Introduction

I.1. Préambule

Bob était tout songeur en s'engouffrant dans la bouche de métro. Voilà deux jours qu'il n'avait pas parlé à Alice. Sa colère s'était peu à peu transformée en amertume, puis en culpabilité. Il comprenait maintenant que sa réaction initiale avait été exagérée, et désirait plus que tout faire la paix avec son amie d'enfance, mais ne savait pas comment l'aborder. Aussi, quel ne fut pas son soulagement lorsqu'il reçut un message d'elle l'invitant à sortir boire un café. Il s'empressa de répondre alors que la rame s'avançait dans le tunnel sombre. Hélas, au plus profond de la ville, le réseau défaillant ne put transmettre son « Évidemment » libérateur.

Alice aimait Bob comme un frère, malgré l'anxiété chronique et le caractère ombrageux de celui qu'elle voyait comme son plus ancien ami. Deux jours plus tôt, il était parti dans une colère noire pour un simple quiproquo et elle n'avait pas pu le raisonner depuis. Même maintenant, la réconciliation semblait difficile : Bob venait d'ignorer purement et simplement son invitation à aller boire un café. Elle tenta une nouvelle stratégie et envoya « tu n'as pas répondu », suivi quelques secondes plus tard d'un humble « tu m'en veux ? ».

Aussi étrange que cela puisse paraître, à partir de ce moment, le destin d'Alice et de Bob dépend énormément du service de messagerie instantanée choisi pour communiquer. Nous avons tenté de reproduire leur conversation avec trois services de messagerie instantanée grand public : Google Hangouts[1] (Hangouts), WhatsApp Messenger[2] (WhatsApp) et Microsoft Skype[3] (Skype). L'expérience a été réalisée avec Matoula Petrolia dans le rôle d'Alice et moi-même dans le rôle de Bob. La

1. www.google.fr/hangouts/
2. www.whatsapp.com/
3. www.skype.com/fr/

perte de connexion de Bob a été modélisée par l'utilisation du mode avion des téléphones. Pour chaque expérience, nous présentons une capture d'écran prise avant reconnexion et une prise après reconnexion pour les deux interlocuteurs. Les trois services de messagerie présentent un comportement très différent face à la même situation. Observons l'impact de ces différences sur les relations entre Bob et Alice dans ce moment de mésentente.

Hangouts (figure I.1). Bob reçut les messages d'Alice en sortant du métro. Il les regarda distraitement en se disant que son propre message avait dû mettre du temps à lui parvenir. Ce n'est qu'en fin d'après-midi, alors qu'il s'apprêtait à préciser l'heure et le lieu du rendez-vous, qu'il remarque le message d'erreur écrit en petites lettres en dessous de sa réponse : « Le message n'a pas été envoyé. Veuillez appuyer pour réessayer. » Il se résigna à appeler Alice pour mettre les choses au clair.

WhatsApp (figure I.2). La réponse de Bob fit à Alice l'effet d'une douche froide : « Évidemment » ! Non seulement il lui en voulait, mais il ne se gênait pas pour le dire le plus sèchement possible. Bob lui demanda plus tard à quelle heure elle voulait le voir. Elle ne comprit pas tout de suite mais sauta sur l'occasion pour accepter. Ce n'est que le soir, quand Bob lui montra son propre fil de messages, qu'elle comprit ce qui s'était passé : leurs messages s'étaient croisés et chacun avait reçu le message de l'autre après avoir envoyé le sien. Bob ne pouvait donc pas savoir qu'elle avait mal interprété son message.

Skype (figure I.3). La deuxième stratégie d'Alice fut aussi infructueuse que la première. Elle eut beau regarder régulièrement son téléphone toute la journée, aucun message de Bob ne vint se placer après les siens sur son fil de messages. Elle se sentit toute bête quand elle réalisa enfin son erreur : elle n'avait pas vu la réponse favorable que Bob lui avait envoyée avant même qu'elle ne lui envoie son deuxième message. Elle s'empressa de se confondre en excuses et de mettre au point une rencontre avec Bob.

À la lumière de ces trois scénarios, on voit que des compromis sont inévitables en cas de déconnexion temporaire, que ce soit un message d'erreur (Hangouts), la présentation d'un état différent aux interlocuteurs (WhatsApp), ou un réordonnancement de messages (Skype). Chaque stratégie a ses propres qualités mais aucune n'est exempte de défauts. Les services de messagerie instantanée ont vocation à être utilisés par des êtres humains, capables de s'adapter à beaucoup de situations. Les programmes, eux, le sont beaucoup moins, et les conséquences d'incohérences imprévues dans des données partagées par des ordinateurs peuvent être beaucoup plus graves. Avoir un moyen efficace de modéliser précisément les différents types d'incohérence qui peuvent survenir est donc un enjeu très important.

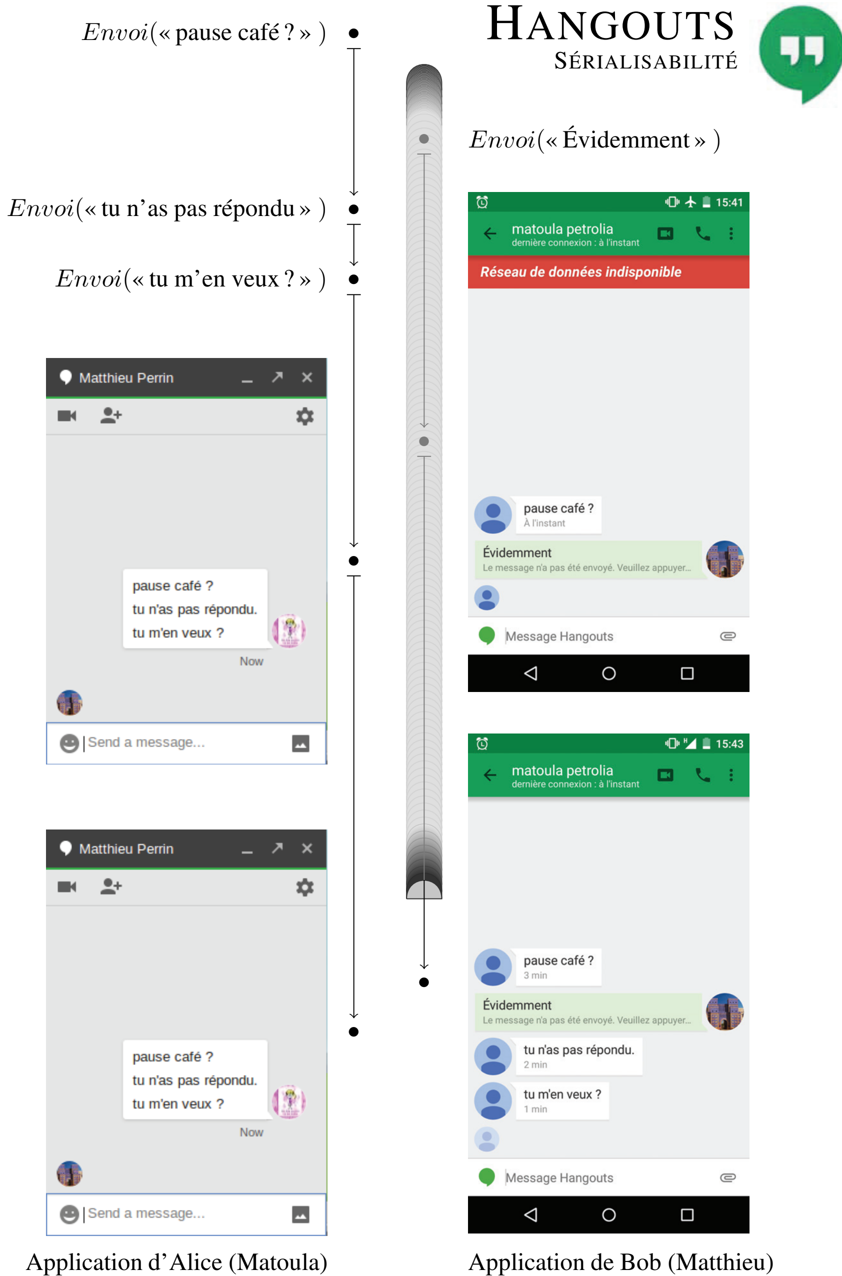


Figure I.1. *Comportement de Google Hangouts après une déconnexion*

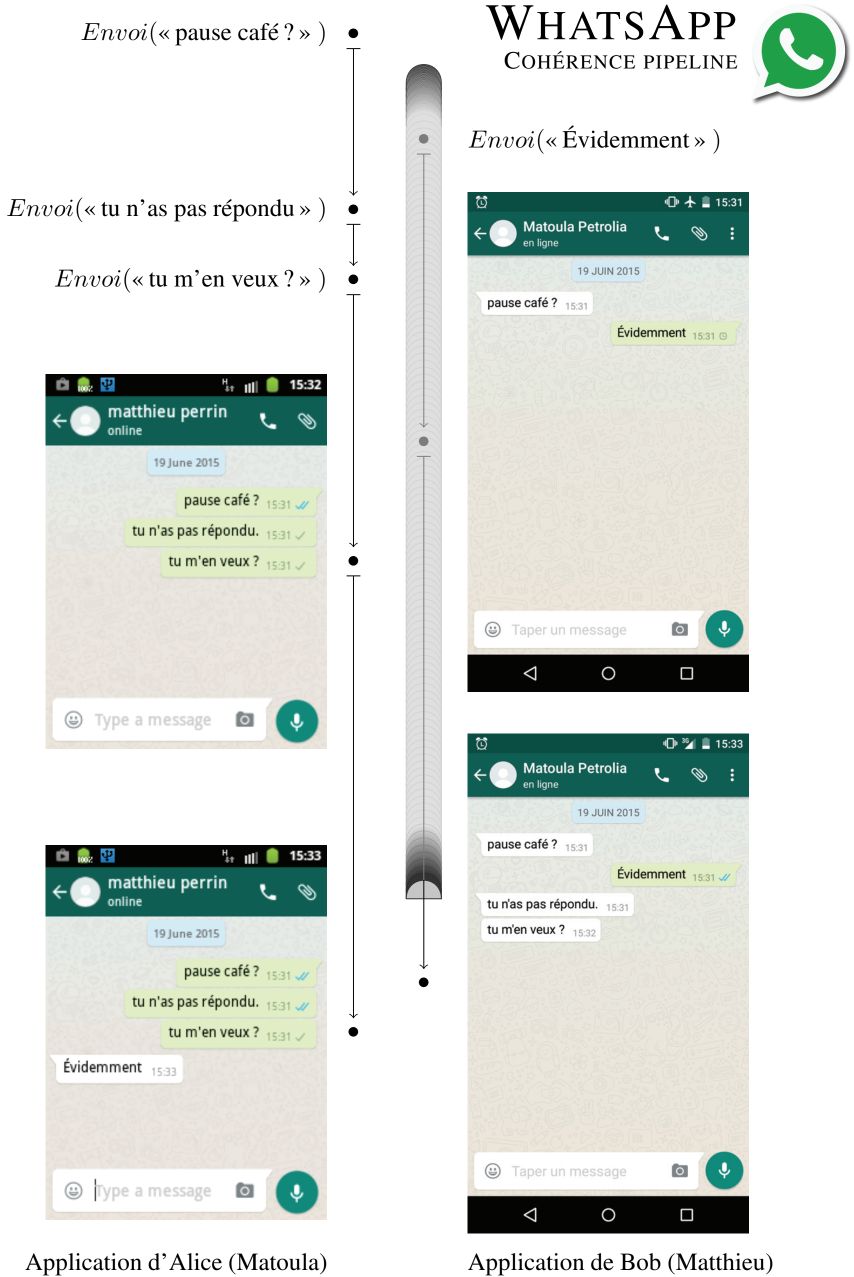


Figure I.2. *Comportement de WhatsApp Messenger après une déconnexion*

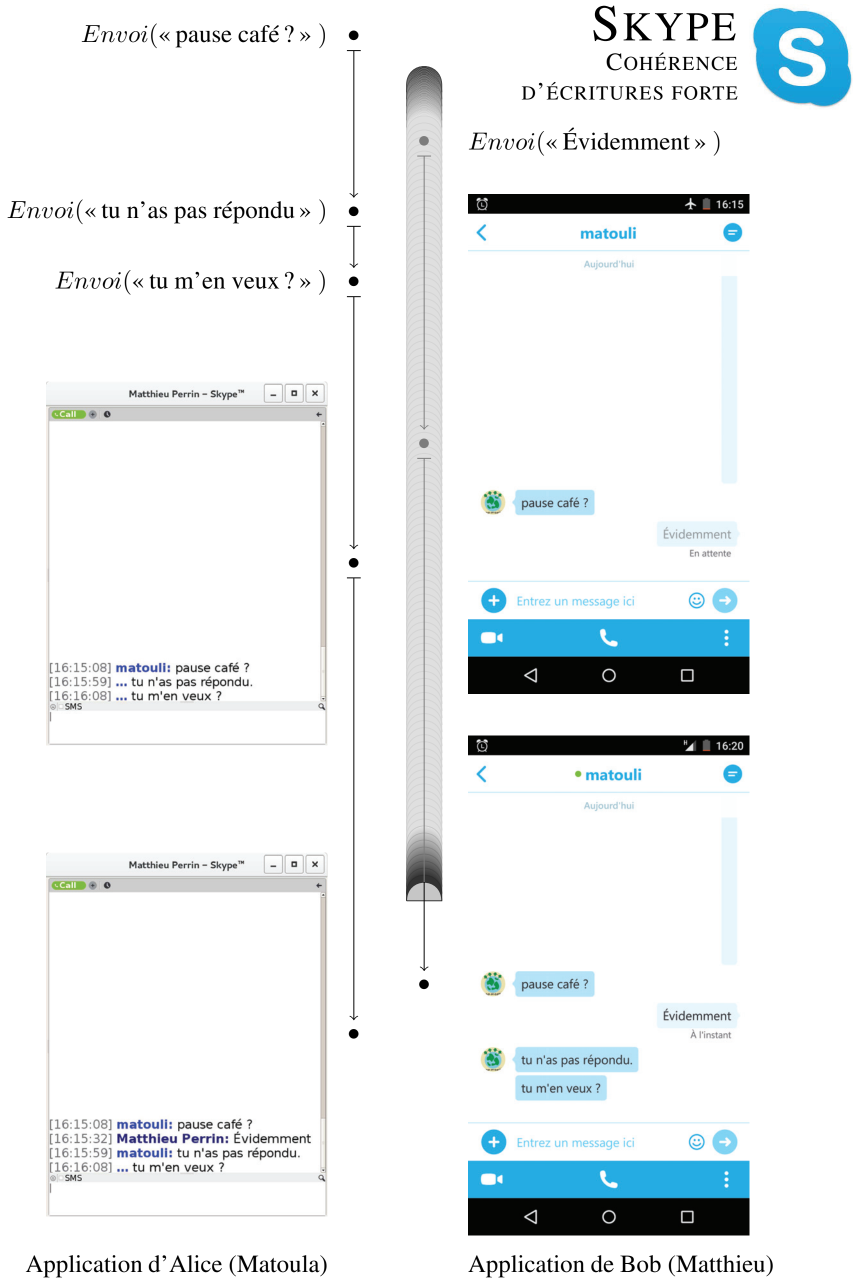


Figure I.3. *Comportement de Microsoft Skype après une déconnexion*

I.2. Systèmes répartis et concurrence

DÉFINITION I.1.– Un *système réparti* est une collection d'entités de calcul autonomes connectées en vue d'accomplir une tâche commune.

Trois éléments, détaillés ci-dessous, sont importants dans cette définition.

Entité de calcul. Par *entité de calcul*, on entend des entités suffisamment complexes pour faire leurs propres choix, soit selon leur volonté propre, pour des êtres humains voire des animaux, soit en fonction du résultat d'un calcul complexe pour les processus d'un ordinateur, voire en réaction à des stimuli extérieurs dans le cas de réseaux de capteurs. On utilisera génériquement le terme de *processus* pour nommer ces entités de calcul, même dans le cas où elles ne correspondent pas à l'exécution d'un programme par un ordinateur.

Dans les exemples précédents, Alice et Bob sont les deux entités de calcul. Au contraire, un système stellaire composé d'une étoile et de plusieurs planètes ne sera pas considéré comme un système réparti, car aucune entité le constituant n'est capable de décision.

Connexion en vue d'accomplir une tâche commune. Pour être considérés comme un système réparti, des processus doivent avoir besoin de communiquer. Par exemple, deux processus exécutant des programmes différents sur un même ordinateur (comme un lecteur de musique et un éditeur de texte) n'ont pas de tâche commune à accomplir. Au contraire, dans l'exemple précédent, Alice et Bob cherchent à résoudre un différend, ce qui forme leur tâche commune. Pour cela, ils utilisent un service de messagerie qui leur permet de se connecter.

Notons que l'exécution d'une *tâche commune* ne signifie pas forcément que les processus ont un intérêt en commun. Les raisons d'être des systèmes répartis sont nombreuses. Il peut s'agir de *collaboration* (édition de documents, orchestration de sites web, etc.) ou au contraire de *compétition* (jeux en réseau, transactions à haute fréquence sur les marchés financiers, etc.), voire de simple *communication* (services de messagerie, sites web, partage de fichier en pair à pair, etc.) entre entités indépendantes, généralement humaines. Enfin, la *réplication* peut être utilisée pour résister aux défaillances : si une machine tombe en panne, d'autres sont disponibles pour prendre le relais.

Autonomie. Dans les exemples précédents, Alice et Bob sont les données du système. Ils existaient avant de chercher à communiquer, et ils existeront encore après. Dans l'analyse des systèmes répartis, on s'intéresse aux problèmes que peuvent avoir les processus pour accomplir leur tâche commune. Le fait qu'ils sont autonomes signifie que leur exécution ne dépend que d'eux-mêmes. En particulier, il ne peut exister un processus maître parmi eux qui décide quand et comment les autres peuvent s'exécuter.

L'autonomie des processus est ce qui différencie les systèmes *répartis* et les systèmes *parallèles*. Dans un système parallèle, le but est d'agréger de la puissance de calcul pour accomplir une tâche particulièrement complexe, comme la simulation de phénomènes physiques ou biologiques. Le programme exécuté par les processus est connu à l'avance et son exécution n'est censée dépendre que des paramètres initiaux. Cependant, un système parallèle peut mettre en œuvre des techniques propres aux systèmes répartis, en particulier pour synchroniser les processus. La différence entre système parallèle et réparti est donc plus une question de point de vue que de réalité du système.

Le monde qui nous entoure est intrinsèquement un système réparti composé d'individus indépendants forcés de communiquer dans leurs tâches quotidiennes, pour élire leurs représentants, éviter les collisions sur la route, etc. Surtout, les systèmes *informatiques* répartis, dans lesquels le comportement des processus est régi par des programmes contrôlables, ont récemment envahi nos vies. En 2015, le monde a dépassé les trois milliards d'internautes[4], le moindre processeur de téléphone portable est aujourd'hui multicœur, l'informatique décentralisée *en nuage* (*cloud computing*) commence à se démocratiser avec l'explosion des services qu'elle offre, la géoréplication est largement utilisée pour garantir la persistance des données sensibles...

Malgré toutes les applications envisageables aux systèmes répartis, les applications réparties sont généralement beaucoup plus difficiles à concevoir que leurs pendants séquentiels : alors que les événements d'un environnement séquentiel se produisent les uns après les autres dans un ordre maîtrisé par le programmeur, la notion de temps entre les événements se produisant dans un système réparti est beaucoup plus floue. Dans l'exemple d'Alice et Bob, il n'est pas clair si Alice a envoyé ses deux derniers messages *avant* ou *après* Bob, d'où les différences de comportement observées dans cette situation pour les trois services de messagerie instantanée. Cette situation, propre aux systèmes répartis, où les événements ne sont pas totalement ordonnés, porte un nom : la concurrence.

DÉFINITION I.2.– Un système est dit *concurrent* si son comportement dépend de l'ordre dans lequel agissent les acteurs du système.

I.3. Systèmes répartis sans-attente

La maîtrise de la concurrence est l'un des enjeux principaux dans les systèmes répartis. Selon le niveau de sécurité exigé pour les applications et le contrôle des incidents susceptibles de se produire dans le système sur lequel elles sont déployées, la

4. www.internetlivestats.com/watch/internet-users/

concurrence sera plus ou moins visible. Par exemple, dans le système de commandes d'un avion, la vitesse relative de chaque composant et les chemins empruntés par les ordres de commande sont parfaitement maîtrisés pour supprimer toute incertitude liée à la concurrence. À l'opposé, dans des éditeurs de documents collaboratifs comme subversion ou git, la majorité des événements d'édition sont faits hors ligne. La concurrence est alors détectée au moment de la synchronisation, résultant en conflits qui doivent être résolus manuellement. Les principaux paramètres qui caractérisent un système réparti sont listés ci-dessous.

Échelle du système. Un système est réparti à partir du moment où il contient au moins deux processus. Un programme dont l'interface graphique s'affiche à l'écran de manière asynchrone peut donc déjà être considéré comme réparti. À l'opposé, Tianhe-2, actuellement le plus puissant superordinateur au monde, possède trois millions cent vingt mille cœurs et BOINC, la plateforme de gestion du projet pair à pair SETI@home [SPA 99] vouée à la recherche de signaux extraterrestres, affirme être installée sur plus de treize millions d'ordinateurs dans le monde[5].

Moyen d'interaction. On distingue généralement les systèmes dans lesquels les processus doivent communiquer en s'échangeant des messages et ceux dans lesquels ils ont accès à une mémoire partagée. Des divisions plus fines peuvent détailler ces familles. Si les processus communiquent par messages, le réseau est-il complet ? Les processus connaissent-ils leurs voisins ? Ceux-ci sont-ils fixes ? Peuvent-ils les choisir ? Pour ceux communiquant par mémoire partagée, à quelles opérations ont-ils droit ? Peuvent-ils écrire dans tous les registres ?

Gestion des fautes. Plus les systèmes sont grands, plus les fautes sont fréquentes. Dans les systèmes pair à pair, des processus partent et d'autres se connectent en permanence ; dans un Internet de plus en plus composé de terminaux mobiles connectés, la fiabilité des transmissions ne peut pas être assurée ; les hackers chercheront toujours à attaquer les applications sensibles comme celles des institutions bancaires et gouvernementales (ce que l'on appelle fautes byzantines)... Ces fautes peuvent être de diverses natures : pannes franches (un processus arrête son exécution sans avertir les autres), pertes de messages (en émission ou en réception) ou comportements non conformes des processus (corruption de la mémoire ou attaques intentionnelles). Le nombre de fautes que l'on peut accepter et la présence d'un serveur fiable connu de tous sont également des caractéristiques déterminantes du système.

Rapport au temps. La présence ou l'absence d'une horloge globale accessible par tous les processus change complètement la nature d'un système. Dans un système complètement synchrone, tous les processus avancent suivant un schéma

5. boincstats.com/fr

temporel bien défini (éventuellement *a priori*) et, s'ils communiquent par messages, ces messages sont acheminés en un temps borné. Ces hypothèses modélisent mal l'hétérogénéité du matériel exécutant les processus et la complexité des réseaux qui acheminent les messages. Au contraire, dans un système complètement asynchrone, il n'y a pas de borne sur la vitesse relative des processus ni sur les délais de transfert des messages (*latence* du réseau). Si des pannes franches peuvent en plus se produire, un processus qui ne reçoit pas un message attendu est dans l'incapacité d'en déduire si l'émetteur est fautif ou extrêmement lent. L'hétérogénéité du matériel exécutant les processus peut être une cause importante d'asynchronisme.

Le but de cet ouvrage est d'étudier la concurrence. Nous devons donc nous placer dans un système suffisamment faible pour que les applications ne puissent pas la masquer complètement. Par exemple, on pourrait accepter, comme dans le prologue, que les processus subissent des déconnexions temporaires. Plus précisément, nous faisons l'hypothèse que les processus communiquent en s'envoyant des messages, mais qu'il n'est pas acceptable ou possible pour un processus d'attendre explicitement des messages d'un autre processus. Cette hypothèse est réaliste pour modéliser de nombreux systèmes, dont les suivants.

– Les systèmes dont le nombre de processus est inconnu, ce qui empêche de savoir le nombre de réponses à attendre, par exemple les systèmes pair à pair.

– Les systèmes pour lesquels le temps passé à attendre est jugé trop coûteux, par exemple pour le calcul à haute performance.

– Les systèmes dans lesquels des partitions peuvent se produire, empêchant momentanément les processus de communiquer entre eux. De telles partitions sont fréquentes dans les architectures en nuage [VOG 09]. Les déconnexions temporaires sur Internet peuvent également être modélisées par des partitions.

– Les systèmes pour lesquels la gestion des fautes est critique, au point où le système doit continuer à fonctionner même si un processus se retrouve seul dans le système. Dans une telle situation, un processus ne peut pas attendre la participation d'un nombre connu *a priori* d'autres processus, car ils peuvent être fautifs. C'est ce cas que nous privilégions pour modéliser l'hypothèse de l'absence d'attente, car elle est plus simple à définir formellement.

Plus précisément, nous considérons les systèmes répartis asynchrones à passage de messages sans-attente. Nous ne donnons que l'intuition ici, ces systèmes étant définis formellement dans la section 1.3.2.

Systèmes répartis à passage de messages. Les systèmes que nous considérons sont composés d'un ensemble de taille fixe et connue de processus séquentiels qui communiquent grâce à des primitives d'envoi et de réception de messages.

Systèmes sans-attente. Tous les processus sont susceptibles de tomber en panne, et le système doit continuer à s'exécuter, même réduit à un seul processus.

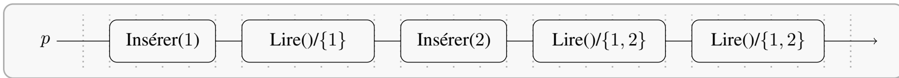
Systèmes asynchrones. Les processus ne s'exécutent pas tous à la même vitesse, et il n'y a pas de borne sur le temps mis par un message pour parvenir à sa destination. Dans ces conditions, un processus qui ne reçoit pas un message qu'il attend ne peut jamais savoir si ce message est très lent à lui parvenir ou si le message n'a jamais été envoyé en raison de la panne de l'émetteur.

I.4. Objets partagés

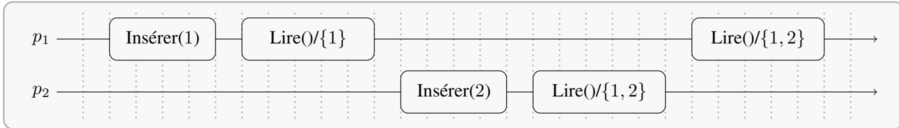
À très haut niveau, les applications réparties ont toutes un modèle semblable : les *processus* interagissent entre eux par l'intermédiaire d'un ou de plusieurs objets partagés qui gèrent la concurrence. Par exemple, dans le cas d'un jeu en ligne, l'objet partagé est la partie en cours, et les règles du jeu explicitent les coups autorisés ; les processus qui calculent les prévisions météorologiques partagent la carte en cours de simulation, etc. La mémoire partagée et les canaux de communication sont d'autres exemples d'objets partagés, mais à plus bas niveau. De manière générale, les applications réparties peuvent souvent être modélisées par couches, chaque couche étant une abstraction, sous forme d'objets partagés, des objets partagés de la couche précédente. Pour résumer, un programme peut être vu comme une hiérarchie d'objets partagés, chacun avec son niveau d'exigence sémantique.

Prenons l'exemple de l'édition collaborative de documents. Au plus haut niveau, l'application donne accès à des opérations d'insertion et de suppression d'un caractère au niveau du curseur et à des opérations de déplacement du curseur, accessibles au clavier. L'implémentation de ces opérations fait appel à une structure de données qui maintient la séquence de caractères du document. Suivant l'application, cette séquence utilise elle-même des objets de plus bas niveau, comme un système de gestion de données abstrayant les primitives de communication du système réparti sur lequel l'application est implantée.

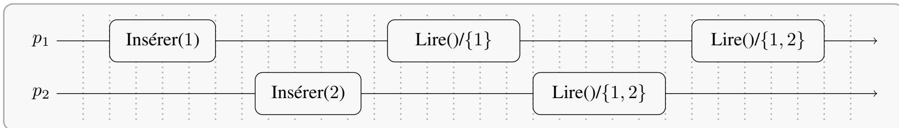
Le fait de se placer dans des systèmes sans-attente a des conséquences importantes sur les objets partagés que l'on peut y construire [ATT 95, ATT 94]. Quand une opération sur un objet partagé est invoquée localement par un processus, elle doit être traitée en utilisant uniquement la connaissance locale du processus. Les lectures doivent être faites localement, ce qui implique que chaque processus ait une copie locale de l'objet. L'impossibilité de se synchroniser pendant les opérations empêche de supprimer toutes les incohérences. Celles-ci doivent donc être spécifiées avec une attention particulière. La question centrale de cet ouvrage peut être formulée de cette façon : *comment spécifier les objets partagés d'un système sans-attente ?*



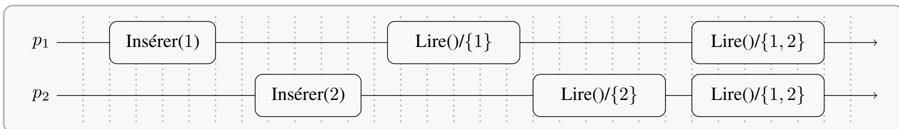
(a) *Exécution séquentielle*



(b) *Exécution linéarisable*



(c) *Exécution séquentiellement cohérente*



(d) *Exécution causalement convergente*

Figure I.4. *Quels comportements sont acceptables ?*

I.5. Spécification des objets partagés

Comment décrire ce qu'est un service de messagerie instantanée ? Si l'on pose la question aléatoirement à des utilisateurs de ces services, fort est à parier que la réponse sera la suivante : « Un service de messagerie instantanée permet l'envoi de messages entre plusieurs correspondants et affiche les messages reçus, du plus ancien au plus récent. » Cela décrit le comportement *normal* de ces services, qui se produit en absence de concurrence, c'est-à-dire en cas d'utilisation séquentielle. De la même manière, les développeurs d'applications concurrentes veulent en général utiliser une version partagée des objets dont ils ont l'habitude dans les programmes séquentiels. La spécification d'un objet partagé doit donc s'appuyer sur sa *spécification séquentielle*.

Bien sûr, la spécification séquentielle ne suffit pas à expliquer les différences entre les trois services exposés dans le prologue. Une propriété supplémentaire, appelée *critère de cohérence*, est nécessaire pour expliciter comment la concurrence est gérée. Pendant une exécution concurrente, les accès aux objets partagés créent des interactions entre les exécutions locales des différents processus. La figure I.4

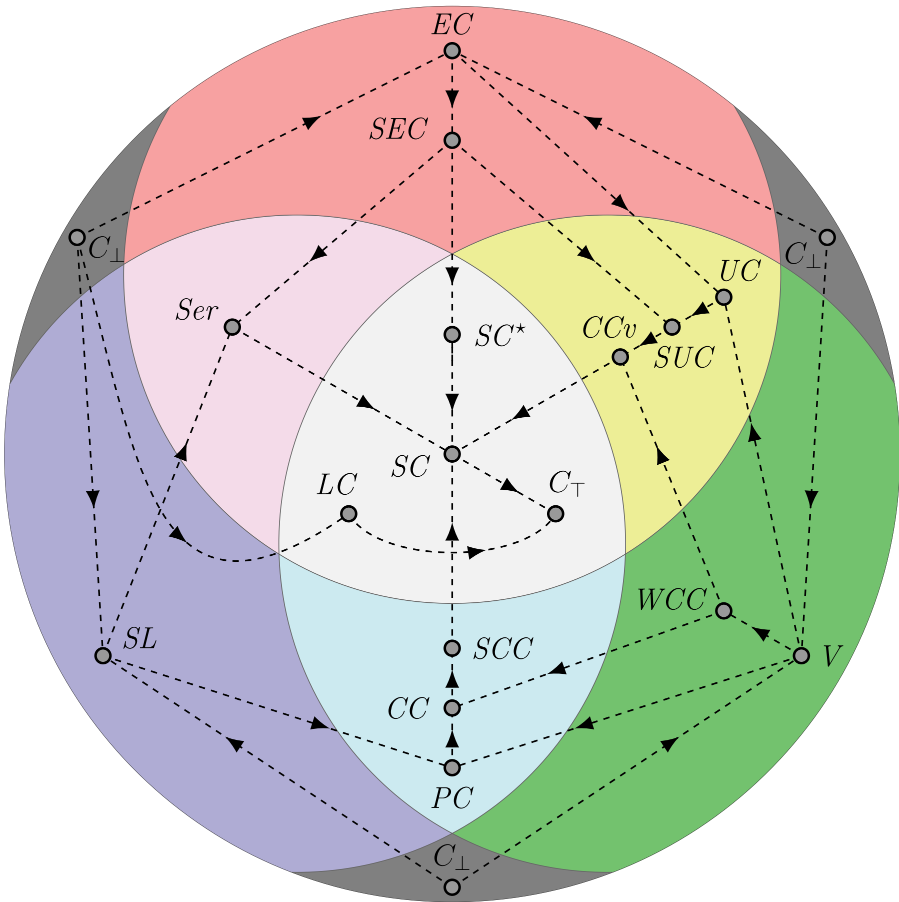
montre quelques exemples d'histoires tirées d'exécutions concurrentes mettant en jeu un ou deux processus partageant un ensemble dans lequel il est possible d'insérer des valeurs, ou dont on peut lire le contenu, c'est-à-dire l'ensemble des valeurs précédemment insérées. La figure montre les opérations exécutées par chaque processus au cours du temps, qui s'écoule de la gauche vers la droite. Un rectangle arrondi délimite le temps entre l'appel et le retour d'une opération. Les valeurs retournées par les lectures sont représentées après une barre oblique.

Quelles histoires sont correctes ? Dans le cas séquentiel, il n'y a pas d'ambiguïté possible. Sur la figure I.4a, lors de sa première lecture, le processus a inséré 1 ; il ne peut donc retourner que $\{1\}$. Les lectures suivant l'insertion du 2 n'ont d'autre choix que de retourner $\{1, 2\}$. L'histoire de la figure I.4b doit également être considérée correcte : toutes les opérations sont consécutives dans le temps et la suite des valeurs retournées est la même que dans l'exécution séquentielle. Qu'en est-il de la figure I.4c ? Les opérations sont également successives, mais la première lecture suit l'insertion du 2 sans la prendre en compte. D'un côté, on pourrait soutenir que cette lecture est fausse, et donc que cette histoire est incohérente. D'un autre côté, chaque processus effectue exactement les mêmes opérations et lit les mêmes valeurs que dans l'exécution précédente. La seule différence entre ces deux histoires est donc la façon dont les deux processus perçoivent leur vitesse relative. Or, dans un système asynchrone, les processus ne peuvent pas faire la différence entre les deux histoires, car l'ordre temporel observé pourrait aussi bien être dû à une désynchronisation des horloges des deux processus, ce qui n'aurait aucun effet sur les programmes qui utilisent l'objet. Il n'y a donc pas de réponse définitive à ce qu'est une histoire cohérente, chaque situation étant simplement mieux décrite par un critère de cohérence particulier. L'histoire de la figure I.4b est dite *linéarisable* alors que celle de la figure I.4c est seulement *séquentiellement cohérente*.

Hélas, la cohérence séquentielle et la linéarisabilité sont des critères trop forts pour être implémentés dans les systèmes sans-attente [ATT 94, GIL 02] : des incohérences comme celle de l'histoire de la figure I.4d ne peuvent pas être évitées. Comment, alors, décrire de telles histoires ? La littérature sur le sujet présente un canevas serré de concepts et de définitions, souvent partiels et incomparables entre eux. L'extension et la formalisation de ces concepts sont une partie importante du travail présenté dans cet ouvrage. Nous nous efforçons de combler les vides pour extraire la structure de l'espace des critères faibles et en dresser une carte, présentée sur la figure I.5.

I.6. Organisation

Le chapitre 1 présente le cadre mathématique commun à tous les autres chapitres : nous y définissons formellement les spécifications séquentielles, les histoires concurrentes et les critères de cohérence. Ce chapitre définit également formellement les systèmes sans-attente.



PC Cohérence pipeline (p. 74)
SEC Convergence forte (p. 67)
SUC Cohérence d'écritures forte (p. 91)
CCv Convergence causale (p. 117)
WCC Cohérence causale faible (p. 115)
SCC Cohérence causale forte (p. 128)

SL Localité d'état (p. 140)
EC Convergence (p. 65)
UC Cohérence d'écritures (p. 90)
Ser Sérialisabilité (p. 63)
V Validité (p. 139)
CC Cohérence causale (p. 121)

SC Cohérence séquentielle (p. 54)
LC Cohérence locale (p. 45)
$SC^\star$ Cohérence de cache (p. 47)
$C_\bot$ Critère minimal (p. 46)
$C_\top$ Critère maximal (p. 46)

Figure I.5. *Carte des critères de cohérence étudiés dans cet ouvrage*

Dans le chapitre 2, nous présentons les principales notions proposées par les différentes communautés confrontées à la concurrence pour spécifier leurs objets répartis. Nous tentons, autant que faire se peut, d'inscrire ces notions dans le cadre présenté au chapitre 1, dans le but de les comparer. Les notions plus difficiles à généraliser feront l'objet des chapitres suivants.

La convergence impose que tous les processus finissent dans un *état de convergence* commun. Spécifier cet état de convergence est un problème difficile. Dans le chapitre 3, nous présentons deux critères de cohérence pour le résoudre : la *cohérence d'écritures* et la *cohérence d'écritures forte*, qui relient l'état de convergence aux écritures grâce à la spécification séquentielle. Nous y présentons aussi trois algorithmes pour les implémenter dans les systèmes sans-attente.

La mémoire causale [AHA 95] est reconnue comme un objet central parmi les modèles de mémoire qui peuvent être implémentés dans les systèmes sans-attente, mais sa définition utilise les liens sémantiques entre les lectures et les écritures. Faire entrer la causalité comme un critère de cohérence dans notre modèle est un défi relevé par le chapitre 4. Nous y définissons quatre critères de cohérence. La *cohérence causale faible* est le plus grand dénominateur commun de ces quatre critères. Elle peut être associée à la cohérence d'écritures pour former la *convergence causale*, ou à la cohérence pipeline, pour former la *cohérence causale*, qui correspond à la mémoire causale quand elle est appliquée à la mémoire. La *cohérence causale forte*, enfin, permet l'étude de la fausse causalité induite par l'implémentation standard de la mémoire causale.

Le chapitre 5 se concentre sur la calculabilité dans les systèmes sans-attente. Nous y dressons une cartographie de la structure de l'espace des critères faibles, divisée en trois familles de critères primaires (*convergence*, *validité* et *localité d'état*), qui peuvent être conjugués deux à deux pour former trois familles de critères secondaires (*cohérence d'écritures*, *cohérence pipeline* et *sérialisabilité*). En revanche, la conjonction des trois critères primaires ne peut pas être implémentée dans les systèmes sans-attente, ce qui justifie les couleurs de la figure I.5.

Le chapitre 6 présente comment ces notions peuvent s'appliquer à la programmation dans les systèmes répartis. La bibliothèque CODS met en œuvre les critères de cohérence faibles dans le langage de programmation orienté objet D. Elle offre une interface de programmation abstraite la plus transparente possible, dans laquelle l'instanciation est seulement remplacée par l'évocation d'une spécification séquentielle, définie par une classe du programme, et d'un critère de cohérence choisi parmi ceux de la bibliothèque, ou bien implémenté comme une extension de cette bibliothèque.

Finalement, la conclusion résume les chapitres précédents et expose les pistes de recherche futures. Après la bibliographie, l'index aide à se retrouver dans le document.