

# Algorithmes distribués et détection de défaillances

Carole Delporte-Gallet

LIAFA, Université Paris 7 - Denis Diderot  
2 pl Jussieu case 7024 5251 PARIS CEDEX 05  
cd@liafa.jussieu.fr

---

Le domaine de l'algorithmique distribuée tolérante aux pannes est un domaine relativement récent : les premiers travaux datent de la fin des années soixante-dix.

Son développement est allé de pair avec celui des réseaux. Avec ceux-ci, un service n'est plus, en général, réalisé par un processus unique s'exécutant sur un ordinateur unique, mais par plusieurs processus s'exécutant sur différents ordinateurs et échangeant des informations.

Répartir un service sur plusieurs processeurs connectés entre eux présente aussi l'avantage de pouvoir résister à des pannes : en dupliquant les divers matériels, on peut espérer maintenir le service même si certains d'entre eux connaissent des défaillances. Les défaillances peuvent provenir aussi bien du réseau que des ordinateurs.

La finalité de l'algorithmique distribuée tolérante aux pannes est d'offrir, malgré les pannes, des services fiables. Le champ d'application de ces services est vaste: téléphonie, système de réservation, gestion de trafic aérien, bourse, systèmes embarqués, etc.

De tels services sont nécessairement complexes. Aussi, une manière classique de les aborder est de les construire à partir de services plus basiques correspondant à des *middlewares*. De tels services basiques seront les *primitives* à partir desquelles le service de plus haut niveau sera développé. Ces dernières seront utilisées comme des boîtes noires. Le développeur n'en connaîtra que l'interface et la spécification. Il ne devra pas connaître, et ne pourra faire aucune hypothèse, sur la manière dont l'implémentation est réalisée.

Le développement ultérieur reposant sur ces primitives, l'étude et le choix de celles-ci sont cruciaux. L'expérience en la matière a permis d'en identifier un certain nombre.

De nombreux efforts ont été faits pour en donner des spécifications. Car, même si l'intuition du service à obtenir est claire, il n'est pas si facile d'en donner des spécifications qui ne soient pas vides ou contradictoires ou encore irréalisables. D'autant que ces services basiques peuvent eux-mêmes être relativement complexes. Il faut trouver un compromis acceptable entre ce que l'on aimerait avoir et ce qui est réalisable.

Ces efforts ont porté leurs fruits et de nombreux services comme, par exemple, ceux assurant la diffusion fiable de messages, le consensus sur une valeur ou la diffusion ordonnée des messages, ont des spécifications parfaitement définies dont la pertinence fait l'unanimité dans la communauté.

Mais, avant tout, il faut préciser les caractéristiques des systèmes dans lesquels on va réaliser ces primitives. Parmi celles-ci on a essentiellement : les propriétés des horloges des processus, les vitesses d'exécution et les délais de transmission des messages. Si dans un réseau local, on peut, dans une certaine mesure, maîtriser et quantifier ces paramètres, dès que le réseau est plus large trop de paramètres entrent en jeu et cela devient impossible.

Aussi, en algorithmique distribuée classique on fait alors l'abstraction de l'asynchronisme. Comme on ne connaît pas les vitesses d'exécution des processus, on suppose simplement que tous fonctionnent réellement, c'est à dire qu'un processus ne fait pas une infinité de pas de calcul entre deux pas d'un autre. De même, comme on ne connaît pas les délais de transmission, on suppose simplement que les messages parviennent un jour à leurs destinataires.

Cette approche a l'avantage de la généralité: les algorithmes développés dans ce modèle pourront être ensuite utilisés quel que soit le système cible.

Mais, cette approche ne fonctionne plus aussi bien pour l'algorithmique distribuée *tolérante aux pannes*. En effet, en l'absence de connaissance sur les vitesses des processus et les délais d'acheminement des messages, la plupart des problèmes que l'on essaie de résoudre dans ce domaine n'ont plus de solution.

En effet, de nombreux services nécessitent la réalisation d'un *accord* entre les processus : accord sur une valeur, accord sur un message, sur les processus en panne etc... Un problème d'accord très simple est celui du *consensus*. Ce problème est un des principaux paradigmes de l'algorithmique distribuée tolérante aux pannes. Pour le problème du consensus, chaque processus dispose d'une valeur initiale. Les processus ont alors à décider sur une valeur commune. Cette valeur commune est choisie parmi les valeurs initiales. Une fois qu'un processus a décidé sur la valeur du consensus, il ne peut plus changer d'avis. Sa décision est irrévocable.

Même si les messages sont acheminés de manière fiable, et, même si, au plus un seul processus peut tomber en panne, il est impossible de réaliser un accord entre les processus : les processus ne peuvent choisir une valeur commune. C'est le résultat classique de Fischer, Lynch et Paterson [4]. La preuve formelle est délicate, mais l'intuition en est simple : on ne peut pas distinguer un processus lent d'un processus en panne. Et cette incertitude conduit à ce qu'aucun des processus ne décide ou bien à ce qu'ils décident sur des valeurs différentes.

Par contre, si on connaît les vitesses d'exécution des processus et les délais d'acheminement des messages, on peut résoudre le problème du consensus. Mais pour ce problème simple, il n'est pas nécessaire d'avoir toute cette connaissance. Par exemple, on peut le résoudre dans un modèle *partiellement synchrone* particulier comme défini dans [2]. Ainsi le consensus peut être résolu, dans le modèle partiellement synchrone suivant : une majorité de processus ne sont jamais en panne, on ne connaît pas les délais de transmission, mais, on sait qu'il existe un instant et des bornes tels que, à partir de cet instant, les délais d'acheminement des messages et les vitesses des processus seront inférieurs à ces bornes.

Une des difficultés de l'algorithmique distribuée tolérante aux pannes est liée à ce résultat d'impossibilité qui empêche de résoudre dans le modèle asynchrone la plupart des problèmes d'accord.

Il n'émerge pas, actuellement, une modélisation unique pour les systèmes distribués avec défaillances.

On peut considérer un modèle où intervient directement le temps : c'est le cas des modèles synchrones, partiellement synchrones ([2] par exemple) ou asynchrones avec temps ([3] par exemple). L'écriture d'algorithmes distribués résistants aux pannes dans de tels modèles peut se révéler délicate et complexe.

On peut aussi avoir une approche modulaire et développer les algorithmes dans un modèle asynchrone enrichi d'une primitive qui abstrait les propriétés temporelles du système cible : le modèle asynchrone enrichi d'un détecteur de défaillances de [1] en est un des exemples les plus classiques. Dans ce deuxième cas, se posera, ensuite, le problème de la réalisation dans le système cible de cette abstraction.

Chaque approche (partiellement synchrone, asynchrone avec temps, asynchrone enrichi d'une abstraction...) donne lieu à une famille de modèles. A l'intérieur de chacune de ces familles, chaque modèle a des propriétés particulières, plus ou moins fortes, qui requièrent des suppositions particulières sur le système sous-jacent. Notons enfin qu'au sein d'une même famille, les modèles peuvent être incomparables.

Les divers types de défaillances considérés font partie du modèle.

Tout d'abord, il y a le nombre de processus défaillant que l'on tolère. La fréquence d'apparition des défaillances peut aussi jouer un rôle. Enfin la nature même des défaillances peut varier. Les processus comme les canaux de communications peuvent être sujets à des défaillances plus ou moins graves.

Pour les processus, les défaillances peuvent être des pannes franches de processus : le processus est correct jusqu'à ce qu'il tombe en panne, il n'exécute alors plus aucun pas de calcul. Le processus défaillant peut, mis à part l'échec sur l'émission ou sur la réception de messages, exécuter correctement le reste de son code. Enfin le processus défaillant peut avoir n'importe quel comportement, on parle alors de comportements byzantins.

Pour les canaux de communication, les défaillances peuvent être des pertes, des duplications ou des corruptions des messages.

Revenons au problème de la réalisation d'une primitive, on peut procéder de la façon suivante.

Si on considère des modèles asynchrones enrichis d'un détecteur de défaillances, les propriétés imposées au modèle pour assurer l'implémentation de la primitive se traduiront en propriétés du détecteur de défaillances. Le système devra alors permettre d'implémenter un détecteur de défaillances satisfaisant ces propriétés.

Dans cette démarche, est cruciale la recherche de conditions minimales: sur le détecteur de défaillances pour assurer un service, sur le système pour implémenter le détecteur de défaillance, etc...

A défaut de conditions minimales, on cherche à obtenir des conditions suffisantes et des conditions nécessaires. En effet ces conditions permettront de mieux cerner les systèmes dans lesquels la primitive est implémentable. Si on dispose de conditions suffisantes, tout système qui les assure pourra implémenter la primitive. Si on dispose de conditions nécessaires, tout système qui ne les assure pas ne pourra pas implémenter la primitive.

Ces conditions sont importantes tant du point de vue théorique, pour mieux maîtriser le domaine que du point de vue pratique, pour la réalisation des services.

Lorsqu'une primitive a été développée dans un modèle particulier, le fait que l'implémentation soit correcte signifie que si le système cible assure les propriétés supposées par ce modèle particulier, alors la spécification de la primitive est assurée. Il est intéressant de savoir ce qui se passe si certaines de ces propriétés ne sont plus assurées par le système cible. Cela revient à déterminer la dégradation du service en cas de dégradation (non tolérée) du système. Par exemple, dans le cas du consensus, est-ce que l'on perd la propriété d'Accord (les processus décident une valeur différente), la propriété d'Intégrité (la valeur de décision n'est pas une des valeurs initiales) ou la propriété de Terminaison (certains processus ne décident pas)?

La manière dont se dégradent les propriétés de la primitive peut se révéler critique pour certaines applications.

Le plus souvent, on peut conserver les propriétés de sûreté (dans le cas du consensus l'Accord et l'Intégrité). Mais on perd la propriété de vivacité (dans le cas du consensus la propriété de Terminaison: soit les processus sont bloqués, soit ils continuent à échanger des messages mais ne parviennent pas à décider).

Il est clair que les algorithmes développés seront différents suivant le type de propriétés que l'on veut préserver.

Jusqu'à présent les directions de recherches indiquées concernent essentiellement la maîtrise de l'algorithmique dans le sens où on se pose uniquement la question de l'existence d'algorithmes pour tel ou tel problème.

Une autre direction concerne la réalisation de ces services avec comme objectif non plus l'existence d'algorithmes mais leur efficacité. On retrouve les soucis classiques de l'algorithmique consistant à obtenir des algorithmes optimaux vis-à-vis de différents critères.

Ici, ces critères peuvent être le temps de calcul, le nombre de messages, le nombre d'appels à une primitive...

Pour résumer, l'objectif de ce domaine est de réaliser une algorithmique maîtrisée et efficace.

## References

- [1] T. Chandra and S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems*. Journal of the ACM, 43(2), March 1996.
- [2] C. Dwork, N. Lynch and L. Stockmeyer. *Consensus in the presence of partial synchrony*. Journal of the ACM, 35(2), 1988.
- [3] C. Fetzer and F. Cristian. *The Timed Asynchronous Distributed System Model*. IEEE Transactions on Parallel and Distributed Systems, 10(6), 1999

- [4] M. Fischer, N. Lynch and M. Paterson. *Impossibility of distributed consensus with one faulty process*. Journal of the ACM, 32(2), 1985.