

# Verteilte Systeme

Verteilte Transaktionen und  
Nebenläufigkeitskontrolle

# Transaktionen



# Motivation

- ◆ Wir haben bereits das Konzept des **gegenseitigen Ausschlusses** bei Zugriff auf kritische Abschnitte betrachtet.
- ◆ **Ziel:** *eine* bestimmte *Ressource* soll nur von *einem Client* zur selben Zeit benutzt werden
- ◆ Ganz generell haben **Transaktionen** dasselbe **Ziel:** Schutz von Ressourcen vor gleichzeitigem Zugriff.
- ◆ Transaktionen gehen jedoch noch wesentlich **weiter:**
  - **Automatisches Erzwingen** der Konsistenzwahrung
  - Es ist möglich, auf *mehrere* Ressourcen in einer *einzigsten atomaren* Operation zuzugreifen, d.h. **Umgang mit verschiedenen kritischen Abschnitten**
  - Diverse Arten von Fehler können abgefangen werden, so daß Prozesse in den *Zustand vor Beginn* der Ausführung einer Transaktion zurückgesetzt werden, d.h. **Konsistenzwahrung auch bei Fehlern und Ausfällen.**

# Beispiel

- ◆ Wir wollen die in diesem Kapitel vermittelten Konzepte anhand **eines durchgängigen Beispiel** erläutern.
- ◆ Es gibt zwei Arten von Ressourcen:
  - **Account-Objekte** (Konto-Objekt), die *Abbuchungen* und *Einzahlungen* gestatten sowie *Abfragen* und *Änderungen* des Kontostands
  - **Branch-Objekte** (Bankfiliale-Objekt), die eine *Filiale repräsentieren* und es gestatten, Konten zu erzeugen, Konten zu suchen und den Gesamtstand aller Konten in dieser Filiale abzufragen.

# Beispiel: Methoden

## Methoden des **Account-Objekts**

- ◆ **deposit (amount)**  
Zahlt `amount` (Betrag) auf Konto ein
- ◆ **withdraw (amount)**  
Hebt `amount` von Konto ab
- ◆ **getBalance () -> amount**  
Gibt Kontostand zurück
- ◆ **setBalance (amount)**  
Setzt Kontostand auf `amount`

## Methoden des **Branch-Objekts**

- ◆ **create (name) -> account**  
Erzeugt neues Konto `name`
- ◆ **lookUp (name) -> account**  
Gibt Verweis auf Konto `name` zurück
- ◆ **branchTotal () -> amount**  
Gibt Gesamtsumme aller Konten zurück

# Einfache Synchronisation

(ohne Transaktion)

- ◆ Im Account-Objekt müssen die Operationen `deposit()` und `withdraw()` **atomar ausgeführt** werden, d.h. sie dürfen in der Ausführung nicht unterbrochen werden.
- ◆ In **Java** läßt sich das auf einfache Weise unter Verwendung des Schlüsselwortes **synchronized** erreichen:  

```
public synchronized void deposit(...);
```
- ◆ **Ergebnis:** wenn mehrere Threads gleichzeitig dieselbe bzw. eine andere synchronisierte Methode dieses Objekts benutzen wollen, wird nur ein Thread zugelassen; die anderen werden blockiert.

# Fehlermodell für Transaktionen

- ◆ B.W. Lampson führte 1981 ein Fehlermodell ein, das heute als **Grundlage aller Transaktionsalgorithmen** verwendet wird. Ein Algorithmus muß demnach folgende (vorhersehbare) Fehler behandeln können:
  - **Fehler beim Schreibzugriff** auf permanenten Speicher
  - **Server-Absturz**
  - Beliebige **Verzögerungen** bei der **Nachrichtenübertragung**
- ◆ Im **Katastrophenfall keine Aussage** über die Folgen.
  - Schreiben in falschen Block ist Katastrophe
  - Gefälschte Nachrichten sowie nicht erkannte fehlerhafte Nachrichten sind Katastrophe

# Transaktionen

- ◆ **Transaktionen** bestehen aus einer **Folge von Operationen** (Anfragen an Server), für die bestimmte Eigenschaften gelten – die ***ACID-Eigenschaften***.
- ◆ **Beispiel** für eine Transaktion in der Bankanwendung:
  - eine Kunde will **verschiedene Operationen** auf drei Konten *a*, *b* und *c* ausführen.
  - Die Operationen sollen **ohne Unterbrechung** ausgeführt werden.
  - **Transaction T:**

```
a.withdraw(100);  
b.deposit(100);  
c.withdraw(200);  
b.deposit(200);
```



# Transaktionen: ACID-Eigenschaft

- ◆ **ACID** ist ein von T. Härder und A. Reuter 1983 vorgeschlagenes **Acronym**.

- ◆ **Bedeutung:**

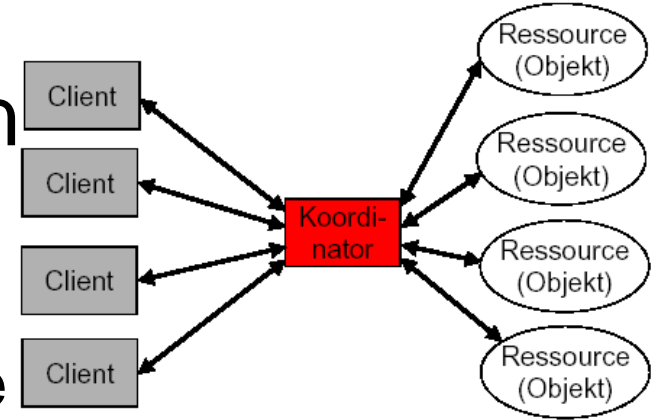
- **Atomicity:** *Alles-oder-Nichts* Eigenschaft: Die Transaktion wird entweder vollständig ausgeführt oder hinterläßt keine Wirkung.
- **Consistency:** eine Transaktion überführt das System von einem konsistenten Zustand in einen konsistenten Zustand.
- **Isolation:** jede Transaktion muß von der Ausführung anderer Transaktionen unabhängig bleiben (Serialisierbarkeit).
- **Durability:** Die Änderungen einer beendeten und bestätigten (*committed*) Transaktion können weder verloren gehen noch rückgängig gemacht werden.

# ACID (1): Atomarität und Dauerhaftigkeit

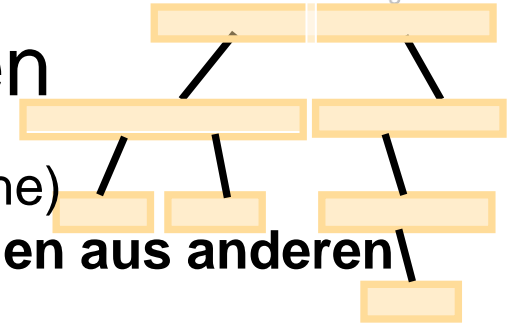
- ◆ *Atomicity* und *Durability*
  - ist **gefährdet** durch eine **fehlerhafte Umgebung** und
  - wird erreicht durch die Verwendung **wiederherstellbarer Objekte**.
- ◆ Wenn ein Server-Prozeß während der Abarbeitung einer Transaktion abstürzt und dann ein neuer Prozeß gestartet wird, dann muß dieser den alten Zustand der Objekte wieder laden können.
- ◆ Wenn die Transaktion abgeschlossen ist, muß das Objekt den neuen Zustand repräsentieren und abgespeichert werden.

# Implementierung: Methoden

- ◆ **openTransaction()**  $\rightarrow$  `trans`;  
startet eine neue Transaktion und gibt die eindeutige TID `trans` zurück. Diese ID wird in den anderen Operationen der Transaktion verwendet.
- ◆ **closeTransaction(trans)**  $\rightarrow$  `(commit, abort)`;  
beendet eine Transaktion: der Rückgabewert `commit` zeigt an, daß die Transaktion festgeschrieben wird; der Rückgabewert `abort` zeigt an, daß sie abgebrochen wurde.
- ◆ **abortTransaction(trans)**;  
bricht die Transaktion `trans` ab.



# Geschachtelte Transaktionen



- ◆ Geschachtelte Transaktionen erweitern das bisherige (flache) Transaktionsmodell, indem sie gestatten, daß **Transaktionen aus anderen Transaktionen zusammengesetzt** sind.
- ◆ Die Transaktion auf der höchsten Ebene wird als **toplevel transaction** bezeichnet, die anderen als **subtransactions**.
- ◆ **Zusätzliche Nebenläufigkeit:**
  - *Subtransactions* auf der selben Hierarchieebene können nebenläufig ausgeführt werden
  - Bankenbeispiel: die Operation `branchTotal()` muß für sämtliche Konten die Methode `getBalance()` aufrufen. Man könnte jeden dieser Aufrufe als Untertransaktion starten
- ◆ **Unabhängiges Commit oder Abort:**
  - Dadurch werden Transaktionen potentiell robuster (hängt von der Anwendung ab)
  - Die Elterntransaktion muß jeweils entscheiden, welche Folge ein Abort der Untertransaktion haben soll.

# Geschachtelte Transaktion: Regeln

1. Eine Transaktion darf nur abgeschlossen werden, *wenn* ihre Untertransaktionen abgeschlossen sind.
2. *Wenn* eine Untertransaktion abschließt, entscheidet sie unabhängig, entweder provisorisch festzuschreiben (committed) oder endgültig abzurechnen.
3. *Wenn* eine Elterntransaktion abbricht, werden auch alle Subtransaktionen abgebrochen.
4. *Wenn* eine Subtransaktion abbricht, entscheidet die Elterntransaktion, was weiter geschieht.
5. *Wenn* eine Elterntransaktion festgeschrieben ist, dann können alle provisorisch festgeschriebenen Untertransaktionen ebenfalls (permanent) festgeschrieben werden.

# Transaktionen: ACID-Eigenschaft

- ◆ **ACID** ist ein von T. Härder und A. Reuter 1983 vorgeschlagenes **Acronym**.
- ◆ **Bedeutung:**
  - **Atomicity:** *Alles-oder-Nichts* Eigenschaft: Die Transaktion wird entweder vollständig ausgeführt oder hinterläßt keine Wirkung.
  - **Consistency:** eine Transaktion überführt das System von einem konsistenten Zustand in einen konsistenten Zustand.
  - **Isolation:** jede Transaktion muß von der Ausführung anderer Transaktionen unabhängig bleiben (Serialisierbarkeit).
  - **Durability:** Die Änderungen einer beendeten und bestätigten (*committed*) Transaktion können weder verloren gehen noch rückgängig gemacht werden.

## ACID (2): Isolierung und Konsistenz

- ◆ Die Operationen aller Transaktionen müssen so synchronisiert werden, daß *Isolation* und *Consistency* erreicht werden.
- ◆ Die beiden Eigenschaften sind **gefährdet** durch **Interferenzen nebenläufiger Transaktionen**.
- ◆ **Einfachste Variante: Serielle Ausführung** der Transaktionen
- ◆ Ist nicht wünschenswert, da die Performance des Servers sehr schlecht wäre und mögliche nebenläufige Ausführungen von Transaktionen würden nicht berücksichtigt.
- ◆ Zur **Maximierung der Leistung** wird deshalb versucht, die Nebenläufigkeit zu maximieren: Zwei Transaktionen dürfen nebenläufig ausgeführt werden, wenn diese Ausführung denselben Effekt hat wie die sequentielle Ausführung – die Ausführung heißt dann **seriell äquivalent** (Serialisierbar).

# Nebenläufigkeitskontrolle





# Nebenläufigkeitskontrolle

- ◆ **Nebenläufigkeitskontrolle** (Concurrency Control) ist die wichtigste Aufgabe des Transaktionsmanagers, um die Nebenläufigkeit maximieren zu können.
- ◆ **Aufgabe:** finde möglichst nebenläufige Ablaufpläne für Transaktionen, ohne das serielle Äquivalentskriterium zu verletzen.
- ◆ Es geht also im wesentlichen darum, miteinander in Konflikt stehende Operationen **korrekt einzuplanen**.

# Interferenzen nebenläufiger Transaktionen

Problem der  
verlorenen Updates

## Transaction T:

```
balance = b.getBalance();
b.setBalance(balance*1.1);
a.withdraw(balance/10)
```

```
$200 balance = b.getBalance();
```

```
$220 b.setBalance(balance*1.1);
```

```
$20 a.withdraw(balance/10);
```

## Transaction U:

```
balance = b.getBalance();
b.setBalance(balance*1.1);
c.withdraw(balance/10);
```

```
balance = b.getBalance(); $200
```

```
b.setBalance(balance*1.1); $220
```

```
c.withdraw(balance/10); $20
```

# Korrekte Interferenz

## Transaction T:

```
balance = b.getBalance();
b.setBalance(balance*1.1);
a.withdraw(balance/10)
```

**\$200** balance = b.getBalance();

**\$220** b.setBalance(balance\*1.1);

**\$20** a.withdraw(balance/10);

## Transaction U:

```
balance = b.getBalance();
b.setBalance(balance*1.1);
c.withdraw(balance/10);
```

balance = b.getBalance(); **\$220**

b.setBalance(balance\*1.1); **\$242**

c.withdraw(balance/10); **\$22**

# Interferenzen nebenläufiger Transaktionen

```
$200 == a.getBalance()
$200 == b.getBalance()
```

Problem der  
inkonsistenten Abrufe

## Transaction v:

```
a.withdraw(100)
b.deposit(100)
```

## Transaction w:

```
aBranch.branchTotal()
```

**\$100** a.withdraw(100);

**\$300** b.deposit(100)

```
total = a.getBalance()
total = total+b.getBalance()
total = total+c.getBalance()
•
•
•
```

**\$100**  
**\$300**

# Korrekte Interferenz

```
$200 == a.getBalance()
```

```
$200 == b.getBalance()
```

## Transaction v:

```
a.withdraw(100)  
b.deposit(100)
```

## Transaction w:

```
aBranch.branchTotal()
```

```
$100 a.withdraw(100);  
$300 b.deposit(100)
```

```
total = a.getBalance() $100  
total = total+b.getBalance() $400  
total = total+c.getBalance()  
•  
•  
•
```

# Wiederherstellung nach Abbrüchen

## Transaction T:

```
balance = a.getBalance();
a.setBalance(balance +10);
```

```
$100 balance = a.getBalance();
$110 a.setBalance(balance +10);
```

Überlagernde Abbrüche

```
abort transaction;
```

```
a.setBalance(105);
```

```
$105 a.setBalance(105);
```

```
abort/commit transaction;
```

## Transaction U:

```
balance = a.getBalance();
a.setBalance(balance +20);
```

Dirty Read

```
balance = a.getBalance(); $110
a.setBalance(balance +20); $130
commit transaction;
```

Nur vorläufiges commit!

```
a.setBalance(110);
```

Vorzeitiges Schreiben

```
a.setBalance(110); $110
commit/abort transaction;
```

# Konflikte zwischen Operationen

- ◆ **Was bedeutet** es, wenn zwei Operationen zueinander im **Konflikt** stehen?
  - ⇒ Ihr kombinierter Effekt hängt von der Reihenfolge ab, in der sie ausgeführt werden.
- ◆ Mit diesem Begriff läßt sich **serielle Äquivalenz** formaler definieren:
  - ⇒ Zwei Transaktionen sind genau dann **seriell äquivalent**, wenn alle Paare von miteinander in Konflikt stehenden Operationen der beiden Transaktionen auf allen betroffenen Objekten in derselben Reihenfolge ausgeführt werden.

# Beispiel: Konfliktregeln für read & write

<i>Operationen unterschiedlicher Transaktionen</i>		<i>Konflikt</i>	<i>Grund</i>
<i>read</i>	<i>read</i>	<i>Nein</i>	<i>Weil die Wirkung eines Paares von read-Operationen nicht von der Ausführungsreihenfolge abhängig ist.</i>
<i>read</i>	<i>write</i>	<i>Ja</i>	<i>Weil die Wirkung einer read- und einer write-Operation von der Ausführungsreihenfolge abhängig ist.</i>
<i>write</i>	<i>write</i>	<i>Ja</i>	<i>Weil die Wirkung eines Paares von write-Operationen von der Ausführungsreihenfolge abhängig ist.</i>



# Beispiel: Serielle Äquivalenz

- ◆ Gegeben seien zwei Transaktionen wie folgt
  - T: `x=read(i); write(i,10); write(j,20);`
  - U: `y=read(j); write(j,30); z=read(i);`
- ◆ Ist der folgende Ablauf seriell äquivalent?  
Warum bzw. warum nicht?

Transaction T:	Transaction U:
<pre>x = read(i) write(i, 10)  write(j, 20)</pre>	<pre>y = read(j) write(j, 30)  z = read(i)</pre>

Der Zugriff auf einzelne Objekte ist serialisiert:

- T greift auf i vor U zu
- U greift auf j vor T zu

Aber: **Nicht äquivalent!**

Korrekt wäre zB:

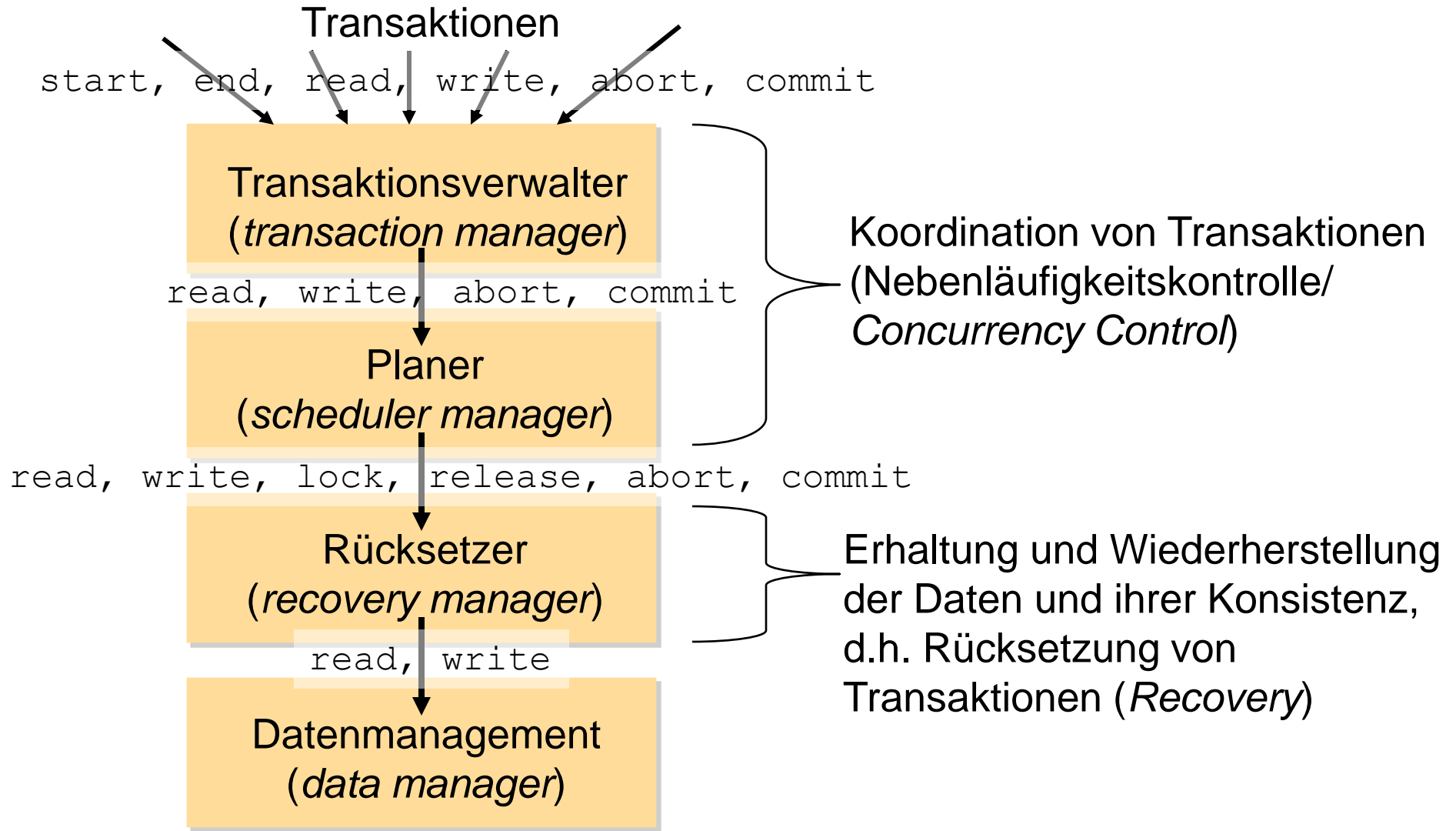
- T greift vor U auf i zu und
- T greift vor U auf j zu

# Beispiel: Serielle Äquivalenz

	x	y	z	i	j
Ausgangswerte	-	-	-	*	**
T vor U	*	20	10	10	20 30
U vor T	*	**	*	10	30 20
Wie im Beispiel	*	**	10	10	30 20

- ◆ **Beispiel ist nicht seriell Äquivalent:**  
weder zu „T vor U“ noch zu „U vor T“
- ◆ Serielle Äquivalenz **entscheidet nicht** über „T vor U“ oder „U vor T“:  
serielle Äquivalenz sorgt dafür, daß **eines von beiden** im Ergebnis erzielt wird!

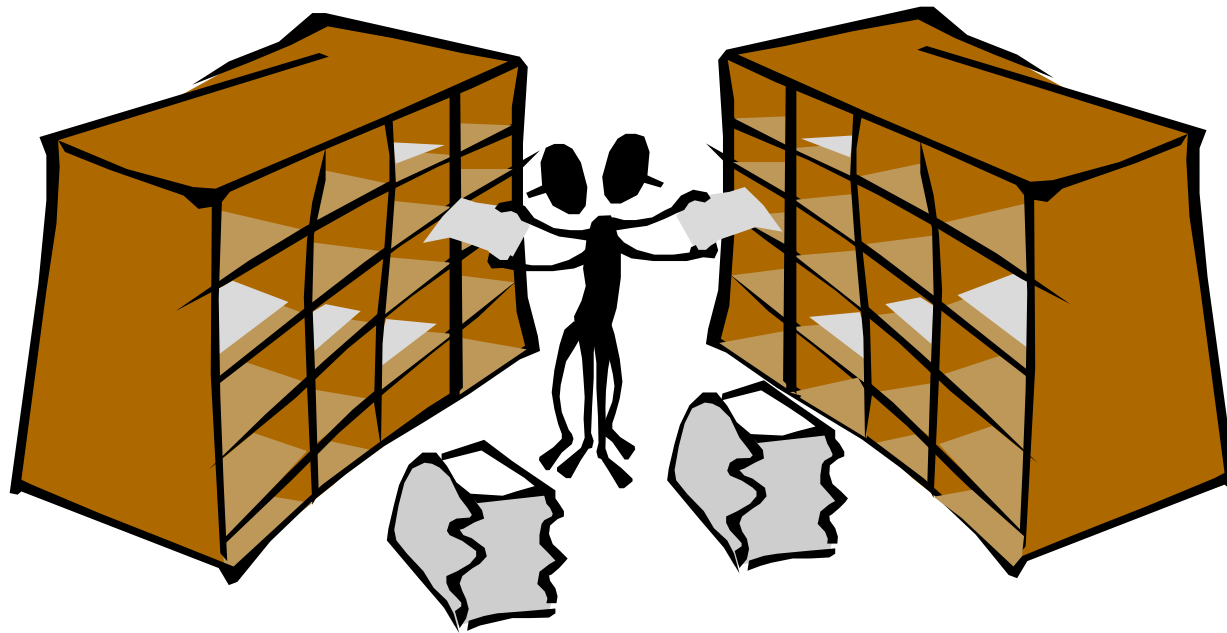
# Architektur eines Transaktionssystems



# Nebenläufigkeitskontrolle: Algorithmen

- ◆ Es geht also nun darum, einen **Ablaufplan** für zueinander in Konflikt stehende Operationen zu finden.
- ◆ Drei gängige Ansätze:
  - **Sperren** (*Locking*): pessimistische Systeme
  - **optimistische Nebenläufigkeitskontrolle** (*Optimistic concurrency control*)  
(Auswertung ganze Transaktion)
  - **Zeitstempel-Reihenfolge** (*Timestamp ordering*)  
(Auswertung einzelne Operation)

# Verteilte Transaktionen

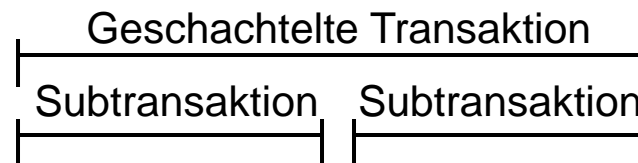


# Verteilte Transaktionen

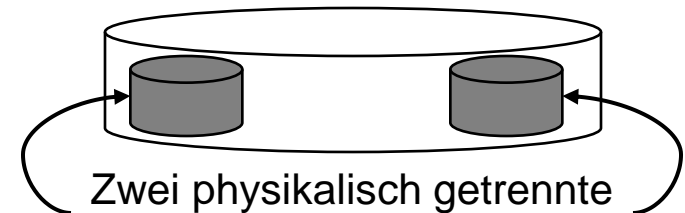
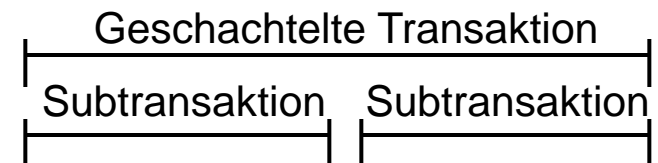
- ◆ Bisher haben wir Transaktionen betrachtet, die auf **Objekte auf einem einzigen Server (Transaktionssystem)** zugreifen (**rechts**).
- ◆ Oft jedoch werden die **Objekte** bzw. Operationen **über mehrere Server** und damit mehrere Transaktionssysteme **verteilt (links)**.
- ◆ **Zwei Arten** von Diensten:
  - **Nicht-transaktionale** Dienste bieten keine Unterstützung für verteilte Transaktionen an, Dienstleistungen werden ohne besondere Garantien erbracht
  - **Transaktionale** Dienste bieten Unterstützung für verteilte Transaktionen an, unterliegen bei entsprechender Abstimmung globalen ACID-Garantien

- ◆ **Beispiel:** Buchung einer Reise

- Flug
- Hotel
- Bustransfers
- Tagesausflüge



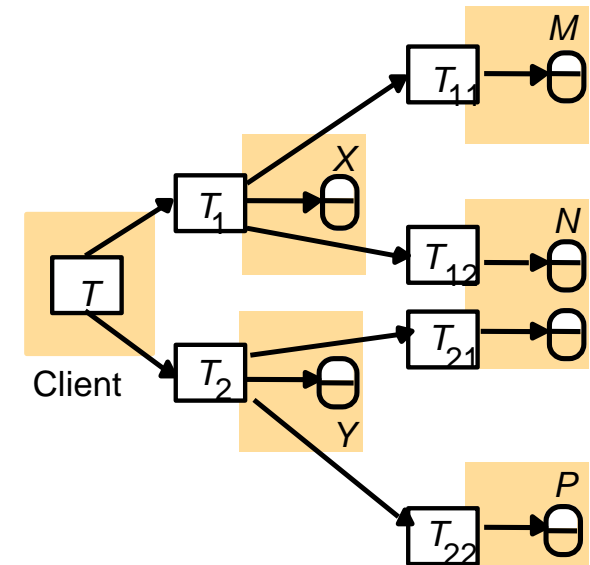
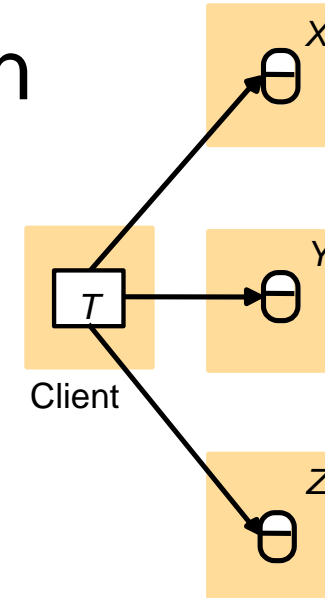
Zwei unterschiedliche und unabhängige Datenbanken



Zwei physisch getrennte Teile einer Datenbank

# Verteilte Transaktionen

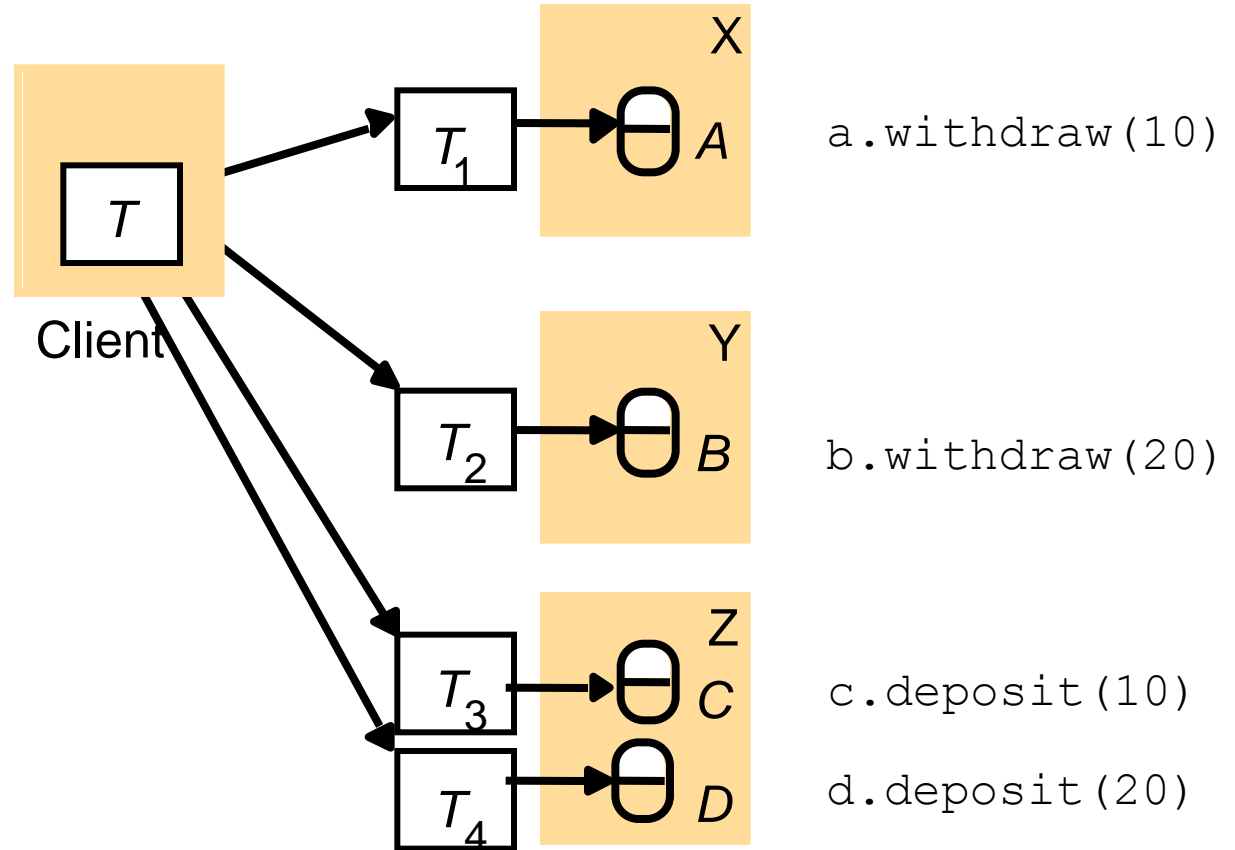
- ◆ **Verteilte Transaktionen** können wiederum **flach** (oben) oder **geschachtelt** (unten) sein.
- ◆ Bei einer **flachen** verteilten Transaktion greift der Client **nacheinander** auf die beteiligten Server zu.
- ◆ Bei der Verwendung von **Subtransaktionen** kann von der **Nebenläufigkeit** der verschiedenen Server Gebrauch gemacht werden.



# Verteilte Transaktionen: Beispiel

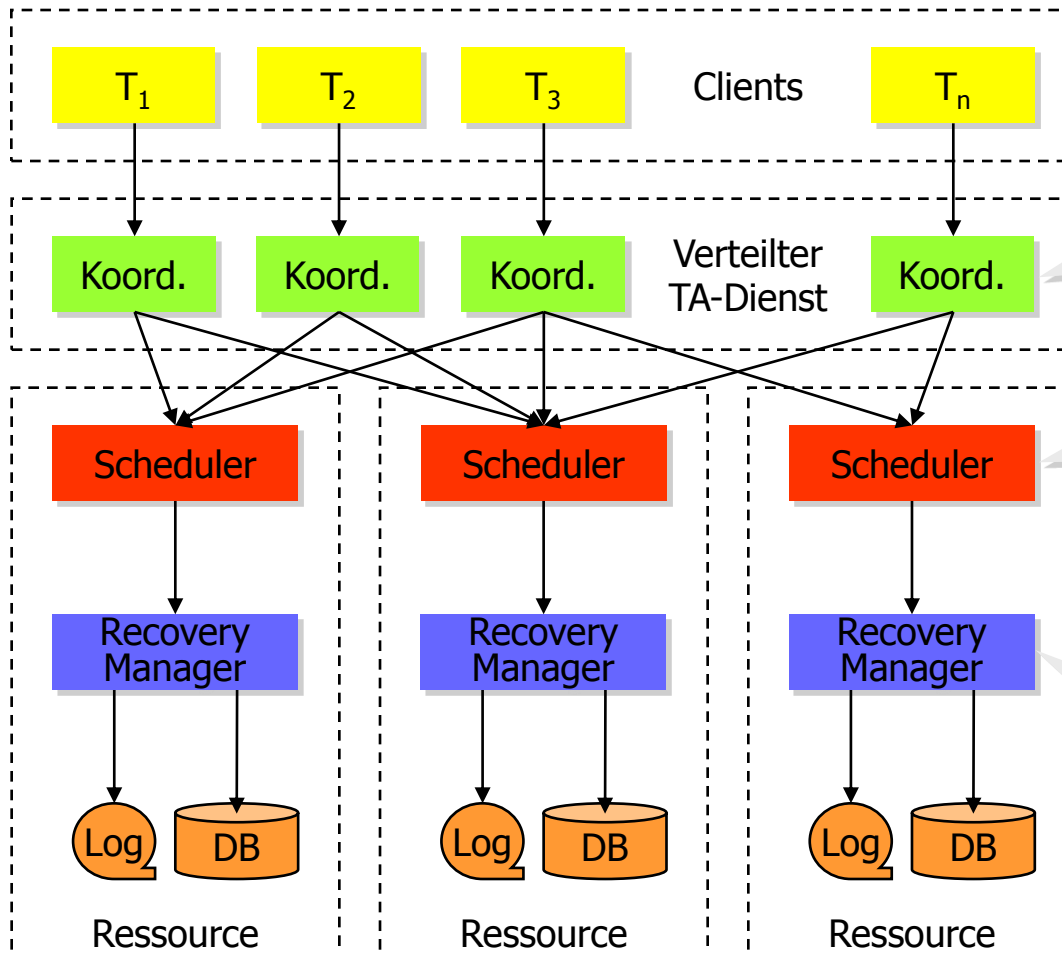
```

T = openTransaction
  openSubTransaction
    a.withdraw(10);
  openSubTransaction
    b.withdraw(20);
  openSubTransaction
    c.deposit(10);
  openSubTransaction
    d.deposit(20);
closeTransaction
    
```





# Beispielarchitektur eines Transaktionssystems



## Verteilte TA-Koordinatoren

- Vergabe von TA-Kennungen (TIDs)
- Zuordnung von TIDs zu Operationen
- Koordination von Festschreiben, Rücksetzen und Wiederanlauf

## Lokale Scheduler

- Lokale Durchführung des 2PS-Prot.
- Lokale Sperrenverwaltung
- Lokale Warteschlangenverwaltung
- Lokale Verklemmungsüberwachung

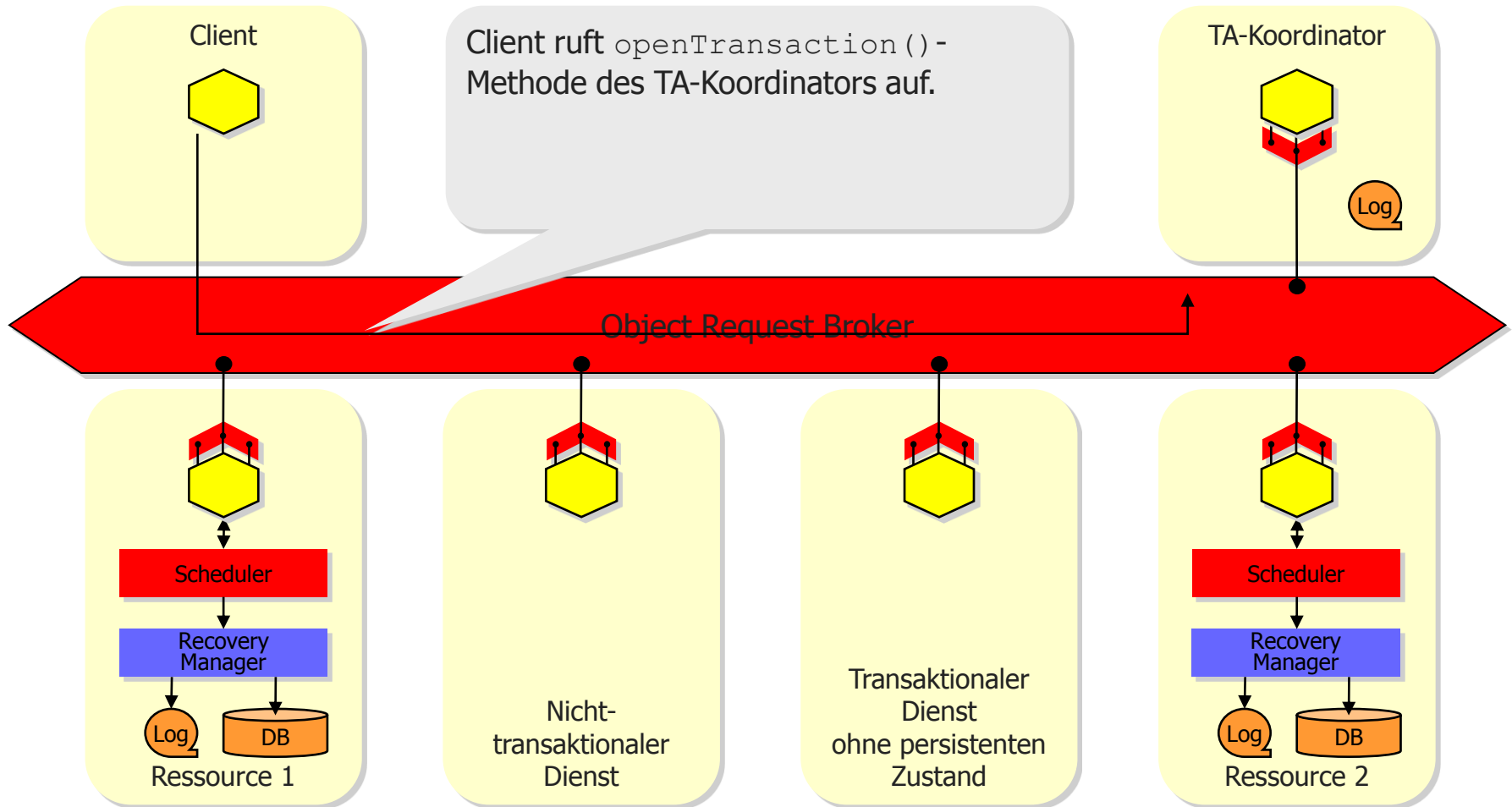
## Lokale Recovery Manager

- Führung der lokalen Protokolldatei
- Lokales Festschreiben von Transakt.
- Lokales Rücksetzen von Transakt.
- Lokaler Wiederanlauf nach Crash

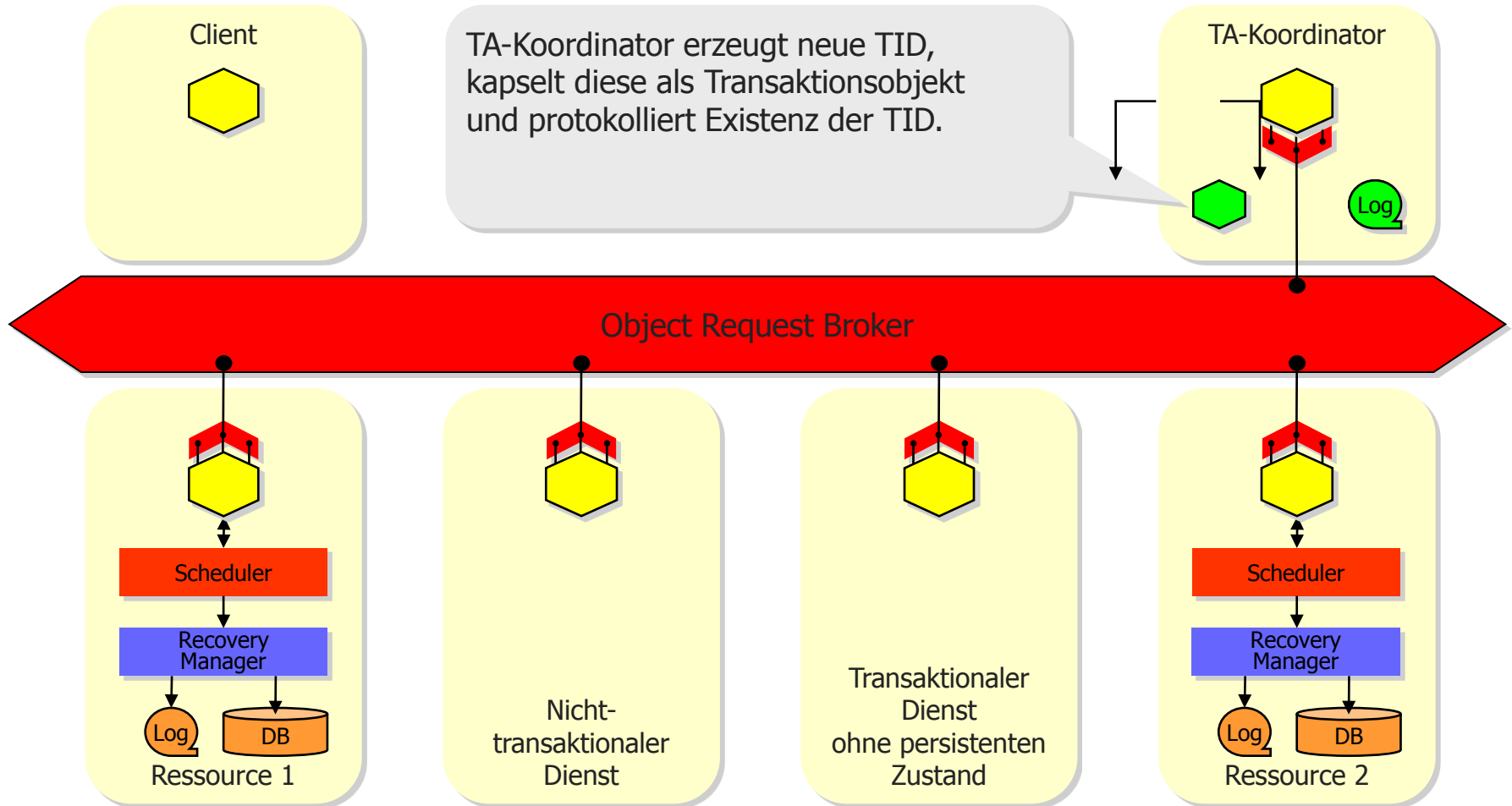
# Koordination

- ◆ Zur korrekten Abwicklung der Transaktion **müssen** die Server ihre Aktionen **koordinieren**.
- ◆ Dazu wird für jede Transaktion ein **Koordinator** (Transaktionsmanager) **bestimmt** (typischerweise in einem der Server).
- ◆ Zum **Start** der Transaktion sendet der Client ein `openTransaction()` an den Koordinator.
  - Dieser startet die Transaktion und liefert eine **eindeutige Transaktions-ID** (z.B. IP-Nummer + lokale TID) zurück.
  - Der **Koordinator entscheidet** am Ende, ob eine verteilte Transaktion **abgebrochen** oder **korrekt** beendet wird.
- ◆ Er **kennt alle** Teilnehmer, die wiederum **alle ihn** kennen.
- ◆ **Neue Teilnehmer** melden sich mit der Methode `join(Transaktion, Verweis auf Teilnehmer)` an.
- ◆ Zur Kooperation wird ein **Commit-Protokoll** (Festschreibungsprotokoll) verwendet.

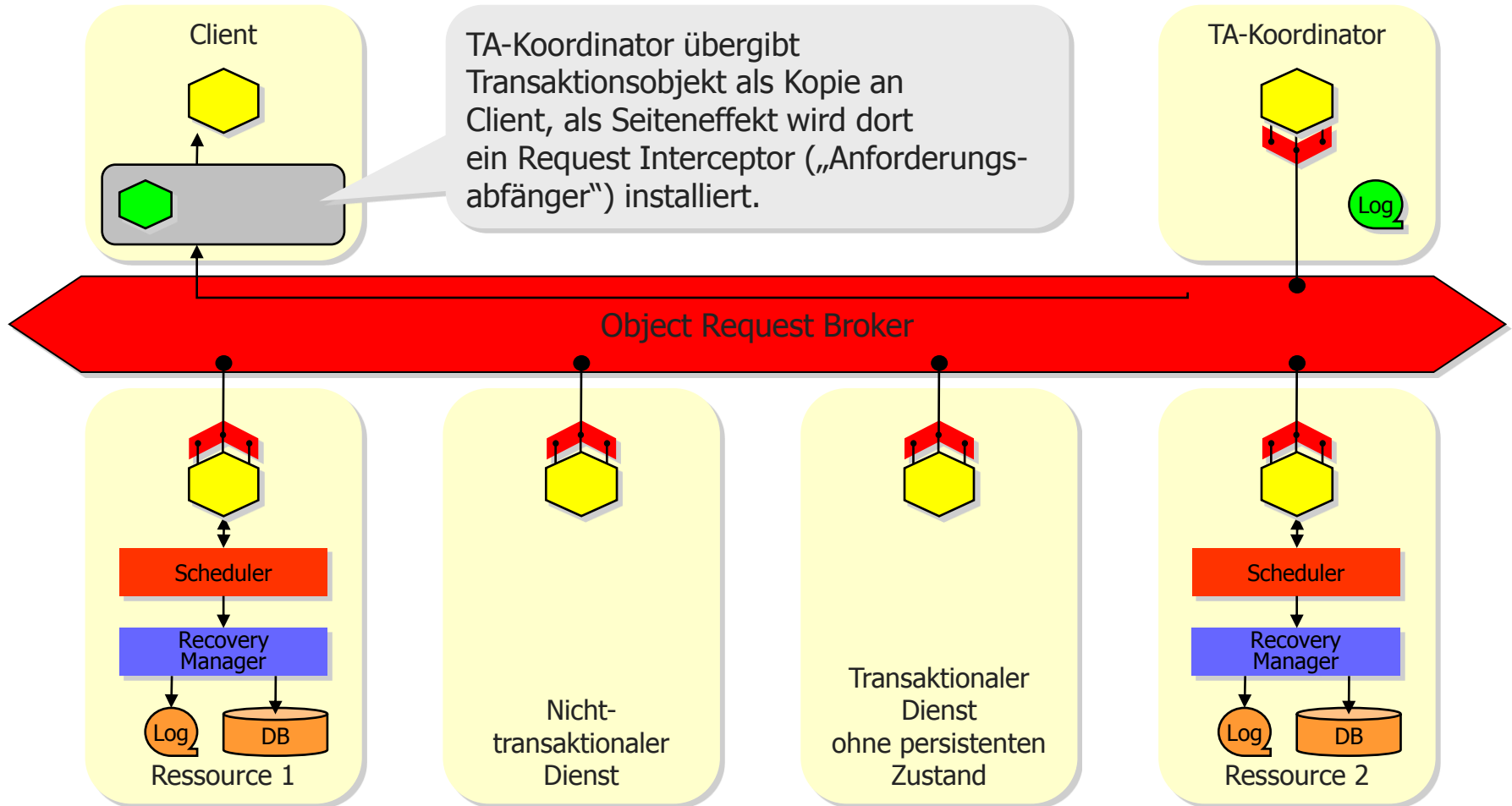
# Globale TID-Erzeugung und -Weiterleitung



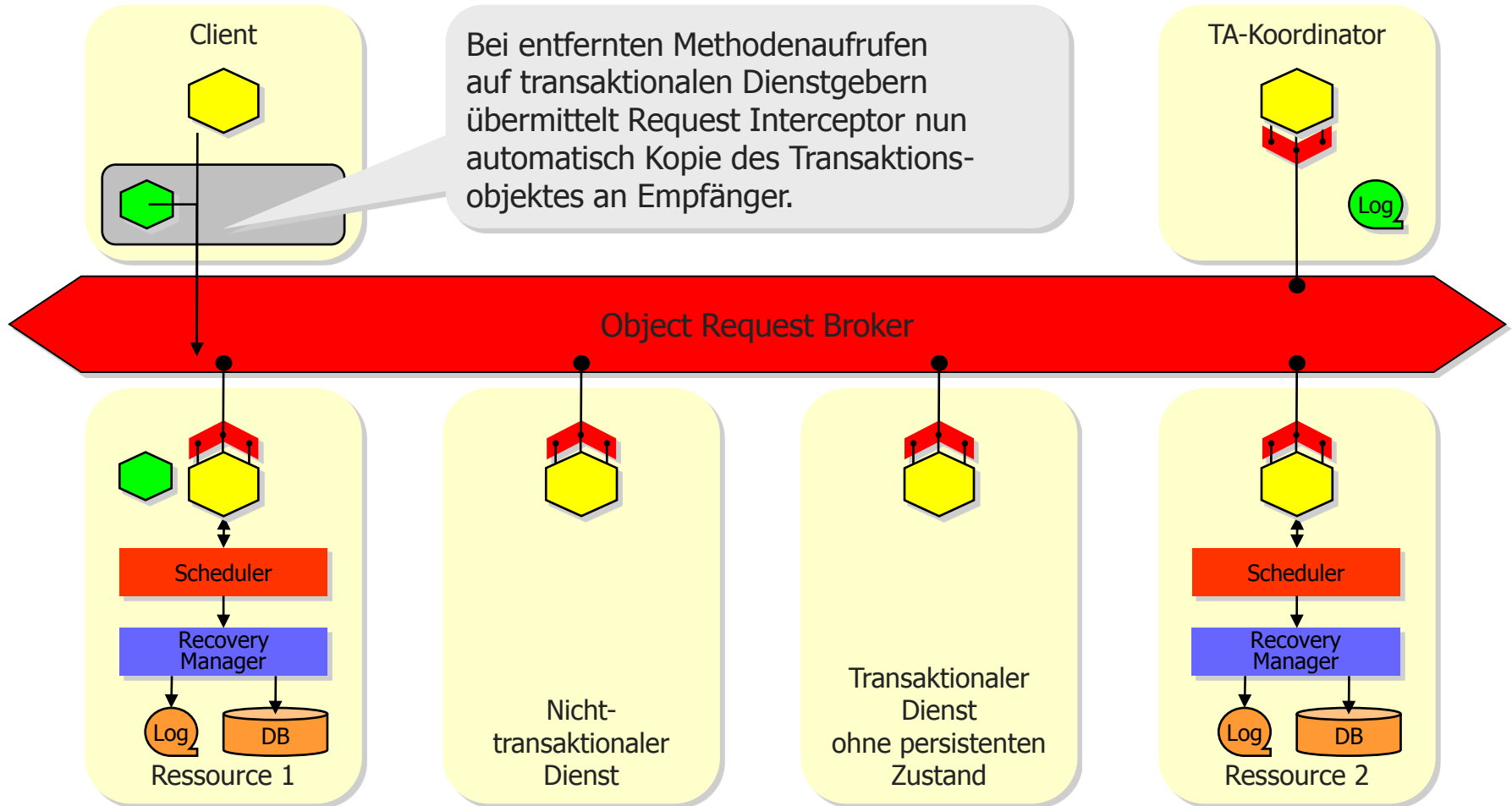
# Globale TID-Erzeugung und -Weiterleitung



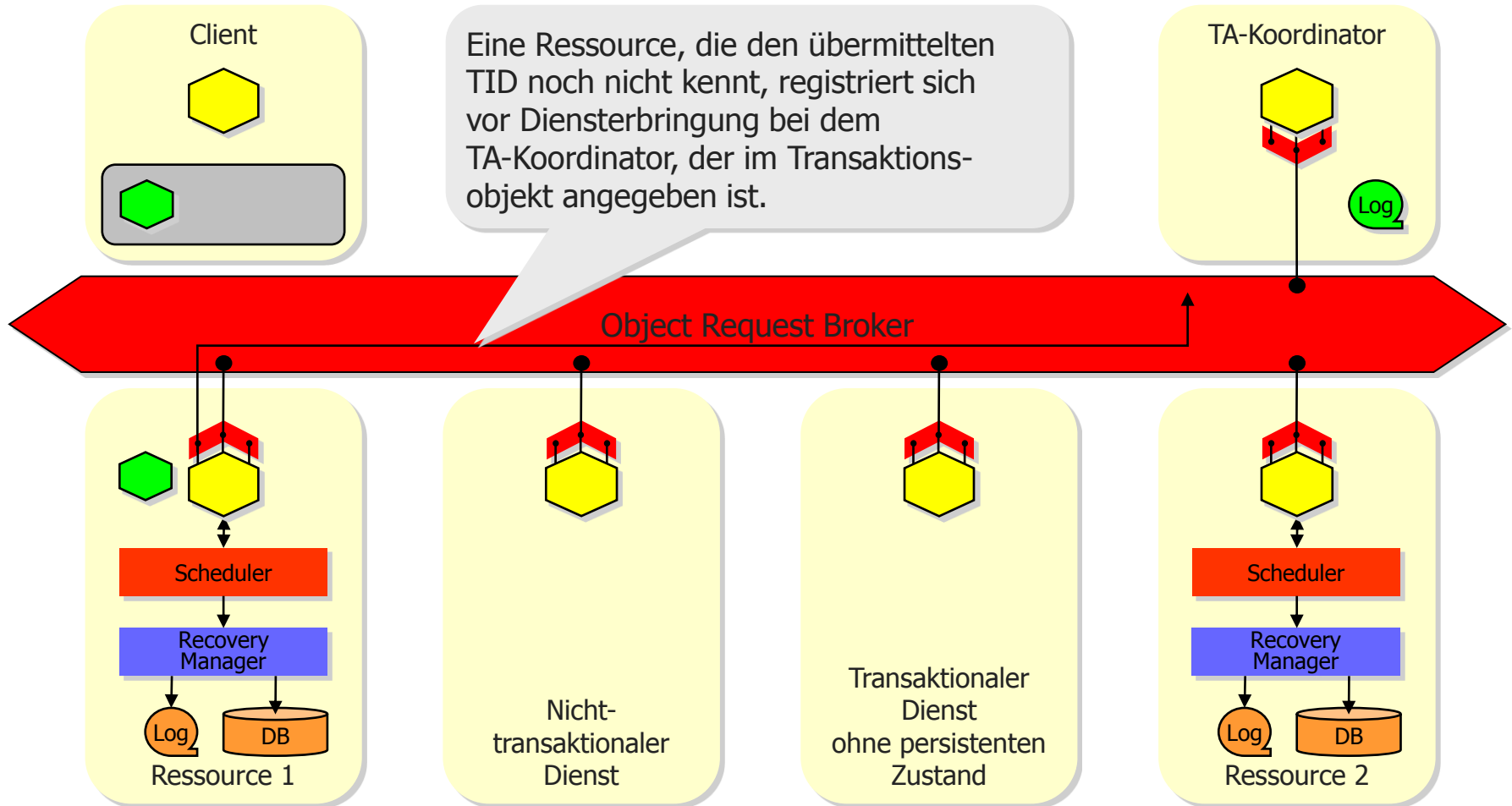
# Globale TID-Erzeugung und -Weiterleitung



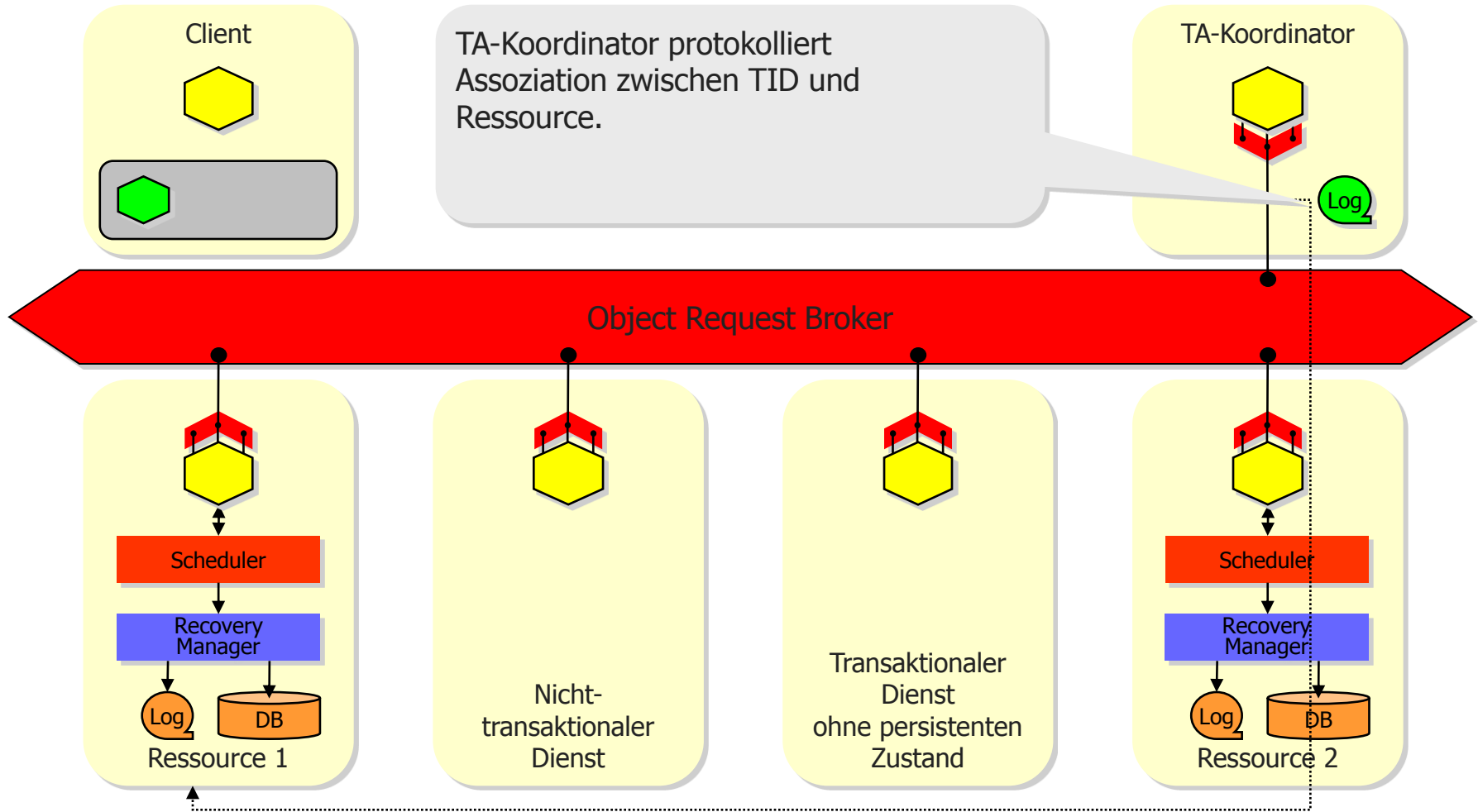
# Globale TID-Erzeugung und -Weiterleitung



# Globale TID-Erzeugung und -Weiterleitung

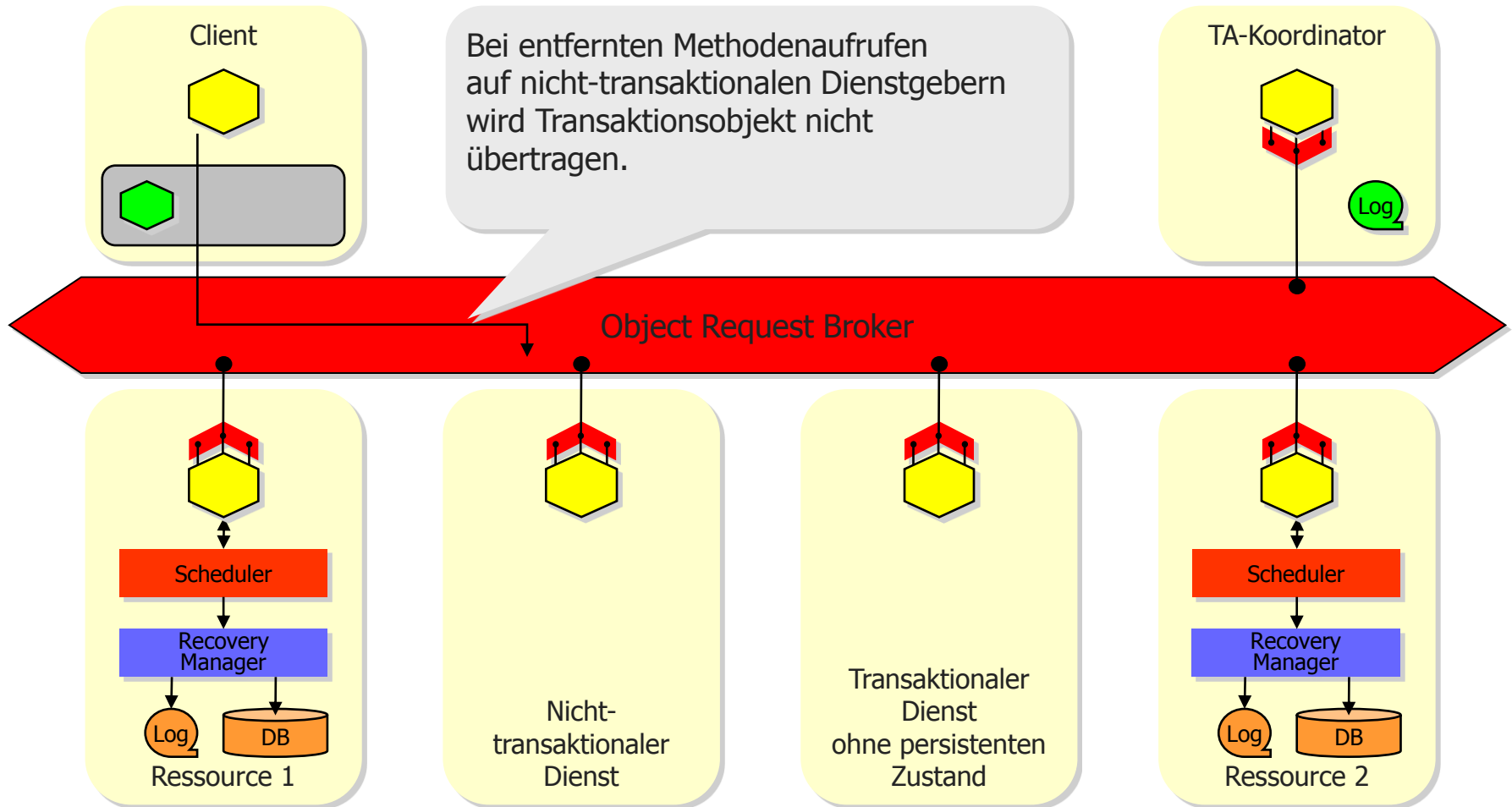


# Globale TID-Erzeugung und -Weiterleitung

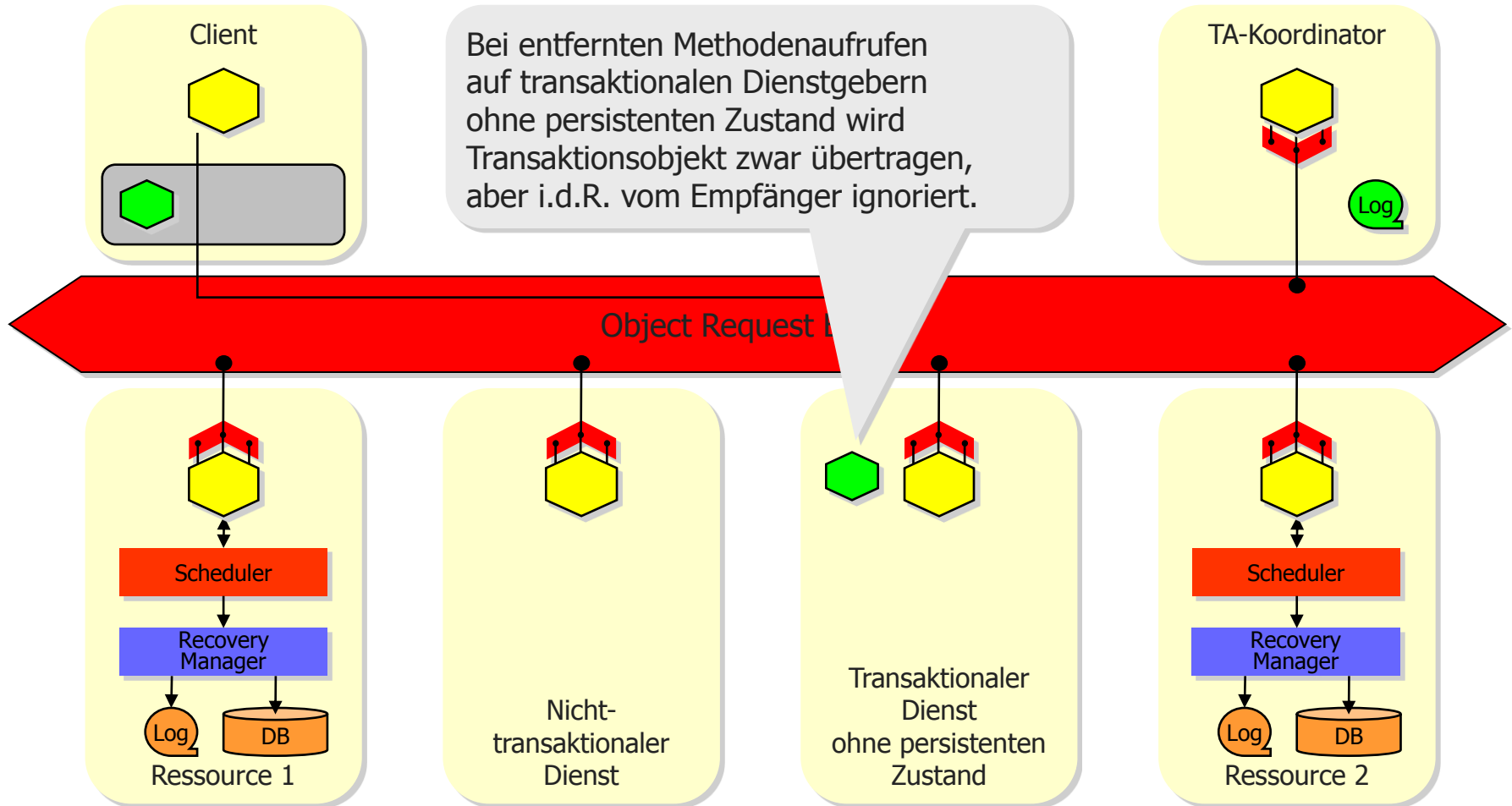




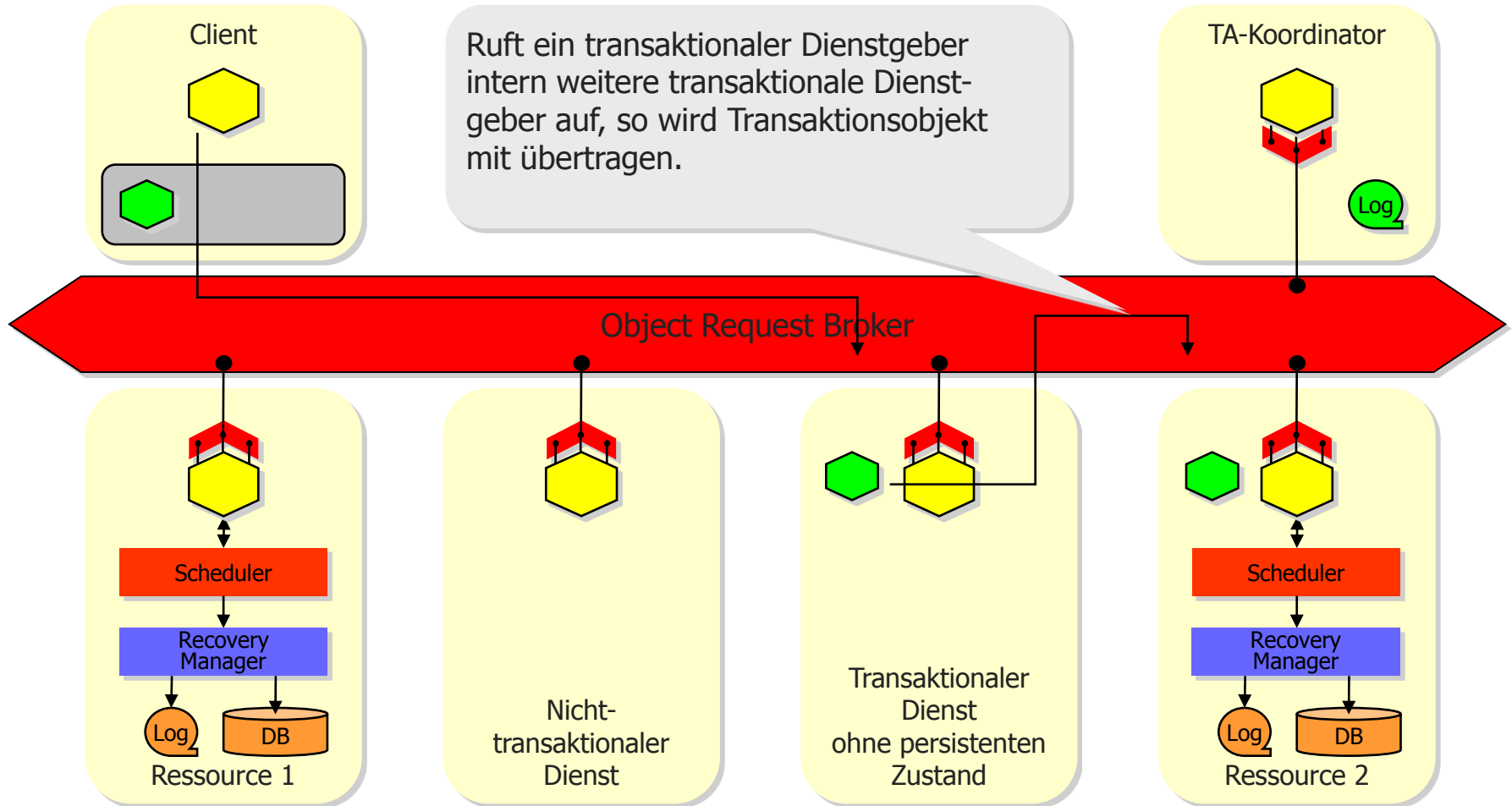
# Globale TID-Erzeugung und -Weiterleitung



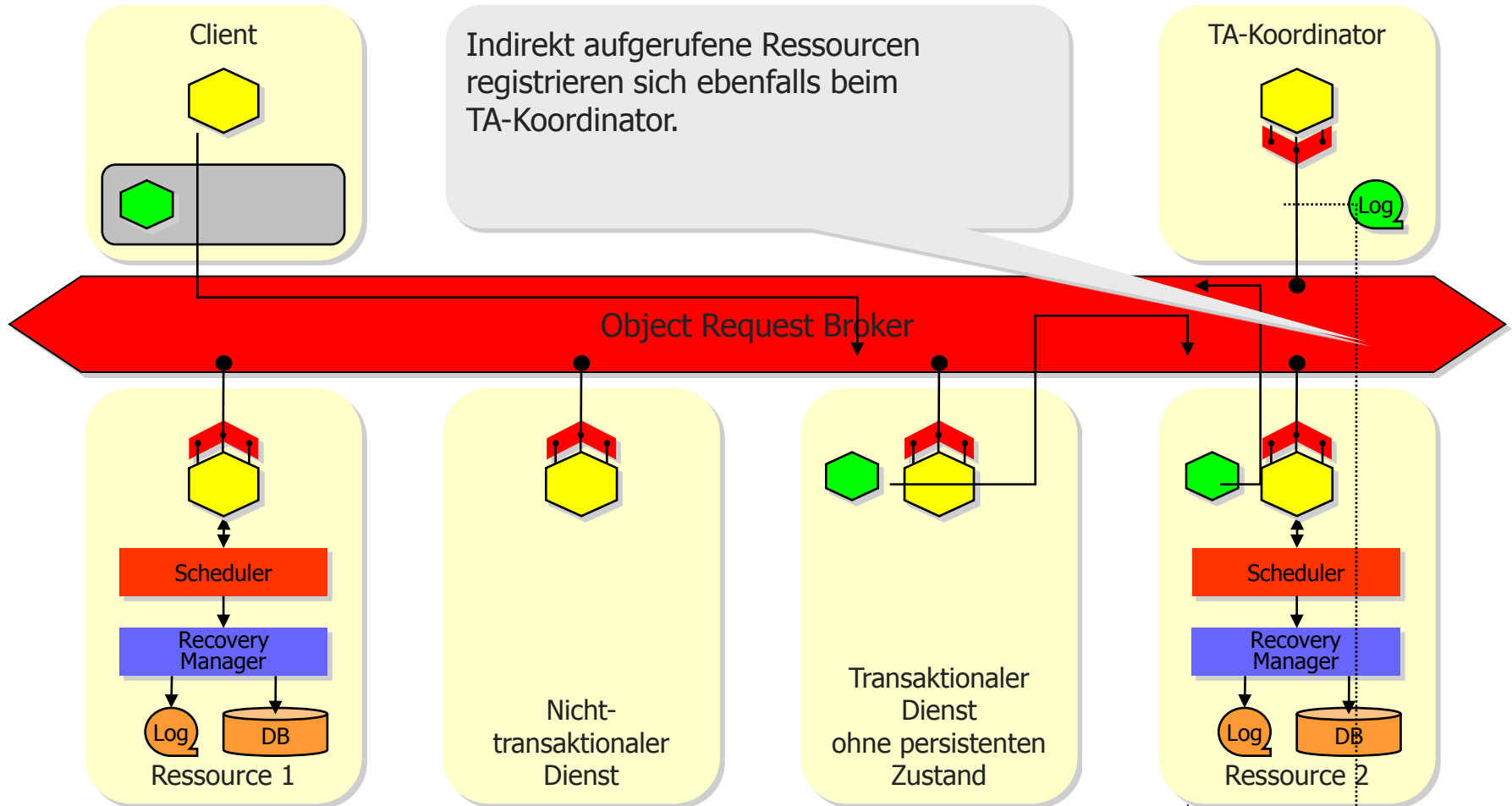
# Globale TID-Erzeugung und -Weiterleitung



# Globale TID-Erzeugung und -Weiterleitung



# Globale TID-Erzeugung und -Weiterleitung



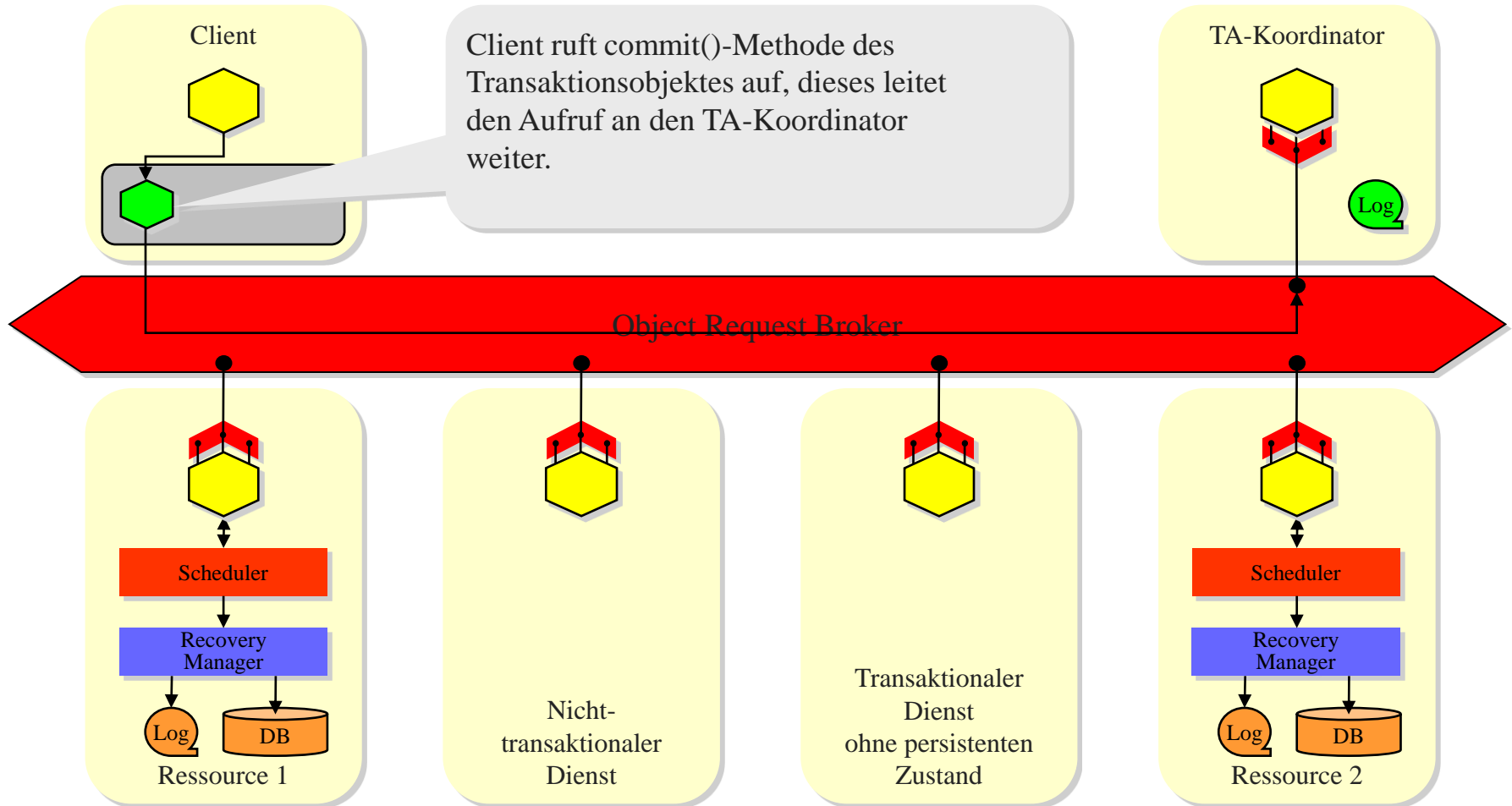
# Commit-Protokoll

- ◆ **Commit-Protokolle** gibt es seit den frühen 70ern; das berühmte *Two-Phase-Commit* wurde 1978 von J. Gray vorgestellt (Turing-Preisträger 1999)
- ◆ **Aufgabe:** Atomarität der **verteilten Transaktion** herstellen
- ◆ Ein **atomares Commit-Protokoll** (ACP) erfüllt folgende **Anforderungen:**
  - (1) Alle Knoten, die eine Entscheidung treffen, treffen dieselbe Entscheidung.
  - (2) Ein Knoten kann seine Entscheidung nicht nachträglich ändern.
  - (3) Die Entscheidung, eine Transaktion zu bestätigen, kann nur von allen Knoten einstimmig getroffen werden.
  - (4) Falls keine Ausfälle vorliegen und alle Knoten zugestimmt haben, wird die Transaktion festgeschrieben (bestätigt).
  - (5) Nach Behebung eventueller Ausfälle muß eine Entscheidung getroffen werden
- ◆ **Ein-Phasen-Commit-Protokolle** sind zu einfach; sie erlauben es den beteiligten Subtransaktionen nicht, einen Abbruch an den Koordinator zu melden, der ja dann wieder die anderen informieren müßte.

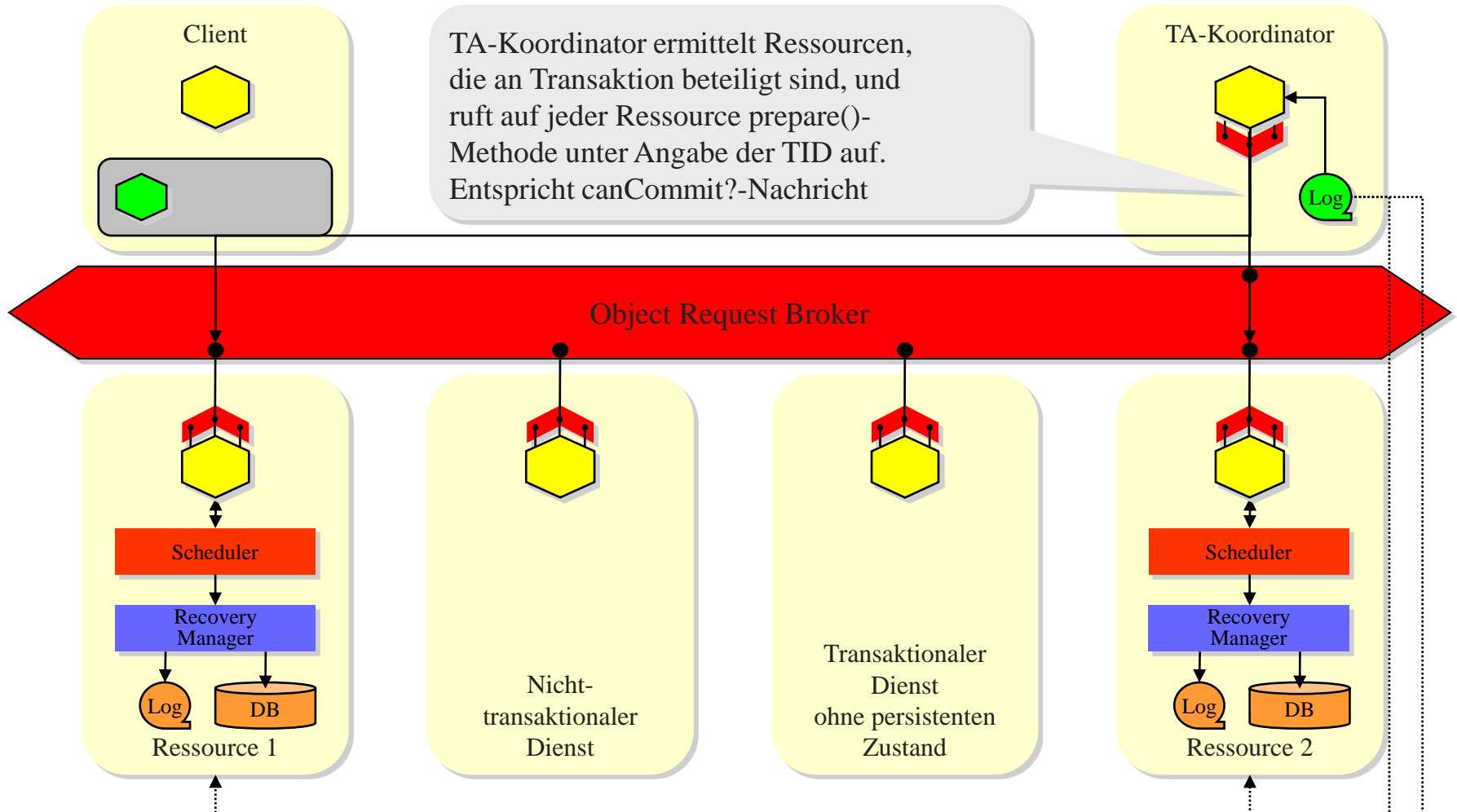
# Zwei-Phasen-Commit-Protokoll

- ◆ Einer der involvierten Knoten (üblicherweise der „Heimatknoten“ der Transaktion) übernimmt die **Rolle des Koordinators**, alle anderen sind **Teilnehmer** (participants)
- ◆ Jeder Knoten unterhält eine **spezielle Log-Datei**, in die alle relevanten Ereignisse geschrieben werden.
- ◆ Das **Protokoll beginnt** erst, wenn der **Client** die Transaktion **beendet**. Vorher gibt es nur ein `join`, um der Transaktion beizutreten.
- ◆ Das Protokoll besteht aus **zwei Phasen** mit jeweils zwei Schritten:
  - Phase 1: **Abstimmungsphase**
    - (1) Aufforderung zur Stimmabgabe (*canCommit? request*)
    - (2) Stimmabgabe (*yes or no vote*)
  - Phase 2: **Abschluß gemäß des Abstimmungsergebnisses**
    - (3) Mitteilung über die Entscheidung (*doCommit/doAbort decision*)
    - (4) Bestätigung der Entscheidung (*acknowledge*)

# Zwei-Phasen-Commit

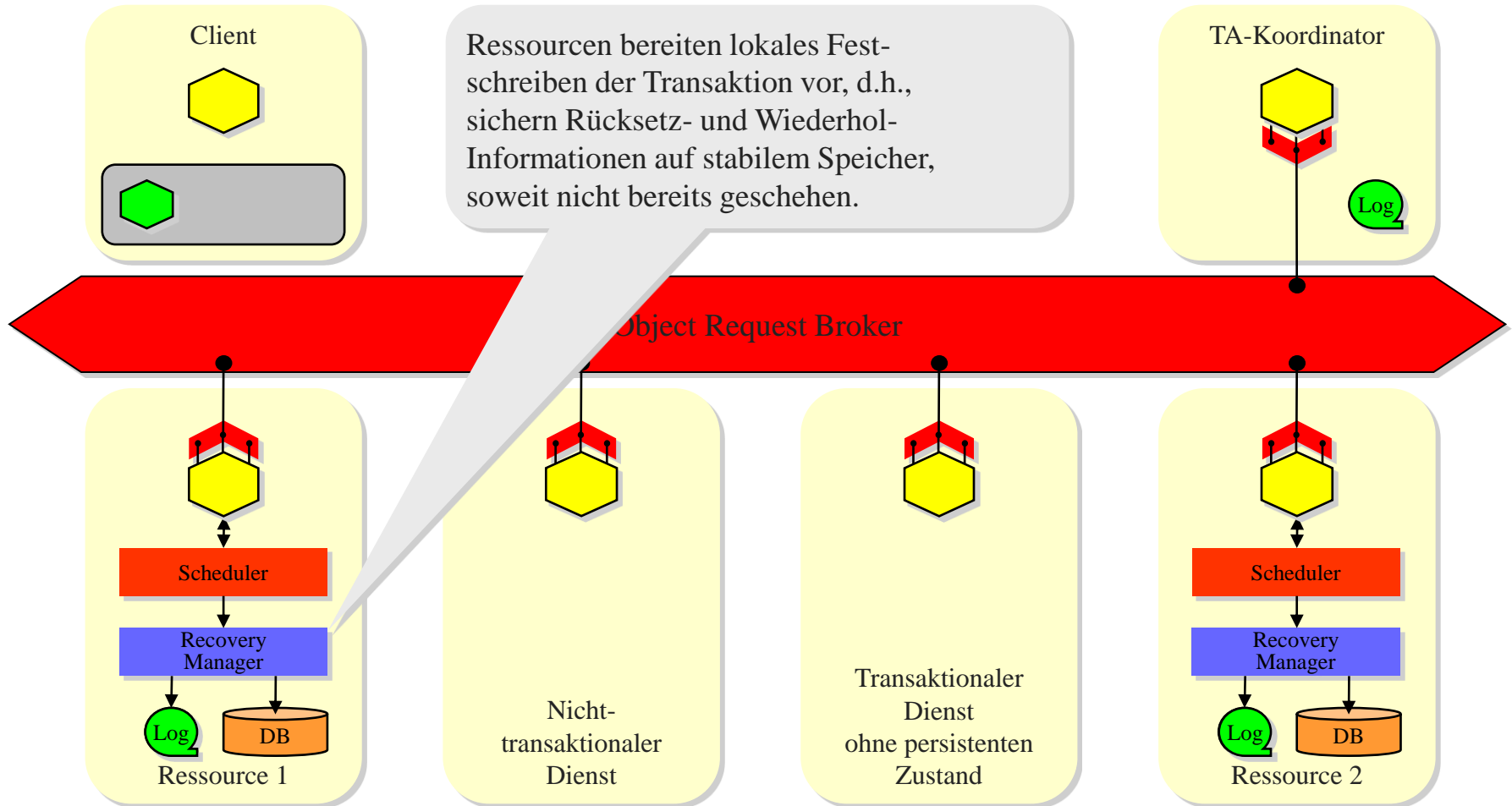


# Zwei-Phasen-Commit

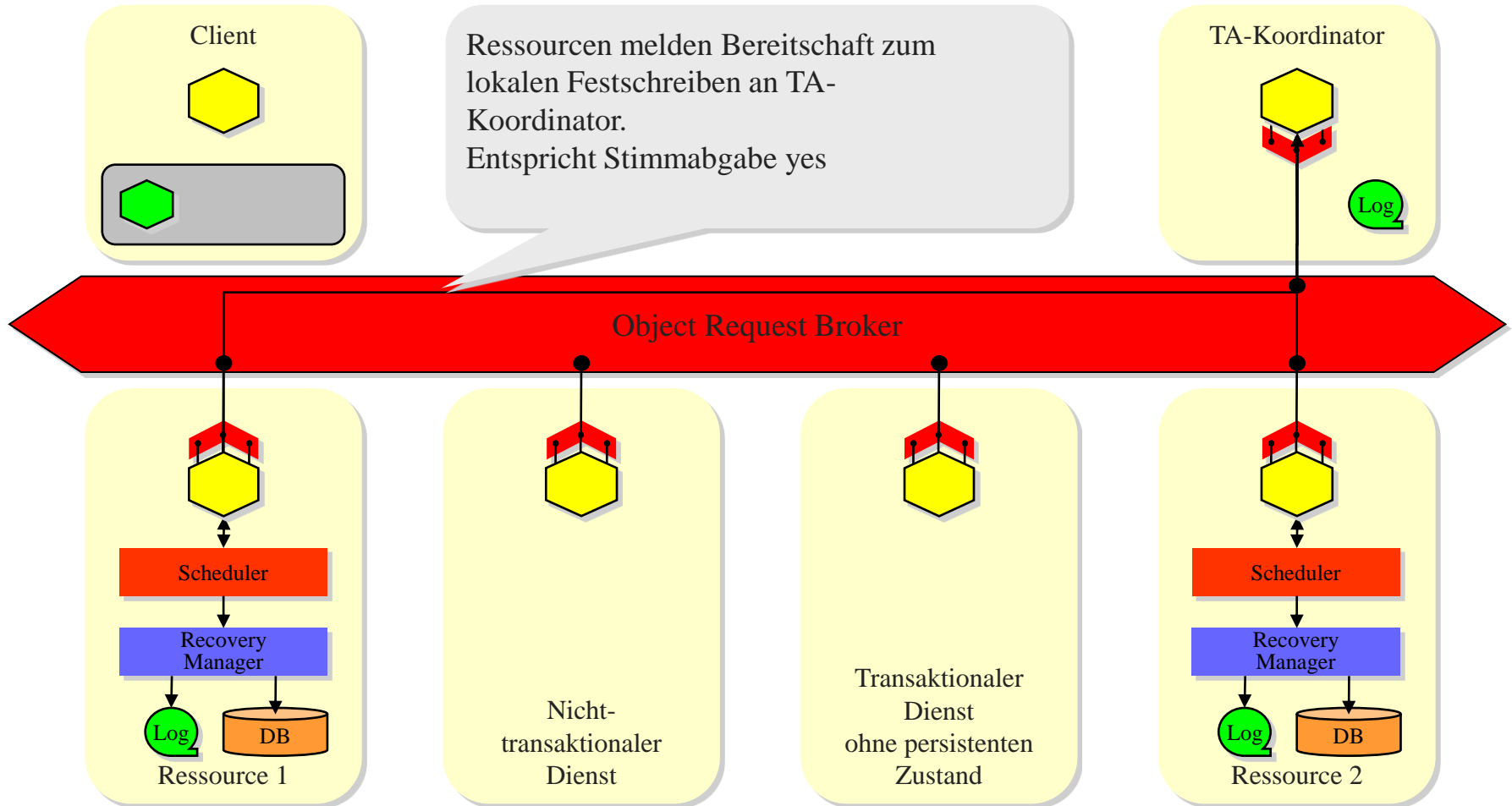




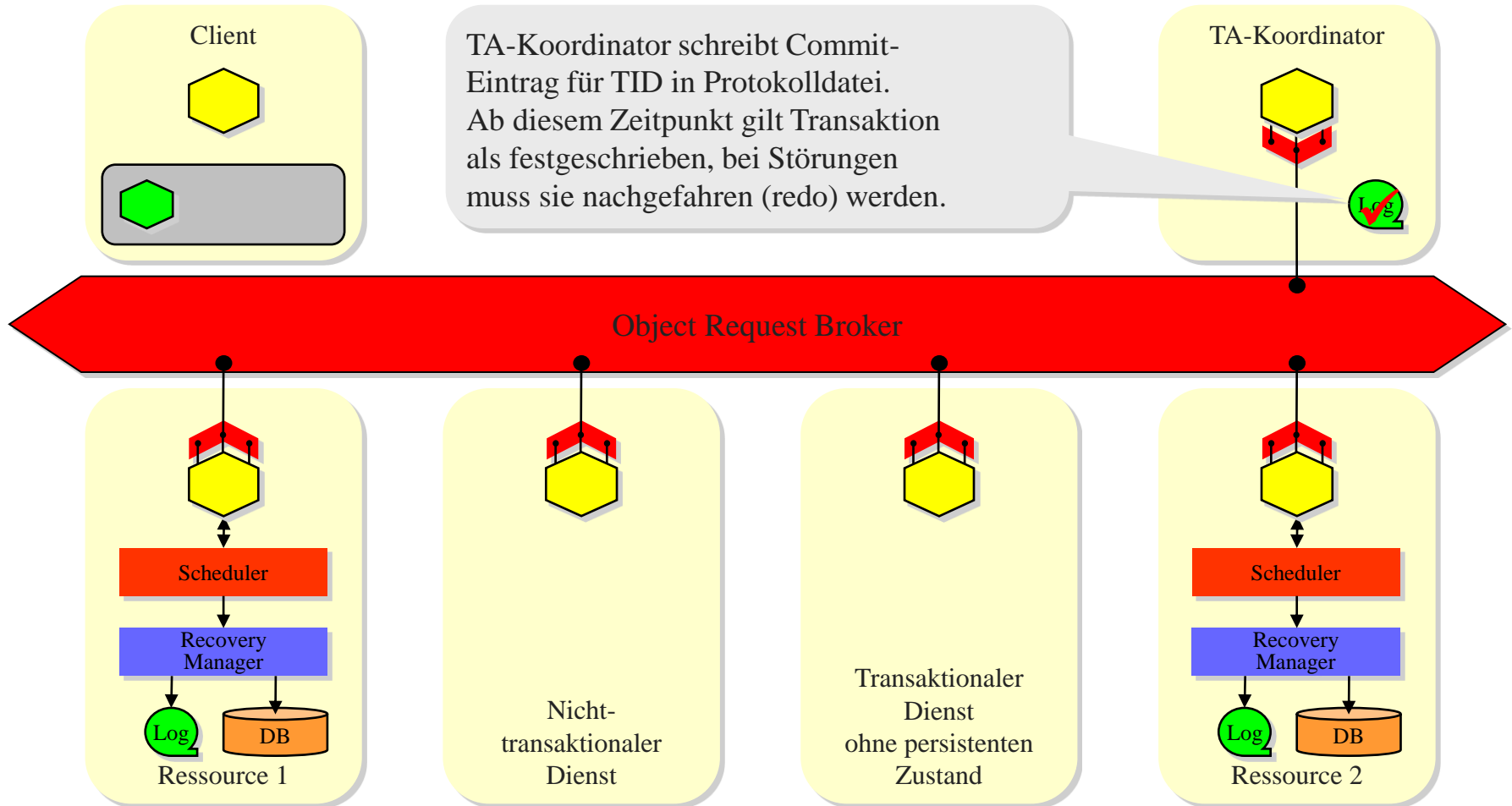
# Zwei-Phasen-Commit



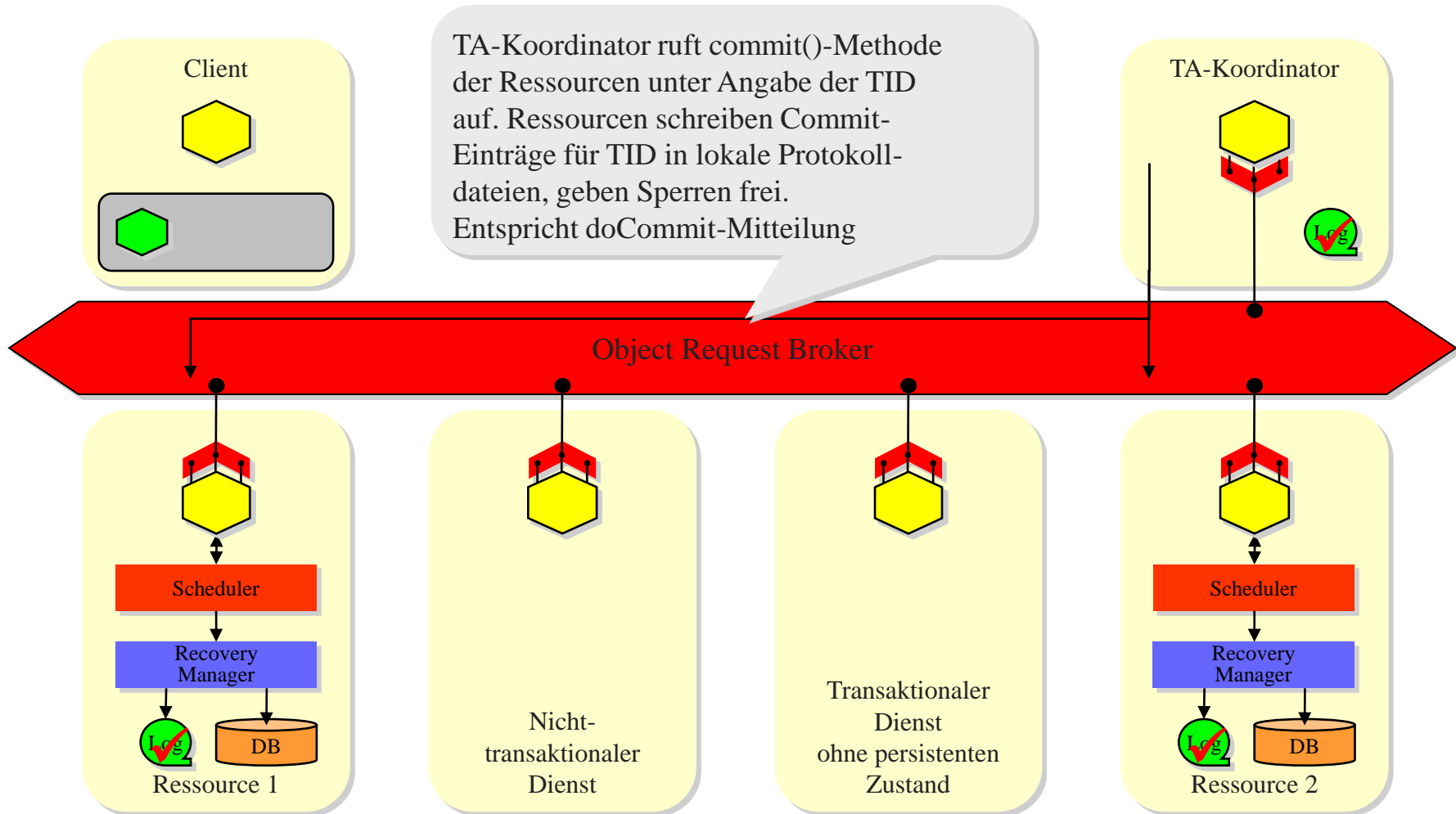
# Zwei-Phasen-Commit



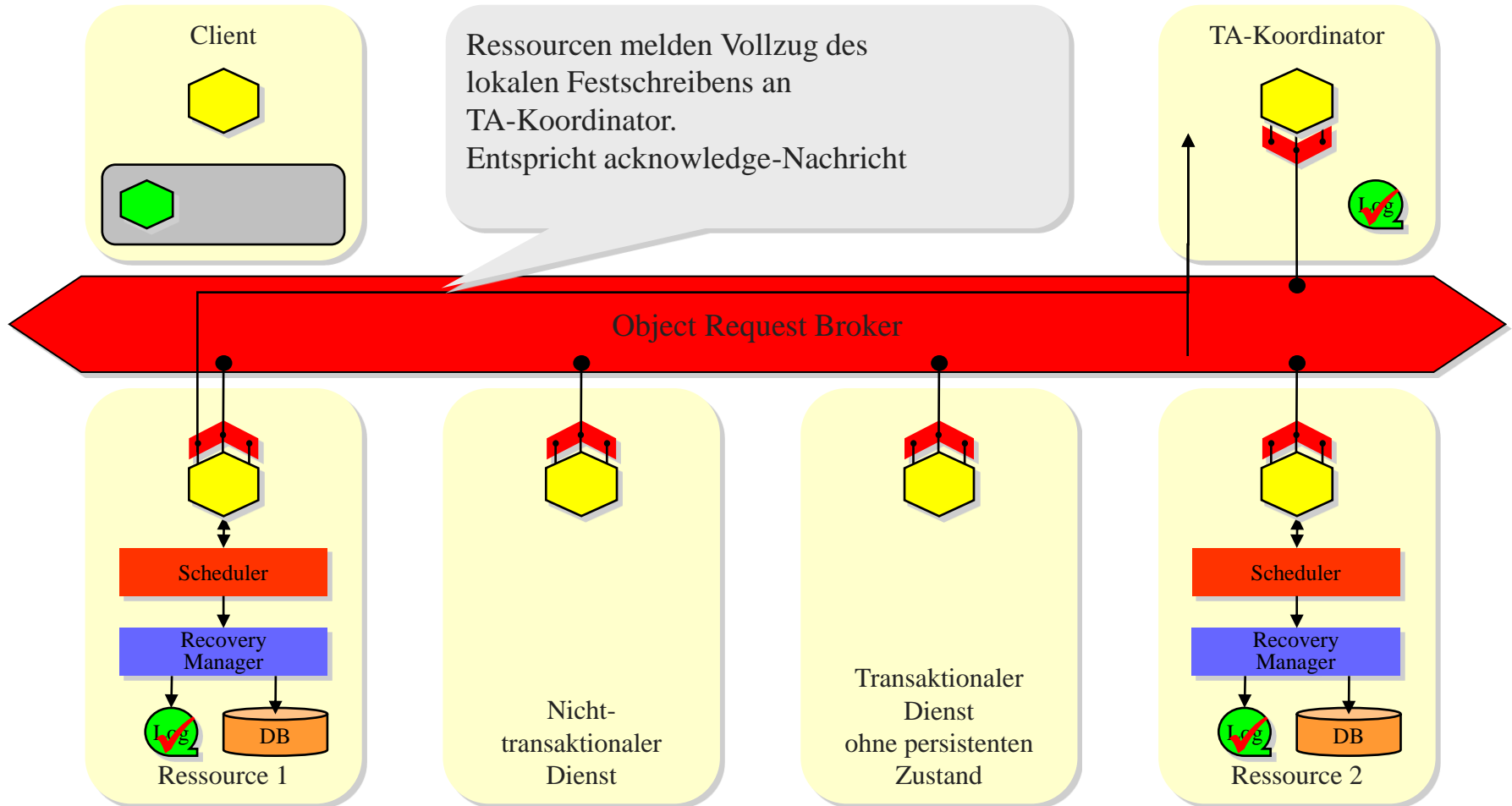
# Zwei-Phasen-Commit



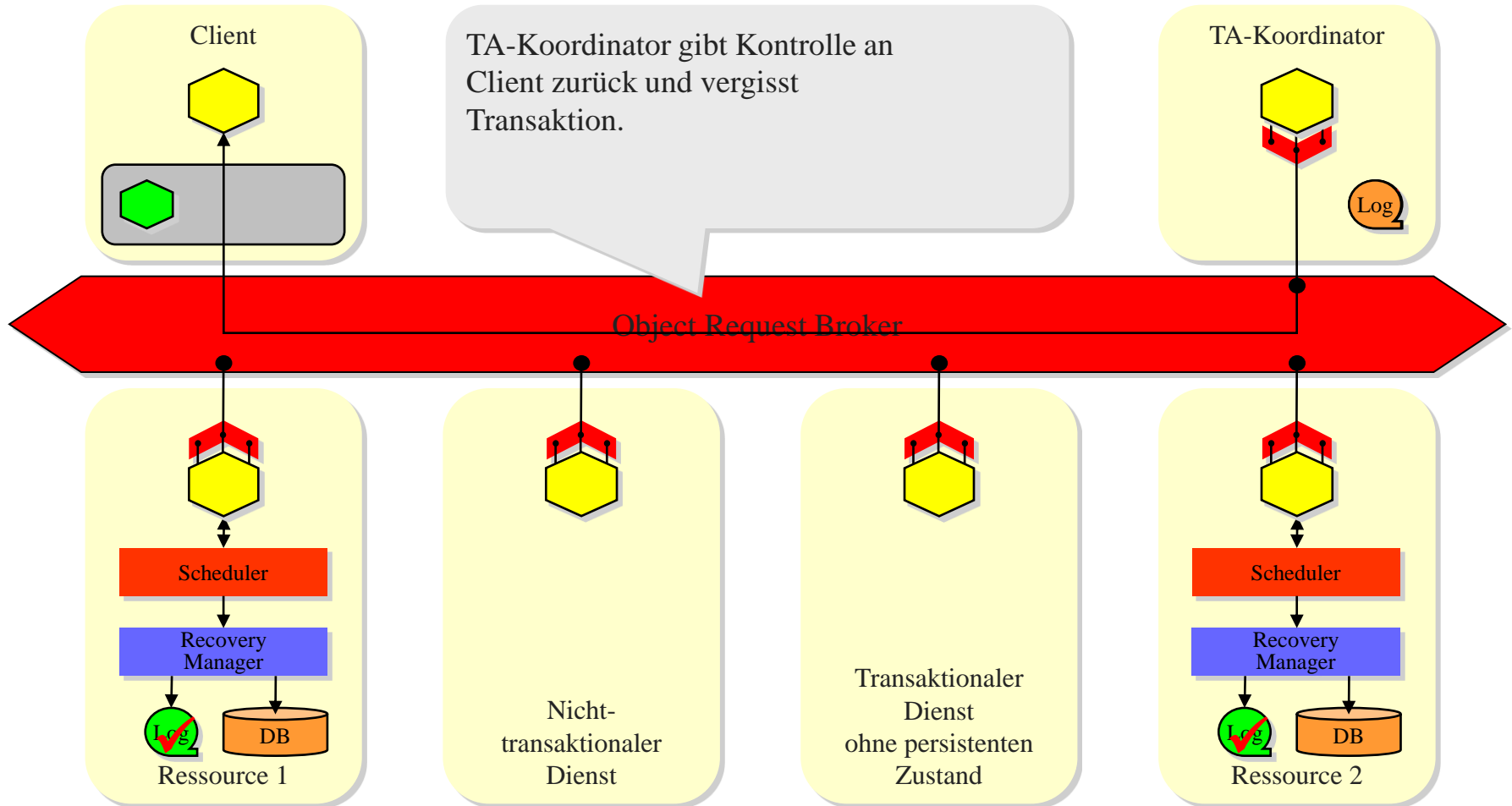
# Zwei-Phasen-Commit



# Zwei-Phasen-Commit



# Zwei-Phasen-Commit



# Verhalten im Fehlerfall

- (1) **Teilnehmer wartet** auf *canCommit? request*: Falls keine Nachricht eintrifft, entscheidet der Teilnehmer *Abort* und stoppt.
- (2) **Koordinator wartet** auf *vote-messages*: Falls nicht alle Stimmabgaben eingetroffen sind, entscheidet der Koordinator *Abort*, sendet *doAbort decision* und stoppt.
- (3) **Teilnehmer wartet** auf *decision-message*: Da die Entscheidung zu diesem Zeitpunkt vielleicht bereits getroffen ist, ist eine einseitige Entscheidung wie in (1) nicht mehr möglich. Um die Ungewißheit zu beenden, muß ein anderer Teilnehmer Q nach dem Ergebnis gefragt werden (Terminierungsprotokoll).

## Drei Möglichkeiten

- (a) Q hat die **Entscheidung** erhalten und gibt sie **weiter**
- (b) Q hat **noch nicht abgestimmt**. Q kann einseitig *Abort* entscheiden und eine entsprechende Nachricht senden
- (c) Q ist selbst in der **Ungewißheitsphase** und kann nicht helfen.

# Eigenschaften

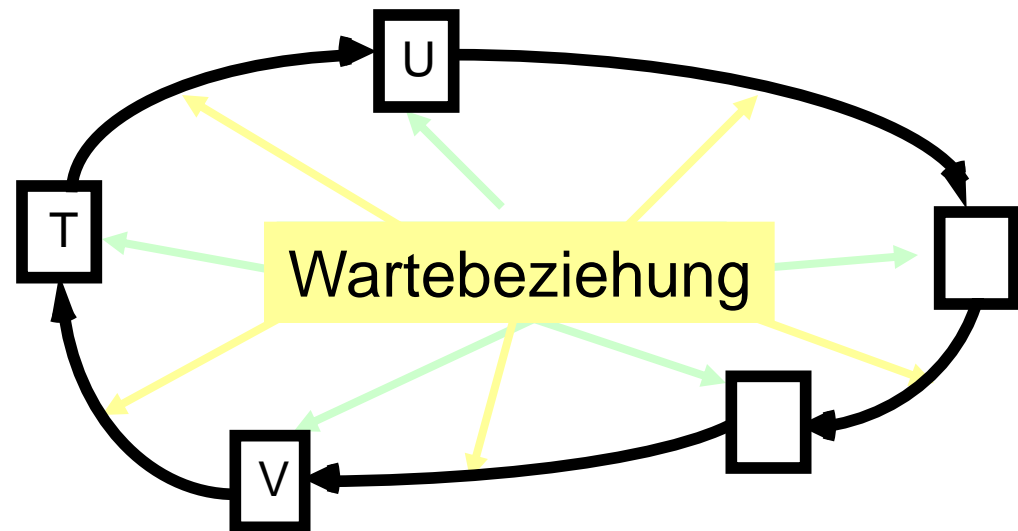
- ◆ **Fehlertoleranz:** Toleriert Knotenausfälle und Kommunikationsfehler (durch Fristablauf)
- ◆ **Blockierung:** Zwei-Phasen-Commit-Protokoll blockiert, wenn bei einem Teilnehmer ein Fristablauf während seiner Ungewißheitsphase entsteht und nur Knoten erreicht werden können, die sich ebenfalls in Ungewißheit befinden.
- ◆ **Zeitkomplexität:**
  - Das 2PC-Protokoll benötigt drei Runden.
  - Im Fehlerfall kommen zwei Runden für das Terminierungsprotokoll hinzu.
- ◆ **Nachrichtenkomplexität:**
  - Bei  $n$  Teilnehmern und 1 Koordinator:  **$3n$  Nachrichten**
  - Im Fehlerfall weitere  $O(n^2)$  Nachrichten



# Deadlock

- ◆ Ein Deadlock (Verklemmung) ist ein **Zustand**, in dem **jedes** Mitglied einer Gruppe von Transaktionen darauf **wartet**, daß ein **anderes** Mitglied eine Sperre freigibt.
- ◆ Je feiner die Granularität bei der Nebenläufigkeitskontrolle ist, desto geringer ist die Gefahr von Deadlocks.
- ◆ **Frage:** kann man Deadlocks erkennen bzw. verhindern?

Warte-Graph



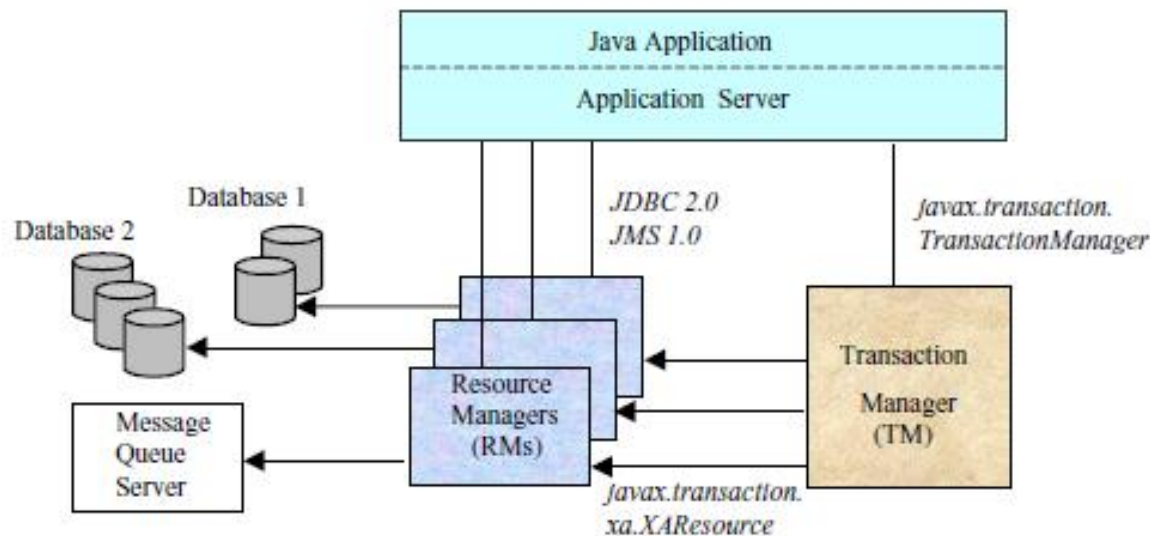
# Verteilte Deadlocks

- ◆ In verteilten Transaktionen wird auch das Problem der Deadlocks noch einmal eine Stufe schwieriger: **verteilte Deadlocks können entstehen.**
- ◆ Verteilte Deadlocks können u.U. nicht am lokalen Wartegraphen erkannt werden. Vielmehr muss ein **globaler Graph** aufgebaut und dieser dann auf Zyklen untersucht werden.
- ◆ Der globale Wartegraph wird **aus den lokalen Graphen** konstruiert.
- ◆ **Einfachste Lösung:**
  1. Der **zentrale Koordinator sammelt** die lokalen Graphen und konstruiert den Graphen.
  2. Anschließend **untersucht** er ihn auf Deadlocks.
  3. Er **fällt eine Entscheidung** und informiert die betreffenden Server über die Notwendigkeit eines Transaktionsabbruchs.
  4. Gehe zu 1
- ◆ Nicht immer gut wegen der üblichen Probleme (Bottleneck, single point of failure)

# Phantom-Deadlock

- ◆ Ein Phantom-Deadlock ist ein entdecktes Deadlock, das in der Realität jedoch gar nicht existiert.
- ◆ **Woher kann das kommen?**
- ◆ Genau von dem Hauptproblem verteilter Systeme, daß sich **ein Zustand zu einem gegebenen Zeitpunkt nicht feststellen läßt.**
- ◆ Das heißt, der Wartegraph kann eine **Mischung** aus älteren und neueren Daten sein; eine bisher belegte Sperre kann inzwischen schon wieder freigegeben sein.
- ◆ Kann **nicht** passieren **bei** Verwendung von **Zwei-Phasen-Sperren!**
- ◆ **Mögliche Lösung:** Verteilte Ermittlung des Graphen

# Implementierung verteilter Transaktionen



Quelle: Oracle JTA Spezifikation 1.1 (<http://download.oracle.com/otndocs/jcp/jta-1.1-spec-oth-JSpec/>)

# Java Transaction API

- ◆ Das Java Transaction API (JTA) stellt ein **Interface** zur Anbindung von Transaktionsmanagern bereit
- ◆ Die Transaktionsmanager werden im jeweiligen Application Server eines Anbieters implementiert
- ◆ Damit ist es möglich, verteilte Transaktionen transparent über Applikationsservergrenzen hinweg durchzuführen
  - In der Praxis hat sich jedoch gezeigt, dass nicht alle Implementierungen kompatibel sind!

## Bestandteile des JTA (1/2)

- ◆ **UserTransaction Interface**
  - Interface um Transaction für den Anwendungsentwickler bereitzustellen.
- ◆ **TransactionManager Interface**
  - Interface, das die Schnittstelle der Implementierung im Application Server definiert.
- ◆ **XAResource Interface**
  - Java Mapping für das X/Open XA Protokoll. Das Protokoll ist eine Implementierung eines 2-Phasen-Commits.

## Bestandteile des JTA (2/2)

- ◆ **Xid Interface**
  - Interface, das eindeutige Transaction Ids (TID) definiert
- ◆ **TransactionSynchronizationRegistry Interface**
  - Interface, das eine Schnittstelle zum registrieren von Ressourcen, die in der Transaktion benötigt werden, definiert.