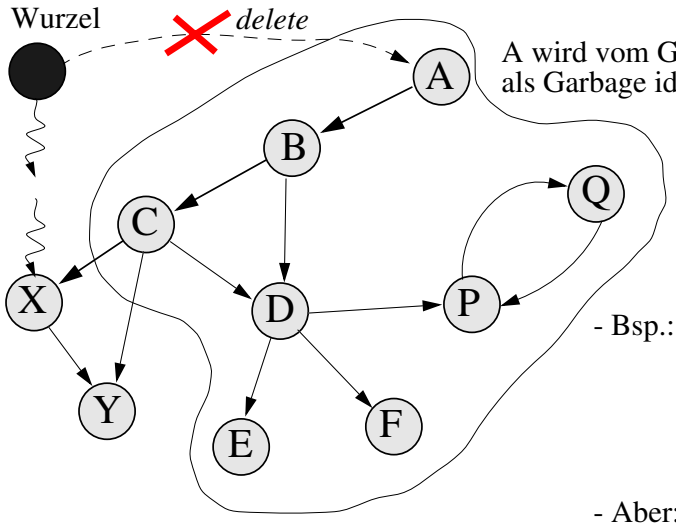
# Rekursives Freigeben

- Falls ein Objekt als Garbage erkannt wird:
  - sollten seine ausgehenden Referenzen gelöscht werden,
  - damit kann ggf. eine grössere daran "aufgehängte" Substruktur als Garbage erkannt werden!
    --> rekursives Erkennen von Garbage-Objekten, "pointer chasing"




A wird vom Garbage-Collector als Garbage identifiziert

- Bsp.: A Garbage
  --> B Garbage
  --> C Garbage
  --> D Garbage
  --> E, F Garbage
- Aber: X, Y kein Garbage!

- P, Q sind dann auch Garbage, manche Garbage-Collectoren können solchen "zyklischen Garbage" jedoch nicht erkennen!



# Freezing: "Stop the World"-Prinzip

*Mark & sweep-Algorithmus:*

Das geschieht meist automatisch, weil Weiterarbeit unmöglich ist

1) Mutator anhalten ("out of memory")
2) Alle erreichbaren Objekte markieren (ausgehend von der Wurzel) --> Graph-Traversierung
3) Garbage = alle unmarkierten Objekte
4) Mutator wieder starten

"sweep" durch den Hauptspeicher und z.B. in eine Freispeicherliste einketten

- "Stop the world"-Paradigma --> schlechte Lösung für Realzeit- und interaktive Anwendungen!
- Erst recht untragbar in verteilten Systemen!

---

- Vergleiche dies mit dem folgenden (schlechten!) Terminierungs-Entdeckungverfahren:
  - friere alle Prozese ein
  - ermittle Gesamtzahl der versendeten und empfangenen Nachrichten
  - falls identisch und alle Prozesse passiv sind: terminiert
  - ansonsten: "auftauen" aller Prozesse
- Eingefrorener globaler Zustand ist konsistent!
  - "in aller Ruhe" ein Objekt nach dem anderen betrachten

# "Behind the back copy"-Problem

*Concurrent / parallel / on-the-fly-G*arbage-Collection:

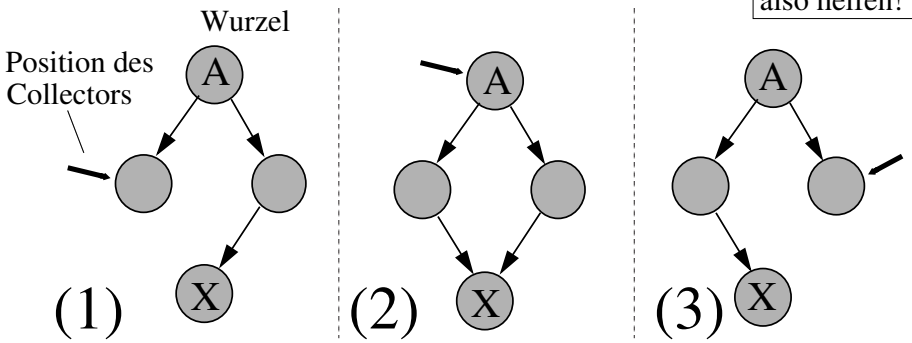
- Collector versucht, Garbage-Objekte zu identifizieren, *während* der Mutator aktiv ist
- Verhindert somit lange Wartezeiten der Anwendung

---

Traversiert den Graphen und markiert erreichbare Objekte

Collector kann von gleichzeitig aktivem Mutator getäuscht werden --> Kooperation notwendig!

Mutator muss dem Collector also helfen!




- Knoten X wird als nicht erreichbar angesehen...
- ...obwohl es immer einen Weg von A zu X gibt!

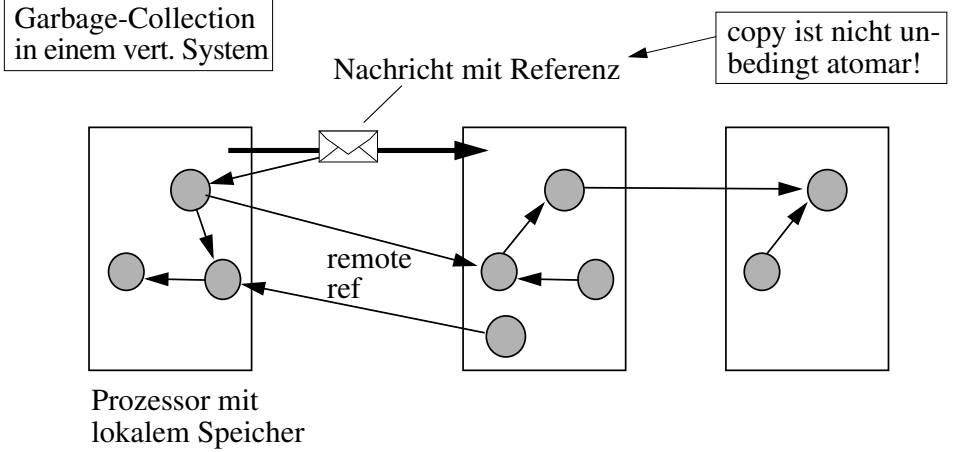
---

- Manipulationen "hinter dem Rücken" des Collectors
- Collector rekonstruiert aus seinen zusammengesetzten lokalen Beobachtungen einen falschen (nie existenten) Graphen
- dabei ist Beobachtungszeitpunkt = Besuchszeitpunkt



# Verteiltes Garbage-Collection

Garbage-Collection in einem vert. System



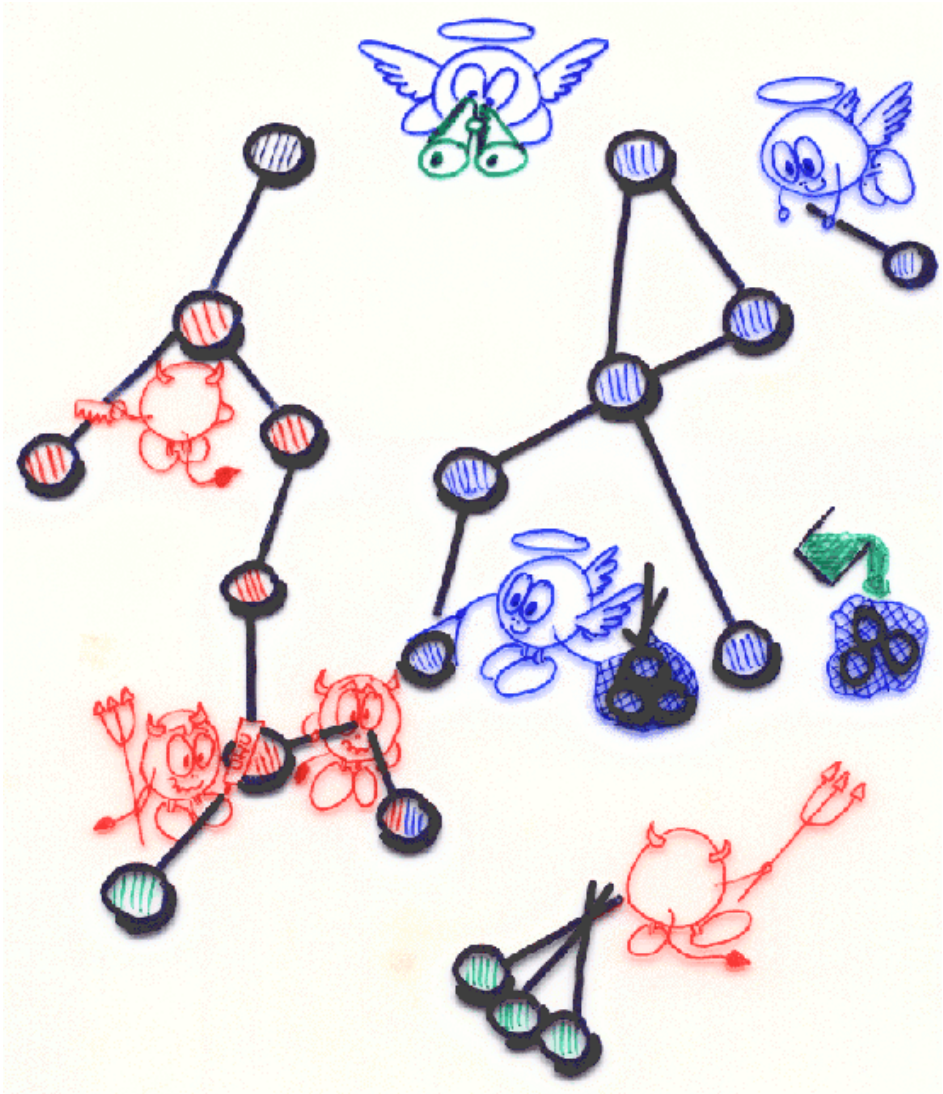

(1) Prozessorüberschreitende Referenzen ("*remote ref*")

(2) Nachrichten enthalten (Objekte mit) Referenzen auf andere Objekte ("*remote copy*")

- Referenzen in Nachrichten dürfen nicht übersehen werden!

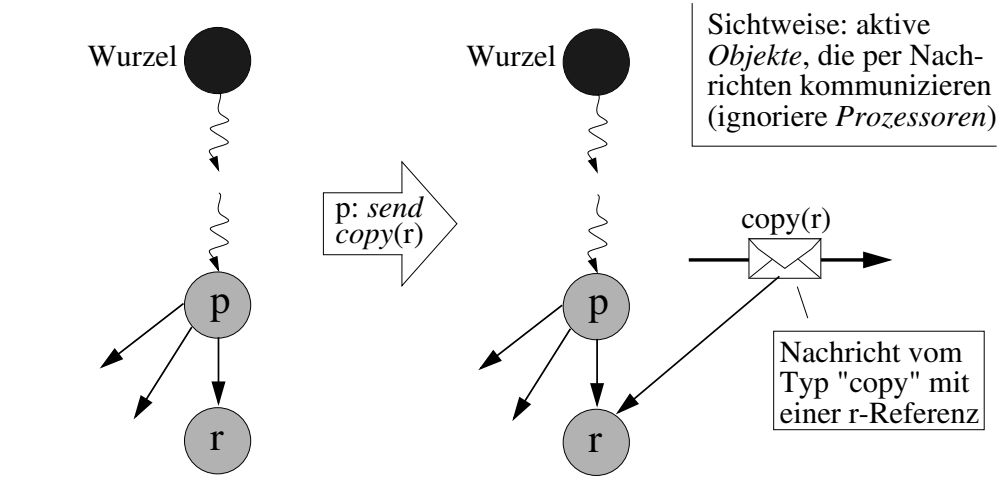
---

- GC typischerweise dezentral, parallel und hierarchisch
  - *lokale* GC (Annahme: alle eingehenden remote refs kommen von einem erreichbaren Objekt)
  - *globale* GC (Suchen prozessorübergreifender "Garbage-Ketten")
- GC aufwendiger als im nicht-verteilten Fall
- Viele Mutator / Collector (pro Prozessor einen?)
  - Synchronisation zwischen den vielen Prozessen notwendig
- Pragmatisches:
  - "Stop the world" ist hier schon gar nicht angemessen
  - Kontrollkommunikation minimieren (Kosten, Effizienz)



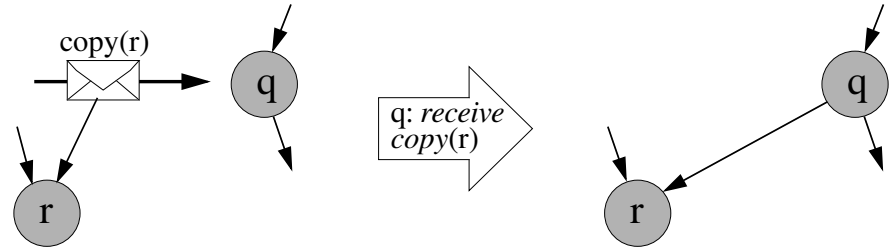


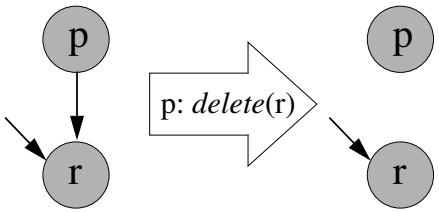
# Wirkung der Mutator-Operationen




Sichtweise: aktive *Objekte*, die per Nachrichten kommunizieren (ignoriere *Prozessoren*)

Beachte: Copy-Operation ist nicht atomar ==> Aufspalten in zwei Aktionen *send / receive copy*

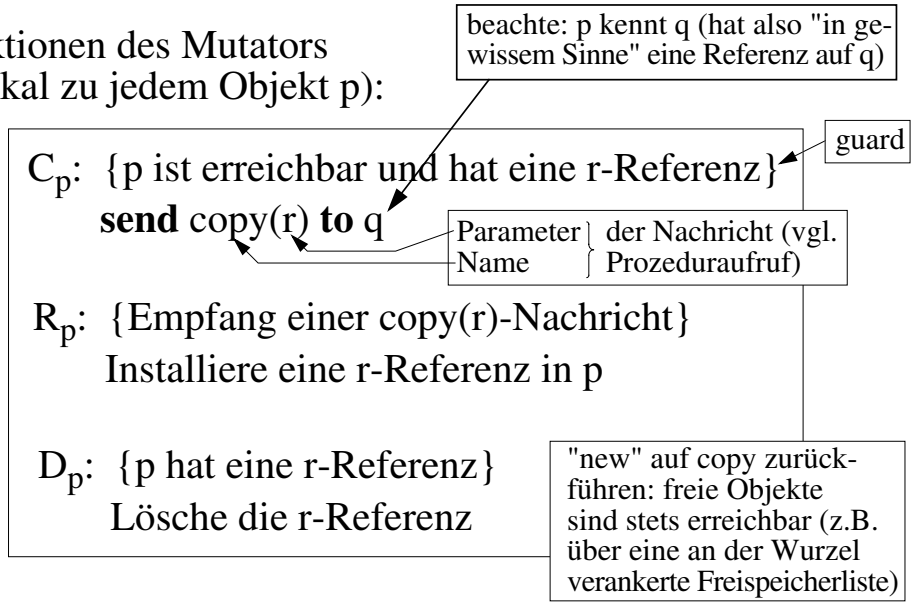



- "new" hier nicht relevant
- jede Aktion ändert den globalen Graphen "etwas"
- Folge solcher Änderungen -->" Berechnung"
- hier: "Interleaving-Modell": Operationen sind atomar ("zeitlos") --> es gibt keine gleichzeitige Aktionen --> verschränkte Ausführung

# Formalisierung des verteilten Garbage-Collection-Problems

Aktionen des Mutators
(lokal zu jedem Objekt p):

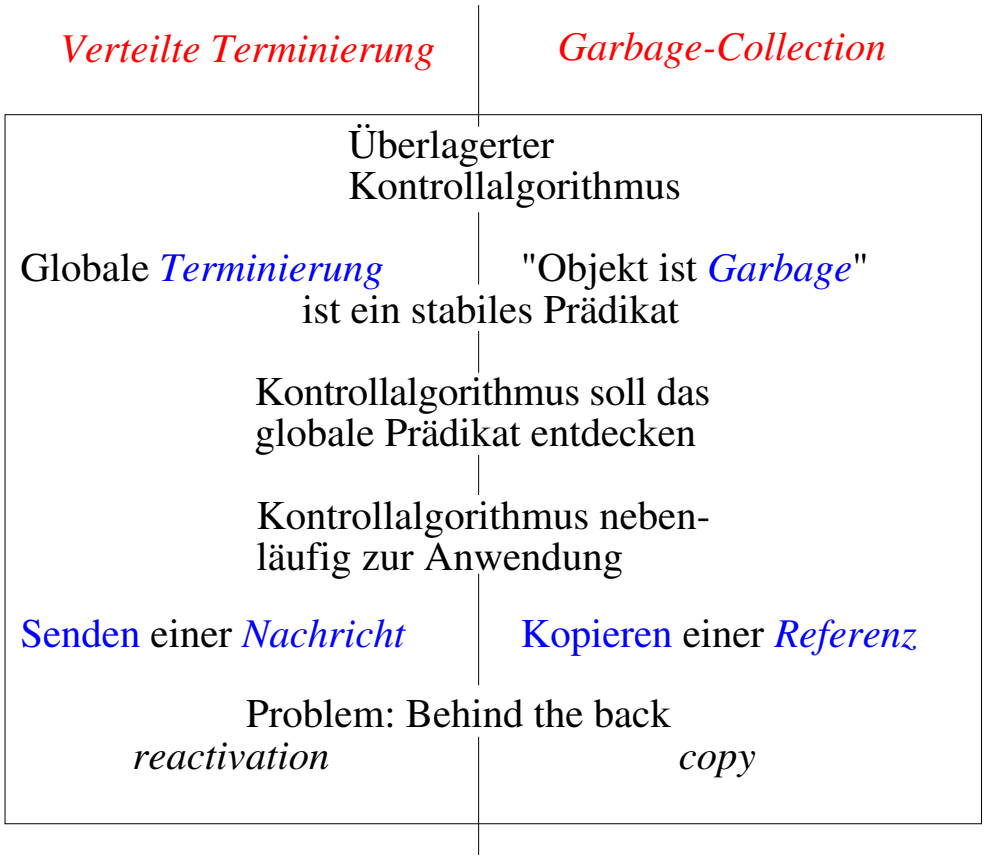



- So "sieht" der Collector die Basisberechnung
- Vergleiche dies mit Basisaktionen beim Problem der verteilten Terminierung!
- Aufgabe: Überlagerung mit Aktionen des Collectors:
  - zusätzliche atomare Aktionen
  - Ergänzung der 3 Aktionen mit weiteren Statements, um die notwendige Kooperation mit dem Collector zu erreichen
- Bedingungen an eine korrekte Lösung:
  - *Safety*: Wenn ein Objekt eingesammelt wird, dann ist es Garbage
  - *Liveness*: Wenn ein Objekt Garbage ist, dann wird es *schliesslich* eingesammelt



# Verteilte Terminierung und Garbage-Collection

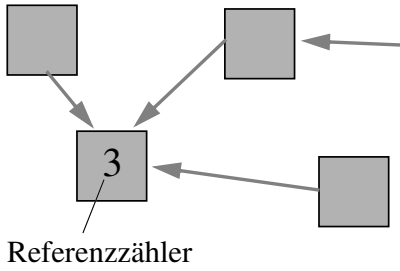
- Interessante Analogie zwischen beiden Problemen:




- Können Lösungen des einen Bereiches auf den anderen Problembereich angewendet werden?

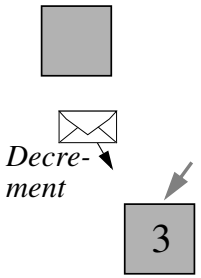
# Referenzzähler-Verfahren

- Idee: Jedes Objekt weiss, wie oft es referenziert wird
  - wird dieser Referenzzähler 0 --> Garbage
- Nachteil: Zyklischer Garbage wird nicht entdeckt
- Zugehörigen Referenzzähler "atomar" zusammen mit der copy- oder delete-Operation aktualisieren
  - relativ einfach in einem nicht-verteilten System
- In einem verteilten System:
  *increment* / *decrement* Nachrichten



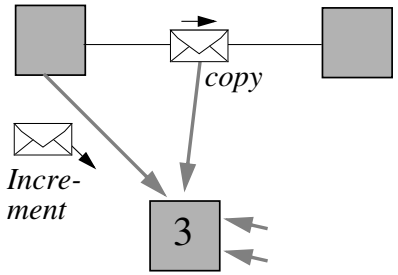

Referenzähler = 0
==> Objekt ist Garbage

*Löschen* einer Referenz:




3 ==> 2

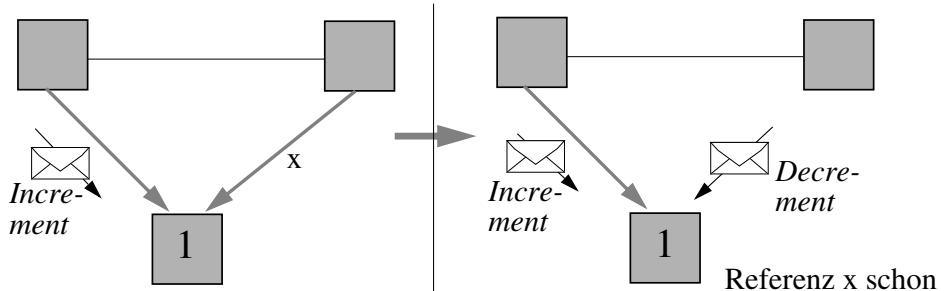
*Kopieren* einer Referenz:




3 ==> 4



Kopieren kann zu *Fehlinterpretationen* führen!




Increment-Nachricht bzgl.
Referenz x ist noch unterwegs

Decrement schneller als vorangehendes Increment
==> Referenzzähler wird 0, jedoch kein Garbage!

*Korrektheitsbedingung:*
Empfang von *Inc* früher als alle kausal abhängigen *Dec*

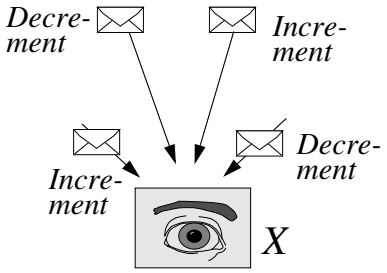
Wie dies garantieren?

1) Synchrone Kommunikation (--> "atomare" Operation)
2) Acknowledge von Inc-Nachrichten (+ warten)
3) "Causal Order" realisieren...

Objekt hat eine *kausaltreue Sicht*
der Ereignisse (die es betreffen)

--> (indirekte) Überholungen vermeiden

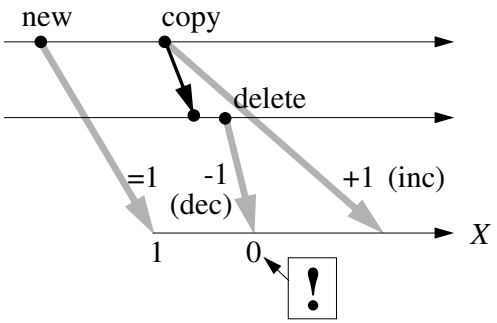
# Garbage-Identifikation als konsistentes Beobachtungsproblem




Objekt X "beobachtet" die copy- und delete-Operationen, die es selbst betreffen

- die increment- und decrement-Nachrichten dienen der Beobachtung
- damit feststellen, ob *alle* copy / new durch delete kompensiert wurden

- Diese Beobachtung sollte kausaltreu sein!
  - zumindest darf ein dec nicht vor "seinem" inc empfangen werden



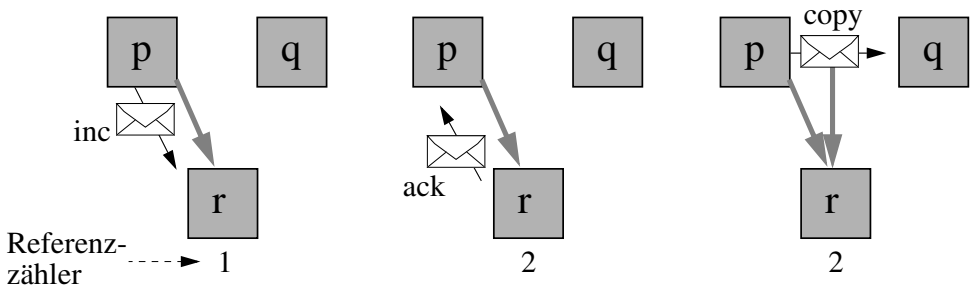

- *Jedes* Objekt beobachtet jedes andere
  - "Causal Order" (Globalisierung von FIFO) bzgl. Kommunikation gefordert

- Wie realisiert man kausaltreue Beobachtungen?
  - in diesem konkreten Fall diverse Möglichkeiten (--> viele Algorithmen!), z.B. copy solange verzögern, bis Bestätigung für inc-Nachricht eingetroffen
  - oder: kausaltreue Beobachter / "Causal Order"-Kommunikation allgemein implementieren?



# Verteiltes Reference-Counting: Lösungen

--> Jede increment-Nachricht bestätigen (warten auf ack):

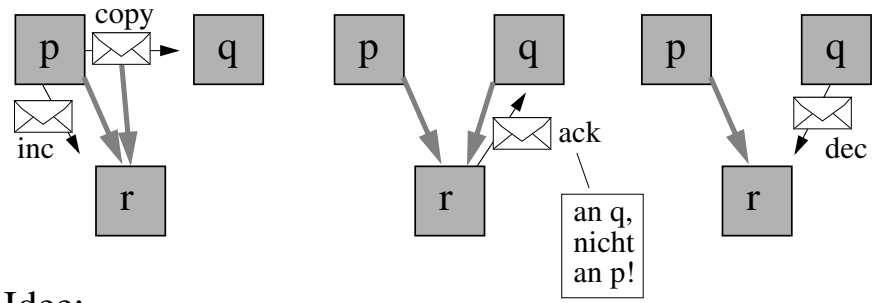



- Korrektheitskriterium erfüllt:
  inc wird vor einem kausal abhängigen dec empfangen
  (d.h. Objekt r erfährt über die Existenz einer Kopie einer r-Referenz, bevor es vom Löschen dieser oder einer "solchen" Referenz erfährt)

- Nachteile der Methode:
  - copy wird verzögert
  - zusätzliche Nachricht (ack)

--> insgesamt 3 Nachrichten pro copy-Operation

# Variante von Lermen und Maurer




Idee:

Senden einer dec-Nachricht (bei Löschen der Referenz) erst dann, wenn das Objekt bereits eine zugehöriges ack (bzgl. inc) vom Zielobjekt der Referenz empfangen hat

==> Korrektheitskriterium erfüllt

- Beachte: Referenz kann stets gelöscht werden, nur das Senden von dec muss verzögert werden

---

- Implementierungsskizze:
  - Zählen von empfangenen copy und ack-Nachrichten
  - dec-Nachricht erst senden, wenn Zähler ACK und COPY übereinstimmen
    - dann die Zähler ACK und COPY beide dekrementieren
    - "individuelle" Zuordnung von copy zu ack nicht notwendig!
    - Abschwächung |ACK|>0 ∧ |COPY|>0 möglich? Konsequenzen?

---

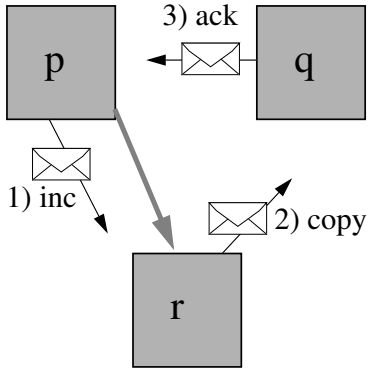
- *Vorteil*: kein Verzögern von Basisaktionen
- *Nachteil*: Ack-Nachricht und FIFO-Kanäle notwendig

wieso?



# Varianten von Rudalics

## 1) 3-Nachrichten-Protokoll ("zyklisch"):




*Idee*: q bekommt Referenz auf r von r selbst; nachdem r seinen Zähler inkrementiert hat (veranlasst durch inc).

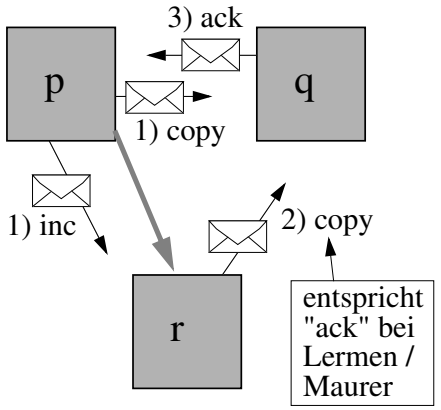
*Bedingung*: p darf seine r-Referenz erst löschen, wenn alle erwarteten ack-Nachrichten eingetroffen sind.

*Frage*: Wäre es auch möglich, dass das ack an p von r (statt q) gesendet wird?

- *Vorteil*: kein FIFO notwendig (falls FIFO garantiert ist: ack-Nachricht einsparen ==> nur zwei Nachrichten pro copy!)
- *Nachteil*: Kopieren dauert länger (2 Nachrichten)

---

## 2) 4-Nachrichten-Protokoll:



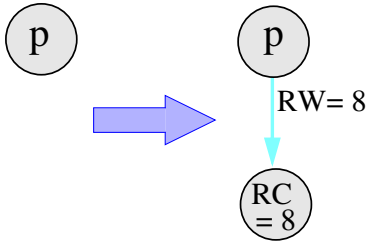

- Idee: ack erst senden, wenn *beide* copy-Nachrichten empfangen wurden
- q installiert die r-Referenz bei Empfang der *ersten* copy-Nachricht
- Unter welchen Bedingungen dürfen p bzw. q dec-Nachrichten senden?

- Vorteil: kein FIFO notwendig; keine Verzögerung
- Nachteil: 4 Nachrichten (aber nur 3 "sequentiell")

# Die Referenzgewichts-Methode

(WRC: "Weighted Reference Counting")

**Neues Objekt generieren ("new"):**



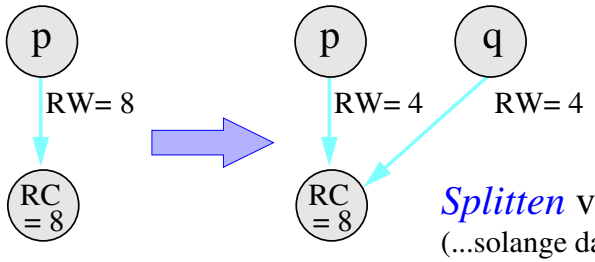

Reference *Weight*

Reference *Count*

(obwohl dadurch die Referenzen nicht tatsächlich "gezählt" werden)

---

**Referenz kopieren ("copy "):**




*Splitten* von RW

(...solange das geht!)

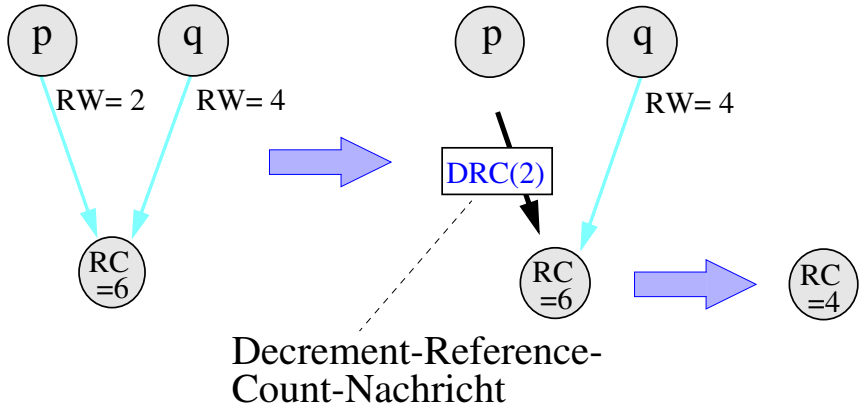
ansonsten (hier nicht genauer behandelt):

1) "Recharge"
2) Indirektionsobjekt

- Beachte: Es wird keine Increment-Nachricht benötigt!



**Referenz löschen ("delete"):**




---

*Invariante*: RC = ∑ RW + ∑ DRC

RC = 0 --> Objekt ist Garbage

--> alle Referenzen dieses Objektes auf andere Objekte löschen (DRC-Nachrichten senden)

---

- Logarithmische Kompression (2er-Potenzen!) von RW
  - mit nur 2 Bit pro Zeiger lassen sich so RW bis max. 8 darstellen
  - statt 8 kann ggf. auch ein (etwas?) grösserer Maximalwert gewählt werden
  - RC so nicht komprimierbar ==> int-Variable mit "vielen" Bits pro Objekt
- Keine Verzögerung bei copy / delete und bei copy keine zusätzlichen Nachrichten!
  - RW = 1 sollte ein eher seltenes Ereignis sein (--> Zusatzaufwand)
- Analogie zur *Kredit-Methode* bei vert. Terminierung!

# Kredit-Methode und WRC

| Terminierungserkennung mit der Kreditmethode | | WRC-Garbage-Collection |
|---|---|---|
| - Senden einer Nachricht (Splitten des Kreditwertes) | ⟷ | - Kopieren einer Referenz (Splitten des RW) |
| - Passiv werden (Kredit an Urprozess zurückgeben) | ⟷ | - R-Referenz löschen (Decrement-Nachricht an R) |
| *Invariante*: ∑ Kredit =1 | ⟷ | *Invariante*: RC = ∑ RW + ∑ DRC |
| - Gesamtkredit beim Urprozess = $2^0$=1 | ⟷ | - RC = 0 bei R |
| *Terminierung* | ⟷ | *R ist Garbage* |

Also: Kreditmethode entspricht WRC-Garbage-Collection!



# Garbage-Collection und Terminierung

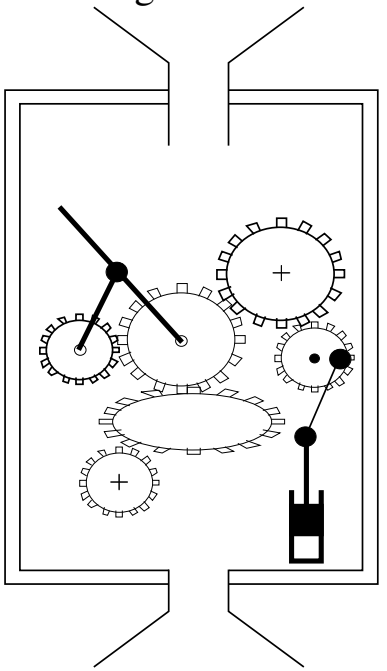
**Theorem:**

nicht notwendigerweise verteilte

Jeder *Garbage-Collection*-Algorithmus kann automatisch in einen Algorithmus zur Feststellung der *verteilten Terminierung* transformiert werden

Bemerkung: Für beide Probleme wurden viele nicht-triviale (und auch manche falsche!) Lösungen publiziert

Ein Garbage-Collection-Algorithmus

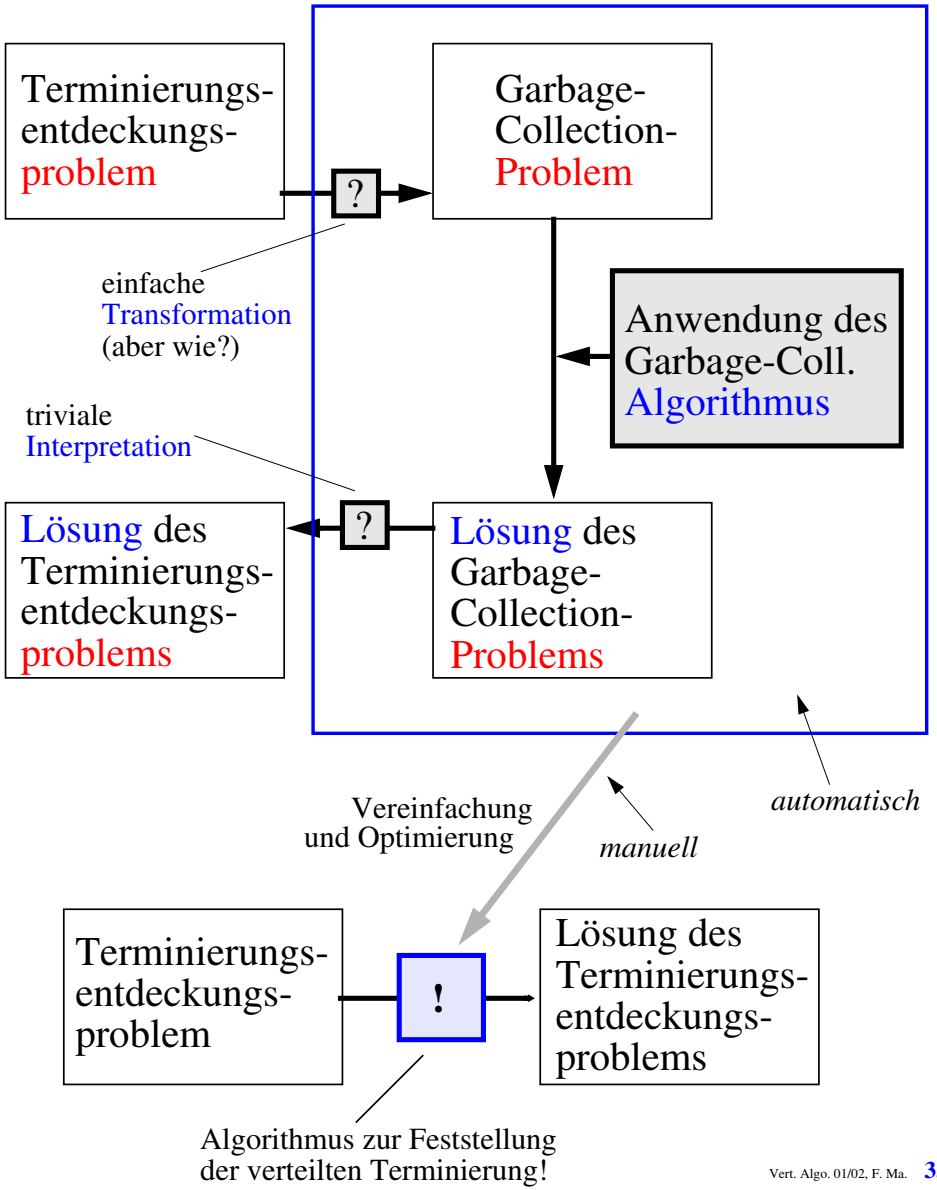


mechanische Transformation

Ein verteilter Terminierungsentdeckungs-Algorithmus

# Problemtransformation

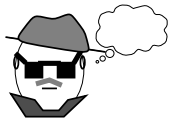
- Es wird das *Problem*, nicht der *Algortihmus* transformiert!



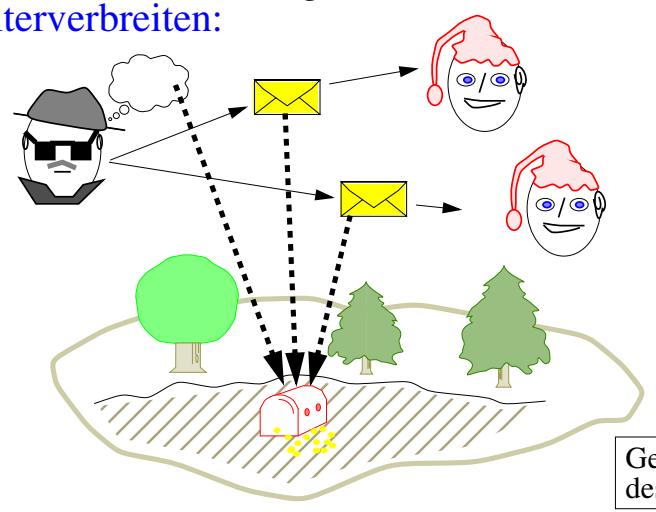



# Vom Ende einer Geheimniskrämerei

*Die vier goldenen Regeln der Geheimniskrämerei:*

1) Es gibt Geheim-nisträger  und unein-geweihte Personen

2) Nur ein Geheimnisträger kann das Geheimnis weiterverbreiten:




3) Wer das Geheimnis erfährt, wird zum Geheimnisträger

4) Ein Geheimnisträger kann das Geheimnis (endgültig) vergessen

---

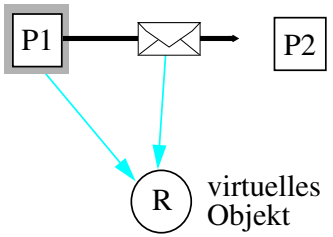
es gibt keine Geheimnisträger und keine Nachrichten mit dem Geheimnis

*Geheimniskrämerei terminiert* <==>
*Schatz nicht mehr zugreifbar* <==> *Schatz ist "Garbage"*

--> "watchdog" beim Schatz meldet Terminierung...

# Die Transformation

- Jeder Prozess wird in ein Wurzelobjekt transformiert
- Ein zusätzliches virtuelles Objekt R wird hinzugefügt




1) Prozess P aktiv <==> P besitzt Referenz auf R

2) Jede Nachricht enthält Referenz auf R

- Die beiden Regeln lassen sich ("induktiv") erfüllen:
  - ein (aktives) Objekt / Prozess sendet eine Kopie seiner R-Referenz mit jeder Nachricht
  - ein reaktivierter Prozess erhält eine R-Referenz
  - ein Prozess, der passiv wird, löscht seine R-Referenz

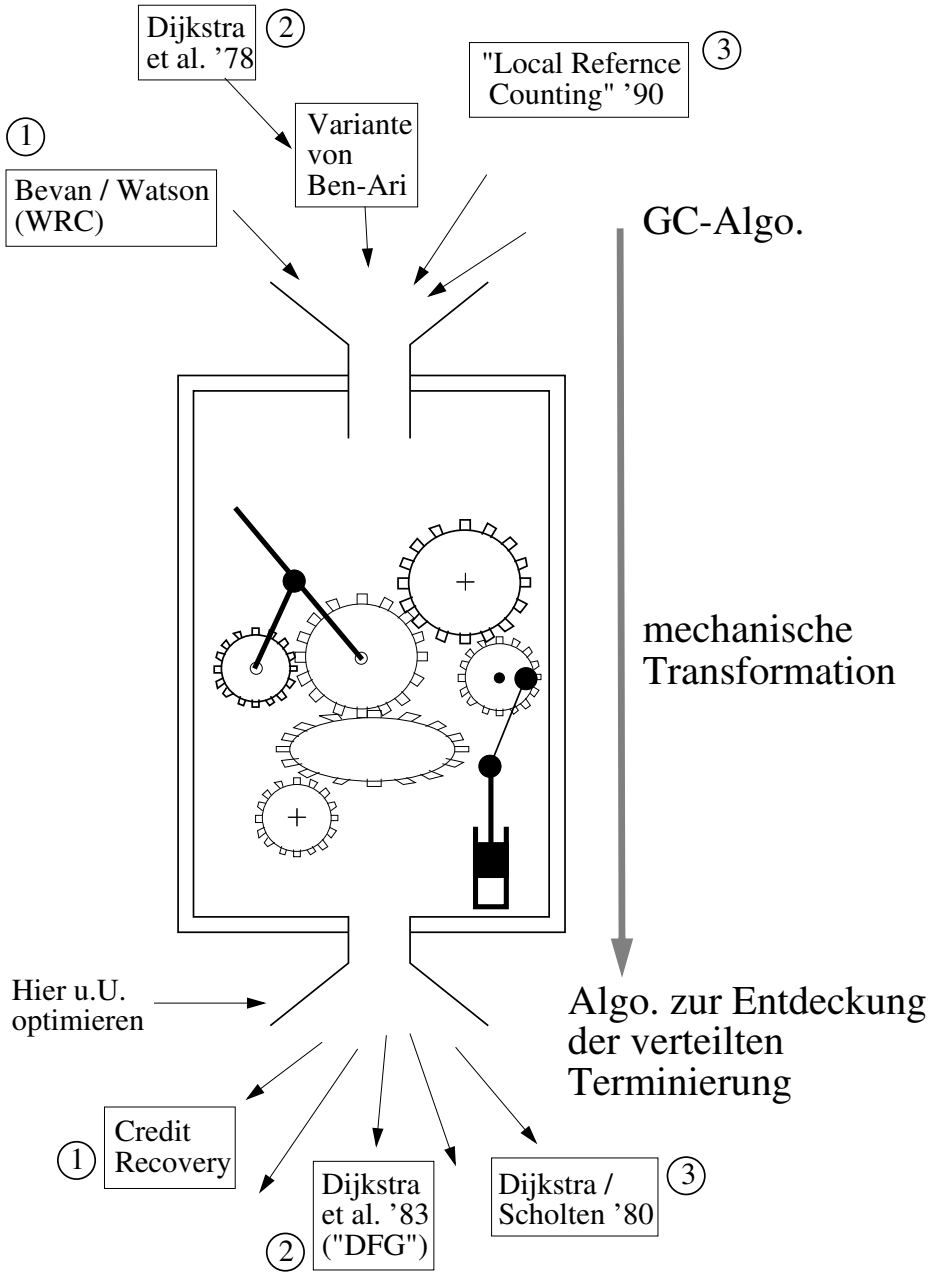
R Garbage <==> Es gibt keine Referenz auf R <==> Alle Prozesse passiv und keine Nachricht unterwegs <==> Verteilte Berechnung terminiert

- Also: verwende *irgendeinen* GC-Algorithmus
  --> interpretiere Berechnung als GC-Problem
  --> melde Terminierung, wenn R als Garbage erkannt

---

- Übung: man mache dies für konkrete GC-Algorithmen aus der Literatur
- beachte: es entstehen keine Zyklen von Referenzen, daher sind auch Referenzzählverfahren anwendbar!

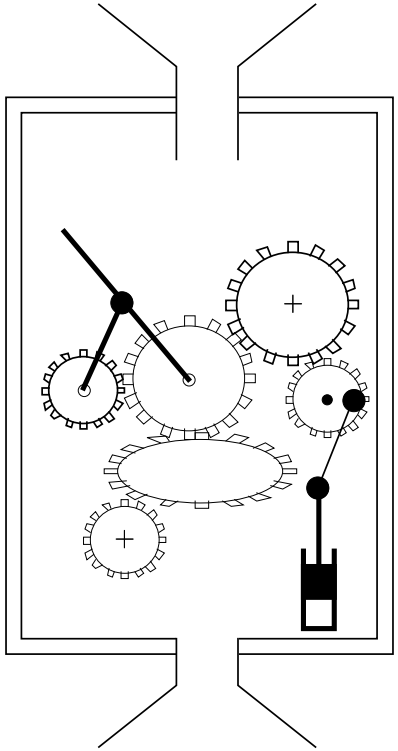





Bekannte Garbage-Collection-Verfahren werden so in bekannte und brauchbare Algorithmen zur Erkennung der verteilten Terminierung transformiert!

# Die Patent-Story

WRC-Garbage-Collection-Algorithm patentiert:
Europäische Patentnummer 86309082.5



Ist der resultierende Terminierungs-erkennungsalgorithmus auch durch das Patent geschützt?



# Das WRC-Patent

Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 225 755 A2**

(12) **EUROPEAN PATENT APPLICATION**

(21) Application number: 86309082.5

(22) Date of filing: 20.11.86

20.11.86

(51) Int. Cl.: **G 06 F 12/02**

ICL

(30) Priority: 04.12.85 GB 8529890

(71) Applicant: INTERNATIONAL COMPUTERS LIMITED ICL House, Putney, London, SW15 1SW (GB)

Watson, Paul

(43) Date of publication of application: 16.06.87 Bulletin 87/25

(72) Inventor: Watson, Paul, 146, Hilda Park Chester-Le-Street, Co-Durham DH2 2JY (GB)

(84) Designated Contracting States: BE DE FR GB NL

(74) Representative: Guyatt, Derek Charles et al, STC Patents Edinburgh Way, Harlow Essex CM20 2SH (GB)

(54) Garbage collection in a computer system

(57) A computer system is described, having memory cells organised in a directed graph structure by means of pointers. Each cell has a reference count, and each pointer a weight value. If a new pointer to a cell is created by copying an existing pointer, the new and existing pointers are given weights whose sum equals the old value of the existing pointer. In this way, the sum of the weights of the pointers to any cell are maintained equal to its reference count. If a pointer is destroyed, the reference count of the cell to which it points is reduced by the weight of the pointer. Thus, when the reference count of a cell reaches zero, it is safe to assume that there are no more existing pointers to it, and hence that cell may be reclaimed (garbage-collected) for re-use.

nächste Folie-->

(84) Designated Contracting States: BE DE FR GB NL

(54) Garbage collection in a computer system

(57) A computer system is described, having memory cells organised in a directed graph structure by means of pointers. Each cell has a reference count, and each pointer a weight value. If a new pointer to a cell is created by copying an existing pointer, the new and existing pointers are given weights whose sum equals the old value of the existing pointer. In this way, the sum of the weights of the pointers to any cell are maintained equal to its reference count. If a pointer is destroyed, the reference count of the cell to which it points is reduced by the weight of the pointer. Thus, when the reference count of a cell reaches zero, it is safe to assume that there are no more existing pointers to it, and hence that cell may be reclaimed (garbage-collected) for re-use.

Invariante!



# Ein hierarchisches Patent

**CLAIMS:-**

**1. A *computer system* having storage means containing a plurality of memory cells, at least some of which contain pointers to others of the cells thereby defining a directed graph structure in which each cell represents a node of the graph, wherein**

**(a) each pointer has a *variable weight value* associated with it,**

**(b) each cell contains a variable reference count value,**

**(c) whenever a cell is *allocated* in the storage means, its reference count is initially set to a predetermined value and the weights of any pointers to that cell are set to non-zero values such that the sum of those weights (or the weight if there is only one such pointer) is equal to the reference count of that cell,**

**(d) whenever a new pointer to a cell is created by *copying* an existing pointer, the new pointer and the existing pointer are given non-zero weight values whose *sum equals the old weight value* of the existing pointer, but the reference count of the cell is unaltered,**

**(e) whenever a pointer is *destroyed*, the reference count of the cell to which it points is reduced by the weight of that pointer, and**

**(f) cells with reference count equal to zero are reclaimed.**

2. A system according to Claim 1 wherein the weight of each pointer is a *power of two*, and wherein, whenever a pointer is created by copying an existing pointer of weight greater than one, the weight of the existing pointer is divided by two and the result is stored as the weight value of the new and existing pointers.

3. A system according to Claim 2 wherein the weight of each pointer is *encoded in a compressed form as the logarithm* to the base two of the weight.

4. A system according to any preceding claim wherein, whenever a pointer is to be created by copying an existing pointer of *weight equal to one*, a *dummy cell* is created into which the existing pointer is copied along with its weight, the dummy cell is given a reference count greater than one, and the new and existing pointers are both made to point to this dummy cell, and are given non-zero weights whose sum equals the reference count of the dummy cell.

5. A system according to any preceding claim in which the system is a *multi-processor distributed computer system*.



## 10 CLAIMS

US-Version 4755939: hier wird nun eine *Methode* patentiert, nicht mehr, wie im europäischen Patent, ein *Computersystem*!

I claim:

1. A method of garbage collection in a computer system having storage means containing a plurality of memory cells, at least some of which contain pointers to others of the cells thereby defining a directed graph structure in which each cell represents a node of the graph, wherein the method comprises steps as follows:
   - (a) each pointer is provided with a variable weight value,
   - (b) each cell is provided with a variable reference count value,
   - (c) whenever a cell is allocated in the storage means, the reference count of the cell is initially set to a predetermined value and the weights of any pointers to that cell are set to non-zero values such that the sum of those weights (or the weight if there is only one such pointer) is equal to the reference count of that cell,
   - (d) whenever a new pointer to a cell is created by copying an existing pointer, the new pointer and the existing pointer are given non-zero weight values whose sum equals the old weight value of the existing pointer, but the reference count of the cell is unaltered,
   - (e) whenever a pointer is destroyed, the reference count of the cell to which the pointer points is reduced by the weight of that pointer,
   - (f) and cells with reference count equal to zero are reclaimed.

2. A method according to claim 1 wherein the weight of each pointer is a power of two, and wherein, whenever a pointer is created by copying an existing pointer of weight greater than one, the weight of the existing pointer is divided by two and the result is stored as the weight value of the new and existing pointers.

3. A method according to claim 2 wherein the weight of each pointer is encoded in a compressed form as the logarithm to the base two of the weight.

4. A method according to claim 3 wherein, whenever a pointer is to be created by copying an existing pointer of weight equal to one, a dummy cell is created into which the existing pointer is copied along with its weight, the dummy cell is given a reference count greater than one, and the new and existing pointers are both made to point to this dummy cell, and are given non-zero weights whose sum equals the reference count of the dummy cell.

5. A method according to claim 3 in which the system is a multi-processor distributed computer system.

6. A method according to claim 2 wherein, whenever a pointer is to be created by copying an existing pointer of weight equal to one, a dummy cell is created into which the existing pointer is copied along with its weight, the dummy cell is given a reference count greater than one, and the new and existing pointers are both made to point to this dummy cell, and are given non-zero weights whose sum equals the reference count of the dummy cell.

7. A method according to claim 2 in which the system is a multi-processor distributed computer system.

8. A method according to claim 1 wherein, whenever a pointer is to be created by copying an existing pointer of weight equal to one, a dummy cell is created into which the existing pointer is copied along with its weight, the dummy cell is given a reference count greater than one, and the new and existing pointers are both made to point to this dummy cell, and are given non-zero weights whose sum equals the reference count of the dummy cell.

9. A method according to claim 8 in which the computer system is a multiple-processor distributed computer system.

10. A method according to claim 1 in which the computer system is a multi-processor distributed computer system.

# Patentsuche

- Sucht man nach Garbage-Collection-Verfahren (z.B. bei den WWW-Sites der Patentämter), findet man einiges...

*Search Results*

*Query: (garbage collection)*

man hätte besser auch nach "garbage collector" etc. gefragt

*139 of 2569032 matched* (Jan. 2000: ein Jahr zuvor: 93 of 2461228)

5652883 Computer method and system for conservative-stack and generational heap garbage collection

5561785 System for allocating and returning storage and collecting garbage using subpool of available blocks

5560003 System and hardware module for incremental real time garbage collection and memory management

5446901 Fault tolerant distributed garbage collection system and method for collecting network objects

5819299 Process for distributed garbage collection

5832529 Methods, apparatus, and product for distributed garbage collection

5033930 Garbage collecting truck

5398334 System for automatic garbage collection using strong and weak encapsulated pointers

4755939 Garbage collection in a computer system

5901540 Garden tool for collection and removal of debris

**...**



# Andere Garbage-Collection-Patente

- Es gibt viele Patente zum Thema "Garbage Collection", und manches hätte man vielleicht selbst erfinden können

- Titel: Fault tolerant distributed garbage collection system and method for collecting network objects
- Veröffentlichungsnr.: US5446901
- Veröffentlichungsdatum: 1995-08-29
- Erfinder: OWICKI SUSAN S (US); BIRRELL ANDREW D (US); NELSON CHARLES G (US); WOBBER EDWARD P (US)
- Anmelder: DIGITAL EQUIPMENT CORP (US)
- Klassifikationssymbol (IPC): G06F12/00

A distributed computer system includes a multiplicity of concurrently active processes. Each object is owned by one process. Objects are accessible to processes other than the object's owner. Each process, when it receives a handle to an object owned by any other process, sends a first "dirty" message to the object's owner indicating that the object is in use. When a process permanently ceases use of an object handle, it sends a second "clean" message to the object's owner indicating that the object is no longer in use. Each object's owner receives the first and second messages concerning usage of that object, stores data for keeping track of which other processes have a handle to that object and sends acknowledgement messages in return. The receiver of an object handle does not use the handle until its first message is acknowledged. Periodically, the object's owner sends status request messages to other processes with outstanding handles to that object to determine if any of those processes have terminated and updates its stored object usage data accordingly. A garbage collection process collects objects for which the usage data indicates that no process has a handle. The first and second messages include sequence numbers, wherein the sequence numbers sent by any process change in value monotonically in accordance with when the message is sent. Object owners ignore any message whose sequence number indicates that it was sent earlier than another message for the same object that previously received from the same process.

# Andere Garbage-Collection-Patente (2)

- Man findet auch Querverweise auf viele andere Patente, auf wissenschaftliche Literatur dazu etc., z.B.:

5355483 : Asynchronous garbage collection
INVENTORS: Serlet; Bertrand, Paris, France
ASSIGNEES: NeXT Computers, Redwood City, CA
ISSUED: Oct. 11, 1994
FILED: July 18, 1991
SERIAL NUMBER: 732453

...With this method, the process being collected communicates its memory state ("a memory snapshot") to a garbage collecting process (GC), and the GC process scans the memory and sends back the information about garbage. As a result, the present invention permits garbage collection to be performed asynchronously. The process being scanned for garbage is interrupted only briefly, to obtain the memory snapshot. The process then runs without interruption while the garbage collection is being performed. The present invention makes the assumption that if an object is garbage at the time of the memory snapshot it remains garbage any time later,...



# Beispiele für Softwarepatente (1)

Es gibt viel ganz offensichtliche Dinge, die patentiert wurden, das gilt auch für Algorithmen. Einiges davon ist in der fachlich beschlagenen Öffentlichkeit bekannt geworden, z.B. das LZW-Komprimierungspatent (US4558302) beim gif-Bildformat. Es gibt aber auch noch viele andere "interessante" Patente. Vieldiskutiert ist die Frage, ob Algorithmen (oder ganz allgemein "Intellectual Property") überhaupt patentiert werden soll.

---

**Any word processor with a separate mode that the user selects when they wish to type in a mathematical formula.** [US5122953]

**A word processor which marks and makes correction to a document using two additional different colors.** [US5021972]

Inventor(s): Nishi; Toshio
June 4, 1991
*A word processor, including a keyboard through which characters can be inputted, a memory device for storing inputted character arrays and a display device capable of multi-color displays, is so programmed that corrections and additions are automatically displayed in a different color from the rest for the convenience of editing. ... When a completed document is finally stored in a document file, however, such color codes are deleted such that the document can be outputted in one color.*

**Statically allocating an initial amount of memory when a program is first loaded according to a size value contained in the program header.** [US5247674]

Applicant(s): Fujitsu Limited, Kawasaki, Japan
Issued/Filed Dates: Sept. 21, 1993 / May. 13, 1991
*A memory allocation system includes a unit for storing the information about the amount of memory required at the time of initializing each executable program in the control information of the file storing the program. The amount required is determined when the program is translated, assembled or compiled and linked. The memory allocation system also includes a unit for reading the information, indicating the amount of memory required at the time of initializing the program stored in the control information of the file, when loading of program is requested.*

# Beispiele für Softwarepatente (2)

**Assigning a client request to a server process by first examining all the server processes not handling the maximum number of clients, and then assigning it to the server process currently servicing the fewest clients.** [US5249290]

Applicant(s): AT&T Bell Laboratories, Murray Hill, NJ
Issued/Filed Dates: Sept. 28, 1993 / Feb. 22, 1991
*A server of a client/server network uses server processes to access shared server resources in response to service requests from client computers connected to the network. The server uses a measured workload indication to assign a received client service request to a server process... a busy indicator provides a measured workload indication for each active process. The server uses the busy indicator to assign a new client service request to the least busy process.*

**Method for canonical ordering of binary data for portable operating systems.** [US4956809]

Applicant(s): Mark Williams Company, Chicago, IL
Issued/Filed Dates: Sept. 11, 1990 / Dec. 29, 1988
*...The method includes converting all binary data accessed from a file or communications channel from the canonical order to the natural order of the host computer before using the binary data in the host computer and converting all binary data which is to be sent to a file or communications channel from the natural order of the host computer to the canonical order before sending the binary data.*

**Remembering file access behavior and using it to control the amount of read-ahead the next time the file is opened.** [US5257370]

**Using of multiple read only tokens and a single read write token to control access to a portion of a file in a distributed file system.** [US5175851]

**Quicksort implemented using a linked list of pointers to the objects to be sorted.** [US5175857]

**Intercepting calls to a network operating system by replacing the first few instructions of an entry point by a call to an intercept routine.** [US5257381]



# Beispiele für Softwarepatente (3)

**Distinguishing nested structures by color.** [US4965765]

Applicant(s): International Business Machines Corp., Armonk, NY
Issued/Filed Dates: Oct. 23, 1990 / May. 16, 1986
*A method of distinguishing between nested expressions, functions, logic segments or other text by using a different color for each nesting level.*

Das berüchtigte "xor-Cursor-Patent": **Method for dynamically viewing image elements stored in a random access memory array.** [US4197590]

Issued/Filed Dates: April 8, 1980 / Jan. 19, 1978
*...An XOR feature allows a selective erase that restores lines crossing or concurrent with erased lines. The XOR feature permits part of the drawing to be moved or "dragged" into place without erasing other parts of the drawing. ...supporting another image on the display without destruction of the initially stored image... logically exclusively ORing together the accessed data for each element of the stored image and the data for the corresponding element of the image to be superimposed, and for reentering the resultant logical data into the same memory locations, said display then being generated from the resultant contents of said memory.*

**A parallelizing compiler that estimates the execution time for each of a number of different parallelization conversions and then selects the one that it thinks will be the fastest.** [US5151991]

**Any document storage system that has a digital camera to scan in documents, stores the documents on an optical disk, and uses character recognition software to construct an index.** [US4941125]

---

Siehe auch:

http://www.freepatents.org/ (bzgl. der aktuellen Debatte, ob auch in Europa Algorithmen patentiert werden sollen)

http://www.base.com/software-patents/examples.html (zu obigen Beispielen)

# Bemerkungen zum WRC-Patent...

From: munnari!tasis.utas.oz.au!ben@uunet.UU.NET (Ben Lian)
Subject: Watson & Watson's Weighted Reference Counting
Date: 19 Jun 89 11:55:52 GMT

... I thought you'd like to know that the Watson & Watson algorithm was PATENTED (yes, patented) by ICL in the early '80s. According to one of my colleagues, the patent covers all uses of the algorithm in which the reference weight is split (in some arbitrary ratio). This suggests that the patent covers most ANY application of the algorithm. Bevan's algorithm, also in the PARLE 87 proceedings, is an elegant subset of Watson & Watson's. (Ironically, Bevan won the 'best paper' award at the conference!) The patent permits the use of the algorithm in research contexts only; use in a commercial product is not allowed. (Which means that our Department's VLSI parallel combinator reduction machine will never see the light of day.)

Ben Lian

P.S. On hearing that the algorithm had been patented, my colleague ran up and down the corridor of our hut, exclaiming: "How could they do this? They might as well patent the addition operator!"

---


```
Benjamin Y H Lian              ACSnet: ben@tasis.utas.oz
Dept. of EE & CS               ARPA  : ben%tasis.utas.oz.au@uunet.uu.net
University of Tasmania         BITnet: munnari!tasis.utas.oz!ben@
GPO Box 252C                           uunet.uu.net
Hobart, Tasmania 7001          UUCP  : {enea,hplabs,mcvax,uunet,ukc}!
A U S T R A L I A                      munnari!tasis.utas.oz!ben
```




# Ein weiterer Brief von Ben Lian

As I see it, since the algorithm itself is patented, then any technique which makes use of weighted reference counts in any form is covered by the patent. This is regardless of different terminology and/or the strategy used to split reference weights.

I hope this has been helpful. If you have any further questions you might perhaps like to write to Andrew Partridge in my Department. He will be able to provide you with more accurate information—I am only an interested by-stander! Alternatively, you can write to Derek Guyatt or Paul Watson (as per the enclosed letter from the latter).

Yours sincerely

Ben Lian
(ben@tasis.uucp)

# Watson war nicht der erste...

From: grit@carlos.llnl.gov (Dale Grit)
Message-Id: <8907101526.AA12531@carlos.llnl.gov>
To: ben@tasis.utas.oz
Subject: weighted reference counting
Status: R

I had heard about weighted reference counts in 1980 at a workshop at MIT, so I asked Arvind what he knew about them. He responded with the following:

"On reference weights: I brought the idea to the attention of Bob Thomas while he was doing his thesis at Irvine. I had heard it from Ken Weng in course of our many informal discussions. It is explained in a foot note in Ken Weng's Ph.D. thesis. I have mentioned the idea in my dataflow course at MIT as early as 1980. Paul (not our old buddy Ian) Watson gave a talk about it at MIT in 1986 (I think). He was quite surprised to learn that we knew about it so thoroughly. He acknowledges it in his Parle paper. He for a while was erroneously attributing the idea to me but I had him correct it. I did not know about the ICL patent. I will be surprised if it was filed in 1982. In any case I would not worry about it because it can't hold up in court if challanged. In fact 70 to 80% patents don't hold up when challanged."

If you need more information, please contact Arvind at

arvind@au-bon-pain.lcs.mit.edu

Dale Grit,
Computer Science Dept.,
Colorado State University
on leave at LLNL
grit@lll-crg.llnl.gov

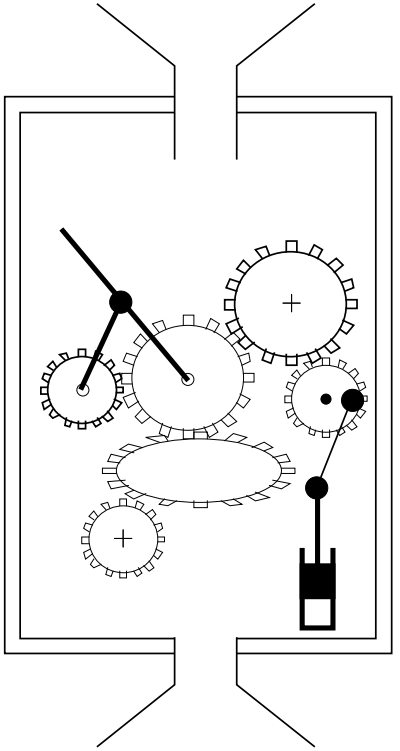


# Local reference counting (LRC)

**Y. Ichisugi, A. Yonezawa**: Distributed Garbage Collection Using Group Reference Counting, TR 90-014, Univ. of Tokyo

**M. Rudalics**: Implementation of Distributed Reference Counts, Internal Report, J. Kepler University, Linz

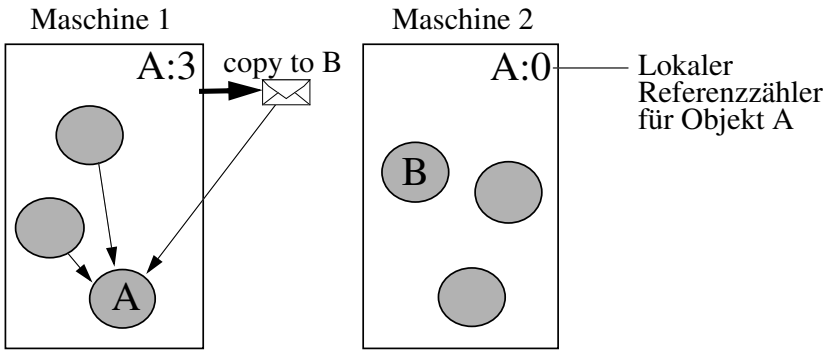
**J. Piquer**: *Indirect Reference Counting,* Proc. PARLE 91, 150-165



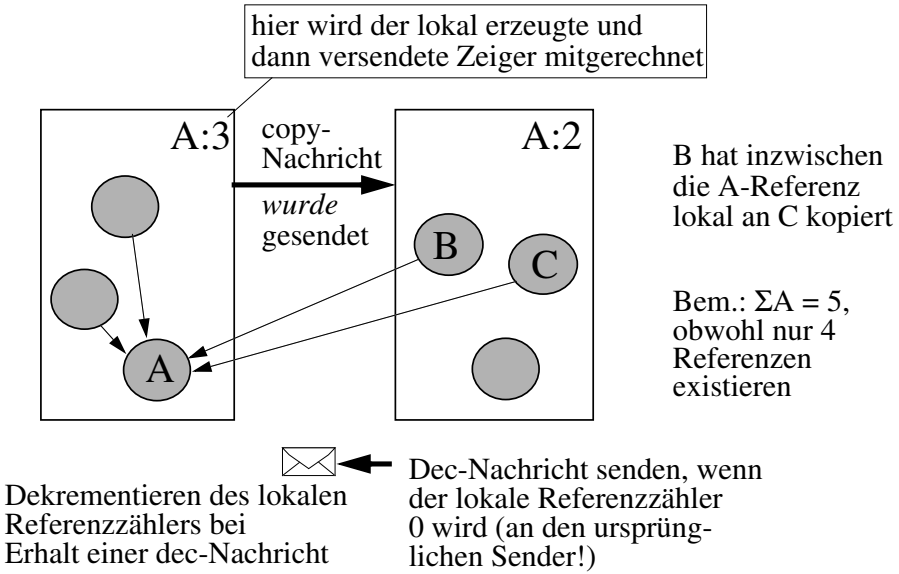
**E.W. Dijkstra, C.S. Scholten**: Termination Detection for Diffusing Computations. Inf. Proc. Lett. 11 (1980), 1-4

# LRC-Garbage-Collection-Verfahren

- Pro *Maschine* wird (potentiell) für jedes Objekt ein Referenzzähler gehalten, z.B. für Objekt A:



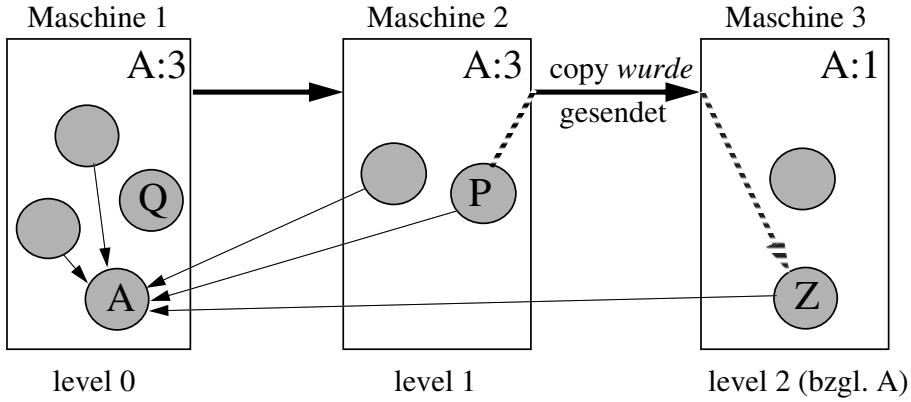

---




Beachte: Es wird angenommen, dass der *lokale* Referenzzähler *atomar* mit den Operationen copy, delete, Empfang einer dec-Nachricht aktualisiert werden kann



# LRC: Weitervererben von Referenzen




- Maschine 1 muss nicht informiert werden, wenn z.B. Objekt P eine (Kopie seiner) A-Referenz an Z schickt!

---

- Korrektheit (safety):

Referenzbaum

Lokaler Referenzzähler auf level i+1 > 0
--> lokaler Referenzzähler auf level i > 0

==> Falls eine Referenz existiert, dann ist der Referenzzähler auf level 0 > 0

- wieso eigentlich *Baum*?
- wie genau ist "level" definiert?

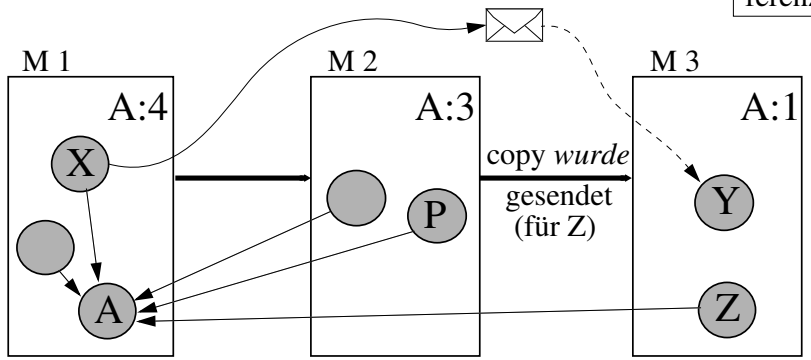
==> "Garbage-Kriterium": Referenzzähler auf level 0 = 0

- Denkübungen: wie steht es mit der *liveness*?
- was geschieht, wenn P eine A-Referenz an Q (Maschine 1) sendet? (--> Zyklen ?)

# Der "Remote-ref"-Baum

- Was geschieht, wenn X auf Maschine 1 eine copy-Nachricht mit einer A-Referenz an Y auf Maschine 3 schickt?
  - Maschine 1 erhöht ihren lokalen A-Zähler beim Senden
  - Maschine 3 erhöht ihren lokalen A-Zähler beim Empfang

die schon eine A-Referenz hat




- Aber: Wann und an wen (hier: Maschine 1 und/oder 2) soll ggf. eine dec-Nachricht von Maschine 3 gesendet werden?

?
- (1) wenn Y seine A-Referenz löscht --> an M1 senden, und wenn Z seine A-Referenz löscht --> an M2 senden
- (2) wenn der lok. Referenzzähler 0 wird auf M3 --> an M1 und M2 senden
- (3) M2 "adoptiert" Y bei Empfang der copy-Nachricht --> bei Empfang des copy eine dec-Nachricht an M1 senden (als hätte Y seine A-Referenz gleich gelöscht und dann sofort eine lokale Kopie von Z erhalten)

- Beachte bei der Lösung (3):
  - eindeutige Vorgängerbeziehung; keine Zyklen --> *Baum* ("level" klar bestimmt)
  - neuer "Adoptivvater" M2 braucht hierbei nicht informiert zu werden
  - genausogut hätte M1 Objekt Z adoptieren können (M3 sendet dann dec-Nachricht an M2 bei Empfang der copy-Nachricht von X) --> Optimierungspotential: wähle stets den Vater mit niedrigstem level... (wieso?)



# LRC: Formale Beschreibung

- Ergänzen existierender atomarer Aktionen bzw. zusätzliche Aktionen:

atomare Aktion: Änderung von LRC "gleichzeitig" mit dem Versenden der copy-Nachricht

$C_p$: {p ist erreichbar und hat eine r-Referenz}
**send** copy(r) **to** q;
$LRC_p(r) := LRC_p(r) + 1$

$R_p$: {Bei Empfang einer copy(r)-Nachricht von q}
Installiere die r-Referenz;
**if** $LRC_p(r) = 0$
**then** {$LRC_p(r) := 1$; $FIRST_p(r) := q$}
**else** {$LRC_p(r) := LRC_p(r)+1$; **send** dec(r) **to** q}
**fi**

r-Referenz existierte schon bzw. p befindet sich schon im Referenzbaum

$D_p$: {p hat eine r-Referenz}
Lösche die r-Referenz;
$LRC_p(r) := LRC_p(r) - 1$;
**if** $LRC_p(r) = 0$ **then call** collect(r) **fi**

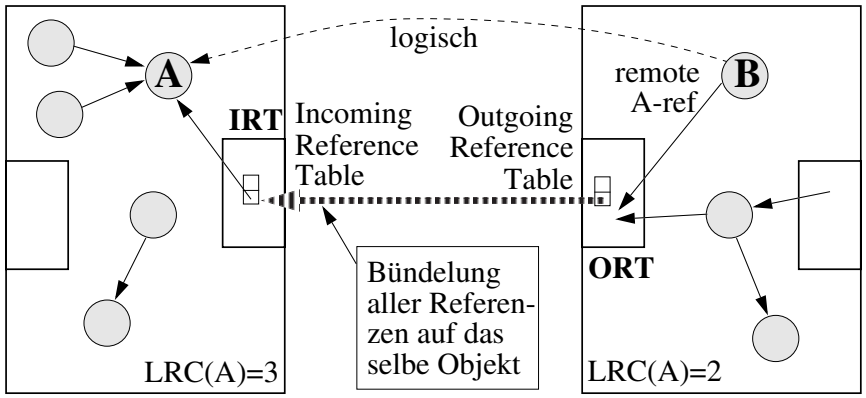
$X_p$: {Bei Empfang einer dec(r)-Nachricht}
$LRC_p(r) := LRC_p(r) - 1$;
**if** $LRC_p(r) = 0$ **then call** collect(r) **fi**

ggf. rekursives Freigeben!

**proc** collect(r):
r wurde lokal erzeugt
**if** $FIRST_p(r) = \varnothing$ **then** "*r is garbage*"
**else send** dec(r) **to** $FIRST_p(r)$; **fi**

- beim *lokalen* Kopieren von r wird lediglich $LRC_p(r)$ erhöht
- wieso wird eine dec-Nachricht gesendet, wenn schon eine r-Referenz existiert; könnte man darauf u.U. verzichten?

# Implementierungssicht: Remote-reference-Tabellen




- ORT: enthält Schattenobjekte als lokale Stellvertreter ("Proxy") für alle entfernten Objekte
  - beachte: hier ist Σ LRC(A) = 5 = Anzahl A-Referenzen (inkl. bei IRT!),

- IRT / ORT: Die Einträge bzgl. eines bestimmten Objektes (z.B. A) lassen sich als ein *einziges* Objekt, *verteilt* auf mehrere Maschinen auffassen

z.B. weil vom *lokalen Collector* als Garbage erkannt

*Idee*: Wenn ein Teilobjekt in ORT *gelöscht* wird, dann wird das entsprechende Gegenstück in der IRT gelöscht
(falls es sonst keine Teile in anderen ORT gibt)

--> ORT muss dazu eine Nachricht an IRT senden

- Remote-Referenzen sind ggf. *mehrfach indirekt*
  - u.U mehrfache Adressumsetzung bei *Zugriff* über eine Remote-Referenz, falls der Zugriff tatsächlich entlang der Referenzkette erfogt



# LRC: Eigenschaften

- Vergleich zu Verfahren von Lermen/Maurer, Rudalics etc:
  - keine Zusatznachricht bei copy
  - kein Verzögern von copy
  - oft: keine Zusatznachricht bei delete (gelegentlich: Abbau des Referenzbaumes)
  - kein Verzögern bei delete
  - FIFO nicht notwendig
  - Gesamtzahl der Nachrichten: genausoviele dec wie copy (wieso?)

Generell gilt: zyklischer Garbage zwischen verschiedenen Objekten lässt sich mit Referenzzählern nicht erkennen!

- Vergleich zu WRC:
  - Zähler einfacher handzuhaben als die Akkumulation von beliebig kleinen Gewichtsfragmenten
  - keine Komplikation bei RW=1

- Nachteil (gegenüber anderen Referenzzähler-Methoden):
  - Objekte besitzen i.a. mehrere (Referenz)zähler (angesiedelt z.B. in mehreren ORT-Tabellen) --> höherer Speicheraufwand

- Implementierung:
  - typischerweise je einen *lokalen Garbage-Collector* pro Maschine; kann nach anderem Verfahren arbeiten, z.B. mark&sweep (Zyklenerkennung!)
  - Proxyobjekte in der IRT werden dabei als Wurzelobjekte angesehen

- Migration von Objekten lässt sich einfach realisieren!
  - nur Zielmaschine und Quellmaschine sind betroffen
  - wie realisiert? (vgl. jeweils Objekte A und B in vorheriger Skizze)
  - wieso einfacher als bei anderen GC-Verfahren?
  - Denkübung: Präzisierung (z.B. Zyklen?)...
    ... und Optimierung (z.B. Verkürzung von Indirektionsketten)