6 Explorer MapReduce

6.1 Composants peu fiables (Unreliable Components)

Le sujet de cette section est les systèmes de calcul distribués. Lorsque les gens parlent de systèmes distribués, ils mentionnent souvent que ces systèmes sont construits à partir de composants peu fiables.

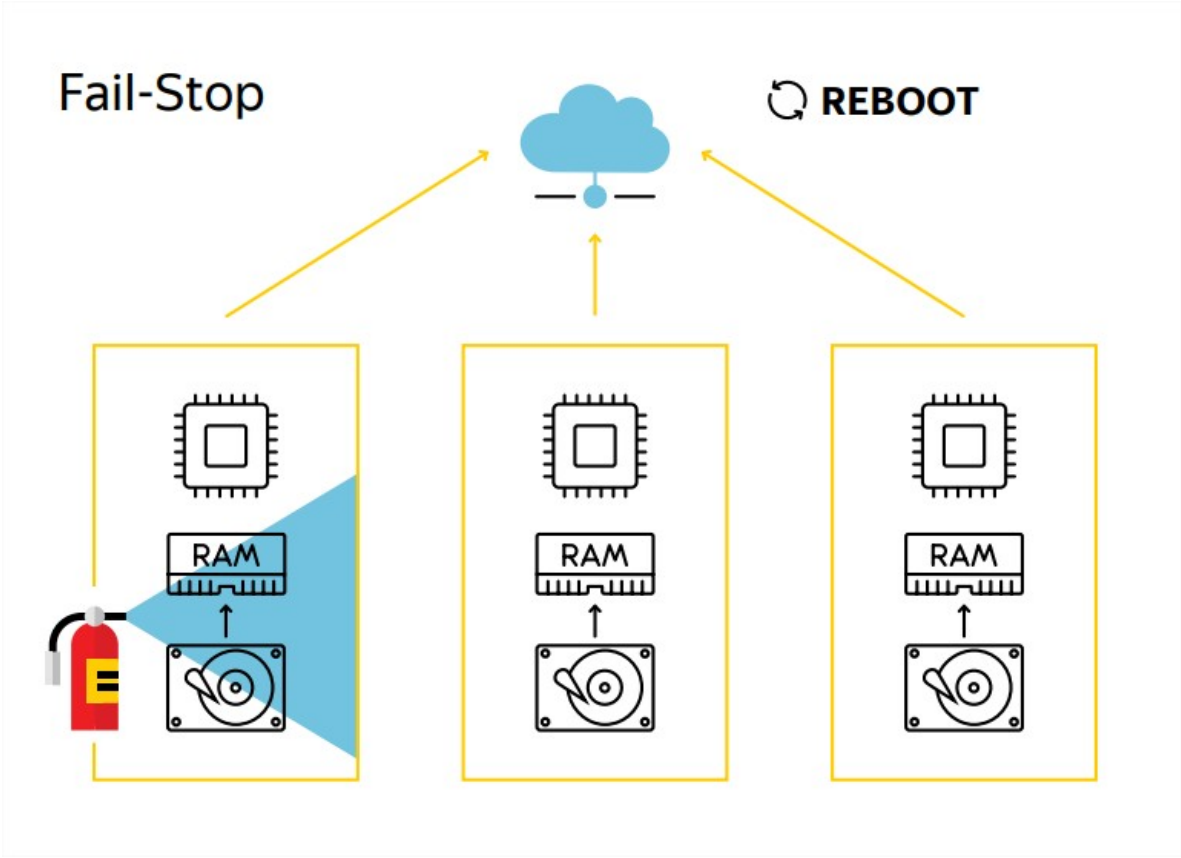
Imaginons que vous ayez un système de fichiers distribué comme HDFS que vous avez créé. Comme vous vous en souvenez, les gros fichiers de ce système de fichiers sont répartis sur différentes machines.

Supposons que vous ayez un énorme fichier texte. Par exemple, de Wikipedia stocké dans un fichier, ayant un article par ligne. Et vous devez trouver les mots les plus populaires ici. Si vous souhaitez trouver une réponse à ce problème, vous devez d'abord lire les données des disques locaux, effectuer des calculs et regrouper les résultats sur le réseau. Quels composants peuvent casser dans ce système? Pour être honnête, tous et chacun.

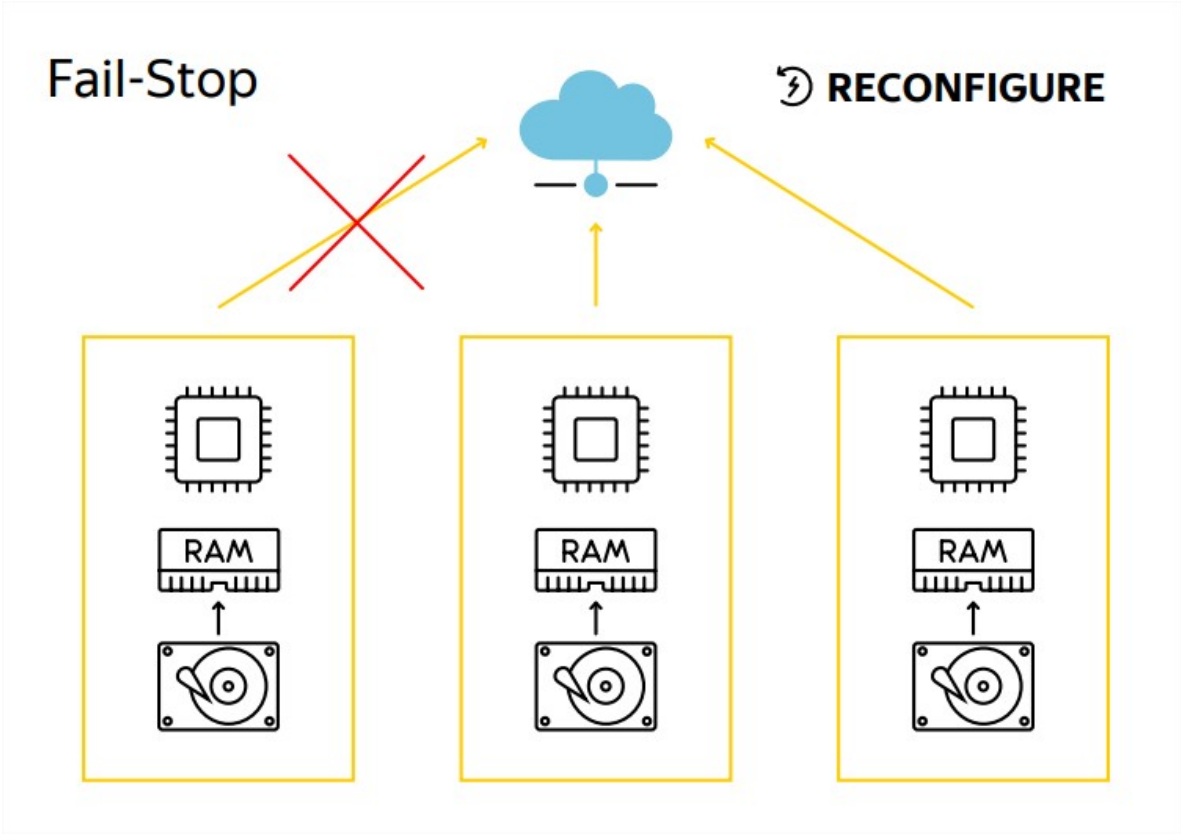
Les nœuds de cluster peuvent être interrompus à tout moment en raison de l'alimentation électrique, des dommages au disque, de la surchauffe des processeurs, etc. Du système distribué, perspective plus claire. Il existe trois types de pannes de nœuds.

6.1.1 Échec Fail-stop

Un échec fail-stop ne signifie pas que si la machine tombe en panne, elle se bloque pour de bon. Cela signifie que si les machines sont hors service pendant un calcul, vous devez avoir un impact externe pour ramener le système à un état de fonctionnement. Par exemple, un administrateur système doit soit réparer le nœud et redémarrer tout le système ou une partie de celui-ci.



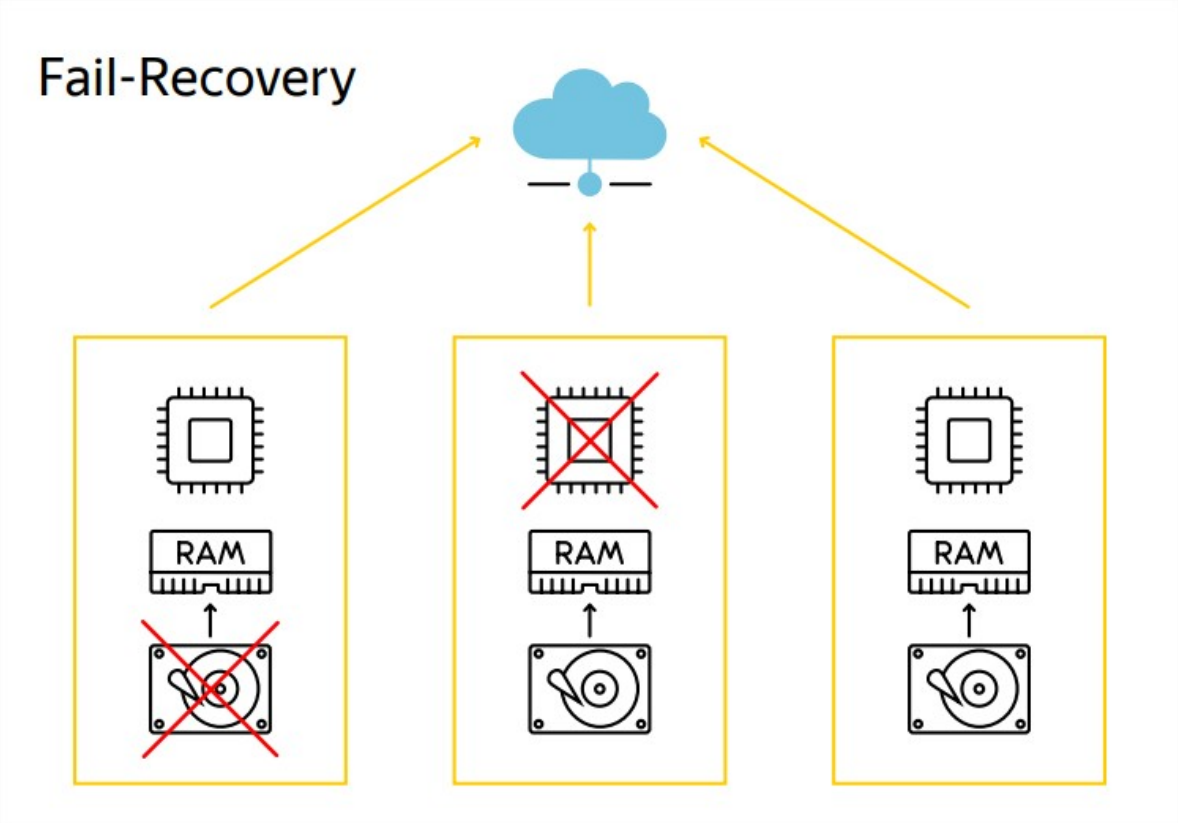
Ou, un administrateur système doit retirer la machine défectueuse et reconfigurer le système distribué. Par conséquent, ces systèmes distribués ne sont pas robustes aux plantages de nœuds.



6.1.2 Échec Fail-recovery

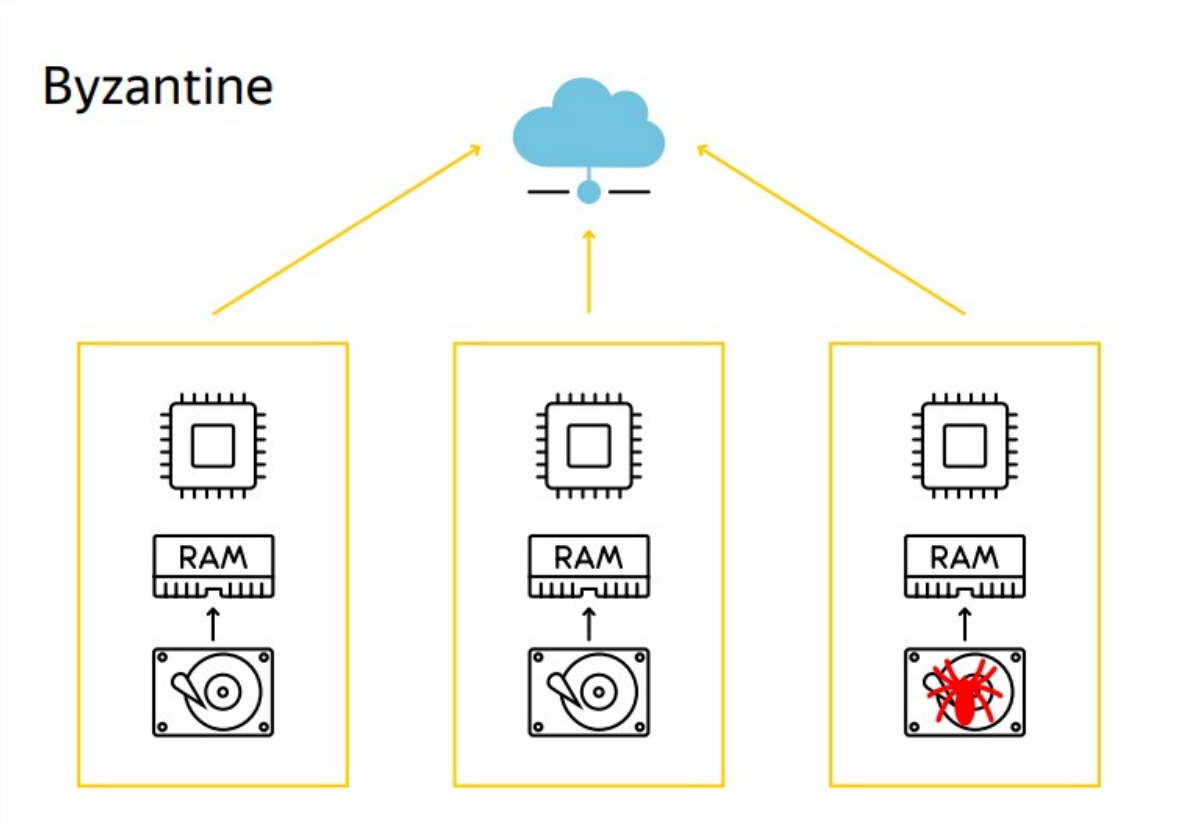
Un échec de fail-recovery signifie que pendant les calculs, les notes peuvent tomber en panne arbitrairement et revenir aux serveurs. Un processus plein de surprises, votre vie ne sera plus jamais ennuyeuse. Ce qui est intéressant, ce comportement n'influence pas l'exactitude et le succès des calculs. Autrement dit, aucun impact externe n'est nécessaire pour reconfigurer le système lors de tels événements.

Par exemple, si un disque dur a été endommagé, un administrateur système peut changer physiquement le disque dur. Et il n'y a pas d'autre étape nécessaire pour remettre ce nœud en service. Après reconnexion, ce nœud sera automatiquement récupéré par un système distribué. Et il pourra même participer aux calculs actuels.

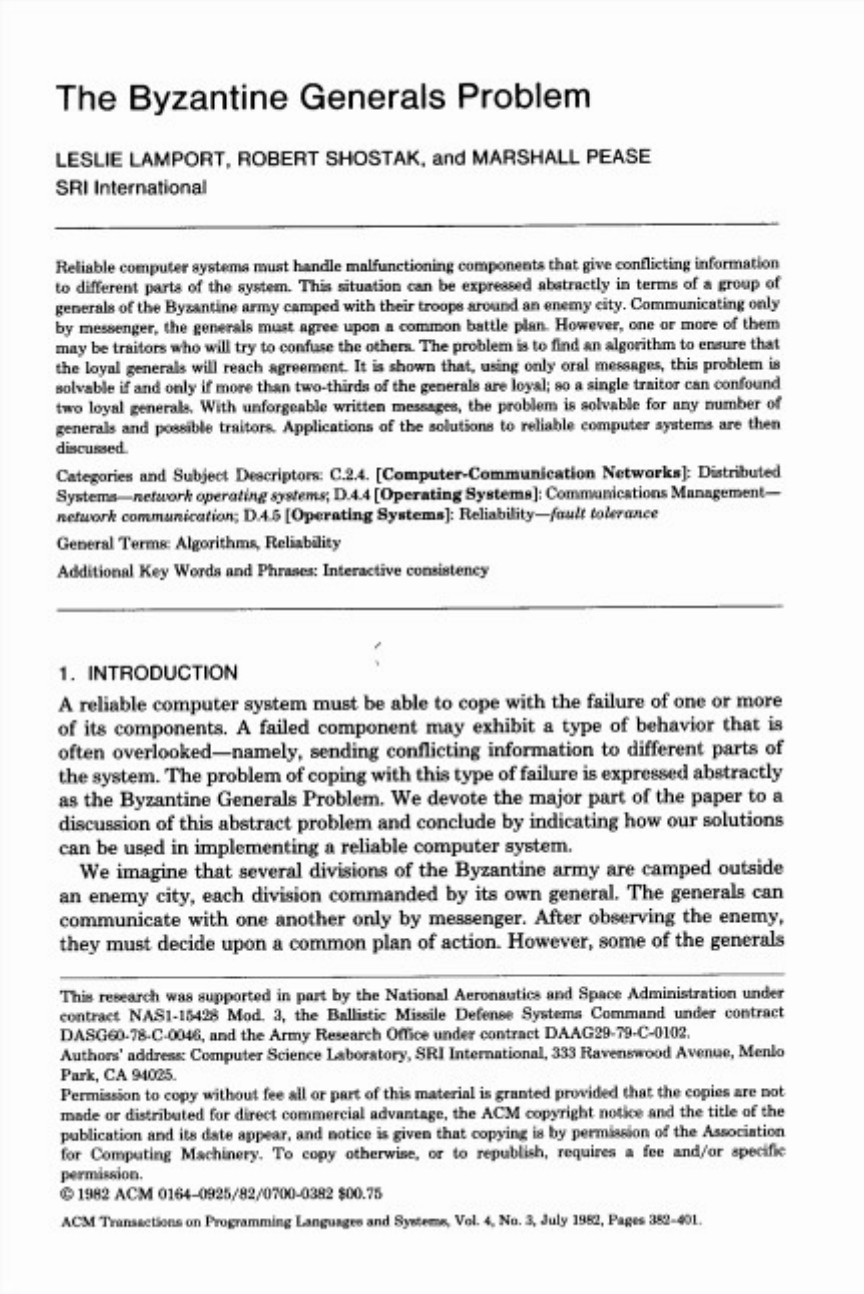


6.1.3 Échec Byzantine

Un système distribué présente des pannes byzantines robustes s'il peut fonctionner correctement malgré le comportement de certains nœuds hors protocole. En d'autres termes, vous avez des nœuds qui vont passer entre leurs petites dents numériques pour déstabiliser le système. Si vous développez un système financier, vous devrez probablement faire face à ces types de défaillances pour protéger vos clients et votre entreprise.



La définition de l'échec byzantin est largement connue en raison de l'article , écrit par Leslie Lamport, Robert Shostak et Marshall Pease, et publié en 1982.



The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. **[Computer-Communication Networks]**: Distributed Systems—*network operating systems*; D.4.4 **[Operating Systems]**: Communications Management—*network communication*; D.4.5 **[Operating Systems]**: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

1. INTRODUCTION

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked—namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem. We devote the major part of the paper to a discussion of this abstract problem and conclude by indicating how our solutions can be used in implementing a reliable computer system.

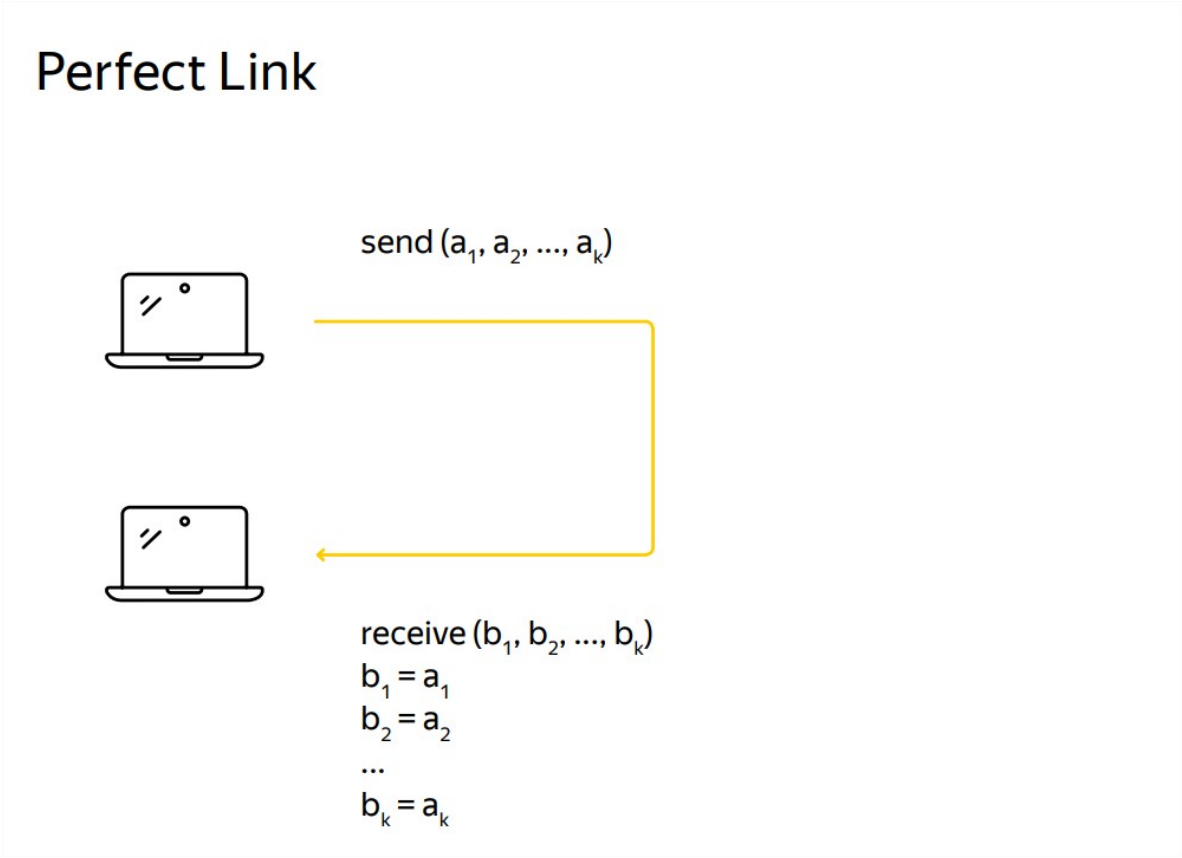
We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals

This research was supported in part by the National Aeronautics and Space Administration under contract NAS1-15428 Mod. 3, the Ballistic Missile Defense Systems Command under contract DASG60-78-C-0046, and the Army Research Office under contract DAAG29-79-C-0102.
Authors' address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

Dans un système distribué, différentes connaissances peuvent échouer et c'est au système de décider quoi faire ensuite. Mais en plus des pannes de nœuds, des problèmes de connexion réseau peuvent survenir. Encore une fois, vous pouvez classer les liens en trois types.

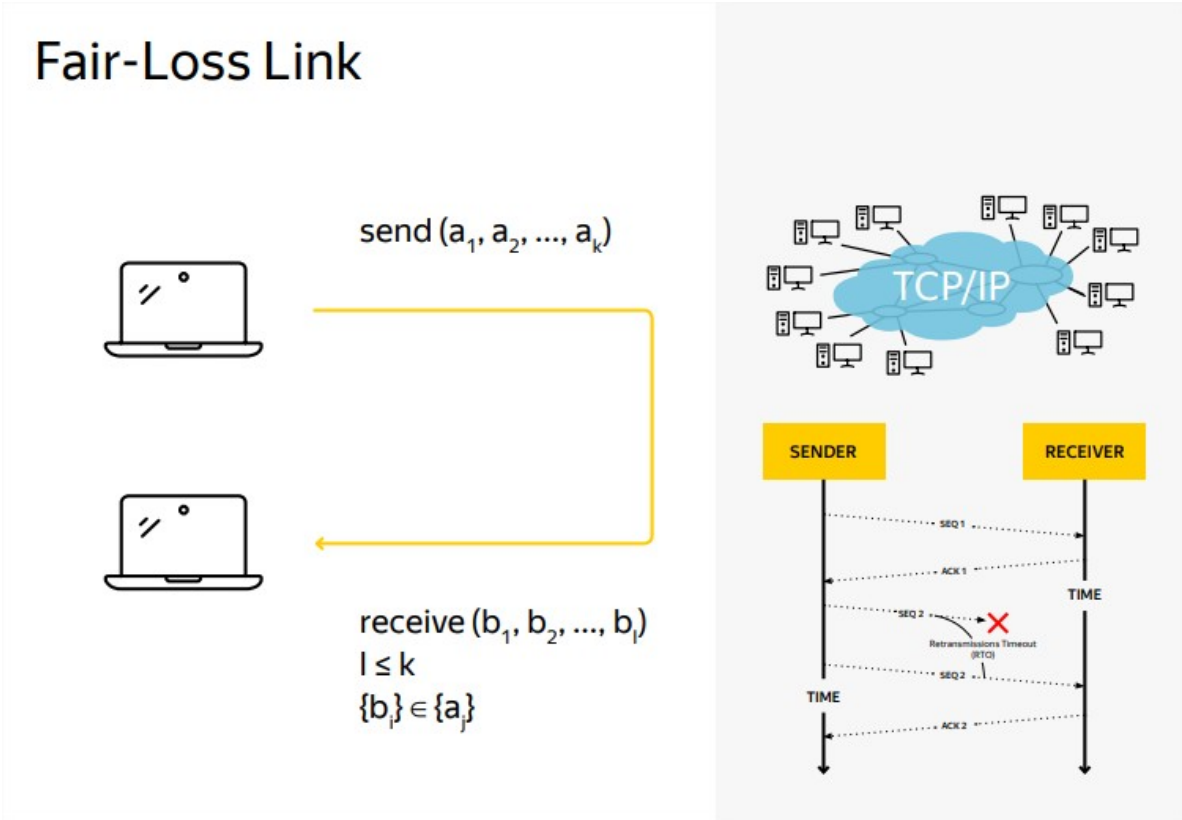
6.1.4 Perfect Link

Si vous avez un lien parfait, cela signifie que tous les messages envoyés doivent être livrés et reçus sans aucune modification dans le même ordre.



6.1.5 Fail-loss Link

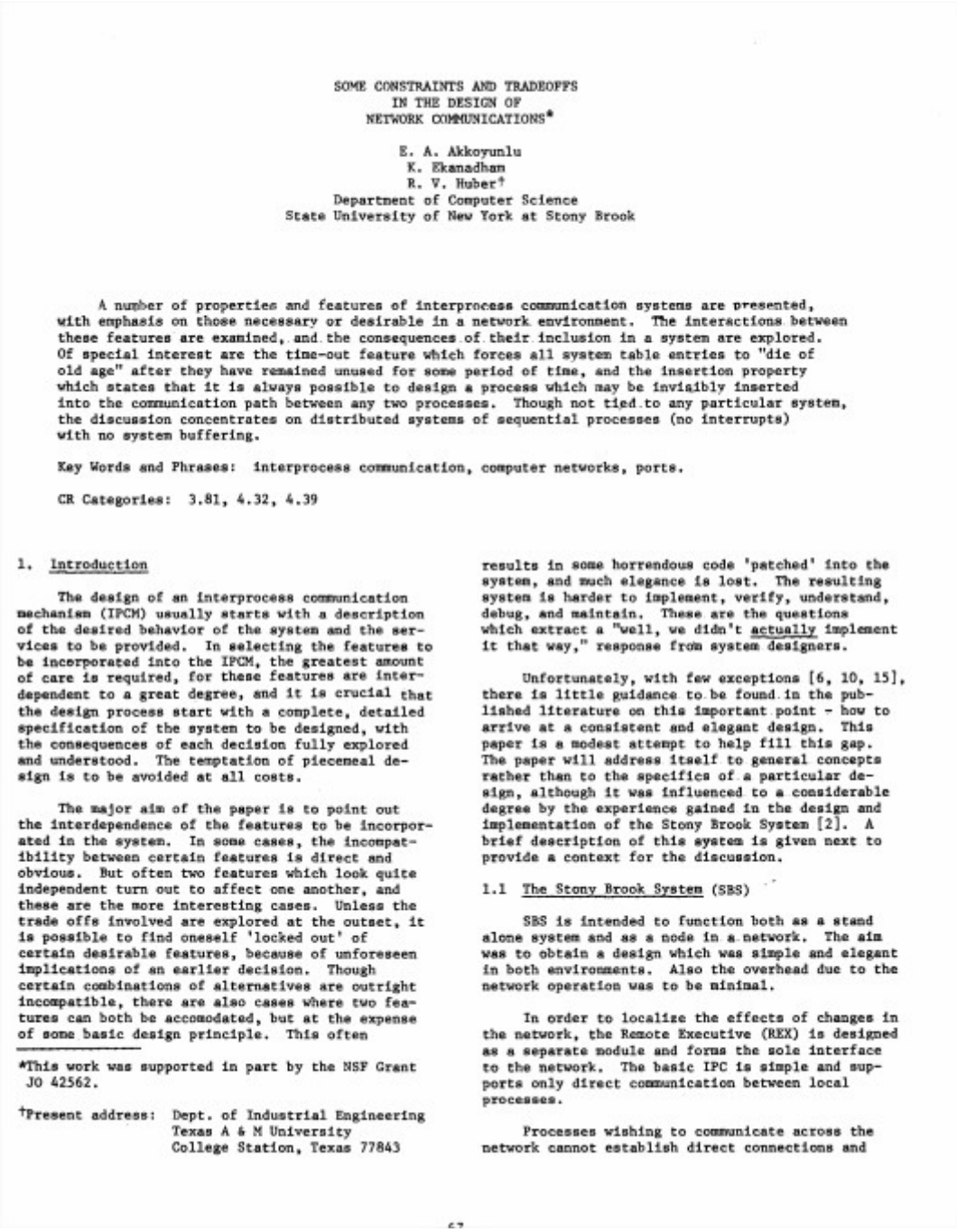
Le lien de Fail-loss signifie qu'une partie des messages peut être perdue mais la probabilité de perte de message ne dépend pas du contenu d'un message. La perte de paquets est un problème très courant pour les connexions réseau.



Par exemple, le protocole TCP / IP bien connu tente de résoudre ce problème en retransmettant des messages s'ils n'ont pas été reçus.

6.1.6 Byzantine Link

Le dernier type de liens est les liens byzantins. Si vous avez des liens byzantins dans le système, cela signifie que certains messages peuvent être filtrés selon une règle, certains messages peuvent être modifiés et certains messages peuvent être créés de nulle part. Les problèmes de liens byzantins sont liés au paradoxe des deux généraux.



SOME CONSTRAINTS AND TRADEOFFS
IN THE DESIGN OF
NETWORK COMMUNICATIONS*

E. A. Akkoyunlu
K. Ekanadham
R. V. Huber†
Department of Computer Science
State University of New York at Stony Brook

A number of properties and features of interprocess communication systems are presented, with emphasis on those necessary or desirable in a network environment. The interactions between these features are examined, and the consequences of their inclusion in a system are explored. Of special interest are the time-out feature which forces all system table entries to "die of old age" after they have remained unused for some period of time, and the insertion property which states that it is always possible to design a process which may be inviably inserted into the communication path between any two processes. Though not tied to any particular system, the discussion concentrates on distributed systems of sequential processes (no interrupts) with no system buffering.

Key Words and Phrases: interprocess communication, computer networks, ports.

CR Categories: 3.81, 4.32, 4.39

1. Introduction

The design of an interprocess communication mechanism (IPCM) usually starts with a description of the desired behavior of the system and the services to be provided. In selecting the features to be incorporated into the IPCM, the greatest amount of care is required, for these features are interdependent to a great degree, and it is crucial that the design process start with a complete, detailed specification of the system to be designed, with the consequences of each decision fully explored and understood. The temptation of piecemeal design is to be avoided at all costs.

The major aim of the paper is to point out the interdependence of the features to be incorporated in the system. In some cases, the incompatibility between certain features is direct and obvious. But often two features which look quite independent turn out to affect one another, and these are the more interesting cases. Unless the trade offs involved are explored at the outset, it is possible to find oneself 'locked out' of certain desirable features, because of unforeseen implications of an earlier decision. Though certain combinations of alternatives are outright incompatible, there are also cases where two features can both be accomodated, but at the expense of some basic design principle. This often results in some horrendous code 'patched' into the system, and much elegance is lost. The resulting system is harder to implement, verify, understand, debug, and maintain. These are the questions which extract a "well, we didn't actually implement it that way," response from system designers.

Unfortunately, with few exceptions [6, 10, 15], there is little guidance to be found in the published literature on this important point - how to arrive at a consistent and elegant design. This paper is a modest attempt to help fill this gap. The paper will address itself to general concepts rather than to the specifics of a particular design, although it was influenced to a considerable degree by the experience gained in the design and implementation of the Stony Brook System [2]. A brief description of this system is given next to provide a context for the discussion.

1.1 The Stony Brook System (SBS)

SBS is intended to function both as a stand alone system and as a node in a network. The aim was to obtain a design which was simple and elegant in both environments. Also the overhead due to the network operation was to be minimal.

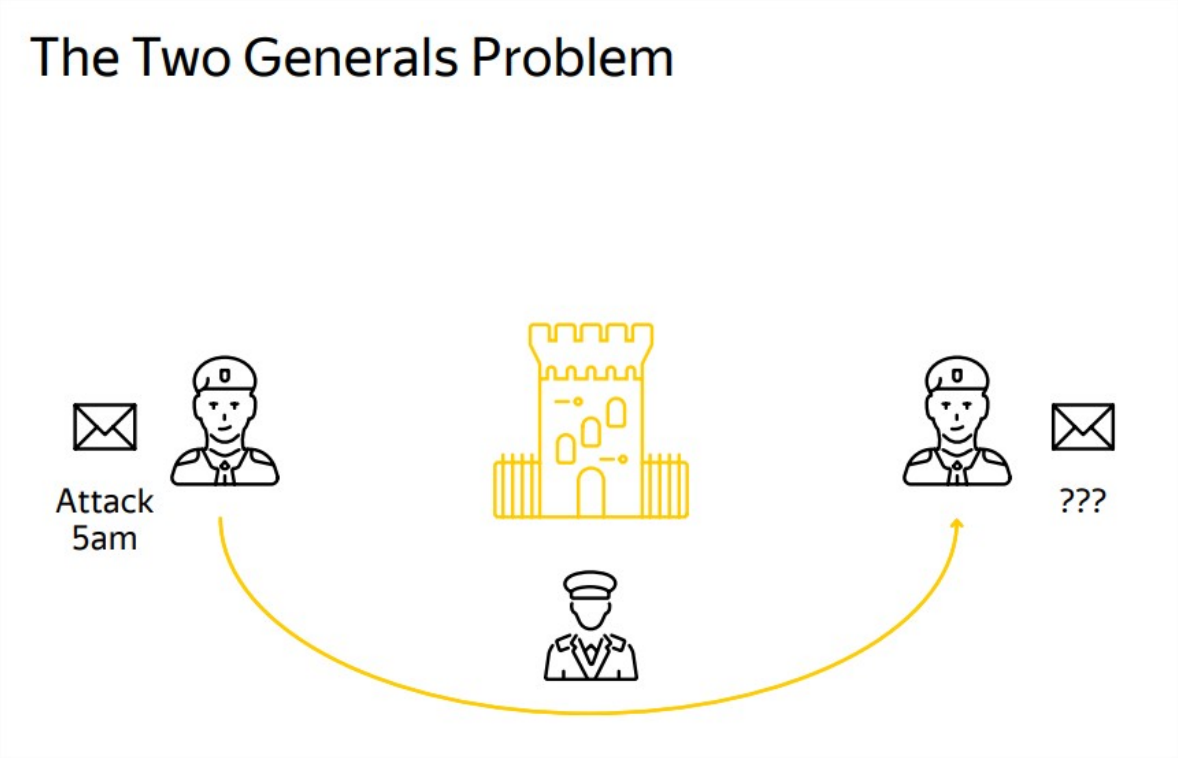
In order to localize the effects of changes in the network, the Remote Executive (REX) is designed as a separate module and forms the sole interface to the network. The basic IPC is simple and supports only direct communication between local processes.

Processes wishing to communicate across the network cannot establish direct connections and

*This work was supported in part by the NSF Grant JO 42562.

†Present address: Dept. of Industrial Engineering
Texas A & M University
College Station, Texas 77843

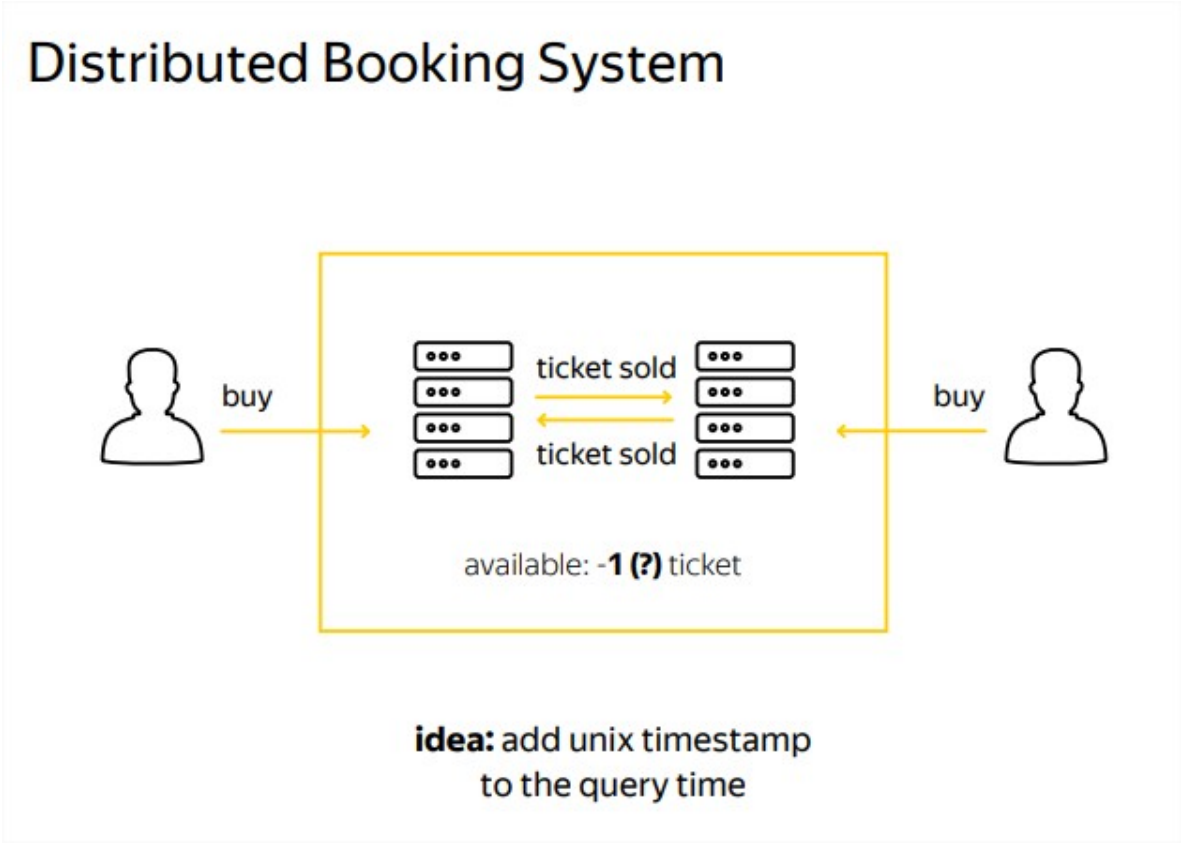
Le problème des deux généraux a été le premier problème de communication informatique à se révéler insoluble. Histoire similaire, il y a deux généraux en campagne. Ils ont un objectif, une tour qu'ils veulent capturer. S'ils marchent simultanément vers l'objectif, ils gagnent. Sinon, celui qui marche sera détruit. En raison de difficultés techniques, ils ne peuvent communiquer que via des messagers. Malheureusement, le monde est occupé par les défenseurs de la ville et il y a une chance qu'une méthode donnée envoyée à travers le mur soit capturée. Le problème est de trouver un protocole qui permette aux généraux de marcher ensemble même, par certaines méthodes, se perdre.



Il existe une simple preuve qu'aucun protocole de plans fixes n'existe. Pensez-y de la manière suivante. Le premier général envoie un message d'attaque à 5 heures du matin. Comment sait-il que le second général recevra le message? Le second général doit envoyer un message d'accusé de réception confirmé. Mais comment sait-il que le premier général recevra le message d'accusé de réception? Vous pouvez continuer indéfiniment, le cycle est fermé.

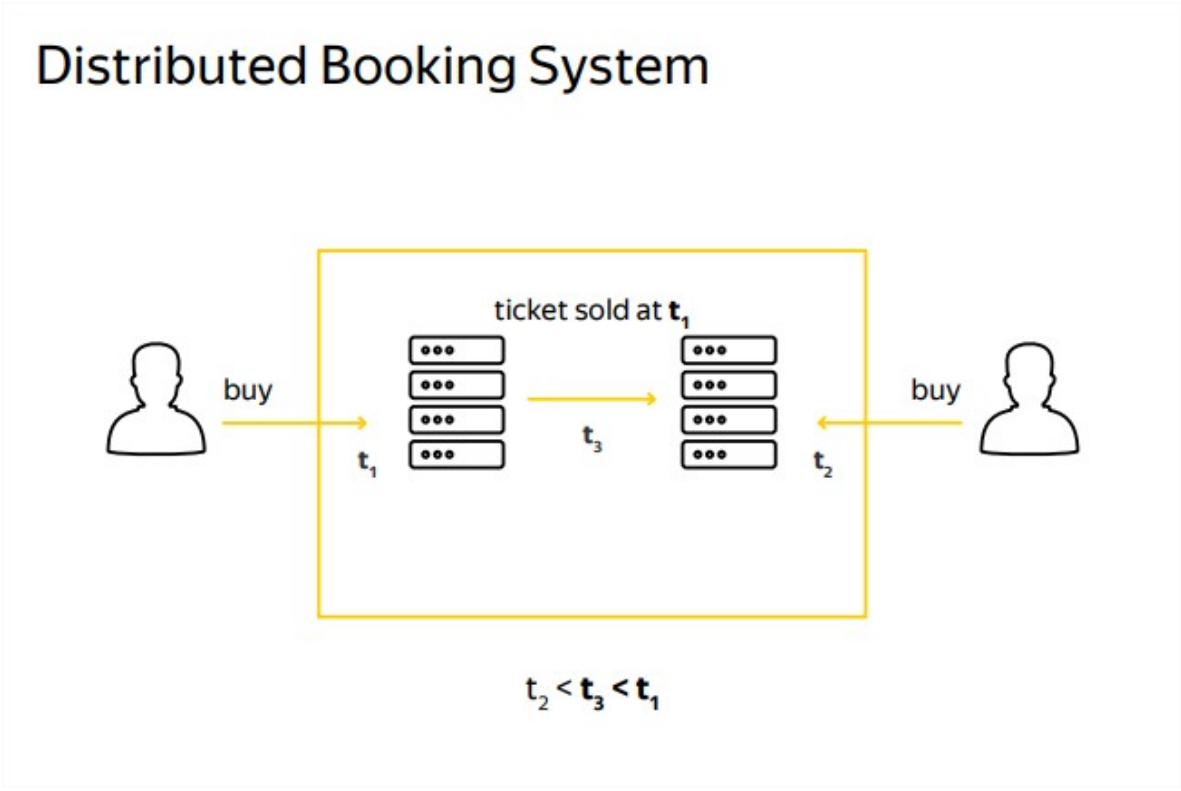
6.1.7 Horloge cassée

Imaginez que vous avez un système de réservation et qu'il ne reste qu'un seul billet. Vous avez deux clients qui ont essayé d'acheter un billet plus ou moins en même temps. Ensuite, chacun des nœuds de notre système distribué informe l'autre de la vente de billets. Qui était là en premier?

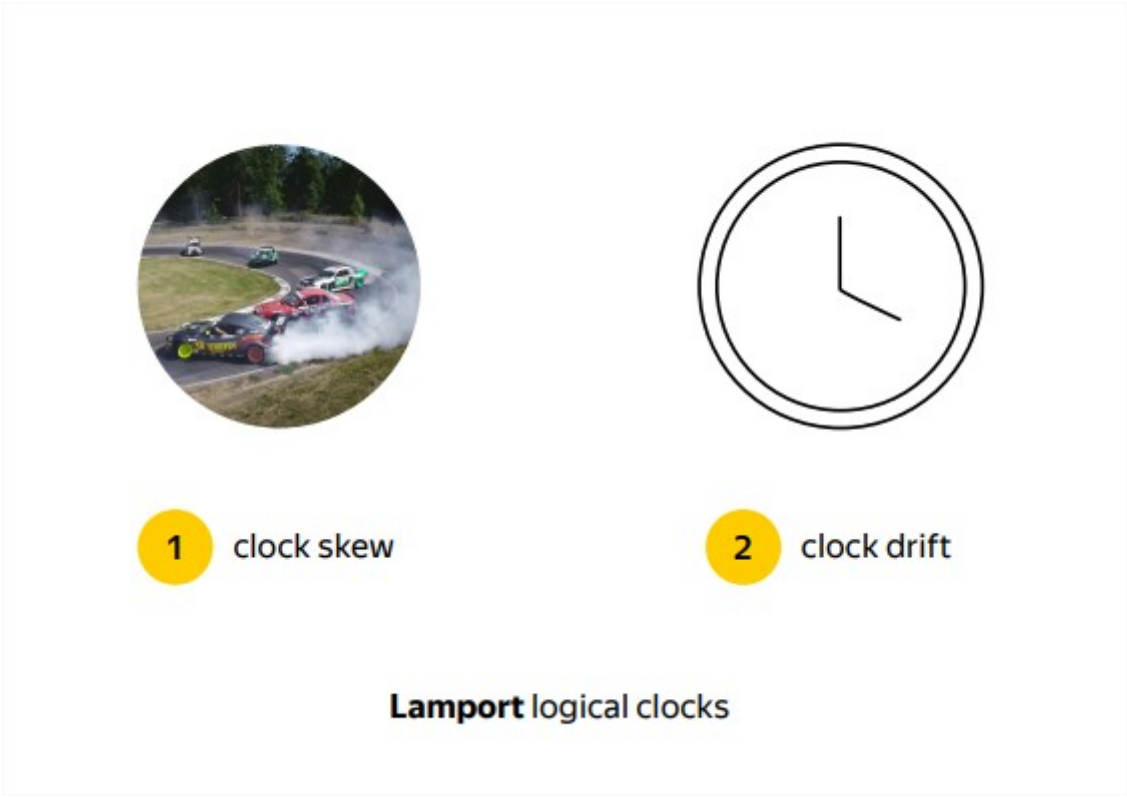


Vous pouvez créer un ID intuitif pour ajouter un horodatage Unix à l'heure de la requête. Parfois, cela ne fonctionnera pas correctement.

Supposons que vous ayez reçu une demande d'acheteur de l'utilisateur un à l'horodatage t_1 sur le nœud un et que vous ayez reçu une demande d'acheteur des utilisateurs deux à l'horodatage t_2 sur le nœud deux. Le nœud un a envoyé un message avec l'horodatage t_1 au nœud deux et il a été reçu par le nœud deux à l'horodatage t_3. En raison de la nature de la simulation d'horloge, il serait possible que l'heure locale t_3 sur le nœud deux soit inférieure à t_1 reçue du nœud un. Alors, qu'est-ce que vous avez ici?



Qui était vraiment là en premier pour acheter le billet dans le cas où les horodatages ne sont pas alignés? Qui s'en soucie maintenant? Maintenant, vous devez faire face aux conséquences.



Je voudrais vous donner une brève idée du problème de synchronisation d'horloge. Ce problème est double. Premièrement, l'heure peut être différente sur différentes machines. Cela s'appelle un décalage d'horloge. Il est similaire aux voitures situées à différents endroits sur une route circulaire. Deuxièmement, il peut y avoir une fréquence d'horloge différente appelée dérive(drift), pour atténuer le premier problème. Métaphoriquement, différentes voitures ont une vitesse de rotation différente sur cette image. Tout mécanisme de synchronisation d'horloge est soumis à une certaine précision. C'est pourquoi les horloges logiques ont été inventées.

Les horloges logiques aident à suivre les événements avant les événements et, par conséquent, à ordonner les événements pour créer des protocoles fiables. Les horloges logiques portent le nom de son inventeur, Leslie Lamport. Lamport est bien connu en tant que développeur initial du système de préparation de documents LaTex et il a été lauréat du prix Turing 2013 pour son travail dans les systèmes distribués. Si vous souhaitez acquérir plus de connaissances théoriques sur les systèmes distribués, je vous encourage à jeter un œil à son travail.

6.1.8 Systèmes synchrones/Asynchrones

Les systèmes peuvent aussi être divisés en synchrones et asynchrones. Dans les systèmes synchrones,

- Chaque paquet réseau doit être livré dans un délai limité
- La dérive d'horloge est limitée en taille
- Chaque instruction CPU est également limitée dans le temps

Au moins une de ces affirmations est toujours erronée dans les systèmes asynchrones.

6.2 MapReduce

Dans cette section, je voudrais vous montrer le monde des calculs MapReduce. MapReduce a été inventé par Jeffrey Dean et Sanjay Ghenawat. Et présenté au Symposium sur la conception et l'implémentation des systèmes d'exploitation en 2004. C'était juste un an après la publication de l'article sur le système de fichiers distribué de Google.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

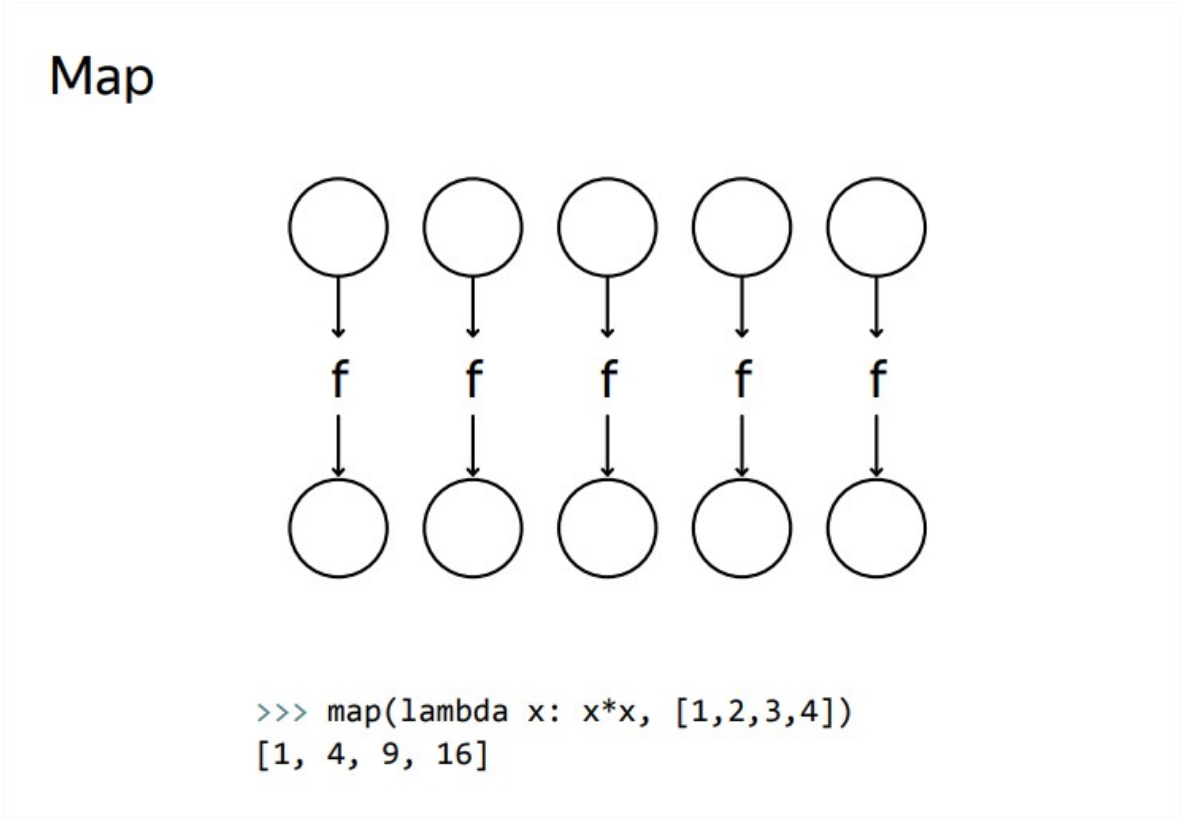
To appear in OSDI 2004

Il y a deux phases lors du calcul, Map(cartographier) et Reduce(réduire).

6.2.1 Map

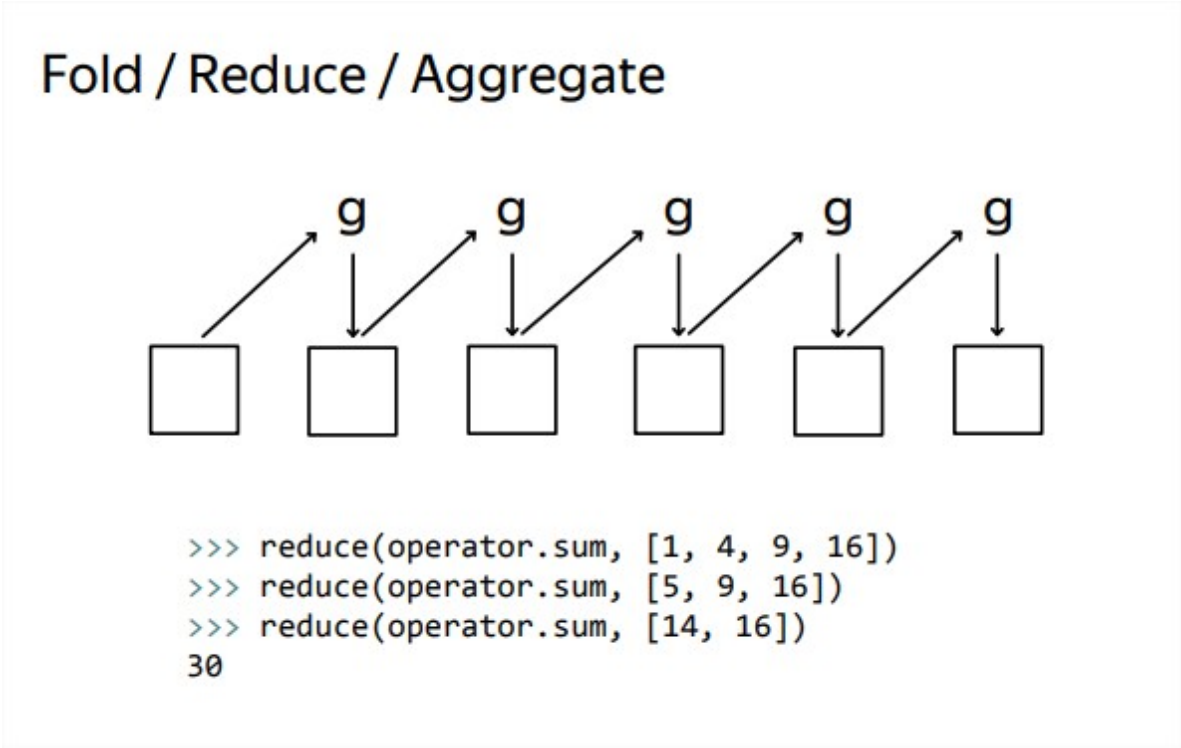
La première phase est appelée Map. Vous appliquez la même fonction à chaque élément de votre collection. En Python, vous avez même une fonction spéciale. Il vous serait donc facile de comprendre le principe principal. À quoi vous attendez-vous en mettant au carré les nombres de 1 à 4? 1, 4, 9 et 16, n'est-ce pas?

```
mapE = map(lambda x : x*x, (1,2,3,4))
print(list(mapE))
```



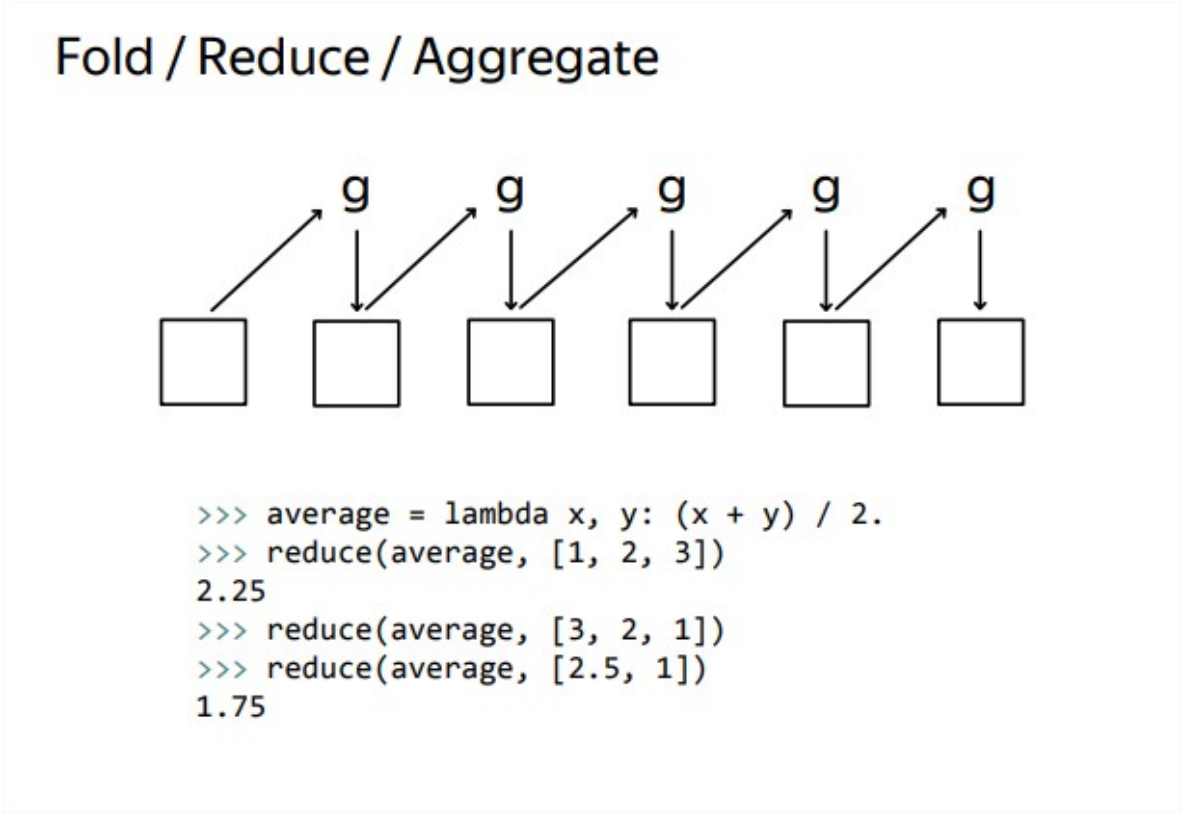
6.2.2 Reduce

La deuxième phase est appelée Reduce. Dans le monde de la programmation fonctionnelle, il aurait pu être nommé défaut ou agrégat. Encore une fois, il existe une fonction Python intégrée avec laquelle vous pouvez jouer pour mieux saisir le concept.
L'opérateur de Reduce provoque une séquence d'éléments en appliquant la procédure suivante de manière itérative. Dès que vous avez plus d'un élément dans la séquence, vous obtenez les deux premiers et les combinez en un seul élément en appliquant la fonction fournie. Si vous réduisez les éléments de séquence 1, 4, 9 et 16 avec la fonction somme, vous en obtiendrez 30. Faites attention à ne pas réduire les fonctions qui ne sont pas associatives.

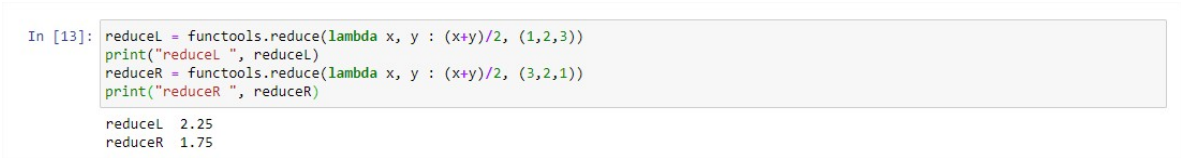



```
import functools
import operator
reduceR = functools.reduce(operator.add, (1,4,9,16))
print(reduceR)
```

Par exemple, la fonction moyenne n'est pas associative. Donc, si vous appliquez cela à une liste de nombres de 1 à 3, vous obtiendrez le résultat 2,25. Mais si vous appliquez la même fonction de droite à gauche, vous obtiendrez 1,75. Ainsi, changer l'ordre des atomes ne change pas la somme. Mais changer l'ordre des applications affecte définitivement le résultat.

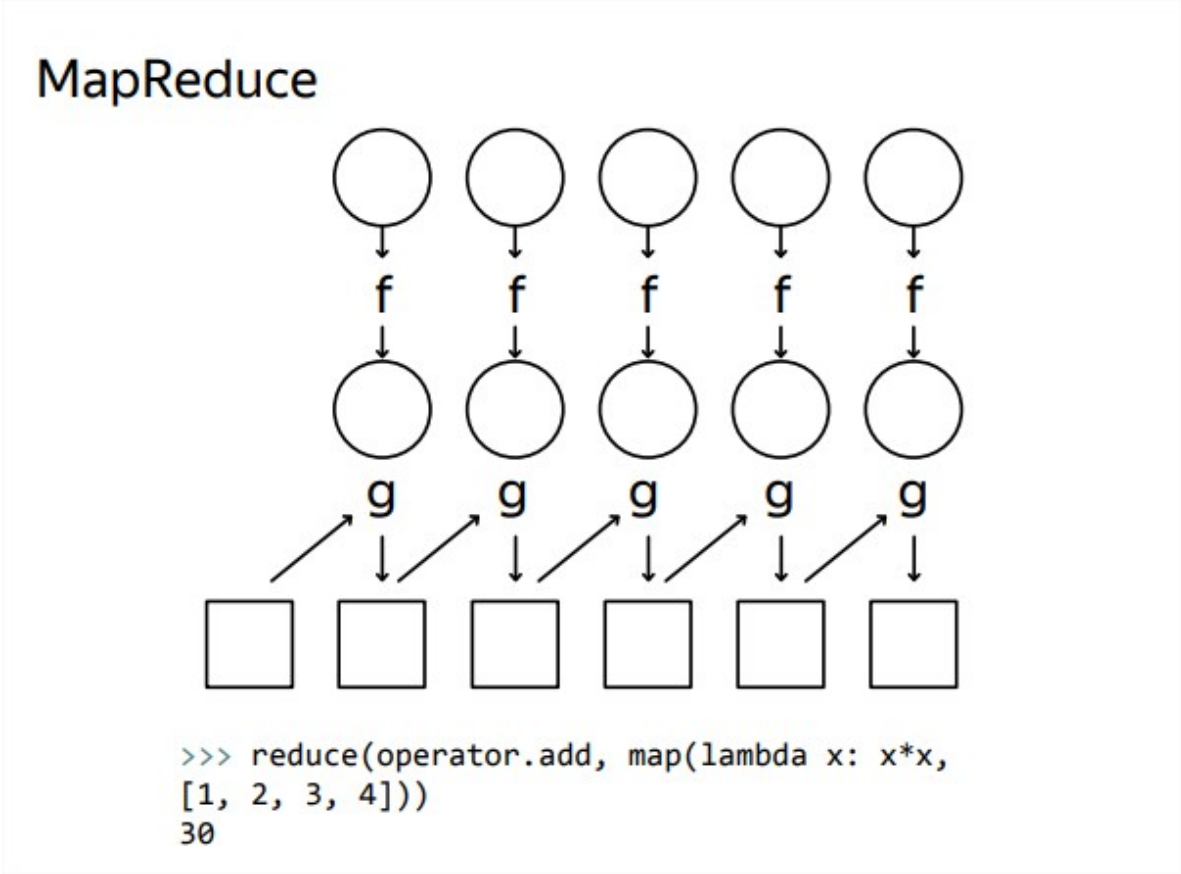



```
reduceL = functools.reduce(lambda x, y : (x+y)/2, (1,2,3))
print("reduceL ", reduceL)
reduceR = functools.reduce(lambda x, y : (x+y)/2, (3,2,1))
print("reduceR ", reduceR)
```



6.2.3 MapReduce

Lorsque vous combinez ces deux étapes ensemble, vous obtiendrez MapReduce. Il s'avère que la classe de problèmes que vous pouvez résoudre avec une carte arbitraire et des fonctions de réduction est assez grande.




```
MapReduce = functools.reduce(operator.add, map(lambda x : x*x, (1,2,3,4)))
print("MapReduce ", MapReduce)
```

```
In [20]: MapReduce = functools.reduce(operator.add, map(lambda x : x*x, (1,2,3,4)))
         print("MapReduce ", MapReduce)

         MapReduce  30
```

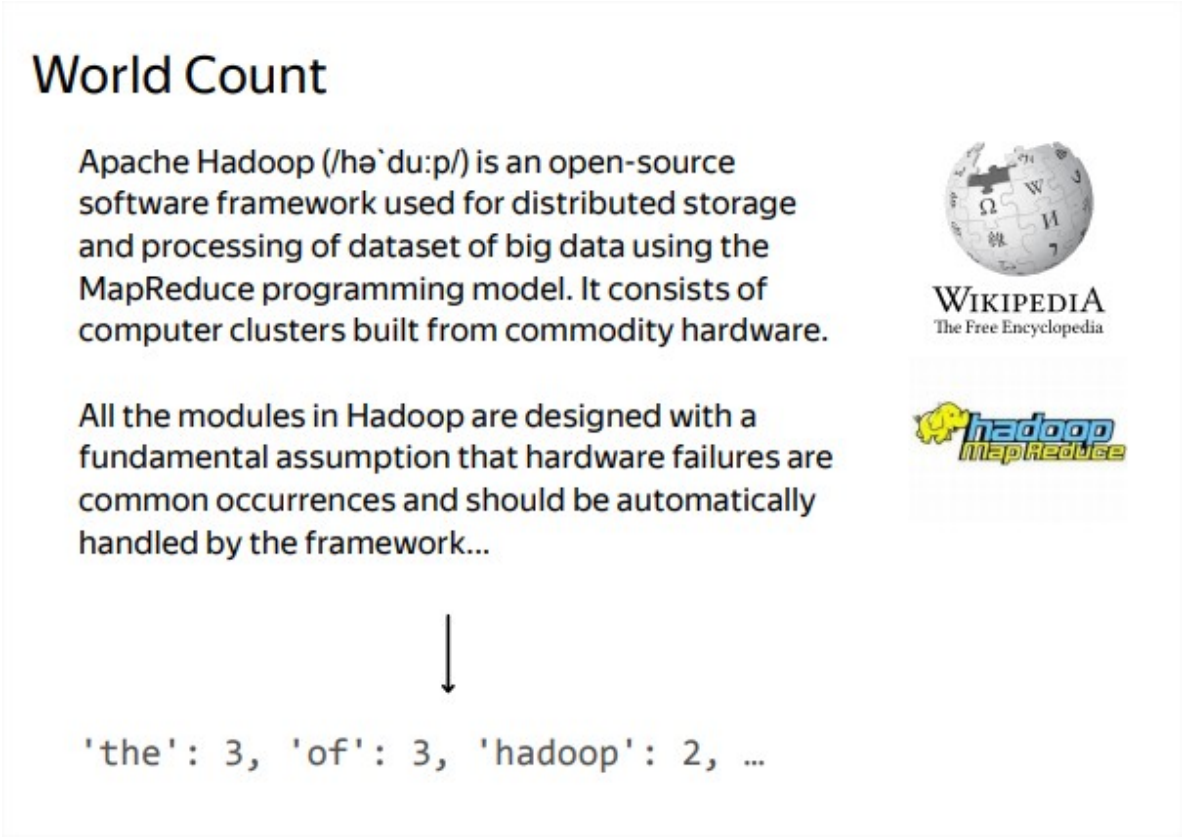
6.2.4 Exemple: Trouver les mots les plus populaires sur Wikipedia - Word Count

Syntaxe

uniq [options] [fichier_entree [fichier_sortie]]

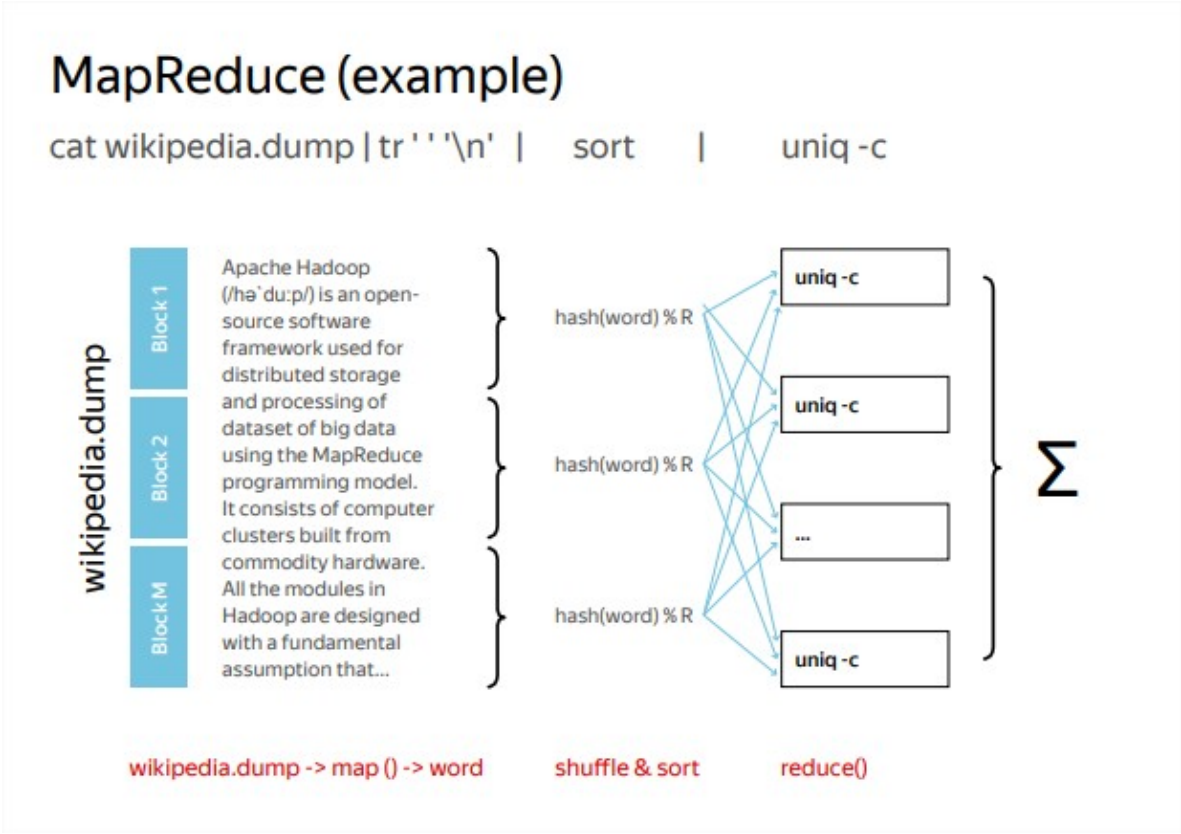
Principales options :
-d : Affichage des doublons
-c : Comptage des doublons

La première étape consiste à compter le nombre de fois où chaque mot apparaît dans un ensemble de données. Ce problème s'appelle, le nombre de mots.



Si vous disposez d'un petit ensemble de données, vous pouvez traiter les données sur un ordinateur avec quelques appels CLI simples. cat, tr, sort et uniq.

Ce sont les abréviations. Pour une application MapReduce distribuée, cat est utilisé pour lire les données. tr est une fonction de carte pour diviser le texte en mots, et uniq est évidemment une fonction réduite. Alors que devez-vous faire avec le tri? Est-ce une carte ou une réduction? D'une part, il ne peut pas s'agir d'un Map, car la map traite les éléments indépendamment. D'un autre côté, il ne peut pas non plus être réduit. Sinon, nous devrions être en mesure d'intégrer l'énorme ensemble de données dans un seul ordinateur. Pour cette raison, le framework MapReduce introduit une phase spéciale appelée shuffle and sort, entre les phases de map et de réduction. Permettez-moi de visualiser une version distribuée du nombre de mots pour vous.

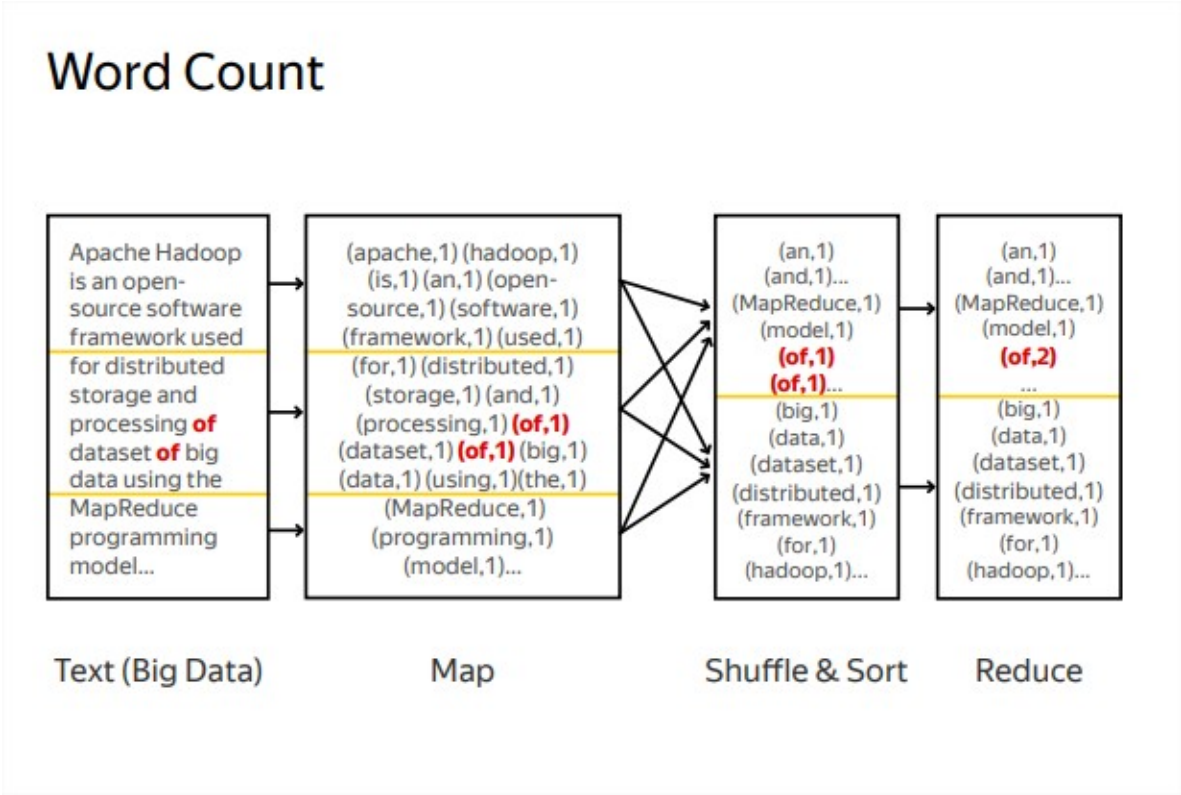


Pendant la phase Map, le texte est divisé en deux mots. Ensuite, pendant la phase de mélange et de tri, les mots sont distribués à une phase de réduction de manière à ce que les fonctions de réduction puissent être exécutées indépendamment sur différentes machines.

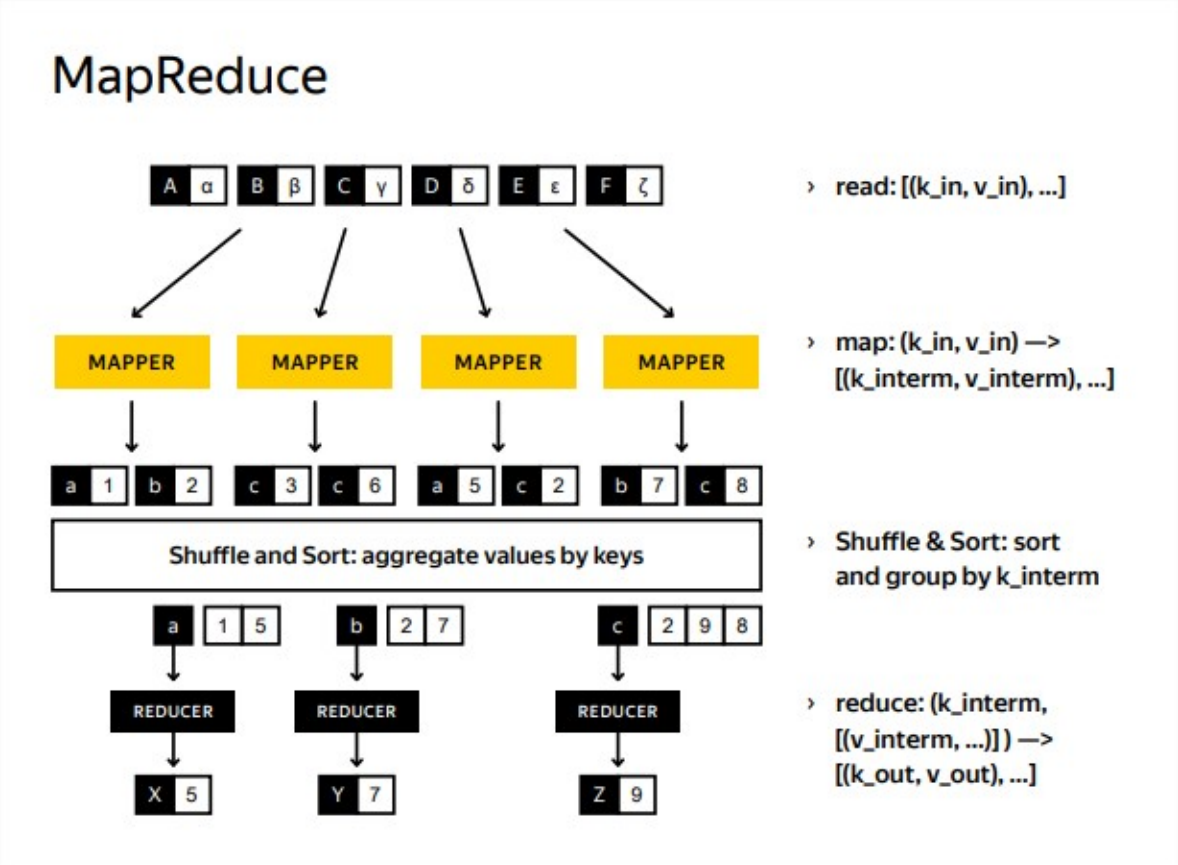
Pour l'application de comptage de mots, cela signifie que les mots sont distribués par un hachage de mot. Dans un exemple simple, si vous avez 26 réducteurs indépendants et uniquement des mots anglais, vous pouvez répartir leurs mots par ordre alphabétique de leur premier caractère de a à z. Comme vous pouvez le constater, il existe également un onglet de tri. Même si tous les mots sont répartis par ordre alphabétique, vous ne disposez peut-être pas de suffisamment d'espace RAM pour insérer ces données en mémoire. Mais si les données sont triées et peuvent être lues sous forme de flux, alors uniq moins c fonctionnera correctement.

Pour que les données soient triées, il vous suffit de disposer de suffisamment d'espace disque. L'algorithme pour cela s'appelle le tri externe. Ce n'est pas si difficile à mettre en œuvre.

Au lieu d'avoir des fonctions arbitraires avec des entrées et des sorties arbitraires dans le modèle MapReduce, les gens ont décidé de faire un certain niveau d'abstraction qui s'est avéré être pratique. Toutes les entrées et sorties des fonctions de mappage et de réduction doivent être une paire valeur / clé. Permettez-moi de transformer l'exemple de comptage de travail précédent en ce modèle.



Tout d'abord, vous lisez des données et obtenez des paires avec un numéro de ligne et un contact de ligne. Ensuite, lors d'une phase cartographique, vous ignorez les numéros de ligne et divisez les lignes en mots. Pour satisfaire ce modèle, vous pouvez ajouter une valeur à chaque sortie qui a fonctionné. Donc, cela signifie que vous avez vu ce mot une fois en lisant une ligne de gauche à droite. Ensuite, une phase de mélange et de tri où vous étalez les mots par les hachages. Ainsi, vous pouvez les traiter sur des réducteurs indépendants. Enfin, vous comptez le nombre de chiffres de un que vous avez pour chaque mot et vous les affichez pour obtenir une réponse. Cette ligne le montrera de manière éclatante. Plus généralement, vous disposez de trois types de paires valeur / clé. Paires de valeurs-clés pour les données d'entrée, paires de valeurs-clés pour les données intermédiaires et paires de valeurs-clés pour les données de sortie.



En plus, vous lisez des blocs de données d'entrée. Ces blocs sont traités par des mappeurs et ont des paires valeur / clé intermédiaires à partir de là. Ensuite, les données sont agrégées par des clés intermédiaires et fournies aux réducteurs. Enfin, les données sont transformées par des réducteurs et peuvent être stockées sur les disques locaux d'un système distribué.