Verteilte Systeme

Replikation

Sinn der Replikation

- **Replikation** bedeutet das *Halten* einer oder mehrerer **Kopien** eines Datums
- Ein Prozess, der auf dieses Datum zugreifen will, kann auf **jede beliebige Replika zugreifen**
- Im **Idealfall** erhält er immer das gleiche Ergebnis
- Was also erreicht werden muss, ist die **Konsistenz** der Kopien wobei unterschiedliche Anwendungen unterschiedliche Anforderungen an die **Striktheit der Konsistenz** haben.

Ziele der Replikation

Zwei große **Ziele**

- **Steigerung** der **Verlässlichkeit** eines Dienstes bzw. der **Verfügbarkeit** eines Datums
  - Wenn ein Replikat nicht mehr verfügbar ist, können andere verwendet werden. (Mobilität oder Absturz)
  - Besserer Schutz gegen zerstörte/gefälschte Daten: bei gleichzeitigem Zugriff auf mehrere Replikate wird das Ergebnis der Mehrheit verwendet

- **Steigerung** der **Leistungsfähigkeit** des Zugriffs auf ein Datum
  Bei großen Systemen:
  - Verteilung der Replikate in verschiedene Netzregionen oder
  - Einfache Vervielfachung der Server an einem Ort

Problem der Replikation

- Die verschiedenen **Kopien** müssen **konsistent gehalten** werden. Das ist insbesondere ein Problem
  - Wenn es **viele** Kopien gibt
  - Wenn die Kopien **weit verstreut** sind
- Es gibt eine Reihe von **Lösungen** zur absoluten Konsistenzhaltung in nicht-verteilten Systemen, die jedoch die **Leistung** des Gesamtsystems **negativ beeinflussen**.
- **Dilemma**: wir wollen bessere Skalierbarkeit und damit bessere Leistung erreichen, aber die dazu notwendigen Mechanismen verschlechtern die Performance.
- **Einzige Lösung**: *keine* strikte Konsistenz

Anwendungsbeispiel

- Ansicht 1:

Bulletin board: OS.interesting

Item	From	Subject
23	A. Hampton	Mach
24	G. Joseph	Microkernels
25	A. Hampton	Re: Microkernels
26	T.L. Heureux	RPC Performance
27	M. Walker	Re: Mach
end		

- Ansicht 2:

Bulletin board: OS.interesting

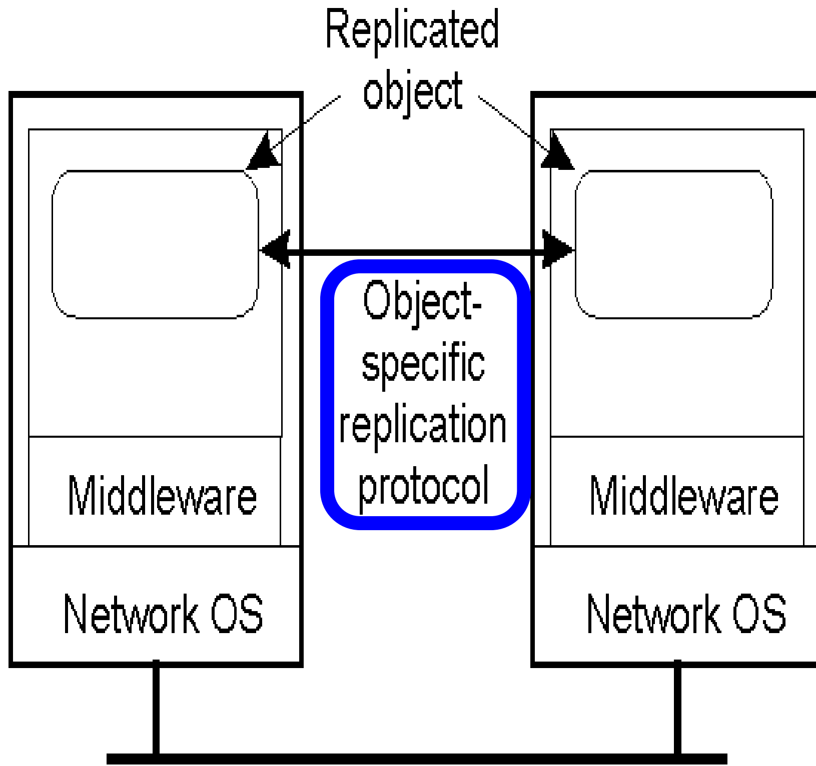
Item	From	Subject
20	G. Joseph	Microkernels
21	A. Hampton	Mach
22	A. Sahiner	Re: RPC performance
23	M. Walker	Re: Mach
26	T.L. Heureux	RPC Performance
27	A. Hampton	Re: Microkernels
end		

- Probleme:
  - Nachrichten tauchen in **unterschiedlicher Reihenfolge** auf.
  - Sie kommen **überhaupt nicht** an.
- Für News ist das OK, aber andere Anwendungen?

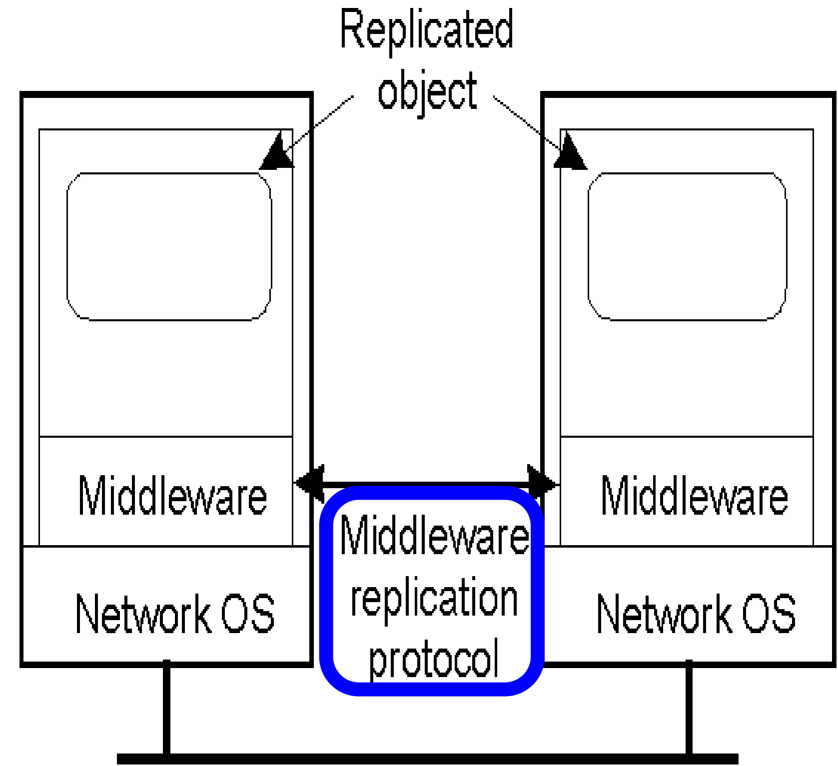
Systemmodell

- **Daten** im System = Sammlung von **Objekten** (Datei, Java-Objekt, etc.)
- Jedes logische Objekt wird durch eine Reihe physischer Objekte realisiert, den **Replikaten**.
- Die **Replikate** müssen **nicht** zu jeder Zeit **absolut identisch** sein – sie *können* es tatsächlich auch nicht sein.
- Die **Replikations-Intelligenz** kann
  - in den Objekten platziert sein oder
  - außerhalb (in einer Middleware). Vorteil hier: Anwendungsprogrammierer ist frei von Überlegungen zur Replikation

Replikations-Intelligenz



Die Objektebene (z.B. Programmierer) ist für das Replikationsmanagement verantwortlich



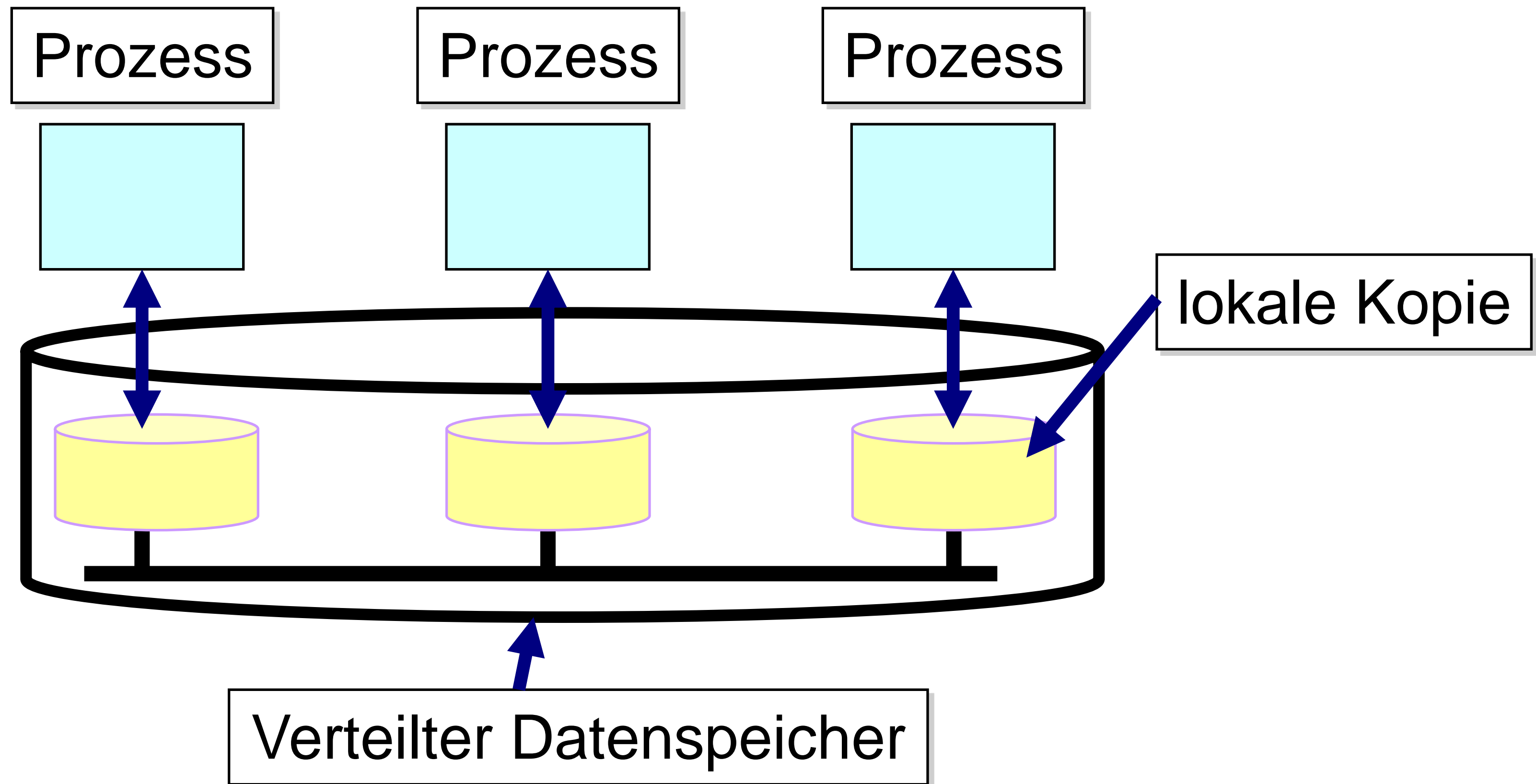
Das Verteilte System (Middleware) ist für das Replikationsmanagement verantwortlich

Konsistenzmodelle

- **Konsistenzmodell**:
  - Im Prinzip ein **Vertrag zwischen** einem **Datenspeicher** und den darauf **zugreifenden Prozessen**
  - „Wenn sich die Prozesse an **gewisse Regeln** halten, arbeitet der Datenspeicher **korrekt**.“
  - **Erwartung**: der Prozess, der ein `Read` ausführt, erwartet als Ergebnis den Wert des letzten `Write`
  - **Frage**: was ist das letzte `Write` in Abwesenheit einer globalen Uhr?
  - **Lösung**: verschiedene Konsistenzmodelle (nicht nur strikt)

- **Daten-zentrierte** Konsistenzmodelle: Sicht des Datenspeichers
- **Client-zentrierte** Konsistenzmodelle: Sicht des Client,
  - weniger starke Annahmen,
  - insbesondere keine gleichzeitigen Updates

Daten-zentriertes Konsistenzmodell



- physikalisch verteilt
- repliziert für verschiedene Prozesse

Strikte/Atomare Konsistenz

- **Konsequentestes** Konsistenzmodell
- Modell: „Jedes `Read` liefert als Ergebnis den Wert der letzten `Write` Operation."
- **Notwendig** dazu: absolute **globale Zeit**
- Unmöglich in einem verteilten System, daher **nicht implementierbar**

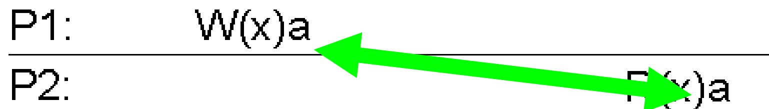
Withdraw $100

Waterloo

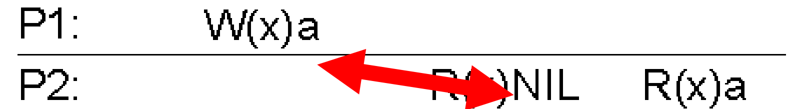
`W(x)a`: `Write`-Operation **auf Replikt** des Objekts x mit Wert a
Die Änderung muss per Nachricht an andere Replikte (direkt oder über Original) verbreitet werden
`P1`: Prozeß 1
Zeit: von links nach rechts

Bitte Warten, das Konto wird gerade geändert

Paris



Strikt Konsistent

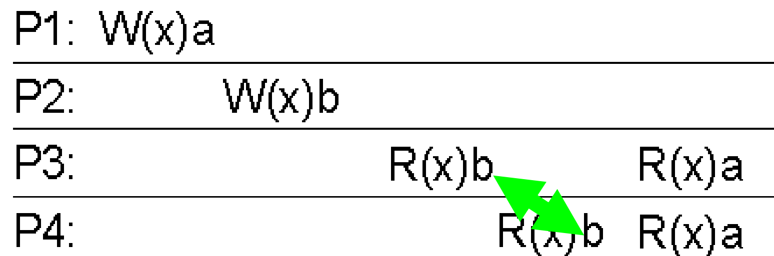


Nicht strikt Konsistent

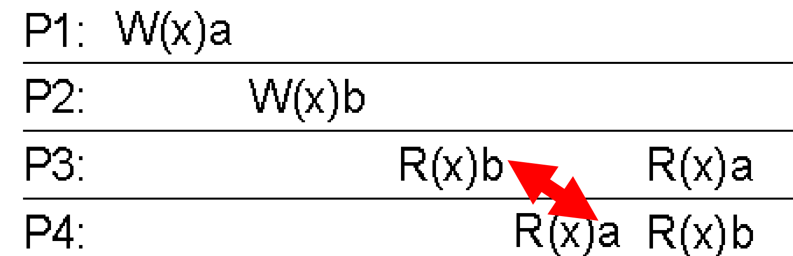
Sequentielle Konsistenz

Entspricht der seriellen Äquivalenz bei den Transaktionen, bezogen auf einzelne Operationen!

- Etwas schwächeres Modell, aber **implementierbar**.
- **Modell**: „Wenn mehrere nebenläufige Prozesse auf Daten zugreifen, dann ist jede gültige Kombination von `Read`- und `Write`-Operationen akzeptabel, solange alle Prozesse dieselbe Folge sehen."
- **Zeit** spielt **keine Rolle**
- **Jede Permutation** der Zugriffe ist **zulässig**, sofern sie von allen Prozessen so wahrgenommen wird
- Unter sequentieller Konsistenz können zwei Läufe desselben Programms **unterschiedliche Ergebnisse** liefern, sofern nicht explizite Synchronisationsoperationen dies verhindern



Sequentiell Konsistent



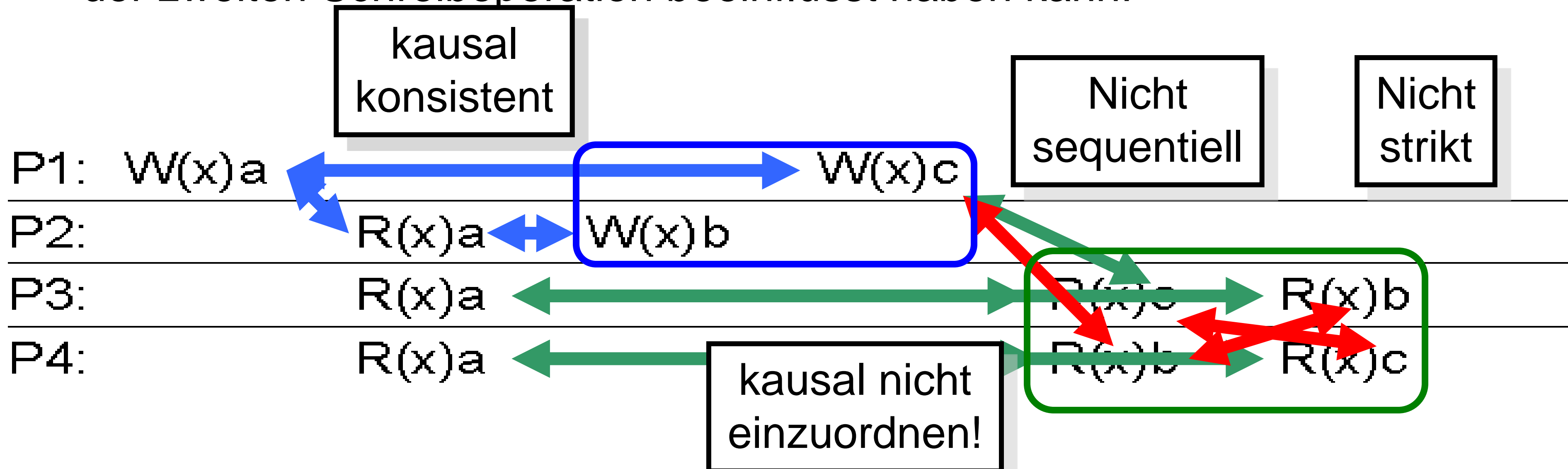
Nicht sequentiell Konsistent

Linearisierbarkeit

- Liegt in der Striktheit **zwischen** strikter und sequentieller Konsistenz, d.h. sie genügt der sequentiellen Konsistenz, erreicht aber keine strikte Konsistenz!
- **Idee**: verwende eine Menge **synchronisierter Uhren**, auf deren Basis Zeitstempel für die Operationen vergeben werden
- Verglichen mit sequentieller Konsistenz ist die **Ordnung** dann nicht beliebig, sondern **auf der Basis dieser Zeitstempel**
- **Komplexe Implementierung** (durch die Forderung der Zeitstempelreihenfolge), wird hauptsächlich eingesetzt zur formalen Verifikation nebenläufiger Algorithmen

Kausale Konsistenz

- **Schwächeres Modell** als die sequentielle Konsistenz
- Vergleichbar mit **Lamports „happened-before“-Relation**
- **Regel**: `Write`-Operationen, die *potentiell* in einem kausalen Verhältnis stehen, müssen bei allen Prozessen in derselben Reihenfolge gesehen werden. Für nicht in dieser Beziehung stehende Operationen ist die Reihenfolge gleichgültig.
- Zwei `Write`-Operationen sind auch *potentiell* kausal abhängig, wenn vor der zweiten Operation eine Leseoperation stattgefunden hat, die den Wert der zweiten Schreiboperation beeinflusst haben kann!



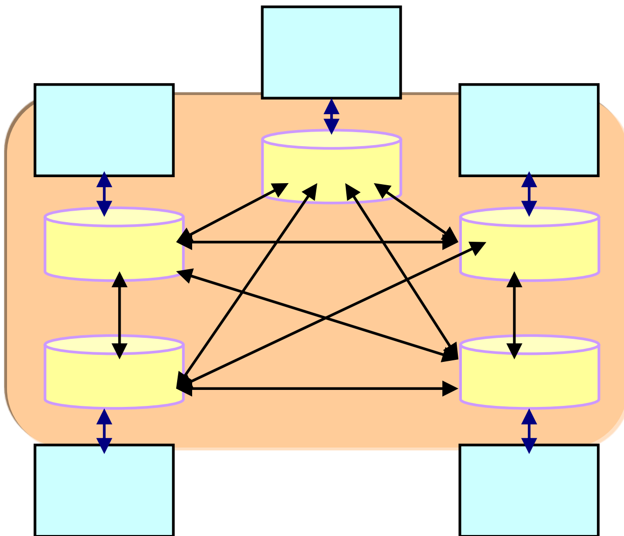
FiFo-Konsistenz

- **Idee**: alle Schreiboperationen eines Prozesses werden von allen anderen Prozessen in derselben Reihenfolge wahrgenommen.
- Schreibzugriffe **verschiedener Prozesse** können von unterschiedlichen Prozessen in **unterschiedlicher Reihenfolge** gesehen werden.
- heißt Pipelined RAM (PRAM) bei DSM‘s

P1: W(x)a					
P2:	R(x)a	W(x)b	W(x)c		
P3:			R(x)b	R(x)a	R(x)c
P4:			R(x)a	R(x)b	R(x)c

Zusammenfassung der Konsistenzmodelle

Sequentiell
Kausal
FIFO



-**eager** Strategie der Aktualisierung, d.h. jedes `write` wird (nach einer bestimmten Ordnung) unmittelbar an alle Kopien übermittelt

-**update-everywhere** Strategie der Aktualisierung, d.h. auf jeder Kopie kann eine Aktualisierung der Daten initiiert werden

Schwache Konsistenz

- In der Regel benötigen Prozesse **nur in bestimmten Situationen** eine **konsistente Sicht** auf den Speicher.
- Schwache Konsistenz **unterscheidet** daher **normale Variablen** von **Synchronisationsvariablen,** die mit einem Datenspeicher assoziiert sind.
- Die **Synchronisation erzwingt** eine **Aktualisierung** der Werte
- nur **eine Operation**: `synchronisiere(S)`; alle lokalen `Write`-Operationen werden zu allen Kopien übertragen und alle `Write`-Operationen bei anderen Kopien werden zur lokalen Kopie übertragen.
- Idee:
  - **Konsistenz für eine Gruppe von Operationen**, und nicht für einzelne Schreib- und Leseoperationen
  - eine **sequenzielle Konsistenz zwischen Gruppen** von Operationen
- Wer seine **Schreibzugriffe sichtbar** machen möchte für andere Prozesse, muss synchronisieren.
- Wer auf die **aktuellsten Daten zugreifen** möchte, muss ebenfalls synchronisieren; Wer vor dem Lesezugriff nicht synchronisiert, kann veraltete Werte sehen

Schwache Konsistenz

```
P1:  W(x)a    W(x)c  S
P2:           R(x)b    R(x)c
P3:  W(x)b  S R(x)b
```

schwach konsistent

P3 synchronisiert: alle Replikate kennen **alle Änderungen von P3** und P3 kennt alle **Änderungen der anderen Replikate!**

```
P1:  W(x)a    W(x)b          S
P2:                            R(x)b
P3:                  R(x)a
```

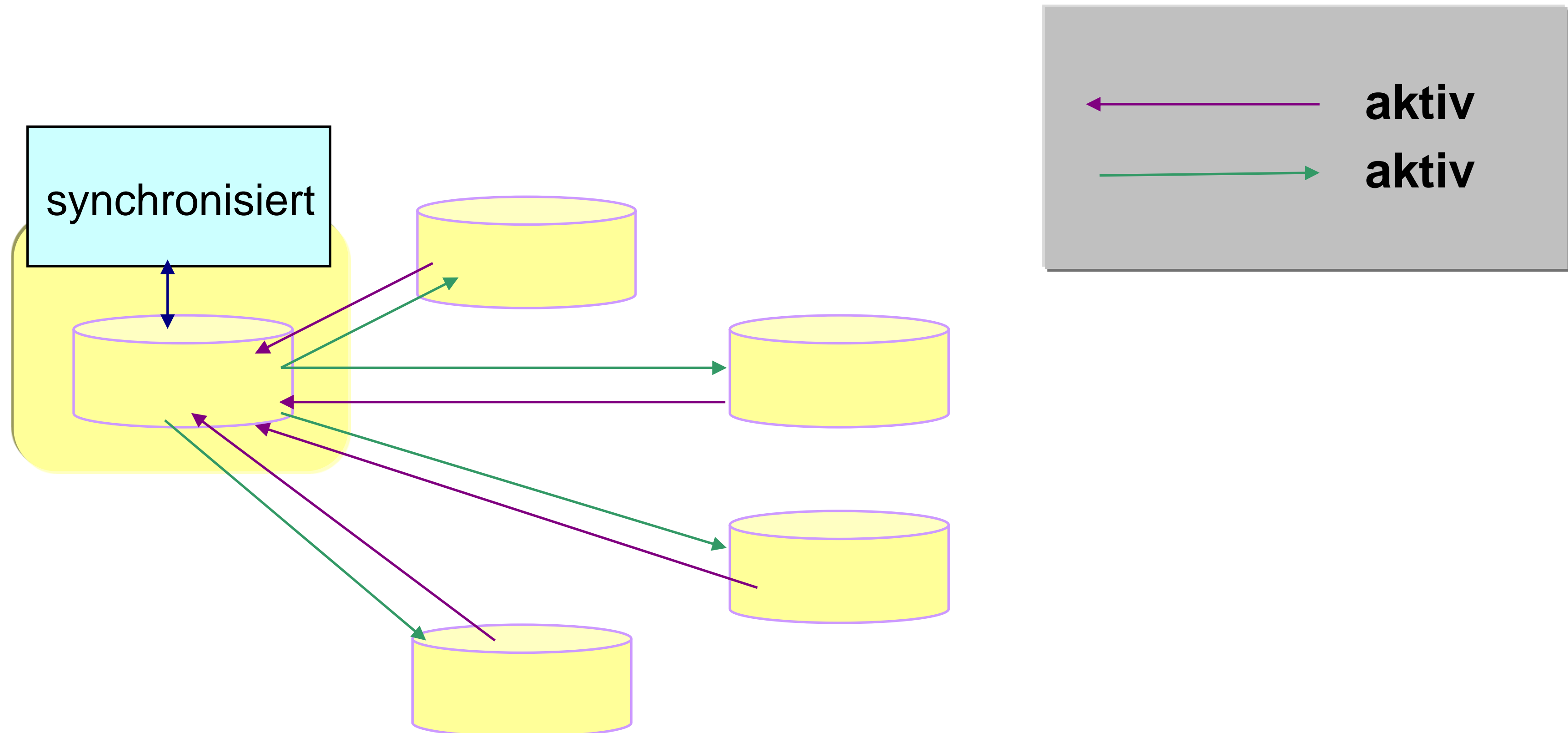
P3 hat nicht synchronisiert: er liest alte Werte!

```
P1:  W(x)a    W(x)b  S
P2:                    R(x)a
```

nicht schwach konsistent

P2 **müsste** `R(x)b` lesen, da vorher P1 synchronisiert hat und damit alle Replikate alle Änderungen von P1 kennen!

Schwache Konsistenz

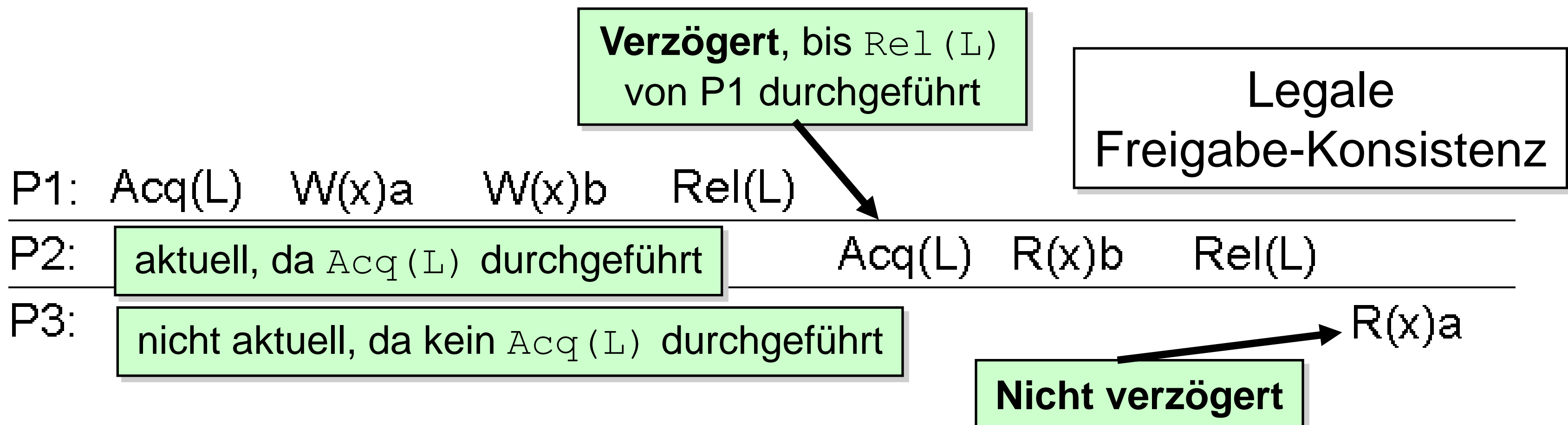


Freigabe/Eintritts-Konsistenz

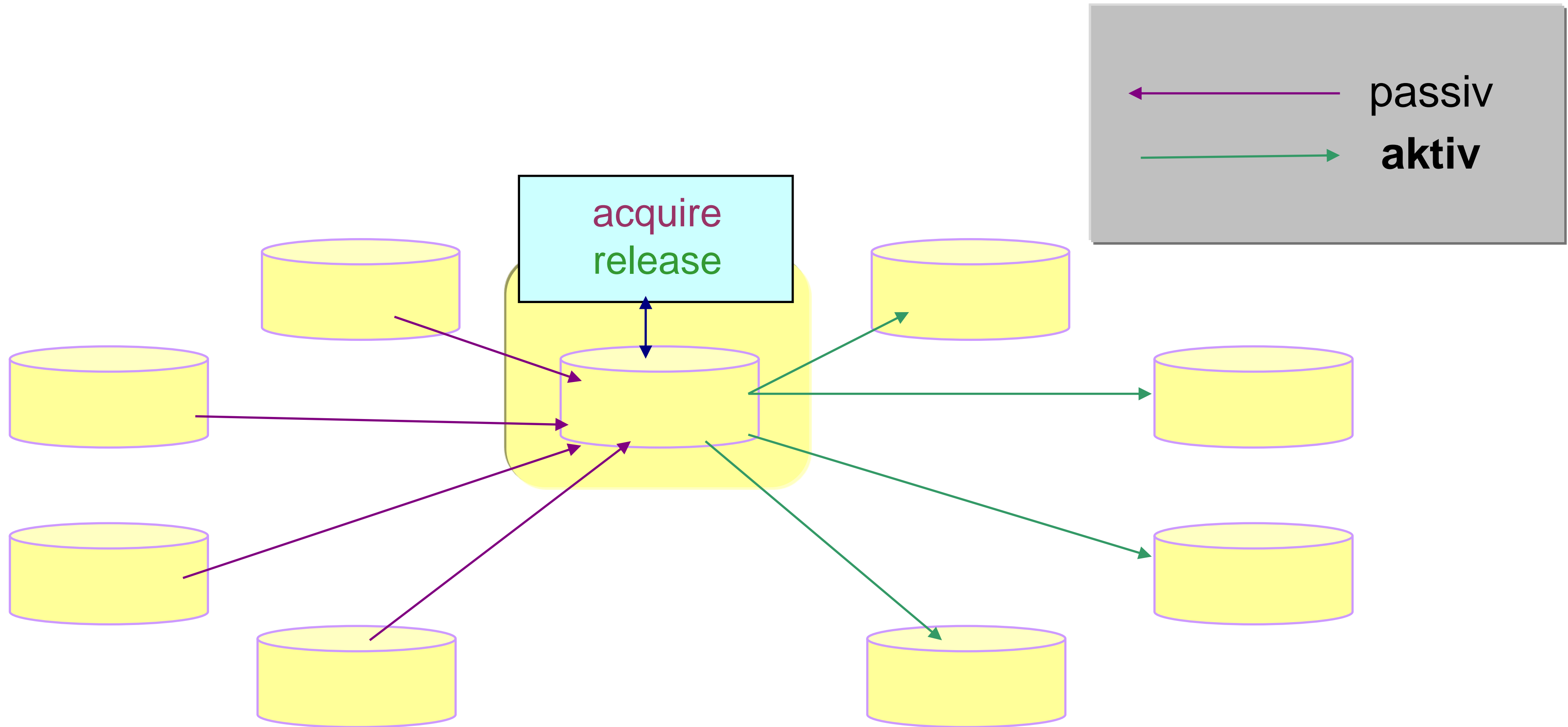
- Bei der **schwachen Konsistenz** werden bei Zugriff auf eine Synchronisationsvariable die **folgenden Aktionen** durchgeführt:
  1. alle **Änderungen** des Prozesses werden **an alle anderen** Kopien propagiert
  2. alle **Änderungen von anderen** Kopien werden zur Kopie des Prozesses propagiert.
- Dies ist i.allg. **nicht nötig**. Idee: **Kritische Zugriffe** können in **kritische Abschnitte** geklammert werden.
- Es gibt **zwei Operationen** für den kritischen Abschnitt:
  - `acquire`: **nur Aktion 2**
  - `release`: **nur Aktion 1**

Freigabe-Konsistenz

- Bei einem `acquire` wird der Prozess **verzögert**, während ein anderer Prozess ebenfalls auf die gemeinsam genutzten Variablen zugreift. Bei einem `release` werden **alle lokalen Änderungen** an den geschützten Daten **sichtbar gemacht** (z.B. an die Kopien verteilt).



Freigabe-Konsistenz



Eintritts-Konsistenz

- Alle Daten benötigen eine Synchronisationsvariable!
- Bei einem `acquire` müssen **alle entfernten Änderungen** an den geschützten Daten **sichtbar gemacht** werden. Bei einem `release` passiert nichts!
- **Vor der Aktualisierung** eines gemeinsam genutzten Datenelements muss ein Prozess **im exklusiven Modus** in einen **kritischen Bereich eintreten**, um sicherzustellen, dass **keine anderen Prozesse gleichzeitig** versuchen, das Datenelement zu aktualisieren!
- Wenn ein Prozess im **nicht-exklusiven Modus** in einen kritischen Bereich eintreten will, muss er sich zuerst mit dem **Eigentümer** der Synchronisierungsvariablen, die den kritischen Bereich schützt, **in Verbindung setzen**, um die **neuesten Kopien** der geschützten gemeinsam genutzten Daten **zu erhalten**.

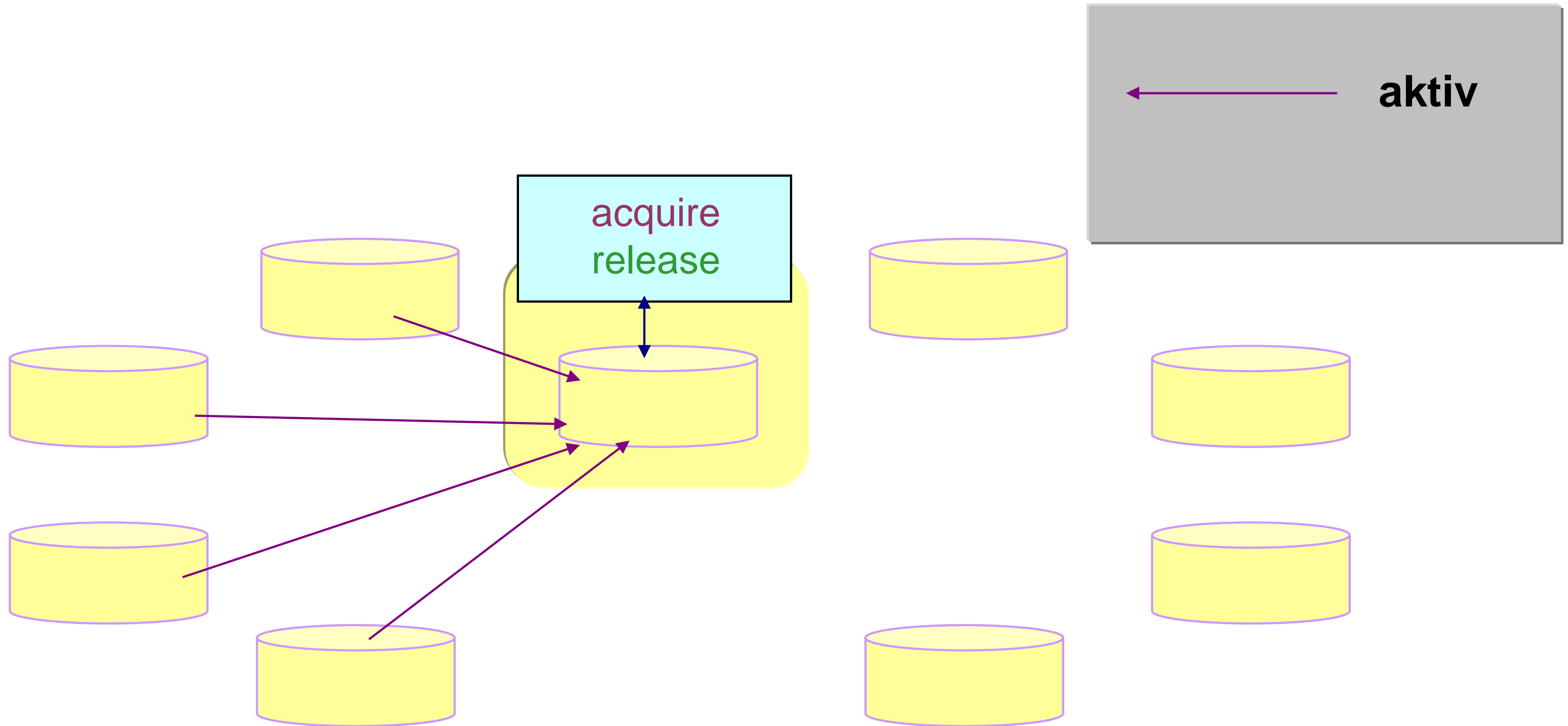
Legale Eintritts-Konsistenz

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)

P2: [nicht aktuell, da kein `Acq(Ly)` durchgeführt] Acq(Lx) R(x)a R(y)NIL

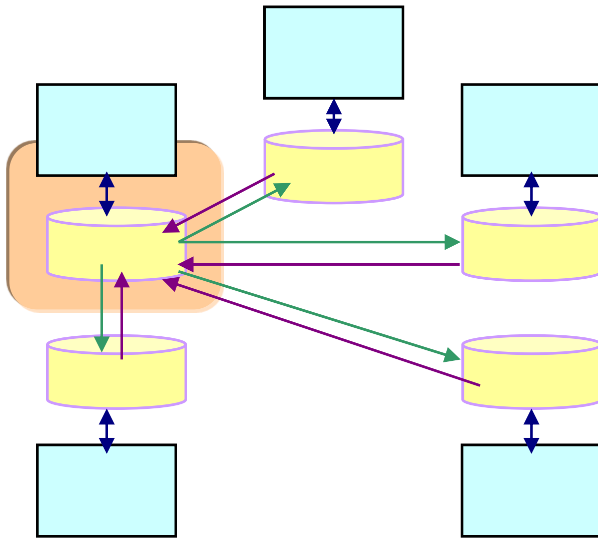
P3: [aktuell, da `Acq(Ly)` durchgeführt] Acq(Ly) R(y)b

Eintritts-Konsistenz



Zusammenfassung der Konsistenzmodelle

Schwach
Freigabe
Eintritt



-**lazy** Strategie der Aktualisierung, d.h. nur Blöcke von `write`'s werden unmittelbar an alle Kopien übermittelt

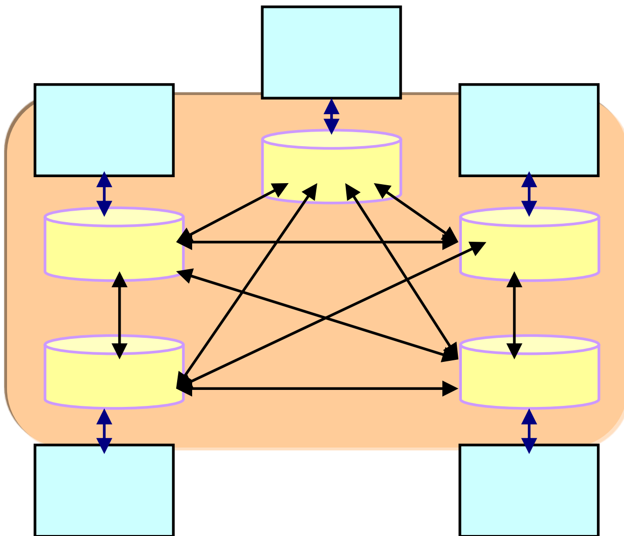
-**update-everywhere** Strategie der Aktualisierung bei der schwachen Konsistenz, d.h. auf jeder Kopie kann eine Aktualisierung der Daten initiiert werden;

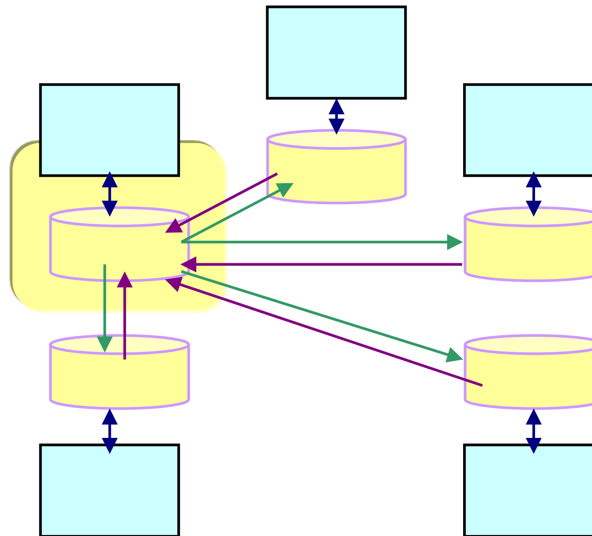
-**Primary-Copy** Strategie bei der Freigabe- und Eintritts-Konsistenz, d.h. eine Kopie, die aktualisieren möchte, muss im Nachhinein oder vorher exklusives Schreibrecht erhalten

Zusammenfassung der Konsistenzmodelle

Sequentiell
Kausal
FIFO

Schwach
Freigabe
Eintritt





Zusammenfassung der Konsistenzmodelle

...die keine Synchronisationsvariable nutzen

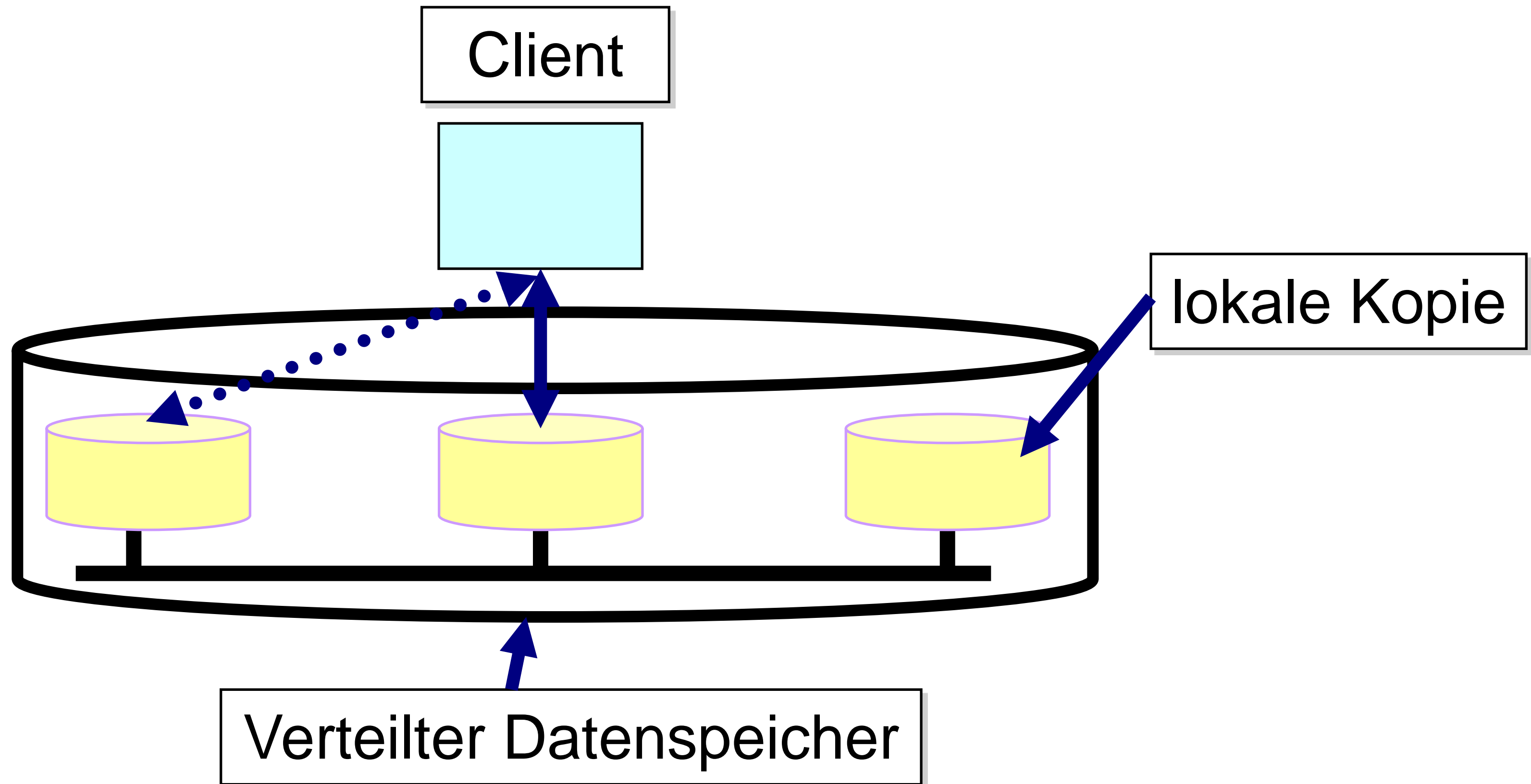
Konsitenz	Beschreibung
Strikt	Absolute Zeitreihenfolge aller gemeinsamen Zugriffe
Linearisier-barkeit	Alle Prozesse sehen alle gemeinsamen Zugriffe in derselben Reihenfolge. Die Zugriffe sind darüber hinaus gemäß einem (nicht eindeutigen) globalen Zeitstempel sortiert.
Sequenziell	Alle Prozesse sehen alle gemeinsamen Zugriffe in derselben Reihenfolge. Die Zugriffe sind nicht der Zeit nach sortiert
Kausal	Alle Prozesse sehen alle kausal verknüpften gemeinsamen Zugriffe in derselben Reihenfolge.
FIFO	Alle Prozesse sehen Schreiboperationen voneinander in der Reihenfolge, in der sie abgesetzt wurden. Schreiboperationen von anderen Prozessen werden möglicherweise nicht immer in der gleichen Reihenfolge gesehen.

Zusammenfassung der Konsistenzmodelle

...die eine Synchronisationsvariable nutzen

Konsitenz	Beschreibung
Schwach	Gemeinsam genutzte Daten sind nur dann verlässlich konsistent, nachdem eine Synchronisierung vorgenommen wurde
Freigabe	Gemeinsam genutzte Daten werden konsistent gemacht, nachdem ein kritischer Bereich verlassen wurde
Eintritt	Gemeinsam genutzte Daten, die zu einem kritischen Bereich gehören, werden konsistent gemacht, sobald ein kritischer Bereich betreten wird

Client-zentriertes Konsistenzmodell



Eventuelle Konsistenz

- **Idee**: über lange Zeitspannen **keine Updates**, dann werden im **Laufe der Zeit** alle Replikate **konsistent** sein, indem sie identische Kopien voneinander werden.

- **Beispiele** für typische Anwendungen, bei denen ein solches Modell ausreicht (meist **wenig Schreibberechtigte** oder/und **selten Veränderungen**)
  1. DNS (Aktualisierung wird langsam weitergegeben)
  2. Web Caching (oft akzeptierte Inkonsistenz)

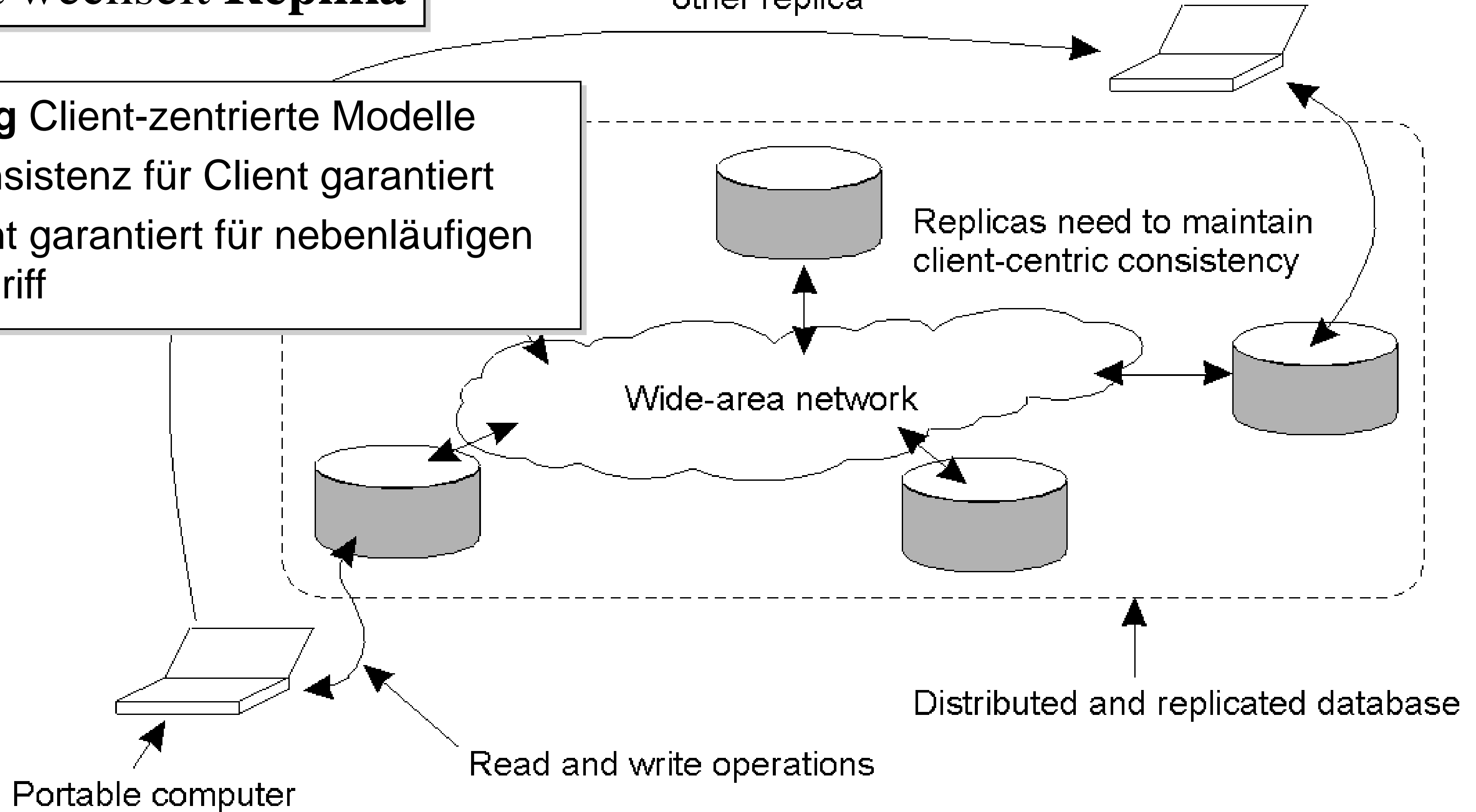
- **Vorteil**: meist sehr **einfach zu implementieren**, Write-Write-Konflikte treten meist nicht auf

Problem

Problem
Client wechselt **Replika**

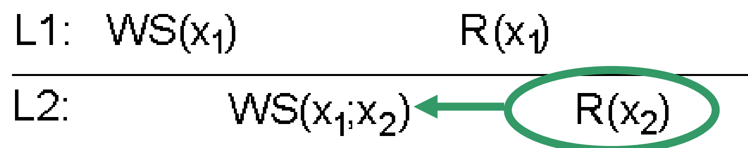
Inkonsistentes Verhalten möglich

Lösung Client-zentrierte Modelle
- Konsistenz für Client garantiert
- nicht garantiert für nebenläufigen Zugriff

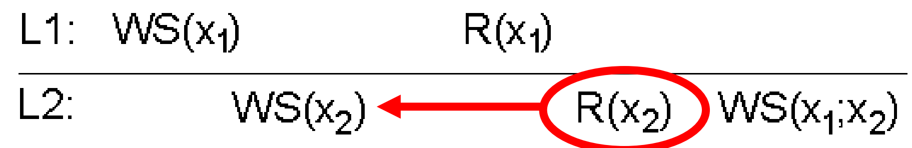


I: Monotones Lesen

- **Regel**: Wenn ein Prozess den Wert einer Variablen `x` liest, dann wird jede weitere `Read`-Operation denselben oder einen neueren Wert von `x` liefern.
- **Beispiel**: Zugriff auf Email-Box von verschiedenen Orten
- **Legende**:
  - `Li:` sind Kopien **eines** Datenspeichers
  - x_i: der `i`-te Zugriff auf `x`
  - `WS` ist Schreibset, `R` Leseoperation, `W` Schreiboperation jeweils des selben (!) Prozesses `P`
  - $WS(x_1;x_2)$: es ist $WS(x_1)$ Teil von $WS(x_2)$



Monotones Lesen garantiert



Monotones Lesen nicht garantiert

II: Monotones Schreiben

- **Regel**: Eine Schreiboperationen durch einen Prozess auf ein Datenelement `x` (ggf. auf mehreren Replika) ist abgeschlossen, bevor eine nachfolgende Schreiboperationen auf `x` durch denselben Prozess stattfindet
- **Keine Daten-zentrierte FIFO-Konsistenz**, da hier Konsistenz nur für einen Prozess berücksichtigt wird!
- **Beispiel**: Aktualisierung einer Software-Bibliothek

L1: $W(x_1)$

L2: $W(x_1)$ ← $W(x_2)$

Monotones Schreiben

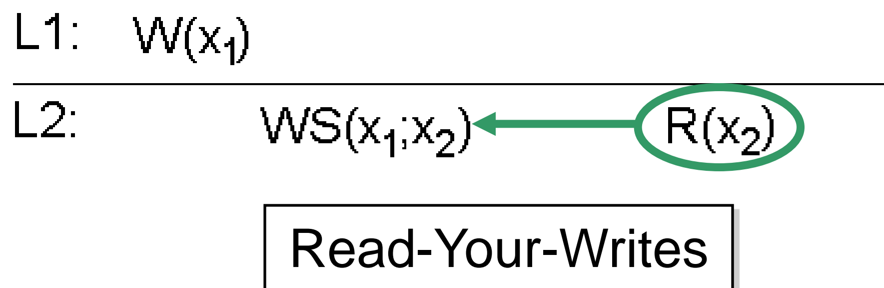
L1: $W(x_1)$

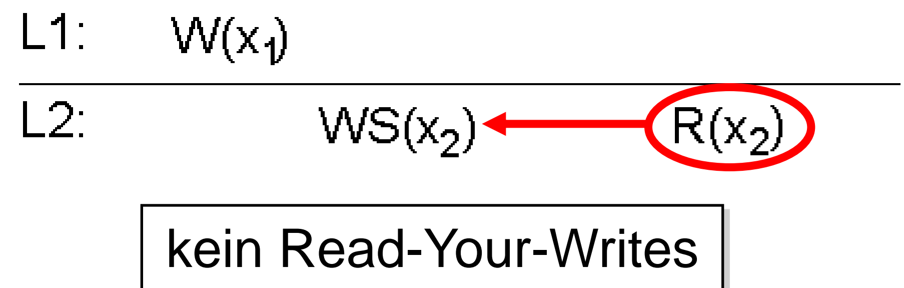
L2: ? ← $W(x_2)$

kein Monotones Schreiben

III: Read-Your-Writes

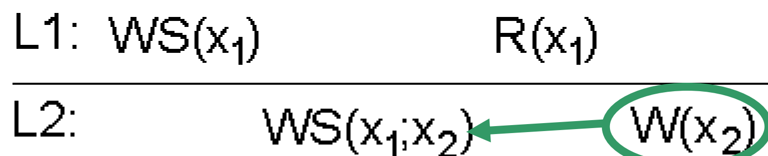
- **Regel**: Die Wirkung einer Schreiboperation durch einen Prozess auf ein Datenelement x wird immer von einer nachfolgenden Leseoperation auf x durch denselben Prozess gesehen (unabhängig davon, wo diese Leseoperation stattfindet)
- **Beispiel**: Aktualisierung von Web-Seiten oder Passwörtern



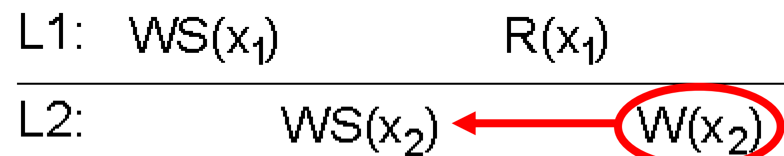


IV: Write Follows Read

- **Regel**: Eine Schreiboperation durch einen Prozess auf einem Datenelement `x`, die einer vorhergehenden Leseoperation für `x` folgt, findet garantiert auf demselben oder einem neueren Wert von `x` statt, der gelesen wurde
- **Beispiel**: Lesen von Newsgroups. Antworten können nur geschrieben werden, wenn die Frage gelesen wurde.



Write Follows Read



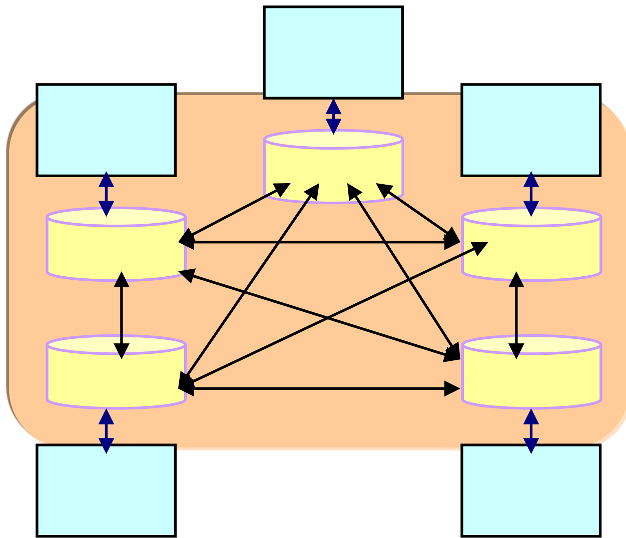
kein Write Follows Read

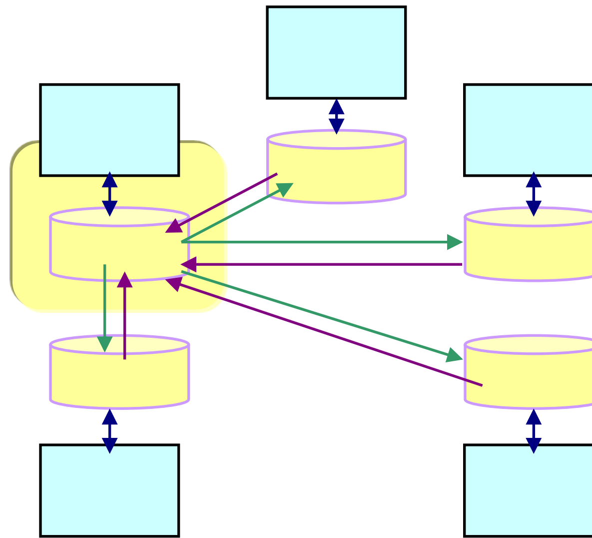
Zusammenfassung der Konsistenzmodelle

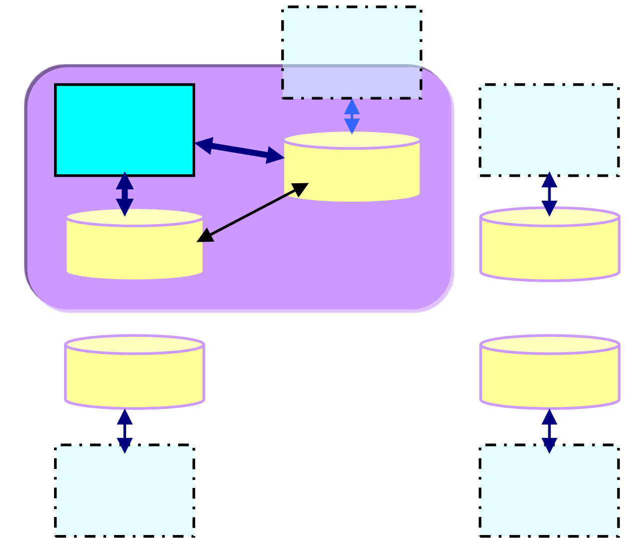
Sequentiell
Kausal
FIFO

Schwach
Freigabe
Eintritt

Eventuell







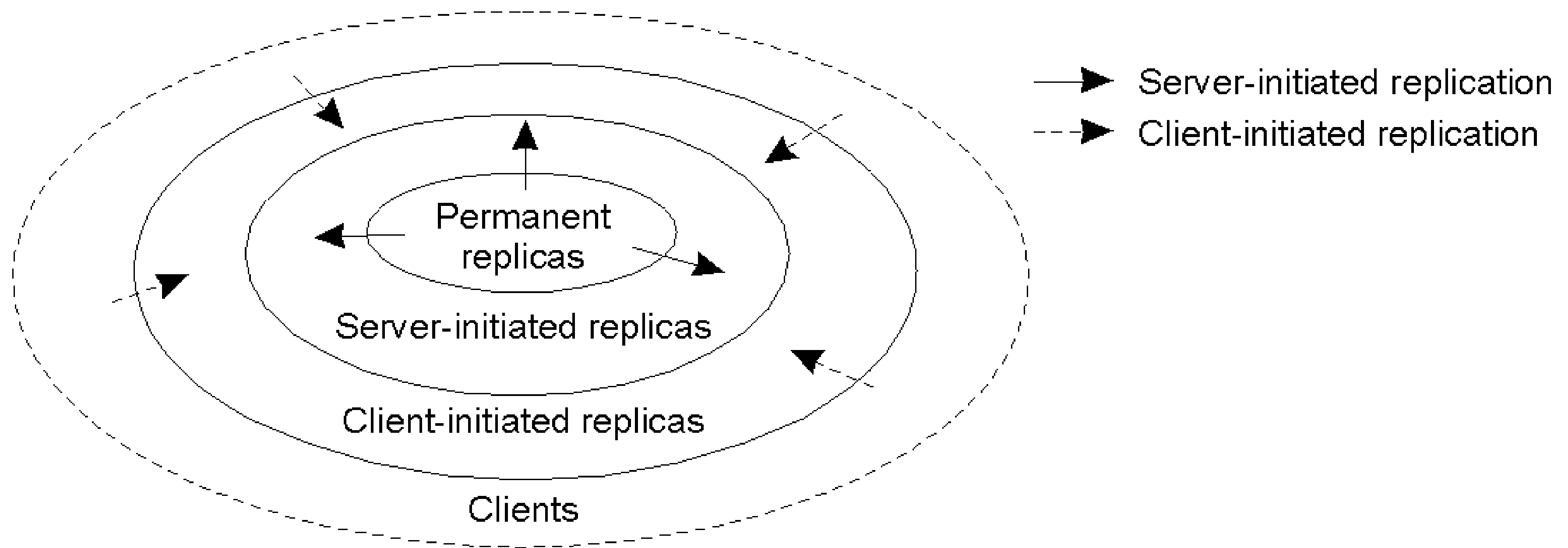
Verteilungsprotokolle

- Welche **Möglichkeiten** gibt es nun, Replikate zu verteilen?
- Wir betrachten **Verteilungsprotokolle** und anschließend spezielle **Konsistenzerhaltungsprotokolle**.
- Beim **Design** solcher Protokolle müssen verschiedene Fragen beantwortet werden
  1. Wo, wann und von wem werden die **Replikate platziert**?
  2. Wie werden **Updates propagiert**?

Platzierung der Replikate

Es können **drei verschiedene Arten** von Kopien unterschieden werden

1. **Permanente** Replikate
2. **Server-initiierte** Replikate
3. **Client-initiierte** Replikate



Permanente Replikate

- **Grundlegende Menge** von Replikaten, die meist **beim Design** eines Datenspeichers schon angelegt werden
- **Beispiele**:
  - replizierte Web-Site (Client merkt nichts von der Replikation),
  - Mirroring (Client sucht bewusst ein Replikat aus)
- Meist nur **sehr wenige** Replikate

Server-initiierte Replikate

- **Kurzfristig** initiiert **bei hohem Bedarf**, meist in der (Netz-) Gegend, in der der Bedarf auftritt
- Wichtige Grundlage für das Geschäftsmodell von *Web Hosting Services* (dynamisches replizieren von Daten)
- Schwierige Entscheidung: **wann und wo** sollen die Replikate **erzeugt** werden bzw. wann **gelöscht** werden?
- Dieser Ansatz kann **permanente Replikas ersetzen**, wenn garantiert ist, dass immer mindestens ein Server ein Datum vorrätig hält.

Client-initiierte Replikation

- Meist als **(Client) Cache** bezeichnet
- Management des Caches bleibt völlig dem Client überlassen, d.h., der **Server kümmert sich nicht um Konsistenzerhaltung**
- **Einziger Zweck**: Verbesserung der Datenzugriffszeiten
- Daten werden meist für **begrenzte Zeit** gespeichert (verhindert permanenten Zugriff auf alte Kopie)
- Der Cache wird meist auf der **Client-Maschine platziert**, oder zumindest in der **Nähe** von vielen Clients.

Propagierung von Updates

- Die **Implementierung** des jeweiligen Konsistenzmodells ist z.T. **nicht** ganz **einfach** und hängt von der Art und Weise ab, wie die Updates lokaler Kopien durchgeführt wird.
- Auch die **Eigenschaften der Kommunikationsoperationen** (FIFO/kausal konsistenter/total geordneter Multicast) beeinflussen das Ergebnis.
- Updates werden **generell von einem Client** auf einer Replika durchgeführt.
- Diese müssen dann an die anderen Replikas **weiter gegeben** werden.
- Verschiedene **Design-Gesichtspunkte** für die entsprechenden Protokolle
  - **Was** wird zu den anderen Replikaten propagiert?
  - Wird **push oder pull** eingesetzt?
  - **Unicast oder Multicast**?

Was wird propagiert ?

- **Spontan** würde man sagen, dass derjenige Server, dessen Replikat geändert wurde, diesen neuen Wert **an alle anderen schickt**. („gierige“ Möglichkeit)
- Das **muss aber nicht** unbedingt so gemacht werden.
- **Alternativen**:
  - Sende **nur eine Benachrichtigung**, dass ein Update vorliegt (wird von Invalidation Protocols verwendet und benötigt sehr wenig Bandbreite). Updates erfolgen nur bei Bedarf („faule“ Möglichkeit)
  - **Transferiere** die das **Update auslösende Operation** (z.B. „`sortiere(Daten)`“) zu den anderen Servern (**aktive Replikation**: benötigt ebenfalls minimale Bandbreite, aber auf den Servern wird mehr Rechenleistung erforderlich)

Push oder Pull ?

- **Push**:
  - die Updates werden auf **Initiative des Servers**, bei dem das Update vorgenommen wurde, verteilt.
  - Die anderen Server schicken **keine Anforderungen** nach Updates
  - Typisch, wenn ein **hoher Grad an Konsistenz** erforderlich ist
- **Pull**: umgekehrtes Vorgehen
  - Server/Clients **fragen nach neuen Updates** für Daten
  - Oft von Client Caches verwendet

Aspekt	**Push-basiert**	**Pull-basiert**
Status am Sever	Liste mit Client-Repliken und -Caches	Keine
Gesendete Nachrichten	Aktualisierung (und möglicherweise später die Aktualisierung laden)	Anfragen und Aktualisieren
Antwortzeit auf dem Client	Unmittelbar (oder Aktualisierung-Laden-Zeit)	Aktualisierung-Laden-Zeit

Unicast oder Multicast ?

- **Unicast**: sende eine Nachricht mit demselben Inhalt an jeden Replika-Server
- **Multicast**: sende nur eine einzige Nachricht und überlasse dem Netz die Verteilung;
  Meist wesentlich effizienter, insbesondere in LANs

- **Multicast** wird meist mit **Push-Protokollen** verbunden, die Server sind dann als Multicast-Gruppe organisiert
- **Unicast** passt besser zu **Pull**, wo immer nur ein Server nach einer neuen Version eines Datums fragt.

Protokolle zur Konsistenzerhaltung

- Wie lassen sich nun die **verschiedenen Konsistenzmodelle** implementieren?
- Dazu **benötigt** man **Protokolle**, mit deren Hilfe sich die verschiedenen Replika-Server abstimmen.
- Im folgenden **Beispiele** für sequentielle Konsitenz und eventuelle Konsistenz
- Man unterscheidet **zwei grundlegende Ansätze** für diese Protokolle:
  1. **Primary-based** Protocols (`Write`-Operationen gehen immer an dieselbe (primäre) Kopie)
  2. **Replicated-Write** Protocols (`Write`-Operationen gehen an beliebige Kopien)

Primary-Based Protocols

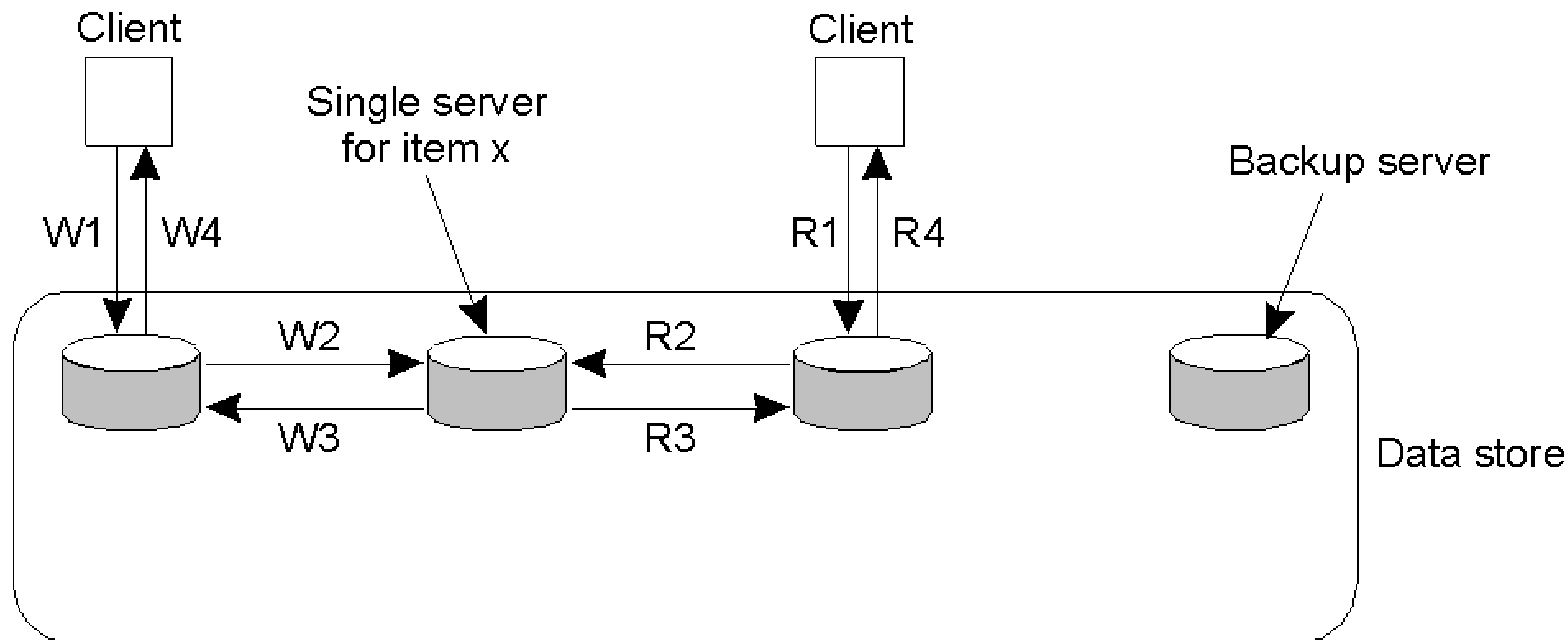
- Wenn alle `Write`-Operationen immer nur an eine Kopie gehen, kann man **noch einmal unterscheiden**,
  - Ob diese **Kopie immer am selben entfernten Platz** bleibt
  - Ob die primäre **Kopie** zu dem schreibenden Client **verlagert** wird
- Dementsprechend werden unterschiedliche Algorithmen und Protokolle verwendet:
  - Remote-Write Protocoll
  - Local-Write Protocoll

Remote-Write Protocols

- alle Updates auf **einem einzigen** entfernten **Server**
- `Read`-Operationen auf **lokalen Kopien** (auch *primary-Backup protocoll* genannt)
- Nach Update der primären Kopie
  1. Aktualisierung der Kopien (z.B. Backup-Server),
  2. Bestätigung zurück an primäre Kopie,
  3. primäre Kopie informiert den Client

  damit bleiben alle Kopien konsistent
- **Problem**: Performance (beim Client), deshalb wird auch *non-blocking Update* eingesetzt (aber hier wieder **Problem mit Fehlertoleranz**)
- Beste Umsetzung für **sequentielle Konsistenz**

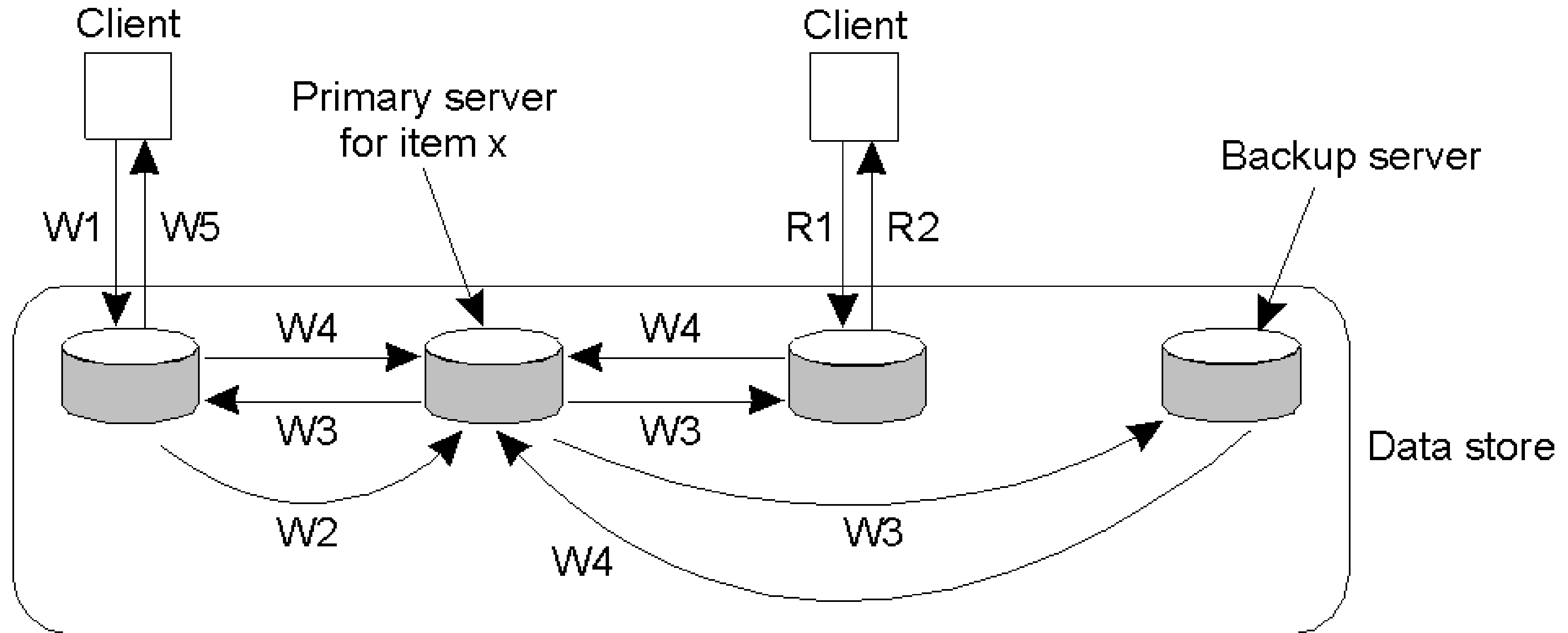
Ablauf: ohne lokaler Kopie (pull)



W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Ablauf: mit lokaler Kopie (push, blocking)



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

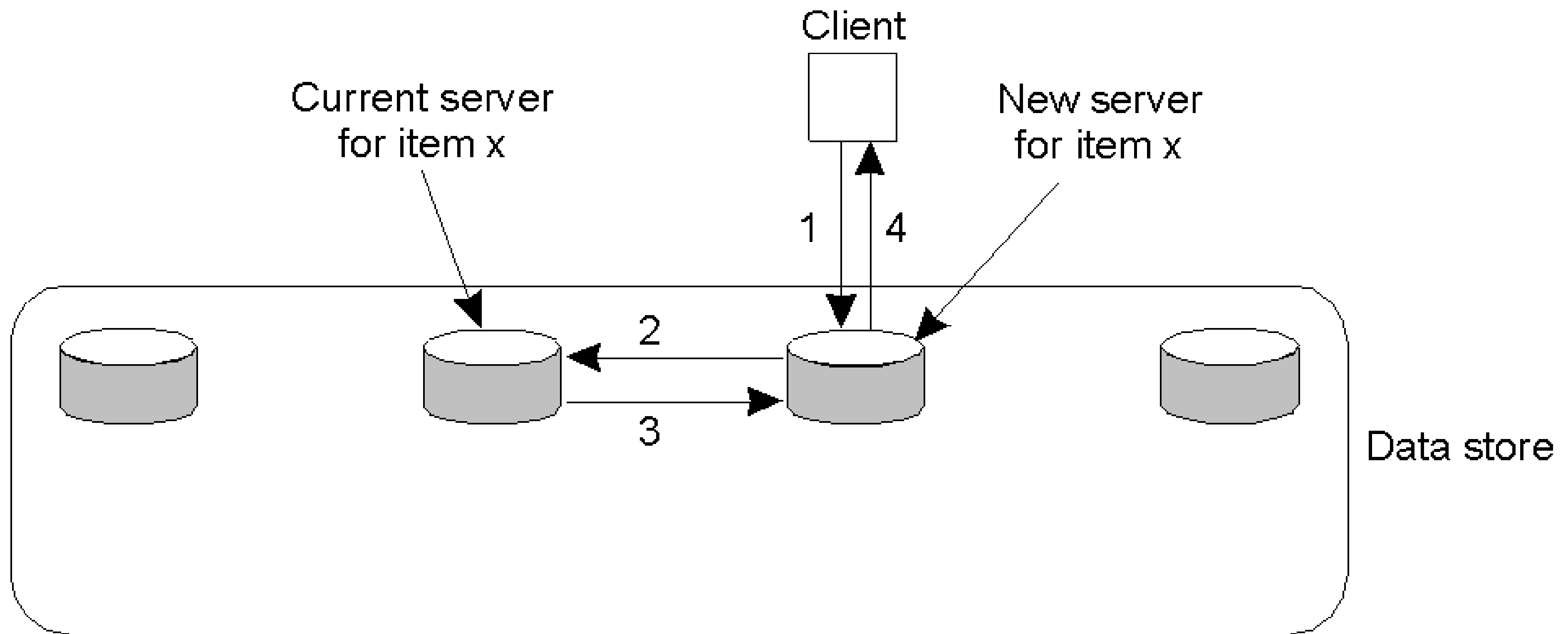
R1. Read request
R2. Response to read

Local-Write Protocols

- Jeder Prozess, der ein **Update ausführen** will, lokalisiert die **primäre Kopie** und **bewegt diese** dann an seinen eigenen Platz.
- Realisiert **sequentielle Konsistenz**
- Gutes Modell auch für mobile Benutzer:

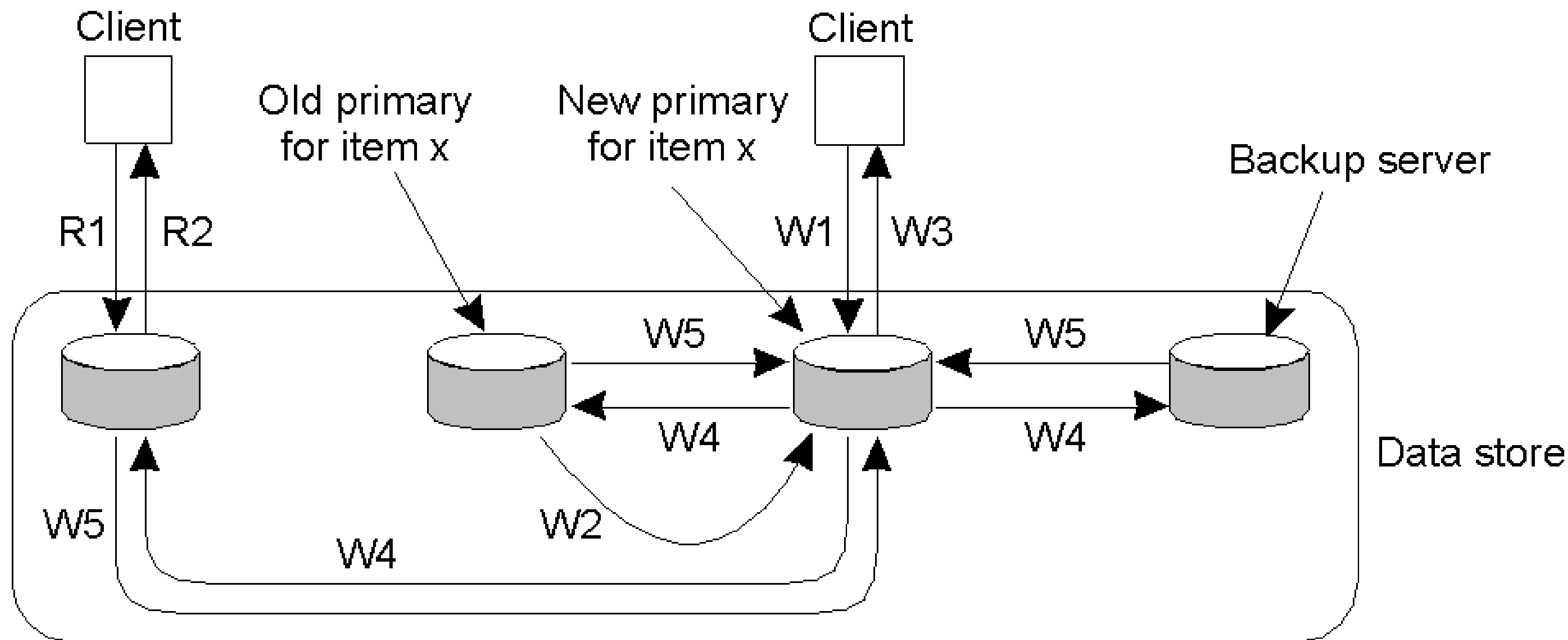
1. hole primäre Kopie
2. breche Verbindung ab
3. Arbeite
4. baue später Verbindung wieder auf
5. keine Updates durch andere Prozesse!

Ablauf: nur genau eine Kopie (pull)



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Ablauf: Migration der primären Kopie (push, nonblocking)



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Replicated-Write Protocols

- Bei dieser Art von Protokollen können `Write`**-Operationen** auf **beliebigen Replikaten** ausgeführt werden.
- Es muss dann **entschieden** werden, welches der **richtige Wert** eines Datums ist.
- Realisiert **sequentielle Konsistenz**
- Zwei Ansätze:
  - **Active Replication**: eine Operation wird an alle Replikas weiter gegeben
  - **Quorum-based**: es wird abgestimmt, die Mehrheit gewinnt

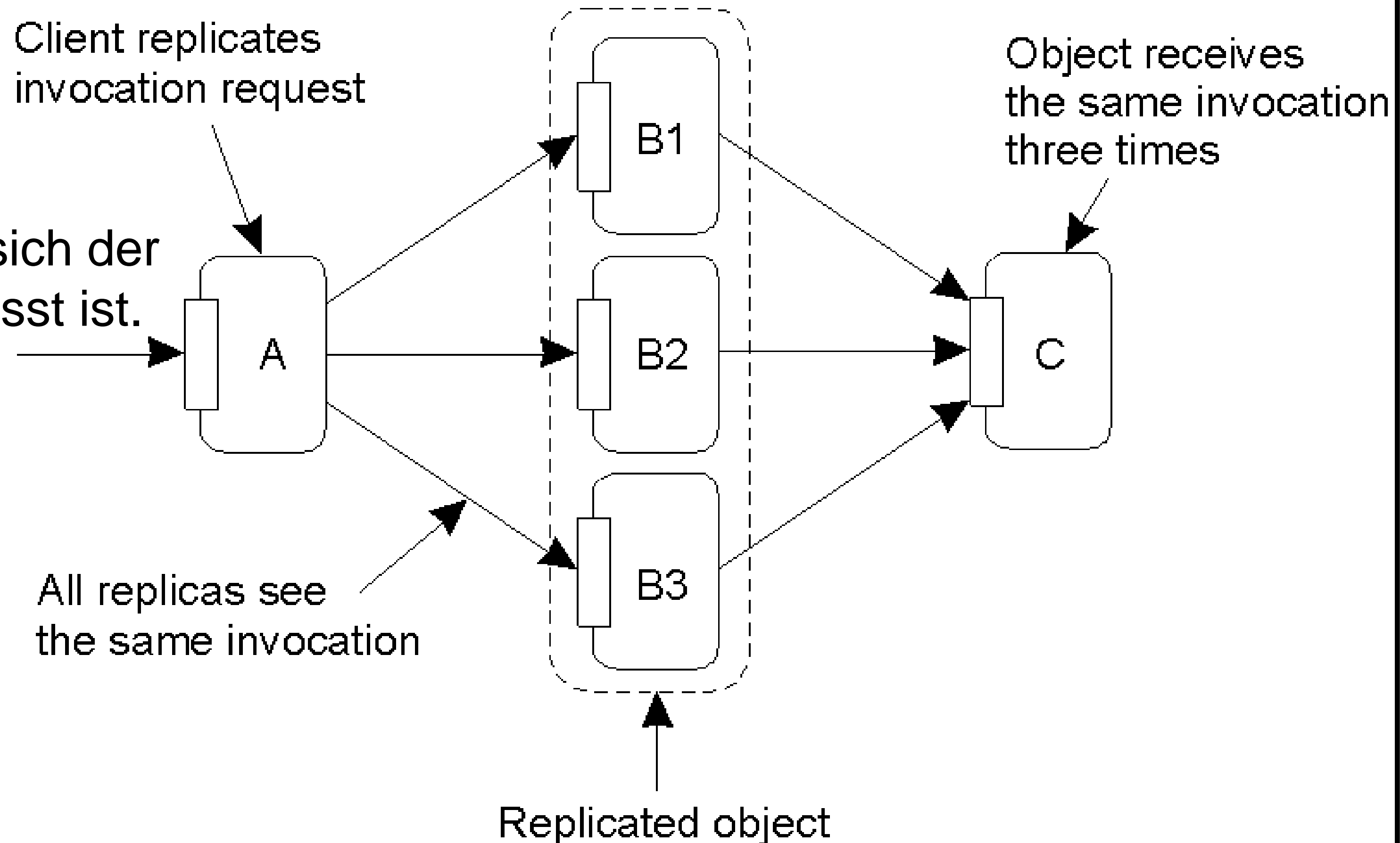
Aktive Replikation

- **Jede Replika** besitzt **einen Prozess**, der die Updates durchführt
- **Updates** werden meist **als Operation** propagiert
- **Wichtigstes Problem**: alle **Updates** müssen auf allen Replikas **in derselben Reihenfolge** ausgeführt werden!
  - Es wird **Multicast mit totaler Ordnung** benötigt(!), implementiert z.B. mittels Lamport-Uhren
  - Lamport-Zeitstempel unterstützen **Skalierung nicht gut**
  - **Alternative**: zentraler Prozess (Sequenzer), der die Sequentialisierung übernimmt (aber nicht besser skaliert...)
  - **Kombination** aus beiden Ansätzen hat sich als **brauchbar** erwiesen

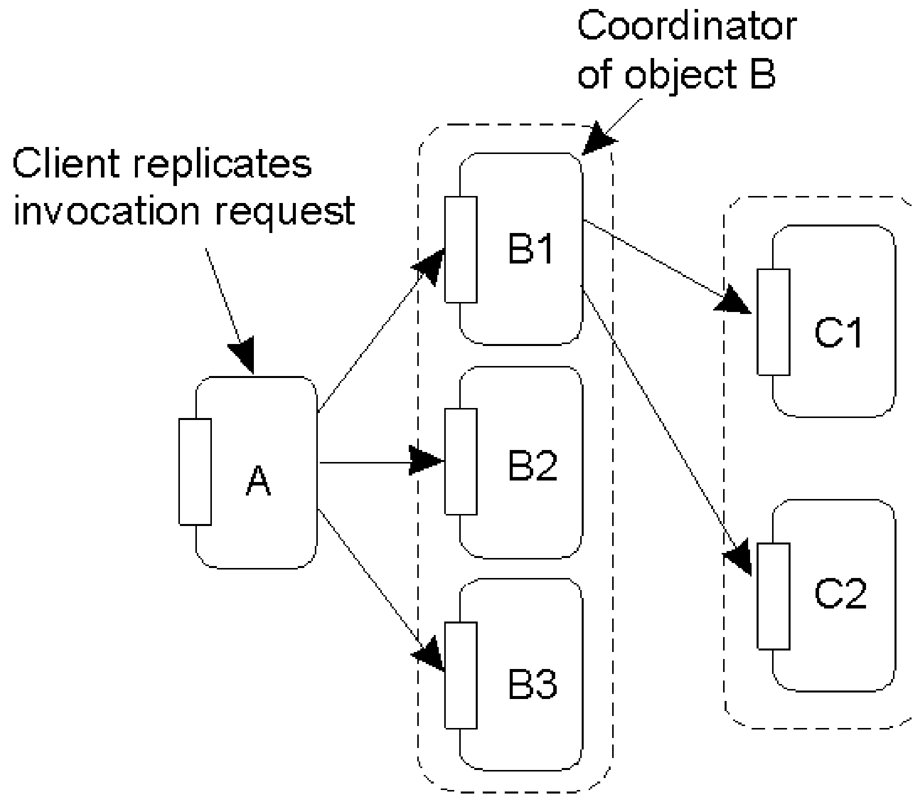
Aktive Replikation: Problem

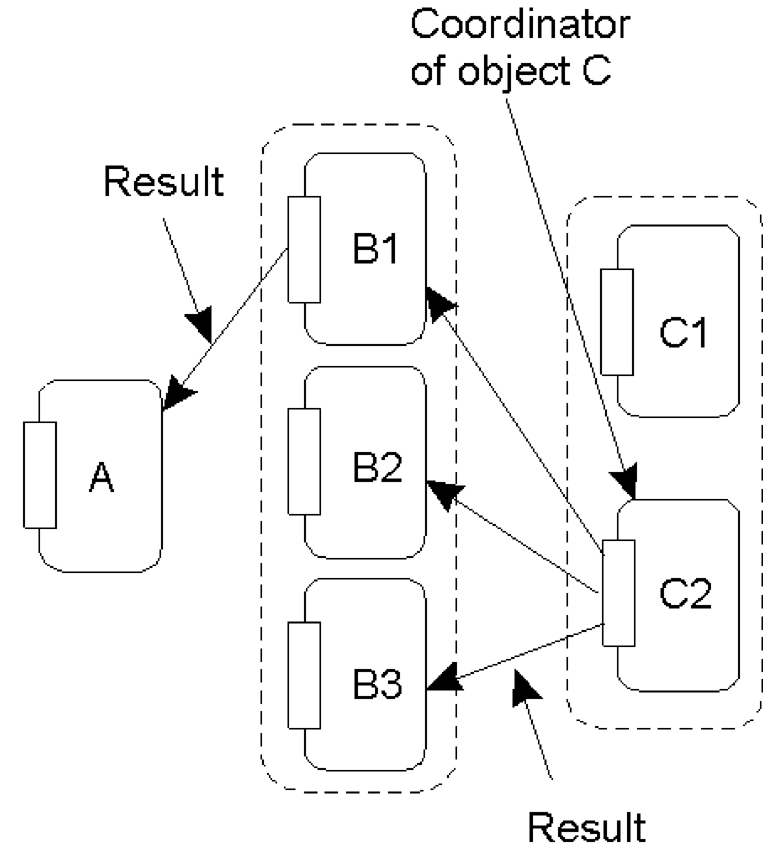
- Was passiert, wenn ein **repliziertes Objekt ein anderes Objekt** aufruft?

  Jede Replika ruft das Objekt auf!
- **Lösung**: verwende eine Middleware, die sich der Replikation bewusst ist.
- Löst auch das Problem der Verarbeitung der Antworten



Koordination der replizierten Objekte





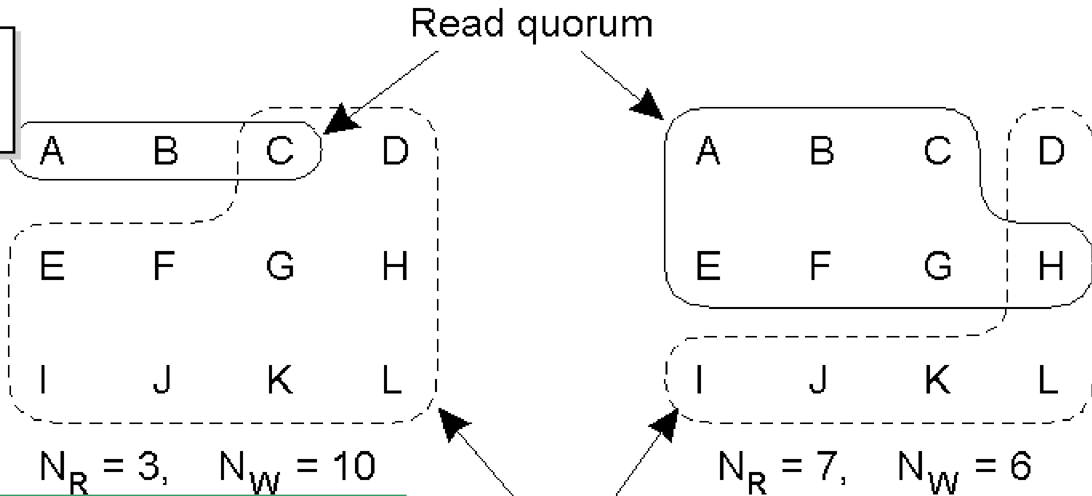
Weiterleitung eines Aufrufs von einem replizierten Objekts an ein anderes: Nur Ein Replikt leitet `request` weiter

Rückgabe der Antwort: Nur Ein Replikt leitet `reply` weiter

Quorum-Based Protocols

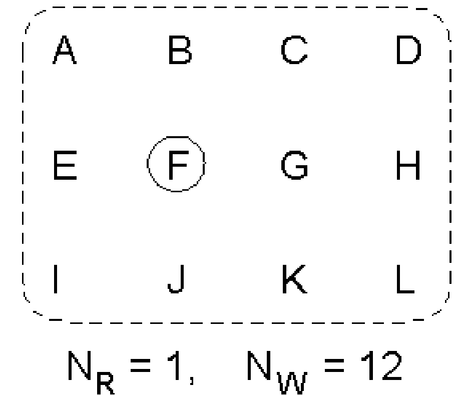
- **Idee**: Clients müssen zur Ausführung einer `Read`- oder `Write`-Operation die **Erlaubnis mehrerer Server** einholen
- Jedes Objekt besitzt eine **Versionsnummer**.
- Wenn der Client ein `Read` oder `Write` durchführen will, muss er eine **Erlaubnis** erhalten. (Bei `Read` Übereinstimmung der Versionsnummer, bei `Write` Zustimmung zur Aktualisierung)
- **Regeln**: $N_W \geq N/2 + 1$; $N_W + N_R \geq N + 1$; $N_R \geq 1$;
- Ist das der Fall, kann kein anderer Client eine entsprechende Operation ausführen.

Beispiele für N = 12



Korrektes Read Quorum: 10 + 3 ≥ 12 + 1

`Write`-`Write` Konflikt: 12/2 + 1 ≥ 6

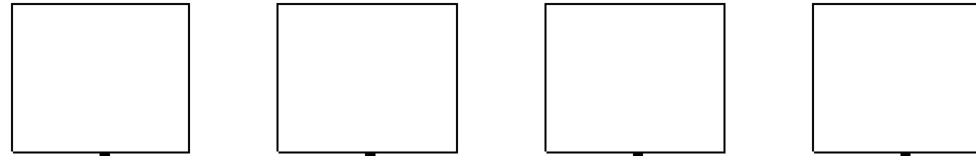


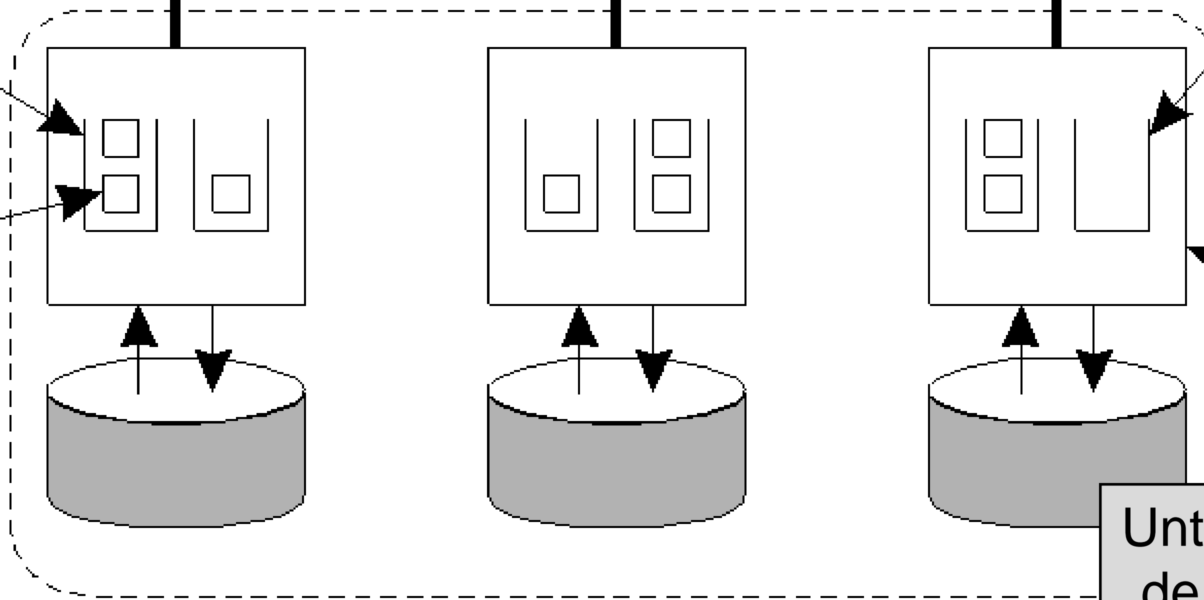
Read One *Write All*: 12 + 1 ≥ 12 + 1

Kausal konsistente lazy-Replikation

- Realisiert **eventuelle Konsistenz**
- gleichzeitig **kausale Beziehungen** zwischen Operationen **berücksichtigt**
- Verwendet in **kausal konsistentem Datenspeicher**
- Nutzt lazy-Form der Replikation, um Aktualisierungen weiterzugeben
- Implementierung: mit Hilfe von **Vektor-Zeitstempeln**
- Clients dürfen **miteinander kommunizieren**, müssen aber **Informationen über Operationen (**auf dem Datenspeicher) austauschen
- **In der Praxis**:
  - Menge der Clients realisiert als **Menge von Frontends** des Datenspeichers
  - Frontends tauschen Informationen aus
  - „reine“ Clients wissen überhaupt nichts über Konsitenz und Replikation

Systemmodell



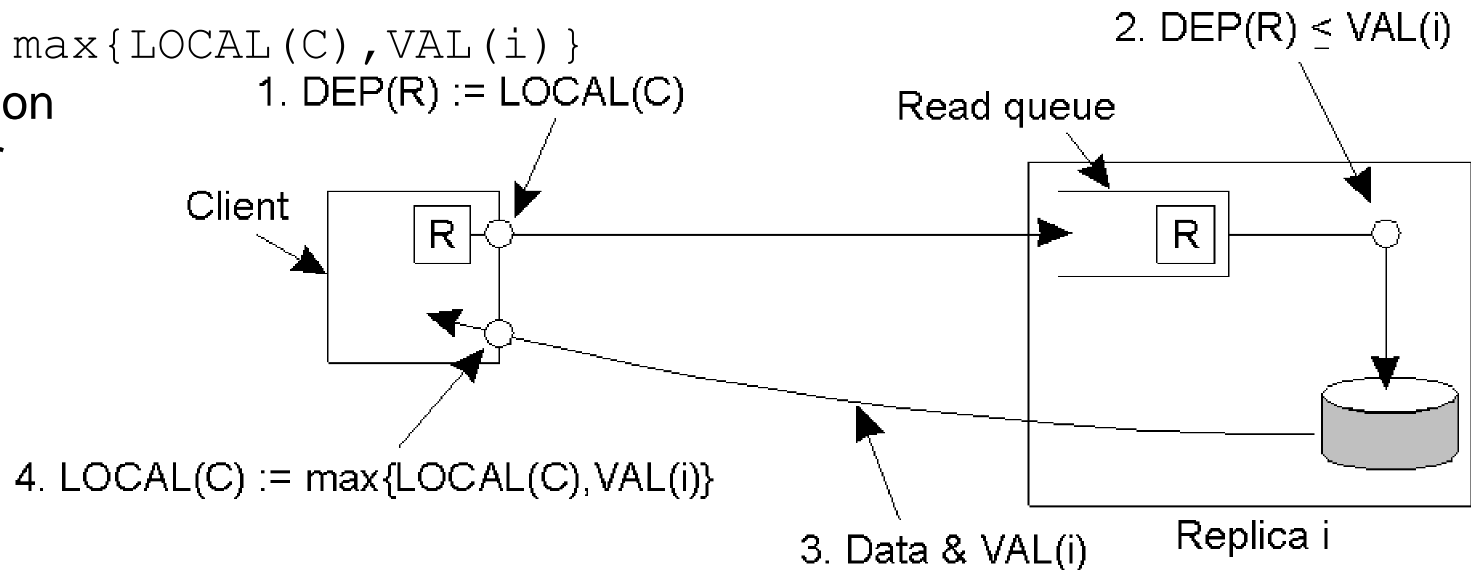


Unter Berücksichtigung der implementierten Konsistenz!

Read-Operation

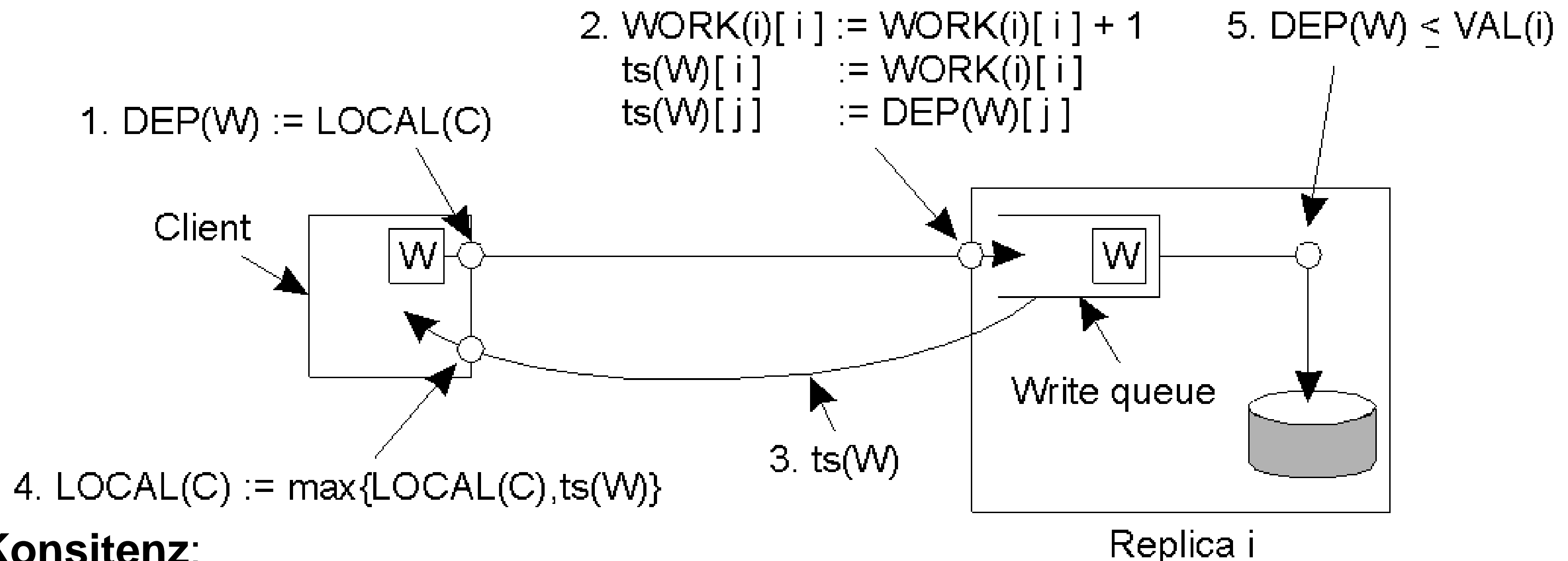
- **Vektor-Zeitstempel**
  - LOCAL(C)[i] aktuellster Stand von L_i gesehen von C
  - VAL(i)[i] aktueller Status von Kopie L_i; VAL(i)[j] Aktualisierungen von L_j
  - WORK(i)[i] todo in L_i; WORK(i)[j] todo-Aktualisierungen von L_j
  - DEP(R) Zeitstempel beschreibt die Abhängigkeiten der Operation

1. Zeitstempel DEP(R) der Leseanforderung := LOCAL(C)
2. DEP(R) <= VAL(i): L_i kennt Status von C
3. Kopie L_i gibt Wert und VAL(i) an Client
4. LOCAL(C):=max{LOCAL(C),VAL(i)}
   Synchronisation der Vektoruhr



Write-Operation

1. Zeitstempel DEP(W) der Leseanforderung := LOCAL(C)
2. WORK(i)[i]:=WORK(i)[i]+1: todo-Liste erhöhen
3. ts(W): Rückgabe Zeitstempel (ist Zustand wenn Warteschlange abgearbeitet)
4. LOCAL(C):=max{LOCAL(C),ts(i)}Synchronisation der Vektoruhr
5. DEP(W)<= VAL(i): L_i kennt Status von C



- **Konsitenz**:
  1. ts(W)[i] = VAL(i)[i] + 1:**Operationen**, die von anderen Clients direkt an L_i gesendet wurden und W **vorausgehen**, wurden **verarbeitet**
  2. ts(W)[j]≤VAL(i)[j]für allej≠i: alle **Aktualisierungen**, von denen W **abhängt**, wurden von L_i **verarbeitet**

Weitergabe von Aktualisierungen

- Kopie L_i tritt (**von Zeit zu Zeit**) in Verbindung mit Kopie L_j:
  - L_i sendet alle Operationen ihrer `Write`-Warteschlange und Vektor-Zeitstempel `WORK(i)` an L_j
  - L_j synchronisiert eigenen Vektor `WORK(j)` und L_j kombiniert empfangene `Write`-Operationen mit eigener `Write`-Warteschlange (nicht vorhandene `W`‘s werden eingetragen)
  - L_j sendet alle Operationen ihrer `Write`-Warteschlange und Vektor-Zeitstempel `WORK(j)` an L_i und L_i aktualisiert sich
- L_i **überprüft Ausführbarkeit** von `Read`- und `Write`-Operationen
  - $U := \{W \mid DEP(W)[k] \leq VAL(i)[k]\}$: L_i kennt Status der Auftragegeber
  - Wähle `W‘` aus `U` mit:

    für alle `W` aus `U` gilt: $DEP(W')[k] \leq DEP(W)[k]$,

    d.h. `W‘` ist unabhängig von `W` (nach Ausführung von `W‘` gilt für die anderen `W`‘s immer noch $DEP(W)[k] \leq VAL(i)[k]$)
- **Noch offen**: Löschen in Warteschlangen, d.h. wann ist bekannt, das ein W bei allen ausgeführt wurde ?