Torben Weis, Christopher Boelmann

Automatismen zur Strukturbildung und Selbst-Organisation in verteilten Systemen

Erhaltung der Konsistenz in einer verteilten virtuellen Welt

In der Informatik gibt es gegenwärtig zwei fundamental unterschiedliche Ansätze, um große verteilte Systeme, die aus vielen einzelnen miteinander vernetzten Computern bestehen, zu betreiben: Client-Server-Systeme und Peer-to-Peer-Systeme. Bei Client-Server-Systemen stehen alle Server unter der Kontrolle einer Instanz (etwa einer Firma wie Google, Amazon oder Apple, oder einer staatlichen Instanz). Im Gegensatz zur zentralisierten Verwaltung von Client-Server-Systemen organisieren sich Peer-to-Peer-Systeme (kurz: P2P) selbstständig. Alle Rechner (meist PCs), welche die P2P-Software ausführen, schließen sich zu einem P2P-Netz zusammen. In diesem Netz gibt es allerdings keine zentrale Administration, die den einzelnen Rechnern ihre Aufgaben zuschreibt und deren Funktion überwacht. Vielmehr müssen die Rechner sich automatisch organisieren und die anfallenden Aufgaben untereinander aufteilen. In kleinen Netzen ist dies relativ einfach, da alle Rechner sich miteinander ins Benehmen setzen können. In großen Systemen mit mehreren tausend PCs ist es aber sinnlos zu versuchen, den Überblick über das gesamte Netz zu erlangen. Dies würde Minuten dauern und in dieser Zeit haben schon wieder Rechner das Netz verlassen und andere sind beigetreten. Der Algorithmus, der das Netz organisiert, kann also nicht von globalem Wissen ausgehen. Jeder Rechner verfügt demnach nur über lokales Wissen, aufgrund dessen er seine Entscheidungen treffen muss. Die Summe aller lokalen Entscheidungen soll aber dazu führen, dass das gesamte P2P-Netz global betrachtet die erwartete Struktur bildet.

Die Algorithmen, die ein verteiltes System zusammenhalten, machen Annahmen über die Kausalität von Ereignissen und die Konsistenz der Realität, die sich oft nicht mit der alltäglichen Erfahrung der Menschen decken. Das kann zu einem unerwarteten Verhalten aus der Sicht des Nutzers führen. Um dies zu illustrieren, verwenden wir als Anwendungsbeispiel eine virtuelle Welt, wie sie auch Basis der meisten Computerspiele ist. Der Nutzer erwartet beim ersten Betreten der Spielwelt, dass diese virtuelle Welt sich analog zur realen verhält. Wir werden zeigen, dass virtuelle Welten oft anders ticken und dies auch für den Nutzer wahrnehmbar ist.

Ein weiteres Problem bei der automatischen Organisation von PCs in einem P2P-Netz ist die Unzuverlässigkeit der einzelnen PCs (Peers). Ein PC kann

jederzeit ausgeschaltet werden, abstürzen oder vom Netz getrennt werden. Daher ist es zwingend notwendig, dass ein P2P-Netz selbststabilisierend ist. Ein P2P-Netz muss sich nicht nur selbstorganisieren, um eine Struktur auszubilden. Es muss auch jederzeit verkraften können, dass Rechner ausfallen und damit die Struktur beschädigt wird. Daher besprechen wir in diesem Artikel sowohl die Selbststabilisierung und darauf aufbauend die Selbstorganisation.

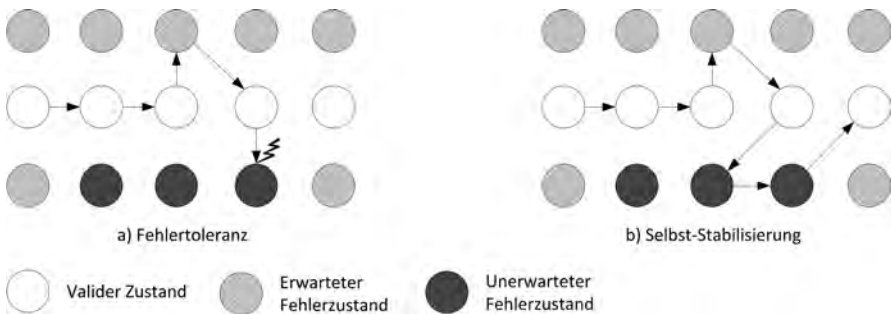
Mit Algorithmen zur Selbstorganisation und Selbststabilisierung können Peers automatisch eine stabile globale Struktur erzeugen, auf deren Basis man dann eine virtuelle Welt betreiben kann. Neben den schon erwähnten Besonderheiten bezüglich Kausalität und Konsistenz (die den verwendeten Algorithmen geschuldet sind), kann es aber auch zu absichtlichen Regelverletzungen kommen, wenn sich betrügerische Spieler hiervon einen Vorteil erhoffen. Daher werden wir im abschließenden Teil unseres Beitrags darauf eingehen, wie sich ein selbstorganisierendes System gegen Betrüger schützen kann und wir werden die Grenzen dieses Schutzes aufzeigen.

Selbststabilisierung

Beim Programmstart befindet sich ein Programm in einem validen (d. h. korrekten) Zustand. Das bedeutet, dass die aktuell gespeicherten Werte und die erhaltenen Eingaben die für die Ausführung erwarteten Werte haben, und dass das Programm korrekt ausgeführt wird. Durch auftretende Fehler während der Laufzeit (z. B. eine Unterbrechung der Netzwerkverbindung) können bei der Programmausführung allerdings Fehler auftreten, durch die eine korrekte Ausführung nicht immer gewährleistet ist.

Der klassische Ansatz für den Umgang mit Fehlern, die zur Programmlaufzeit auftreten, ist, die Fehler abzufangen und im Programmcode explizit eine Fehlerbehandlung festzulegen. Diese Herangehensweise hat den Vorteil, dass das Verhalten des Programms zur Laufzeit zu jedem Zeitpunkt, auch im Fehlerfall, vorhersehbar ist. Allerdings versagt das Konzept der klassischen Fehlerbehandlung im Falle von Fehlerzuständen, die bei der Entwicklung nicht in Betracht gezogen wurden. Hierbei kann zwischen zwei Fehlerzuständen unterschieden werden, die sich aus einem unerwarteten Fehler ergeben können: Zum einen kann beispielsweise eine fehlerhafte Berechnung zu einem fatalen Fehler (z. B. einer Division durch Null) und somit zu einem Programmabbruch führen (siehe Abbildung 1a). Zum anderen kann das System aber auch durch einen Fehler (z. B. fehlerhaft übertragene Werte) in einen Zustand gelangen, in dem das Programm zwar fehlerhafte Werte enthält, aber dennoch weiter ausgeführt wird. In diesem Falle werden die Fehler nicht entdeckt und es wird von einer korrekten Programmausführung ausgegangen, obwohl das Programm durch die fehlerhaften gespeicherten Werte dauerhaft falsche Ausgaben liefert.

Einen Gegensatz zu fehlertoleranten Systemen stellen selbststabilisierende Systeme dar, deren Konzept auf die Veröffentlichung „Self-Stabilizing Systems in Spite of Distributed Control“[1] des niederländischen Informatikers Edsger W. Dijkstra im Jahre 1974 zurückgeht. Ein selbststabilisierendes System konvergiert aus jedem möglichen Zustand in konstanter Zeit k automatisch wieder zu einem validen Zustand (d. h. es stabilisiert sich), solange es sich bei dem Fehler um einen *temporären* Fehler handelt (z. B. hitzebedingte Fehlberechnungen oder Netzwerkfehler). Sollte es sich nicht um einen temporären Fehler handeln, würde der Zustand des Systems unabhängig von den Selbststabilisierungs-Maßnahmen immer wieder in einen Fehlerzustand übergehen, ohne dass sich das Programm stabilisieren könnte, da die Fehlerquelle bestehen bleibt. Der Vorteil der Selbststabilisierung nach konstanter Zeit wird allerdings auf Kosten der Vorhersehbarkeit des Programmablaufs erkauft. Da auftretende Fehler nicht verhindert (d. h. nicht abgefangen) werden, kann während der Stabilisierungszeit keine Aussage über die Korrektheit der aktuellen Berechnungen und somit auch nicht über die Korrektheit des Programmverhaltens gemacht werden. Die Selbststabilisierungs-Eigenschaft garantiert nur, dass sich das Programm nach konstanter Zeit wieder korrekt verhalten wird, und dass die Einflüsse durch den Fehler wieder behoben werden (d. h., dass wieder ein valider Zustand erreicht wurde; siehe Abbildung 1b). Die Entscheidung, ob ein System selbststabilisierend oder fehlertolerant realisiert werden soll, ist dementsprechend eine Abwägung von Ausfallsicherheit gegen die gesicherte Korrektheit des Programmablaufs zu jedem Zeitpunkt.



1 – Programmzustands-Übergänge im Fehlerfall

Selbstorganisation

Automatismen zur Selbstorganisation setzen Mechaniken zur Selbststabilisierung ein und sorgen dafür, dass ein System eine Struktur annimmt, die den ge-

[1] Edsger W. Dijkstra, „Self-Stabilizing Systems in Spite of Distributed Control“, in: *Communications of the ACM* 17, 11 (1974), S. 643-644.

gebenen Regeln entspricht und ohne Einflussnahme von außen eine solche Struktur aufrechterhält. Im Kontext von verteilten Systemen in der Informatik kann solch eine Struktur beispielsweise einen Zusammenschluss von Computern zu einer Ringstruktur bedeuten. Im Falle einer Ringstruktur müsste sich jeder Computer einen Vorgänger und einen Nachfolger aussuchen. Dabei trifft jeder Computer die Entscheidung für einen Vorgänger und einen Nachfolger basierend auf seinem lokalen Wissen. Die Entscheidungen werden periodisch wiederholt, damit die Systemstruktur immer wieder an die aktuellen Gegebenheiten und das aktualisierte lokale Wissen angepasst wird. Auf globaler Ebene konvergiert das System durch die lokalen Entscheidungen zu einem System mit der vorher festgelegten Struktur und durch die periodische Wiederholung werden auch Veränderungen wie hinzukommende und ausfallende Computer in die Struktur einbezogen. Ein Beispiel für die Umsetzung selbstorganisierender Systeme sind strukturierte P2P-Systeme wie z. B. Chord[2]. In strukturierten P2P-Systemen ist es wichtig, dass eine eindeutige Struktur erreicht wird, um ein effizientes Auffinden von Daten innerhalb des Systems zu ermöglichen. Da allerdings kein einzelner Peer (ein Computer innerhalb des P2P-Systems) globales Wissen besitzt, werden Automatismen zur Selbstorganisation eingesetzt, die dafür sorgen, dass nach einer gewissen Zeit das P2P-System eine gewisse Struktur annimmt.

Das Forschungsprojekt *Peers@Play*[3] (*P@P*) beschäftigt sich mit der Erforschung und Umsetzung eines Massively Multiplayer Online Games (MMOG) ohne zentralen Server. Ein MMOG ist ein Spiel, in dem sehr viele Spieler gemeinsam in einer virtuellen Welt mit den Spielinhalten interagieren können. Das wohl bekannteste Beispiel ist das MMORPG (Massively Multiplayer Online Role-Playing Game) *World of Warcraft*[4], das 2004 vom Spieleentwickler Blizzard Entertainment veröffentlicht wurde und das zeitweise (Oktober 2010) von bis zu 12 Millionen Spielern weltweit gespielt wurde.[5]

In einem serverbasierten MMOG gibt es eine Menge von Spieleservern, die vom Entwickler betrieben und gewartet werden. Diese Server kennen und verwalten die komplette virtuelle Welt, den Zustand der Objekte innerhalb der Spielwelt und die Spieler-Benutzerkonten. Zum Zustand der Spielwelt gehören u. a. Informationen über die Spieler und *Non-Player Characters* (NPCs), die sich in der Welt befinden, die Reihenfolge der Aktionen/Ereignisse, die in der Spielwelt stattgefunden haben und die aktuellen Positionen von Objekten. Jeder

---

[2] Ion Stoica/Robert Morris/David Liben-Nowell/David R. Karger/M. Frans Kaashoek/Frank Dabek/Hari Balakrishnan, „Chord: a Acalable Peer-to-Peer Lookup Protocol for Internet Applications", in: *IEEE/ACM Transactions on Networking* 11, 1 (2003), S. 17-32.

[3] Forschungsprojekt *Peers@Play*, online unter: http://www.peers-at-play.org/, zuletzt aufgerufen am 29.09.2013.

[4] *World of Warcraft*, online unter: http://eu.battle.net/wow/, zuletzt aufgerufen am 29.09.2013.

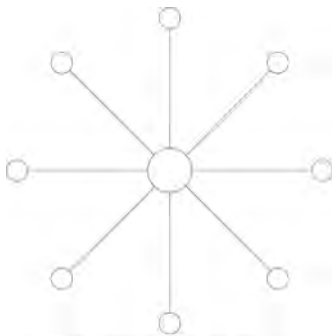
[5] „World of Warcraft Subscriber Base Reaches 12 Million Worldwide", in: *Blizzard Entertainment*, online unter: http://us.blizzard.com/en-us/company/press/pressreleases.html?id=2847881, zuletzt aufgerufen am 29.09.2013.

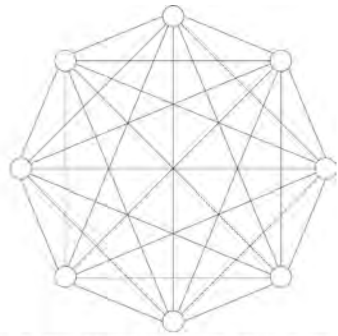
Spieler hat eine Verbindung zum Server, die während des Spielens nicht unterbrochen werden darf und tauscht mit dem Server Informationen aus (siehe Abbildung 2a). Über diese bestehende Verbindung sendet jeder Spieler seine Aktionen (z. B. seine Bewegungen, Angriffe oder Interaktionen) an den Server. Der Server wertet die empfangenen Aktionen aus und überprüft, ob die Aktionen den Regeln entsprechen, wie sich die Position des Spielers durch seine Bewegung verändert hat und ob die Angriffe oder Interaktionen andere Spieler beeinflussen. Solch eine Beeinflussung könnte beispielsweise sein, dass ein Spieler einem anderen Spieler Schaden zufügt oder dass ein Spieler versucht mit einem anderen Spieler zu handeln. Bei gleichzeitigen Aktionen hat der Server des Weiteren die Möglichkeit, die Aktionen in eine feste Reihenfolge zu bringen. Nach der Verarbeitung der Aktionen durch den Server sendet dieser die Ereignisse zurück an alle Spieler, die von den Aktionen betroffen sind. Bei den Spielern wird anschließend die Anzeige der Spielwelt anhand der empfangenen Ereignisse aktualisiert. Diese Ereignisse können u. a. Veränderungen im Zustand der Spielwelt sein oder aber auch die veränderten Positionen der eigenen Spielfigur und anderer Charaktere in der Spielwelt. Dadurch, dass der Server globales Wissen hat und den gesamten Zustand der Spielwelt und der Spieler kennt, kann gewährleistet werden, dass es immer eine eindeutige Reihenfolge für alle Ereignisse in der Spielwelt gibt. Somit wird sichergestellt, dass alle Spieler die gleiche konsistente Welt sehen und dass sich alle Spieler an die Regeln der Spielwelt halten.

Peers@Play

Im Rahmen des *Peers@Play*-Projektes wird untersucht, wie ein MMOG ohne einen zentralen Server funktionieren kann. Die Motivation des Projektes sind u. a. die hohen Kosten für die Serverbetreiber, die es nicht ermöglichen, auch kleinere innovative MMOGs rentabel zu betreiben. Um auf einen zentralen Server verzichten zu können, nutzt der *Peers@Play*-Prototyp ein P2P-System zum Austausch von Nachrichten zwischen den einzelnen Spielern und zur Speicherung des Zustands der Spielwelt. Dadurch trägt jeder Spieler automatisch dazu bei, die Infrastruktur des Spiels (die bisher von den Servern geleistet wurde) zu verstärken, indem er die Ressourcen seines Computers (unter anderem CPU-Rechenzeit und Speicherplatz) mit in das Netzwerk einbringt. Je mehr Spieler spielen, desto leistungsstärker wird die P2P-Infrastruktur, d. h. das System skaliert automatisch mit der Anzahl der Spieler. In einem P2P-System sind alle Teilnehmer (Peers) gleichberechtigt und bauen untereinander Verbindungen auf. Abbildung 2b stellt zur Veranschaulichung ein vollvermaschtes P2P-System dar. In der Realität baut jeder einzelne Peer allerdings nur Verbindungen zu wenigen anderen Peers auf. Die Kommunikation mit den restlichen Peers des P2P-Netzes wird dabei über die verbundenen Peers weitergeleitet. Da es keinen zentralen Server gibt, der globales Wissen hat, müs-

sen die Aufgaben des MMOG-Spiele-Servers von den einzelnen Peers im Netzwerk übernommen werden.





2 – Netzwerk-Topologien Vergleich

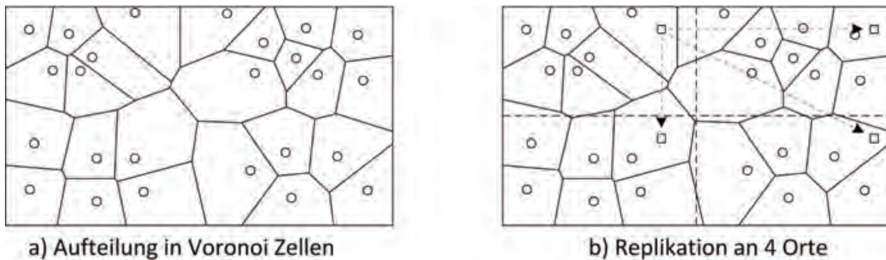
Wie bereits beschrieben, nutzen P2P-Systeme Automatismen, um die Struktur des Netzwerkes ohne Eingriffe von außen zu erhalten. Der Verzicht auf einen zentralen Server bringt verschiedene Probleme für das MMOG mit sich, auf die im Folgenden weiter eingegangen wird:

1. Es gibt ohne einen Server keinen zentralen Punkt im System, an dem der Zustand der Spielwelt sowie der in der Spielwelt befindlichen Objekte gespeichert werden kann.
2. Es fehlt eine zentrale Instanz, die über die Reihenfolge von Ereignissen in der Spielwelt entscheidet.
3. Ohne einen zentralen Punkt, an dem alle Ereignisse der Spielwelt und Interaktionen der Spieler verarbeitet und auf die Spielwelt angewendet werden, kann es passieren, dass sich die lokalen Zustände der Spielwelt bei den Spielern voneinander unterscheiden.
4. Ohne eine zentrale Instanz, die die Korrektheit und Regelkonformität von Spielerinteraktionen überprüft, ist eine Durchsetzung von Regeln nicht leicht zu realisieren.

Innerhalb eines P2P-Systems ist jeder Peer für einen Teil der gespeicherten Daten zuständig. Die Zuständigkeiten für die einzelnen Teile der Spielwelt müssen so unter den vorhandenen Spielern aufgeteilt werden, dass zum einen die Last der Spielberechnungen fair auf die Spieler verteilt wird und dass die im P2P-System gespeicherten Informationen anschließend auch von anderen Spielern aufgefunden werden können. Das bedeutet auf der einen Seite, dass die Orte, an denen Daten zu finden sind, durch ein festes Schema errechenbar sein müssen. Auf der anderen Seite muss verhindert werden, dass ein *böser Spieler*, also ein Spieler, der beabsichtigt gegen die Regeln zu verstoßen oder das Spiel negativ zu beeinflussen, sich frei auswählen kann, für welche Daten er zuständig ist. Ansonsten hätte er beispielsweise die Möglichkeit, sich *so*

innerhalb des P2P-Systems zu positionieren, dass er verantwortlich für den Zustand seines eigenen Inventars ist. In diesem Falle könnte er die Daten, die sein Inventar beschreiben, verändern und z. B. neue Gegenstände hinzufügen.

In *P@P* werden die Daten der Spielwelt auf einen zweidimensionalen Zustandsraum abgebildet. Haben Daten Ortsbezug, so ergibt sich ihre Position im Zustandsraum trivial. Die Position von ortsunabhängigen Daten lässt sich dadurch ermitteln, dass man den Namen des gesuchten Datums in eine Einwegfunktion (eine Hash-Funktion) gibt, die als Ausgabe einen Zahlenwert hat, der als Position im Zustandsraum interpretiert wird. Auch die einzelnen Peers werden auf eine Position im Zustandsraum abgebildet. Jeder Peer ist im Zustandsraum verantwortlich für die Daten, die ihm gemäß einer euklidischen Distanz am nächsten liegen. Durch diese Annahme und die zufällige Verteilung der Peers im Zustandsraum ergibt sich eine *Voronoi*-Zerlegung des Zustandsraumes in unterschiedlich große Zellen[6], die in Abbildung 3a beispielhaft dargestellt ist.



3 – Zerlegung des Zustandsraums in Voronoi-Zellen

Algorithmen zur Selbstorganisation sorgen dafür, dass die Struktur des Systems (d. h. die korrekte Zerlegung des Zustandsraumes in *Voronoi*-Zellen und die gerechte Verteilung der Zuständigkeiten für Teile der Spielwelt auf die einzelnen Spieler) auch in Anbetracht von neu hinzukommenden und von ausfallenden Peers erhalten bleibt. Dazu tauschen die Peers mit ihren direkten Zellennachbarn Statusinformationen aus. Anhand dieser Informationen treffen die Peers lokale Entscheidungen darüber, für welche Positionen sie selbst zuständig sind und für welche Positionen ihre direkten Nachbarn zuständig sind. Sobald ein neuer Peer *A* in das System eintritt, bestimmt er seine Position und sucht den Peer *B*, der für diese Position zuständig ist. Peer *B* benachrichtigt seine Nachbarn über die Position des neuen Peers *A* und alle betroffenen Peers berechnen ihre Zuständigkeitsbereiche neu. Falls ein Peer ausfällt, wird dies von seinen Nachbarn bemerkt, die periodisch versuchen Informationen auszutauschen. Wird ein Ausfall entdeckt, werden die Nachbarn des ausgefallenen Peers über den Ausfall informiert und alle betroffenen Peers errechnen ihre

---

[6] Vgl. Franz Aurenhammer, „Diagrams – a Survey of a Fundamental Geometric Data Structure“, in: *ACM Computing Surveys* 23, 3 (1991), S. 345-405.

Zuständigkeit neu. So wird erreicht, dass sich das System automatisch und ohne explizite Einwirkung von außen an Veränderungen anpasst, d. h. die Struktur wahrt und repariert.

Wenn jedes im Zustandsraum gespeicherte Datum nur von genau einem Peer verwaltet würde, gingen die gespeicherten Informationen bei einem Ausfall des Peers verloren. Für die Persistenz der Spielwelt hätte das fatale Folgen. Aus diesem Grund wird dafür gesorgt, dass Daten repliziert und von unterschiedlichen Peers verwaltet werden. Eine Möglichkeit zur Umsetzung der Replikation ist es, den Zustandsraum aufzuteilen und jedem Datum statt einer einzelnen Position mehrere Positionen zuzuweisen. Abbildung 3b zeigt die Aufteilung des Zustandsraumes in vier Teile. Dies bedeutet, dass ein Datum, das nur zu einer Position in der virtuellen Welt gehört, an mehreren (hier: vier) Positionen innerhalb des Zustandsraumes gespeichert wird und dort von anderen Peers abrufbar ist. Wenn ein Peer ausfällt, kann sich der Peer, der die Zuständigkeiten übernimmt, die Replikate der verlorenen Daten zusammensuchen.

Damit Peers die Daten, die innerhalb des Peer-to-Peer-Systems gespeichert wurden, schnell auffinden können, werden Wegfindungs-Algorithmen (*Routing Algorithms*) genutzt. Um eine Zielposition im Zustandsraum zu erreichen, kann eine Nachricht (z. B. eine Anfrage für bestimmte Informationen) immer zu einem direkten Nachbarn weitergeleitet werden, der näher an der Zielposition ist. Sollte der entsprechende Nachbar nicht selbst für die Position verantwortlich sein, so wiederholt er die Weiterleitung der Nachricht. Im beschriebenen Zustandsraum mit einer *Voronoi*-Zerlegung würde man allerdings im Durchschnitt $O(\sqrt{n})$ Schritte benötigen, um eine Position zu erreichen (wobei n die Anzahl der Peers im System ist). Dies würde für 40.000 Peers, die sich im Spiel befinden, bereits eine durchschnittliche Strecke von 100 Schritten bedeuten. Um die Anzahl benötigter Schritte zu reduzieren, nutzen Routingalgorithmen Abkürzungen. Jeder Peer führt eine Sprungtabelle mit bekannten Peers an bestimmten Punkten im System. Diese Punkte sind so verteilt, dass mit jedem Routingschritt der Abstand zum Ziel mindestens halbiert werden kann. Somit ergibt sich eine Routingkomplexität von durchschnittlich $O(\log(n))$ Schritten zum Ziel.[7] Für 40.000 Peers ergibt sich somit beispielsweise eine erwartete Anzahl von 15 Schritten.

[7] Vgl. Sebastian Holzapfel/Sebastian Schuster/TorbenWeis, „VoroStore – a Secure and Reliable Data Storage for Peer-to-Peer-Based MMVEs“, in: *International Conference on Computer and Information Technology*, 2011, S. 35-40.

Kausalität und Konsistenz

Das zweite Problem, das sich aus dem Verzicht auf einen zentralen Server ergibt, ist die Festlegung einer Reihenfolge für alle Ereignisse, die sich in der Spielwelt zugetragen haben.

Den meisten Menschen erscheint es so, dass jede Menge von Ereignissen, die sich beobachten lässt, chronologisch eindeutig sortiert werden kann. Das bedeutet, dass der Zeitpunkt zweier Ereignisse *a* und *b* entweder exakt gleich war oder notwendigerweise ein Ereignis vor dem anderen stattfand. Diese Sortierung von Ereignissen nach der Zeit erfolgt mithilfe der (physikalischen) Uhrzeit. Wird eine Menge von Ereignissen derart sortiert, genügt diese Sortierung auch den Erfordernissen der Kausalität, d. h. wenn ein Ereignis *a* Einfluss auf Ereignis *b* hatte (*b* ist also kausal abhängig von *a*), so muss der Zeitpunkt von *a* vor dem des Ereignisses *b* liegen, denn ansonsten hätte die Zukunft Einfluss auf die Vergangenheit. Allerdings können Ereignisse auch kausal unabhängig voneinander sein. Dies gilt insbesondere für Ereignisse, die auf Rechnern eines verteilten Systems auftreten. Insofern diese Rechner nicht miteinander kommunizieren, sind ihre Ereignisse kausal unabhängig. Die Ereignisse dürfen in der zeitlichen Sortierung also in beliebiger Reihenfolge vorkommen, denn die Kausalität kann nicht verletzt werden.

Auf das Beispiel *P@P* übertragen bedeutet das, dass zwei Spieler, die sich gegenseitig mit ihren Spielfiguren zu beeinflussen suchen (etwa mit Waffen), dies kausal unabhängig voneinander tun, denn eine kausale Abhängigkeit wird erst erzeugt, nachdem die Rechner der Spieler Nachrichten ausgetauscht haben. In zentral verwalteten virtuellen Welten nimmt der Server eine Ordnung der Ereignisse vor. Er versieht jede eingehende Nachricht mit einem eindeutigen Zeitstempel. Dadurch ergibt sich eine global eindeutige totale Ordnung auf allen Ereignissen. Diese geordneten Ereignisse bilden die virtuelle Realität, d. h. im Idealfall sehen alle Spieler dieselben Ereignisse in derselben Reihenfolge. In *P@P* gibt es aber keine solchen Server, die alle Ereignisse sammeln, zeitlich ordnen und dann an alle Spieler weiter verteilen.

Die scheinbar nahe liegende Lösung, jedes Ereignis mit der Uhrzeit des jeweiligen Rechners zu versehen, ist leider technisch nicht sinnvoll. Computeruhren sind niemals für längere Zeit synchron, da die Uhrzeiten immer auseinander driften.[8] So könnte es passieren, dass ein Rechner *A* ein Ereignis zur Zeit t_1 erzeugt, eine Nachricht an Rechner *B* schickt, der daraufhin zum Zeitpunkt t_2 das Ereignis *b* auslöst. Es folgt, dass *b* kausal von *a* abhängig ist und demnach *a* vor *b* passiert sein muss. Wegen des Uhrendrifts kann es aber sein, dass t_1 einen späteren Zeitpunkt markiert als t_2. Daher kann man in einem vollständig dezentralen System keine perfekte physikalische Uhr annehmen. In Systemen mit zentralen Servern ist dies anders, denn dort werden alle Ereignisse an

8 Vgl. Leslie Lamport, „Time, Clocks, and the Ordering of Events in a Distributed System", in: *Communications of the ACM* 21, 7 (1978), S. 558-565.

die Server geschickt, welche sie zeitlich ordnen (z. B. nach dem Zeitpunkt ihres Eingangs beim Server) und dann an alle anderen Spieler verteilen. Wenn die Serveruhr zu langsam oder zu schnell läuft, ist das demnach kein Problem. Es ist nur wichtig, dass die Uhr des Servers eine streng monoton steigende Uhrzeit erzeugt.

In dezentralen, verteilten Systemen wird daher gerne eine logische Zeit konstruiert, die der Kausalität niemals widerspricht, d. h. wenn *b* kausal abhängig von *a* ist, dann ist die logische Zeit *C*(a) auch kleiner als *C*(b). Es gibt mehrere Arten von logischen Uhren. Die einfachsten sind die skalaren Uhren nach *Lamport*[9]. Eine Uhr ist hier ein diskreter Zähler. Bei jedem Ereignis wird die Uhr um eins erhöht. Schickt ein Rechner eine Nachricht, inkludiert er seine aktuelle logische Uhrzeit t_γ in die Nachricht. Empfängt ein Rechner eine Nachricht, so setzt er seine Uhrzeit t gemäß der Formel $t = \max(t, t_\gamma) + 1$. Dadurch ist sichergestellt, dass jedes Ereignis ausgehend vom Empfang der Nachricht eine höhere Uhrzeit hat als das Senden der Nachricht, denn das Empfangen ist vom Senden kausal abhängig.

Aus den technischen Gegebenheiten kann man erkennen, dass die Teilnehmer eines verteilten dezentralen Systems eine logische Zeit erzeugen müssen, weil Rechner keine perfekten physikalischen Uhren haben. Diese logischen Uhren decken sich aber nicht immer mit dem menschlichen Verständnis von Zeit. Das bleibt solange unproblematisch, wie der Nutzer den Unterschied weder wahrnimmt noch gezielt ausnutzen kann. Wendet man aber logische Zeit auf eine virtuelle Welt an, von der der Nutzer erwartet, dass sie sich an die physikalische Zeit hält, so wird der Unterschied zwischen den zwei Zeitbegriffen für den Nutzer spürbar und für Betrüger nutzbar.

Beispielsweise füge die Spielfigur *A* einer Spielfigur *B* eine tödliche Verletzung zu. Der Rechner von *A* schickt sodann eine Nachricht darüber an die Rechner aller anderen Spieler mit dem Zeitstempel t_A. Der Rechner von *B* könnte nun so tun, als hätte er diese Nachricht (noch) nicht gesehen. Daraufhin erzeugt er seinerseits eine Nachricht, in der steht, dass die Spielfigur *B* einen tödlichen Schlag gegen die Spielfigur *A* geführt habe. Eigentlich müsste diese Nachricht den Zeitstempel $t_B = t_A + 1$ tragen. Aber *B* betrügt und setzt $t_B = t_A$. Dadurch haben sich beide Spielfiguren scheinbar gleichzeitig gegenseitig verletzt, obwohl das faktisch nicht richtig ist. Der Betrug ist schwer nachzuweisen, weil es technisch im Internet keine Möglichkeit gibt, *B* nachzuweisen, dass er eine Nachricht von *A* erhalten hat, bevor er seine Nachricht *B* erzeugt und verschickt hat. Das liegt daran, dass Kommunikation im Internet so ähnlich funktioniert wie der Versand von Postkarten. Diese kommen bekanntlich *irgendwann* in *irgendeiner* Reihenfolge an (oder auch gar nicht) und der aktuelle Stand der Sendung kann nicht nachverfolgt werden. Wie sollte man da jemandem nachweisen, dass er eine gewisse Postkarte erhalten hat, bevor er seine eigene verfasst und verschickt hat?

---

[9] Lamport (1978), Time, Clocks, and the Ordering of Events in a Distributed System.

Ein weiteres Problem mit der Kausalität entsteht, wenn sich Ereignisse gegenseitig ausschließen. Beispielsweise liegt auf dem Boden ein Gegenstand. Zwei Spieler versuchen zeitgleich, den Gegenstand vom Boden aufzuheben. In der realen Welt kann dies nur einem gelingen, d. h. das Ereignis „*A* hebt den Gegenstand auf“ schließt das Ereignis „*B* hebt den Gegenstand auf“ aus. Wenn aber beide Spieler dies unabhängig voneinander an zwei Rechnern tun, so sind beide Ereignisse eingetreten und sie sind kausal unabhängig. Das widerspricht aber der Forderung, dass die Ereignisse nicht beide eingetreten sein können.

In virtuellen Welten versucht man daher, solche Exklusivität von Ereignissen nicht zuzulassen. Die einfachste Lösung besteht darin, dass ein Gegenstand, der vom Boden aufgehoben wurde, dort sofort wieder erscheint. Damit schließen sich die beiden Ereignisse nicht gegenseitig aus und es gibt keine Probleme mit der Kausalität. Dafür widerspricht das Wiedererscheinen eines Gegenstandes den Erwartungen des Nutzers. Dieser könnte das auch ausnutzen, indem er denselben Gegenstand mehrfach aufhebt. Handelt es sich dabei um Geld oder einen Wertgegenstand, könnte ein Spieler sich unbeschränkt bereichern. Daher ist diese einfache Lösung zwar konform mit der Kausalität, aber das Ergebnis ist der Spielmechanik abträglich.

Das Problem wird daher oft so gelöst, dass ein Gegenstand nach dem Aufheben durch Spieler *A* für diesen nicht mehr sichtbar ist. Auf die Sicht von Spieler *B* hat das aber keine Auswirkung. Er sieht nach wie vor den Gegenstand und hebt ihn auf, damit verschwindet er auch für Spieler *B*. Auch hier ist die Kausalität gewahrt (indem die Ereignisse nicht mehr kausal abhängig sind), dafür wird aber die Konsistenz verletzt, denn Spieler *A* und *B* haben Ereignisse beobachtet, die sich scheinbar gegenseitig ausschließen und dennoch sind sie beide eingetreten. Auf dieses Problem gehen wir im nächsten Abschnitt genauer ein. Es ist festzuhalten, dass Entwickler von virtuellen Welten aus technischen Gründen versuchen, spielinterne Kausalitäten zu vermeiden, da diese zu Problemen führen. Dadurch handelt man sich aber Probleme mit der Konsistenz ein.

Eine Momentaufnahme des Zustandes einer virtuellen Welt (Snapshot) ist durch eine Menge von Daten repräsentiert, die ihren Zustand beschreibt. In einem interaktiven System hingegen wird die Welt durch die Menge aller Ereignisse beschrieben, die in ihr ausgelöst wurden. Ausgehend vom initialen Zustand der Welt kann mit diesen Ereignissen der aktuelle Zustand der Welt ausgerechnet werden. Das Problem ist aber, dass sich zu einem physikalischen Zeitpunkt die Sichten der einzelnen Rechner auf die virtuelle Welt unterscheiden, denn einige Ereignisse reisen noch als Nachrichten durchs Netz, so dass sie einigen, aber nicht allen Rechnern bekannt sind. Dies führt zwangsläufig zu einer Inkonsistenz.

Der Effekt ist aber nicht zwingend störend. Wenn sich die Spieler nicht in Sichtweite zueinander aufhalten, werden sie gar nicht merken, dass ihre virtuelle Welt sich nicht synchron mit den Welten anderer Spieler ändert. Stellt

man aber beide Rechner nebeneinander, wird der Effekt sofort sichtbar. Um diesen Effekt zu minimieren, wird ein sogenannter *Local-Lag*[10] eingeführt. Sobald ein Spieler ein Ereignis auslöst, wird dieses nicht sofort ausgeführt, sondern erst an alle anderen Spieler verschickt. Nach einer Wartezeit von max. 100 bis 200 ms wird das Ereignis dann tatsächlich durchgeführt und angezeigt. Das Kalkül ist, dass zu diesem Zeitpunkt alle Rechner die Nachricht erhalten haben und die Ereignisse dann weitgehend synchron auf allen Rechnern ausgeführt werden.

Allerdings gibt es auch hier Probleme und Angriffsmöglichkeiten. Rechner mit einer besonders langsamen Verbindung zum Netzwerk (hohe Latenz) erhalten die Nachrichten zu spät. Daher sehen ihre Nutzer die Ereignisse anderer Spieler mit Verzögerung, was zu Nachteilen im Spiel führen kann. Im Gegensatz dazu sind solche Spieler im Vorteil, die die Ereignisse deutlich vor der Deadline des *Local-Lag* erhalten. Sie könnten die Ereignisse vorzeitig auswerten und haben einen Vorteil gegenüber den anderen Spielern, denn der Rechner kann gewissermaßen ‚in die Zukunft blicken', die sich als Liste von Ereignissen in seiner *Local-Lag*-Warteschlange darstellt. Nun könnte man den *Local-Lag* so weit erhöhen, dass auch die langsamsten Rechner die Nachrichten rechtzeitig erhalten. Das nutzt aber wenig, weil schnelle Rechner dann immer noch ‚in die Zukunft schauen' können und weil der Nutzer deutlich merken wird, dass nach einem Mausklick eine ganze Zeit nichts passiert, weil sein Ereignis erst einmal in der Warteschlange hängt. Das bedeutet, dass das maximale *Local-Lag*, das gewählt werden kann, durch die Wahrnehmung des Spielers beschränkt ist. Es wird ersichtlich, dass die Geschwindigkeit der Netzwerkverbindung einem Nutzer Vorteile bietet, die durch technische Maßnahmen nicht vollständig nivelliert werden können.

Selbst wenn alle Rechner untereinander in derselben Geschwindigkeit Nachrichten austauschen könnten (eine fiktive Annahme), gäbe es noch Probleme mit der Konsistenz. Als Beispiel denke man sich einen virtuellen Zug, der in einer virtuellen Welt auf eine Weiche zufährt. Die Spieler können die Weiche verstellen. In Anlehnung an das Beispiel mit dem Aufheben eines Gegenstandes kann die Situation eintreten, dass beide Spieler die Weiche zeitgleich in unterschiedliche Positionen bringen und zwar kurz bevor der Zug die Weiche erreicht. So könnte es dazu kommen, dass der eine Spieler den Zug nach links fahren sieht (weil er die Weiche entsprechend gestellt hat) und der andere sieht den Zug nach rechts fahren. Die Nachrichten über die Stellung der Weiche sind noch im Netzwerk unterwegs, während der Zug die Weiche schon passiert hat. Hier ist es nicht einfach damit getan, die Kausalität aufzuheben und den Fehler in der Konsistenz billigend in Kauf zu nehmen. Wenn ein Zug eine andere Richtung nimmt, kann das für das Spiel große Bedeutung

[10] Martin Mauve/Jürgen Vogel/Volker Hilt/Wolfgang Effelsberg, „Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications", in: *IEEE Transactions on Multimedia* 6, 1 (2004), S. 47-57.

haben, weil viele der folgenden Ereignisse kausal abhängig sind. Daher würde ein verteiltes Computerspiel solche Szenarien gar nicht erst erlauben. Es gibt also besonders zeitkritische Vorgänge in der realen Welt (Zug fährt über Weiche), die sich in der virtuellen Welt nicht abbilden lassen, weil Kausalität und Konsistenz nicht forciert werden können.

Ein weiteres interessantes Beispiel sind zwei Spieler, die gleichzeitig durch einen engen Spalt laufen, durch den sie nicht gleichzeitig nebeneinander gehen können. Auch hier kann es wegen mangelnder Kausalität dazu kommen, dass beide Spieler unabhängig voneinander durch den Spalt gehen. Dabei sieht sich jeder Spieler zuerst gehen, denn die Nachricht, dass der andere auch gegangen ist, erreicht ihn erst, nachdem er schon im Spalt ist. In diesem Fall kann man die Konsistenz opfern, d. h. man akzeptiert, dass die Spieler sich gegenseitig in unterschiedlicher Reihenfolge durch den Spalt gehen sehen. Die meisten Spiele gehen sogar noch weiter und erlauben gar nicht, dass Spieler gegeneinander prallen. Laufen zwei Spieler aufeinander zu, so gehen sie einfach geisterhaft durcheinander hindurch.

Auch hier gilt, dass zentrale Server die Situation verbessern können. Wenn beispielsweise ein Spieler durch den erwähnten Spalt gehen will, wird eine Nachricht an den Server geschickt. Der lässt den ersten passieren und bescheidet dem zweiten Spieler zu warten. Solange der Server nicht antwortet, geschieht auf dem Bildschirm nichts. Wenn der Server hinreichend schnell ist, wird der Nutzer den Zeitversatz kaum bemerken. In einem dezentralen, verteilten System hingegen gibt es keine zentrale Instanz, die kurzerhand die Spielfigur bestimmen kann, die zuerst durch den Spalt klettert. Hierfür wäre ein verteilter Konsensus[11] notwendig, der aber viel zu langsam ist. Außerdem könnten Betrüger auf die Idee verfallen, den verteilten Konsensus zu ihren Gunsten zu beeinflussen, um somit unliebsame Ereignisse einfach niederzustimmen.

Regelverstöße

Neben Problemen mit Zeit, Kausalität und Konsistenz muss ein dezentrales verteiltes System auch mit mutwilligen Regelverstößen umgehen können. Ähnlich wie bei einem Gesellschaftsspiel werden die Regeln von denen bestimmt, die spielen. Bei serverbasierten Spielen geben hingegen die Server die Regeln vor, an die sich alle Klienten halten müssen. Diese starke normative Instanz fehlt in P2P-Systemen. Die meisten Nutzer werden allerdings ein fertiges Programm herunterladen und dieses nicht modifizieren, so dass dessen Implementierung die Regeln implizit vorgibt. Dies ist allerdings nur eine schwache normative Instanz, denn ein erfahrener Programmierer kann das Pro-

[11] Vgl. Friedemann Mattern, „Verteilte Basisalgorithmen", in: *Informatik-Fachberichte* 226 (1989), S. 28-30.

gramm jederzeit modifizieren und die modifizierte Variante auch Nutzern zur Verfügung stellen, die über keine Programmierkenntnisse verfügen.

Daraus folgt, dass bestenfalls jene Regeln gelten können, die von der Mehrheit der Peers befolgt werden. In *Peers@Play* müssen wir daher davon ausgehen, dass mehr als die Hälfte der Nutzer bereit ist, von Betrug freiwillig abzusehen, denn ansonsten wäre der Betrug die Regel und damit wäre das Spiel nicht mehr interessant. Ob dies in der Praxis Bestand hätte, ist mangels praxisnaher Evaluation noch eine offene Frage.

Aber selbst wenn wir dies als gegeben annehmen, stellt sich noch die Frage, wie mit einer betrügerischen Minderheit zu verfahren ist. Regelverstöße werden irgendwann sichtbar und sei es nur, weil ein Spieler ungewöhnlichen Erfolg hat. Ein Mensch mag sich so von einem Regelverstoß überzeugen, aber ein unbestreitbarer Nachweis eines Verstoßes, der von Rechnern automatisch überprüft werden kann, ist schwieriger zu erlangen. Dieser ist allerdings notwendig, um einen Spieler aus dem P2P-Netz zu verbannen. Ein Ausschluss aus dem System kann nur funktionieren, wenn eine Mehrheit der Peers nicht mehr mit dem Betrüger kommuniziert. Allerdings ist der Betrug oft nur einer Minderheit offenbar, denn die Spieler verteilen sich in der virtuellen Welt und verarbeiten nur Ereignisse in ihrer Nähe, schließlich hat kein Peer eine globale Sicht auf das System. Regelverstöße können also nur von jenen wahrgenommen werden, die in der Nähe des Betrügers sind, und das ist global gesehen mit hoher Sicherheit nur eine Minderheit aller Spieler. Daher muss die Minderheit die Mehrheit von dem Vergehen überzeugen. Einfache Anschuldigungen genügen nicht, sonst könnte eine betrügerische Minderheit durch falsche Anschuldigungen regeltreue Spieler ausschließen lassen, bis schließlich die Betrüger die Mehrheit stellen.

In der Informatik bieten sich für einen sicheren Nachweis von Regelverstößen kryptografische Beweise an. Zum Beispiel könnte jeder Spieler von ihm ausgelöste Ereignisse digital signieren. Hat er seine Ereignisse nicht regelkonform erzeugt, kann man diese samt digitaler Unterschrift sammeln und im P2P-Netz verbreiten. Aufbauend darauf können alle Peers den Regelverstoß nachvollziehen und den Peer ausschließen. Allerdings kann man nur Ereignisse unterschreiben, die man ausgelöst hat, aber nicht solche, die man unterlassen hat, obwohl man sie hätte auslösen müssen. Besonders bei Betrügereien mit der Reihenfolge von Ereignissen ist es schwierig bis unmöglich kryptografisch nachzuweisen, dass ein Spieler eine Nachricht erhalten hat, bevor er ein Ereignis auslöste. Ganz allgemein kann Kryptografie die Echtheit eines Dokumentes nachweisen, aber sie kann nicht nachweisen, dass es kein solches Dokument gibt bzw. geben kann. Ein Betrüger kann daher leicht behaupten, eine Nachricht nicht oder zu spät erhalten zu haben, denn im Internet gibt es keine verlässlichen Empfangsbestätigungen, die garantieren, dass der Betrüger den Empfang einer Nachricht nicht leugnen kann.

Anhand der Regelverstoß-Problematik kann man schon erkennen, dass selbstorganisierende Systeme nicht für sicherheitskritische Anwendungen ge-

eignet sind. Die Automatismen zur Selbstorganisation und Selbststabilisierung funktionieren zwar sehr gut, aber nur solange niemand mit betrügerischen Absichten eingreift. Ein Betrüger oder Angreifer innerhalb des P2P-Netzes, der sich nicht an der Durchführung des Automatismus beteiligt oder absichtlich neue Fehler erzeugt, kann die erfolgreiche Durchführung des Automatismus verhindern. Ein Server als prüfende, vertrauenswürdige Instanz existiert in reinen P2P-Netzen nicht. Daher müssen in P2P-Netzen zusätzliche Maßnahmen ergriffen werden, um das System gegen Betrugsversuche abzusichern. Allerdings ist dies, wie bereits beschrieben, immer nur in einem begrenzten Maße möglich. Sobald beispielsweise die Anzahl der Betrüger innerhalb des Netzes größer ist als die Anzahl ehrlicher Spieler, kann die Durchsetzung der Regeln nicht mehr garantiert werden. P2P-Netze sind daher nur dann sinnvoll, wenn durch Fehler niemand verletzt wird und auch kein substanzieller finanzieller Schaden entsteht und wenn die Peers eine intrinsische Motivation haben, sich an die Regeln zu halten.

Das System muss daher so entwickelt werden, dass ein Abweichen von den Regeln mehr schadet als nutzt. Dies ist beispielsweise bei der Datei-Tauschbörse BitTorrent der Fall. Das Ziel der Teilnehmer ist das Herunterladen von Dateien. Das Hochladen hingegen nutzt nur den anderen und wird daher gerne gedrosselt. Wenn alle das Hochladen drosseln, wird aber auch für alle das Herunterladen langsamer. Deshalb schicken regelkonforme BitTorrent-Peers nur Daten an solche Peers, die ihrerseits mit annehmbarer Geschwindigkeit Daten hochladen. Drosselt ein Betrüger die Geschwindigkeit beim Hochladen, drosselt er so automatisch auch seine eigene Empfangsgeschwindigkeit, was nicht in seinem Interesse liegt.

Das Paradebeispiel BitTorrent lässt sich aber leider nicht generalisieren. Es funktioniert nur, weil zwei kommunizierende Peers etwas voneinander wollen (Daten) und diese jeweils nur im Austausch hergeben. Bei virtuellen Welten ist das nicht immer der Fall. Der Sieg liegt schließlich nur im Interesse des Siegers, nicht aber im Interesse des Verlierers.

Zusammenfassung und Ausblick

Forschungsergebnisse auf dem Gebiet der Selbst-Organisation zeigen, dass Computer in einem großen verteilten System auf Grund lokaler Entscheidungen eine globale Struktur erzeugen und diese auch erhalten können. Aufbauend auf dieser Struktur können Anwendungen verschiedenster Art realisiert werden. Aus einer rein wissenschaftlichen Perspektive betrachtet, sind Algorithmen zur Selbstorganisation schon allein deshalb interessant, weil sie die Frage beantworten, wo die Grenzen von Selbstorganisation liegen. Die Arbeit an *Peers@Play* hat allerdings auch aufgezeigt, dass solche Systeme deutlich schwieriger zu konstruieren sind als zentralisierte Systeme. Ohne eine zentrale Autorität werden schon grundlegende Probleme wie Zeit, Kausalität und Kon-

sistenz berührt. Zentrale Server lösen auch dort nicht alle Probleme, machen aber vieles einfacher.

Aus technischer Sicht betrachtet, ist dem zentralisierten Ansatz des Cloud-Computing der Vorzug zu geben. Denn alles was ein P2P-System technisch leisten kann, vermag auch eine Serverlösung zu leisten, wenn jemand bereit ist, die Rechnung hierfür zu bezahlen. Umgekehrt gilt das hingegen nicht. Der Vorteil des sich selbstorganisierenden P2P-Systems in der Praxis liegt darin, dass Strukturen zu jeder Zeit automatisch erzeugt werden können, ohne erst einen Spender zu finden oder ein Geschäftsmodell zu entwickeln, um die Kosten für die Server-Infrastruktur zu tragen. Insbesondere für freie Software ist dies ein wichtiger Punkt, denn freie Software nutzt dem Einzelnen wenig, wenn er auf einen Anbieter angewiesen bleibt, der die passende Server-Infrastruktur betreibt.

Literatur

„World of Warcraft Subscriber Base Reaches 12 Million Worldwide“, in: *Blizzard Entertainment*, online unter: http://us.blizzard.com/en-us/company/press/pressreleases.html?id=2847881, zuletzt aufgerufen am 29.09.2013.

Aurenhammer, Franz, „Diagrams – A Survey of a Fundamental Geometric Data Structure“, in: *ACM Computing Surveys* 23, 3 (1991), S. 345-405.

Dijkstra, Edsger W., „Self-Stabilizing Systems in Spite of Distributed Control“, in: *Communications of the ACM* 17, 11 (1974), S. 643-644.

Forschungsprojekt *Peers@Play*, online unter: http://www.peers-at-play.org/, zuletzt aufgerufen am 29.09.2013.

Holzapfel, Sebastian/Schuster, Sebastian/Weis, Torben, „VoroStore – a Secure and Reliable Data Storage for Peer-to-Peer-Based MMVEs“, in: *International Conference on Computer and Information Technology*, 2011, S. 35-40.

Lamport, Leslie, „Time, Clocks, and the Ordering of Events in a Distributed System“, in: *Communications of the ACM* 21, 7 (1978), S. 558-565.

Mattern, Friedemann, „Verteilte Basisalgorithmen“, in: *Informatik-Fachberichte* 226 (1989), S. 28-30.

Mauve, Martin/Vogel, Jürgen/Hilt, Volker/Effelsberg, Wolfgang, „Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications“, in: *IEEE Transactions on Multimedia* 6, 1 (2004), S. 47-57.

Stoica, Ion/Morris, Robert/Liben-Nowell, David/Karger, David R./Kaashoek, M. Frans/Dabek, Frank/Balakrishnan, Hari, „Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications“, in: *IEEE/ACM Transactions on Networking* 11, 1 (2003), S. 17-32.

World of Warcraft, online unter: http://eu.battle.net/wow/, zuletzt aufgerufen am 29.09.2013.