Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Verteile Systeme: Synchronisation und Gruppenkommunikation

Übergang von eng zu lose gekoppelten Systemen

Wo liegen die Unterschiede in der Prozesskommunikation zu eng gekoppelten SM Systemen?

Wie geschieht die Synchronisation bei Fehlern und Asynchronität?

Map & Reduce

- Ein Web-Programmiermodell für die skalierbare Datenverarbeitung in großen Clustern über große Datenmengen basierend auf impliziter Master-Worker Gruppenkommunikation (aber via peer-to-peer Nachrichten).
- Das Modell wird häufig in Web-Scale-Search- und Cloud-Computing-Anwendungen eingesetzt.
- Aber auch funktionale und parallele Programmierung macht von MapReduce Methoden Gebrauch

- **Methode**:
  - Es wird eine Map-Funktion angegeben, um eine Gruppe von Schlüssel/Wert-Zwischenpaaren zu generieren.
  - Dann wird eine Reduce-Funktion auf diesen Paaren angewendet, um alle Zwischenwerte mit demselben Zwischenschlüssel zusammenzuführen.

- MapReduce ist hochgradig skalierbar, um hohe Parallelitätsgrade auf verschiedenen Arbeitsebenen zu erreichen.

Map & Reduce

- Ein typischer MapReduce-Berechnungsprozess kann Terabytes an Daten auf Zehntausenden oder mehr Client-Computern verarbeiten. Hunderte von MapReduce-Programmen können gleichzeitig ausgeführt werden.
  - Tatsächlich werden jeden Tag Tausende von MapReduce-Jobs in Clustern von Google ausgeführt.
  - Das Hadoop Framework bietet für WEB Anwendungen MapReduce Services an → Master-Slave Architektur!

Die Map-Funktion verarbeitet ein Paar (Schlüssel, Wert) und gibt eine Liste von Zwischenpaaren (Schlüssel, Wert) zurück:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

Die Reduzierungsfunktion führt alle Zwischenwerte zusammen, die die gleichen Zwischenschlüssel haben:

$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

Map & Reduce

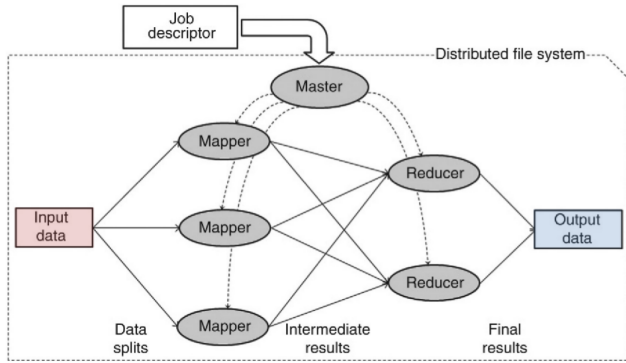


Abb. 1. Ausführungsphasen einer generischen MapReduce Applikation

Map & Reduce

MapReduce Phasen

1. Ein Master-Prozess erhält einen Jobdeskriptor, der den auszuführenden MapReduce-Job angibt. Der Jobdeskriptor enthält neben anderen Informationen den Ort der Eingabedaten, auf die unter Verwendung eines verteilten Dateisystems zugegriffen werden kann.

2. Gemäß dem Jobdeskriptor startet der Master eine Anzahl von Mapper- und Reducer-Prozessen auf verschiedenen Maschinen. Gleichzeitig startet es einen Prozess, der die Eingabedaten von seinem Speicherort liest, diese Daten in eine Gruppe von Aufteilungen unterteilt und diese Aufteilungen an verschiedene Zuordner verteilt.

3. Nach dem Empfang seiner Datenpartition führt jeder Zuordnungsvorgang die Zuordnungsfunktion aus (die als Teil des Jobdeskriptors bereitgestellt wird), um eine Liste von Zwischenschlüssel / Wert-Paaren zu erzeugen. Dann werden diese Paare auf der Basis ihrer Schlüssel gruppiert.

Map & Reduce

4. Alle Paare mit den gleichen Schlüsseln sind dem gleichen Reduziervorgang zugeordnet. Daher führt jeder Reduzierprozess die Reduktionsfunktion (definiert durch den Jobdeskriptor) aus, die alle Werte vereinigt, die mit dem gleichen Schlüssel assoziiert sind, um einen möglicherweise kleineren Satz von Werten zu erzeugen.

5. Dann werden die von jedem Reduktionsprozess erzeugten Ergebnisse gesammelt und an einen durch den Jobdeskriptor spezifizierten Ort geliefert, um die endgültigen Ausgabedaten zu bilden.

Map & Reduce

Beispiel

```
local options = {workers = 2}
local data = {34,35,36,37,38,39,40,41}

local function worker (id,set)
  local results = T{}
  for i = 1,#set do
    results:push(fib(set[i]))
  end
  return results
end
-- Parallel Processing
local Parallel = require('parallel')
local p = Parallel:new(data,options)
function sum (x,y) return x+y  end
p:time():
  map(worker):
  apply(function (r) print(r:print()) end):
  reduce(sum):
  apply(print):
  time()
```

Gruppenkommunikation

- Die bisherigen Synchronisationsmethoden basierten auf einem sicheren Lock Objekt und Master-Worker Hierarchien
- Verteilte Systeme sind i.A. strikt asynchron und können nicht direkt synchronisiert werden
- Der verteile mutuale Ausschluss ist zentrales Problem in verteilten Systemen
- In verteilten Systemen kann es lose gekoppelte Gruppen aus Prozessen geben und es muss zunächst ein Master/Leader gewählt werden!

MPI

Message Passing Interface: MPI

Ziele und Eigenschaften

- Anwendungsprogrammierschnittstelle (nicht unbedingt für Compiler oder eine Systemimplementierungsbibliothek).
- Effiziente Kommunikation:
  - Vermeidung von Arbeitsspeicher-Arbeitsspeicher Kopien
  - Überlappung von Berechnung und Kommunikation
  - Auslagerung auf Kommunikations-Koprozessoren, soweit verfügbar.

MPI

- Implementierungen, die in einer heterogenen Umgebung verwendet werden können (verschiedene Hostplattformen).
- Einfache Einbindung in Programmiersprachen und Bibliotheken und Plattformunabhängigkeit
- Zuverlässige Kommunikation: Nutzer/Programmierer muss sich nicht um Kommunikationsfehler kümmern

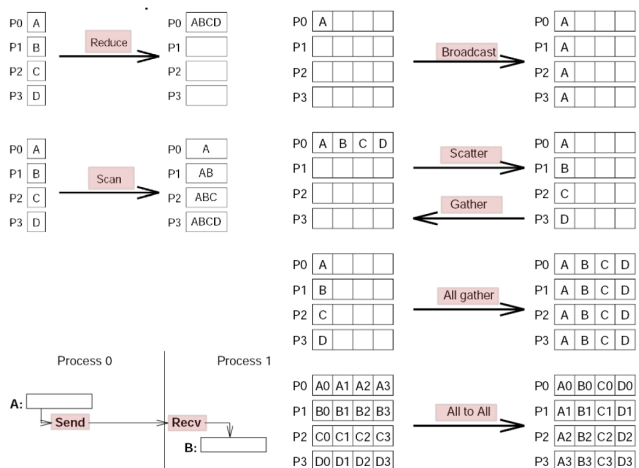
MPI

API

- Point-to-point communication,
- Datatypes,
- Collective operations,
- Process groups,
- Communication contexts,
- Process topologies,
- Environmental management and inquiry,
- The Info object,
- Process creation and management,
- One-sided communication,
- External interfaces,
- Parallel file I/O,

MPI

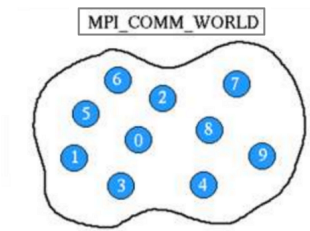
Operationen



MPI

Communicator

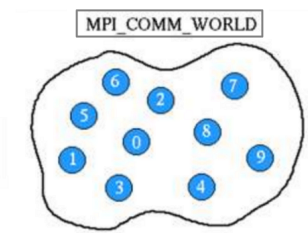
- Kommunikationswelt mit einer Gruppe aus Prozessen die gemeinsam Nachrichten austauschen können
- Nachrichten können nur innerhalb der Kommunikationswelt ausgetauscht werden
- MPI_COMM_WORLD ist die Standardwelt



MPI

Rank (Rang)

- Einheitliche Prozessnummer innerhalb einer Kommunikationswelt
- Werden vom System bei der Initialisierung vergeben und werden fortlaufend ab 0 nummeriert
- Rank IDs werden bei der Kommunikation zur Identifikation von Empfänger und Sender verwendet (Kommunikationsendpunkte)
- Rank IDs dienen zur Programmdifferenzierung (*if rank==0 then do this else do that*)

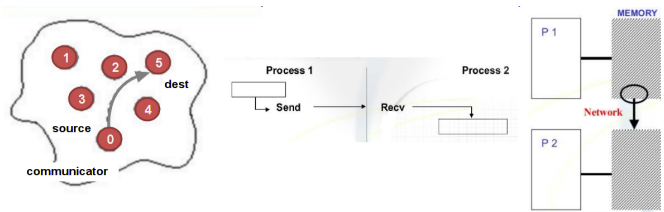


MPI

Point-to-Point Kommunikation

- Kommunikation zwischen zwei Prozessen
- Quellprozess sendet eine Nachricht (Typ, Daten) an einen Zielprozess unter Angabe der Rank ID
- Kommunikation kann nur innerhalb eines Communicators stattfinden

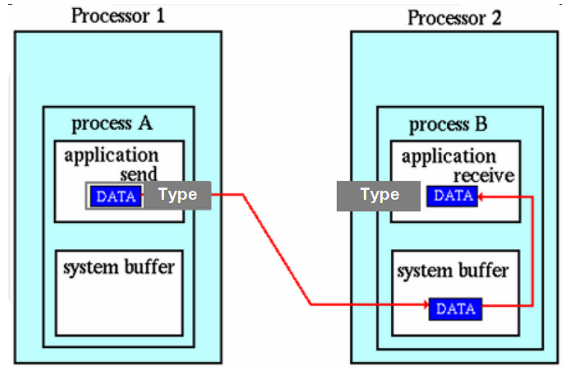
```
MPI.send(dest:number,message:{type:string,content:string})
```



MPI

- Damit der Zielprozess die Nachricht empfangen kann muss er einen Handler für den entsprechenden Nachrichtentyp einrichten:

```
MPI.recv(type:string,callback:function (message))
```



MPI

Broadcast Kommunikation

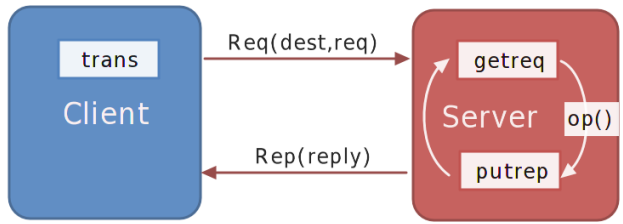
- Ein Quellprozess kann eine Nachricht an alle Prozesse innerhalb eines Communicators senden

```
MPI.broadcast(message:{type:string,content:string})
```

RPC

Remote Procedure Call Interface

- Ähnlich MPI
- Klienten-Server Architektur
- Es gibt drei Operationen:
  - *getreq* → Server; Auf eine Anfrage warten
  - *putrep* → Server: Eine Annfrage beantworten
  - *trans* → Klient: Eine Anfrage senden und auf Antwort warten



RPC

LUA API (CSP)

- Die Kommmunikation findet über das IP-UDP/TCP Protokoll statt.
- Daten werden serialisiert und es können beliebige Daten übertragen werden (inkl. seitenfreier Funktionen)

Rpc(options?:{}) → rpc
Erzeugt eine RPC Instanz (für Klient und Server)

rpc:getreq(ip:string,port:number,callback:function (req) → rep)
Server Handlerfunktion (*putrep* wird implizit mit dem Rückgabewert des Callbackhandlers ausgeführt)

rpc:trans(ip_string,port:number,request:*) → reply:*
Klienten Transaktion

RPC

Beispiel

```
// Server
require('Csp')
local rpc = Rpc({verbose=2})
rpc:getreq('127.0.0.1',12345,function (req)
  if req.cmd=='iabs' then
    return {
     result=math.sqrt(math.pow(req.x,2)+
                      math.pow(req.y,2)),
     stat='OK'
    }
  else return {stat='EINVALID'} end
end)
loop.start()
```

```
// Client
require('Csp')
local rpc = Rpc({verbose=2})
local stat,reply = rpc:trans('127.0.0.1',12345,
                             {cmd='iabs',x=1,y=2})
print(reply)
```

Tupelräume

- Tupel-Räume stellen ein **assoziiertes Shared-Memory-Modell** dar, wobei die gemeinsam genutzten Daten als **Objekte** mit einer Reihe von **Operationen** betrachtet werden, die den Zugriff der Datenobjekte unterstützen
- Tupel sind in **Räumen** organisiert, die als abstrakte Berechnungsumgebungen betrachtet werden können.
- Ein Tupelraum verbindet verschiedene Programme, die **verteilt** werden können, wenn der Tupel-Space oder zumindest sein operativer Zugriff verteilt ist.
    - Oder: **Mobile Rechner** als Tupel Verteiler!
- Das Tupelraum Organisations- und Zugangsmodell bietet **generative Kommunikation**, d.h. Datenobjekte können in einem Raum durch Prozesse mit einer Lebensdauer über das Ende des Erzeugungsprozesses hinaus gespeichert werden.
- Ein bekanntes Tupelraum-Organisations- und Koordinationsparadigma ist **Linda** [GEL85].

Tupelräume

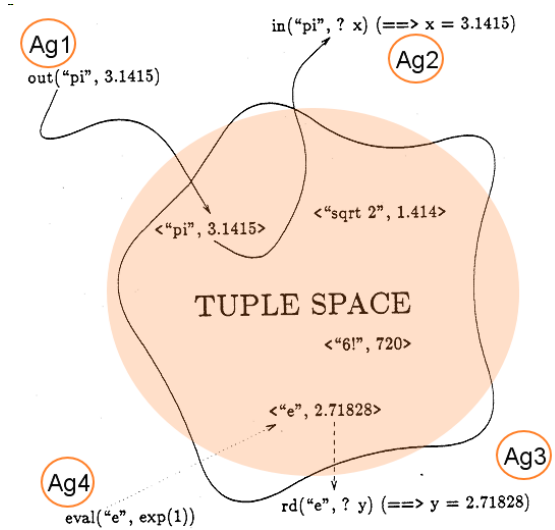


Abb. 2. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen [11]

Tupelräume

- Kommunikation von Sensorknoten über Tupelräume ist eine **Koordionierungssprache**.

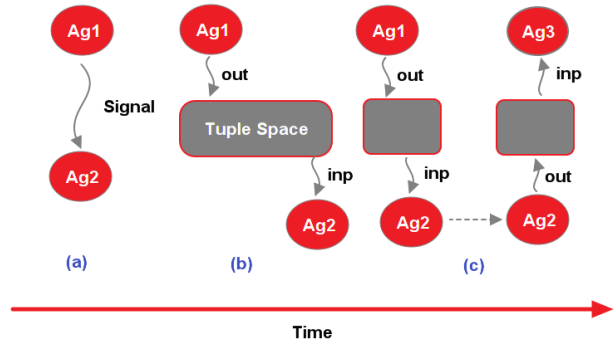


Abb. 3. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Sensorknoten)

Tupelräume - Datenmodell

- Die Daten sind mit Tupeln organisiert.
- Ein Tupel ist eine lose gekoppelte Verbindung einer beliebigen Anzahl von Werten beliebiger Art /Typ/
- Ein Tupel ist ein Wert und sobald es in einem Tupelraum gespeichert ist, ist es persistent.
- Tupeltypen ähneln den Datenstrukturtypen, sie sind jedoch dynamisch und können zur Laufzeit ohne statische Beschränkungen erstellt werden.
- Auf die *Elemente von Tupeln* kann nicht direkt zugegriffen werden, was üblicherweise Mustererkennung und *musterbasierte Dekomposition* erfordert, im Gegensatz zu Datenstrukturtypen, die einen benannten Zugriff auf Feldelemente bieten, obwohl die Behandlung von Tupeln als Arrays oder Listen diese Beschränkung lösen kann.
- Ein Tupel mit *n* Feldern heißt *n*-stellig und wird in der Notation $<v_1, v_2, ..>$ angegeben.

Tupelräume - Datenmodell

Beispiele von Tupel

```
<'SENSOR',1000>
<'SENSOR2',[10,100,2]>
<1,3,5,7,11>
<'SLEEPMODE',True,2500.34>
<0,'OFF'>
<1,'ON'>
```

- Formal werden Tupel als **Vektoren** durch die folgende Generierungsregel mit *Werten v*, *Ausdrücken* ε und *Variablen x* definiert, die als tatsächliche Parameter betrachtet werden (d.h. Variablen *x*, die mit Wertsemantik verwendet werden):

$$t = \left\langle \vec{d} \right\rangle, \text{with } \vec{d} ::= d | d, \vec{d} \text{ and } d ::= v | \varepsilon | x$$

Tupelräume - Datenmodell

- Tupelwerte erfordern einen **Mustervergleich** basierend auf dem *Vorlagenmuster* mit der folgenden Generierungsregel, bestehend aus tatsächlichen (*v*, ε, *x*) und formalen Parametern (*x*?, Variablen, die mit Referenzsemantik verwendet werden):

$$p = \left\langle \overrightarrow{dt} \right\rangle, \text{with } \overrightarrow{dt} ::= dt | dt, \overrightarrow{dt} \text{ and } dt ::= v | \varepsilon | x | x? | \perp$$

- Ein Suchmuster kann ein Wildcard (⊥) anstelle von formalen Parametern verwenden.
- Jedes Tupel *t* hat eine Typsignatur Sig (*t*) = S_t = <T_1; T_2; ...; T_n>, ein Tupel mit der gleichen Stelligkeit wie *t*, das den Typ jedes Tupelfeldes angibt.
- Ein Tupel kann nur durch seine Verknüpfung mit Templates *p* angesprochen werden.

Tupel Räume - Datenmodell

- Üblicherweise wird das **erste Feld** eines Tupels als symbolischer **Schlüssel** behandelt, der eine Tupelunterklasse identifiziert, indem Textzeichenfolgen oder aufgezählte symbolische Konstantenwerte verwendet werden.

Mustersuche

Sei $t = <d_1, d_2, .., d_n>$ ein Tupel, $p = <dt_1, dt_2, .., dt_m>$ eine Vorlage; dann wird t durch p abgedeckt (bezeichnet durch match (t, p) = true), wenn die folgenden Bedingungen gelten: (i) m = n. (ii) $\forall\ dt_i = d_i$ oder $dt_i = \perp$, $1 \leq i \leq n$. Bedingung (1) prüft, ob t und p die gleiche Stelligkeit haben, während (2) prüft, ob jedes Nicht-Wildcard-Feld von p gleich ist dem entsprechenden Feld von t.

Def. 1. Mustersuche

Tupelräume - Operationale Semantik

- Es gibt eine Reihe von Operationen, die von Prozessen angewendet werden können, bestehend aus
  - einer Reihe reiner Datenzugriffsoperationen, die Tupel als passive Datenobjekte behandeln,
  - und Operationen, die Tupel als eine Art von aktiven Rechenobjekten behandeln (genauer gesagt, zu berechnende Daten).
  - RPC-Semantik (Remote Procedure Call).

out(*t*)
Die Ausführung der Ausgabeoperation fügt das Tupel *t* in den Tupelraum ein. Mehrere Kopien desselben Tupelwerts können eingefügt werden, indem die Ausgabeoperation iterativ angewendet wird. Die gleichen Tupel können nach dem Einfügen in den Tupelraum nicht unterschieden werden.

Beispiel: `out("Sensor",1,100); out("Sensor",2,121);`

Tupelräume - Operationale Semantik

inp(*p*)

Die Ausführung der Eingabeoperation entfernt ein Tupel *t* aus dem Tupelraum, der der Mustervorlage *p* entspricht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiel: `inp("Sensor",1,s1?); inp("Sensor",i?,s?);`

rd(*p*)

Die Ausführung der Leseoperation gibt eine Kopie eines Tupels *t* zurück, dass der Vorlage *p* entspricht, entfernt sie jedoch nicht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiele: `rd("Sensor",1,s1?); rd("Sensor",i?,s?);`

Tupelräume - Operationale Semantik

inp?(*p*), rd?(*p*)
: Nichtblockierende Version von *inp*/*rd*. Wird kein passendes Tupel gefunden wird die Operation ergebnislos terminiert.

Beispiel: `res:=inp?('SENSOR',a?,b?);`

inpw?(*tmo*,*p*), rdw?(*tmo*,*p*)
: Teilblockierende Version von *inp*/*rd*, Wird einer Zeit von *tmo* kein passendes Tupel gefunden wird die Operation abgebrochen.

Beispiel: `res:=inpw?(1000,'SENSOR',a?,b?);`

- Die Verwendung von zeitlich unbegrenzt blockierenden Operationen kann unter Betrachtung der Lebendigkeit von Agenten nachteilig sein. Daher sollte immer eine zeitliche Begrenzung und anschließende Abfrage des Operationsstatus erfolgen (abgebrochen?)

Tupelräume - Operationale Semantik

test(t), testandset(*p*,function (*t*)→*t*)

Nicht blockierender Test eines Tupels und atomare Veränderung eines Tupels, dass der Vorlage *p* entspricht. Das zweite Argument ist eine Abbildungsfunktion. Das Ergebnistupel ersetzt das ursprüngliche.

Markierungen

- Tupel sind persistent und können für immer in einem Tupelraum verbleiben!
- Daher ist die Verwendung von *Markierungen* häufig sinnvoll.
- Eine Markierung ist ein Tupel mit einer Lebenszeit τ
- Nach Ablauf der Lebenszeit wird das Tupel - sofern es nicht entfernt wurde - durch einen Garbagecollector entfernt.

$$m = \left\langle \tau, \overrightarrow{d} \right\rangle, \text{with } \overrightarrow{d} ::= d | d, \overrightarrow{d} \text{ and } d ::= v | \varepsilon | x,\ \tau : \text{timeout}$$

mark(*tmo*,*t*)

Ausgabe eines Tupels *t* mit einer Lebenszeit τ (im Tupelraum).

Tupelräume - Synchronisationsmodell

- Es gibt **Produzenten- (Generator) und Verbraucherprozesse**.

1. Ein *Produzent* erzeugt ein Tupel, das von einem Konsumentenprozess entfernt werden kann.
    - Die Tupelausgabeoperation endet unmittelbar (asynchron), alternativ nachdem das Tupel im Tupelraum gespeichert wurde (synchron).
2. Ein Verbraucher-Prozess wird blockiert, wenn die Anfrage nicht bearbeitet werden kann, wenn im Tupel-Bereich tatsächlich kein passendes Tupel vorhanden ist.
3. Nachdem ein übereinstimmendes Tupel im Tupelraum gespeichert wurde, wird es sofort einen der wartenden Verbraucherprozesse zugewiesen.

- Daher ist die Eingabeoperation immer synchron. Einzige Ausnahme sind die nicht permanent blockierenden Versionen, die das Warten auf eine obere Zeitgrenze begrenzen (Timeout).
- Es gibt keine anfängliche zeitliche Anordnung von Erzeuger- und Verbraucheroperationen.

Tupelräume - Beispiele von Operationen in Lua

```
out({'SENSOR',10,20});
out({'SENSOR',9,23{);
out({'PI',3,14{);
out({'DATA',{1,2,3,4}});
result = inp({'SENSOR',nil,nil});
>> { 'SENSOR', 9, 23 }
result = inp({'SENSOR',nil,nil});
>> { 'SENSOR', 10, 20 }
inp({'SENSOR',nil,nil});
>> nil
rd({nil,nil})
>> { 'DATA', { 1, 2, 3, 4 } }
rd({nil,nil,nil,nil},100)
>> nil -- Timeout
```

Sicherheit und Lebendigkeit

- Ein Algorithmus, z.B. die Wahl eines Leaders in einer Prozessgruppe, gelte als **sicher** wenn *maximal ein Leader* unter allen Umständen gewählt wird. Dies ist auch der Fall wenn einzelne Prozesse der Prozessgruppe fehlerhaft sind (nicht erreichbar sind oder terminiert sind).

- Ein Algorithmus, z.B. die Wahl eines Leaders in einer Prozessgruppe, gelte als **lebendig** wenn *irgendwann mindestens ein Leader* unter allen Umständen gewählt wird.

Robustheit und Fehler

- Angenommen eine Prozessgruppe bestehe aus *N* Prozessen die in einem Netzwerk verteilt sind.
- Die Netzwerkknoten sind miteinander verbunden.
- Nun wird aufgrund einer technischen Störung das Netzwerk partitioniert (z.B. in zwei getrennte Bereiche geteilt).

Sicherheit und Lebendigkeit

- Der verteilte Algorithmus wird in jeder der Partitionen unabhängig arbeiten. Dann ist das verteilte System zwar noch lebendig (es werden unabhängig zwei Leader in den Gruppenpartitionen gewählt), aber nicht mehr sicher !!!!
  - Die Invariante des Algorithmus ist verletzt worden durch Ausfall/Störung!

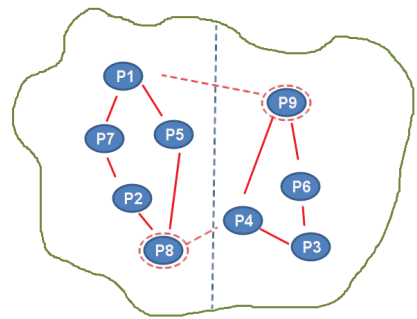


Abb. 4. Leaderelection: Eine ursprünglich zusammenhängende Gruppe wird durch Netzwerkpartitionierung (Störung der Kommunikation) zweigeteilt und es werden jetzt zwei Leader gewählt!

Mutualer Ausschluss

Verteilter Algorithmus nach Lamport

LA1: Um einen kritischen Bereich zu erlangen (Mutex Acquire), sendet ein Prozess eine zeitgestempelte Anforderung an jeden anderen Prozess im System und fügt die Anforderung auch in seiner lokalen Queue *Q* hinzu.

LA2: Wenn ein Prozess eine Anforderung empfängt, wird sie in *Q* platziert. Wenn sich der Prozess nicht in seinem *CS* befindet, sendet er eine zeitgestempelte Bestätigung an den Absender. Andernfalls wird das Senden der Bestätigung bis zum Verlassen des *CS* verzögert (Mutex Release).

Mutualer Ausschluss

LA3: Ein Prozess tritt in seinen *CS* ein, wenn (1) seine Anfrage vor allen anderen Anfragen (d.h. der Zeitstempel seiner eigenen Anfrage ist kleiner als die Zeitstempel aller anderen Anfragen) in seinem lokalen *Q* angeordnet ist und (2) Es hat die Antworten von jedem anderen Prozess als Antwort auf seine aktuelle Anfrage erhalten.

LA4: Um die *CS* zu verlassen, löscht ein Prozess (1) die Anfrage von seiner lokalen Warteschlange *Q* und (2) sendet eine zeitgestempelte Freigabenachricht an alle anderen Prozesse.

LA5: Wenn ein Prozess eine Freigabenachricht erhält, entfernt er die entsprechende Anforderung aus seiner lokalen Warteschlange Q.

Mutualer Ausschluss

Verteilter Algorithmus nach Ricart–Agrawala'

- Verbesserte Version

RA1: Jeder Prozess, der den Eintritt in seinen *CS* anfordert (Mutex Acquire), sendet eine zeitgestempelte Anfrage an jeden anderen Prozess im System.

RA2: Ein Prozess, der eine Anforderung empfängt, sendet eine Bestätigung an den Absender zurück, nur wenn

- (1) der Prozess nicht an dem Eintritt in seinen *CS* interessiert ist (Mutex Acquire) oder
- (2) der Prozess versucht, seine *CS* zu erlangen, aber sein Zeitstempel größer ist als der des Absenders.
- Wenn sich der Prozess bereits in seinem *CS* befindet (besitzt den Lock) oder sein Zeitstempel kleiner als der des Absenders ist, puffert er alle Anforderungen bis zum Verlassen des *CS*.

Mutualer Ausschluss

RA3: Ein Prozess tritt in seine *CS* ein, wenn er von jedem der verbleibenden (n-1) Prozesse eine Bestätigung erhält.

RA4: Nach dem Verlassen seiner *CS* muss ein Prozess eine Rückmeldung an jede der ausstehenden Anfragen senden, bevor er eine neue Anfrage macht oder andere Aktionen ausführt.

Mutualer Ausschluss

Token Algorithmen

- Eine andere Klasse verteilter mutualer Ausschlussalgorithmen verwendet das Konzept eines expliziten variablen Tokens, das als eine Erlaubnis für den Eintritt in die CS dient (Mutex Acquire) und von einem anfordernden Prozess durch das Prozesssystem weitergereicht werden kann.
- Immer wenn ein Prozess in seinen CS eintreten möchte, muss er den Token erwerben. Der erste bekannte Algorithmus, der zu dieser Klasse gehört, ist auf Suzuki und Kasami zurückzuführen.
- Da es nur ein Token gibt ist der mutuale Ausschluss (Sicherheit) garantiert. Die Lebendigkeit aber nicht unbedingt (Verlust des Tokens).

Mutualer Ausschluss

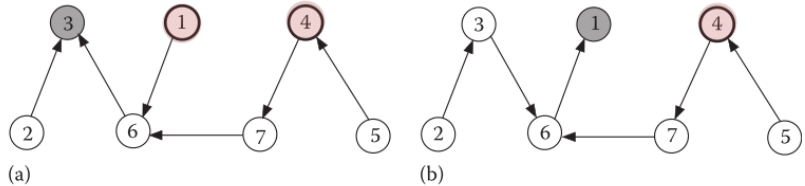


Abb. 5. Prozess 3 hält den Token und Prozesse 1 und 4 fordern ihn (über 6) an; schließlich erhält ihn 4

Verteilter Konsens

- Ein verteilter Konsensalgorithmus hat das Ziel in einer Gruppe von Prozessen oder Agenten eine gemeinsame Entscheidung zu treffen
- Zentrale Eigenschaften:
  - Zustimmung/Übereinstimmung
  - Terminierung; Lebendigkeit und Deadlockfreiheit
  - Gültigkeit; Robustheit gegenüber Störungen wie fehlerhaften Nachrichten oder Ausfälle von Gruppenteilnehmern
- Beim Konsens kann ein Master-Slave Konzept oder ein Gruppenkonzept mit Leader/Commander und Workern verwendet werden.
  - Beim Master-Slave Konzept kommunizieren nur Slaves mit dem Master
  - Bei Gruppenkonzept (i.A. mit einem Leader) kommunizieren auch alle Gruppenteilnehmer untereinander
- Durch Störung (Fehler oder Absicht) kann es zu fehlerhaften bis hin zu fehlgeschlagenen Konsens kommen.

Verteilter Konsens

- Bedingungen für Interaktive Konsistenz:
  - IC1: Jeder Worker empfängt die *gleiche* Anweisung vom Leader!
  - IC2: Wenn der Leader *fehlerfrei* arbeitet, dann empfängt jeder *fehlerfreie* Worker die Anweisung die der Leader sendete!

Verteilter Konsens

Byzantinisches Generalproblem

- Beispiel: In einer Gruppe aus drei Prozessen/Agenten ist einer fehlerhaft bzw. versendet fehlerhafte Nachrichten (durch Störung oder Absicht) mit Anweisungen (schließlich ein Konsensergebnis) [E]

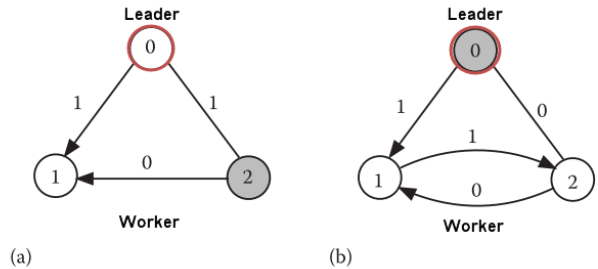


Abb. 6. Byzantinisches Generalproblem: (a) Leader 0 ist fehlerfrei, Worker 2 ist fehlerhaft (b) Leader 0 ist fehlerhaft, Worker 1 und 2 sind fehlerfrei [E]

Verteilter Konsens

- Jeder Worker der Nachrichten empfängt ordnet diese nach direkten und indirekten (von Nachbarn)
- **Fall (a)**: Prozess 2 versendet fehlerhafte Nachricht mit Anweisung 0, Prozess 1 empfängt eine direkte Nachricht mit Anweisung 1 und eine indirekte mit (falschen) Inhalt Anweisung 0
  - Bedingung IC1 ist erfüllt. Unter Annahme von Bedingung IC2 wird Worker 1 die direkte Anweisung 1 von Prozess 0 (Commander) auswählen → Konsens wurde gefunden
- **Fall (b)**: Prozess 0 (Leader) versendet an Prozess 1 richtige Nachricht mit Anweisung 1 und falsche Nachricht mit Anweisung 0 an Prozess 1
  - Würde Prozess 1 wieder zur Erfüllung von IC2 eine Entscheidung treffen (Anweisung 1 auswählen), dann wäre IC1 verletzt. Wie auch immer Prozess 1 entscheidet ist entweder IC1 oder IC2 verletzt → **Unentscheidbarkeit** → Kein Konsens möglich

Verteilter Konsens

Das nicht-signierte Nachrichtenmodell erfüllt die Bedingungen:

1. Nachrichten werden während der Übertragung nicht verändert (aber keine harte Bedingung).
2. Nachrichten können verloren gehen, aber die Abwesenheit von Nachrichten kann erkannt werden.
3. Wenn eine Nachricht empfangen wird (oder ihre Abwesenheit erkannt wird), kennt der Empfänger die Identität des Absenders (oder des vermeintlichen Absenders bei Verlust).

- Algorithmen zur Lösung des Konsensproblems müssen *m* fehlerhafte Prozesse annehmen (bzw. fehlerhafte Nachrichten)

Verteilter Konsens

Der OM(m) Algorithmus

- Ein Algorithmus der einen Konsens erreicht bei Erfüllung der Bedingungen IC1 und IC2 mit bis zu m fehlerhaften Prozesse bei insgesamt $n \geq 3m+1$ Prozessen mit nicht signierten ("mündlichen") Nachrichten.
  i. Leader i sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq$ i. ii. Jeder Worker j akzeptiert den Wert von i als Befehl vom Leader i.

Verteilter Konsens

1. Leader *i* sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq$ i.
2. Wenn $m > 0$, dann beginnt jeder Worker *j*, der einen Wert vom Leader erhält, eine neue Phase, indem er ihn mit OM(m-1) an die verbleibenden Worker sendet.
   - In dieser Phase fungiert *j* als Leader.
   - Jeder Arbeiter erhält somit (n-1) Werte: (a) einen Wert, der direkt von dem Leader *i* von OM (m) empfangen wird und (b) (n-2) Werte, die indirekt von den (n-2) Workern erhalten werden, die aus ihrem Broadcast OM(m-1) resultieren.
   - Wird ein Wert nicht empfangen wird er durch einen Standardwert ersetzt.
3. Jeder Worker wählt die Mehrheit der (n-1) Werte, die er erhält, als Anweisung vom Leader *i*.

Def. 2. Algorithmus OM(m)

Verteilter Konsens

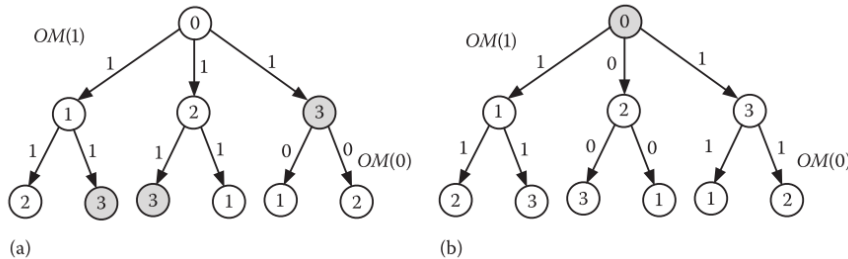


Abb. 7. Eine Illustration von OM(1) mit vier Prozessen und einem fehlerhaften Prozess: die Nachrichten auf der oberen Ebene spiegeln die Eröffnungsnachrichten von OM(1) wider und die auf der unteren Ebene spiegeln die OM(0)-Meldungen wider, die von den Mitteilungen der oberen Ebene ausgelöst werden. (a) Prozess 3 ist fehlerhaft. (b) Prozess 0 (Leader) ist fehlerhaft. [E]

Verteilter Konsens

Der Paxos Algorithmus

- Paxos ist ein Algorithmus zur Implementierung von fehlertoleranten Konsensfindungen.
- Er läuft auf einem *vollständig verbundenen Netzwerk* von n Prozessen und toleriert bis zu m Ausfälle, wobei $n \geq 2m+1$ ist.
- Prozesse können abstürzen und Nachrichten können verloren gehen, byzantinische Ausfälle (absichtliche Verfälschung) sind jedoch zumindest in der aktuellen Version ausgeschlossen.
- Der Algorithmus löst das Konsensproblem bei Vorhandensein dieser Fehler auf einem *asynchronen System von Prozessen*.
  - Obwohl die Konsensbedingungen Zustimmung, Gültigkeit und Terminierung sind, garantiert Paxos in erster Linie die Übereinstimmung und Gültigkeit und nicht die Beendigung - es ermöglicht die Möglichkeit der Beendigung nur dann, wenn es ein ausreichend langes Intervall gibt, in dem kein Prozess das Protokoll neu startet.

Verteilter Konsens

- Ein Prozess kann drei verschiedene Rollen wahrnehmen:
  - Antragsteller,
  - Akzeptor und
  - Lerner.

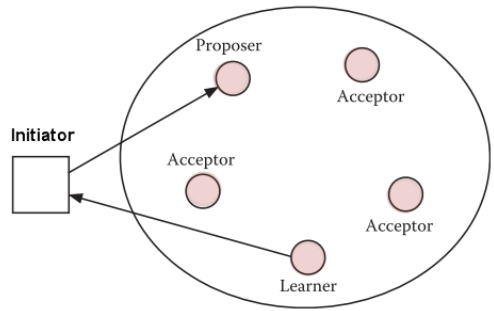


Abb. 8. Typische Rollenverteilung beim Paxos Algorithmus

Verteilter Konsens

- Die **Antragsteller** reichen die vorgeschlagenen Werte im Namen eines Initiators ein,
- die **Akzeptoren** entscheiden über die Kandidatenwerte für die endgültige Entscheidung und
- die **Lernenden** sammeln diese Informationen von den Akzeptoren und melden die endgültige Entscheidung dem Initiator zurück.

- Ein Vorschlag, der von einem Antragsteller gesendet wird, ist ein Tupel (v, n), wobei v ein Wert und n eine Sequenznummer ist.

- Wenn es nur einen Akzeptor gibt, der entscheidet, welcher Wert als Konsenswert gewählt wird, dann wäre diese Situation zu einfach. Was passiert, wenn der Akzeptor abstürzt? Um damit umzugehen, gibt es mehrere Akzeptoren.

- Ein Vorschlag muss von mindestens einem Akzeptor bestätigt werden, bevor er für die endgültige Entscheidung in Frage kommt.

Verteilter Konsens

- Die Sequenznummer wird verwendet, um zwischen aufeinander folgenden Versuchen der Protokollanwendung zu unterscheiden.
- Nach Empfang eines Vorschlags mit einer größeren Sequenznummer von einem gegebenen Prozess, verwerfen Akzeptoren die Vorschläge mit niedrigeren Sequenznummern.
- Schließlich akzeptiert ein Akzeptor die Entscheidung der Mehrheit.

Phasen des Paxos Algorithmus

1. **Die Vorbereitungsphase**
   - Jeder Antragsteller sendet einen Vorschlag (v, n) an jeden Akzeptor
   - Wenn n die größte Sequenznummer eines von einem Akzeptor empfangenen Vorschlags ist, dann sendet er ein *ack* (n, $\perp$, $\perp$) an seinen Vorschlager
   - Hat der Akzeptor einen Vorschlag mit einer Sequenznummer $n' < n$ und einem vorgeschlagenen Wert v akzeptiert, antwortet er mit *ack* (n, v, n').

Verteilter Konsens

2. **Aufforderung zur Annahme eines Eingabewertes**

   - Wenn ein Antragsteller *ack* (*n*, ⊥, ⊥) von einer Mehrheit von Akzeptoren empfängt, sendet er an alle Akzeptoren *accept* (*v*, *n*) und fordert sie auf, diesen Wert zu akzeptieren.
   - Wenn ein Akzeptor in Phase 1 einen *ack* (*n*, *v*, *n*') an den Antragsteller zurücksendet, muss der Antragsteller den Wert *v* mit der höchsten Sequenznummer in seiner Anfrage an die Akzeptoren einbeziehen.
   - Ein Akzeptor akzeptiert einen Vorschlag (*v*, *n*), sofern er nicht bereits zugesagt hat, Vorschläge mit einer Sequenznummer größer als *n* zu berücksichtigen.

3. **Die endgültige Entscheidung**

   - Wenn eine Mehrheit der Akzeptoren einen vorgeschlagenen Wert akzeptiert, wird dies der endgültige Entscheidungswert. Die Akzeptoren senden den akzeptierten Wert an die Lernenden, wodurch sie feststellen können, ob ein Vorschlag von einer Mehrheit von Akzeptoren akzeptiert wurde.

Zusammenfassung

Zuverlässige und sichere Synchronisation in asynchronen Systemen mit Fehlern ist theoretisch nicht möglich!