Verteilte Systeme

Erstellt von Christoph Redl

Inhalt

Inhalt 1
Einführung 4
Definitionen 4
Ziele von verteilten Systemen 4
Transparenzarten 5
Arten von verteilten Systemen 5
VS mit Betonung auf der Verteilung der Rechenkapazität 5
VS mit Betonung auf der Verteilung der Daten 5
Distribted Pervasive Systems 6
Architekturen 7
Typen 7
Communication System 7
Middleware 8
Prozesse, Threads, Clients und Server 9
Prozesse und Threads 9
Multithreaded Clients 9
Multithreaded Server 9
Binding 10
Remote Procedure Call 10
Remote Method Invocation 11
Code Migration 12
Virtualisierung 12
Message Orientated Communication 13
Stream-orientated Communication 14
Naming 15
Definition 15
Flache Namen 16
Hierarchische Namen 16
Definition 16

DNS....16
Attribute-based Naming....17
Discovery Services....18
JNDI....18
Distributed Garbage Collection....18
Synchronisation....20
Definition....20
Zeitdefinition....20
Synchronisation mit einem externen Zeitgeber....20
Interne Synchronisation....21
Logische Synchronisation....21
Mutal Exclusion....22
Election Algorithms....23
Global State....23
Replikation und Konsistenz....25
Definition....25
Arten von Replikaten....25
Konsistenzmodelle....25
Datenzentrierte Konsistenzmodelle....25
Clientzentrierte Konsistenzmodelle....26
Abgleich von Replikaten....26
Konsistenzprotokolle....27
Fault Tolerance....29
Begriffe....29
Erreichen von Fehlertoleranz....29
Redundanzen....30
Verlässliche Client/Server-Kommunikation....31
Reliable Multicast....32
Ordnungen von Multicast-Nachrichten....33
Distributed object-based Systems....34
Definition....34
Beispiele....34
Komponenten....35
Object-based Messaging....35
Dokumentenbasierte Distributed Systems....36

Distributed coordination-based Systems 36
Definition 36
Beispiele 37
Quellen 38

Einführung

Definitionen

Verteiltes System

Ein aus einzelnen, unabhängigen und miteinander verbundenen Computern bestehendes System, das aus Benutzersicht wie ein einziges kohärentes System aussieht.

Middleware

Eine Softwareschicht zwischen Betriebssystem und Anwendungsprogrammen bzw. Usern, die nach oben einheitliche Schnittstellen anbieten. Die Heterogenität der darunter liegenden Systeme, der Hardware und der Netzwerktopologien werden verborgen, ebenso die eventuelle Verteilung einer Anwendung. Die Middleware bietet Services an, die von vielen verschiedenen Anwendungen in gleicher oder ähnlicher (parametrisierbarer) Weise benötigt werden. Sie verhindert somit dass gleiches immer wieder neu implementiert werden muss.

Ziele von verteilten Systemen

Ressourcen leicht zugänglich machen um diese zu sharen, oder Daten auszutauschen. Das wird zum Beispiel mittels Naming und Discovery erreicht.

Verteilungstransparenz: Vor dem Benutzer wird verborgen, dass es sich um ein verteiltes System handelt.

Openness:

VS sollten sich an Standards (anerkannte Definition von Schnittstellen und Formaten) orientieren. Dies unterstützt die *Interoperabilität* (Zusammenschluss von unterschiedlichen Herstellern stammender Systeme) sowie *Portabilität* (Austauschbarkeit einzelner Komponenten). Für die *Interoperabilität* ist die Vollständigkeit der Standards besonders wichtig (weil sonst herstellerspezifische Erweiterungen entstehen), für die *Portabilität* die Neutralität der Standards (sie sollen nur das Was, aber nicht das Wie spezifizieren).

Saubere Schnittstellen sollten nicht nur zum User bzw. zur Applikation hin bestehen, sondern auch innerhalb des Systems (Komponententrennung), um z.B. das Einhängen von Modulen zu ermöglichen.

Nicht erwünscht ist Openness manchmal aus Kostengründen oder aus wegen dem verwendeten Geschäftsmodell.

Skalierbarkeit:

Beschreibt, wie sich das System verhält wenn es wächst (z.B. ob signifikant langsamer wird) und wie gut ein solches Wachstum praktisch möglich ist. Es gibt mehrere Dimensionen.

User- bzw. Ressourcenanzahl (size): Ist aufgrund von zentralisierter Rechenleistung für Services, zentralisierten Daten oder zentralisierten Algorithmen oft schwierig.

Geographisch: Probleme sind ebenfalls zentralisierte Lösungen sowie Systeme, die für LAN-Bedingungen (synchrone Kommunikation, verlässliche Netzwerkverbindungen, Broadcasting) ausgelegt sind, die in WANs nicht gegeben sind.

Administrativ: Wenn sich VS plötzlich über mehrere Organisationen erstrecken kann es wegen den unterschiedlichen Richtlinien zur Sicherheit, zur Administration und zur Abrechnung zu Problemen kommen. Diese müssen abgegelichen werden.

Lösungen: Übertragungszeiten verbergen (z.B. asynchrone Kommunikation), Distribution auf mehrere Rechner (z.B. DNS) und Replikation um Last zu verteilen(z.B. Caching).

Nicht funktionale Anforderungen

Dazu zählen zum Beispiel Security, Fault Tolerance, Dependability, Quality of Service, usw.

Transparenzarten

Verteilungstransparenz lässt sich unterteilen in folgende Transparenzarten:

access: Zugriff ohne Rücksicht auf Darstellungsunterschiede (Big/Little Endian, ...)

location: physische Ressourcenplatzierung

migration: physische Verschiebung einer Ressource

relocation: physische Verschiebung einer Ressource während diese in Betrieb ist

replication: mehrere Kopien von Ressourcen, die der User als eine Ressource wahrnimmt

concurrency: Verbergen des gleichzeitigen Zugriffs durch mehrere User

failure: Ausfall einzelner Komponenten wird vom Benutzer nicht gesehen

Transparenz ist nicht immer anstrebenswert, z.B. aus Kostengründen oder weil sie ganz einfach aus Benutzersicht nicht wünschenswert ist (z.B. bei embedded Systems will man z.T. ganz bewusst auf die aktuelle Location eingehen).

Arten von verteilten Systemen

VS mit Betonung auf der Verteilung der Rechenkapazität

Cluster: Mehrere sehr ähnliche Rechner innerhalb einer Organisation bilden eine Art Superrechner

Grid computing: Die einzelnen Rechner können sehr unterschiedlich sein und über ein WAN verbunden sein (sich also in unterschiedlichen Organisationen befinden). Die Benutzer, die Zugriff auf das VS haben, befinden sich in einer virtuellen Organisation.

VS mit Betonung auf der Verteilung der Daten

Transaction Processing Systems: Basieren darauf, dass Folgen von Operationen entweder ganz oder gar nicht ausgeführt werden (z.B.*Transactional RPC*). Auch die anderen ACID-Eigenschaften gelten. In

VS besonders wichtig sind Subtransaktionen, die auf verschiedenen Computern durchgeführt werden können. Sie werden von einem TP Monitor verwaltet.

Enterprise Application Integration: Das sind VS, bei denen die Anwendungen direkt (und nicht nur über die DB) miteinander kommunizieren können.

Distribted Pervasive Systems

Das sind VS, bei denen ständig Hosts eingefügt und entfernt werden, die relativ klein sind und zum Teil Mobilgeräte darstellen. Ausfälle sind dann nicht die Ausnahme, sondern die Regel.

Voraussetzungen für solche Systeme sind, dass die Geräte ständig auf Änderungen der Umgebungsbedingungen vorbereitet sind, dass sie sich automatisch konfigurieren und dass *sharing* zum Standard wird.

Architekturen

Typen

Grundsätzlich kann ein verteiltes System nach folgenden Typen einteilen:

- Layered
- Object-based
- Event-based
- Shared Dataspace

Communication System

Auf unterster Ebene (im sogenannten Communication System – OSI Layer 1 – 4) findet jedoch immer ein Message Passing statt. Alle anderen Typen werden auf Basis dieses Mechanismus realisiert, wobei die Middleware die Umsetzung vornimmt.

Um zu vermeiden dass jede Entität (ein allgemeiner Begriff für den Endpunkt einer Verbindung) mit jeder anderen verbunden werden muss (any-to-any-Topologie), was nicht nur hohe Kosten sondern auch unnötig komplexe Hardware zur Folge hätte, werden stattdessen Leitungen per time-division-Multiplexing genutzt. Dies kann entweder synchron (circuit switched) oder asynchron (packet switched) erfolgen. Circuit switched (so wie beim Telefon) hat den Vorteil, dass die Verbindung zuerst fix durchgeschaltet wird und daher ein Routing einzelner Pakete nicht notwendig ist. Es besteht auch keine Risiko dass die Pakete in einer anderen Reihenfolge angekommen als sie gesendet wurden, was Korrekturen auf höherer Ebene nicht notwendig macht. Packet Switched nutzt dafür die Verbindungen besser aus, da bei datenverarbeitenden Anwendungen üblicherweise eine nicht gleichmäßige Verteilung der Last über die Zeit stattfindet.

Neben synchron oder asynchron lässt sich auch noch zwischen verbindungslos und verbindungsorientiert unterscheiden. Verbindungsorientiert hat den Vorteil, dass die Verbindung zuerst hergestellt wird und daher ein garantiert fehlerfreies Übertragen möglich ist (z.B. mittels TCP), jedoch ist hier ein Overhead durch Verbindungsauf und –abbau gegeben.

Während für Verteilte Systeme die oben genannten Architekturen möglich sind basieren Communication Systems praktisch immer auf Layern. Diese haben den Vorteil dass komplexe Systeme nicht als ganzes implementiert werden müssen, sondern dass verschiedene Teilaufgaben getrennt umgesetzt werden können. Außerdem wird die Austauschbarkeit von Layern gewährleistet, was bei schnell weiterentwickelten Technologien vorteilhaft ist.

Die Kommunikation erfolgt logisch gesehen horizontal (gleiche Layer kommunizieren miteinander), in Wirklichkeit kann jeder Layer nur auf die Services des darunter liegenden aufsetzen. Physikalische Kommunikation erfolgt nur auf Layer 1.

Praktisch wurde das OSI-Modell auch in OSI-Protokollen umgesetzt, die sich aber nicht durchgesetzt haben. Stattdessen hat sich ein weiteres Modell, das TCP-Modell mit seinen Implementierungen IP auf Layer 2 und TCP/UDP auf Layer 3 durchgesetzt. IP ist ein verbindungsloses, nicht verlässliches Ent-to-End-Protokoll. TCP baut darauf einen verbindungsorientierten verlässlichen Kanel auf, UDP einen verbindungslosen unverlässlichen. Endpunkte (Adresse und Port zur Identifikation einer

Applikation auf einem Host) werden als Sockets bezeichnet, 2 Sockets gemeinsam identifizieren eine Verbindung eindeutig.

Middleware

Middleware erweitert die Funktionalität des Communication Systems um Services, die üblicherweise in vielen Applikationen in gleicher oder ähnlicher Weise benötigt werden. Somit wird dafür gesorgt, dass gleiches nicht immer wieder implementiert werden muss. Zu diesen Services zählt z.B. Naming, Security, Replication, Transaktionsunterstützung, usw.

Unterschiede zwischen den Applikationen können durch Flexibilitäten in der Middleware ausgeglichen werden. Dazu zählen z.B. klassische Parameter, Interceptors (Codestücke, die an verschiedenen Stellen in der Middleware eingehängt werden können), Aspektorientierte Programmierung, Refletion-Mechanismen von Programmiersprachen, usw.

Prozesse, Threads, Clients und Server

Prozesse und Threads

Ein Prozess ist eine Einheit für unabhängige Ausführung und für Resourcenzuteilung. Ein Thread ist (innerhalb eines Prozesses) nur eine Einheit für unabhängige Ausführung, jedoch teilen sich mehrere Threads die Resourcen (insbesondere auch den Speicher).

Die Kommunikation zwischen mehreren Threads ist aus technischer Sicht einfacher als zwischen Prozessen und erfordert weitaus geringeren Overhead, dafür ist mehr Vorsicht bei der Implementierung geboten, da Zugriffe unter Umständen synchronisiert werden müssen. Ein Kontext-Switch zwischen Prozessen ist weitaus aufwändier als zwischen Threads (nur bei User Level Threads).

Als Alternativen für die Implementierung stehen zur Verfügung:

- User Level Threads: Im User Level wird eine Bibliothek angeboten, die für Erzeugung, Zerstörung und Scheduling von Threads zuständig ist.
- Kernel Level Threads: Threads werden durch Systemaufrufe verwaltet. Scheduling ist dadurch aufwändiger, dafür blockieren Threads bei synchronen Systemaufrufen nicht den ganzen Prozess. Außerdem kann Thread Scheduling von mehreren physischen CPUs Gebrauch machen und der Kernel kann selbst multi-threaded aufgebaut sein.
- Lightweight Processes: User Level Threads werden durch LWPs ausgeführt (genau einer pro LWP), wobei LWPs wiederum auf Kernel Level Threads gemappt werden. Dadurch kann beim Blockieren eines User Level Threads (samt seinem LWP) auf einen anderen LWP umgeschaltet und ein anderer User Level Prozess ausgeführt werden, während gleichzeitig das Teure Erzeugen und Zerstören von Kernel Level Thredas vermieden wird (LWPs werden vergleichsweise selten geändert, hauptsächlich nur beim Prozessstart).

Abgesehen von Kernel Level Threads und LWPs können blockierende Threads, die zu blockierenden Prozessen führen, auch durch sogenannte Scheduler Activation oder durch Jacketing vermieden werden. Scheduler Activation erlaubt dem Kernel Upcalls, also das Benachrichtigen den Thread Schedulers auf User Level, wenn ein Thread blockiert. Jacketing meint das Ersetzen blockierender durch nicht blockierende Systemfunktionen.

Multithreaded Clients

Wenn Clients mit Hilfe mehrere Threads realisiert sind können sie einerseits leichter implementiert werden (separate Implementierung logischer Teilaufgaben) und außerdem effizienter ausgeführt werden. Beispielsweise können Kommunikationszeiten verborgen werden oder Hintergrundaufgaben durchgeführt werden während Benutzereingaben entgegengenommen werden. Falls serverseitig Vorkehrungen getroffen sind kann auch Load Balancing ausgenutzt werden.

Multithreaded Server

Mehrere Threads sind nur eine Möglichkeit einen parallelen Server zu implementieren. Alternativ dazu könnten (theoretisch) auch mehrere Prozesse gestartet werden, oder die Parallelität wird durch manuellen Nachbau von Threads mittels Finite-State-Machines (Nicht-blockierende Systemcalls und

ein Thread, der Client- oder I/O-Requests von einer Queue nacheinander bearbeitet) simuliert werden.

Ein multithreaded Server besteht grundsätzlich aus einem Dispatcher-Thread, der auf eingehende Client-Request wartet, und für jeden davon einen Worker Thread startet, der sich von nun an um diese Client-Verbindung kümmert. Der Dispatcher kann in der Zwischenzeit auf weitere Anfragen warten.

Eine weitere Frage bei multithreaded Server ist, ob gleich ein Vorrat mehrere Threads gehalten werden soll (Thread Pool), oder ob für jede Anfrage ein separater Thread gestartet werden soll. Erstere Variante hat den Vorteil, dass zur Laufzeit keine Threads erzeugt werden müssen, was über viele Anfragen hinweg Overhead einspart. Threads können, falls nicht die Pool-Lösung gewählt wird, pro Request (pro Methodenaufruf), pro Objekt oder pro Verbindung erzeugt werden.

Ein komplett iterativer Server kann immer nur einen Client gleichzeitig behandeln. Multithreaded Server haben also den Vorteil, dass sie mehrere Clients gleichzeitig abfertigen können, was zu kürzeren Antwortzeiten und einer insgesamt besseren Auslastung des Serverrechners führt. Im Vergleich zu anderen Lösungen für parallele Server sind multithreaded Server einfacher zu implementieren.

Binding

Ein Client kann einen Serverprozess auf mehrere Arten auffinden. Eine Möglichkeit ist die direkte Angabe von IP+Port, eine weitere die Verwendung eines Dämons am Server, der auf einem Well-Known-Port Client-Anfragen über den Port eines spezifischen Sevices beantwortet. Eine letzte Alternative ist ein sogenannter Superserver, der auf einem oder mehreren Well-Known-Ports auf Anfragen an die verschiedensten Services wartet und diese dann je nach Bedarf startet und ihnen die (bereits angenommene) Client-Verbindung mittels TCP-Handover übergibt.

Remote Procedure Call

Um die Verteilungstransparenz zu fördern werden durch RPC Methodenaufrufe derart umgesetzt, dass die Methode auf einem anderen Rechner läuft als der aufrufende Code. Dazu werden auf beiden Seiten virtuelle Repräsentationen der jeweils anderen Seiten, sogenannte Stubs, eingesetzt. Diese kapseln die Parameter und den Returnwert in Nachrichten (marshalling) und kommunizieren miteinander. Für den aufrufenden Code sowie für die Methodenimplementierung sieht ein RPC wie ein lokaler Aufruf aus. RPCs können synchron oder asynchron sein, wobei 2 asynchrone Aufrufe auch kombiniert werden können (deferred synchronous RPC) um den Returnwert an den Client zu übermitteln, nachdem dieser inzwischen schon weitergearbeitet hat.

RPC übergibt sämtliche Parameter als Wert. Referenzen werden entweder durch call by copy-and-restore oder durch Callback-Funktionen aufgelöst.

Die Stubs werden aus einer Spezifikation in einer IDL (Interface Definition Language) automatisch generiert.

Um einen Client-Stub an einen Server zu binden ist es notwendig den Server überhaupt erst zu finden. Dies kann durch direkte Angabe von IP+Port des Servers erfolgt, hat aber den Nachteil, dass

Migration der Server-Prozedur nicht möglich ist. Mehr Flexibilität erreicht man durch Verwendung eines Dämons am Server, der auf einem Well-Known-Port auf Anfragen wartet. Bei diesem Dämon kann dann der genaue Port für RPC-Calls erfragt werden. Um auch mehr Unabhängigkeit von der konkreten IP-Adresse des Servers zu bekommen, kann man zusätzlich einen Directory-Rechner verenden, bei dem die Adresse eines bestimmten Servers durch Angabe eines logischen Namens gefunden werden kann (eine einfache Form von flachem Naming). Ein Service muss sich natürlich sowohl beim lokalen Dämon als auch beim Directory-Rechner registrieren.

Typen von RPC:

- Synchron (Warten des Clients auf das Ergebnis)
- Asynchron (Warten des Clients bis die Nachricht beim Server angekommen ist, aber nicht auf ein Ergebnis – nur für Aufrufe ohne Returnwert)
- One-Way-RPC (kein Warten, nicht einmal auf die Ankunft der Nachricht)
- 2 asynchrone Aufrufe (deferred synchronous RPC) um trotz asynchronem Verhalten einen Returnwert zurückgeben zu können

Remote Method Invocation

Durch RMI wird RPC auf Methodenaufrufen in Objekten erweitert. Im Unterschied zu RPC können nun auch Referenzen auf (verteilte) Objekte übergeben werden. Lokale Objekte werden nach wie vor als Wert übergeben, ebenso wie primitive Datentypen.

Das Prinzip der Stubs bleibt unverändert. Der Client-Stub wird hier als Proxy, der Server-Stub als Sekelton bezeichnet.

Als Objekte können nicht nur Compile-Time-Objects (Instanzen von Klassen einer Programmiersprache) verwendet werden, sondern beliebige Implementierungen, solange sie sich dem User gegenüber wie ein Objekt verhalten. Es ist also auch prozeduraler Code, der auf Dateien operiert, als Objekt auffassbar. Die Implementierungsdetails können von einem sogenannten Object Adapter (der als Wrapper fungiert) verborgen worden, der sich (neben den Activation Policies) auch darum kümmert, dass die Implementierung für den Benutzer wie ein Objekt aussieht. Zur Unterscheidung von Compile-Time-Objects werden solche Objekte als Servants bezeichnet.

Es weitere Unterscheidung betrifft persistente und transiente Objekte, je nachdem ob sie nach Beendigung der erzeugenden Serverprozesses weiterhin existieren oder nicht.

Das Binden eines Clients an ein distributed Object kann explizit oder implizit erfolgen. Implizit bedeutet, dass Methoden direkt über die Referenz auf ein remote Object (die z.B. über einen Directory-Server und den Dämon am Server gefunden wird) aufgerufen werden. Explizit bedeutet, dass man im Sourcecode zuerst zur Remote Reference die Referenz auf den lokalen Stub ermitteln und darüber Methoden aufrufen muss. Im Hintergrund werden implizite Aufrufe immer in explizite umgewandelt.

Aufrufe können entweder statisch oder dynamisch erfolgen. Statisch bedeutet, dass der Aufruf direkt über ein vordefiniertes Interface, das in der jeweiligen Programmiersprache implementiert ist, erfolgt. Der Aufruf sieht daher einem lokalen sehr ähnlich. Der Clientcode muss jedoch rekompiliert werden, falls sich die Interfaces ändern. Bei einem dynamischen Aufruf wird dagegen eine spezielle Call-Funktion der Middleware verwendet, der als Parameter (neben den eigentlichen Parametern für

die aufgerufene Methode) auch die Methoden-ID übergeben wird. Aufrufe werden also zur Lazufzeit zusammengebaut, was auch mehr Flexibilität bringt, es jedoch dem Compiler erschwert Fehler frühzeitig zu erkennen.

Globale Objektreferenzen müssen alle notwendigen Informationen enthalten um das Objekt eindeutig finden zu können. Implementierungsalternativen sind:

- IP+Port+ObjectID des Servers
- IP + ObjectID des Servers (Auflösung durch Dämon)
- IP eines Directory Servers + ObjectID (Auflösung über Directory Server und Dämon)
- Implementation Handle

Daneben werden Informationen über das verwendete Protokoll (z.B. TCP oder UDP), Datenformate und Fehlerbehandlung gespeichert.

Code Migration

Das Verschieben von Code zwischen verschiedenen Rechnern kann grundsätzlich auf 2 Arten erfolgen: während der Laufzeit (strong migration) oder während der Code nur auf der Platte steht (weak migration). Wird er zur Laufzeit übertragen, so ist der Aufgwand wesentlich größer (da auch Resourcen und Codesegmente mitübertragen werden müssen), dafür ist die Flexibilität höher. Der Code kann dann entweder vom Empfangsprozess direkt ausgeführt werden, oder er wird gespalten (fork), wobei ein Teil den übermittelten Code ausführt.

Wird der Code übermittelt während er nicht läuft, so sind nur der statische Programmcode und eventuell Initialisierungsparameter zu übertragen.

Die Migration kann entweder durch den Client (z.B. Java-Applet) oder den Server (z.B. mobile Agenten) initiiert werden. Code Migration wird zum Load Balancing sowie zur Platzierung des Codes in der Nähe der verarbeiteten Daten verwendet. Beides erhöht die Scalability.

Wie die Migration von Resourcen durchgeführt hängt davon ab, wie stark die Resource an den aktuellen Ort des Codes gebunden ist (leicht, schwer oder nicht übertragbar) und wie sehr der Code an die Resource gebunden ist (namentlich, nach dem Wert und nur nach Typ). Alternativen sind: Kopieren, Verschieben, Verwendung globaler Referenzen.

Die Übertragung des Speicherinhalts bei Migration von laufendem Code kann durch Pushing mit eventuell wiederholter Page-Übertragung (falls sie sich inzwischen geändert haben), Stoppen und Pushing der Memory-Pages oder durch Pulling (langsam) erfolgen.

Virtualisierung

Virtualisierung meint die Simulation eines Interfaces, indem die Aufrufe auf ein anderes vorhandenes Interface abgebildet werden. Virtuelle CPUs werden nun also auch durch andere virtualle Resourcen erweitert.

Dies kann entweder auf Prozessebene erfolgen (z.B. JVM) oder für ganze Betriebssysteme (VMM – Virtual Machine Monitor, z.B. VMWare). Historisch wurde Virtualisierung zur Unterstützung alter Systeme verwendet. Heute sind daneben andere Gründe auch noch die Sicherheit (Isolation

mehrerer Prozesse) sowie die einfachere Administrierbarkeit: es werden einfach lauter gleiche Maschinen aufgestellt, die Anwendungen werden samt ihren Umgebungen (also auch inklusive Betriebssystem) darauf installiert. Die Heterogenität wird dadurch reduziert.

Message Orientated Communication

Im Unterschied zu RPC und RMI handelt es sich bei MoM (Message-orientated Middleware) um eine Lösung zur expliziten Kommunikation, während die anderen Technologien die Verteilungstransparenz fördern.

Bei MoM hat jeder Prozess eine Sende- und eine Empfangsqueue. Nachrichten, die in die Sendequeue eingetragen werden, werden von der Middleware automatisch in die Empfangsqueue des Empfängers übertragen. Die Sendequeue ist dabei entweder direkt am Senderechner oder an einem nahe gelegenen anderen Rechner (im LAN) eingerichtet.

Man kann unterscheiden zwischen persistenter und transienter, sowie zwischen synchroner und asynchroner Kommunikation.

Es gibt zwischen synchron und asynchron auch noch Zwischenvarianten, bei denen zumindest auf den Empfang einer Nachricht durch die Middleware bzw. auf das Zustellen gewartet wird (nicht aber auf die Verarbeitung). Diese Zwischenlösungen werden meist bei Message-orientierten Systemen noch als synchron bezeichnet, während sie bei RPCs bereits als asynchron gelten.

Konkret gibt es folgende Alterantiven:

- Asynchron und Persistent (klassische Variante für MoM)
- Synchron und Persistent (warten bis zur Ablegung in der Empfänger-Mailbox)
- Asynchron und Transient
- Synchron und Transient
    - Warten auf den Empfang in der Empfängerqueue
    - Warten auf den Empfang durch die Applikation
    - Warten auf die Verarbeitung und die Antwort

Transient bzw. persistent sagt aus, ob eine Nachricht von der Middleware zwischengespeichert wird oder ob sie nur solange existiert, solange Sender und Empfänger beide ausgeführt werden.

Nachrichten werden mit einer logischen Adresse des Empfängers adressiert. Der Queuing-Layer der MoM ist dafür zuständig, diese Adresse in die IP-Adresse des Empfängers (bei transienter Kommunikation), bzw. in die Adresse eines nahe gelegenen Message Routers umzuwandeln.

Message Router werden verwendet um persistente Kommunikation zu ermöglichen (andernfalls wäre nur direktes Senden an den Empfänger möglich, was aber erfordert dass beide Kommunikationspartner gleichzeitig online sind) und die Last im Netzwerk im Netzwerk einzugrenzen. Man muss Nachrichten nur noch an die Ausgangskanäle schicken, an denen der Empfänger angeschlossen ist (was anhand einer Routingtable feststellbar ist). Darüber hinaus können Router auch noch Umwandlungen der Nachrichten vornehmen, um etwa das Format anzupassen (Enterprise Application Integration). Dabei handelt es sich dann um sogenannte Message Broker.

Stream-orientated Communication

Unter kontinuierlichen Streams (im Unterschied zu Datenströmen) werden Verbindungen verstanden, über die sehr viele, sehr kleine und überdies zusammenhängene Informationspakete geschickt werden. Für solche Verbindungen bestehen neben Obergrenzen für die Verzögerungen auch noch Untergrenzen (sogenannte isochrone Verbindungen): es handelt sich um zeitkritische Anwendungen. Auch die Reihenfolge der Pakete ist hier wesentlich. Bei gewöhnlichen Datenströmen ist meistens die Korrektheit der Anwendung durch fehlerhaftes Zeitverhalten nicht gefährdet.

Die Eigenschaften von kontinuierlichen Streams sind also:

- Übermittlung von zeitabhängigen Informationen
- Gleichmäßige Verzögerungen sind erforderlich
- Reihenfolge der Pakete ist wesentlich

Kontinuierliche Streams werden vor allem für Multimedia-Anwendungen verwendet.

Mehrere Streams gemeinsam (z.B. mehrere Kanäle bei Audiosignalen) werden zu sogenannten komplexen Streams zusammengefasst.

Die Anforderungen an eine Verbindung für die Übertragung von kontinuierlichen Streams sind also wesentlich anders als die für diskrete Daten. Die zeitlichen Anforderungen fallen in den Bereich Quality of Service, also nicht-funktionale Anforderungen. Diese können der Middleware üblicherweise als Parameter mitgeteilt werden, und werden intern durch Ausnutzung von IP-Flags sowie Buffer (vor allem beim Empfänger) umgesetzt. Eine Möglichkeit um lange Aussetzer bei Multimediaanwendungen zu vermeiden ist es, die Pakete zu verzahnen und in einer anderen Reihenfolge zu senden als sie eigentlich benötigt werden. Kurze Störungen zerstören somit Pakete, die im Endprodukt nicht direkt hintereinander liegen.

Naming

Definition

Namen werden verwendet um auf Resourcen zugreifen zu können. Genauer: Namen identifizieren Resourcen. Es gibt folgende Spezialformen von Namen:

Menschenlesbare Namen

Adresse: Das ist der Name eines Access Points einer Resource. Direkte Verwendung einer Adresse ist problematisch, da dadurch die Location-, Replication- und Migration-Transparenz nicht unterstützt wird und Adressen auch oft für Menschen schwer lesbar sind. Gute Namen sollten daher die Adresse weder direkt noch versteckt codieren.

Identifier: Ein Name mit folgenden Eigenschaften:

- Jede Entität hat genau einen Identifier
- Jeder Identifier gehört zu höchstens einer Entität
- Identifier werden niemals wieder verwendet wenn sie bereits einmal in Verwendung waren

Identifier können z.B. durch fortlaufende Zahlen oder durch einen Zufallszahlengenerator (der jedoch mit einer geringen Wahrscheinlichkeit doppelte Identifier erzeugt) erzeugt werden.

Attribute: Schlüssel/Werte-Paare für attributbasiertes Naming

Orthogonal dazu ist folgende andere Klassifizierung:

Flache Namen

Hierarchische Namen

Unter einem Namensraum versteht man die Menge aller Namen. Im Falle von hierarchischen Namen ist das ein gerichteter zyklenfreier Graph.

Ein Name Service dient der Auflösung von Namen in zugehörige Attribute (vor allem Adressen). Nehmen der lookup()-Operation unterstützt es noch die bind()-Operation, mit der neue Namen in den Namensraum eingetragen werden können. In großen Systemen ist vor allem die Skalierbarkeit wichtig, was durch Verteilung und Replikation erreicht werden kann.

Wünschenswerte Attribute on Namen sind:

- Keine Codierung der Location im Namen
- Eindeutigkeit (z.B. durch Verwendung von Kontexten oder ID-Generatoren erreichbar)
- Unterstützung von Alias-Namen

Ein Beispiel für eine einfach Namensauflösung ist ARP (Address Resolution Protocol), bei dem die IP-Adresse eines Hosts als Broadcast ausgeschickt wird. Jeder Rechner inspiziert dieses Paket und der, dem diese Adresse gehört, wird mit seiner MAC-Adresse antworten.

Flache Namen

Als einfache Alternativen kommen Broad- oder Multicasts (z.B. ARP) in Fragen. Komplexere Lösungen verwenden Forwarding Pointers (vor allem bei oft veränderlichen Locations), Home-based Approaches, hierarchische Lösungen (nur in Bezug auf die Architektur des Naming Services, die Namen selbst sind flach).

Hierarchische Namen

Definition

Der Namensraum ist ein gerichteter azyklischer Graph. Oft gibt nur einen Pfad zu einem bestimmten Knoten, es kann aberauch mehrere geben (Hard Links → Alias). Soft Links sind dagegen Blattknoten, die als Daten (z.B. als Attribute) den absoluten Pfad eines anderen Knotens speichern.

Namensräume können auch verbunden werden. Dies kann durch Mounting erfolgen (ein Namensraum oder ein Teil davon – der Mounting Point – wird am Mount Point eines anderen Namnesraums eingehängt). Eine Alternative ist es, 2 Namensräume einem neuen gemeinsamen Root Wurzelknoten unterzuordnen. Bisher gültige Namen können trotzdem noch aufgelöst werden, da diese ohnehin implizit immer den Wurzelknoten codieren, auf den sie Bezug nehmen.

Für Mounting sind Informationen über das Zugriffsprotokoll (z.B. NFS), den Server (z.B. IP) und den Mouting Point im fremden Namensraum (ein absoluter Pfad in diesem) erforderlich.

Das Auflösen (Resolution) in einem hierarchischen Namensraum erfolgt grundsätzlich in einer Folge von 2 Teilschritten:

- Suchen eines Namens in einem Directory um den Identifier des nächsten Nameservers (oder des Blattknotens) zu finden. In diesem Schritt wird ein (Teil-)Name also immer wieder einem anderen Namingcontext präsentiert (ein Namingcontext ist ein Pfad zu einem inneren Knoten im Namensraum, ab dem dann relative Adressen angegeben werden können).
- Kontaktieren des nächsten Nameservers

Im Falls von NFS bedeutet das:

- Laden des Datenblocks zu einem inode um darin die inode-Nummer des nächsttiefer gelegenen Verzeichnisses zu finden
- Zugriff auf den nächsten inode

Neben der Resolution ist noch ein sogenannter Closure-Mechanismus erforderlich, der verwendet wird um den initialen Knoten zu finden. Dieser ist notwendigerweise implizit, beispielsweise hard-coded im Betriebssystem-Kernel.

DNS

Das Domain Name Service dienst der Auflösung von DNS-Namen im Internet. Der Namensraum ist in 3 Layer eingeteilt: global, administrational, managerial. Von oben nach unten werden Änderungen immer häufiger, weshalb DNS-Server für Zonen (Teile des Namensrausm, die von einem bestimmten Name Server aufgelöst werden) im oberen Bereich des Namensraums zwar ausfallssicherer sein sollten, dafür aber die Antwortzeiten nicht so kurz sein müssen wie bei Servern für untere Zonen, da ihre Ergebnisse effektiver gecacht werden können.

Für die Auflösung eines DNS-Namens müssen die Name Server für die beteiligten Zonen Top-Down kontaktiert werden, wobei jeder Server den Teil des Namens auflöst, der in seinen Zuständigkeitsbereich fällt. Die Resolution kann dabei entweder vom Client oder vom Server geleitet werden, wobei beim Server die Unterscheidung zwischen rekursiver und iterativer Namensauflösung gemacht werden kann. Clients lösen immer iterativ auf (d.h. ein Zonen-NS wird nach dem anderen vom Client kontaktiert, die NS kommunizieren untereinander nicht).

Die Namensserver speichern Schlüssel/Werte-Paare für ihre Zone, sogenannte Resource Records. Es gibt verschiedene Typen von Resource Records, etwa Adressen von Hosts (A), Textinformationen (TXT), Mailserver für E-Mails an diese Zone (MX), Start of Authority (SOA, Zuständigkeit für diese Zone), CNAME (Canonical Name, Alias), PTR (eine Adresse aus in-addr.arpa. für Reverse Lookup). Nameserver für eine Zone müssen immer mindestens doppelt ausgeführt sein (ein primärer und ein oder mehrere sekundäre NS), wobei die sekundären Server nicht direkt auf die DNS-Datenbank (ein Textfile) zugreifen, sondern Änderungen vom primären Server mittels Zone Transfer übernehmen. Daneben können auch noch nicht autorisierte Server eine Antwort auf eine Anfrage liefern (anhand ihrer Caches). Primäre und sekundäre NS liefern autorisierte Antworten.

Vorteile einer rekursiven Namensauflösung sind, dass Caching besser ausgenützt werden kann (nicht nur beim Client sondern auch bei den Namensservern) und dass lange Latenzzeiten durch große Entfernungen zwischen Namensserver und Clients vermieden werden. Caching wiederum führt zu weniger ausgelasteten Namensservern, sorgt für höhere Verfügbarkeit (Namen können auch aufgelöst werden wenn der Namensserver selbst ausgefallen ist) und reduziert den Netzwerkverkehr. Caches bringen aber das Risiko mit sich, dass unter Umständen ein veraltetes Ergebnis zurückgegeben wird.

Attribute-based Naming

Diese Form des Namings (auch Directory Service genannt) ermöglicht die Suche von Services anhand ihrer Attribute und nicht nur ihres Namens. Dazu wird ein sogenannter Attributbaum aufgebaut, dessen Kanten mit Attributen und zugewiesenen Werten beschriftet sind. Sucht man nun nach einer Eigenschaft, so muss man nur den Attributbaum von der Wurzel ausgehend verfolgen und immer den Zweig weiterverfolgen, der mit dem gesuchten Attribut/Wert-Paar beschriftet ist.

Unterschiede zu DNS sind, dass nun auch Attribute ausgelassen werden können (während bei DNS nicht einfach Teile eines Namens ausgelassen werden können). In diesem Fall müssen an der fraglichen Stelle mehrere Zweige nach unten verfolgt werden.

Abgesehen davon ist ein weiterer Unterschied, dass die inneren Knoten im Attributbaum nicht nur der weiteren Navigation dienen, sondern selbst nützliche Informationen enthalten können. Aus diesem Grund gibt es auch 2 Operationen auf innere Knoten: read (zum Auslesen des Knotens selbst) und list (zum Auslesen der Subknoten).

Ein Beispiel dafür ist der X.500-Namensraum, ein attributbasiertes System für das weltweite Informationssystem. X.500 normiert einige Attribute und gibt den Organisationen die Freiheit selbst noch welche hinzuzufügen. Genormte Attribute sind z.B. Country, Organisation, Organisational Unit.

DAP und die vereinfachte Version LDAP (Lightweight Directory Access Protocol) ist ein Zugriffsprotokoll auf einen gemäß X.500 aufgebauten Namensraum. Er ermöglicht die Durchführung von read- und list-Operationen. In Windows 2000 ist es als Active Directory implementiert.

Discovery Services

Im Unterschied zu Naming werden diese Services in ad hoc-Netzen verwendet, um Services von einem Host aus anbieten und von den anderen aus auffinden zu können. Die Spezielle an ad hoc-Netzen ist, dass sich diese in relativ kurzen Intervallen ändern. Discovery-Services haben die Möglichkeit, Services zu registrieren und nach welchen suchen zu können.

In hoch mobilen Netzen ist es vorteilhaft, das Mapping von Namen auf Entitäten und das Mapping von Entitäten auf Locations (Adressen) zu trennen. Das Naming Service kümmert sich nur um den ersten Teil, ein Location Service um den zweiten. Verwendet wird das zum Beispiel in der Mobiltelefonie, bei der zuerst ein Mapping einer Rufnummer auf eine SIM-Karte erfolgt (Naming Service), und erst in einem weiteren Schritt diese SIM-Karte lokalisiert wird. Der Grund, wieso direktes Mapping schwierig ist, ist, dass traditionelle Name Services (egal welcher Art) davon ausgehen, dass die Einträge einigermaßen stabil sind. Deswegen wird Caching eingesetzt, was aber bei hoch mobilen Netzen zu vielen inakzeptablen Fehlzugriffen führen würde.

JNDI

Java Naming and Discovery ist eine Java-Schnittstelle für den Zugriff auf verschiedenste Namensräume. Die JNDI-API setzt auf den JNDI-Manager auf, derwiederum auf die verschiedenen Implementierungen (etwa für DNS, RMI, CORBA, LDAP, COS und NIS) zurückgreift um die Aufrufe Namespace-spezifisch umzusetzen.

Distributed Garbage Collection

Entitäten, die nicht mehr benötigt werden, sollten aus dem System entfernt werden. Wenn sich diese explizit abmelden ist das kein Problem, jedoch will man solche nicht mehr benötigten Teilsystem automatisch erkennen können.

Eine Möglichkeit dazu ist Reference Counting. Zeigt keine Referenz mehr auf eine Entität (z.B. ein Skelton), so kann das Objekt entfernt werden. Schwierigkeiten verursachen bei diesem Ansatz aber Zyklen sowie Referenzen, die sich in noch nicht fertig übertragenen Nachrichten befinden. In letzterem Fall könnte ein Objekt seinen Counter bereits auf 0 herabsetzen und gelöscht werden, obwohl im nächsten Moment wieder eine Referenz eingerichtet wird. Das Problem kann behoben werden indem der Host, der eine Referenz versendet, diese zuerst beim global Object anmeldet. Die ursprüngliche Referenz wird erst gelöscht, wenn die gesendete beim Objekt angemeldet und beim 2. Objekt eingerichtet wurde.

Eine Alternative ist es, nicht nur einen Counter, sondern eine vollständige Liste aller Referenzen mitzuführen. Das hat den Vorteil, dass Einfüge- und Entfernoperationen idempotent sind und daher bei unverlässlicher Kommunikation gefahrlos wiederholt werden können.

Synchronisation

Definition

Synchronisation kann auf 3 Arten erfolgen:

- Synchronisation mit einem externen Zeitgeber (Wall Clock)
- Synchronisation in einer abgeschlossenen Gruppe (relative Synchronisation)
- Synchronisation logischer Uhren (lediglich durchnumerieren von Ereignissen)

Wichtig ist Synchronisation für die Korrektheit verteilter und zeitabhängiger Informationsverarbeitung. Bei Single-Prozessor Systemen ist das deswegen kein Problem, weil ohnehin alle Prozesse auf die gleiche Zeitquelle zugreifen (mittels Kernelfunktionen).

Zeitdefinition

Ursprünglich war die Sekunde definiert als 1/86.400 eines mittleren Sonnentages (mittlerer Sonnentag deswegen, weil der Abstand zwischen 2 Transitstellungen der Sonne nicht immer gleich ist.

Inzwischen wurde diese Definition zu ungenau, da die Erdrotation abnimmt und Sonnentage daher immer länger werden. Die astronomische Definition wurde durch eine physikalische (anhand der Periodendauer des Cesium-133) ersetzt. Die TAI (Internationale Atomzeit) tickt genau nach dieser Definition. Um sie mit der astronomischen Zeit in Einklang zu bringen wurde die UTC (Coordinated Universal Time) definiert, die im wesentlichen der TAI entspricht, hin und wieder um Schaltsekunden erweitert wird (um die länger werdenden Tage zu kompensieren).

In elektronischen Geräten werden Timer eingesetzt. Das sind Quarze, die durch Spannung zu Schwingungen angeregt werden. Diese Quarze werden verwendet um einen Counter hinunterzuzählen. Immer wenn er auf 0 gesetzt wird, wird er wieder mit einem vordefinierten Wert geladen und ein Interrupt ausgelöst, der zur Zeitmessung (z.B. zum Zählen einer Sekunde) verwendet werden kann.

Unterschiedliche Timer zählen produktionsbedingt (geringfügig) unterschiedlich schnell. Das führt zu einer Abweichung zwischen gezählter Sekunde und einer Sekunde nach der TAI. Die clock skew steht für diese Abweichung und ist wie folgt definiert: $s(t) = f'(t) - 1$, wenn $f(t)$ die gemessene Zeit in abhängigkeit von der richtigen Zeit ist.

Synchronisation mit einem externen Zeitgeber

Die Genauigkeit von Zeitgebern wird durch das sogenannte Stratum definiert. Je niedrigeres ist, umso genauer ist eine Zeitquelle.

Mit dem NTP (Network Timing Protocol) werden Uhren mit einem höheren Stratum mit einer mit einem niedrigeren (0 = genaueste Zeit anhand einer Atomuhr) synchronisiert. Ein Rechner sendet dazu eine Nachricht an den Zeitsever und merkt sich den Sendezeitpunkt t1. Der Server zeichnet den Empfangszeitpunkt t2 sowie den Zeitpunkt kurz vor dem Absenden der Antwort t3 auf und schickt das Ergebnis zurück. Der erste Rechner vermerkt den Zeitpunkt des Empfangs t4. t1 und t4 sind nach

der lokalen Zeit des einen Rechners, t2 und t3 nach der des anderen. Durch ((t4 – t1) – (t3 – t2)) / 2 kann jedoch auf die Dauer der einfachen Übertragung geschlossen werden (wenn das Delay in beide richtungen annähernd gleich ist). Somit kann durch t4 – t1 – einfaches Delay darauf geschlossen werden, um wie viel die Zeit des Sender nach- oder vorgeht. Dies kann durch beschleunigen oder bremsen der lokalen Uhr über einen längeren Zeitraum korrigiert werden.

Der ganze Vorgang wird 8 Mal wiederholt und das Ergebnis mit dem kürzesten Delay verwendet. Ist das Delay über einem Grenzwert, so wird die Zeitquelle als nicht mehr vertrauenswürdig eingestuft, da sonst ein Angreifer (selbst bei verschlüsselten Nachrichten) durch Verzögerung der Nachrichten die Synchronisation beeinflussen könnte.

Interne Synchronisation

Sollen mehrere Uhren nur untereinander synchronisiert werden, so kann der Berkeley Algorithmus verwendet werden. Bei diesem sendet ein Koordinator seine lokale Zeit an die zu synchronisierenden Rechner (auch an sich selbst). Jeder antwortet mit seiner Abweichung von dieser Referenzzeit. Daraufhin berechnet der Koordinator die durchschnittliche Abweichung und schickt jedem zurück, um wie viel er seine Zeit vor- oder zurückstellen muss. Ein Zurückstellen erfolgt dabei durch Verlangsamung der Zeit über einen längeren Zeitraum.

Eine Alternative, vor allem für Funknetze, ist Reference Broadcast Synchronisation (RBS). Dabei sendet ein Koordinator (der hier selbst nicht mitsynchronisiert wird) einen Broadcast aus. Der Inhalt der Nachricht ist irrelevant (was ein großer Vorteil ist, da die Zeit, die die Nachricht in der Applikation und der Netzwerkkarte verbringt, unwichtig ist). Dieser Nachricht sollte bei allen ungefähr zum gleichen Zeitpunt ankommen (das Netzwerkdelay wird als relativ konstant angenommen). Jeder Prozess zeichnet seine lokale Zeit beim Empfang des Broadcasts auf. Die Prozesse wissen, dass die aufgezeichneten Zeiten gleich sein sollten und können deren Abweichungen interpretieren als Abweichungen zwischen ihren lokalen Zeiten (oft erfolgt gar keine Synchronisation im Sinne des Gleichstellens der Uhren, sondern nur eine Berücksichtigung der Abweichung bei künftigen Berechnungen).

Logische Synchronisation

Lamport Timestamps numerieren Ereignisse nur durch, sorgen also für eine logische Zeit. Potentiell kausal abhängige Ereignisse (im Sinne der Happened-Before-Relation) bekommen dabei ihre Nummern so zugewiesen, dass das abhängige Ereignis eine größere Nummer hat als die Ursache, und dass diese Nummern bei allen Prozessen gleich sind.

Dazu führt jeder Prozess seine lokale logische Uhr (einen Counter). Er zählt diesen bei jedem Sendeereignis (und eventuell auch bei Berechnungen) um 1 hoch und hängt diese Zahl der Nachricht an. Der Empfänger setzt beim Empfang seinen Counter auf das Maximum seines aktuellen Counters und dem mitgeschickten Wert, und erhöht anschließend seinen Counter um 1 (um den Empfang zu zählen).

Damit wird erreicht dass gilt:

1. T(b) > T(a), falls a und b in dieser Reihenfolge im gleichen Prozess stattfinden
2. T(b) > T(a), falls a das Senden einer Nachricht und b ihr Empfang ist

Die Eigenschaften an Lamport Timestamps sind also:

1. 2 hintereinander folgende Ereignisse a und b im gleichen Prozess haben Zeitstempel so dass T(a) < T(b)
2. Der Empfang einer Nachricht hat immer einen höheren Zeitstempel als ihr Absenden
3. Die Relation ist transitiv abgeschlossen

Bei Lamport Timestamps kann von den Zeitstempeln aber nicht auf die kausale Abhängigkeit geschlossen werden, d.h. T(a) < T(b) bedeutet nicht unbedingt dass a → b gilt (nur in die andere Richtung).

Eine Erweiterung sind daher Vector Timestamps, bei denen jeder von n Prozessen einen Vektor der Länge n hat (initialisiert auf den Nullvektor). Bei jedem Ereignis in Prozess i zählt dieser Element i um 1 hoch. Der Vektor wird an gesendete Nachrichten angehängt. Beim Empfang durch Prozess j wird das Weitergeben der Nachricht an die Applikation solange verzögert bis gilt:

1. m[i] = Vj[i] + 1
2. m[k] <= Vj[k] für alle k außer i

Damit wird erreicht, dass potentiell kausal abhängige Nachrichten in dieser Reihenfolge an die Applikation geliefert werden. Ob 2 Nachrichten potentiell kausal abhängig sind lässt sich nun feststellen, indem man ihre Vektoren elementweise vergleich. Ist keine Reihung feststellbar, dann sind die Nachrichten concurrent (somit unterliegen Vector Timestamps einer schwächeren Bedingung als Lamport Timestamps). Es handelt sich also um eine partielle Ordnung von Ereignissen. Werden Vector Timestamps bei Multicasts verwendet, so spricht man von Causally-Ordered Multicasts. Bei Annahme einer Nachricht wird elementweise das Maximum des lokalen und des mitgesendeten Vektors übernommen.

Mutal Exclusion

Durch Mutal Exclusion sollen die Aktivitäten mehrere Prozesse so koordiniert werden, dass:

- Auf eine bestimmte Resource immer maximal ein Prozess zugreifen kann
- Liveness: Jeder einmal drankommt (no deadlock, no starvation)
- Der Zugriff fair abläuft (z.B. mittels FIFO-Policy)

Die beste Lösung dafür ein zentralisierter Ansatz, bei dem ein Prozess den Zugriff auf die Resource regelt. Jeder, der sie verwenden will, meldet sich beim Koordinator. Dieser antwortet mit ack, falls die Resource frei ist, und gar nicht, falls sie belegt ist (in diesem Fall wird der Request auch in eine Queue eingetragen). Wenn ein Prozess mit der Verwendung der Resource fertig ist gibt er sie durch eine Mitteilung an den Koordinator wieder frei. Dieser kann dann den nächsten wartenden Prozess aus der Queue nehmen und ihm den Zugriff erlauben.

Alternative Lösungen wäre ein dezentralisierter Ansatz (mehrere Koordinatoren werden befragt, wobei nur eine Mehrheit notwendig ist um zugreifen zu dürfen), ein verteilter Algorithmus (jeder fragt jeden anderen Prozess, bei 2 interessierten Prozessen entscheidet der niedrigere logische Zeitstempel der Anfrage) oder ein Token Ring Algorithmus. Man kann aber zeigen, dass jeder alternative Ansatz im Endeffekt schlechter ist als der zentralisierte, da dann mehr Nachrichten

versendet werden und mehrere Fehlerquellen ins Spiel kommen. Die beste Lösung bleibt daher der zentralisierte Ansatz, wobei besonders viel Mühe in die Absicherung des Koordinators gesteckt wird.

Election Algorithms

Oft ist die Festlegung eines (in irgendeiner Weise) ausgezeichneten Prozesses notwendig. Es ist dabei meist unwichtig wer diese Rolle übernimmt, nur dass es genau einer ist. Das ist zum Beispiel für die Suche nach einem Koordinator für Mutal Exclusion notwendig. Solche Algorithmen sind ein Spezialfall von Agreement Algorithmen.

Beim Bully-Algorithmus schickt ein Prozess, der den Ausafall des aktuellen Koordinators bemerkt hat, eine Election-Nachricht an alle Prozesse mit einer höheren Prozessnummer als er selbst. Diese Antworten ihm mit einer Stop-Nachricht und setzen in gleicher Weise fort. Somit erhält jeder Prozess, mit Ausnahme des Prozesses mit der höchsten Nummer, eine Stop-Nachricht. Der Prozess mit der höchsten Nummer erkennt dass er keine Stop-Nachricht erhält und erklärt sich daher zum Koordinator, was er allen anderen durch eine eigene Koordinator-Nachricht mitteilt.

Ein anderer Ansatz ist der Ring-Algorithmus. Der Prozess, der den Ausfall des aktuellen Koordinators bemerkt, schickt dazu einen Token mit siner eigenen Prozessnummer in einem logischen Ring aus. Jeder andere Prozess trägt seine Nummer ein und schickt den Token weiter. Kommt er zum ursprünglichen Sender zurück, so erkennt er dies und ändert den Nachrichtenttyp auf „Coordinator“. Er schickt den Token noch einmal im Kreis. Alle Prozesse erkennen anhand des Nachrichtentyps, dass die höchste Prozessnummer im Token die Nummer des neuen Koordinators ist.

In einem ad hoc-Netzwerk ist es oft wichtig, dass nicht irgendein Koordinator gewählt wird, sondern der beste (anhand irgendeines Kriteriums, z.B. der Restlaufzeit der Batterien). Es wird ein logischer Baum (ein sogenanntes Overlay-Netzwerk) im ad hoc-Netz aufgebaut, indem ein Prozess beginnt Election-Nachrichten an alle Nachfolger zu senden. Jeder Nachfolger wird beim Empfang der ersten Election-Nachricht diese an alle Nachfolger weiterschicken und beim Empfang einer 2. (und jeder weiteren) Election-Nachricht sofort antworten und diese zurückweisen. Sobald ein Prozess antworten von allen Nachfolgern erhalten hat, wählt er die beste Prozessnummer von seinen Nachfolgern und sich selbst aus, und schickt diese an den Parent zurück. Somit kommt beim Initiator die beste Prozessnummer an, die er allen als neuen Koordinator mitteilen kann.

Für die Superpeer-Selektion (Auswahl mehrerer ausgezeichneter Knoten, die sich jeweils um eine Anzahl anderer Knoten kümmern) existiert ein Algorithmus, der nach physikalischem Vorbild funktioniert. Es werden dazu für n Superpeers n Token zufällig im Netz verteilt. Erkennt ein Knoten mit Token, dass sich in der Umgebung bereits andere Token befinden, so berechnet er die logisch wirkenden Kräfte dieser Token auf seinen und gibt ihn dann in die entsprechende Richtung weiter. Es werden sich die Token dadurch automatisch gleichmäßig im Netzwerk verteilen. Sobald ein Knoten seinen Token eine bestimmte Zeit lang gehalten hat gilt er als fixiert.

Global State

Für manche Anwendungen ist es notwendig den Status des gesamten verteilten Systems erfassen zu können. Dazu zählen die lokalen Stati der einzelnen Knoten und die Nachrichten im Netzwerk, die noch unterwegs sind. Man will dabei den globalen Status so erfassen, dass keine Wirkung ohne Ursache aufgezeichnet wird, d.h. es sollen beispielsweise keine Auswirkungen einer Nachricht

aufgezeichnet werden, ohne dass auch deren Absenden aufgezeichnet wurde. Man bezeichnen so einen Schnitt als konsistent.

Ein Algorithmus von Chandy und Lamport erzeugt wie folgt einen Snapshot vom Global State. Dazu schickt ein Initiator einen Marker aus. Beim Empfang des 1. Markers zum ersten mal zeichnet jeder Rechner seinen lokalen Status auf und schickt den Marker weiter. Bis zum Empfang des Markers zum zweiten mal zeichnet jeder Prozess alle eingehende Nachrichten des Kanels auf, von dem er den Marker erhalten hat. Beim Empfang des Markers zum zweiten mal ist die Aufzeichnung abgeschlossen: der lokale Status und die aufgezeichneten Nachrichten können an den ursprügnlichen Initiator gesendet werden, wo dann die Auswertung passieren kann.

Anforderungen an einen globalen Status sind:

- Stability: Ein Prädikat, das im Startzustand true ist, ist auch in allen Folgezuständen true.
- Safety: Wenn ein unerwünschtes Prädikat im Startzustand nicht erfüllt ist, dann ist es auch in allen Folgezuständen nicht erfüllt.
- Liveness: Jedes erwünschte Prädikat ist in jeder Linearisierung, beginnend bei einem Startzustand, erreichbar.

Der von diesem Algorithmus aufgezeichnete Status ist garantiert konsistent, enthält aber unter Umständen Kombinationen von lokalen Zuständen, die so nie aufgetreten sind.

Replikation und Konsistenz

Definition

Und Replikation versteht man die mehrfache Ausführung von Entitäten (Prozesse, Datenbanken, etc.) zum Zwecke der Verfügbarkeitserhöhung (Fehlertoleranz: Umschaltemöglichkeit auf ein Backup, Voting-Mechanismen) und/oder der Performancesteigerung (Lastverteilung, Zugriff auf Replikate in der näheren Umgebung).

Der Preis für die Replikation ist jedoch ein Zusatzaufwand durch Konsistenzerhaltung (neben den eigentlichen Kosten für die Einrichtung von Replikaten). Replikation ist daher nicht immer sinnvoll: es kommt auf Anwendung, das verfügbare Kapital, Zugriffsmuster (nicht sinnvoll ist es z.B. bei sehr vielen Schreib- und sehr wenigen Lesezugriffen), usw. an.

Aus diesem Grund führt man abgeschwächte Versionen der Konsistenz ein. Die Replikate müssen nicht mehr zu jedem Zeitpunkt vollständig übereinstimmen, sondern sich nur gemäß einem Konsistenzmodell verhalten. Ein Konsistenzmodell ist eine Beschreibung der Erwartungen, die die User und Prozesse an die Replikate richten können (welche Werte zu welchem Zeitpunkt geliefert werden können).

Arten von Replikaten

Unabhängig davon wie die Daten konsistent gehalten werden und wie Konsistenz überhaupt definiert ist, kann man mehrere Arten von Replikaten unterscheiden. Grundsätzlich können Replikate vom Server oder vom Client eingerichtet werden, wobei beim Server weiter unterschieden werden kann ob die Replikate permanent vorhanden sind (im Voraus geplant, z.B. Cluster oder Mirrors) oder ob sie zur Laufzeit eingerichtet und entfernt werden wie es gerade günstig ist (z.B. Platzierung von Kopien auf Edge-Servern des Internets, z.B. ISPs).

Werden Replikate von Clients eingerichtet (die sich dann auch um die Konsistenzerhaltung kümmern müssen), so spricht man von Caches. Diese können auch in LANs geteilt werden, um gleiche Zugriffe von unterschiedlichen Clients aus dem Cache beantworten zu können. Solche gemeinsamen Caches können in flacher oder hierarchischer Weise angeordnet werden.

Konsistenzmodelle

Man kann unterscheiden zwischen Client- und Datenzentrierten Konsistenzmodellen. Erstere geben nur für Zugriffe von einem einzigen Client Garantien ab, dass bestimmte Bedingungen eingehalten sind. Es werden keine Aussagen über gleichzeitige Zugriffe gemacht. Datenzentrierte Modelle hingegen erlauben es mehreren Clients gleichzeitig zuzugreifen und enthalten in ihrem Beschreibungen auch die möglichen Ergebnissen von Leseoperationen im Falle von concurrent Lese- und Schreibzugriffen.

Datenzentrierte Konsistenzmodelle

Die strengste Form ist strict consistency. Es wird verlangt, dass ein Lesezugriff immer die aktuellste Version eines Datum liefert, also die, die zuletzt geschrieben wurde. Dies ist jedoch praktisch nicht

möglich, da es aufgrund des fehlens eines globalen Clocks gar nicht möglich ist, die letzte Schreiboperation zu ermitteln (Zeit ist relativ).

Eine Abschwächung ist sequential consistency. Dabei wird nur verlangt, dass die Operationen eines Prozesses in der im Programm definierten Reihenfolge stattfinden, und dass die Reihenfolge der Operationen im Gesamtsystem von allen Prozessen gleich gesehen wird. Wie die Operationen zweier Prozesse verzahnt werden ist irrelevant, solange es von allen gleichgesehen wird.

Eine weitere Abschwächung ist die causal consistency. Dabei müssen nicht einmal alle Operationen von allen Prozessen in der gleichen Reihenfolge gesehen werden, sondern nur potentiell kausal abhängige Operationen. 2 Schreiboperationen sind dann potentiell kausal abhängig, wenn eine Schreiboperation in einem Prozess i einen Wert überschreibt, die vorher vom gleichen Prozess i gelesen und vor diesem Lesezugriff von einem anderen Prozess j geschrieben wurde.

Clientzentrierte Konsistenzmodelle

Eventual Consistency ist ein Modell, bei dem geänderte Daten nicht sofort mit den Replikaten abgeglichen werden, sondern nur hin und wieder. Finden längere Zeit keine Änderungen statt (und nur in diesem Fall ist das Modell sinnvoll anwendbar), dann ist davon auszugehen dass irgendwann alle Replikate wieder am aktuellsten Stand sein werden.

Es gibt daneben noch folgende clientzentrierte Konsistenzmodelle:

Monotonic Read: Wenn ein Client ein Datum gelesen hat, dann kann er später nur noch die gleiche ode reine aktuellere Version lesen (egal bei welchem Replikat)

Monotonic Write: Wenn ein Client schreibt, dann überschreiben zukünftige weitere Schreiboperationen nur die zuvor geschriebene, oder eine noch aktuellere Version, aber niemals eine ältere.

Read your Writes: Wenn ein Client geschrieben hat, dann liest er später nur die zuvor geschriebene oder eine noch aktuellere Version, niemals eine ältere

Writes follow Reads: Wenn ein Client ein Datum gelesen hat, dann überschreiben alle zukünftigen Schreibzugriffe das Datum in der zuvor gelesenen oder einer noch aktuelleren Version.

Abgleich von Replikaten

Unabhängig davon welche Konsistenzbedingungen an die Replikate gestellt werden, kann man folgende Varianten für den Abgleich der Replikate unterscheiden:

- Übertragung der Zustände
- Übertragung der Operationen
- Benachrichtigung, ohne Daten zu übertragen

Die ersten 2 Varianten können entweder vom Server oder vom Client initiiert werden (push vs. pull). Push-Methoden sind bei vielen Lesezugriffe zu bevorzugen, da dann die aktualisierten Daten möglichst schnell verfügbar sind. Bei wenigen Lesezugriffen sind Pull-Methoden besser, da dann unnötige Updates vermieden werden. Eine Hybridform ist die Verwendung von Leases: in diesem Fall wird der Server eine Zeit lang seine Updates mittels Push propagieren. Danach muss der Client

wieder anfragen (pull oder Lease verlängern). Die Dauer einer Lease kann verändert werden, abhängig davon wie stark der Server ausgelastet ist, wie oft auf das Replikat zugegriffen wird und wie wahrscheinlich es ist, dass sich die Daten in naher Zukunft ändern.

Konsistenzprotokolle

Konsistenzprotokolle sind Implementierungen von Konsistenzmodellen. Alle beschriebenen Protokolle setzen diesequential consistency um.

Grundsätzlich lässt sich unterscheiden zwischen primary-based-Protokollen und replicated-write-Protokollen.

Primary-based-Protokolle erlauben den Schreibzugriff nur auf einem Replikat. Ein Schreibrequest auf einem beliebigen anderen Replikat wird an den Primary-Server weitergereicht (remote-write). Der Primary kann dabei für jedes Datenitem ein anderer Host sein. Alternativ dazu gibt es noch die Möglichkeit, das Datenitem zuerst auf den Server zu migirieren, auf dem der Schreibzugriff abgesetzt wird (Änderung des Primarys) und die Operation dann dort auszuführen (local-write). Remote-write kann entweder synchron oder asynchron durchgeführt werden, je nachdem ob die Updates an alle Replikate propagiert werden bevor der Client seine Antwort bekommt oder nicht.

Replicated-Write-Protokolle erlauben den Zugriff hingegen auf beliebigen Replikaten. Die Änderungen werden dann an alle anderen Replikate propagiert, entweder über Active Replication (die Operationen werden ausgeschickt und überall nachvollzogen – nur bei deterministischen Operationen möglich; gleiche Reihenfolge wird durch totally-ordered Multicast und deterministisches Thread Scheduling erreicht) oder über quorumbasierte Ansätze (es wird immer eine bestimmte Anzahl von Replikaten gelesen bzw. geschrieben, so dass es keine Schreib/Schreib- oder Lese/Lese-Konflikte geben kann).

Eine Spezialform von Replicated-Write-Protokollen ist Coordinator-Cohort-Replication. Dabei wird die Anfrage an ein (beliebiges) Replikat gesendet, das die Updates synchron an alle anderen propagiert. Dazu ist ein distributed-Locking-Mechanismus erforderlich.

Epidemic Protokolle funktionieren wie folgt. Ein Host, der die Updates bereits erhalten hat, versucht diese an möglichst viele noch nicht aktualisierte Hosts weiterzugeben. Das Prinzip ist Ähnlich wie bei der Ausbreitung von Krankheiten. Die Skalierbarkeit des Protokolls ist sehr gut. Gossip-Protocols sind eine Erweiterung, bei die die Häufigkeit des Austauschs geringer wird, wenn ein Host erkennt dass die meisten anderen ohnehin schon notifiziert sind.

Bei Anti-Entropy-Protokollen wählt ein Server P einen anderen Server Q zufällig aus und gleicht seine Daten mit diesem ab: entweder durch Push, durch Pull, oder durch beides. Problematisch ist dabei jedoch, dass bei sehr vielen bereits aktualisierten Servern die zufällige Auswahl eines noch nicht aktualisierten Servers sehr gering wird, weshalb push und pull immer kombiniert werden sollten.

Speziell für Datenbanken wurden leichte Abwandlungen dieser Protokolle entwickelt. Grundsätzlich lässt sich dabei unterscheiden zwischen Eager und Lazy synchronisation (synchron und asynchron) sowie zwischen primary update und update everywhere. Es ergeben sich 4 Kombinationen:

- eager, primary: Keine Inkonsistenzen, lange Updatezeit

- eager, update everywhere: Inkonsistenzen möglich, Deadlocks möglich, symmetrischer Ansatz
- lazy, primary: Inkonsistenzen möglich, kürzere Antwortzeiten
- lazy, update everywhere: Inkonsistenzen möglich, Zurückrollen kann erforderlich werden, kürzere Antwortzeiten, symmetrischer Ansatz

Praktisch implementiert werden solche Konsistenzprotokolle durch mehrere Komponenten.

1. Interception der Client-Call im Stub
2. Replica-Management: Anlegen und ändern von Replikaten und deren Rollen (Primary, Backup)
3. Replikationslogik (Implementierung des eigentlichen Protokolls)
4. Fehlererkennung (Node- oder Link-Fehler)
5. Reliable Multicasting
6. Transaktionsunterstützung

Fault Tolerance

Begriffe

Dependability meint die Eigenschaft eines Systems, Failures zu verhinden, die häufiger oder schwerwiegender sind als akzeptiert werden kann. Die ist eng verbunden mit Fault Tolerance, was bedeutet dass ein System auch dann noch sein Service fehlerfrei anbieten kann, wenn Faults im System vorhanden sind (diese müssen maskiert werden). Das ist nur durch Redundanzen möglich, weshalb Fault Tolerance und Redundanz gewissermaßen Synonyme sind.

Anforderungen an fehlertolerante Systeme sind:

- Availability: Wahrscheinlichkeit der Verfügbarkeit zu einem gegebenen Zeitpunkt
- Reliability: Zeitspanne zwischen 2 Ausfällen
- Safety: Keine katastrophalen Folgen von Failures
- Maintainability: Fehler müssen schnell behoben werden können
- Integrity: Es dürfen keine ungültigen Systemzustände (Errors) angenommen werden

Aus dem Bereich Security kommt noch hinzu:

- Confidentiality: Keine unberechtigten Lesezugriffe auf Daten oder Nachrichten
- Integrity: Speziell im Bereich Security meint es, dass ungültige Zustände nicht von unberechtigten Personen hergestellt werden dürfen
- Availability: Für berechtigte Benutzer

Dependable Systems werden bedroht durch:

- Faults: Defekte innerhalb oder außerhalb einer Komponente
- Errors: Ungültige Systemzustände, die potentiell zu Failures führen
- Failures: Fehlverhalten des Systems, also Abweichungen vom erwarteten Verhalten

Faults direkt zu erkennen ist oft schwierig. Meistens beobachtet man lediglich Failures, von denen man dann auf Errors schließt (z.B. mittels Debuggen, wo man versucht die Errors zu reproduzieren) und schließlich die Ursachen für diese Errors (also die Faults) indirekt aufdeckt.

Errors können im Sinne der Error Propagation zu weiteren Errors oder auch zu Failures führen.
Failures eines Systems sind aus Sicht eines darauf aufbauenden Systems Faults.

Erreichen von Fehlertoleranz

Mittel zur Unterstützung der Fehlertoleranz sind:

- Fault Prevention: Versuchen Faults zu vermeiden, z.B. durch Qualitätssicherung
- Fault Tolerance: Die Möglichkeit weiterzuarbeiten trotz Faults
- Fault/Error Removal: Faults entfernen wenn diese aktiviert wurden (z.B. wenn die Toleranz überschritten wird)
- Fault/Error Forecasting: Konsequenzen von Faults frühzeitig abfangen, indem z.B. noch schlafende Faults bereits aufgedeckt und rechtzeitig korrigiert werden

Validierung meint das Bestärken des Vertrauens in die Tatsache, dass ein System im Bereich Security, Depandability und Funktionalität den Erwartungen entspricht.

Faults so gut wie möglich zu verhindern reicht aber noch nicht aus um von fehlertoleranten Systemen sprechen zu könen. Dazu müssen auch auftretende Fehler wieder korrigiert werden können. In fehlertoleranten Systemen sind üblicherweise folgende Phasen implementiert die zu einem fehlerfreien Systemverhalten führen:

- Error Detection: Fehler müssen überhaupt erkannt werden bevor diese korrigiert werden können. Dies kann entweder während einer Wartungsphase oder (vorteilhaft) zur Laufzeit erfolgen.
- Assessment und Limitation: Für erkannte Fehler muss die Schwere des Fehlers abgeschätzt und eine Error Propagation verhindert werden.
- Error Correction: Fehler müssen korrigiert werden um das System wieder in einen gültigen Zustand zu bringen. Dies kann durch Rollback, Rollforward (nur bei Systemen, bei denen ein gültiger Zustand in der Zukunft vorausgesagt werden kann) oder durch Verbergen des Fehlers (bei genug Redundanz) erfolgen.
- Fault Correction: Schließlich muss die Ursache des Errors auch behoben werden um ein Wiederholen des Errors zu verhinden. Dies kann durch Ausschluss der defekten Komponente (Isolation), durch Reinitialisierung oder ähnliches erfolgen.

Redundanzen

Um auch in der Zwischenzeit, während ein Fault vorhanden ist, fehlerfreies Verhalten garantieren zu können, sind Redundanzen notwendig. Diese können entweder zeitlich (z.B. mehrmalige Übertragung wie bei TCP), datentechnisch (Prüfsummen, Hamming-Distanzen, etc.) oder physikalisch (doppelte Ausführung eines Servers wie z.B. im DNS) realisiert sein.

Im Falle von physikalischen Redundanzen kann man unterscheiden ob die redundanten Systeme ständig im Gesamtsystem mitlaufen (aktiv) oder nur im Falle eines Faults angekoppelt werden (passiv). Bei passiven Systemen kann weiter unterschieden werden ob die (nicht angeschlossenen) redundanten Teile nebenbei mitlaufen (hot standby) oder komplett abgeschaltet sind (cold standby).

Ein Beispiel für eine redundante Systemarchitektur ist Triple Modular Redundancy. Dabei wird jedes Element einer sequentiellen Aneinanderreihung von Komponenten 3 Mal ausgeführt. Als Input bekommt jedes Element das (von einem Voter ermittelte) Mehrheitsergebnis der vorherigen Komponente. Die Voter müssen, da diese selbst fehlerhaft sein können, ebenfalls repliziert sein.

Auch Prozesse können mehrfach ausgeführt werden und ein Mehrheitsergebnis ermittelt werden. Man spricht in diesem Fall von Process Recilience. Solche Prozessgruppen können hierarchisch oder flach angeordnet sein, je nachdem ob es einen Koordinator gibt oder ob alle Prozesse gleichberechtigt sind und ein Gemeinschaftsergebnis ermitteln.

Wie viele Prozesse in eine Gruppe sind hängt einerseits davon ab wie viele Fehler man korrigeren können möchte (dies ist in den Requirements festgelegt), welche Art von Fehlern man korrigieren möchte und ob die Gruppe hierarchisch oder flach ist.

Fail-stop-Systeme (eine Komponente meldet sich ab wenn sie nicht mehr richtig funktioniert) oder fail-sailent-Systeme (eine fehlerhafte Komponente verschwindet einfach, z.B. bei einem Crash-

Failure) erfordern nur k+1 Prozesse um k Fehler korrigieren zu können. Fail-consistent-Systeme (fehlerhafte Komponenten arbeiten weiter, liefern aber falsche Ergebnisse – jedoch immer die gleichen falschen Ergebnisse bei gleichen Eingaben) erfordern 2k+1 Prozesse bei k zu korrigierenden Fehlern. Bei fail-inconsistent-Systemen (auch genannt Byzantinische Fehler – selbst bei gleichen Eingaben können die fehlerhaften Teile verschiedene Ergebnisse liefern) erfodern bei hierarchischen Gruppen mit funktionierendem Koordinator ebenfalls 2k+1 Prozesse, bei flachen Gruppen sogar 3k+1 Prozesse, wenn k Fehler korrigiert werden sollen.

Bei hierarchischen Gruppen muss beim Ausfall des Koordinators mittels eines Election Algorithmus (z.B. Bully- oder Token-Algorithmus) ein neuer Koordinator gewählt werden. Solange der Koordinator aber läuft, ist diese Struktur einfacher als die flache. Der Koordinator kann die Arbeit an alle anderen Prozesse aufteilen und schließlich die Ergebnisse einsammen und daraus die Mehrheit ermitteln. Von einer solchen Gruppe kann auch gesprochen werden wenn der Client selbst einfach mehrere Prozesse kontaktiert und daher selbst sozusagen als Koordinator fungiert.

Bei flachen Gruppen ist das Protokoll komplizierter. Hier schickt jeder Prozess (oder nur ein Koordinator, je nach Anwendung) sein Ergebnis an alle anderen. Somit hat jeder Prozess einen Vektor (evtl. auch nur einen 1-elementigen Vektor) von Ergebnissen, den er wieder an alle anderen Weiterschickt. Jeder von n Prozessen hat somit insgesamt n – 1 Vektoren mit je n Ergebnissen. Er kann diese Vektoren nun komponentenweise vergleichen und somit das Ergebnis jedes einzelnen Prozesses ermitteln. Z.B. kann in Komponente 1 eingesehen werden, zu welchem Ergebnis Prozess 1 gekommen ist. Diese Komponente muss nicht in allen Vektoren gleich sein, weil auch andere Prozesse, die ihren Vektor weiterschicken, diesen verändern können falls sie fehlerhaft sind. Eine Mehrheitsentscheidung ist erforderlich. Sobald jeder Prozess das Ergebnis jedes anderen Prozesses kennt, kann durch eine weitere Mehrheitsentscheidung ein Konsens über das Gesamtergebnis gefunden werden.

Solche Agreements in Gruppen mit möglicherweise fehlerhaften Prozessen sind notwendig um Fehler maskieren zu können: auch bei vorhandenen Fehlern kann noch immer ein gültiges Gesamtergebnis nach außen präsentiert werden. Man kann aber zeigen, dass solch ein Agreement nicht immer erreichbar ist. Es kommt ganz auf die angebotenen Services der darunter liegenden Übertragungsschicht an.

Im Falls von synchronen, geordneten Nachrichten ist ein Agreement immer möglich. Ebenso bei geordneten Multicast-Nachrichten (z.B. mittels totally-ordered Mutlicast). Als letzte Alternative ist es möglich ein Agreement zu erreichen, wenn synchrone Unicast-Nachrichten verschickt werden, solange diese geordnet sind.

In asynchronen Systemen ist bei unzuverlässigen Verbindungen oder bei möglicherweise fehlerhaften Prozessen grundsätzlich kein Agreement möglich.

Verlässliche Client/Server-Kommunikation

Bei Client-Server-Kommunikation können Fehler in verschiedenen Phasen passieren:

1. Server ist nicht erreichbar
2. Request geht verloren
3. Server stürzt während der Verarbeitung ab

4. Response geht verloren
5. Client stürzt währen der Verarbeitung ab

Die Phasen 2 – 4 sind aus Client-Sicht nicht unterscheidbar. Deswegen muss in diesem Fall unterschieden werden ob der Request noch einmal gesendet wird oder nicht. Beides kann zu Fehlern führen (doppelte oder gar keine Ausführung, was bei idempotenten Operationen beides falsch ist).

Als Abhilfe kann man Nachrichten künstlich idempotent machen indem man eine Sequenznummer anhängt. Jedoch ist in diesem Fall die exactly-once-Semantik (die Operation wird genau 1 Mal ausgeführt) auch nur dann garantiert, falls der Client die Anfrage im Extremfall endlos wiederholt schicken kann (was praktisch nicht möglich ist).

Das Problem am 5. Fall ist, dass eventuell Resourcen am Server verschwendet werden obwohl der Auftraggeber eines Requests inzwischen gar nicht mehr zum System gehört. Die noch laufenden Anfragen werden dann als Orphans des abgestürzten Clients bezeichnet und müssen entfernt werden. Alternativen dazu sind, dass die Prozesse periodisch beim Client um weitere Rechenzeit anfragen müssen (ist er in der Zwischenzeit abgestürzt, so ist das bei der nächsten Anfrage erkennbar), oder indem der Server für noch laufende Anfragen die Besitzer regelmäßig auffordert sich zu melden. Eine weitere Alternative ist es, dass der Client nach dem neuen Hochfahren alle Orphans aktiv entfernt.

Reliable Multicast

Grundsätzlich ist eine Nachricht reliable, wenn sie vom Empfänger sicher empfangen wird. Dies ist z.B. bei TCP der Fall. Reliable Multicast bedeutet, dass die Nachricht von allen Empfängern garantiert empfangen wird. Reliable Multicasts werden in vielen verteilten Algorithmen benötigt, z.B. im oben genannten Algorithmus zum finden eines Mehrheitsergebnisses in flachen Gruppen.

Garantiert kann das Empfangen werden, indem jede Nachricht von allen Empfängern bestätigt wird. Wird in einer gewissen Zeit kein Ack empfangen, so wird sie noch einmal übertragen.

Eine Optimierung sieht vor, dass lediglich verloren gegangene Nachrichten negativ bestätigt werden (Nack). Das ist aber nur möglich, wenn laufend Kommunikation mit steigenden Sequenznummern stattfindet.

Noch weiter optimiert kann der Algorithmus vorstehen, dass jeder Prozess eine Multicastnachricht neu ausschicken darf, und nicht nur der ursprüngliche Sender, oder dass das Nack nach Ablauf einer Zufallszeit ausgeschickt wird, so dass mehrere Nacks für die gleiche Nachricht wenn möglich vermieden werden.

Eine hierarchische Lösung sieht vor, dass die Gruppe der Empfänger in Subgruppen unterteilt wird, jede mir ihrem eigenen Koordinator. Die Gruppen werden hierarchisch angeordnet. Jeder Koordinator sendet eingehende Multicast-Nachrichten an alle direkt angeschlossenen Clients und an seine Subkoordinatoren weiter. Bestätigt werden Nachrichten von Clients bei ihrem Koordinator. Dieser bestätigt den Erhalt wiederum wenn er von allen Clients und Subkoordinatoren ein Ack erhalten hat. Sobald ein Koordinator von all seinen Subkoordinatoren und direkt angeschlossenen Clients erhalten hat, kann er die Nachricht aus seinem History-Buffer löschen. Andernfalls schickt er sie nach einer gewissen Zeit neu aus.

Ordnungen von Multicast-Nachrichten

Mutlicast-Nachrichten können in verschiedenen Reihenfolgen an den Empfänger zugestellt werden.

Ungeordnet: Es ist nichts definiert

FIFO: Nur Nachrichten vom selben Prozess werden in einer definierten Reihenfolge zugestellt (nämlich in der, in der sie gesendet wurden).

Causal: Kausal abhängige Nachrichten (2 Nachrichten sind abhängig, wenn sie im Sinne von FIFO abhängig sind oder wenn eine gesendet wurde nachdem eine andere empfangen wurde – siehe Vector Timestamps) werden in der Reihenfolge der Abhängigkeit zugestellt.

Orthogonal dazu gibt es totally-ordered Multicasts, bei dem die Nachrichten bei allen Prozessen in der gleichen Reihenfolge zugestellt werden.

Außerdem gibt es noch die Klassifizierung der virtual synchron Multicasts, was bedeutet, dass eine Nachricht entweder von allen oder von gar keinem der Empfänger empfangen wird. Wird das mit totally-ordered Multicasts kombiniert, so spricht man von Atomic Multicasts.

Totally-ordered Multicasts können durch eine Holdback-Queue implementiert werden. Diese buffert eingehende Nachrichten und ordnet sie nach einem Zeitstempel von einer logischen Uhr (z.B. ein Lamport-Timestamp). Jeder Prozess bestätigt den Erhalt einer Nachricht allen anderen über einen Mutlicast. Eine Nachricht wird von einem Prozess erst dann von der Holdback- in die In-Queue verschoben, wenn sie sich ganz oben in der Holdback-Queue befindet und außerdem bereits von allen anderen bestätigt wurde. Dadurch ist garantiert, dass die Nachrichten von allen Prozessen in der gleichen Reihenfolge empfangen werden, es wird aber noch nicht nichts darüber ausgesagt, in welcher Reihenfolge (FIFO, Causal) sie empfangen werden, und auch nichts darüber, ob sie überhaupt von allen empfangen werden (virtual synchron Mutlicasting). Totally-Ordered Causal Ordering kann z.B. mittels Vector-Timestamps kombiniert mit Ack-Multicasts erreicht werden.

Virtual Synchron Multicasting kann wie folgt realisiert werden. Jeder Prozess in der Gruppe buffert sämtliche Mutlicast-Nachrichten. Sobald ein Prozess die Gruppe verlassen will, beitreten will oder den Ausfall eines Prozesses enteckt hat schickt er eine View-Change-Nachricht an alle in der Gruppe. Das veranlasst die Prozesse sämtlichevon ihnen gebufferten Nachrichten in die Gruppe auszuschicken und mit einer Flush-Nachricht abzuschließen (was bedeutet, dass der Prozess keine gebufferten Nachrichten mehr hat). Damit wird gewährleistet, dass vor dem View Change alle Nachrichten an alle Prozesse in der Gruppe gehen, falls diese mindestens einen Prozess erreicht haben (eventuell haben sie, als sie ursprünglich ausgeschickt wurden, nicht alle Prozesse erreicht, weil der Sender abgestürzt ist bevor er den Multicast abschließen konnte). Sobald ein Prozess die Flush-Nachricht von allen anderen erhalten hat kann er sicher sein dass er keine an diese Gruppe adressierten Nachrichten verpasst hat und kann somit den View Change durchführen.

Distributed object-based Systems

Definition

Das grundlegende Item bei solchen Systemen ist das Objekt. Traditionellerweise handelt sich sich dabei, so wie etwa auch beim File und beim Document, um ein Item, das in nicht-verteilten Systemen verwendet wurde. Ziel von solchen Systemen ist daher die Unterstützung der Verteilungstransparenz.

Beispiele

Bei Beispiel für ein object-based Distributed System ist CORBA. CORBA selbst ist nur eine Spezifikation für eine sprachübergreifende object-based Middleware, von der es mehrere Implementierungen wie z.B. JacORB gibt. Kern ist der sogenannte Object Request Broker (ORB), der serverseitig eingehende Requests zum passenden (portable) Object Adapter weiterleitet. Darauf setzen statische und dynamische Skeletons auf, welche die eigentlichen Objekte (Servanten) aufrufen. Clientseitig bauen auf ORB direkt statische (aus der IDE generierte Stubs) und dynamische Method Invocation Interfaces auf. Die IDE (Interface Definition Language) ist eine neutrale Sprache zur Beschreibung der Interfaces eines verteilten Objekts. Daraus können mit speziellen IDE-Compilern die Stubs für verschiedene Programmiersprachen generiert werden. Man erspart sich damit die manuelle Implementierung von standardisierten Stubs, die sich lediglich in den Prototypen der Methoden unterscheiden, sonst aber immer gleich aussehen.

Interfaces werden in einem sogenannten CORBA Interface Repository abgelegt. Dieses Repository kann für dynamische Methodenaufrufe verwendet werden. ORB selbst verwendet es ebenfalls.

Wichtige CORBA-Services:

- Naming: Auffinden von Objekten
- Trading: Damit können Services eines bestimmten Typs angeboten und von anderen Prozessen aufgefunden werden, ohne dass deren genauer Name bekannt sein muss.
- Transaction: Mehrere Aufrufe auf verteilten Objekten werden entweder alle oder ger nicht durchgeführt.
- Persistency: Mit CORBA verteilte Objekte können persistiert werden
- Security: Authentifizierung, Autorisierung
- Property: Objekten können Attribut/Wert-Paare zugeordnet werden

CORBA verwendet sogenannte language-specific Interfaces. Das bedeutet, dass die in IDL beschriebenen Interfaces zuerst in Interfaces der jeweiligen Programmiersprache und diese dann mittels Sprachecompiler in binären Code übersetzt werden.

DCOM verwendet dagegen sogenannte binary interfaces. Dabei wird das in IDL beschriebene Interface direkt in binären Code übersetzt, der in einer Memory Map für jede angebotene Methode einen Pointer auf die Implementierung dieser Methode (in einer beliebigen Programmiersprache die den erwarteten Aufrufkonventionen entspricht) speichert.

RMI ist ähnlich wie CORBA, unterstützt aber nur Java. Lokale Objekte werden dabei durch Serialisierung übergeben, Remote Objekte grundsätzlich per Implementation Handle.

Enterprise Java Beans ist eine weitere objektbasierte Technologie, bei der Objekte (EJBs) in einem Container auf dem Object Server gehalten werden. Der Zugriff darauf wird über verschiedene Mechanismen wie JDBC, RMI und JNDI ermöglicht.

Komponenten

Objekte haben oft nicht die geforderte Granularität für Einheiten für die Wiederverwendung. Ein weiterer Schritt ist daher die Wiederverwendung ganzer Komponenten. Komponenten werden im Unterschied zu Objekten oft von Drittherstellern angeboten, können in Binärform wiederverwendet werden, können als Black-Box verwendet werden und kapseln eine abgeschlossene Funktionseinheit.

Object-based Messaging

Eine Mischung aus object-based und message-orientated Middleware ist object-based Messaging. Dabei werden zwar serverseitig klassische Methodenaufrufe durchgeführt, und aus Serversicht können solche Methoden auch wie gewohnt implmenetiert werden, das Runtime-System am Client ersetzt jedoch einen gewöhnlichen Aufruf durch Message-Processing.

Der Proxy wird dermaßen abgeändert, dass er keine Returnwert und keine Out-Parameter liefert. Stattdessen gibt es eine Callback-Funktion, die von der Middleware aufgerufen wird, sobald eine Antwort vom Server eingetroffen ist. Diese erhält als Parameter den Returnwert und die Out-Parameter.

Alternativ dazu ist auch eine Polling-Funktion denkbar, mit der die Applikation regelmäßig auf eingetroffene Antworten des Servers prüfen kann. Diese Polling-Funktion liefert den Returnwert und die Out-Parameter selbst als Out-Parameter, falls eine Antwort auf einen vorhergehenden Call eingetroffen ist.

Dokumentenbasierte Distributed Systems

Das zentrale Konzept ist hier das Dokument, auf das von den verschiedenen Clients zugegriffen werden kann. Das bekannteste System dieser Art ist das WWW.

Neben statischen Dokumenten werden meist auch noch dynamisch generierte unterstützt. Das kann zum Beispiel mittels CGI-Scripts erfolgen (Common Gateway Interface), die serverseitig in einem separaten Prozess ausgeführt werden, den Markup-Code (HTML oder XML) generieren und an den Webserver übergeben, der diesen an den Client weiterleitet.

Webservices erweitern das Konzept auf Services, die nicht (so wie beim WWW) direkt vom Enduser in Anspruch genommen werden, sondern von anderen Applikationen. Das bedeutet es werden maschinenlesbare Services angeboten. Als Zugriffsprotokoll für Webservices (aber auch zum Datenaustausch zwischen verschiedenen Prozessen) wurde SOAP entwickelt (Simple Object Access Protocoll), das jedoch selbst auf http aufbaut. Es wird damit einfach XML-Code in einer http-Nachricht verschickt. Die Nachricht spezifiziert dabei genau welches Service (per RPC) aufzurufen ist, was vom Server interpretiert werden kann. Die Antwortnachricht verpackt die Ergebnise des Aufrufs wieder als SOAP-Nachricht. Alternativ zu http kann auch noch SMTP als Trägerprotokoll für SOAP verwendet werden, was dann aber eher für Kommunikation zwischen menschlichen Usern verwendet wird statt für Prozesse.

Insgesamt stellt SOAP gekoppelt mit Webservices ein virutelles Komponentenmodell zur verfügung, indem Services, die traditionell von Komponenten angeboten werden, nun in einem verteilten System über dokumentenbasierte Systeme zur Verfügung gestellt werden.

SOAP kann aber auch für andere Zwecke, etwa zum Zugriff auf Datenbanken verwendet werden. Anstatt direkt den Datenbankzugriff mittels JDBC zu ermöglichen (z.B. aus Sicherheitsgründen) kann bei Verwendung von SOAP genau spezifiziert werden welche Aktionen erlaubt und welche verboten sind.

Distributed coordination-based Systems

Definition

Im Unterschied zu anderen Typen von verteilten Systemen (object-based, document-based, etc.) wird hier die Verteilung nicht verborgen sondern ist dem System immanent. Das ermöglicht bessere Skalierbarkeit und Offenheit. Es wird ganz bewusst auf die Verteilung von Rechenleistung (computation) Wert gelegt, wobei die Prozesse koordiniert werden müssen (coordination).

Die Koordination zwischen den Prozessen kann auf verschiedene Weise erfolgen, je nachdem ob die einzelnen Prozesse referentiell und/oder zeitlich ge- oder entkoppelt sind.

Beides gekoppelt: Direkte Kommunikation, z.B. mittels TCP-Verbindungen oder RMI.

Nur referentiell gekoppelt: Mailbox-Systeme

Nur zeitlich gekoppelt: Event-basierte Systeme (auch genannt Meeting-basierte Systeme) mittels Publish/Subscribe-Mechanismus.

Beides entkoppelt: Generative Communication mittels Shared-Dataspaces in denen Tupel abgelegt und von anderen Prozessen wieder ausgelesen werden können.

Beispiele

Jini setzt Generative Communication um. Es speichert Tupel von Java-Objekten in einem Shared Dataspace (JavaSpace), aus dem sie von interessierten Prozessen durch Matching mit einem Template (ebenfalls ein Tupel aus Objekten, das aber nicht vollständig ausgefüllt sein muss) wieder abgefragt werden können. Das Abfragen kann dabei auch das gelesene Tupel auch gleich aus dem Datastore entfernen (take-Operation). Neben dem Zugriff auf JavaSpaces unterstützt Jini auch noch Naming und Security (Authentification, so dass nur berechtigte Prozesse auf den Datastore zugreifen können).

ein anderes Beispiel ist ein sogenanntes Distributed Shared Memory. Das ist ein gemeinsamer Arbeitsspeicher, der konzeptionell geshared wird, d.h. Schreibzugriffe von einem werden durch Lesezugriffe von einem anderen Prozess berücksichtigt. Das Runtime System sorgt dafür, dass die Daten abgeglichen werden (in Wirklichkeit wird dazu Message Passing verwendet).

Quellen

[1] Distributed Systems: Principles and Paradigms Second Edition – Andrew S. Tanenbaum, Maarten VanSteen
[2] Slides zum Buch – Andrew S. Tanenbaum, Maarten VanSteen
[3] Folien zur Vorlesung – Dr. Karl Göschka, TU Wien