Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Lehrstuhl für Rechnernetze und Telematik

SS 2009

Seminararbeit

Network Coding in P2P live streaming

Niklas Goby

24. Juli 2009

Betreut durch Prof. Dr. Schindelhauer

Abstract

Viele Forscher haben in den letzten Jahren immer mehr Gefallen daran gefunden, Network Coding mit bereits bestehenden Technologien zu verbinden um damit Verbesserungen zu erziehlen. So auch MẆang und BL̇i die in ihren Arbeiten „R^2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming“ und „Lava: A Reality Check of Network Coding in Peer-to-Peer Live Streaming“ den Einsatz von Network Coding in P2P Live Streaming testen. In beiden Arbeiten werden die verwendeten Protokolle ausführlich erklärt und jeweils Tests und Experimente durchgeführt, die den Einsatz von Network Coding unter bestimmten Bedingungen überprüfen sollen. Die Analyse der erhaltenen Daten zeigt einige Verbesserungen duch den Einsatz.

Inhaltsverzeichnis

1 Einleitung

Die Möglichkeit, über das Internet kostenlose Fernsehprogramme aus aller Welt zu empfangen, wird immer beliebter. Herkömmliche Programme wie beispielsweise PPLive greifen dabei auf die Peer-to-Peer Technologie zurück um die Streams an die Clients zu verteilen. Zum Einsatz kommen dabei für gewöhnlich *data-driven* pull-basierte Peer-to-Peer Protokolle. Die beiden Autoren, M. Wang und B. Li, untersuchen in ihren Arbeiten [2] und [3] nun den Einsatz von Network Coding in Peer-to-Peer Live Streaming. Dabei gehen sie der Frage nach: *Wie hilfreich ist Network Coding in Peer-to-Peer Live Streaming?*.

2 Network Coding

Network Coding gewinnt immer mehr an Beliebtheit und gilt in der Literatur als vielversprechende Möglichkeit die Performance in Peer-to-Peer Netzwerken deutlich zu verbessern. Seit der im Jahr 2000 veröffentlichen Definition in dem Paper *„Network Information Flow“*(siehe [1]) beschäftigen sich unzählige Paper mit diesem Thema.

Die Idee von Network Coding besteht darin, Daten in Blöcke zu unterteilen und diese mittels einer Linear Kombination zu codieren. Die so entstandenen codierten Blöcke, können dann verschickt und beim Empfänger decodiert werden. Die Coding Koeffizienten werden dabei zufällig aus einem Galois Feld gewählt. Die bevorzugte Größe des Galois Feldes beträgt GF(2^8), dass heißt unser Galois Feld enthält 2^8 Elemente [2].

2.1 Network Coding in Lava

Eine Genaue Erklärung für Lava ist in Abschnitt 3 zu finden. Der Daten Stream wird in Segmente unterteilt. Jedes Segment entspricht einer gewissen Wiedergabedauer (zum Beispiel entspricht ein Segment einer Sekunde Wiedergabe). Jedes Segment wird in n Blöcke $[b_1, b_2, ..., b_n]$ geteilt. Jeder Block b_i ist dabei k Bytes groß wobei $k = (r \cdot l)/n$ mit r gleich der Streamingrate und l gleich der representierten Abspiellänge eines Segments. Um Blöcke zu codieren, wählt jeder Peer zufällig n Coding Koeffizienten $[c_1^p, c_2^p, ..., c_n^p]$ aus dem Galois Feld GF(2^8). Codierte Blöcke x werden dann durch eine Linear Kombination wie folgt berechnet:

$$x = \sum_{i=1}^{n} c_i^p \cdot b_i^p \tag{1}$$

Die zum decodieren benötigten Koeffizienten, werden im Header eines Codierten Blocks mitgeführt. Dadurch entsteht ein Header overhead von n Bytes pro Codierten Block.

Sind n Codierete Blöcke empfangen worden, kann die vollständige Decodierung durchgeführt werden. Dazu bildet ein Peer eine $n \times n$ Matrix A, in der jede Reihe, den Coding Koeffizienten eines Codierten Blocks entspricht. Mit der folgenden Gleichung, können die originalen Blöcke wieder hergestellt werden:

$$\boldsymbol{b} = \boldsymbol{A}^{-1}\boldsymbol{x}^T \tag{2}$$

Damit der Decodierungsprozess nicht erst gestartet werden muss, wenn n Codierte Blöcke empfangen wurden, verwenden die Autoren die Gauss-Jordan Elimination. Laut [2] ist es damit möglich, schon mit dem empfangen des ersten Codierten Blocks, das Decodieren zu beginnen. Eine Genauer Beschreibung über das Decodieren mit der Gauss-Jorden-Elimination, ist in [2] nachzulesen.

2.2 Network Coding in R^2

Eine genaue Erklärung für R^2 ist wieder in Abschnitt 3 zu finden. Wie auch in Lava, findet die Codierung nur innerhalb eines Segments statt. Auch hier werden Segmente in n Blöcke $[b_1, b_2, ..., b_n]$

geteilt wobei jeder Block eine feste Anzahl an Bytes k hat. Anders als in Lava, werden in R^2 anschließend m Coding Koeffizienten $[c_1^p, c_2^p, ..., c_m^p]$ mit $m \leq n$ aus GF(2^8) gewählt. Weiter werden zufällig m Blöcke $[b_1^p, b_2^p, ..., b_m^p]$, aus allen bisher Empfangenen Blöcken des Segments gewählt und ein codierter Block x der größe k Bytes mit folgender Formel produziert:

$$x = \sum_{i=1}^{m} c_i^p \cdot b_i^p \tag{3}$$

Die Dichte wird dabei bestimmt durch $\frac{m}{n}$. In einer anderen Arbeit der Autoren, M. Wang und B. Li, konnte anhand von Experimenten zeigen, dass die Dichte bis zu 6% hinunter geschraubt werden kann, ohne auf lineare Abhängigkeit zwischen den codierten Blöcken zu führen.

Auch wenn nur m Koeffizienten ausgesucht werden, werden n Bytes Overhead benötigt. Die n Koeffizienten die in den Header eines codierten Blocks eingefügt werden, lassen sich einfach berechnen. Man multipliziere einfach die gewählten $[c_1^p, c_2^p, ..., c_m^p]$ mit der $m \times n$ Matrix der coding Koeffizienten der Empfangenen bzw. vorhandenen Blöcke.

Hier ein kurzes Beispiel: Ein Segment unterteilt in vier Blöcke $b_1, \cdots, b_4$ daraus folgt $n = 4$. Gewählte coding Koeffizienten mit $m = 2$: c_1 und c_2. Wahl der Blöcke b_1 und b_3. Berechne den mitzuschickenden coding Koeffizienten Vektor wie folgt:

Erstelle die $m \times n$ Matrix B für die gewählten Blöcke

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Multipliziere den coding Koeffizienten Vektor mit B und ermittle dadurch den mitzuschickenden coding Koeffizienten Vektor

$$\begin{aligned} \begin{pmatrix} c_1 & c_2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} &= \begin{pmatrix} c_1 \cdot 1 & 0 & c_2 \cdot 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} c_1 & 0 & c_2 & 0 \end{pmatrix} \end{aligned}$$

Der Decodierungsprozess in R^2 entspricht dem in Lava, siehe 2.1.

3 Lava und R^2

Die Folgenden Erläuterungen und Ergebnisse sind alle aus den beiden Paper, der Autoren M. Wang und B. Li, [2] und [3].

Mit der Hilfe von *Lava* [2] soll die Frage geklärt werden, ob der Einsatz von *Network Coding* in P2P Live Streaming Protokollen hilfreich ist. Aus diesem Grund, wurde eine Testumgebung geschaffen, die realen Traffic und eine hoch optimierte Implementation von Network Coding zu Verfügung stellt.

Die Testumgebung ist ein Cluster aus 44 dual-CPU Servern in Lava. Um brauchbare Ergebnisse aus den Messungen zu erhalten und diese fair vergleichen und bewerten zu können, ist es wichtig eine gewisse Anzahl an Design und Implementations Bedingungen zu erfüllen die in Abschnitt 3.1 nachzulesen sind.

Die Analyse der erhalten Daten durch die Experimente zwischen Network Coding und Vanilla (siehe 3.3), führt laut Autoren zu folgenden Ergebnissen [2]: Der Einsatz von Network Coding, ...

- ... verringert den Bandbreiten Verbrauch.
- ... verbessert die Widerstandsfähigkeit auf Netzwerk Dynamiken.

- ... nützt am meisten, wenn die Bandbreite gerade noch über der Streaming Nachfrage liegt.

R^2 verbessert laut den Autoren noch diese Verbesserungen und schafft es eine Verkürzung der Wiedergabeverzögerung zu erziehlen [3]. Ergebnisse die laut Autoren in Tests und Vergleichen der Lava Komponenten mit R^2 ermittelt wurden, sind folgende [3]:

3.1 Die Testumgebung in Lava

Die 44 Server sind untereinander über ein Gigabit Ethernet verbunden und emulieren auf der Anwendungs Schicht für jeden Peer eine korrekte Upload Bandbreite. Um bessere Ergebnisse zu erzielen, stellen die Autoren weitere Bedingungen auf, die ihre Testumgebung erfüllen muss [2]:

1. eine große Anzahl an TCP Verbindungen und UDP Traffic muss von jedem Peer effizient geleitet werden.
2. eine optimierte Implementation der Protokolle, mit Augenmerk auf maximale Performance.
3. Um Live Streaming Protokolle mit und ohne Network Coding fair vergleichen und bewerten zu können, müssen diese mit gleichen Parametereinstellungen laufen können.
4. Ankommende und gehende Peers sollen simuliert werden können.

Der Kern von Lava ist der sogenannte *Algorithmus* der in jedem Peer implementiert ist. In diesem *Algorithmus* sind das Vanilla Protokoll und das randomisierte Network Coding Plugin enthalten. Außerdem, ermöglicht es einem Peer mehrere TCP Verbindungen aufzubauen und zu verwalten [2]. Jeder, in einer Session enthaltene Multimedia Stream, ist in mehrere Segmente unterteilt, die einer gewissen Laufzeit entsprechen (hier: nach [2] eine Sekunde). Im Falle des Gebrauchs von Network Coding, werden diese Segmente nochmals in mehrere Blöcke unterteilt. Genauere Implementierungsdetails sind in [2] nachzulesen.

3.1.1 Streaming in Lava

Um der Bedingung 3. aus Abschnit 3 gerecht zu werden, wird der Network Coding Teil als Plugin auf Vanilla aufgesetzt. Vanilla ist ein *data-driven* pull-based P2P Streaming Protokoll in dem wie gewöhnlich die Peers periodisch Informationen über Segment Verfügbarkeit austauschen (genannt *buffer maps* [2]). Diejenigen Peers, die laut buffer map ein Segment zur Verfügung stellen, werden *seeds* dieses Segments genannt.

Jeder Peer hält pro Session einen *playback buffer* in dem empfangene Segmente nach ihrem Abspielungszeitpunkt sortiert, gespeichert werden. Liegt ein Abspielungszeitpunkt in der Vergangenheit, wird dieses Segment gelöscht. Die Wiedergabe erfolgt nach einer kurzen Zeitspanne, dem so genannten *initial buffering delay*. Peers fordern zu jeder Zeit benötigte Segmente von deren Seeds an wobei die Anzahl der Anforderungen beschränkt ist. Hier ist auch gleich ein Unterschied zum Streaming mit Network Coding denn dort können Peers von mehreren Seeds ihre Blöcke eines Segments herunterladen. Werden Segmente nicht rechtzeitig erhalten, werden diese übersprungen. Die genaue Funktionsweise wann und wieviel Segmente ein Peer anfordert ist in [2] genauer nachzulesen.

3.2 Testumgebung in R^2

Die Testumgebung in denen R^2 getestet wurde, ähneln sehr denen von Lava. Was auch verständlich ist, da alle Protokolle untereinander verlgeichbar sein sollen (siehe Abschnitt 3.1). Alle Experimente, falls nicht anderst angegeben, haben unter folgenden Parametern stattgefunden: Jedes Segment representiert vier Sekunden Wiedergabezeit und ist in 128 Blöcke geteilt. Jede Streaming Session dauert zehn Minuten. Die Puffer Größe wird auf 32 Sekunden gesetzt. Die Wiedergabeverzögerungsdauer beträgt sechzehn Sekunden und die priorisierte Region acht Sekunden (aus [3]).

3.2.1 Streaming in R^2

Anders wie zur Zeit gängigen Peer-to-Peer Protokolle, verfolgt R^2 den Ansatz des *Random Push*. Bei diesem Verfahren, entfällt das Anfordern der fehlenden Segmente durch einen Peer bei den Seeds. An Stelle dessen, wählen die Seeds zufällig ein Segment aus, von dem sie wissen, dass der verbundene Peer dieses noch nicht vollständig besitzt, generieren einen codierten Block und senden diesen dem Peer. Da alle codierten Blöcke gleichwertig sind, können alle Seeds den Peer mit fehlenden Segmenten bedienen, ohne dafür Protokol Nachrichten austauschen zu müssen [3].

Um zu garantieren, das Segmente die dringlicher benötigt werden, weil sie nahe an ihrer Abspielzeit sind, rechtzeitig beim Peer ankommen, besitzt jeder Peer eine so genannte *Priorisierte Region*, siehe Abbildung 1. Alle Segmente, welche in dieser Region liegen, bekommen von den Seeds eine höhere Priorität. Das bedeutet für einen Peer dessen fehlendes Segment innerhalb der Priorisierten Region liegt, dass alle seine Seeds drängen dieses fehlende Segment zu vervollständigen. Unter der Annahme, dass der Peer auch genügend Download Kapazität besitzt, sollte dies den Seeds laut Paper [3] gelingen.

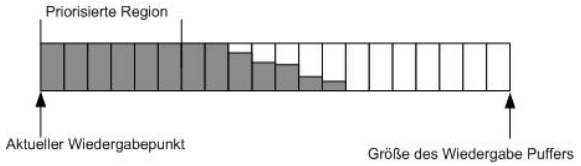


Abbildung 1: Der Widergabe Puffer in R^2 (nach [3]). Der dunkel gezeichnete Füllstand, gibt den Status der Segmente an. Je voller desto vollständiger

Sind alle Segmente innerhalb der Priorisierten Region vorhanden, wählen die Seeds zufällig ein Segment welches sich ausßerhalb befindet. Um dieses Auswahlverfahren zu gewährleisten, verwenden die beiden Autoren als Wahrscheinlichkeitsverteilung eine Weibull Verteilung.

Bleibt noch zu klären, wie die Seeds die Informationen über die fehlenden Segmente erhalten. In *pull*-basierten Protokollen werden dazu periodisch sogenannte *Buffer Maps* [1] unter den Peers ausgetauscht. Die dadurch entstehenden Verzögerungen, wären für das R^2 Protokoll verheerend, da Peers möglicherweise etliche nicht benötigte Blöcke empfangen würden. Um dieses Problem zu umgehen, senden Peers ihre Buffer Maps nicht mehr periodisch, sondern immer dann, wenn sich der Status ihres Wiedergabe Puffers geändert hat. Dies geschieht, wenn etwa ein Segment erfolgreich abgespielt wurde oder eine Segment vollständig empfangen wurde. Um den Overhead möglichst gering zu halten, wird die Buffer Map wann immer es geht in einem zu versendendn codierten Block integriert. Ist die nicht möglich, so wird die Buffer Map an die benachbarten Peers verteilt [3].

Eine weitere Neuerung in R^2 entsteht aus der Verbindung von *Random Push* und *Random Network Coding*. Dadurch ist es nämlich möglich, dass mehrere Seeds ein und das selbe Segment bedienen, während bei *pull*-basierten Protokollen ein Segment immer nur von einem Seed bedient werden kann (dargestellt in Abbildung 2).

Die Anzahl der Peers die ein Seed gleichzeitig bedient ist jedoch beschränkt. Für die Festlegung der oberen Schranke, empfehlen die beiden Autoren, MẆang und BL̇i, eine Lineare Beziehung zwischen der oberen Grenze und der Upload Kapazität [3]. Je größer die Upload Kapazität, desto größer die obere Grenze. Durch diese Grenze erhoffen sich die Autoren, dass neue Segmente so schneller vollständig empfangen und somit weiter verteilt werden können [3].

Beim beitreten neuer Peers in eine Streaming Session, bekommen diese zuerst Buffer Maps ihrer Seeds zusammen mit dem aktuellen Segment welches gerade abgespielt wird, zugeschickt. Drauf hin findet eine Wiedergabe Synchronisation mit den anderen Peers statt. Dies geschiet,

[1] Ist eine Bitmap, die die Verfügbarkeit der Segmente aufzeigt

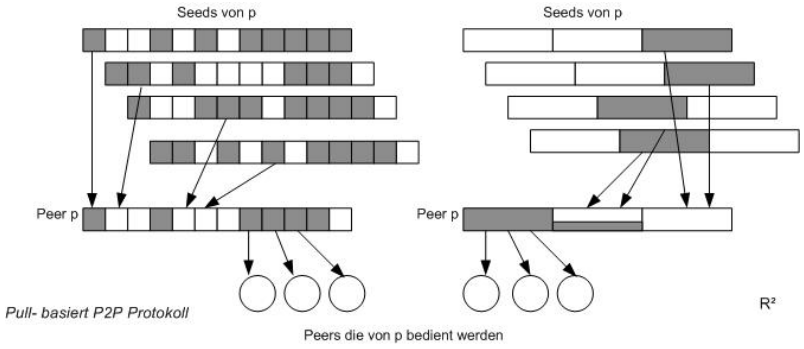


Abbildung 2: Der Vergleich zwischen einem *pull*-basierten Protokoll (links) und R^2 (rechts) (nach [3]). Deutlich zu erkennen: Die Möglichkeit Segmente in R^2 größer zu lassen.

indem die Seeds nur Segmente verschicken, die nach der Wiedergabeverzögerungsdauer liegen. Die Wiedergabe beginnt genau nach dieser Zeit, egal wie weit der Puffer gefüllt ist [3].

3.3 Die Testergebnisse

Im folgenden Abschnitt werden alle drei Protokolle (R^2, Vanilla und Vanilla mit Network Coding Pugin) mit einander an hand von Test verglichen. Die Testbedingungen und Parameter, nach [3], sin in Abschnitt 3.2 zu finden.

Vorstellung der Vergleichswerte nach [3]:

- *Playback Skips*: Anzahl der ausgelassenen Segmente während der Wiedergabe. Ein Segment wird ausgelassen, wenn es zum Wiedergabezeitpunkt nicht vollständig geladen ist.
- *Bandbreiten Redundanz*: Prozentualer Anteil der verworfenen Segmente oder Blöcke aller empfangenen Segmente oder Blöcke.
- *Buffering Level*: Gemessener prozentualer Anteil aller empfangenen Segmente und Blöcke im Wiedergabe Puffer.
- *Uplink Bandbreiten Verbrauch*: Gemessen am Streaming Server.

Alle Messungen sind Durchschnittswerte über alle Peers einer Session.

Test der Skalierbarkeit

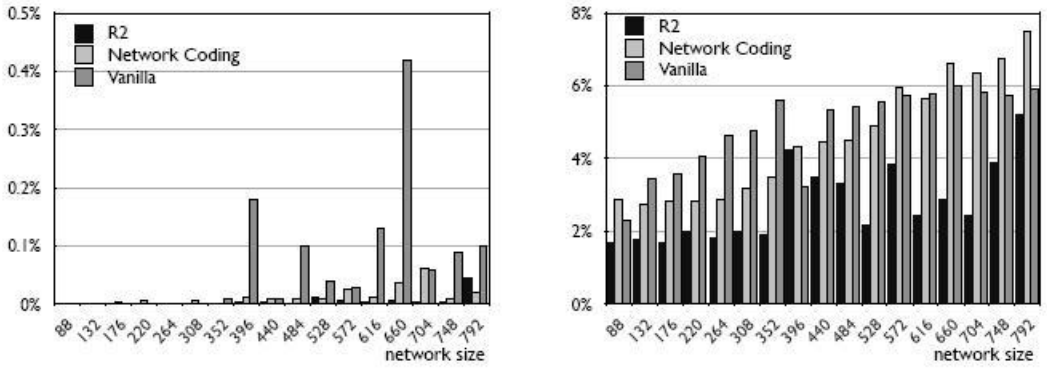
Hier wird die Anzahl der Teilnehmer von Anfangs 88 nach und nach auf 792 erhöht. Die Auswertung des Experiments zeigt Abbildung 3.

Wie man in Abbildung 3 gut sehen kann, sind die beiden Network Coding Protokolle, was die Playback Skips betrifft,klar besser und haben fast konstant bleibende Werte unter 0.1%. Allerdings relativiert sich das Ergebniss, wenn man bedenkt, dass sich alle Werte nur im zehntel Prozent Bereich bewegen und nie über 0.45% hinaus gehen. Ob diese *Peaks* im täglichen Gebrauch wirklich ins Gewicht fallen ist fraglich.

Betrachtet man die Bandbreiten Redundanz, erkennt man auch hier das bessere Abschneiden der Network Coding Protokolle. R^2 weißt sogar durchweg den niedrigsten Wert auf. Auffallend ist jedoch, dass Vanilla ab einer Peer Anzahl von 660, einen nahezu konstanten Wert hält während die Network Coding Protokolle schwanken und Network Coding (hier gemeint als Vanilla mit Network Coding Plugin) sogar höhere Werte als Vanilla annimmt.

Beobachtung der Puffer Level

Auch in Betrachtung des Puffer Levels haben die Network Coding Protokolle die Nase vorne,



(a) Average playback skips.

(b) Average bandwidth redundancy.

Abbildung 3: aus [3]

siehe Abbildung 4. Während bei steigenden Teilnehmer Zahlen R^2 und Vanilla Veränderungen zeigen, beleibt bei Network Coding der Füllstand relativ konstant. Dennoch kann R^2 mit dem höchsten Wert auftrumpfen. Gemeinsam haben alle drei Protokolle, dass sie in der Anfangsphase sehr schnell sind und schon nach kurzer Zeit ein hohes Level erreicht haben.

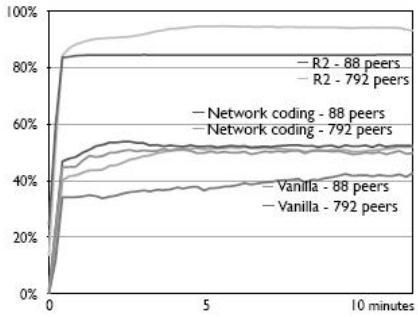


Abbildung 4: aus aus [3]

Weitere Test mit Ergebnissen sind in den Paper [2] und [3] zu finden und können aus Platz Gründen hier nicht weiter erläutert werden.

4 Fazit

Der Einsatz von Network Coding in Peer-to-Peer Live Streaming, führt unter „Labor“ Bedingungen in den meisten Fällen zu Verbesserungen der Wiedergabequalität , einer besseren Reaktionsmöglichket auf Peer Dynamiken. Dennoch fehlen Erfahrungswerte und ausgiebige Tests unter realen Bedingungen. User die neben dem Internet-Fernseh auch noch im Internet surfen und Filesharing betreiben, erzeugen wieder ganz andere Szenarien die es zu Testen und zu Analysieren gilt.

Der Grund weshalb meiner Meinung nach, so hochgepriesene Protokolle wie R^2 noch nicht genutzt werden, liegt darin, dass die bisherigen Angebote vollkommen zufriedenstellend Funktionieren und keiner sich groß den Kopf zerbrechen will, wie ein so kompliziertes Protokoll wie R^2 für den alltäglichen Einsatz implementiert werden kann. Die Autoren schreiben selbst, dass aktuelle Pro-

tokolle mit nur ein paar tausend Zeilen Code schnell und funktionsfähig implementiert werden können. R^2 hat einen bedeutent höheren Aufwand zu verzeichnen.

Literatur

[1] R. Ahlswede, Ning Cai, S. Y. R. Li, and R. W. Yeung. Network information flow. 46(4):1204–1216, July 2000.

[2] Mea Wang and Baochun Li. Lava: A reality check of network coding in peer-to-peer live streaming. In *Proc. INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1082–1090, 6–12 May 2007.

[3] Mea Wang and Baochun Li. R2: Random push with random network coding in live peer-to-peer streaming. 25(9):1655–1666, December 2007.