Nicht-blockierende Synchronisation

Seminar "Effiziente Programmierung"

Joshua Krüger

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2018-12-13

Gliederung (Agenda)

Nachteile von blockierender Synchronsation:

- Ein wartender Thread ist ein nutzloser Thread
- Wenn alle Threads aufeinander Warten, ist die Abfolge sequentiell
- Verlust von Parallelität
  (häufig gewollter Verlust, aber ist er immer notwendig?)

Gefahren von blockieren mit Locks:

- Deadlocks
- Suspension oder Interrupt des 'Lockenden' Threads blockiert alle anderen

Kritischer Abschnitt

```
1 void safe_replace(array, old_element, new_element) {
2   \\lock
3     int old_element_index =
          ↪ find_index(array, old_element);
4     if(old_element_index == DOES_NOT_EXIST)
5         add(new_element);
6     else
7         array[old_element_index] = new_element;
8   \\unlock
9 }
```

Listing 1: List - Safe Replace Operation (Pseudo Java)

Deadlock

http://mcs109.bu.edu/site/files/deadlock/citydeadlock.jpg

Schlafende Threads

```
1  Lock lock1 = new Lock();
2
3  thread-1: requests lock1
4  thread-1: gets lock1
5  thread-1: look at some array
6  thread-2: requests lock1
7  thread-1: sleep(1000 * 60 * 10) //in ms
8  thread-2:
9  thread-1:
10 thread-1:
11 thread-2:
```

Listing 2: Sleeping threads (Pseudo Code)

Schlafende Threads

```
1  Lock lock1 = new Lock();
2
3  thread-1: requests lock1
4  thread-1: gets lock1
5  thread-1: look at some array
6  thread-2: requests lock1
7  thread-1: sleep(1000 * 60 * 10) //in ms
8  thread-2: waits for 10 minutes..
9  thread-1:
10 thread-1:
11 thread-2:
```

Listing 3: Sleeping threads (Pseudo Code)

Schlafende Threads

```
1  Lock lock1 = new Lock();
2
3  thread-1: requests lock1
4  thread-1: gets lock1
5  thread-1: look at some array
6  thread-2: requests lock1
7  thread-1: sleep(1000 * 60 * 10) //in ms
8  thread-2: waits for 10 minutes..
9  thread-1: look at some other array
10 thread-1: releases lock1
11 thread-2: gets lock1
```

Listing 4: Sleeping threads (Pseudo Code)

Interrupted Threads

```
1  Lock lock1 = new Lock();
2
3  thread1: requests lock1
4  thread1: gets lock1
5  thread1: looks at some array
6  thread2: requests lock1
7
8  os|mainThread interrupts thread1
9
10 thread2:
```

Listing 5: Interrupted threads (Pseudo Code)

Durchaus möglich z.B. bei Verwendung von Java's 'Thread.stop'

Interrupted Threads

```
1  Lock lock1 = new Lock();
2
3  thread1: requests lock1
4  thread1: gets lock1
5  thread1: looks at some array
6  thread2: requests lock1
7
8  os|mainThread interrupts thread1
9
10 thread2: waits forever
```

Listing 6: Interrupted threads (Pseudo Code)

Durchaus möglich z.B. bei Verwendung von Java's 'Thread.stop'

Lock Freedom

Der gesamte Prozess(alle Threads zusammen betrachtet) macht garantiert Fortschritt.

Lock Freedom

Der gesamte Prozess(alle Threads zusammen betrachtet) macht garantiert Fortschritt.

1. Keine Deadlocks

Lock Freedom

Der gesamte Prozess(alle Threads zusammen betrachtet) macht garantiert Fortschritt.

1. Keine Deadlocks
2. Interrupt fähig

Wait Freedom

Lock Free

+

Jeder Thread endet nach einer endlichen Anzahl an Operationen, egal was andere Threads tun.

1. Keine Deadlocks
2. Interrupt fähig
3. Threads blockieren sich untereinander nicht

Problem: Sehr schwer zu implementieren, zum Teil unmöglich zu implementieren.

Wait Freedom - Example

```
1 void increment_atomic_reference_counter() {
2     counter.atomic_increment();
3 }
```

Listing 7: Atomic Reference Counter (Pseudo Java)

WaitFree, weil es auf die CPU-Instruction 'Fetch-And-Add' abbildet. Diese holt einen Wert aus dem Speicher, inkrementiert ihn und schreibt ihn zurück. Atomar und geschützt gegen Interrupts.

No communication

```
1  for (int i=0;i<10;i++) {
2      final int i_copied = i;
3      new Thread(() -> {
4          String result="";
5          for (long = 0; i < Long.MAX_VALUE; i++) {
6              result+=i_copied;
7          }
8          writeToFile(result, i_copied+".txt");
9      }).start();
10 }
```

Listing 8: Calculating Something Long (Pseudo Java)

No communication

```
1  for (int i=0;i<10;i++) {
2      final int i_copied = i;
3      new Thread(() -> {
4          String result="";
5          for (long = 0; i < Long.MAX_VALUE; i++) {
6              result+=i_copied;
7          }
8          writeToFile(result, i_copied+".txt");
9      }).start();
10 }
```

Listing 9: Calculating Something Long (Pseudo Java)

Wait Free

No communication

```
1  for (int i=0;i<10;i++) {
2      final int i_copied = i;
3      new Thread(() -> {
4          String result="";
5          for (long = 0; i < Long.MAX_VALUE; i++) {
6              result+=i_copied;
7          }
8          writeToFile(result, i_copied+".txt");
9      }).start();
10 }
```

Listing 10: Calculating Something Long (Pseudo Java)

Wait Free

OutOfMemoryError

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 11: A simple Echo Server (Pseudo Java)

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 12: A simple Echo Server (Pseudo Java)

- So funktioniert 'ping'

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 13: A simple Echo Server (Pseudo Java)

- So funktioniert 'ping'
- So funktioniert der Echo Service auf Port 7

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 14: A simple Echo Server (Pseudo Java)

- So funktioniert 'ping'
- So funktioniert der Echo Service auf Port 7
- Blockiernde Synchronisation

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 15: A simple Echo Server (Pseudo Java)

- So funktioniert 'ping'
- So funktioniert der Echo Service auf Port 7
- Blockiernde Synchronisation
- Wait Free

Linked Stack

```
1  Node<E> head = null;
2
3  void push(E e) {
4      head = new Node<>(e, head);
5  }
6
7  E pop() {
8      if(head==null) {
9          return null;
10     } else {
11         E val = head.val;
12         head = head.next;
13         return val;
14     }
15 }
```

Listing 16: LinkedStack (Java)

Deadlock free, but not lock-free Stack

```
1  Node<E> head = null;
2
3  void push(E e) {
4      //lock
5      head = new Node<>(e, head);
6      //unlock
7  }
8  E pop() {
9      //lock
10     if(head==null) {
11         return null;
12     } else {
13         E val = head.val;
14         head = head.next;
15         return val;
16     }
17     //unlock
18 }
```

Lock Free Stack

```
1  AtomicReference<Node<E>> head = new
       ↪ AtomicReference<>(null);
2  void push(E e) {
3      Node<E> oldHead;  Node<E> newHead;
4      do {
5          oldHead = head.get();
6          newHead = new Node<>(e, oldHead);
7      } while(!head.compareAndSet(oldHead, newHead));
8  }
9  E pop() {
10     Node<E> oldHead;  Node<E> newHead;
11     do {
12         oldHead = head.get();
13         if(oldHead==null)    return null;
14         else                 newHead = oldHead.next;
15     } while(! head.compareAndSet(oldHead, newHead));
16     return oldHead.val;
17 }
```

Exkurs - Compare and Swap:

```
1 \\atomic
2 boolean compare_and_set(int* pointer_to_val,
      ↪ int expected_val, int new_val) {
3     int current_val = *pointer_to_val;  //obtain value
4     if(current_val == expected_value) { //compare value
5         *pointer_to_val = new_val;      //set value
6         return true;
7     }
8     return false;
9 }
```

Listing 19: Compare and Swap (CAS - Pseudo Java)

Meist als Compiler bzw. CPU-Instruktion implementiert.
In Java eine 'native' Function (abgebildet auf C Code).

Lock Free Stack

```
1  AtomicReference<Node<E>> head = new
       ↪ AtomicReference<>(null);
2  void push(E e) {
3      Node<E> oldHead;  Node<E> newHead;
4      do {
5          oldHead = head.get();
6          newHead = new Node<>(e, oldHead);
7      } while(!head.compareAndSet(oldHead, newHead));
8  }
9  E pop() {
10     Node<E> oldHead;  Node<E> newHead;
11     do {
12         oldHead = head.get();
13         if(oldHead==null)    return null;
14         else                 newHead = oldHead.next;
15     } while(! head.compareAndSet(oldHead, newHead));
16     return oldHead.val;
17 }
```

ABA - Problem

Mögliche, problematische Abfolge:

1. Original Stack: A->null
2. Thread-2: peek() => returns A
3. Thread-1: push(B) => Stack: B->A->null
4. Thread-3: pop() => Stack: A->null
5. Thread-2: peek() => returns A

Thread-2 kann nun (fälschlicherweise) denken, dass sich nichts verändert hat.
(Bei unserem Stack kein Problem.
Es wäre allerdings ein Problem wenn compareAndSwap den Inhalt der Node's für den Vergleich zugrunde legen würde.)

Stack Performance - Single Thread

1,000,000 iterations of pop, push and peek

Test-Name	Successes	Average Time
one - notLocked	26	8.293s
one - locked	26	9.055s
one - synchronized	26	8.599s
one - lockfree	26	8.629s

Stack Performance - Single Writer/Many Readers

1000 writer 1000^2 push/pop - AND - 1000 reader 1000x peek

Test-Name	Successes	Average Time
oneMany - locked	47	2.718s
oneMany - lockless	47	2.680s
oneMany - notLocked	35	2.675s
oneMany - synchronized	47	2.663s

Stack Performance - Many Writers/Many Readers

1000 thread 1000x push/pop/peek

Test-Name	Successes	Average Time
manyMany - locked	41	13.156s
manyMany - lockless	41	12.923s
manyMany - notLocked	15	13.026s
manyMany - synchronized	41	12.839s

Stack Performance - Many Writers/Many Readers + Suspension

1000 thread 1000x push/pop/peek

+

Every 100th Thread is suspended for 500ms within push/pop/peek

Stack Performance - Many Writers/Many Readers + Suspension

1000 thread 1000x push/pop/peek

+

Every 100th Thread is suspended for 500ms within push/pop/peek

Test-Name	Successes	Average Time
manyMany(sleep) - locked	47	15.181s
manyMany(sleep) - lockless	47	10.815s
manyMany(sleep) - notLocked	32	1.821s
manyMany(sleep) - synchronized	47	15.146s

Copy On Write

Wenn geschrieben wird, wird kopiert, verändert und atomar gesetzt.

Pros	Cons
■ Reader/getter müssen nicht gelockt werden. ■ Überhaupt kein locking, sofern nur ein Writer.	■ Mindestens verdoppelte Space-Complexity. ■ Mehrere Writer: Die Writer/Setter müssen weiterhin geblockt werden

Tabelle: Pros and cons of Copy On Write

Replace in Locked (Array-)List

```
1 void safe_replace(array, old_element, new_element) {
2   \\lock (entire list)
3     int old_element_index =
          ↪ find_index(array, old_element);
4     if(old_element_index == DOES_NOT_EXIST)
5         throw Error
6     array[old_element_index] = new_element;
7   \\unlock (entire list)
8 }
```

Listing 21: List - Safe Replace Operation (Pseudo Java)

Problem: Reader müssen gelockt werden (warum?)

Warum Reader gelockt werden müssen:

```
1  int find_index(array, element) {
2      for(int i=0; i < array.length; i++) {
3          //wenn der Thread hier unterbrochen wird,
               ↪ gibt es eine IndexOutOfBoundsException
               ↪ in der naechsten Zeile
4          if(array[i] == element) {
5              return i;
6          }
7      }
8      return DOES_NOT_EXIST;
9  }
```

Listing 22: List - Safe Replace Operation (Pseudo Java)

Motivation

Replace in Copy On Write List

```
1 void replace(array, old_element, new_element) {
2   //lock (writers only)
3   Array array_new = array_old.clone();
4
5   int old_element_index =
        ↪ find_index(array_new, old_element);
6   array_new[old_element_index] = new_element;
7   //unlock (writers only)
8 }
```

Listing 23: List - Safe Replace Operation (Pseudo Java)

Vorteil: Reader müssen nicht gelockt werden. Nachteil: Schreiben ist langsam, speziell für viele Daten

Replace in Copy On Write List

```
1  void replace(array, old_element, new_element) {
2    //'array' vom Java-Typ AtomicReference
3    Array array_old;
4    Array array_new;
5    do {
6      array_old = array.load();
7      array_new = array_old.clone();
8
9      int old_element_index =
           ↪ find_index(array_new, old_element);
10     array_new[old_element_index] = new_element;
11
12   } while(! array.cas(array_old, array_new));
13 }
```

Listing 24: List - Lock-Free Replace (Pseudo Java)

Non-Blocking IO - Überblick

1. Blocking IO
2. Futures
3. Non-Blocking + Asynchronous Server

Ein einfacher Echo Server

```
1 while(true) {
2     Socket incoming = server.acceptConnection();
3     new Thread(() -> {
4         while(incoming.isOpen()) {
5             String message = incoming.readMessage();
6             incoming.write(message);
7         }
8     }).start();
9 }
```

Listing 25: A simple Echo Server (Pseudo Java)

- So funktioniert 'ping'
- So funktioniert der Echo Service auf Port 7
- Blockiernde Synchronisation
- Wait Free

Futures

```
1  var incoming = Socket.connect("localhost", 12345);
2  var future = incoming.read_async();
3
4  //1. Weg - Blockierend & Synchron
5  var message = future.get();
6  handleMessage(message);
7
8  //2. Weg - Nicht Blockierend
9  while(!future.isDone()) {
10     doSomeWork();
11 }
12 var message = future.get();
13 handleMessage(message);
```

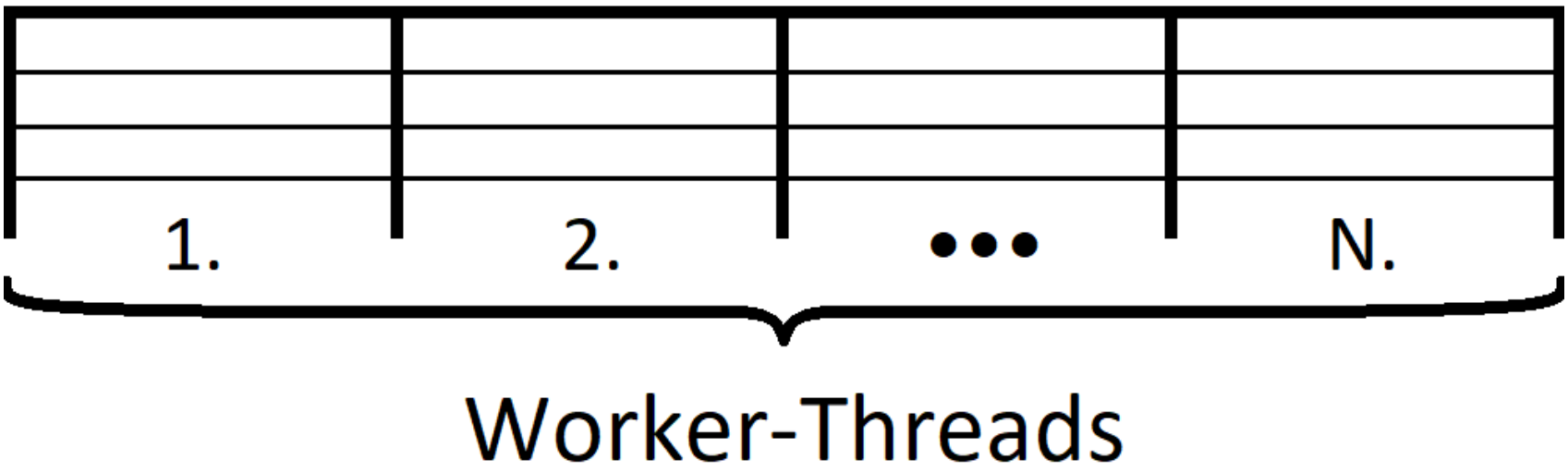
Listing 26: List - Examples of using Futures (Pseudo Code)

Ein einfacher Echo Server - (mit Futures)

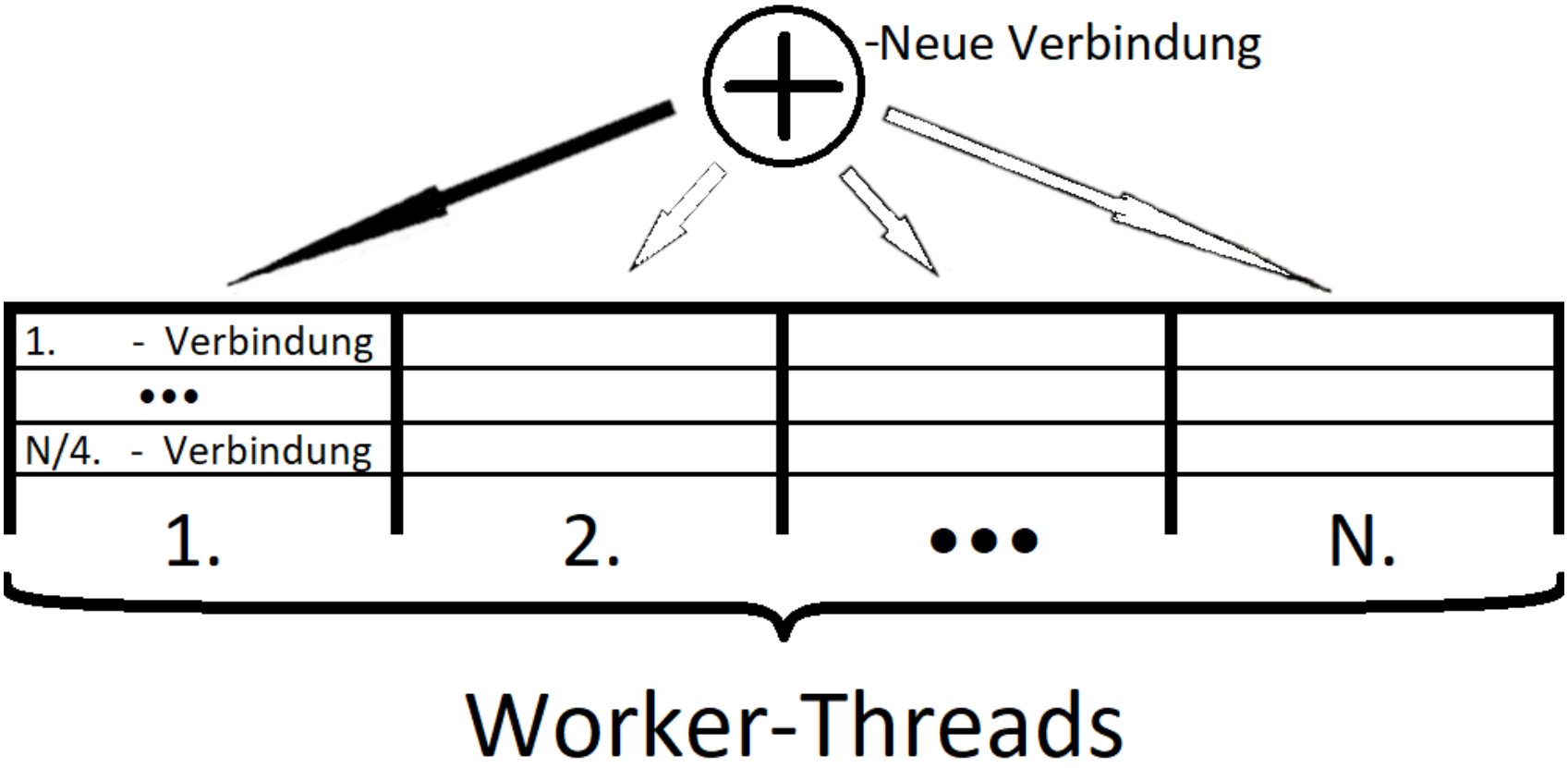
```
1  while(true) {
2      AsyncSocket incoming =
           ↪ server.acceptConnection();
3      new Thread(() -> {
4          while(incoming.isOpen()) {
5              Future<String> messageFuture =
                   ↪ incoming.readMessage();
6              var message = messageFuture.get();
7              Future writeCompleteFuture =
                   ↪ incoming.write(message);
8              writeCompleteFuture.wait();
9          }
10     }).start();
11 }
```

Listing 27: A simple Echo Server - (with futures) (Pseudo Java)

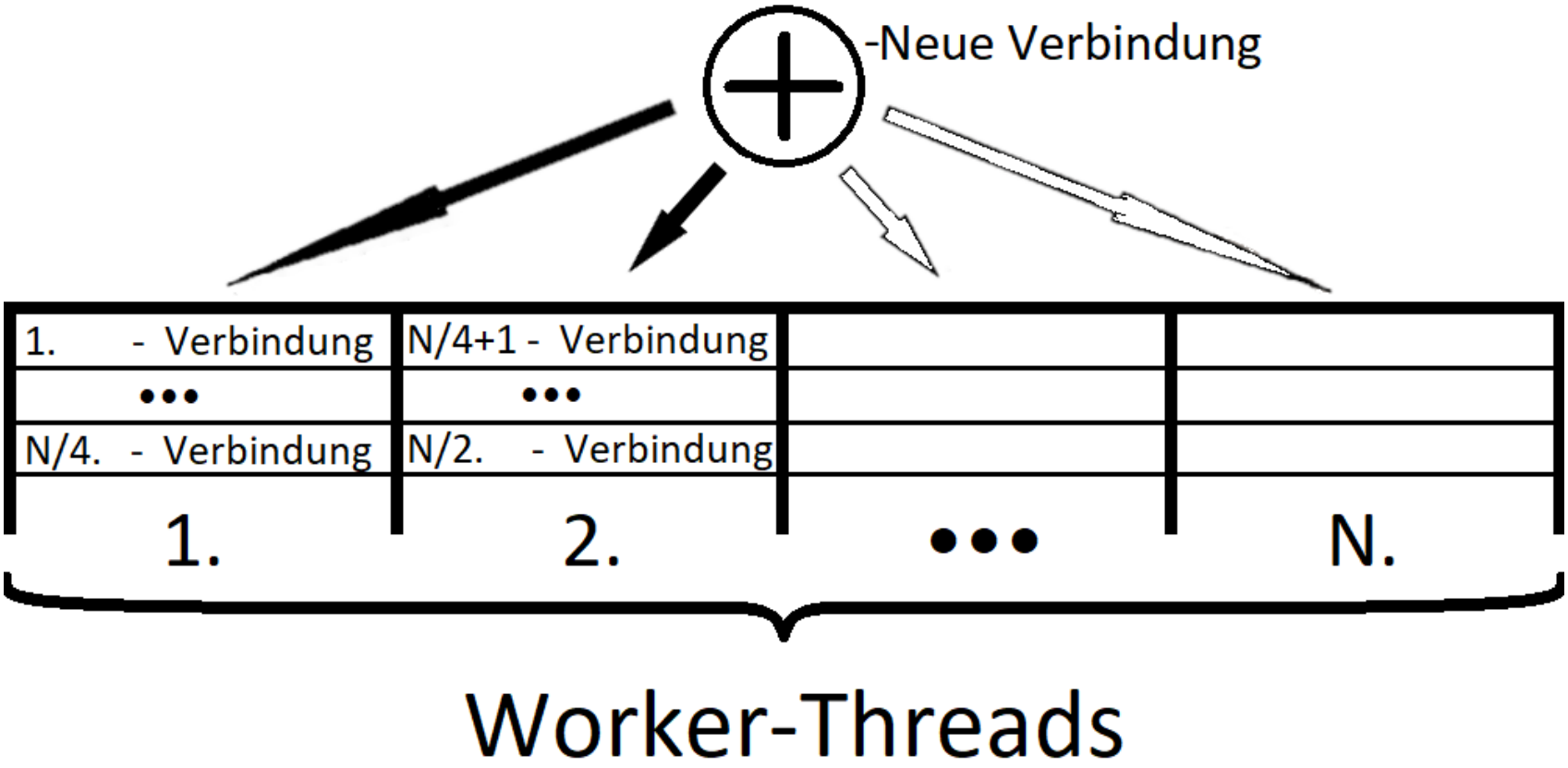
Scalable Echo Server



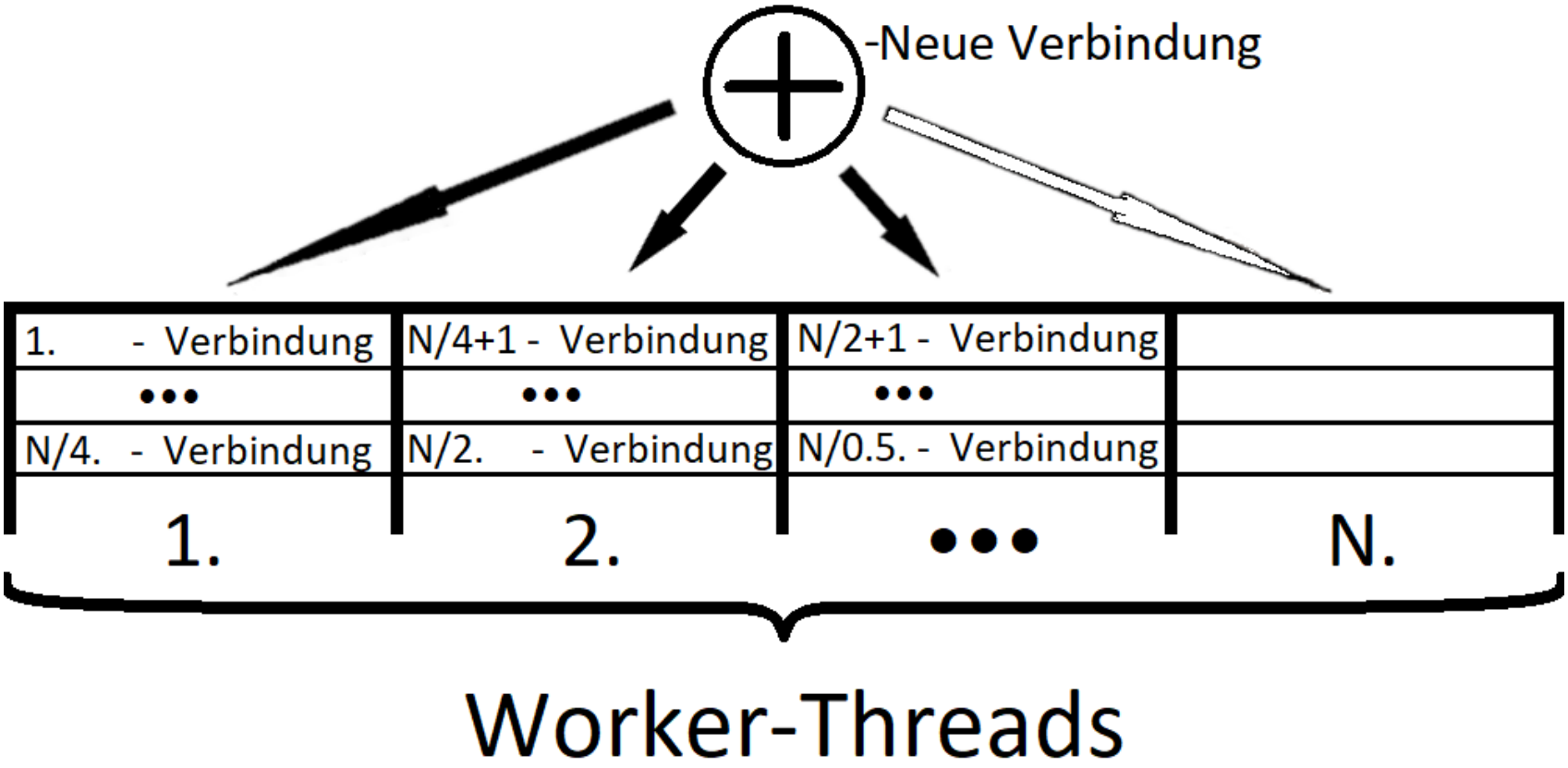
Scalable Echo Server



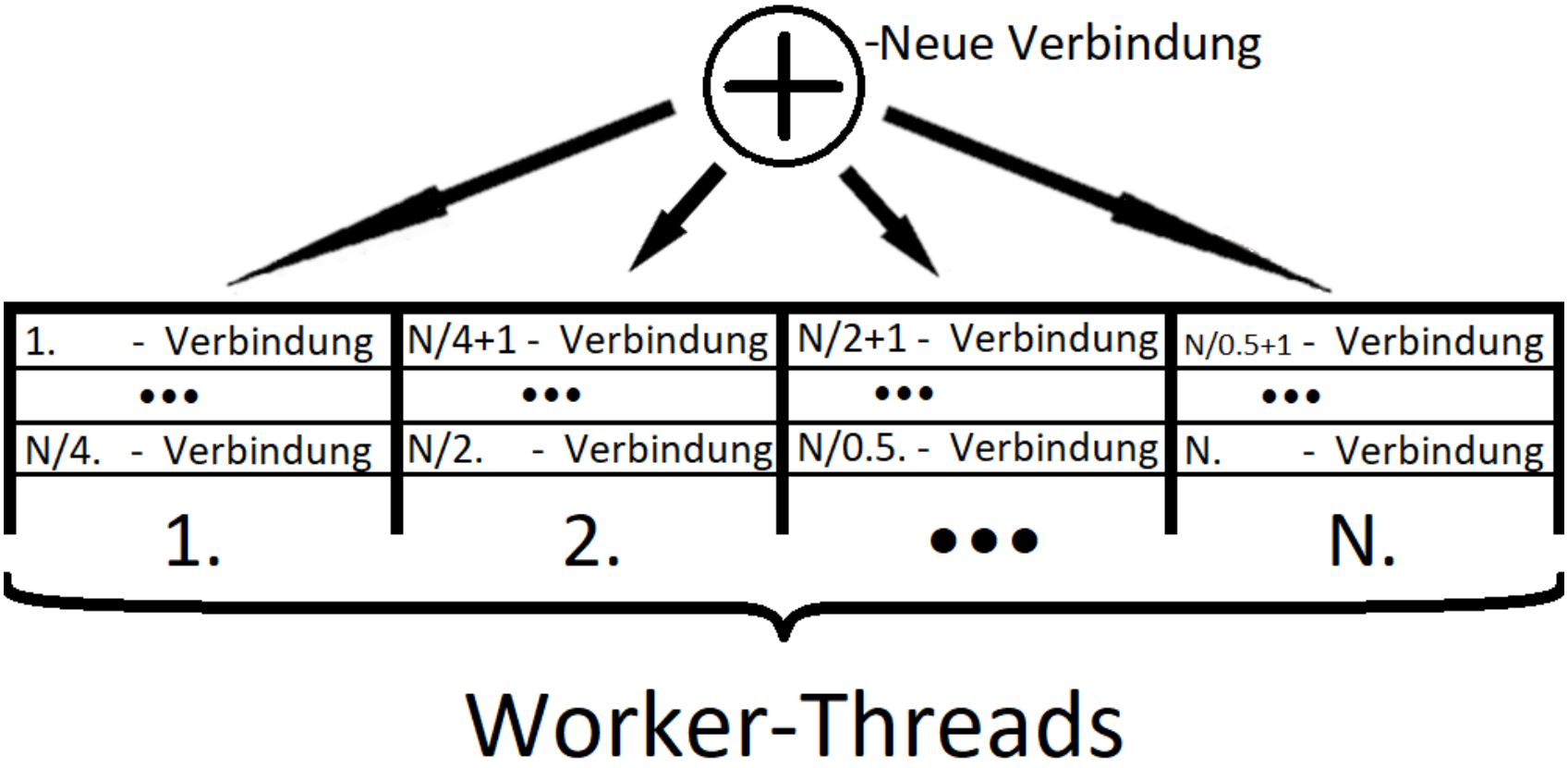
Scalable Echo Server



Scalable Echo Server



Scalable Echo Server



Echo Worker Thread

```
1  new Thread(() -> {
2      while(true) {
3          for(AsyncSocket socket:sockets) {
4              if(socket.readFuture.isDone()) {
5                  var message = socket.readFuture.get();
6                  Future writeCompleteFuture =
                       ↪ socket.write(message);
7                  socket.writeFuture = writeCompleteFuture;
8              } else if(socket.writeFuture.isDone()) {
9                  Future messageFuture =
                       ↪ socket.readMessage();
10                 socket.readFuture = messageFuture;
11             } else if(socket.isClosed()) {
12                 sockets.remove(socket);
13             }
14         }
15     }
16 }).start();
```

Threads vs Async

Threads haben Overhead:

1. Jeder(auch wartende) Thread braucht Speicher(Stack, etc.)
2. Das OS muss Threads schedulen (Kontextwechsel)
3. Begrenzt

Async hat auch Overhead:

1. Wir müssen selber schedulen (kann besser auf die Situation abgestimmt passieren).
2. Kein eigener Stack, potentiell langsamer Heap Zugriff

Empfehlung:

- Immer IO
- **Außer:**
  1. Arbeiten beim Warten
  2. VIELE Verbindungen

Test Results:

NBIO gegen faule Affen

https://www.pinterest.com/pin/437482551295127796

NBIO gegen faule Affen

Slow Loris Attack - DOS-Angriff auf Thread-per-Connection Server-Modell

Server:

1. Für jede neue Verbindung, erstelle einen Thread und warte auf Nachrichten.

[1] z.B. durch das Windows Thread-limit

NBIO gegen faule Affen

Slow Loris Attack - DOS-Angriff auf Thread-per-Connection Server-Modell

Server:

1. Für jede neue Verbindung, erstelle einen Thread und warte auf Nachrichten.

Client:

1. Baue viele Verbindungen zu einem Server auf.
2. Sende auf jeder Verbindung alle x Sekunden 1 byte (timeouts).

Da die Ressource 'Threads' begrenzt ist[1], können irgendwann keine neuen Threads mehr aufgebaut werden und neue Verbindungen müssen abgelehnt werden.

[1] z.B. durch das Windows Thread-limit

Go's way

1. Golang hat sogenannte Goroutinen.
2. Im Prinzip leichtgewichtige Threads(dynamic stack, 'besserer scheduler', 1 Thread pro CPU-Kern).
3. IO Operationen in GO sind asynchron(sofern möglich).
4. **aber:** GO versteckt diese Komplexität gegenüber dem Programmierer.
5. => Schöner Code und schöne Scalability.

Zusammenfassung

- Lock Freedom
  - schwierig
  - Per Definition Deadlock free
    Was dem Programmier der Algorithmen nicht hilft
  - Per Definition Interrupt fähig
  - Je mehr Low Level, desto sinnvoller
  - Besser + Schwerer: Wait-Free Programming
- NBIO
  - Nicht immer den Aufwand wert
  - Nützlich bei begrenzter Rechenkapazität oder paralleler Kommunikation
  - Mitigiert eine Art von DOS Attacke.

Literatur und Quellen I

- https://docs.microsoft.com/en-us/windows/desktop/DxTechArts/lockless-programming
- http://developer.classpath.org/doc/java/util/concurrent/CopyOnWriteArrayList-source.html
- www.intel.com/content/dam/www/public/us/en/documents/guides/intel-dpdk-programmers-guide.pdf
- http://www.cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf
- http://www.1024cores.net/home/lock-free-algorithms/introduction

Literatur und Quellen II

- https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html
- https://www.researchgate.net/publication/221597191_Design_and_Evaluation_of_Nonblocking_Collective_IO_Operations
- http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html