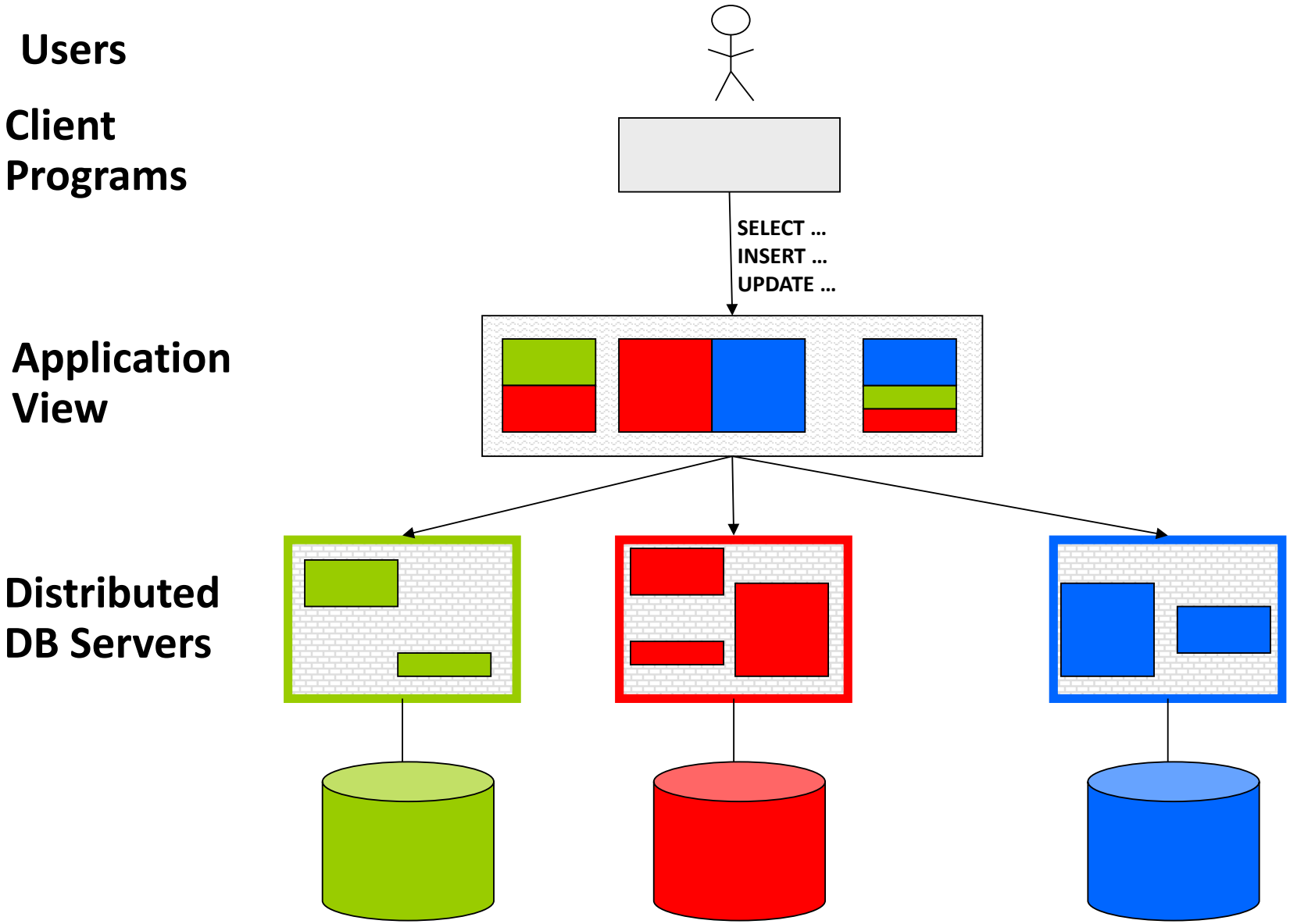
Modell einer verteilten Datenbank



Universität Trier

Bemerkung 6.1 (Modell einer verteilten Datenbank)

1. Die Anwendungssicht kann rein virtuell sein (es muss kein zentrales Interface geben).
2. Zur Vereinfachung nehmen wir an, dass die Daten nur fragmentiert wurden, aber nicht repliziert.

Arten verteilter DBMS

Wir betrachten zwei unterschiedliche Arten von verteilten DBMS:

- Homogen (DDBMS)
  - Alle teilnehmenden Rechner gehören zum selben logischen System.
  - Alle teilnehmenden Rechner verwenden die gleiche Software und die gleichen Verfahren.
  - Die Verteilung ist voll transparent für die Benutzer.
- Heterogen (Multi-DBMS, Föderation)
  - Die teilnehmenden Rechner sind autonom und unabhängig.
  - Die teilnehmenden Rechner verwenden verschiedene Protokolle, Datenmodelle und Anfragesprachen.

Arten von Transaktionen

Abhängig von der Art des DDBMS werden uns die folgenden Arten von Transaktionen begegnen:

Definition 6.2 (Globale Transaktion)

Eine *globale Transaktion* greift auf Daten über das verteilte System zu, dabei können die Daten auf einem oder auf mehreren Rechnern liegen; sie kommt sowohl in homogenen als auch in heterogenen Systemen vor.

Definition 6.3 (Lokale Transaktion)

Eine *lokale Transaktion* arbeitet nur auf einem einzelnen Rechner und ist dem verteilten System nicht bekannt; sie kommt nur in heterogenen Systemen vor.

Wir befassen uns im Moment nur mit homogenen Systemen und betrachten heterogene Systeme zu einem späteren Zeitpunkt.

ACID-Eigenschaften von Transaktionen

Ein (zentrales oder verteiltes) Datenbanksystem muss die folgenden vier Eigenschaften für Transaktionen gewährleisten:

Definition 6.4 (ACID-Eigenschaften)

- *Atomarität*: Alles-oder-nichts-Effekt von Änderungen über ein oder mehrere Systeme hinweg, einfache (aber nicht vollständig transparente) Fehlerbehandlung
- *Konsistenzerhaltung*: Abbruch von Transaktionen, die (globale) Konsistenzbedingungen verletzen
- *Isolation*: Nur konsistente Daten sind für eine Anwendung sichtbar, als ob sie als einzige im System laufen würde; Parallelität wird vor den Anwendungsprogrammierern versteckt
- *Dauerhaftigkeit (Persistenz)*: Mit Commit abgeschlossene Änderungen überstehen Systemfehler und Ausfälle

ACID-Eigenschaften von Transaktionen . . .

Transaktionen werden durch besondere SQL-Anweisungen oder API-Aufrufe markiert. Das System stellt dann ACID-Garantien für die Transaktionen bereit (“ACID-Vertrag”).

- begin transaction
- commit transaction (“commit work” in SQL)
- rollback transaction (“rollback work” in SQL)

Wir betrachten in dieser Vorlesung, wie ACID-Garantien für globale Transaktionen bereitgestellt werden, indem wir die ACID-Garantien der lokalen Systeme ausnutzen.

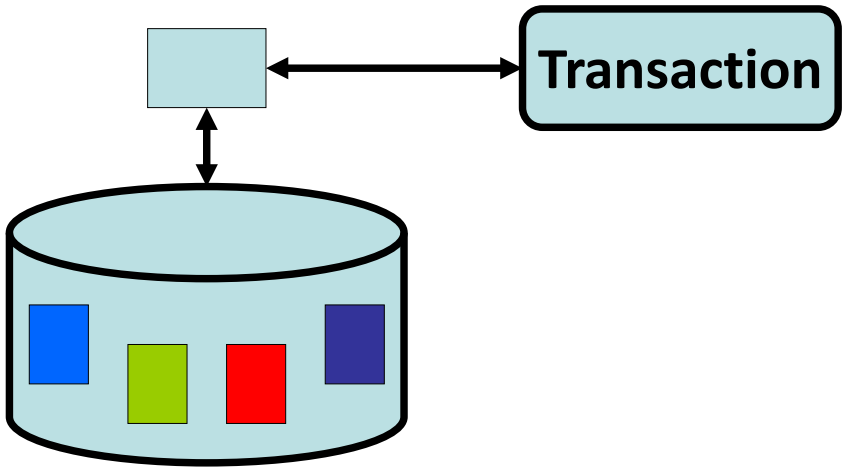
Hintergrund zu zentraler Transaktionsverwaltung

Zentrale Atomarität und Dauerhaftigkeit

Zentrale Atomarität und Dauerhaftigkeit

Um Atomarität und Dauerhaftigkeit zu gewährleisten, muss ein zentrales Datenbanksystem die folgenden beiden Probleme berücksichtigen:

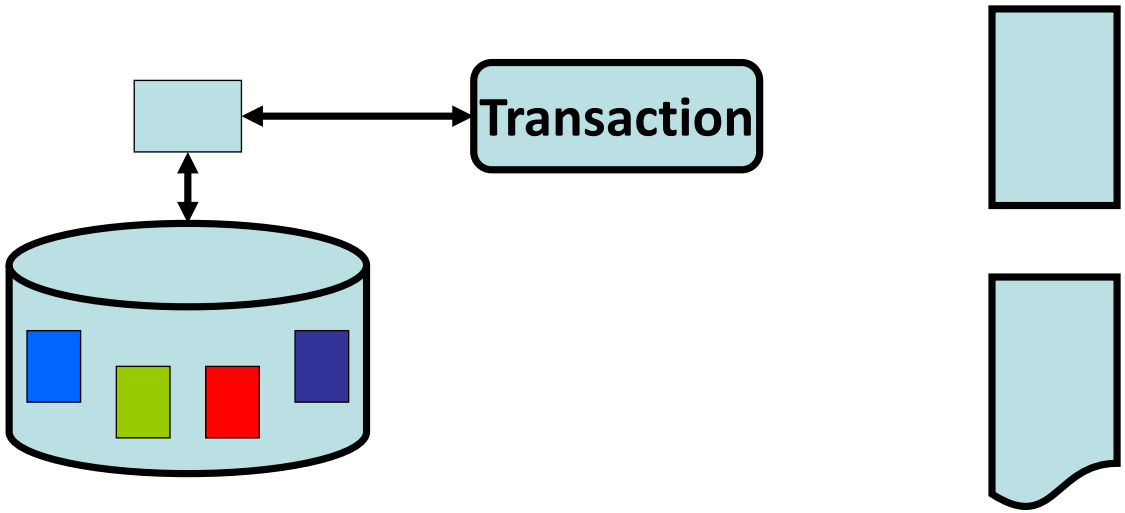
- Änderungen werden an im Speicher gecachten Kopien der Tupel/Seiten vorgenommen und nur gelegentlich auf die stabile Platte geschrieben
  - Wenn eine Transaktion abbricht, müssen alle Änderungen, die bereits auf Platte geschrieben wurden, rückgängig gemacht werden
  - Wenn eine Transaktion mit Commit beendet wurde, müssen alle Änderungen auf Platte geschrieben werden
- Wenn die Software, die Platte, oder der Rechner ausfällt, muss es möglich sein, den Zustand der Datenbank mit allen Änderungen von mit Commit beendeten Transaktionen (und keinen anderen) wieder herzustellen.



Zentrale Atomarität und Dauerhaftigkeit

Lösung: Das System unterhält eine Logdatei in persistentem Speicher.

- Es schreibt alle Änderungen in die (gecachte) Logdatei.
- Es überträgt den gecachten Inhalt der Logdatei auf die Platte, wenn eine Änderung aus dem Tupel- bzw. Seitencache auf die Platte geschrieben oder eine Transaktion mit Commit beendet werden soll.
- Es verwendet die Logdatei, um Änderungen von abgebrochenen Transaktionen zurückzunehmen.
- Es verwendet die Logdatei, um Änderungen nach einem (Software- oder Platten-)Fehler zu wiederholen.



Zentrale Isolation

Definition 6.5 (Transaktion)

Eine *Transaktion* T ist eine Sequenz von Schritten (Aktionen, Operationen) der Form $r(x)$ (lies x) oder $w(x)$ (schreibe x), wobei x ein Objekt (eine Seite, ein Tupel) der Datenbank ist.

Beispiel 6.6

$r(s)\ w(s)\ r(t)\ w(t)$

Wir verwenden die folgende allgemeine Semantik der Schritte einer Transaktion:

Definition 6.7 (Interpretation)

Die *Interpretation* des j-ten Schritts, p_j, einer Transaktion T ist wiefolgt definiert:

- Ist $p_j = r(x)$, so wird der aktuelle Wert von Objekt x einer lokalen Variable v_j zugewiesen, d.h. $v_j := x$
- Ist $p_j = w(x)$, so wird dem Objekt x der Wert $x := f_j(v_{j_1}, ..., v_{j_k})$ zugewiesen, wobei f_j eine unbekannte (anwendungsabhängige) Funktion ist und $j_1, \ldots, j_k$ den vorherigen Leseoperationen von T entsprechen.

Notation und (vereinfachtes) Modell ...

Zur Laufzeit können Transaktionen folgende Zustände annehmen

- *aktiv* (die zugehörige Anwendung arbeitet)
- *committed*, d.h. erfolgreich beendet (die Pseudooperation c_i stellt das *Commit* von Transaktion T_i dar)
- *abgebrochen* oder *zurückgesetzt*, d.h. sie wurde erfolglos beendet und alle ihre Effekte wurden zurückgenommen (die Pseudooperation a_i stellt den Abbruch (den *Abort*) von Transaktion T_i dar)

Definition 6.8 (Schedule)

Ein *Schedule* s ist eine Sequenz von (möglicherweise verschränkten) Schritten mindestens einer Transaktion.
Die Operationsmenge $op(s)$ eines Schedules s besteht aus den Operationen seiner Transaktionen.

Beispiel 6.9

$r_1(s)\ w_1(s)\ r_2(s)\ r_2(t)\ w_2(t)\ c_2\ r_1(t)\ w_1(t)\ c_1$

Definition 6.10 (Serieller Schedule)

Ein Schedule s ist *seriell*, wenn, für zwei beliebige Transaktionen T_i und T_k aus s ($i \neq k$), alle Operationen von T_i (einschließlich c_i oder a_i) vollständig vor allen Operationen von T_k in s angeordnet sind, oder umgekehrt.

Transaktionen in einem seriellen Schedule sind *isoliert* voneinander nach Konstruktion, daher ist die Isolation trivialerweise garantiert. Das Ziel der Concurrency Control ist daher sicherzustellen, dass alle Schedules, die während der Ausführng entstehen, *äquivalent zu einem seriellen Schedule* sind (serialisierbar), ohne die Ausführung auf serielle Schedules zu beschränken (wäre zu ineffizient).

Definition 6.11 (Konflikte und Konfliktrelation)

Sei s ein Schedule und T, T' Transaktionen in s, $T \neq T'$. Zwei Datenoperationen $p \in T$ und $q \in T'$ sind *in Konflikt* in s, wenn sie auf das gleiche Objekt zugreifen und mindestens eine von ihnen eine Schreiboperation ist.

Die *Konfliktrelation* von s ist definiert als $conf(s) := \{(p, q) | p, q \in op(s)$ sind in Konflikt und p vor q in $s\}$.

Zwei Schedules s und s' sind *konflikt-äquivalent*, geschrieben als $s \approx_c s'$, wenn sie die gleiche Menge von Operationen und die gleiche Konfliktrelation haben, d.h. wenn $op(s) = op(s')$ und $conf(s) = conf(s')$.

Ein Schedule s ist *konflikt-serialisierbar*, wenn es einen seriellen Schedule s' gibt mit $s \approx_c s'$.
CSR bezeichnet die Klasse der konflikt-serialisierbaren Schedules.

Definition 6.12 (Konfliktgraph)

Sei s ein Schedule. Der Konfliktgraph $G(s) = (V, E)$ ist ein gerichteter Graph, dessen Knoten die mit Commit beendeten Transaktionen von s sind und dessen Kanten wiefolgt definiert sind: $E := \{(T, T') | T \neq T'$ und es gibt Operationen $p \in T, q \in T'$ mit $(p, q) \in conf(s)\}$.

Theorem 6.13

Sei s ein Schedule. Dann $s \in CSR$ genau dann, wenn $G(s)$ azyklisch ist.

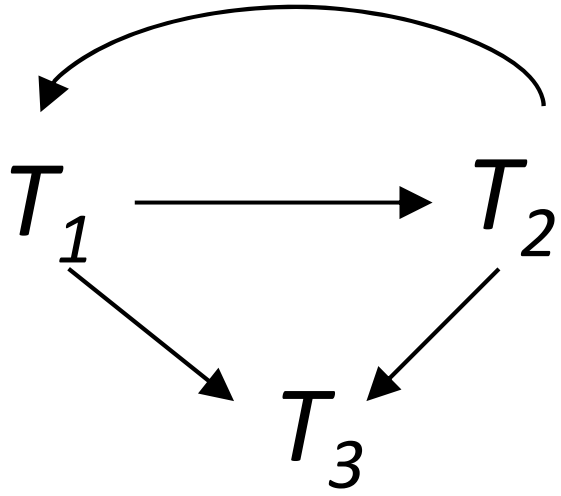
Korollar 6.14

Es kann in polynomieller Zeit in der Anzahl der Transaktionen des Schedules geprüft werden, ob ein Schedule in CSR ist. Wieso?

Beispiel 6.15

$s = r_1(y)\ r_3(w)\ r_2(y)\ w_1(y)\ w_1(x)\ w_2(x)\ w_2(z)\ w_3(x)\ c_1\ c_3\ c_2$

Ist s in CSR?



Konfliktgraph

Der folgende Algorithmus zur Concurrency Control stellt sicher, dass der resultierende Schedule konfliktserialisierbar ist:

Algorithmus 6.16

Für jede Operation einer Transaktion, die ausgeführt werden soll, werden die entsprechenden Kanten zu $G(s)$ hinzugefügt, bevor sie ausgeführt wird. Die Transaktion wird abgebrochen, wenn eine solche Kante zu einem Zyklus im Graph führt.

(Zwei-Phasen-) Sperren

Sperren ist eine einfache, aber effektive Methode, um konflikt-serialisierbare Schedules zu erzeugen. Wir betrachten das Zweiphasen-Sperrprotokoll (2PL, 2-phase-locking).

Algorithmus 6.17 (2PL)

- Vor jeder Operation fordert die Transaktion eine Sperre auf dem zu bearbeitenden Objekt in einem Modus an, der zur Operation passt (lesen oder schreiben).
- Wenn das Objekt nicht bereits in einem unverträglichen Modus gesperrt ist, wird die Sperre gewährt; andernfalls gibt es einen Sperrkonflikt und die Transaktion wird blockiert, bis die Sperre freigegeben wird. Nur Lesesperren sind untereinander verträglich.
- Eine Transaktion muss alle ihre Sperren freigeben, bevor sie endet.
- Eine Transaktion kann keine neuen Sperren mehr anfordern, wenn sie eine Sperre freigegeben hat.

Theorem 6.18
2PL erzeugt nur Schedules, die konfliktserialisierbar sind.

Bemerkung 6.19
Es gibt Varianten von 2PL, die Unterklassen von *CSR* erzeugen, die stärkere Eigenschaften haben:

- Bei striktem Zweiphasensperren (S2PL) behalten Transaktionen ihre Schreibsperren bis zu ihrem Ende.
- Bei starkem Zweiphasensperren (SS2PL) behalten Transaktionen alle Sperren bis zu ihrem Ende.

Theorem 6.20
SS2PL erzeugt nur Schedules, bei denen die Reihenfolge der Transaktionen im äquivalenten seriellen Schedule der Commitreihenfolge entspricht (COCSR).

Definition 6.21 (Regel für Timestamp ordering (TO Regel))

Jeder Transaktion T_i wird ein eindeutiger Zeitstempel (timestamp) $ts(T_i)$ zugewiesen (z.B. die Anfangszeit von T_i).

Wenn $p_i(x)$ und $q_k(x)$ in Konflikt sind, dann muss folgendes für jeden Schedule s gelten, in dem sie auftauchen: $p_i(x)$ vor $q_k(x)$ in s genau dann wenn $ts(T_i) < ts(T_k)$.

Algorithmus 6.22 (Basic timestamp ordering Protokoll (BTO))

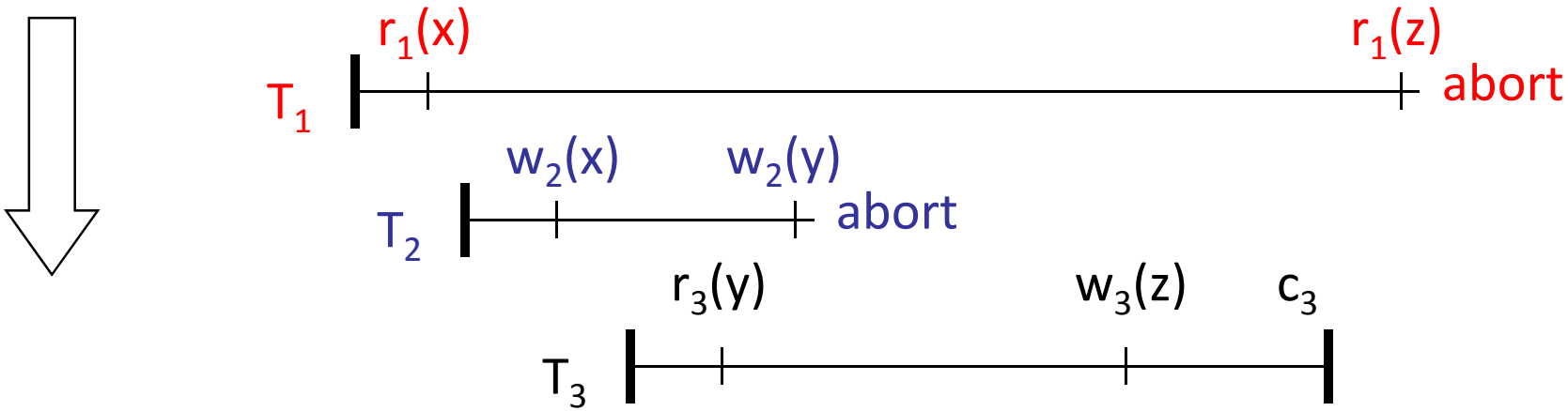
- Unterhalte für jedes Objekt x max-r$(x) := max\{ts(T_k)|r_k(x)$ ist gescheduled worden$\}$ und max-w$(x) := max\{ts(T_k)|w_k(x)$ ist gescheduled worden$\}$.
- Die Operation $p_i(x)$ wird verglichen mit max-q(x) für jedes in Konflikt stehende q:
  - wenn $ts(T_i) <$ max-q(x) für ein q dann breche T_i ab
  - andernfalls führe $p_i(x)$ aus und setze max-p$(x) = ts(T_i)$

Theorem 6.23

BTO erzeugt konfliktserialisierbare Schedules.

Beispiel für BTO

$s = r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$



$r_1(x)\ w_2(x)\ r_3(y)\ a_2\ w_3(z)\ c_3\ a_1$

Beobachtungen:

- Schedules sind oft nicht serialiserbar, weil “alte” Daten nicht mehr verfügbar sind, nachdem sie überschrieben wurden
- Serialisierbarkeit ist für manche Anwendungen nicht notwendig

Snapshot Isolation (SI):

- Transaktionen sehen immer den Zustand der Datenbank zum Zeitpunkt ihres Beginns.
- Zwei parallele Transaktionen dürfen nicht das gleiche Objekt schreiben.

Theorem 6.24

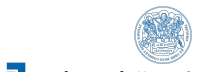
SI erlaubt Schedules, die nicht “serialisierbar” sind (wir haben Serialisierbarkeit nicht formal für Mehrversions-Schedules definiert).

Beispiel 6.25

$s = r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(y)\ c_1\ c_2$

Verteilte Atomarität

Das Zweiphasen-Commitprotokoll



Das grundsätzliche Problem des verteilten Commits

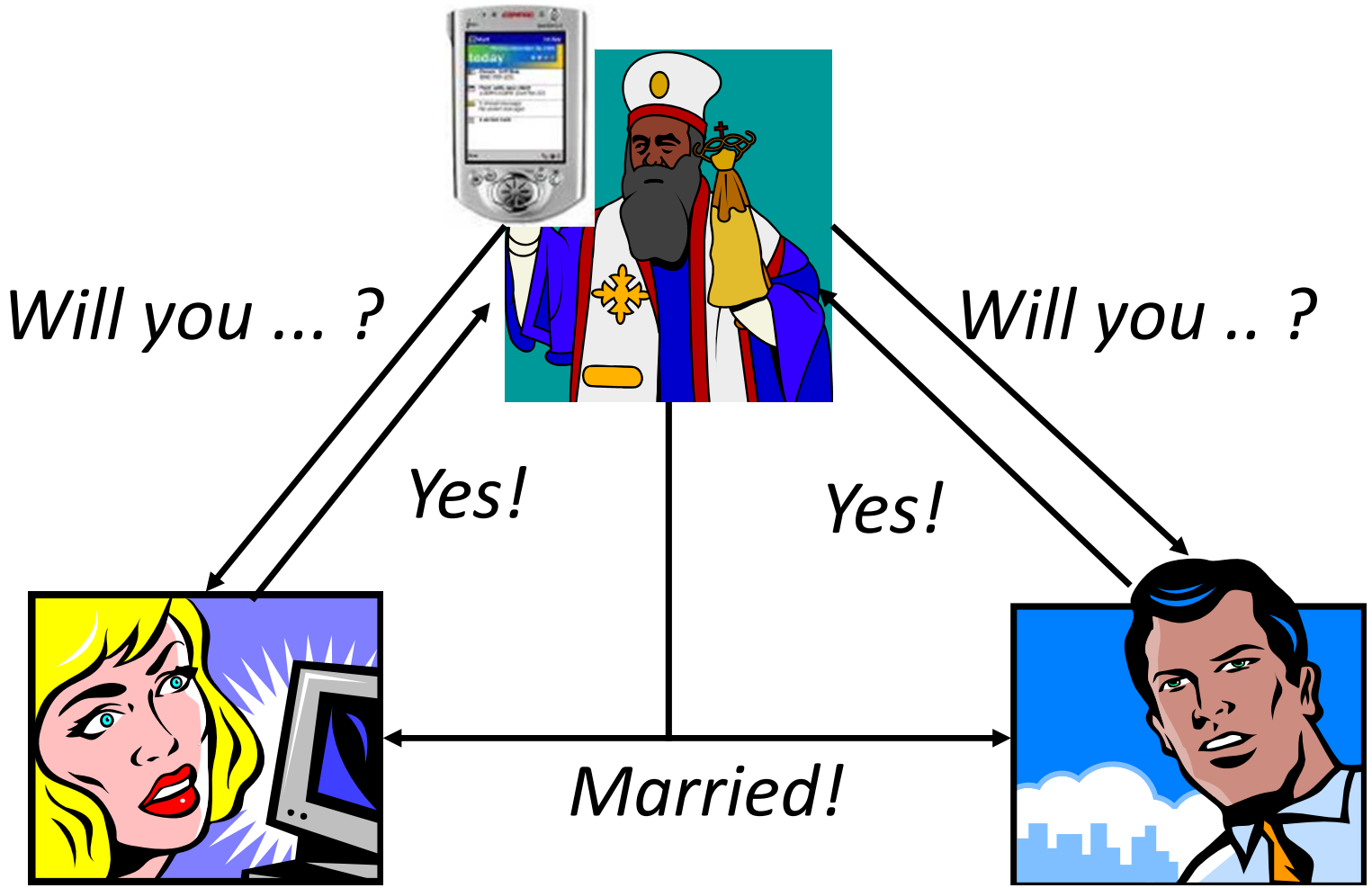
Problem:

- Transaktion arbeitet auf mehreren Servern (Ressourcenmanagern)
- Globales Commit erfordert einstimmige Entscheidung zu lokalen Commits aller Teilnehmer (Agenten)
- Das verteilte System könnte teilweise ausfallen (Serverausfälle, Netzausfälle) und damit zu inkonsistenten Entscheidungen führen

Lösungsansatz:

- Verteiltes Handshake-Protokoll, bekannt als Zweiphasen-Commit (2PC)
- Koordinator übernimmt Verantwortung für einstimmiges Ergebnis
- Transaktionen mit unklarem Ausgang müssen zurückgesetzt werden

Idee: E-Marriage über das Web



Algorithmus 6.26 (Zweiphasen-Commitprotokoll, 2PC)

Erste Phase (Abstimmung):

- Der Koordinator schreibt einen **Begin-Logeintrag** ins stabile Log (forced), schickt **Prepare**-Nachrichten an alle Teilnehmer, und wartet auf ihre Antworten (**yes** oder **no** Stimmen)
- Teilnehmer senden yes/no, schreiben **Prepared-Logeintrag** und markieren Transaktion als **in-doubt** (mit unklarem Ausgang)
  $\Rightarrow$ potentielle Gefahr der Blockierung, Verletzung der lokalen Autonomie

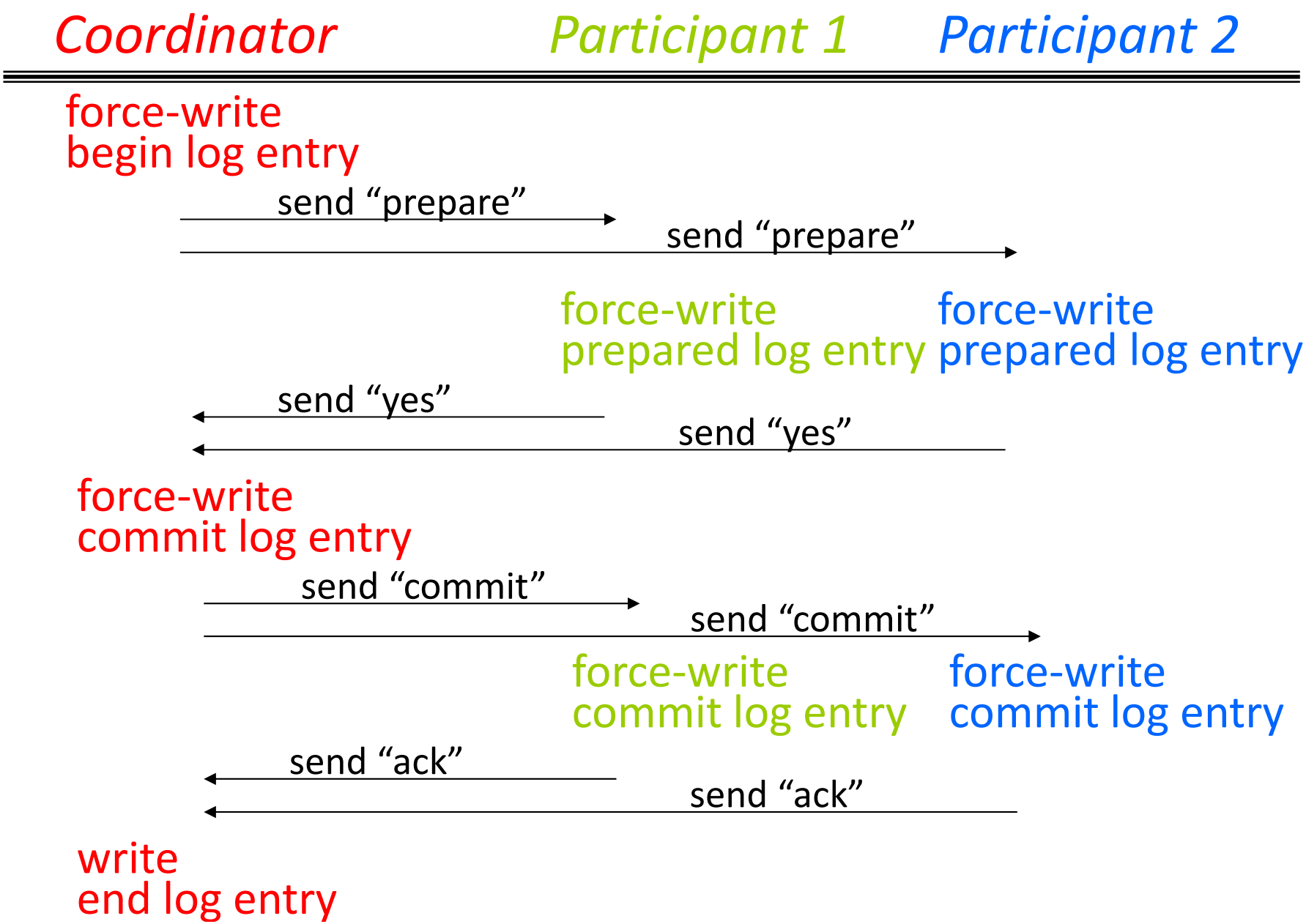
Zweite Phase (Entscheidung):

- Der Koordinator entscheidet auf Commit, wenn alle Antworten yes sind, und auf Abort sonst
- Der Koordinator schreibt einen **Commit**- oder **Abort-Logeintrag** ins stabile Log, sendet **Commit-** oder **Abort-Nachrichten** an die Teilnehmer und wartet auf ihre Bestätigungen (**ack**)
- Die Teilnehmer schreiben **Commit-** oder **Abort-Logeinträge** in ihr stabiles Log und senden ack
- Der Koordinator kann nun dem Client den Ausgang der Transaktion mitteilen und schreibt einen **Ende-Logeintrag** in das Log, um Garbage Collection zu ermöglichen

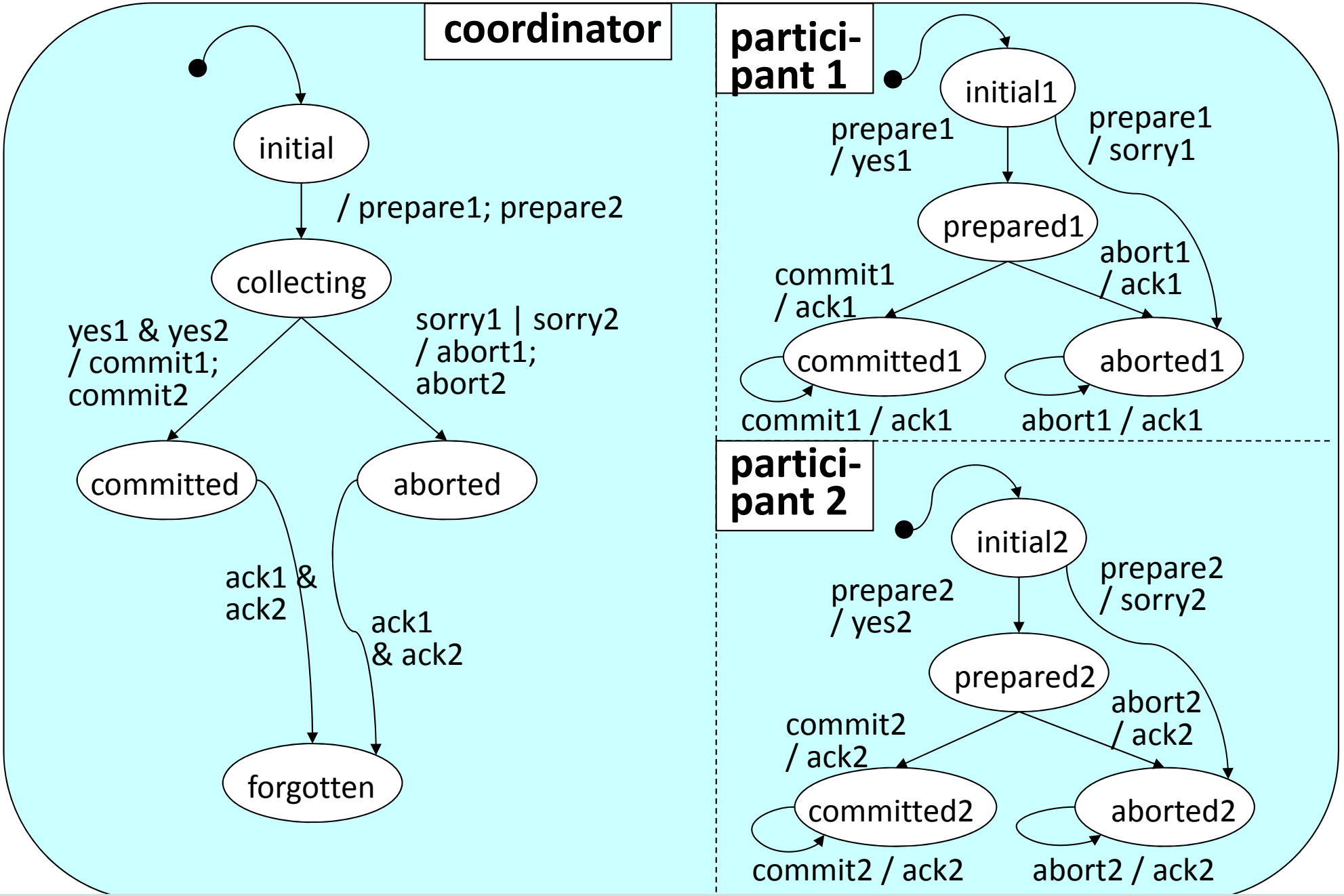
Bemerkung 6.27

Mit n Teilnehmern und einem Koordinator erfordert 2PC $4n$ Nachrichten, $2n + 2$ Logeinträge im stabilen Log, und einen weiteren Logeintrag.

Illustration von 2PC



Darstellung von einfachem 2PC als Statechart



Protokoll zum Restart und zur Terminierung

In einem realistischen System können Komponenten (Prozesse und das Netz) ausfallen. Das 2PC-Protokoll kann erweitert werden, um solche Ausfälle zu berücksichtigen.

Definition 6.28 (Fehlermodell)

Wir betrachten die folgenden Fehlerarten:

- *Prozessfehler*: Vorübergehende (transiente) Serverausfälle
- *Netzfehler*: Verluste oder Duplikate von Nachrichten

Bemerkung 6.29

Wir nehmen an, dass es keine Byzantinischen Fehler gibt (d.h. jeder Fehler wird erkannt und hält das System an) und keine böswilligen Teilnehmer vorkommen, die sich nicht an das Protokoll halten.
Um diese Annahme abzuschwächen, kann man byzantinische Agreement-Protokolle verwenden.

Bemerkung 6.30

Wir machen keine Annahmen, wie Netzfehler erkannt und behandelt werden. Ein System könnte zum Beispiel sitzungsbasierte Kommunikation verwenden.

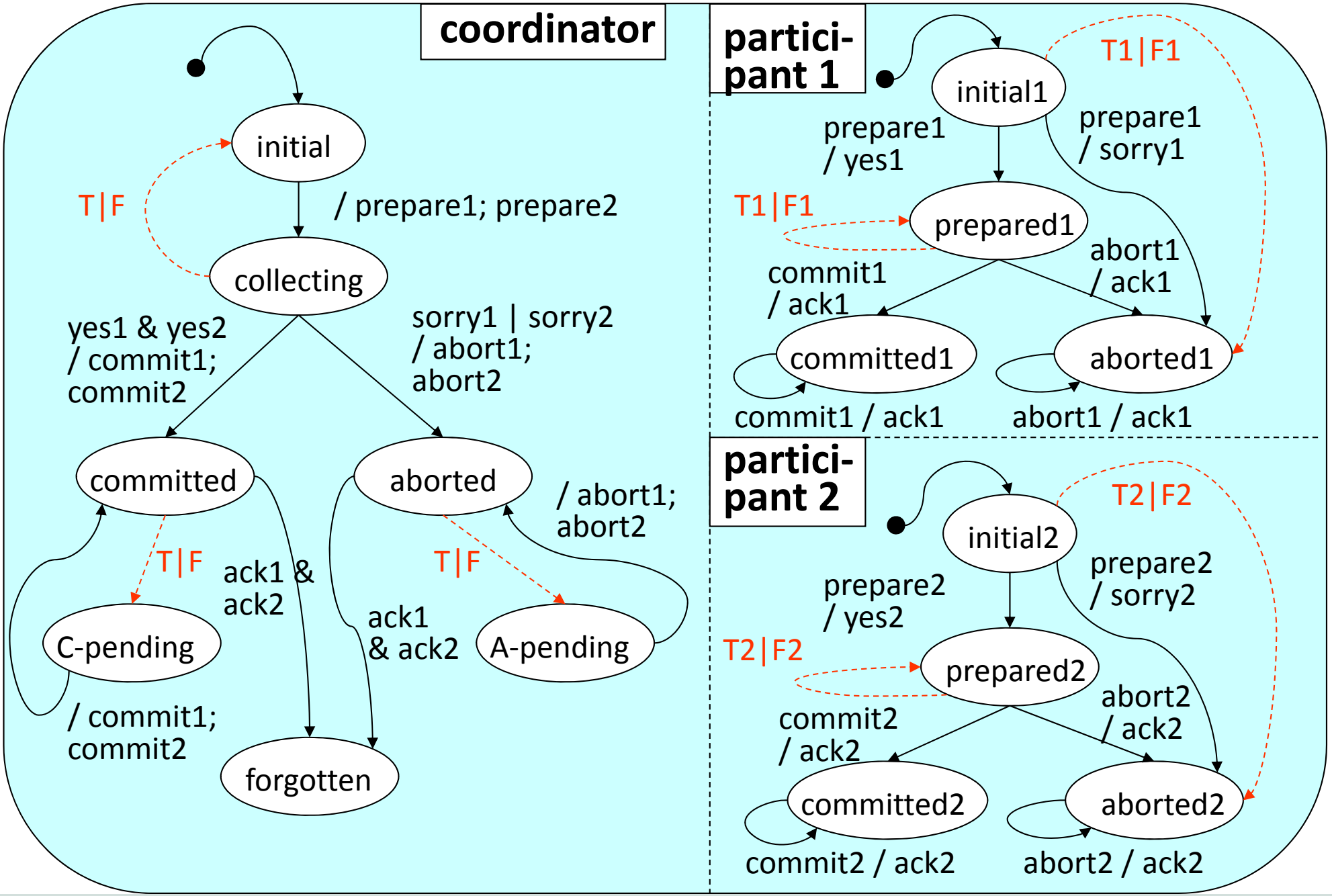
Algorithmus 6.31 (Restart-Protokoll nach Fehlern (F-Transitionen))

- Der *Koordinator* gelangt beim Neustart in seinem letzten gemerkten Zustand (aus dem Log) und sendet die ensprechenden Nachrichten erneut.
- Ein *Teilnehmer* gelangt beim Neustart in seinen letzten gemerkten Zustand (aus dem Log) und sendet die entsprechenden Nachrichten erneut oder wartet auf eine Nachricht des Koordinators.

Algorithmus 6.32 (Terminierungsprotokoll bei Timeout)

- Der *Koordinator* sendet Nachrichten erneut und kann in der ersten Phase entscheiden, die Transaktion abzubrechen.
- Ein *Teilnehmer* kann die Transaktion in der ersten Phase einseitig abbrechen und in der zweiten Phase auf den Koordinator warten oder ihn kontaktieren.

Statechart für einfaches 2PC mit Restart/Terminierung



Bemerkung 6.33 (Blockierung im 2PC)

Eine Blockierung, die nicht aufgelöst werden kann, tritt immer auf, wenn alle Teilnehmer mit yes abgestimmt haben und danach der Koordinator ausfällt, bevor er mindestens einem Teilnehmer seine Entscheidung verkündet hat. Die Teilnehmer dürfen nun keine eigene Entscheidung treffen, sondern müssen die Rückkehr des Koordinators abwarten.

Theorem 6.34 (Korrektheit von 2PC)

2PC garantiert, dass immer dann, wenn ein Prozess in einen Endzustand gelangt, alle Prozesse entweder in ihrem “Committed”-Zustand oder alle Prozesse in ihrem “Aborted”-Zustand sind.

Beweisidee.

Wir betrachten die möglichen Berechnungspfade des Protokolls, die im globalen Zustand (initial, initial, ..., initial) beginnen, und argumentieren über Invarianten von Zuständen auf den möglichen Berechnungspfaden. □

Theorem 6.35 (Lebendigkeit des 2PC)

Bei einer endlichen Zahl von Fehlern wird das 2PC-Protokoll schließlich in einer endlichen Zahl von Zustandsübergängen einen globalen Endzustand erreichen.

Unabhängige Recovery

Definition 6.36 (Unabhängige Recovery)

Ein wegen eines Fehlers neugestarteter Prozess kann eine *unabhängige Recovery* durchführen, wenn er seinen Teil des Protokolls beenden kann, ohne mit anderen Prozessen zu kommunizieren.

Bemerkung 6.37

Unabhängige Recovery verhindert das Blockieren von neugestarteten Prozessen.

Theorem 6.38 (Unabhängige Recovery bei einzelnem Prozessfehler)

Unter der Annahme, dass höchstens ein einzelner Prozessfehler geschieht, während das Protokoll läuft (und kein Netzfehler), ist es möglich, ein verteiltes Commit-Protokoll zu entwerfen, das unabhängige Recovery für einen fehlerhaften Prozess garantiert.

Beweisidee.

Wir müssen globale Zustände vermeiden, in denen ein lokaler Prozess in einem Endzustand ist, während ein andere noch mehrere Endzustände erreichen kann.
Der Koordinator fügt einen neuen Zustand *willing-to-commit* zwischen den collecting und commit Zuständen ein, und macht eine T-Transition von collecting nach abort. Wenn ein Teilnehmer in seinem prepared-Zustand ausfällt, wird er auf die Nachricht nicht antworten.
Ein Teilnehmer hat einen neuen prepared-to-commit Zustand nach seinem prepared-Zustand, macht eine F-Transition von prepared nach abort, und macht eine F-Transition von prepared-to-commit nach commit. □

Unmöglichkeit der unabhängigen Recovery bei mehreren Fehlern

Theorem 6.39

Es existiert kein verteiltes Commit-Protokoll, das unabhängige Prozessrecovery garantieren kann, wenn es mehrere Fehler gibt (z.B. Partitionierung des Netzes).

Beweisidee.

Betrachte zwei Prozesse, ein Koordinator und ein Teilnehmer, mit dem Berechnungspfad G_0=(initial, initial), ..., G_m=(committed, committed) ohne das Auftreten von Fehlern.

Mit unabhängiger Recovery können beide Prozesse in den Zuständen G_0 bis G_{k-1} in ihre lokalen abort-Zustände gelangen. In Zustand G_m ändert sich dieses Verhalten und bei einem der Prozesse führt die unabhängige Recovery in den lokalen commit-Zustand. Der andere verhält sich wie in Zustand G_{m-1}, da ein nachrichtenbasiertes Protokoll die Komponenten eines globalen Zustandes immer einzeln ändert. Wenn nun beide Prozesse ausfallen, kommt es zu einem inkonsistenten Ergebnis. □

Algorithmus 6.40 (Hierarchisches 2PC)

Während der Ausführung einer Transaktion bildet sich ein Prozessbaum, dessen Wurzel der Initiator der Transaktion ist und bei dem die Kanten bilateralen Kommunikationslinks entsprechen.

Beim Commit kann der Prozessbaum

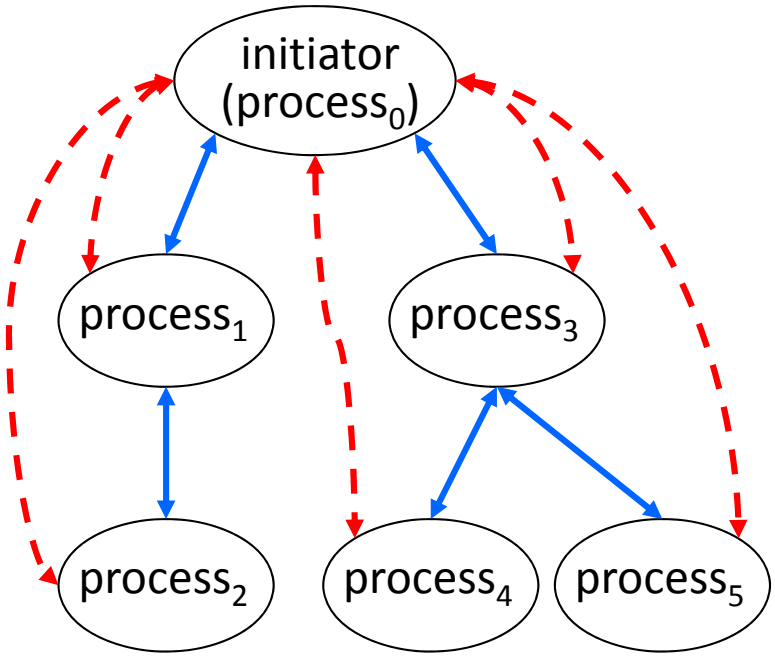
- **flachgeklopft** werden, falls der Koordinator alle Teilnehmer kennt und mit ihnen einfach Kommunizieren kann
- **wiederverwendet** werden, indem eine hierarchische Form des 2PC verwendet wird, in der die Zwischenknoten sowohl als Teilnehmer als auch als Koordinator auftreten
- **restrukturiert** werden, um einen anderen Koordinator als den Initiator auszuwählen, unter Berücksichtigung von Stabilität und Geschwindigkeit von Prozessen und Netzkommunikation

Bemerkung 6.41 (Transfer des Koordinators)

Bei hierarchischem 2PC ändert sich die Rolle des Koordinators während des Ablaufs des Protokolls.

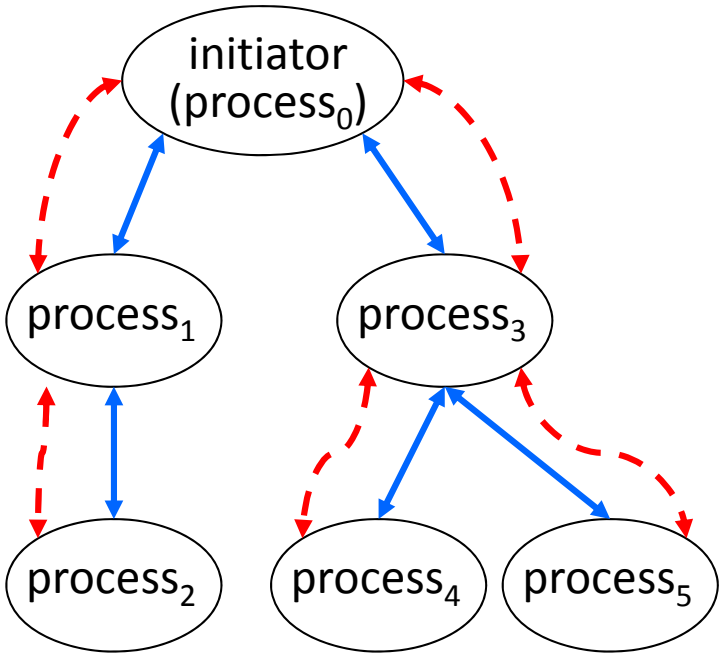
Flattened vs. Hierarchical Commit Tree

Flattened 2PC:



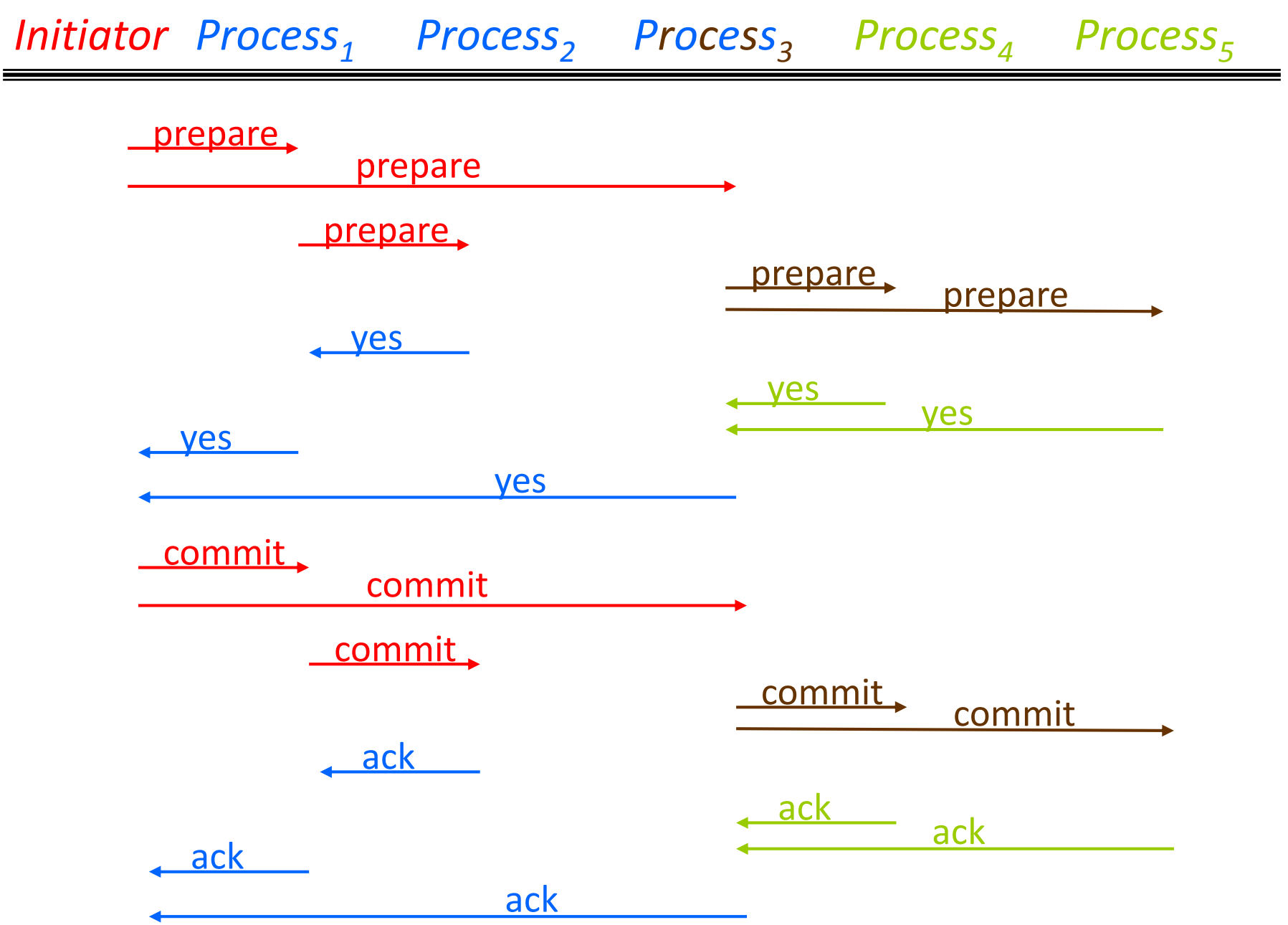
Kommunikation während Transaktionsausführung

Hierarchical 2PC:



Kommunikation während Commitprotokoll

Illustration von hierarchischem 2PC



2PC Engpässe und Optimierungen

Bemerkung 6.42

Das ursprüngliche 2PC-Protokoll hat eine Reihe von Nachteilen:

- Es erfordert eine relativ große Zahl von Nachrichten und Schreiboperationen auf dem stabilen Log.
- Es dauert relativ lange, bis lokale Sperren freigegeben werden können.
- Die Wahrscheinlichkeit für eine Blockierung ist relativ hoch.

Wir diskutieren nun Erweiterungen von 2PC, die diese Probleme teilweise lösen:

- Reduziere die Zahl der Nachrichten und die Zahl der Schreibzugriffe auf das stabile Log, indem Annahmen für den Fall fehlender Information getroffen werden
- Eliminiere so früh wie möglich Unterbäume, in denen nur gelesen wurde
- Transferiere die Rolle des Koordinators an einen schnellen Rechner

Bemerkung 6.43

Wie man leicht sieht, ist es normalerweise nicht notwendig, den begin-Logeintrag direkt ins stabile Log zu schreiben; es ist nie ein Problem, wenn man ihn verliert

2PC mit Presumption

Idee:

- Teilnehmer müssen commit/abort-Logsätze nicht ins stabile Log schreiben, sie können immer den Koordinator fragen
- Wenn der Koordinator die Transaktion bereits vergessen hat, antwortet er mit einem **angenommenen Ausgang der Transaktion**
- Der Koordinator kann Transaktionen manchmal sogar vergessen, ohne dass er die ack-Nachrichten der Teilnehmer empfangen hat

Problem:
Wir müssen sicherstellen, dass das angenommene Ergebnis des Koordinators zu den folgenden beiden Fällen passt:

a) Die Entscheidung über die Transaktion hat noch nicht begonnen (früher Verlierer)

b) Die Entscheidung über die Transaktion wurde bereits getroffen und der Koordinator hat sie vergessen (vergessene Transaktion)

oder der Koordinator muss beide Fälle unterscheiden können

Algorithmus 6.44 (Presumed-abort (PA) Protokoll)

- Das angenommene Ergebnis ist abort
- Abort-Logeinträge müssen nicht direkt ins stabile Log geschrieben werden und es sind keine ack-Nachrichten für Verlierer-Transaktionen notwendig
- Aber: beides notwendig für Gewinner-Transaktionen
- Spart $n + 1$ Schreibzugriffe auf das stabile Log und n Nachrichten für Verlierer-Transaktionen

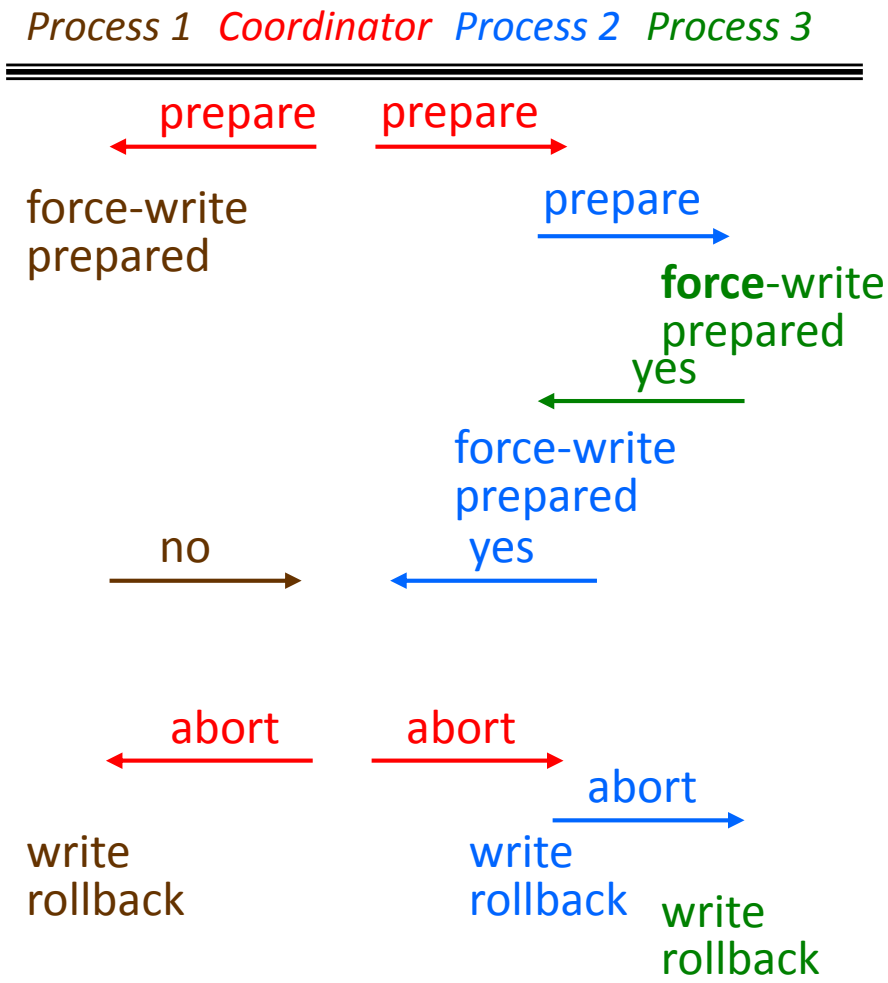
Bemerkung 6.45

Presumed-abort ist das Protokoll des XA-Standards.

Algorithmus 6.46 (Presumed-commit (PC) Protokoll)

- Das angenommene Ergebnis ist commit
- Teilnehmer müssen keine Commit-Logeinträge mehr direkt ins stabile Log schreiben und es sind keine ack-Nachrichten für Gewinner-Transaktionen notwendig
- Der Koordinator muss Begin- und Commit-Einträge ins stabile Log schreiben, um frühe Verlierer und vergessene Gewinner unterscheiden zu können
- Spart $n - 1$ Schreibzugriffe auf das stabile Log und n Nachrichten für Gewinner

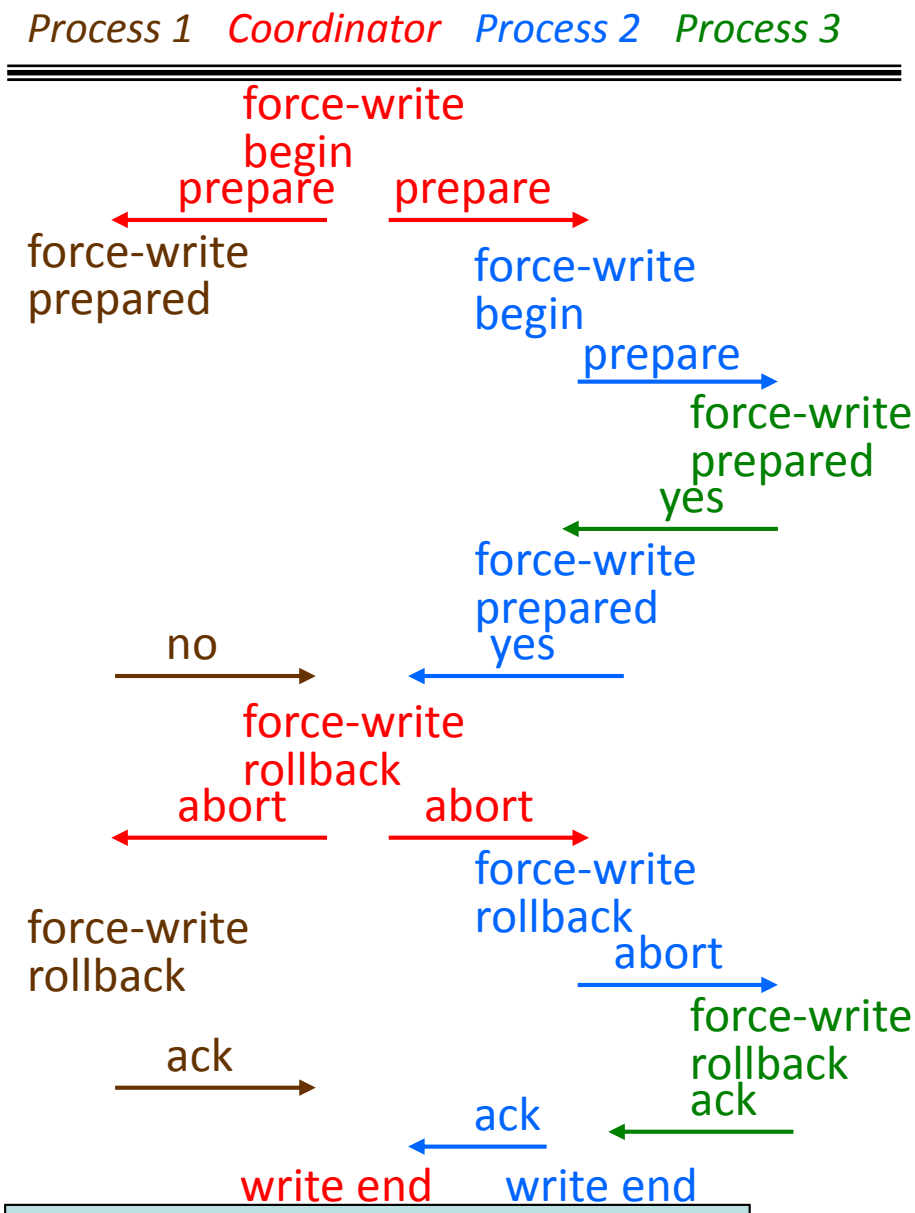
Illustration des Presumed-Abort Protokolls



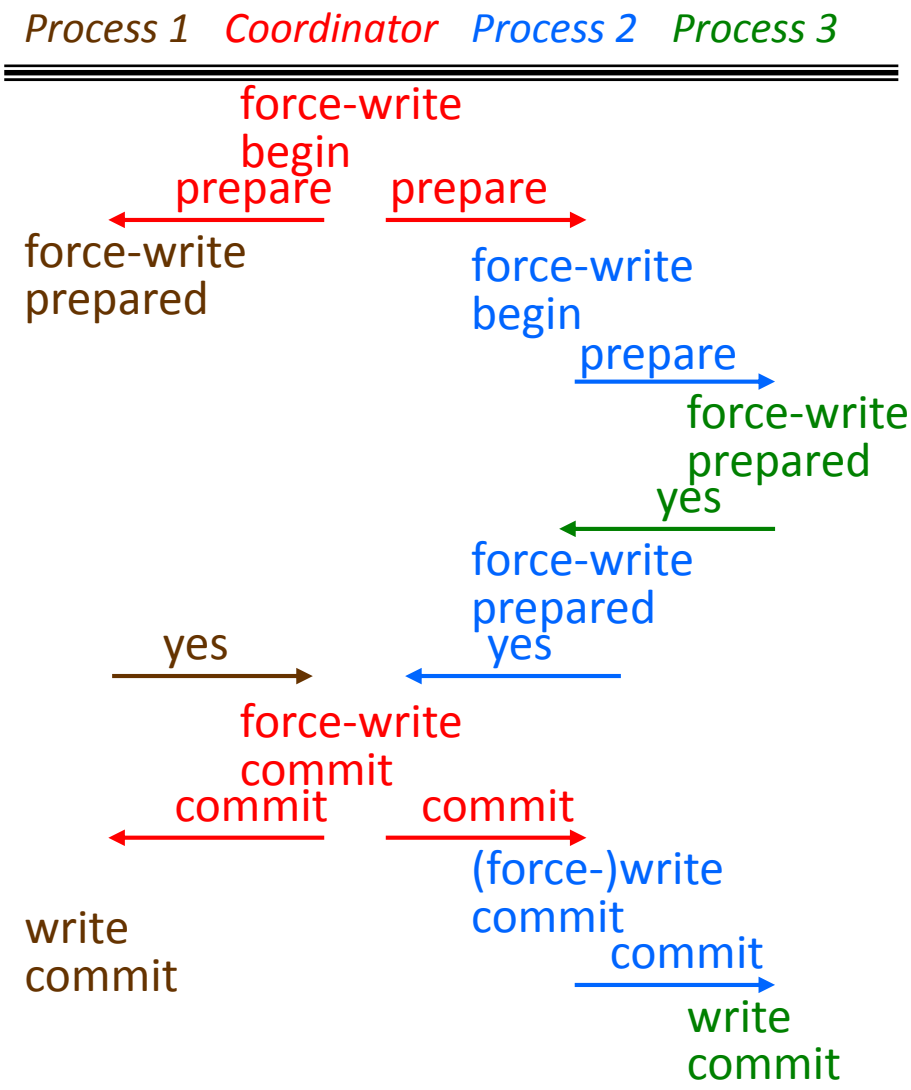
Fall 1: Abort der Transaktion

Fall 2: Commit der Transaktion

Illustration des Presumed-Commit Protokolls



Fall 1: Abort der Transaktion



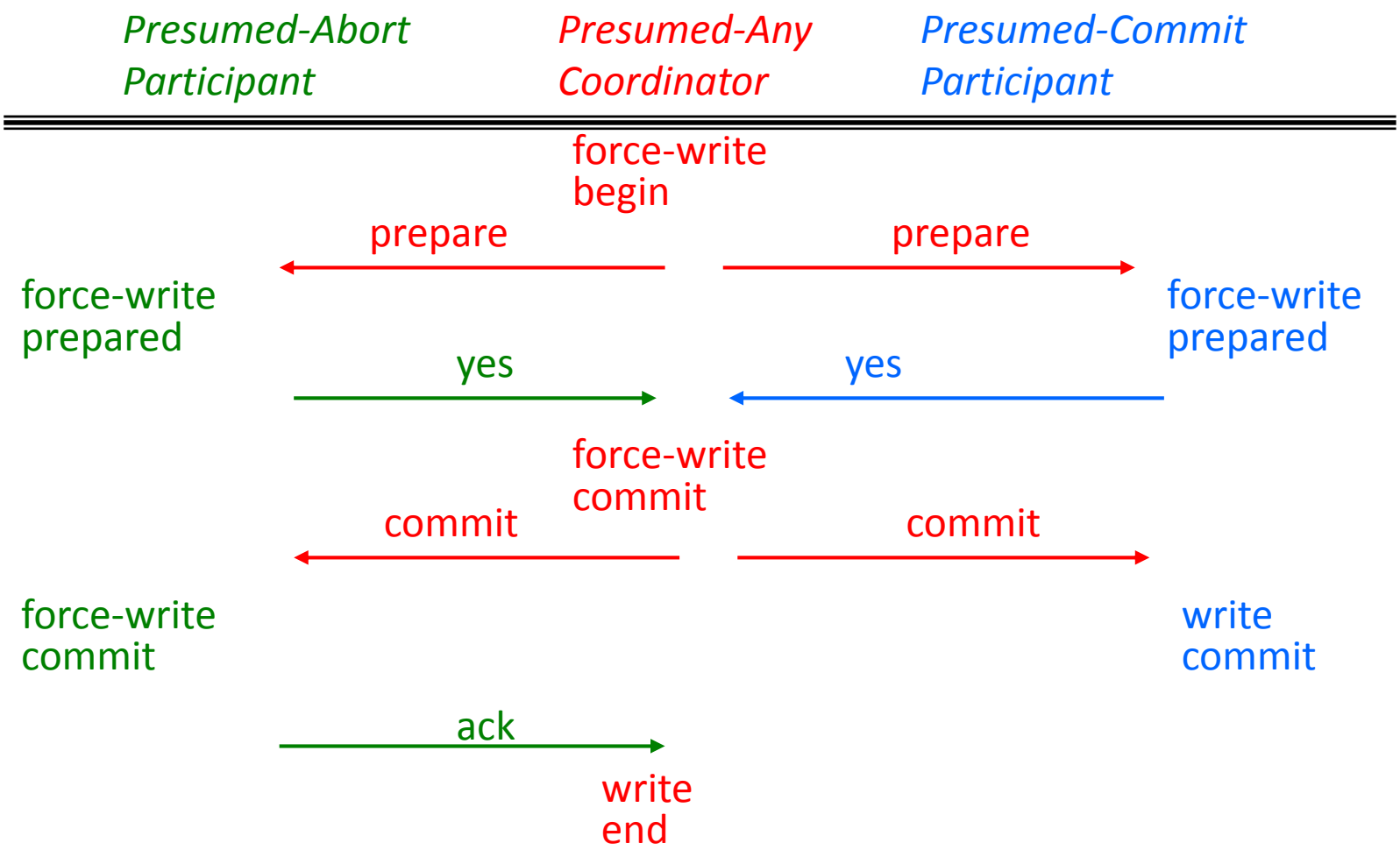
Fall 2: Commit der Transaktion

Universität Trier

Koexistenz von PA- und PC-Teilnehmern in heterogenen Systemem (presumed-any Protokoll)

- Der Koordinator schreibt Logeinträge ins stabile Log, so dass er mit PA- und PC-Teilnehmern umgehen kann
- Der Koordinator muss den Typ des Teilnehmers kennen (wird im Begin-Logsatz vermerkt) und schickt/erwartet Nachrichten gemäß dem Teilnehmertyp

Illustration eines Presumed-Any Koordinators



Optimierung für nur-lesende Teilbäume

- Ein Teilnehmer in einem Blatt des 2PC-Baums, der nur gelesen hat, stimmt mit “read-only” und gibt alle seine Sperren frei.
- Ein Teilnehmer im Innern des 2PC-Baums, der selbst nur gelesen hat, stimmt mit “read-only”, wenn er alle Stimmen seiner Kinder erhalten hat und keine davon “yes” oder “no” ist.
- Der Koordinator eliminiert nur-lesende Unterbäume aus der Entscheidungsphase.
- Read-only Optimierungen sind nützlicher für PA als für PC, weil PC immer noch einen Begin-Logeintrag im stabilen Log benötigt
- PA kann read-only Transaktionen ohne Loggingkosten beenden.

Transfer des Koordinators

Beobachtungen:

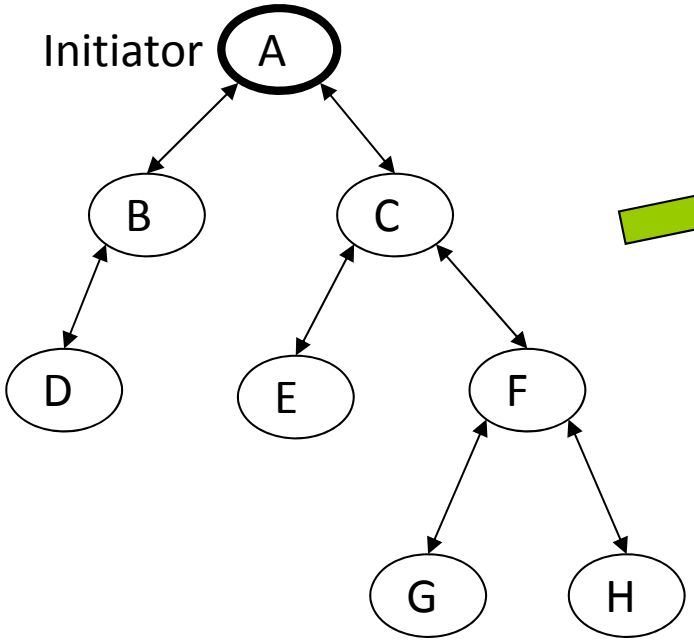
- Der Initiator der Transaktion ist der Default-Koordinator, aber oft keine gute Wahl
- Die Rolle des Koordinators kann im Laufe des Protokolls an einen anderen Prozess übertragen werden auf Basis von Verlässlichkeit, Geschwindigkeit, Netzverbindungen, etc.

Ansätze:

- **rotiere den Prozessbaum** um den Koordinator, verwende bestehende Kommunikationsverbindungen weiter
- Lineares 2PC, bei der der letzte Agent Koordinator wird:
  - Transferiere die Verantwortlichkeiten des Koordinators entlang der Prozesskette
  - Verwende Nachrichten der Art “I’m prepared, you decide”
  - Transaktion mit zwei Teilnehmern erfordert nur 3 Nachrichten und 3 Einträge im stabilen Log
- **Dynamisches 2PC**: verallgemeinert die last-agent Optimierung auf beliebige Bäume

Initiator- vs. Coordinator-Rooted Process Tree

Während der Transaktionsausführung



Während des Commitprotokolls

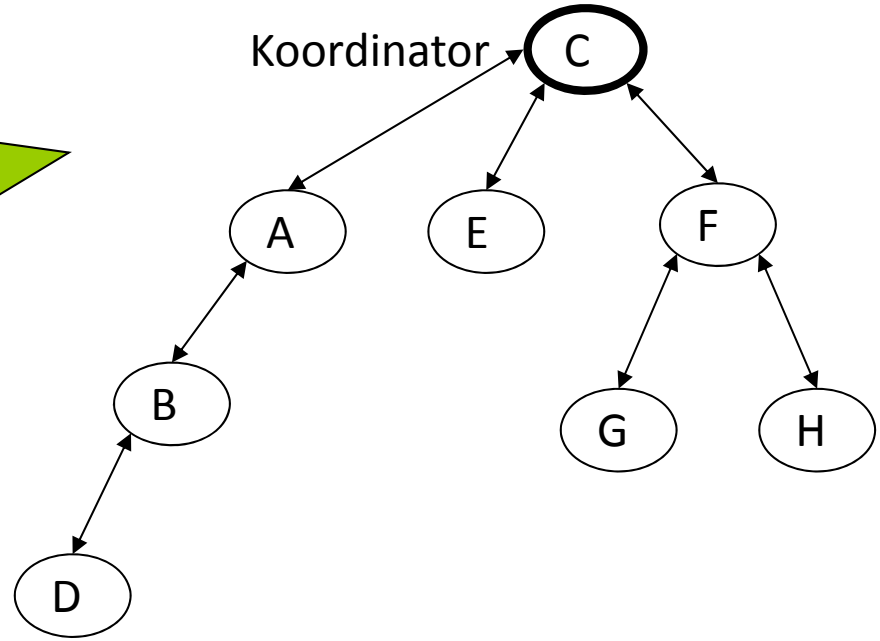
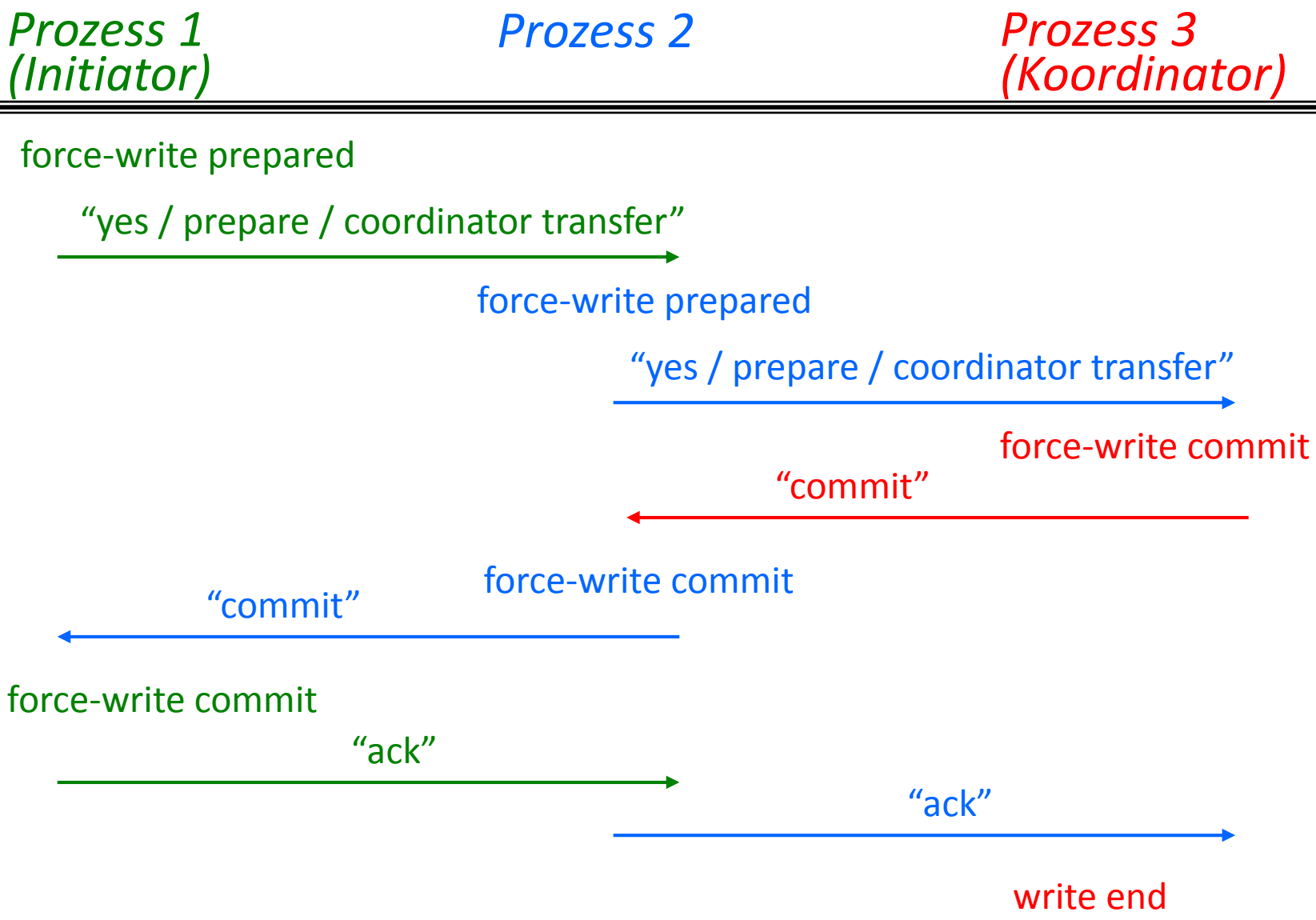


Illustration der Last-Agent Optimierung



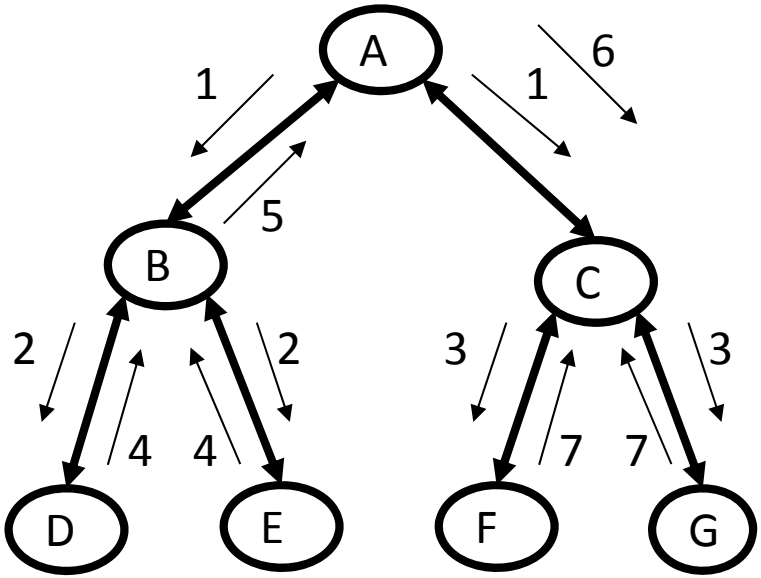
Ziel:

- Bestimme den Koordinator so schnell wie möglich (mit mehr Nachrichten als nötig, um langsame Äste im Baum zu kompensieren)

Algorithmus 6.47 (Dynamisches 2PC-Protokoll)

- Knoten im Prozessbaum senden Prepare-Nachrichten an alle ihre Kinder
- Blattknoten, die eine Preparenachricht erhalten, führen lokal das Prepare durch und antworten ihrem Elternknoten mit yes
- Zwischenknoten, die yes-Stimmen von allen Nachbarn außer einem erhalten haben, führen lokal das Prepare durch und senden eine yes-Stimme an ihren fehlenden Nachbarn (verallgemeinerte last-agent-Optimierung)
- Der Prozess, der zuerst yes-Stimmen von allen seinen Nachbarn hat, wird Koordinator

Illustration von dynamischem 2PC



1: A →prepare B, A →prepare C
2: B →prepare D, B →prepare E
3: C →prepare F, C →prepare G
4: D →yes B, E →yes B
B prepared
5: B →yes A
A prepared
6: A →yes C
7: F →yes C, G →yes C
C becomes coordinator

Verteilte Atomarität: Lessons Learned

- 2PC ist ein fundamentaler, vielseitiger und standardisierter Baustein für verteilte transaktionale Systeme
- Anfälligkeit für Blockierung ist inhärent und der Preis für die Konsistenz (siehe CAP-Theorem)
- Skalierbarkeit ist beschränkt auf wenige Teilnehmer
- Optimierungen zielen auf
  - Reduzierung der Schreibzugriffe auf das stabile Log und der Nachrichten (PA und PC)
  - Verkürzung der Zeit bis zur Freigabe der Sperren (read-only Stimmen, last-agent Optimierung, dynamisches 2PC)
- Transfer des Koordinators ist wesentlich, um die Abhängigkeit von einem schlecht administrierten Initiator oder instabilen Server zu vermeiden