Globale Transaktions-Verwaltung

Mehrbenutzer-Synchronisation,
Recovery/Fehlertoleranz,
Atomarität, Replikat-Konsistenz,

ACID-Paradigma

Atomicity (Atomarität)

- Transaktion ist kleinste, nicht mehr weiter zerlegbare Einheit
- Entweder werden alle Änderungen der Transaktion festgeschrieben oder gar keine
- Man kann sich dies auch als „alles-oder-nichts"-Prinzip merken

Consistency

- Transaktion hinterläßt einen konsistenten Datenbasiszustand
- Anderenfalls wird sie komplett (siehe *Atomarität*) zurückgesetzt
- Zwischenzustände während der TA-Bearbeitung dürfen inkonsistent sein
- Endzustand muß die im Schema definierten Konsistenzbedingungen (z.B. referentielle Integrität) erfüllen

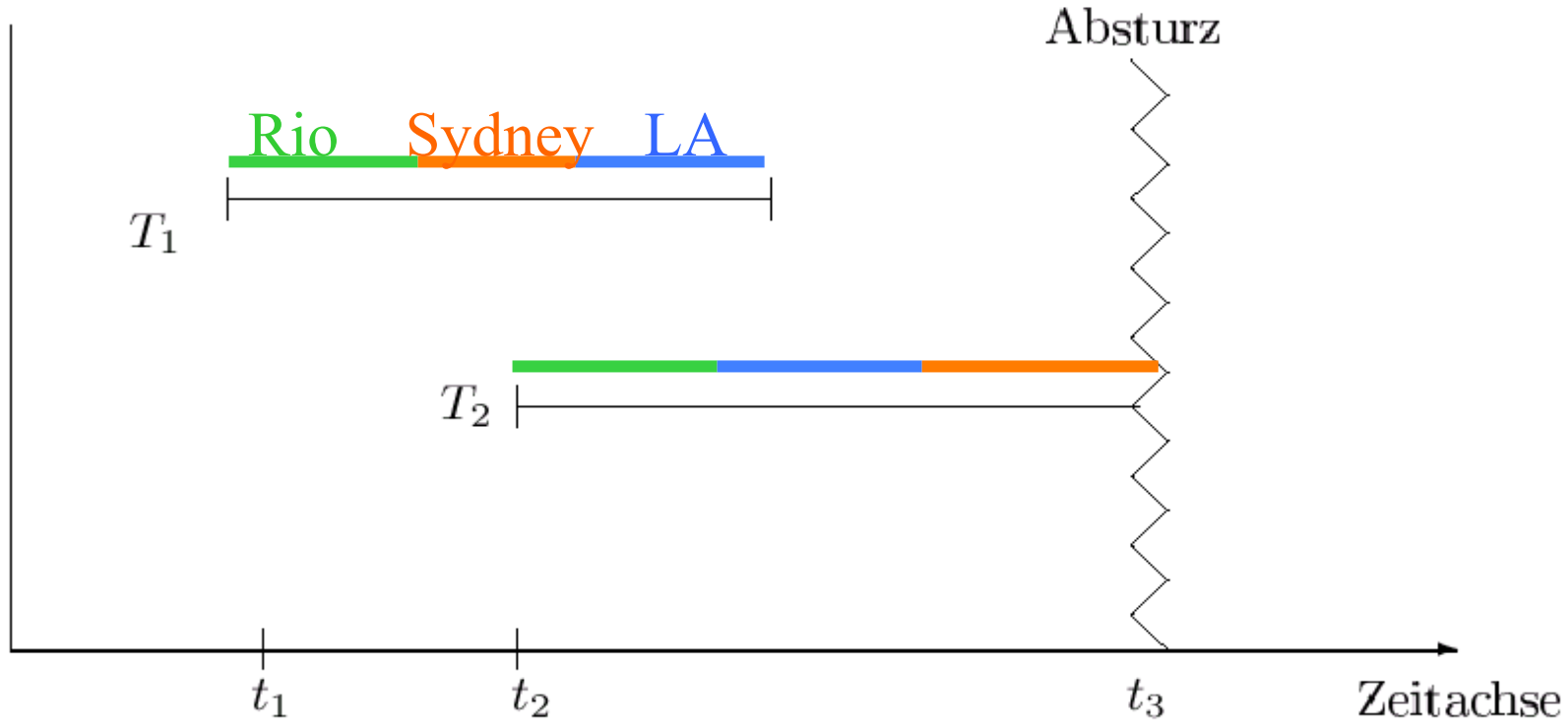
Isolation

- nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen
- alle anderen parallel ausgeführten Transaktionen bzw. deren Effekte dürfen nicht sichtbar sein

Durability (Dauerhaftigkeit)

- Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten
- Transaktionsverwaltung muß sicherstellen, daß dies auch nach einem Systemfehler (Hardware oder Systemsoftware) gewährleistet ist
- Wirkungen einer einmal erfolgreich abgeschlossenen Transaktion kann nur durch eine sogenannte kompensierende Transaktion aufgehoben werden

Transaktionsbeginn und -ende relativ zu einem Systemabsturz



- Transaktionen der Art T_1 müssen hinsichlich ihrer Wirkung vollständig nachvollzogen werden. Transaktionen dieser Art nennt man *Winner*.
- Transaktionen, die wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese Transaktionen bezeichnen wir als *Loser*.

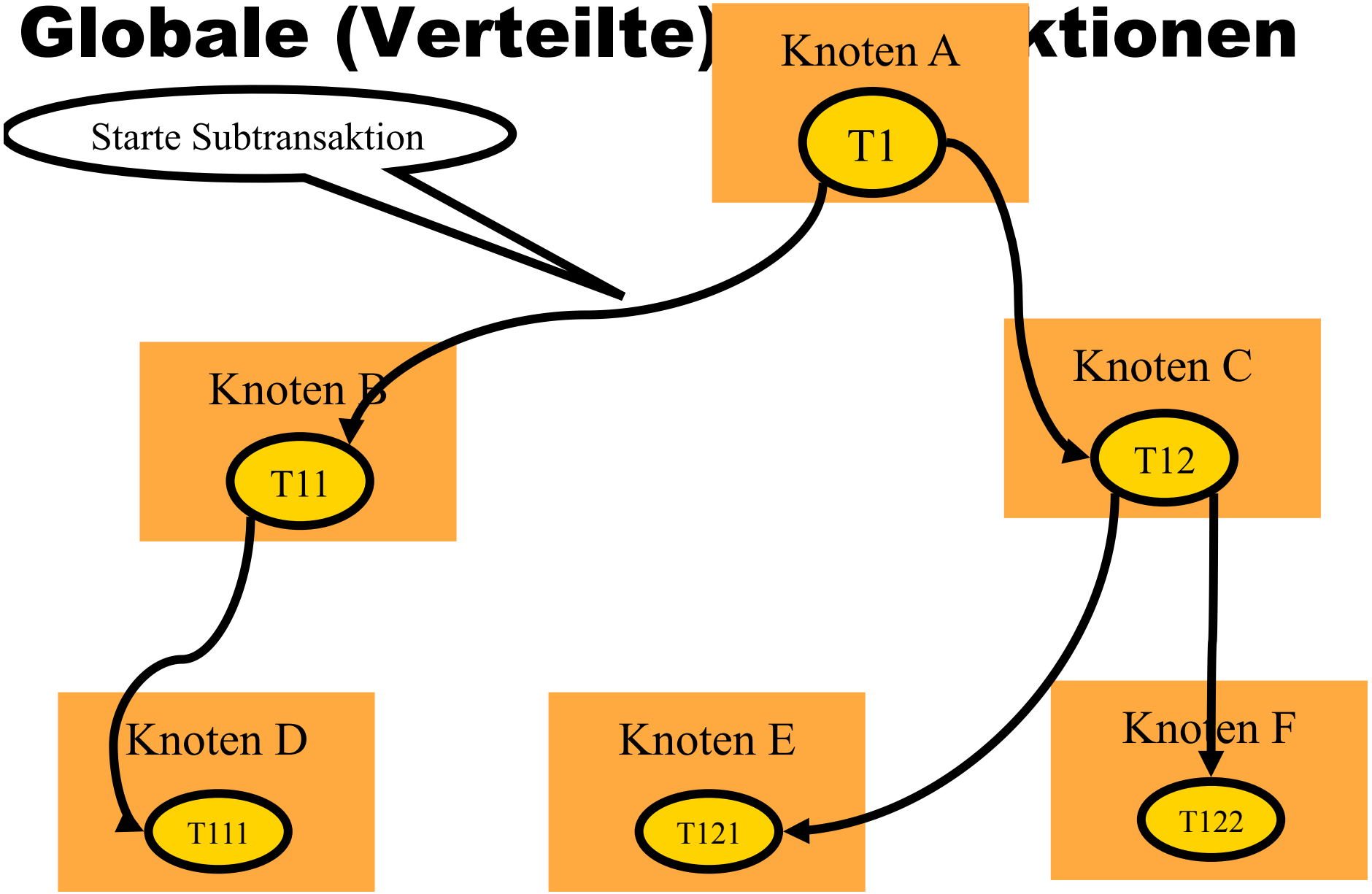
Write Ahead Log-Prinzip

1. Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle „zu ihr gehörenden“ Log-Einträge ausgeschrieben werden.

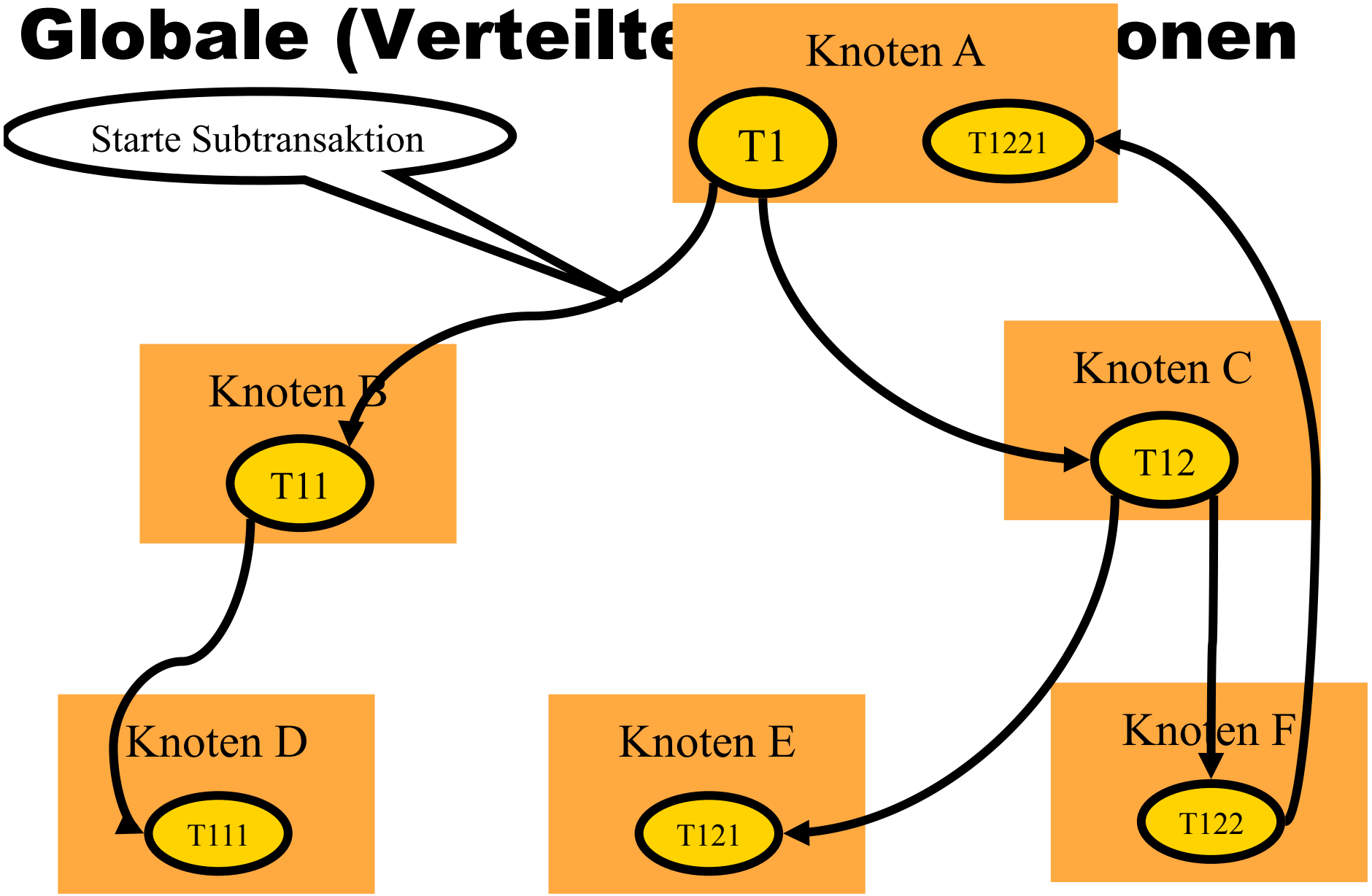
2. Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden.

- Log-Record wird für jede Änderungsoperation geschrieben
  - before-image für das „Undo“
  - after-image für das Redo

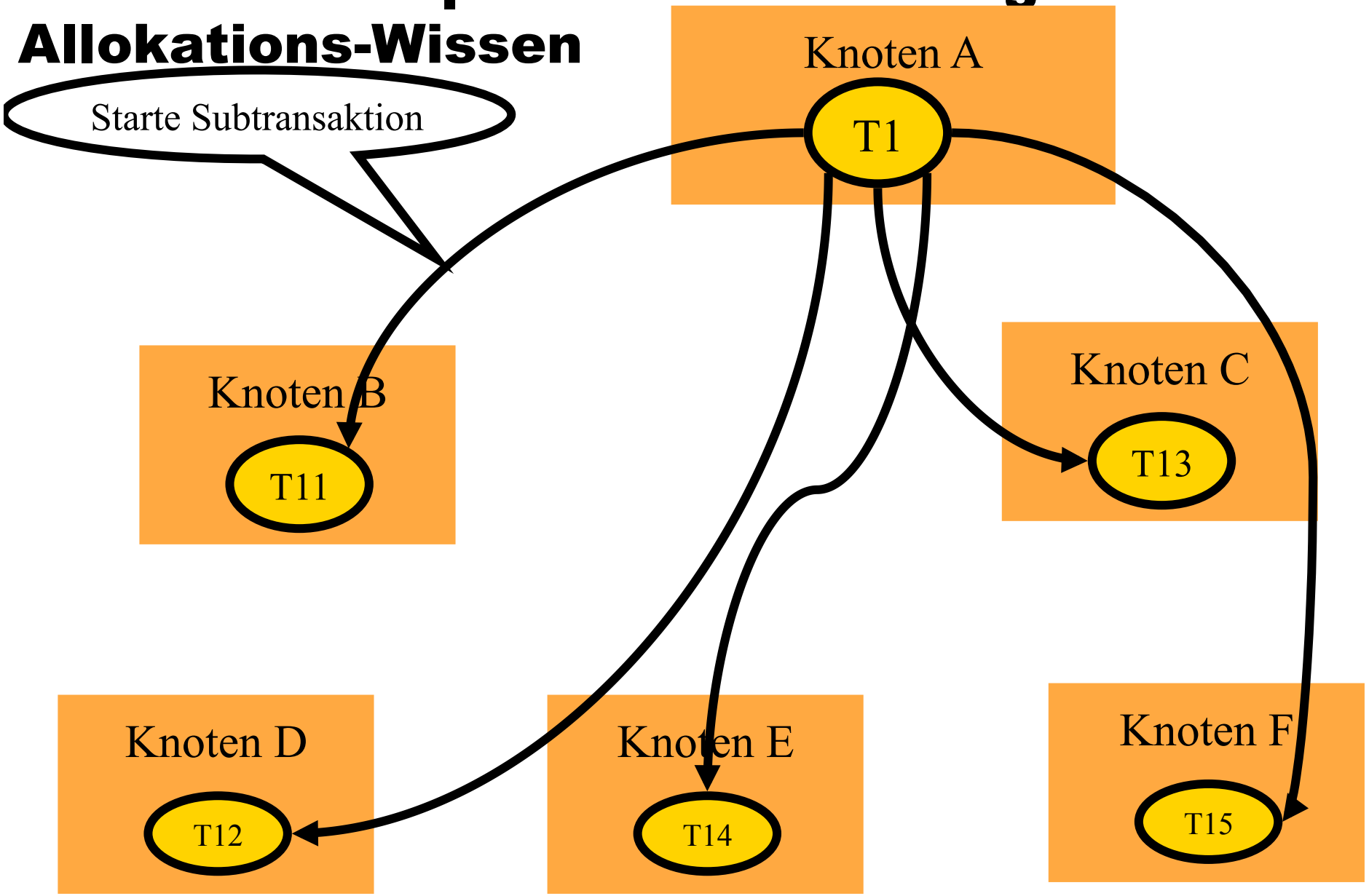
Globale (Verteilte) Transaktionen



Globale (Verteilte onen

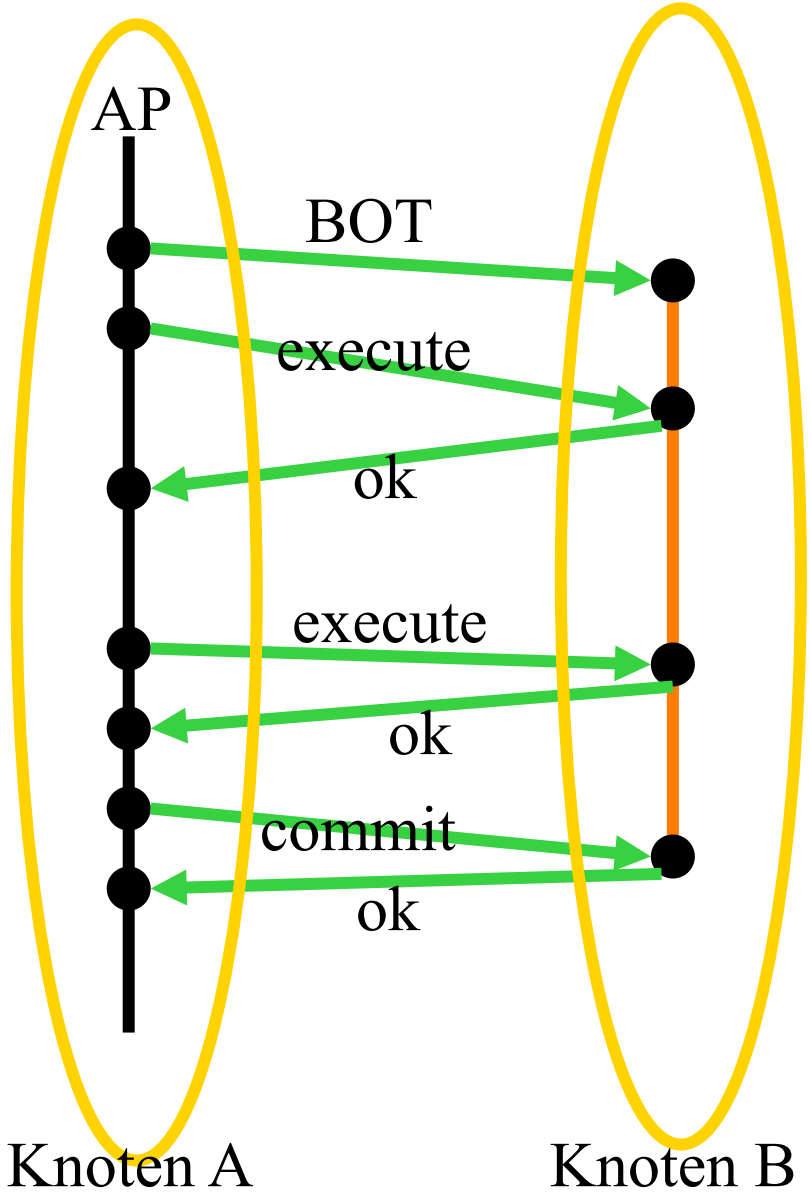


Flache TA-Graphen bei vollständigem Allokations-Wissen

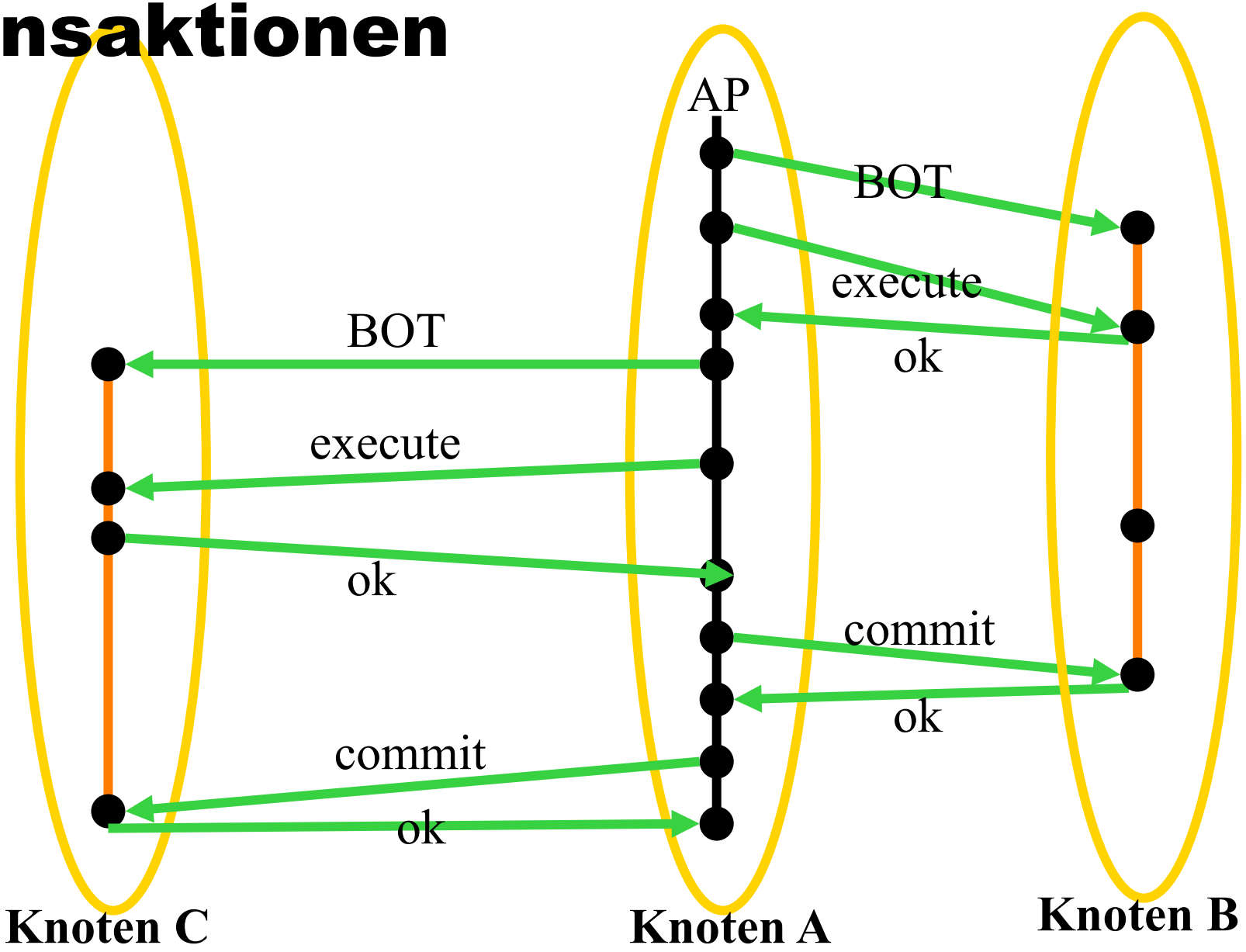


Entfernte Ausführung einer Transaktion

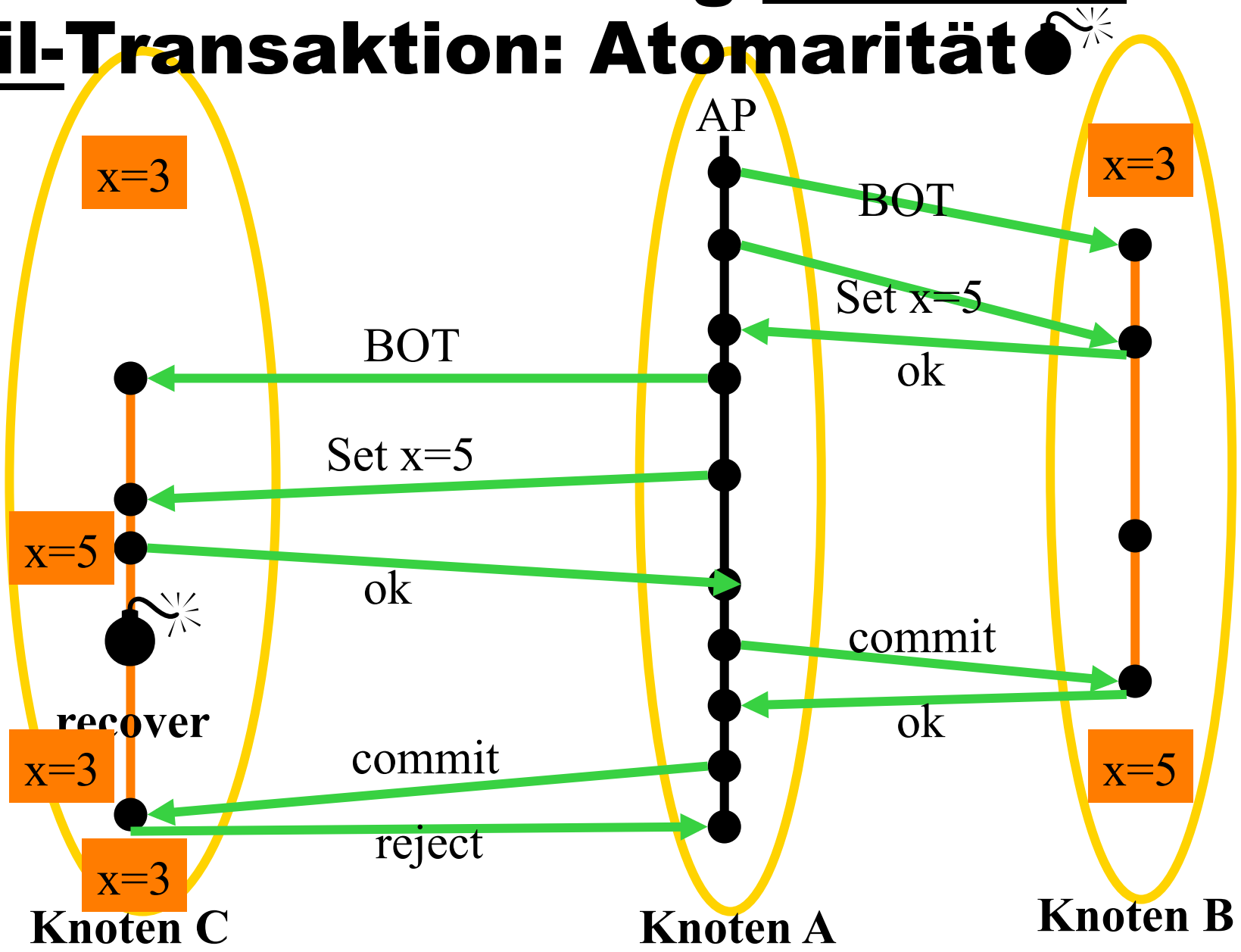
- Voraussetzung: Entfernter Knoten unterstützt Transaktionskonzept (BOT, commit)
- Entfernter Knoten wird im AP (Anwendungsprogramm) explizit spezifiziert
- Zugriff wird bspw. über ODBC oder JDBC abgewickelt
- keine systemseitige Fortpflanzung der entfernten TA auf andere Knoten



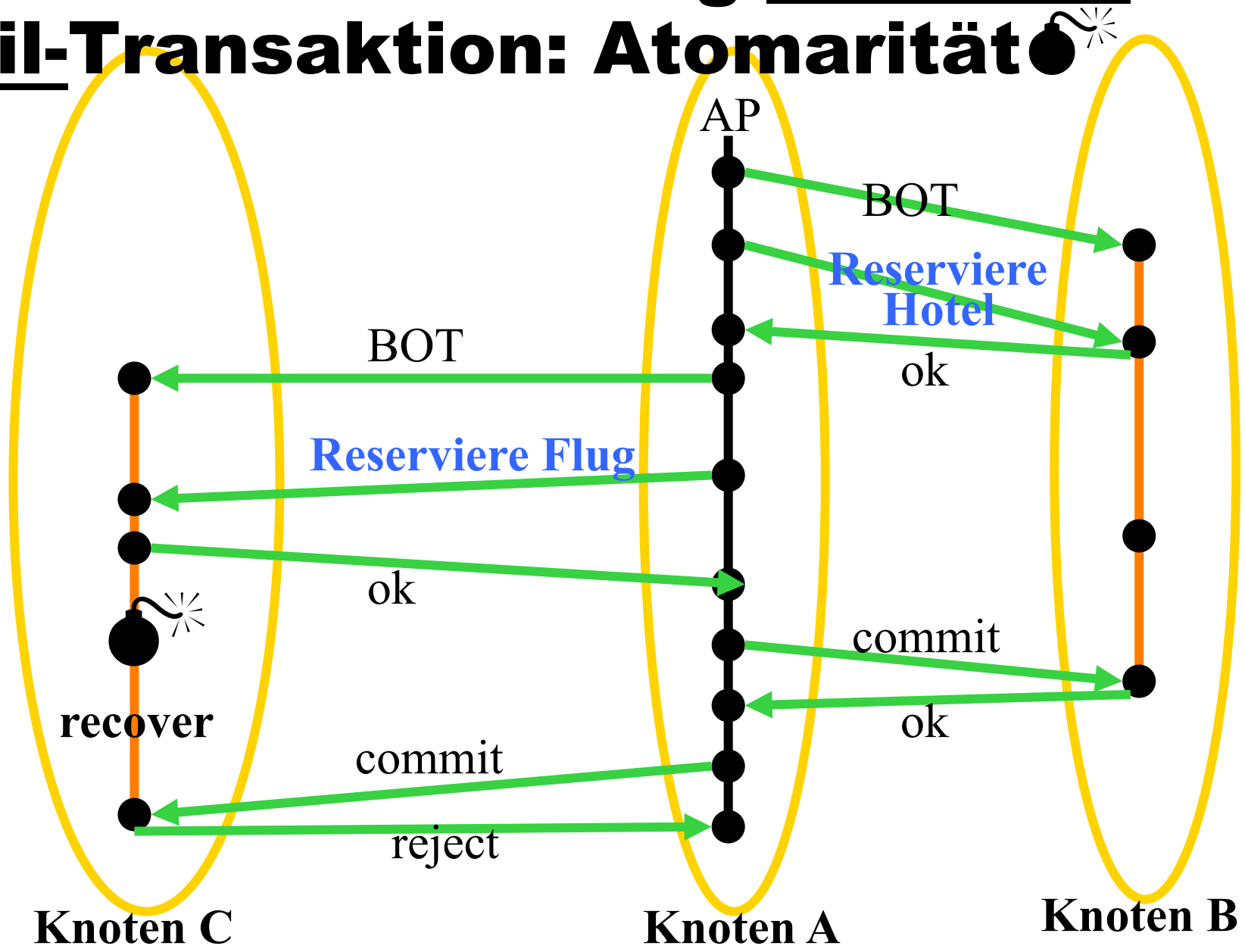
Entfernte Ausführung zweier Transaktionen



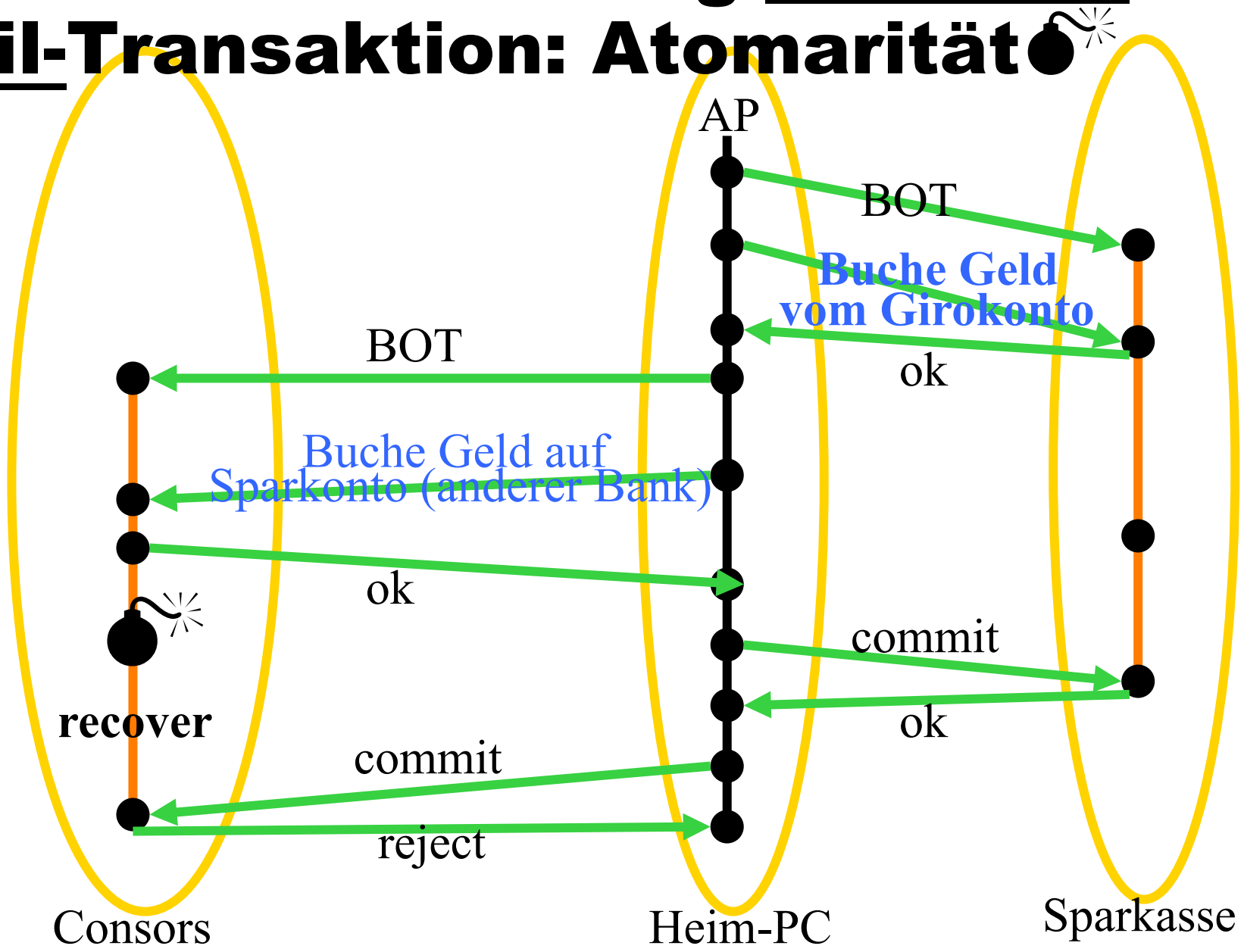
Entfernte Ausführung mehrerer Teil-Transaktion: Atomarität



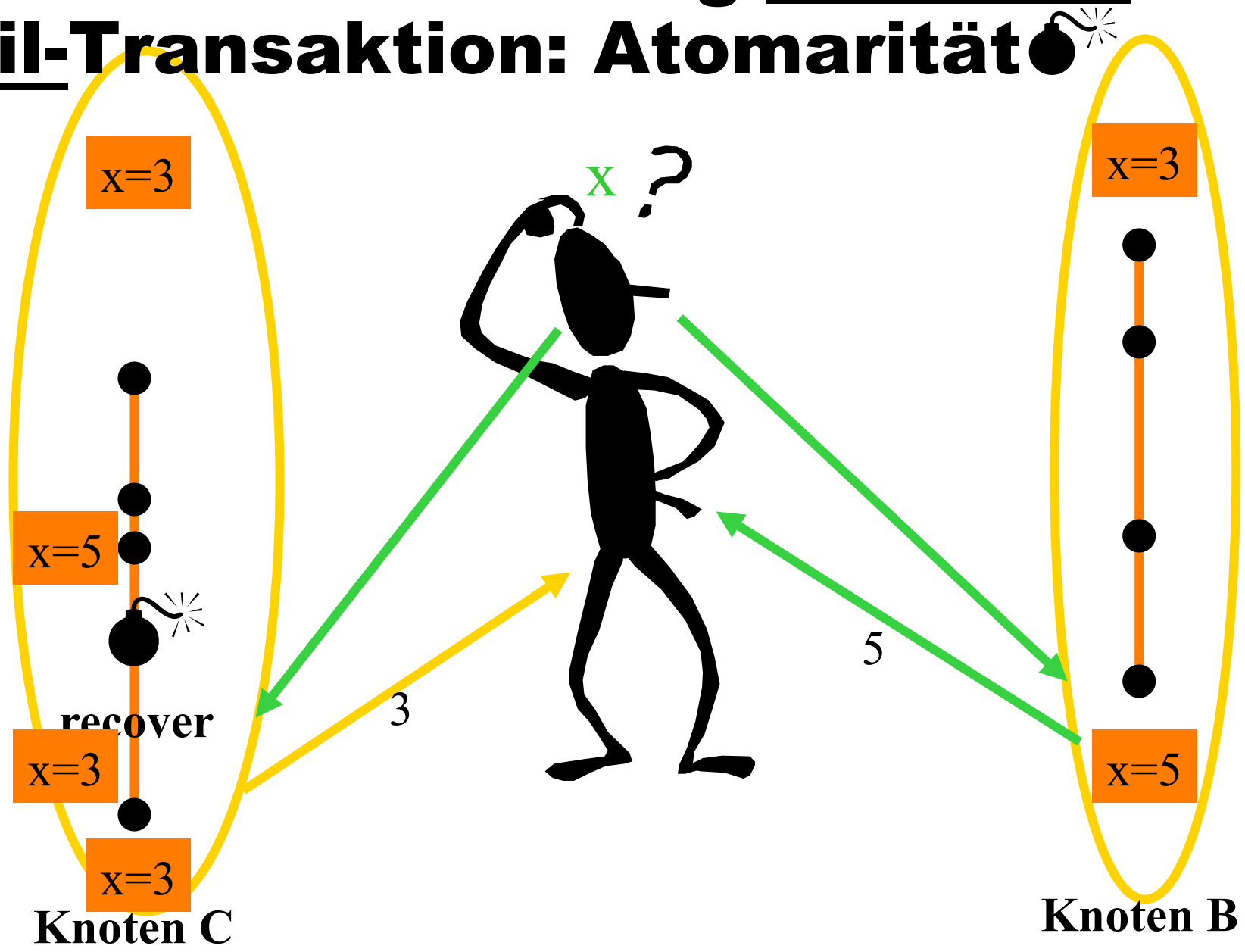
Entfernte Ausführung mehrerer Teil-Transaktion: Atomarität



Entfernte Ausführung mehrerer Teil-Transaktion: Atomarität



Entfernte Ausführung mehrerer Teil-Transaktion: Atomarität



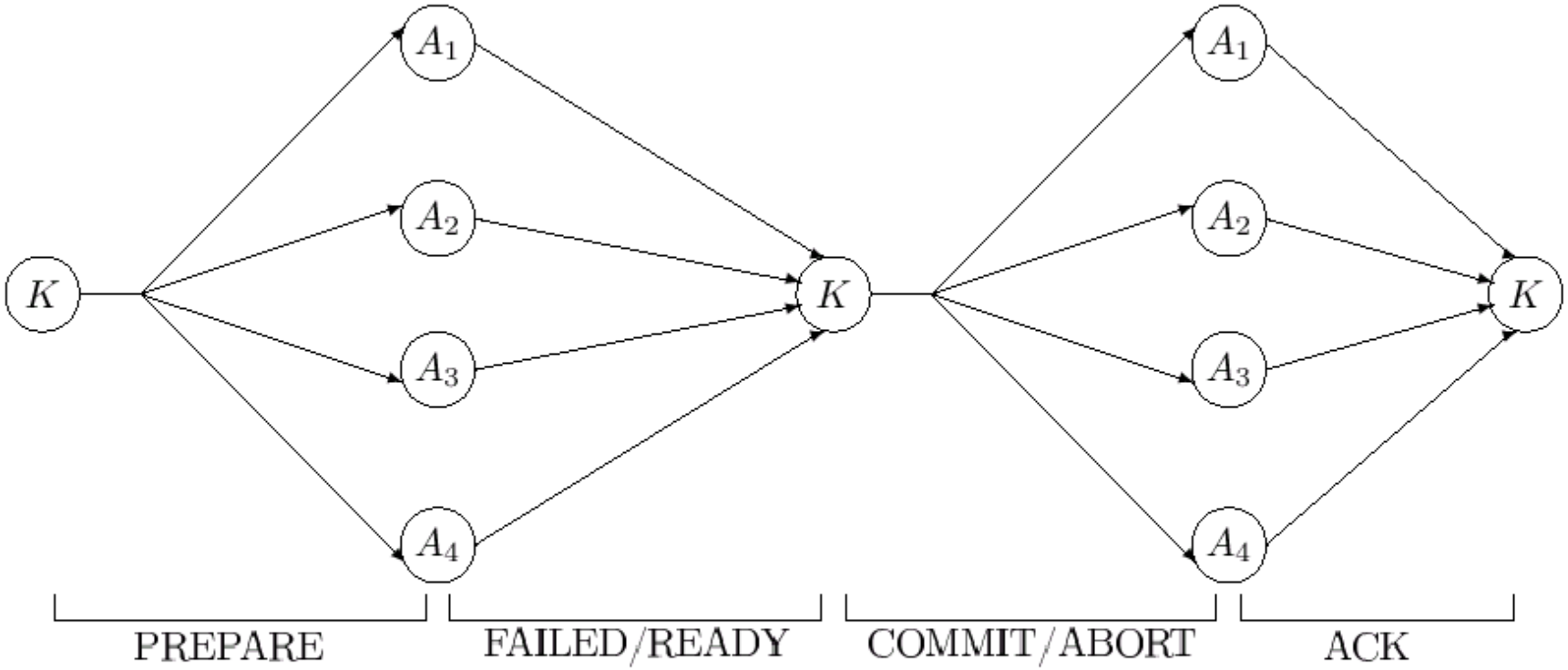
Atomarität durch Zwei-Phasen-Commit (2PC)

- 1. Phase: Abstimmung aller beteiligten Knoten
  - prepare to committ
    - nein = Veto
    - ja: dann muss ich es auch können, komme was wolle
- 2. Phase: Entscheidung durch den Koordinator
  - commit: nur möglich wenn alle mit ja geantwortet haben
  - Mitteilung an alle
- Nicht zu verwechseln mit Zwei-Phasen-Sperrprotokoll (2PL)

Recovery in VDBMS

- *Redo*: Wenn eine Station nach einem Fehler wiederanläuft, müssen alle Änderungen einmal abgeschlossener Transaktionen – seien sie lokal auf dieser Station oder global über mehrere Stationen ausgeführt worden – auf den an dieser Station abgelegten Daten wiederhergestellt werden.
- *Undo*: Die Änderungen noch nicht abgeschlossener lokaler und globaler Transaktionen müssen auf den an der abgestürzten Station vorliegenden Daten rückgängig gemacht werden.

Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

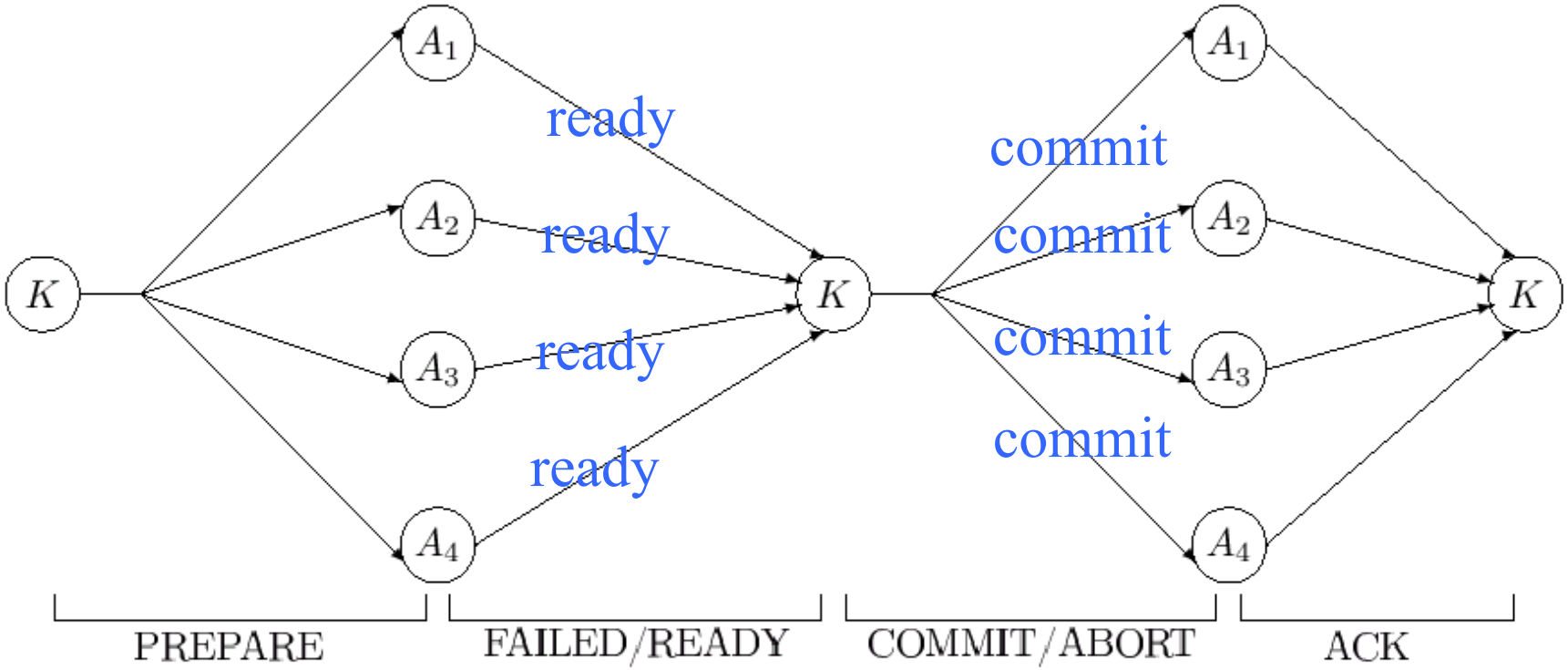


1. Phase

2. Phase

Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

Alles wird gut

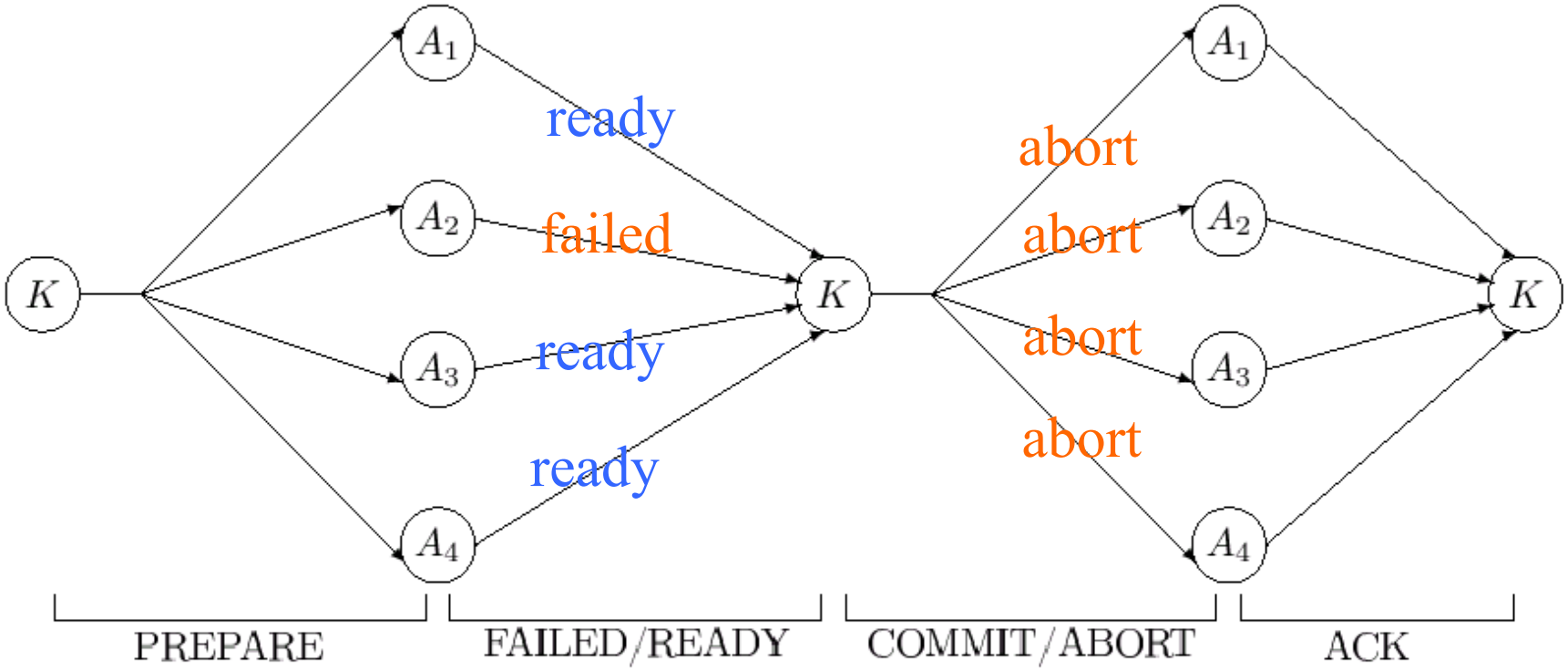


1. Phase

2. Phase

Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

„Nix“ wird gut: einer kann/will nicht

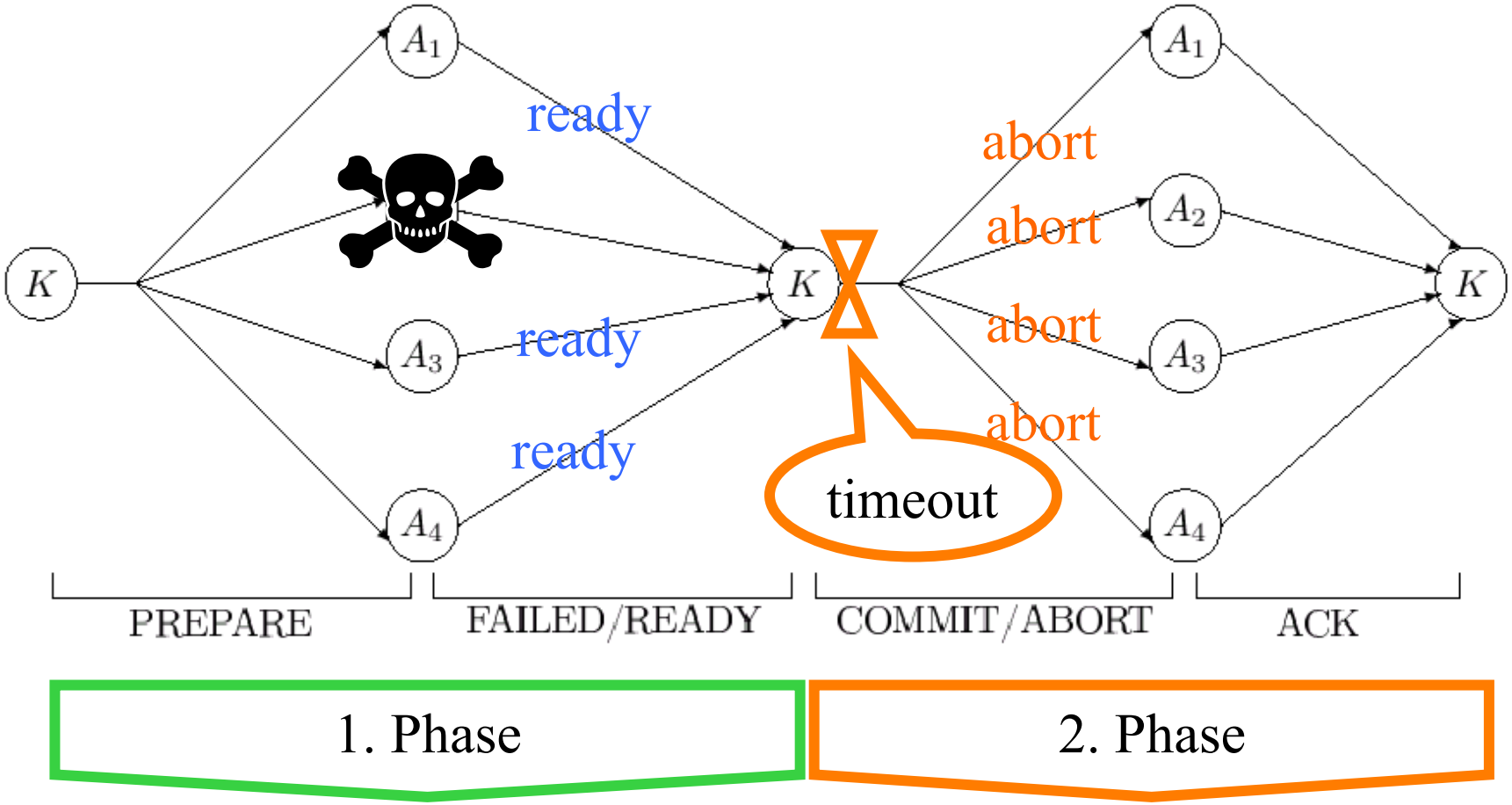


1. Phase

2. Phase

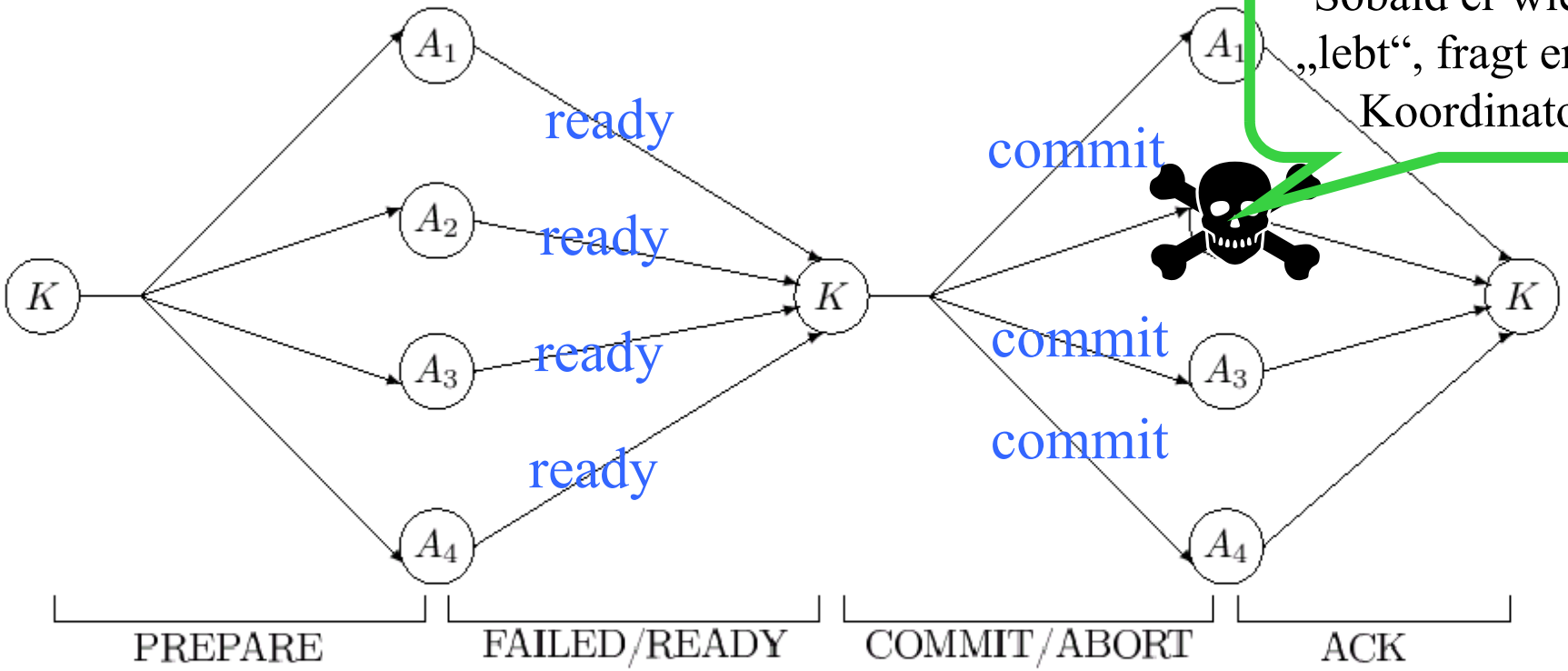
Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

„Nix“ wird gut: einer antwortet nicht



Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

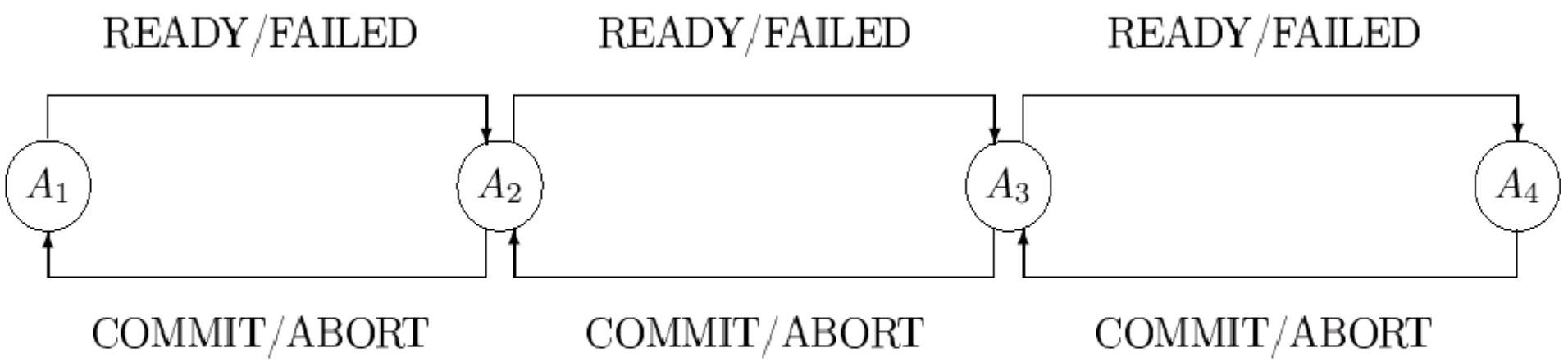
Alles wird gut, aber einer „crashed“ zwischendurch



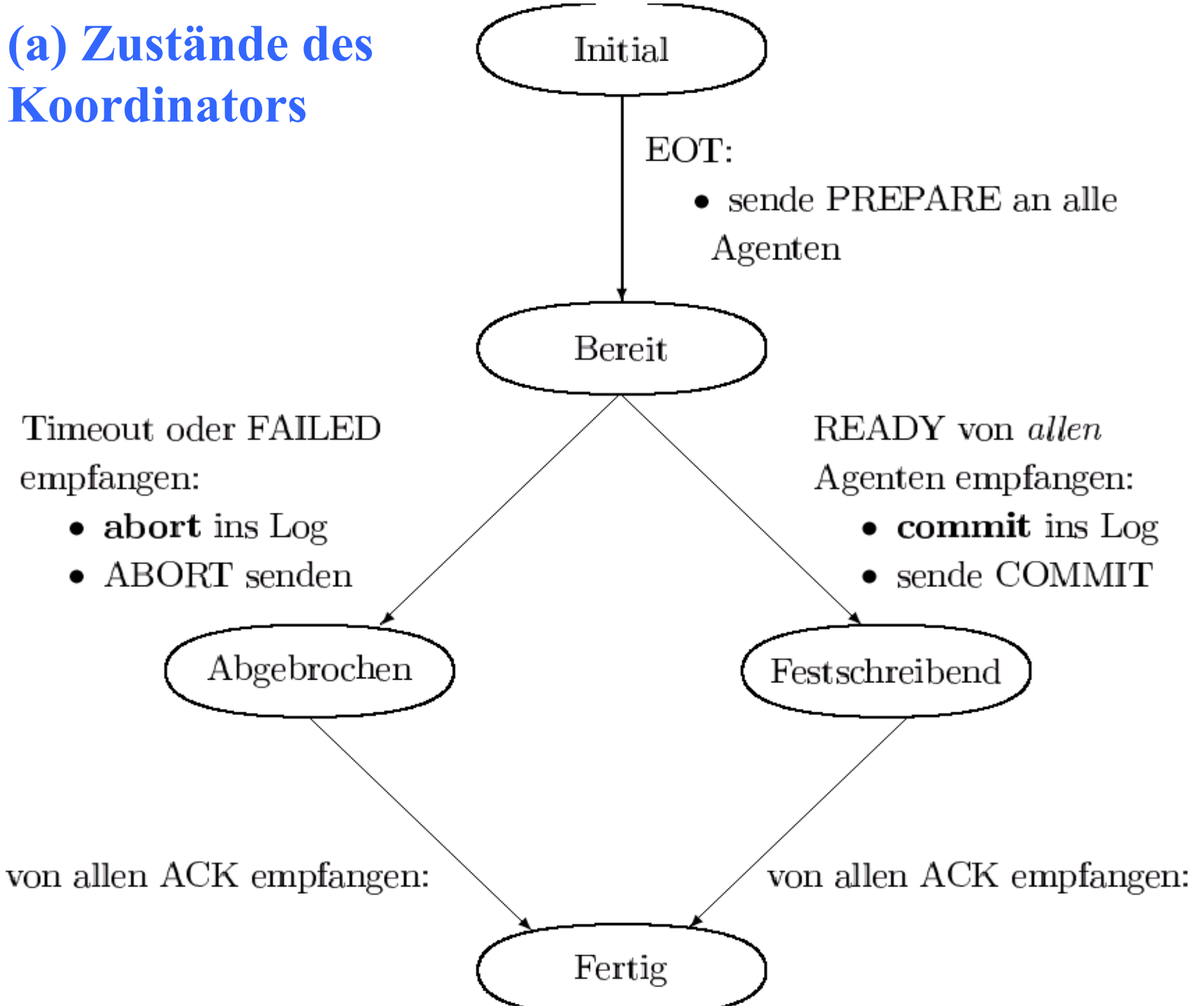
1. Phase

2. Phase

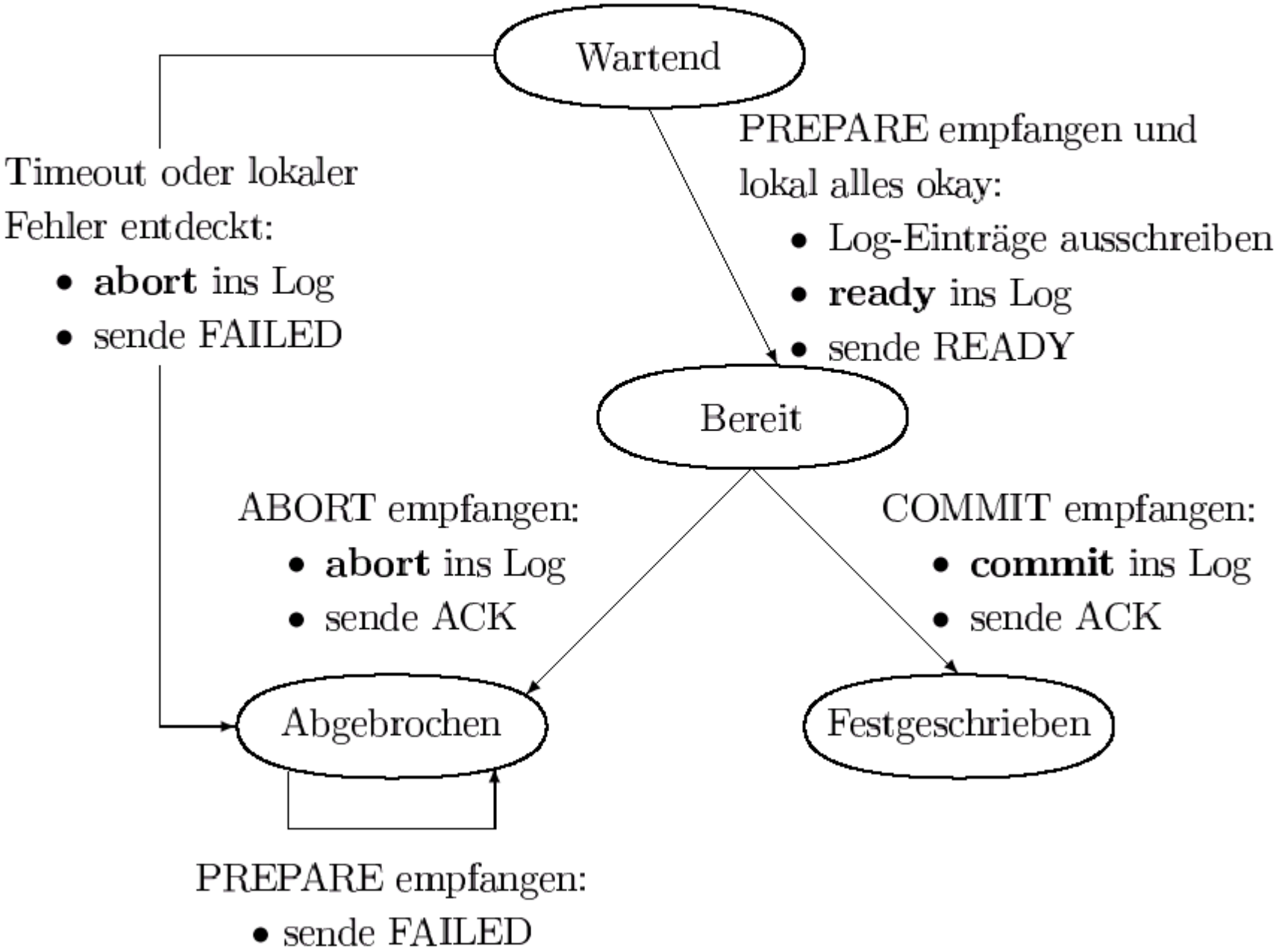
Lineare Organisationsform bei 2PC-Protokoll



(a) Zustände des Koordinators



(b) Agent


Wartend
Timeout oder lokaler Fehler entdeckt:
• **abort** ins Log
• sende FAILED
PREPARE empfangen und lokal alles okay:
• Log-Einträge ausschreiben
• **ready** ins Log
• sende READY
Bereit
ABORT empfangen:
• **abort** ins Log
• sende ACK
COMMIT empfangen:
• **commit** ins Log
• sende ACK
Abgebrochen
Festgeschrieben
PREPARE empfangen:
• sende FAILED

Analogie: 2PC == Trauungszeremonie

- Prepare to commit
  - alle antworten
  - ein nein führt zum Scheitern
  - wer mit ja (ready) stimmt, hat sich verpflichtet, die TA zu vollziehen
- Commit/Abort-Entscheidung
  - commit nur wenn alle ja gesagt haben
  - abort sonst

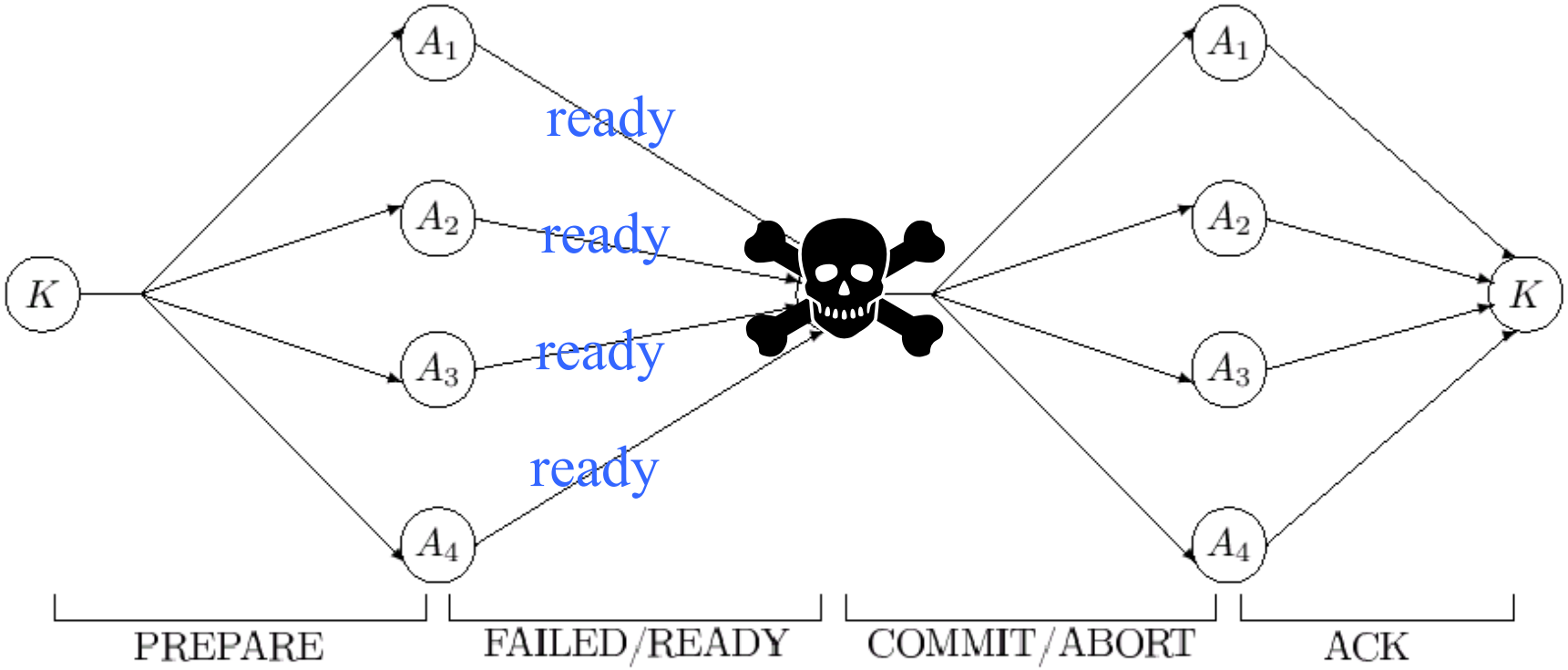
- Abstimmung
  - Standesbeamter = Koordinator
  - Bräutigam: willst Du ...
    - Nein = Veto
    - Ja: kann nicht mehr zurückgezogen werden
  - Braut: willst Du ...
  - Alle anderen: Hat jemand was dagegen ...
- Vollzug der Trauung
  - Protokollierung in den Standesamt-Archiven

Blockieren des 2PC-Protokolls

- Das 2PC kann immer noch in einen Zustand gelangen, wo niemand weiß, welche Entscheidung getroffen wurde
- Dies passiert, wenn der Koordinator „stirbt“, bevor er die Entscheidung verkündet hat
- Die Agenten müssen warten bis der Koordinator wieder „lebt“
  - sie müssen alle Sperren halten
- In einigen Fällen können die Agenten eigenmächtig entscheiden
  - ein Agent hat noch nicht geantwortet oder er hat mit **failed** gestimmt
    - er kann jetzt abort entscheiden
  - einer der Agenten hat das Ergebnis schon erfahren und teilt es den anderen mit
- Erweiterung zum 3PC-Protokoll, um das Blockieren ganz zu vermeiden

Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

Keiner weiß, was los ist ⇒ Warten/Blockieren ⇒ alle Sperren halten

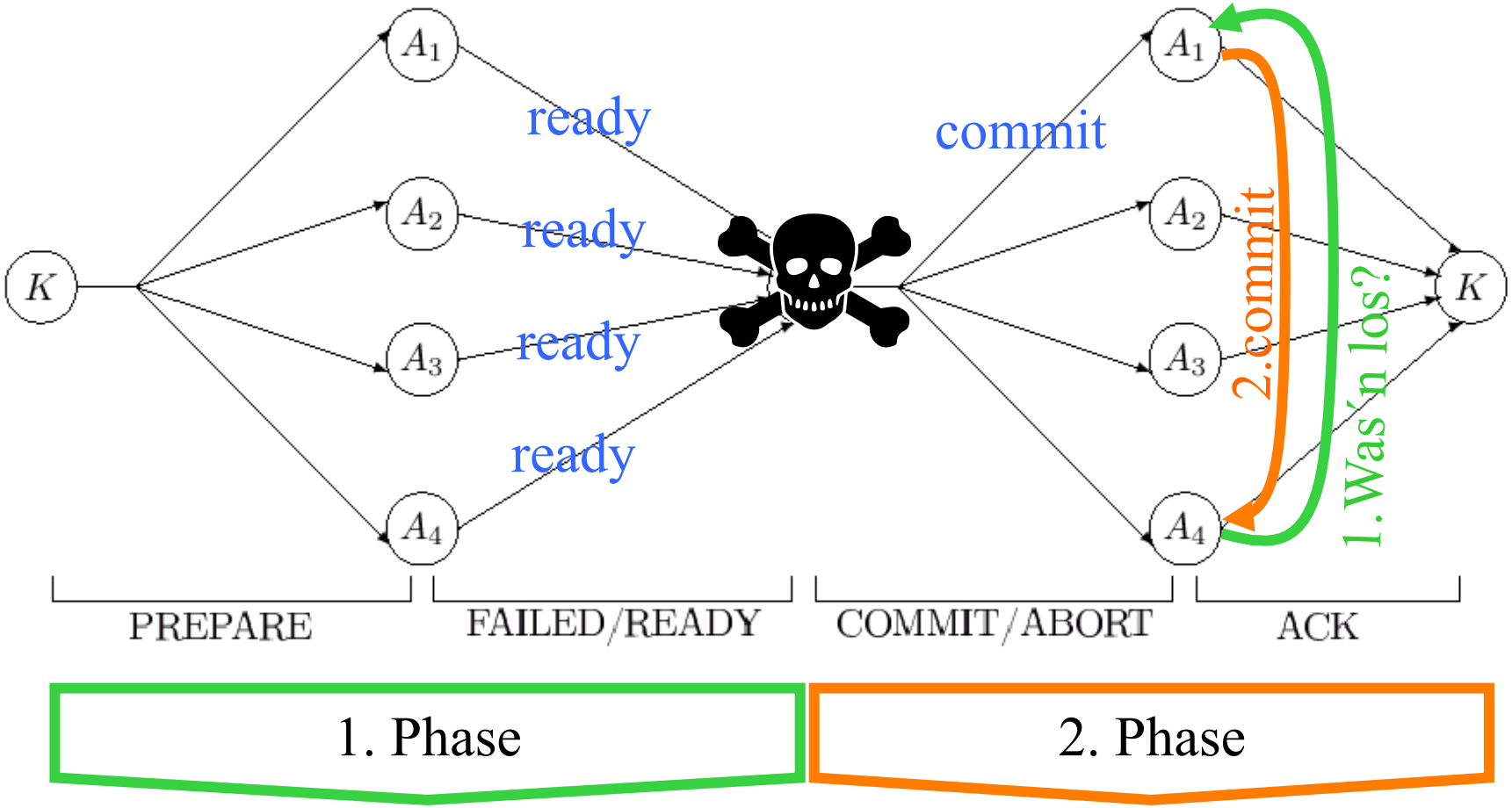


1. Phase

2. Phase

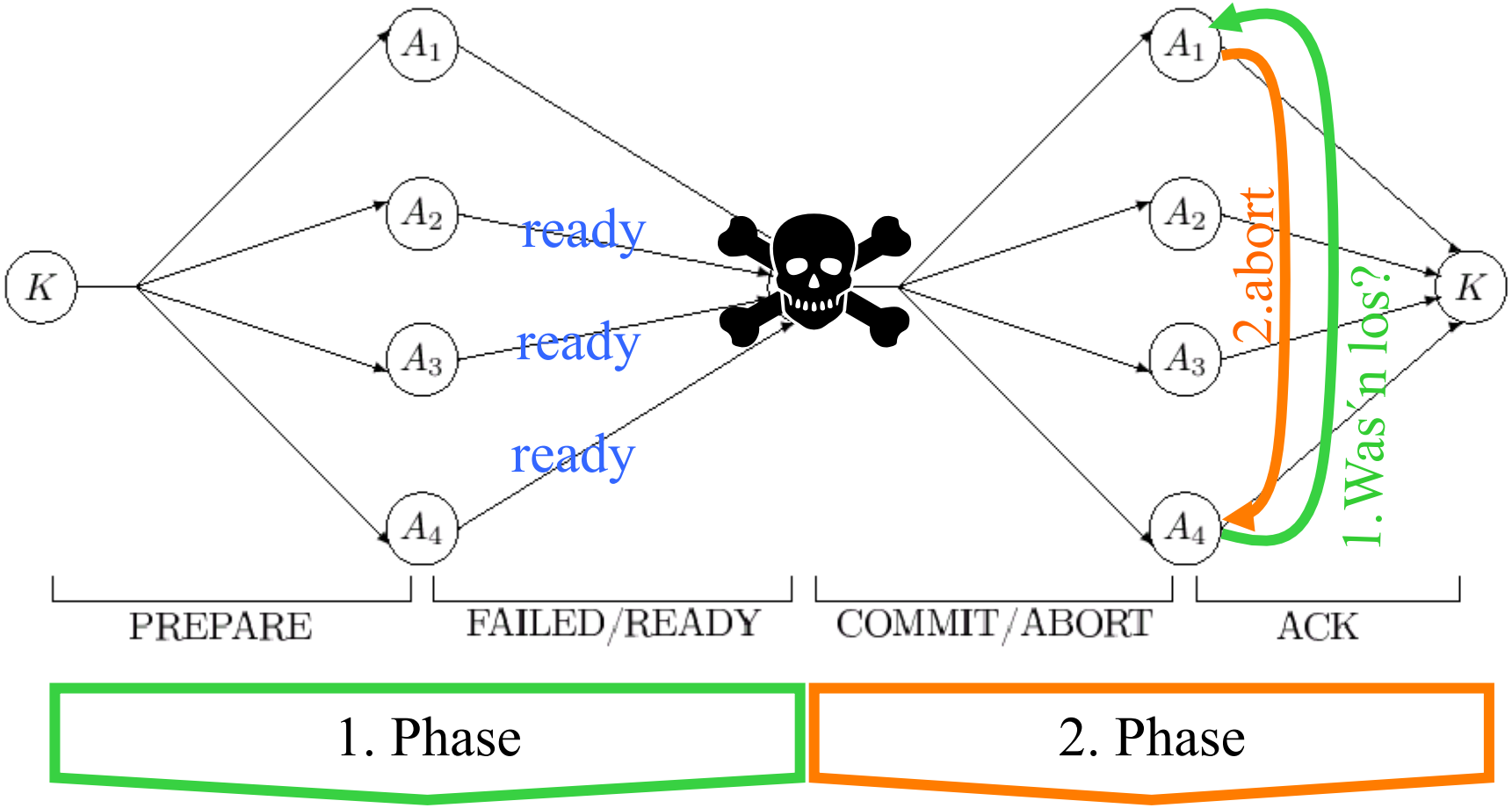
Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

einer weiß, was los ist ⇒ Entscheidung steht fest ⇒ allen anderen mitteilen



Nachrichtenaustausch zwischen *Koordinator* und *Agenten* beim 2PC-Protokoll

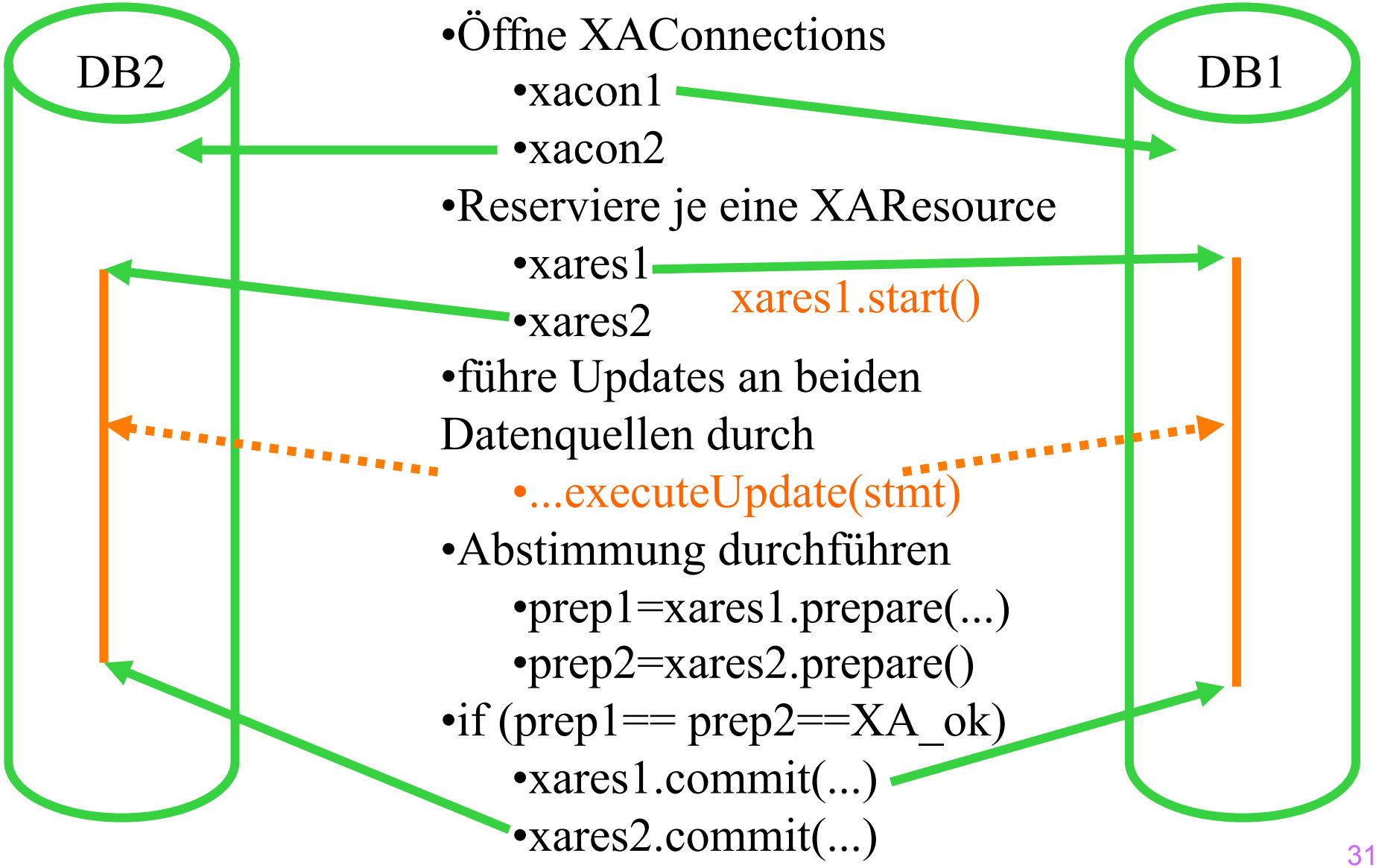
einer hat´s noch in der Hand⇒fällt abort Entscheidung⇒anderen mitteilen



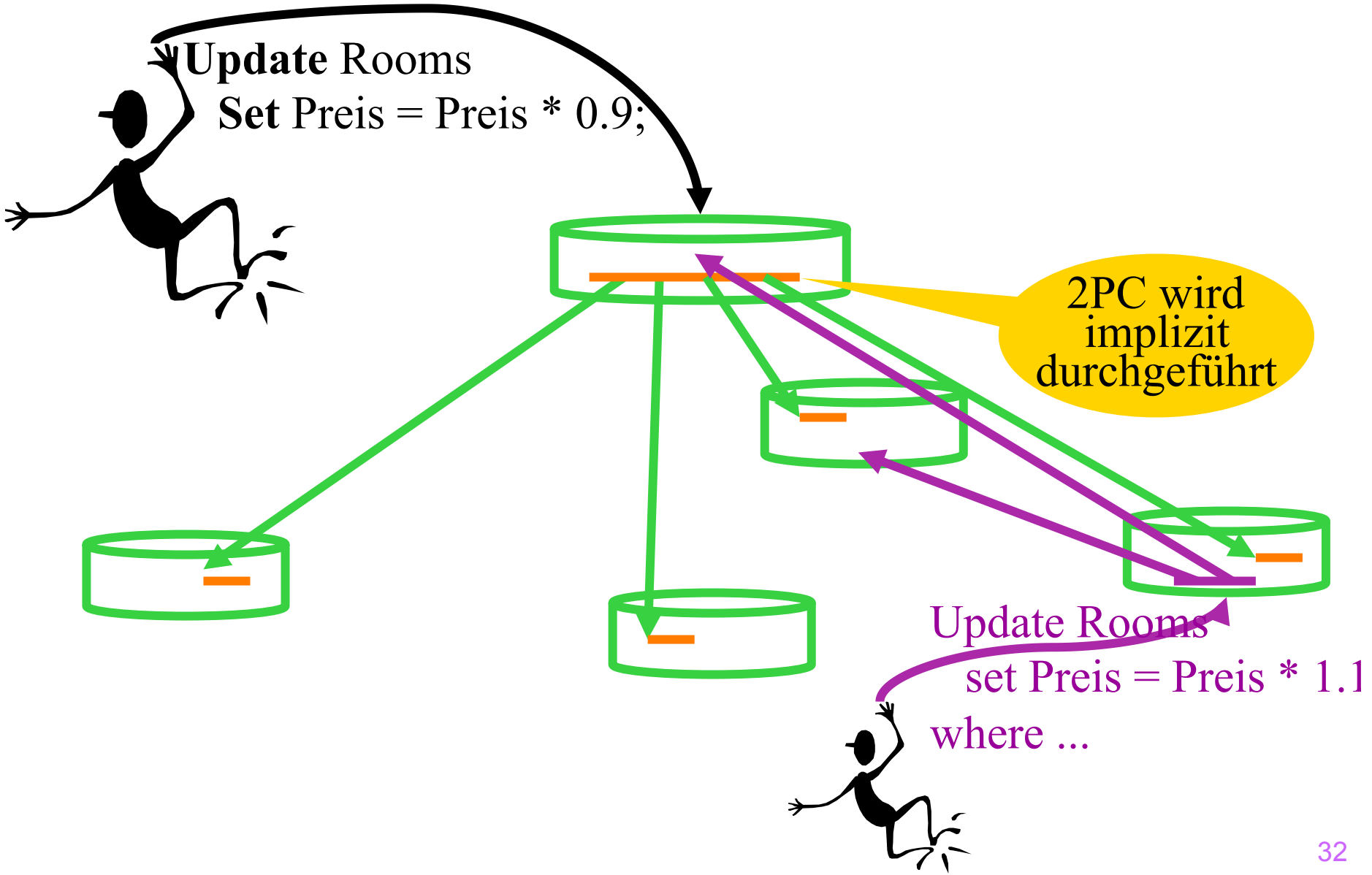
Atomare Ausführung zweier Transaktionen mit Java/JDBC

- 2PC ist als XA-Standard der X/Open-Spezifikation standardisiert
- XA-Ressourcen sind
  - DBMS (aller Hersteller)
  - Message Queue Server (z.B. von IBM)
  - Transaktionsmonitore
  - etc.
- Realisierung des 2PC im Java-Programm
  - Java-AP entspricht dem Koordinator
- Verbindung mit zwei unterschiedlichen Datenbanken
- Übungsaufgabe
- Literatur: G. Saake und K. U. Sattler: Datenbanken und Java. Dpunkt Verlag, 2000.

Vorgehensweise im Java-Programm (via Java Transaction Service)



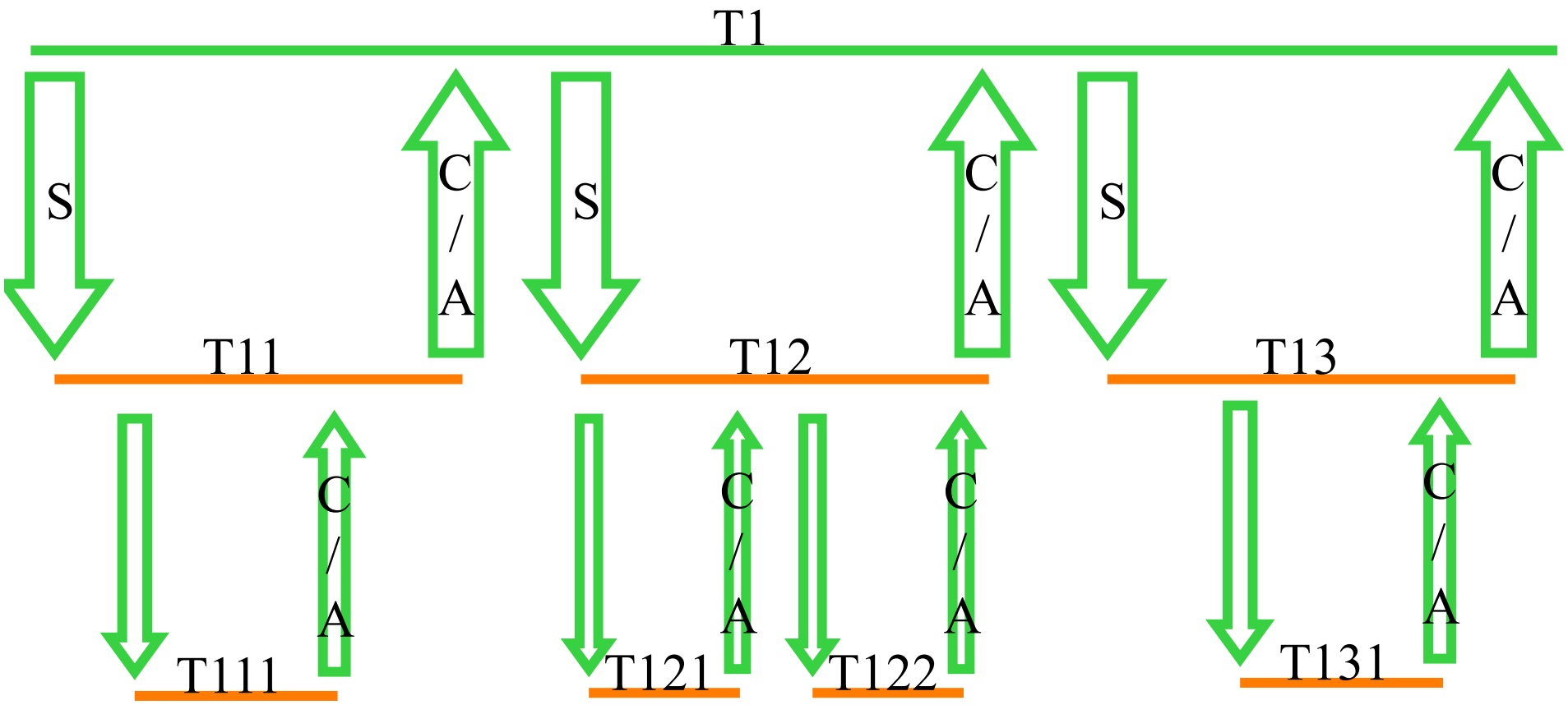
Atomare Ausführung verteilter Transaktionen in VDBMS



Geschachtelte (Nested) Transaktionen

- Baumartige Transaktionsstruktur
  - Vater kann Subtransaktionen starten
  - Kinder können ihrerseits wieder Sub-TAs starten
- Sub-TA erbt automatisch alle Sperren des Vaters
- Vater-TA erbt alle Sperren der Kinder, nach deren commit/abort
- also sind deren Ergebnisse (dirty data) nur für Vater-TA sichtbar
  - **geschlossene geschachtelte Transaktionen**
- Top-Level TA befolgt ACID-Prinzip
  - verhält sich wie eine große globale TA
  - gibt alle Sperren auf einmal frei

Beispiel einer geschachtelten Transaktion



Offen geschachtelte Transaktionen

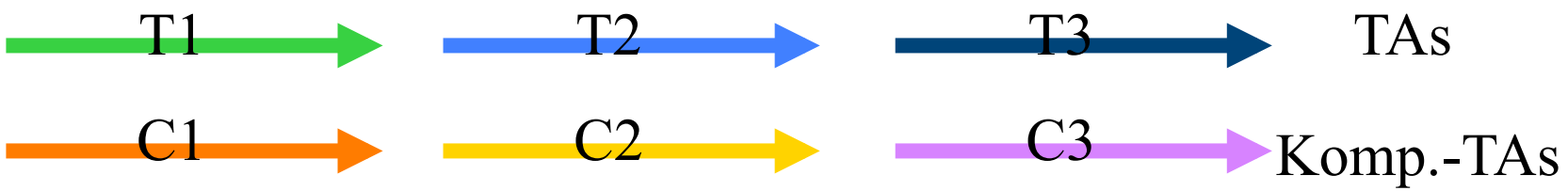
- Subtransaktionen machen ihre Änderungen bei ihrem commit nach außen sichtbar
- Serialisierbarkeit als Korrektheitskriterium für Gesamttransaktion nicht mehr anwendbar
- einfaches Rollback abgeschlossener Teiltransaktionen bei Abort der Gesamt-TA nicht mehr möglich
  - kaskadierendes Rollback der Leser dieser Daten notwendig
- Lösung: Kompensation der Änderungen abgeschlossener Teiltransaktionen
  - Saga/Contract-Modell

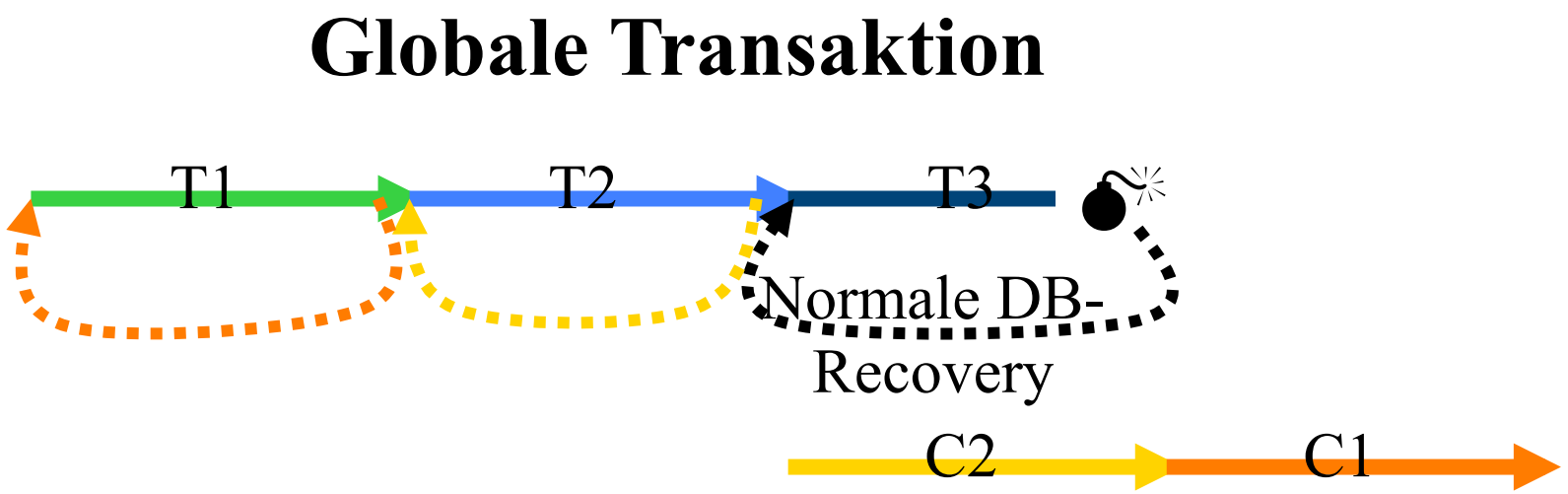
Sehr lange verteilte Transaktionen: Sagas/Contracts

- Sagas: Garcia-Molina et al. (Stanford)
- Contracts: Reuter und Wächter (Stuttgart)
- Eine Saga/ein Contract besteht aus mehreren Datenbank-Transaktionen
- Um eine/n Saga/Contract zuruecksetzen zu können, benötogt man spezielle Kompensations-TAs

- Bsp: Flugbuchung, Buchungsstornierung

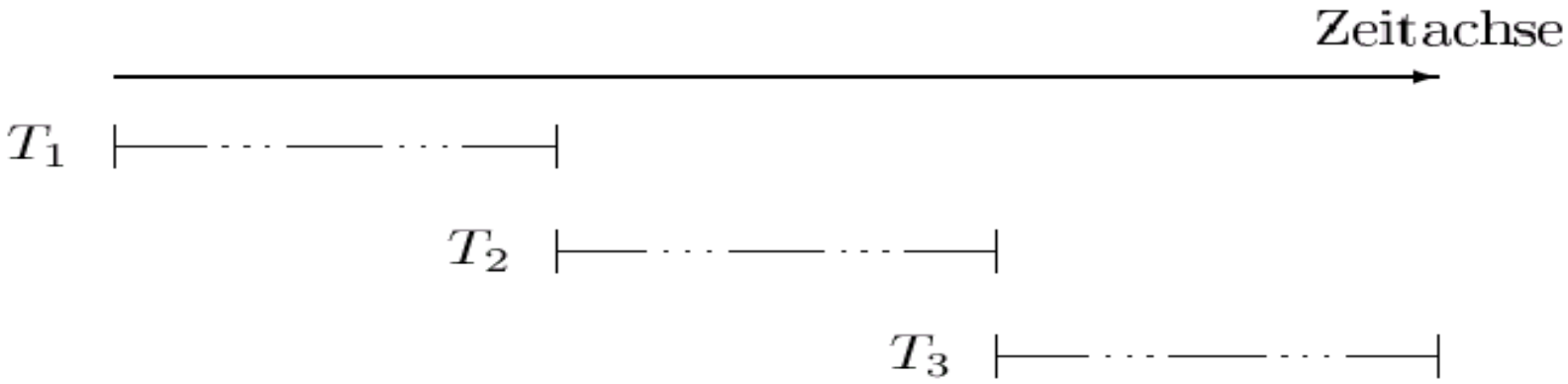
Rücksetzen (Undo) einer/s Saga/ Contract



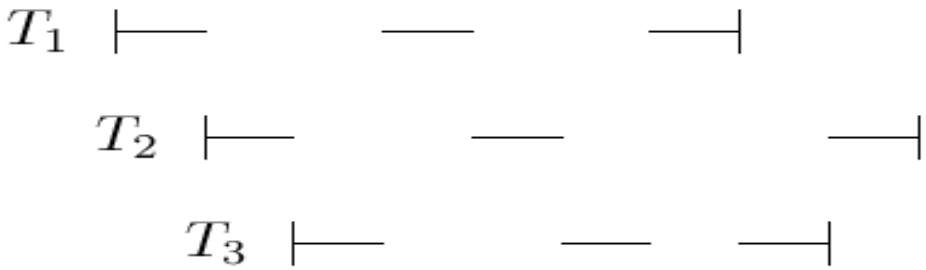


Synchronisation paralleler Transaktionen (Mehrbenutzer-S.)

(a) im Einbenutzerbetrieb und

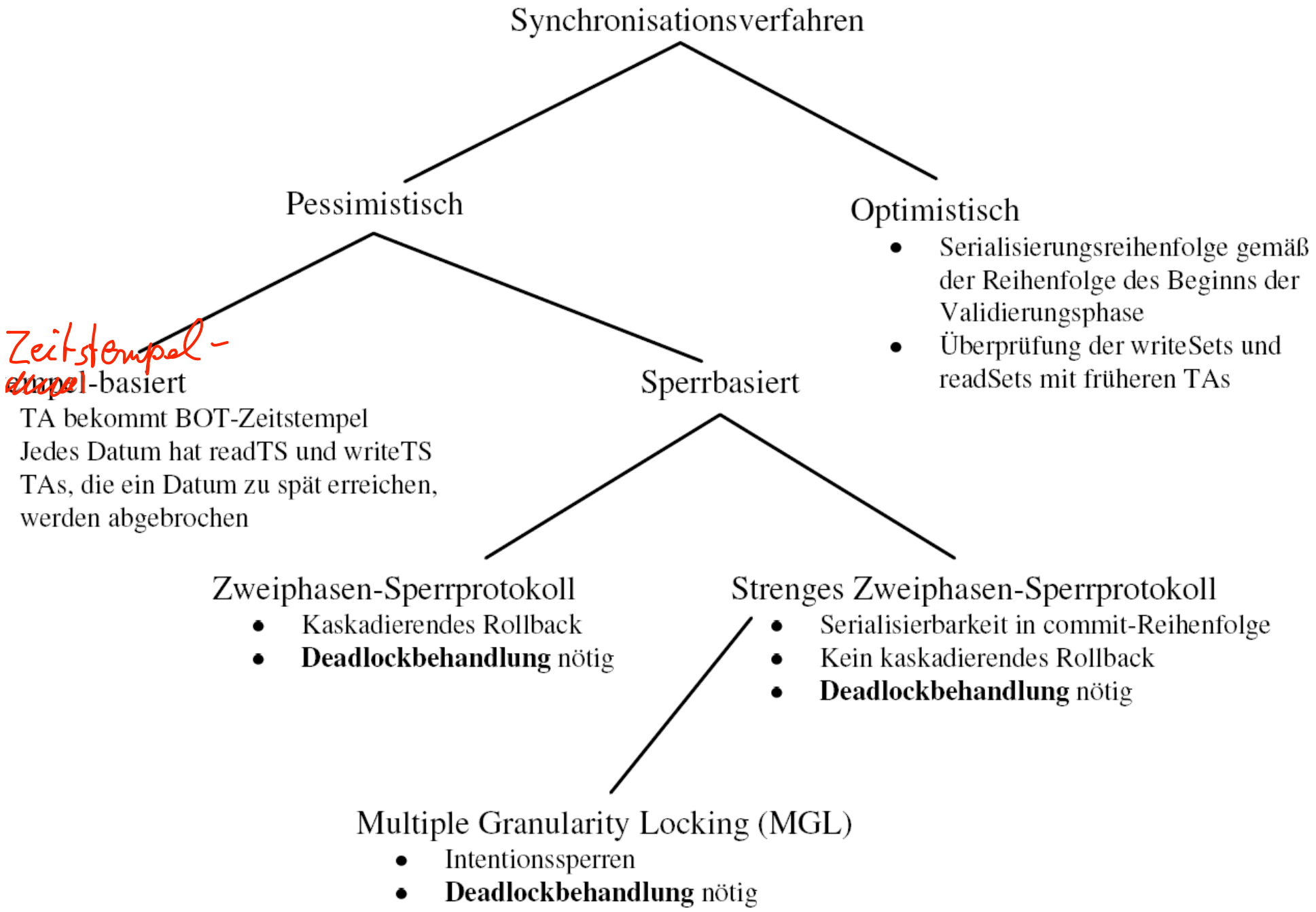


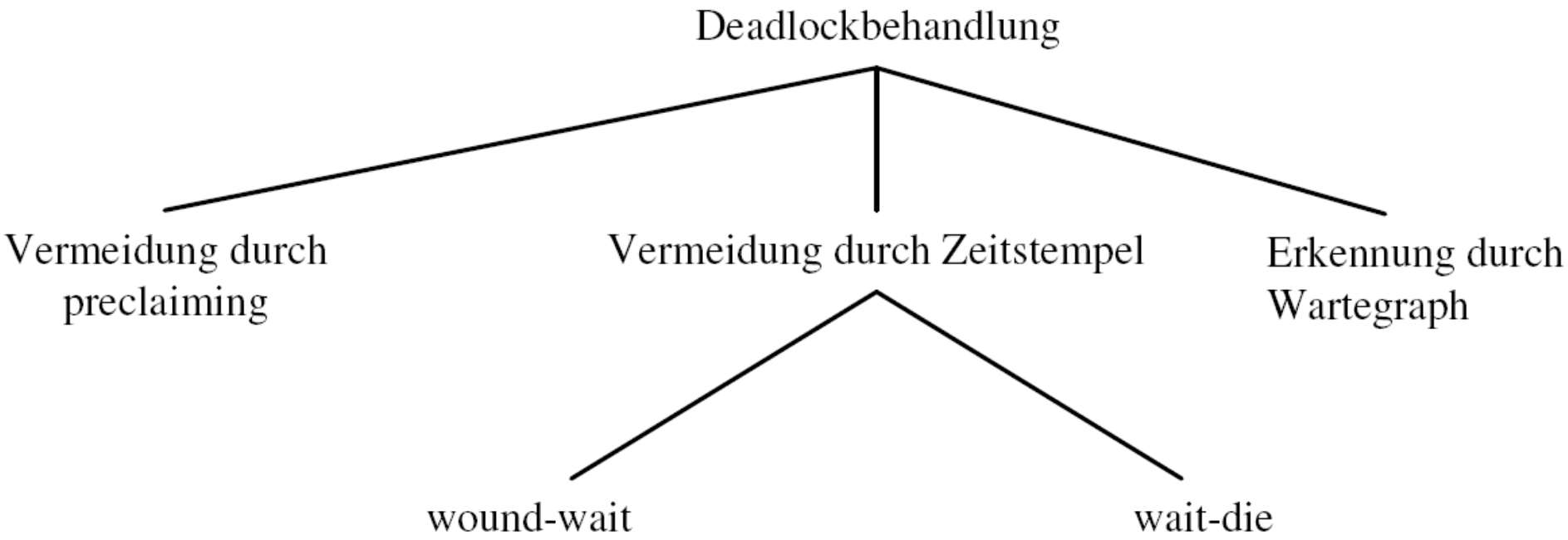
(b) im (verzahnten) Mehrbenutzerbetrieb (gestrichelte Linien repräsentieren Wartezeiten)



Korrektheitskriterium: Serialisierbarkeit

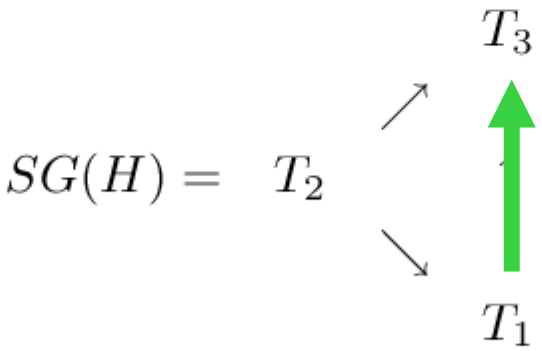
- Die parallele Ausführung ist äquivalent (irgend-)einer seriellen Ausführung der Transaktionen
- In der Historie (~Schedule) werden die Elementaroperationen der Transaktionen partiell geordnet
  - Konfliktoperationen müssen geordnet sein:
    - $w_i(A)$? $w_j(A)$ (Transaktion Ti schreibt A bevor oder nachdem Tj das Datum A schreibt)
    - $r_i(A)$? $w_j(A)$ (Ti liest A bevor oder nachdem Tj das Datum A schreibt)
    - Entweder Transaktion Ti ist erfolgreich (commit = c_i) oderwird zurückgesetzt (abort = a_i)
- Der Serialisierbarkeitsgraph SG ist eine komprimierte Darstellung der Historie (nur noch Tas, keine Ops)
- Der SG ist zyklenfrei genau dann wenn die Historie serialisierbar ist
- Topologische Sortierung = äquivalente serielle Ausführungsreihenfolge


Synchronisationsverfahren
Pessimistisch
Optimistisch
• Serialisierungsreihenfolge gemäß der Reihenfolge des Beginns der Validierungsphase
• Überprüfung der writeSets und readSets mit früheren TAs
Zeitstempel-basiert
TA bekommt BOT-Zeitstempel
Jedes Datum hat readTS und writeTS
TAs, die ein Datum zu spät erreichen, werden abgebrochen
Sperrbasiert
Zweiphasen-Sperrprotokoll
• Kaskadierendes Rollback
• Deadlockbehandlung nötig
Strenges Zweiphasen-Sperrprotokoll
• Serialisierbarkeit in commit-Reihenfolge
• Kein kaskadierendes Rollback
• Deadlockbehandlung nötig
Multiple Granularity Locking (MGL)
• Intentionssperren
• Deadlockbehandlung nötig


Deadlockbehandlung
Vermeidung durch preclaiming
Vermeidung durch Zeitstempel
Erkennung durch Wartegraph
wound-wait
wait-die

Historie und zugehöriger Serialisierbarkeitsgraph

$$
\begin{array}{lccccccc}
 & r_1(A) & \rightarrow & w_1(A) & \rightarrow & w_1(B) & \rightarrow & c_1 \\
 & & & \uparrow & & \uparrow & & \\
H = & & & r_2(A) & \rightarrow & w_2(B) & \rightarrow & c_2 \\
 & & & \downarrow & & & & \\
 & r_3(A) & \rightarrow & w_3(A) & \rightarrow & c_3 & &
\end{array}
$$



- $w_1(A) \rightarrow r_3(A)$ der Historie H führt zur Kante $T_1 \rightarrow T_3$ des SG
- weitere Kanten analog

Zwei-Phasen-Sperrprotokoll

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muß vorher entsprechend gesperrt werden.

2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.

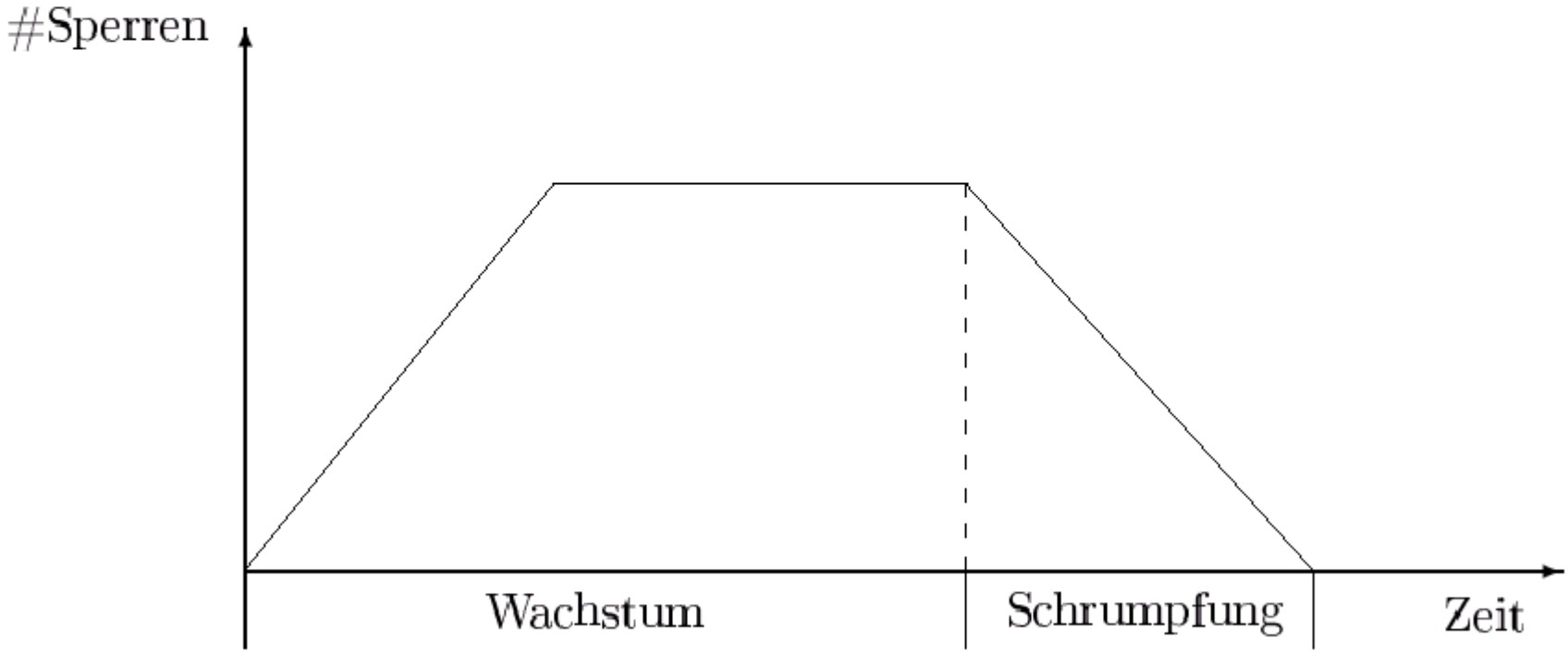
3. eine Transaktion muß die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.

4. Jede Transaktion durchläuft zwei Phasen:

   - Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
   - Eine *Schrumpfungsphase*, in der sie ihre bisher erworbenen Sperren freigibt, aber keine weiteren anfordern darf.

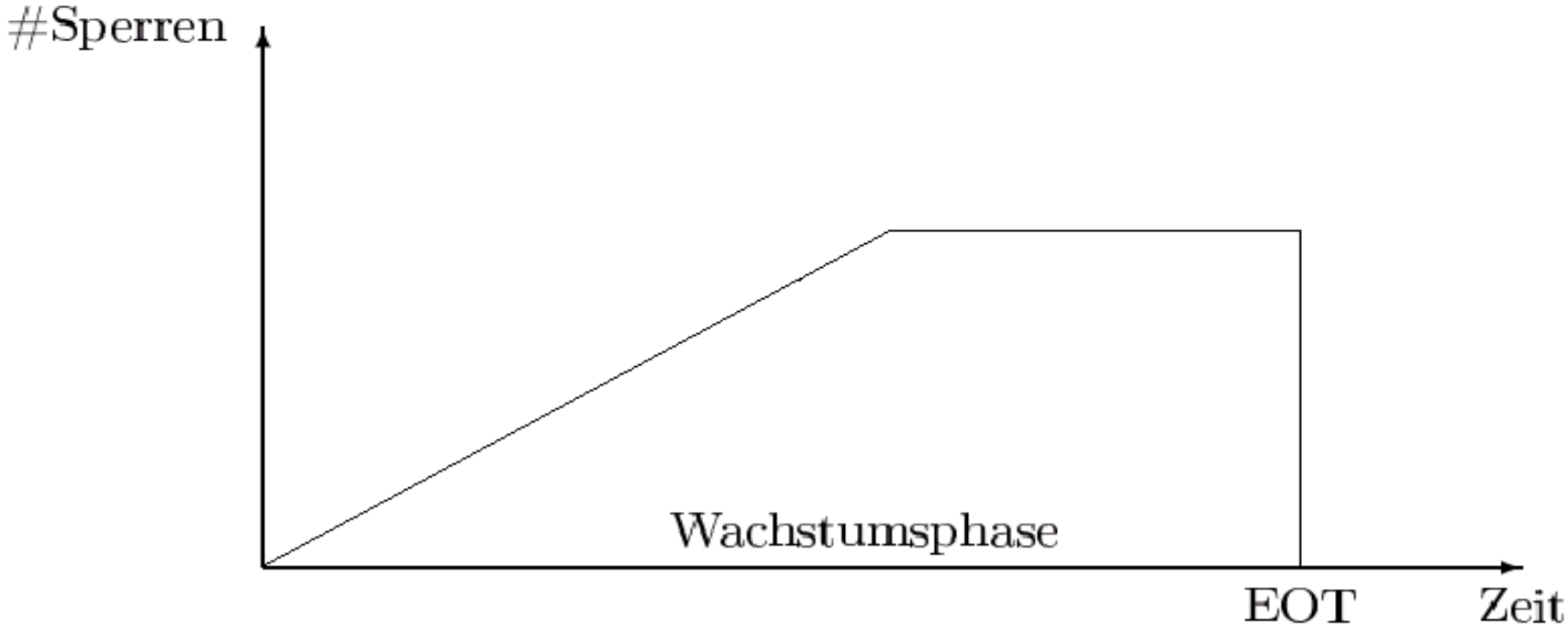
5. Bei EOT (Transaktionende) muß eine Transaktion alle ihre Sperren zurückgeben.

Zwei-Phasen-Sperrprotokoll

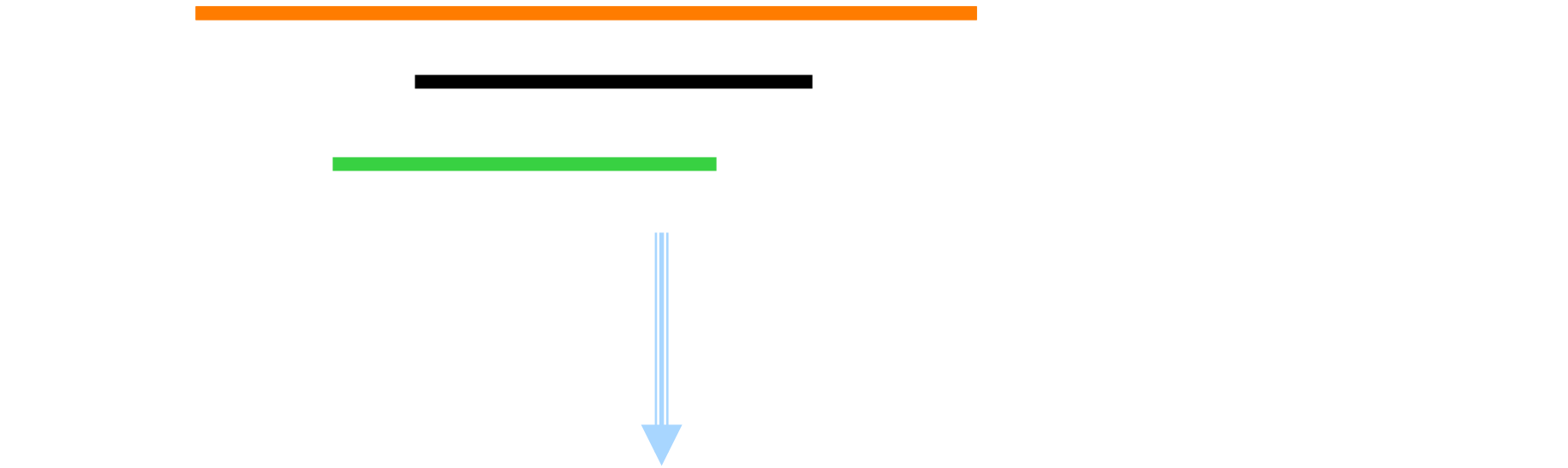


Strenges Zwei-Phasen-Sperrprotokoll

- 2PL schließt kaskadierendes Rücksetzen nicht aus
- Erweiterung zum *strengen* 2PL:
  - alle Sperren werden bis EOT gehalten
  - damit ist kaskadierendes Rücksetzen ausgeschlossen

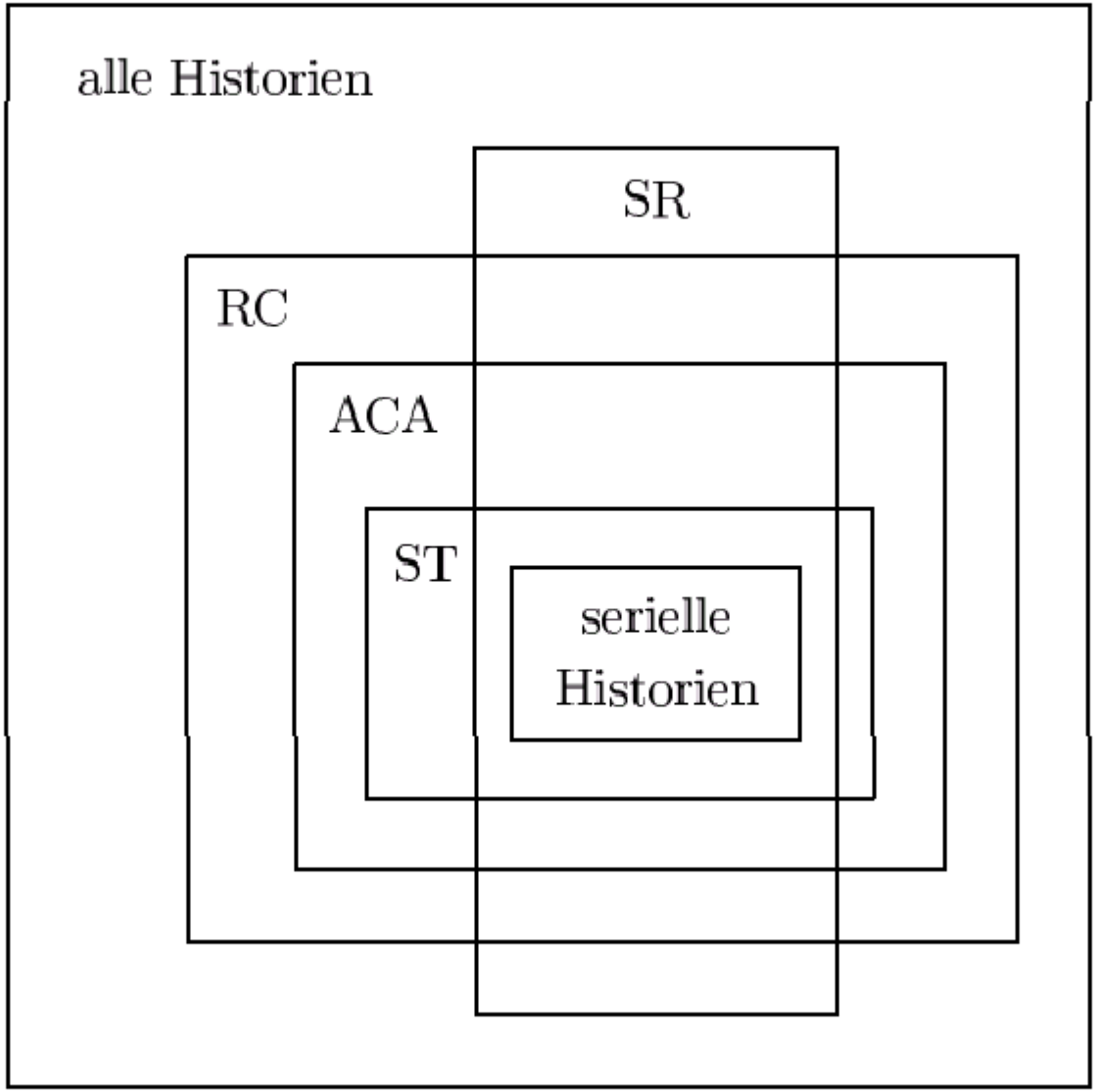


Zwei-Phasen-Sperrprotokoll = Commit Order Serialisierung

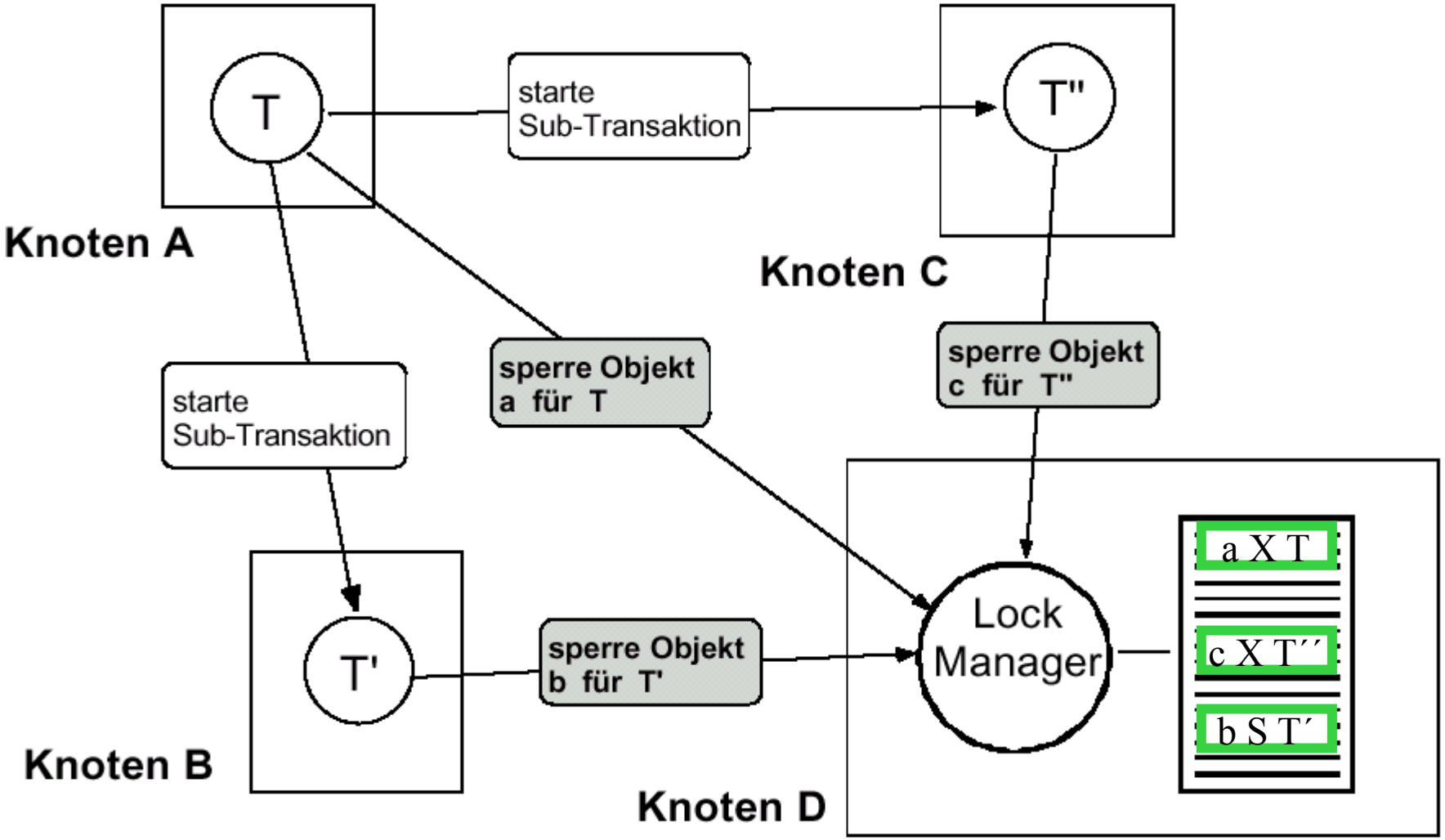


Äquivalenter serieller Schedule

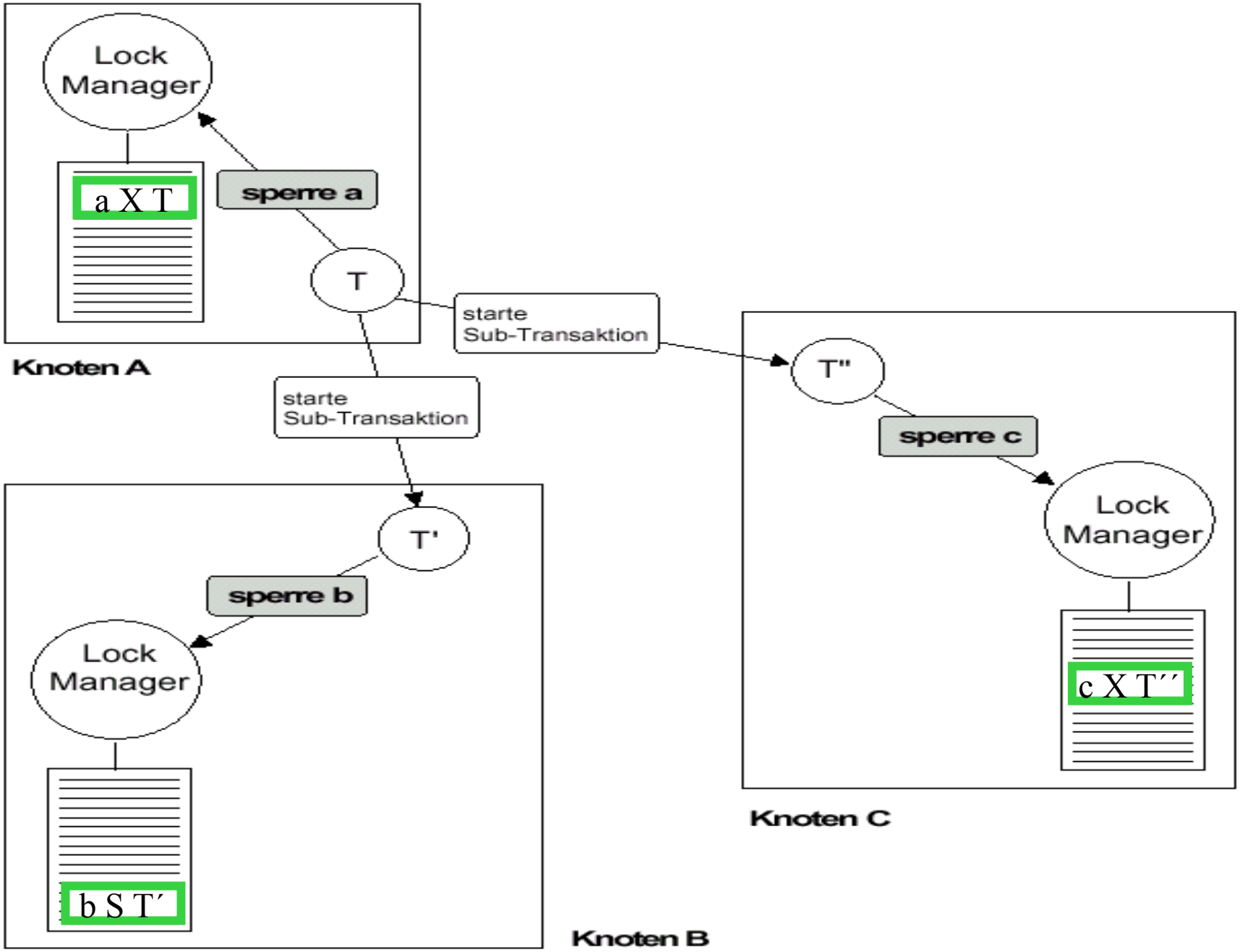
Die unterschiedlichen Historien



Synchronisation mittels zentraler Sperrverwaltung



Dezentrale (=lokale) Sperrverwaltung

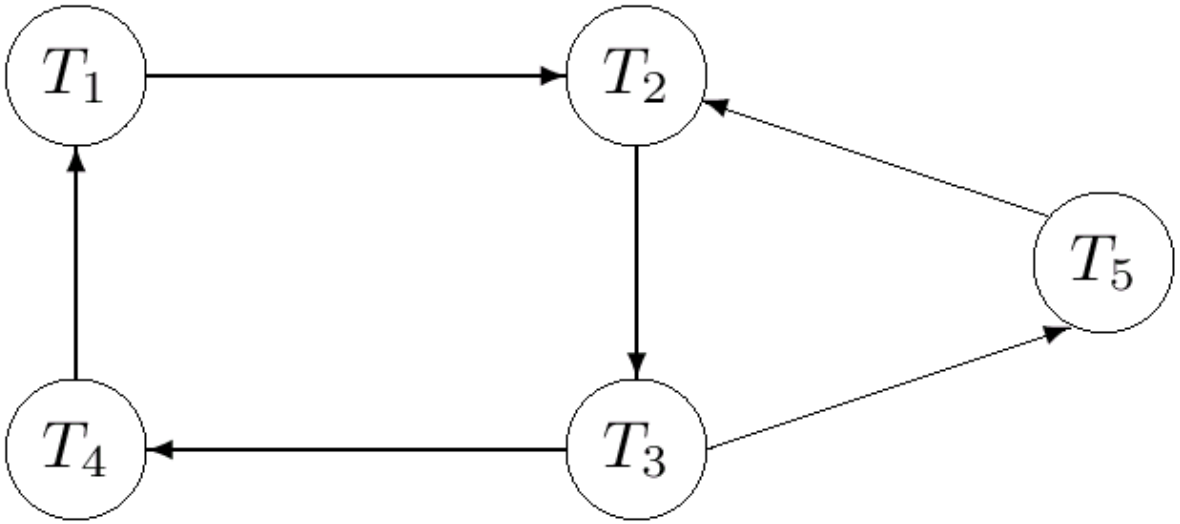


Ein verklemmter Schedule

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 muß warten auf T_2
9.		**lockS**(A)	T_2 muß warten auf T_1
10.	...	...	$\Rightarrow$*Deadlock*

Wartegraph mit zwei Zyklen:

- $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$
- $T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



- beide Zyklen können durch Rücksetzen von T_3 „gelöst“ werden
- Zyklenerkennung durch Tiefensuche im Wartegraphen

Serialisierbarkeit

Lokale Serialisierbarkeit an jeder der an den Transaktionen beteiligten Stationen reicht nicht aus. Deshalb muß man bei der Mehrbenutzersynchronisation auf globaler Serialisierbarkeit bestehen.

Beispiel (lokal serialisierbare Historien):

S_1

Schritt	T_1	T_2
1.	r(A)	
2.		w(A)

S_2

Schritt	T_1	T_2
3.		w(B)
4.	r(B)	

Reihenfolge: $T_1 \rightarrow T_2 \rightarrow T_1$

Deadlocks in VDBMS

- Erkennung von Deadlocks (Verklemmungen)
  - zentralisierte Deadlock-Erkennung
  - dezentrale (verteilte) Deadlock-Erkennung
- Vermeidung von Deadlocks

„Verteilter“ Deadlock

S_1

Schritt	T_1	T_2
0.	**BOT**	
1.	**lockS**(A)	
2.	r(A)	
6.		**lockX**(A)
		~~

S_2

Schritt	T_1	T_2
3.		**BOT**
4.		**lockX**(B)
5.		w(B)
7.	**lockS**(B)	
	~~	

Verklemmungs-vermeidende Sperrverfahren

- Jeder Transaktion wird ein eindeutiger Zeitstempel (TS) zugeordnet
- ältere TAs haben einen kleineren Zeitstempel als jüngere TAs
- TAs dürfen nicht mehr „bedingungslos“ auf eine Sperre warten

wound-wait Strategie

- T_1 will Sperre erwerben, die von T_2 gehalten wird
- Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen und zurückgesetzt, so daß T_1 weiterlaufen kann.
- Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.

wait-die Strategie

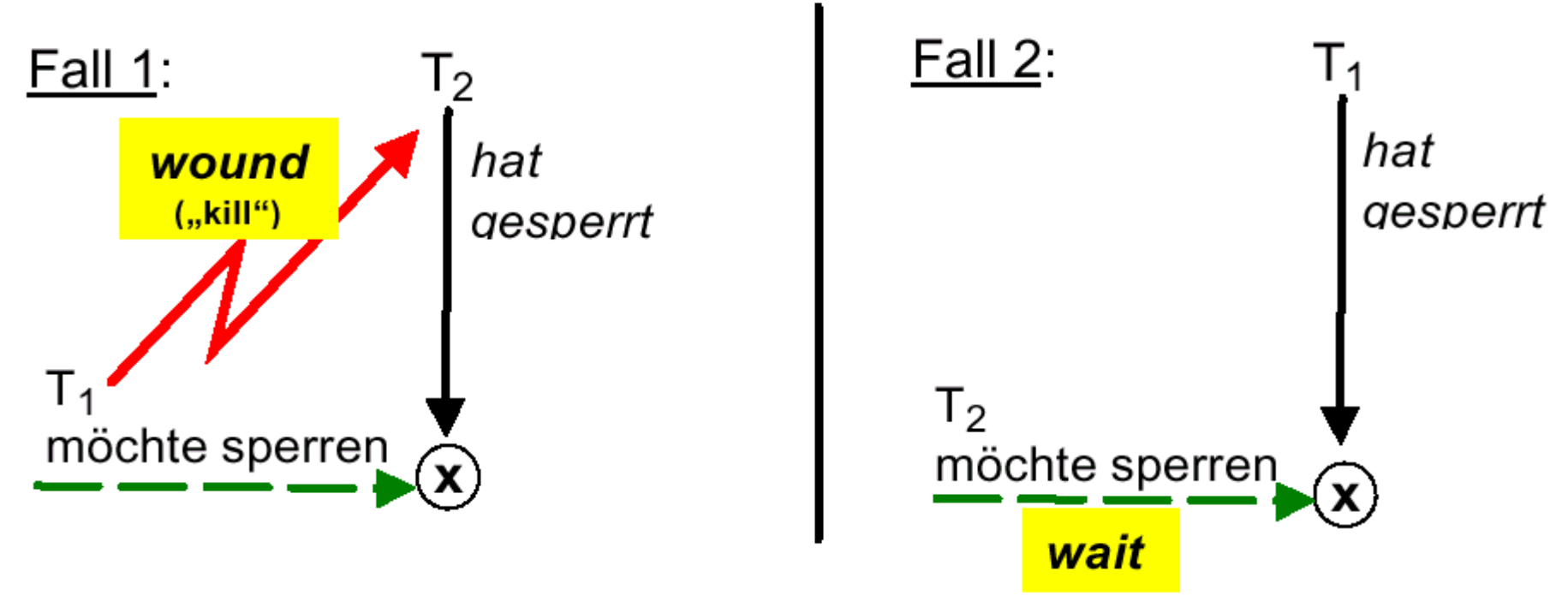
- T_1 will Sperre erwerben, die von T_2 gehalten wird
- Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre.
- Sonst wird T_1 abgebrochen und zurückgesetzt.

Wound/ Wait

Falls Tj im Zwei-Phasen-Commit Protokoll schon mit READY gestimmt hat, kann Tj nicht abgebrochen werden. In diesem Fall wartet Ti bis Tj fertig ist (es kann ja keine zyklische Wartebeziehung mit Tj geben. Warum? Deshalb heißt die Regel „wound“ und nicht „kill“.)

if $TS(T_i) < TS(T_j)$ then T_j *is wounded* else T_i *waits*

Wenn T_j "verwundet" wird, wird T_j abgebrochen und neu gestartet.

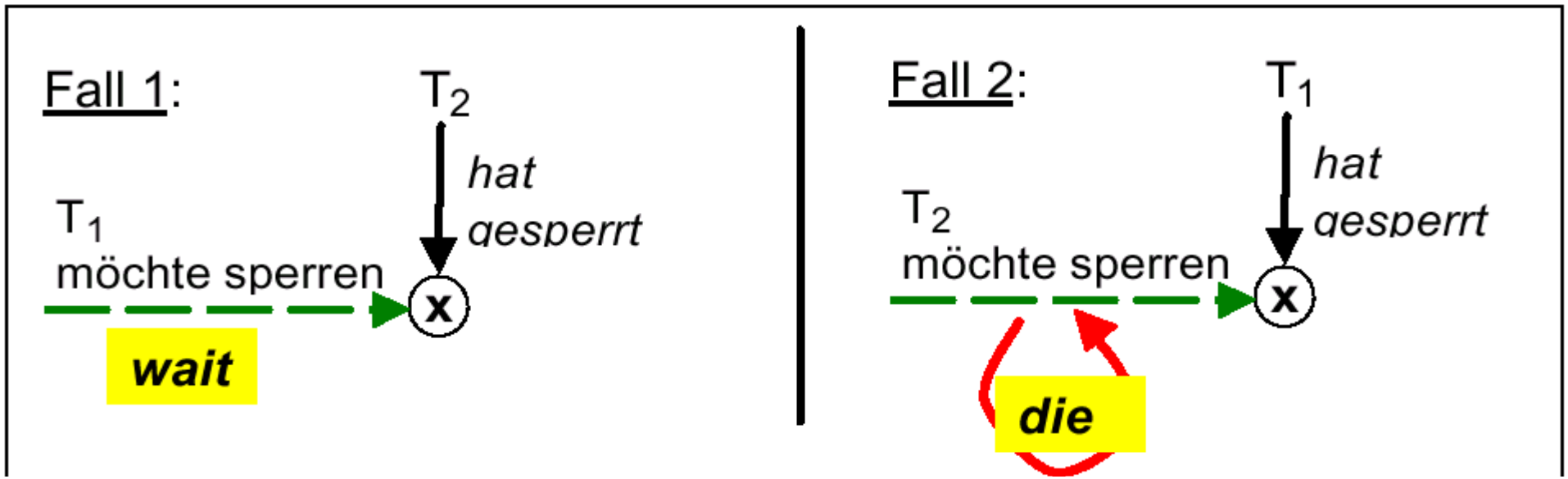


Wait/Die-Regel

if $TS(T_i) < TS(T_j)$ then T_i *waits* else T_i *dies*

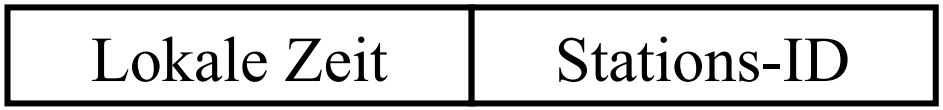
D.h. Transaktion T_i darf auf die Freigabe der Sperre nur dann warten, wenn sie "älter" als T_j ist.

Bei Abbruch (dann ist sie die "jüngere" Transaktion) wird T_i mit der alten Zeitmarke erneut gestartet.



Generierung global eindeutiger Zeitstempel

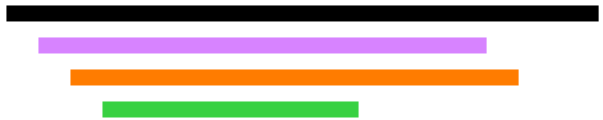
- Konkatenation lokaler Zeit und Stationskennung



- Für die Effektivität (nicht für die Korrektheit) wird die Genauigkeit der lokalen Uhren vorausgesetzt
- Stationen könnten sich durch Vor/Zurückstellen der lokalen Uhren einen Vorteil verschaffen

Zeitstempel-basierende Synchronisation

- TAs werden à priori gemäß ihres BOT-Zeitpunkts geordnet
- Äquivalenter serieller Schedule entspricht dann der BOT-Reihung



- schwarz vor violett vor rot vor grün
- Wenn eine alte TA zu spät zu einem Datum kommt (jüngere war vorher da), muss sie zurückgesetzt werden
- Beim 2PL wird die Reihung zweier TA dynamisch erst zum Zeitpunkt des ersten Konflikts vorgenommen
  - Sperrenhalter **vor** Sperrenanforderer

Zeitstempel-Basierende Synchronisation

Jedem Datum A in der Datenbasis werden bei diesem Synchronisationsverfahren zwei Marken zugeordnet:

1. $readTS(A)$:
2. $writeTS(A)$:

Synchronisationsverfahren

- T_i will A lesen, also $r_i(A)$
  - Falls $TS(T_i) < writeTS(A)$ gilt, haben wir ein Problem:
    - Die Transaktion T_i ist älter als eine andere Transaktion, die A schon geschrieben hat.
    - Also muß T_i zurückgesetzt werden.
  - Anderenfalls, wenn also $TS(T_i) \geq writeTS(A)$ gilt, kann T_i ihre Leseoperation durchführen und die Marke $readTS(A)$ wird auf $max(TS(T_i), readTS(A))$ gesetzt.

Zeitstempel-Basierende Synchronisation (Fortsetzung ...)

- T_i will A schreiben, also $w_i(A)$
  - Falls $TS(T_i) < readTS(A)$ gilt, gab es eine jüngere Lesetransaktion, die den neuen Wert von A, den T_i gerade beabsichtigt zu schreiben, hätte lesen müssen. Also muß T_i zurückgesetzt werden.
  - Falls $TS(T_i) < writeTS(A)$ gilt, gab es eine jüngere Schreibtransaktion. D.h. T_i beabsichtigt einen Wert einer jüngeren Transaktion zu überschreiben. Das muß natürlich verhindert werden, so daß T_i auch in diesem Fall zurückgesetzt werden muß.
  - Anderenfalls darf T_i das Datum A schreiben und die Marke $writeTS(A)$ wird auf $TS(T_i)$ gesetzt.

Optimistische Synchronisation

1. *Lesephase*:
   - In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen.
   - Gegenüber der Datenbasis tritt die Transaktion in dieser Phase aber nur als Leser in Erscheinung, da alle gelesenen Daten in lokalen Variablen der Transaktion gespeichert werden.
   - alle Schreiboperationen werden (zunächst) auf diesen lokalen Variablen ausgeführt.
2. *Validierungsphase*:
   - In dieser Phase wird entschieden, ob die Transaktion möglicherweise in Konflikt mit anderen Transaktionen geraten ist.
   - Dies wird anhand von Zeitstempeln entschieden, die den Transaktionen in der Reihenfolge zugewiesen werden, in der sie in die Validierungsphase eintreten.
3. *Schreibphase*:
   - Die Änderungen der Transaktionen, bei denen die Validierung positiv verlaufen ist, werden in dieser Phase in die Datenbank

Validierung bei der optimistischen Synchronisation

Lesephase | Validierungsphase | Schreibphase

Lesephase | Validierungsphase | Schreibphase

Vereinfachende Annahme: Es ist immer nur eine TA in der Validierungsphase!

Wir wollen eine Transaktion T_j validieren. Die Validierung ist erfolgreich falls für **alle** älteren Transaktionen T_a – also solche die früher ihre Validierung abgeschlossen haben – eine der beiden folgenden Bedingungen gelten:

1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen – einschließlich der Schreibphase.

2. Die Menge der von T_a geschriebenen Datenelemente, genannt $WriteSet(T_a)$, enthält keine Elemente der Menge der gelesenen Datenelemente von T_j, genannt $ReadSet(T_j)$. Es muß also gelten:

$$WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$$

Zwei-Phasen-Sperrprotokoll ... Deadlocks

- Alle empirischen Untersuchungen beweisen (deuten darauf hin), dass 2PL in der Praxis das beste Verfahren ist
- ... Auch für verteilte Datenbanken
- also müssen wir uns mit dem Problem der Deadlocks auseinandersetzen
- Nicht-serialisierbare Schedules werden beim 2PL in einen Deadlock „transformiert“ (geleitet)
- (Leider auch einige serialisierbare Schedules ... Nur um auf der sicheren Seite zu sein)

„Verteilter“ Deadlock

S_1

Schritt	T_1	T_2
0.	**BOT**	
1.	**lockS**(A)	
2.	r(A)	
6.		**lockX**(A)
		~~

S_2

Schritt	T_1	T_2
3.		**BOT**
4.		**lockX**(B)
5.		w(B)
7.	**lockS**(B)	
	~~	

Erkennung und Auflösung von Verklemmungen (Deadlocks)...1

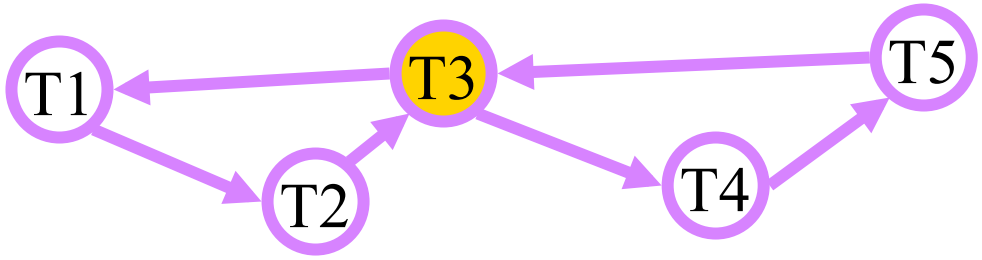
- **Durch Timeout**
  - sobald eine TA bei einer Teiloperation eine Zeitschranke überschreitet, wird eine Verklemmung angenommen
  - TA gibt sich selber auf, wird zurückgesetzt, danach Neustart
  - Problem: richtige Wahl der Zeitschranke
    - zu großer Wert: Deadlocks bleiben lange unerkannt und verstopfen das System
    - zu kleiner Wert: „falsche" Deadlocks werden erkannt un TAs werden unnötigerweise zurückgesetzt
  - Gefahr: ein stark belastetes System ist nur noch mit Rücksetzen und Neustart beschäftigt
  - Oracle wendet „richtige" Deadlockerkennung lokal und Timeout global an.Lokale Deadlocks werden schnell erkannt.

Erkennung und Auflösung von Verklemmungen (Deadlocks) ...2

- **Durch verklemmungsvermeidende Verfahren**
  - optimistische Synchronisation
    - bei Erkennung eines Konflikts in der Validierungsphase wird die Transaktion zurückgesetzt
    - viel Arbeit geht verloren
    - Gefahr: in stark belastetem System „kommt keiner durch"
  - verklemmungsvermeidende Sperrverfahren
    - TA darf nicht bedingungslos auf Sperrfreigabe warten
    - wound/wait: nur jüngere warten auf ältere TAs
    - wait/die: nur ältere warten auf jüngere TAs
  - Zeitstempelverfahren
    - globale Uhr zur Reihung der TAs
    - wer „zu spät kommt" wird zurückgesetzt und mit neuem/jüngeren Zeitstempel neu gestartet

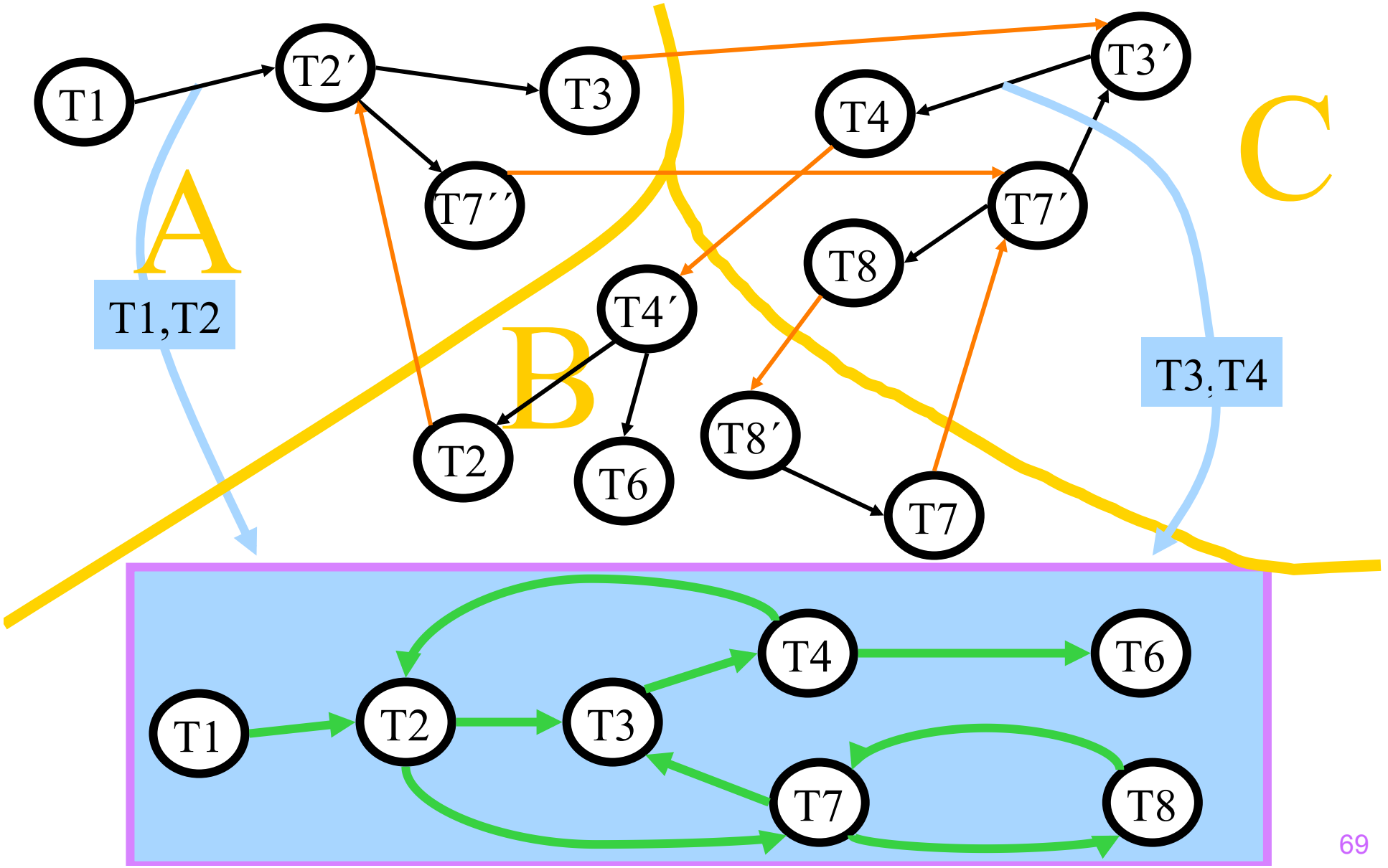
„Echte“ Deadlockerkennung

- **Zentralisierte Deadlocksuche**
  - lokale Stationen melden der zentralen Stelle die Wartebeziehungen (T1,T2) ~ T1 wartet auf T2
  - Aufbau des globalen Wartegraphen
  - vollständiges Wissen ⇒ gute Entscheidungen
  - nur echte Deadlocks werden erkannt
  - keine Phantomdeadlocks
  - Versuch, die Anzahl der Opfer zu minimieren (NP hart)



  - verschiedene Strategien, ein „gutes“ Opfer zu finden
    - junge TA: wenig Aufwand beim Rücksetzen
    - alte TA: viele Ressourcen werden frei und erlauben dem Gesamtsystem wieder „voran zu kommen“

Zentrale Deadlockerkennung



Dezentrale (=verteilte) Deadlockerkennung

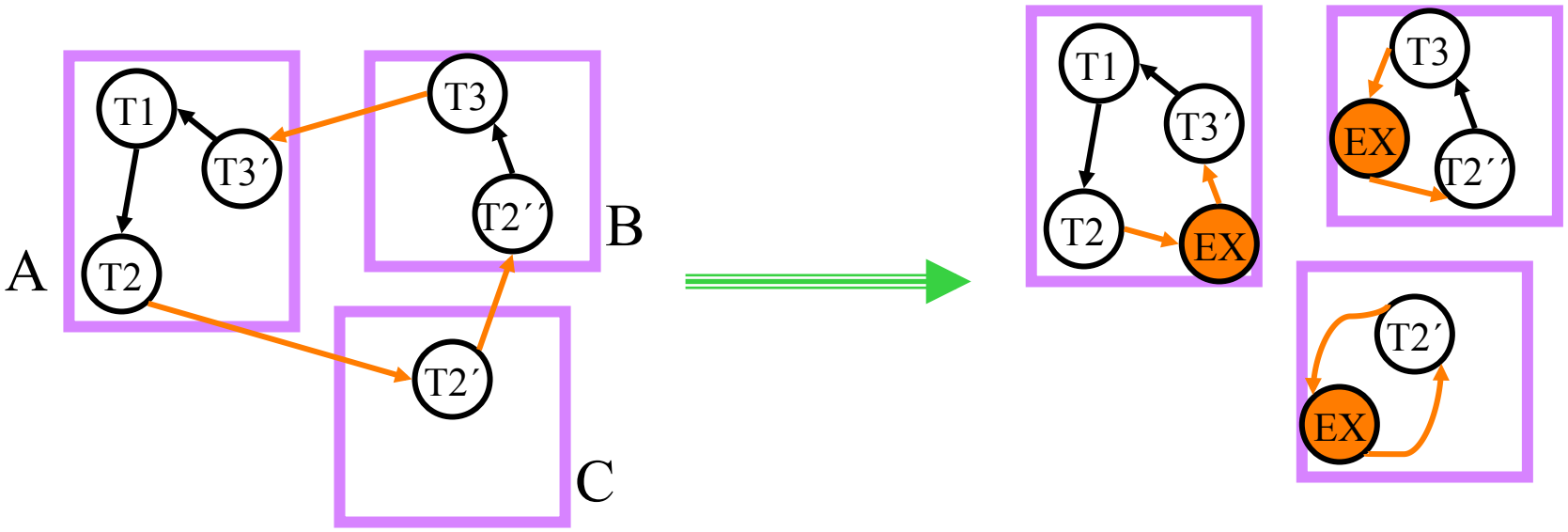
- Verzicht auf den Aufbau eines globalen Wartegraphen
  - dadurch oft die Gefahr, Phantom-Deadlocks zu erkennen (wg. Veraltetem Wissen)
- verteilte Deadlockerkennung ist ein sehr schwieriges Problem
  - es gibt viele falsche Algorithmen in der Literatur
- Hier in der Vorlesung (sowie in allen einschlägigen Lehrbüchern) wird der Obermarck Algorithmus vorgestellt
- wurde bei IBM für System R* entwickelt
- ist aber eigentlich kein wirklich guter Algorithmus
- aber pädagogisch wertvoll :-)
- Neuere Arbeit: N. Krivokapic´, A. Kemper, E. Gudes. Deadlock Detection in Distributed Databases: A New Algorithm and a Comparative Performance Evaluation. The VLDB Journal, Vol. 8, No. 2, pages 79-100, October 1999
- http://www.db.fmi.uni-passau.de/publications/journals/vldbj1999.pdf

Unterschiedliche Ansätze für echte verteilte Deadlockerkennung

- Path Pushing
  - Teile des globalen Wartegraphen (=Pfade) werden zwischen den Stationen ausgetauscht
  - die einzelnen Stationen haben unvollständiges Wissen
- Probing-basierte Algorithmen
  - Edge Chasing (entlang virtuellen Kanten des Wartegraphen)
    - eine wartende Transaktion sendet eine Nachricht an die TA, auf die sie wartet
    - eine TA leitet solch eine Nachricht an andere TA weiter, auf die sie selber wartet
    - kommt die Nachricht an den Initiator zurück = Deadlock/Zyklus
  - Diffusing Computation
    - Overkill für Deadlockerkennung in Datenbanken
    - Zyklen kann man besser/schneller mit Edge Chasing finden
- Global State Detection (unser DDA Algorithmus, später)

Der Obermarck-Algorithmus:
Path Pushing Algorithmus

- Alle Stationen sind an der Suche nach Deadlocks beteiligt
- externe Wartebeziehungen werden mittels lokalem EX-Knoten modelliert



- Kritisch: EX→ ... →EX (mögliche globale Deadlockzyklen)
- Informationen über solche Zyklen werden an andere Stationen verschickt.
- (EX,T3,T1,T2,EX) wird an C geschickt, da T2 auf eine SubTA an C wartet.

Reduktion des Nachrichtenaufkommens

- Ein Pfad EX → Ti → ..... → Tj → EX wird an die Station weitergereicht, bei der Tj eine SubTA gestartet hat.
- Er wird nur dann weitergereicht, wenn gilt
  - TransID(Ti) > TransID(Tj)
- Dadurch Reduktion des Nachrichtenvolumens auf die Hälfte
- Korrektheit? Wenn es einen echten Zyklus
  - Tz → ... → Ti → ..... → Tj → ... → Tz
- gibt, dann muss an mindestens einer Station ein Zyklus
  - EX → Tl → ... → Tk → EX

  vorkommen mit (Tl → ... → Tk) als Teilpfad von (Tz → ... → Ti → ..... → Tj → ... → Tz) und TransID(Tl) > TransID(Tk)
- So wird dann der Gesamtzyklus Schritt für Schritt aufgebaut

Der Algorithmus im Detail (1)

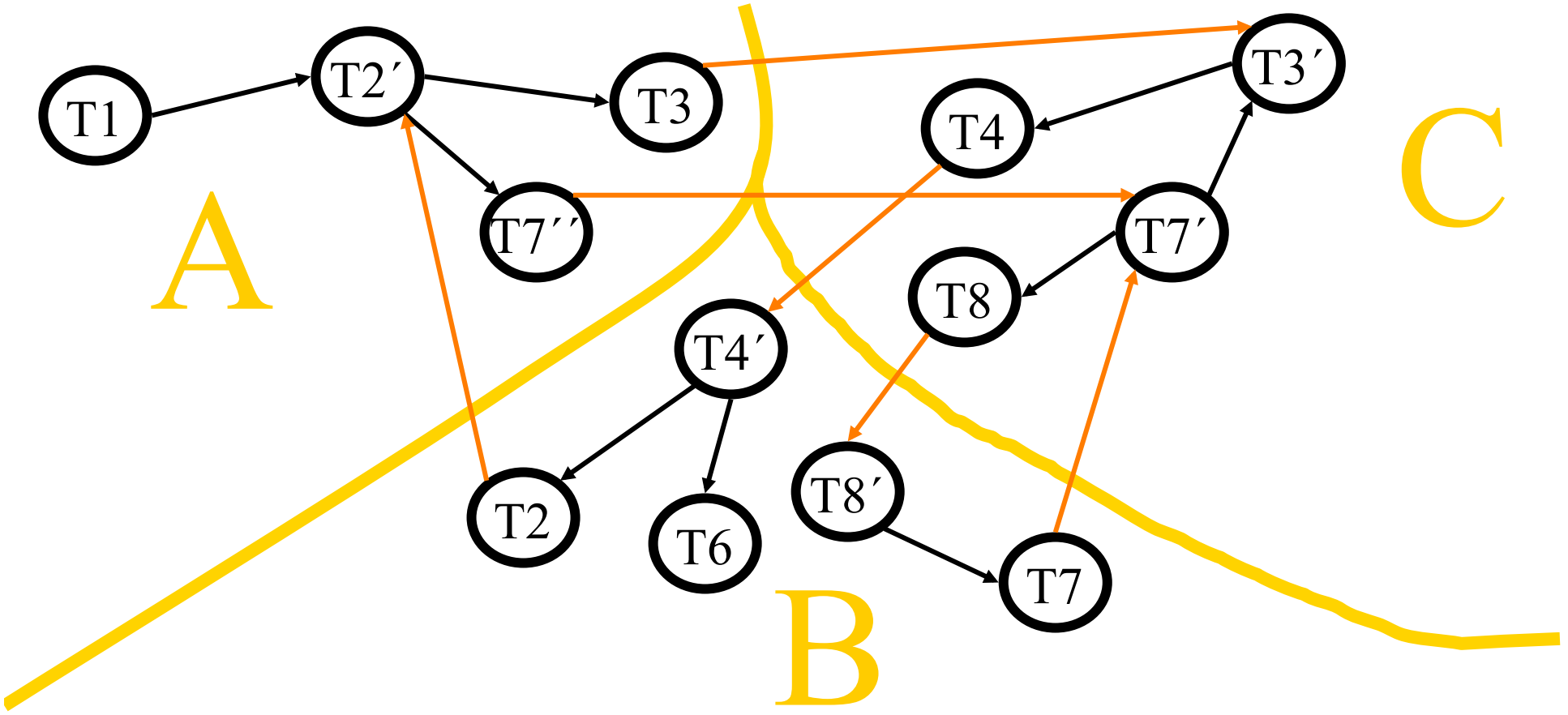
- Konstruiere lokalen Wartegraphen mit EX-Knoten. Gehe zu Schritt 5.
- Eintreffende Information von anderen Knoten wird eingebaut
  - Opfer und deren Kanten werden entfernt
  - eintreffende Zyklen mit bekannten Opfern werden ignoriert
  - noch nicht bekannte TA werden als Knoten eingefügt
- für jedes T´auf das von „außen" gewartet wird, füge die Kante EX → T´ ein
- für jedes T das auf Nachricht einer nicht-lokalen TA wartet füge die Kante T → EX ein
- Analysiere den Wartegraphen und erstelle Zyklenliste (nur elementare Zyklen ~ enthalten keine Subzyklen)
- Wenn ein Zyklus Tz → ... → Tv ... → Tz ohne EX gefunden wurde, bestimme ein Opfer Tv
- entferen Tv aus Wartegraph und Zyklenliste und falls Tv eine globale TA war, informiere andere Knoten

Der Algorithmus im Detail (2)

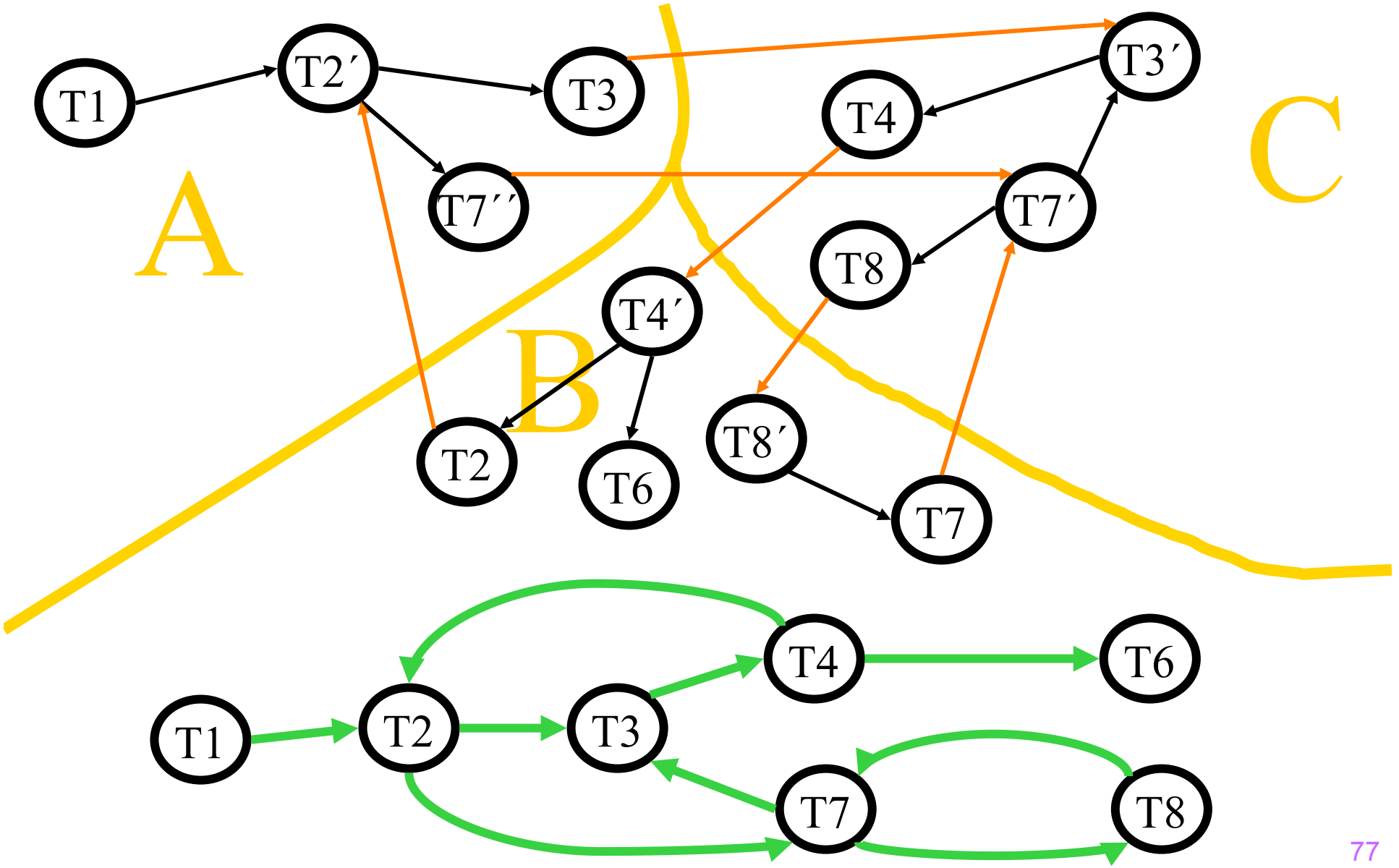
Zyklenliste enthält nur noch solche EX → ... → EX

8. Ermittle Zyklen der Form EX → Ti → ..... → Tj → EX
   - wird an die Station weitergereicht, bei der Tj eine SubTA gestartet hat.
   - Er wird nur dann weitergereicht, wenn gilt
     - TransID(Ti) > TransID(Tj)
   - Gehe zu Schritt 2.

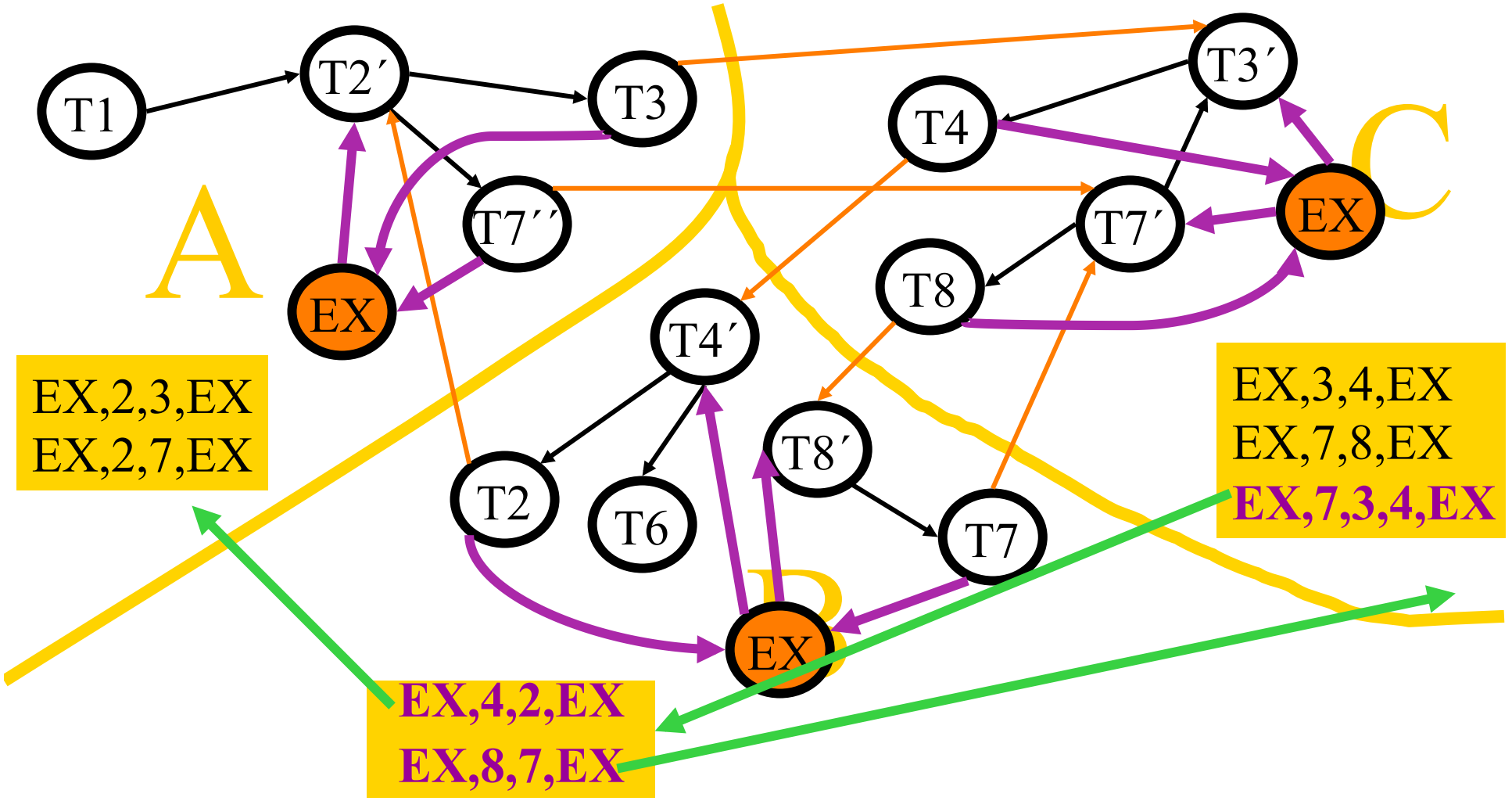
Beispiel-Szenario



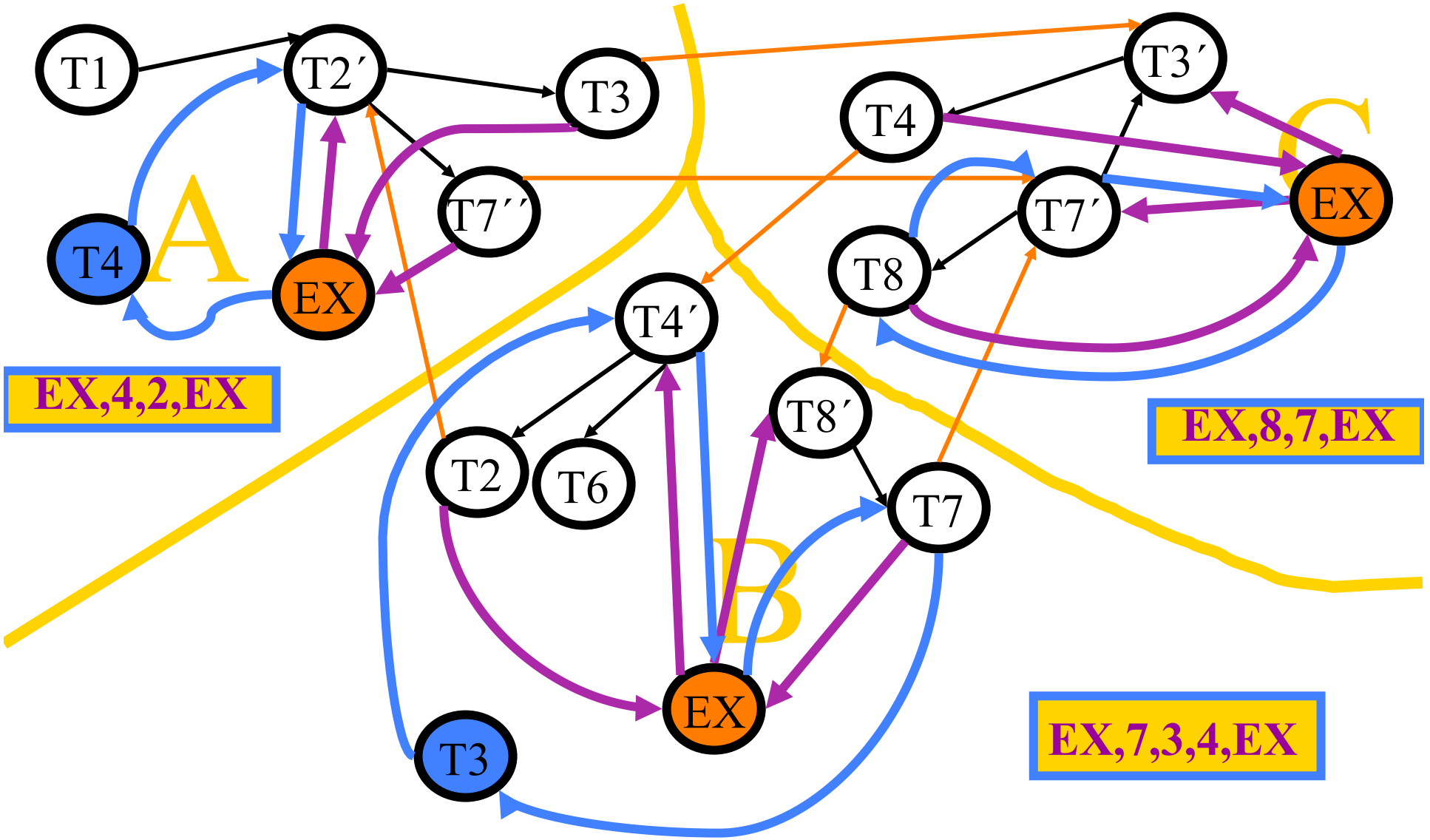
Beispiel-Szenario: Globaler Wartegraph



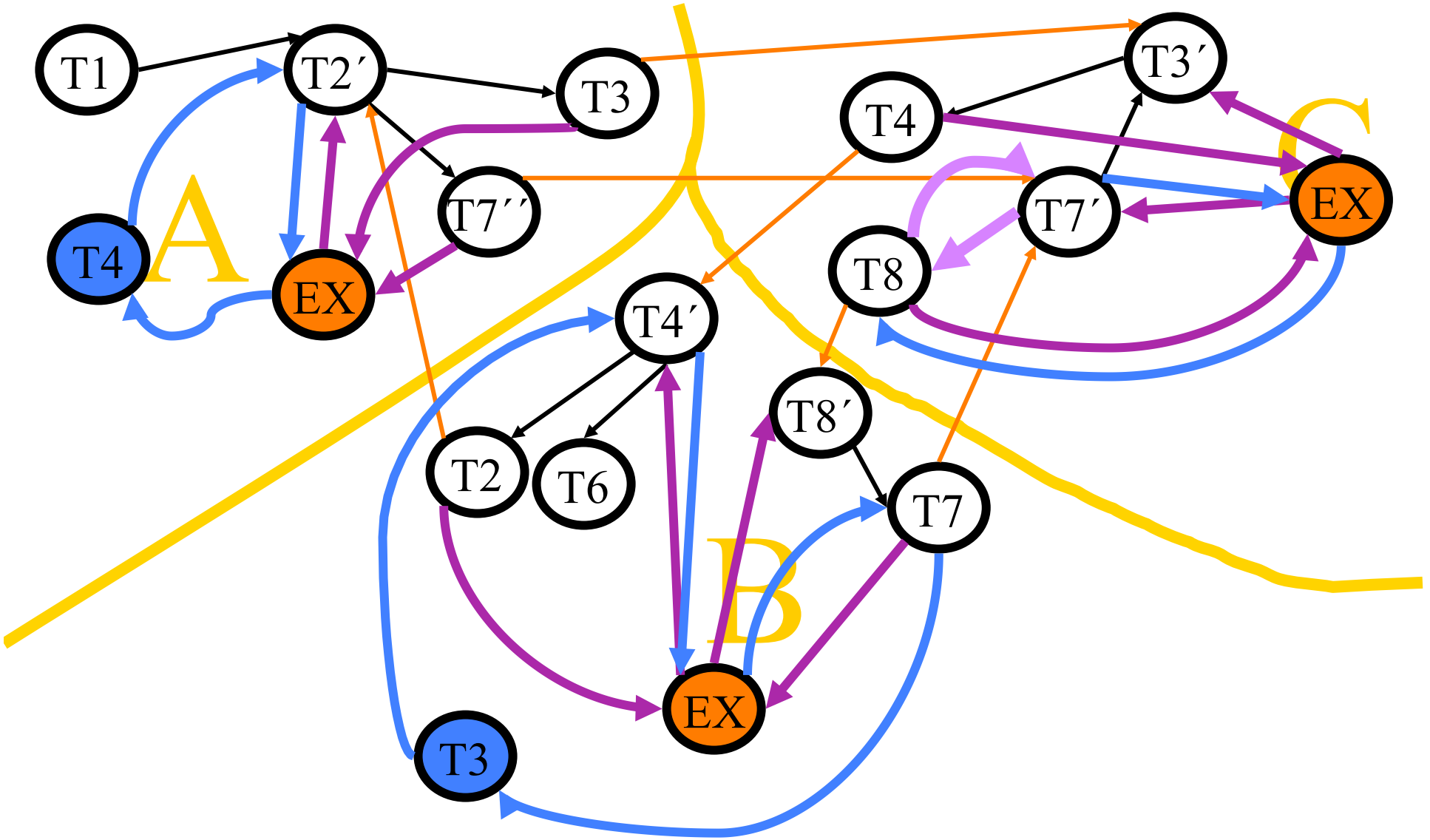
Beispiel-Szenario: Lokaler Wartegraph mit EX-Knoten



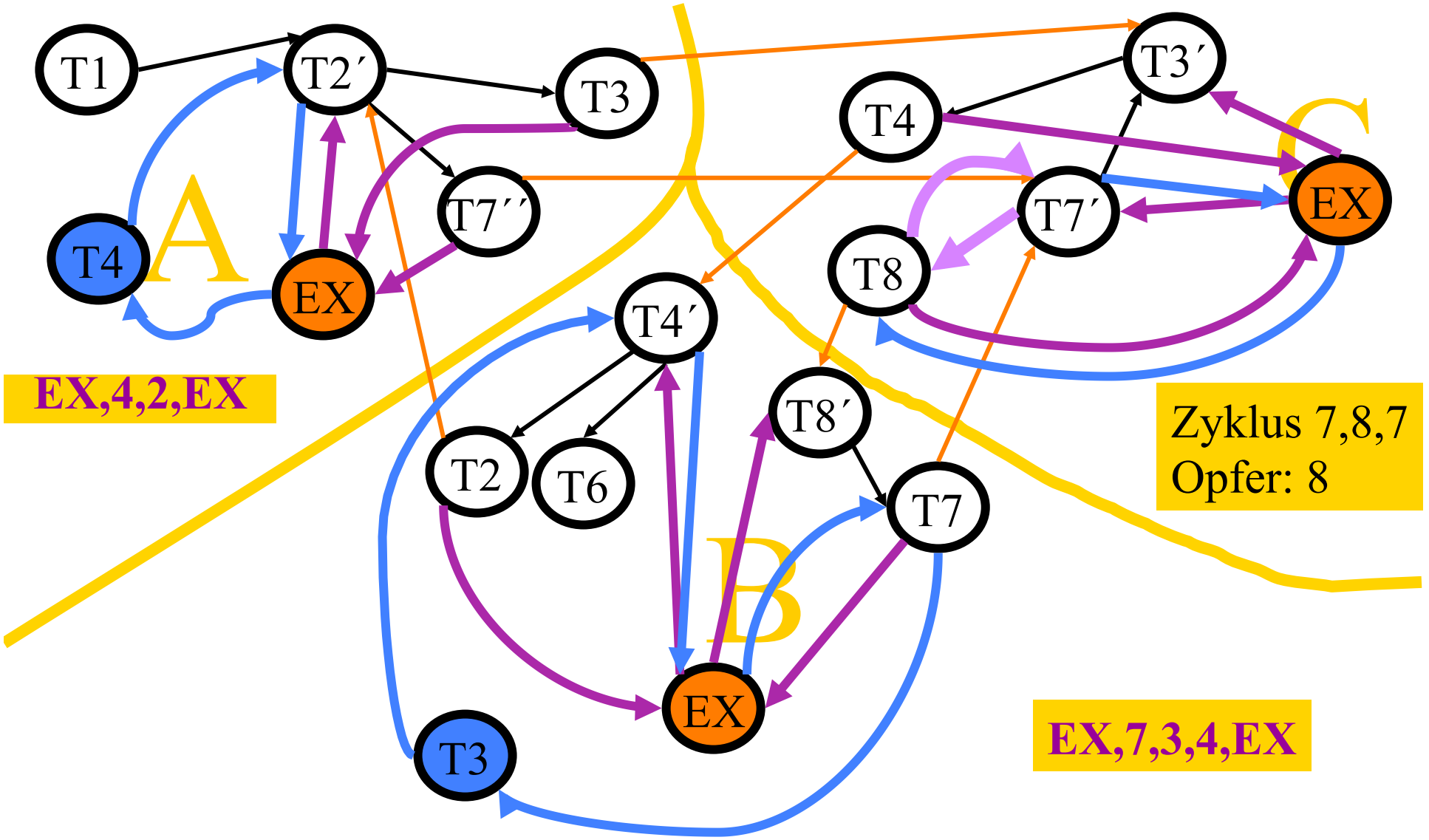
Lokale Wartegraphen nach Informationsaustausch (Path Pushing)



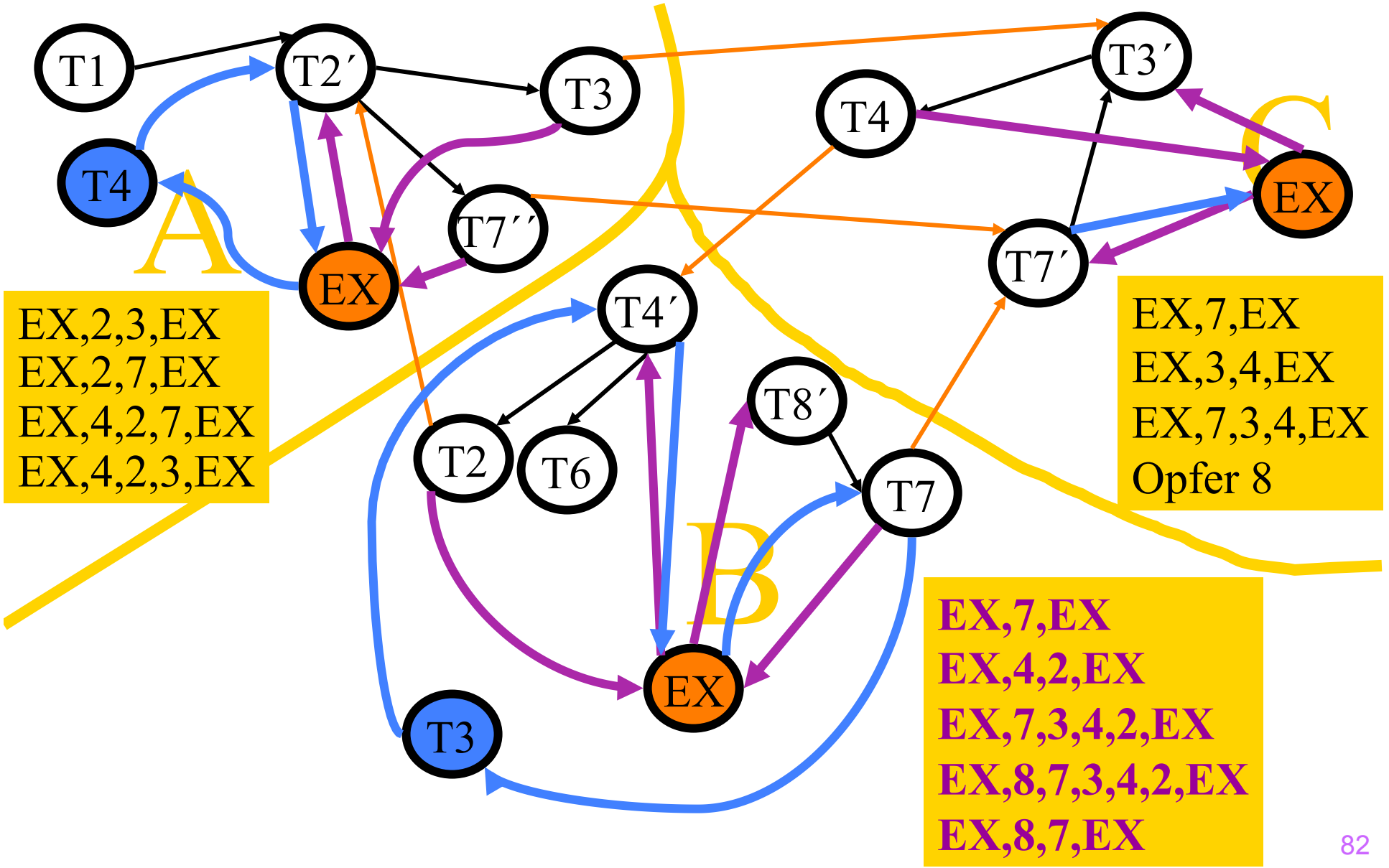
Beispiel-Szenario der lokalen Wartegraphen



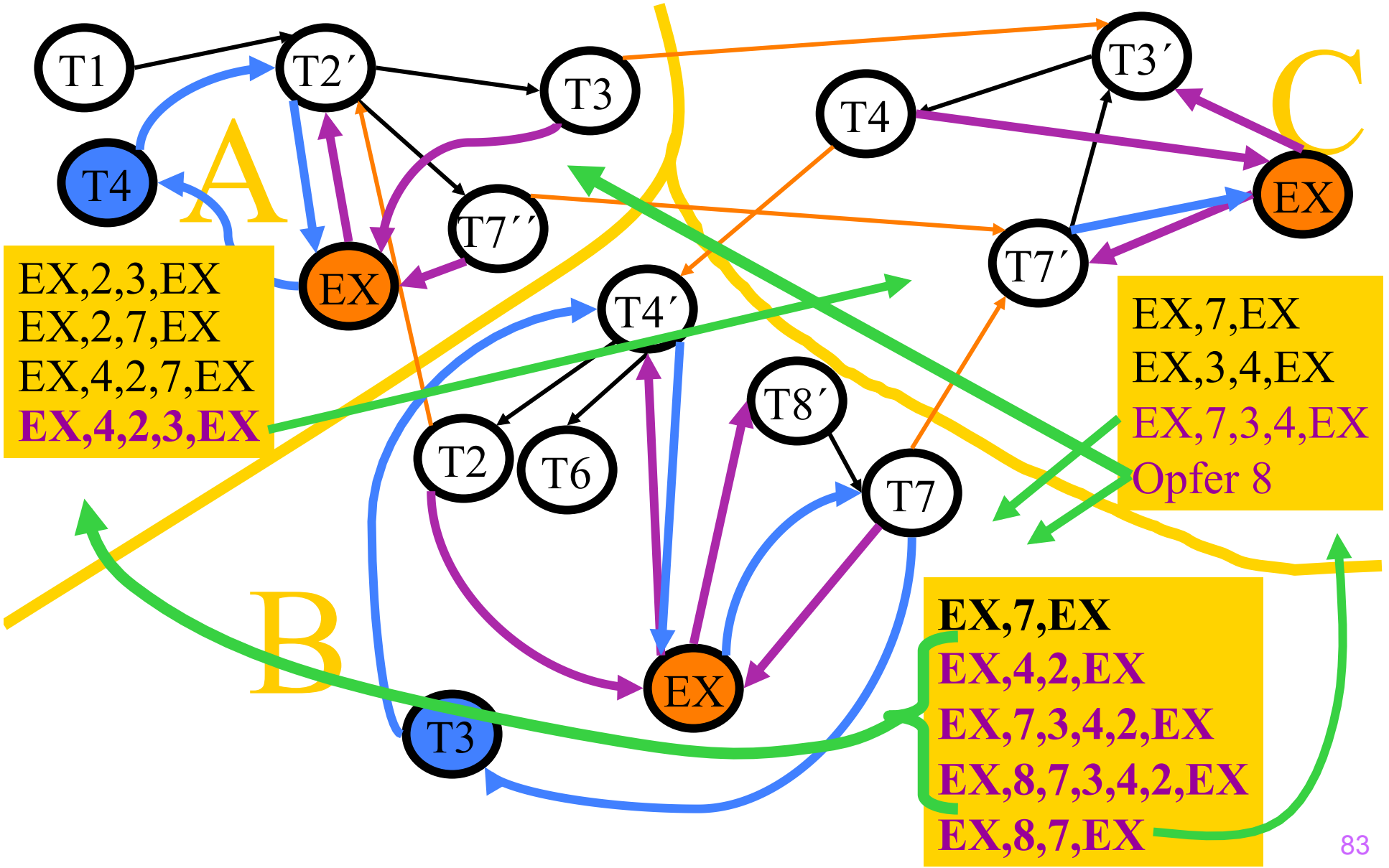
Zyklensuche in lokalen Wartegraphen



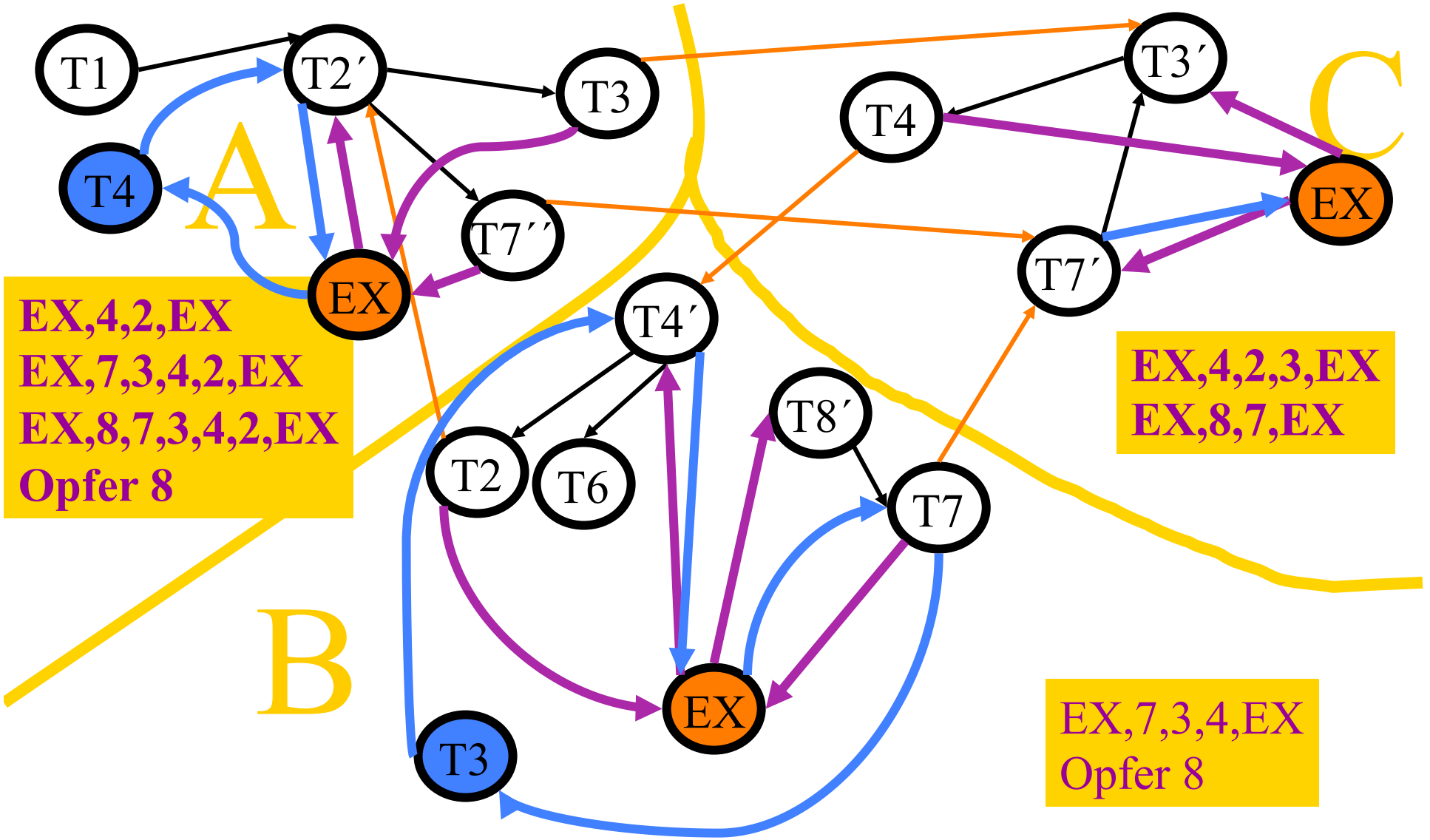
Rücksetzen des Opfers und Erstellung der Zyklenliste



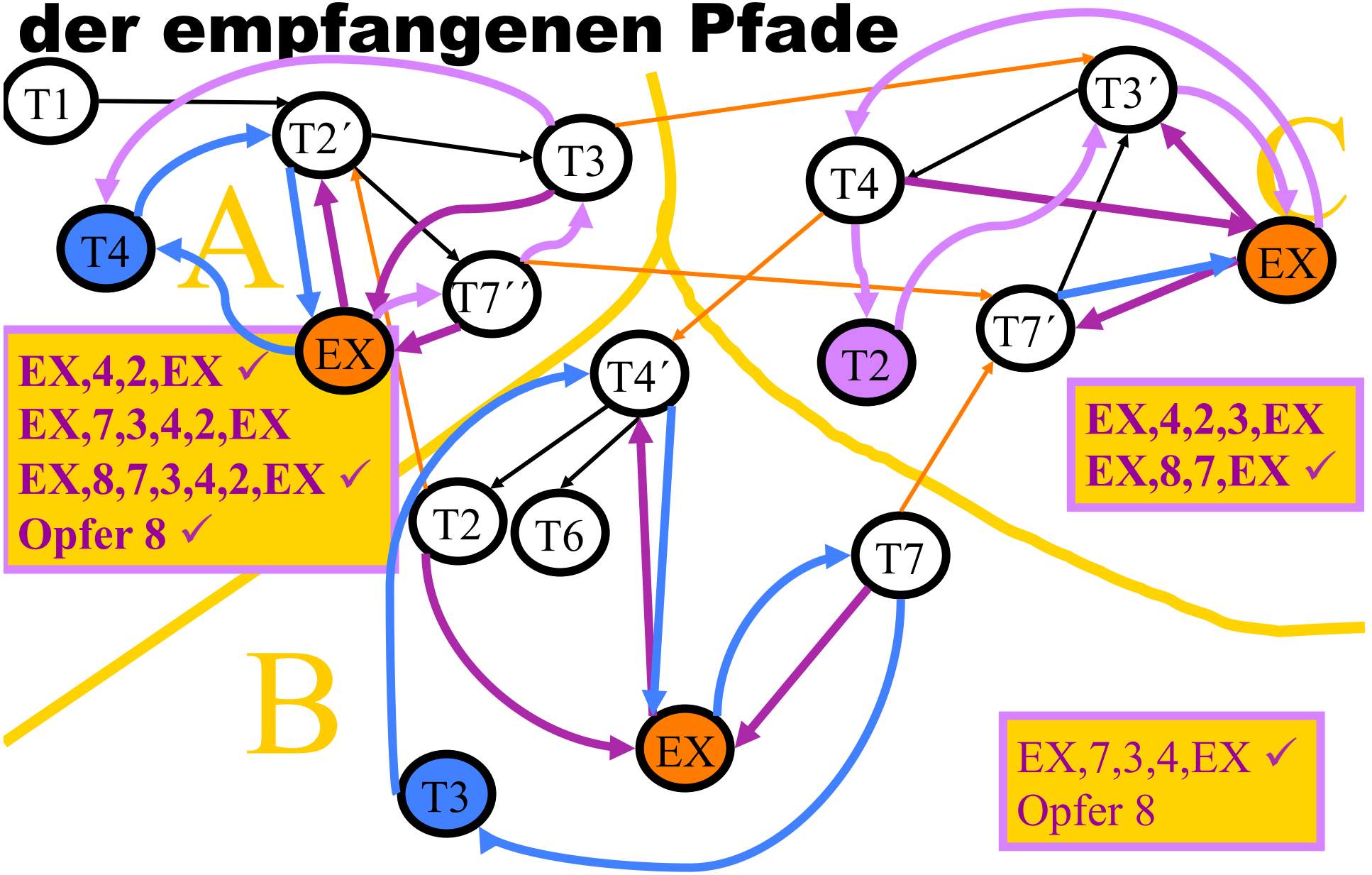
Path Pushing



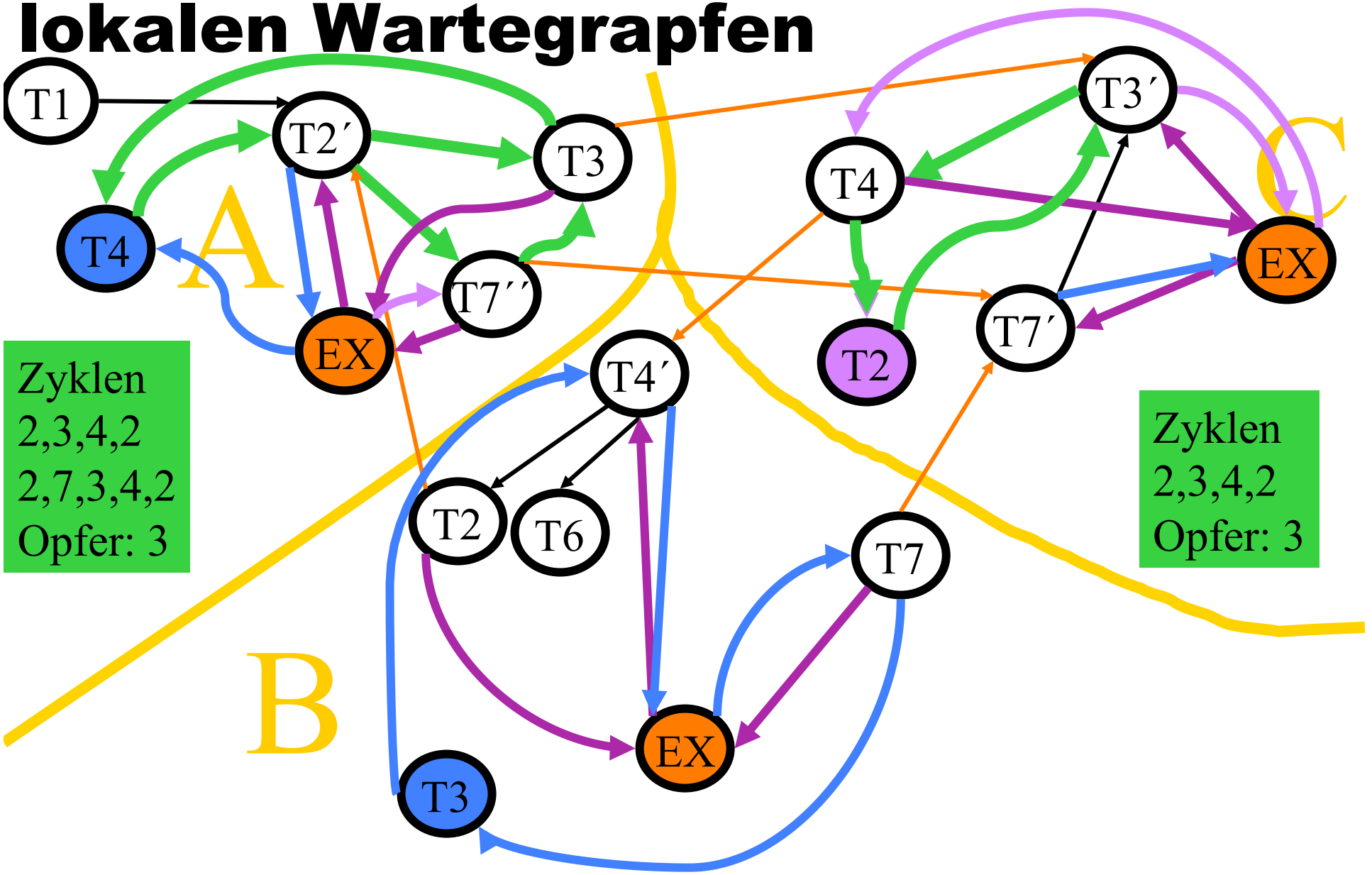
Nach Erhalt der Informationen



Bereinigung (Opfer 8) und Einbau der empfangenen Pfade

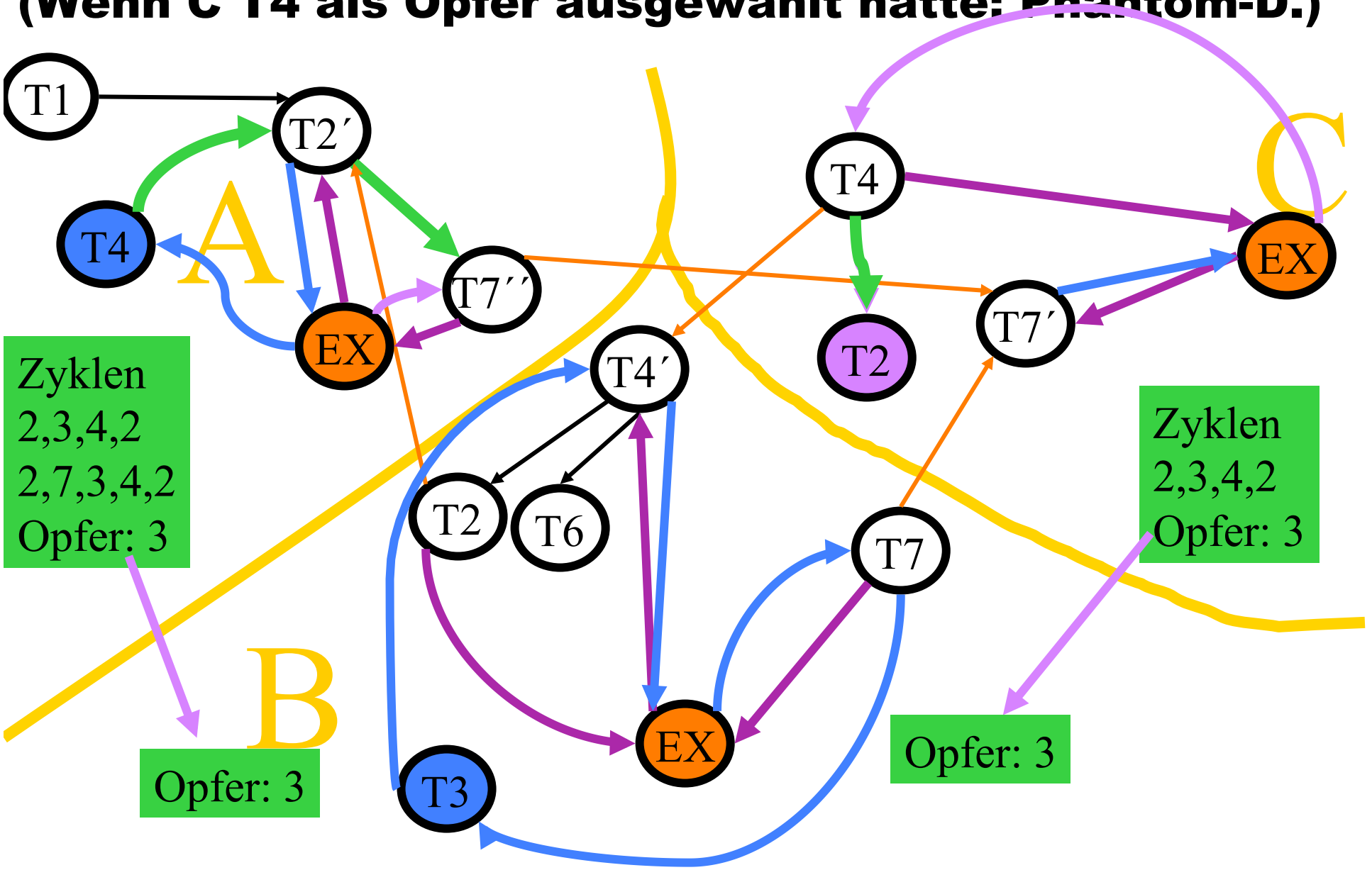


Ermittlung weiterer Zyklen in lokalen Wartegrapfen

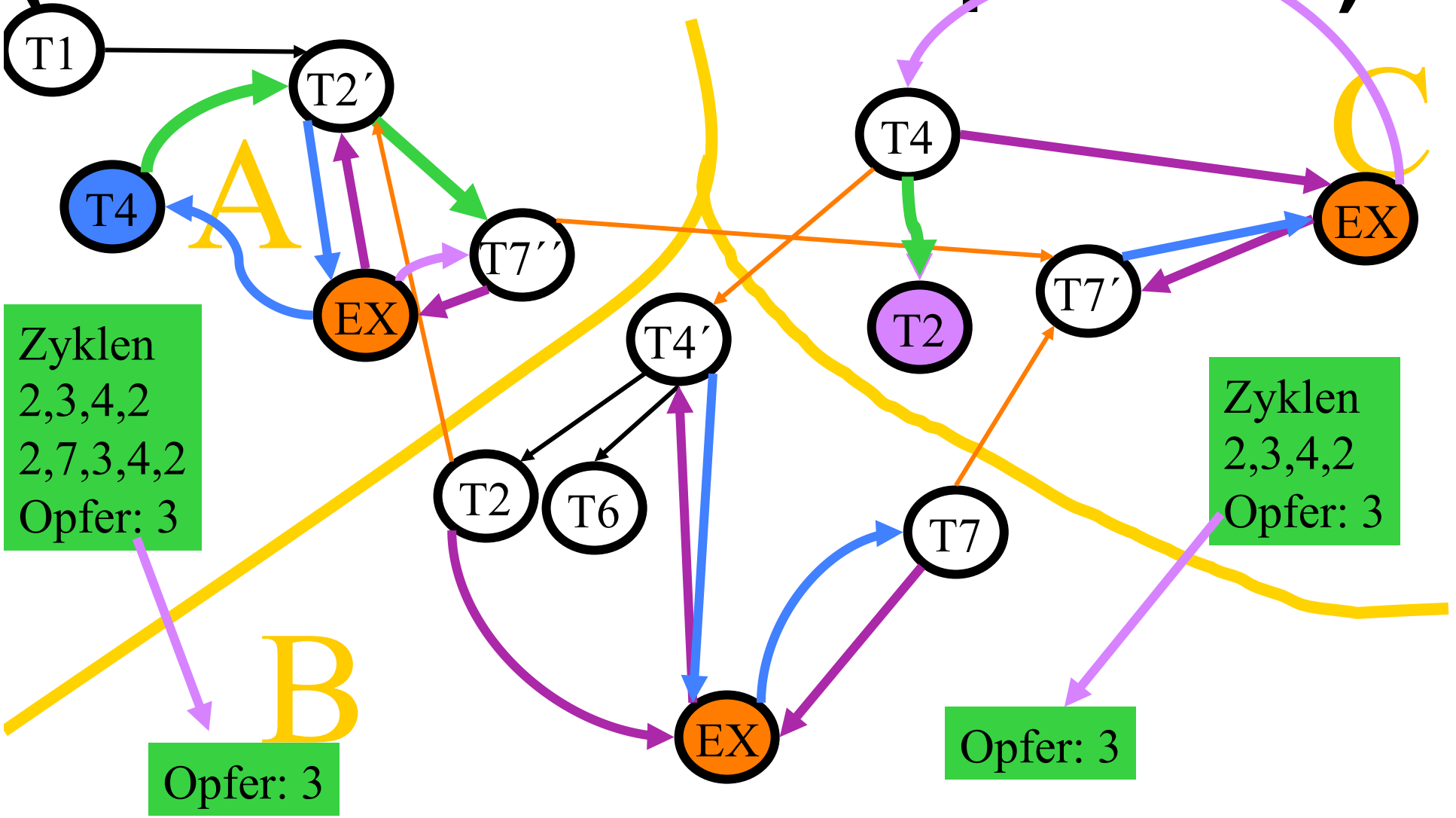


Rücksetzen des Opfers

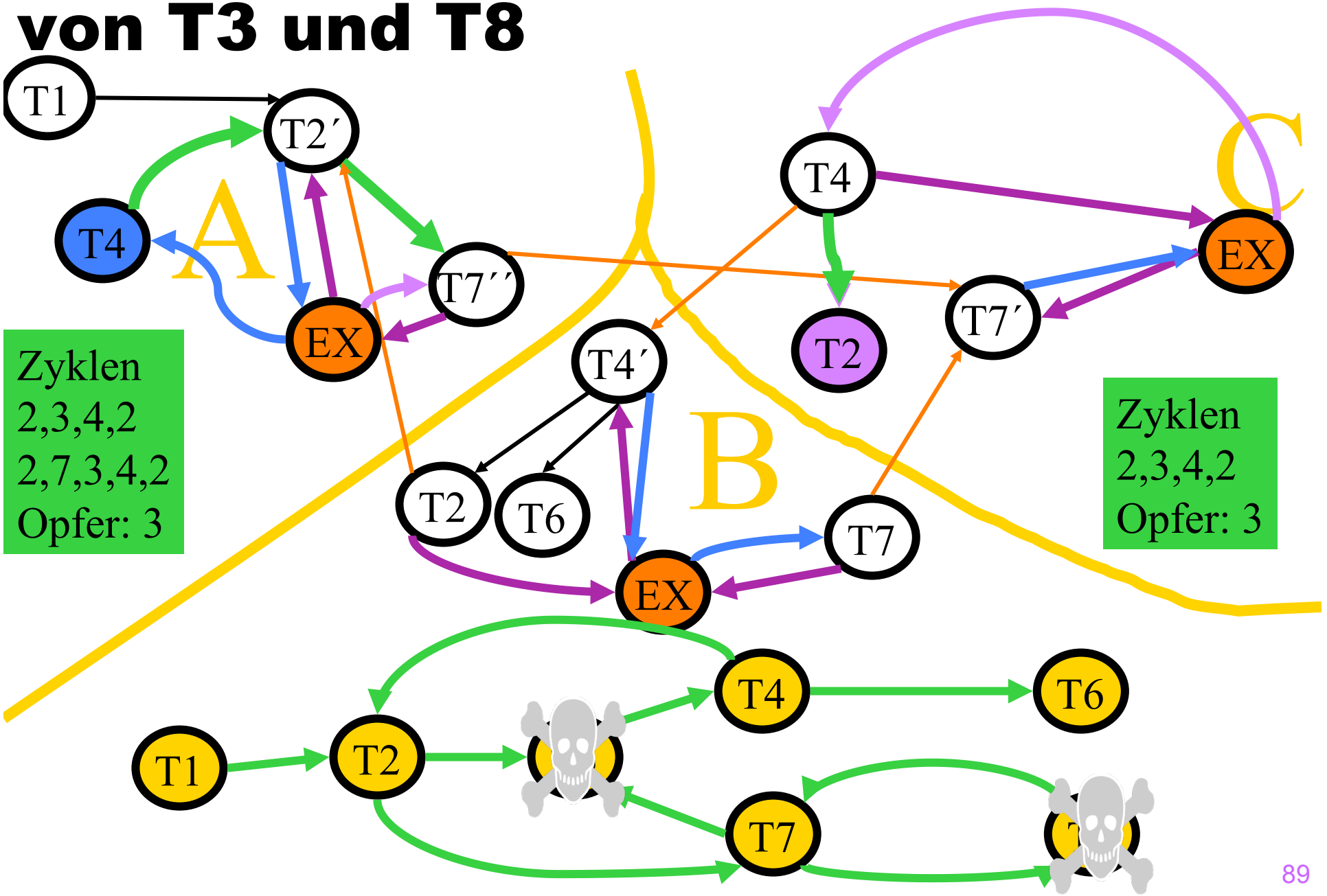
(Wenn C T4 als Opfer ausgewählt hätte: Phantom-D.)



Beispiel-Szenario (nach Elimination des Opfers in B)

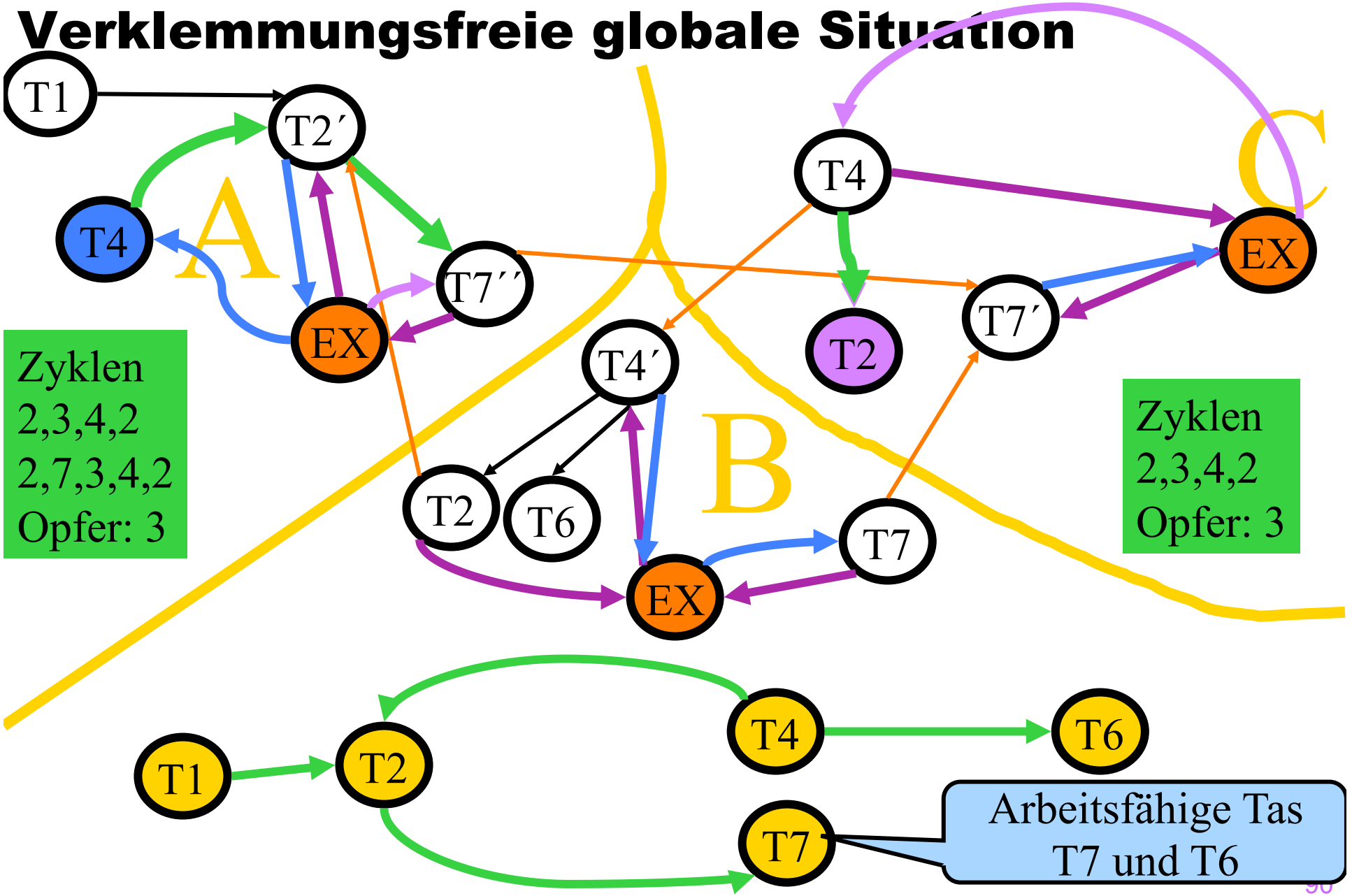


Globale Situation nach Rücksetzen von T3 und T8



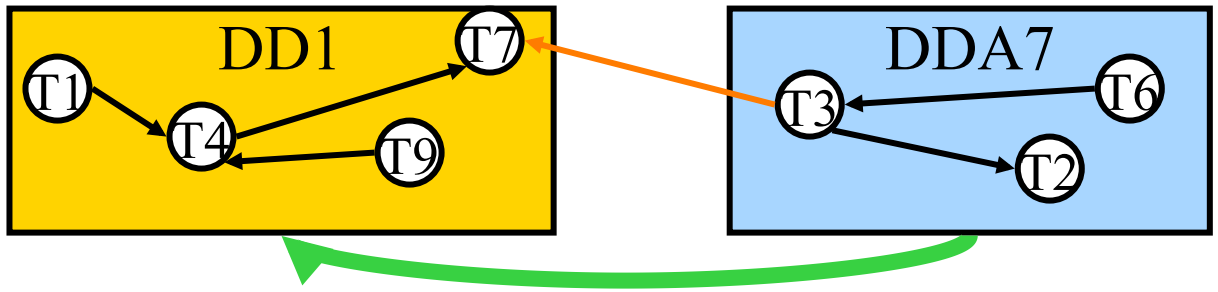
Und jetzt geht´s weiter ...

Verklemmungsfreie globale Situation



Deadlock Detection Agents

- Global state detection; aber verteilte Agenten
- DDAs werden zunächst wartenden Transaktionen zugeordnet
- jeder Agent ist zuständig für eine (oder mehrere) Zusammenhangskomponenten des Wartegraphen
- Sobald zwei Zusammenhangskomponenten verbunden werden, (hier rote Kante) werden die zugehörigen DDAs verschmolzen
  - der jüngere DDA (hier DDA7) verschmilzt mit dem älteren (hier DDA1)
  - alle Informationen werden dann an den älteren weitergeleitet
  - irgendwann darf der jüngere „sterben" (großer Timeout-Wert)



Instanziierung eines neuen DDA
(keine der beteiligten TAs hatte einen)

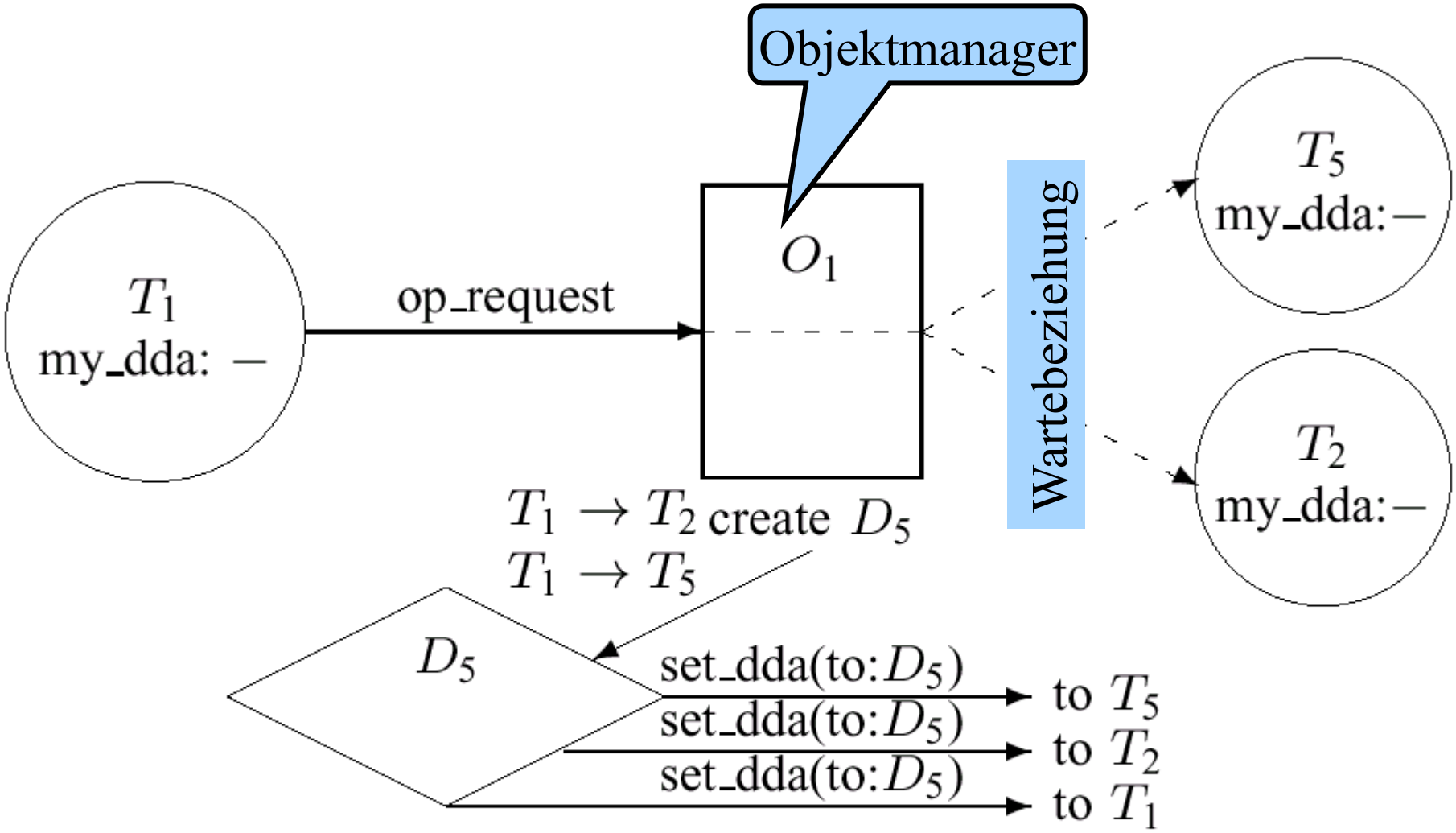
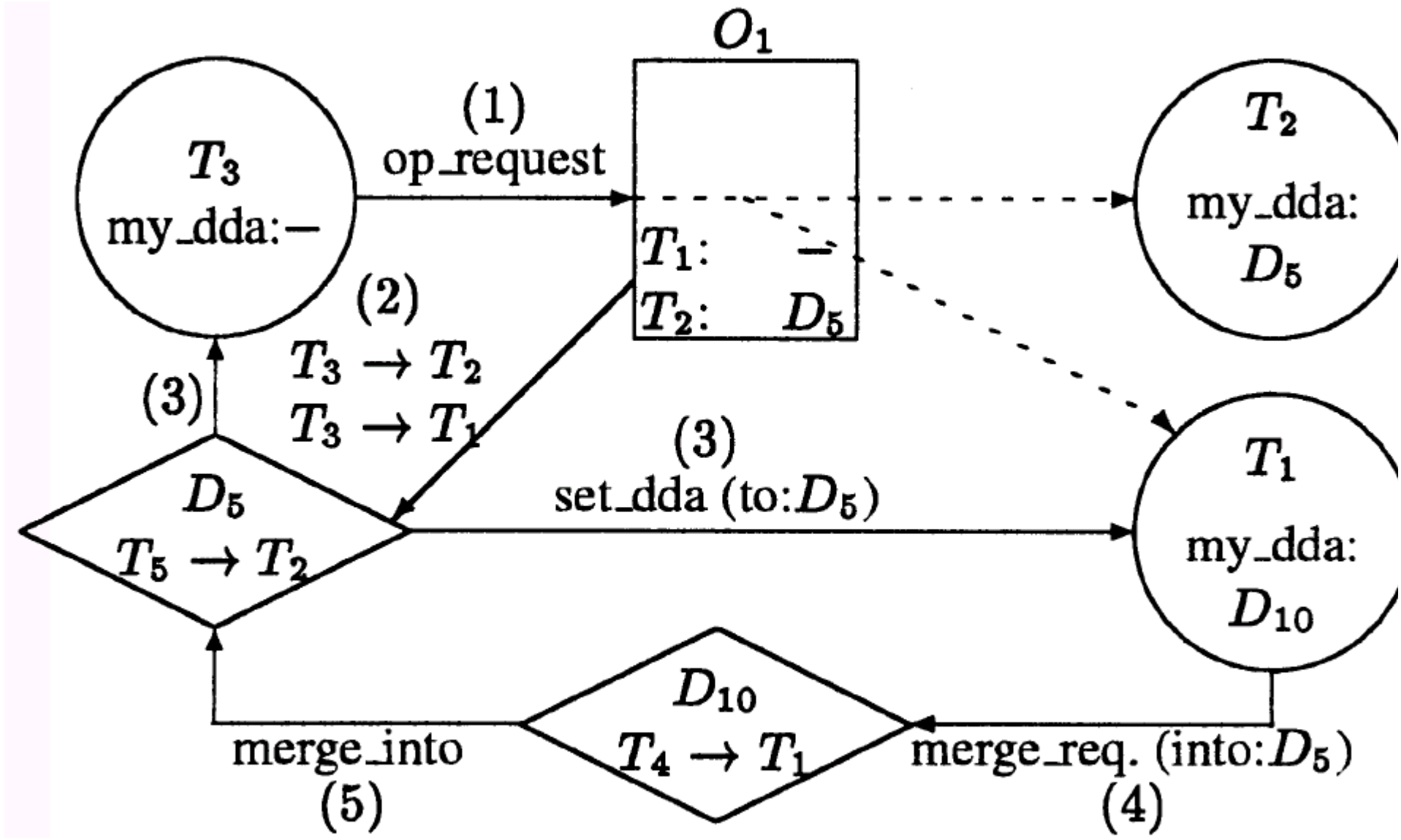


Fig. 1. Creation of a DDA

Verschmelzung zweier DDAs wird von TA (hier T1) initiiert



Zeitlicher Ablauf einer DDA-Verschmelzung

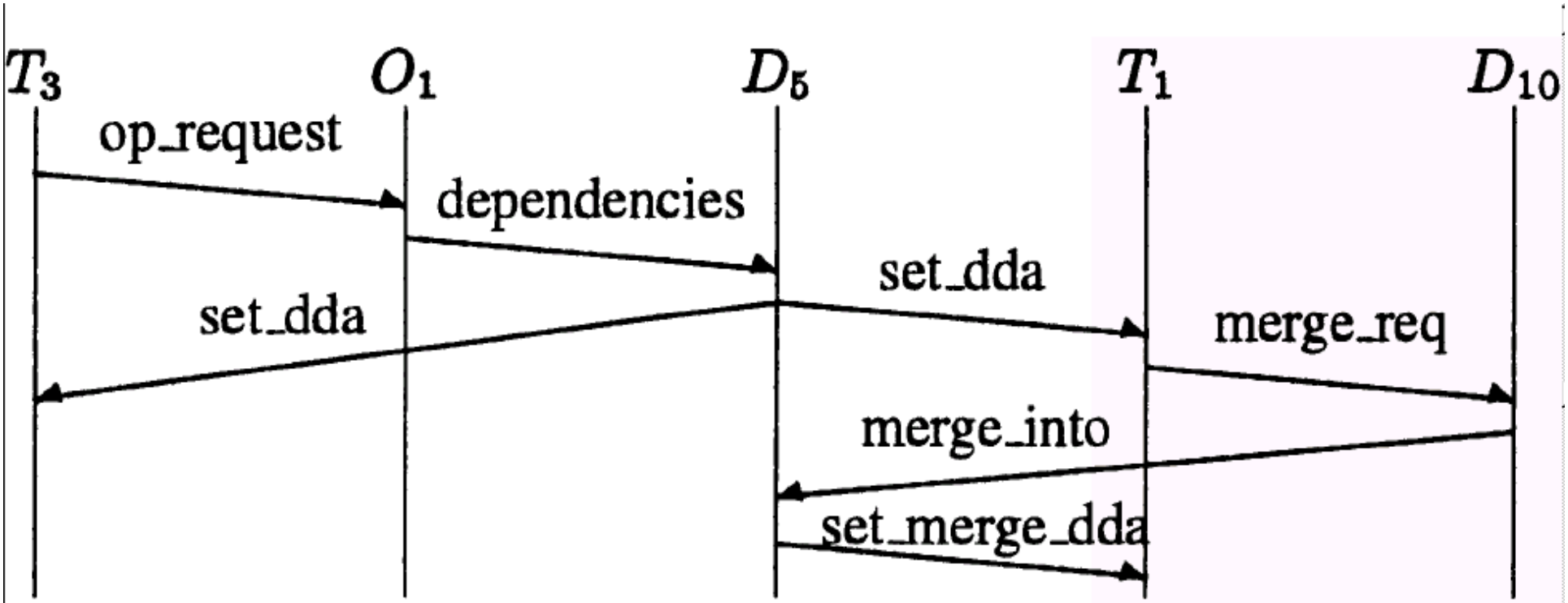


Fig. 4. Messages sent on a merge initiation by a transaction

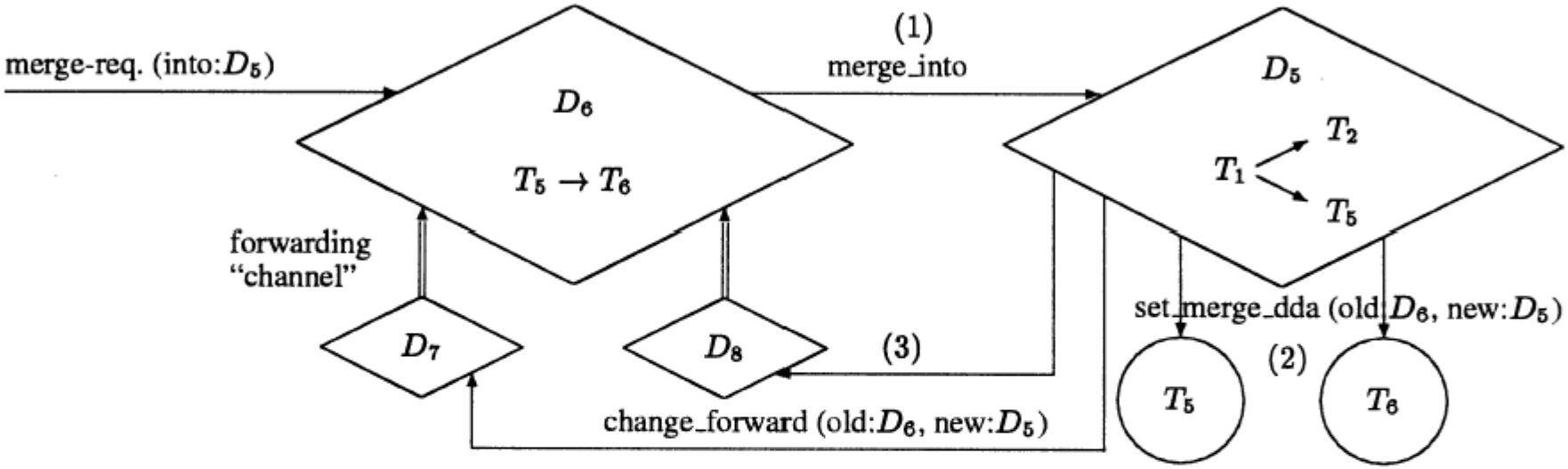


Fig. 2. Merging of two DDAs

Korrektheits- und Effizienz-überlegungen (mehr dazu im Seminar)

- Der Verschmelzungsprozess terminiert auf jeden Fall, da jüngere DDAs immer in ältere übergehen (irgendwann ist man beim ältesten DDA angekommen)
- Irgendwann wird die gesamte Zusammenhangskomponente des Wartegraphen (in der sich der Zyklus befindet) in einem DDA landen, der den Zyklus dann entdeckt
- DDAs werden nicht mehr aufgespalten, auch wenn die Zusammenhangskomponente „zerfällt". Dann sind sie halt für mehrere Zusammenhangskomponenten zuständig
- Es werden keine Phantom-Deadlocks erkannt (global state detection)
- DDAs bilden sich neu an den Aktivitätszentren
  - wenn die wandern (z.B. Tageszeit-abhängig), dann bilden sich dort neue DDAs
  - Reduktion der Kommunikationskosten

Performance: Deadlock-Erkennung