Think different - Think bigger

Markus Völter, voelter@acm.org, www.voelter.de
Eberhard Wolff

Neue Softwarearchitekturen und Infrastrukturen geben uns die Möglichkeit in ein neues IT Zeitalter vorzustoßen. Um das Potential dieser Entwicklungen nutzen zu können, müssen Softwarearchitekten aber auch mit Gewohntem brechen.

Um Enttäuschung zu vermeiden, soll hier vorweggenommen werden, dass dieser Artikel keine Lösungen bietet, ja noch nicht mal versucht sie zu liefern. Ziel dieses Artikels ist es, aufzuzeigen, dass es neben den derzeitigen Mainstream Enterprise- und Verteilungstechnologien noch andere Lösungsansätze gibt, die der steigenden Komplexität des Internets möglicherweise besser gewachsen sind.

Um sehen zu können, wieso die Zukunft so radikal anders aussieht und damit ein so bedeutender Wechsel bevorsteht, sollten wir zunächst den aktuellen Zustand moderner Technologien betrachten.Tut man dies, so wird man im Enterprise Umfeld Technologien wie Enterprise JavaBeans, CORBA und .NET finden. Man findet Projekte die sich damit beschäftigen, Content ins Internet zu bringen und Legacy Systeme zu integrieren. Moderne Middleware Technologien stellen ohne Zweifel eine wichtige Errungenschaft dar und helfen bei der Entwicklung aktueller Systeme, vor allem bei der Realisierung der nötigen Skalierbarkeit und Ausfallsicherhheit. Aber bei genauer Betrachtung stellt man fest, dass diese Technologien nicht wirklich ein neues Paradigma darstellen. Letztendlich sind sie zwar mehrere Evolutionsschritte vom guten alten RPC entfernt und entsprechend komfortabler und mächtiger, aber revolutionär sind sie sicherlich nicht: Das heist, das grundlegende Primzip ist noch immer synchrones RPC. Allerdings haben durch das Thema Middleware und Integration von Legacy Systemen verteilte Technologien ihren Weg in den Mainstream gefunden und stellen nun tatsächlich Schlüsseltechnologien dar, die einer breiteren Masse von Enticklern zugänglich sind.

Losgelöst von Softwarearchitekturen haben sich die Infrastrukturen ebenfalls weiterentwickelt. Der Schritt von der Insellösung bei der Rechner nicht oder nur lokal vernetzt waren, zum globalen Netzwerk ist vollzogen, Bandbreite wird in naher Zukunft kein allzu grosses Problem mehr sein. Nichts desto trotz sind die meisten Anwendungen noch eher der LAN Denkweise verhaftet und nichts anderes als Client-Server-Anwendungen mit mehr oder weniger intelligenten Clients. Auch Webtechnologien sind „nur“ Client-Server-Systeme. Auch hier sind die Forschritte eher zaghaft.

Doch wo sind nun die Technologien die die IT Welt erschüttern werden? Derzeit sind einige interessante Ansätze erkennbar, die mehr oder weniger bekannt sind. Wir möchten in diesem Artikel auf die folgenden vier Punkte eingehen: Peer-to-Peer Systeme, Agenten,

Mobiler Code, und fehlertolerante, skalierbare Systeme. Der letzte Abschnitt wird dann aufzeigen, wie diese Technologien zusammen genutzt werden können.

Peer-to-Peer Systeme

Eine der Technologien, die heute schon einen Paradigmenwechsel ankündigen, ist die Peer-to-Peer Technik. Sie ist aufgrund von „anarchistischen Systemen" wie Napster derzeit nicht ganz unumstritten. Aber die Tatsache, dass es schon eine marketing-tauglich „Hype-Abkürzung" gibt, prophezeit der Technologie eine grosse Zukunft, auch wenn sie Techniker eher mit Misstrauen denn mit Zuversicht erfüllt. Doch was steckt hinter P2P, wie lässt sich dieses Paradigma definieren? Eine Recherche im Internet bringt eine Reihe sich stark voneinander unterscheidender Definitionen zu Tage. Das Destillat dieser Definitionen könnte heißen:

> *Ein System ist ein Peer-to-Peer System wenn die beteiligten Komponenten einen gleichberechtigten Stellenwert einnehmen können.*

Die aktuell so beliebten Filesharing-Ausprägungen wie Napster, Gnutella, Freenet und Co. sind nur eine Spielart von P2P. Bei Napster kann sowieso bezweifelt werden ob es sich überhaupt um P2P System handelt, denn es gibt beispielsweise eine zentrale Benutzerverwaltung. P2P kann also auch bedeuten, dass mehrere gleichberechtigte aktive Einheiten für einen bestimmten Zeitraum zusammen arbeiten. In [1] wird das Computerspiel Doom als Beispiel für eine nicht offensichtliche P2P Anwendung aufgeführt. Man kann auch behaupten, dass Seti@Home eine P2P Anwendung ist, denn die Berechnung findet verteilt mit gleichberechtigten Systemen statt, allerdings gibt es einen dedizierten Server für die Datenhaltung.

In [1] wird erläutert wieso es für echtes P2P wichtig ist, die streng hierarchische (und starren) Namesstrukturen von DNS zugunsten sich dynamisch ändernder „Netzwerkkonfigurationen" aufzugeben. Daraus leiten die Autoren einen Lackmus Test ab, mit dem bestimmt werden kann ob ein System ein P2P System ist. In P2P Systemen finden also mehrere Systeme auf unterschiedliche Arten für einen begrenzten Zeitraum zusammen um eine Aufgabe zu erledigen und werden nicht zu einem festen System konfiguriert, wie sonst üblich.

All das beantwortet allerdings noch nicht die Frage, ob P2P eben doch nur ein Hype ist, oder eine zukunftsträchtige Technologie. Dabei muss beachtet werden, dass P2P in seinen Grundzügen nicht wirklich neu ist. Schon in den Windows-Anfangszeiten und sicherlich auch schon zuvor gab es Peer-to-Peer Netzwerke, die allerdings reines Filesharing zur Verfügung gestellt haben. Sie haben sich allerdings nicht auf breiter Front durchgesetzt. Im Gegenteil: Man hat erkannt, dass es ein Problem ist, Resourcen "normaler" Clients gemeinsam zu nutzen, denn die Clients können weder die Performance noch die Verfügbarkeit eines dedizierten Servers bieten. In der Praxis zeigt sich noch ein anderes Phänomen, dass gegen die Idee "viele kleine sind besser als ein grosser" spricht: Es gibt

eine Konsolidierung hin zu Mainframes oder großen UNIX-Servern. Oft setzten sich die Lösungen gegen einen Cluster von kleinen Maschinen durch oder übernehmen sogar Aufgaben von kleineren Servern.

Dies scheint zunächst gegen P2P zu sprechen. Die neue Generation von P2P Systeme unterscheiden sich hier jedoch in einem wichtigen Punkt: Sie beziehen die Möglichkeit, dass ein teilnehmendes System ausfällt, in die Planung mit ein. Schon durch die obige Definition ist die eingebaute Ausfallsicherheit angedeutet: Wenn alle Systeme gleichberechtigt sind, gibt es keinen Single Point of Failure und damit eine höhere Verfügbarkeit des Gesamtsystems: Einem "reinen" P2P System ist der Ausfall eines einzelnen Rechners völlig egal. Möglicherweise wird ein kleiner Teil des Systems unbenutzbar (der ausgefallene Rechner), aber die Verfügbarkeit des Gesamtsystems ist davon nicht betroffen.

Warum sollte diese Technologie ein neuer Trend sein? Wenn man jenseits des IT-Tellerrands nachschaut, wird man schnell erkennen, dass sehr viele Biologische Systeme und Sozialsysteme der obiger P2P Definition genügen. Weder das Gehirn noch unsere Gesellschaft sind starr und hierarchisch strukturiert sondern hoch dynamische Systeme, die immer neue Konfigurationen ausbilden. Eine interessante Analogie ist die Folgende: Wenn eine Ameise zum Beispiel Futter findet, geht sie zum Ameisenhaufen zurück und hinterlässt dabei eine Spur aus Pheromonen Duftstoffen). Andere Ameisen werden davon angezogen und beteiligen sich dann am Abtragen dieser Futtermittel. Da die Pheromone mit der Zeit verdunsten, werden kurze Strecken mit vielen Ameisen regelrecht "markiert", während dies bei längeren Strecken kaum passiert, da die Pheromone über eine weitere Strecke verteilt sind. Dadurch werden nähere Futtervorkommen zuerst abgebaut. Gleiches wird oft bei P2P-Systemen verwendet: Eine Suchanfrage wandert von Knoten zu Knoten, wobei ein „Hop-Count“ heruntergezählt wird. Ist dieser bei null angekommen, wird die Suchanfrage nicht weiter transportiert. Damit wird sichergestellt, dass nähere Knoten die Suchanfrage zuerst beantworten, und das Netz nicht vollkommen mit Suchfragen „zugemüllt“ wird.

Wenn sich also diese Art von Systemdesign in der Natur durchgesetzt hat um Systeme hoher Komplexität hand zu haben, ist es vermutlich ein vielversprechender Ansatz auch für komplexe ITSysteme.

Agenten

Eine weitere Technologie, die bei komplexen Systemen Verbesserung verspricht sind Agenten. Diese Technologie macht zwar derzeit keine so starken Schlagzeilen, ist aber mindestens ebenso interessant ist wie P2P. In [2] ist ein guter Überblick über Agententechnologien zu finden und deshalb wollen wir an dieser Stelle auf eine ausführliche Beschreibung verzichten. Agenten sind schon lange in der akademischen Welt Ziel umfangreicher Forschungsvorhaben, es sind auch schon einige Systeme im praktischen Einsatz. Eine Definition eines Agenten könnte die folgende sein:

> *Ein Agent ist ein Softwarebaustein, der autonom (also ohne zusätzlichen Benutzerkontakt) einen Dienst durch Kommunikation mit anderen Agenten oder anderer Software erbringt.*

Im Gegensatz zu „normaler" Software haben Agenten keinen festen Algorithmus implementiert. Sie besitzen einige grundlegende Mechanismen, um z.B. Informationen von ihrere Umgebung aufzunehmen, oder auf die Umgebung einzuwirken. Diese ergatterten Informationen werden vom Agent entsprechend bestimmter grundlegender Verhaltensmuster verarbeitet. Zum Zwecke des Informationsaustauschs zwischen Agenten (oder mit anderer Software) wurden spezielle Protokolle entwickelt, die nicht nur die reine Information übertragen, sondern auch „Gefühlszustände" und ähnliches, um die Kommunikation aussagekräftiger zu machen und die zwischenmenschlichen Kommunikationsmittel Gestik, Tonfall, etc. nachzubilden. KQML ist ein solches Protokoll.

Eine typische Aufgabe für einen Agent könnte z.B. die folgende sein: ihr Agent verhandelt zum einen mit den Verschiedensten Anbietern um einen günstigen Preis zu bekommen, fragt auf der anderen Seite Ihren Fernseher ob der potentiell neue Videorecorder zu dem Fernsehgerät passt, dass sie bereits besitzen und man mit einer Fernbedienung beide Geräte steuern kann. Danach handelt er noch mit der Bank eine für sie günstige Ratenfinanzierung aus.

Weitaus interessantere Chancen ergeben sich aus der Kollaboration von vielen Agenten, die zusammen ein bestimmtes Ziel erreichen, ohne dass jeder Agent den Algorithmus für die Gesamtlösung kennen müsste. Der einzelne Agent kennt einige grundlegende Verhaltensmuster, die, wenn sie von vielen kollaborierenden Agenten angewandt werden, eine sehr komplexes Problem lösen können. Schauen wir uns ein weiteres Mal die Natur als Vergleich an. Die Ameisen aus obigem Beispiel kennen nur die folgenden einfachen Verhaltensmuster zum Thema „fressen" und „Futter beschaffen".

1. Wenn Nahrung gefunden, gehe zurück zum Ameisenhaufen und hinterlasse eine Pheromonspur.
2. Folge Pheromonen.

Obwohl dies nur sehr einfache Verhaltensmuster sind, hat die konsequente Anwendung dieser Muster zur Folge, dass ein Ameisenstamm Futter welches sich in der Nähe des Baus befindet gemeinsam abholen.

Interessante Experimente bietet hier [3] von der Gruppe am MIT, die auch Lego Mindstorms mitentwickelt hat. Eine interessantes Buch zu dem Thema ist John Holland's „Hidden Order" [4].

Mobiler Code

In traditionellen Systemen ist ein Stück Programm ortsfest, d.h. es wird auf einer Maschine gestartet, und bleibt während seines gesamten Lebenszyklusses auf dieser Maschine.

Ausgetauscht werden hingegen nur die Daten, die von diesem Programm bearbeitet werden. Dies ist bei den derzeitigen typischen Anwendungen meist noch kein Problem, da sie typischerweise auf einer Client/Server Architektur beruhen, wo auf dem Server praktisch beliebig viel Rechenpower zur Verfügung steht, und die Endgeräte homogen sind (d.h. Es sind praktisch alles Webbrowser, die HTML verstehen). Wenn man das Problem nun aber im Zusammenhang mit kleineren web-enabled Devices unter die Lupe nimmt, so stellt man fest, dass dies in Zukunft nicht mehr so sein wird. Nehmen wir das Beispiel von mobilen Agenten, einer Kombination aus Agententechnologie und mobilem Code.

Ein solcher mobiler Agent wandert tatsächlich von Plattform zu Plattform und agierte somit weitgehend losgelöst von Systemgrenzen. Stellen Sie sich vor, sie konfigurieren einen Agenten auf Ihrem Mobiltelefon und schicken ihn danach ins Internet damit er seine Aufgaben erfüllt. Dort kann er von System zu System wandern, mit anderen Agenten verhandeln und letztendlich mit beliebigen am Internet angeschlossenen Geräten kooperieren und deren Resourcen nutzen. Dies hat zwei bedeutende Vorteile. Erstens können sie Ihr Handy wieder vom Netz nehmen, und zweitens hat der Agent auf größeren Maschinen viel mehr Rechenpower zur Verfügung als auf einem Handy.

Dies ist auch ein Hinweis auf einige Probleme, die im Zusammenhang mit mobilen Agenten noch zu lösen sind: Das System, auf dem der Agent läuft, muss vor übermäßigem Gebrauch der Resourcen geschützt werden, während der Agent gegen Ausspionieren geschützt sein muss, insbesondere wenn er geheime Informationen wie eine Kreditkartennummer zu Bezahlung von Waren mitführt. Desweiteren müssen die Host-Systeme, auf denen die Agenten laufen, noch zumindest so zuverlässig sein, dass nicht irgendwo unbeabsichtigt ein Agent auf der Strecke bleibt.

Was macht nun mobilen Code aber so zukunftsträchtig? Wie bereits oben beschrieben, hat sich die Hardwareinfrastruktur in den letzten Jahren derart verändert, dass man von einer globalen Vernetzung sprechen kann. Nicht nur riesige Mengen an Rechenpower auch gigantische Mengen an Speicherplatz sind Grossteils ungenutzt. Diese zu nutzen und zu verwalten wird mit Hilfe traditioneller Systeme sehr komplex bzw. fehleranfällig. Aktive verteilte Dienste die miteinander kommunizieren können hier mit eher der Komplexität umgehen.

Eine wichtige Grundlage für derartige Systeme sind Bytecode sprachen wie Java, die weitestgehende Hardware Unabhängigkeit bieten, und – zumindest im Falle von Java – auch schon einige Sicherheitsmechanismen mitbringen. Wir Software Techniker haben damit die Möglichkeit bekommen, Programme sicher auf unterschiedlichster Hardware ablaufen zu lassen. Erst damit ist mobiler Code (und mobile Agenten im Besonderen) im großen Rahmen realisierbar geworden.

Flexible Infrastrukturen

Der Begriff „spontaneous networking“, also das spontane, ungeplante „Zusammenfinden“ von Systemen, wurde durch Sun's Jini das erste mal so richtig populär. Die dahinter stehende Idee ist die folgende: Es ist nicht möglich, ein komplexes System von vornherein fest zu konfigurieren, weil einige Teile des Systems zeitweise offline sein werden, oder ausfallbedingt nicht verfügbar sind. Was man braucht um mit solchen Systemen trotzdem arbeiten zu können ist, eine dynamische Konfiguration der Dienste.

Nehmen wir an, sie möchten von Ihrem PDA eine Adressliste ausdrucken. Ihr PDA fragt im Netz herum, wer einen Druck-Dienst zur Verfügung stellt, und verwendet diesen, um seine Adressen auszudrucken. In traditionellen Systemen sucht er dazu in einem „Dienstverzeichnis“ (auch Trader genannt) nach einem passenden Dienstprovider. Was passiert aber, wenn dieser Trader ausfällt? Dies ist ein klassischer Single-Point-of-Failure, der in zuverlässigen, verteilten Systemen auf jeden Fall vermieden werden muss. Daher verwendet Jini ein Konzept, bei dem ein Dienst immer bei mehreren Lookup-Servern registriert wird – wenn einer davon ausfällt, haben andere die Information immer noch, das System ist resistent gegen Ausfälle einzelner Komponenten.

Desweiteren muss sichergestellt werden, dass ein Dienstprovider die Chance hat, sich kontrolliert wieder aus dem System zurückzuziehen. Ein Client darf einen Dienst also nicht „für immer“ verwenden, wenn er ihn per Lookup gefunden hat. Statt dessen bekommt er ein „Lease“, welches ihm für eine bestimmte Dauer das Recht zuspricht, den Dienst zu benutzen. Nach Ablauf der im Lease angegebenen Frist muss es erneuert werden. Wenn der Dienstprovider dies nicht tut, weil er sich eben aus dem System zurückziehen möchte, muss sich der Client nach einem neuen Dienstprovider per Lookup erkundigen.

Die oben genannten Eigenschaften eines Systems sind bereits sehr wichtig, um eine flexible, verteilte, und ausfallredundante Infrastruktur zu ermöglichen. Jedoch reicht dies noch lange nicht aus. Stellen Sie sich vor, sie möchten von IhremPDA aus ein Backup „im Netz“ ablegen. Dazu holen sie sich per Lookup einen Dienst, der es Ihnen ermöglicht, Daten abzulegen und später wieder auszulesen. Soweit kein Problem. Allerdings hat ein Backup nur Sinn, wenn Sie es im Notfall auch wieder zurückspielen können, d.h. sie müssen später den gleichen Datenablegedienst wiederfinden, bei dem Sie ihre Daten abgelegt haben. Wenn der zwischenzeitlich nicht mehr im Netz ist, oder seine Platte gecrasht ist, oder ... Daher muss für solch einen Fall auch die Datenhaltung komplett verteilt sein, sodass bei Ausfall eines Systems die Daten immer noch vorhanden sind. Diese im Netz redundante Datenhaltung muss für die Clients transparent passieren, ähnlich wie das bei Festplatten-RAID-Controllern passiert. Die University of Berkeley hat im Rahmen des Ninja Projekts [4] verschiedene verteilte Datenstrukturen implementiert, die dieser Anforderung genügen. Beipielsweise wurde eine verteilte Hashtable implementiert, die redundant sehr grosse Datenmengen bei erstaunlich guter Performance speichern kann.

Das Ninja Projekt weist unserer Meinung nach auch noch in verschiedenen anderen Hinsichten einen interessanten weg auf. Ziel des Projektes ist die Erstellung einer verteilten Infrastruktur, wobei die darauf laufenden Dienste skalierbar, fehlertolerant, hoch verfügbar und von verschiedensten Clients aus ansprechbar sein sollten. Ninja basiert auf den folgenden Prinzipien:

- *Partitionierung des Zustandes:* Harter Zustand, also solcher, der nicht ohne weiteres neu erzeugt werden kann, wird in skalierbaren, verteilten, kontrollierten Umgebungen gehalten, den sogenannten Bases. Weicher Zustand (z.B. client-seitiger EMail Cache) kann irgendwo im Netz liegen, wenn er verloren geht, kann er neu erzeugt werden.
- *Operatoren und Pfade:* Alle Dienste nehmen einen typisierten Datenstrom an, verarbeiten ihn auf definierte Weise und geben einen anderen Datenstrom aus. Solche Dienste heissen Operatoren, und sie können manuell oder automatisch verkettet werden um komplexere Datenverarbeitungsschritte zu ermöglichen. Operatoren in einem solchen Pfad können auf verschiedenen Maschinen laufen.
- *Automatische Service-Komposition:* Durch Ausstattung der Services mit entsprechenden Metadaten wird es möglich, dass automatisch ein Pfad von „PDA-Daten“ zu „Postscript-Drucker“ erzeugt wird. Die Ninja-Infrastruktur kümmert sich um die automatische Erstellung und um das Wiederaufräumen eines solchen Pfades.
- *Aktive Proxies:* Zwischen den Endgeräten und den stabilen Bases kann auf jedem Knoten ein anonymer Aktiver Proxy liegen, im Rahmen eines Pfades Operatoren ausführt, die nicht direkt auf harten Zustand zugreifen können müssen. Je nach Endgerät können solche Proxies auch auf dem Endgerät selbst vorhanden sein. Mobiler Code erlaubt das Ausführen beliebiger Operatoren in einem Aktiven Proxy. Da sie keinen harten Zustand besitzen, stellt ein Ausfall eines Aktiven Proxies kein Problem dar – der Operator wird einfach auf einem anderen neu instantiiert.

Für weitere Details von Ninja sei auf die entsprechende Webseite unter [5] verwiesen.

Resümee

Wenn man nun die oben geschilderten Technologien betrachtet und deren Gemeinsamkeiten oder Grundprinzipien herausstellt, so könnte das folgendermassen aussehen. Aufgrund der steigenden Komplexität (und der wachsenden Grösse) des Netzes wird es verschiedenartige, auch unzuverlässige Systeme geben. Es ist allerdings möglich, aus einer grossen Zahl unzuverlässiger Systeme im Endeffekt wieder ein zuverlässiges Gesamtsystem zu erstellen. Um die Komplexität in den Griff zu bekommen, muss man dafür sorgen, dass die einzelnen Bausteine einfache Dinge ausführen (also eine geringe

Komplexität aufweisen), sie aber durch verschiedenartige Mechanismen zu einem „grossen Ganzen“ zusammengefügt werden können.

Welche der vorgestellten Technologien das Rennen machen wird, oder ob es eine Kombination oder eine ganz andere Technologie sein wird ist schwer absehbar. Das einzige was sicher sein dürfte ist dass ein Paradigmenwechsel notwendig ist und kommen wird. Nur so wird die steigende Komplexität beherrschbar bleiben.

[1] http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html

[2] James Odell: Key issues for Agent Technologie; Journal of Object-Oriented Programming, Januar/2001

[3] http://lcs.www.media.mit.edu/groups/el/Projects/starlogo/information/index.html

[4] John H. Holland, Hidden Order – How Adaption builds Complexity, Addison-Wesley 1996

[5] Ninja Project at http://ninja.cs.berkely.edu