6 Transaktionen

6.1 Allgemeines

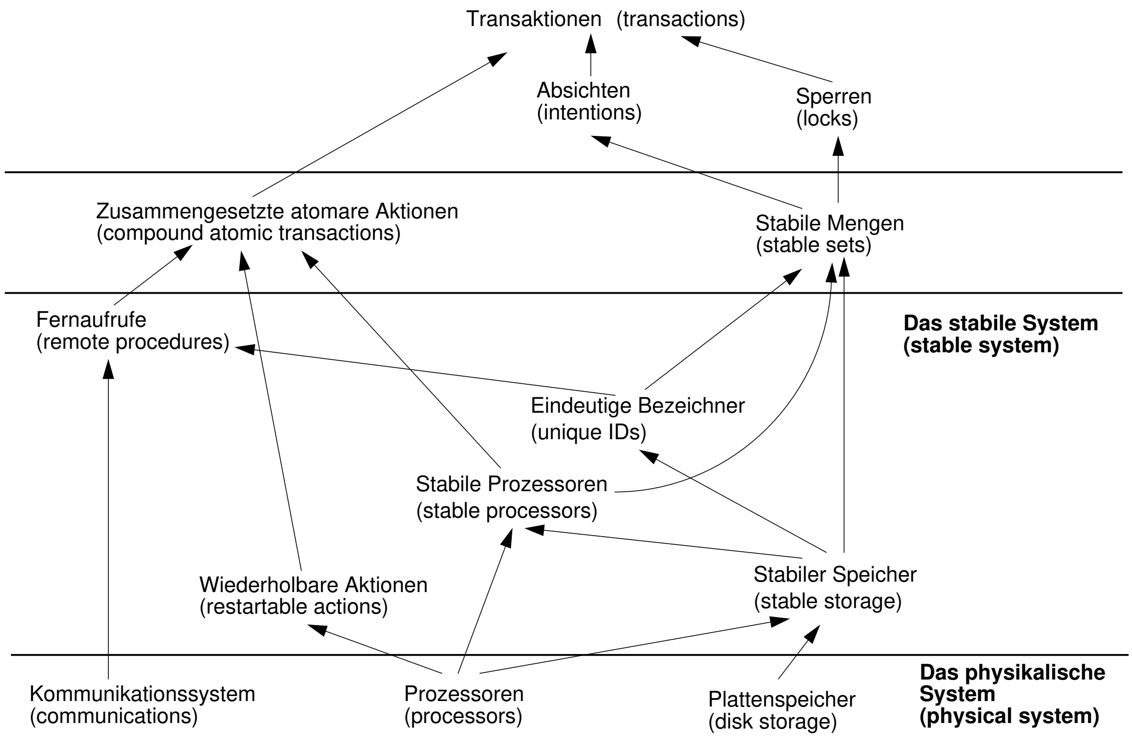
- Problembeispiel: Überweisung von $5 von A an B
  - read account A (obtaining $10);
  - read account B (obtaining $15);
  - write $5 to account A;
  - write $20 to account B;
- Vorstellung: Kunde A hat sein Konto bei der Bank A
  Kunde B hat sein Konto bei der Bank B
  Er tätigt die Anweisung bei der Bank C
  Jede Bank hat ihren eigenen Rechner usw.
- Problem: Systemzusammenbruch während oder zwischen den Schreibanweisungen
  Festplatte mit den Daten defekt
  Nachrichtenverlust oder Nachrichtenverdoppelung
- Ziel: Alle Schreibanweisungen oder keine werden durchgeführt und das Ergebnis ist bekannt

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- Transaktionen bei Betriebssystemen:
  - auf die Dauerhaftigkeit kann oft verzichtet werden
  - kein Logbuch notwendig
- Betrachtung am Modell von Lampson
  - die Probleme werden in dem Artikel detailliert dargestellt
  - prinzipielle Lösungsmöglichkeiten werden vorgestellt
  - effektive Implementierung
  - keine effiziente Implementierung
  - einige der Annahmen sind zu hinterfragen

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

6.2 System-Überblick



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

6.3 Konsistenz und Transaktionen

- **Konsistenz**

  System erfüllt in jedem "beobachtbaren" Zustand ein bestimmtes Prädikat, die sogenannte Invariante.

- **Systemverhalten definiert durch Übergangsfunktion F**

  state := F(state)

- **Ausgabemenge von F**

  Modifizierbare Zustandsanteile.

- **Eingabemenge von F**

  Zustandsanteile, die das Ergebnis beeinflussen.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

- **Transaktionen (atomare Aktionen)**

  Eine Folge von Lese-/Schreibaufrufen mit der Eigenschaft, daß entweder alle oder keiner der Schreibaufrufe sichtbar sind.
  Sie werden durch Klammerung mit den (System-)Funktionsaufrufen Begin und End gekennzeichnet.

- **Datenbefehle:**

  Read, Write

- **Kontrollbefehle:**

  Begin, End, Abort

6.4 Das physikalische System

6.4.1 Der Plattenspeicher

- **Operationen:**

  procedure put(at: address; data: datablock)

  procedure get(at:address)
  returns (status: (good, looksBad); data: datablock)

◆ **get**

- **erwünscht:**

  Seite ist (good, d) und get liefert (good, d)

  Seite ist (bad, ...) und get liefert (looksBad, ...)

- **Fehler:**

  Seite ist (good, d) und get liefert (looksBad, ...)

  (Wiederholung, wie oft?)

- **Ruin:**

  Seite ist (good, d) und get liefert n_R-mal (looksBad, ...)

  (Ist hier eingeordnet wegen der noch zu erläuternden Auswirkungen auf die Überlegungen zur Verlustverwandtschaft)

  Seite ist (bad, ...) und get liefert (good, ...)

  Seite ist (good, d) und get liefert (good, d') mit d≠d'

◆ **put**

- **erwünscht:**

  auf Plattenspeicher (good, d)

- **Fehler:**

  auf Plattenspeicher unverändert

  auf Plattenspeicher (bad, ...)

- **spontan**

  Verlust:

  Verlustmenge: **Menge aller Seiten, die durch eine Verlustursache betroffen werden können,**

  z. B. **Spur durch Ausfall eines Schreib-/Lesekopfes**

  Zylinder durch Justierungsfehler

  Oberfläche durch Kopfabsturz

  Einheit durch Fehler in der Steuerelektronik

- **Zwei Seiten heißen genau dann verlustverwandt, wenn es eine Verlustmenge gibt, der beide angehören.**

  - **Fehler:** (good, d) geht selten nach (bad, d), d. h. höchstens ein Verlust in T_D (bad, d) geht nach (good, d)

    (bad, d) geht über in (good, d)

  - **Ruin:** (good, d) geht häufig nach (bad, d), d. h. mehrere Verluste in T_D

    (good, d) geht nach (good, d') mit $d \neq d'$

6.4.2 Prozessoren

- **Feste Zahl von Prozessen**

- **Ein Prozeß liefert Zeit für Messung von Verlustintervallen T_D (Timeout)**

- **Fehler:**

  Zusammenbruch

  Wiederaufsetzen mit irgendeinem Standardzustand, d. h. Speicher flüchtig

- **Annahme:**

  Alle anderen Fehlfunktionen werden entdeckt und führen zu einem Zusammenbruch, bevor Plattenspeicher oder das Kommunikationssystem betroffen werden.

  Zusammenbrüche werden entdeckt.

6.4.3 Kommunikationssystem

- Operationen:

  send(to: processor; data: messageblock)

  receive() returns (status: (good, bad), data: messageblock)

- receive
  - erwünscht:

    es existiert (good, d, p) und
    es wird (good, d) abgeliefert

    es existiert (bad, d, p) und
    es wird (bad, ...) abgeliefert.

- send
  - erwünscht:

    nach Ausführung existiert (good, d, p)

- spontan
  - Fehler:

    Nachricht geht verloren

    Nachricht wird vervielfacht

    (good, d, p) geht nach (bad, ..., ...)

  - Ruin:

    (bad, d, p) geht nach (good, ..., ...)

    (good, d, p) geht nach (good, d’, p’) mit $d \neq d'$ oder $p \neq p'$.

6.4.4 Einfache, zusammengesetzte und wiederholbare Aktionen

- Einfache Aktionen
  - atomar:
    - Ganzheit des Effekts (alles oder nichts beobachtbar)

      kein Zusammenbruch: $\{P\}\ S\ \{Q\}$

      Zusammenbruch: $\{P\}\ S\ \{P \vee Q\}$
    - Serialisierbarkeit
- Zusammengesetzte Aktionen
  - Nicht alle Aktionen aller Abstraktionsebenen können atomar gemacht werden.
- Allgemeine Eigenschaften:
  - kein Zusammenbruch: $\{P\}\ S\ \{Q_{ok}\}$
  - Zusammenbruch: $\{P\}\ S\ \{Q_{crash}\}$

    Die Gültigkeit von Q_{crash} wird beim Wiederanlauf (recovery) hergestellt.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- Im Vergleich zur Atomarität abgeschwächte Forderungen:

  a) $Q_{ok} \Rightarrow Q_{crash}$

  Zusammenbruch unmittelbar vor return möglich

  b) $Q_{crash} \Leftrightarrow P \vee Q_{ok}$

  Gleichwertig zu Ganzheit

  c) $Q_{crash} \Rightarrow P$

  Wiederholbarkeit (Idempotenz) wird notwendig, wenn Atomarität nicht gewährleistet werden kann

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

6.5 Das stabile System

6.5.1 Stabiler Speicher

- Höhere Abstraktionen des Plattenspeichers:
  - realer Plattenspeicher → sorgfältiger Plattenspeicher
    → stabiler Plattenspeicher
- Sorgfältiger Plattenspeicher:
  - sorgfältiges Lesen:
    Wiederholung bis Zustand good oder n_R-mal
  - sorgfältiges Schreiben:
    Evtl. wiederholtes Schreiben und jeweils anschließendes einfaches Lesen, bis Leseergebnis good

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- Stabiler Plattenspeicher:
  - stabile Seite:
    Seite und nicht verlustverwandte Schattenseite
  - stabiles Lesen:
    Seite sorgfältig lesen;
    falls Zustand bad Schattenseite sorgfältig lesen
  - stabiles Schreiben:
    Seite sorgfältig schreiben,
    anschließend Schattenseite sorgfältig schreiben

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

- Bereinigung:
  - Seite und Schattenseite sorgfältig lesen;
    Beide im Zustand good, dann fertig;
    falls eine davon im Zustand bad,
    dann andere sorgfältig lesen und die "bad"-Seite damit sorgfältig beschreiben;
    falls beide im Zustand good, aber mit verschiedenen Daten,
    dann eine der beiden auswählen und in die andere sorgfältig schreiben.
- Bereinigung bei Systemstart, nach Zusammenbruch und mindestens alle T_D Zeiteinheiten.
- Es muß eine Zuordnungsfunktion vorgesehen werden,
  - mit deren Hilfe alle stabilen Seiten aufgezählt werden können und
  - die zu jeder stabilen Seite die sie darstellende Seite und Schattenseite zu ermitteln gestattet.

6.5.2 Stabile Prozessoren

- Plattenspeicher als verlängerter Arbeitsspeicher ⇒ flüchtig
- Stabiler Speicher zur Sicherung von Prozeßzuständen;
  - save: Speichert PCB, Aufrufparameter, ...
  - reset
  - restart: Durch diese Operation kann nach Zusammenbruch ein Prozeß in seinem zuletzt gesicherten Zustand wieder aufgesetzt werden.
- Stabile Monitore
  - Lokale Daten, ausgenommen die Sperre zum gegenseitigen Ausschluß, werden im stabilen Speicher geführt.
  - Modifikation der lokalen Variablen durch eine Monitorprozedur erfolgt mit einem einzigen stabilen Schreibaufruf.
  - Prozeßzustände werden in Monitoren nicht gesichert.
  - Nur Monitore können Daten enthalten, die nicht prozeßlokal sind.
  - Höchstens ein stabiles Put (also atomar!)

6.5.3 Fernaufrufe

- Kommunikation zwischen Maschinen erfolgt nur durch Fernaufrufe.
- At-least-once-Semantik reicht wegen der Wiederholbarkeit

6.6 Zusammengesetzte Aktionen

- Bisher bereitgestellte Basis:
  - Stabiler Speicher
  - Stabile Prozesse
  - Stabile Monitore
  - Keine flüchtigen Daten

6.6.1 Stabile Mengen

- Eine stabile Menge besteht aus
  - Verbunden, die kleiner sind als eine stabile Seite,
  - einem eindeutigen Bezeichner.

- Alle Operationen zur Manipulation stabiler Mengen sind wiederholbar.

- Alle Operationen liefern eine Fehlermeldung, wenn sie nicht zwischen Create und dem nächsten Erase ausgeführt werden (Create und Erase ausgenommen).

Transaktionen: Zusammengesetzte Aktionen

- Atomare Operationen:
  - Create(ID i)

    Erzeugt eine neue stabile Menge mit Bezeichner i. Falls sie schon existiert, ist die Operation wirkungslos.

  - Insert(StableSet s, StableSet t, Record new)

    Setzt voraus, daß new noch nicht in s oder t enthalten ist, und nimmt es in s und t auf. Einer der beiden Parameter s oder t kann auch nil sein.

  - Replace(StableSet s, StableSet t, Record old, Record new)

    Setzt voraus, daß old in s und t enthalten ist. Old wird entfernt und new aufgenommen.

  - IsEmpty(StableSet s)

    Liefert true bzw. false, je nachdem ob s leer ist oder nicht.

  - IsMember(StableSet s, Record r)

    Liefert true, wenn r in s enthalten ist, sonst false.

Transaktionen: Zusammengesetzte Aktionen

- Nicht atomare Operationen:
  - Enumerate(StableSet s, procedure p)

    Ruft p für jedes Element in s auf.

    In den Programmen geschrieben als: for r in s do p;

  - Erase(StableSet s)

    Entfernt alle Elemente aus s. Wurde ein Element bei seiner Einfügung mittels Insert oder Replace gleichzeitig in eine weitere stabile Menge eingefügt, so wird es auch dort entfernt.

- Implementierungsmöglichkeiten
  - Pool

    lokal auf einem Prozessor

  - Wide Stable Set

    Root auf einem Processor kennt die Verteilung der Leafs auf andere Prozessoren

6.6.2 Zusammengesetzte atomare Aktionen

Gestalt einer Methode:

```
A(...) { R; }
```

Implementierung (nur für wiederholbare Prozeduren):

```
A(...)
{    Save /* Process State */;
     R;
          Reset  /* Process State */
}
```


6.7 Transaktionen

- **Eindeutiger Transaktionsbezeichner:**

```
typedef UniqueId TI
```

- **Seitenadresse:**

```
class SA {   FileName Dateiname;
             int Seitennummer;
         };
```

- **Aktion:**

```
enum RW {read, write};
```

- **Absicht:**

```
class Intention  {  TI      t;
                    SA      p;
                    RW      a;
                    char    d[...];   // zu modif. Daten;
             };
```

Transaktionen: Transaktionen

- **Phase:**

  enum Phase
  {nonexistent, running, committed, aborted};

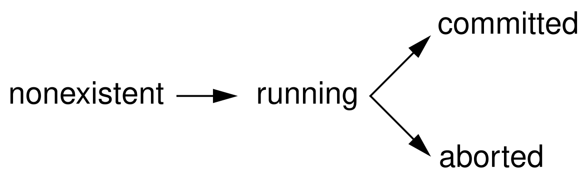
- **Transaktionsmarke:**

  ```
  class TM {    TI        t;
                Phase     ph;
           };
  ```


Transaktionen: Transaktionen

```
entry Phase GetPhase(TI t)
   {
      for r in s.flags do {
         if (r.t == t)
              return r.ph;
      }
      return nonexistent;
   }
```

```
entry SetPhase(TI t, Phase desiredPhase)
    /* desiredPhase may not be nonexistent} */
  { switch (GetPhase(t)) {
      case committed:
      case aborted:   break;              /* do nothing */
      case running:   overwrite(s.flags, TM(t, desiredPhase));
                      break;
      case nonexistent: if (desiredPhase == running)
                          replace(s.flags, nil,
                                  TM(t, GetPhase(t)),
                                  TM(t, running));
                        break;
    }
  };
```



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

- **Für jede Dateiseite auf dem Server-Rechner:**
  - **Stabile Menge p.locks (mit Absichten als Elementen)**
  - **Stabiler Monitor zur Bearbeitung dieser Menge**
  - **Bedingungsvariable p.lockFreed, an der auf Sperren gewartet wird.**
- **Für jede Transaktion beim Koordinator:**
  - **Wurzel für verteilte stabile Menge**
  - **t.intentions (mit Absichten als Elementen; sie sind zu schreiben, wenn die Transaktion mit End beendet wird.)**
  - **Blatt bei jedem betroffenen Server**
- **Bei jedem Server:**
  - **Stabile Menge s.flags (mit Transaktionsmarken als Elementen)**
  - **Stabiler Monitor zur Bearbeitung dieser Menge.**

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

```
entry function Read(t: TI; p: SA): Data;
   var noConflict: Boolean;
   var i, i0: intention;
   begin
      repeat noConflict := true; i0.t = 0;
         for i in p.locks do
            if i.a=write
               then if i.t≠t
                     then begin Wait(p.lockFreed);
                                noConflict := false end;
                     else i0 = i;
      until noConflict;
      if i0.t = t
         then return(i0.d);
         else begin Insert(p.locks, t.intentions, <read, nil>);
                  return StableGet(p); end
   end;
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann


```
entry procedure Write(t: TI; p: SA; d: Data);
   var noConflict: Boolean;
   var d': Data;
   begin
      repeat  noConflict := true; d' := d;
         for i in p.locks do  /* enumerate(p.locks, q) */
         /* The next 5 lines correspond to procedure q */
            if i.t <> t
               then begin Wait(p.lockFreed);
                          noConflict := false end
               else if i.a = write then d' := i.d;
      until noConflict;
      if d'<> d
         then Replace(p.locks, t.intentions, <write,d'>, <write,d>)
         else
            Insert(p.locks, t.intentions, <write,d>
   end;
```


Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann
Reproduktion jeder Art oder Verwendung dieser Unterlage zu Lehrzwecken außerhalb der Universität Erlangen-Nürnberg ist ohne Genehmigung des Autors unzulässig

Transaktionen: Transaktionen

```
procedure Begin(): TI;
   const t = UniqueID();
   begin
      SetPhase(t, running);
      CreateWideStableSet(t);
      return t
   end;

function End(t: TI): Phase;
    return Complete(t, committed);

function Abort(t: TI): Phase;
    return Complete(t, aborted);
```


Transaktionen: Transaktionen

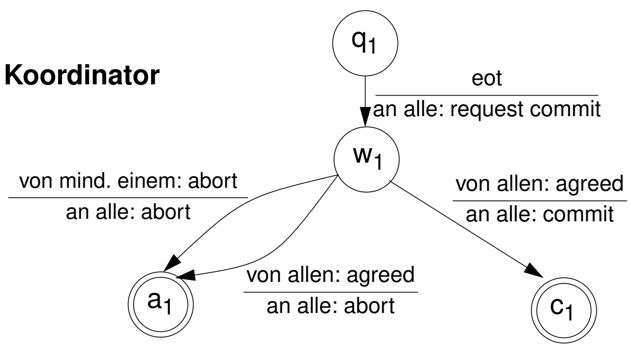
```
function Complete(t: TI; desiredResult: Phase
      /* committed or aborted only */): Phase;
   begin
      if GetPhase(t)=nonexistent then return nonexistent;
      Save /* process state */;
      SetPhase(t,desiredResult);
      /* now the transaction is committed or aborted */
      if GetPhase(t) = committed
         then for i in t.intentions do
                 if i.a=write then StablePut(i.p, i.d);
      Erase(t.intentions)
      /* also erases all corresponding entries in all p.locks and
         signals all p.lockFreed conditions */;
      Reset /* process state */;
      return GetPhase(t)
   end;
```

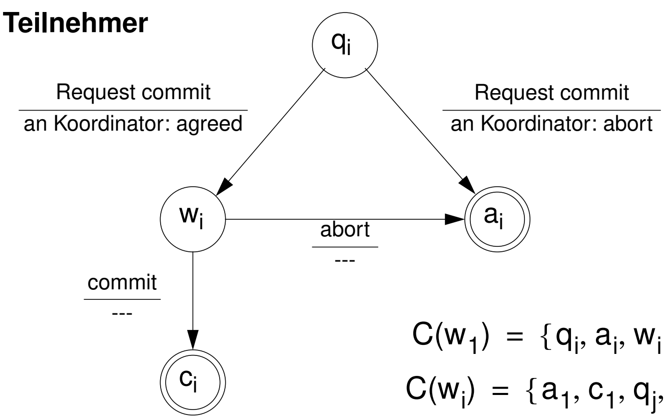
6.8 Commit-Protokolle

- **Systemzustand: Tupel der lokalen Zustände**
- **Atomare Aktionen:**
  - **Übergabe einer Nachricht an das Kommunikationssystem**
  - **Entnahme einer Nachricht aus dem Kommunikationssystem**
  - **Änderung des lokalen Zustandes**
- **Nur atomare Aktionen bewirken eine Änderung des Systemzustandes.**
- **Wenn die beteiligten Prozesse fehlerfrei sind, treten nur durch das Protokoll bedingte Übergänge zwischen Systemzuständen auf.**
- **Die Nebenläufigkeitsmenge C(s) eines lokalen Zustandes s besteht aus allen lokalen Zuständen, die zusammen mit s in einem erreichbaren Systemzustand vorkommen können.**
- **Die Sendermenge S(s) eines lokalen Zustandes s ist die Menge aller lokalen Zustände, die der Nebenläufigkeitsmenge von s angehören und von denen Nachrichten an s gerichtet sein können.**

BP 1 **Transaktionen: 2-Phasen-Commit**

Der Einfachheit wegen werden die folgenden Untersuchung auf den Fall zweier Prozessoren beschränkt. Die Ergebnisse sind aber verallgemeinerbar.



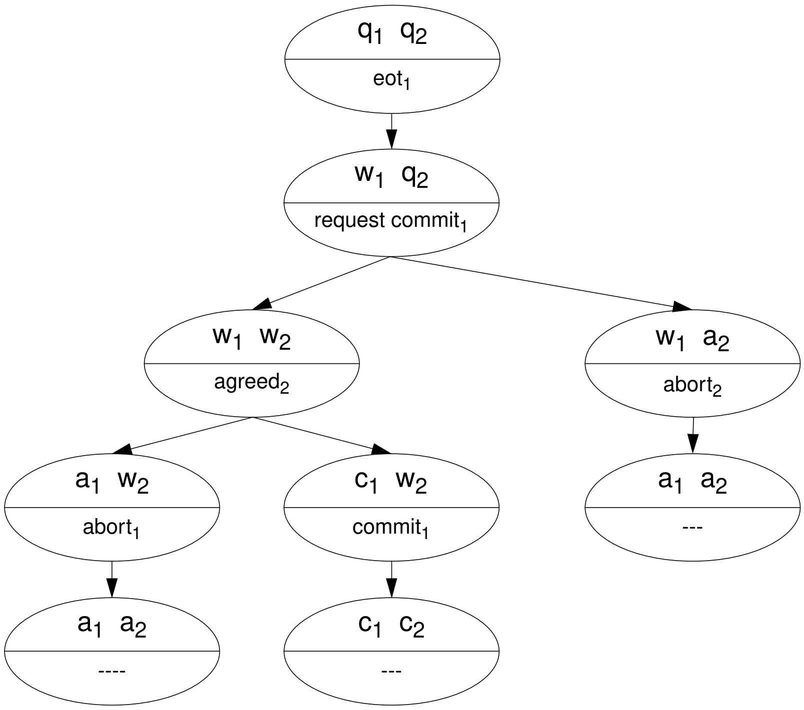


$$C(w_1) = \{q_i, a_i, w_i\}$$

$$C(w_i) = \{a_1, c_1, q_j, w_j, a_j, c_j\}$$

Transaktionen: 2-Phasen-Commit

Erreichbare Zustände



$$
\begin{aligned}
C(q_1) &= \{q_2\}\\
C(w_1) &= \{q_2, w_2, a_2\}\\
C(a_1) &= \{w_2, a_2\}\\
C(c_1) &= \{w_2, c_2\}\\
C(q_2) &= \{q_1, w_1\}\\
C(w_2) &= \{w_1, a_1, c_1\}\\
C(a_2) &= \{w_1, a_1\}\\
C(c_2) &= \{c_1\}
\end{aligned}
$$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Transaktionen: 2-Phasen-Commit

S6.1 **Satz**

Besitzt ein Protokoll einen Zustand s, dessen Nebenläufigkeitsmenge sowohl 'Abort'- als auch 'Commit'-Zustände enthält, so kann es bei unabhängigem Wiederanlauf keinerlei Ausfälle - auch nur eines Prozessors - tolerieren.

- **Fehler-Regel**

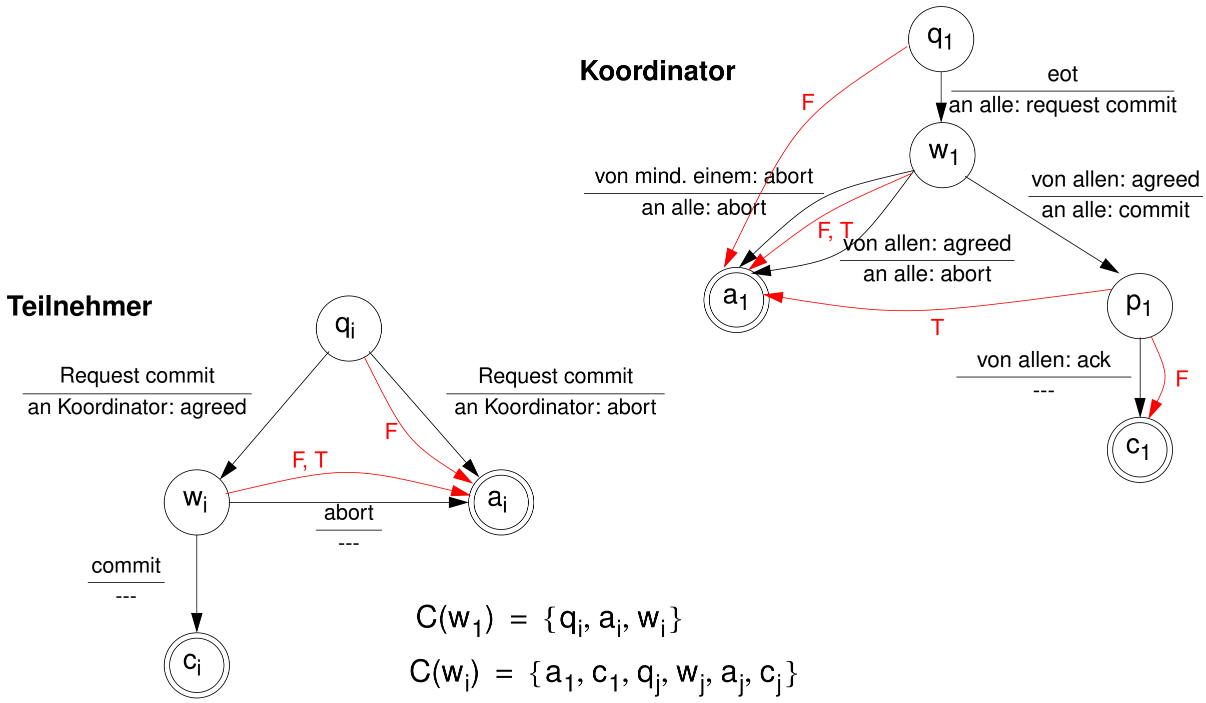
  Von jedem lokalen Zustand s aus, der kein Endzustand ist, wird eine F-Kante nach commit eingefügt, wenn C(s) einen Commit-Zustand enthält, sonst eine F-Kante nach abort.

- **Timeout-Regel**

  Von jedem lokalen Zustand s aus, der kein Endzustand ist, wird eine T-Kante nach commit (abort) eingefügt, wenn ein t $\in$ S(s) existiert, von dem eine F-Kante nach commit (abort) führt. Bei einer T-Kante nach abort wird an alle abort gesandt.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

BP 1 Transaktionen: 2-Phasen-Commit

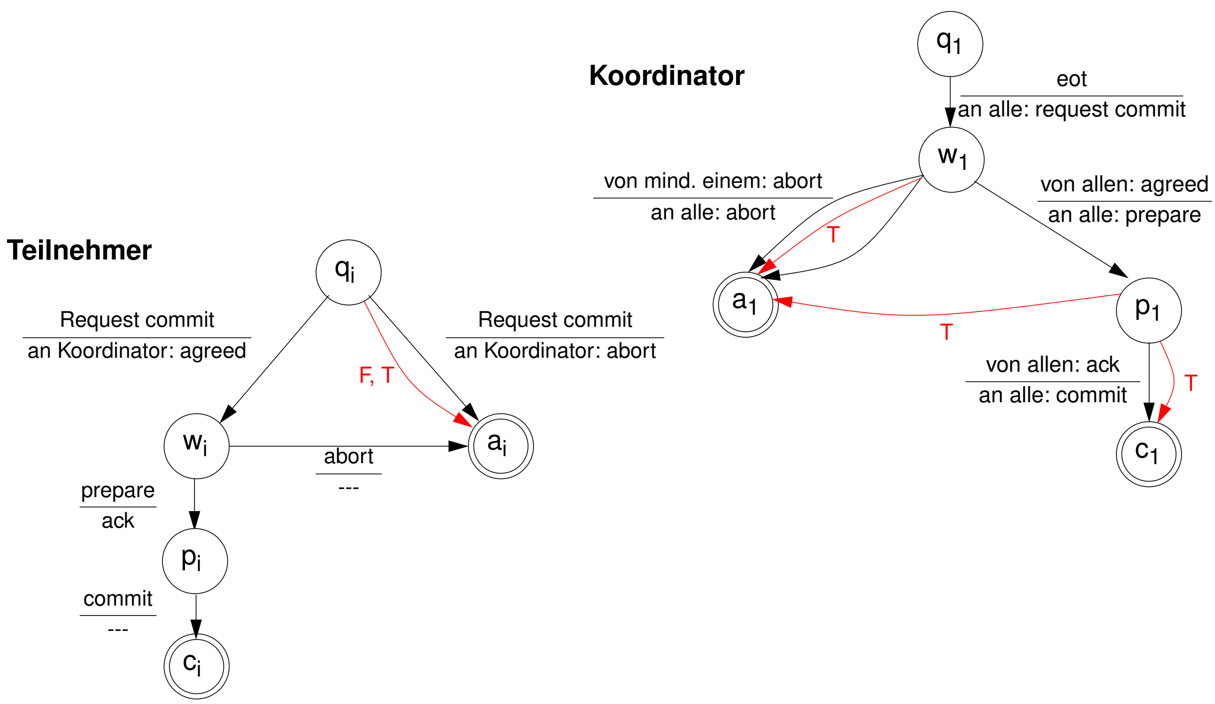


$$C(w_1) = \{q_i, a_i, w_i\}$$

$$C(w_i) = \{a_1, c_1, q_j, w_j, a_j, c_j\}$$

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

BP 1 Transaktionen: 2-Phasen-Commit



Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Transaktionen: 2-Phasen-Commit

6.2 **Satz**

Protokolle,

- **die keine lokalen Zustände besitzen, deren Nebenläufigkeitsmenge sowohl Abort- als auch Commit-Zustände enthält, und**
- **die der Fehler- und der Timeout-Regel genügen,**

können den Ausfall eines Beteiligten (falls das keinen Einfluß auf die Zustellbarkeit von Nachrichten hat) tolerieren.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Transaktionen: Nichtexistenz blockierungsfreier Commit-Protokolle

6.3 **Satz**

Es existiert kein Commit-Protokoll mit unabhängigem Wiederanlauf, das für den Fall von zwei fehlerhaften Teilnehmern funktioniert.

Beweis:

Es sei P **ein Commit-Protokoll; die Teilnehmer** i **und** j **seien fehlerhaft.**

Die Zustandsfolge $G_0, G_1, \ldots, G_m$ **sei eine fehlerfreie Abarbeitung des Protokolls, die zu einem globalen Commit-Zustand führt. Jeder globale Zustand** G_k **ist darstellbar als Vektor** $G_k = \langle s_{k,1}, s_{k,2}, \ldots, s_{k,N} \rangle$ **der lokalen Zustände.**

F_k **sei gleich** G_k**, lediglich** $s_{k,i}$ **und** $s_{k,j}$ **seien ersetzt durch** $f(s_{k,i})$ **und** $f(s_{k,j})$**.**

Nach Definition muß gelten: $f(s_{0,i}) = a_i$ **und** $f(s_{0,j}) = a_j$

sowie : $f(s_{m,i}) = c_i$ **und** $f(s_{m,j}) = c_j$**.**

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann

Also muß ein kleinstes k mit folgender Eigeinschaft existieren:

Sequenz G	**F, wenn** i **und** j **fehlerhaft**
$\langle \ldots, s_{0,i}, \ldots, s_{0,j}, \ldots \rangle$	$\langle \ldots, a_i, \ldots, a_j, \ldots \rangle$
$\langle \ldots, s_{k-1,i}, \ldots, s_{k-1,j}, \ldots \rangle$	$\langle \ldots, a_i, \ldots, a_j, \ldots \rangle$
$\langle \ldots, s_{k,i}, \ldots, s_{k,j}, \ldots \rangle$	**entw.** $f(s_{k,i}) = c_i$ **oder** $f(s_{k,j}) = c_j$
$\langle \ldots, s_{m,i}, \ldots, s_{m,j}, \ldots \rangle$	$\langle \ldots, c_i, \ldots, c_j, \ldots \rangle$

Zwei aufeinanderfolgende Zustände unterscheiden sich nämlich nach Definition in genau einer Komponente. Deshalb ist $f(s_{k-1,i}) = f(s_{k,i})$ **oder** $f(s_{k-1,j}) = f(s_{k,j})$**. Also muß einer ein Abort-Zustand sein.**

Nach Annahme muß dann der andere ein commit-Zustand sein, also ist der erreichte Endzustand inkonsistent.

Universität Erlangen-Nürnberg, IMMD IV, F. Hofmann