Un "R" est apposé au niveau de la pointe de l'arc retour de la boucle comme sur la figure 3.17. Le composant recevant cet arc est donc l'initiateur de la boucle. Tout circuit du graphe doit comprendre un arc identifié de la sorte. Si, dans un graphe, il existe un circuit sans aucun arc marqué d'un identifiant de retour, le graphe est incorrect. Le concepteur doit préciser le comportement qu'il attend du système.

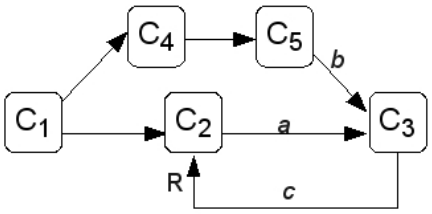


FIGURE 3.17 – Graphe avec circuit et identifiant de retour de boucle

3.6.3 Analyse des fuseaux

En analysant la figure 3.17, deux fuseaux apparaissent : un fuseau A de source C_1 et de puits C_3 comprenant les deux chemins séparés (C_1, C_4, C_5, C_3) et (C_1, C_2, C_3) ainsi qu'un fuseau B de C_1 à C_2 comprenant les chemins séparés (C_1, C_2) et $(C_1, C_4, C_5, C_3, C_2)$. Cependant, le fuseau B a la spécificité de comporter un arc marqué comme retour de boucle. Le concepteur ne souhaite donc pas que les marquages de C_1 arrivant à C_2 par les deux chemins séparés du fuseau aient obligatoirement la même valeur. En effet, le retour de boucle signifie précisément que les estampilles portées par cet arc sont des estampilles du "passé" par rapport à celles provenant d'un chemin sans retour. Le respect de la cohérence stricte n'a donc aucun sens. Un fuseau comprenant au moins un arc de retour de boucle n'implique donc pas les conditions de cohérence que provoque un fuseau sans un tel arc. Il ne doit pas être considéré comme générateur de contraintes. En conséquence, lors de la recherche graphique de fuseaux, les arcs de retour de boucle sont ignorés.

3.6.4 Gestion de la cohérence

Avant toute analyse, un nouveau graphe est construit en supprimant tous les arcs de retour de boucle. Un graphe sans circuit est alors obtenu et l'analyse de graphe du paragraphe 3.3 peut lui être appliqué. Nous identifions ainsi les sources et puits du système nous permettant de limiter les marquages utilisés.

Lors de l'exécution du système, nous avons affaire au problème de gestion d'estampilles décrits dans 3.6.1. Si un composant C reçoit une estampille comportant un marquage qu'il a lui même créé, que se passe-t-il ?

Si l'optimisation des marquages proposé dans la section 3.5.3 est appliquée, le fait d'avoir recherché les sources du système en ignorant les arcs de retour de boucle peut supprimer certains marquages parasites. Mais rien n'empêche un composant initiateur de boucle d'être également source de fuseau et donc de générer un marquage.

Ce problème d'estampillage n'est pas uniquement porté par le marquage du composant initiateur de la boucle. Dans le graphe 3.17, l'estampille de l'évènement interne de C_2 peut comporter plusieurs marquages de C_2 (si pas d'optimisation des marquages) mais également plusieurs marquages de C_1 qu'il reçoit par le retour de boucle et en entrée directe.

Pour résoudre ce problème, il est donc nécessaire, tout comme lors de l'analyse de graphe, d'ignorer les estampilles d'un arc de retour de boucle. Les composants doivent donc avoir un comportement particulier : l'estampille de tout évènement interne est créée en ignorant l'estampille portée par un retour de boucle. En adoptant ce principe, tous les modes de gestion de la cohérence présentés dans ce mémoire peuvent être appliqués.

Le graphe initial avec circuit peut tout de même être utilisé afin de fournir des analyses temporelles sur le comportement des arcs de retours de boucle, par exemple

pour connaître le temps minimal et maximal entre l'émission d'une donnée par un composant et la réception par ce même composant d'une donnée qui en dépend.

3.6.5 Extension du principe des systèmes avec boucles : autre arc ignoré

Les arcs de retour de boucle sont ignorés pour la gestion de la cohérence car un tel arc porte par définition une estampille du passé. Seul le concepteur peut signaler quels sont les arcs considérés comme des retours de boucle afin qu'ils soient traités de façon particulière. Sur le même principe, le concepteur peut également signaler d'autres arcs spécifiques. Il peut s'agir d'un arc appartenant à un chemin sur lequel le concepteur ne souhaite pas garantir la cohérence. De la même manière que les arcs de retour de boucle, cet arc particulier est ignoré.

3.7 Approche proposée

L'étude présentée dans cette thèse peut être menée dans deux optiques :
- Analyse d'un système du point de vue de la problématique de l'association des données
- Mise en place de méthodes de gestion de la cohérence dans un système donné

Du point de vue de l'analyse, nous avons jusqu'ici proposé une méthode d'analyse de l'architecture des systèmes afin d'identifier les configurations pouvant poser des problèmes d'association de données. Il s'agit de détecter la présence de fuseaux dans un système et d'identifier leurs sources et leurs puits.

Dans les deux chapitres suivants, nous allons nous intéresser plus particulièrement à deux types de systèmes : des systèmes périodiques où les composants ont tous la même fréquence et des systèmes périodiques avec des composants de fréquences diverses. Nous allons calculer certaines propriétés temporelles des données suivant le type de système étudié et ses paramètres. Ces propriétés vont nous permettre de quantifier l'impact des fuseaux sur la cohérence des données, nous permettant ainsi d'approfondir l'analyse du système.

D'autre part, nous proposerons également des méthodes afin de gérer en pratique la cohérence des données suivant le type de système. Dans certains cas, nous aurons alors besoin du codage par estampillage présenté précédemment. Ce codage prendra alors part à la gestion opérationnelle de l'association des données.

La méthodologie à suivre pour toute étude de système sera de débuter par une phase d'analyse. Ensuite, suivant les résultats de cette analyse et les besoins du concepteur du système, nous choisirons la méthode à appliquer pour gérer l'association des données. Il pourra alors s'agir d'utiliser une des méthodes que nous allons présenter dans ce qui suit. Mais il peut également être possible de ne pas mettre en place de méthodes particulières car le système ne le nécessite pas ou, au contraire, d'être obligé de modifier l'architecture du système car les méthodes proposées ne pourront pas être mises en place, pour cause de ressources insuffisantes par exemple.

Chapitre 4

Systèmes monopériodiques

4.1 Modèle des systèmes monopériodiques

Nous étudions dans ce chapitre des systèmes particuliers qui, nous le verrons, permettent de simplifier la gestion de la cohérence des données.

Nous étudions des systèmes synchrones, c'est-à-dire des systèmes où tous les composants partagent la même horloge. De plus, tous les composants ont la même période et leurs cycles ne sont pas déphasés. Tous les composants débutent leurs périodes simultanément

A l'intérieur d'une période, nous ne faisons pas d'hypothèse sur la durée des pas de calcul des composants et nous ne connaissons pas leur ordonnancement. Nous imposons cependant que tous les composants aient terminé leur pas de calcul à la fin de leur période.

Chaque composant peut avoir plusieurs entrées et plusieurs sorties. Toutes les entrées d'un composant sont lues simultanément au début de son pas de calcul, ses données de sorties sont émises simultanément à la fin de son pas de calcul.

Il n'y a pas de communication entre composants au cours d'une période. Un composant ne peut disposer d'une donnée que si cette donnée à été produite par un autre composant au plus tard à la période précédente. Nous supposons que la transmission d'une données entre deux cycles est toujours possible, c'est-à-dire que nous ne nous préoccupons pas de temps de communication entre composants. Ces règles sont comparables à celles du formalisme Giotto [31].

Nous supposons que les communications entre composants sont FIFO et fiables.

4.2 Gestion de la cohérence

Nous avons vu dans le paragraphe 3.1, définition 3 que si un composant C' utilise plusieurs données dont les valeurs dépendent de sorties d'un même composant C, alors les données d'entrée de C' seront considérées comme strictement cohérentes si elles dépendent de données produites par le même pas de calcul de C.

La propriété 16 du paragraphe 3.4.1 formalise cette définition en précisant qu'une exécution est cohérente si tout évènement interne ne contient dans son passé d'influence qu'au plus un évènement interne par composant.

Dans un fuseau de source C et de puits C', il s'agit de contrôler les données utilisées par C' afin qu'elles soient cohérentes par rapport à C. Afin de trouver une méthode pour contrôler ces données, nous analysons tout d'abord les propriétés du type de système étudié dans ce chapitre.

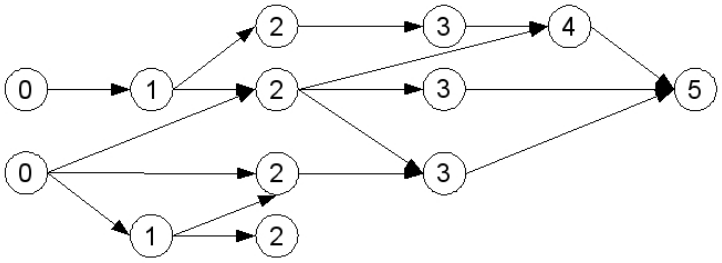


FIGURE 4.1 – Rangs des composants

4.2.1 Propriétés du système

4.2.1.A Disponibilité d'une donnée

D'après le modèle de systèmes que nous avons défini dans le paragraphe 4.1, nous voyons que si un composant C_1 communique directement avec un composant C_2, alors une donnée produite par C_1 à la période T, est utilisée au plus tôt par C_2 à la période $T+1$.

Soit un chemin de composant $P = (C_1, C_2, ..., C_n)$ de longueur n. Si C_1 produit une donnée d à la période T, C_n pourra disposer d'une donnée influencée par d à la période $T+n-1$.

4.2.1.B Rang d'un composant

Nous analysons les systèmes en tant que graphe orienté. Dans ce cadre, nous pouvons nous appuyer sur des notions de théorie des graphes pour les analyser. Nous rappelons donc les notions que nous allons utiliser.

Définition 24 (Origine d'un graphe) *Une origine est un nœud sans prédécesseur.*

Définition 25 (Rang d'un composant) *Dans un graphe sans circuit, le rang d'un nœud C est la distance maximale de ce nœud à une origine du graphe. C'est la longueur du chemin simple le plus long d'extrémité finale C.*

La longueur d'un chemin étant le nombre d'arcs utilisés par ce chemin, elle est égale au nombre de nœuds du chemin moins un. Il n'existe pas de chemin allant d'un sommet à un autre de même rang ou de rang inférieur.

Pour déterminer les rangs des composants nous appliquons l'algorithme suivant :
- Les origines du graphe sont de rang zéro. Dans le cadre de l'analyse d'un fuseau seul, seule la source est de rang zéro.
- On attribue le rang r à un nœud si tous ses prédécesseurs possèdent un rang et si le maximum des rangs de ses prédécesseurs est de rang $r-1$
- L'algorithme se termine lorsqu'un rang a été attribué à tous les nœuds

Nous avons appliqué cet algorithme sur le graphe de la figure 4.1. Les rangs des nœuds sont indiqués à l'intérieur de chacun d'eux.

Nous appliquons également l'algorithme sur le graphe de la figure 4.2 :
- C_1 est de rang 0.
- C_2 et C_4 sont de rang 1.
- C_5 est de rang 2.
- C_3 est de rang 3.

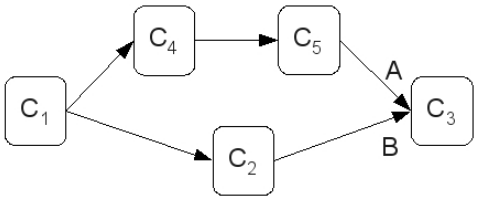


FIGURE 4.2 – Exemple de système monopériodique

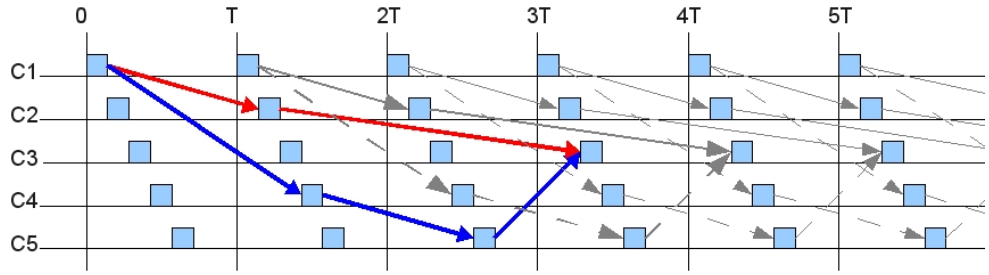


FIGURE 4.3 – Comportement cohérent du système 4.2

4.2.2 Contrôle de la cohérence dans un fuseau simple

4.2.2.A Exemple applicatif

Nous nous focalisons tout d'abord sur l'étude d'un système composé d'un seul fuseau. Nous étudions l'exemple de la figure 4.2 représentant un fuseau entre C_1 et C_3. Nous voyons que sans contrôle, si C_1 émet une donnée d à la période T, C_3 recevra une donnée influencée par d :

- à la période $T+2$ sur son port B
- à la période $T+3$ sur son port A

Ou si nous considérons les actions sous un autre angle, quand C_3 lit ses données d'entrée à la période T', elles sont influencées par des pas de C_1 démarrés à :

- à la période $T'-2$ sur son port B
- à la période $T'-3$ sur son port A

Ainsi, C_3 dispose de données influencées par des pas différents de C_1. Il ne disposera donc jamais de données cohérentes sur ces entrées. Nous constatons alors que, pour fournir des données cohérentes à C_3, il faut décaler d'une période la délivrance des données arrivant sur le port B. Pour cela, nous pouvons utiliser une file d'attente sur le port B. Cette file servira de tampon afin de décaler la délivrance des données sur B. Dans cet exemple, il s'agit pour C_3 de lire à chacun de ses pas, non pas la dernière donnée reçue sur B mais l'avant dernière. Ainsi, à la période T', C_3 ne lit pas des données influencées par des pas de C_1 exécutés lors de la période $T'-2$ mais exécutés à la période $T'-3$.

Avec ce principe, C_3 accède donc pour chaque période T' à des données influencées par des pas de C_1 s'étant déroulés lors de la période $T'-3$ sur chacun de ses deux ports. Le puits C_3 dispose donc de données cohérentes car influencées par le même pas d'exécution de la source C_1.

La figure 4.3 illustre le comportement que nous voulons que le système suive. Sur ce graphe, la queue d'une flèche représente le moment où une donnée est produite, sa tête le moment où elle est utilisée. Nous voyons que la donnée produite par C_2 lors de la deuxième période est utilisée par C_3 lors de la quatrième période. Si nous suivons

les influences entre données nous remarquons que les donnée utilisées par C_3 via les ports A et B lors de cette quatrième période sont bien influencées par le même pas d'exécution de C_1.

En résumé, dans ce cas, la solution repose sur un recalage des données fixé par le chemin le plus long.

4.2.2.B Cas général d'un fuseau simple

De façon générale, si nous analysons un fuseau entre C_α et C_β composé de plusieurs chemins, nous constatons que, pour fournir des données cohérentes au puits C_β, il faut que les délivrances de certaines données soient décalées afin que C_β reçoive simultanément des données influencées par le même pas de C_α.

Le critère imposant le décalage de délivrance des données est la longueur du chemin le plus long entre la source et le puits. En effet, si ce chemin là est de longueur n, une donnée influencée par un pas de C_α se déroulant à la période T ne pourra pas être disponible pour C_β avant la période $T+n-1$. Nous ne pouvons pas avancer la délivrance de cette donnée mais il est possible de retarder la délivrance des données parvenant sur d'autres ports de C_β.

La longueur du plus long chemin nous est donnée par le calcul des rangs des composants du fuseau (voir définition 25). Le rang du puits nous donne la taille du chemin le plus long. Si le puits est de rang r_{C_β}, alors le chemin le plus long reliant la source au puits comprend $n = r_{C_\beta} + 1$ composants. Il s'agit alors d'égaliser tous les chemins afin que si C_α produit une donnée d à la période T, des données influencées par d soit utilisées par C_β à la période $T+n-1 = T+r_{C_\beta}$ sur tous ses ports.

Si C_x est un composant de rang r_x appartenant au fuseau, alors une donnée émise par C_α à la période T est reçue par C_x à la période $T+r_{C_x}$. Si ce composant C_x communique directement avec C_β, C_β doit utiliser les données émises par C_x à la période $T+r_{C_\beta}$. Nous pouvons alors avoir un décalage entre l'émission d'une donnée par C_x et son utilisation par C_β. Nous notons $decalage_{C_xC_\beta}$ ce décalage. Nous avons donc à résoudre :

$$T + r_{C_\beta} = T + r_{C_x} + decalage_{C_xC_\beta}$$

$$decalage_{C_xC_\beta} = r_{C_\beta} - r_{C_x}$$

Nous obtenons alors la condition suivante :

Propriété 4 (Condition de cohérence) *Soit un fuseau entre C_α et C_β composé de plusieurs chemins. Soit C_x un composant de rang r_{C_x} faisant partie d'un chemin du fuseau et communiquant directement avec le composant C_β de rang r_{C_β}. Si C_x émet une donnée durant une période T, alors C_β doit l'utiliser lors de la période $T+(r_{C_\beta}-r_{C_x})$.*

4.2.3 Modes de gestion de la cohérence

Les décalages entre production et utilisation de données peuvent être gérées de différentes façons. Nous pouvons utiliser des files d'attente en entrée des composants puits ou bien en sortie des composants communiquant avec eux.

Dans le cas où les files sont présentes en entrée des puits, il s'agit alors pour le puits d'aller chercher une donnée toujours au même rang dans la file. Sur l'exemple 4.2, il s'agit pour C_3 de ne rien faire durant les trois premiers cycles. Ensuite, à chacun de ses pas, il enregistre la nouvelle donnée qu'il reçoit sur le port B mais consomme la donnée qu'il avait reçue un pas avant.

A l'inverse, ce peut être C_2 qui gère une file d'attente. Il conserve alors la première donnée qu'il produit et il ne commence à transmettre des données qu'au cycle suivant.

A chaque cycle, il transmet la donnée produite au pas précédant et il en produit une nouvelle qu'il stocke dans la file. A chaque cycle, C_2 produit une nouvelle donnée mais envoie à C_3 celle qu'il avait calculée un pas avant.

Il est également possible que ce soit le support de communication et non le composant lui même qui gère les données en attente d'être traitées. Par exemple, toutes les communications peuvent circuler au travers d'un bus logiciel. Ce bus a alors à prendre en charge l'envoi différé des données lorsque cela est nécessaire.

4.2.4 Taille des files d'attente

La condition de cohérence de la propriété 4 nous permet d'obtenir immédiatement la taille des files d'attente à utiliser entre composants. L'objectif est d'obtenir que lorsque C_x émet une donnée durant une période T, alors C_β l'utilise lors de la période $T+(r_{C_\beta}-r_{C_x})$.

Sans contrôle, une donnée émise par C_x à la période T est reçue par C_β à la période $T+1$. Ce fonctionnement revient à l'utilisation d'une file de taille un : durant la période T, C_x dépose une donnée dans cette file, C_β la lit à la période $T+1$. Si l'on souhaite alors modifier ce comportement afin que C_β lise la donnée à $T+(r_{C_\beta}-r_{C_x})$, il faut une file de taille égale à $N=(r_{C_\beta}-r_{C_x})$. A chacun de ses pas, C_β utilise alors la plus vieille donnée de la file. Si la file est plus grande que N, C_β utilise la donnée de rang $r_{C_\beta}-r_{C_x}$ dans la file (le rang 1 correspondant à la dernière donnée entrée dans la file).

Résultat 1 (Taille des files) *Soit un fuseau entre C_α et C_β composé de plusieurs chemins. Soit C_x un composant de rang r_{C_x} faisant partie d'un chemin du fuseau et communiquant directement avec le composant C_β de rang r_{C_β} via le port X. Afin de gérer la cohérence des données, nous devons avoir sur le port X, une file de taille N :*

$$N = r_{C_\beta} - r_{C_x}$$

Après une phase d'initialisation remplissant la file, à chacun de ses pas, C_β lit la plus vieille donnée de la file.

Dans l'exemple de la figure 4.2, C_3 étant de rang 3 et C_2 de rang 1, C_3 a besoin d'une file d'attente de taille 2 sur son port B.

Nous constatons que dans le type de système monopériodique que nous analysons, le principe d'estampillage des données présenté au paragraphe 3.5 est inutile. En effet, le système s'exécutant de façon régulière, le décalage fixe introduit par les files est suffisant. Après une phase d'initialisation du système, les composants se comportent de manière régulière en respectant les contraintes de lecture dans les files ou grâce aux recalages assurés par un bus logiciel.

4.2.5 Contrôle de la cohérence dans les systèmes complexes

Nous nous sommes focalisés sur l'analyse d'un seul fuseau simple. Dans un système, plusieurs fuseaux peuvent être impliqués et ils peuvent interagir entre eux. Nous devons donc tenir compte des imbrications de fuseaux lors de l'analyse d'un système entier.

Prenons l'exemple de la figure 4.4, si nous analysons les fuseaux indépendamment :

– Pour le fuseau F_1 entre C_1 et C_5 composé des chemins (C_1,C_3,C_5) et (C_1,C_4,C_5), nous déduisons que nous avons besoin de files de taille 1 sur les ports A et B car les chemins sont de même taille.

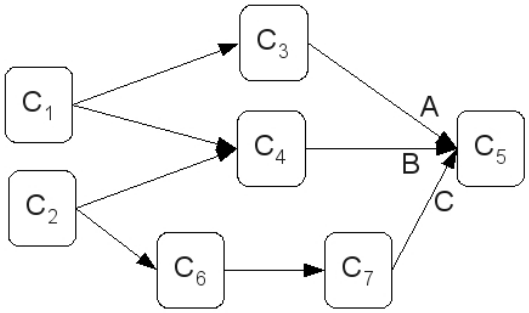


FIGURE 4.4 – Fuseaux imbriqués

– Pour le fuseau F_2 entre C_2 et C_5 composé des chemins (C_2,C_4,C_5) et (C_2,C_6,C_7,C_5), nous déduisons que nous avons besoin d'une file de taille 2 sur le port B car C_5 est de rang 3 et C_4 de rang 1. Nous utilisons une file de taille 1 sur le port C.

Nous voyons alors qu'un premier fuseau impose une file de taille 1 sur le port B alors que le deuxième fuseau impose une taille de 2. Si nous retenons alors la plus grande taille de file et donc le décalage de lecture dans la file correspondant, alors les données ne seront plus cohérentes sur le fuseau F_1. En effet, C_5 utilise alors l'avant dernière donnée qu'il a reçue sur le port B alors qu'il utilise la dernière reçue sur A. Les chemins du fuseau F_1 étant égaux, cela aboutit à des ensembles incohérents.

Nous concluons alors qu'en plus d'imposer une file de taille 2 sur B, il faut également le faire pour le port A afin d'égaliser les chemins du fuseau F_1. En résumé, nous utiliserons une file de taille 1 sur C et des files de taille 2 sur A et B. Nous constatons alors qu'une prise en compte du système global est nécessaire.

Plusieurs méthodes d'analyse se présentent alors. Nous pouvons soit analyser chaque fuseau indépendamment pour en déduire les contraintes que chacun impose, soit analyser directement le système global. Nous verrons que chacune des méthodes est adaptée à des situations différentes.

4.2.5.A Analyse globale du système

Nous pouvons analyser le système global sans même avoir à détecter explicitement les fuseaux au sein de ce système. Nous nous appuyons alors sur les rangs des composants. En effet, nous avons analysé que les rangs permettent de détecter des différences de longueur de chemins. En s'appuyant sur les rangs des composants, nous pouvons donc égaliser tous les chemins. Nous appliquons alors la propriété suivante :

Propriété 5 *Soit un composant C de rang r communicant avec un composant C' de rang r'. Si C produit une donnée à la période T, alors C' doit la consommer à la période $T+(r'-r)$.*

Résultat 2 *Soit un composant C de rang r communiquant avec un composant C' de rang r' via le port X. Pour gérer la cohérence des données le port X doit avoir une file de taille $(r'-r)$.*

Ce résultat générique nous garantit qu'en l'appliquant, la cohérence des données sera traitée sans même avoir identifié où étaient les fuseaux dans le système.

Si nous appliquons ce résultat sur le système 4.4, nous constatons que C_5 est de rang 3 alors que C_4 et C_3 sont de rang 1 et C_7 est de rang 2, nous déduisons donc

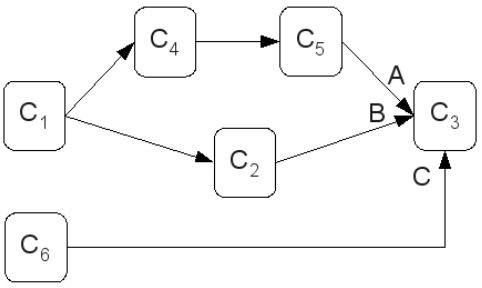


FIGURE 4.5 – Ajout d'un composant au système 4.2

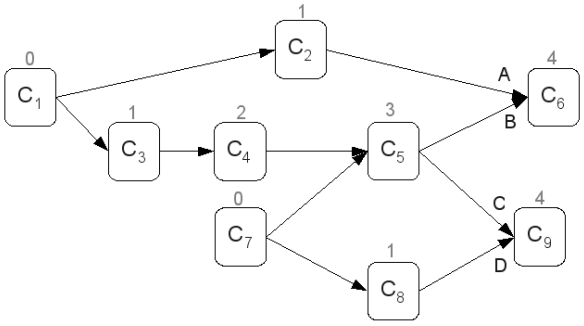


FIGURE 4.6 – Calcul global des rangs des composants

immédiatement qu'il faut utiliser des files de taille 2 sur les ports A et B et une file de taille 1 sur le port C. Nous retrouvons le même résultat que précédemment.

Cette méthode a tout de même un premier point faible. Sur la figure 4.5, nous avons ajouté un composant à la figure 4.2. Le composant C_6 est de rang 0 alors que C_3 est de rang 3. D'après l'analyse globale, il faut alors utiliser une file de taille 3 entre C_6 et C_3. Cependant, le chemin liant ces deux composants n'est pas impliqué dans un fuseau de puits C_3, cette file est donc inutile. De plus, le retard introduit peut être en désaccord avec les objectifs du concepteur qui souhaite que C_3 utilise les valeurs fraîches de C_6.

Nous pouvons donc ignorer de l'analyse globale tout composant ne faisant pas partie d'un fuseau.

Même en retirant les composants ne prenant pas part à un fuseau, l'analyse globale n'est cependant pas toujours optimale. Prenons l'exemple de la figure 4.6. Les rangs globaux sont inscrits à côté de chacun des composants. Nos voyons que C_7 de rang 0, C_5 de rang 3, C_8 de rang 1 et C_9 est de rang 4. Nous dirons alors qu'une file de taille 3 est nécessaire entre C_0 et C_5 ainsi qu'entre C_8 et C_9. Or nous constatons que les chemins composant le fuseau entre C_7 et C_9 sont de même taille. Ici, le fuseau entre C_1 et C_6 influence les rangs de C_5 et C_9 alors qu'il n'influence pas la disponibilité des données sur le fuseau entre C_7 et C_9. L'analyse globale impose un retard dans l'utilisation des données influencées par C_7. Les calculs précédant garantissent bien la cohérence sur C_6 et C_9 mais ils imposent des files qui ne sont pas nécessaires et un comportement qui peut être contraire à la volonté du concepteur.

Dans cet exemple, le rang de C_9 provient de la longueur du chemin le plus long reliant C_0 et C_9. Or ce chemin n'est impliqué dans aucun fuseau. En prenant en

compte le calcul des rangs globaux pour égaliser les chemins, tout est comme si nous avions une source unique virtuelle liée à toutes les sources du graphe.

Dans certains cas, nous pouvons alors adopter un autre mode de résolution en traitant les fuseaux indépendamment.

4.2.5.B Analyse indépendante des fuseaux principaux

Méthode d'analyse Afin de ne pas aboutir à des files inutiles, nous essayons de gérer la cohérence des données en plaçant uniquement des files en entrées des composants puits de fuseaux. Nous verrons plus loin que nous ne pouvons pas toujours gérer la cohérence de cette façon.

Nous commençons par distinguer les fuseaux principaux du système, c'est à dire les fuseaux n'étant pas sous-fuseau d'un autre fuseau (définition 15). Un fuseau principal peut contenir plusieurs sous-fuseaux. Ils seront ici traités tous ensemble.

Nous analysons chaque fuseau principal indépendamment en calculant les rangs de ses composants comme s'il était seul. Nous déduisons de cette analyse les rapports entre les décalages de lecture qu'il est nécessaire d'imposer sur les entrées des puits et nous les exprimons sous forme d'équations. Après avoir analysé tous les fuseaux du système, nous aboutissons alors à un système d'équations. Nous résolvons ce système en cherchant les plus petites solutions entières positives.

Soit un port d'entrée A du composant C_2 recevant les données émises par C_1. Nous notons $decalage_{C_1C_2}$ ou $decalage_A$, le décalage nécessaire entre l'émission d'une donnée par le composant C_1 et sa lecture par C_2 sur le port A. Un décalage égal à 1 signifie qu'une donnée émise par C_1 à la période T est utilisée par le composant C_2 à la période $T+1$. C'est le décalage minimal. L'objectif est de déterminer ces décalages pour tous les ports d'entrée de tous les ports du système. Ces décalages correspondent à la taille de la file qu'il faudra mettre en place sur chacun des ports.

Dans chaque fuseau principal, nous exprimons les rapports entre décalage de lecture sur les entrées des puits. Nous notons r_C le rang du composant C. Soit un puits C_β avec C_β possédant n ports d'entrée A_i recevant chacun respectivement des données d'un composant C_i avec $1 \leq i \leq n$. Pour tout couple (C_i, C_j) avec $i \neq j$:

- si $r_{C_j} \geq r_{C_i}$, nous exprimerons le rapport des décalages sous la forme :

$$decalage_{C_iC_\beta} - decalage_{C_jC_\beta} = r_{C_j} - r_{C_i}$$

- Si $r_i \leq r_j$, alors nous n'écrivons pas d'équation.

L'ensemble des équations obtenues ainsi forme un système que le puits analysé doit respecter afin de gérer la cohérence des données. Nous appliquons ce principe à tous les composants puits appartenant au fuseau principal.

Pour tous les fuseaux principaux du système, nous établissons ces systèmes d'équations. Notons qu'un composant peut être puits de plusieurs fuseaux, nous établissons alors plusieurs systèmes d'équations que les ports de ce puits doivent respecter. Nous retrouvons ce cas dans le système 4.4 où deux fuseaux principaux partagent le même puits C_5.

Nous traitons le système 4.6. Nous obtenons :

- Pour le fuseau entre C_1 et C_6, l'analyse de ce fuseau seul aboutit à des rangs identiques à ceux présentés sur la figure. Nous obtenons :

$$decalage_A - decalage_B = 2$$

- Pour le fuseau entre C_7 et C_9, C_7 est de rang 0, C_5 et C_8 de rang 1 et C_9 de rang 2. Nous obtenons alors :

$$decalage_C - decalage_D = 0$$

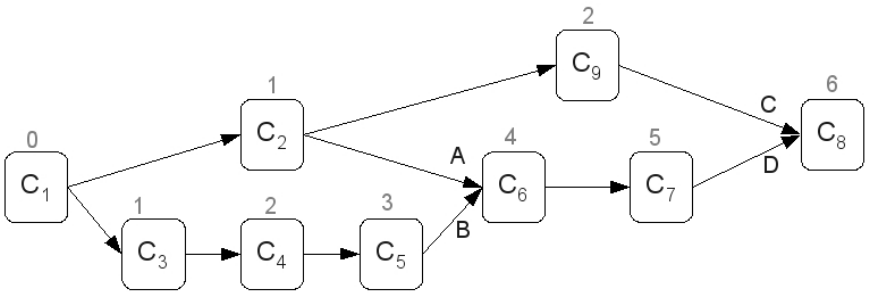


FIGURE 4.7 – Système comprenant des sous-fuseaux

Le plus petite solution entière de ce système est :

$$decalage_B = decalage_C = decalage_D = 1 \text{ et } decalage_A = 3$$

En utilisant la méthode d'analyse des fuseaux principaux indépendamment les uns des autres, nous n'utilisons que des files en entrée des puits. Les autres composants non puits ont uniquement des files de taille 1.

Résolution des équations L'ensemble de toutes les équations obtenues forment un système que nous devons résoudre en trouvant des solutions entières. Nous aboutissons alors à un problème de programmation linéaire en nombre entiers. Des algorithmes de résolutions ainsi que des outils de résolutions sont disponibles pour ce type de problèmes. Nous pouvons citer par exemple l'outil lp_solve, un solveur libre d'équations linéaires. Le critère que nous devons minimiser est l'addition de tous les décalages sur tous les ports d'entrée.

La forme générale du problème que nous souhaitons résoudre est la suivante :

$$\begin{cases} decalage_{A_i} - decalage_{A_j} = k_p \\ k_p \in \mathbb{N} \\ decalage_{A_i} \in \mathbb{N}^* \\ Minimisant \sum decalage_{A_i} \end{cases}$$

Seuls les ports appartenant à un même composant seront liés par des équations. Ainsi, nous pouvons traiter chaque puits indépendamment en résolvant, pour chaque puits, un système d'équations portant uniquement sur ses ports d'entrée.

La programmation linéaire en nombre entier est un problème NP-complet. Cependant nos équations sont posées sous une forme simple (différence entre deux variables) et un composant n'a généralement qu'un nombre restreint de ports d'entrées. Si ce nombre est n, nous obtenons alors au maximum $\frac{n(n-1)}{2}$ équations. Ainsi, une résolution manuelle du système est en général facilement réalisable.

Importance de la notion de fuseau principal Il est important de réaliser l'analyse à partir des fuseaux principaux et non pour chaque fuseau simple du système. Si plusieurs fuseaux sont sous-fuseau d'un autre, alors ces sous-fuseaux influent sur les longueurs des chemins du fuseau principal. En effet, la présence de sous-fuseau au sein d'un autre fuseau implique qu'un puits est présent dans les chemins du fuseau supérieur et éventuellement dans le chemin d'un autre sous-fuseau. Le comportement du puits peut influer sur le cheminement des données à cause de l'attente de données cohérentes qu'il impose. Le temps de parcours d'une donnée sur un chemin peut donc être allongé à cause de son attente en entrée d'un puits.

Prenons l'exemple de la figure 4.7 sur laquelle figurent les rangs des composants obtenus en analysant le fuseau principal (ici, tout le système). Si nous analysons le fuseau entre C_2 et C_8 indépendamment, nous en déduisons que C_8 est de rang 3, C_9 de rang 1 et C_7 de rang 2. Nous obtenons alors que $decalage_{C_9C_8} - decalage_{C_7C_8} = 1$.

Si nous analysons le fuseau principal entre C_1 et C_8, nous calculons que C_8 est de rang 6, C_9 de rang 2 et C_7 de rang 5. Nous obtenons alors que $decalage_{C_9C_8} - decalage_{C_7C_8} = 3$.

Cette différence de décalage est due à la présence du sous-fuseau entre C_1 et C_6. L'utilisation par C_6 des données émises par C_2 est retardée par la présence du chemin $(C_1, C_3, C_4, C_5, C_6)$. En effet, C_6 doit attendre d'avoir des données cohérentes par rapport à C_1. Cet attente sur C_6 influence alors le chemin (C_2, C_6, C_7, C_8) et se retrouve en entrée C_8 qui doit alors décaler l'utilisation des données fournies par C_9 par rapport au retard des données qu'il reçoit de C_7. L'analyse du fuseau entre C_2 et C_8 seul ignore ce retard pourtant bien présent au niveau du système global.

Ce système doit respecter les contraintes suivantes :

$$decalage_A - decalage_B = 2$$

$$decalage_C - decalage_D = 3$$

La plus petite solution entière est :

$$decalage_A = 3, decalage_B = 1, decalage_C = 4, decalage_D = 1$$

Existence d'une solution L'analyse indépendante des fuseaux n'aboutit pas toujours à une solution. En effet, avec ce principe nous restreignons nos possibilités en utilisant uniquement des files en entrée des puits. Cependant, le système d'équations des décalages requis peut ne pas avoir de solution. Cela est possible lorsque le système est composé de fuseaux de sources indépendantes (non reliées par un chemin) et partageant plusieurs composants non reliés par un chemin. En effet, cette configuration impose qu'il existe dans le système un composants puits de deux fuseaux de sources différentes et partageant plusieurs chemins. Ce puits doit donc égaliser les chemins des deux fuseaux afin de gérer la cohérence des données. Suivant la configuration du système, cela n'est pas toujours possible, les contraintes sur un fuseau n'étant pas compatibles avec les contraintes de l'autre.

Nous retrouvons cette configuration dans la figure 4.8. Les rangs globaux sont représentés sur la figure. Ce système est composé de deux fuseaux, un entre C_1 et C_4 et l'autre entre C_6 et C_4. Les fuseaux partagent les deux composants non connexes C_2 et C_1. Le composant C_4 est puits de deux fuseaux de sources indépendantes et partageant deux chemins. Les chemins arrivant sur C_4 doivent donc être égalisés en fonction de contraintes de deux fuseaux non totalement indépendants.

Nous analysons les contraintes imposées par chacun des deux fuseaux sur la figure 4.8. Nous obtenons les équations suivantes :

– Pour le fuseau entre C_1 et C_4 :

$$decalage_B - decalage_A = 1$$

– Pour le fuseau entre C_6 et C_4 :

$$decalage_B - decalage_A = 0$$

Nous constatons que ce système d'équation n'a pas de solution. Les données entrant sur A et B sont influencées par C_1 et C_6. En matière de marquage, elles portent

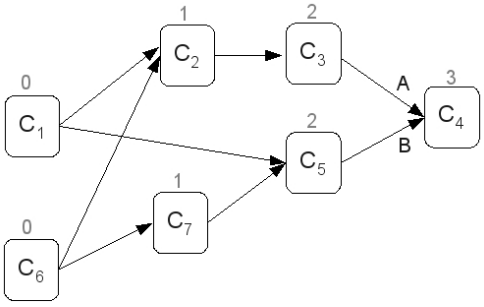


FIGURE 4.8 – Système nécessitant une analyse globale

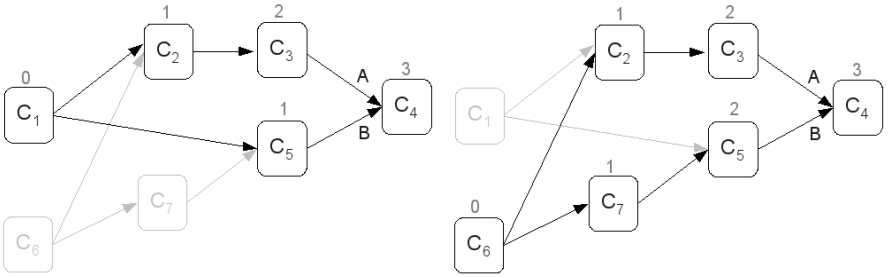


FIGURE 4.9 – Analyse des fuseaux du système 4.8

donc toutes des marquages de ces deux composants. Si nous faisons un contrôle uniquement sur le puits, ces couples de marquages ne seront jamais compatibles. Quelque soit le décalage choisi, si des données sont cohérentes vis-à-vis de C_1, elles ne le seront pas vis-à-vis de C_6 et vice-versa. Nous aurons par exemple sur A une donnée d'estampille $\langle C_1, 2\rangle, \langle C_6, 2\rangle$ et sur B une donnée d'estampille $\langle C_1, 2\rangle, \langle C_6, 1\rangle$. Ces données sont cohérentes par rapport à C_1 mais non par rapport à C_6.

Si l'analyse des fuseaux principaux aboutit à un système sans solution, alors nous devons utiliser l'analyse globale. En effet, elle garantit que chaque composant "attend" la donnée qui prend le plus de temps à se propager jusqu'à lui à partir d'une des sources du système pour utiliser un ensemble de données. Ainsi, toutes les sources sont synchronisées et donc toutes les données influencées par plusieurs sources portent des estampilles compatibles.

Sur ce dernier exemple, le calcul global des rangs aboutit aux files suivantes : des files de taille 1 sur les deux entrées du puits C_4 et une file de taille 2 entre C_1 et C_5. Le décalage de lecture en entrée de C_5 permet de garantir que C_5 et C_2 utilisent les mêmes données en entrée et donc de présenter à C_4 des données portant des estampilles toujours compatibles. Plus précisément, le fait que cette analyse globale impose une source unique virtuelle implique que chaque donnée arrivant à C_4 est influencée par des pas de C_1 et C_6 démarrés lors de la même période du système.

4.3 Conclusion

Nous avons proposé dans ce chapitre une méthode de gestion de la cohérence basée sur l'utilisation de file sur les ports d'entrée des composants. Ces files permettent un

décalage entre la production d'une donnée et sa lecture par un composant destinataire de cette donnée. La cohérence des données est ainsi assurée en équilibrant l'utilisation des différentes données impliquées dans un fuseau. Nous avons également proposé une méthode d'analyse des systèmes afin de calculer les tailles de files nécessaires. Nous avons soulevé le fait que l'imbrication des fuseaux était une contrainte importante à prendre à compte dans cette analyse.

Les systèmes étudiés dans ce chapitre correspondent à un modèle d'exécution très restreint : une unique période pour tous les composants, aucun déphasage et pas de communication entre composants durant une période. Les résultats obtenus ici ne sont donc applicables qu'à un nombre réduit de systèmes réels.

Le modèle pourrait être facilement élargi en intégrant un déphasage entre composants ou en utilisant des composants de périodes différentes mais étant toutes multiples d'une période de base. Nous choisissons de ne pas partir dans cette voie mais de nous intéresser directement à des systèmes plus complexes et dont les contraintes sont beaucoup plus lâches qu'un élargissement du modèle ne le permettrait.

Dans le chapitre suivant, nous nous intéressons à des systèmes multipériodiques, dans lesquels les différentes périodes peuvent être totalement indépendantes et où les communications entre composants ne sont pas liées à la notion de période.

Chapitre 5

Systèmes multipériodiques

5.1 Modèle des systèmes multipériodiques

Nous étudions des systèmes périodiques et nous cherchons à analyser ces systèmes tôt dans leur processus de développement. Plus particulièrement, nous intervenons avant que l'ordonnancement ou le type d'ordonnanceur utilisé ne soient connu. Notre modèle se base alors sur le nombre minimal de paramètres nécessaires pour nous permettre de résoudre le problème d'association de données sans restreindre trop fortement le type de système sur lequel notre méthode peut être appliquée.

5.1.1 Modèle d'exécution

Les systèmes étudiés sont dirigés par le temps et sont composés de composants périodiques. Les périodes respectives des composants sont indépendantes et peuvent donc être différentes.

Un pas d'exécution d'un composant peut être instantané ou durer jusqu'à la totalité d'une période. Durant deux périodes consécutives, les pas d'exécution peuvent avoir des durées différentes. Leur date de début d'exécution peut aussi être différente par rapport à la date de début de leur période

Durant un pas d'exécution, un composant lit une fois chacun de ses ports d'entrée, ensuite il réalise son exécution et pour finir il écrit une fois sur chacun de ses ports de sortie. Si nous considérons un modèle à messages, un composant reçoit exactement un message sur chaque port d'entrée et envoie exactement un message sur chaque port de sortie. La lecture des différents ports ne se fait pas forcément simultanément, de même pour les écritures. Un composant doit finir son exécution avant la fin de sa période. Il ne peut émettre de donnée avant de s'être exécuté pendant son temps d'exécution minimal

Le pas d'exécution d'un composant peut être instantané : la lecture, l'exécution et l'écriture peuvent être réalisées à la même date. Il peut également prendre toute la durée d'une période. Nous utilisons le temps d'exécution minimal s'il est connu, sinon il peut être considéré comme nul. Un pas peut être découpé en morceau, par un ordonnanceur préemptif par exemple.

Nous n'avons pas besoin d'utiliser une horloge globale pour le système. En effet, nous gérons la cohérence par rapport à chaque composant à travers les marquages qu'il produit. Cependant, nous avons tout de même besoin que les composants partagent la même unité de temps. Afin de calculer les propriétés des systèmes que nous étudions, nous calculons des temps globaux. Si le système ne dispose pas d'une horloge globale, il est donc nécessaire qu'il dispose d'un mécanisme de synchronisation ou de recalage

d'horloges. Sur les systèmes orbitaux, la câble PPS (2.2) peut fournir cette unité de temps globale.

5.1.2 Modèle de communication

Nous supposons que les communications sont FIFO et fiables. Les bornes minimales et maximales des temps de communication entre chaque couple de composants sont connues. La borne minimale peut être nulle, permettant ainsi la modélisation d'une mémoire non-transactionnelle. Le fait de pouvoir avoir des temps de communication non nuls permet par contre de modéliser un vrai réseau de communication. La borne supérieure des temps de communication est naturelle dans un contexte temps réel, par exemple lorsque les communications sont réalisées par un bus synchrone ou à qualité de service (QoS) garantie.

5.1.3 Paramètres du système et notations

Pour pouvoir analyser un système, les paramètres suivants doivent être connus :
- Les périodes des composants. Nous notons T_C, la période du composant C.
- Les délais de communication maximaux entre chaque couple de composants. Des bornes supérieures sont suffisantes. Nous notons $\Delta_{CC'}$ le délai de communication maximal entre C et C'.

Des paramètres optionnels peuvent également être connus :
- Les temps d'exécution minimaux des composants. Ces temps correspondent au temps de traitement minimal des données. Entre la lecture d'une entrée et l'émission d'une donnée par un composant, il s'écoule au moins ce temps. Nous notons e_c, le temps d'exécution minimal du composant C. Une borne inférieure est suffisante.
- Les délais de communication minimaux entre chaque couple de composant. Des bornes inférieures sont suffisantes. Nous notons $\delta_{CC'}$ le délai de communication minimal entre C et C'.

5.2 Gestion de la cohérence stricte

Nous avons vu dans le paragraphe 3.5, que nous pouvons identifier des ensembles de données cohérents grâce à l'utilisation d'estampilles sur les données. Un composant peut donc choisir parmi un ensemble de données quelles sont celles qui peuvent composer un ensemble cohérent de données. Nous avons également identifié dans le paragraphe 3.5.3 que nous pouvons réduire les composants marqueurs en n'utilisant comme marqueur que les composants sources d'un puits. Les composants devant vérifier la cohérence des données sont uniquement les composants puits.

En utilisant des files sur les entrées des composants puits, nous pouvons stocker les données arrivant sur ces entrées pour que chaque composant puits choisisse parmi les données disponibles lesquelles utiliser afin d'avoir un ensemble de données cohérent à chacun de ses pas d'exécution. Nous déterminons dans ce qui suit les tailles que doivent avoir ces files afin que le puits ait la garantie de pouvoir construire un ensemble cohérent en utilisant les données stockées dans les files. Nous allons obtenir ces tailles après une analyse des caractéristiques du système. Nous pouvons obtenir des tailles bornées car les composants sont périodiques et les communications entre composants sont bornées.

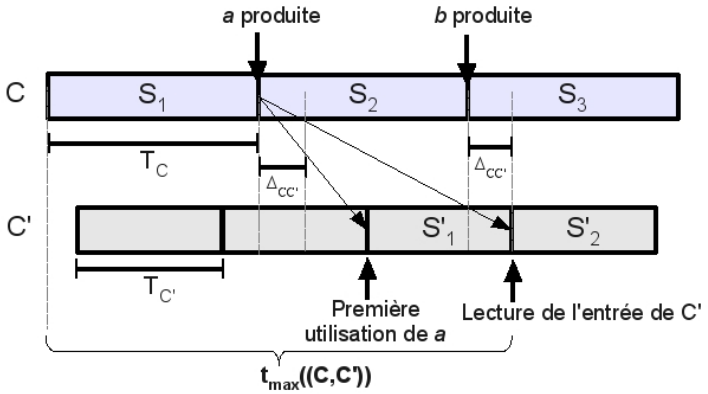


FIGURE 5.1 – Temps de propagation maximal

5.2.1 Analyse d'un fuseau

Afin de calculer les tailles des files à utiliser sur chaque entrée d'un puits, nous analysons les propriétés temporelles de chaque fuseau du système indépendamment.

5.2.1.A Temps de propagation

Définition 26 (Propagation d'une donnée) *Une donnée d se propage via un chemin de composants $(C_1, C_2, ..., C_n)$ si chacun de ses composants utilise, pour au moins un de ses pas d'exécution, une donnée influencée par d.*

Définition 27 (Temps de propagation) *Le temps de propagation d'une donnée d sur un chemin de composants $(C_1, C_2, ..., C_n)$ est le temps s'écoulant entre le début du pas d'exécution de C_1 qui produit la donnée d et l'utilisation par C_n d'une donnée influencée par d.*

Temps de propagation maximal Soit un chemin $P = (C_1, C_2, C_3, ..., C_n)$

Nous définissons $t_{max}(P)$ comme étant le temps maximal pouvant se dérouler entre le début de l'exécution de C_1 qui produit une donnée d et l'utilisation par C_n d'une donnée dépendant de d. $t_{max}(P)$ est le temps de propagation maximal sur le chemin P.

Afin de trouver ce temps maximal, nous considérons que chaque composant du chemin étudié utilise ses données d'entrée au début de sa période et envoie ses données de sortie à la fin de sa période. Nous imposons également que les données utilisent les temps maximaux de communication pour transiter entre composants. Nous fixons enfin la différence de phase entre les composants afin de maximiser l'intervalle entre la production d'une donnée et son utilisation.

La figure 5.1 illustre le décalage maximal qu'il est possible d'obtenir entre la production d'une donnée et son utilisation. Nous étudions la communication entre deux composants C et C', tels que C' utilise les données produites par C

C produit une donnée a à la fin du pas S_1. Il produit ensuite une donnée b à la fin du pas S_2. C' peut utiliser une donnée produite par C après un délai de communication compris entre $\delta_{C_1C_2}$ et $\Delta_{C_1C_2}$. Au plus tard, la donnée b pourra donc être disponible pour C' après un délai $\Delta_{C_1C_2}$. Si C' commence son exécution simultanément (pas S'_2), alors nous ne pouvons déterminer s'il possède sur son port la valeur b. Afin d'obtenir le temps de propagation maximal, nous supposons donc que lorsque C' démarre son pas

d'exécution la valeur b n'est pas encore disponible. Il utilise donc la valeur a. Ainsi, le temps de propagation maximal de la donnée a, et plus généralement le temps de propagation maximal d'une donnée via le chemin (C, C') est :

$$t_{max}((C, C')) = 2T_C + \Delta_{CC'}$$

Nous obtenons alors :

Résultat 3 (Temps de propagation maximal)

$$\textit{Pour un chemin } P = (C_1, C_2, C_3, ..., C_n),$$

$$t_{max}(P) = \sum_{i=1}^{n-1} [2T_{C_i} + \Delta_{C_i C_{i+1}}]$$

Notons que les conditions pour qu'une donnée se propage avec un temps maximal de propagation peuvent être plus souples que ce que nous annonçons au début du paragraphe. En effet, il n'est pas nécessaire que chaque composant utilise ses données d'entrée au début de sa période et envoie ses données de sortie à la fin de sa période. Sur la figure 5.1, le temps de propagation serait le même si la donnée a était produite avant la fin du pas S_1 et si le pas S_2 ne commençait pas dès le début de la période. La condition suffisante est que S_1 démarre au début de sa période et que S_2 émette sa donnée à la fin de sa période. Sur un chemin de donnée, la condition est qu'une donnée suivant le temps de propagation le plus long doit être lue par chaque composant au début d'une période alors que lors de la période suivante, il émet ses données à la fin de la période.

Temps de propagation minimal Soit un chemin $P = (C_1, C_2, C_3, ..., C_n)$

Nous définissons t_{min} comme étant le temps minimal pouvant se dérouler entre le début de l'exécution de C_1 qui produit une donnée d et l'utilisation par C_n d'une donnée dépendant de d. t_{min} est le temps de propagation minimal sur le chemin P.

Ce temps minimal est obtenu en considérant que les composants s'exécutent en n'utilisant que leurs temps minimaux d'exécution et que les temps de communication minimaux sont utilisés entre chaque paire de composants. De plus, la différence de phase entre les exécutions des composants est telle qu'une donnée produite est utilisée le plus rapidement possible, c'est-à-dire après un délai de communication minimal.

Nous obtenons alors :

Résultat 4 (Temps de propagation minimal)

$$\textit{Pour un chemin } P = (C_1, C_2, C_3, ..., C_n),$$

$$t_{min}(P) = \sum_{i=1}^{n-1} [e_{C_i} + \delta_{C_i C_{i+1}}]$$

5.2.1.B Écart de cohérence maximal entre données

Nous considérons le fuseau de la figure 5.2 entre C_α et C_β composé de 2 chemins : $P_A = (C_\alpha, C_2, ...C_{n-1}, C_\beta)$ et $P_B = (C_\alpha, C'_2, ...C'_{m-1}, C_\beta)$. P_A est de taille n et P_B est de taille m. Le composants C_β a deux entrées : A et B. C_{n-1} envoie des données à C_β à travers le port A, C'_{m-1} envoie des données à C_β à travers le port B.

C_β utilise les données des ports A et B. Elles sont influencées par les données produites par C_α. Lorsque C_β lit les données présentes sur ses deux entrées, elle ne sont pas forcément influencées par la même donnée émise par C_α.

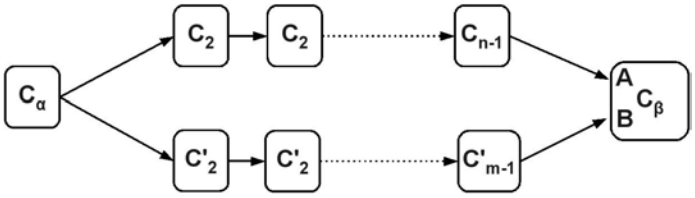


FIGURE 5.2 – Fuseau générique à deux chemins

Nous calculons ici l'écart maximal qu'il peut y avoir entre les deux données de C_α qui influencent les deux données des ports A et B.

Au maximum, une donnée peut en influencer une autre en utilisant le temps de propagation le plus long. Au minimum, elle utilise le temps de propagation le plus court. Nous notons $ecart_{AB}$, l'écart maximal entre les deux pas de C_α qui influencent les deux données des ports A et B lorsque la donnée qui influence la donnée sur A a suivi le temps de propagation le plus long et la donnée qui influence la donnée sur B a suivi le temps de propagation le plus court.

Notre modèle d'exécution n'impose pas que les données d'entrée soient lues en même temps. Pour trouver $ecart_{AB}$, nous nous plaçons donc dans le pire cas : la donnée sur A est lue en début de période de C_β et la donnée sur B est lue le plus tard possible, c'est à dire de façon à ne laisser que le temps minimal d'exécution avant la fin de la période. Nous obtenons alors les formules suivantes :

Résultat 5 (Ecart de cohérence maximal entre données)

$$ecart_{AB} = t_{max}(P_A) + T_\beta - e_\beta - t_{min}(P_B)$$

$$ecart_{BA} = t_{max}(P_B) + T_\beta - e_\beta - t_{min}(P_A)$$

Notons que si les données étaient lues en même temps, nous aurions :

$$ecart_{AB} = t_{max}(P_A) - t_{min}(P_B)$$

$$ecart_{BA} = t_{max}(P_B) - t_{min}(P_A)$$

Si $ecart_{AB}$ est négatif, cela signifie que les données propagées via le chemin P_A seront toujours propagées plus rapidement que sur le chemin P_B.

Nous considérons que le temps de propagation d'une donnée est compté à partir de la date de début d'exécution de C_α qui produit cette donnée. Si nous considérons que ce temps est comptabilisé à partir de la date d'émission d'une donnée, alors, le temps de propagation maximal d'une donné de C_α est $t_{max} - e_\alpha$ et le temps minimal est $t_{min} - e_\alpha$. Notons alors que la formule de $ecart_{AB}$ reste inchangée.

5.2.1.C Exemple d'application

Nous considérons le fuseau de la figure 5.3. Les périodes sont indiquées à l'intérieur des composants. P_A est le chemin (C_1, C_2, C_3, C_4). P_B est le chemin (C_1, C_5, C_4).

Nous supposons que tous les composants ont le même temps d'exécution minimal et que les délais de communication sont les mêmes entre tous les composants.

- Temps d'exécution minimal : $e = 5$
- Délai de communication minimal : $\delta = 5$

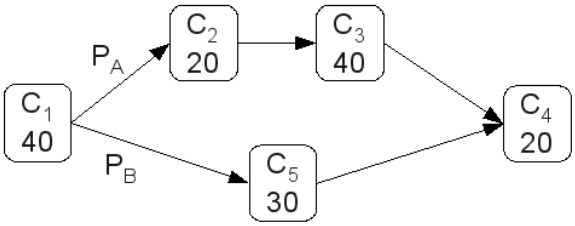


FIGURE 5.3 – Exemple d'application

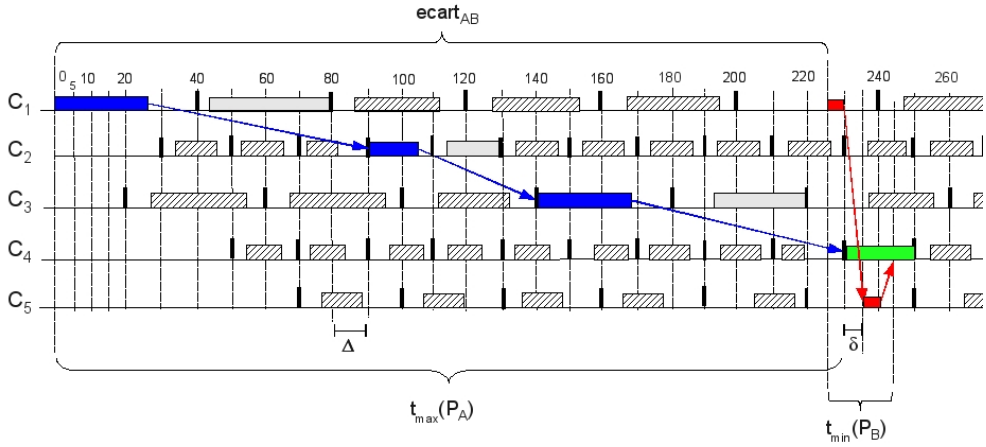


FIGURE 5.4 – Décalage AB

– Délai de communication maximal : $\Delta = 10$

Nous pouvons alors calculer les différentes caractéristiques de ce fuseau :

$$t_{max}(P_A) = (40+20+40)*2+10*3 = 230$$
$$t_{min}(P_A) = 3*5+3*5 = 30$$

$$t_{max}(P_B) = (40+30)*2+10*2 = 160$$
$$t_{min}(P_B) = 5*2+5*2 = 20$$

$$ecart_{AB} = 230+20-5-20 = 225$$
$$ecart_{BA} = 160+20-5-30 = 145$$

Les figures 5.4 et 5.5 illustrent des exécutions pour lesquelles les écarts de données $ecart_{AB}$ et $ecart_{BA}$ sont obtenus. Sur ces figures, nous suivons l'exécution des composants. La queue d'une flèche représente le moment où une donnée est produite, la tête représente le moment où elle est utilisée. Tous les débuts et fins de période sont matérialisés par des traits épais. Chaque composant réalise un pas d'exécution à chacune de ses périodes. Les pas hachurées illustrent des périodes pour lesquelles le composant peut avoir n'importe quel comportement sans influer sur l'écart illustré. Pour les autres périodes, les pas d'exécutions sont représentés par des rectangles pleins.

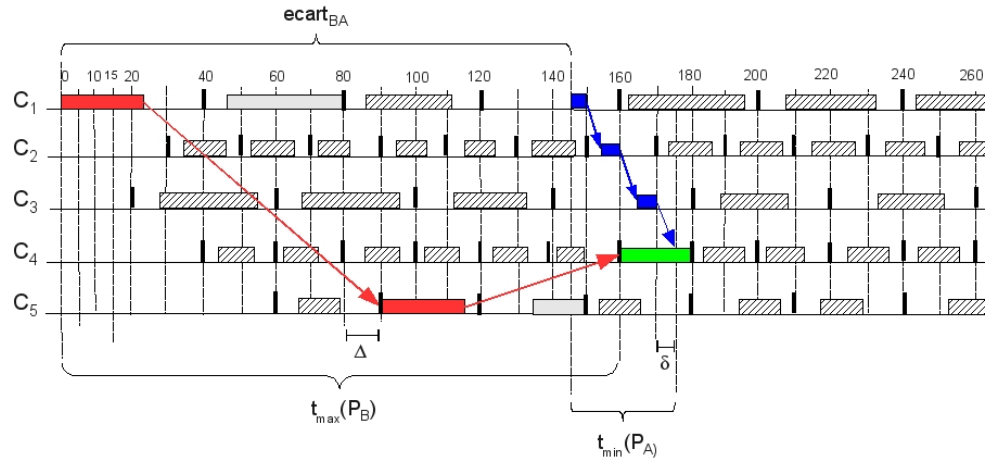


FIGURE 5.5 – Décalage BA

5.2.2 Utilisation de files d'attente

Nous étudions tout d'abord un fuseau entre C_α et C_β composé de deux chemins P_A et P_B tel que décrit dans la figure 5.2 et le paragraphe 5.2.1.B. Nous généraliserons à des fuseaux plus complexes ensuite.

Une donnée d arrivant sur le port A et influencée par un pas S de C_α doit être conservée jusqu'à ce qu'une donnée influencée par ce même pas soit disponible sur le port B. C_β pourra alors composer un ensemble cohérent avec les données dont il dispose en entrée. En attendant la donnée cohérente avec d sur B, d'autres données peuvent arriver sur le port A. Ces données pourront être utilisées par la suite donc il est nécessaire de les conserver également. La file du port A doit donc être assez grande pour stocker toutes les valeurs lui parvenant en attendant que le puits les utilise.

Seuls les ports d'entrée des puits du système nécessitent de gérer la cohérence des données, seuls ces derniers ont donc besoin d'avoir des files d'attente. Pour tous les composants non puits, chaque nouvelle donnée arrivant sur une entrée écrase la dernière donnée stockée.

5.2.2.A Identification des ports nécessitant des files

Une file est nécessaire sur une entrée du puits C_β s'il est possible que l'on ait sur cette entrée une donnée influencée par un pas S de C_α alors que sur l'autre entrée de C_β, une donnée également influencée par S n'est pas encore disponible.

La nécessité de l'utilisation de files est dépendante des temps de propagation minimaux et maximaux d'une donnée produite par C_α au travers des chemins P_A et P_B. Ainsi, nous avons les propriétés suivantes :

- Si $t_{max}(P_A) \geq t_{min}(P_B)$, cela signifie qu'une donnée d émise par C_α peut se propager plus rapidement sur le chemin P_B que sur le chemin P_A. Ainsi, à un moment donné, nous pouvons avoir une donnée influencée par d sur le port B alors que le port A n'a pas encore reçu de donnée influencée par d. Il est donc nécessaire d'utiliser une file sur l'entrée B.
- Si $t_{max}(P_B) \geq t_{min}(P_A)$, alors, par le même principe, une file est nécessaire sur le port A.
- Si les deux conditions précédentes sont vérifiées, alors une file est nécessaire sur les deux entrées de C_β.

En utilisant ces propriétés, nous identifions les ports de C_β qui nécessitent l'utilisation de files.

5.2.2.B Comportement des files

Nous utilisons des files bornées dont nous allons déterminer la taille. Le bornage des files est possible car nous disposons de données bornées sur le comportement du système et notamment car les temps de propagation des données sont eux même bornés. Les files utilisées respectent par ailleurs les propriétés suivantes :

- lorsqu'une valeur est utilisée dans la file, les données plus vielles que cette dernière sont supprimées de la file mais la valeur utilisée est conservée ;
- si une donnée arrive alors que la file est pleine, la donnée la plus ancienne de la file est effacée ;
- le dernier ensemble de données utilisé par un composant est stocké en mémoire. Ainsi, si la fréquence du composant receveur est supérieure à celle du composant émetteur, le receveur utilise les mêmes données pour plusieurs de ses pas.

5.2.2.C Tailles des files

Nous supposons que $t_{max}(P_A) \geq t_{min}(P_B)$. La taille de la file sur le port P_B doit être déterminée. L'objectif est de déterminer le nombre maximal de données qui doivent être conservées sur B jusqu'à ce qu'il soit possible de composer un ensemble de données cohérent avec une donnée présente sur le port A.

Nous rappelons que si la file est pleine et qu'une nouvelle donnée arrive, la donnée d la plus ancienne de la file est effacée et la nouvelle valeur est enregistrée. Si cette situation se produit, nous souhaitons que cela signifie qu'une donnée cohérente avec d ne parviendra jamais sur l'autre port.

La taille de file maximale sur B est obtenue en supposant qu'une donnée issue de C_α se propage sur P_A jusqu'à C_β avec le temps de propagation maximal alors que les données produites par C_α sur P_B utilisent le temps de propagation minimal pour se propager jusqu'à C_β.

Nous avons défini que lorsque C_B utilise une données stockée dans une file, la donnée est conservée dans la file mais les données plus anciennes sont effacées. La taille de la file requise sur B correspond au nombre maximal de données qui peuvent être accumulées sur ce port entre deux suppressions de données par C_β.

Tout d'abord, à partir du moment où une première donnée d influencée par un pas S de C_α arrive sur le port B, des données peuvent s'accumuler sur la file en attendant qu'une donnée d' influencée également par S soit disponible sur le port A.

Pour trouver le pire cas, la donnée d' est le résultat de la propagation d'une donnée de C_α suivant le temps de propagation le plus long. Le délai entre le début du pas S et la dernière utilisation de d' par C_β est ainsi maximisé. Ainsi, avant la dernière utilisation de d', des données peuvent s'accumuler sur B durant $t_{max}(P_A) - t_{min}(P_B)$.

Lorsque C_β utilise d et d', il n'efface que les données plus anciennes que d' dans la file de B. Dans le pire cas, d' est la plus ancienne donnée de la file donc aucune donnée n'est supprimée. Il y a alors une suppression de données uniquement lorsque C_β utilise une autre donnée que d' dans la file. Cela survient lors du pas suivant la dernière utilisation de d'. Nous plaçons cette dernière utilisation de d' au début d'une période de C_β. Lors de la période suivante, la lecture d'une nouvelle donnée est ensuite placée au plus tard, c'est à dire telle qu'il ne reste que le temps d'exécution minimal avant la fin de la période de C_β. Ainsi entre la dernière utilisation de d' et l'utilisation d'une nouvelle donnée, il s'écoule un temps de $2T_{C_\beta} - e_{C_\beta}$.

Au final, des données peuvent s'accumuler sur B durant un temps d'accumulation, noté t_a, égal à :

$$t_a = t_{max}(P_1) - t_{min}(P_2) + 2T_{C_\beta} - e_{C_\beta}$$

Afin d'évaluer le nombre de données pouvant alors s'accumuler, nous devons calculer le nombre maximal de données reçues sur B durant t_a. Ces données sont produites par le composant C'_{m-1}.

Le nombre de périodes complètes n de C'_{m-1} pouvant se dérouler durant t_a est égal à :

$$n_{complete} = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

Une donnée est produite par C'_{m-1} à chacune de ses périodes. Si nous ajoutons à ces données, la première donnée d' déjà présente dans la file au début de t_a, nous constatons qu'il peut donc s'accumuler au moins $n_{complete} + 1$ données sur B.

Il reste également un temps t_i durant lequel C'_{m-1} peut avoir des périodes incomplètes :

$$t_i = t_a - n_{complete} T_{C'_{m-1}}$$

Deux cas se présentent alors :

- Si $t_i < e_{C'_{m-1}}$ alors C'_{m-1} ne peut pas avoir un pas d'exécution complet durant t_i et donc il ne peut pas produire de données supplémentaires dans cet intervalle. Le nombre maximal de données accumulées sur B est donc $N = n_{complete} + 1$.
- Si $t_i \geq e_{C'_{m-1}}$ alors C'_{m-1} a le temps d'exécuter un pas complet durant t_i et donc de produire une donnée supplémentaire. Dans ce cas, $N = n_{complete} + 2$.

Ainsi, nous obtenons une approximation suffisante du nombre de données à conserver dans la file du port B et donc la taille nécessaire de cette file :

Résultat 6 (Taille de file suffisante)

$$N = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor + 2$$

$$N = \left\lfloor \frac{t_{max}(P_A) - t_{min}(P_B) + 2T_{C_\beta} - e_{C_\beta}}{T_{C'_{m-1}}} \right\rfloor + 2$$

Calcul précis de la taille nécessaire Nous pouvons être plus précis sur cette formule de calcul de N. Ce calcul nous sera utile dans un paragraphe suivant pour pouvoir comparer la taille de file obtenue ici avec une taille obtenue pour un autre type de file. Nous cherchons donc à intégrer dans la formule finale les différents cas suivant la valeur de t_i.

Premier cas : $t_i < e_{C'_{m-1}}$

$$t_i < e_{C'_{m-1}} \Leftrightarrow t_a - \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor T_{C'_{m-1}} < e_{C'_{m-1}}$$

$$\Leftrightarrow \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} < \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

$$\left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor \leq \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} < \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

$$\text{Or } 0 \leq e_{C'_{m-1}} \leq T_{C'_{m-1}}$$

$$\Leftrightarrow \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor - 1$$

Dans ce cas, nous avons déterminé que la taille N est égale à :

$$N = n_{complete} + 1 = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor + 1$$

Ce résultat est donc égal dans ce cas à :

$$N = \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

Deuxième cas : $t_i \geq e_{C'_{m-1}}$

$$t_i \geq e_{C'_{m-1}} \Leftrightarrow t_a - \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor T_{C'_{m-1}} \geq e_{C'_{m-1}}$$

$$\Leftrightarrow \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \geq \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

$$\frac{t_a}{T_{C'_{m-1}}} \geq \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \geq \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

$$\Leftrightarrow \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor$$

Dans ce cas, nous avons déterminé que la taille N est égale à :

$$N = n_{complete} + 2 = \left\lfloor \frac{t_a}{T_{C'_{m-1}}} \right\rfloor + 2$$

Ce résultat est donc égal dans ce cas à :

$$N = \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

Nous concluons que dans tous les cas, la taille de N est égale à :

$$N = \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

Résultat 7 (Calcul précis de la taille de file nécessaire)

$$N = \left\lfloor \frac{t_a - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

$$N = \left\lfloor \frac{t_{max}(P_A) - t_{min}(P_B) + 2T_{C_\beta} - e_{C_\beta} - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

Cette formule plus précise ajuste la taille de la file obtenue avec le résultat 6 à une donnée près.

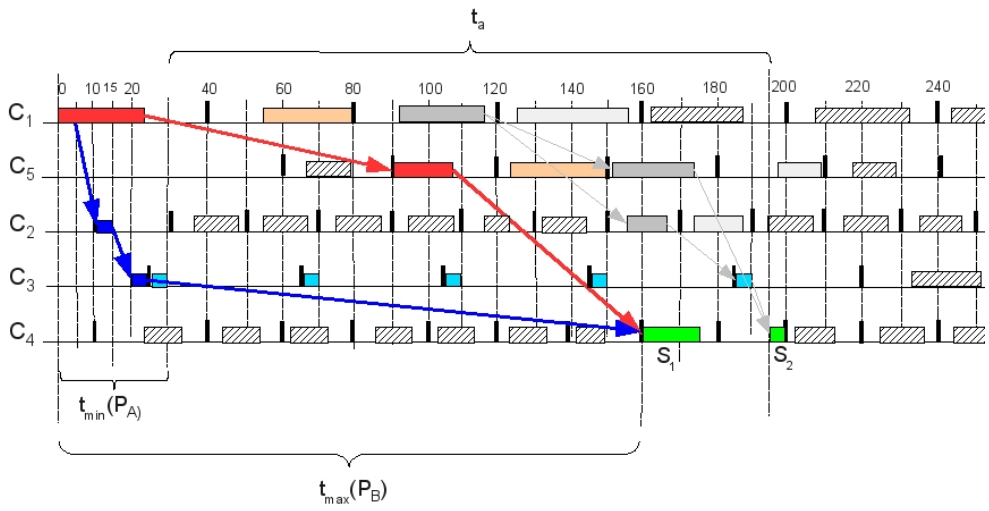


FIGURE 5.6 – Accumulation de données entre C_3 et C_4

Exemple d'application Nous utilisons l'exemple de système du paragraphe 5.2.1.C. Nous appliquons le résultat 7 pour calculer les files nécessaires en entrée du composant C_4.

Sur cet exemple, $t_{max}(P_A) > t_{min}(P_B)$ et $t_{max}(P_B) > t_{min}(P_A)$. Nous avons donc besoin de files sur les deux entrées de C_4.

Entre C_3 et C_4, nous devons utiliser une file de taille :

$$N = \left\lfloor \frac{t_{max}(P_B) - t_{min}(P_A) + 2T_{C_4} - e_{C_4} - e_{C_3}}{T_{C_3}} \right\rfloor + 2$$

$$N = \left\lfloor \frac{160 - 30 + 40 - 5 - 5}{40} \right\rfloor + 2 = 6$$

Entre C_5 et C_4, nous devons utiliser une file de taille

$$N = \left\lfloor \frac{t_{max}(P_A) - t_{min}(P_B) + 2T_{C_4} - e_{C_4} - e_{C_5}}{T_{C_5}} \right\rfloor + 2$$

$$N = \left\lfloor \frac{230 - 20 + 40 - 5 - 5}{30} \right\rfloor + 2 = 10$$

La figure 5.6 illustre une exécution du système pour laquelle le nombre maximal de données s'accumulent entre C_3 et C_4. Sur le chemin P_B une donnée utilise le temps maximal de propagation jusqu'à C_4. Sur le chemin P_A, les données utilisent elles le temps minimal de propagation. De plus, l'exécution de C_3 se déroule de façon à maximiser le nombre de données parvenant dans la file de C_4. Des données accumulées ne seront effacées de la file que lors du pas S_2 de C_4. Nous voyons sur la figure que C_3 aura alors eu le temps d'envoyer 6 données à C_4. Nous retrouvons nos calculs précédents.

5.2.2.D Généralisation à des fuseaux comportant plus de deux chemins

Nous avons calculé la taille des files nécessaires pour un puits comprenant deux entrées et faisant part d'un fuseau composé de deux chemins. Nous allons maintenant présenter la méthodologie à suivre pour des fuseaux plus complexes. Un fuseau

quelconque peut être composé de plus de deux chemins et son puits avoir plus de deux entrées. Remarquons en particulier qu'il peut exister plusieurs chemins reliant la source au puits via un même port d'entrée du puits. Nous retrouvons alors la présence de sous-fuseaux.

L'utilité des files reste la même : permettre de conserver une donnée en attendant qu'elle soit utilisée dans un ensemble de donnée cohérent. Pour un puits à plus de deux entrées, il s'agit alors pour une donnée influencée par un pas S de la source d'attendre que des données influencées par le même pas soient disponibles sur tous les ports du puits.

Nous étudions ici des fuseaux entre C_α et C_β composé de p chemins, notés P_i avec $i \in [1..p]$. Le composant C_β possède q ports d'entrée, notés X_i avec $i \in [1..q]$.

Nous notons $P = (C_\alpha, C_2, ..., C_{n-1}, [X_i]C_\beta)$, un chemin P tel que C_{n-1} est connecté à C_β via le port d'entrée X_i de C_β.

Ports nécessitant l'utilisation de files Comme précédemment, nous identifions tout d'abord les ports du puits nécessitant l'utilisation d'une file. Un port a besoin d'une file s'il est possible qu'une donnée parvenant sur ce port ne puisse pas être utilisée immédiatement dans un ensemble cohérent car il manque des données sur les autres ports du puits pour construire cet ensemble.

Un port d'entrée X_i de C_β nécessite une file si :

$$\exists X_j \neq X_i : \exists P_B = (C_\alpha, ..., [X_i]C_\beta), \exists P_A = (C_\alpha, ..., [X_j]C_\beta) : \\ t_{max}(P_A) \geq t_{min}(P_B)$$

Tailles des files Pour calculer la taille des files nécessaires, il s'agit de déterminer pour chaque port quel est le chemin qui permet de propager une donnée le plus rapidement et le plus lentement possible de la source au port. Supposons qu'une file soit nécessaire sur le port X_i. Le pire cas d'accumulation de données sur un port X_i correspond au cas où :

- les données se propagent de la source à X_i en utilisant le temps de propagation minimal sur le chemin où ce temps est le plus petit. Ce minimum des temps minimaux de propagation de la source au puits via le port Xi est noté $t_{min}[X_i]$:

$$t_{min}[X_i] = min(t_{min}(P_B) : P_B = (C_\alpha, ..., [X_i]C_\beta))$$

- les données se propagent de la source aux autres ports de C_β en utilisant les temps de propagation maximaux sur les chemins où ces temps sont les plus grands. Le maximum des temps de propagation entre la source et le puits via le port X_j est noté $t_{max}[X_j]$:

$$t_{max}[X_j] = max(t_{max}(P_A) : P_A = (C_\alpha, ..., [X_j]C_\beta))$$

Ce qui conditionne la taille de la file sur X_i est l'attente de la donnée prenant le plus de temps pour se propager de la source au puits. Ainsi, seul le chemin emprunté par cette donnée nous intéresse pour calculer la taille de la file. Le temps de propagation maximal entre C_α et C_β via un port différent de X_i est égal à :

$$max(t_{max}[X_j] : j \in [1..q] \wedge X_j \neq X_i)$$

Résultat 8 (Taille des files dans un fuseau complexe) *D'après le résultat 7, la taille nécessaire de la file sur un port X_i est égal à :*

$$N = \left\lfloor \frac{max(t_{max}[X_j] : X_j \neq X_i) - t_{min}[X_i] + 2T_{C_\beta} - e_{C_\beta} - e_{C_\gamma}}{T_{C_\gamma}} \right\rfloor + 2$$

avec C_γ, le composant communiquant avec C_β via le port $[X_i]$.

Étude d'un système entier Au niveau d'un système global, chacun des fuseaux est analysé indépendamment. Si le port d'un composant se trouve impliqué dans plusieurs fuseaux, alors la taille de la file sur ce port sera la taille la plus grande obtenue lors de l'analyse des fuseaux séparés.

5.3 Gestion de la cohérence relâchée

Le concepteur peut ne pas désirer une cohérence stricte sur les entrées de certains composants du système. Dans ce cas, des méthodes autres que l'utilisation de files standards peuvent être utilisées suivant les besoins du concepteur et les propriétés du système.

Nous rappelons la définition du paragraphe 3.1.3 : Si un composant C' utilise plusieurs données dont les valeurs dépendent de sorties d'un même composant C, alors les données d'entrée de C' seront considérées comme cohérentes avec une tolérance τ si, pour tout couple de données, ces données dépendent de pas de C dont les dates de début d'exécution sont séparées au plus d'un temps τ. Nous employons également le terme de données cohérentes τ-relâchées.

Dans le cadre d'une cohérence τ-relâchée, de nouveaux modes de gestion de la cohérence peuvent être utilisés. Bien entendu, la gestion de la cohérence par l'utilisation de file simple comme décrit dans le paragraphe 5.2 est toujours possible. Cependant, nous pouvons profiter de la tolérance de cohérence pour réduire la taille des files. Nous utilisons alors ce que nous appelons des *files filtrantes*.

D'autres méthodes peuvent également être exploitées en fonction de la tolérance voulue et de l'écart de cohérence maximal que nous pouvons avoir entre données sans contrôle de cohérence. Cet écart a été calculé dans le paragraphe 5.2.1.B. En fonction de ces deux paramètres, nous pouvons soit nous abstenir de tout contrôle soit utiliser un procédé retardateur afin de ralentir la délivrance de certaines données.

Notons que parmi un ensemble de données, il peut exister plusieurs sous-ensembles respectant la contrainte de cohérence relâchée. Ainsi, sans plus de précisions, un composant puits peut faire plusieurs choix. Afin de l'orienter, nous pouvons adopter deux objectifs : construire l'ensemble le plus récent ou construire l'ensemble le plus cohérent. Dans l'ensemble le plus récent, les données doivent être influencées par des données les plus récentes possibles de la source. Ce choix est déterministe. Dans l'ensemble le plus cohérent, l'écart de cohérence entre chaque paire de données doit être minimisé. Ce choix peut être indéterministe. Nous pouvons alors ajouter des contraintes en précisant par exemple que parmi plusieurs possibilités, la jeunesse d'une donnée portée par un arc précis doit être privilégiée sur les autres. Pour notre part, nous utilisons le protocole de choix des données les plus récentes. Cependant, les résultats présentés dans ce chapitre ne dépendent pas de ce choix.

5.3.1 Files filtrantes

Nous définissons les files filtrantes comme étant des files qui n'enregistrent pas toutes les données qu'elles reçoivent mais qui les conservent suivant un taux fixe. Elles n'enregistrent qu'une donnée sur un certain nombre, par exemple une donnée est enregistrée sur trois reçues. Excepté ce principe de filtrage, leur comportement est identique aux files simples décrites dans le paragraphe 5.2.2.B.

Ce type de file peut être utilisé afin d'avoir des files de taille plus réduite qu'avec des files classiques telles que celles décrites dans le paragraphe 5.2.2. Elles peuvent être utiles dans deux cas.

En premier lieu, nous pouvons utiliser des files filtrantes lorsque qu'une cohérence stricte n'est pas nécessaire. C'est cet aspect qui va être développé dans les paragraphes suivants.

Mais elles peuvent être utilisées lorsqu'un composant envoie plusieurs fois des données identiques au puits d'un fuseau de façon régulière. Ce phénomène peut avoir lieu si, sur un chemin $(C_1, ..., C_{n-2}, C_{n-1}, C_n)$ reliant la source au puits, le composant C_{n-2} a une fréquence beaucoup plus faible que C_{n-1} et que C_{n-1} a lui une fréquence beaucoup plus élevé que le composant puits. Ainsi, C_{n-1} utilise pour plusieurs pas la même donnée fournie par C_{n-2} et donc une donnée dépendant de la même donnée de la source. Il émet donc également plusieurs fois une donnée dépendant d'une même donnée source. Des données portant des marquages identiques s'accumulent sur le puits. Il est donc inutile de conserver toutes ces données dans la file et nous pouvons utiliser des files filtrantes. Attention, si le fuseau est imbriqué avec d'autres fuseaux, l'utilisation de files filtrantes peut perturber le fonctionnement des autres fuseaux du système. En effet, si C_{n-1} n'est connecté qu'à C_{n-2}, ses données ne sont influencées que par C_{n-2}. Mais si C_{n-1} utilise des données provenant d'un autre composant C', alors même s'il émet plusieurs données influencées par le même pas de C_{n-2}, elles peuvent être également influencées par des pas différents de C'. Si C' fait lui même partie d'un fuseau, alors ignorer certaines de ces données peut influer sur le comportement du fuseau dont il fait partie.

Nous nous concentrerons sur l'utilisation des files filtrantes dans le cadre de la cohérence relâchée. Notre objectif est alors d'une part de déterminer le rythme auquel la file doit enregistrer les données et quelle doit être sa taille.

5.3.1.A Position des files filtrantes

Nous modélisons les files filtrantes comme étant présentes en entrée des composants. Cependant, ces files peuvent être positionnées en sortie des composants ou encore être gérées par le bus de communication entre composants. Le comportement de ces files peut également correspondre à un comportement dégradé volontaire du bus [34], par exemple choisissant de ne laisser passer qu'une donnée sur un certain nombre pour cause de surcharge. Nous choisissons de modéliser les files en entrée des composants plutôt qu'en sortie car cela facilite la gestion des configurations dans lesquelles plusieurs destinataires utilisent une donnée produite par un même composant.

5.3.1.B Propriétés des files filtrantes

Une file filtrante n'enregistrant qu'une donnée sur un certain nombre R, la valeur la plus vieille conservée dans ce type de file est plus vieille que dans une file simple. De même, deux données consécutives dans une file filtrante possèdent des marquages de la source du fuseau plus espacés que dans une file simple.

Nous étudions un fuseau entre C_α et C_β. Nous nous concentrons sur un chemin $P = (C'_1, C'_2, ..., C'_m)$ de ce fuseau, avec $C'_1 = C_\alpha$ et $C'_m = C_\beta$ et nous utilisons une file filtrante de taille N' entre C'_{m-1} et C_m.

Âge de la donnée la plus vieille dans une file filtrante Afin de calculer la taille que doit avoir une file filtrante, nous cherchons à savoir de combien de temps en arrière la file peut nous faire remonter. Plus précisément, nous voulons savoir quel pas de la source la file nous permet d'atteindre. Il s'agit donc de savoir par quel pas de la source la plus vieille donnée de la file est influencée. La file enregistre une donnée sur R reçues avec $R \geq 2$.

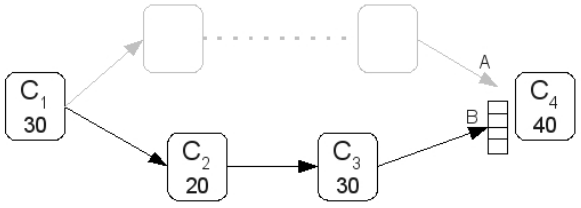


FIGURE 5.7 – Système utilisant une file filtrante

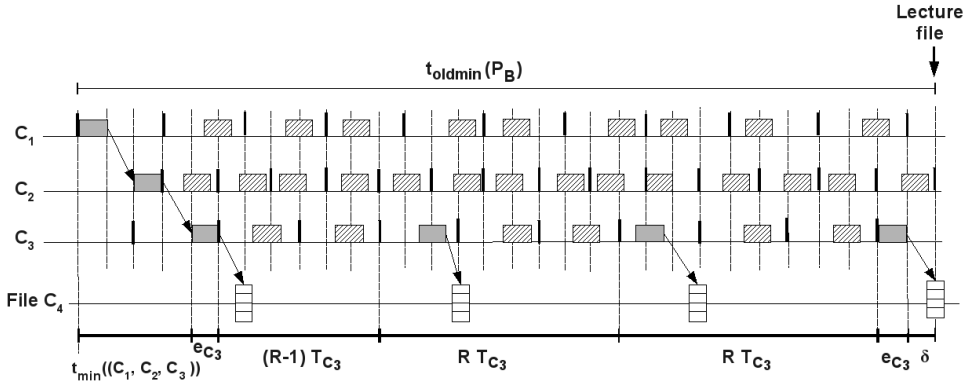


FIGURE 5.8 – Calcul de t_{oldmin}

Nous calculons quel est l'âge minimal du pas influençant la plus vieille donnée de la file. Pour trouver cet âge minimal, nous devons nous placer dans le cas où la file est pleine et où une valeur vient juste d'être ajoutée.

La figure 5.8 illustre le cas où la plus veille donnée de la file est influencée par un pas de la source le plus récent possible. Cette figure trace une exécution du système de la figure 5.7. Dans ce système, les périodes des composants sont indiquées à l'intérieur d'eux, le temps d'exécution minimal est de 10 pour tous les composant et les délais de communications sont fixés à 10.

Sur la figure 5.8, la queue d'une flèche représente l'instant où une donnée est produite. Sa tête représente l'instant où elle est utilisée quand il s'agit d'un pas d'exécution, l'instant où elle entre dans la file quand il s'agit d'une file.

Si nous sommes au temps t, la donnée qui vient d'être ajoutée dans la file provient d'un pas de C'_{m-1} démarré au plus tôt à $t-(e_{C'_{m-1}}+\delta_{C'_{m-1}C_m})$. En effet, C'_{m-1} doit au minimum s'exécuter durant son temps d'exécution minimal avant d'émettre une donnée et la donnée met le temps de communication minimal avant de parvenir à la file.

Ensuite, chaque donnée supplémentaire dans la file permet de reculer de R périodes de C'_{m-1}. Si une donnée provient de la période x, la donnée précédente provient de la période $x-R$. Au sein de cette période, C'_{m-1} peut s'exécuter au plus tard de telle manière qu'il ne lui reste que son temps d'exécution minimal. C'est ce cas que nous retiendrons pour calculer notre temps minimal.

Ainsi, l'avant-dernière donnée nous permet de reculer au minimum d'un temps supplémentaire de $(R-1)T_{C'_{m-1}}+e_{C'_{m-1}}$. Ensuite, chaque donnée précédente nous fait alors reculer d'un temps supplémentaire égal au minimum à $RT_{C'_{m-1}}$.

La plus ancienne donnée de la file provient d'un pas de C'_{m-1} démarré au plus

tard à :

$$t - [(N'-2)RT_{C'_{m-1}} + (R-1)T_{C'_{m-1}} + 2e_{C'_{m-1}} + \delta_{C'_{m-1}C_m}]$$
$$= t - [(N'R-R-1)T_{C'_{m-1}} + 2e_{C'_{m-1}} + \delta_{C'_{m-1}C_m}]$$

Au minimum, cette plus ancienne donnée est influencée par une donnée de la source qui s'est propagée avec le temps de propagation minimal jusqu'à C'_{m-1}. Nous ajoutons ce temps à nos résultats précédent. Nous obtenons donc que la plus ancienne donnée de la file est influencée par un pas de C'_1 démarré au minimum à $t - t_{oldmin}(P)$ avec :

$$t_{oldmin}(P) = (N'R-R-1)T_{C'_{m-1}} + 2e_{C'_{m-1}} + \delta_{C'_{m-1}C_m} + \sum_{i=1}^{m-2}[e_{C'_i} + \delta_{C'_iC'_{i+1}}]$$

$$\Leftrightarrow t_{oldmin}(P) = (N'R-R-1)T_{C'_{m-1}} + e_{C'_{m-1}} + t_{min}(P)$$

Résultat 9 (Age de la donnée la plus vieille dans une file filtrante)

$$t_{oldmin}(P) = (N'R-R-1)T_{C'_{m-1}} + e_{C'_{m-1}} + t_{min}(P)$$

En accédant au temps t à une file filtrante de taille N' et de taux d'enregistrement R, nous avons la garantie que les données stockées dans cette file sont influencées par des pas de C'_1 démarrés au plus tard à $t - t_{oldmin}(P)$.

Âge de la donnée la plus vieille dans une file simple Une file simple est une file dont le taux d'enregistrement R est égal à 1. Nous obtenons donc le résultat suivant :

Résultat 10 (Age de la donnée la plus vieille dans une file simple)

$$t_{oldmin}(P) = (N-2)T_{C'_{m-1}} + e_{C'_{m-1}} + t_{min}(P)$$

Écart entre données consécutives dans une file filtrante Nous cherchons à déterminer l'écart maximal qu'il peut y avoir entre deux pas d'exécution de la source qui influencent deux données consécutives dans la file. Cette information va nous permettre de calibrer le taux d'enregistrement R en fonction de la tolérance de cohérence souhaitée τ.

La figure 5.9 illustre l'écart maximal que nous pouvons obtenir entre deux données consécutives dans la file. Cette figure trace une exécution du système de la figure 5.7.

Sur la figure 5.9, la queue d'une flèche représente l'instant où une donnée est produite. Sa tête représente l'instant où elle est utilisée quand il s'agit d'un pas d'exécution, l'instant où elle entre dans la file quand il s'agit d'une file. Durant les périodes où les pas sont hachurés, les composants peuvent s'exécuter de façon quelconque sans influer sur l'écart représenté ici.

L'écart maximal que nous pouvons obtenir entre deux données consécutives dans la file par rapport aux débuts de pas d'exécution de C'_{m-1} se trouve dans la configuration suivante :

- C'_{m-1} exécute un pas au début de sa période. La donnée produite d est alors stockée dans la file.
- C'_{m-1} s'exécute indifféremment dans sa période durant $(R-1)$ périodes. Les données produites ne sont pas stockées dans la file.

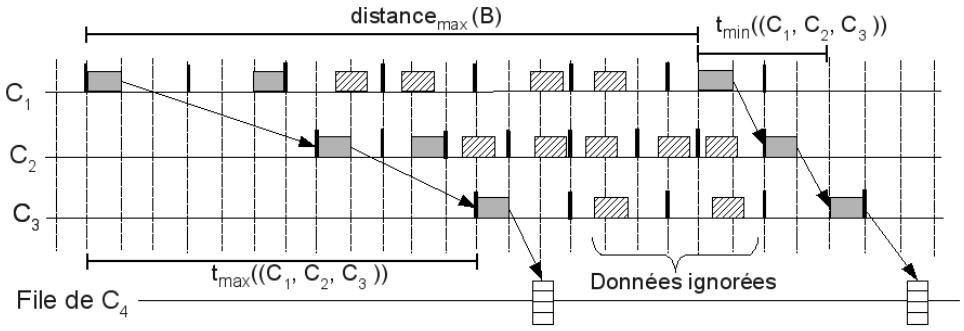


FIGURE 5.9 – Calcul de l'écart maximal entre deux données consécutives

– La période suivante produira une donnée d' qui sera stockée. Afin de maximiser l'écart entre le pas d'exécution ayant produit d et celui produisant d', nous plaçons cette exécution au plus loin dans la période de C'_{m-1}, c'est à dire lorsqu'il ne reste que le temps d'exécution minimal pour s'exécuter.

Ainsi, si nous reprenons les différentes étapes, l'écart maximal par rapport aux débuts de pas d'exécution de C'_{m-1} est :

$$T_{C'_{m-1}} + (R-1)T_{C'_{m-1}} + T_{C'_{m-1}} - e_{C'_{m-1}}$$

$$= (R+1)T_{C'_{m-1}} - e_{C'_{m-1}}$$

L'écart maximal par rapport aux pas d'exécution de la source correspond au cas où la donnée d est le résultat d'une propagation depuis la source jusqu'à C'_{m-1} avec le temps de propagation maximal alors que d' est le résultat d'une propagation avec le temps de propagation minimal. Ainsi, deux données consécutives dans la file du port X dépendront de données produites par la source avec un écart maximal $distance_{max}(X)$ égal à :

$$distance_{max}(X) = (R+1)T_{C'_{m-1}} - e_{C'_{m-1}} + t_{max}(C_1,...,C'_{m-1}) - t_{min}(C_1,...,C'_{m-1})$$

$$distance_{max}(X) = (R+1)T_{C'_{m-1}} - e_{C'_{m-1}} + \sum_{i=1}^{m-2}[2T_{C'_i} + \Delta_{C'_iC'_{i+1}}] - \sum_{i=1}^{m-2}[e_{C'_i} + \delta_{C'_iC'_{i+1}}]$$

$$distance_{max}(X) = \sum_{i=1}^{m-1}[2T_{C'_i} + \Delta_{C'_iC'_{i+1}}] - \sum_{i=1}^{m-1}[e_{C'_i} + \delta_{C'_iC'_{i+1}}] + (R-1)T_{C'_{m-1}} - \Delta_{C'_{m-1}C_m} + \delta_{C'_{m-1}C_m}$$

$$distance_{max}(X) = t_{max}(P) - t_{min}(P) + (R-1)T_{C'_{m-1}} - \Delta_{C'_{m-1}C_m} + \delta_{C'_{m-1}C_m}$$

Résultat 11 (Ecart entre données consécutives dans une file filtrante) *Si la file est placée sur le port X :*

$$distance_{max}(X) = t_{max}(P) - t_{min}(P) + (R-1)T_{C'_{m-1}} - \Delta_{C'_{m-1}C_m} + \delta_{C'_{m-1}C_m}$$

Écart entre données consécutives dans une file simple Dans le cas d'une file simple, R est égal à 1 et donc :

Résultat 12 (Ecart entre données consécutives dans une file simple) *Si la file est placée sur le port X :*

$$distance_{max}(X) = t_{max}(P) - t_{min}(P) - \Delta_{C'_{m-1}C_m} + \delta_{C'_{m-1}C_m}$$

5.3.1.C Calcul du taux d'enregistrement

Nous voulons un taux d'enregistrement qui permette de minimiser la taille de la file tout en garantissant que des ensembles cohérents pourront être construits avec le taux de tolérance défini.

Nous étudions un fuseau entre C_α et C_β composé des chemins $P_A = (C_1, C_2, ..., C_n)$ et $P_B = (C'_1, C'_2, ..., C'_m)$ avec $C_1 = C'_1 = C_\alpha$ et $C_n = C'_m = C_beta$. Nous utilisons une file filtrante sur le port B entre C'_{m-1} et C_β. Nous ne contraignons pas le type de file utilisée sur le port A entre C_{n-1} et C_β.

Au maximum, dans la file filtrante, nous avons des données influencées par des pas de la source démarrés avec un intervalle de $distance_{max}(B)$ entre eux. Le pire cas que nous pouvons avoir en correspondance sur le port A sont des données influencées par des pas de la source démarrant régulièrement à égale distance des dates des pas influençant les données de B. Par exemple, si nous avons sur le port B des données influencées par les pas datés aux temps 10, 20, 30, 40 et sur le port A nous recevons des données datées de 15 et 35. Il est ainsi possible de ne jamais avoir mieux qu'un écart de $distance_{max}(B)/2$ entre les pas d'exécution de C_α qui ont influencés les données disponibles sur les ports A et B.

Afin que même dans le pire cas la construction d'ensemble cohérent soit possible, il faut donc que la tolérance τ soit supérieure à la moitié de $distance_{max}(B)$:

$$distance_{max}(B) \leq 2\tau$$

De cette condition, nous déduisons le taux R à utiliser :

$$t_{max}(P) - t_{min}(P) + (R-1)T_{C'_{m-1}} - \Delta_{C'_{m-1}C_\beta} + \delta_{C'_{m-1}C_\beta} \leq 2\tau$$

$$\Leftrightarrow RT_{C'_{m-1}} \leq 2\tau - t_{max}(P) + t_{min}(P) + T_{C'_{m-1}} + \Delta_{C'_{m-1}C_\beta} - \delta_{C'_{m-1}C_\beta}$$

$$\Leftrightarrow R \leq \frac{2\tau - t_{max}(P) + t_{min}(P) + \Delta_{C'_{m-1}C_\beta} - \delta_{C'_{m-1}C_\beta}}{T_{C'_{m-1}}} + 1$$

De plus R est un entier supérieur à 1, donc R est un entier respectant la condition suivante :

Résultat 13 (Taux d'enregistrement d'une file filtrante)

$$R \leq max(1, \frac{2\tau - t_{max}(P) + t_{min}(P) + \Delta_{C'_{m-1}C_\beta} - \delta_{C'_{m-1}C_\beta}}{T_{C'_{m-1}}} + 1)$$

Si le taux R de la file filtrante ne respecte pas cette condition, alors il est possible qu'avec une cohérence de tolérance τ le puits ne puisse jamais construire d'ensemble cohérent.

5.3.1.D Calcul de la taille de la file

Lorsque le puits lit une donnée dans la file filtrante au temps t, la valeur la plus vieille de la file est influencée par un pas de la source démarré avant $t - t_{oldmin}(P_B)$.

Nous nous plaçons dans le cas où nous utilisons une file filtrante sur le port B et une file simple sur A.

Au temps t, dans le pire cas, la donnée d présente sur le port A est le résultat d'une propagation depuis la source avec le temps de propagation maximal $t_{max}(P_A)$. Ainsi, la donnée lue sur le port A peut être influencée au maximum par une donnée produite à $t - t_{max}(P_A)$. Afin de gérer l'association cohérente des données, nous devons être capable de fournir au puits une donnée cohérente avec d sur le port B. Dans un contexte de cohérence relaxée, cette donnée cohérente sera influencée par un pas de la source démarrée entre $(t - t_{max}(P_A) - \tau)$ et $(t - t_{max}(P_A) + \tau)$:

$$t - t_{max}(P_A) - \tau \leq t - t_{oldmin}(P_B) \leq t - t_{max}(P_A) + \tau$$
$$\Leftrightarrow t_{max}(P_A) - \tau \leq t_{oldmin}(P_B) \leq t_{max}(P_A) + \tau$$

La taille minimale de la file est obtenue quand :

$$t_{oldmin}(P_B) = t_{max}(P_A) - \tau$$

Nous avons donc :

$$(N'R - R - 1)T_{C'_{m-1}} + e_{C'_{m-1}} + t_{min}(P_B) = t_{max}(P_A) - \tau$$

$$\Leftrightarrow N' = \frac{t_{max}(P_A) - \tau - t_{min}(P_B) - e_{C'_{m-1}} + (R+1)T_{C'_{m-1}}}{RT_{C'_{m-1}}}$$

La taille N' étant un entier et avec la condition $t_{oldmin}(P_B) \geq t_{max}(P_A) - \tau$:

$$N' = \left\lceil \frac{t_{max}(P_A) - \tau - t_{min}(P_B) - e_{C'_{m-1}} + (R+1)T_{C'_{m-1}}}{RT_{C'_{m-1}}} \right\rceil$$

Cette taille N' nous permet de stocker suffisamment de données sur le port B pour assurer une correspondance avec le port A. Cependant, tout comme dans le paragraphe 5.2.2, nous devons prendre en compte le pire cas d'utilisation des données par le puits. Le pire cas correspond au cas où les données sont utilisées par C_β au début d'une période et où l'utilisation d'une nouvelle donnée n'est faite qu'au plus tard dans la période suivante, c'est à dire lorsqu'il ne reste que le temps d'exécution minimal avant la fin de la période de C_β. Rappelons que lorsqu'un puits utilise une donnée dans une file, il n'efface que les données plus vieille que cette dernière. Ainsi, une donnée utilisée est effacée au pire après un temps $2T_{C_\beta} - e_{C_\beta}$ après sa dernière utilisation.

Nous avons donc également besoin de prévoir de l'espace pour les données pouvant s'accumuler sur B pendant ce temps supplémentaire. La taille définitive N de la file est donc :

Résultat 14 (Taille d'une file filtrante)

$$N = \left\lceil \frac{t_{max}(P_A) - \tau - t_{min}(P_B) - e_{C'_{m-1}} + (R+1)T_{C'_{m-1}} + 2T_{C_\beta} - e_{C_\beta}}{RT_{C'_{m-1}}} \right\rceil$$

Comparaison avec le calcul des tailles des files simples Une file simple est équivalente à une file filtrante de taux $R = 1$. La cohérence stricte revient à une cohérence de tolérance zéro. Ainsi, la taille N calculée ci-dessus devient :

$$N_b = \left\lceil \frac{t_{max}(P_A) - t_{min}(P_B) - e_{C'_{m-1}} + 2T_{C'_{m-1}} + 2T_{C_\beta} - e_{C_\beta}}{T_{C'_{m-1}}} \right\rceil$$

$$\Leftrightarrow N_b = \left\lceil \frac{t_{max}(P_A) - t_{min}(P_B) + 2T_{C_\beta} - e_{C_\beta} - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rceil + 2$$

Comparons avec la taille précise du résultat 7 calculé dans le paragraphe 5.2.2.C :

$$N_a = \left\lfloor \frac{t_{max}(P_A) - t_{min}(P_B) + 2T_{C_\beta} - e_{C_\beta} - e_{C'_{m-1}}}{T_{C'_{m-1}}} \right\rfloor + 2$$

La seule différence constatée est l'utilisation d'une valeur plafond pour une et d'une valeur plancher pour l'autre. Cette différence est due à la différence d'approche entre les deux calcul. Dans l'approche de calcul pour la file filtrante, nous cherchons à garantir que pour toute donnée influencée par un pas de la source démarré à $t - x$, il existe une valeur influencée par un pas égal ou plus vieux que $t - x$. Cela nous amène à reculer un peu plus loin dans le temps qu'avec une file simple où nous nous arrêtons à une cohérence exacte et nous n'avons donc pas à englober des pas environnant celui qui nous intéresse.

5.3.2 Autres méthodes de gestion de la cohérence relâchée

En fonction de la tolérance voulue et de l'écart de cohérence maximal que nous pouvons avoir entre données sans contrôle de cohérence (calcul du paragraphe 5.2.1.B), nous pouvons utiliser d'autres méthodes que l'utilisation d'estampilles et de files filtrantes. En effet, la définition de la tolérance peut nous laisser une importante marge de manœuvre dans la gestion de la cohérence. En fonction de la tolérance et des paramètres du système, nous pouvons parfois soit nous abstenir de tout contrôle soit utiliser un procédé retardateur afin de ralentir la délivrance de certaines données.

Nous considérons le fuseau de la figure 5.2 entre C_α et C_β composé de deux chemins P_A et P_B. Le composant C_β a deux entrées, A et B. P_A relie C_α à C_β via le port A, P_B via le port B.

5.3.2.A Absence de contrôle de la cohérence

Si la tolérance τ fixée par le concepteur est supérieure à $ecart_{AB}$ et $ecart_{BA}$, alors, sans aucun mécanisme de contrôle, le système respecte les contraintes de cohérence. En effet, le pire cas de décalage entre les débuts des pas d'exécution de C_α influençant les données sur les ports de C_β est inférieur au décalage autorisé par le concepteur. Donc, l'exécution de ce système sera toujours cohérente τ-relâchée sans aucun contrôle.

L'absence de mécanisme de contrôle peut aussi se justifier dans le cas où l'on souhaite utiliser les valeurs les plus fraîches possibles sur chaque port sans tenir compte d'une quelconque cohérence entre ces données.

5.3.2.B Introduction de retards

Si $ecart_{AB}$ ou $ecart_{BA}$ sont supérieurs à la tolérance fixée, alors nous pouvons chercher à réduire ces écarts. Supposons que le problème se situe au niveau de $ecart_{AB} > \tau$.

$$ecart_{AB} = t_{max}(P_A) + T_\beta - e_\beta - t_{min}(P_B)$$

Pour réduire $ecart_{AB}$, nous cherchons à augmenter $t_{min}(P_B)$ en introduisant un retard sur le chemin P_B. Nous pouvons le gérer de deux façons : en utilisant une file sur le composant communiquant avec le port B ou en introduisant un composant retardateur sur le chemin P_B.

Nous constatons toutefois qu'en augmentant $t_{min}(P_B)$ nous augmentons inévitablement $t_{max}(P_B)$ et donc nous augmentons $ecart_{BA}$. Les méthodes présentées ici n'ont donc d'intérêt que s'il existe de grandes disparités de temps de propagation entre les chemins P_A et P_B. Si les temps de propagations sur P_A sont beaucoup plus importants que sur P_B, alors nous pouvons chercher à réduire $ecart_{AB}$ par les méthodes suivantes.

Utilisation de files en sortie des composants Dans le cas d'une file sur le composant C_x communiquant avec le port B, nous imposons qu'à chacun de ses pas d'exécution, C_x produise une donnée qui est alors introduite en queue de file. Il transmet par contre la donnée qui est en tête de file. Les données prennent donc du temps pour se propager jusqu'en tête de file et être envoyées à C_β.

Notons qu'il est différent d'utiliser une file en sortie de C_x ou de placer cette file sur un port d'entrée de C_β. Le rythme d'entrée des données est le même mais pas le rythme d'utilisation de ces données. En effet, en sortie de C_x, les données sont retirées de la file suivant le rythme d'envoi de C_x alors qu'en entrée de C_β, elles sont retirées suivant le rythme de lecture de C_β. Suivant les propriétés des composants, la dynamique de la file peut être très différente.

Pour une file de taille F placée en sortie de C_x, le délai minimal introduit entre la production d'une donnée et son émission se déroule dans le cas où :

- Lors du pas S, C s'exécute à la fin de sa période de façon à avoir uniquement son temps d'exécution minimal de disponible.
- Puis il s'exécute indifféremment pendant $(F-2)$ pas.
- Enfin, lors du pas $S+F-1$, il s'exécute au début de sa période et avec le temps d'exécution minimal

Ainsi la donnée produite lors du pas S sera transmise après un délai $D = 2e_C + (F-2)T_C$ depuis le début de l'exécution de S.

L'introduction d'une file de taille F, augmente donc $t_{min}(P_B)$ de D et diminue donc $ecart_{AB}$ d'autant.

Mais en augmentant $t_{min}(P_B)$, nous augmentons également $t_{max}(P_B)$ car toutes les données produites par C_x sont retardées par la file. En diminuant $ecart_{AB}$, nous augmentons donc $ecart_{BA}$. Nous devons donc faire attention à ce double effet, d'autant plus que cette augmentation est encore plus importante qu'une augmentation de D. En effet, l'augmentation maximale introduite par la file se déroule dans le cas où :

- Lors du pas S, C s'exécute au début de sa période.
- Puis il s'exécute indifféremment pendant $(F-2)$ pas.
- Enfin, lors du pas $S+F-1$, il s'exécute à la fin de sa période et avec le temps d'exécution minimal

Ainsi la donnée produite lors du pas S sera transmise après un délai $D' = FT_C$ depuis le début de l'exécution de S.

L'introduction d'une file de taille F sur C, diminue $ecart_{AB}$ de $D = 2e_C + (F-2)T_C$ mais augmente $ecart_{BA}$ de $D' = FT_C$.

Nous obtenons alors les nouveaux écarts suivants :

$$ecart'_{AB} = ecart_{AB} - 2e_C + (F-2)T_C$$

$$ecart'_{BA} = ecart_{BA} + FT_C$$

Si $ecart_{AB} > ecart_{BA}$, on obtient l'équilibre $ecart'_{AB} = ecart'_{BA}$ lorsque

$$F = \frac{ecart_{AB} - ecart_{BA} + 2e_C}{2}$$

Si avec cette taille de file F, $ecart'_{AB}$ et $ecart'_{BA}$ sont inférieurs à la tolérance τ alors l'utilisation de cette file nous permet de garantir que l'exécution du système sera cohérente. Si ce n'est pas le cas, alors le mécanisme de file seul ne pourra pas nous permettre de gérer la cohérence.

Utilisation de composants retardateurs Un composant retardateur est un composant chargé uniquement de transmettre des données avec un certain retard. Ce composant lit les données et les transmet telles quelles après un certain délai D. En plaçant un tel composant sur le chemin P_B, nous augmentons le temps $t_{min}(P_B)$ de D. Nous réduisons donc $ecart_{AB}$ de D.

Mais en augmentant $t_{min}(P_B)$, nous augmentons également $t_{max}(P_B)$ et donc $ecart_{BA}$. Ainsi, en cherchant à résoudre le problème de décalage dans un sens, nous sommes peut être en train d'en créer un nouveau dans l'autre. Le calcul du délai D à utiliser doit donc tenir compte de ce phénomène.

L'effet maximal que nous pouvons attendre de cette méthode dépend donc de l'influence de ce double effet. Nous introduisons un composant retardateur C sur P_B entre les composant C_x et C_y appartenant à P_B. Le composant retardateur est apériodique, déclenché lorsqu'il reçoit une donnée et introduit un délai D. Nous obtenons alors les nouveaux écarts suivant :

$$ecart'_{AB} = ecart_{AB} - (D + \delta_{C_x,C} + \delta_{C,C_y} - \delta_{C_x,C_y})$$
$$ecart'_{BA} = ecart_{BA} + D + \Delta_{C_x,C} + \Delta_{C,C_y} - \delta_{C_x,C_y}$$

En négligeant les délais de communication pour simplifier la formule, si $ecart_{AB} > ecart_{BA}$, on obtient l'équilibre $ecart'_{AB} = ecart'_{BA}$ lorsque

$$D = \frac{ecart_{AB} - ecart_{BA}}{2}$$

Si alors $ecart'_{AB}$ et $ecart'_{BA}$ sont inférieurs à la tolérance τ fixée par le concepteur, alors l'introduction de ce composant retardateur nous permet de garantir que l'exécution du système sera cohérente. Si ce n'est pas le cas, alors l'introduction seule d'un composant retardateur ne pourra pas nous permettre de gérer la cohérence.

Nous avons simplifié les résultats en supposant que le composant retardateur C est apériodique et déclenché lorsqu'il reçoit une donnée. Si ce composant est périodique, alors le délai D devient le temps minimal d'exécution de C.

Mais son effet sur $t_{max}(P_B)$ est encore plus important. En effet, nous retrouvons alors le même phénomène que lors d'un calcul de temps de propagation maximal avec des composants périodiques. Si nous négligeons les temps de communication, le composant peut donc augmenter le temps maximal de $2T_C$. Pour minimiser alors cette influence, le mieux que nous pouvons faire est de choisir la période de C comme étant égale au délai D et donc égale à son temps minimal d'exécution.

Si nous intégrons les temps de communication le composant C placé entre C_x et C_y sur le chemin P_B augmente $t_{max}(P_B)$ de $2T_C + \Delta_{C_x,C} + \Delta_{C,C_y} - \delta_{C_x,C_y}$.

5.4 Conclusion

Nous avons traité dans ce chapitre le cas de systèmes dont les composants sont de fréquences diverses. Excepté le fait que tous les composants doivent respecter

leurs échéances, nous n'imposons aucune autre contrainte sur l'ordonnancement du système. De la même manière, les modes de communication entre composants sont libres. Il est uniquement nécessaire de connaître les bornes des temps de communication. Pour gérer la cohérence, nous utilisons le mécanisme d'estampillage présenté chapitre 3 associé à des files d'attente sur certains ports d'entrée des composants. Nous calculons la taille nécessaire de ces files.

La cohérence relâchée est ensuite traitée. Dans ce cadre, nous utilisons ce que nous nommons des files filtrantes. Ces files n'enregistrent qu'une donnée sur un certain nombre de données reçues. Elles permettent ainsi de réduire les tailles des files nécessaires pour gérer la cohérence. Nous calculons la taille de ces files ainsi que le rythme d'enregistrement des données qu'elles doivent respecter.

Chapitre 6

Vivacité de l'association cohérente

Dans ce chapitre, nous étudions la vivacité de l'association cohérente des données. Nous disons qu'un puits est vivace s'il est capable de construire infiniment souvent de nouveaux ensembles cohérents. En sens inverse, le fait qu'un puits ne soit pas vivace signifie qu'à partir d'un moment, il réalise tous ses pas de calculs avec un même ensemble d'entrées sans jamais parvenir à obtenir un nouvel ensemble cohérent. Nous allons montrer deux types de situations où il est impossible de garantir la vivacité.

D'une part, du fait du rapport des périodes et de la relative liberté d'exécution laissée au système, certains composants agissent comme des filtres en ne transmettant pas toutes les données qu'ils reçoivent. Par exemple, si C_1 est un composant de période T et qu'il communique avec un composant C_2 de période $2T$, nous voyons qu'en moyenne C_2 n'utilisera qu'une donnée sur deux fournie par C_1. C_2 agit alors comme un filtre pour les données de C_1. Ainsi, un puits ne reçoit pas sur toutes ses entrées des données influencées par tous les pas de la source. Il n'est alors pas garanti qu'il soit capable de construire des ensembles cohérents à partir de ses entrées.

D'autre part, certaines architectures de système comportant des fuseaux imbriqués peuvent également influer sur la vivacité. Dans ce cas, ce sont des contraintes incompatibles sur plusieurs fuseaux qui s'opposent.

6.1 Phénomène de pertes de données

Prenons le fuseau entre C_1 et C_4 de la figure 6.1 composé des chemins $P_1 = (C_1, C_2, C_4)$ et $P_2 = (C_1, C_3, C_4)$ tel que C_1 est de période T et C_2 et C_3 sont de période $2T$. Les composants C_2 et C_3 ne laissent passer en moyenne qu'une donnée sur deux émises par C_1. Supposons que les composants C_1, C_2 et C_3 ont un rythme

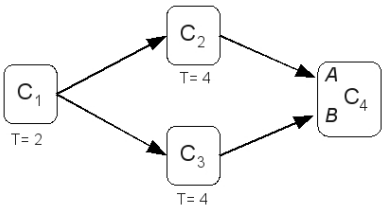


FIGURE 6.1 – Système pouvant générer des pertes de données

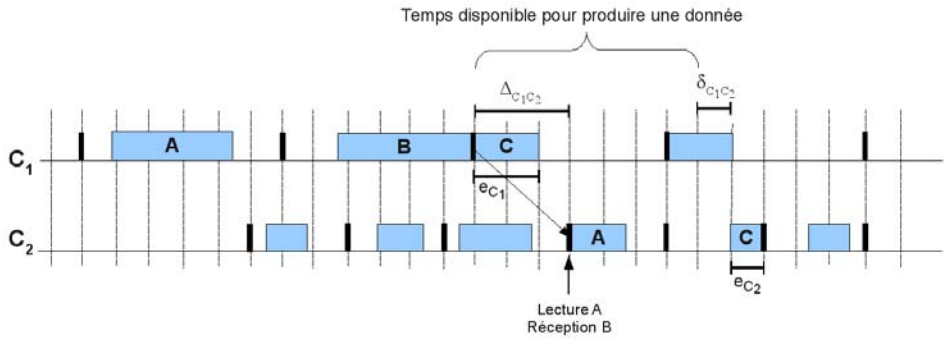


FIGURE 6.2 – Perte d'une donnée

d'exécution régulier c'est-à-dire que chaque composant a un temps d'exécution fixe et une date de début d'exécution fixe elle aussi par rapport au début d'une période. Avec cette exécution régulière, C_2 et C_3 lisent exactement une donnée sur deux consécutives produites par C_1. C_2 et C_3 peuvent être déphasés de telle manière qu'ils ne lisent jamais la même donnée émise par C_1. Quand les données portent des estampilles, nous obtenons que les données propagées de C_1 à C_4 via le chemin P_1 portent des marquages de C_1 de valeurs paires alors que celles propagées via P_2 portent des valeurs impaires (ou l'inverse). Le composant C_4 ne pourra alors jamais construire d'ensemble de données strictement cohérent car il n'aura jamais à sa disposition des données avec le même marquage sur C_1 sur ses deux entrées. L'utilisation de files d'attente ne changerait rien.

Nous constatons que le phénomène de filtrage que peut provoquer un composant sur un chemin de données peut avoir un impact sur la construction d'ensemble de données sur un puits. En général, il diminuera le nombre d'ensembles cohérents constructibles en entrée du puits. Dans le pire cas, comme ci-dessus, il peut amener à un puits ne pouvant jamais construire d'ensembles cohérents.

Le rapport des périodes n'est pas la seule cause possible de ce phénomène de pertes de données. En effet, dans le modèle choisi dans cette thèse, nous ne fixons pas une exécution régulière des composants. Ainsi, des données peuvent être perdues même lorsque les composants partagent la même période, voire quand le composant destinataire a une période plus petite que son composant en entrée. Cela signifie qu'il existe des configurations dans lesquelles il existe des exécutions où il est toujours impossible de construire un ensemble cohérent. Nous allons identifier ces pires cas.

6.1.1 Conditions de pertes de données

6.1.1.A Condition d'absence de perte de données

Soit un composant C_1 produisant des données lues par un composant C_2. Une donnée d produite par C_1 n'est pas lue par C_2 si entre deux lectures de C_2, C_1 a eu le temps de transmettre d suivie d'une autre donnée d'. La donnée d' écrase la donnée d en entrée de C_2 et C_2 utilise alors d'. Nous disons alors qu'il y a eu une perte de donnée ou que la donnée d a été perdue.

La figure 6.2 illustre cette configuration. Le composant C_1 de période 6 et de temps d'exécution minimal 2 communique avec le composant C_2 de période 3 et de temps d'exécution minimal 1. Le délai maximal de communication entre ces deux composants est égal à 3, le temps minimal est égal à 2. Nous suivons la production et l'utilisation des données. A l'intérieur des pas d'exécution de C_1 nous indiquons les

données qu'il émet à la fin de ces pas et pour ceux de C_2, nous indiquons les données qu'il utilise pour ces pas.

Nous indiquons les données que C_1 produit à chacun de ses pas en les faisant figurer à l'intérieur des pas d'exécution correspondants. Pour C_2, nous indiquons à l'intérieur de ses pas les données qu'il utilise en entrée.

Le temps maximal s'écoulant entre deux lectures de C_2 se déroule lorsque, lors d'une première période, C_2 lit sa donnée d'entrée en début de période alors que, lors de la période suivante, il la lit le plus tard possible, c'est-à-dire alors qu'il ne lui reste que son temps d'exécution minimal avant la fin de la période. Le temps maximal s'écoulant entre deux lectures de C_2 est donc égal à $2T_{C_2} - e_{C_2}$.

La pire situation est que C_1 mette à disposition de C_2 une donnée d simultanément avec la première lecture de C_2. Comme ces évènements sont simultanés, nous prenons le pire cas pour nos calcul : C_2 démarre sans avoir pris en compte d. Il utilise la donnée précédemment fournie par C_1.

Le composant C_1 a le temps de transmettre une autre donnée d' avant la lecture suivante de C_2 s'il dispose, depuis l'envoi de d et jusqu'à cette deuxième lecture, du temps minimal d'exécution et du temps minimal de communication.

Au temps disponible entre deux lectures de C_2 nous devons ajouter l'influence des temps de communication. Dans le pire cas, la première donnée d est parvenue à C_2 en utilisant le temps maximal de communication entre C_1 et C_2 alors que la deuxième donnée d' peut prendre le temps minimal de communication. C_1 peut disposer ainsi d'un temps encore plus important pour produire une donnée. Le temps disponible est égal à $2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2}$.

Une donnée produite par C_1 peut donc être ignorée de C_2 si

$$2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} \geq e_{C_1}$$

$$\Leftrightarrow T_{C_2} \geq \frac{1}{2}\,(e_{C_1} + e_{C_2}) - \Delta_{C_1C_2} + \delta_{C_1C_2}$$

Ainsi C_2 perd aucune donnée de C_1 si

$$T_{C_2} < \frac{1}{2}\,(e_{C_1} + e_{C_2)} - \Delta_{C_1C_2} + \delta_{C_1C_2}$$

Cette condition traduit le fait qu'entre deux lectures de C_2, C_1 ne peut jamais produire deux données. Avec cette condition, C_2 lit toutes les données produites par C_1.

Résultat 15 *Si un composant C_1 produit des données pour un composant C_2, alors C_2 lira toutes les données émises par C_1 si :*

$$T_{C_2} < \frac{1}{2}\,(e_{C_1} + e_{C_2)} - \Delta_{C_1C_2} + \delta_{C_1C_2}$$

Nous voyons ici que la perte d'une donnée peut survenir facilement. Il ne s'agit pas de rapport de périodes mais essentiellement d'un rapport entre période du lecteur et temps d'exécution minimal des deux composants. Remarquons que la période de C_1 n'intervient pas.

6.1.1.B Bornes sur les pertes de données

Nous avons vu précédemment la condition à respecter pour ne perdre aucune donnée. Cette condition étant très stricte pour les paramètres du système, nous voulons pouvoir quantifier la perte maximale de données si le système ne la respecte pas.

Nous établissons tout d'abord la condition de perte d'une seule donnée. Le raisonnement est similaire au précédent. Deux données de C_1 peuvent être perdues si C_1 transmet trois données entre deux lectures de C_2. Nous prenons les mêmes conditions que précédemment : nous maximisons le temps entre deux lectures de C_2 et nous supposons que C_1 produit une donnée simultanément avec la première lecture de C_2. Ensuite, il s'agit donc pour C_1 d'avoir le temps de transmettre deux données avant la deuxième lecture de C_2. Le composant C_1 doit alors disposer d'au moins un temps de $T_{C_1} + e_{C_1}$.

C_2 peut donc ignorer deux données de C_1 si :

$$2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} \geq T_{C_1} + e_{C_1}$$
$$T_{C_2} \geq \frac{1}{2}\left(T_{C_1} + e_{C_1} + e_{C_2} - \Delta_{C_1C_2} + \delta_{C_1C_2}\right)$$

Il y aura donc au maximum une seule perte de donnée entre deux lectures de C_2 si :

$$T_{C_2} < \frac{1}{2}\left(T_{C_1} + e_{C_1} + e_{C_2} - \Delta_{C_1C_2} + \delta_{C_1C_2}\right)$$

Généralisons maintenant au cas d'une perte de p données, tel que $p > 1$. Un nombre p de données peuvent être perdues si C_1 transmet $(p+1)$ données entre deux lectures de C_2.

Nous prenons les même hypothèses que précédemment concernant les lectures et émissions des données entre C_1 et C_2. Le composant C_1 produit p données en un temps de $pT_{C_1} + e_{C_1}$

Pour avoir au maximum p pertes entre C_1 et C_2 il faut respecter la condition suivante :

$$2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < pT_{C_1} + e_{C_1}$$
$$T_{C_2} < \frac{1}{2}\left(pT_{C_1} + e_{C_1} + e_{C_2} - \Delta_{C_1C_2} + \delta_{C_1C_2}\right)$$

6.1.1.C Calcul du nombre maximal de pertes

Nous voulons pouvoir déterminer, suivant les paramètres des composants, quelle sera la perte maximale de données entre deux composants.

La condition garantissant qu'il y aura au maximum p pertes est la suivante :

$$2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < pT_{C_1} + e_{C_1}$$
$$\Leftrightarrow p > \frac{2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} - e_{C_1}}{T_{C_1}}$$

Le plus petit entier respectant cette condition est :

$$p = \left\lfloor \frac{2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} - e_{C_1}}{T_{C_1}} \right\rfloor + 1$$

Résultat 16 (Nombre maximal de données perdues) *Si un composant C_1 produit des données pour un composant C_2, alors entre deux lectures de C_2, au maximum p données de C_1 peuvent être perdues, où*

$$p = \left\lfloor \frac{2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} - e_{C_1}}{T_{C_1}} \right\rfloor + 1$$

Si $2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} \geq e_{C_1}$, alors $p = 0$. Nous retrouvons bien le résultat 15.

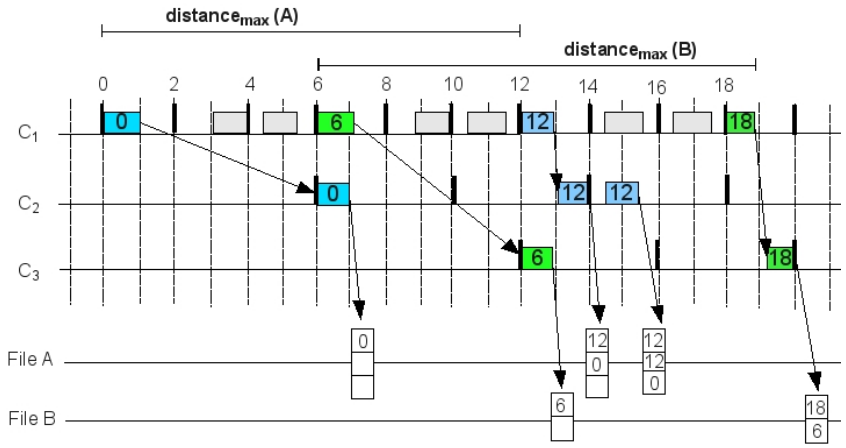


FIGURE 6.3 – Exécution avec pertes du système de la figure 6.1

6.1.2 Impact des pertes de données sur la vivacité

Nous venons de détailler les conditions aboutissant à la perte de p données entre deux composants communicants. Si nous considérons un chemin de composants entre C et C', les pertes peuvent s'enchaîner. Ainsi, seul un sous-ensemble des données émises par C est propagé jusqu'à C'. Si C' dispose d'une file de données sur son entrée, les données conservées ne sont influencées que par un sous-ensemble des données produites par C.

Ce phénomène de pertes sur un chemin de données est sous-jacent dans le calcul de l'écart maximal entre données consécutives stockées dans une file présenté au paragraphe 5.3.1.B. En effet, nous avons constaté que deux valeurs consécutives dans une file (simple ou filtrante) peuvent être influencées par des pas de la source distants de plus d'une période. Cela signifie que certaines données émises par la source ont été "perdues" dans le chemin des composants.

S'il existe un fuseau entre C et C', les données disponibles en entrée de C' sur ces différents ports peuvent être influencés par des pas différents de C. Dans ce cas, comme nous l'avons évoqué au début du paragraphe 6.1, il est possible qu'aucun ensemble strictement cohérent ne puisse être construit.

6.1.2.A Exemple applicatif

La figure 6.3 représente une trace d'exécution du système 6.1 composé des deux chemins $P_A = (C_1, C_2, C_4)$ et $P_B = (C_1, C_3, C_4)$. Dans ce système, le composant C_1 est de période 2 alors que C_2 et C_3 sont de périodes 4. Le temps minimal de communication est de zéro et le temps maximal est égal à 1. Le temps minimal d'exécution de tous les composants est égal à 1.

Sur la figure 6.3, nous indiquons les dates du temps global. Pour illustrer les écart de datation des données, les chiffres à l'intérieur des pas d'exécution des composants représentent, pour C_1, la date à laquelle le pas à démarré et pour C_2 et C_3 la date du pas de C_1 qui a produit la donnée qu'ils utilisent en entrée. Enfin, C_2 et C_3 produisent des données qui sont stockées respectivement dans les files des ports A et B. Chacune de ces données porte également la datation de la donnée de C_1 qui l'influence.

Sur le système étudié, nous appliquons les formules de calcul de pertes proposées précédemment. Nous obtenons qu'entre C_1 et C_2, tout comme entre C_3 et C_4, il peut

survenir au maximum p pertes avec p égal à :

$$p = \left\lfloor \frac{2T_{C_2} - e_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} - e_1}{T_{C_1}} \right\rfloor + 1$$

$$p = \left\lfloor \frac{8-1+2-0-1}{2} \right\rfloor + 1 = 5$$

La figure 6.3 illustre ce pire cas de perte. Elle illustre également le pire cas en terme d'écartement maximal entre données consécutives dans les files. En effet, si nous reprenons le calcul du paragraphe 5.3.1.B, nous obtenons pour la file du port A

$$distance_{max}(A) = t_{max}(P_A) - t_{min}(P_A) - \Delta_{C'_{m-1}C_m} + \delta_{C'_{m-1}C_m}$$

$$distance_{max}(A) = 12 - 2 - 2 + 0 = 12$$

Le chemin P_B ayant les mêmes caractéristiques que P_A, $distance_{max}(A) = distance_{max}(B)$.

Nous constatons que nous pouvons avoir, dans chacune des files, des données consécutives influencées par des pas de la source espacées de leur écart maximal. Le pire cas de correspondance entre les données des deux files est tracé. Il correspond au cas où les pas influençant les données de la file A sont décalés de $distance_{max}(A)/2$ par rapport à ceux influençant les données de la file B. Autrement dit, les pas influençant les données de la file A démarrent aux dates x, $x + distance_{max}(A)$, $x + 2distance_{max}(A)$, etc, alors que les pas influençant les données de la file B démarrent aux dates $x + distance_{max}(A)/2$, $x + distance_{max}(A)/2 + distance_{max}(A)$, $x + distance_{max}(A)/2 + 2distance_{max}(A)$ etc.

Nous voyons dans ce cas que la construction d'ensembles strictement cohérents à partir des données stockées dans les files A et B est impossible. Tout ensemble possible est composé de données dépendant de pas de C_1 espacés au minimum de $distance_{max}(A)/2$. Si une tolérance de cohérence supérieure à $distance_{max}(A)/2$ est acceptée, il est par contre possible de construire des ensembles cohérents. En acceptant cette tolérance, il est garanti que le système du graphe 6.1 pourra construire des ensembles cohérents sur son puits lors de son exécution. Si la tolérance est inférieure à $distance_{max}(A)/2$, alors il est possible que le système ne puisse jamais composer d'ensemble cohérent. Nous ne pouvons pas garantir sa vivacité.

6.1.2.B Cas général

Soit un fuseau quelconque entre C et C' composé de deux chemins P_A et P_B connectés respectivement à C' via les ports A et B. Les données stockées dans la file du port A sont influencées par des pas de C espacés au maximum de $distance_{max}(A)$, celles du port B par des pas de C espacés au maximum de $distance_{max}(B)$. Nous supposons que $distance_{max}(A) < distance_{max}(B)$. Ces paramètres conditionnent la précision de cohérence que l'on peut demander au puits si l'on veut garantir que des ensembles cohérents soient constructibles.

Dans le pire des cas, nous avons sur la file du port A, des données influencées par des pas de C espacés de $distance_{max}(A)$. En correspondance, le pire des cas se produit lorsque les données stockées dans la file de B sont influencées par des pas de C démarrés régulièrement à égale distance des pas influençant les données de la file A. Par exemple, nous pouvons avoir dans la file A des données influencées par des pas démarrés aux dates 10, 20, 30, 40 et sur la file de B, des pas démarrés à 15 et 35.

Quel que soit l'exécution du système et quel que soit la donnée d considérée dans la file de B, il existera toujours dans la file de A une donnée influencée par un pas de

C au maximum distant de $distance_{max}(A)/2$ par rapport au pas qui a influencé d. La tolérance de cohérence à accepter si nous voulons un système capable de garantir la construction des ensembles cohérents correspond donc à $distance_{max}(A)/2$.

Dans un fuseau composé de plus de deux chemins, nous comparons les files d'entrée du puits deux à deux. Pour tout couple de files A et B, si $distance_{max}(A) < distance_{max}(B)$, alors la tolérance à accepter est égale à $distance_{max}(A)/2$. Dans un ensemble de données, le pire cas de cohérence sera conditionné par le couple pouvant avoir le plus grand écart de cohérence. Après avoir analysé tous les couples, nous conservons donc le maximum des tolérances obtenues. C'est au minimum cette tolérance (τ_{min}) qui devra être acceptée par le système pour garantir qu'il soit vivace. Plus simplement, si les écarts maximaux des files sont rangés par ordre croissant, τ_{min} correspond à l'avant dernier écart divisé par 2.

Résultat 17 *Soit un fuseau entre C et C'. Le puits C' possède n files d'entrée notées X_i $(1 \leq i \leq n)$. Il est garanti que le puits peut construire des ensembles cohérents si une tolérance supérieure à τ_{min} est acceptée, avec :*

$$S = \{\{distance_{max}(X_i) : i \in [1..n]\}\}$$

$$\tau_{min} = \frac{Max(S \setminus \{\{Max(S)\}\})}{2}$$

En acceptant une tolérance de cohérence égale à τ_{min}, il est garanti que le puits du fuseau est vivace, c'est-à-dire qu'il a la possibilité de construire différents ensembles cohérents lors de son exécution. Par contre, si une tolérance plus petite est demandée, alors la vivacité du puits dépendra de l'exécution exacte de système.

6.1.3 Simplification du modèle

Nous constatons dans le paragraphe 6.1.1 que des pertes de données peuvent facilement se produire, entraînant alors le besoin d'accepter parfois une tolérance de cohérence importante sur le puits d'un fuseau. A titre d'exemple, nous allons montrer comment un modèle d'exécution moins libre, plus régulier, réduit significativement cette tolérance. Nous considérons que les temps d'exécution des composants sont fixes et que les débuts des pas d'exécution sont toujours les mêmes par rapport aux débuts des périodes. Nous cherchons à déterminer l'impact de ce modèle sur le phénomène de perte de données.

Sous ces contraintes, deux lectures de C_2 sont espacées de T_2 et deux productions de données de C_1 sont espacées de T_1.

Nous reprenons les mêmes principes de calcul dans le paragraphe 6.1.1. Après une première production de donnée, le composant C_1 a dans ce nouveau modèle un temps égal à $T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2}$ pour en produire d'autres.

Pour n'avoir aucune perte, il faut que C_1 n'ait pas le temps de produire de donnée durant ce temps. Il faut donc :

$$T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < T_{C_1}$$

Pour avoir au maximum une perte, il faut que C_1 n'ait pas le temps de produire deux données durant ce temps

$$T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < 2T_{C_1}$$

Nous aurons au maximum p pertes si :

$$T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < (p+1)T_{C_1}$$

A partir des paramètres des composants, nous pouvons déterminer le nombre maximal de pertes.

$$T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2} < (p+1)T_{C_1}$$
$$\Leftrightarrow \frac{T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2}}{T_{C_1}} - 1 < p$$

Le plus petit entier respectant cette condition est :

$$p = \left\lfloor \frac{T_{C_2} + \Delta_{C_1C_2} - \delta_{C_1C_2}}{T_{C_1}} \right\rfloor$$

Si nous considérons que les temps de communication sont nuls ou fixes :

$$p = \left\lfloor \frac{T_{C_2}}{T_{C_1}} \right\rfloor$$

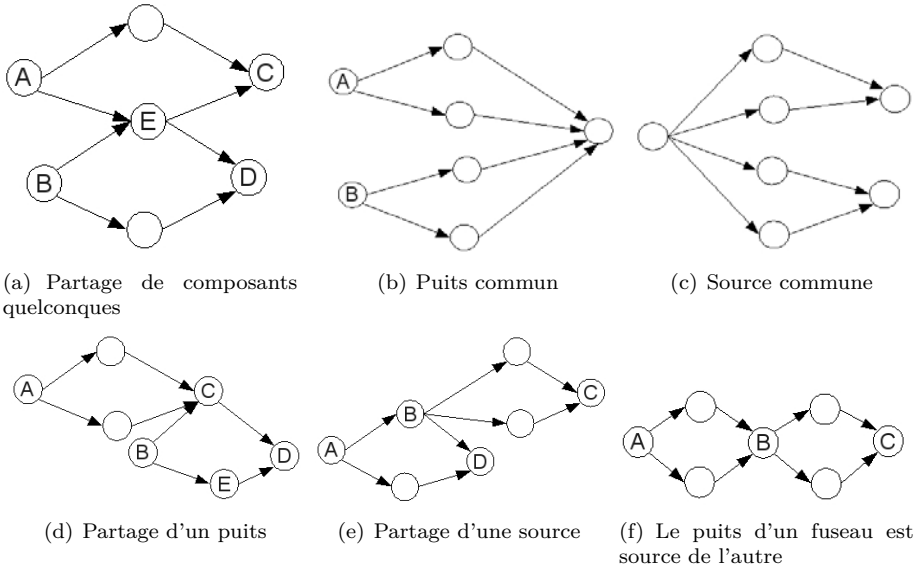
Nous voyons qu'une perte de donnée peut arriver même si les deux composants sont de la même période. Cela correspond au cas où C_2 lit ses données exactement en même temps que C_1 lui en transmet une. Sans précision, nous ne savons pas comment le système se comporte : C_2 prend-il en compte la nouvelle donnée ou non ? Si pour un pas, C_2 ne prend pas en compte la donnée d que C_1 lui met à disposition et que, pour le pas suivant, il prend par contre en compte la donnée d' que C_1 lui transmet, alors d est perdue. Cependant, vu les hypothèses définies sur le système, nous pouvons supposer la régularité d'un tel mécanisme. Nous établissons alors que, soit C_2 lit toujours son entrée après avoir pris en compte la donnée provenant de C_1, soit l'inverse. Dans ce cas, il n'existe pas de perte de données entre composants de même période.

Nous constatons qu'avec ce nouveau modèle d'exécution, les pertes de données entre composants sont réduites. Ainsi, les écarts maximaux entre données stockées dans les files du puits seront également réduits. D'après l'approche proposée dans le paragraphe 6.1.2, nous déduisons alors que la tolérance de cohérence à accepter pour garantir la vivacité du puits peut être réduite. L'ajout de contrainte de régularité sur le modèle d'exécution permet de gérer une cohérence plus forte entre données.

6.2 Influence de l'architecture du système sur la vivacité

Nous avons soulevé dans le chapitre 4 la problématique particulière des fuseaux imbriqués. En effet, dès lors qu'il existe des fuseaux imbriqués, ces fuseaux peuvent interagir entre eux et cette interaction peut influer sur la gestion de la cohérence dans les différents fuseaux. Avec le modèle d'exécution utilisé dans le chapitre 4, nous avons constaté qu'il n'existe pas toujours une solution à la gestion de la cohérence. Nous allons analyser dans cette section l'influence de l'architecture des systèmes dans le cadre du modèle d'exécution du chapitre 5, ou plus généralement dans tout modèle n'imposant aucune contrainte sur l'ordonnancement des composants.

La méthodologie appliquée est de considérer deux fuseaux F_1 et F_2 ayant un certain comportement lorsqu'ils fonctionnent indépendamment, puis d'imbriquer ces deux fuseaux en imposant la contrainte que certains composants appartenant à des chemins de F_1 appartiennent également à des chemins de F_2. Nous disons alors que F_1 et F_2 partagent des composants. Nous cherchons à déterminer si, après imbrication, les fuseaux ont toujours le même comportement en terme de gestion de cohérence que lorsqu'ils s'exécutent séparément. Nous pouvons distinguer les différents cas de partage de composants en distinguant les composants sources d'un fuseau, ceux étant



(a) Partage de composants quelconques

(b) Puits commun

(c) Source commune

(d) Partage d'un puits

(e) Partage d'une source

(f) Le puits d'un fuseau est source de l'autre

FIGURE 6.4 – Partage d'un composant sans influence sur la vivacité

puits et ceux n'appartenant à aucune de ces catégories que nous appelons alors composants quelconques.

Le puits d'un fuseau apparie des données de façon à ce qu'elles composent un ensemble cohérent. En s'appuyant sur les estampilles, il doit construire un ensemble de données tel que l'union des estampilles des données ne contienne pas différents marquages provenant d'un même composant. Nous nous plaçons dans le cadre où seules les sources de fuseau génèrent des marquages. S'il existe plusieurs fuseaux dans le système, alors des marquages provenant de composants différents circulent dans le système.

6.2.1 Configurations n'influençant pas la vivacité

6.2.1.A Partage d'un seul composant

Le premier cas d'imbrication possible est le partage d'un seul composant. Ce composant peut alors être :

- un composant quelconque pour les deux fuseaux (figure 6.4(a))
- un composant puits des deux fuseaux (figure 6.4(b))
- un composant source des deux fuseaux (figure 6.4(c))
- un composant puits pour un fuseau mais quelconque pour l'autre (figure 6.4(d))
- un composant source pour un fuseau mais quelconque pour l'autre (figure 6.4(e))
- un composant puits pour un fuseau mais source pour l'autre (figure 6.4(f))

Analysons le graphe 6.4(a). Il est composé de deux fuseaux partageant le composant quelconque E. Le composant E produit des données à la fois influencées par A et par B. Ces données portent donc des marquages de ces deux composants. Cependant, son comportement n'est pas perturbé par ce phénomène. Les puits C et D reçoivent des données portant les marquages de A et B via E. Le composant C reçoit également des données marquées par A sur son autre entrée alors que le composant C reçoit sur son autre entrée des données marquées par B. Le composant C contrôle alors la cohérence des données par rapport à A et le composant D la contrôle par

rapport à B. Ce comportement est identique à celui que les fuseaux ont lorsqu'ils sont pris indépendamment.

Le partage d'un composant quelconque n'a donc aucun effet sur la gestion de la cohérence des deux fuseaux.

Analysons plus rapidement les autres cas de partage d'un composant :

- Dans le graphe 6.4(b), le puits commun contrôle la cohérence des données deux à deux, par rapport à A d'une part et B d'autre part. Son comportement est donc identique à celui qu'il a lorsqu'il appartient à un seul des deux fuseaux. Le partage d'un puits commun n'a donc aucun effet sur la gestion de la cohérence.
- Dans le graphe 6.4(c), le fait d'avoir une source commune n'a aucune influence sur les fuseaux. Les deux puits vérifient simplement la cohérence des données par rapport à un même composant.
- Dans le graphe 6.4(d), le puits C partagé laisse passer les données de B comme un composant quelconque le ferait. Il contrôle uniquement la cohérence par rapport à A. Le puits D contrôle lui la cohérence uniquement par rapport à B. Il reçoit de la part de C des données marquées par A et B mais uniquement des données marquées par B de la part de E.
- Dans le graphe 6.4(e), le fuseau entre A et D n'est pas influencé par le partage de B avec un autre fuseau. C reçoit sur ces deux entrées des données portant des marquages de A et B. Cependant, à chaque marquage de B correspond un seul A. La compatibilité des marquages de B implique donc la compatibilité des marquages de A. Le fuseau entre B et C conserve son comportement initial.
- Dans le graphe 6.4(e), le fuseau partageant son puits B ne connaît aucun changement de comportement si ce n'est que le puits génère des marquages. Le fuseau partageant sa source voit passer des marquages des deux sources mais, comme précédemment, ces marquages sont liés.

De l'analyse exhaustive décrite dans ce paragraphe, nous pouvons déduire la propriété suivante :

Propriété 6 *Si deux fuseaux ne partagent qu'un seul composant, alors leur comportement en terme de gestion de cohérence est identique au comportement qu'ils ont lorsqu'ils sont analysés séparément.*

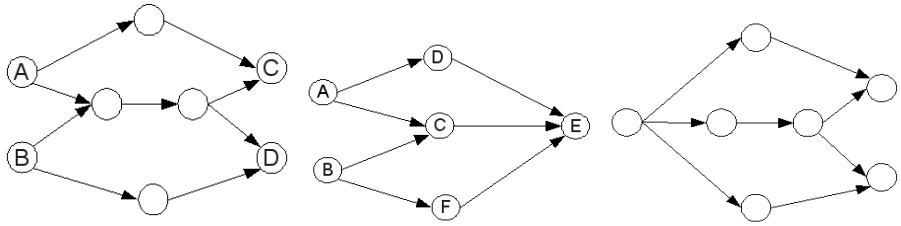
6.2.1.B Partage de plusieurs composants appartenant au même chemin

Nous pouvons étendre les différents types de partage décrits précédemment en considérant des fuseaux partageant non pas un seul composant mais une partie de chemin, c'est-à-dire partageant plusieurs composants appartenant à un seul chemin. Nous ajoutons aux différents cas précédents, le partage supplémentaire de composants quelconques. Nous obtenons alors :

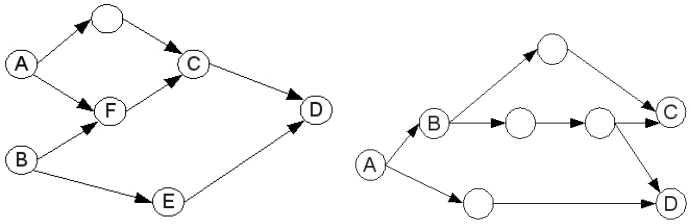
- à partir du graphe 6.4(a), le graphe 6.5(a)
- à partir du graphe 6.4(b), le graphe 6.5(b)
- à partir du graphe 6.4(c), le graphe 6.5(c)
- à partir du graphe 6.4(d), le graphe 6.5(d)
- à partir du graphe 6.4(e), le graphe 6.5(e)

Les systèmes des graphes 6.5(a), 6.5(c) et 6.5(e) ont un comportement comparable à leur équivalent ne partageant qu'un seul composant.

Par contre, le graphe 6.5(b) a un comportement différent. Dans ce système, un composant est puits de deux fuseaux. Les deux fuseaux du système partagent également le composant C. E reçoit des données marquées par A via D, des données marquées par A et B via C et des données marquées par B via F. Dans ce système, E peut ne pas pouvoir créer d'ensemble cohérent, une donnée de C cohérente avec une donnée de D par rapport aux marquages de A peut ne pas être cohérente en



(a) Partage de composants quelconques (b) Puits commun (c) Source commune



(d) Partage d'un puits (e) Partage d'une source

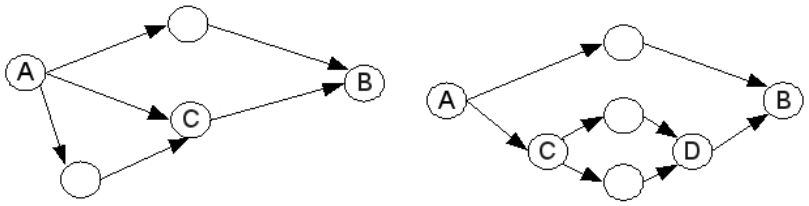
FIGURE 6.5 – Partage de composants appartenant à un seul chemin

même temps avec une donnée de F par rapport à B. Nous aurons par exemple des données provenant de D d'estampilles $\{\langle A,1\rangle\},\{\langle A,3\rangle\}$, des données provenant de C d'estampilles $\{\langle A,1\rangle,\langle B,2\rangle\},\{\langle A,2\rangle,\langle B,2\rangle\}$ et des données provenant de F d'estampilles $\{\langle B,1\rangle\},\{\langle B,3\rangle\}$. L'appariement deux à deux est possible, mais un ensemble cohérent composé des trois entrées est impossible. Le problème d'appariement des données est causé par des pertes de données. En effet, si D et F utilisent toutes les données émises respectivement par A et B, alors il n'existe aucun problème de construction d'ensemble cohérent puisque tout couple de marquage provenant de A et B pourra être apparié avec une donnée marquée par A et une donnée marquée par B. Cependant, sans garantie de ce type, il est également possible que le puits ne puisse jamais construire d'ensemble cohérent dans ce type de configuration. Le comportement des deux fuseaux composant ce système sont donc différents des mêmes fuseaux pris séparément.

Dans la figure 6.5(d), le composant C fait partie d'un fuseau entre B et D tout en étant le puits d'un autre fuseau de source A. Le composant C doit donc gérer la cohérence par rapport à A. Cette attente de cohérence peut filtrer la propagation des données émises par B et empruntant le chemin (B,F,C,D). En effet, les données circulant entre F et C sont influencées à la fois par A et par B. Au niveau de C, elles doivent être en cohérence avec les données provenant de A. Leur propagation peut donc soit être ralentie en attente de cohérence, soit être arrêtée car aucune donnée cohérente avec elle n'est parvenue au puits. Le composant C agit donc comme un filtre pour les données influencées par B. Le comportement du fuseau entre B et D est perturbé par ce filtre, son comportement n'est pas le même que si C était un composant quelconque.

Des observations détaillées dans ce paragraphe, nous pouvons déduire la propriété suivante :

Propriété 7 *Si deux fuseaux partagent plus de deux composants appartenant à un seul chemin tels qu'aucun de ces composants n'est puits d'un des fuseaux, alors leur*



(a) Partage d'un puits et source commune

(b) Partage d'un puits et d'une source

FIGURE 6.6 – Partage d'un puits

comportement est identique au comportement qu'ils ont lorsqu'ils sont analysés séparément.

6.2.2 Filtrage des données par le puits d'un fuseau

Nous nous plaçons maintenant dans le cas exclu dans la propriété précédente : un système où deux fuseaux partagent plus de deux composants appartenant à un seul chemin tels qu'un des composants partagés est puits d'un des fuseaux. Nous distinguons alors deux cas :

- le puits partagé est le puits commun des deux fuseaux ;
- le puits partagé est puits uniquement d'un des deux fuseaux.

Le premier cas correspond au système de la figure 6.5(b) dont le comportement est détaillé dans le paragraphe précédent.

Nous considérons dans ce paragraphe le deuxième cas. Il impose qu'un fuseau comprenne sur un de ses chemins le puits d'un autre fuseau et partage également un autre composant avec ce deuxième fuseau. Si les fuseaux ne partagent qu'un puits, nous sommes dans la situation de la propriété 6. Nous distinguons alors trois cas :

- les fuseaux partagent le puits d'un fuseau et un ou plusieurs composants quelconques ;
- les fuseaux partagent le puits d'un fuseau et ont une source commune ;
- les fuseaux partagent le puits d'un fuseau et la source d'un fuseau.

Le premier cas est celui de la figure 6.5(d) décrit dans le paragraphe précédent. Dans ce système, le composant C agit comme un filtre pour les données influencées par B.

Le cas où les fuseaux partagent un puits et une source commune aboutit à un système où un fuseau est sous-fuseau de l'autre. Sur la figure 6.6(a), le fuseau entre A et B contient un sous-fuseau entre A et C. Les données influencées par A doivent être cohérentes sur C avant de continuer leur propagation vers B. Comme précédemment, le composant C agit comme un filtre. Tout sous-fuseau provoque le même phénomène.

La figure 6.6(b) présente le dernier cas où les fuseaux partagent un puits et une source. Le système résultant présente également un fuseau et son sous-fuseau. Le sous-fuseau entre C et D agit sur la propagation des données émises par A.

Nous obtenons alors la propriété suivante :

Propriété 8 *Si deux fuseaux F_1 et F_2 partagent plus de deux composants appartenant à un seul chemin tels qu'un de ces composants est puits du fuseau F_1, alors ce puits filtre les données propagées par la source de F_2 et change le comportement de F_2.*

La présence d'un puits sur le chemin d'un fuseau entraîne un filtrage des données se propageant à travers ce puits. Si l'on considère l'imbrication de deux fuseaux, ce

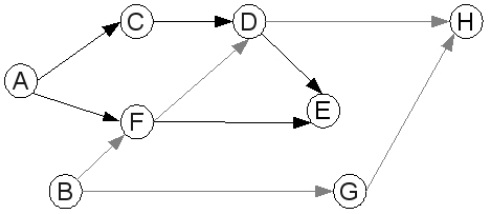


FIGURE 6.7 – Apparition d'un sous-fuseau

puits peut être le puits d'un des deux fuseaux considéré indépendamment, mais il peut également apparaître à cause de l'imbrication. Dans ce cas, un nouveau sous-fuseau est créé avec son propre puits. Nous observons ce phénomène sur la figure 6.7. Les fuseaux considérés indépendamment sont :

- le fuseau F_1 entre A et E de chemins $\{A, C, D, E\}$ et $\{A, F, E\}$
- le fuseau F_2 entre B et H de chemins $\{B, F, D, H\}$ et $\{B, G, H\}$

Initialement, le composant D n'est donc puits d'aucun de ces deux fuseaux. Cependant, l'imbrication crée un sous-fuseau de source A et de puits D. Le composant D doit construire des ensembles cohérents par rapport à A. Il agit donc comme un filtre pour les données influencées par A. Les données influencées par B empruntent le même chemin et sont donc également filtrées. La présence du puits D influence le comportement des fuseaux F_1 et F_2.

De nombreux cas d'imbrication de fuseaux créent un nouveau sous-fuseau et donc un nouveau puits filtrant des données. Il s'agit de cas où les composants partagés sont reliés par un chemin dans un fuseau alors que ce n'est pas le cas dans l'autre (ici F et D). Nous ne détaillons pas tous ces cas menant aux mêmes résultats.

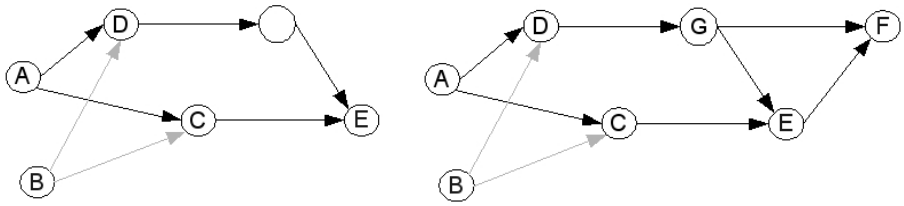
Propriété 9 *Si un fuseau de source C et de puits C' comporte sur un de ses chemins un composant puits d'un autre fuseau, alors ce puits agit comme un filtre sur la propagation des données de la source C jusqu'à C'.*

Dans la première partie de ce chapitre, nous avons détaillé le phénomène de pertes de données dû aux paramètres temporels des composants et à leur liberté d'exécution. Nous constatons ici qu'un autre type de pertes de données peut survenir à cause du filtrage créé par un puits.

6.2.3 Architectures ne garantissant pas la vivacité

Nous avons traité jusqu'ici les cas de partage d'un seul composant et les partages de composants liés par un chemin. Il nous reste à traiter le cas où plusieurs composants sont partagés alors qu'ils ne sont pas liés par un chemin.

L'existence de plusieurs fuseaux dans le système impose le fait que des puits ont à apparier des données dont les estampilles comportent des marquages provenant de plusieurs composants. Nous nous trouvons dans un cas d'appariement impossible si les données dont dispose un puits ne sont jamais compatibles entre elles. Par exemple, prenons le cas où le puits dispose sur une entrée de données portant les estampilles $\{\langle A,1\rangle, \langle B,1\rangle\}$ et $\{\langle A,2\rangle, \langle B,2\rangle\}$ alors que sur son autre entrée il dispose de données portant les estampilles $\{\langle A,1\rangle, \langle B,2\rangle\}$ et $\{\langle A,2\rangle, \langle B,3\rangle\}$. A partir de ces données, il ne peut pas construire d'ensemble cohérent. En effet, si deux données sont compatibles par rapport aux marquages de A, elles ne sont pas compatibles par rapport à B et inversement.



(a) (b)

FIGURE 6.8 – Fuseaux partageant des composants indépendants

Nous obtenons cette situation dans le cas où A et B sont deux sources indépendantes (c'est-à-dire non reliées par un chemin) et qu'elles appartiennent à des fuseaux partageant plusieurs composants indépendants.

La figure 6.8(a) illustre cette configuration. Ce système est composé de deux fuseaux imbriqués. Tous deux sont de puits E mais un est de source A et l'autre de source B. Les composants C et D appartiennent aux deux fuseaux et ne sont pas reliés entre eux. Utilisant des données fournies par A et B, leurs données de sorties portent les marquages de A et B. Le puits E a donc à apparier des données portant des marquages de deux composants. Dans ce système, aucun mécanisme ne contraint C et D à utiliser les mêmes couples de données en entrée. L'exemple d'estampilles incompatibles précédemment décrit peut donc se produire. Il est possible que E ne puisse jamais construire d'ensemble cohérent à partir de ses entrées.

La figure 6.8(b) étend le cas de la figure 6.8(a). Nous considérons cette fois ci le fuseau entre A et F composé des chemins $\{A, D, G, F\}$ et $\{A, C, E, F\}$ et le fuseau entre B et E composé des chemins $\{B, D, G, E\}$ et $\{B, C, E\}$. Ces fuseaux sont de sources indépendantes et ils partagent des composants indépendants. Mais, contrairement à la précédent configuration, ils partagent également deux composants liés par un chemin dans un fuseau mais pas dans l'autre : G et E. La même problématique que précédemment reste sur le composant E. Le composant F, tout comme E, reçoit sur ses deux entrées des données marquées par A et B. Cependant, le comportement des données n'est pas le même. Les données circulant entre G et F portent des marquages de A et B identiques aux marquages des données circulant entre G et E car elles sont émises par le même composant G. Si E est capable de produire des données, alors leurs marquages correspondent à un sous-ensemble des marquages portés par les données provenant de G (E n'utilise pas forcément toutes les données fournies par G). Il en résulte que le composant F reçoit de E et de G des données dont les marquages sont compatibles. Il lui est donc possible de construire des ensembles cohérents. Le fait que F n'ait pas à faire face à une impossibilité de composition d'ensemble cohérent est dû à la présence d'un arc entre G et E. Cet arc impose que toutes les données émises par E soient compatibles avec les données émises par G. Une incompatibilité de données peut survenir sur E alors qu'elle ne se produit pas sur F.

A partir de ces cas, nous pouvons établir la propriété suivante :

Propriété 10 *Un puits peut être dans l'incapacité de construire des ensembles cohérents s'il respecte les trois conditions suivantes :*

- *il est puits de deux fuseaux de sources indépendantes (non liées par un chemin),*
- *ces fuseaux partagent au moins deux composants indépendants,*
- *ces fuseaux ne partagent pas deux composants liés par un chemin dans un fuseau et non dans l'autre.*

6.2.4 Bilan de l'influence de l'architecture

Soit deux fuseaux A et B imbriqués. Nous voulons déterminer si les fuseaux imbriqués conservent le même comportement que lorsqu'ils sont considérés indépendamment :

- Si A et B ne partagent qu'un seul composant : le comportement des fuseaux est inchangé.
- Sinon (partage de plusieurs composants) :
  - Si uniquement partage de composants non puits appartenant à un même chemin : le comportement des fuseaux est inchangé.
  - Sinon :
    - Si le puits de A appartient au fuseau B :
      - Si partage du même puits : Le comportement du puits est modifié, un blocage est possible en cas de pertes de données.
      - Sinon : filtrage par le puits de A des données propagées par B
    - Si apparition de sous-fuseaux : filtrage par le nouveau puits des données propagées par la source du (ou des) fuseaux auquel il appartient.
    - Si les sources de A et B sont indépendantes et qu'ils partagent deux composants indépendants :
      - Si A et B partagent deux composants liés par un chemin dans A et non dans B :
        - Si le puits de A fait partie des composants partagés : blocage possible du puits de A. Filtrage par ce puits des données propagées par B.
        - Sinon : blocage possible d'un nouveau puits qui apparaît. Filtrage par ce puits des données propagées par A et B.
      - Sinon : possible blocage des puits des fuseaux A et B.

6.3 Conclusion

Dans ce chapitre, nous avons analysé comment certaines caractéristiques d'un système peuvent influer sur la vivacité de l'association cohérente des données. Un phénomène de pertes de données peut être provoqué à cause de la grande liberté laissée au système pour s'exécuter ou à des types d'architecture particuliers. Il est alors impossible de garantir la vivacité du système sans introduire une tolérance dans la gestion de la cohérence. Certaines architectures de système peuvent aller jusqu'à provoquer un blocage des puits. Des contraintes contradictoires s'opposent et rendent alors impossible la construction d'ensemble cohérent.

Nous constatons que les problèmes de vivacité décrits dans ce chapitre sont indépendants des tailles de files utilisées en entrée des puits. Certaines configurations de système imposent donc de considérer ces aspects de vivacité en complément de la gestion de la cohérence des données à l'aide de files correctement dimensionnées.

Chapitre 7

Cas d'étude

7.1 Description du cas d'étude

Nous appliquons les résultats de cette thèse sur une mission spatiale développée par Thales Alenia Space : la mission FUEGO. La mission FUEGO est une mission de surveillance de la Terre. Elle est dédiée à la surveillance et à la détection de feux de forêts ou d'éruptions volcaniques. La mission possède deux aspects :

- la surveillance systématique de la surface terrestre pour détecter des départs de feux ou de nouvelles activités volcaniques. Dans le cas d'une détection, une alarme est envoyée au sol.
- l'observation des zones actives sur demande du sol ou après détection à bord d'activité nouvelle. Après observation, les données sont stockées en mémoire puis télédéchargées au sol.

Cette mission est réalisée par une constellation de satellites composée de plusieurs satellites défilant assurant une couverture suffisante des zones sensibles du globe. Des satellites géostationnaires sont utilisés pour relayer les alarmes vers le sol. Des stations sols sont chargées de recevoir les données envoyées par les satellites et d'envoyer de nouvelles requêtes d'observation. Un centre de contrôle dédié s'occupe de vérifier régulièrement l'état des satellites défilants. Les satellites géostationnaires sont utilisés uniquement comme support de communication et leur gestion est réalisé indépendamment de la mission FUEGO.

Pour la surveillance des déclenchements de feux ou d'activité volcanique, chaque satellite utilise un instrument champ large fonctionnant en permanence dans une bande de fréquences appartenant aux infrarouges. Une barrette de détecteurs dont la projection au sol forme une bande perpendiculaire à la trace au sol de 2500 km de large (appelée fauchée) balaie le sol au fur et à mesure de la progression du satellite sur son orbite. L'instrument vise 30 degrés en avant (400 km au sol) ce qui laisse un délai d'une minute entre la détection d'un point chaud non connu (éruption, feu, fumée) à un endroit et le passage du satellite à l'aplomb de cet endroit. Le traitement des données s'effectue à la volée, et les parties intéressantes sont conservées en mémoire afin d'être télédéchargées vers les stations sol.

L'observation des zones désirées s'effectue en utilisant un instrument champ étroit (176 km de fauchée) fonctionnant sur requête. Il vise en dessous du satellite, tandis qu'un miroir orientable lui permet d'atteindre tout point de la zone couverte par l'instrument de détection. Quatre bandes spectrales sont disponibles pour l'observation : visible (VIS), infrarouge proche (NIR), moyen infrarouge (MIR) et infrarouge thermique (TIR). Les bandes spectrales VIS et NIR sont gérées par un même instrument. Les autres bandes ont des instruments dédiés. Les combinaisons utilisées dépendent

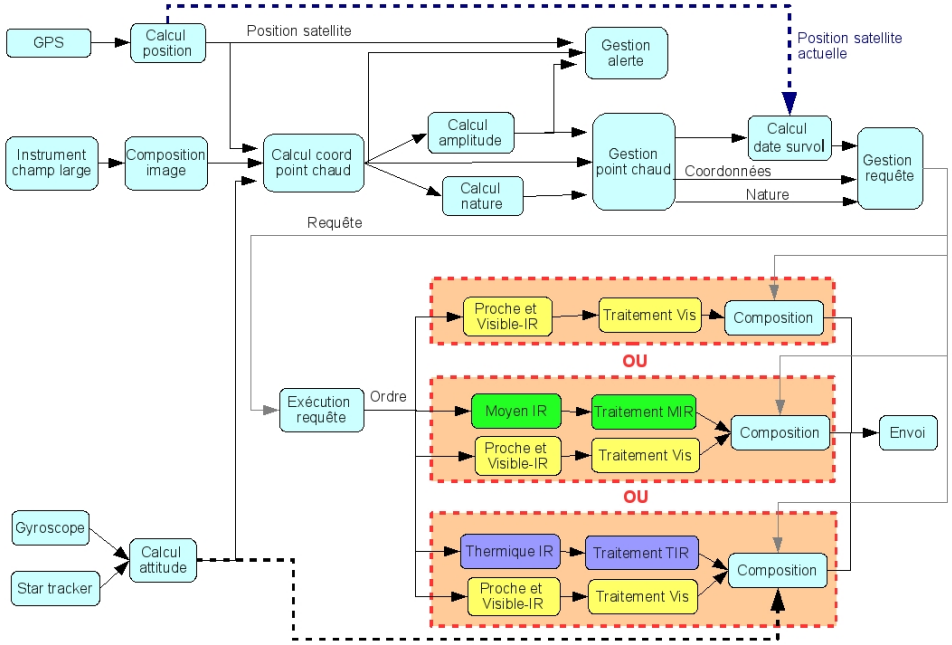


FIGURE 7.1 – Logiciel applicatif d'un satellite de la mission FUEGO

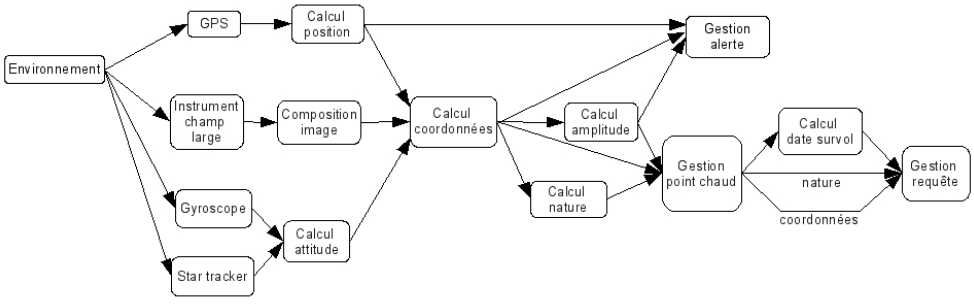


FIGURE 7.2 – Logiciel applicatif lié au champ large

du type d'observation : VIS-NIR pour les volcans, VIS-NIR + MIR pour les incendies et VIS-NIR + TIR pour les fumées.

Lorsqu'une source de chaleur est détectée par l'instrument champ large, la nature de l'objet (volcan/feu/fumée) est aussitôt déterminée. Les catalogues d'objets actifs connus à bord permettent ensuite de savoir s'il s'agit réellement d'un événement nouveau. Si c'est effectivement le cas, les coordonnées géographiques du phénomène sont relevées et envoyées avec sa nature sous forme d'alarme. Une nouvelle zone à observer de priorité maximale est finalement construite, centrée sur le cœur du phénomène.

Chaque satellite défilant connaît à chaque instant un ensemble de zones à observer qui varie suivant les requêtes venant de l'extérieur et les détections à bord. Ces requêtes d'observation sont caractérisées par un intervalle temporel d'activation de l'instrument d'observation (date de début et de fin), un angle du miroir de l'instrument, des canaux (bandes spectrales) à utiliser, et une priorité. Ces paramètres étant préalablement fixés, il est possible que certaines requêtes soient incompatibles entre-elles, notamment si elles se chevauchent temporellement, si la transition de l'une à l'autre est impossible parce que le temps que va prendre le moteur du miroir pour passer de l'angle de la première à celui de la seconde est supérieur au délai qui les sépare ou si les niveaux des batteries ou de la mémoire disponible sont insuffisants. Il faut donc décider de la réalisation de chaque requête d'observation en fonction de tous ces paramètres.

La figure 7.1 représente le diagramme de composants d'un satellite défilant de la mission FUEGO. A partir des données fournies par l'instrument champ large, une image est composée. Sur cette image, un point chaud est recherché. En combinant la position du satellite (calculé à partir du GPS) et son attitude (orientation du satellite, calculée à partir du star tracker et du gyroscope), les coordonnées du point chaud peuvent être déterminées. L'amplitude et la nature de ce point sont calculées et confiées au gestionnaire de point chaud. D'autre part, une alerte est envoyée au sol composée des coordonnées et de l'amplitude du point chaud ainsi que de la position qu'avait le satellite lors de la détection de ce point. La date prévisionnelle de survol du point chaud est calculée et envoyée au gestionnaire de requêtes qui mémorise également la nature et les coordonnées du point. Ce gestionnaire est chargé de déterminer, parmi une liste de requêtes, les prochaines requêtes à accomplir. C'est également cet élément qui reçoit les requêtes provenant du sol. En fonction de la nature du point à observer, différents instruments sont activés. Les trois configurations représentées sont possibles. Une image est composée à partir des observations des différents instruments et elle est envoyée vers le sol.

Les arcs en pointillés représentent des arcs sur lesquels une cohérence des données n'est pas souhaitée. Au contraire, il est par exemple nécessaire d'utiliser la position actuelle du satellite lors du calcul de la date de survol du point chaud.

Nous nous intéressons uniquement à la partie liée à l'observation champ large, c'est-à-dire la partie supérieure du schéma. Nous reprenons uniquement cette partie dans la figure 7.2 faite de manière à bien distinguer les différents fuseaux du système. Les paramètres des composants sont détaillés dans le tableau 7.1. Les temps de communication sont supposés nuls. Ils sont effectivement négligeables par rapport aux autres paramètres du système.

7.2 Analyse du système

Sur le graphe de composants de la figure 7.2 plusieurs fuseaux apparaissent, certains étant imbriqués.

Nom	Période (ms)	Temps d'exécution minimal (ms)	Temps d'exécution maximal (ms)
GPS	1000	100	200
Calcul position	60	20	40
Gestion alerte	1000	50	200
Instrument champ large	100	60	70
Composition image	1000	200	400
Calcul coordonnées	1000	100	500
Calcul amplitude	1000	20	30
Calcul nature	1000	30	40
Gestion point chaud	1000	50	200
Calcul date survol	1000	30	30
Gestion requêtes	1000	50	150
Gyroscope	60	20	30
Star tracker	120	40	60
Calcul attitude	60	20	30

TABLE 7.1 – Paramètres du cas d'application

Nous identifions des fuseaux entre les composants suivants :
- entre l'environnement et le gestionnaire d'alerte (Fuseau F_1). Ce fuseau comprend des sous-fuseaux :
  - entre l'environnement et le calcul de coordonnées (Fuseau F_{11}). Lui même contient un sous-fuseau entre l'environnement et le calcul d'attitude (Fuseau F_{111}).
  - entre calcul position et gestion d'alerte (Fuseau F_{12}). Lui-même contient un sous-fuseau entre calcul coordonnées et gestion d'alerte (Fuseau F_{121}).
- entre le calcul de coordonnées et la gestion point chaud (Fuseau F_2). Ce fuseau est imbriqué avec le fuseau F_{121}. Il est donc imbriqué avec tous les fuseaux englobant ce dernier.
- entre gestion point chaud et gestion requête (Fuseau F_3)

Analysons tout d'abord l'imbrication entre le fuseau F_2 de source *calcul de coordonnées* et de puits *gestion point chaud* et le fuseau F_{121} de source *calcul coordonnées* et de puits *gestion d'alerte*. Ces deux fuseaux partagent la même source, un composant quelconque (*calcul amplitude*) et ont des puits différents. Ces deux fuseaux n'ont donc aucun impact l'un sur l'autre.

Les sous-fuseaux F_{11} et F_{111} ont eux un impact potentiel sur le fuseau F_1. En effet, ils introduisent des puits sur les chemins de F_1 et donc un potentiel filtrage.

7.3 Gestion de la cohérence

Pour déterminer les méthodes à utiliser pour gérer la cohérence des données dans le système, il faut tout d'abord déterminer les critères de cohérence que chaque puits du système doit respecter. Nous n'analyserons pas tous les fuseaux du système mais seulement quelques-uns des plus significatifs.

7.3.1 Fuseau entre l'environnement et le calcul d'attitude (F_{111})

Pour calculer de manière la plus précise possible l'attitude du satellite, il faut disposer des données les plus fraîches possibles provenant du *star tracker* et du gyroscope. Dans ce cas, nous ne voulons pas assurer une quelconque cohérence des données sur le composant de calcul d'attitude. Nous sommes dans le cas où la fraîcheur des données est préférée à la cohérence. Le calcul d'attitude utilise les dernières données reçues.

7.3.2 Fuseau entre *gestion point chaud* et *gestion requête* (F_3)

Le composant de gestion de requête nécessite qu'une cohérence stricte soit appliquée sur ses entrées. En effet, il doit mémoriser un ensemble composé des coordonnées, de la nature et de la date de survol d'un point chaud.

Nous décomposons le fuseau entre *gestion point chaud* et *gestion requête* en trois chemins :
- P_1 = gestion point chaud, calcul date survol, gestion requête
- P_2 = gestion point chaud, gestion requête par le lien "nature"
- P_3 = gestion point chaud, gestion requête par le lien "coordonnées"

Nous simplifierons nos calculs en ne considérant que les chemins P_1 et P_2. Les temps de communication étant les mêmes sur tous les liens, les propriétés temporelles de P_2 sont les mêmes que celles de P_3.

D'après les formules présentées dans ce mémoire, si P est un chemin de taille n :

$$t_{max}(P) = \sum_{i=1}^{n-1} [2T_{C_i} + \Delta_{C_iC_{i+1}}]$$

$$t_{min}(P) = \sum_{i=1}^{n-1} [e_{C_i} + \delta_{C_iC_{i+1}}]$$

Les temps de propagation maximaux sont :
- $t_{max}(P_1) = 2 * 1000 + 2 * 1000 = 4000$
- $t_{max}(P_2) = 2 * 1000 = 2000$

Les temps de propagation minimaux sont :
- $t_{min}(P_1) = 50 + 30 = 80$
- $t_{min}(P_2) = 50$

Nous souhaitons gérer une cohérence stricte en entrée de gestion requêtes. Nous déterminons tout d'abord les entrées nécessitant des files :
- $t_{max}(P_1) \geq t_{min}(P_2)$
- $t_{max}(P_2) \geq t_{min}(P_1)$

Il faut donc des files sur les deux entrées. Nous obtenons leurs tailles d'après le résultat 7 du paragraphe 5.2.2.C.

Entre *calcul date survol* (noté *calcul_d*) et *gestion requête* (noté *gestion_r*), il faut une file de taille N_1 avec,

$$N_1 = \left\lfloor \frac{t_{max}(P_2) - t_{min}(P_1) + 2T_{gestion\_r} - e_{gestion\_r} - e_{calcul\_d}}{T_{calcul\_d}} \right\rfloor + 2$$

$$N_1 = \left\lfloor \frac{2000 - 80 + 2 * 1000 - 50 - 30}{1000} \right\rfloor + 2 = 5$$

Entre *gestion point chaud* (noté *gestion_p*) et *gestion requête*, il faut une file de taille N_2 avec,

$$N_2 = \left\lfloor \frac{t_{max}(P_1) - t_{min}(P_2) + 2T_{gestion\_r} - e_{gestion\_r} - e_{gestion\_p}}{T_{gestion\_p}} \right\rfloor + 2$$

$$N_2 = \left\lfloor \frac{4000 - 50 + 2 * 1000 - 50 - 50}{1000} \right\rfloor + 2 = 7$$

En utilisant ces tailles de files, le composant *gestion requête* peut composer des ensembles strictement cohérent quel que soit l'ordonnancement du système.

Si ces tailles sont considérées comme trop importantes vis à vis des ressources disponibles, alors il est nécessaire de modifier l'architecture du système. Le fuseau est effacé en supprimant les liens transportant les informations de coordonnées et nature des points chauds. Nous faisons transiter ces informations via le composant de calcul de date de survol.

7.3.3 Fuseau entre *calcul coordonnées* et *gestion point chaud* (F_2)

Le fuseau F_2 possède des caratéristiques temporelles proches du fuseau F_3 précédemment analysé. Les périodes et les temps minimaux d'exécution sont comparables.

Nous décomposons le fuseau en trois chemins :
- P_1 = calcul coordonnées, calcul amplitude, gestion point chaud

– P_2 = calcul coordonnées, gestion point chaud
– P_3 = calcul coordonnées, calcul nature, gestion point chaud

Les temps de propagation maximaux sont :
– $t_{max}(P_1) = 2*1000+2*1000 = 4000$
– $t_{max}(P_2) = 2*1000 = 2000$
– $t_{max}(P_3) = 2*1000+2*1000 = 4000$

Les temps de propagation minimaux sont :
– $t_{min}(P_1) = 100+20 = 120$
– $t_{min}(P_2) = 100$
– $t_{min}(P_3) = 100+30 = 130$

Pour la file entre *calcul amplitude* (noté *calcul_a*) et *gestion point chaud* (noté *gestion_p*), nous devons avoir une file de taille N_1 avec :

$$N_1 = \left\lfloor \frac{max(t_{max}[P_j] : P_j \neq P_1) - t_{min}[P_1] + 2T_{gestion\_p} - e_{gestion\_p} - e_{calcul\_a}}{T_{calcul\_a}} \right\rfloor + 2$$

$$N_1 = \left\lfloor \frac{4000 - 120 + 2*1000 - 50 - 20}{1000} \right\rfloor + 2 = 7$$

Pour la file entre *calcul coordonnées* (noté *calcul_c*) et *gestion point chaud*, nous devons avoir une file de taille N_2 avec :

$$N_2 = \left\lfloor \frac{max(t_{max}[P_j] : P_j \neq P_2) - t_{min}[P_2] + 2T_{gestion\_p} - e_{gestion\_p} - e_{calcul\_c}}{T_{calcul\_c}} \right\rfloor + 2$$

$$N_2 = \left\lfloor \frac{4000 - 100 + 2*1000 - 50 - 100}{1000} \right\rfloor + 2 = 7$$

Pour la file entre *calcul nature* (noté *calcul_n*) et *gestion point chaud*, nous devons avoir une file de taille N_3 avec :

$$N_3 = \left\lfloor \frac{max(t_{max}[P_j] : P_j \neq P_3) - t_{min}[P_3] + 2T_{gestion\_p} - e_{gestion\_p} - e_{calcul\_n}}{T_{calcul\_n}} \right\rfloor + 2$$

$$N_3 = \left\lfloor \frac{4000 - 130 + 2*1000 - 50 - 30}{1000} \right\rfloor + 2 = 7$$

En utilisant ces tailles de files, le composant *gestion point chaud* peut composer des ensembles strictement cohérents quel que soit l'ordonnancement du système.

7.3.4 Fuseau entre *calcul position* et *gestion alerte* (F_{12})

La particularité du fuseau entre *calcul position* et *gestion alerte* est une importante différence de période entre un composant communicant avec le puits et le puits lui même. En effet, le composant *calcul position*, qui est de période 60 ms, communique directement avec le puits qui a une période de 1000 ms. De plus, les autres composants ont également des périodes de 1000 ms, ce qui provoque des temps de propagation importants sur les autres chemins du fuseau. Le composant de calcul de position a une fréquence élevée car ses données sont également utilisées afin de contrôler le satellite ou pour calculer une date précise de survol d'un point.

Nous décomposons le fuseau en trois chemins :
– P_1 = calcul position, gestion d'alerte
– P_2 = calcul position, calcul coordonnées, gestion d'alerte
– P_3 = calcul position, calcul coordonnées, calcul amplitude, gestion d'alerte

D'après les formules présentées dans ce mémoire, les temps de propagation maximaux sont :

- $t_{max}(P_1) = 2*60 = 120$
- $t_{max}(P_2) = 2*60+2*1000 = 2120$
- $t_{max}(P_3) = 2*60+2*1000+2*1000 = 4120$

Les temps de propagation minimaux sont :

- $t_{min}(P_1) = 20$
- $t_{min}(P_2) = 20+100 = 120$
- $t_{min}(P_3) = 20+100+20 = 140$

Si nous souhaitons gérer une cohérence stricte en entrée du gestionnaire d'alerte, il est nécessaire d'utiliser entre *calcul position* (noté *calcul_p*) et *gestion d'alerte* (noté *gestion_a*) une file de taille N_1 avec,

$$N_1 = \left\lfloor \frac{max(t_{max}[P_j] : P_j \neq P_1) - t_{min}[P_1] + 2T_{gestion\_a} - e_{gestion\_a} - e_{calcul\_p}}{T_{calcul\_p}} \right\rfloor + 2$$

$$N_1 = \left\lfloor \frac{4120 - 20 + 2*1000 - 50 - 20}{60} \right\rfloor + 2 = 102$$

Nous obtenons sans surprise une taille de file très importante. Cela traduit l'important déséquilibre entre les temps de propagation des chemins du fuseau mais également l'important rapport entre la période du composant de calcul de position et celle de gestion d'alerte.

En réalité, l'utilisateur n'a pas besoin d'une cohérence stricte dans ce fuseau. La position envoyée avec l'alarme n'a pas besoin d'être précise à 60 ms près. Nous tolérons un écart de cohérence de 300 ms. Il est alors possible d'utiliser une file filtrante entre calcul de position et gestion d'alerte.

Nous cherchons tout d'abord à déterminer le taux d'enregistrement R de la file. D'après nos travaux,

$$1 \leq R \leq \frac{2\tau - t_{max}(P_1) + t_{min}(P_1) + \Delta - \delta}{T_{calcul\_p}} + 1$$

$$1 \leq R \leq \frac{2*300 - 120 + 20}{60} + 1$$

$$1 \leq R \leq 9,33$$

Nous retenons un taux d'enregistrement $R = 9$, c'est à dire que la file n'enregistre qu'une données sur 9 reçues. Calculons maintenant la taille N de la file filtrante

$$N = \left\lceil \frac{t_{max}(P_3) - \tau - t_{min}(P_1) - e_{calcul\_p} + (R+1)T_{calcul\_p} + 2T_{gestion\_a} - e_{gestion\_a}}{RT_{calcul\_p}} \right\rceil$$

$$N = \left\lceil \frac{4120 - 300 - 20 - 20 + 10*60 + 2*1000 - 50}{8*60} \right\rceil = 12$$

Dans le cadre d'une cohérence relâchée de tolérance 300 ms, nous avons considérablement réduit la taille de la file, passant de 102 à 12. Nous remarquons l'efficacité du principe de file filtrante et de cohérence relâchée pour réduire la taille des files.

7.4 Conception d'un outil d'analyse et d'un simulateur d'exécution

Afin d'automatiser l'analyse des systèmes, un premier outil a été développé avec l'aide d'un stagiaire. Ce logiciel permet d'éditer un graphe de composants et de définir leurs paramètres temporels. A partir de ces informations, les fuseaux sont détectés dans le graphe et peuvent être visualisés à l'écran. En utilisant les paramètres temporels des composants, les tailles de file nécessaires en entrée de chaque puits sont calculées afin de gérer la cohérence des données. Seule la cohérence stricte a été implémentée, l'ajout de calcul pour la cohérence relâchée serait facilement réalisable.

Dans un deuxième temps, un simulateur d'exécution a été développé. Il permet de simuler l'exécution d'un système comportant des fuseaux avec des contraintes de cohérence sur les puits. Une cohérence stricte ou relâchée peut être demandée. Les paramètres temporels des composants ainsi que la taille de leurs files d'attente sont définis. Nous obtenons alors, pour une exécution donnée, le nombre d'ensembles cohérents différents qu'un puits a réalisé sur ces entrées. Cela nous permet de mesurer la vivacité de ce puits et de vérifier que les tailles des files sont bien suffisantes. Dans l'état actuel, le simulateur produit des exécutions dans lesquelles les composants s'exécutent à date fixe par rapport au début de leur période. Il est facilement envisageable de modifier le programme pour qu'il permette des exécution plus aléatoires. L'utilisation du simulateur nous permet de vérifier, par le test, les formules élaborées dans cette thèse.

Chapitre 8

Conclusion et perspectives

Pour la description d'un système complexe, une modélisation par composant est adaptée. Cependant, une telle modélisation fait apparaître de multiples chemins de données au sein du graphe de composants. Il est alors nécessaire d'introduire une coordination entre ces différents chemins afin que les données soient cohérentes les unes vis-à-vis des autres. Une notion de cohérence que nous jugeons adaptée a tout d'abord été décrite et formalisée, puis nous avons proposé plusieurs méthodes pour la gérer. Cette gestion se base sur une mémorisation de données qui sont conservées jusqu'à ce qu'elles puissent être utilisées de façon cohérente. Nous avons déterminé que la capacité de mémorisation nécessaire est bornée. Plusieurs aspects de la cohérence ont été explorés et notamment la cohérence relâchée permettant d'alléger les contraintes sur le système et de réduire les espaces de mémorisation. Tous ces travaux ont été inspirés du domaine spatial et appliqués sur un cas d'étude provenant de ce domaine. L'originalité de notre approche a été d'identifier l'existence de ce problème de cohérence et de l'étudier en se détachant le plus possible de l'ordonnancement des composants, ce qui est inhabituel dans un contexte temps réel.

Différentes perspectives sont envisageables. En premier lieu, nous n'avons traité dans ce mémoire que des systèmes composés de composants périodiques et l'intégration de composants apériodiques est une extension possible de nos travaux. Dans l'industrie spatiale, lors des phases de simulation ou de test, les tâches apériodiques sont ramenées à des tâches périodiques dont la période correspond au pire cas de déclenchement des évènements apériodiques. Il serait intéressant d'analyser si, pour traiter la gestion de la cohérence, nous pouvons adopter la même approche.

Par ailleurs, nous considérons des architectures composées de composants élémentaires. Pour aller jusqu'au bout d'une approche par composant, les composants doivent pouvoir être fusionnés pour n'en composer qu'un seul de plus haut niveau. Nous pouvons par exemple représenter un fuseau par un unique composant ayant de nouvelles propriétés. Cependant, nous avons montré que le puits d'un fuseau a un comportement non comparable à un composant quelconque. En conséquence, les données émises en sortie d'un fuseau en fonction des données utilisées en entrée de ce fuseau ont des propriétés particulières. Les paramètres qu'aurait un composant représentant un fuseau ne sont donc pas évidentes. Le principe est d'autant moins évident s'il existe des fuseaux imbriqués dans le système.

D'autre part, nous avons adopté un modèle d'exécution laissant l'ordonnancement des composants relativement libre. Cependant, nous avons constaté dans le chapitre 6 que l'ajout de contraintes à l'ordonnancement du système peut améliorer ses capacités à construire des ensembles cohérents. Il lui est notamment plus facile de gérer des cohérences de données dont la tolérance de cohérence est faible. Une piste plus clas-

sique reste alors à explorer : quels critères d'ordonnancement permettent de faciliter la gestion de la cohérence des données ? Nous pouvons alors adopter deux approches symétriques. D'une part, nous pouvons prendre en compte des contraintes particulières d'ordonnancement pour analyser leurs effets sur la gestion de la cohérence. Par exemple, de manière similaire à ce que nous avons présenté section 6.1.3, si les dates d'activation des composants sont régulières, les tailles des files nécessaires sont réduites. D'autre part, nous pouvons produire des indications sur l'ordonnancement à adopter dans un système pour gérer au mieux la cohérence des données. Par exemple, nous pouvons imposer un séquencement d'exécution des composants compatible avec leurs positions dans un fuseau, de manière à régulariser les temps de propagation des données le long des chemins.

Enfin, nous pouvons considérer la problématique de l'adaptation dynamique du système provoquée par un changement des besoins de l'application mais aussi par une évolution des ressources. Actuellement, les tolérances de cohérence des différents puits du système sont définies à la conception et restent fixes durant l'exécution du système. Une adaptation dynamique de ces tolérances est possible au cours de l'exécution du système en fonction des ressources disponibles, de la surcharge du bus de communication, de changement de modes du système... Les tolérances peuvent s'adapter aux besoins de l'application : un puits peut avoir à gérer une cohérence précise dans un certain mode alors qu'il lui est demandé une cohérence plus lâche dans un autre. Les tolérances peuvent également s'adapter aux ressources et contraintes du système : un comportement dégradé du bus, pour cause de surcharge ou d'économie de ressources, peut rendre impossible la gestion d'une cohérence précise car toutes les données ne sont pas transmises. Ce comportement doit donc être répercuté sur les composants afin qu'ils soient capables de disposer d'ensembles cohérents relâchés sur leurs entrées, si ce comportement a un sens. Cette évolutivité du système pose plusieurs problèmes. D'une part, il faut disposer d'un espace de stockage adapté à toutes les configurations. D'autre part, le passage d'une configuration à une autre peut être délicat et nécessiter une période de transition. Par exemple, si nous passons d'une gestion de cohérence relâchée avec file filtrante à une cohérence stricte, le composant ne peut pas immédiatement composer d'ensembles strictement cohérents avec les données dont il dispose. Il doit attendre de recevoir de nouvelles données. Dans la même optique d'adaptation, nous pouvons considérer un système évolutif dans lequel des composants peuvent être ajoutés ou supprimés et des liens entre composants peuvent être modifiés. La détection des fuseaux et la mise en place des files doivent alors être réalisés dynamiquement à l'exécution, et les phases de transition entre les différentes configurations doivent être traitées.

Glossaire

AADL	Architecture Analysis and Design Language
ADL	Architecture Design Language
CCM	Corba Component Model
CNES	Centre National d'Etudes Spatiales
CORBA	Common Object Request Broker Architecture
DRE	Distributed Real Time Embedded
FIFO	First In First Out
GALS	Globally Asynchronous Locally Synchronous
IDL	Interface Description Language
KPN	Kahn Process Network
LTTA	Loosely Time-Triggered Architecture
MIR	Mid InfraRed
NIR	Near InfraRed
PPS	Pulse Per Second
PTIDES	Programmong Temporal Integrated Distributed Embedded Systems
QoS	Quality of Service
SDF	Synchronous Data Flow
TDMA	Time-Division Multiple Access
TIR	Thermal InfraRed
TTA	Time-Triggered Architecture
TTP	Time-Triggered Protocol
VIS	Visible

Bibliographie

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9) :125–141, 1990.

[2] Gul Agha. Abstracting interaction patterns : A programming paradigm for open distributed systems. In *Formal Methods for Open Object-Based Distributed Systems*, 1997.

[3] Gul Agha, Svend Frølund, Wooyoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology : Systems and Applications*, 1 :3–21, 1993.

[4] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1) :1–72, 1997.

[5] Yamine Ait Ameur, Bruno D'Ausbourg, Frédéric Boniol, and Virginie Wiels. A component based methodology for description of complex systems. an application to avionics systems. In *4th European Systems Engineering Conference (EuSEC)*, 2002.

[6] S. Anderson and J. Küster Filipe. Guaranteeing temporal validity with a real-time logic of knowledge. In *Proceedings of the 23rd Int'l Conference on Distributed Computing Systems (ICDCS '03)*, pages 178–183. IEEE Computer Society, 2003.

[7] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2) :42–49, 1982.

[8] John A.Sharp. *Data flow computing : theory and practice*. Ablex Publishing Corp., 1992.

[9] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal. Efficient Δ-causal broadcasting. *International Journal of Computer Systems Science and Engineering*, 13(5) :263–269, 1998.

[10] S. Baruah, S. Goddard, and K. Jeffay. Feasibility concerns in PGM graphs with bounded buffers. In *ICECCS '97 : Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems*, pages 130–139. IEEE Computer Society, 1997.

[11] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64– 83, Jan 2003.

[12] A. Benveniste, P. Caspi, M. Di Natale, C. Pinello, A. Sangiovanni Vincentelli, and S. Tripakis. Loosely time-triggered architecture based on communication-by-sampling. Publication interne Irisa, 2007.

[13] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for loosely time-triggered architectures. In *EMSOFT '02 : Proceedings of the Second International Conference on Embedded Software*, pages 252–265. Springer-Verlag, 2002.

[14] Albert Benveniste, Paul Guernic, and Christian Jacquemot. Synchronous programming with events and relations : the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2) :103–149, 1991.

[15] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. Research Report RR-0842, INRIA, 1988.

[16] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *J. VLSI Signal Process. Syst.*, 21(2) :151–166, 1999.

[17] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4) :445–457, 1998.

[18] K. M. Chandy and J. Misra. Distributed simulation : A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, SE-5(5) :440–452, 1979.

[19] D. M. Chapiro. *Globally-asynchronous locally-synchronous systems*. PhD thesis, Stanford University, CA., 1984.

[20] Vilgot Claesson and Neeraj Suri. TTET : Event-triggered channels on a time-triggered base. In *IEEE International Conference on Engineering of Complex Computer Systems*, pages 39–46, Los Alamitos, CA, USA, 2004.

[21] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Science*, pages 511–523, 1971.

[22] Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti). Le modèle de composants CORBA. Technical report, ACCORD, 2002.

[23] Ministère de l'économie des finances et de l'industrie. Technologies clés 2010. Technical report, Ministère de l'économie, des finances et de l'industrie, 2006.

[24] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376. Springer-Verlag, 1974.

[25] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *Proceedings of the International Symposium on Theoretical Programming*, pages 187–216. Springer-Verlag, 1974.

[26] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL) : An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

[27] C. Fong. Discrete-time dataflow models for visual simulation in Ptolemy II. Master's thesis, Electronics Research Laboratory, University of California, Berkeley, 2001.

[28] Steve Goddard and Kevin Jeffray. Analizing the real-time properties of a U.S. navy signal processing system. In *fourth IEEE International symposium on high assurance systems engineering*, pages 141–150, nov 1999.

[29] T. Gustafsson and J. Hansson. Data freshness and overload handling in embedded systems. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 173–182. IEEE Computer Society, 2006.

[30] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[31] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto : a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, 2003.

[32] Maurice P. Herlihy and Jeannette M. Wing. Linearizability : a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12 :463–492, 1990.

[33] Abhay Kumar Jha, Ming Xiong, and Krithi Ramamritham. Mutual consistency in real-time databases. In *Real-Time Systems Symposium (RTSS)*, pages 335–343, 2006.

[34] Ning Jia, Ye-Qiong Song, and Françoise Simonot Lion. Graceful degradation of the quality of control through data drop policy. In *European Control Conference (ECC 2007)*, jul 2007.

[35] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74 : Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[36] Gilles Kahn and David B. Macqueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998. North Holland Publishing Company, 1977.

[37] Richard M. Karp and Rayamond E. Miller. Properties of a model for parallel computations : Determinacy, termination, queueing. *SIAM Journal*, 14 :1390–1411, nov 1966.

[38] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Jounal of Computer and System Sciences*, 3(2) :147–195, 1969.

[39] Mohamed Khalgui, Xavier Rebeuf, and Françoise Simonot Lion. Component based deployment of industrial control systems : a hybrid scheduling approach. In *11th IEEE International Conference on emerging Technologies and Factory Automation*, pages 1293–1300, sept 2006.

[40] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101. Springer-Verlag, 1991.

[41] Hermann Kopetz, Günther Bauer, and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 1988.

[42] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Transactions on Computers*, 28(9) :690–691, 1979.

[43] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, jul 1978.

[44] E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Invited Paper in Foundations of Interface Technologies (FIT)*. Satellite event to the 16th International Conference on Concurrency Theory, 2005.

[45] Edward A. Lee and Eleftherios Matsikoudis. *The Semantics of Dataflow with Firing*. Cambridge University Press, 2007. Chapter from "From Semantics to Computer Science : Essays in memory of Gilles Kahn".

[46] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1) :24–35, 1987.

[47] Edward A. Lee and Thomas Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–799, 1995.

[48] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[49] F. Mattern. Virtual time and global state in distributed systems. In *Int'l Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1989.

[50] Mokhoo Mbobi and Frédéric Boulanger. Le paradigme acteur dans la modélisation des systèmes embarqués. In *IEEE Canadian Conference on Electrical and Computer Engineering*, pages 418–421, Ottawa, Canada, 2006.

[51] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin. Evaluation of component technologies with respect to industrial requirements. In *30th EUROMICRO conference*, pages 56–63. IEEE Computer Society, 2004.

[52] Medvidovic Nenad and Richard N.Taylor. A classification and comparaison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1) :70–93, 2000.

[53] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4) :631–653, 1979.

[54] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, Berkeley, CA, USA, 1995.

[55] James L. Peterson. Petri nets. *ACM Computer Surveys*, 9(3) :223–252, 1977.

[56] D. Pilone. UML 2.0. O'Reilly, 2005.

[57] Nadège Pontisso, Gérard Padiou, and Philippe Quéinnec. Real time data consistency in component based embedded systems. In *8th International Conference on New Technologies in Distributed Systems (NOTERE '08)*, pages 1–6, Lyon, 2008. ACM.

[58] Nadège Pontisso, Philippe Quéinnec, and Gérard Padiou. Temporal data matching in component based real time systems. In *IEEE Industrial Symposium on Industrial Embedded System (SIES 2009)*, pages 62–65, Lausanne, Suisse, 2009. IEEE Computer Society.

[59] Nadège Pontisso, Philippe Quéinnec, Gérard Padiou, Guillaume Veran, and Gérald Garcia. Data consistency in a component based space system. In *Data System In Aerospace (DASIA)*, pages 277–280, Istanbul, Turquie, 2009. ESA.

[60] K. Ramamritham, S. H. Son, and L. Cingiser DiPippo. Real-time databases and data services. *Real-Time Systems*, 28(2-3) :179–215, 2004.

[61] G. B. Shippen and J. K. Archibald. A tagged token dataflow machine for computing small, iterative algorithms. *SIGARCH Computer Architecture News*, 15(6) :9–18, 1987.

[62] Xiohui Song and Jane W.-S. Liu. Maintaining temporal consistency : Pessimistic vs. optimitic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5) :786–796, 1995.

[63] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[64] T. Tachikawa, H. Higaki, and M. Takizawa. Delta-causality and ϵ-delivery for wide-area group communications. *Computer Communications*, 23(1) :13–21, 2000.

[65] Francisco Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Lifetime based consistency protocols for distributed objects. In *Proc. 12th International Symposium on Distributed Computing, DISC'98*, pages 378–392. Springer-Verlag, 1998.

[66] Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. Timed consistency for shared distributed objects. In *PODC '99 : Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 163–172. ACM, 1999.

[67] B. Urgaonkar, A. G. Ninan, M. S. Raunak, P. J. Shenoy, and K. Ramamritham. Maintaining mutual consistency for cached web objects. In *21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 371–380. IEEE Computer Society, 2001.

[68] Thomas Vergnaud, Laurent Pautet, and Fabrice Kordon. Using the AADL to describe distributed applications from middleware to software components. In *10th Ada-Europe International Conference on Reliable Software Technologies*, volume 3555 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2005.

[69] M. Xiong, S. Han, and K. Lam. A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In *Real-Time Systems Symposium (RTSS)*, pages 27–37. IEEE Computer Society, 2005.

[70] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints : Exploiting data semantics. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RSS'96)*, pages 240–253, 1996.

[71] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Iyengar. Engineering web cache consistency. *ACM Transactions on Internet Technology*, 2(3) :224–259, 2002.

[72] Yang Zhao, Jie Liu, and Edward A.Lee. A programming model for time-synchronized distributed real-time systems. *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '07)*, pages 259–268, 2007.