Verteilte Systeme Zusammenfassung

Benedikt Lüken-Winkels

10. April 2019

Inhaltsverzeichnis

1 Grundlagen

Verteiltes System

- Zusammenschluss unabhängiger Computer
- kein gemeinsamer Speicher
- Kommunikation über Nachrichten

1.1 Verschiedene Systemarten

- Client-Server-System: Viele Clients greifen auf einen oder mehrere Server zu.
- Verteilte Anwendung: Durch die Programmierung der Anwendung wird das verteilte System erstellt.
- Verteiltes Betriebssystem: Das Betriebssystem selbst ist verteilt, für Benutzer und Anwendungen ist dies nicht sichtbar.

1.2 Vorteile

- Speedup
  - $T_1(n)$ = Rechenzeit auf einem Kern, $T_k(n)$ Rechenzeit auf k Kernen
  - $\frac{T_1(n)}{T_k(n)} \leq k$
  - zu viel Lastverteilung, kann zu $k < 1$ führen und damit verlangsamen
- Redundanz $\Rightarrow$ Ausfallsicherheit. Naive Redundanz zu teuer: Primary Backup Approach. Einfacher 2. Rechner
- Räumliche Verteilung der Partner

1.3 Probleme

- Wahrscheinlichkeit von Fehlverhalten wächst mit Anzahl der beteiligten Prozesse
- Erkennen von Teilausfällen $\Rightarrow$ zB regelmäßiges Anpingen
- Synchronisation der Prozesse $\Rightarrow$ Wechselseitiger Ausschluss beim Dateizugriff (Mutex)
- Deadlocks (Zyklische Wartesituation mehrerer Prozesse)
- Debugging bei verteilter Berechnung kompliziert
- Nachrichtenbasierte Kommunikation fehleranfällig (Latenz bei der Übermittlung)

1.4 Aktormodell

- Aktoren (aktiv/passiv): Einfache, beschreibbare Programme (single Threads) die sequentiell und deterministisch arbeiten
  - aktiv: Verarbeitet, verändert Zustände, verschickt Nachrichten etc. Kann sich selbst in Passivmodus versetzen
  - passiv: kein Ressourcenverbrauch. Nur wieder aktivierbar durch Nachricht von Außen
- Kommunikation über Nachrichten
  - Jede Nachricht, die Versendet wird kommt an (Latenz kann aber beliebig sein).
- Terminierung eines Programms
  - Problem: Viele Aktoren im Verteilten System
  - Terminiert, wenn alle Aktoren passiv sind **und** keine Nachricht unterwegs ist, die andere Aktoren aufwecken (schwierig zu überprüfen)
  - Trick: Jeder Aktor hat Ein und Ausgangzähler (funktioniert nur wenn es eine globale Zeit gäbe)

$\Rightarrow$ Einschränkungen (kein neuer Thread, kann sich nicht selbst aufwecken) sorgt für Einfachheit der Lösung.

$\Rightarrow$ Multi-Aktorlösung nicht so effizient, wie Mutil-Threadedlösung

- Beispiele Go, Clojure

Single-System-Image Mehrere Rechner agieren als ein einzelnes System. Verwendung verteilter Software (Beispiel: Data-Center mit schnellem Verbindungsnetz im räumlich begrenzten Umfang)

2 Internet of Trust

CAP-Theorem Ein Verteiltes System kann bestimmten Grad erreichen. $\Rightarrow$ nur 2 der 3 Möglichkeiten kombinierbar

- Consistency: Replikate von Datensätzen müssen gleich sein
- Availability: Zuverlässige Antwort
- Partitioned Tolerance: Auslagerung, Ausfalltoleranz eines Knotens

Beispiele:

- CA: Datenbanksystem
- AP: zB verschiedener Cache
- CP: Bankautomat

3 Synchronisation

Problem Es gibt keine global genau gleiche Zeit, weil sich Information nur mit endlicher Geschwindigkeit bewegt.

Idee Entweder ein Rechner hat die exakte Uhr oder jeder Rechner hat eine ungefähr exakte Uhr.

Grundannahme Uhren haben eine lineare Abweichung von der Idealzeit.
Abweichung $\exists p > 0 : 1 - p < \frac{dC}{dt} < 1 + p$

3.1 Pragmatische Uhrensynchronisation

Idee Uhren werden 'gezwungen' sich zu synchronisieren

- F. Cristian: passiver Zeitserver, der auf Anfrage Timestamps liefert
- Berkeley: aktiver Zeitserver synchronisiert die Clients
- NTP (Network Time Protocol): 3.1.1
- DCF77: Zentraler Zeitserver in Braunschweig, Sender in Frankfurt. Senden per Funk (1500 km Reichweite)

3.1.1 NTP

- Verwendet UDP
- Baumstruktur aus Stratum Servern mit Atomuhren: Jeder Stratum Server synchronisiert sich mit kleinerem S-Server (Stratum$_2$ mit Stratum$_1$, Stratum$_3$ mit Stratum$_2$...). Kann mit GPS-Synchro auf eigenem Gerät simuliert werden.
- Vorteile eines eigenen Stratum$_1$-Servers: Sehr geringe Zeitbasis für sehr genaue Netzwerkmessungen
- ntpq = lokaler Dienst auf Unix-Geräten

3.2 Theoretische, logische Uhrensynchronisation

Idee Aus kausaler Abhängigkeit folgt zeitliche. Vergleichbare Zeitstempel $\Rightarrow$ Ereignis$_1$ ist vor Ergeignis$_2$.

- Alles was einen Einfluss auf die Ausführung des Programms hat ist ein **Ereignis**
  - Lokale Ereignis
  - Kommunikationsereignisse
- Zwischen 2 Ereignissen gibt es immer ein weiteres Ereignis
- Kausalität
  - Ursache findet vor der Wirkung statt
  - Kausalitätskette: aufeinander folgende Ereignisse sind voneinander abhängig
  - Kausalunabhängig: Ereignisse bedingen sich nicht gegenseitig

$\Rightarrow$ Ereignisse erhalten Zeitstempel wo die Kausalität aus dem Vergleich gezogen wird.

$\Rightarrow$ **Uhrenbedingung**

- $e_n <_k e_m \Rightarrow LC(e_n) < LC(e_m)$ (LC ist Zeitstempel)

- Lamport-Zeit 3.2.1
- Vektor-Zeit 3.2.2

3.2.1 Lamport-Zeit

Idee Einfachste logische Uhr. Jeder Rechner erhält n Bit-Zähler. Passiert ein Ereignis, wir die Uhr erhöht.

- Lokales Ereignis: Erhöhe Clock um 1

- Sendeereignis: Erhöhe Clock um 1, sende eigenen LC
- Empfangsereignis: Maximum aus eigenem und empfangenen LC ist neuer LC, $LC + 1 \Rightarrow$ garantiert größerer Zeitstempel für Empfangsereignis
- Nicht injektiv: sind 2 Zeitstempel gleich, folgt daraus nicht, dass die Ereignisse gleich sind

Uhrenbedingung? Klar für lokale Ereignisse; Empfangsereignis immer größer, als Sendeereignis
Umkehrung gilt nicht

Erweiterte Lamport-Zeit Zusatzkriterium um Vergleichbarkeit total zu machen

- Jedes Gerät hat eine eindeutige Adresse (zB IP-Adresse)
- Zeitstempel werden mit Adresse versehen
- Totale Ordnung
  - Sind 2 Zeitstempel gleich, wird auf die Ordnung auf den Adressen zurückgegriffen (willkürlich)
- Lamport-Zeit funktioniert nur bei geeigneter Größe für den Zähler $\Rightarrow$ Muss bei jeder Nachricht übermittelt werden

3.2.2 Vektor-Zeit

Idee Zeitstempel ist ein Vektor. $|Vektor|$ = Menge an Rechnern im System.

- Jeder Rechner k hat eine Vektorclock VC_k
- Lokales Ereignis: $VC_k[k] + +$
- Sendeereignis: $VC_k[k] + +$, sende VC_k
- Empfangsereignis: Empfänger erhöht seinen Wert im VC_k, nimm komponentenweise das Maximum
- Kausal abhängig: ein Vektor ist komponentenweise $\leq$ als der vorherige
- Kausal unabhängig: keiner der Vektoren ist größer als der andere

4 Mutex - Verteilter Wechselseitiger Ausschluss

Idee Nur ein Gerät darf in die Critical Section. Anwendung: Primary schmiert ab $\Rightarrow$ Nur einer der Backups wird neuer Primary.

Theorie	Nachteile	Komplexität
Zentraler Ansatz/Server	SPOF	$O(1)$
Token-Ring Le Lann	Token kann hängen bleiben	$O(1 \dots \infty)$
Verteilte Queue Lamport		$O(3n)$
Maekawa	Deadlockgefahr	$O(\sqrt{n})$
Raymond		$O(log_k n)$

Symmetrie In einer symmetrischen Lösung führen alle beteiligten exakt die selben Operationen aus, um ein Problem zu löse: Lamport; Maekawa eher symmetrisch

Asymetrie Führt schnell zum SPOF: Zentraler Server; Raymond eher asymmetrisch (Blätter haben fast keine Funktion)

Fazit Jeder Algorithmus ist Optimal für seine Symmetrieklasse

4.1 Zentraler Ansatz

- Zentraler Server serialisiert die Anfragen
- Request des Clients an Server
- Erster Client in der Queue bekommt das Grant
- Release von Client an Server 'bin fertig'
- Pop Queue
- ↯ SPOF, 3 Nachrichten (minimum), Bottleneck bei Server möglich

4.2 Token-Ring - Le Lann

Le Lann's Algorithmus

- Rechner sind in einem physischen Ring angeordnet
- Nur wer das Token hat, kann senden
- Wer in den kritischen Abschnitt will, wartet auf das Token
- ↯Token kann hängen bleiben

4.3 Verteilte Warteschlange - Lamport

Lamport, Verhinderung des SPOF eines zentralen Servers

- Anfragen sind in Warteschlange serialisiert
- alle Rechner haben eine Kopie der Warteschlange

$\Rightarrow$ Einigkeit über Reihenfolge der Requests in der Warteschlage durch **Agreement-protokoll**, zB Blockchain

- aus Unicast für Anfrage wird Multicast an alle
- erweiterte Lamport-Zeitstempel als Ordnung auf den Queues
- $\lightning$einige Requests möglicherweise noch unterwegs

$\Rightarrow$ Zwischen zwei Nachrichten besteht eine FIFO Ordnung: Nachricht$_2$ kommt nie vor Nachricht$_1$ an.

- Alle die ein Request empfangen, senden ein Acknowledge zurück
- Ack ist kausale Folge aus Request (größerer ZS)
- Min über alle Acks $\Rightarrow$ Requester bekommt keinen kleineren Zeitstempel als Min(Acks) mehr
- Queue wird in 2 Hälften geteilt: Ab dem Minimum ist die Queue statisch, da sich nichts mehr davor einreihen kann. Der Rest kann sich weiterhin verändern.

Laufzeit

- Request an alle: $0(n)$ oder $0(1)$, je nach Multicastfähigkeit
- Acks von allen: $O(n)$, n-1 Unicasts
- Release als Muticast an alle $O(n)$

$\Rightarrow$ Verbesserung der Laufzeit durch Verzögerung unnötiger Acks

4.4 Maekawa

Idee

- $n = m^2$ Knoten werden im Gitter angeordnet
- Will ein Knoten in die Critical Section wird das Lamport-Verfahren mit den Knoten in derselben Spalte und Zeile gestellt, also $2m = 2\sqrt{n}$ Unicasts
- Es gibt immer 2 Schnittknoten, die beide Requests sehen $\Rightarrow$ einer von beiden ist kleiner und erhält Ack
- $\lightning$Deadlockgefahr: die Schnittknoten senden verschiedene (falsche) Grants $\Rightarrow$ Revoke nach vergleich der Lamport-Zeit

4.5 Raymond

- Essentiell ein Token-Ring
- Ausbalancierte Baumstruktur mit logaritmischer Tiefe und gerichteten Kanten
- Kanten zeigen auf Knoten mit dem Token
- Request wird mit ZS in Tokenrichtung geschickt
- 🗲Der Knoten mit dem Token darf nicht ausfallen
- Je höher der Grad der Kanten, desto mehr nähert sich die Baumstruktur dem zentralen Server. $k \leq n - 1$

5 RPC - Remote Procedure Call

Idee Unterprogramme kommunizieren mit Remotegerät, ohne dass der Aufrufer dies mitbekommt $\Rightarrow$ Verbergen von Send/Receive

5.1 State-Variable

- Lokale Variablen: lokal auf Heap allokiert
- State-Variablen: Zugriff auf Variable nach außen/für andere geöffnet
  - Anfrage nach Content an Gerät mit Variable. Zurücklieferung des Wertes
  - Gefahr in Polling-Schleife zu geraten

Bäckerei-Algorithmus Array mit $|Array| = |beteiligte\ Prozesse|$. Jeder Prozess hat seinen Eintrag im Array und nimmt das Max(Array) + 1 für seinen Wert

6 Verteilte Terminierung

Frage Ist das Programm fertig? $\Rightarrow$ Trivial für sequentiellen Fall

- Ergebnisorientierte Terminierung: Ist ein Prädikat true, ist das System terminiert $\Rightarrow$ Komplexität im Prädikat: UND sind gleichzeitig zu überprüfen. Erfüllbarkeit ist nicht immer gegeben
- Kommunikationsbasierte Terminierung: Aktormodell $\Rightarrow$ alle Aktoren passiv und keine Nachricht unterwegs. Nachrichten zählen und Observer für Aktoren
  - Basisnachrichten
  - Kontrollnachrichten: Nachrichten des Koordinators. Fragt Zustandsinformationen ab. $|Empfangen| = |Gesendet|$ bedeutet Terminierung

6.1 Kommunikationsbasierte Terminierung

Doppelzählverfahren Koordinator fragt erneut nach $|Empfangen|$ und $|Gesendet|$ nachdem von allen die Antwort kam $|Empfangen_1| = |Gesendet_1| = |Empfangen_2| = |Gesendet_2|$

6.2 Vektormethode

Idee Wie bei Vektorzeit $\Rightarrow$ Vektor mit n Einträgen für n Aktoren.

- Senden: addieren, Empfangen: subtrahieren
- Kein Koordinator
- Null-Vektor $\Leftrightarrow$ terminiert

Nachlaufender Kontrollvektor Warte bis Aktor passiv und wandere dann zu Aktor mit Wert $\neq 0$

7 Election

- Fainess wird nicht verlangt $\Rightarrow$ es darf mehrfach der selbe 'gewinnen'
- (Bei Mutex immer ein anderer)
- $\lightning$Brechen von Symmetrie erzeugt SPOF
- Verteilte Minimum/Maximumsuche

Anwendungsfall Backup übernimmt, falls Primary versagt (Check via Heartbeat oder Watchdog-Timer)

- Mehrere Backups $\Rightarrow$ einer muss gewählt werden

7.1 Einfacher Election-Algorithmus

Funktioniert für kleine Graphen

- Prozessidentifikator M wird bei allen Prozessen vermerkt
- Eigener Wert wird an alle Nachbarn geschickt
- Ist eigener Wert ¡ Nachricht $\Rightarrow$ M aktualisieren

$\Rightarrow$ Alle haben Maximum der anfragenden Prozesse

8 Schnappschüsse

Liefert konsistente Schnitte

- Foto vom Systemzustand $\Rightarrow$ Entscheidungen im Algorithmus
- Koordinator
- **Konsistenz eines Schnitts**: Ist ein Ereignis links des Schnitts $\Rightarrow$ dann ist das kausalabhängige Ereignis auch im Schnitt
- Gummibandtransformation, um Abfrage gleichzeitig zu habe

8.1 Einfärben

Idee Garantieren eines konsistenten Schnitts

- Koordinator fragt Zustand ab: alle Prozesse und Nachrichten rot
- Hat ein Prozess die Antwort zurückgesandt wird er schwarz
- Erhält ein roter Prozess eine schwarze Nachricht vor der Nachricht des Koordinators $\Rightarrow$ Koordinatornachricht ist auf dem Weg. Schicke Nachricht an Koordinator und ändere Farbe
- Erhält ein schwarzer Prozess eine rote Nachricht $\Rightarrow$ schicke weiter an Koordinator

9 Lastverteilung

Kreative Lastverteilung Programmierung: sehr komplex

Mechanische Lastverteilung Lastpakete werden automatisch an Rechner verteilt. Entscheidung, welches Gerät geeignet ist. Je aufwändiger die Suche, desto geringer ist der Speedup

- Proaktiv: Rechner melden Last regelmäßig
- Reaktiv: Last muss angefragt werden
- Systemspezifisch: Prozesse
- Anwendungsspezifisch: Verteilbare Berechnungsabschnitte
- Probleme:
  - Paketgröße muss stimmen
  - Trägheit des Systems: Veraltete Informationen können zu Überlastung führen

Lastmetrik Vergleichbarkeit der Lastverteilung

- Prozessorauslatung: zB Ready Queue
- Speicherauslastung
- Kommunikationslast

Verteilung der Lastwerte Push: Verteile Info über eigene Last; Pull: Frage nach Last der anderen

- Alle n-1 fragen
- Random
- n-Hop

Lösungsansatz

- Lokale Informationen über Lasten anderer merken
- Keine großen Pakete
- Last fällt Exponentiell ab
- Statisch: Vor der Ausführung wird eine Lastverteilung bestimmt
- Dynamisch: Reaktion auf Lastpakete im Enstehungsmoment (Mit/Ohne Migration)

- Mit Migration: Während der Ausführung kann die Last neu verteilt werden
- Wechseln des Ausführenden kann teuer sein

⇒ Copy-On-Reference: Wird eine Seite erzeugt, wird die Seitentabelle dupliziert. Die Kopie zeigt auf die Urseite. Duplikat wird erst bei Referenzierung erstellt.

Beispiele

- Nutzung inaktiver Workstations
- Condor
- SetiAtHome
- Boinc: Plattform für verteilte Berechnung

10 Fehlertoleranz

Redundante Ausführung des Codes $\Rightarrow$ Ausfallwahrscheinlichkeit durch Menge der Replikate steuerbar p^n

Fehlermodele Hierarchie: Kann man schwere Fehler lösen, kann man auch einfache Lösen. Maskieren eines Fehlers = Ausblenden/Bearbeitungs eines Fehlers. Hardware 1. - 3. Software 4. - 5.

1. Crash: Beobachtbar, Rechner antwortet nicht mehr, Rechner wird auf Eis gelegt (Fail-and-Stop/Fail-Silent), Watchdog überprüft Ausführung
2. Ommission: Auslassung, Nachricht geht verloren
3. Timing: Nachrichten kommen zu früh oder zu spät an
4. Arbitrary: Beliebiger Fehler, Nachricht hat falschen Inhalt aber nicht böswillig (Softwarefehler)
5. Byzantinisch: Nachrichten können absichtlich verfälscht werden (Authentifizierung)

Um 1-3 zu kompensieren/maskieren braucht man n+1 Geräte
Um 4 zu kompensieren/maskieren eine falsche Antwort muss durch 3 überprüft werden $\Rightarrow$ Mehrheitsentscheid über Korrektheit 2k+1 Geräte nötig. Für 5 3k+1 (Mehrheitsentscheid + Authentifizierung)

Redundanz

- passiv: **Primary-Backup-Approach** Nächste Instanz übernimmt bei Fehler
  - Backups warten, solange Server antwortet
  - Funktioniert nur bis Fehlerklasse 3, weil Fehlerhafter Code weiterhin ausgeführt wird
  - Hot Standby: Jedes Kommando wird bei den Backups aktualisiert
  - Warm Standby: periodische Aktualisierung
  - Cold Standby: Reaktion auf Ausfall

  $\Rightarrow$ Umsetzung SCSI-Bus
- aktiv: Mehrere Server, die das gleiche tun
  - sehr teuer und daher nur in Bereichen, wo in Milisekunden entschieden wird
  - Replikatgruppe
  - Alle Fehlerklassen maskierbar
  - Lösung durch State-Machine-Approach 10.1

Byzantinisches General-Problem Nur bei mehreren Angriffen kann die Burg erobert werden. Einzelne Angriffe abgewehrt werden

⇒ Absprache eines gemeinsamen Angriffs

- Keine 100%ige Sicherheit in der Nachrichtenübermittlung
- Fehler in der Übermittlung:
  - Byzantiner ersetzen Nachricht mit böswilligem Inhalt

10.1 State-Machine-Approach

- Beteiligte Server sind Zustandsmaschinen
- Server hat einen Zustand, der durch Kommandos geändert werden kann
- Deterministisch und Atomar
- **Idempotenz:** gleiche Eingabe erzeugt immer die gleiche Ausgabe, unabhängig der Vorgeschichte
- Beispiel: File Server fängt in Initialzustand an und arbeitet die Aufgaben squentiell ab. Kollidieren Clients, entsteht Inkonsistenz

⇒ Seiteneffekte schränken Determinismus ein

- **Ensemble**: Mehrere State-Machines entscheiden über Mehrheit, welche Antwort richtig sind.
- Nur wenn die Aufträge in der selben Reihenfolge ankommen, sind die Antworten idempotent
- Geordneter Muticast für Ensemble (Reihenfolge der Anfragen)
- Fehlerhafte Clients können Ensemble stören

Arbitrary Einfache Replizierung einer State-Machine reicht nicht. N-Version-Programmierung/Verschiede
Sprachen.

10.2 Multicast

Nachricht an eine Gruppe

- Multicastgruppe = Liste von Empfängern (naiv)
- Empfänger/Mitglieder einer Multigruppe sind vertelt auf Netze
- Eventuell Flutung des Netzwerks nötig um alle Empfänger zu erreichen

- Zuverlässigkeitsgrad
  - Keine Garantie
  - K-Zuverlässigkeit: min k Empfänger erhalten die Nachricht, nicht unbedingt die gleichen
  - Atomar: Alle oder kein Empfänger erhalten die Nachricht
- Ordnungsgrad
  - FIFO
  - Kausale Ordnung
  - Totale Ordnung

10.2.1 Amoeba

- Verteiltes OS
- Verteilte Computer sollen wie ein einzelner Computer agieren (Single-System-Image)
- Nutzer verwenden plattenlose Rechner
- Dateiserver/Verzeichnisdienste etc laufen auf speziellen Rechnern (RPV-basierte Kommunikation)
- Sequencer übernimmt Hauptaufgabe
- Beim Ausfall nächster Teilnehmer in der totalen Ordnung
- Multicast: nur innerhalb Multicast-Gruppe
  1. Anfrage an Amoeba-Kern
  2. Kern blockiert SEND
  3. Kern fragt Sequencer per RPC
  4. Sequencer nummeriert die Nachricht und sendet
  5. Kern erhält die Nachricht zurück und deblockiert SEND

10.2.2 Raft

- Verteilen einer State-machine auf verschiedene Rechner
- Alle Beteiligten sind in einem Zustand

$\Rightarrow$ Leader-State-Machine aktualisiert den Status der anderen Maschinen

- Leader übernimmt die Hauptaufgabe

- Election, welcher Rechner der Leader ist

1. Timer eines Teilnehmers läuft aus, vote me
2. Empfängt ein Teilnehmmer eine Nachricht bevor sein Timer abläuft, wähle ihn (FIFO)

10.2.3 Netzwerkflutung

Idee Nachricht enthält Info über Multicastgruppe und Nachricht

- Erhält ein Knoten eine MC-Anfrage, sende an alle Nachbarn außer den Sender
- $\Rightarrow$ Redundante Kanten; Spannbaum entsteht (alle Knoten sind ohne Zyklen enthalten)
- Verkleinerung durch logisches Netz über physischem Netz (Overlaynetze)

Echo-Algorithmus

- Blätter des Spannbaums antworten in Richtung des Senders
- Innere Knoten warten, bis sie von allen Knoten, denen Sie zuvor geschickt haben einee Antwort erhalten haben
- Fügen ihre eigenen Infos hinzu und schicken weiter

11 Dateisysteme

Wurde bisher nicht in Klausuren abgefragt (daher Mut zur Lücke)

12 Häufig abgefragte Begriffe

- Raft-Protokoll 10.2.2
- Amoeba 10.2.1
- Vor/Nachteile Verteilter Systeme 1.3 1.2
- Idempotenz 10.1
- Aktor-Modell 1.4
- State-Machine-Approach 10.1
- Speedup 1.2
- Vektorzeit $\rightarrow$ Uhrenbedingungen zeigen 3.2.2
- erweiterte Lamportzeit $\rightarrow$ Uhrenbedingungen zeigen 3.2.1
- Fehlerklassen (Crash, Omission, Timing...) 10
- Raymond/Maekewa/Verteilte Warteschlange/Zentraler Ansatz 4
- SSI (Single-System-Image) 1.4
- Echo-Verfahren 10.2.3
- CAP-Theorem 2
- Aktive/Passive Toleranz 10
- Lastverteilung - Copy-on-Reference 9