Grundlagen Rechnernetze und Verteilte Systeme

SoSe 2012

Kapitel 6: Verteilte Systeme

Prof. Dr.-Ing. Georg Carle

Stephan M. Günther, M.Sc.
Nadine Herold, M.Sc.
Dipl.-Inf. Stephan-A. Posselt

Fakultät für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste
Technische Universität München

Worum geht es in diesem Kapitel?

1 Motivation

2 Homogene, skalierbare Paradigmen
- MPI
- MapReduce
- Pipes

3 Remote Procedure Call

4 Shared Memory
- NUMA
- Paging
- Sharing auf Objektebene

5 Einbettung in Programmiersprachen

Was ist ein verteiltes System?

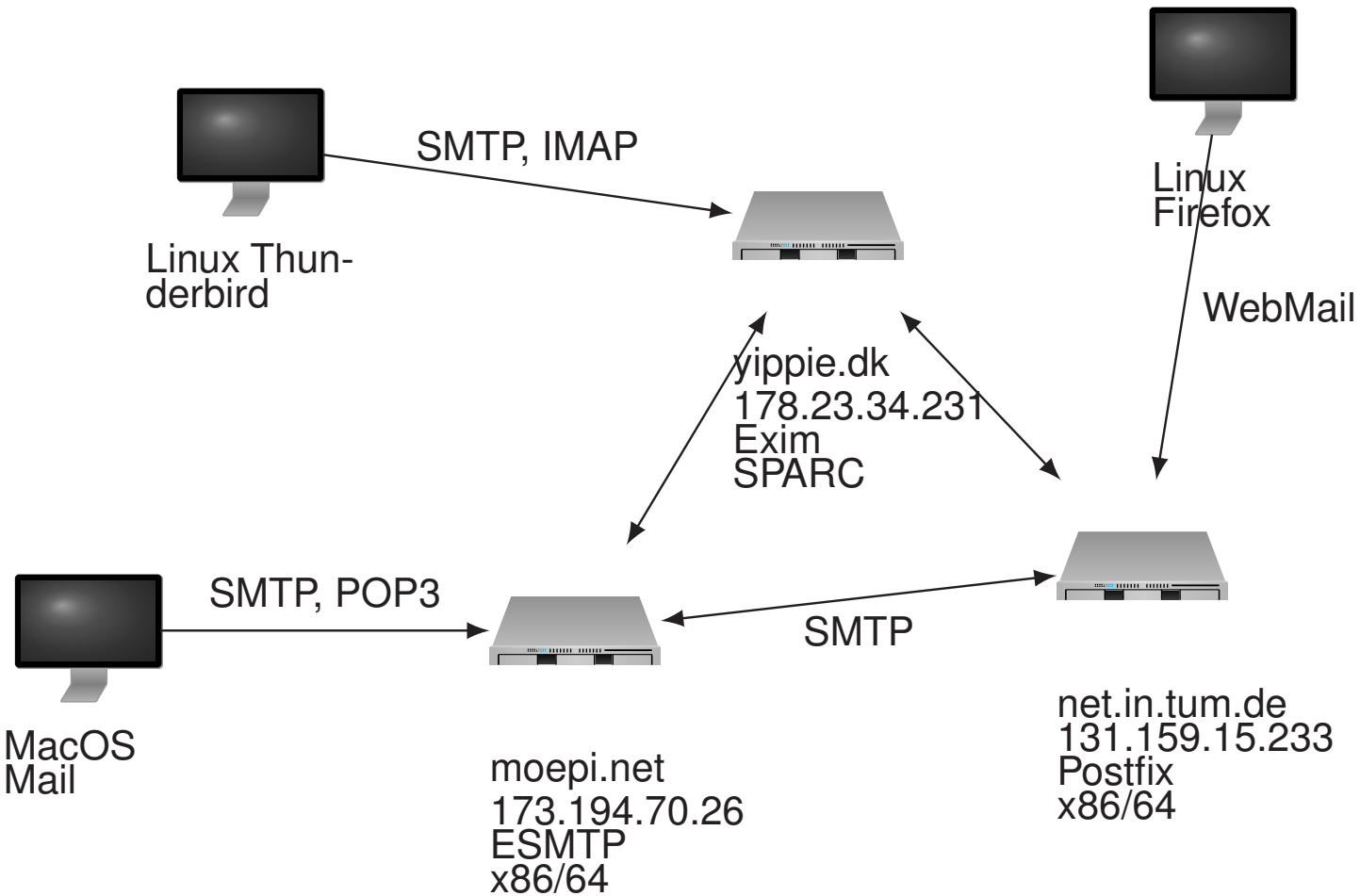
Definition

„Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.“ [TvS07]

Warum verteilte Systeme?

- Skalierbarkeit
  Viele Rechner stellen im Verbund mehr Ressourcen zur Verfügung.
- Ausfallsicherheit
  Der Dienst als Ganzes kann auch noch angeboten werden, wenn ein einzelner Rechner ausfällt.
- Heterogenität
  Rechner mit verschiedener Hardware oder unterschiedlichen Eigentümern können einen gemeinsamen Dienst anbieten.

Beispiel: E-Mail



Beispiel: E-Mail

- verteilt
  - mehrere Hosts nehmen teil
  - Server und Clients können unterschiedlichen Parteien gehören
- heterogen
  - Netzwerkanbindung
    z. B. Schicht 2: Ethernet, WLAN, ...
    z. B. Schicht 3: IP, IPX, ATM, ...
  - Hardware
    z. B. Architektur: x86, x64, ARM, SPARC, ...
    z. B. Endianness
  - Software
    z. B. Betriebssystem: UNIX, Windows, MacOS, ...
    z. B. Mail Transfer Agent: Postfix, Exim, MS Exchange Server, ...
  - Protokolle
    z. B.: SMTP, SMTP+TLS, POP3, IMAP, WebMail, ...
- Kommunikation ist zwischen allen Teilnehmern dank standardisierter Protokolle trotzdem möglich.

Fokus

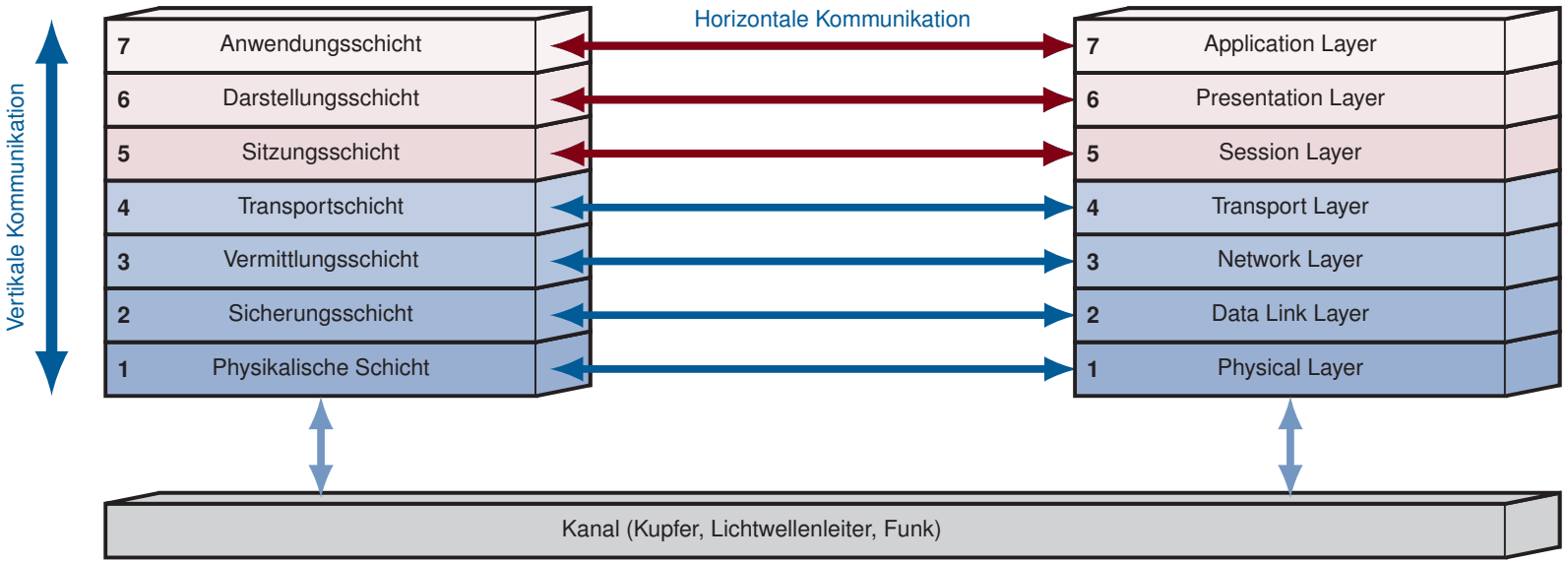
Es gibt Zahlreiche Problemfelder im Bereich der Verteilten Systeme, zum Beispiel:

- Entwurf von Kommunikationsprotokollen (vgl. Kapitel 5)
- Abstraktion vom Basissystem durch sog. “Middleware”
  - Kommunikation
  - Persistenter Datenspeicher
- Migration/Mobilität von Systemkomponenten
- Fehlertoleranz
- Skalierbarkeit

Schwerpunkt ist in dieser Vorlesung:

- Probleme Verteilter Systeme verstehen und klassifizieren zu können
- Methoden zur Implementierung Verteilter Systeme am Beispiel bestehender Lösungen zu vermitteln.

Einordnung im ISO/OSI-Modell



Worum geht es in diesem Kapitel?

1 Motivation

2 Homogene, skalierbare Paradigmen
- MPI
- MapReduce
- Pipes

3 Remote Procedure Call

4 Shared Memory
- NUMA
- Paging
- Sharing auf Objektebene

5 Einbettung in Programmiersprachen

Problemstellung

Szenario: Ein Super-Computer besteht aus 10 000 Knoten, die alle die gleiche Hardware haben. Alle sind in einem regelmäßigen Muster in einem gemeinsamen Netzwerk angeordnet.

Problem: Wie kann man regulärer Datenstrukturen wie Matrizen effizient verarbeiten?

Idee: Man kann einfach die Datenstruktur auf die Topologie des Rechnernetzes abbilden.

Message Passing Interface (MPI)

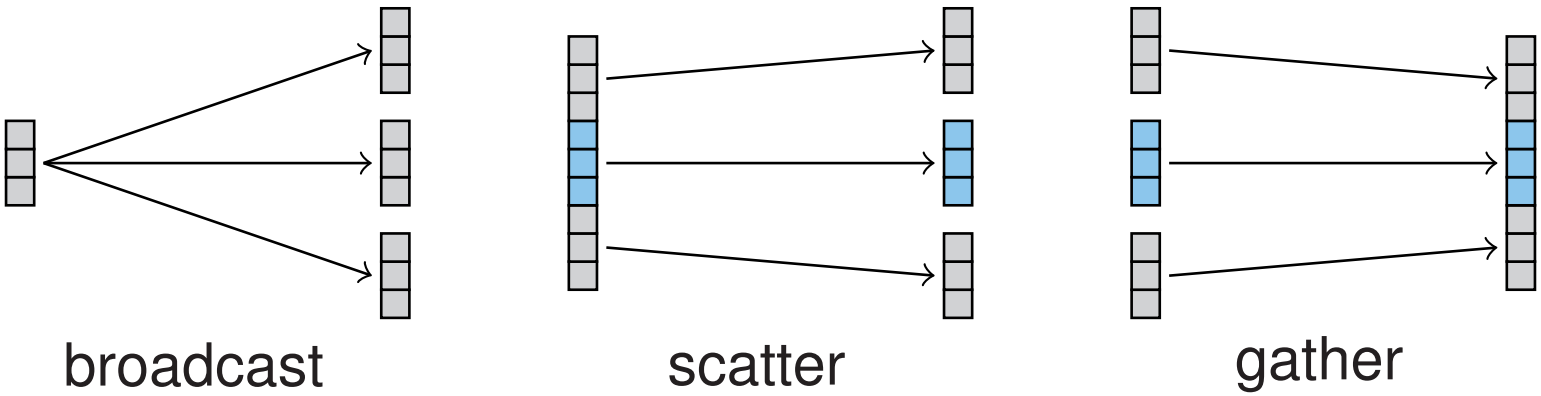
- Ist praktisch Standard auf Super-Computern und Compute Clustern.
- MPI hat ein standardisiertes Interface, was es erlaubt, Anwendungen auf unterschiedliche Implementierungen zu portieren.
- Die bekanntesten, freien Implementierungen sind:
  - Open MPI (`http://www.open-mpi.org/`)
  - MPICH2 (`http://www.mcs.anl.gov/research/projects/mpich2/`)
- Üblicherweise wird MPI mit C oder Fortran verwendet.
- MPI bietet verschiedene Abstraktionen von der reinen Socket-Programmierung:
  - Nachrichtenorientierte Kommunikation – MPI übernimmt den Aufbau von (TCP-)Verbindungen. Nachrichten können eine (fast) beliebige Länge haben.
  - MPI bietet Primitive für häufige Kommunikationsmuster wie z. B. “1 zu N” (`MPI_BCast`).
  - Man kann MPI Knoten in virtuellen k-dimensionalen kartesischen Räumen oder Tori anordnen lassen. Auch hierfür werden Primitive für die einfache Kommunikation angeboten.

Code-Beispiel: MPI

- Alle N MPI-Prozesse werden von 0 bis $N-1$ durchnummeriert.
- Folgendes Code-Beispiel zeigt ein vollständiges MPI-Programm, das jedoch noch keine programmspezifische Kommunikation durchführt.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int rank; // Index dieses Prozesses
5  int size; // Anzahl aller Prozesse
6
7  int main(int argc, char **argv) {
8      MPI_Init(&argc, &argv);
9      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10     MPI_Comm_size(MPI_COMM_WORLD, &size);
11     printf("I␣am␣%d␣of␣%d.\n", rank, size);
12     MPI_Finalize();
13 }
```

Beispiele für Kommunikationsmuster



Code-Beispiel: MPI

Deklaration

```
1  int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int
       root, MPI_Comm comm);
```

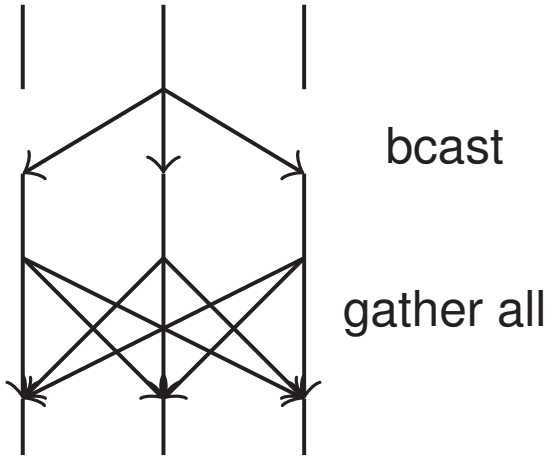
Beispiel-Aufruf

```
1  MPI_Bcast(in, inCount, MPI_INT, MASTER, MPI_COMM_WORLD);
```

- `MPI_Bcast` kopiert `count` Bytes aus dem `buffer` des Prozesses mit dem Index `root` zu allen anderen.

- datatype: MPI hat eigene Datentypen $\rightarrow$ Plattformunabhängigkeit!
- root: der Index des Prozesses, der die Daten verteilt
- comm: Rechner können in Kommunikationsdomänen unterteilt werden alle Rechner befinden sich automatisch in `MPI_COMM_WORLD`

Kosten



- In jedem Kommunikationsschritt muss auf den langsamsten Rechner gewartet werden.
- Die gesamten Kommunikationskosten sind die Summe der Kosten der einzelnen Kommunikationsschritte.
- Die Gesamtkosten können sich erheblich verteuern, wenn die Latenz im Netz zu groß ist.

Probleme

- Kommunikationsaktionen überlappen sich nicht, sondern blockieren das Programm $\rightarrow$ Performance-Verlust:
  - Amdahls Gesetz:
$$\text{Speedup } s = \frac{1}{(1-p) + p/n}$$
  - bei $n = 100$ Rechnern und einem parallelen Anteil von $p = 90\%$ beträgt der Speedup $s = 9.2$, also nur 9.2% des Optimums von 100

- Die Topologie der Rechner und die Aufteilung der Daten sollten gut zueinander passen: “Verschnitt” führt zu schlechter Auslastung

- MPI leistet kein Load-Balancing: Ein einzelner Rechner, der mit anderen Tasks ausgelastet ist, bremst alle MPI-Prozesse aus.

- MPI beinhaltet prinzipbedingt nur sehr begrenzte Fehlertoleranz. Der Ausfall eines Rechners führt zu:
  - Abbruch des ganzen Programms
  - oder ein anderer Knoten übernimmt die Aufgabe: im schlechtesten Fall bedeutet doppelte Arbeit halbe Performance!

Feature-Tabelle

	MPI
reguläre Datenstrukturen	×
skaliert automatisch	×

Anmerkungen

- Der Programmierer bestimmt explizit, wie die Daten auf die Rechner aufgeteilt werden. Mit dem entsprechenden Vorwissen kann so für reguläre Datenstrukturen der Durchsatz maximiert werden.

MapReduce

- Manche Probleme lassen sich als eine Abbildung (Map) gefolgt von einer Reduktion (Reduce) formulieren. Beispiele:
  - Wörter in Dokumenten zählen
  - Sortieren
  - Index-Invertierung
- MapReduce
  - . . . erwartet vom Programmierer nur die Implementierung der sequentiellen Funktionen Map und Reduce.
  - . . . übernimmt Parallelisierung, Last-Verteilung und Fehlertoleranz.
- Beispiel: Wir wollen für jedes Assignment zählen, wieviele Teams bestanden haben.
  - Map

    Eingabe: die `grades.txt` eines Teams, z.B. `team000/grades.txt`
    Ausgabe: eine Liste aus Paaren der Form: „(Assignment-Nummer, bestanden)“, wobei „bestanden“ 0 oder 1 ist
  - Reduce

    Eingabe: ein Paar der Form: „(Assignment-Nummer, Liste aus bestanden = 0 oder 1)“
    Ausgabe: ein Paar der Form: „(Assignment-Nummer, Summe der Liste)“

Map und Reduce

- Die Funktionen „map“ und „reduce“ stammen aus der funktionalen Programmierung und haben üblicherweise folgende Signaturen:

$$\text{map} : (X_1 \to X_2) \times (X_1)^* \to (X_2)^*$$
$$\text{reduce} : (Y_1 \times Y_1 \to Y_2) \times (Y_1)^* \to (Y_2)^*$$

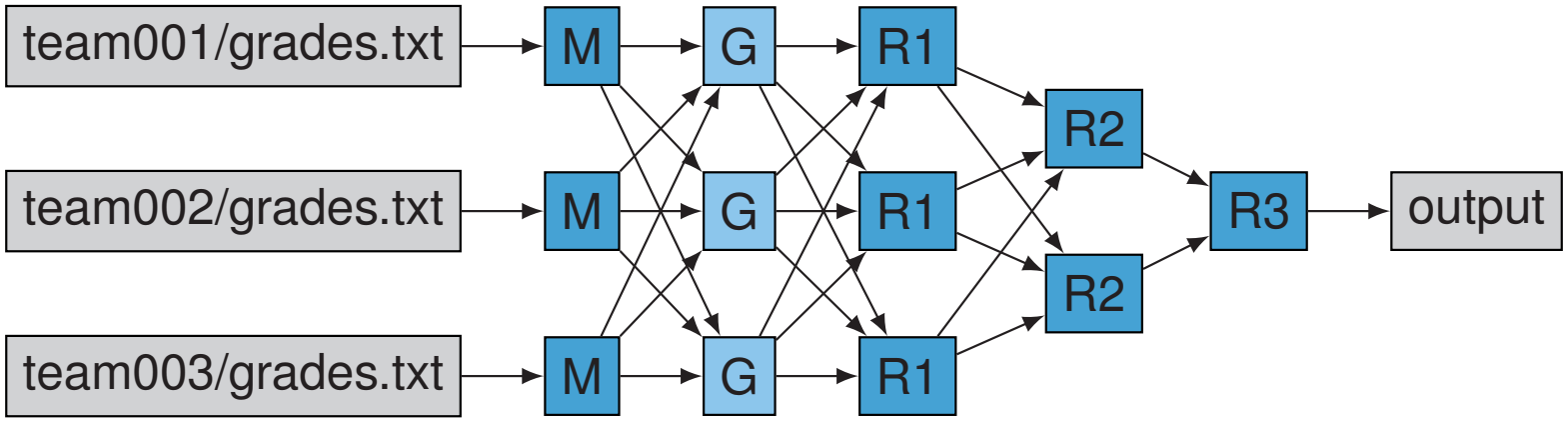
- Bei MapReduce entsprechen die Funktionen “Map” und “Reduce” dem ersten Parameter in obigen Signaturen – allerdings mit spezielleren Typen:

$$\text{Map} : K_1 \times V_1 \to (K_2 \times V_2)^* \qquad \text{wobei } X_1 \mathrel{\widehat{=}} K_1 \times V_1$$
$$\text{Reduce} : K_2 \times (V_2)^* \to (V_2)^* \qquad X_2 \mathrel{\widehat{=}} (K_2 \times V_2)^*$$

- Zwischen der „Map“- und der „Reduce“-Phase gruppiert das Framework die Werte nach Schlüsseln. Diese Funktion wird nicht vom Programmierer implementiert:

$$\text{group} : (K_2 \times V_2)^* \to (K_2 \times (V_2)^*)^*$$

Datenfluss



- Für jede Datei wird eine Map-Operation (M) instanziiert. Diese Aufrufe sind völlig unabhängig voneinander und können parallel ausgeführt werden.
- Die beiden Reduktionsoperationen, Group (G) und Reduce (R), können bereits anlaufen, sobald Map genügend Ausgabe-Paare produziert hat.

  Die Ausführung der Phasen kann sich also überlappen, sodass die Rechner am Anfang nur Map und erst am Schluss nur noch Reduce-Operationen ausführen.
- Ein (fehlbarer) Master erzeugt die Arbeitspakete und verteilt sie erneut, wenn ein Slave abstürzt.

Code-Beispiel: MapReduce

- Auszählen bestandener Assignments

```
1  virtual void Map(const MapInput& input) {
2      const string& grades = input.value();
3      // grades parsen, assignment und passed zuweisen ...
4      Emit(assignment, Emit(IntToString(passed));
5  }
6
7  virtual void Reduce(ReduceInput* input) {
8      int64 sum = 0;
9      while (!input->done()) {
10         sum += StringToInt(input->value());
11         input->NextValue();
12     }
13     Emit(IntToString(sum));
14 }
```

MapReduce

- MapReduce wurde von Google – ursprünglich für den internen Gebrauch – entwickelt [DGI04].
- Apache Hadoop MapReduce ist eine freie Implementierung von MapReduce.

- Nachteile
  - MapReduce ist nur für Probleme geeignet, die sich in genau eine Abbildung und/oder eine Reduktion zerlegen lassen – MapReduce kennt keine anderen Kommunikationsmuster.
  - Der Programmierer hat keinen Einfluss darauf, wie die Tasks im System verteilt werden. Das könnte zu schlechterer Performance führen, wenn der Programmierer dadurch a priori-Wissen über die Struktur des Systems nicht in die Lösung einbringen kann.

Feature-Tabelle

	MPI	MapReduce
reguläre Datenstrukturen	×	
skaliert automatisch	×	×
automatische Verteilung		×
Fehlertoleranz		×

Pipes

Definition (Pipe)

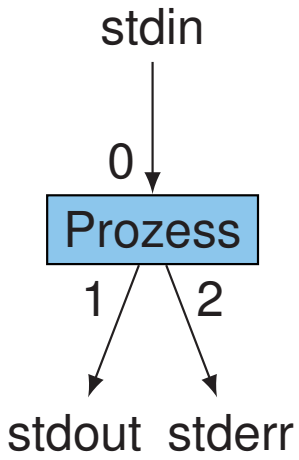
Ein Simplex-Datenstrom, der durch ein File Handle repräsentiert und durch das Betriebssystem gepuffert wird, nennt man Pipe.

- Pipes ermöglichen Inter-Prozess-Kommunikation (IPC).
- Jeder Prozess verfügt automatisch über drei Pipes mit vorgegebenen Nummern (File Handles):

stdin	0, Eingabe, oft Tastatur oder Datei
stdout	1, Ausgabe, normaler Ausgabedatenstrom
stderr	2, Ausgabe, für Fehlermeldungen gedacht

- Zugriff in C:

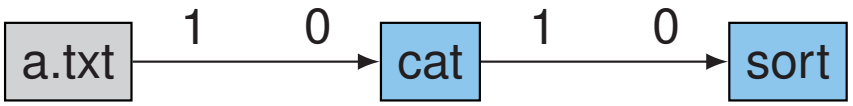
```
1  scanf("%d", &num); // stdin
2  printf("Hello, world!\n"); // stdout
3  fprintf(stderr, "Fehler %d!\n", errno);
```



Pipes umleiten mit netcat

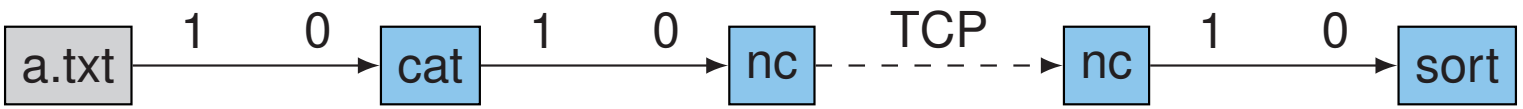
- Beispiel 1: Die Datei `a.txt` soll vom Programm `cat` eingelesen und der Inhalt an `sort` zum Sortieren weitergereicht werden (IPC):

```
host1> cat a.txt | sort
```



- Beispiel 2: Der Dateiinhalt soll auf `host1` gelesen und auf `host2` sortiert werden. Dazu können die Daten zum Beispiel via netcat oder SSH auf einen anderen Rechner übertragen werden:

```
host2> nc -lp 1234 | sort
host1> cat a.txt | nc host2 1234
```



- Problem: Bei einem komplexeren Prozess-Graph müssen zuerst die Pipes angelegt werden und dann auf möglicherweise vielen Rechnern die nötigen Prozesse gestartet werden. Das ist umständlich so.

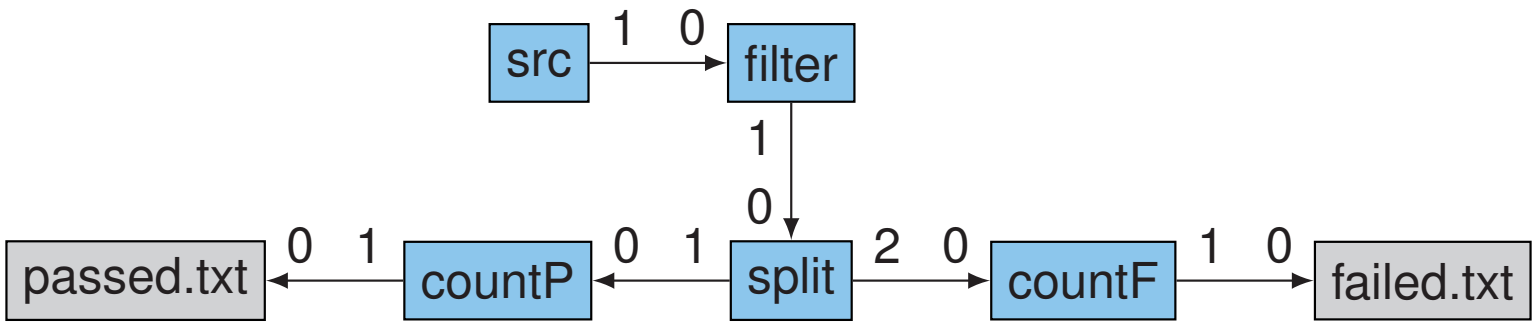
DUP

- Das DUP-System [GGR+10] dient dem einfachen Aufbau eines Graphen aus Prozessen, die über Pipes miteinander verbunden sind.
- Der DUP-Dämon (`dupd`) wird einmal pro teilnehmendem Rechner gestartet und wartet dann auf eingehende TCP-Verbindungen. Er übernimmt folgende Aufgaben:
  - Pipes erstellen
  - Prozesse mit Pipes verbinden und starten
- Der Prozess-Graph wird mit einer einfachen, domänenspezifischen Sprache beschrieben.
- DUP-Syntax (vereinfacht):
  Name `@` Host `[` Umleitungen `]` `$` Shell-Befehl `;`
- Umleitungen beschreiben die Struktur des Graphen als Adjazenzliste; Beispiele für Umleitungen:
  - `0 < foo.txt` – Inhalt der Datei “foo.txt” nach stdin kopieren
  - `1 | foo:0` – stdout zu stdin vom Prozess mit dem Namen “foo” umleiten
  - `2 >> errors.log` – stderr in die Datei “errors.log” umleiten

Code-Beispiel: DUP

- Wieviele Gruppen haben Assignment 1 bestanden und nicht bestanden?

Datenfluss



Quell-Code

```
1    src@192.168.1.1 [ 1>filter:0 ] $ cat team*/grades.txt;
2  filter@192.168.1.2 [ 1>split:0 ] $ grep 'assignment01:';
3   split@192.168.1.3 [ 1>countP:0, 2>countF:0 ] $ mgrep 'PASS';
4  countP@192.168.1.4 [ 1>passed.txt ] $ wc -l;
5  countF@192.168.1.5 [ 1>failed.txt ] $ wc -l;
```

Feature-Tabelle

	MPI	MapReduce	DUP
reguläre Datenstrukturen	×		×
skaliert automatisch	×	×	
automatische Verteilung		×	
Fehlertoleranz		×	
heterogen			×

Anmerkungen

- Der Prozess-Graph in einem DUP-Skript ist statisch und passt sich weder an die Größe des Systems noch an die Datenmenge an.
- DUP ermöglicht es, bestehende Konsolenprogramme in den Prozess-Graphen zu integrieren.

Worum geht es in diesem Kapitel?

1 Motivation

2 Homogene, skalierbare Paradigmen
- MPI
- MapReduce
- Pipes

3 Remote Procedure Call

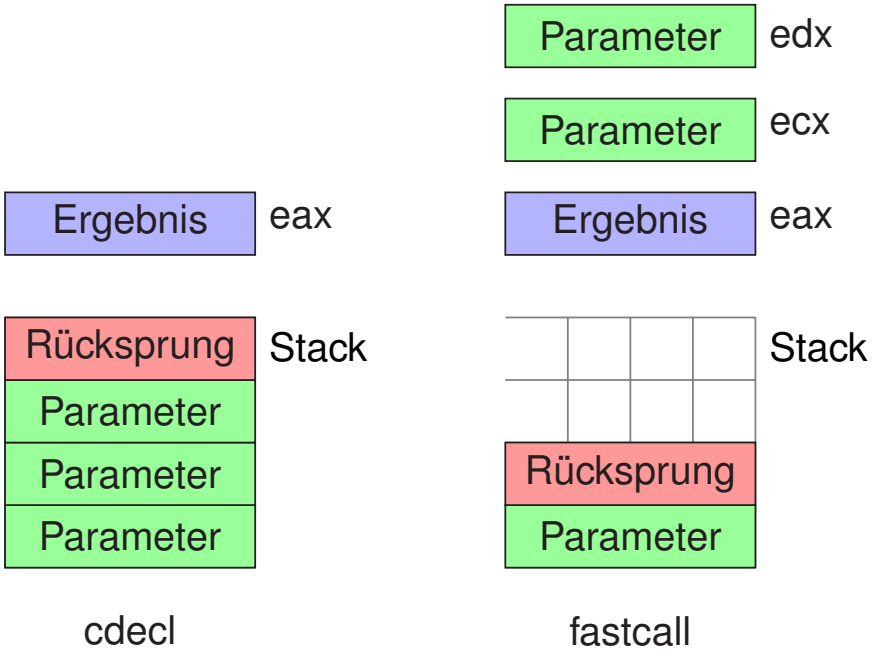
4 Shared Memory
- NUMA
- Paging
- Sharing auf Objektebene

5 Einbettung in Programmiersprachen

Remote Procedure Call

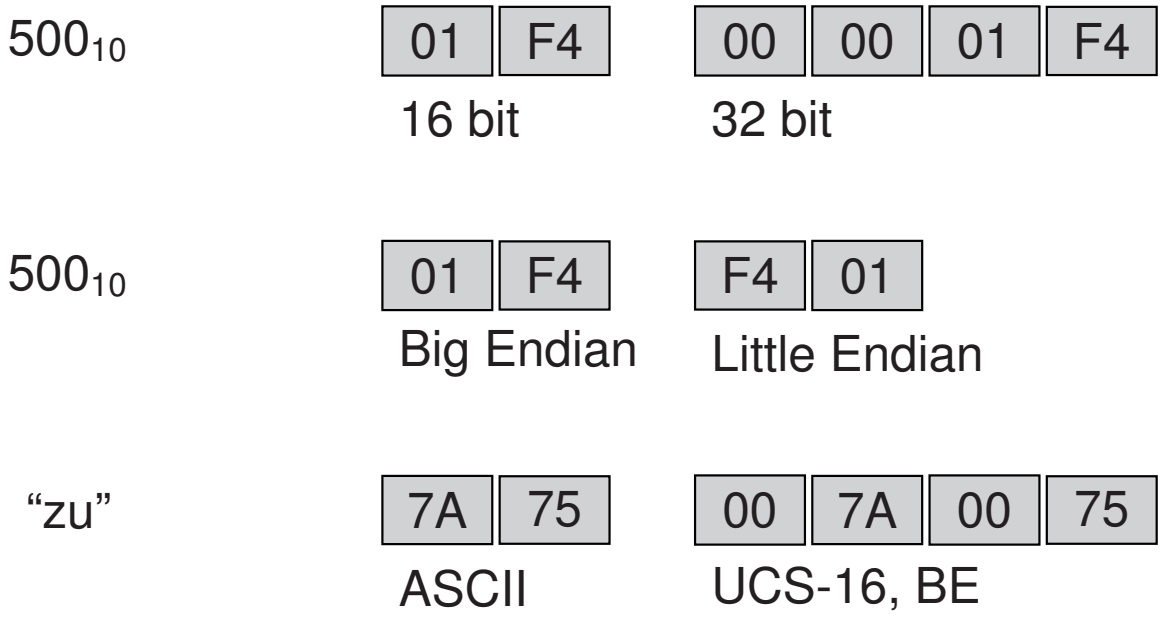
- Die meisten Programmiersprachen unterstützen Funktionen (oder Prozeduren), um Teile des Quell-Codes zu abstrahieren.
- Eine Funktion isoliert den inneren Code vom äußeren und hat ein klar definiertes Interface. Funktionen bieten sich daher für die Kommunikation zwischen Komponenten auf verschiedenen Rechnern an.
- Ein Remote Procedure Call ist ein Funktionsaufruf über Prozessgrenzen hinweg.
- Dabei liegt das Client-/Server-Modell zugrunde:
  Der Server bietet eine Funktion an, der Client ruft sie auf.

Was ist ein Funktionsaufruf?



- Es gibt viele verschiedene, inkompatible Konventionen (calling conventions). Die Daten werden je nach Konvention über den Stack und/oder Register übergeben.

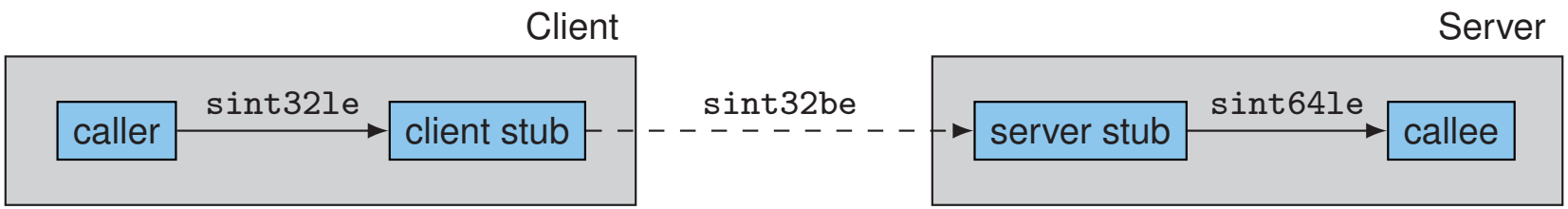
Wie sind die Parameter codiert?



- Fazit: Eine Funktion in einem anderen Prozess kann nicht direkt aufgerufen werden.

Stubs

- Für jede Funktion, die per RPC aufrufbar sein soll, wird sowohl für die Server- als auch die Client-Seite ein Stub erzeugt.
- Ein Stub ist eine Funktion, die anstelle der eigentlichen Funktion aufgerufen wird. Der Stub . . .
  1. konvertiert die Parameter der Remote Procedure in eine gemeinsame Darstellung und packt sie in eine Nachricht (Marshalling).
  2. verschickt die Nachricht an den Server.
  3. blockiert, bis eine Antwort eintrifft.
  4. konvertiert die Daten der Antwort zurück in die interne Darstellung (Unmarshalling).
- Auf der Server-Seite befindet sich ebenfalls ein Stub, der die Parameter in die Server-interne Darstellung konvertiert, die gewünschte Funktion aufruft und das Ergebnis zurücksendet.



IDL

- Stubs müssen nicht von Hand implementiert werden, sondern können automatisch für verschiedene Programmiersprachen erzeugt werden.

- Die Interface Description Language (IDL) soll Funktionen und Datentypen weitgehend sprachunabhängig definieren. Aus IDL-Definitionen werden die Stubs generiert.

Beispiel:

IDL
```
long min([in] long a, [in] long b);
```

C
```
long min(long a, long b);
```

Java
```
int  min(int a, int b);
```

Binding

- Da der Client-Stub zur Kompilierungszeit noch nicht die Adresse des Servers kennt (Dynamic Binding), fragt er zur Laufzeit einen dedizierten Dienst, rpcbind.

- rpcbind erwartet Anfragen normalerweise auf Port 111.

- Die Server-Prozesse registrieren sich bei rpcbind und teilen ihm folgende Informationen mit:
  - ID des Netzwerks dem der Server angehört
  - Programmnummer und Versionsnummer
  - Adresse und Port unter der der Server erreichbar ist

Java RMI

- Java ist eine objektorientierte Sprache mit komplexen Datentypen, Referenzen und Garbage Collection.
- Java Remote Method Invocation (RMI) ist eine RPC-Variante, die auf Java zugeschnitten ist. RMI ermöglicht es, komplexe Objekte und Objektgraphen als Parameter zu übergeben und unterstützt die verteilte Garbage Collection.
- Entfernte Objekte bleiben referenziert und werden nicht kopiert. Anstelle von Stubs, also Funktionen, erfüllt für RMI auf Client-Seite ein Proxy-Objekt und auf Server-Seite ein Skeleton-Objekt diese Rolle.
- Interfaces können bei RMI nicht mehr durch IDL beschrieben werden. Die Common Object Request Broker Architecture (CORBA) bietet jedoch eine zusätzliche Kapselung der Objekte, sodass trotzdem verschiedene Programmiersprachen verwendet werden können.

RPC/RMI

Vorteile

- Dank Marshalling können heterogene verteilte Systeme konstruiert werden, die aus unterschiedlichen
  - Hardware-Plattformen,
  - Betriebssystemen und
  - Programmiersprachen

  bestehen.

Nachteile

- Marshalling ist teuer, weil die Daten vor dem Versand bearbeitet und zum Teil umkopiert werden müssen.
- RPC ist ein Low-Level-Konstrukt, das weder Fehlertoleranz noch automatische Lastverteilung bietet.

Feature-Tabelle

	MPI	MapReduce	DUP	RPC
reguläre Datenstrukturen	×		×	×
skaliert automatisch	×	×		
automatische Verteilung		×		
Fehlertoleranz		×		
heterogen			×	×
Marshalling	·			×

Worum geht es in diesem Kapitel?

1 Motivation

2 Homogene, skalierbare Paradigmen
- MPI
- MapReduce
- Pipes

3 Remote Procedure Call

4 Shared Memory
- NUMA
- Paging
- Sharing auf Objektebene

5 Einbettung in Programmiersprachen

Shared Memory

Definition

Shared Memory ist Arbeitsspeicher, der von mehreren Prozessen gleichzeitig verwendet werden kann.

- Innerhalb eines sequentiellen Programms können Programmteile über Variablen und Objekte miteinander kommunizieren.
- Threads, die demselben Prozess angehören, arbeiten im gleichen Adressraum und können daher ebenfalls über den Arbeitsspeicher miteinander kommunizieren.
- Wenn Shared Memory zur Verfügung steht, kann eine verteilte Anwendung wie eine Multi-Threaded-Anwendung geschrieben werden. Dadurch erspart man sich das manuelle Kopieren von Daten zwischen den Prozessen via Message-Passing.
- Für die Performance ist aber auch in einem Shared Memory-System wichtig, wie die Daten auf die Rechner verteilt werden.

NUMA

- Mit NUMA-Architektur (Non-Uniform Memory Access) bezeichnet man einen Cluster, dessen Netzwerk den Prozessoren erlaubt, direkt auf die Speicher der anderen Prozessoren zuzugreifen.
- Die einzelnen Arbeitsspeicher der Rechner erscheinen dem Betriebssystem dabei wie ein einzelner, großer Arbeitsspeicher (Single System Image).
- Vorteile:
  - Zur Parallelisierung muss ein Programm lediglich so angepasst werden, dass es mehrere Prozessorkerne nutzen kann.
  - Da es sich um eine Hardware-Lösung handelt, ist die Latenz beim Zugriff auf entfernten Speicher geringer als bei Software-Lösungen.
- Nachteile:
  - “Non-Uniform” bezieht sich darauf, dass ein Prozessor aufgrund der Netzwerklatenz auf Adressen, die in den eigenen Arbeitsspeicher anstatt in entfernten verweisen, weitaus schneller zugreifen kann. Folglich müssen die Daten trotz der einheitlichen Sicht auf den Speicher sorgfältig angeordnet werden.
  - Die Caches aller Prozessoren kohärent zu halten skaliert schlecht. Ohne Cache-Kohärenz können die Programme wesentlich komplexer werden.

Virtueller Speicher

- Moderne Prozessoren verfügen über eine Memory Management Unit (MMU), die in Kooperation mit einem geeigneten Betriebssystem den Arbeitsspeicher des Systems virtualisieren kann.

  Beispiele: Linux, BSD, Windows (ab NT), MacOS (ab OS X)
- Der physische Arbeitsspeicher wird in Kacheln unterteilt, die typischerweise 4 KiB groß sind. Jedem Prozess wird ein eigener Adressraum zugeordnet, der in Seiten der gleichen Größe untergeteilt wird.
- Die Seitentabelle bildet für jeden Prozess die benutzten Seiten auf Kacheln ab. Die Seitentabelle entpricht dabei einer Abbildung: Seitentabelle : Prozesse $\times$ Seiten $\rightarrow$ Kacheln
- Die MMU verwendet bei jedem Speicherzugriff eines Prozesses die Seitentabelle, um die virtuelle Adresse des Prozesses in eine physische Adresse zu übersetzen.
- Dieser Vorgang ist für die Prozesse völlig transparent. Jeder Prozess kann nur auf seinen eigenen Adressraum zugreifen.

Auslagern

- Sollte in der Seitentabelle zu einer virtuellen Adresse kein gültiger Eintrag vorhanden sein, wird das Betriebssystem benachrichtigt (Interrupt Request). Das Betriebssystem hat dann die Möglichkeit, die Seite auf eine gültige Kachel abzubilden und den Prozess fortzusetzen.

- Diesen Umstand nutzt das Betriebssystem, um Seiten in einen Massenspeicher auszulagern und bei Bedarf zurückzuholen.

Distributed Shared Memory über Paging

- Ansatz: Anstatt zwischen Hauptspeicher und Massenspeicher werden die Seiten zwischen den Hauptspeichern eines Clusters getauscht.

- Der Speicher eines Prozesses kann sich so über viele Rechner erstrecken. Die Rechner erscheinen dabei wie ein einzelner Rechner mit vielen Rechenkernen. Der lokale Arbeitsspeicher verhält sich dabei wie ein Cache im Prozessor.

- [LH89] vergleicht verschiedene Methoden, den Zugriff auf und Austausch von Seiten zu verwalten.

Probleme

- Kacheln sind typischerweise 4 KiB groß. Diese Größe wird durch die Hardware vorgeschrieben. Die meisten Variablen und Objekte sind kleiner. Folge: Sehr verschiedene Daten könne in derselben Seite liegen.

- Über das Speichersystem können nur ganze Seiten angefordert werden. Wenn zwei verschiedene Prozessoren ständig auf zwei verschiedene Variablen zugreifen, die in derselben Seite liegen, muss diese Seite ständig zwischen den beiden Prozessoren hin- und hergeschoben werden, obwohl sie auf unterschiedlichen Daten arbeiten und daher unabhängig voneinander arbeiten könnten. Dieses Problem bezeichnet man als False Sharing.

- Unter anderem aufgrund der Performance-Probleme durch False Sharing hat sich dieser Ansatz in der Praxis nicht durchgesetzt.

Konsistenz in parallelen Programmen

- Die Funktion `remove` löscht ein Element aus einer doppelt-verketteten Liste:

```
1  void remove(list_element *el) {
2    el->prev->next = el->next;
3    el->next->prev = el->prev;
4    free(el);
5  }
```

- Problem: Nebeneinanderliegende Elemente können gleichzeitig von verschiedenen Threads gelöscht werden. Dabei befindet sich die Liste zwischen Zeile 2 und Zeile 3 in einem ungültigen Zustand.

- Traditionell wird dieses Problem behoben, indem sich ein Thread mit Hilfe von Locks exklusiven Zugriff auf die benötigten Daten sichert.

Code-Beispiel: Locking

- Thread-sicheres Entfernen eines Elements aus einer Linked List mit Hilfe von Locks

```
1  void remove(list_element *el) {
2    acquire(el->prev->lock);
3    acquire(el->lock);
4    acquire(el->next->lock);
5
6    el->prev->next = el->next;
7    el->next->prev = el->prev;
8
9    release(el->prev->lock);
10   release(el->lock);
11   release(el->next->lock);
12
13   free(el);
14 }
```

Code-Beispiel: Locking

- Achtung: Diese Variante führt zu Dead-Locks! Beispiel: Das aktuelle Element und sein Vorgänger werden gleichzeitig gelöscht. Beide sperren sich zunächst selbst. Am Ende wartet der Vorgänger auf dieses Element (Z. 4) und dieses Element auf den Vorgänger (Z. 3).

```
1  void remove(list_element *el) {
2    acquire(el->lock);
3    acquire(el->prev->lock);
4    acquire(el->next->lock);
5
6    el->prev->next = el->next;
7    el->next->prev = el->prev;
8
9    release(el->lock);
10   release(el->prev->lock);
11   release(el->next->lock);
12
13   free(el);
14 }
```

Software Transactional Memory

- Das Problem mit der Funktion `remove` ist, dass inkonsistenter Zwischenzustand beobachtet werden kann, während die Funktion ausgeführt wird, weil die Operationen nacheinander aus dem Speicher lesen und darauf schreiben.
- Das Problem würde nicht bestehen, wenn die Funktion atomar wäre, d. h., wenn sie gleichzeitig alle ihre Eingaben lesen und alle Ausgaben schreiben könnte.
- Software Transactional Memory bietet die Möglichkeit, einen Block aus Anweisungen als Transaktion zu markieren.
- Transaktionen sind atomar. Sie werden entweder ganz oder gar nicht ausgeführt und verhalten sich so, als ob sie während ihrer Ausführung eine Momentaufnahme des Gesamtsystems gesehen hätten.

Software Transactional Memory

- Der Code innerhalb des atomic Blocks wird in einer Transaktion ausgeführt:

```
1  void remove(list_element *el) {
2    atomic {                        // begin transaction
3      el->prev->next = el->next;
4      el->next->prev = el->prev;
5    }                               // commit transaction
6    free(el);
7  }
```

- Am Ende betritt eine Transaktion die Commit-Phase. Hier überprüft das System, ob während der Ausführung der Transaktion andere Transaktionen abgeschlossen wurden, die in Konflikt stehen.
- Ein Konflikt entsteht z. B. dann, wenn zwei laufende Transaktionen auf die gleiche Variable schreiben. In diesem Fall wird eine der beiden Transaktionen zurückgerollt und wiederholt.

Veranschaulichung

Umgangssprachlich ausgerückt kann man den Zwischenzustand einer Operation nicht beobachten, weil . . .

Atomar: zwischen Anfang und Ende keine Zeit vergeht.

Locking: es verboten ist, hinzusehen.

STM: man sich hinterher darauf einigt, dass man es nicht gesehen hat.

Distributed Software Transactional Memory

- Verteilter STM benötigt eine zentrale Anlaufstelle oder ein Konsens-Protokoll, damit die beteiligten Prozesse im Konfliktfall entscheiden können, welche Transaktionen zurückgerollt werden müssen.
- Beispiel: DecentSTM (dezentrales verteiltes Konsensprotokoll)

Nachteile

- Innerhalb einer Transaktion dürfen außer dem Lesen und Schreiben von Variablen keine Operationen mit Nebenwirkungen durchgeführt werden, weil deren Wirkung nicht rückgängig gemacht werden kann.
  Beispiele: Auf die Konsole schreiben, Atomraketen starten, ...
- Um auf Konflikte prüfen zu können, müssen sich die beteiligten Prozesse abstimmen. Diese Aufgabe übernimmt
  - ein zentraler Server, der ein Single Point of Failure ist — oder
  - ein Konsens-Protokoll, das einen hohen Kommunikationsaufwand haben kann.

Feature-Tabelle

	MPI	MapReduce	DUP	RPC	NUMA	Seitenbasierter DSM	Distributed STM
reguläre Datenstrukturen	×		×	×	×		
skaliert automatisch	×	×			·	×	×
automatische Verteilung		×					·
Fehlertoleranz		×					·
heterogen			×	×			
Marshalling	·			×			

Worum geht es in diesem Kapitel?

1 Motivation

2 Homogene, skalierbare Paradigmen
- MPI
- MapReduce
- Pipes

3 Remote Procedure Call

4 Shared Memory
- NUMA
- Paging
- Sharing auf Objektebene

5 Einbettung in Programmiersprachen

Erlang

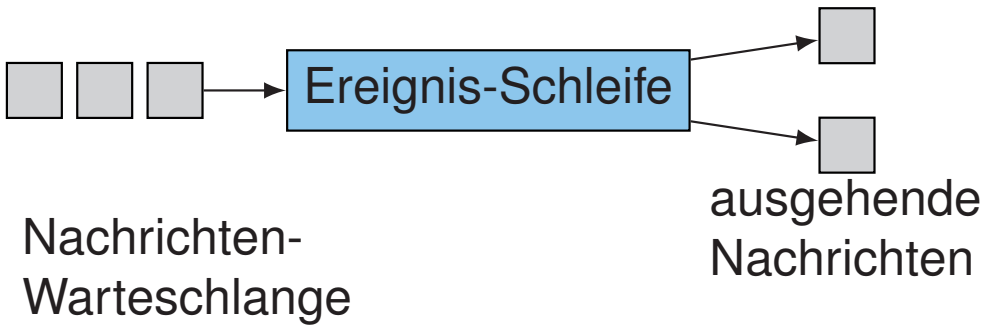
- Erlang ist eine dynamische, funktionale Programmiersprache mit primitiven Kommunikationsoperatoren.
- Ericsson hat inzwischen Erlang zusammen mit der verteilten In-Process-Datenbank „Mnesia“ und der „Open Telecommunication Platform“ (OTP) unter einer freien Lizenz veröffentlicht.
- Erlang wurde von Ericsson hauptsächlich entwickelt, um die Software für ihre ATM-Switches in einer Hochsprache implementieren zu können, die Kommunikation in einem verteilten System möglichst einfach macht.
- Bekannteste Anwendungen außerhalb der IT-Branche: ejabberd (Jabber-Server), CouchDB (NoSQL-Datenbank).

Was ist an Erlang hier interessant?

- Erlang integriert das Actor Model direkt in die Programmiersprache.
- Erlang bietet verteilte Fehlerbehandlung.

Das Actor Model

- Ein Actor besteht aus einer Nachrichten-Warteschlange (event queue) und einem Prozess, der in einer Ereignisschleife (event handler) die Nachrichten abarbeitet.
- Actors identifizieren sich gegenseitig über Process Identifiers (PIDs) und können sich darüber gegenseitig Nachrichten zusenden. Da die PID auch den Rechner identifiziert, können Nachrichten über Rechnergrenzen hinweg gesendet werden.



- Explizite Synchronisationskonstrukte, wie z. B. Mutexes, sind nicht notwendig, weil die Nachrichtenwarteschlange ein natürlicher Synchronisationspunkt ist.
- Der Nachrichtenversand ist ungeordnet: Die Nachrichten treffen nicht in der Reihenfolge in der Warteschlange ein, in der sie versandt wurden.

Erlang: Sequentielle Syntax

Primitive Datentypen

`a` Atom, ein Wert, der nur für sich selbst steht (wird meistens verwendet, um Nachrichten-Typen zu unterscheiden)

`{a, 23, "a"}` 3-Tupel, das aus einem Atom, einem Integer und einem String besteht

`foo(42)` Funktionsaufruf: Funktion “foo” mit Parameter 42

Definition einer Funktion

```
1  dist(X, Y) ->      % Funktion 'euclid' mit zwei Parametern
2    A = X*X + Y*Y,   % lokale Variable (read-only binding)
3    sqrt(A).         % Rueckgabewert ist Ergebnis von 'sqrt'
```

- Hinweis: Variablennamen beginnen in Erlang immer mit einem Großbuchstaben, Atome mit einem Kleinbuchstaben.

Erlang: Kommunikationsoperationen

Prozess starten

```
1  Pid = spawn(a, fun, args) % Neuen Prozess starten, der die
2                            % ... Funktion 'fun' ausfuehrt
```

Nachricht senden

```
1  Pid ! Msg   % Dem Prozess mit der PID 'Pid' den Wert 'Msg'
2              % ... als Nachricht senden
```

Nachrichten empfangen

```
1  receive                                    % Auf Nachricht warten
2    hello          -> doSomething();         % Mustervergleiche auf
3    {"hi", Name} -> doOtherThing(Name) % ... empfangener Nachricht
4  end.
```

Anmerkung: Erlang hat eine eigenwillige Syntax mit verschiedenen Interpunktionszeichen (Komma, Semikolon, Punkt), um Befehle voneinander zu trennen. Das liegt daran, dass die Entwickler ursprünglich eine PROLOG-ähnliche Sprache implementieren wollten.

Code-Beispiel: Erlang

- Ein einfacher Echo-Server. Siehe `http://www.erlang.org/doc/getting_started/users_guide.html` für ausführlichere Beispiele.

```
1  echo_server() ->
2    receive                  % Warte auf 'ping'-Nachricht.
3      {ping, Sender} ->
4        Sender ! pong,       % Antworte mit 'pong'.
5        echo_server()        % Rekursion, warte auf die naechste
6      end.                   % ... Nachricht.
7
8  start_echo_server() -> spawn(demo, echo_server, []).
9
10 main() ->
11   Echo = start_echo_server(), % Variable 'Echo' enthaelt PID
12   Echo ! {ping, self()},      % Sende 'ping' mit eigener PID.
13   receive                     % Warte auf Antwort.
14     pong -> io:format("Pong.~n", [])
15   end.
```

Programmieren in Erlang

- Das Starten neuer Prozesse ist in Erlang auch ein Mittel, um das Programm zu strukturieren — das bedeutet, man startet auch ohne Notwendigkeit neue Prozesse. Dadurch stehen manchmal schon genügend Prozesse zur Parallelisierung zur Verfügung, wenn das Programm auf einem größeren System ausgeführt wird.
- Prozesse werden nicht automatisch verteilt. Über einen zusätzlichen Parameter teilt man `spawn` mit, auf welchem Rechner der Prozess gestartet werden soll.

Verteilte Fehlerbehandlung in Erlang (1/2)

- Fehler (signals) werden in Erlang wie spezielle Nachrichten der Form `{'EXIT', FromPid, Reason}` behandelt. Empfängt eine Warteschlange diese Nachricht, wird der Prozess sofort beendet.
- Der Befehl link verbindet zwei Prozesse so miteinander, dass Fehlernachrichten in beide Richtungen weitergeleitet werden und folglich auch die Beendigung des Partners auslösen. Diese Verbindung ist transitiv.

Zwei Prozesse miteinander verbinden (link)

```
1  start() ->
2    Pid = spawn(demo, some_proc, []),
3    link(Pid),
4    % erledige andere Dinge
```

Verteilte Fehlerbehandlung in Erlang (2/2)

- Das Standard-Verhalten eines Prozesses bezüglich Signalen kann geändert werden:

```
1 process_flag(trap_exit, true)
```

- Dadurch wird die Fehlernachricht in eine gewöhnliche Nachricht umgewandelt, die der Prozess normal über die Warteschlange empfangen kann. Sie hat nicht mehr die Beendigung des Prozesses zur Folge und wird auch nicht mehr an verbundene Prozesse weitergeleitet.

```
1 receive
2   {'EXIT', FromPid, Reason} -> % ... Fehler behandeln
3 end.
```

- Strategie:
  - Alle Prozesse, die direkt voneinander abhängen, werden miteinander verbunden. Dadurch wird durch einen einzelnen Fehler eine Kaskade an Prozessen beendet.
  - Nur solche Prozesse fangen Fehlernachrichten ab, die den Fehler sinnvoll behandeln können. Diese Prozesse starten dann rekursiv die beendeten Prozesse neu.

Feature-Tabelle

	MPI	MapReduce	DUP	RPC	NUMA	Seitenbasierter DSM	Distributed STM	Erlang
reguläre Datenstrukturen	×		×	×	×			×
skaliert automatisch	×	×			·	×	×	
automatische Verteilung		×					·	
Fehlertoleranz		×					·	×
heterogen			×	×				
Marshalling	·			×				·

Bibliography I

[DGI04] Jeffrey Dean, Sanjay Ghemawat, and Google Inc, Mapreduce: simplified data processing on large clusters, In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation, USENIX Association, 2004.

[GGR+10] Christian Grothoff, Krista Grothoff, Matthew J. Rutherford, Kai Christian Bader, Harald Meier, Craig Ritzdorf, Tilo Eissler, Nathan Evans, and Chris GauthierDickey, Dup: A distributed stream processing language, IFIP International Conference on Network and Parallel Computing (Zhengzhou, China), Springer Verlag, Springer Verlag, 2010.

[LH89] Kai Li and Paul Hudak, Memory coherence in shared virtual memory systems, 1989.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen, Verteilte systeme, 2., aufl. ed., PEARSON STUDIUM, 2007.