Vortrag im Sommersemester 2000
Seminar „Verteilte Algorithmen“

„IMPROVING THE FAULT TOLERANCE OF ALGORITHMS“

von Kevin Köser

basierend auf Kapitel 12 von
Distributed Computing, Attiya/Welch, McGraw Hill, 1998

IMPROVING THE FAULT TOLERANCE OF ALGORITHMS

1. Einleitung

Zielstellung

2. Wiederholung

Voraussetzungen

3. Verbesserung der Fehlertoleranz (synchroner Fall)

a) Protokoll-Schichtung:
identical byzantine/ omission / crash

b) „identical byzantine“ - Schicht

c) „omission failure“ - Schicht

d) „crash failure“ - Schicht

e) Korrektheitsbeweis am Beispiel des Algorithmus b)

4. beispielhafte Anwendung im synchronen Fall

„consensus“ auf einem System byzantinischer Fehler

5. Ansätze für den asynchronen Fall

2. Wiederholung

Voraussetzungen:

„synchronous message passing systems“:

- Netzwerk aus n Prozessoren p_i
- rundenbasiert, in jeder Runde:
  - jeder kann Nachricht an jeden anderen senden
  - Empfang der Nachrichten
  - Berechnungen

=> „admissible execution“ eines Algorithmus:

- fairness (=> synchronous)
- user-compliance (korrekte Umgebung)
- correctness (korrekte „Basis“)

„crash failure“

Prozessor hält an, verschickt ab sofort keine weiteren Nachrichten mehr

„byzantine failure“

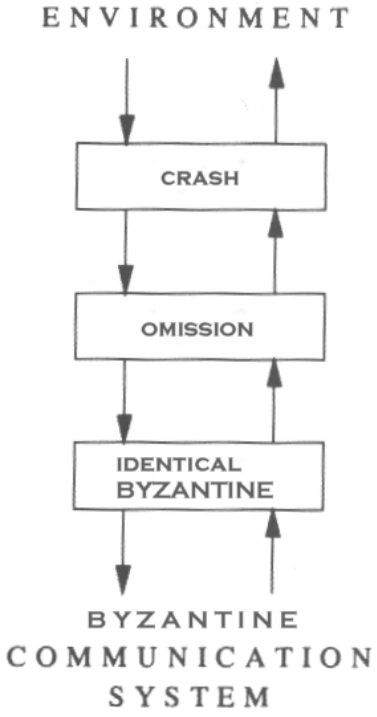
Prozessor verschickt Nachrichten beliebigen Inhalts (bzw. verschiedene/keine Nachrichten)

3-stufiges Aufheben der Unterschiede zwischen „crash“ und „byzantine“

1.) identical byzantine:
alle p_i bekommen die gleichen Nachrichten

2.) omission:
falsche Nachrichten werden nicht empfangen

3.) crash:
falls Fehler auftritt, wird der Sender danach ignoriert



prinzipielle Funktionsweise Algorithmus „identical byzantine“

a) Verfahren beim Eingang einer Nachricht auf byzantine-Ebene (byz-receive):

1.) Kopie („Echo“) der Nachricht an alle schicken (byz-send)

2.) bei genug „Echos“ Nachricht akzeptieren (idbyz-receive)

b) Verfahren, wenn keine Nachricht empfangen wird

falls genug „Echos“ für eine Nachricht da sind Nachricht akzeptieren (idbyz-receive)

=> doppelte Rundenzahl nötig durch Bestätigungsverfahren

Bemerkungen zur „identical byzantine“-Simulation

- Rundenzahl wird verdoppelt

- Für jedes Nachrichtenbit auf byzantine-Ebene werden $O(n^2)$ Nachrichtenbits auf „identical byzantine“-Ebene verschickt

- Voraussetzung $n > 4f$

prinzipielle Funktionsweise Algorithmus „omission“

Nachrechnen (via „idbyz-receive“) empfangener Nachrichten, bei Korrektheit Empfang (via „om-receive“)

benötigtes „Wissen“ bei jedem Prozessor:

- benutzter Algorithmus auf höherer Ebene
- Eingangsdaten („support set“) des Senders

Algorithmus „omission“

Initially $valid = \emptyset$, $accepted = \emptyset$, and $pending = \emptyset$

In response to om-send$_i(m)$:
1: id-send$_i(\langle m, accepted \rangle)$
2: id-recv$_i(R)$
3: add R to *pending*
4: validate(*pending*)
5: $accepted := \{m'$ with sender $p_j : (m', j, k) \in valid\}$
6: om-recv$_i$(*accepted*)

procedure validate(*pending*):
1: for each $\langle m', support, k' \rangle \in pending$ with sender p_j,
in increasing order of k', do
2: if $k' = 1$ then
3: if m' is an initial state of the A process on p_j then
4: add $(m', j, 1)$ to *valid* and remove it from *pending*
5: else // $k' > 1$
6: if $(m'', h, k' - 1) \in valid$ for each $m'' \in support$ with sender p_h
and $(v, j, k' - 1) \in valid$ for some v
and $m' = \text{transition}_A(j, v, support)$
7: then add (m', j, k') to *valid* and remove it from *pending*

Bemerkungen zur „omission“-Simulation

- keine zusätzlichen Runden benötigt

- beträchtliche Erhöhung der Nachrichtengröße, da auch „support sets“ übergeben werden

- Anwendungs-Algorithmen ALLER höherliegenden Schichten müssen bekannt sein !

- Prinzip „jeder rechnet alles nach“ nur für sehr sicherheitsbedürftige Aufgaben, nicht zur Problemteilung anwendbar

prinzipielle Funktionsweise Algorithmus „crash“

- für jede Nachricht ein Echo senden

- nach dem eigenen Senden Echos zählen

- bei zu wenig Echos selbst „crashen“

Algorithmus „crash“

round $(k, 1)$: in response to crash-send$_i(m)$:

1: om-send$_i(\langle init,m \rangle)$
2: om-recv$_i(R)$
3: $S := \{ \langle echo,m',j \rangle : \langle init,m' \rangle$ with sender p_j is in $R\}$

round $(k, 2)$:

4: om-send$_i(S)$
5: om-recv$_i(R)$
6: if $< n - f$ messages in R contain $\langle echo,m,i \rangle$ then crash self
7: crash-recv$_i$ ($\{m'$ with sender $p_j : \langle echo,m',j \rangle$ is contained in a message in $R\}$)

Bemerkungen zum „crash“-Algorithmus

- Rundenzahl verdoppelt sich wieder

- durch Echo-Nachrichten werden erneut ($O(n^2)$) soviele Nachrichten-Bits verschickt wie zuvor

- Voraussetzung $n > 2f$

Zusammenfassung der drei Schichten

Voraussetzungen der Gesamtsimulation:

- $n > 4f = \max(2f,4f)$
- Algorithmus auf höherliegender Schicht bekannt !

Kosten der Simulation:

- Vervierfachung der Rundenzahl (2x1x2)
- ca. $O(n^5)$ Nachrichtenbits pro Bit auf unterer Schicht

=> Eignung nur falls einfache Algorithmen für obere Schicht existieren, nicht zur „Arbeitsteilung“

4. Anwendung

„consensus“ bei „byzantine failures“

Algorithm 5.1 Consensus algorithm in the presence of crash failures: code for processor p_i, $0 \leq i \leq n-1$.

Initially $V = \{x\}$ // V contains p_i's input

round k, $1 \leq k \leq f+1$:

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all processors
2: receive S_j from p_j, $0 \leq j \leq n-1$, $j \neq i$
3: $V := V \cup \bigcup_{j=0}^{n-1} S_j$
4: if $k = f+1$ then $y := \min(V)$ // decide

z.B. Algorithmus 5.1 statt „phase king“-Algorithmus

Effizienzvergleich:

	phase king	5.1 auf „crash-simulation“
Runden	2(f+1)	4(f+1)
Toleranz	n > 4f	n > 4f

aber: in 5.1 „omission“ bereits integriert,
=> „crash“-Schicht kann weggelassen werden
=> nur (2f+1) Runden

5.) Ansätze für den asynchronen Fall

„asynchronous message passing system“

Unterschiede zum synchronen Fall

- keine obere Schranke für Auslieferungsdauer
- aber: jede gesendete Nachricht kommt schließlich an
- jeder Prozessor macht unendlich viele Schritte

Verbesserung der Fehlertoleranz:

- „identical byzantine“ im asynchronen Fall analog
- „omission“ nur implementierbar für deterministische Algorithmen, da Umgebung mit einfließt

=> größtes Problem (inhaltliche Fehler) im asynchronen Fall praktisch nicht durch Protokoll behebbar, da kaum asynchrone deterministische Algorithmen verfügbar