# ZetaGrid

**Dr. Sebastian Wedeniwski, IT Architect**

IBM Deutschland Entwicklung GmbH

Postfach 1380

71003 Böblingen

*Schlüsselworte: Internet Computing, Grid Computing, Riemannsche Hypothese*

## 1 Ungenutzte Ressourcen

### 1.1 Große Organisationen und Firmen

In großen Organisationen oder Firmen wie zum Beispiel der IBM Corporation werden die Prozessorleistungen moderner Büroarbeitsplätze zu höchstens 20% genutzt, trotz aller Klagen, dass der verwendete Rechner für bestimmte Anwendungen zu langsam sei, wie zum Beispiel eine Anwendung zu starten, ein Spielprogramm zu verwenden oder eine Datei zu öffnen. Die meiste Zeit nämlich, beispielsweise bei Meetings, Surfen im Web oder bloßem Tippen von Dokumenten, nutzt der Rechner seine Prozessorleistung kaum, bei Pausen wird ein Teil sogar für nett gestaltete Bildschirmschoner verbraucht. Neben den Büroarbeitsplätzen gibt es selbstverständlich auch noch größere Server-Rechner, die entweder Dienste anbieten oder als Rechenknechte intensiv eingesetzt werden; aber selbst hier fällt auf, dass diese Rechner oft zwar 24 Stunden am Tag laufen, doch kaum während der gesamten Betriebszeit genutzt werden. Ja noch schlimmer, da sie im Gegensatz zu den Büroarbeitsplätzen nicht abgeschaltet werden, sind sie je nach Aufgabenfeld vor allem an Feiertagen und Wochenenden vollkommen ungenutzte Ressourcen.

### 1.2 Netzwerke bilden den mächtigsten Leerlauf-Computer der Welt

Netzwerke bieten heutzutage deutlich mehr Flexibilität und Datendurchsatz als die früheren Terminals, die nur an einen Host-Rechner angeschlossen waren und

deshalb Prozesse nur zentral ablaufen lassen konnten, was diesen zentralen Host fast immer überlastet hat. Mit dem Aufkommen der PCs hat sich dann schnell die Client/Server-Architekur durchgesetzt, um Prozessabläufe dezentralisieren und besser paralellisieren zu können. Früher waren dabei die zentralen Server die eindeutig leistungsstärkeren Rechner, weil jegliche Kommunikation über sie lief und wesentliche Informationen auf ihnen konzentriert wurden; die Clients agierten dabei überwiegend als komfortable Benutzerschnittstellen. Heute werden dagegen aufgrund der neuen Anforderungen, die Ressourcen flexibel einsetzen zu können, oft Querverbindungen zwischen verschiedenen Clients gesetzt, was durch die bessere Ausstattung der Clients ermöglicht wird. So kann mittlerweile jeder handelsübliche PC, der mit einem modernen Betriebssystem läuft, problemlos fast alle Dienste erfüllen, die früher nur ein Server bieten konnte. Jeder PC kann somit sowohl Dienste als Server anbieten als auch gleichzeitig als Client die Dienste anderer Rechner im Netz nutzen, was unter dem Begriff der Peer-to-peer Architektur (P2P) zusammengefasst wird. Dieser Begriff und diese Architektur sind also nicht neu,[1] neu ist lediglich die eingesetzte Masse an Einzelressourcen innerhalb der Architektur. In solch einer mächtigen, aber auch komplexen P2P-Architektur entstehen neben den neuen Möglichkeiten aber auch neue Probleme, wie zum Beispiel, wenn – wie meist der Fall – die beteiligten PCs nur zeitweise zur Verfügung stehen: Sie fallen dementsprechend als (alleinige) Server aus, es sei denn, man nutzt die vorhandene Redundanz an Ressourcen für einen hintereinandergeschalteten Sicherungsmechanismus, der bisher erstaunlich gut funktioniert. Die uns zur Verfügung stehenden Netzwerke vereinigen also die einzelnen PCs zu einem mächtigen System, das von keinem einzelnen Supercomputer übertroffen werden kann – mit dem wesentlichen Unterschied jedoch, dass die Ressourcen eines Supercomputers nahezu ausgelastet sind. Auch wenn sich auf dem Gebiet der besseren Ressourcennutzung in den letzten Jahren viel bewegt hat (siehe zum Beispiel [2] oder [5]), gibt es noch einiges zu tun, was sich besonders daran zeigt, dass es zum Beispiel noch keinen einheitlichen Standard für ein Kommunikationsprotokoll gibt.

## 1.3 Computeranwendungen in Forschung und Wirtschaft

Der Fokus produktorientierer Organisationen wie IBM Global Services ist ein ganz anderer als der forschungsorientierter, wie er an Universitäten üblich ist. Eine Anwendung, die in einer forschungsorientierten Organisation entwickelt wird, setzt in der Regel ihren Schwerpunkt darauf, die besten Algorithmen zu finden und den Code in jedem Detail auszufeilen. Diese Punkte werden selbstverständlich auch innerhalb eines Firmenprojektes mit einem Geschäftskunden berücksichtigt, doch wird der Projektverlauf insgesamt

[1] Diese Systemarchitektur prägte sogar die Anfänge des Internets Ende der 60er Jahre (siehe [5]).

wesentlich mehr von der zur Verfügung stehenden Zeit und den entstehenden Kosten beeinflusst. Denn in der Regel interessiert sich der Kunde mehr dafür, dass die entwickelte Lösung stabil läuft und vor allem bezahlbar ist, als für technische Einzelheiten. Existiert zum Beispiel ein bereits etablierter Algorithmus, der vor mehreren Jahren vielleicht sogar noch in Cobol geschrieben wurde, nun jedoch seine Aufgaben für den Kunden zu langsam erfüllt, so werden meist eher schnellere Rechner eingesetzt beziehungsweise ein Netzwerk von mehreren Rechnern aufgebaut, um den Durchsatz zu erhöhen, als dass dieser Algorithmus im Code verbessert oder sogar komplett neu entwickelt wird. Dieser Ansatz ist in den meisten Fällen – besonders in einer Client/Server-Architektur – korrekt, weil er deutlich weniger Risiken enthält und geringere Kosten beim Kunden verursacht, falls nicht zusätzliches Personal für neue Hardware eingesetzt werden muss. Aber gerade dadurch entsteht ein Netz mit vielen leistungsstarken Ressourcen, die gar nicht mehr vollständig durch ihre eigentliche Aufgabe ausgelastet sind und somit unglaublich hohe Leerlaufzeiten haben.

## 1.4 Die Nutzung freier Ressourcen für ungelöste Probleme

An diesen Punkt habe ich nun angesetzt, um eine flexible und einfache Möglichkeit zu schaffen, die Rechner besser auszunutzen. Es geht dabei nicht darum, den Rechner mit fremden Diensten vollständig zu blockieren, sondern lediglich eine entstehende Leerlaufzeit durch Aufgaben auszutauschen, so dass der Benutzer weiterhin immer über die volle Leistung seines Rechners verfügt. Des Weiteren ist es hierbei wichtig, dass die Aufgabenstellung flexibel und der Verwaltungsaufwand gering ist. Kurz gesagt: Der Ressourcen-Anbieter will keine Kenntnisse über Netzwerke und großflächig verteilte Berechnungen haben müssen, dagegen aber Vorteile für die Bereitstellung seiner Dienste erhalten und die Zugriffsrechte und den Preis mitbestimmen können.

Prinzipiell sind für die Freigabe von eigenen Ressourcen zwei verschiedene Ansätze vorstellbar: Erstens kann der Bildschirmschoner im Betriebssystem durch ein Programm ersetzt werden, das sich für den Benutzer genauso wie ein Bildschirmschoner verhält, dabei aber die zur Verfügung stehenden Ressourcen zur Lösung von Problemen nutzt; denn genau dann, wenn der Bildschirmschoner aktiv ist, wird der Rechner ja gewöhnlich nicht genutzt, falls er nicht schon für andere Rechner Dienste anbietet oder mit einer größeren lokalen Aufgabe beschäftigt ist. Die zweite Möglichkeit besteht darin, im Hintergrund einen permanenten Prozess als Dienst im Betriebssystem oder als Befehl in der Eingabeaufforderung laufen zu lassen, der nur die kontinuierlich vorhandene Leerlaufzeit in Anspruch nimmt und daher ebenso wenig den Benutzer in seinem Ablauf stört. Bei der bisherigen Installation von ZetaGrid im IBM Labor Böblingen hat sich gezeigt, dass sich Manager eher für die erste Lösung und Entwickler eher für die zweite Lösung entschieden haben.

So verlockend sich die theoretischen Vorteile solch eines Netzwerkes auch anhören, in dem eine große Anzahl verschiedener Ressourcen, die sich zudem in unterschiedlichen Organisationen befinden und nur zeitweise im Internet zur Verfügung stehen, um die Lösung einer flexiblen Aufgabe kümmern, ist in der Praxis doch mehr nötig als nur die einzelnen Bausteine zusammenzusetzen. Mein Ansatz ist daher, sich den Aufbau eines solchen „Grid“[2] als eine Vier-Stufen-Pyramide vorzustellen (siehe Abbildung 1).

Die unterste und einfachste Ebene besteht aus den Ressourcen einer Einzelperson, für die lediglich notwendig ist, dass die zu lösende Aufgabe stabil läuft. Sonstige Kriterien sind meist weniger wichtig, weil dieses Projekt nur nebenher laufen soll. Die zweite Ebene umfasst die Ressourcen der gesamten Firma, welche intern angesprochen werden können. Hierfür ist neben der Voraussetzung der ersten Ebene nötig, das Vertrauen der einzelnen Ressourcen-Anbieter in die zu installierende Software und die mit ihr zu verrichtenden Aufgaben zu gewinnen, also im wesentlichen die Furcht vor ungewolltem oder unkontrolliertem Zugriff auf die eigene Ressource zu beseitigen. Eine weitere Absicherung wie zum Beispiel vor Sabotageakten ist in diesem durch die Firmenzugehörigkeit geschützten Rahmen weniger gefordert, weil derartiges in aller Regel zur fristlosen Kündigung führt und damit eine genügend große „Firewall“ darstellt. Die dritte Ebene wird von Kunden der Firma gebildet, die in der Regel leichter erreicht werden können als völlig fremde Ressourcen-Anbieter im Internet, weil die Infrastruktur und die verantwortlichen Vertragspartner bekannt sind. Für den Kunden müssen jedoch wesentliche Vorteile wie zum Beispiel Kostenreduzierungen, Produktionsverbesserungen oder neue Absatzmärkte entstehen, damit er auch Interesse hat, seine Ressourcen einzubinden. Zusätzlich gilt spätestens ab dieser dritten Ebene, dass die Datenumgebung geschützt sein muss. Im Internet Beteiligte sind in einer übergeordneten vierten Ebene angesiedelt, die sich nur schwer einschätzen lässt, weil hier zu viele unterschiedliche Interessen präsent sind. Neben den Voraussetzungen der vorigen drei Ebenen gilt aber hier grundsätzlich, die Privatspähre des einzelnen von Seiten der Software oder Hardware zu sichern, um Hackern keinen Einlass zu bieten.

[2] Es gibt unterschiedlich strenge Definitionen von dem, was heute mit Grid bezeichnet wird; die auch im folgenden verwendete urspüngliche Definition ist in [2] vorgestellt.

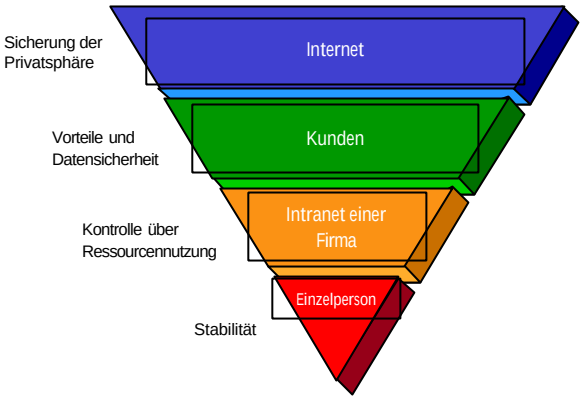


Abbildung 1: Ressourcen-Pyramide

Seit mehreren Jahren existieren Clients im Netz, die sich auf eine bestimmte Aufgabe spezialisiert haben. Das bekannteste öffentlich erreichbare Projekt ist SETI@home (www.setiathome.ssl.berkeley.edu), das Funksignale aus dem Äther filtert, um außerirdisches Leben ausfindig zu machen; eine Liste weiterer Projekte findet sich unter www.rechenkraft.de. Diese Clients schützen aber höchstens die beim Server abgelieferten Ergebnisse und auch das nur sehr simpel; die Privatsphäre des Ressourcen-Anbieters auf der Client-Seite bleibt ungeschützt. So öffnet der Client für die Kommunikation zwar eine Socket-Verbindung zum Server, überprüft aber nicht, ob die hereinkommenden Daten authentisch und gegen in der Kryptoanalyse bekannte Attacken resistent sind. Dieser Punkt ist für größere Organisationen wie IBM jedoch entscheidend, wenn Aufgaben nicht mehr lokal von einem Anbieter gelöst werden, sondern auf sehr viele verteilt werden sollen.

# 2 ZetaGrid als ein IBM-Beispiel für die Nutzung freier Ressourcen

Grid-Computing ist auch bei IBM offiziell zu einem ihrer strategischen Ziele ernannt worden. Dementsprechend werden Privatinitiativen von Mitarbeitern auf diesem Gebiet nicht nur gern gesehen, sondern auch gefördert, was auch ZetaGrid den entscheidenden Auftrieb gegeben hat. Die Anwendung beschäftigt sich zur Zeit damit, grundlegende Erkenntnisse über die Riemannsche Hypothese zu sammeln,[3] was ursprünglich noch auf meine Promotionsarbeit [7] zurückgeht.

[3] Zur praktischen Bedeutung der Hypothese siehe das Kapitel „Nutzen der Nullstellenberechnung und der Riemannschen Hypothese“.

Im folgenden werden deshalb zunächst in Grundzügen die Geschichte und die mathematischen Grundlagen der Riemannschen Hypothese dargelegt und gezeigt, warum sie sich als Beispiel für Grid Computing geradezu anbietet, bevor im nächsten Abschnitt der Fokus auf der technischen Realisierung von ZetaGrid liegt.

## 2.1 Die Riemannsche Hypothese

Die Riemannsche Hypothese besagt, dass alle nichttrivialen Nullstellen der analytischen Fortsetzung der Riemannschen Zeta-Funktion

$$z(s) = \sum_{k=1}^{\infty} \frac{1}{k^s} = \prod_{p \in P}^{\infty} \frac{1}{1 - p^{-s}}$$

auf der kritischen Geraden ½+*it* liegen, wobei *t* eine reelle Zahl, *s* eine komplexe Zahl mit einem Realteil größer als 0 und *P* die Menge alle Primzahlen ist. Diese Hypothese schreibt durch die folgenden Zeilen von B. Riemann [6] seit 1859 mathematische Geschichte: „...es ist sehr wahrscheinlich, dass alle Wurzeln reell sind. Hiervon wäre allerdings ein strenger Beweis zu wünschen; ich habe indess die Aufsuchung desselben nach einigen flüchtigen vergeblichen Versuchen vorläufig bei Seite gelassen, da er für den nächsten Zweck meiner Untersuchung entbehrlich schien."

Seit nun über 140 Jahren ist es noch niemandem gelungen, diese grundlegende Vermutung zu beweisen oder zu widerlegen. Bisher wurde erstens gezeigt, dass unendlich viele nichttriviale Nullstellen der Zeta-Funktion auf der kritischen Geraden liegen, und zweitens, dass mindestens 40% aller nichttrivialen Nullstellen auf dieser Geraden liegen. Mittlerweile gehen viele Mathematiker davon aus, dass die Vermutung korrekt ist.

Da es eines der wichtigsten Probleme der modernen Mathematik ist, hat im Jahr 2000 das Clay Mathematics Institut einen Preis von $1.000.000 für den Beweis der Riemannschen Hypothese ausgesetzt (http://ww.claymath.org/prizeproblems/riemann.htm).

## 2.2 Empirische Grundlagenforschung

Der einfachste Weg, diese Vermutung zu widerlegen, besteht darin, eine nichttriviale Nullstelle zu finden, die nicht auf der kritischen Geraden liegt, womit sie also einen Realteil ungleich ½ hat. Dieser Weg, nämlich Gegenbeispiele für eine Vermutung zu finden, ist sehr verbreitet nach dem Motto „Ein Beispiel ist kein Beweis, aber ein Gegenbeispiel ist ein Gegenbeweis", weil es oft einfacher ist, eine falsche mathematische Aussage anhand eines Gegenbeispiels zu

widerlegen, als eine richtige zu beweisen.[4] Gerne setzt man dafür Computer ein, bei denen man sich darauf verlassen kann, dass sie auch bei langwierigen und stupiden Rechnungen nicht nachlässig werden.

Für die Riemannsche Zeta-Funktion sind beispielsweise als die ersten vier nichttrivialen Nullstellen berechnet worden

$$r_1 \approx \tfrac{1}{2} + 14{,}135i$$
$$r_2 \approx \tfrac{1}{2} + 21{,}022i$$
$$r_3 \approx \tfrac{1}{2} + 30{,}011i$$
$$r_4 \approx \tfrac{1}{2} + 32{,}935i.$$

Wie die folgende Tabelle zeigt, ist das Interesse an dieser Nullstellenberechnung im letzten Jahrhundert kontinuierlich gestiegen. Dabei gibt $n$ die Anzahl der ermittelten Nullstellen an, die sich im positiven Imaginärteil befinden und fortlaufend hintereinander liegen.

| Jahr | Autor | $n$ |
|---|---|---|
| 1903 | J. P. Gram | 15 |
| 1914 | R. J. Backlund | 79 |
| 1925 | J. I. Hutchinson | 138 |
| 1935 | E. C. Titchmarsh | 1.041 |
| 1953 | A. M. Turing | 1.104 |
| 1955 | D. H. Lehmer | 10.000 |
| 1956 | D. H. Lehmer | 25.000 |
| 1958 | N. A. Meller | 35.337 |
| 1966 | R. S. Lehman | 250.000 |
| 1968 | J. B. Rosser, J. M. Yohe, L. Schoenfeld | 3.500.000 |
| 1977 | R. P. Brent | 40.000.000 |
| 1979 | R. P. Brent | 81.000.001 |
| 1982 | R. P. Brent, J. van de Lune, H. J. J. te Riele, D. T. Winter | 200.000.001 |
| 1983 | J. van de Lune, H. J. J. te Riele | 300.000.001 |
| 1986 | J. van de Lune, H. J. J. te Riele, D. T. Winter | 1.500.000.001 |
| 2001 | S. Wedeniwski | 10.118.665.300 |

Da weitere Details und mathematische Grundlagen zu diesem Gebiet den vorliegenden Beitrag sprengen würden, verweise ich an dieser Stelle Interessenten

[4] Daneben hilft die Suche nach einem Gegenbeispiel manchmal dabei, eine Idee für einen Beweis zu finden.

auf die Web-Seite [8] des ZetaGrid-Projektes, auf der auch weiterführende Literatur zu finden ist.

## 2.3 Nutzen der Nullstellenberechnung und der Riemannschen Hypothese

Eine Frage, die sich bei diesem Thema sofort stellt, ist: Warum ist die Riemannsche Hypothese überhaupt so wichtig und welcher Nutzen entsteht durch die Ermittlung vieler Nullstellen?

Die letzte Frage ist schnell beantwortet: Je mehr Nullstellen berechnet werden, desto größere Bereiche kann man eingrenzen, innerhalb derer die Hypothese sicher gilt. Entweder man weitet den Umfang des gültigen Bereichs also irgendwann soweit aus, dass die Korrektheit der Riemannschen Hypothese insgesamt für einige Theoreme nicht mehr so entscheidend ist, oder es wird ein Gegenbeispiel gefunden, wodurch sich das Problem (zumindest vordergründig) erledigt hat.[5] Der praktische Nutzen der Gültigkeit der Riemannschen Hypothese liegt, grob gesagt, auf dem wichtigen und immer größer werdenden Gebiet der Datensicherung. Für mathematisch Versierte kann hinzugefügt werden, dass die Riemannsche Zeta-Funktion im Bereich der Zahlentheorie grundlegend ist, und zwar für Primzahlen, ohne die zum Beispiel in der Kryptographie elliptische Kurven nicht korrekt verwendet werden könnten oder der RSA-Algorithmus zu leicht zu entschlüsseln wäre.

Da es schwierig und zeitintensiv ist, beliebig viele Primzahlen zu erzeugen und zu testen, werden stattdessen bei der praktischen Umsetzung eines kryptographischen Systems „Pseudoprimzahlen“ verwendet, von denen bekannt ist, dass sie mit sehr hoher Wahrscheinlichkeit Primzahlen sind. Die dadurch entstehenden Unsicherheitsfaktoren und Probleme gäbe es nicht, wenn die Erweiterte Riemannsche Hypothese[6] korrekt wäre, weil dann der schnellste zur Zeit bekannte Primzahltest [4] verwendet werden könnte.[7] Doch bisher geht man dieses Risiko ein, da erstens die Wahrscheinlichkeit gering genug ist, dass eine solche Pseudoprimzahl in der Praxis einen schweren Fehler bewirkt, und zweitens eine absolute Sicherheit sowieso nie zugesagt werden kann. Wenn man also die Rechenkraft vieler Computer dazu einsetzt, durch Berechnung der Nullstellen der Riemannschen Zeta-Funktion die Gültigkeit der Riemannschen Hypothese zu

---

[5] Natürlich entstehen dadurch wieder andere Schwierigkeiten, die aber hier nicht das Thema sind.

[6] Die Erweiterte Riemannsche Hypothese ist eine Verallgemeinerung der Riemannschen Hypothese, sie macht also die gleiche Aussage wie für die Zeta-Funktion, nur dass die verwendete Funktion allgemeiner gefasst ist.

[7] Eine genauere Auseinandersetzung mit Primzahlen, Primzahltests und der Notwendigkeit der Riemannschen Hypothese findet in [7] statt.

erweitern, leistet man einen aktiven Beitrag zur allseits geforderten Datensicherheit.[8]

## 2.4 Wie die Nullstellen ermittelt werden

In der Numerischen Mathematik und speziell bei der Kurvendiskussion in der Analysis gibt es verschiedene Verfahren, die Nullstellen einer Kurve zu ermitteln. Die einfachste Regel, die wir für reelle und stetige Funktionen kennen, ist, dass mindestens eine Nullstelle zwischen einem positiven und negativen Funktionswert liegen muss. Diese Regel ist auch für unsere Anwendung grundlegend, weil wir nicht eine genaue Nullstelle ermitteln wollen, sondern nur ermitteln wollen, dass in bestimmten Intervallen auch eine bestimmte Anzahl von Nullstellen liegen. In einem Theorem hat A. M. Turing[9] gezeigt, wie viele Nullstellen in einem bestimmten Intervall liegen müssen, damit die Riemannsche Hypothese für dieses Intervall bewiesen ist.[10]

Außerdem hat bereits 1903 J. Gram festgestellt, dass der Vorzeichenwechsel der Kurve der Riemannschen Zeta-Funktion (siehe Abbildung 2), der, wie oben erwähnt, mindestens eine Nullstelle verursacht, fast immer an ganz bestimmten Punkten festgestellt werden kann. Diese Gram-Punkte beginnen mit der Folge 9,67, 17,85, 23,17 usw. und werden durch eine Funktion beschrieben. Diese Feststellung ist in insgesamt 68% der Fälle richtig, wobei sie zu Beginn stets stimmt und erst ab dem 126. Punkt Ausnahmen aufweist. Diese Regel wurde 1969 von J. B. Rosser derart erweitert, dass nun mehrere Gram-Punkte gleichzeitig in einem Block betrachtet werden können, wodurch in 99,9991% aller auftretenden Fälle die fehlenden Nullstellen schnell gefunden werden können, wenn die Regel von Gram nicht zutrifft.[11] Ich habe seine Regel noch dahingehend erweitert, dass bei der jetzigen Aufgabe von ZetaGrid mehrere dieser Blöcke gleichzeitig betrachtet werden, weil selbst diese wenigen Ausnahmen bei einer Berechnung von mehreren Milliarden Nullstellen sehr störend sind. Selbst nach 25 Milliarden Nullstellen gab es noch keine Ausnahme zu dieser Erweiterung.

Somit liegt der größte Aufwand der momentanen Aufgabe von ZetaGrid darin, den Funktionswert der Zeta-Funktion an den verschiedenen Gram-Punkten bzw. den wenigen dazwischen liegenden Punkten zu berechnen. Empirisch hat sich

---

[8] Andere grundlegende Theoreme der Mathematik und Physik, die nur dann korrekt sind, wenn die Riemannsche Hypothese korrekt ist, finden sich zum Beispiel in [1].

[9] Informatikern bekannt sein dürfte die Turingmaschine, die 1936 vom britischen Mathematiker A. M. Turing (1912-1954) als universelles Automatenmodell vorgeschlagen wurde.

[10] Falls also weniger Nullstellen in einem Intervall empirisch gefunden werden, wäre ein Gegenbeispiel zur Riemannschen Hypothese gefunden.

[11] Die erste Ausnahme der Regel von Rosser tritt beim 13999525. Punkt auf.

dabei gezeigt, dass nach durchschnittlich 1,22 Funktionsberechnungen eine Nullstelle gefunden wird.

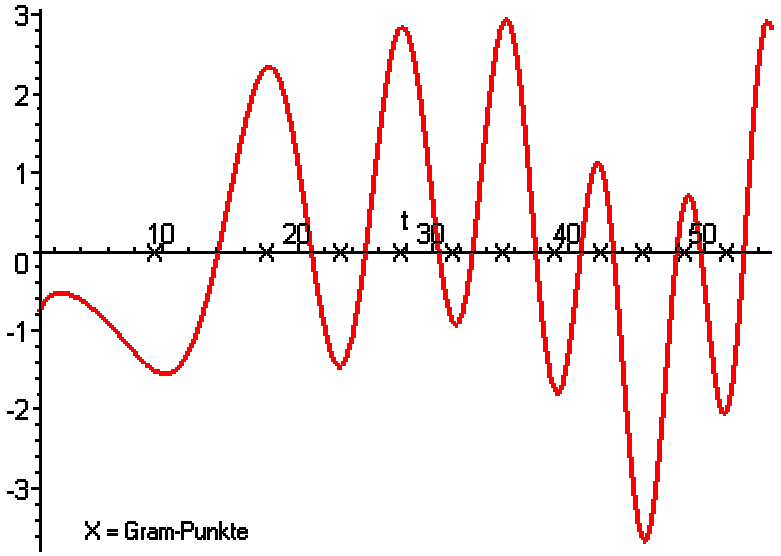


Abbildung 2 : Graph der Funktion $t \mapsto z\ (1/2 + it)e^{i(\mathrm{Im}(\ln(\Gamma(1/4+1/2\,i)))-1/2\cdot\ln(p))}$

# 3 Technische Aspekte von ZetaGrid

Die ZetaGrid Web-Seiten [8] spielen eine entscheidende Rolle in diesem Projekt. Auf diesen Seiten können Nutzer den Kernel herunterladen, genauere Informationen über ZetaGrid bekommen und aktuelle Statistiken wie den Stand der Berechnung überwachen. Die Seiten werden durch Servlets und andere zusätzliche Programme erzeugt, welche die aktuellen Informationen aus der zentralen Datenbank herausholen.

Der Client von ZetaGrid besteht aus einem Kernel in Java, einer Bibliothek, die sich um die Kryptographie kümmert und den authentifizierten öffentlichen Schlüssel enthält,[12] und einem austauschbaren Paket, das sich um die Lösung der verteilten Aufgabe kümmert. Diese Schichten sind folgendermaßen aufgebaut:

[12] Da die Bibliothek eine Public-Key-Infrastruktur bildet, ist nur der öffentliche Schlüssel freigegeben; der geheime Schlüssel verbleibt beim Autor der Bibliothek.

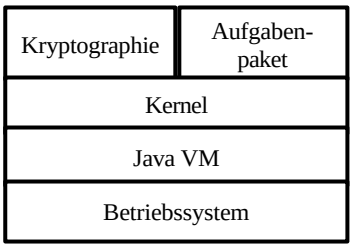


Abbildung 3 : Schichtenmodell des Clients von ZetaGrid

## 3.1 Kernkomponente in Java

Große oder komplexe Netzwerke, welche viele Clients involvieren, enthalten automatisch verschiedene Betriebssysteme und Prozessorarchitekturen. Ich habe mich deshalb bei ZetaGrid für Java als Programmiersprache entschieden, weil sie den entscheidenden Vorteil der Plattformunabhängigkeit hat. Dabei ist die immer als Nachteil diskutierte Performance von Java, die vorgibt, dass der durch den Java-Compiler erzeugte Bytecode noch während der Ausführung des Programms in den Maschinencode interpretiert oder compiliert werden muss, kein Problem, denn in der Praxis gilt die Faustregel: Die Performance von Java ist schnell für komplexe Systeme, bei denen mindestens 80% der Gesamtlaufzeit in mindestens 80% des Codes stattfindet. Demnach liefert Java also eine gute Lösung für „große dünne Probleme“, aber weniger für „kleine dicke Probleme“, bei denen nur wenige Zeilen Code (oft sogar nur eine einzige Schleife) den Großteil der Programmlaufzeit in Anspruch nehmen. Für diese Art von Problemen liefern C/C++ oder Fortran meist die schnelleren Ergebnisse, wobei dann aber sichergestellt sein muss, dass der Code für alle verwendeten Prozessoren und Betriebssysteme stets auf den neuesten Stand gebracht wird.

Der Kernel von ZetaGrid kümmert sich um die folgenden beiden Aufgaben:

1. Der erste Teil bereitet alles für die zu erledigenden Arbeitseinheiten vor. Hierbei wird eine Anforderung in HTTP an ein Servlet des Aufgabenservers geschickt und auf eine unverschlüsselte Zip-Datei als Antwort gewartet. Diese Zip-Datei enthält als ersten Eintrag eine Datei mit den digitalen Signaturen aller weiteren Dateien der übertragenen Zip-Datei. Anhand des lokalen öffentlichen Schlüssels und der übertragenen digitalen Signaturen werden dann diese Dateien verifiziert, bevor sie zur Vorbereitung der Arbeitseinheiten auf die lokale Platte geschrieben werden. Die hierfür verwendete Schlüssellänge und das Verfahren wird weiter unten im Unterkapitel „Die Sicherheitsaspekte“ besprochen.
2. Der zweite Teil kümmert sich um die gesamte Verwaltung der vorgesehenen Arbeitseinheiten. Dazu gehört, die Bereiche, die berechnet werden sollen, über

Anweisungen in HTTP an bestimmte Servlets vom Aufgabenserver zu holen und die abgeschlossenen Arbeitseinheiten an den Ergebnisserver zu liefern. In diesem Teil des Kernels kann der Ressourcen-Anbieter auch einstellen, wie viele Arbeitseinheiten er reservieren will und ob er sie nacheinander oder (bei Multiprozessoren) gleichzeitig verarbeiten will, bevor sich der Kernel wieder mit dem Server verbindet, um neue Arbeitseinheiten anzufordern; denn während der gesamten Berechnung, die vom Kernel über eine Schnittstelle aufgerufen wird, besteht keine Verbindung zum Server. Zusätzlich kann auch die Größe der zu berechnenden Bereiche definiert werden, die, in fünf Stufen gegliedert, zwischen einer und fünf Stunden für einen Intel Pentium III mit 800 MHz pro Arbeitseinheit liegen kann, wodurch die unterschiedlichen Computer flexibel eingebunden werden. Die abgeschlossenen Arbeitseinheiten werden vor der Übertragung komprimiert und mit einem öffentlichen Schlüssel kodiert, der sich nach jeder Transaktion automatisch ändert. Für die Komprimierung wird der Algorithmus des Tools bzip2[13] verwendet. Diese Komprimierung reduziert die Ergebnisdatei um einen Faktor vier, wodurch die Netzbelastung besonders bei Modems deutlich verringert wird; so ist zum Beispiel die Ergebnisdatei einer mittelgroßen Arbeitseinheit etwa 7,2 MB groß, nach der Komprimierung demnach nur noch knapp 1,8 MB. Auch die hierfür verwendete Schlüssellänge und das Verfahren selbst werden im Unterkapitel „Die Sicherheitsaspekte“ weiter unten besprochen.

## 3.2 Die zentrale Datenbank und die eingeschränkten Zugriffe

Die relationale Datenbank ist der zentrale Punkt dieses Projektes, da sie die Daten enthält, die für dieses Projekt relevant sind. Der Grund für diese Organisation der Daten in einer Datenbank ist, dass dadurch die Transaktionen stabil erfolgen und die Skalierbarkeit der persistenten Daten sowie die Performance des Systems ausbaufähig ist. Die Datenbank verwaltet im wesentlichen sechs voneinander getrennte Tabellen, die für verschiedene Serveraufgaben zuständig sind und auch nur bestimmte Zugriffsrechte besitzen. Diese Tabellen umfassen die Verwaltung der Arbeitseinheiten, der Ergebnisse, der Ressourcen-Anbieter, der involvierten Ressourcen, des vom Betriebssystem abhängigen Aufgabenpools[14] und der Systemparameter.

Kein Client hat direkten Zugriff auf den Datenbankserver; der Zugriff auf ihn ist nur über den Web-Server möglich. Kein Server oder andere Beteiligte im Netzwerk haben Löschrechte auf irgendeine Tabelle des Systems. So kann der Aufgabenserver beispielsweise nur Arbeitseinheiten erzeugen, nicht aber löschen

[13] Das Tool bzip2 ist unter der Adresse http://sourceware.cygnus.com/bzip2/ erhältlich und mittlerweile Bestandteil jeder neueren Linux-Installation.

[14] Der Aufgabenpool enthält die nach Plattformen aufgeteilten Bibliotheken, die als Umgebung zur Lösung der ausgewählten Aufgabe benötigt werden.

oder ändern. Änderungsrechte bestehen für einen Kunden lediglich im Aufgabenpool und der Verwaltung der beteiligten Computer. Falls trotz aller Vorsichtsmaßnahmen ein Hacker dennoch diese Sicherheitsvorkehrungen durchbrechen sollte und Änderungen an den beiden Tabellen vornehmen könnte, hätte er trotzdem keinerlei Einfluss auf die beteiligten Clients, die ja nur den authentifizierten Kernel verwenden (siehe das Unterkapitel „Die Sicherheitsaspekte“).

Die Hauptaufgabe der Datenbank ist die Verwaltung der Ergebnisse, welche eine Datenmenge von mehreren Gigabytes am Tag darstellen, die deshalb auch nur komprimiert im System vorkommt. Für diese Verwaltung und Überprüfung der Ergebnisse existiert im System ein separater Server (Ergebnisprüfer); er liest in definierten Zeitintervallen (zur Zeit alle 30 Minuten) alle neu eingetroffenen Ergebnisse aus der Datenbank und entfernt sie aus der Datenbank, wenn sie einige Prüfungen durchlaufen haben und auf anderen Medien im Archiv gesichert wurden.

Bildlich lässt sich der beschriebene Ablauf mit der folgenden Skizze darstellen:

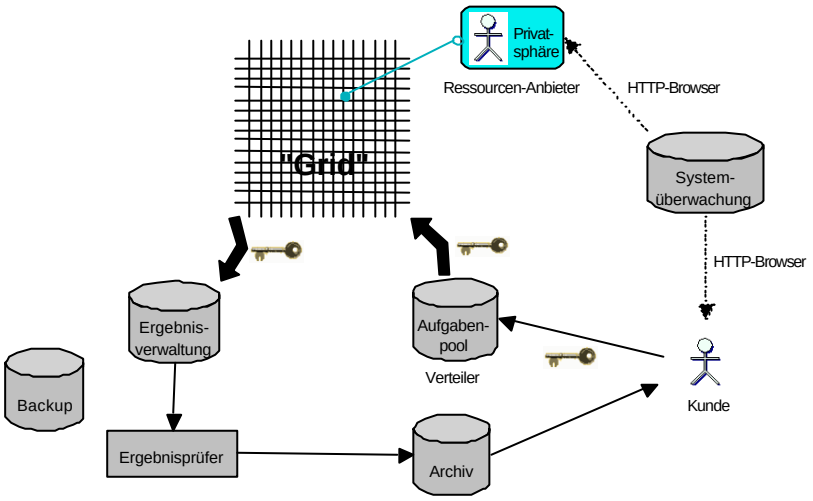


Abbildung 4: Die Architektur von ZetaGrid

## 3.3 Berechnung der Nullstellen der Riemannschen Zeta-Funktion in C++ oder Java?

Die erste Aufgabe von ZetaGrid wurde in C++ geschrieben, da ich im Jahr 2000 noch davon überzeugt war, dass diese Sprache zwar einen höheren Verwaltungsaufwand für mehrere Plattformen mit sich bringt, aber die beste Performance erreicht; der Source-Code besteht nämlich aus etwa 3000 Zeilen

Code, wobei 99,3% der Laufzeit in einer Schleife von 20 Zeilen Code stattfinden, womit diese Aufgabe ein Paradebeispiel für ein „kleines dickes Problem“ ist (siehe oben). Auch wenn ich von der Leistung der verwendeten C++-Compilern sehr überzeugt bin, wurde ich beim Performancevergleich von Java und C++ auf verschiedenen Plattformen überrascht: So benötigt das Programm in Java[15] zum Beispiel unter Windows NT nur 64% der Laufzeit des Programms, dessen Code in C++ geschrieben und mit Microsoft Visual C++ 6.0 erzeugt wurde; fast dasselbe Ergebnis lieferte der durch Intel C++ 5.0.1 erzeugte Code. Noch deutlicher war der Unterschied mit Java unter AIX (PowerPC) im Vergleich mit dem C++-Code, der durch GNU C++ 2.9 erzeugt wurde. Hier benötigte nämlich das Programm mit Java nur noch 31% der Laufzeit des Programms, dessen Code in C++ geschrieben wurde. Der größte mir bekannte Unterschied ist auf den Mainframe-Rechnern S/390 unter Linux aufgetreten; hier benötigte das Java-Programm nur 21% der Laufzeit des C++-Programms, das mit GNU-C++ 2.95.2 erzeugt wurde. Als Ausnahme habe ich unter Linux (x86) mit dem Compiler GNU C++ 2.95.3 ein Beispiel gefunden, bei dem die C++-Version nur 91% der Laufzeit der Java-Version benötigt, weswegen ich speziell für diesen Prozessor den durch die verschiedenen Compiler erzeugten Maschinen-Code untersucht habe. Unter Verwendung dieser Ergebnisse gibt es nun für die 20 Zeilen, die fast die gesamte Laufzeit des Programms beanspruchen, eine etwas aufwändigere Assembler-Version,[16] die speziell für den Pentium-Prozessor optimiert wurde und nur noch 72% der Laufzeit der Java-Version benötigt.

Als Konsequenz dieser leider recht späten Erkenntnisse wird es im ZetaGrid-Projekt bald nur noch eine reine Java-Version geben.

## 3.4 Die Qual der Wahl: Bildschirmschoner oder permanenter Prozess?

Eine der wichtigsten und kniffligsten Eigenschaften, die eine Aufgabe in einem verteilten System besitzen muss, in dem die einzelnen Ressourcen sich beliebig ein- und auskoppeln können, ist, dass sie zu jeder Zeit sofort beenden werden kann und deshalb in der Lage ist, intern eigenständig Check-Points zu verwalten, damit die Berechnung zu einem späteren Zeitpunkt an derselben Stelle fortgeführt werden kann.
Bei ZetaGrid kann der Ressourcen-Anbieter wählen, ob er die Aufgabe in Form eines Bildschirmschoners oder eines permanenten Prozesses laufen lassen möchte.

---

[15] Verwendet wurde die IBM Java VM 1.3 auf den Plattformen AIX, Linux und Windows: Classic VM (build 1.3.0, J2RE 1.3.0 IBM build cx130-20010626 (JIT enabled: jitc)).

[16] Als Vorlage habe ich den vom Intel-Compiler erzeugten Assembler-Code übernommen, weil er bis auf einige externe Funktionsaufrufe für viele Fließkommaoperationen sehr gute Ergebnisse liefert.

Die Option des Bildschirmschoners wurde durch zwei Voraussetzungen ermöglicht: Erstens wurde diese Lösung als dem Kernel vorgeschalteter Prozess realisiert, der sich nur um die vom Betriebssystem abhängigen Eigenschaften des Bildschirmschoners[17] kümmert und den Kernel als separaten Prozess startet; dadurch wird der Gesamtaufwand für beide Optionen gering gehalten. Zweitens wurde die Aufgabe so formuliert, dass sie tatsächlich jederzeit sofort beendet werden kann und selbst intern Check-Points setzt, um zu einem späteren Zeitpunkt die Aufgabe an derselben Stelle fortführen zu können. Dieser Mechanismus von Check-Points ist plattformunabhängig und funktioniert auch, wenn die Maschine abstürzt oder der Prozess durch das System abgeschossen wird.

Die beiden Vorteile dieser Option liegen darin, dass der Berechnungsprozess mit normaler Priorität im System laufen kann und es bisher nur hier möglich ist, dass der Ressourcen-Anbieter nebenher über den aktuellen Stand seiner Berechnung informiert wird.

Die Lösung als permanenter Prozess kann nur mit geringerer Priorität laufen, weil die Ressource ja gleichzeitig auch für andere Zwecke verwendet werden soll. Hierbei hat sich gezeigt, dass zusätzliche Pausen im Prozess eingebaut werden sollten, um den Prozessor nicht zu 100% zu belasten, auch wenn die Aufgabe mit niedriger Priorität erledigt wird. Denn gerade bei Notebooks würde dann die Platte oder der Lüfter die ganze Zeit über laufen, was manchmal lästige Nebengeräusche verursacht. Wir haben festgestellt, dass sich bei einer Belastung von 40-60% des Prozessors dieser unerwünschte Nebeneffekt vermeiden lässt.

## 3.5 Die Sicherheitsaspekte

Das Ziel eines geschützten Systems sollte sein, soviel Vertrauen und Sicherheit bei einem Ressourcen-Anbieter zu gewinnen, dass er einer Beteiligung bei ZetaGrid genauso traut, wie er seiner Steckdose vertraut, dass sie immer genau 220V Strom liefert und auch nur soviel Strom abgerechnet wird, wie er tatsächlich verbraucht. Sicherheit meint dabei heutzutage nicht, möglichst geschickt Bits zu vertauschen und ggf. den Algorithmus dafür geheim zu halten, sondern man versteht darunter vielmehr ein Verfahren, das durch einen mathematisch fundierten Beweis unter vorgegebenen Annahmen in seiner Sicherheit bewiesen ist. Zusätzlich wird dabei das enorme Potential der Öffentlichkeit ausgenutzt, um etwaige Schwachstellen zu finden. Insgesamt lautet die Devise also nicht wie früher, die Codierung als ganzes geheim zu halten, sondern sie im Gegenteil gerade öffentlich zu machen und dadurch zu sichern, dass viele Köpfe überprüft haben, dass das mathematische System dicht hält. Zugegeben, dieser Ansatz

[17] Hierzu gehört zum Beispiel auch die optionale Anmeldemaske unter Windows 98, die von Windows NT/2000 selber verwaltet wird.

verlangt eine gehörige Portion Vertrauen in die Mathematik als Hilfsmittel – aber genau dafür ist sie unter anderem ja konstruiert wo rden.

Das Anliegen eines Ressourcen-Anbieters liegt darin, die richtigen Daten von der richtigen Quelle auf gesichertem Wege zu erhalten und seine vollbrachte Leistung honoriert zu bekommen. Hierbei reicht es also nicht, nur den Datenkanal zwischen der Quelle und der Senke zu verschlüsseln, weil es einem Hacker vielleicht gelingen könnte, direkt auf den Server zuzugreifen, sondern es muss auch sichergestellt werden, dass nur die gewünschten Daten und auch nur von dem vorgesehenen Absender beim Empfänger ankommen. Dieses Problem wird innerhalb von ZetaGrid durch digitale Signaturen gelöst, was bedeutet, dass jedes Aufgabenprogramm vor der Übertragung vom Aufgabensteller anhand seines privaten Schlüssels signiert wurde, den kein Server oder anderer Client im Netzwerk kennt. Der Client akzeptiert nur Programme, welche die Signatur des autorisierten Aufgabenstellers besitzen und zum lokalen öffentlichen Schlüssel passen. Dabei wird das ElGamal-Signaturverfahren mit einer Schüssellänge von 1024 Bit verwendet, welches mathematisch stabil ist und sich als sehr sicher etabliert hat.[18]

Demjenigen, der die Aufgabe stellt, ist es dagegen wichtig, dass die Ergebnisse vom richtigen Client erzeugt und korrekt übertragen werden. Dies zu kontrollieren, gibt es leider zur Zeit nur eingeschränkte Möglichkeiten, weil man postuliert, dass das Netz nur gering belastet wird und außer einem PC beim Ressourcen-Anbieter keine zusätzliche Hardware notwendig sein soll. Hierfür ist das ElGamal-Key-Agreement Protokoll geeignet, weil der Client so mit demselben lokalen öffentlichen Schlüssel ständig andere Verschlüsselungen erzeugen kann und nur der Server in der Lage ist, die Entschlüsselung durchzuführen, ohne dass der Client den Schlüssel übertragen muss. Auch dieses Protokoll verwendet eine Schlüssellänge von 1024 Bit.[19] Eigentlich wäre es gut, wenn alle temporären Daten durch dieses Verfahren geschützt würden, bevor sie auf die lokalen Platte geschrieben werden. Doch hätte diese Sicherheit auch Nachteile, falls eine Arbeitseinheit einer Aufgabe vor einem Abschluss noch einmal komplett durchlaufen werden muss, wie es zum Beispiel bei den ermittelten Nullstellen geschieht, um die Datenmenge auf Redundanz zu überprüfen. Denn dazu müsste der Client die Daten auch entschlüsseln können, was aber in dieser Lösung nicht vorgesehen ist. Indem ich auf diesen Zusatz verzichtet habe, ist die zu sendende Datenmenge neben der Komprimierung auf 25% der Gesamtmenge (siehe oben das Unterkapitel „Kernkomponente in Java“) nochmals um mindestens 22% reduziert.

---

[18] Details zu diesem Signaturverfahren können beispielsweise in [3] nachgeschlagen werden.

[19] Eine genauere Beschreibung findet man in [3].

# 4 ZetaGrid in der Praxis

In die Praxis umgesetzt wurde die Idee von ZetaGrid erstmals im Februar 2001 auf meinem eigenen Rechner; ab August 2001 lief es dann mit Genehmigung von IBM erstmals auf 10 verschiedenen Computern einer Abteilung, womit die Anwendung auf der ersten Ebene der Ressourcen-Pyramide (siehe Abbildung 1) hinreichend getestet und im kleinen Rahmen der Übergang zur zweiten Ebene vorbereitet werden konnte. Allein über Mundpropaganda und die ZetaGrid-Seite im Intranet weitete sich dann der Benutzerkreis bis Anfang Januar 2002 auf 46 Kollegen mit insgesamt 225 Computern aus.

Aufgrund dieser Resonanz und da das System ohne größere Probleme reibungslos lief, wurde seit Mitte Januar unter anderem durch Vorträge die Anwendung im gesamten IBM Labor Böblingen verbreitet, so dass bis Ende Januar für die Berechnung von mehr als 26 Milliarden Nullstellen der Riemannschen Zeta-Funktion etwa 53.000 Arbeitseinheiten auf 413 Computern unter 4 Plattformen und 186 Ressourcen-Anbietern für ZetaGrid verteilt worden sind, was mehr als 13 CPU-Jahre eines Intel P4 Prozessors mit 1.600 MHz entspricht.

Damit durchläuft ZetaGrid nun laborintern eine größere Testphase, bevor es auf die gesamte IBM Deutschland ausgedehnt wird. Daneben wird bereits analysiert, wie sich der Kern von ZetaGrid für andere Aufgaben bei Kunden einsetzen lässt, so dass auch die letzten beiden Ebenen greifbare Ziele darstellen. Interessenten können sich bis zur Realisierung dieser auf der Web-Seite [8] über den jeweils aktuellen Entwicklungsstand informieren.

# 5 Literaturverzeichnis


[1] H. M. Edwards, *Riemann's Zeta Function*, 1974.

[2] I. Foster, C. Kesselman, *The Grid – Blueprint for a New Computing Infrastructure*, 1999.

[3] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of Applied Cryptography*, 1997.

[4] G. L. Miller, *Riemann's Hypothesis and Tests for Primality*, Journal of Computer and System Sciences 13 (1976), 300-317.

[5] A. Oram (Hrsg.), *Peer-to-peer: Harnessing the Power of Disruptive Technologies*, 2001

[6] B. Riemann, *Ueber die Anzahl der Primzahlen unter einer gegebenen Grösse*, Monatsberichte der Berliner Akademie, November 1859.

[7] S. Wedeniwski, *Primality Tests on Commutator Curves*, Dissertation Tübingen, 2001.

[8] S. Wedeniwski, *ZetaGrid – Verification of the Riemann Hypothesis*, zu finden unter http://www.hipilib.de/zeta/index.html.