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

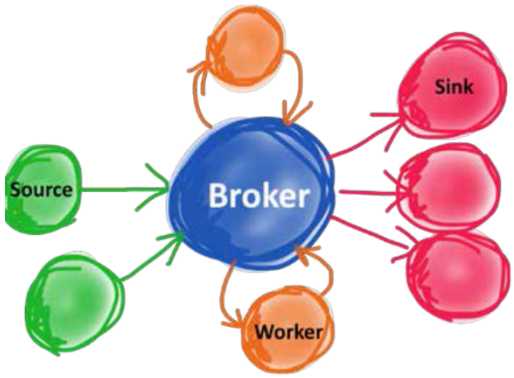
UDP

Best effort, max 64kb

ZMQ

Message Protocol: Daten werden als Nachrichten versendet
Dabei bietet ZMQ ein hohes Abstraktionslevel

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

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):

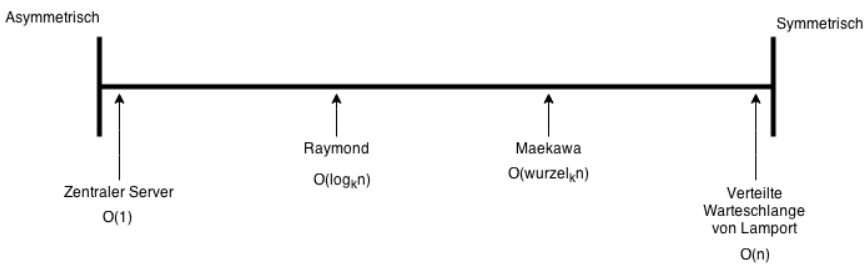
Clients sind als Matrix angeordnet.
Ein Client sende Requests an alle in seiner Reihe und seiner Spalte.
Wenn alle ein Grant gesendet haben, darf der Abschnitt betreten werden.
Deadlock: Zwei Requests -> zwei gemeinsame Knoten. Einer sendet dem einen ein Grant, der andere dem anderen. Beide warten nun 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



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
- 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.
Fehlerfall: Ein Bote fällt aus.
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.

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: Daten zentral, Backupserver schaltet sich ein, wenn Main nicht mehr geht. Problem: Main manipuliert Daten, obwohl Backup schon läuft

- 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