

Distributed Operating Systems

11th Chapter
Memory

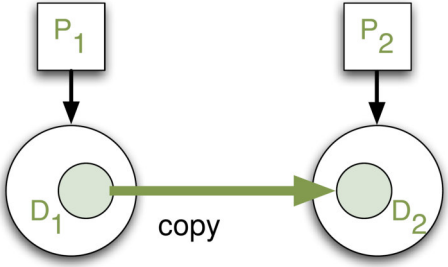
Matthias Werner
Chair Operating Systems

11.1 Foundations

Repetition “Operating Systems”

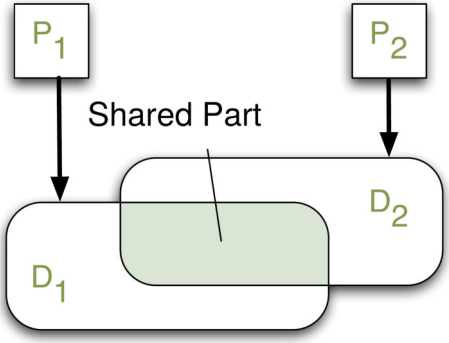
- There are two programming models for the interaction of concurrent processes:

Message Passing
(=explicit data transport)



directed relation

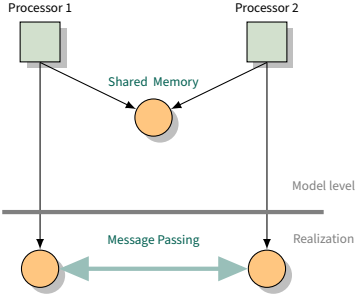
Shared Memory
(=implicit data transport)



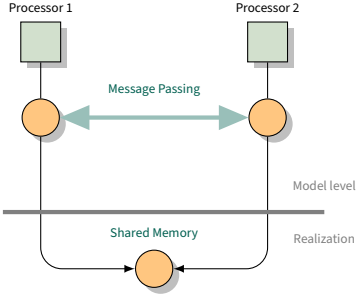
symmetric relation

Realization

- The two forms of information exchange are **concepts**
- They do **not** state anything about the implementation



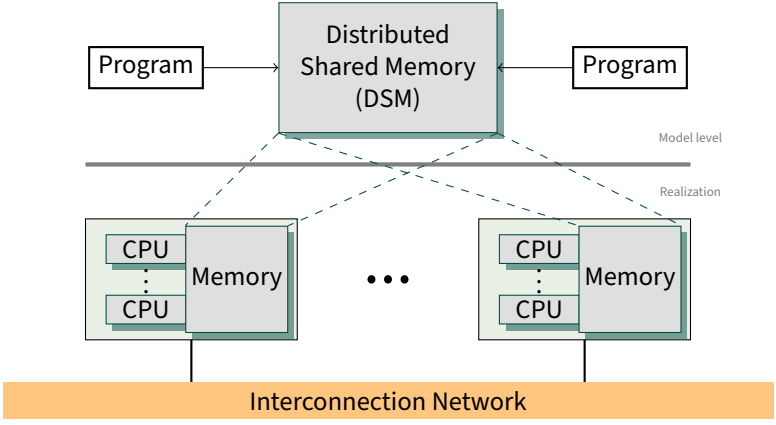
Shared memory realized through message passing



Message passing through shared memory

Distributed Memory

- Following, we discuss the realization of shared memory in a distributed system



Distributed Memory (cont.)

- Even cooperation and communication (shared memory and message passing) are equivalent mighty, most programmers prefer cooperation
  - No need to make exchange of information explicit
  - Default case

- However, programming of concurrent applications with shared memory is error-prone, too
- **Example**: Is the following mutex implementation correct?

```
// Prozess A
while(1){
  while(var == 1);
  var = 1;
  /* Kritischer Abschnitt */
  var = 0;
}
```

```
// Prozess B
while(1){
  while(var == 1);
  var = 1;
  /* Kritischer Abschnitt */
  var = 0;
}
```

Distributed Memory (cont.)

- We are “accustomed” to a specific behavior of memory, and assume it frequently without care: Memory access
  - immediately ➡ a written value can always be read
  - ordered ➡ accesses do not pass each other
  - uniform ➡ all read accesses at a given time result in the same value
- These assumptions do rarely hold in non-distributed case, distribution makes it worse
- Have to consider consistency, again

11.2 Consistency

Consistency problem

- Example: Is “ok” always printed?

Process 1

```
br:= b;
ar:= a;
if (ar>=br) then print("ok");
```

Process 2

```
a:=b:=0;
a:=a+1;
b:=b+1;
```

- Obviously, there is an unwanted order-related effect
- Synchronization may solve this problem, but:
  - Performance decreases through synchronization overhead
  - Overhead increases with increasing number of nodes
- Consensus protocols pay this price
- Alternatively, think about graduation of consistency

Notation

- We compare several **kinds** of consistency
- Finally, a requirement to the order from a global point of view (see Chapter 4)
  - Global total order of all accesses
  - Global total order regarding specific data
  - Global causal order
  - Total order regarding the accesses of an application
  - ...
- Notations:
  - r_2(y=23) ➡ Read operation for variable y that reads value 23
  - w_1(x=42) ➡ Write operation of node 1 for x that assignes value 42 to x
  - Die Benennung des lesenden/schreibenden Knoten wird mitunter ausgelassen
- Wenn nichts anderes gesagt, wird davon ausgegangen, dass die Variablen anfangs den Wert 0 haben

Strict Consistency

Definition 1 (Strict Consistency)

A distributed memory realizes **strict consistency** if every read access returns the value of the globally last write operation.

Annotations:

- Strict consistency is what one expects from a non-distributed system
- Other name: “atomic consistency”
- It assumes that there is a definition of “last access”
  ➡ global time-/order model
- Problem occured first at the end of the 70s in data bases [Lin79]

Examples

- The following scenario fulfills strict consistency
  - Abscissa represents global time

P1:	w_1(x=1)		w_1(y=2)	
P2:		r_2(x=1)		r_2(y=2)

- The following scenario is **not** strict consistent

P1:	w_1(x=1)	w_1(y=2)		
P2:		r_2(x=0)	r_2(x=1)	r_2(y=1)

Problem: Globale Zeit

- Lokal liegen keine Informationen über eine globale Zeit vor, zumindest i.d.R. nicht mit der notwendigen Granularität
- Daher muss bei jeder lokalen Zeitachse von einer anderen Zeitskala ausgegangen werden
- Die folgenden beiden Läufe sind dann (bei Wegfall der globalen Abzisse) gleich:

P1:	w_1(x=1)		w_1(y=2)
P2:		r_2(x=1)	r_2(y=2)

P1:		w_1(x=1)	w_1(y=2)
P2:	r_2(x=1)		r_2(y=2)

- Die EInführung eines Mechanismus zur Sicherstellung, dass jeder Leseoperation eine eindeutige globale Zeit zugeordnet werden kann (Lock-Step) würde das verteilte System extrem ausbremsen

Sequential Consistency

Definition 2 (Sequential Consistency)

A memory realizes **sequential consistency** if all processes see the same order of memory accesses

Annotations:

- Defined by Lamport to characterize multiprocessor systems
- Each process recognizes its own order in the global order
- Interleaving over all processes is arbitrary
- The value of the last global write operation is not necessarily delivered
- Two runs of the same program may deliver different results

P1:	w_1(x=1)		
P2:		w_2(y=2)	
P3:			r_3(y=2) r_3(x=0) r_3(x=1)
P4:			r_4(x=0) r_4(x=1)

Example

- 3 processes work on shared variables a, b and c, that were initialized with 0

P1:
```
a=1;
print(b,c);
```

P2:
```
b=1;
print(a,c);
```

P3:
```
c=1;
print(a,b);
```

- For 6 instrcutions there are theoretically $6! = 720$ execution orders
- The **outer** perception counts, instrcutions may overtake eachother
- 90 out of 720 possibilities are sequentially consistent, i.a.:

Order 1:
```
a=1;
print(b,c);
b=1;
print(a,c);
c=1;
print(a,b);
```
Ergebnis: `001011`

Order 2:
```
a=1;
b=1;
print(b,c);
print(a,c);
c=1;
print(a,b);
```
Ergebnis: `101011`

Order 3:
```
b=1;
c=1;
print(a,b);
print(a,c);
a=1;
print(b,c);
```
Ergebnis: `010111`

Order 4:
```
a=1;
b=1;
c=1;
print(b,c);
print(a,c);
print(a,b);
```
Result: `111111`

Causal Consistency

Definition 3 (Causal Consistency)

A memory realizes causal consistency if all write operations, which are potentially causally dependent on eachother, are seen in the same correct order by all processes.

Annotations:

- Weaker than sequential consistency, since the definition refers to causally related write operations only
- Two write operations are potentially causally dependent if a read operation, which may have influenced the result of the second write operation, has taken place before the second write operation
- If $w_2()$ is potentially causally dependent on $w_1()$, the order $w_1()$ $w_2()$ is correct

Example

- The following execution is causally consistent, but not sequentially consistent
- w_1(x=1) and w_2(x=2) are causally dependent, since P2 recognized the write operation
- P3 and P4 recognize the updates in different orders, but the corresponding read operations are not causally dependent
- There is no global timeline anymore, but a global order per process only!

P1:	w_1(x=1)			w_1(x=3)	
P2:		r_2(x=1)	w_2(x=2)		
P3:		r_3(x=1)		r_3(x=3)	r_3(x=2)
P4:		r_4(x=1)		r_4(x=2)	r_4(x=3)

FIFO Consistency

Definition 4 (FIFO Consistency)

A memory realizes **FIFO consistency** if all **write** operations of each single process are recognized by all other processes in the same order, while **write** operations of **different** processes may be recognized in different orders.

Example:

- A process P1 executes w_{11} and w_{12}, and a process P2 w_{21} and w_{22}
- ➡ Execution fulfills FIFO consistency from a global point of view if P1 and P2 recognize any (also different) of these orders:
  (w_{11} w_{12} w_{21} w_{22}), (w_{11} w_{21} w_{12} w_{22}), (w_{11} w_{21} w_{22} w_{12}), (w_{21} w_{11} w_{12} w_{22}),
  (w_{21} w_{11} w_{22} w_{12}), (w_{21} w_{22} w_{11} w_{12})
- Considers different latencies for communication

FIFO Consistency (cont.)

- Write accesses of different processes can be seen in different orders by different processes
- The term FIFO consistency is used in this manner by TANENBAUM/VAN STEEN only; other names are **PRAM consistency** or **processor consistency** [1]
- FIFO consistency may lead to unexpected results:

P1

```
a=1;
if (b==0) kill(P2);
```

P2

```
b=1;
if (a==0) kill(P1);
```

Under FIFO consistency **both** processes can (possibly) be terminated !

[1] **Attention:** In literature the term “processor consistency” is assigned to (at least) three different semantics, thus we prefer the less common term “FIFO consistency” in this lecture.

Example

- P3 and P4 see both write operations in different orders
- Since they were performed by different processes, the execution fulfills FIFO consistency, but not causal consistency, since both write operations are (potentially) causally dependent
- There is no global timeline anymore, but a global order per process only!

P1:	$w_1(x=1)$			
P2:		$r_2(x=1)$	$w_2(x=2)$	
P3:			$r_3(x=1)$	$r_3(x=2)$
P4:			$r_4(x=2)$	$r_4(x=1)$

Weak Consistency

- Usually, processes need a consistent view on the memory in specific situations only
- Hence, weak consistency differs **normal** variables from **synchronization variables**

Definition 5 (Weak Consistency)

A memory realizes **weak consistency** if the following conditions hold:

- All accesses to synchronization variables fulfill sequential consistency
- All previous write operations are everywhere finished, before an access to a synchronization variable is granted
- All previous accesses to synchronization variables are finished, before an access to a normal variable takes place

Example

- Synchronization forces an update of the values in the memory
- If a process wants to render its write accesses visible to other processes, it has to synchronize
- If a process wants to access the latest data, it has to synchronize too
- If a process does not synchronize before a read access, it **may** see outdated values

P1:	w_1(x=1)	w_1(x=2)	S		
P2:		r_2(x=1)	S	r_2(x=2)	
P3:		r_3(x=2)	r_3(x=1)	S	

Discussion

- The presented consistency models are not exhaustive
- Other models are *local consistency*, *slow consistency*, *release consistency*, *memory coherence* or *entry consistency*
- Models have dependencies, e.g.:
  - Sequential consistency implies causal consistency.
  - Causal consistency implies FIFO consistency.
- Auch sind die hier dargestellten Modelle etwas vereinfacht: eigentlich müsste genauer zwischen den Zugriff auf eine und dieselbe Variable (**Koheränz**) und auf unterschiedliche Variablen unterschieden werden
  - Unter diesem Gesichtspunkt gibt es eine ganze Reihe von Modellen, die im Bereich FIFO/Prozessor/PRAM angesiedelt sind
- A program is correct under a model, if and only if it delivers an acceptable result for all possible sequences
- Generally: The “closer” a consistency model is to the non-distributed case, the more costly is the implementation

11.3 Linda

- Consistency ist **just** a problem because our thinking is deeply rooted in the memory architecture of the von-Neumann computers
  - memory: value that can be accessed via an address
- Alternative approach Linda: **associative memory**
  - value and address are no longer separated ➡ tuple
  - classical memory cell can be seen as a 2-tupel (address, value)
  - Linda-tupel is a general n-tupel
  - Known from caches, e.g. TLB
- **Tupelspace:**
  - Process writes and reads tupel from/to a tupel space
  - Tupel has no (specific) **place** ➡ there can be multiple identical tupel **simultaneously**
- Property: Consistency is **not** a problem because there is no outrun possible

Memory primitives

- Primitives in Linda:
  - `in` reads and deletes a tupel from tupel space (atomic)
  - `rd` reads a tupel non destructive
  - `out` writes a tupel to tupel space
  - `eval` generates a new process to evaluate the tupel and writes the result to tupel space
- In case of read operations the tupel components form a logical name
- Example: `in(42,?,'a',?)` returns *a* tupel whose fist component is 42 and the third one is 'a'
  - If there is no such tupel, the operation blocks until one shows up in tupel space

Example

- Tokenpassing-Algorithms in C-Linda [CaGe93]

```
#define NPROC 4

ring(id){
  if (id) in("token", id);
  out("token",(id+1)%NPROC);
  if (!id) in("token",id);
}

real_main(){
  int i;
  for(i=1;i<NPROC;i++) eval("ring node", ring(i));
  ring(0):
 }
```

11.4 Garbage Collection

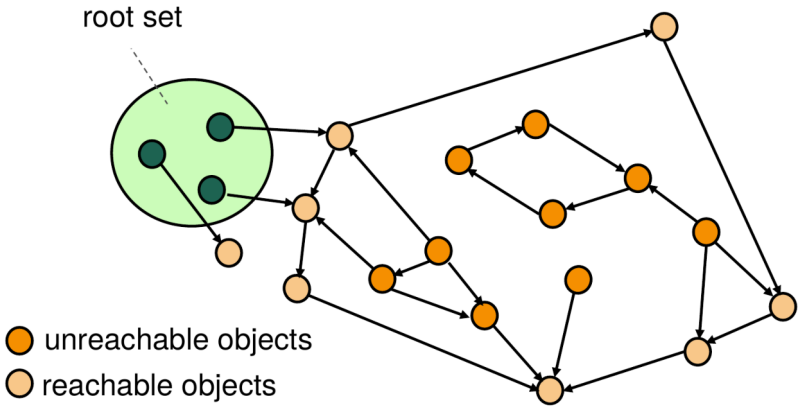
- Originally, memory management was manually performed by the programmer
  - `new` and `delete` in C++
  - `malloc` and `free` in C
- Manual memory management is even in centralized systems highly complex!
  - **Program failures** through faulty memory management (e.g. use of released memory)
    ➡ **dangling pointer**
  - **Memory leaks** through forgotten release of allocated memory
    ➡ **memory leak**

Garbage Collection (cont.)

- Due to error-proneness, memory management is increasingly automated by runtime systems (e.g. in Java, Go, C#)
  ➡ **transparent** garbage collection
- **Goal**: Release of memory blocks (e.g. objects), which cannot be accessed anymore
- Consider/repeat the non-distributed case first

Graph of Object References

- All objects that are reachable from the **root set** are still required; all others may be released



Requirement: Security and Liveliness

- As for other algorithms, there are security and liveliness requirements
  - **Security**
    Only objects that are not accessable anymore, are released
  - **liveliness**
    An object that is not accessable anymore, is released after a finite amount of time

Concepts

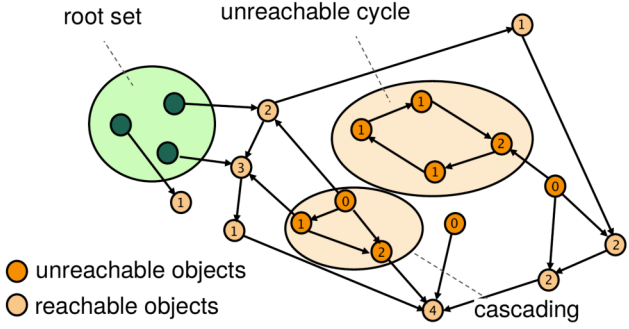
- In this context, we distinguish two typs of algorithms:
  - **Mutator**: Manipulates variables that contain references to objects
    ➡ application (base algorithm)
  - **Collector**: Finds objects that may be released and releases them
    ➡ control algorithm
- Mutator and collector are partially entangled, partially truely concurrent
- Principally, there are two classes of collector approaches:
  - Tracing algorithms
  - Reference counting
- We start with the latter

11.4.1 Reference Counting

- Each object gets a reference counter
  - If a new reference is created, the reference counter is incremented
  - If a reference is deleted, the reference counter is decremented
  - If a reference counter reaches zero, there is no reference to this object and it is deleted
- **Use cases**: Apple (Objective-C, Swift), Perl, COM

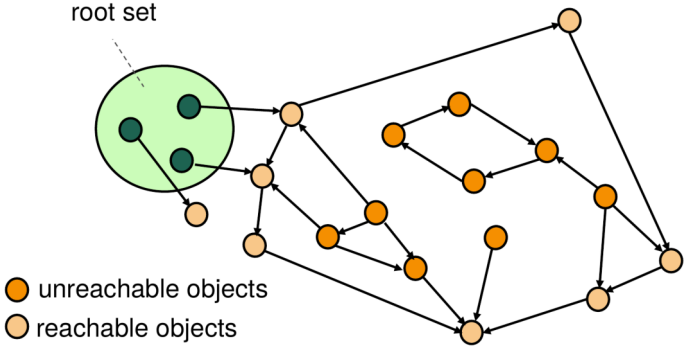
Discussion

- **Advantages**:
  - Effort for garbage collection is distributed over runtime
  - No stop-the-world approach necessary
- **Disadvantages**:
  - Cascade effects possible
  - No detection of reference-cycles



11.4.2 Tracing

- **Basic idea**:
  - “Tracing” of reachability from the root set
- **Classical algorithm**: Mark & Sweep
- **Use cases**: JVM-Sprachen (Java, Scala, Conjure), Smalltalk, Python



Mark & Sweep

- The algorithm uses a mark for each object and proceeds in two phases
- 1. Phase: **Mark**
  - The system is stopped
  - The mark of all objects is deleted
  - Starting at the root set, all references are traced and the referenced objects are marked
  - This procedure is recursively continued, until no unmarked objects are reached anymore (tree search)
- 2. Phase: **Sweep**
  - All unmarked objects are released
  - The system is continued

Discussion

- **Advantages:**
  - All objects to release are securely identified
  - No/very low effort at the objects (1 Bit)
- **Disadvantages:**
  - Freeze the system (*stop the world*)
  - The whole (used) memory needs to be inspected
    ➡ inconvenient in combination with **demand paging**
- Mark & Sweep is rarely used in this pure form ➡ variety of improvements

Discussion – Improvements

- **Tricolor**:
  - Distinction of three classes: “securely releasable”, “securely not releasable” and “maybe releasable”
  - Only last class needs evaluation
  - ➡ performance gain
- **Snapshot**:
  - Observation: once unreachable objects remain unreachable
  - Approach: make concurrent (consistent) snapshot and perform garbage collection based thereon
  - In the meantime, mutator generates new unreachable objects that are released in the next run
  - ➡ improved runtime behavior, but higher memory rerquirements
- **Generational algorithms**:
  - Observation: many objects “live” very short, others longer
  - Approach: distinction by lifespan and adaptation of strategies
  - ➡ better tradeoff

11.4.3 Garbage Collection in Distributed Systems

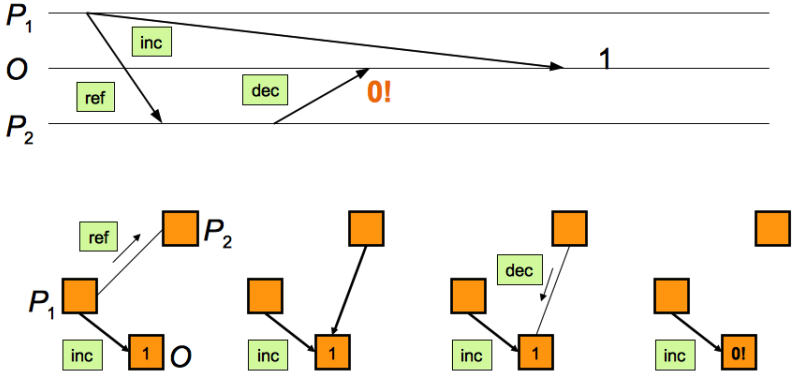
- Consider distributed (operating) systems
- More complex than in centralized systems
- Remote references allow the access to objects that are located on other computers
- **Important**: remote references may be **sent** by messages
- Coordination (as always) by messages only

Garbage Collection vs. Termination

- The garbage collection problem is obviously similar to the termination problem:
  - In both, base algorithm and control algorithm are overlayed and executed concurrently
  - “Global termination” and “object can be released” are both stable predicates
- Control algorithm is supposed to detect the stable predicate
- **Problem**: actions behind the back of the control algorithm
  - **Termination**: send message that reactivates another process
  - **Garbage collection**: copy and send reference that still allows to access the object
- Con solutions for one problem be transferred to the other? (Spoiler: partially yes)

Misinterpretation

- first consider reference counting
- Increment and decrement messages can outrun each other directly and indirectly
- This can lead to an inconsistent view and hence a misinterpretation



Solution

- Approaches to eradicate possible misinterpretations
- **Synchronous** communication
  - The reference is send after the increment message
  - The reference is send after the increment message has been received due to synchronous communication.
- **Acknowledgement** of the increment message
  - The increment message is send
  - The increment message is acknowledged by the receiver
  - After receiving the acknowledgement the reference is send
- Receiving messages is forced to be in **causal order**
  - From the perspective of the receiver messages can neither directly nor indirectly outrun each other

Weighted reference counting – Credit method

- Based on an algorithm by Shing-Tsaan Huang (1989), that was constructed initially to detect termination
- **Idea**:
  - Each newly (outside of the root set) created object has a **credit value** of 1
  - By sending an object reference the message contains half of the current credit; the credit of the referenced object is decreased accordingly
  - If a reference is deleted the respective credit is send to the referenced object
  - If an active process receives a message, its credit increases by the credit value of the message

Weighted reference counting – Credit method (cont.)

- **Invariant**
  - The credit sum of all references for a certain object is always 1
  - Reference messages always have a credit value > 0
  - Referenced objects always have a credit > 0
- **Clean up**
  - If a credit sum for a certain object reaches 1 again then there are no references anymore and it can be deleted

Optimization: Description of credit fractions

- floating point inappropriate ➡ rounding error
- **Better:** Save the negative dual logarithm of the credit fraction $c = -\log_2 k$
  - $k = 1$ ➡ $c = 0$
  - $k = \frac{1}{2}$ ➡ $c = 1$
  - $k = \frac{1}{4}$ ➡ $c = 2$
  - ...
- Just divide and double necessary; corresponds to increment/decrement of c
  - $c' = c \pm 1$
- Referenced object has a set of credit values (bitvector)
- Two identical credit values are unified to the next bigger one
- **Problem:** By using an integer representation credits cannot be split arbitrarily

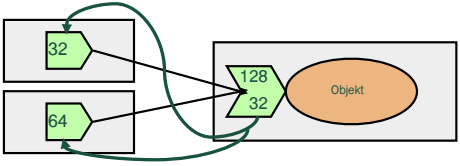
Weighted reference counting – Variant

The following approach solves the problem of weight distribution

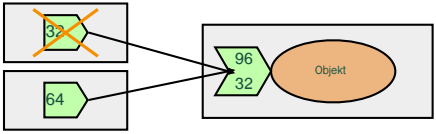
- Object has **two** weights: **partial weight** and **complete weight**
- Both are initially equal and are powers of two $g = 2^c$
- References have just one partial weight
- The weight is saved as a natural number in normal format (not as a logarithm)
  ➡ Division is c-times possible

Weighted reference counting – Variant (cont.)

- Output of a remote-reference or copy of a remote-reference like before (partial weight is divided by two)



- Deletion of a remote-reference: partial weight is send to the object and is **substracted** from its complete weight



Weighted reference counting – Variant (cont.)

- **Invariant**
  Sum of all partial weights = complete weight of the object
- Object can be cleaned up if partial weight of the object = complete weight of the object
- What happens if a partial weight is 1 and it has to be divided by two?
  - Increase complete weight and partial weight of the object or reference by $g - 1$ (e.g. 127
  - Invariant is **preserved**, since both sides of the equation are increased by the same number

Distributed Mark & Sweep

- The system is freezed in the beginning and hence the algorithm can be used in the distributed case
- Freeze and unfreeze can be done with e.g. a with wave algorithm
- Freezing is in the distributed case a even stronger intervention than in the local (just concurrent) case
- Solution approach:
  - **Logically Centralized Reference Service** (LADIN und LISKOV)
  - **Using time stamps** (HUGHES)

Approach of Ladin and Liskov

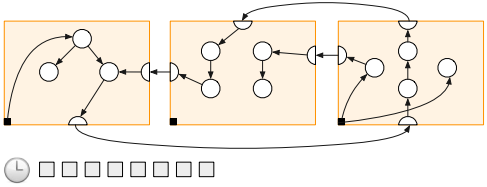
- All nodes send their local view to a central server
- There a global reference graph is constructed ➡ snapshot
- Mark & Sweep on central reference graph; clean up information is send to the nodes

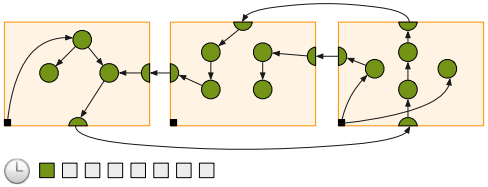
- **Diskussion**
  - Consistent snapshot is non-trivial (see Chapter) 10
  - Clean up is not up to date
  - Central server is the bottleneck

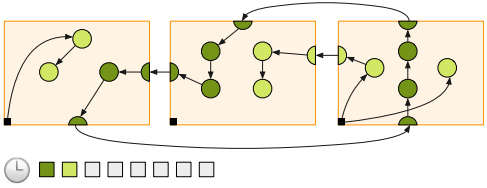
Approach of Hughes

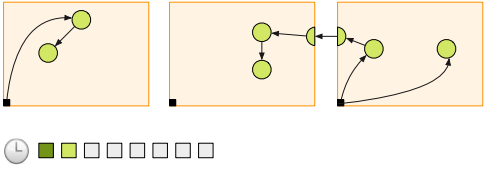
- Requirement: gobal time ➡ clock synchronization
- Marking via timestamps, not bits
- Each node has a local marking phase and propagates extern reference information
- received reference informations are updated
- Objects with a timestamp smaller than the smallest global timestamp can be cleaned up
- Discussion
  - not robust ➡ slow or malfunctioning nodes are a problem
  - Clock synchronization necessary

Approach of Hughes (cont.)



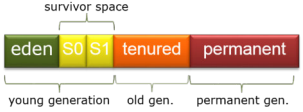






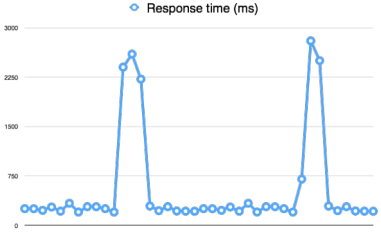
11.4.4 Discussion

Case example, Java



Bildquelle: Wikipedia

- Java uses different approaches:
  - **Intern**: generational tracking algorithm
  - **RMI** (*remote message invokation* ➡ Java's RPC-Variant) a distributed reference counting algorithm
- **Problem in the distributed case**:
  - Undecidable if a reference holding object is alive or crashed / messages have been lost
  - Solution: decrement the reference counter after a fixed time ➡ danger of incorrect clean up
  - Alternative (standard case): global GC from time to time (1h from version 6 on, 1min before) ➡ performance problem



Bildquelle: plumbr.io

11.4.4 Discussion (cont.)

- There is not perfect approach for distributed memory clean up
- Many other approaches, inclusive hybrid algorithms
- Practical consequence: in performance critical applications one should not use distributet references (e.g. Java-RMI)
- Current field of research ➡ potential for new solutions

Literature

[CDK05] Coulouris, G., J. Dollimore and T. Kindberg: *Distributed Systems: Concepts and Design*. Prentice Hall, 2005, Kapitel 18

[StNu04] Steinke, R. C. and G. J. Nutt: "A Unified Theory of Shared Memory Consistency". *Journal of the ACM*, 51(5)2004, 800–849

[CaGe93] Carriero, N. and D. Gelernter: *Linda and Message Passing: What have we learned?*, technical report, Yale University, Department of Computer Science

[ACG86] Ahuja, S., N. Carriero and D. Gelernter: "Linda and Friends". *IEEE Computer*, 19(8)1986, 26–34

[TaSt02] Tanenbaum, A. S. and M. van Steen: *Distributed Systems*. Prentice Hall, 2002 , Abschnitt 4.3

[JoLi96] Jones, R. and R. D. Lins: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. WileyJohn Wiley & Sons, 1996