IT-Infrastruktur

WS 2014/15

Hans-Georg Eßer
Dipl.-Math., Dipl.-Inform.

Foliensatz G: v1.1, 2015/01/22

- Zentrale und verteilte IT-Infrastrukturen
- Zusammenfassung der Vorlesung

Dieser Foliensatz

Vorlesungsübersicht
Seminar
Wiss. Arbeiten
Datenformate und Wandlung
PC als Arbeitsplatz
Ergonomie und Arbeitsschutz
Rechnerstrukturen
(Telekommunikation)
Infrastruktur-Technologie
Folien G → Zentrale / verteilte IT-Infrastrukturen

Zentrale und verteilte IT-Infrastrukturen

- Parallelität
- Verteilte Systeme
- Client / Server
- Grid
- Cloud
- Virtualisierung

Parallelität (1)

- Cloud-Computing basiert auf Grundlagen in den Bereichen
  - parallele und verteilte Systeme
  - Client/Server, Thin Clients
  - Kommunikationsprotokolle

Parallelität (2)

- Parallele Systeme
  - Probleme lösen, für welche die Ressourcen einer einzelnen Maschine nicht ausreichen
  - Benötigte Zeit für die Problemlösung reduzieren („Speed-up“)
  - Erfolg abhängig vom Anteil α des nicht-parallelisierbaren Codes
  - Amdahls Gesetz (1967):
    Speed-up $S(n) \rightarrow 1 / \alpha$ (für $n \rightarrow \infty$)

    z. B. α = 5 % = 0,05: 1/0,05 = 20 ist max. Speed-up

Parallelität (3)

- Parallele Systeme
  - Beispiel zu Amdahl:

    α = 5 %; Code braucht insgesamt 1000 Zeit-einheiten (ZE), 950 parallelisierbar, 50 sequentiell

n	Par.	Seq.	Summe	Speed-up
1	950	50	1000	1
2	475	50	525	1,90
10	95	50	145	6,90
50	19	50	69	14,49
950	1	50	51	19,6
∞	0	50	50	20,0

Parallelität (4)

- Was verhindert Parallelisierbarkeit?
  - Zwei Ereignisse sind nebenläufig, wenn keines die Ursache des anderen ist.
  - Datenabhängigkeit: z. B. RAW-Konflikt (vgl. Pipelines)
- Was verlangsamt parallele Threads?
  - Synchronisation (Mutex, Semaphore, Barrieren, Inter-Thread-Kommunikation)

Parallelität (5)

- Arten der Parallelität
  - **Bit-Level-Parallelität:** Zahl der Bits, die eine CPU-Instruktion bearbeitet (entspricht i.d.R. der Breite der Register)
  - **Instruction-Level-Parallelität:** Pipelining
  - **Daten-/Schleifen-Parallelität:** Schleifen können parallel bearbeitet werden
  - **Task-Parallelität:** Programm in unabhängige Threads aufteilen

Verteilte Systeme (1)

- Verteiltes System:
  - Zusammenschluss mehrerer Computer, auch mit unterschiedlicher Architektur
  - präsentiert sich gegenüber Nutzer als ein System
  - Komponenten arbeiten autonom, auch bzgl. Scheduling und Ressourcen-Verwaltung
  - ist durch Hinzufügen weiterer Rechner skalierbar
  - kein gemeinsamer Speicher, darum Synchronisation nur durch Nachrichtenversand möglich

Verteilte Systeme (2)

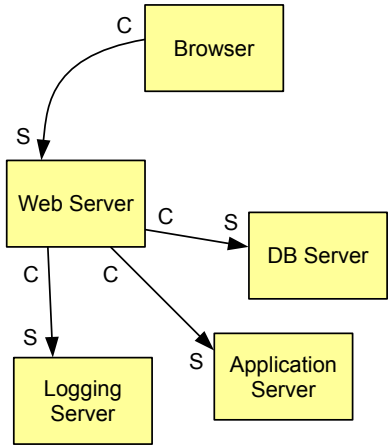
- Beispiele:
  - P2P-Netze (z. B. für Filesharing)
  - Verteiltes Dateisystem (z. B. IBM GPFS, Google Filesystem)
  - HPC, inkl. Cluster- und Grid-Computing, auch Seti@Home u. ä.

Client / Server (1)

- Server und Client sind Prozesse, die auf derselben Maschine oder auf verschiedenen, übers Netz verbundenen Rechnern laufen
- Server bietet eine Dienstleistung
- Clients können diese nutzen, indem sie eine Anfrage an den Server schicken
- wahlweise verbindungsbasiert (z. B. TCP) oder verbindungslos (z. B. UDP)
- Thin Client: nutzt überwiegend Dienste eines oder mehrerer Server

Client / Server (2)

- Client / Server-Beispiele:
  - WWW (HTTP, HTTPS)
  - FTP
  - Datenbank
  - Authentifikation



Client / Server (3)

- Remote Procedure Call (RPC)
  - Server erlaubt den Aufruf von Programm-Prozeduren (Funktionen) durch Clients
  - Aufruf (und Ergebnisrückgabe) durch Message Passing
  - Problem: Client und Server können verschiedene Architektur haben → andere Art der Datenablage (z. B. Little-Endian/Big-Endian, verschiedene Kodierungen für Strings)
  - **Marshalling** / **Serialisieren**: Packen der Prozedur-Argumente in ein universelles Format (Unmarshalling: auspacken)

Client / Server (4)

- RPC-Beispiele
  - Sun RPC (heute: Open Network Computing RPC)
    - Dienste über einen Portmapper registrieren, der auf Port 111 Anfragen annimmt
    - unterstützt Authentifikation
    - serialisiert Daten mit XDR (External Data Representation) → nächste Folie
    - z. B. NFS (Network File System) nutzt Sun RPC
  - XML-RPC
  - Web Services Description Language (WSDL), SOAP (Nachfolger von XML-RPC)

XDR-Beispiel: Datei-Versand

```
const MAXUSERNAME = 32;    /* Länge username  */
const MAXFILELEN = 65535; /* max. Dateigröße */
const MAXNAMELEN = 255;    /* Länge Dateiname */

/* Dateitypen: */
enum filekind {
   TEXT = 0,        /* ASCII-Text        */
   DATA = 1,        /* Binäre Daten      */
   EXEC = 2         /* ausführbare Datei */
};

/* Datei-Informationen: */
union filetype switch (filekind kind) {
case TEXT:
   void;
case DATA:
   /* Ersteller */
   string creator<MAXNAMELEN>;
case EXEC:
   /* Interpreter */
   string interpretor<MAXNAMELEN>;
};

/* komplette Datei: */
struct file {
   string filename<MAXNAMELEN>; /* Name     */
   filetype type;               /* Infos    */
   string owner<MAXUSERNAME>;   /* Besitzer */
   opaque data<MAXFILELEN>;     /* Inhalt   */
};
```

Beispiel:

Versand einer Lisp-Datei „sillyprog“ mit Inhalt „(quit)“

Pos.	Hex	ASCII	Kommentare
0	00 00 00 09	....	Länge Dateiname = 9
4	73 69 6c 6c	sill	Dateiname
8	79 70 72 6f	ypro	...
12	67 00 00 00	g...	... (und 3 Füller-Bytes)
16	00 00 00 02	....	Dateityp 2 (EXEC)
20	00 00 00 04	....	Interpreter-Länge 4
24	6c 69 73 70	lisp	Interpreter-Name
28	00 00 00 04	....	Besitzer-Länge 4
32	6a 6f 68 6e	john	Besitzer
36	00 00 00 06	....	Dateigröße = 6
40	28 71 75 69	(qui	Dateiinhalt
44	74 29 00 00	t)..	... (und 2 Füller-Bytes)

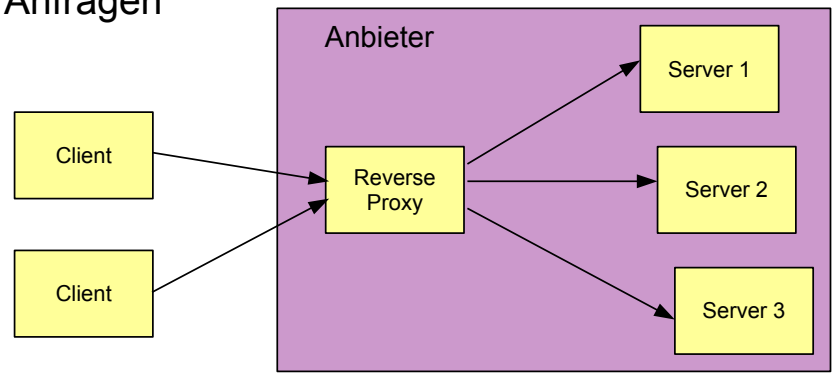
Quelle: http://tools.ietf.org/html/rfc1014

Vorteile Client/Server / RPC

- klare Trennung der Aufgaben
- Kommunikation über definiertes Protokoll
- leichter Austausch des Servers (oder des Clients)
- wenn komplexe (aufwendige) Aufgaben von Servern erledigt werden, kann Client klein sein (→ Thin Client)
- Server kann auch ein **Reverse Proxy** sein und Anfragen an einen Server weiterleiten

Reverse Proxy

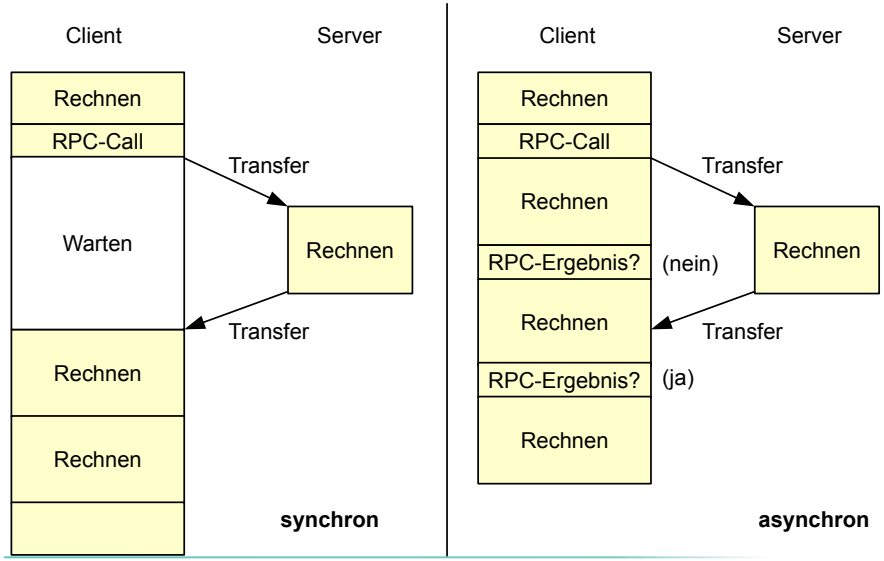
- Lastverteilung
- Puffern / Cachen der Antworten auf wiederholte Anfragen



RPC-Anwendung

- Aufruf von RPC-Prozeduren kann wie Aufruf lokaler Funktionen arbeiten
- normales Verhalten: blockierend (wie Funktionsaufruf)
- Alternative: Asynchronous RPC
  - Zugriff auf langsame / überlastete Server → nicht warten, bis Ergebnis da ist
  - Transfer großer Datenmengen
  - Rückgabewert muss dann (später) separat abgefragt werden → Callback

Asynchronous RPC



Cluster / Super-Computer (1)

- Super-Computer
  - zahlreiche (tausende) CPUs
  - gemeinsamer Speicher oder separater Speicher je CPU (→ Message Passing)
  - Plattform für High Performance Computing (HPC)
- Cluster
  - zahlreiche separate Rechner, typischerweise mit identischer Hardware
  - verbunden durch spezielles schnelles Netzwerk
  - Kooperation via Message Passing

Cluster / Super-Computer (2)

- Message Passing mit MPI
  - MPI-Standard (Message Passing Interface)
  - wird in parallelen Programmen zur Aufteilung der Arbeit verwendet
  - typische Operationen:
    - **Send** (an einzelnen) und **Recv** (von einzelnen)
    - **Broadcast** (Nachricht an alle)
    - **Gather** (Daten von allen Prozessen erhalten; entspricht *n* mal Recv) und **Reduce** (z. B. Summe der Werte von allen *n* Prozessen erhalten)
    - **Barrier** (Schranke, die alle Prozesse erreichen müssen)

MPI-Beispiel

```
int main(int argc, char *argv[]) {
  char idstr[32]; char buff[BUFSIZE];
  int numprocs; int myid; int i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);  // wie viele?
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);      // wer bin ich?
  if(myid == 0) {  // Master (Rank 0)
    printf("%d: We have %d processors\n", myid, numprocs);
    for (i=1; i<numprocs; i++) {
      sprintf(buff, "Hello %d! ", i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
    }
    for (i=1; i<numprocs; i++) {
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf("%d: %s\n", myid, buff);
    }
  } else {         // Slave (Rank != 0)
    // Nachricht von Prozess 0 empfangen
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "active", BUFSIZE-1);
    // Nachricht an Prozess 0 senden
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }
  MPI_Finalize(); return 0;
}
```

```
$ gcc -lmpi mpitest.c
$ mpirun -n 8 ./a.out
0: We have 8 processors
0: Hello 1! Processor 1 active
0: Hello 2! Processor 2 active
0: Hello 3! Processor 3 active
0: Hello 4! Processor 4 active
0: Hello 5! Processor 5 active
0: Hello 6! Processor 6 active
0: Hello 7! Processor 7 active
```

Quelle: http://en.wikipedia.org/wiki/Message_Passing_Interface#Example_program, leicht verändert

Grid Computing (1)

- Idee auch beim Grid: Lösung einer Aufgabe auf mehrere Maschinen verteilen
- Funktion: ähnlich wie Cluster, aber
  - zum Grid gehörende Rechner sind räumlich weit entfernt (keine schnelle Verbindung)
  - Rechner haben verschiedene Architektur und Hardware-Ausstattung
  - schlecht für Aufgaben, die viel Kommunikation der beteiligten Prozesse benötigen
  - schlecht, wenn parallele Anwendung voraussetzt, dass alle Nodes in etwa gleich schnell arbeiten
  - Normalfall: völlig unabhängige Prozesse

Grid Computing (2)

- „Shared Computing“: Idle-Zeiten (ungenutzte Ressourcen) nutzen
  - SETI@Home
  - kryptographische Verfahren brechen (→ z. B. RSA Secret-Key Challenge)
  - BOINC (Berkeley Open Infrastructure for Network Computing), Projekte: siehe http://boincstats.com/en/stats/projectStatsInfo
  - Auswertung riesiger Datenmengen, die nicht realistisch von Einzelnpersonen oder Unternehmen verarbeitbar sind

Grid Computing (3)

- Beteiligung an Shared-Computing-Projekten
  - Account anlegen
  - Client installieren und laufen lassen
  - bei Interesse Statistik über eigenen Beitrag abrufen
- Grid Computing (und auch Cluster / Super-computer) eher im wissenschaftlichen Umfeld relevant
- Kommerzielle Bedeutung von parallelen Systemen erst durch Cloud Computing

Cloud Computing (Überblick, 1)

- erste Umsetzung verteilter Systeme mit starker kommerzieller Bedeutung
- Anbieter u. a. Amazon, Google, Oracle, SAP, Microsoft, Dropbox … (Liste der Top-100-Cloud-Provider: http://talkincloud.com/tc100)
- drei sehr unterschiedliche Angebotsvarianten:
  - Software as a Service (SaaS)
  - Platform as a Service (PaaS)
  - Infrastructure as a Service (IaaS)

Cloud Computing (Überblick, 2)

- Clouds bieten *skalierbare* und *elastische* Services
  - *skalierbar*: tatsächlich genutzte Ressourcen vom Anwender konfigurierbar
  - *elastisch*: Überwachung der Nutzung; je nach Bedarf wird Rechenleistung, Plattenplatz, Netzwerk-Bandbreite automatisch angepasst
- Messbarer Ressourcenverbrauch → individuelle Gebühren entsprechen der Nutzung
- Cloud-Provider übernehmen Verwaltung und Sicherheit
- zentralisierte Cloud-Rechnerfarmen

Cloud Computing (Überblick, 3)

- Cloud-Anbieter setzen **Virtualisierung** ein und profitieren dadurch von der Lastverteilung (nicht alle Instanzen arbeiten ständig mit Höchstlast)
- geringere Kosten für RZ-Betrieb können an Cloud-Anwender weitergegeben werden
- Daten werden in der Nähe der Rechner vorgehalten, die sie verwenden

Cloud Computing (Überblick, 4)

- Cloud-Arten
  - **Public Cloud:** Infrastruktur gehört einem Cloud-Anbieter, Anwender greifen über das Internet darauf zu. Keine Einschränkung der Benutzergruppe
  - **Private Cloud:** Infrastruktur wird von/für ein einzelnes Unternehmen betrieben
  - **Community Cloud:** ähnlich Public Cloud, aber mit eingeschränkter Benutzergruppe, z. B. nur Krankenhäuser, öffentliche Verwaltungen etc.
  - **Hybrid Cloud:** Kooperation mehrerer Clouds (public, private, community) mit Schnittstellen

Cloud Computing (Überblick, 5)

- Vorteile beim Cloud-Einsatz (für Anwender):
  - keine Anfangsinvestition für eigenes RZ
  - „pay as you go“
  - Elastizität: Leistungen können mit wachsender Benutzergruppe dynamisch mitwachsen; in Zeiten der Nicht-Nutzung keine Kosten
  - Virtualisierung lässt Benutzer (bei IaaS) mit gewohnten Umgebungen arbeiten

Cloud Computing (Überblick, 6)

- Clouds erfolgreicher als Grid, Cluster etc.:
  - Fokus auf Enterprise Computing, nicht wissenschaftliches Rechnen (HPC)
  - Homogene Hardware (anders als beim Grid)
  - Standard-Hardware (anders als beim Supercomputer)
  - Alle Hardware unter einheitlicher Kontrolle des Cloud-Anbieters → Sicherheit, Fehlertoleranz, Quality of Service: leichter umsetzbar

Cloud Computing (Überblick, 7)

- Software as a Service
  - Service Provider stellt Anwendungen zur Verfügung
  - Benutzer verwaltet *nicht* die zugrundeliegende Cloud-Infrastruktur oder Anwendungseinstellungen
  - Services:
    - Enterprise services: workflow management, groupware and collaborative, supply chain, communications, digital signature, customer relationship management (CRM), desktop software, financial management, geo-spatial, search.
    - Web 2.0 applications: metadata management, social networking, blogs, wiki services, portal services.

Cloud Computing (Überblick, 8)

- Software as a Service
  - ungeeignet, wenn Daten nicht extern gehostet werden dürfen
  - Beispiele: GMail, Google-Suchmaschine

Cloud Computing (Überblick, 9)

- Platform as a Service
  - Service Provider stellt Plattform zur Verfügung, auf der vom Anwender entwickelte Anwendungen laufen
  - Programmiersprachen und Tools vom Provider vorgegeben
  - Benutzer verwaltet *nicht* die zugrundeliegende Cloud-Infrastruktur (wie etwa Betriebssystem, Netzwerk-Konfiguration etc.)

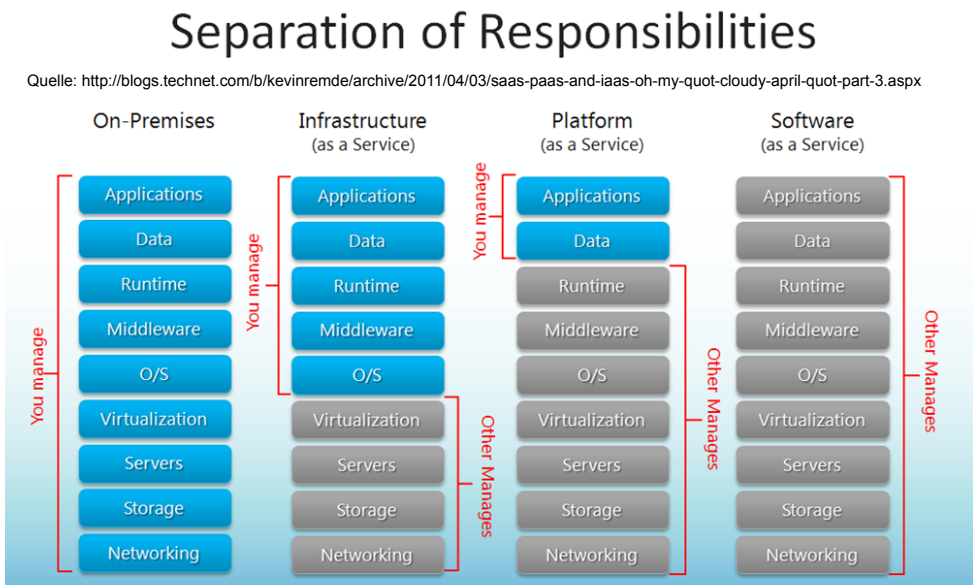
Cloud Computing (Überblick, 10)

- Platform as a Service
  - ungeeignet, wenn Optimierung der „Hardware“ und Software nötig ist oder spezielle (proprietäre) Programmiersprachen verwendet werden
  - Beispiele: Microsoft's Windows Azure (PaaS und IaaS), Google App Engine

Cloud Computing (Überblick, 11)

- Infrastructure as a Service
  - Benutzer können beliebige Software ausführen, darunter auch (weitgehend) beliebige Betriebssysteme
  - Benutzer kontrolliert *nicht* die Cloud-Infrastruktur, *aber* die in seinen VMs laufenden Betriebssysteme, Netzwerk- und Firewall-Einstellungen, Software und deren Konfiguration

Cloud Computing (Überblick, 12)



Ausblick

- Nächster Termin:
  - technische Grundlagen der Clouds
  - ausgewählte Beispiele
  - Sicherheit in der Cloud
  - Entwickeln von Cloud-Anwendungen (nur ein grober Überblick)

Cloud Computing (Überblick, 13)

- Gründe für den Einsatz von Public Clouds:

Reason	Percentage who agree
Improved system reliability and availability	50%
Pay only for what you use	50%
Hardware savings	47%
Software license saving	46%
Lower labor costs	44%
Lower maintenance costs	42%
Reduced IT support needs	40%
Ability to take advantage of the latest functionality	40%
Less pressure on internal resources	39%
Solve problems related to updating/upgrading	39%
Rapid deployment	39%
Ability to scale up resources to meet the needs	39%
Ability to focus on core competencies	38%
Take advantage of the improved economics of scale	37%
Reduced infrastructure management needs	37%
Lower energy costs	29%
Reduced space requirements	26%
Create new revenue streams	23%

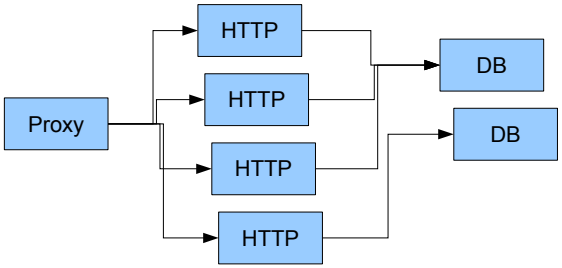
Cloud: Technische Details

- Verteilte Konfiguration mit ZooKeeper
- Software-Entwicklung mit MapReduce
- Verteilte Dateisysteme

22.01.2015: ab hier neue Folien

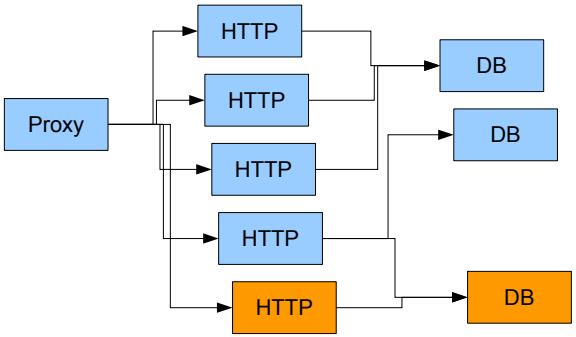
ZooKeeper (1)

- Verteiltes Repository von Konfigurationsdaten
- Beispiel: elastische HTTP/DB-Umgebung



ZooKeeper (2)

- Elastisch: neue VM mit HTTP/DB starten



- Proxy/HTTPs müssen neue Server kennen

ZooKeeper (3)

- ZooKeeper verwaltet eine Art Verzeichnisbaum

```
/
/config
/config/http-servers
/config/http-servers/http1
/config/http-servers/http2
/config/http-servers/http3
/config/http-servers/http4
/config/db-servers
/config/db-servers/db1
/config/db-servers/db2
```

speichern u.a.
– IP-Adresse
– Versionsinformation

dito

- neue Server tragen sich hier ein
- Proxy und HTTPs fragen Informationen ab

ZooKeeper (4)

- Load-Verwaltung bemerkt wachsende Last
  - neue VM mit HTTP und DB starten
  - VM im ZooKeeper registrieren
  - Wegen *WATCH* werden Proxy und HTTPs benachrichtigt

```
/
/config
/config/http-servers
/config/http-servers/http1
/config/http-servers/http2
/config/http-servers/http3
/config/http-servers/http4
/config/http-servers/http5
/config/db-servers
/config/db-servers/db1
/config/db-servers/db2
/config/db-servers/db3
```

ZooKeeper (5)

- Zwei DB-Server wollen gleichzeitig einen Datensatz ändern → Locking!
  - ZooKeeper bietet systemübergreifende Locks
  - Nur eine der Maschinen kann das Lock erhalten, die übrigen müssen warten
  - Bei Freigabe des Locks werden die wartenden Maschinen benachrichtigt (WATCH)

```
/
/locks
/locks/db
/locks/db/table_customers
```


ZooKeeper (6)

- *Persistant* vs. *Ephemeral* Nodes
  - *persistant*: Node bleibt permanent erhalten
  - *ephemeral*: Node verschwindet, wenn sich die erzeugende Maschine abmeldet (oder ausfällt)

Funktionale Programmierung (1)

Basics

- Listen
  - `[]` ist leere Liste
  - `a::list` – a wird vorne an Liste `list` angehängt (Ergebnis ist neue Liste)
  - `::` heißt **Prepend**-Operator (vgl. Append)

Funktionale Programmierung (2)

- Map-Operation
  - `map(f,[])  = []`
  - `map(f,a::list) = f(a)::map(f,list)`
  - z. B.: `f=(x->3*x)`, `map(f,[1,2,3])=[3,6,9]`

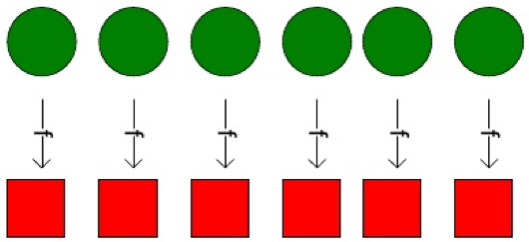


Bild: http://de.slideshare.net/tugrulh/presentations (Google-Präsentation)

Funktionale Programmierung (3)

- Fold-Operation
  - wendet eine Funktion `f(x,y)` mit zwei Argumenten der Reihe nach auf alle Elemente der Liste an

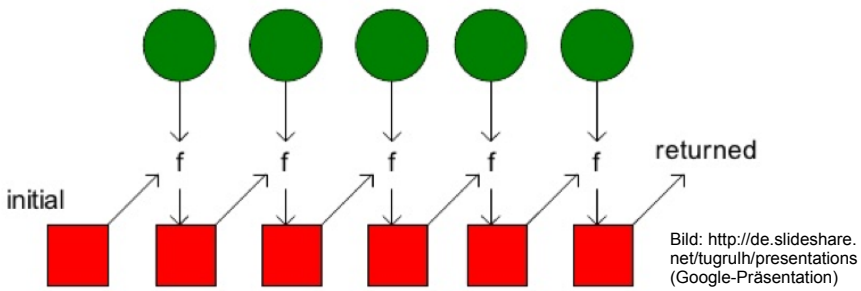


Bild: http://de.slideshare.net/tugrulh/presentations (Google-Präsentation)

Funktionale Programmierung (4)

- Fold-Operation (Forts.)
  - braucht Startwert
  - `foldl(add,0,[1,2,3]) = 3+(2+(1+0)) = 6` (Summe aller Listenelemente)
  - `foldl(mult,1,[3,4,5]) = 5*(4*(3*1)) = 60` (Produkt aller Listenelemente)
  - `foldr`: arbeitet sich von rechts nach links durch

Funktionale Programmierung (5)

- Implementierung von `foldl` und `foldr`
  - `foldl(f, a, []) = a`
  - `foldl(f, a, x::rest)`
    `= foldl(f, f(x,a), rest)`
  - `foldr(f, a, []) = a`
  - `foldr(f, a, x::rest)`
    `= f(x, foldr(f, a, rest))`
  - a heißt „Akkumulator“

Funktionale Programmierung (6)

- Dabei muss `foldl` nicht zwingend ein einzelnes Element als Ergebnis erzeugen

  → kann auch Liste erzeugen
- Verwende „Prepend“-Operator `::` als Funktion

Funktionale Programmierung (7)

- Beispiel: Reverse-Funktion für Listen
  - `f(x,a) = x :: a` (a ist Liste)
  - `reverse (list) = foldl (f, [], list)`
  -
```
reverse ([1,2,3])
= foldl (f, [], [1,2,3])
= foldl (f, f(1,[]), [2,3])

= foldl (f, [1], [2,3])
= foldl (f, f(2,[1]), [3])

= foldl (f, [2,1], [3])
= foldl (f, f(3,[2,1]), [])

= foldl (f, [3,2,1], [])        = [3,2,1]
```


Funktionale Programmierung (8)

- Beobachtung: `map` verarbeitet alle Listenelement unabhängig (während `foldl` und `foldr` sequentiell arbeiten)
- Darum lässt sich `map` gut parallelisieren.

MapReduce (1)

- Verarbeitung großer Datenmengen, z. B. Terabytes
- parallelisieren: auf hunderte oder tausende CPUs verteilen
- MapReduce erlaubt
  - automatische Parallelisierung und Verteilung
  - Fehler-Toleranz
  - Statusanzeige, Monitoring

MapReduce (2)

- übernimmt Konzepte der funktionalen Programmierung
  - zu verarbeitende Daten auf mehrere Maschinen aufteilen
  - jede Maschine führt auf den (ihr zugeteilten) Daten eine **Map**-Funktion aus
  - nach Abschluss aller Berechnungen werden die Ergebnisse gesammelt
  - Dann kombiniert eine **Reduce**-Funktion (wie etwa fold) alle Ergebnisse

MapReduce (3) – Beispiel

- Beispiel: gemeinsame Freunde (Facebook o. ä.)
  - Ausgangssituation: Für jeden Teilnehmer X eine Liste seiner Freunde (`key -> value`)

```
A -> B C D          D -> A B C E
B -> A C D E        E -> B C D
C -> A B D E
```

  - Jede Zeile an eine Maschine schicken und dort eine **Map**-Funktion starten

```
A -> B C D   ──────►   (A B) -> B C D
                       (A C) -> B C D
                       (A D) -> B C D
```

Quelle für Beispiel: http://stevekrenzel.com/finding-friends-with-mapreduce

MapReduce (4) – Beispiel

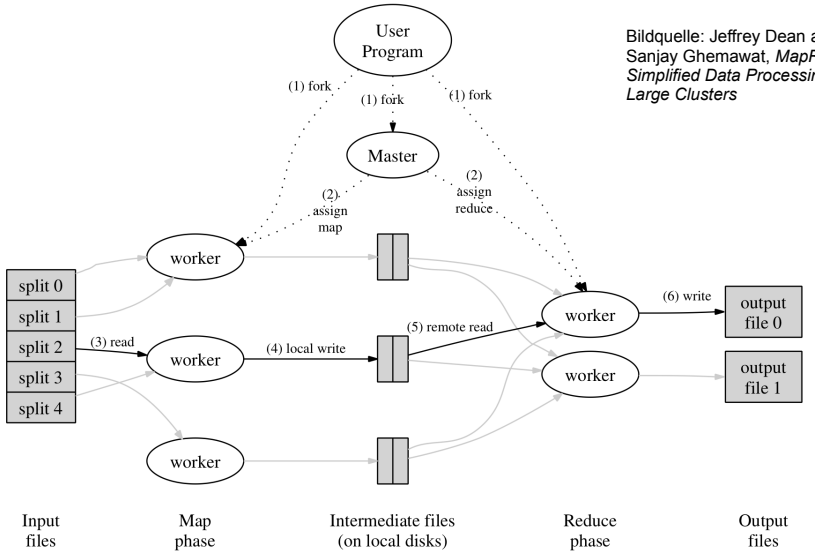
- Nach Abschluss der Berechnungen alle Ergebnisse zusammenfassen, nach *key* gruppiert

```
(A B) -> (A C D E) (B C D)        (B E) -> (A C D E) (B C D)
(A C) -> (A B D E) (B C D)        (C D) -> (A B C E) (A B D E)
(A D) -> (A B C E) (B C D)        (C E) -> (A B D E) (B C D)
(B C) -> (A B D E) (A C D E)      (D E) -> (A B C E) (B C D)
(B D) -> (A B C E) (A C D E)
```

- Abschluss: für jeden Key die **Reduce**-Funktion aufrufen (hier: Schnittmenge berechnen)

```
(A B) -> (C D)        (B E) -> (C D)
(A C) -> (B D)        (C D) -> (A B E)
(A D) -> (B C)        (C E) -> (B D)
(B C) -> (A D E)      (D E) -> (B C)
(B D) -> (A C E)
```


MapReduce (5)



Bildquelle: Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*

Verteilte Dateisysteme (1)

- Gemeinsames (globales) Dateisystem für alle Mitglieder der Cloud
- Einfacher NFS-Server, dessen Shares an alle Clients exportiert werden?
  - Daten sind immer remote
  - langsame Übertragung aller Daten
  - NFS-Server ist ein Bottleneck, weil etliche Rechner in der Cloud gleichzeitig darauf zugreifen
  - skaliert nicht

Verteilte Dateisysteme (2)

- Statt zentralem Fileserver: verteilten Server nutzen
- mehrere Server im Netz bilden gemeinsam einen Fileserver
  - Ausfallsicherheit durch Duplikation (Redundanz)
  - Datenlokalität: Daten dort speichern, wo sie benötigt werden

IBM Global Parallel Filesystem (GPFS)

- Verteiltes Dateisystem
- ab 2000; zunächst für HPC-Cluster entworfen
- parallele Zugriffe mehrerer Rechner
- Locking schützt Schreibvorgänge
  - Locking verwendet Tokens mit Ablaufzeit
    → keine explizite Rückgabe des Locks nötig
    → reduzierter Traffic
- File Placement Optimizer (FPO)
  - Daten liegen auf Platten in der Nähe des Servers
  - für MapReduce-Berechnungen vorbereitet

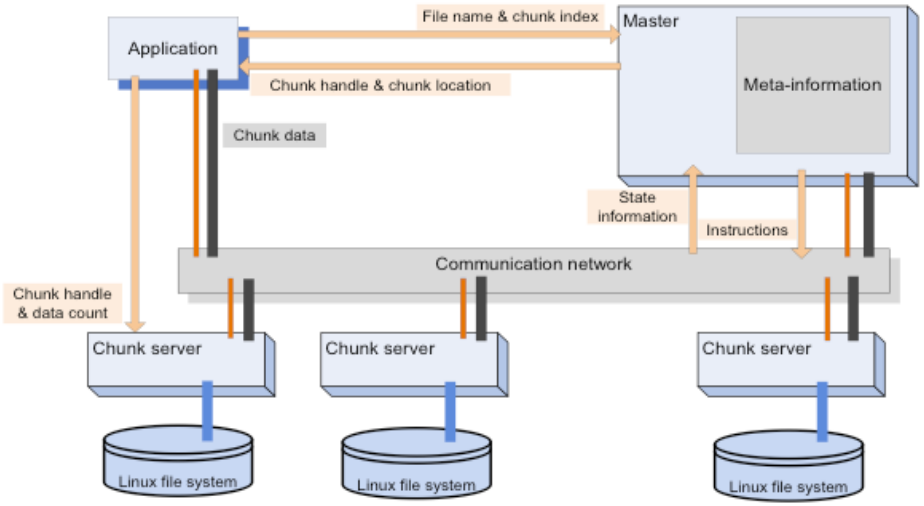
Google-Filesystem (1)

- Ausgangssituation bei Google
  - einige Millionen Dateien mit jeweils > 100 MByte
  - Dateien vom Typ „write-once“, mit Append
  - Lese-Zugriff auf diese Dateien über Streaming
  - bei Zugriff: hoher Durchsatz wichtiger als niedrige Latenz

Google-Filesystem (2)

- Design-Entscheidungen für GFS
  - Dateien in 64-MByte-Chunks aufteilen
    → Performance-Optimierung für große Dateien
  - Atomare Append-Operation; mehrere Prozesse können an gleiche Datei Daten anhängen
  - Replikation: jeder Chunk auf ≥3 Chunk-Servern
  - ein Master speichert Metadaten, koordiniert Zugriffe
    → soll Konsistenz garantieren (aber: Bottleneck)

Google-Filesystem (3)



Bildquelle: Marinescu, *Cloud Computing, Theory and Practice*

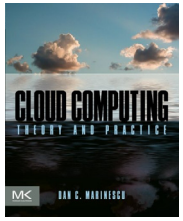
Google-Filesystem (4)

- GFS Chunks
  - **Chunk**-Größe 64 MByte, aber zerlegt in **Blöcke** der Größe 64 KByte
  - Jeder 64-KByte-Block hat 32-Bit-Prüfsumme
- Locking erfolgt auch Chunk-Ebene

Google-Filesystem (5)

- Schreiben eines Chunks
  - Master wählt einen der Chunk-Server als **Primary** aus, der schreib-berechtigt ist
  - Client (der schreiben will) bittet Master um Schreibberechtigung
  - Master schickt Client Information über Primary und alle Secondary Chunk Servers
  - Client schickt Daten an alle (!) Chunk-Server
  - Alle Chunk-Server schicken ACK an Client
  - Client schickt Write-Request an Primary
  - Primary schickt Write-Request an alle Seconarys

Literatur



- Dan C. Marinescu: *Cloud Computing – Theory and Practice*, Morgan Kaufmann 2013
- Jimmy Lin and Chris Dyer: Data Intensive Text Processing with MapReduce, 2010, http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

ZUSAMMENFASSUNG
und letzte Fragen

A: Datenformate und Wandlung (1)

- Information vs. Daten
- Bits, Bytes, dual/oktal/hex
- ASCII, Unicode (UTF-8, UTF-16)
- Digitalisierung:
  - rastern (diskretisieren, samplen) und
  - quantisieren (Wertebereich einschränken)
- Kompression (verlustfrei/verlustbehaftet)
- Prüfsummen, Fehlerkorrektur
- Mark-up: HTML/CSS, LaTeX, Wiki, XML

A: Datenformate und Wandlung (2)

- Mark-up: HTML/CSS, LaTeX, Wiki,
- XML, XSLT, DTD, XHTML, DocBook, Open Document Format (XML)
- Rastergrafiken (JPG vs. PNG), Vektorgrafiken
- PostScript, PDF
- MPEG, Frames
- Kommunikation
  - Client / Server; HTTP, FTP, SMTP

A: Datenformate und Wandlung (3)

- Datenträgeraustauschformat (DTaus)
- Applikationsformate (CAD, GIS, DTP)
- Archive und Software-Pakete
  - zip, tar, gz, tar.gz (plattform-übergreifend?)
  - Linux: RPM, Debian;
    - Pakete mit Metadaten
    - Abhängigkeiten, Konflikte
    - Repositories: Abh. auflösen, Upgrade
  - Windows: MSI, OS X: DMG mit *.app-Ordnern

B: PC als Arbeitsplatz

- Exkurs: LaTeX, BibTeX
- Statistik: Programmiersprache R
- Numerik: GNU Octave
- Computer-Algebra-Systeme, Wolfram Alpha
- GIS: Mark-up-Sprachen, Nielsen-Gebiete
- Versionsverwaltung, Mercurial (hg), Klonen, Merge-Operation

D: Rechnerstrukturen (1)

- Universalrechner, von-Neumann vs. Harvard
- ISA (Instruction Set Architecture)
  - Maschinenbefehle, Register
  - Adressierungsarten, Interruptbehandlung
  - 1-/2-/3-Adress-*Maschinen* bzw. -*Befehle*
  - Spezialregister (IP, SP, Status)
- Load, Store, Push, Pop, arithm. Operationen

  → RISC, CISC

D: Rechnerstrukturen (2)

- Pipelining
  - 5-stufige RISC-Pipeline (Fetch, Decode, Execute, Memory Access, Write-back)
  - 6-stufige CISC-Pipeline (Fetch, Decode, Calculate Operands, Fetch Operands, Execute, Write-back)
  - Pipeline-Hemmnisse
    - strukturell (Speicherzugriff bei **M**, **F**)
    - Datenabhängigkeiten (RAW-Konflikt)
    - ablauf-bedingt (bedingter Sprung; Sprungvorhersage)

D: Rechnerstrukturen (3)

- Superskalare Architekturen
  - mehrere Execute-Einheiten, nach Aufgaben getrennt (z. B. FPU, Int, MUL)
  - abhängige vs. unabhängige Pipelines
  - Datenabhängigkeiten (RAW, WAR, WAW; RAR)
  - Abhängigkeitsgraph (keine RARs, keine transitiven Abhängigkeiten)
  - Reorder-Buffer
    - Platz im Buffer, Anzahl der Ausführ-Einheiten (pro Kategorie), Anzahl der pro Takt zuteilbaren (issue) und bestätigten (commit) Befehle

E: Ergonomie und Arbeitsschutz (1)

- „Gebrauchstauglichkeit“
- Nutzungskontext(e), Zielgruppe(n)
- effektiv, effizient, zufrieden stellend
- Kriterien
  - Aufgabenangemessenheit
  - Selbstbeschreibungsfähigkeit
  - Steuerbarkeit
  - Erwartungskonformität
  - Fehlertoleranz
  - Individualisierbarkeit
  - Lernförderlichkeit

E: Ergonomie und Arbeitsschutz (2)

- Beispiele:
  - Microsoft Ribbons
  - Seriennummern-Eingabe
  - Secure Shell (ssh, scp; Optionen -p, -P)
- Kriterien auch auf Webdesign anwendbar
- Arbeitsschutz

G: Zentrale/verteilte IT-Infrastrukturen (1)

- Parallele Systeme
- Programm parallelisieren, Speed-up
  → Amdahls Gesetz: $S(n) \rightarrow 1 / \alpha$
- Arten der Parallelität (Bit-Level, Instruction-Level, Daten/Schleifen-Parall., Task-Parall.)
- Verteilte Systeme
- Client / Server, RPC, Reverse Proxy, Asynchronous RPC
- Cluster, Super Computer (HPC, MPI), Grid

G: Zentrale/verteilte IT-Infrastrukturen (2)

- Cloud Computing
  - skalierbar, elastisch, abrechenbar, zentralisiert
  - Virtualisierung
  - public / private / community / hybrid cloud
  - Cloud-Vorteile: pay as you go, keine Investitionen
  - Varianten:
    - Software as a Service (SaaS),
    - Platform as a Service (PaaS),
    - Infrastructure as a Service (IaaS)

G: Zentrale/verteilte IT-Infrastrukturen (3)

- Technische Umsetzung
  - Verteilte Konfiguration mit ZooKeeper
  - Map und Fold (funktionale Programmierung)
  - Software-Entwicklung mit MapReduce
  - Verteilte Dateisysteme (GPFS, GoogleFS statt NFS)