Distributed Systems
Zusammenfassung

Frédéric Vogel

ETH Zürich, HS14

Contents

1 Teil 1

1.1 Definition und Historie

> A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network.
>
> H. Bal

1.2 Architekturen verteilter Systeme

1.2.1 Peer-to-Peer (P2P)

- Jeder Rechner gleichzeitig Informationsanbieter und -konsument

1.2.2 Client-Server

- Server als Informationsanbieter (reagierender Prozess)
- Client als Konsument (initiierender Prozess)
- Client gleichzeitig Benutzungsschnittstelle
- WWW-dominiertes Internet

1.2.3 3-Tier

- Verarbeitung wird auf mehrere physikalische Einheiten verteilt
- Logische Schichten mit minimierten Abhängigkeiten
- Leichtere Wartung, einfaches Austauschen

1.2.4 Multi-Tier

- Weitere Schichten, mehrere physikalische Einheiten pro Schicht → erhöht Skalierbarkeit und Flexibilität
- Mehrere Server ermöglichen Lastverteilung
- Verteilte Datenbanken bietet Sicherheit (Replikation, hoher Durchsatz)

1.2.5 Compute-Cluster

- Vernetzung kompletter Einzelrechner
- Räumlich konzentriert (wenige Meter)
- Sehr schnelles Verbindungsnetz
- Diverse Netztopologien, sehr unterschiedlich hinsichtlich
  - ∞ Skalierbarktei
  - ∞ Routingkomplexität
  - ∞ usw.

1.2.6 Service-Oriented Architecture (SOA)

- Unterteilung der Applikation in einzelne, unabhängige Abläufe innerhalb eines Geschäftsprozess → erhöht Flexibilität
- Lose Koppelung zwischen Services über Nachrichten und Events
- "Development by composition": Services können bei Änderungen der Prozesse einfach neu zusammengestellt werden
- Services können von externen Anbietern bezogen werden
- Oft in Zusammenhang mit Web-Services

1.2.7 Cloud-Computing

- Massive Bündelung der Rechenleistung an zentraler Stelle
- Outsourcen von Applikationen in die Cloud
- Internet nur noch als Vermittlungsinstanz

1.3 Charakteristika & Phänomene

- Neue Probleme durch räumliche Separation und Autonomie der Komponenten:
  - ∞ partielles Fehlverhalten möglich (statt totaler Absturz)
  - ∞ fehlender globaler Zustand / exakt synchronisierte Zeit
  - ∞ mögliche Inkonsistenzen (z.B. zwischen Datei und Verzeichnis/Index)
- Heterogenität in Hard- und Software
- Hohe Komplexität
- Sicherheit notwendiger aber schwieriger

1.3.1 Beispiele konzeptioneller Probleme

Schnappschussproblem Wie viel Geld ist im Umlauf?

- Ständige Transfers
- Keine globale Sicht
- Keine gemeinsame Zeit

Deadlock Zyklische Wartebedingung

Uhrensynchronisation Uhren gehen nicht gleich schnell, keine globale Zeit

Kausaltreue Beobachtungen Gewünscht: Ursache stets vor ihrer (u.U. indirekten) Wirkung beobachten

Verteile Geheimnisvereinbarung Einigung eines gemeinsames geheimes Passwort über unsicheren Kanälen

1.4 Kommunikation — Nachrichten

Prozesse sollen kooperieren, daher untereinander Information austauschen können.

- Globaler Speicher (physisch oder virtuell)
- Nachrichten

1.4.1 Nachrichtenbasierte Kommunikation

- Send → Receive
- Implizierte Synchronisation, Senden vor Empfangen

1.4.2 Ordungserhalt von Nachrichten

FIFO First In, First Out
Empfangsreihenfolge = Sendereihenfolge (zwischen zwei Prozessen)

Kausale Ordnung Keine Information erreicht Empfänger auf Umwegen schneller als auf direktem Wege
Globalisierung von FIFO auf mehrere Prozesse

1.4.3 Fehlermodelle

Nachrichtenfehler beim Senden/Übertragen/Empfangen
Verlorene Nachricht

Crash / Fail-Stop: Ausfall eines Prozessors
Nicht mehr erreichbarer/mitspielender Prozess

Zeitfehler
Ereignis geschieht zu spät oder zu früh

Byzantinische Fehler
Beliebiges Fehlverhalten, z.B.:
- Verfälschte Nachrichteninhalte
- Prozess, der unsinnige Nachrichten sendet

1.4.4 Mitteilungsorientierte Kommunikation

- Einfachste Form der Kommunikation
- Unidirektional

1.4.5 Auftragsorientierte Kommunikation

- Bidirektional
- Ergebnis des Auftags wird als Antwortnachricht zurückgeschickt

1.5 Kommunikation — synchron/asynchron

Blocking send
Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert
Sender hat Garantie Nachricht wurde zugestell/empfangen

Synchrone Kommunikation
Send und receive geschehen (im Prinzip) gleichzeitig

Virtuelle Gleichzeitigkeit
Bei Abstraktion von Realzeit ist ein Ablauf durch ein äquivalentes Zeitdiagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen
Nur stetige Deformation erlaubt (verschieben auf Zeitachse, nicht kreuzen)

Asynchrone Kommunikation / No-wait send
Send und receive nicht gleichzeitig
Sender nur bis zur lokalen Ablieferung der Nachricht an das Transportsystem blockiert

1.5.1 Asynchrone $\leftrightarrow$ synchrone Kommunikation

Vorteile asynchroner Kommunikation
- sendender Prozess kann weiterarbeiten während Nachricht übertragen wird
- stärkere Entkopplung von Sender und Empfänger
- höherer Grad an Parallelität möglich
- geringere Gefahr von Deadlocks

Nachteile
- Sender weiss nicht, ob/wann Nachricht angekommen ist
- Debugging der Anwendung oft schwierig
- System muss Puffer verwalten

1.5.2 Synchron $\stackrel{?}{=}$ blockierend

Blockierung Rein senderseitiger Aspekt

blockierend Sender wartet, bis Nachricht lokal vom Kommunikationssystem abgenommen wurde
nicht-blockierend Sender informiert Kommunikationssystem, dass & wo zu versendende Nachricht ist

Synchron/asynchron Nimmt Bezug auf Empfänger

synchron Nach Ende der Send-Operation wurde Nachricht dem Emfpänger zugestellt
asynchron Nicht garantiert

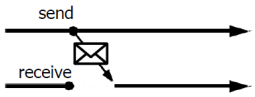
1.5.3 Hauptklassifikation von Kommunikationsmechanismen

	asynchron	synchron
Mitteilung	no-wait send	Rendezvous
Auftrag	asynchroner RPC	Remote Procedure Call (RPC)

Häufigste Kombination: Mitteilung asynchron, Auftrag synchron

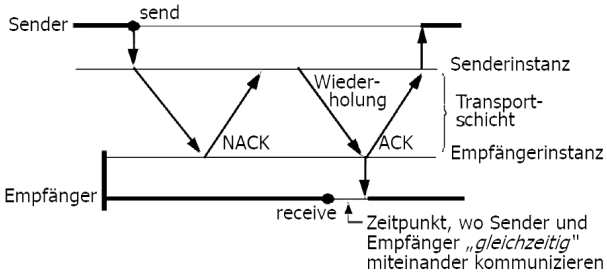
No-wait send

- $\oplus$ weitgehende zeitliche Entkopplung von Sender und Empfänger
- $\oplus$ einfache, effiziente Implementierung (bei kurzen Nachrichten)
- $\ominus$ keine Erfolgsgarantie für Sender
- $\ominus$ Notwendigkeit der Zeishcenpufferung
- $\ominus$ Gefahr des "Überrennens" des Empfängers bei zu häufigen Nachrichten



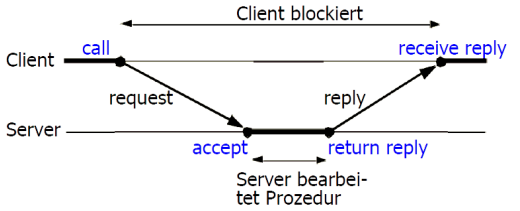
Rendezvous-Protokolle

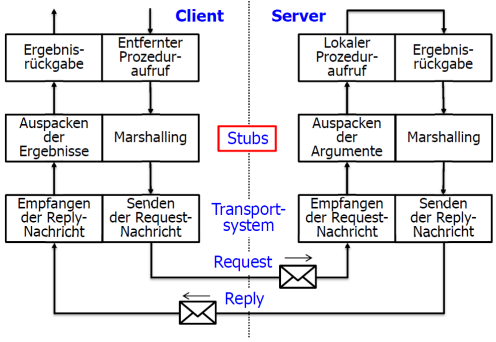
- Der erste wartet auf den anderen (Synchronisationspunkt)
- Mit NACK/ACK wenig Puffer nötig aber aufwändiges Protokoll (busy waiting)



1.6 Kommunikation — RPC

- Aufruf einer entfernten Prozedur
- Soll klassischem Prozeduraufruf möglich gleichen (klare Semantik für Anwender)
- Einfaches Programmieren
  - ∞ kein Erstellen von Nachricht, kein Quittieren, etc. auf Anwendungsebene
  - ∞ Syntax analog zu lokalem Prozeduraufruf
  - ∞ Typsicherheit (Datentypprüfung auf Client- und Serverseite möglich)





1.6.1 RPC: Stubs

`call S.X(out: a; in: b);`

wird ersetzt durch längeres Programmstück (Stub), welches:

- Parameter a in Nachricht packt
- Nachricht an Server S sendet
- Timeout für Antwort setzt
- Antwort entgegennimmt
- Erreignisparameter b mit Werten der Antwortnachricht setzt

Stubs

- wirken als proxy (lokale Stellvertreter des entfernten Gegenüber
- simulieren einen lokalen Aufruf
- sorgen für Zusammenbau und Entpacken von Nachrichten
- konvertieren Datenrepräsentationen
- können weitgehend automatisch generiert werden
- steuern das Übertragungsprotokoll

1.6.2 Probleme mit RPC

Soll möglichst aussehen wie lokaler Prozeduraufruf, doch

- Server kann unerreichbar/abgestürtzt sein
- RPCs dauern länger
- Anwender hat keine Kontrolle über Server
- Ungewisse, variable Verzögerungen
- Keine Kommunikation über globale Variablen
- Keine Pointer/Referenzparameter als Parameter möglich
- mehr Fehlerfälle (Server-/Client-Absturz, Nachrichtenverlust, etc.)

Verlorene Request-Nachricht

Gegenmassnahme
Nach Timeout Request-Nachricht erneut senden

Probleme

- Wie viele Wiederholungsversuche maximal?
- Wie gross soll Timeout sein?
- Was wenn Nachricht nicht verloren sondern untypisch langsam?

Probleme wenn Nachricht tatsächlich nicht verloren

- Doppelte Request-Nachricht (gefährlich bei nicht-idempotente serverseitige Operation)
- Server sollte solche Duplikate erkennen

Verlorene Reply-Nachricht

Gegenmassnahme 1
Analog zu verlorener Request-Nachricht

Probleme

- Vielleicht ging Request verloren?
- Server war langsam und arbeitet noch?
- Nicht unterscheidbar aus Client-Sicht

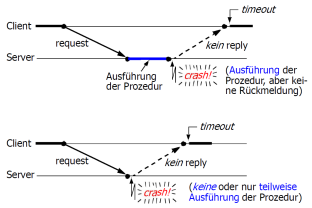
Gegenmassnahme 2
Server hält Historie versendeter Replies

- Falls Duplikat erkannt wird und Auftrag bereits ausgeführt: letztes Reply erneut senden
- Pro Client nur letztes Reply speichern
- Nach gewisser Zeit Historie löschen

Server-Crash

Probleme

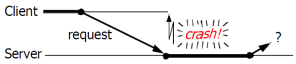
- Wie soll Client untere Fälle unterscheiden?
- Client meint evtl. zu Unrecht, dass Auftrag nicht ausgeführt wurde



Client-Crash

Probleme

- Reply des Servers wird nicht abgenommen → blockiert Ressourcen bei Server
- Orphans beim Server
- Server fragt ab und zu ob Client noch an Antwort interessiert (bzw. ob Client existiert)



1.6.3 RPC-Fehlersemantik-Klassen

Maybe-Semantik

- Keine Wiederholung von Requests
- Einfach und effizient
- Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
- Mögliche Anwendungen: Auskunftsdienste

At-least-once-Semantik

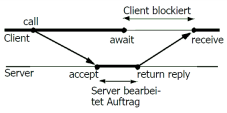
- Automatische Wiederholung von Requests
- Keine Duplikatserkennung (zustandsloses Protokoll auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

At-most-once-Semantik

- Erkennen von Duplikaten (Sequenznummern, log-Datei, etc.)
- Keine wiederholte Ausführung der Prozedur
- Geeignet auch für nicht-idempotente Operationen

1.6.4 Asynchroner RPC

- Auftragsorientiert → Antwortverpflichtung
- Parallelverarbeitung von Client und Server möglich (solange Client nicht auf Resultat angewiesen)



1.7 Client/Server

Client
Konsument, initiierender Prozess, Benutzungsschnittstelle

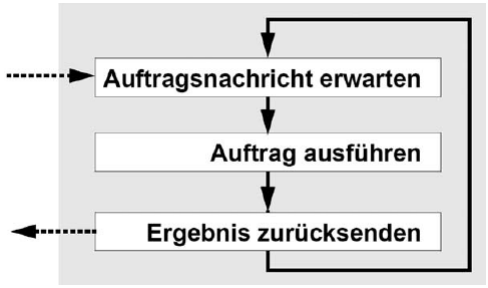
Server
Informationsanbieter, reagierender Prozess

1.7.1 Gleichzeitige Server-Aufträge

Problem: Oft viele gleichzeitige Aufträge

Iterativer Server: Bearbeitet nur einen einzigen Auftrag pro Zeit

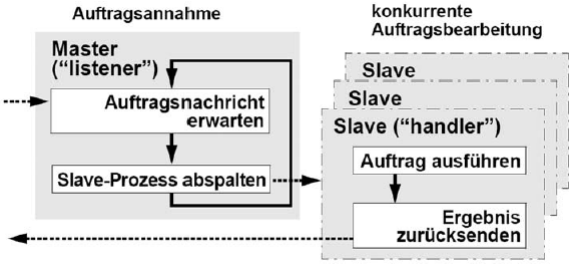
- einfach zu realisieren
- eintreffende Anfragen puffern, abweisen oder ignorieren
- sinnvoll bei trivialen Diensten mit kurzer Bearbeitungszeit



Konkurrent-Server: Quasi-gleichzeitige Bearbeitung mehrerer Aufträge
Sinnvoll bei längeren Aufträgen

Konkurrente Server mit dynamischen Handler-Prozessen: Master gründet Slave-Prozess für jeden Auftrag

- Neu gegründeter Slave übernimmt den Auftrag
- Client kommuniziert direkt mit Slave
- Slaves typischerweise Leichtgewichtprozesse
- Slaves terminieren nach Auftrag
- Anzahl gleichzeitiger Slaves begrenzt



1.7.2 Stateless/statefull Server

- Hält Server Zustandsinformation über Aufträge hinweg?
- Bei zustandslosen Servern muss jeder Auftrag vollständig beschrieben sein (Position Dateizeiger, etc.)
- Dateisperren bei zustandslosen Servern nicht einfach möglich
- Zustandsbehaftete Server im Allg. effizienter
- Zustandsbehaftete Server können wiederholte Aufträge erkennen → Idempotenz
- Crash eines Servers gibt weniger Probleme im zustandslosen Fall

1.7.3 Wiedererkennung von Kunden

URL Rewriting und dynamische Websites

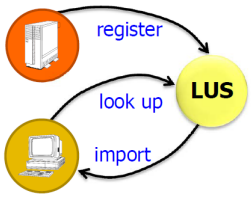
- der Einstiegsseite eindeutige Identität anheften wenn Kunde erstmals Seite aufruft
- Identität jedem Link auf Seite anheften und mit übertragen
- Gefahr: Identität klauen durch wissen der URL

Cookie als Context-Handle

- Textdatei, die Server an Browser schickt und dort gespeichert wird
- Server kann Cookie später wieder lesen und so Kunde erkennen

1.7.4 Lookup-Service

- Wie finden sich Client und Server → Lookup-Service (LUS)
- Server macht Service LUS bekannt
  - ∞ register: RPC-Schnittstelle exportieren (Name, Parameter, Typen, ...)
  - ∞ evtl. später wieder abmelden
- Client erfragt bei LUS Adresse von geeignetem Server
  - ∞ Angabe des gewünschten Typs von Service bei lookup
  - ∞ importieren der RPC-Schnittstelle



Vorteile

- mehrere Server für gleichen Server registrieren
- Autorisierung etc. überprüfen
- duroch Polling der Server Verfügbarkeit eines Services testen
- verschiedene Versionen eines Dienstes verwalten

Probleme

- lookup kostet Ausführungszeit
- zentraler LUS ist potentieller Engpass

1.8 Web Services, Middleware

Übersprungen

1.9 REST

Übersprungen

1.10 Jini

Übersprungen

1.11 Broadcast/Multicast

Broadcast
Senden an die Gesamtheit aller Teilnehmer

Multicast
Senden an eine Untergruppe aller Teilnehmer (Broadcast bezogen auf Untergruppe)

1.11.1 "Best Effort" Broadcast

- Euphemistische Bezeichnung da keine extra Anstrengung (einfache Realisierung ohne ACK etc.)
- Keinerlei Garantien: unbestimmt wie viele / welche Empfänger erreicht wurden, unbestimmte Empfangsreihenfolge
- Im Erfolgsfall effizient
- Geeignet für Verbreitung unkritischer Informationen

1.11.2 "Reliable" Broadcast

- Ziel: Unter gewissen Fehlermodellen "möglichst zuverlässigen" Broadcast-Dienst realisieren
- Idee: ACK für jede Einzelnachricht
  ∞ viele ACKs → Belastung des Servers → schlechte Skalierbarkeit
- Idee: NACKs (negative ACKs):
  ∞ Alle Broadcasts werden nummeriert → Lücke wird erkannt beim Empfänger
  ∞ NACK bzgl. fehlender Nachricht wird gesendet
  ∞ Fehlende Nachricht wird nachgeliefert

1.11.3 Broadcast: Empfangsreihenfolge

FIFO-Broadcast
Alle Broadcasts ein und des selben Selbers an eine Gruppe kommen bei allen Mitgliedern in FIFO-Reihenfolge an

Kausale Nachrichtenabhängigkeit
Von links nach rechts verlaufender Pfad von X nach $Y \rightarrow Y$ hängt kausal von X ab

Kausaler Broadcast
N hängt kausal von M ab, Prozess P empfängt M und $N \rightarrow P$ muss M vor N empfangen

Atomarer Broadcast
Prozesse P_i und P_j empfangen beide M und N. P_i empfängt M vor N genau dann, wenn P_j M vor N empfängt

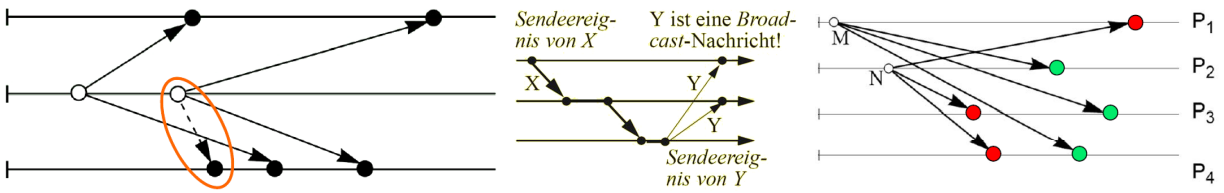


Figure 1: FIFO, Kausal, Atomar

Eigenschaften:
- Atomar $\not\Rightarrow$ kausal
- Atomar $\not\Rightarrow$ FIFO
- Atomar + FIFO $\not\Rightarrow$ kausal

1.12 Logische Zeit

Definition Kausalrelation $\prec$
x $\prec$ y genau dann, wenn:
1. x und y auf dem gleichen Prozess stattfinden und x vor (links von) y kommt, oder
2. x ist ein Sendeereignis und y ein korrespondierendes Empfangsereignis, oder
3. $\exists$ z mit x $\prec$ z $\wedge$ z $\prec$ y

Definition C(e) (Zeitstempel)
Wenn C(e) < C(e') dann nennt man e früher als e'

Uhrenbedingung
e $\prec$ e' $\Rightarrow$ C(e) < C(e') (Zeit ist kausaltreu)

1.12.1 Lamport-Uhren

- Bei jedem Ereignis tickt lokale Uhr (=Zähler) des Prozesses
- Senden: aktuellen Uhrwert mitsenden (Zeitstempel der Nachricht)
- Empfangen: Uhr = max(Uhr, Zeitstempel der Nachricht) (zuerst max(), anschliessend ticken)

Injektive Variante
Lexikographische Ordnung (C(e),i), i ist Prozessnummer auf dem e stattfindet
(a,b) < (a',b') $\Leftrightarrow$ a < a' $\vee$ a = a' $\wedge$ b < b'

1.13 Wechselseitiger Ausschluss

1.13.1 Anforderungen

Safety Nothing bad will ever happen

Liveness Something good will eventually happen

Fairness Jeder kommt mal zum Zug

1.13.2 Zentraler Manager

- Manager ordnet Ressourcen exklusiv aber fair zu
- Problem: Single point of failure

1.13.3 Warteschlange

Globale Warteschlange

- Zentraler Manager-Prozess hält zeitlich geordnete FIFO Warteschlange von Requests
- Problem: Single point of failure

Replizierte Warteschlange

- Warteschlange bei jedem Prozess replizieren
- Alle Prozesse sollen gleiche Sicht haben
- Konsistenz wird mit Nachrichten und logischer Zeit erreicht
- Voraussetzungen: FIFO-Kommunikationskanäle, eindeutige Zeitstempel, FIFO-Broadcast, Request werden geACKt

1.13.4 Lamport Algorithmus

1. Bewerbung um BM: Request mit Zeitstempel und Absender an alle senden und in eigene Queue einfügen
2. Empfang eines Requests: Request in eigene Queue einfügen, ACK versenden
3. Freigabe des BM: Aus eigener Queue entfernen und Release an alle senden
4. Emfpang eines Releases: Zugehörigen Request aus eigener Queue entfernen

Ein Prozess darf BM nuten wenn:

a. Eigener Request ist frühester in seiner Queue

b. Von jedem anderen Prozess irgendeine spätere Nachricht bekommen

1.14 Sicherheit

Übersprungen