TU Braunschweig
Institut für Betriebssysteme
und Rechnerverbund

Verteilte Systeme

Prof. Dr. Stefan Fischer

Kapitel 11:

Konsistenz, Replikation und Fehlertoleranz

Überblick

- Motivation:
  - warum Replikation?
  - warum ein Konsistenzproblem?
- Konsistenzmodelle
- Verteilungs- und Konsistenzprotokolle
- Fehlertoleranz

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.
    - Besserer Schutz gegen zerstörte/gefälschte Daten: gleichzeitiger Zugriff auf mehrere Replikate, das Ergebnis der Mehrheit wird 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

Das große Problem

- 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: News

- Ansicht 1:

Bulletin board: os.interesting		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

- Ansicht 2:

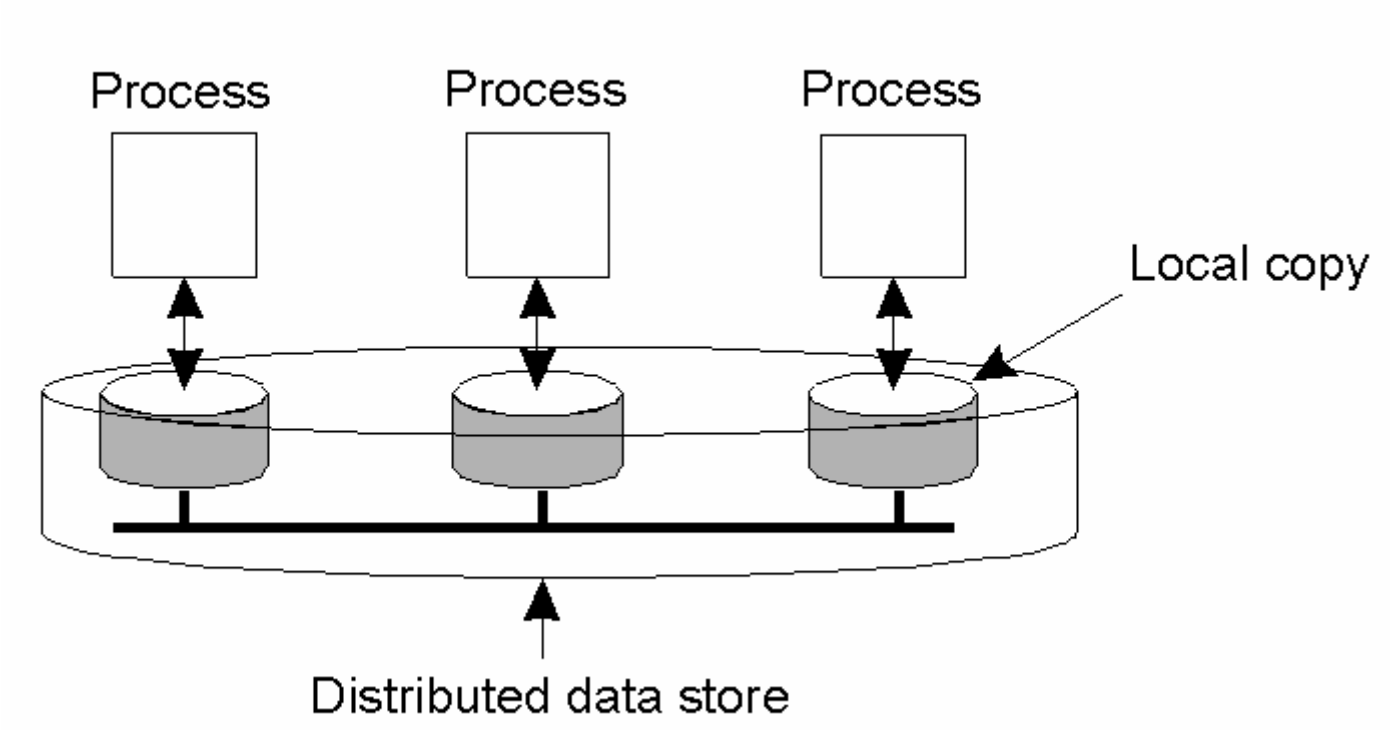
Bulletin board: os.interesting		
Item	From	Subject
20	G.Joseph	Microkernels
21	A.Hanlon	Mach
22	A.Sahiner	Re: RPC performance
23	M.Walker	Re: Mach
24	T.L'Heureux	RPC performance
25	A.Hanlon	Re: Microkernels
end		

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

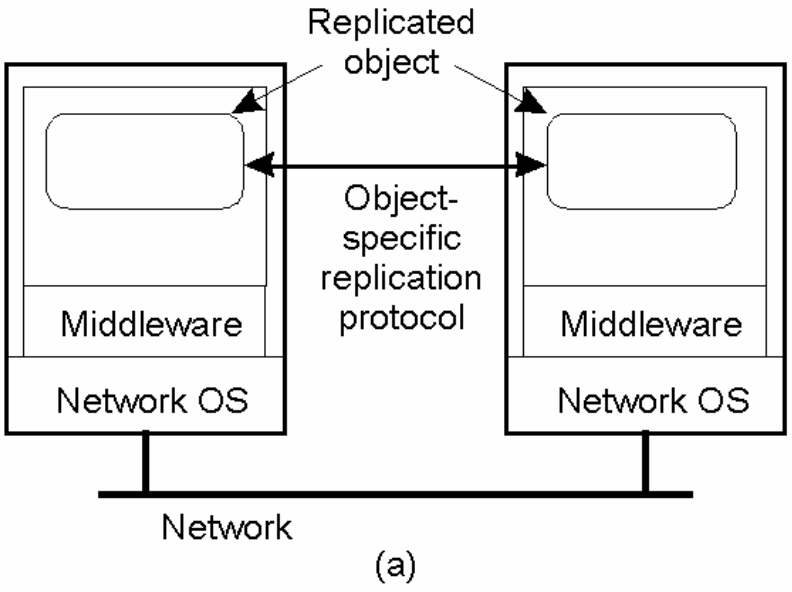
System-Modell

- 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 der letzteren Methode: Anwendungsprogrammierer ist frei von Überlegungen zur Replikation

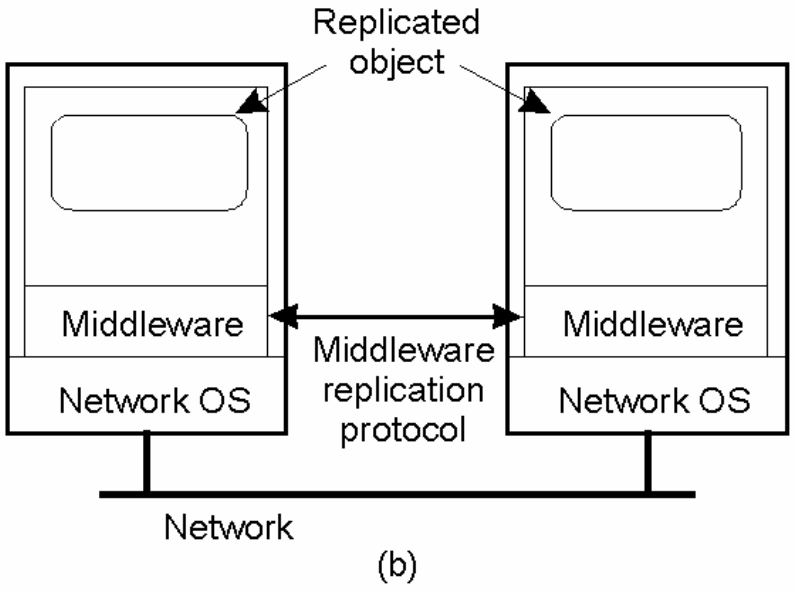
Modell eines verteilten Datenspeichers



Replika-Management



(a)



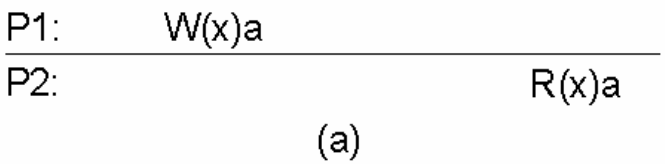
(b)

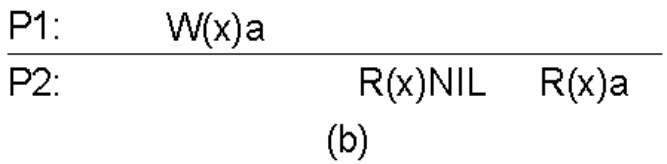
Konsistenzmodelle

- 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

Strikte Konsistenz

- Daten-zentriert
- 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

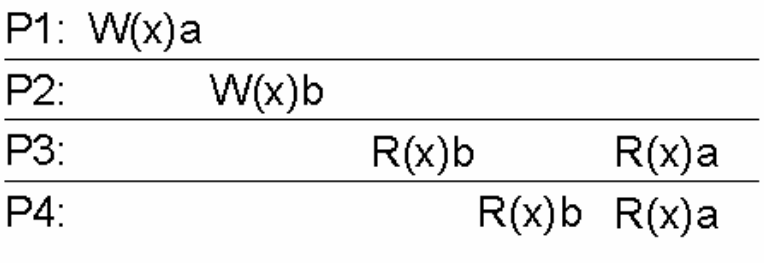




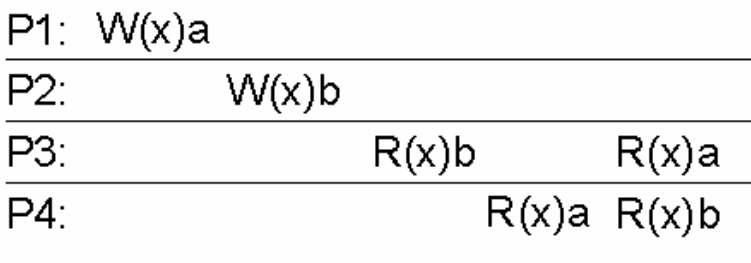
- (a) korrekt
- (b) inkorrekt

Sequentielle Konsistenz

- Etwas schwächeres Modell, aber implementierbar.
- Aussage: 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
- Beispiel: (a) ist korrekt, (b) nicht



(a)



(b)

Linearisierbarkeit

- Liegt in der Striktheit zwischen strikter und sequentieller 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, 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.

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

Kausal konsistent, aber nicht sequentiell oder strikt

Vergleich der Modelle

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

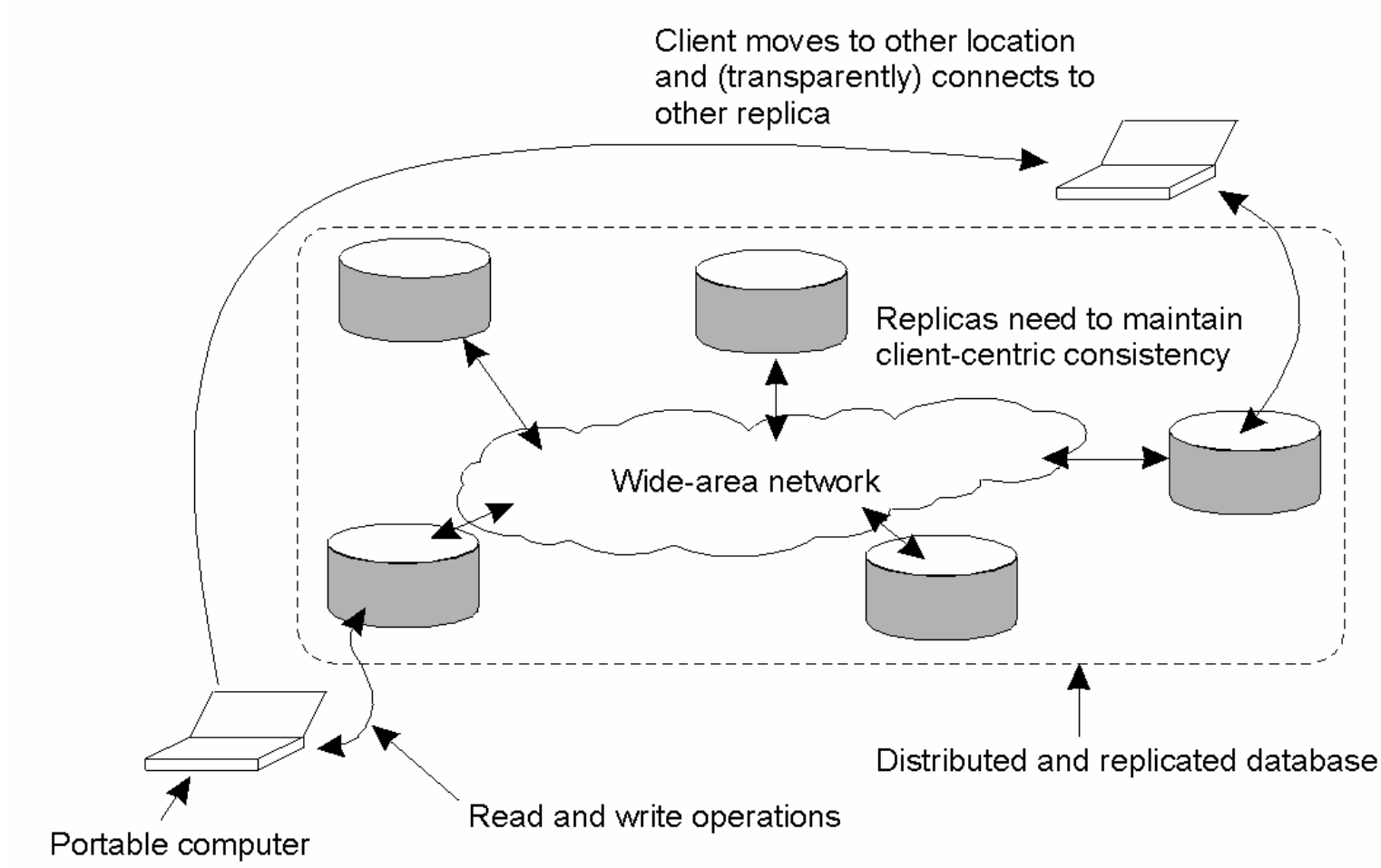
Eventual Consistency

- Idee: über lange Zeitspannen keine Updates, dann werden im Laufe der Zeit alle Replikate konsistent sein
- Beispiele für typische Anwendungen, bei denen ein solches Modell ausreicht
  - DNS
  - Web Caching
- Vorteil: meist sehr einfach zu implementieren, Write-Write-Konflikte treten meist nicht auf

Problem bei Eventual Consistency

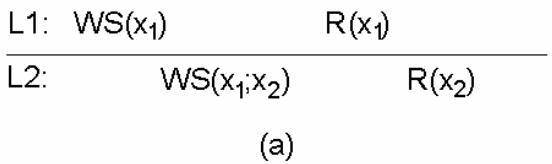
- Ein Problem tritt auf, wenn der Client die Replika wechselt, auf die er zugreift.
- Beispiel: mobiler Benutzer macht Updates, wechselt dann an anderen Platz. Updates sind eventuell noch nicht dort angekommen -> Benutzer stellt inkonsistentes Verhalten fest
- Lösung: client-zentrierte Modelle, bei denen für einen Client die Konsistenz garantiert wird, jedoch nicht für nebenläufigen Zugriff durch mehrere Clients

Illustration des Problems

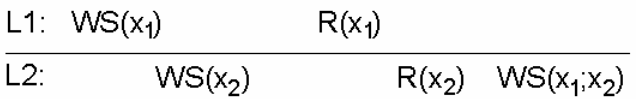


Monotonic Read Consistency

- Beispiel für ein client-zentriertes Konsistenzmodell
- Regel: Wenn ein Prozess den Wert einer Variable x liest, dann wird jede weitere Read-Operation denselben oder einen neueren Wert von x liefern.



(a)



(b)

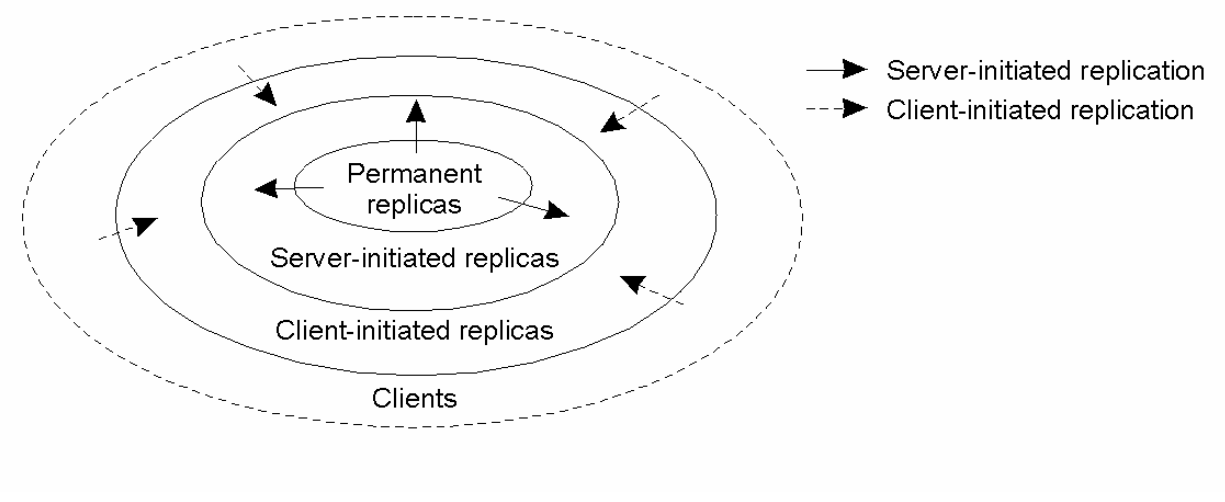
- (a) korrekt (b) nicht korrekt
- Beispiel: Zugriff auf Email-Box von versch. Orten

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
  - Wo, wann und von wem werden die Replikate platziert?
  - Wie werden Updates propagiert?

Platzierung der Replikate

- Es können drei verschiedene Arten von Kopien unterschieden werden
  - Permanente Replikate
  - Server-initiierte Replikate
  - 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*
- Schwierige Entscheidung: wann und wo sollen die Replikate erzeugt werden?
- Existierende Algorithmen verwenden die Namen der gesuchten Dateien, die Anzahl und Herkunft der Requests zur Verteilung von Dateien (Web-Seiten)
- Dieser Ansatz kann permanente Replikas ersetzen, wenn garantiert ist, dass immer mindestens ein Server ein Datum vorrätig hält.

Client-initiierte Replikate

- 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

- 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 pull oder push 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.
- 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)
  - Transferiere die das Update auslösende Operation zu den anderen Servern (benötigt ebenfalls minimale Bandbreite, aber auf den Servern wird mehr Rechenleistung erforderlich)

Pull oder Push?

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

Issue	**Push-based**	**Pull-based**
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

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.
- Man unterscheidet zwei grundlegende Ansätze für diese Protokolle:
  - Primary-based Protocols (Write-Operationen gehen immer an dieselbe Kopie)
  - Replicated-Write Protocols (Write-Operationen gehen an beliebige Kopien)

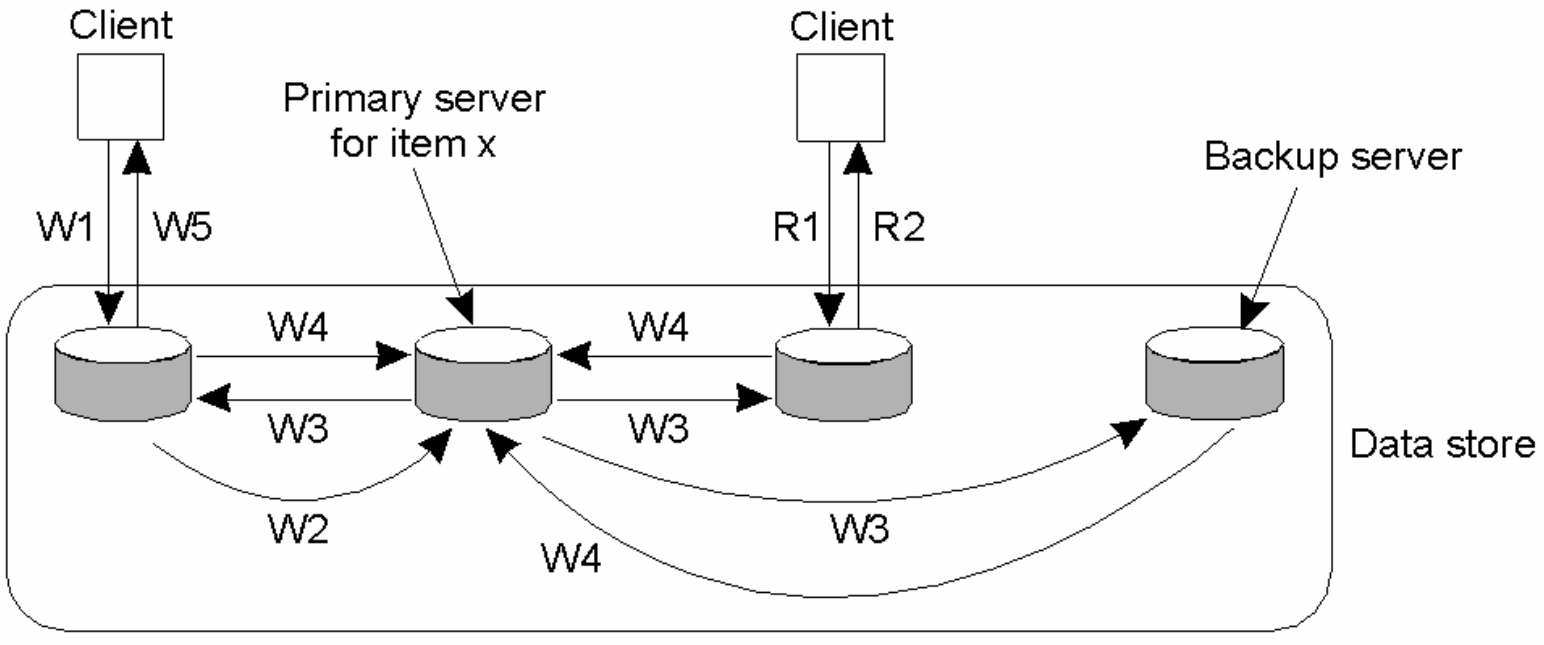
Primary-Based Protocols

- 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 Primärkopie zu dem schreibenden Client verlagert wird.
- Dementsprechend werden unterschiedliche Algorithmen und Protokolle verwendet.

Remote-Write-Protokolle

- alle Updates auf einem einzigen entfernten Server
- Lese-Operationen auf lokalen Kopien
- Nach Update Aktualisierung der Kopien, ACK zurück an Primary, der dann den Client informiert → damit bleiben alle Kopien konsistent
- Problem: Performance, deshalb wird auch non-blocking Update eingesetzt (aber hier wieder Problem mit Fehlertoleranz)
- Beste Umsetzung für sequentielle Konsistenz

Ablauf von Remote Write



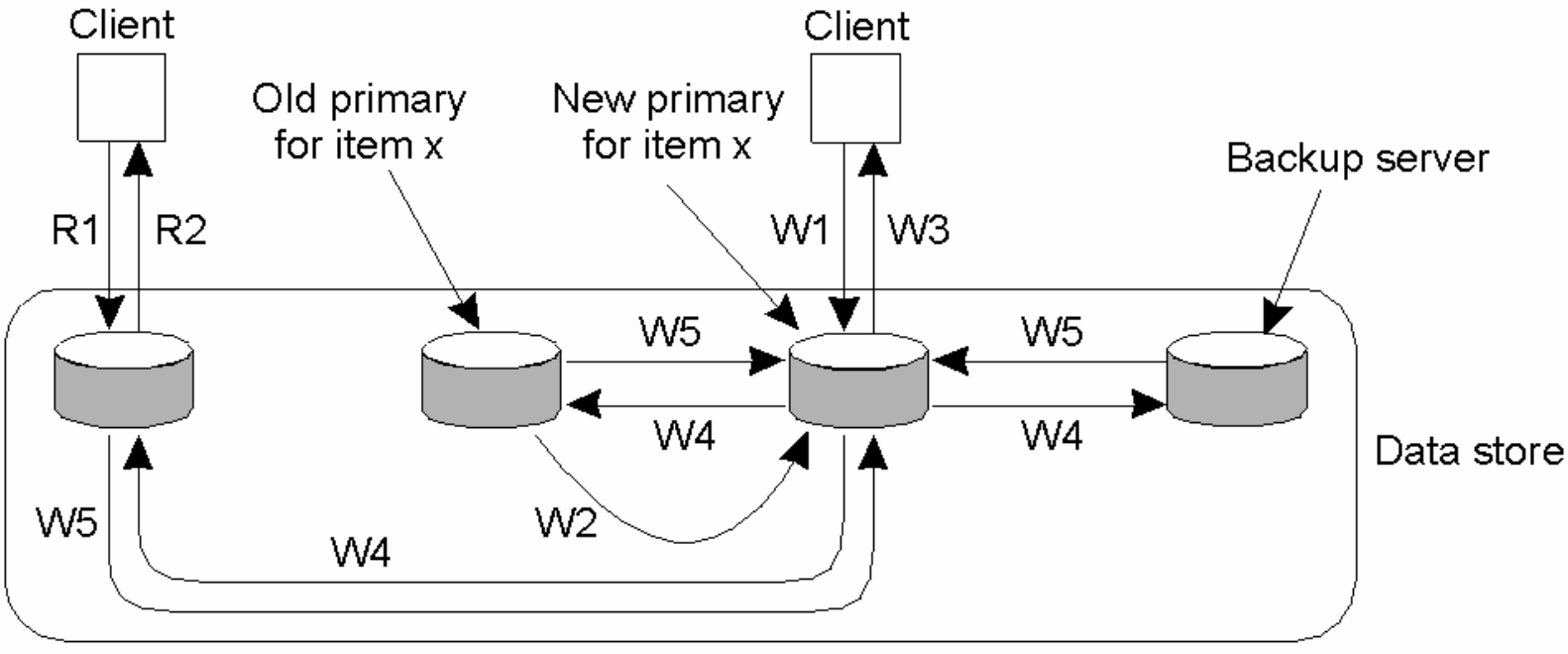
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Local-Write-Protokolle

- Jeder Prozess, der ein Update ausführen will, lokalisiert die Primary Copy und bewegt diese dann an seinen eigenen Platz.
- Gutes Modell auch für mobile Benutzer:
  - hole primary copy
  - breche Verbindung ab
  - Arbeite
  - baue später Verbindung wieder auf
  - keine Updates durch andere Prozesse!

Ablauf von Local Write



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

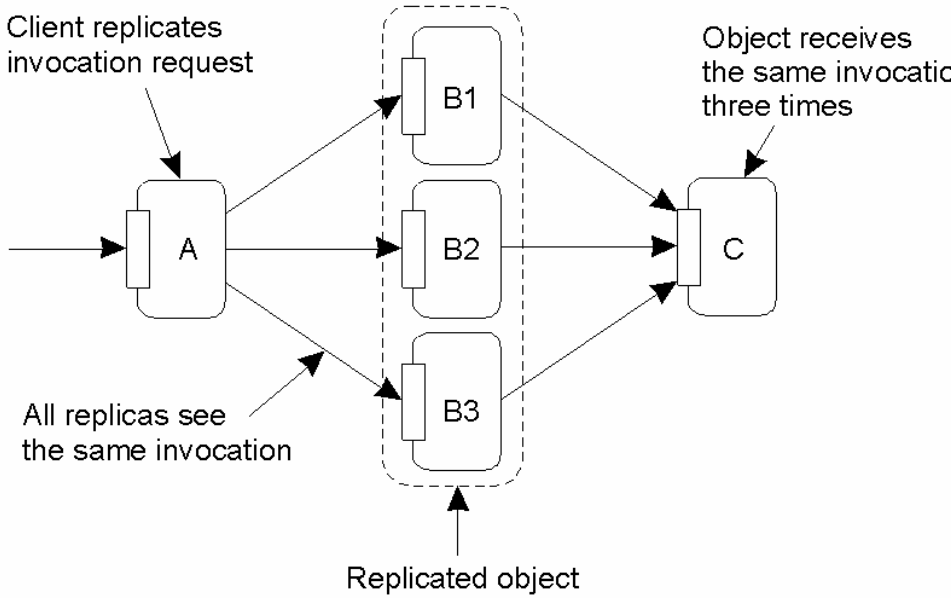
- Bei dieser Art von Protokollen können Write-Operationen auf beliebigen Replikaten ausgeführt werden.
- Es muss dann entschieden werden, welches der richtige Wert eines Datums ist.
- Zwei Ansätze:
  - Active Replication: eine Operation wird an alle Replikas weiter gegeben
  - Quorum-based: es wird abgestimmt,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

Aktive Replikation

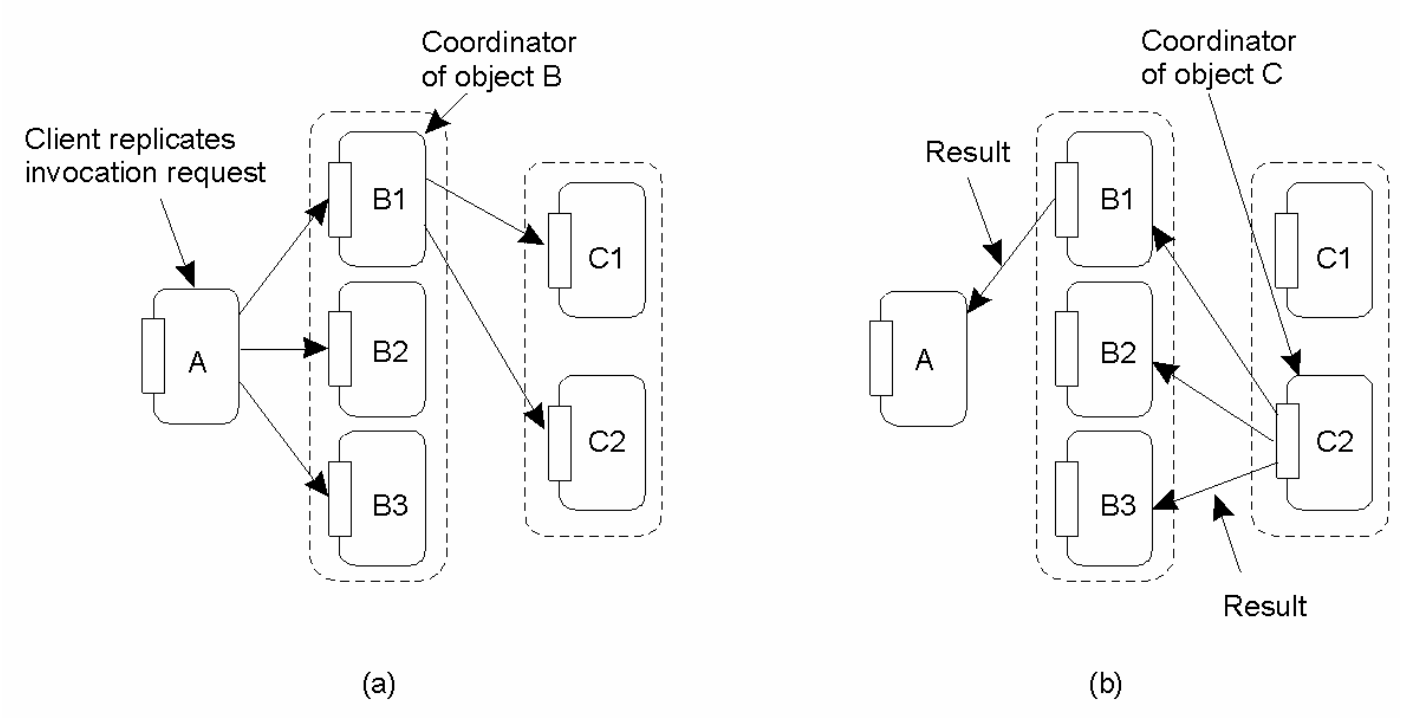
- Jede Replika besitzt einen Prozess, der die Updates durchführt.
- Updates werden meist als Operation propagiert
- Wichtigstes Problem: alle Updates müssen auf allen Replikas in derselben Reihenfolge ausgeführt werden,
- D.h., es wird Multicast mit totaler Ordnung benötigt (s. Kapitel über Zeit.), implementiert mittels Lamport-Uhren
- Skaliert aber nicht gut
- Alternative: zentraler Prozess, der die Sequentialisierung übernimmt
- Kombination aus beiden Ansätzen hat sich als brauchbar erwiesen

Replizierte Objektufrufe

- 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 Ver-arbeitung der Antworten



Koordination der replizierten Objekte



a) Weiterleitung eines Aufrufs von einem replizierten Objekts an ein anderes

b) Rückgabe der Antwort

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 die Erlaubnis von N/2+1 aller N Server erhalten.
- Ist das der Fall, kann kein anderer Client eine entsprechende Operation ausführen, da er auf keinen Fall mehr als die Hälfte der Server „hinter sich“ hat.

Fehlertoleranz

- Charakteristische Eigenschaft verteilter Systeme: partielle Fehler/Ausfälle (anders als Ein-Maschinen-Systeme)
- Durch partielle Fehler können Komponenten beeinflusst werden, während andere problemlos weiter arbeiten können
- Daher Ziel in vert. Systemen: konstruiere System so, dass es partielle Ausfälle verkraften kann, ohne dass die Leistungsfähigkeit zu sehr leidet
- Insbesondere sollte es während der Reparatur weiter arbeiten
- Das System ist dann fehlertolerant (fault-tolerant)

Verlässliche Systeme

- Fehlertoleranz hängt eng mit verlässlichen Systemen zusammen.
- Verlässlichkeit umfasst mehrere Konzepte:
  - Verfügbarkeit: das System ist sofort benutzbar
  - Zuverlässigkeit: das System läuft fortwährend ohne Fehler
  - Sicherheit (Safety): „nothing bad happens“
  - Wartbarkeit: sagt aus, wie leicht und schnell ein ausgefallenes System repariert werden kann
- Die Fähigkeit, verlässliche Systeme zu bauen, hängt stark von der Fähigkeit ab, auf Ausfälle reagieren zu können.

Fehlermodelle (Wdhg.)

- Zweck:
  - Klassifikation von Fehlern
  - Angabe der Fehlertoleranz von Programmen mit Bezug auf die Fehlerklassen

Type of failure	**Description**
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Maskierung von Fehlern

- Beste Vorgehensweise: Verbergen der Fehler vor anderen Komponenten (Maskierung)
- Wird erreicht durch *Redundanz*
  - Informationsredundanz:
    - Verwendung zusätzlicher Bits, um den möglichen Ausfall anderer Bits abzufangen
    - Beispiele: Forward Error Correction, Audio-CD
  - Zeitliche Redundanz:
    - Wiederholung von Aktionen
    - Beispiel: Transaktionen
  - Physische Redundanz
    - Zusätzliche Hardware oder Prozesse

Massnahmen zur Steigerung der Fehlertoleranz

- Wie kann man Fehlertoleranz erreichen?
- Basismechanismus: Replikation, daher enger Zusammenhang zu Fehlertoleranz
- Wir betrachten drei Massnahmen:
  - Ausfallsicherheit von Prozessen
  - Zuverlässiger RPC
  - Zuverlässige Gruppenkommunikation

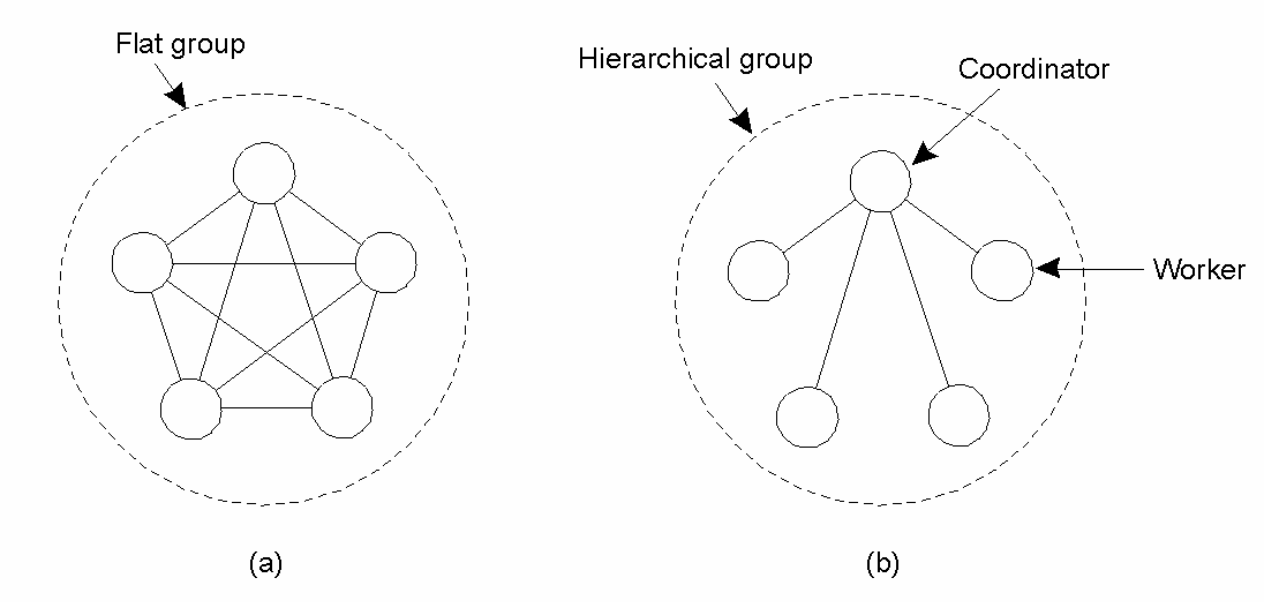
Prozess-Ausfallsicherheit

- Generelles Vorgehen:
  - Replikation von Prozessen (die dann die gleiche Aufgabe erfüllen)
  - Zusammenfassung in Prozessgruppen
  - Ergebnis: fehlertolerante Gruppe von Prozessen
  - Wird eine Nachricht an die Gruppe geschickt, bekommt jeder Prozess die Nachricht
- Wir betrachten
  - Organisation der Gruppen
  - Fehlermaskierung und Replikation
  - Erzielung von Übereinstimmung in fehlerhaften Systemen

Gruppenorganisation

- Dynamische Gruppen
  - Können erzeugt und gelöscht werden
  - Prozesse können ein- und austreten
  - Prozesse können Mitglied mehrerer Gruppen sein
- Zweck des Einsatzes von Gruppen: Abstraktion von den Einzelprozessen, Darstellung gegenüber der Außenwelt als eine Einheit
- Gruppenorganisation:
  - strukturlos oder
  - Hierarchisch (ein Koordinator)

Gruppenorganisation



- Kein single-point-of-failure
- Jede Entscheidung erfordert Abstimmung

- Schneller im Normalbetrieb
- Problem bei Ausfall des Koordinators

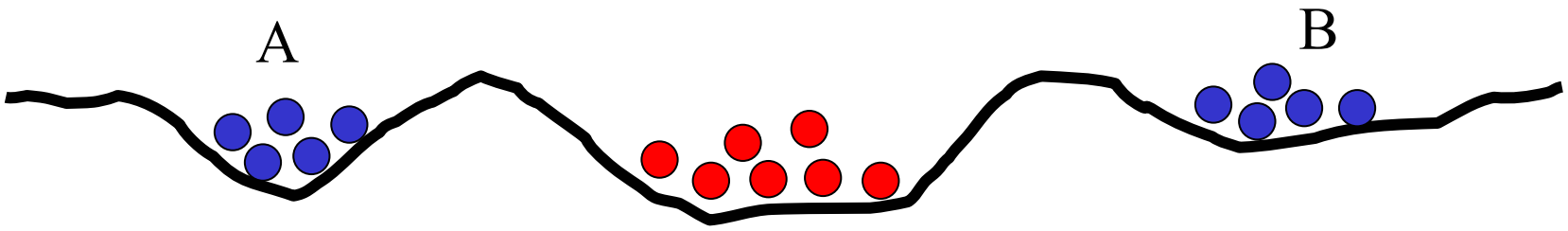
Maskierung von Fehlern

- Anwendung der Replikationsstrategien
  - Primary-Based:
    - Koordinator, der alle Write-Operationen koordiniert
    - Wenn der Koordinator abstürzt, wählen die übrigen Prozesse einen neuen
  - Active Replication:
    - Passt gut zu flachen Gruppen
- Wichtige Frage: wieviel Replikation ist nötig?
  - Hängt stark vom Fehlerverhalten ab
  - Wenn Prozesse byzantinische Fehler zeigen, dann benötigt man 2k+1 Prozesse, um k Fehler abfangen zu können
  - Kann man aber so genau sagen, dass maximal k Fehler auftreten? -> statistische Analysen

Übereinstimmung in fehlerhaften Systemen

- Schon kurz besprochen in Kapitel 8
- Problem: wie kommt man zu einer Übereinstimmung über eine durchzuführende Aktion im Falle
  - Perfekter Prozesse, aber fehlerhafter Kommunikationskanäle
  - Fehlerhafter Prozesse, aber perfekter Kommunikation
- Analogien:
  - Das 2-Armeen-Problem
  - Das Problem der byzantinischen Generäle

Das 2-Armeen-Problem

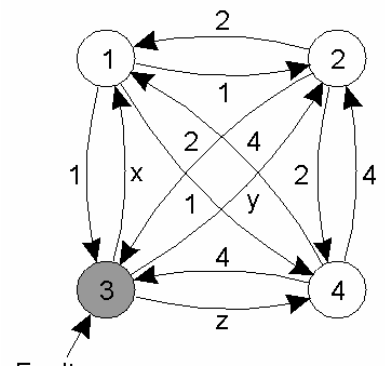


- Blau will rot angreifen, kann aber nur gemeinsam gewinnen (Überzahl)
- Notwendig: Abstimmung über Zeitpunkt des Angriffs
- Bote muss durch das rote Lager, kann gefangen werden -> unzuverlässige Übertragung
- Welches Problem hat B, wenn der Bote sicher bei A angekommen ist?
- Man kann zeigen, dass die beiden Prozesse nicht zu einer Übereinstimmung kommen können.

Die byzantinischen Generäle

- Szenario: rote Armee im Tal, n blaue Armeeteile in den Hügeln
- Kommunikation über zuverlässige Verbindung (Telefon)
- Problem: m der n Generäle sind Verräter, versuchen die Übereinstimmung der anderen zu verhindern
- Frage: können die loyalen Generäle trotzdem eine Übereinstimmung erzielen?
- Unser Freund Lamport hat auch hierzu eine Lösung ☺. Funktioniert, wenn mehr als 2/3 aller Generäle loyal sind.
- Beispiel: Übereinstimmung über die Truppenzahl soll erreicht werden

Lösung für die Generäle



(a)

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

- Szenario: n=4, m=1
- G1, G2 und G4 nennen die korrekte Stärke, G3 lügt zu allen
- 4 Schritte
- Schritt 1 (Bild (a)): alle Generäle melden ihre Stärke an alle anderen →in Schritt 2 Ergebnis in Vektor Bild (b)
- Schritt 3: jeder General meldet seinen Vektor aus (b) an alle anderen (G3 lügt wieder heftig) → Ergebnis in (c)
- Schritt 4: Jeder General überprüft die Spalten, wenn es eine Mehrheit gibt, ist der Wert korrekt
- Übereinstimmung: (1,2,unknown, 4)

Zuverlässiger RPC

- RPC/RMI abstrahiert von einer Kommunikationsbeziehung, aber dahinter steht immer Kommunikation über ein Netz
- Wenn Kanal und Prozesse perfekt funktionieren, ist RPC zuverlässig.
- Mögliche Probleme:
  - Client findet den Server nicht
  - Die Anfrage geht verloren
  - Der Server stürzt nach Erhalt der Anfrage ab
  - Die Antwort geht verloren
  - Der Client stürzt nach Senden der Anfrage ab.
- Behandlung dieser Probleme?

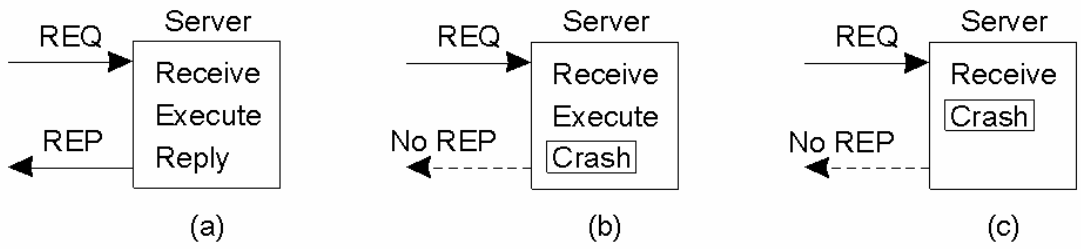
Client findet den Server nicht

- Ursachen
  - Server down
  - Unterschiedliche Versionen von Stub und Skeleton
- Lösung:
  - Exception auf der Client-Seite
  - Allerdings geht dadurch etwas Transparenz verloren
  - Nicht alle Sprachen unterstützen Exceptions

Anfrage geht verloren

- Leichtestes Problem
- Betriebssystem startet Timer beim Abschicken der Nachricht
- Nachricht wird erneut geschickt, wenn der Timer vor Eintreffen einer Antwort abläuft
  - Ist die Nachricht tatsächlich verloren gegangen, merkt der Server keinen Unterschied
  - Erhält der Server die Nachricht doppelt, dann muss er dies erkennen → siehe bei verlorenen Antwortnachrichten

Server-Crash



- Der Server kann abstürzen, bevor (c) oder nachdem (b) er die Operation ausgeführt hat.
- Beide Fälle müssen unterschiedlich behandelt werden
  - Bei (b) wird ein Ausnahmefehler erzeugt (beim Client)
  - Bei (c) kann die Nachricht einfach erneut (an einen anderen Server) geschickt werden
- Problem: wie erkennt das Client-BS die Situation? Kein erkennbarer Unterschied.

Server-Crash

- Lösung: verschiedene Ansichten
  - „At least once“ Semantik:
    - führe die Operation solange aus, bis sie erfolgreich war, d.h., bis eine Antwort eingetroffen ist.
    - Die Operation wird mindestens einmal ausgeführt
  - „At most once“ Semantik:
    - Sofortige Aufgabe, Fehlermeldung
    - RPC wurde höchstens einmal ausgeführt, evtl. gar nicht
  - Keine Garantie
    - Client wird allein gelassen
    - Leicht zu implementieren

Antwort geht verloren

- Problem: für den Client nicht zu unterscheiden von einem der vorherigen Fehler
- Kann eine Operation einfach wiederholt werden, wenn sie schon ausgeführt wurde?
- Idempotente Prozeduren: ein Aufruf ändert nichts am Zustand, können beliebig oft aufgerufen werden
- Versuche, möglichst nur idempotente Prozeduren zu verwenden (geht nicht immer)
- Mögliche Lösungen:
  - zustandsbehaftete Server, die sich Sequenznummern von Client-Requests merken
  - Identifikation von Wiederholungsanfragen durch ein Bit im Header

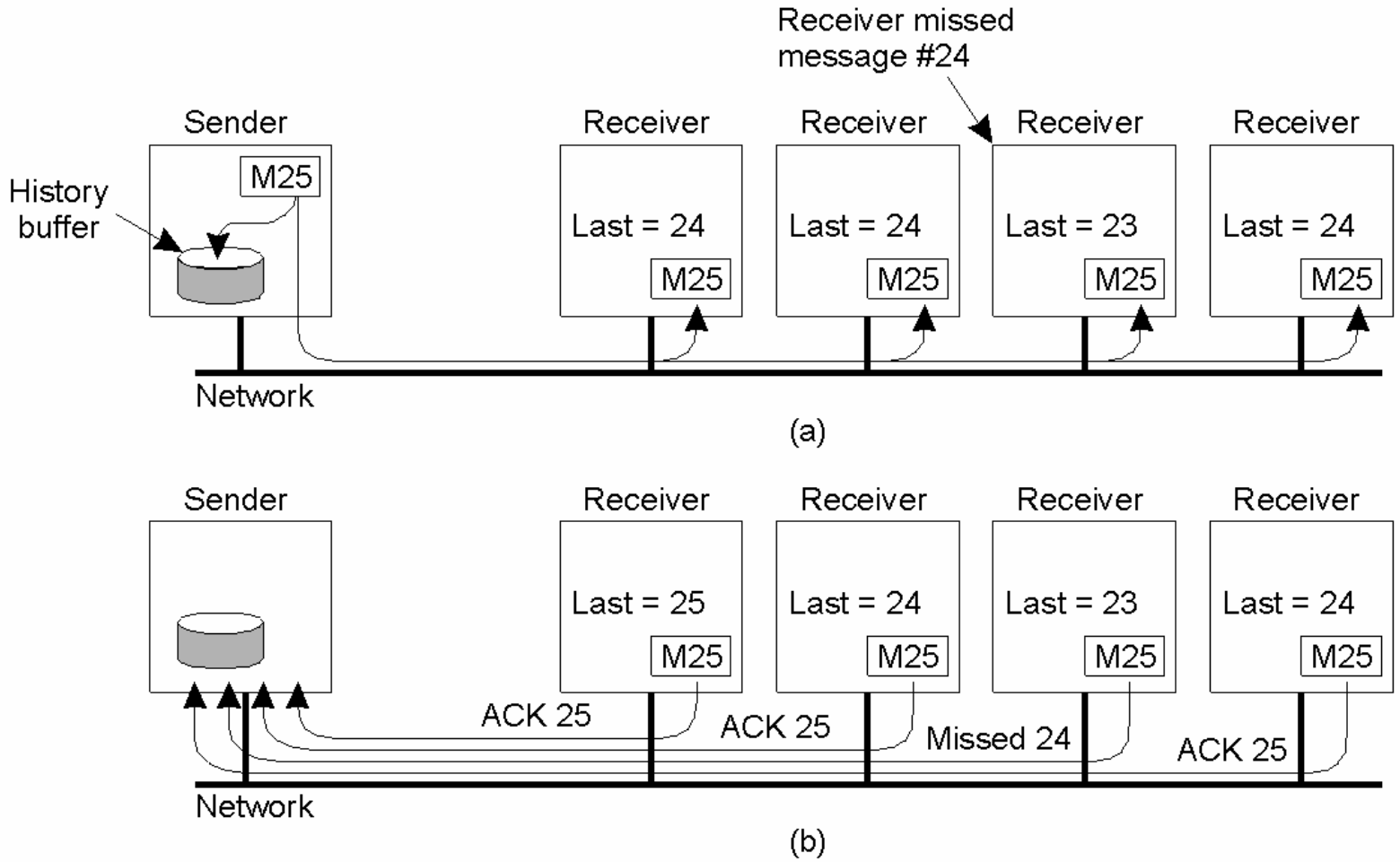
Client-Crash

- Client stürzt ab, bevor Server Antwort sendet → die entstehenden „elternlosen“ Antworten können zu Problemen führen
  - Ressourcenverschwendung
  - Konfusion bei Neustart des Clients und anschließendem Empfang der Antwort
- Mögliche Lösungen
  - Extermination: merke Dir alle angestossenen Operationen in permanentem Speicher, lösche dann nach Neustart noch eintreffende Antworten
  - Reinkarnation: Verwendung von „Epochen“, Lebenszeit des Client. Wenn Client neu startet, beginnt neue Epoche, alte Requests werden gelöscht

Zuverlässige Gruppenkommunikation

- Prozessgruppen sind ein wichtiges Mittel zur Erzielung von Fehlertoleranz.
- Deswegen spielt auch die zuverlässige Gruppenkommunikation eine große Rolle
- Unglücklicherweise ist diese viel schwerer zu erreichen als die einfache Gruppenkommunikation.
  - Insbesondere Skalierbarkeit!
  - Gruppendynamik

Grundlegendes Schema



Skalierbarkeit

- Das Schema funktioniert nicht gut, wenn es viele Empfänger gibt.
- Lösungen:
  - NACK-basiertes Schema: nur nicht empfangene Nachrichten werden gemeldet
  - Bei vielen verlorenen Nachrichten kehrt sich der Vorteil um
  - Weiterer Nachteil: Sender muss im Zweifel Nachrichten für immer im Puffer behalten
  - Weitere Idee: Hierarchische Feedback-Kontrolle

Hierarchische Feedback-Kontrolle

- Mehrstufiges Verfahren, mehrere Zwischenknoten halten Nachrichten und übernehmen die Retransmissions.
- Wichtigste Aufgabe: Konstruktion des Baumes