Verteilte Systeme

Definition: Rechnernetz ohne gemeinsamen Speicher, Datenaustausch per Nachrichtenkommunikation
Vorteile: Leistungssteigerung, Verfügbarkeit
Nachteile: Konsistenz, Uhr, Komplexität
Speedup: $T_1(n)$, Rechenzeit auf einem Kern, $T_k(n)$, Rechenzeit auf k Kernen
$\frac{T_1(n)}{T_k(n)} \leq k$
Eine schlechte Partitionierung (Aufteilung der Arbeit) kann zu einem Speedup < 1 führen

ISO Referenzmodell:

4. Transportlayer: TCP/UDP
5. Session Layer: Sicherungspunkte, Recovery
6. Presentation Layer: Verschlüsselung, Umwandlung von Datenformaten
7. Application Layer: FTP,SMTP etc.

IP Adressen

V4: 32 Bit, 7/14/21 Bit Netzteil und 24/16/8 Hostteil
V6: 128 Bit

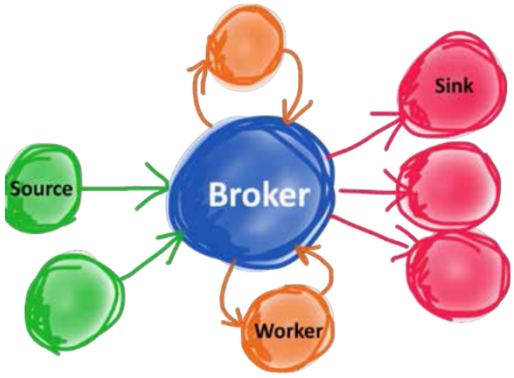
UDP

Best effort, max 64kb

ZMQ

Message Protocol: Daten werden als Nachrichten versendet
Dabei bietet ZMQ ein hohes Abstraktionslevel
Message Queue: Asynchron, Buffer für Nachrichten

Patterns:
Request-Reply: Synchron, Request -> warten auf Reply
Push-Pull: Keine Antwort erwartet
Pub-Sub: 1 zu n, Datenverteilung
Beispiel:
Source: Request
Broker: Reply (Source),Publish(Sink), Push(worker)
Worker: Pull und Request
Sink: Sub
Transportvarianten:
INPROC: Zwischen Threads desselben Prozesses
IPC: Zwischen Prozessen
TCP: TCP halt
PGM: pragmatic multicast
EPGM: encapsulated pgm



Lastverteilung

Kreative Lastverteilung: Verteilte Programmierung
Mechanische Lastverteilung: Auslastung mehrerer Rechner
Aufteilung der Last in kleinere Pakete, je kleiner, desto eher haben alle Rechner die gleiche Last, aber: Auf-/Verteilen kostet, zu kleine Pakete erhöhen Last
Grundlage: Lastwert auf Rechner verteilen, dann Lastpakete verteilen

Metriken: Prozessorauslastung (#Prozesse, Auslastung, Zeit), Speicherauslastung, Kommunikationslast, etc.

Dabei ist die Vergleichbarkeit nicht einfach: Unterschiedl. Prozessoren, andere Systeme

Pull-Metrik: Rechner ziehen Last an

Push-Metrik: Verteiler verteilen Last an wenig beschäftigte Rechner

Broadcast, Teilmenge oder zufällig

Lastwerte, die ein Verteiler mitbekommt sind aufgrund der Laufzeit immer leicht veraltet

Interpolieren und aus Last/Paket lernen

Bei Verteilung 2 Möglichkeiten: Code ist vorhanden -> Ausführen

Code nicht vorhanden -> Code übertragen

Verteilungsverfahren:

Statische Verfahren: Optimale Verteilung vor dem Start ermitteln

Dynamische Verteilung: Ausführungsort für jedes Paket bei Erzeugung ermitteln

Ohne Migration: Lastpaket wechselt Ort nicht

Mit Migration: Lastpaket kann Ort wechseln

Statische Lastverteilung: Als Graph darstellbar und berechenbar

Dynamisch mit Migration: Identische Umgebung auf beiden Systemen notwendig, Adressraum und Prozessstatus übertragen

Dynamisch ohne Migration: Einfach, schnell und gute Ergebnisse. Bei Erstellung von Paket, Zielort bestimmen (gerne auch zufällig)

Verteilte Zeit

Es fehlt eine globale genau gleiche Zeit

Ansätze: Synchronisation der Uhren (z.B. NTP)

Logische Uhren (Lamport, Vektor)

Uhrensynchronisation

1. 1 Rechner hat eine genaue Uhr, daran wird angeglichen
2. Jeder Rechner gleicht sich den anderen Uhren an

Grundannahme: Abweichung linear

DCF77: Zentraler Zeitserver in Braunschweig

Senden per Funk, Reichweite 1500km

F. Christian: Passiver Zeitserver, der auf Anfrage Timestamp sendet

Transportzeit ignoriert

NTP: Hierarchisch, liefert Offset, Roundtrip und Fehlerrate

Stratum 1 (höchste Stufe) = GPS/Atomuhr, auch selbst baubar

Verteilter Abgleich: In Intervallen senden alle Rechner aktuelle Zeit per Broadcast

Jeder berechnet Mittelwert

Logische Uhren

Ereignis: Lokales oder Sende-/Empfangsereignis, vom Entwickler definiert, oder Anweisung

Kausalität: e ist kausal abhängig von f, wenn f Auswirkungen auf e hat

Transitiv, Partielle Ordnung

Logische Uhr: $LC: E \rightarrow H$ mit E: Ereignisse mit Kausalrelation, H: Zeitbereich

Uhrenbedingung: $e_n <_k e_m \rightarrow LC(e_n) < LC(e_m)$

Wenn em kausal abhängig von en ist, dann muss der Zeitstempel von em größer sein, als der von en

Umgekehrt ist das nicht notwendig (umgekehrte Uhrenbedingung)

Lamportzeit:

Jeder Rechner hat eine eigene Logische Uhr (LC)

Lokales Ereignis: LC = LC+1

Sendeereignis: LC = LC+1; Send(Message,LC);

Die aktuelle Uhrzeit nach Erhöhen wird mitgesendet

Empfangsereignis: Receive(Message, LC_s); LC = max(LC, LC_s)+1

Die Uhrzeit wird auf die des Senders gesetzt, wenn diese höher ist, danach wird sie um 1 erhöht

Uhrenbedingung gilt durch Empfangsereignis

Umkehrung der Uhrenbedingung gilt nicht!

$LC(a) < LC(b) \rightarrow (a <_k b) \vee (a||b)$ (|| = unabhängig)

Zähler muss groß genug sein, sonst überlauf (64Bit ist noch genug)

Erweiterte Lamportzeit:

Um totale Ordnung zu erreichen wird dem Zeitstempel noch eine RechnerID angefügt

$$LC_E(A,a) < LC_E(B,b) \leftrightarrow LC(a) < LC(b) \vee (LC(a) = LC(b) \wedge A < B)$$

Vektorzeit:

Feste Menge von Rechnern, Erweiterung allerdings möglich

Zeitstempel ist ein Vektor der Größe n, die Anzahl der Rechner

$$VC(a) = \begin{pmatrix} vc_0 \\ \dots \\ vc_n \end{pmatrix}$$

Initialisiere alle Komponenten auf 0.

Lokales Ereignis auf Rechner k: $vc_k[k] = vc_k[k] + 1$

In der lokalen Uhr wird die lokale Komponente erhöht

Sendeereignis auf Rechner k: $vc_k[k] = vc_k[k] + 1$; Send(Message, VC_k)

Empfangsereignis auf Rechner k:

$vc_k[k] = vc_k[k] + 1$

Receive(Message, vc_{sender})

for(i=0;i<n;i++)

$vc_k[i] = \max(vc_k[i], vc_{sender}[i])$

Kausale Abhängigkeit: $vc(a) \leq vc(b)$

Kausale Unabhängigkeit:

$vc_a||vc_b \rightarrow \exists i,j \in \{0,\dots,n-1\}: (vc_a[i] < vc_b[i]) \wedge (vc_b[j] < vc_a[j]$

Uhrenbedingung und Umkehrung gelten

Wechselseitiger Ausschluss

Zentraler Ansatz:

Server verwaltet Zustände, Clients fragen an (request), Server gibt grant, Client gibt frei (release)

SPOF, nur 3 Nachrichten, asymmetrisch

Token-Ring:

Ein Token wird im Ring herumgereicht, nur der Client mit Token darf den kritischen Abschnitt betreten.

z.T. warten bis Token einmal rum ist, viele bekommen Token obwohl nicht benötigt

Problem bei: Tokenverlust, es darf nur ein neues generiert werden, Prozessabsturz.

Verteilte Warteschlange (Lamport):

Jeder Client verwaltet eine Queue

Ein Client sendet einen Multicast als Request mit Zeitstempel. Jeder andere Client nimmt diesen in seine Queue auf und sendet eine Bestätigung. Erst wenn alle Bestätigungen eingetroffen sind, kann der kritische Abschnitt betreten werden. Die Queue ist nach der

erweiterten Lamportzeit sortiert.
Jeder Client berechnet das Minimum aller eingetroffenen Bestätigungen. Die Queue wird so geteilt, dass vorne die sicheren, mit kleinerere Lamportzeit und hinten die noch abhängigen mit größerem Zeitstempel liegen.
Der erste in der Liste darf den Abschnitt betreten, wenn alle Bestätigungen eingetroffen sind und ein Release kommt.
Annahme: Kommunikation ist zeitlich korrekt, nach Empfang einer Nachricht kann keine mit kleinerer Zeit vom selben Client kommen.
O(3(n-1))

Matrix (Maekawa):

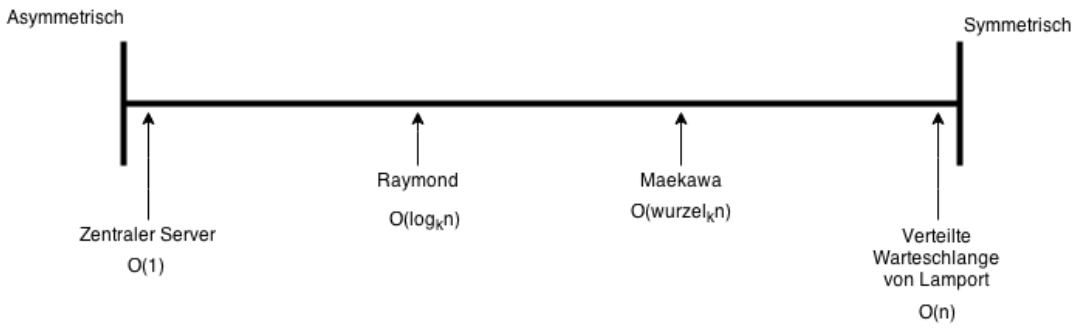
Clients sind als Matrix angeordnet.
Ein Client sende Requests an alle in seiner Reihe und seiner Spalte.
Wenn alle ein Grant gesendet haben, darf der Abschnitt betreten werden.
Deadlock: Zwei Requests -> zwei gemeinsame Knoten. Einer sendet dem einen ein Grant, der andere dem anderen. Beide warten nun auf einen Knoten, der nie Grant senden wird.
Deadlock ist mit Lamport oder Grant-Revoke lösbar.
$O(c * \sqrt{n})$

Baum (Raymond):

Topologie ist ein Baum, die Wurzel ist der Knoten mit dem Token.
Jeder Knoten weiß, in welcher Richtung das Token (die Wurzel) ist.
Ein Client sendet einen Reqeust zum nächsten Knoten in Richtung Wurzel. Dieser leitet den Request weiter. Bekommt ein Knoten zwei Requests, wird als erstes der weitergeleitet, der den kleinen Lamport Zeitstempel hat. Danach der zweite.
Jeder Knoten merkt sich von wem er einen Request bekommen hat, um das Token weiterzuleiten.
Wenn ein Knoten, dessen Request unterwegs ist ein anderen Reqeust bekommt, hält er diesen zurück, bis er das Token ist.
Das Token wandert durch Weitergabe in Richtung des Reqeusts.
$O(\log_k n)$ k=Grad des Baum (Anzahl direkter Kinder eines Knotens)

Symmetrie:

Alle Methoden optimal, da unterschiedliche Symmetrieebene
Raymond: Knoten müssen je nach Position im Baum unterschiedlich viele Requests behandeln. Für k -> n asymmetrischer
Maekawa: k->n symmetrischer
Je symmetrischer desto redundanter, aber hohe Nachrichtenkommunikation
Je asymmetrischer -> SPOF
Asymmetrisch: Server mit eigenenem Code
Symmetrisch: Alle gleich, durch hohe Nachrichtenkomplexität nur in kleinen Systemen
Ziel: Maximal Symmetrisch, Single-System-Image



State Variablen:

Programm hat zwei Arten von Variablen:
lokal: Nur er kann lesen/schreiben
state: Er kann schreiben, alle dürfen lesen.
Einfach umzusetzen mittels Nachrichtenkommunikation
Logik und Implementierung gemischt, unschön

Fehlertoleranz:

Fehlermodelle:

Crash: System-/Prozessausfall
Ommission: Auslassung
Timing: Zu früh/zu spät (Echtzeitfehler/Performancefehler)
Arbitrary: Falsch, aber „gut gemeint“
Byzantinisch: Totalausfall

Byzantinisch:

Zwei Armeen die hinter Hügeln auf zwei Seiten von Byzanz lagern, wollen dieses einnehmen, können es aber nur gemeinsam erreichen. Sie müssen sich also absprechen. Zum absprechen werden Boten geschickt. A schickt B eine Nachricht mit einen weit genug in der Zukunft liegenden Termin. B schickt einen Boten zurück, der die Nachricht bestätigt. Nun weiß B nicht, ob die Nachricht angekommen ist. Es kann unendlich weiter Boten geschickt und Nachrichten bestätigt werden.
Wenn man mit k Ausfällen rechnet, muss man mind. k+1 schicken, also Redundanz.
Weiteres Problem: Nachricht kann kompromittiert werden (abgefangen und verändert). Wenn man k gefälschte Nachrichten kompensieren möchte, muss A 2k+1 Nachrichten abschicken.
Mit Wahrscheinlichkeit p fällt die Komponente k aus. Bei n Komponenten ist die Ausfallwahrscheinlichkeit:$p^n < p$

Redundanz:

Abhängig von Fehlerklasse
k-Zuverlässigkeit: Gleichzeitiger Ausfall von k Teilen toleriert
Replikationsgrad k+1: Crash, Ommission, Timing
Replikationsgrad 2k+1: Arbitrary, Mehrheitsentscheid
Replikationsgrad 3k+1: Arbitrary mit Message authentication

Passive Redundanz: Primary-Backup-Approach

Daten zentral, Backupserver schaltet sich ein, wenn Main nicht mehr geht.
Backupzustände:
Hot-Standby: Jede Operation wird gespiegelt, umschalten dauert nur ms
Warm-Standby: Regelmäßige übernahme des Primary Zustands
Cold-Standby: Übernahme des Zustands nach Ausfall (dauert lange)
Problem: Main manipuliert Daten, obwohl Backup schon läuft
Lösung durch STONITH, Shoot the other node in the head
Bei Ausfall wird per Election ein Backupserver gewählt
Ausfall z.B. durch Heartbeat erkannt

Aktive Redundanz: State-Machine-Approach

Zustandsvariablen, Zustandsändernde Operationen
Replikation durch gleichen Initialzustand und gleiche Taktung
Ensemble: Kollektiv mehrere Replikate von SM
Ausgabe von Kollektiv per Voter (Realisiert im Client, da sonst SPOF)
Hoher Aufwand (Material/Entwicklung)
Determinismus und Atomarität: SM hängt nur von Initialzustand und Abarbeitungsreihenfolge ab

Multicast:

Nachricht an mehrere Empfänge
Gruppenmanagement: Konsistente sichten
Ethernet, Tokenring einfach mit Multicast
In nicht multicast-fähigen Netzwerken: Überflutungsalgorithmen

Zuverlässigkeitsgrad:

- Keiner: Keine Garantie wer und wieviele die Nachricht empfangen
- k-Zuverlässigkeit: Mindestens k Mitglieder erhalten Nachricht
- Atomar: Keiner oder alle empfangen die Nachricht

Ordnungsgrad (Reihenfolge):

- Keine Ordnung
- FIFO: Nachrichten des Senders kommen in FIFO Ordnung an
- Kausale Ordnung: Abhängige Nachrichten kommen später an
- Totale Ordnung: Global geordnet und dementsprechend zugeteilt

Bei höheren Zuverlässigkeits-/Ordungsgraden müssen Nachrichten evtl. durch das Multicast-System zurückgehalten werden

Überflutungsalgorithmen:

- Sende Nachricht an allen ausgehenden Kanten (und weiter so)
- Am Rand entstehen redundante Nachrichten (Reflektion)
- Effizienz in kleinen Gruppen schlechter
- Verbesserung: Knoten speichern Informationen, ob untergeordnete Knoten noch zur Gruppe gehören

Überflutungsalgorithmen mit Rückantwort (Echo):

- Ausdehnungsphase wie Überflutungsalgorithmus
- Kontraktionsphase: Blätter senden spezielle Rückantwort
  - Innere Knoten fassen Rückantworten zusammen und leiten sie in Richtung Sender

Amoeba-Multicast:

- Jede Node hat einen Amoeba-Kern und einen Sequencer, nur genau ein Sequencer ist aktiv
- Wenn Anwendung Multicast senden will und der eigene Sequencer nicht aktiv ist leitet der Amoeba-Kern per Unicast einen Request an den aktiven Sequencer. Der aktive Sequencer sendet in der Reihenfolge der Requests.
- Der Ursprung des Multicasts (der den Request gesendet hat) blockiert ab dem Request, bis zum Empfang des „eigenen“ Multicasts
- Total geordnet, k-Zuverlässig

Multicast-Gruppen:

- IGMP: Verwaltung von IP Abonnenten

RPC

Client sendet Funktionsaufruf an Server, der führt aus und sendet die Antwort zurück.

Serveraufbau:

- Verlagerung Kernfunktionen in Mikro-Prozesse
- Ein Prozess = Ein Dienst

Kommunikationsvarianten:

- Kommunikationsserver, eigener Dienst
  - Erweiterbar, einfacher Systemkern
  - Mehr Kontextwechsel, Latenz
- Kernkommunikation:
  - Minimale Verzögerung, schnell
  - Komplexer Kern

Marshalling:

Viele verschiedene Datentypen/Speicherung der Datentypen

Lösung: Standards (JSON, XML etc.)

RPC-Compiler:

Interface in besonderer Sprache beschrieben.

Compiler erstellt daraus Client und Server stubs

Serverlokalisierung:

Eingabe der Adresse im RPC Aufruf oder während Runtime

Benutzung eines Portmanagers:

Verwaltet RPC Server und deren verwendete Ports

Probleme:

RPC ist noch nicht = LPC

Transparenz nur schwer erreichbar

Ausfälle behandeln?

Performance!

Client/Server nicht immer trennbar (Client wird zum Server etc.)

Ausführungssemantiken:

Exactly-once: Externer Aufruf findet genau einmal statt

Realität: Nachrichtenverlust, Serverabsturz...

At-least-once: Externer Aufruf mindestens einmal ausgeführt

At-most-once: Maximal einmal ausgeführt, kann also auch nicht ausgeführt werden

Fehlerfälle:

Nachrichtenverlust (im Rahmen eines Timeouts)

Nachrichtenwiederholung, dadurch mehrfacher Empfang

Zustandsverlust durch Serverfehler

At-least-once: Ignoriere mehrfache Replies

At-most-once: Schwer herauszufinden, ob es ausgeführt wurde

Zustandsbehaftete Server:

Server speichert Zustand lokal (Auftragsfolgen, Verbindungsinformationen, Liste der ausgeführten Aufträge)

Zustandslose Server:

Client speichert Zustand (auch in verschlüsselter Form)

Server sendet Zustand zurück an Client, bei weiteren Aufträgen sendet Client erst den Zustand mit an den Server

Große Nachrichten

Idempotenz:

Der Aufruf einer Funktion mit den selben Parametern liefert immer das selbe Ergebnis.

Viele Funktionen sind von sich aus idempotent.

Nicht-idempotente Funktionen können durch Festlegen des Startzustanden auch idempotent werden.

Verteilte Dateisysteme

Ram des Nachbarn schneller als Festplatte

Update in place /Log:

In Place: Spurwechsel, Aufzugalgorithmen

Log: Schreiben erst nach Erreichen bestimmter Menge

Vorteile durch Verteilung:

Leistungssteigerung, Mobilität, Redundanz, Lastverteilung, Availability

Transparenz:

Ziel: Single-System-Image
Zugangstransparenz: Identische Programmierschnittstellen
Ortstransparenz: Dateiort nicht ermittelbar (physisch)
Namenstransparenz: Namen überall gleich
Migrationstransparenz: Name bleibt auch bei Migration gleich
Leistungstransparenz: Zugriffsgeschwindigkeit gleich
Fehlertransparenz: Verteilte Fehler werden maskiert

Caching:

Serverseitig: Konsistent aber hohe Netzwerklast
Clientseitig: Geringe Netzwerklast aber achtung Inkonsistenzen
Lösung durch festlegen des Schreibzeitpunktes:
Write-Through: sofort schreiben
Delayed: Zeitfenster für Konflikte
Write on close

Zustandslos vs Zustandsbehaftet:

Zustandslos: Fehlertoleranz einfach, Öffnen und Schließen von Dateien überflüssig, Client-Abstürze egal, höhere Latenz
Zustandsbehaftet: Read-Ahead möglich, Idempotenz leichter erreichbar, sperren von Dateien, höhere Leistung

NFS:

Zugangstransparent, Ortstransparent, Fehlertransparent, (Leistungstransparent)
Zustandslose Server
Bei Datenzugriff wird ein Dateidescriptor mitgegeben. Server speichert Descriptoren

AFS – Andrew File System:

Server sind vertrauenswürdig, Clients cachen
Transparenter Verzeichnisbaum /afs

CODA:

Erlaubt Netzwerkpartitionierung
Optimistische Replikation: Arbeiten auf lokalem Cache
Konfliktlösung nötig
Volume Storage Group: Speichern gleiche Daten
Disconnected Operation: Horde Daten (Hoarding)
Emuliere Serverrolle
Reintegriere Daten beim Wiedereintritt ins Netzwerk
Surrogate: Proxy, sendet Replays an Server, die die Veränderungen vornehmen

ZEBRA:

Großes sequentielles Logfile.
Regelmäßig durch Stripe Cleaner aufgeräumt (etwa Defragmentiert)
Zentraler File Manager inklusive Stripe Cleaner

P2P (Oceanstore):

Überall verfügbar, Peers kommen online und offline

Tapestry (Namensgebung):

Verwendung von Distributed Hash Tables
Netzwerk ist ein Ring
Zugriff auf Datei f:
Wenn hash(f) ungleich ID vom aktuellen Rechner -> leite in die richtige Richtung weiter
Wenn sich der Hashwert stark unterscheidet können Abkürzungen im Ring genommen werden

Verteilte Terminierung

Kommunikationsorientert:

Problem bei Asynchronen Vorgängen z.B. Interrupts
2 Arten von Nachrichten: Kontrollnachrichten und Basisnachrichten
Kontrollnachrichten machen passive Node nicht aktiv (**Actor Modell**)
Fertig wenn alle passiv, keine Basisnachrichten unterwegs
Zähle gesendete und empfangene Nachrichten im ganzen System
gesendet = empfangen -> fertig wenn alle passiv
Startzustand durch initiale Zählerwerte bedingt
Koordinator fragt Zustand der nodes ab.

Ergebnisorientiert:

Zustand des Gesamtsystems ist erfüllt

Doppelzählverfahren (Koordinator):

Zwei Abfragen, zweite beginnt, wenn alle Nachrichten der ersten Welle eingetroffen sind

Vektormethode (Koordinator):

Jeder Prozess besitzt einen Vektor der Größe n der beteiligten Nodes
Startwert alles 0
Gespeichert werden alle Basisnachrichten mit diesem Knoten:
V(x) ++ bei senden zu x
V(x) -- bei empfangen von x
Koordinator addiert alle Vektoren
Terminierung = Nullvektor

Election

Anhand eines Maßes (min/max): Problem: Wann sind alle Nachrichten empfangen?

CAP Theorem

CAP-Theorem: Wünschenswerte Eigenschaften verteilter Systeme, es können immer nur 2 davon gleichzeitig gelten.

C: Consistency. Alle Knoten sehen die gleichen Daten. Ein Lesezugriff sieht alle vorhergegangenen Schreibezugriffe.

A: Availability. Alle Anfragen an ein System werden beantwortet. Jeder Knoten kann immer Lese- und Schreibzugriffe ausführen.

P: Partition Tolerance. Das System lässt sich teilen. Auch wenn Teile ausfallen sollten, funktionieren die restlichen noch.