Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht - SoSe 2010

André Goldflam

Implementierung einer Last-Test-Anwendung als Vorbereitung für den Vergleich von Interest Management Algorithmen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

Längst ist der Bereich der sogenannten „Massive-Multiplayer-Online-Games“ (MMOG) zu einem lukrativen Wirtschaftszweig geworden. Da die Kommunikations-Ressourcen zum Austausch von Spiel-Informationen dennoch begrenzt sind, werden viele verschiedene Techniken eingesetzt, um eine möglichst große Anzahl von gleichzeitigen Spielern zu ermöglichen. Dabei gilt die Skalierbarkeit als ein Kernproblem von MMOGs. Würde ein Spieler in einem MMOG alle Daten erhalten, die alle anderen Spieler erzeugen (z.B. durch Bewegung oder Interaktion mit der Umwelt), wäre das Datenaufkommen auf Seiten des Servers sowie des Clients nicht zu bewältigen. Das als „Interest Management“ (siehe Morse u. a. (2000)) bekannte Verfahren bestimmt dabei, welche Informationen jeder einzelne Spieler von dem Server erhält und reduziert so maßgeblich die benötigten Ressourcen. Soll das Interest Management dabei in einem realitätnahen Umfeld getestet werden, ist eine performante Server-Struktur notwendig, die dem Interest Management gewisse Verantwortlichkeiten abnimmt (z.B. Verbindungsmanagement, Nebenläufigkeit, etc.). Das in diesem Projekt eingesetzte Photon-Framework [1] wird in Abschnitt 2.1 näher beschrieben. Weiterhin soll ein Vergleich von verschiedenen Interest Management Algorithmen erfolgen, ist zum einen ein bestimmte Art von Vorgehen und zum anderen die Verwendung verschiedener Werkzeuge notwendig. Innerhalb dieses Projektes soll daher auf der Grundlange des Photon-Frameworks ein Last-Test implementiert werden. Ziel ist es einerseits, eine Aussage über die Nutzbarkeit des Frameworks für die Umsetzung eines Interest Management Frameworks auf Basis von Photon treffen zu können und andererseits, grundlegende Erfahrungen in Bezug auf die Herausforderungen beim Messen und Vergleichen von nebenläufigen Anwendungen zu machen. Diese Projektarbeit ist daher als Vorbereitung auf die Masterarbeit „Vergleich von Interest Managment Algorithmen“ zu sehen.

1.1. Last-Test

Das Photon-Framework ist eine Multiplayer-Middleware der Firma Exit Games [2] und wird als sehr effizient und performant beworben. Photon ist auch bereits mehrfach kommerziell im Einsatz (z.B. Paradise Paintball [3] oder World Golf Tour [4]) und gilt als bewährt. Paradise Paintball wird dabei von ca. 500.000 Menschen im Monat über das Internet gespielt (siehe cmune (2010)) und benötigt eine gewisse Infrastruktur, um diesen Bedarf zu decken. Während des Betriebes eines Multiplayer-Spieles, wie Paradise Paintball, sind die Unterhaltungskosten kein unerheblicher Faktor. Spieleserver mit einer schnellen und vor allem garantierten Bandbreite sind mit relativ hohen monatlichen Kosten verbunden. Daher stellt sich für den Betreiber und

[1] http://www.exitgames.com/Photon
[2] http://www.exitgames.com/
[3] http://paradisepaintball.cmune.com/
[4] http://www.wgt.com/

Entwickler eines Spieles die Frage, wie viele Spieler er mit seinem Server bedienen kann. Neben der betriebswirtschaftlichen Frage ist es auch während der Weiterentwicklung des Photon-Frameworks wichtig, mögliche Auswirkungen auf die Leistungsfähigkeit des Framworks schnell zu erkennen. Manche Problematiken treten z.B. erst zu Tage, wenn der Server eine gewisse Gesamt-Last verarbeiten muss oder nur in speziellen Situation, wenn z.B. alle Spieler eines Spieles sich nur in einem kleine Bereich aufhalten. Zusätzlich ist eine Bewertung der Skalierbarkeit des Framworks interessant. Die genannten Punkte treffen ebenso auf den in der Masterarbeit angestrebten Vergleich von Interest Management Algorithmen zu. Daher soll im Rahmen eines Last-Tests festgestellt werden, wie das Photon-Framework das im folgenden Abschnitt beschriebene Szenario bewältigt.

1.2. Szenario

Es soll ein Spieleserver simuliert werden, auf dem mehrere Spiele gleichzeitig stattfinden können. In jedem Spiel können bis zu 8 Personen teilnehmen. Das zu simulierende Spiel soll aus dem Bereich der 3D-Echtzeit-Spiele stammen. Um dem Spieler die Illusion zu geben, dass sich seine Aktionen in dem Spiel unmittelbar auswirken, sollte das Spiel mindestens 30 Bilder pro Sekunde zeichnen. Da es zu Aufwändig wäre, bei jedem gezeichneten Bild eine Zustandsaktualisierung zu berechnen und zu versenden, werden verschiedene Techniken wie z.B. Dead Reckoning (siehe Singhal und Zyda (1999)) eingesetzt, um die notwendigen Aktualisierungen zu reduzieren. Im Rahmen dieses Szenarios soll davon ausgegangen werden, dass eine Aktualisierung bei ca. jedem dritten Frame ausreichend ist. Somit sind für jeden Spieler ca. zehn Aktualisuerngen pro Sekunden notwendig. Als Obergrenze der zur Verfügung stehenden Bandbreite für den Server soll 100 MBit gelten.

1.3. Analyse des Szenarios

Die Spieleranzahl wird durch die zur maximale Bandbreite von 100MBit begrenzt. Zum Speichern der Zustandsaktualisierungen sollen 20Bytes genügen (3 shorts für Koordinaten = 6 Bytes, 14 Bytes für restliche Informationen). Die folgende Aufstellung wurde unter der Verwendung von Wireshark[5] erarbeitet. Die Tabelle 1 zeigt, wie viele Bytes bei einer Übertragung durch die Protokoll-Header entstehen. Eine Beschreibung des ENet-Protokolls ist in Abschnitt 2.1 zu finden. Die Tabelle 2 zeigt die Bytes, die von Photon zur Übertragung einer Zustandsaktualisierung notwendig sind. Insgesamt werden also 105 Bytes pro Aktualisierung an Bandbreite benötigt.

Nimmt man weiterhin an, dass nicht die vollständigen 100MBit ausgelastet werden dürfen, da ansonsten zu viele Pakete wegen der Last neu gesendet werden müssen, soll mit 80%

[5] *http://www.wireshark.org/*

Netzwerkkarte	IP	UDP	ENet	ENet-Command	unreliableSequenceNumber	Gesamt
14	20	8	12	12	4	= 70

Tabelle 1: Anzahl Bytes durch Protokolle

Photon	Operation-Code	Data-Set	pro Parameter	pro Typ in Daten	Last-Daten	Gesamt
9	1	3	1	1	20	= 35

Tabelle 2: Anzahl Bytes durch Photon-Last

der verfügbaren Bandbreite, also 80 MBit /s gerechnet werden. Daraus ergibt sich eine mögliche Bandbreite von 10.485.760 Bytes / s. Die zu verarbeitende Obergrenze liegt also bei $\frac{10.485.760 Bytes}{105 Bytes}$ =99864 Aktualisierungen / s.

Die Zustandsaktualisierung eines Spielers wird durch den Spieleserver an alle anderen Teilnehmer gesendet, die sich in demselben Spiel befinden. Eine derartige Duplizierung einer Zustandsaktualisierung soll im Folgenden Ereignis genannt werden. Befinden sich also 8 Spieler gemeinsam in einem Spiel und jeder Spieler erzeugt 10 Aktualisierungen pro Sekunden (= 80 Aktualisierungen / s), werden durch den Spieleserver 560 Ereignisse pro Sekunde erzeugt. Addiert man die Aktualisierungen mit den Ereignisse (80 + 560), werden insgesamt 640 Zustandsaktualisierungen / s gesendet. Wenn man nun die verfügbaren Aktualisierungen / s durch die Anzahl der Aktualisierungen teilt ($\frac{99864}{560}$), sind bei 8 Spielern pro Spiel insgesamt 156 Spiele möglich. Die Tabelle 3 verdeutlicht die Anzahl der Spiele in Relation zu der Anzahl ihrer Teilnehmer.

# Spieler pro Spiel	8	7	6	5	4
# Aktualisierungen pro Spieler	10	10	10	10	10
# Aktualisierungen pro Spiel	80	70	60	50	40
# Ereignisse pro Spiel	560	420	300	200	120
# Summe Ereignisse + Aktualisierungen	640	490	360	250	160
# Anzahl mögliche Spiele	156	204	277	399	624
# Gleichzeitige Spieler	1248	1427	1664	1997	2497

Tabelle 3: Anzahl Spiele in Relation zu Teilnehmern

Zur Überprüfung der vorher genannten Zahlen sollen bei der Last-Test-Anwendung und dem zu testenden Server folgende Performance-Indikatoren überwacht werden:

- Server
  - CPU-Auslastung

  - Speicher-Verbrauch
  - eingehende sowie ausgehende Bytes über Netzwerkschnittstelle / s
  - Anzahl Spieler

- Client
  - CPU
  - Memory
  - durchschnittliche ausgehende Operations / s, pro simulierten Client

Das Ziel dieser Projektarbeit ist es, eine Last-Test-Anwendung zu implementieren, mit der der hier beschriebene Spieleserver, wie vorgestellt, simuliert und getestet werden kann.

2. Implementierung

2.1. Eingesetzte Technologien

Photon-Framework

Als Basis der Implementierung soll das Photon-Framework (Games (2010)) der Firma Exit Games zum Einsatz kommen. Das Framework bietet dabei eine Basis für die Entwicklung des Clients sowie des Servers. Folgend soll das Photon-Framework in seinen Kernmerkmalen vorgestellt werden.

Kern Der Kern des Frameworks, bestehend aus einem Socket-Server sowie der darüber liegende Netzwerkschicht, wurde in nativem C / C++ geschrieben und besonders effizient implementiert, um auch hohen Latenzanforderungen gerecht zu werden. Die darüber liegende Serverlogik ist in C# realisiert. Nachdem die empfangenen Daten durch die Netzwerkschicht verarbeitet und der höheren Programmlogik übergeben worden sind, werden die Daten einer interne Operation-Queue hinzugefügt und so eine totale Ordnung hergestellt (siehe Paragraph „Nebenläufigkeit“). Die Programmlogik für Client und Server kann dabei in .NET / C# entwickelt werden und bietet dem Entwickler den entsprechenden Komfort einer Hochsprache.

Netzwerk Das TCP-Protokoll garantiert, dass die Daten zuverlässig das gewünschte Ziel erreichen. Dabei ist ein entsprechender Overhead notwendig, um diese Zustellungsgarantie zu ermöglichen. Bei Multiplayer-Spielen besitzen manche der zu übertragenden Daten nur eine sehr kurze Lebensdauer (z.B. Positionsdaten durch Bewegungen). Kommen diese Daten erst verspätet an, sind sie bereits veraltet und nutzlos. Es ist daher wichtig, dass diese Daten besonders kompakt und schnell gesendet werden können. Andererseits gibt es auch Daten, welche eine Zustellungsgarantie wie bei TCP voraussetzen, z.B. der Schuss in einem Action-Spiel. In Photon wird daher das Protokoll ENet (Salzman (2010)) eingesetzt, welches das UDP-Protokoll so erweitert, dass auch über UDP Daten mit Zustellungsgarantie gesendet werden können. Dies ermöglicht das effizienten Versenden und Empfangen von den beschriebene Datentypen, ohne einen Protokollwechsel. Die Tabelle 1 im Anhang zeigt, wie viele Bytes pro Paket durch ENet verbraucht werden.

Nebenläufigkeit Retlang (Rettig und Nash (2010)) ist eine hoch performante C# Threading-Bibliothek welche Nebenläufigkeit in Anlehnung an das Aktor-Modell (siehe Agha (1990)) implementiert. Das Aktor-Modell löst das Problem der Nebenläufigkeit, indem jeder Aktor nur über Nachrichten mit anderen Aktoren kommunizieren kann. Dabei werden die Nachrichten in der Reihenfolge bearbeitet, in der sie empfangen worden sind. In Retlang können an sogenannte Fibers Nachrichten gesendet werden, wobei die Nachrichten in der Regel einen Callback auf eine auszuführende Methoden darstellen. Beim Verarbeiten der Nachricht wird der Callback aufgerufen und nach Beendigung des Callbacks kann die nächste Nachricht abgearbeitet werden. Retlang baut dabei auf den Threading-Grundlagen der .NET-Plattform auf und gilt als sehr effizient. Retlang ist dabei nur für die lokale Nebenläufigkeit gedacht, eine wie durch das reine Aktor-Modell mögliche Verteilung über das Netzwerk ist nicht möglich.

PerfMon

Die Messungen der Abschnitt 1.3 angegebenen Daten erfolgt über PerfMon, ein Leistungsüberwachungstool welches in allen Windows-Versionen verfügbar ist. Um die Daten während eines Testlaufes zu speichern, müssen für jeden Datensatz sogenannte benutzerdefinierte Sammlungssätze angelegt werden. Der Photon-Server registriert seine eigenen Performance-Counter in PerfMon. So kann auf die System- und Server-Daten in derselben Art und Weise zugegriffen werden.

2.2. Architektur

Die Last-Test-Anwendung besteht aus einer Client- sowie einer Serverkomponente. Die Last-Test-Clients kommunizieren dabei über den Koordinationsserver um das Vorgehen abzustim-

men, der Server selbst enthält keine für den Last-Test notwendige Logik. Während des Last-Tests senden die Clients Nachrichten an den Spieleserver. Im folgenden Abschnitt „Last-Test-Serverkomponenten“ wird beschrieben, wie auf Grundlage des Photon-Frameworks der Spiele- sowie der Koordinationsserver realisiert worden sind.

Last-Test-Serverkomponenten

Der Photon-Server selbst ist nur ein Container für Photon-Server-Anwendungen. Ein Photon-Server kann dabei mehrere Anwendungen enthalten. Ein Client verbindet sich immer mit einer konkreten Server-Anwendung. Sendet der Client Nachrichten an den Server, leitet dieser die Nachrichten an die zuständige Anwendung weiter. Die zwischen Photon-Server und Photon-Anwendung ausgetauschten Nachrichten werden Operations genannt. Eine Nachricht die von einem Client über den Server an alle anderen Clients gesendet werden soll, wird als Photon-Event bezeichnet. Es existiert bereits eine Server-Anwendung namens „Lite“, welche folgende grundlegende Kommunikations-Mechanismen zur Verfügung stellt:

- Während die Clients mit Lite verbunden sind, wird die Verbindung automatisch aufrechterhalten
- Clients können einem bestimmten durch einen Namen gekennzeichneten Bereich beitreten
- Wenn ein Client einem Bereich beigetreten ist, werden die Photon-Events nur an alle Clients innerhalb des Bereiches gesendet

Die Klasse ExitGames.Client.Photon.LitPeer stellt dabei die für den Client notwendige Implementierung zur Kommunikation mit der Lite-Serveranwendung zur Verfügung. Die vorgestellte Lite-Anwendung ermöglicht bereits den Betrieb eines Spieleservers, wie in der Analyse in Abschnitt 1.3 beschrieben wurde. Die zur Koordination notwendige Kommunikation kann ebenso über eine Lite-Anwendung abgewickelt werden, wobei sich alle beteiligten Last-Test-Clients in einem eigenen Bereich befinden müssen. Um die Ergebnisse bei der Durchführung eines Last-Tests nicht zu verfälschen, sollten die Lite-Anwendungen, die zur Koordination und als Spieleserver eingesetzt werden sich nicht auf demselben Photon-Server befinden. Zusätzlich hat dieses Vorgehen den Vorteil, dass auf keinem Rechner eine für den Last-Test spezifische Serveranwendung installiert werden muss. So ist es besonders einfach, den Client auf den verschiedenen Rechnern auszuführen.

Last-Test-Clientkomponente

Der Last-Test-Client wird durch die Klasse LoadClient realisiert. Der LoadClient selbst wird von der Klasse GameClient, welche wiederum von BasicClient abgeleitet wird. BasicClient besitzt einen ExitGames.Client.Photon.PhotonPeer und verfügt bereits über die grundlegenden Implementierung, die zum Verbinden mit einer Anwendung auf dem Photon-Server notwendig ist. Der GameClient fügt über den von ExitGames.Client.Photon.PhotonPeer abgeleiteten ExitGames.Client.Photon.LitePeer die Funktionalität zum Betreten und Verlassen eines Bereiches einer Lite-Anwendung hinzu. Dabei wrappen GameClient und BasicClient die Funktionalität des PhotonPeer- bzw. GamePhotonPeerListener und ermöglichen es vererbten Klassen, direkt auf die erfolgreiche Herstellung einer Verbindung sowie dem erfolgreichen Beitreten eines Bereiches zu reagieren. Entsprechend wird bei jeder weiteren Vererbung die Funktionalität erweitert. Abbildung 1 zeigt die Vererbungsstruktur der verschiedenen Client-Klassen.

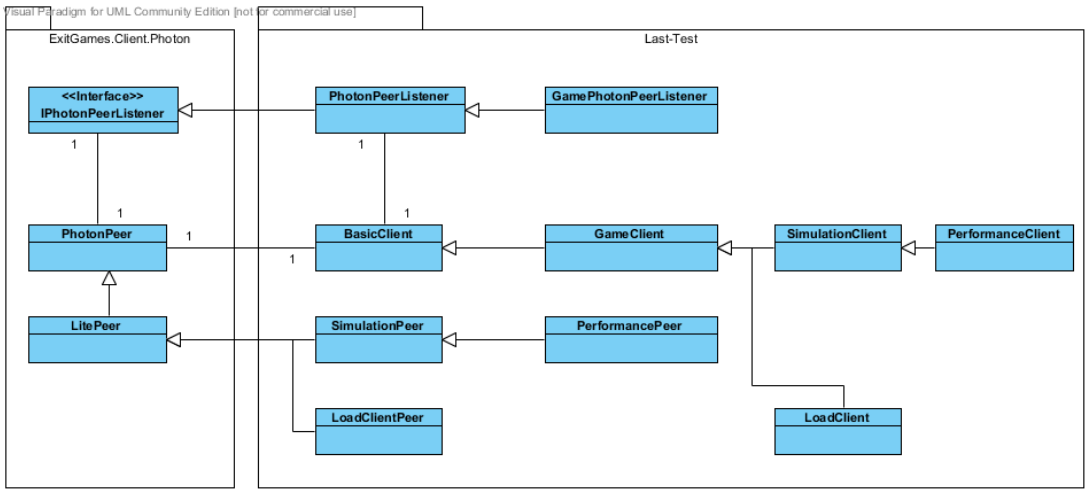

Abbildung 1: Überblick der Clients

Der LoadClient enthält die Logik, die für die Koordination aller beteiligten LoadClients eines Last-Tests notwendig ist. Ein LoadClient kann entweder im Master- oder im Slave-Modus agieren. Es darf dabei immer nur einen Master geben. Der Master koordiniert den Last-Test und führt zusätzlich alle Aktionen aus, die auch die Slaves ausführen. Abbildung 2 zeigt den Aufbau der LoadClient-Klasse. Die Kommunikation der Komponenten wurde über C#-Events realisiert, um eine enge Bindung der verschiedenen Komponenten zu verhindern. Ein Last-Test wird in folgenden Schritten ausgeführt:

1. Alle LoadClients melden sich beim Koordinationsserver an. Nach der Anmeldung befinden sich alle LoadClients im Slave-Modus

2. Ein LoadClient wechselt in den Master-Modus
3. Der Master sendet das StartPerformanceTest-Photon-Event mit Informationen über die zu testende Simulation an alle Slaves
4. Die Slaves überprüfen, ob bereits ein Performance-Test für diese Simulation durchgeführt wurde.
5. Hat der Slave noch keinen Performance-Test durchgeführt, beantragt er einen Freigabe für seinen Performance-Test beim Master
6. Hat der Slave bereits einen Performance-Test durchgeführt, sender er seine Performance-Ergebnisse an den Master
7. Der Master empfängt alle Performance-Anträge der Slaves und gibt dabei dem Slave, der zuerst den Antrag gestellt hat eine Freigabe
8. Der Slave, der die Freigabe erhält, führt den Performance-Test durch und sendet seine Ergebnisse an den Master
9. Der Master empfängt die Ergebnisse und sendet dem nächsten Slave die Freigabe. Sind alle Slaves fertig, führt der Master selbst seinen Performance-Test durch
10. Nachdem alle Performance-Tests abgeschlossen sind, sendet der Master das Startsignal für die Last-Test-Simulation und jeder Slave sowie der Master führen die Simulation entsprechend ihres vorherigen Performance-Tests aus

Die Last-Test-Simulation wird über den SimulationTestManager abgewickelt. Der Manager erhält das auszuführende SimulationSetup, welches eine Liste mit den auszuführenden Operations enthält, sowie die Informationen, wie viele Spiele mit wie vielen Spielern simuliert werden sollen und wann die Simulation als abgeschlossen gilt. Anschließend instanziiert der SimulationTestManager einen MultipleGameSimulatorTest, der wiederum für jedes gefordertes Spiel ein SingleGameTest erstellt. Der Ablauf einer einzelnen Last-Test-Simulation wird wie folgt durchgeführt:

1. SimulationTestManager startet die Vorbereitung des Tests. Alle SimulationClients verbinden sich mit dem Spieleserver
2. SimulationTestManager erhält die Nachricht, dass sich alle SimulationClients erfolgreich verbunden haben. SimulationTestManager startet die Ausführung des Tests
3. Nachdem alle SimulationClients den Test ausgeführt haben, wird dies an den SimulationTestManager gemeldet
4. Nach Abschluss der Simulation weist der SimulationTestManager alle SimulationClients an, die Verbindung zum Spieleserver abzubrechen

5. Schließlich meldet der SimulationTestManager die erfolgreiche Durchführung der Simulation

Dabei werden bei der Durchführung eines Last-Tests die verschiedenen LoadClients jeweils über die Simulationsphasen „Vorbereitung“,„Ausführung“ und „Verbindungsabbruch“ synchronisiert.

Der Performance-Test wrappt dabei die Funktionalität der Simulation und zählt bzw. misst alle notwendigen Kennzahlen. Die wichtigste Kennzahl ist dabei die durchschnittliche Anzahl Operations / s pro SimulationClient. Sie legt fest, ob der LoadClient einen weiteren Performance-Test mit mehr Spielen durchführt oder den Performance-Test abschließt und die maximale Anzahl an gleichzeitigen Spielen als Ergebnis an den Master sendet. Ausgeführt wird der Performance-Test über den PerformanceTestManager.

Die Eingabe der Steuerbefehle für den LoadClient erfolgt über eine einfache Konsolenanwendung.

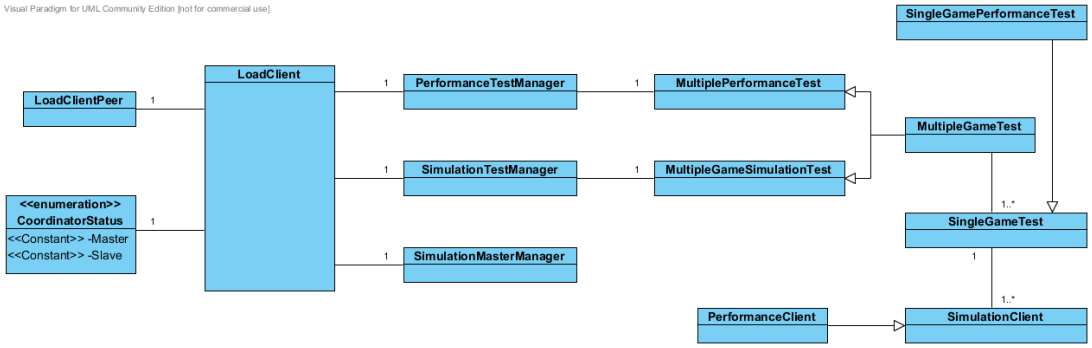

Abbildung 2: Überblick des LoadClients

2.3. Implementierungsstand

Der LoadClient und die vorgestellten Abläufe sind vollständig implementiert. Allerdings kann die LoadClient-Anwendung nur einen vollständigen Last-Test durchführen. Danach muss die Anwendung neu gestartet werden, um einen fehlerfreien Ablauf zu gewähren.

3. Last-Test

Im Rahmen dieser Ausarbeitung soll nur ein exemplarischer Last-Test den Ablauf und die Ergebnisse des Tests demonstrieren. Die folgende Hardware war Grundlage der anschließenden Ergebnisse.

- Client A
  - Core 2 Duo T8300, 2.40 GHz
  - 4 GB RAM
  - Windows 7, 64 Bit
- Client B und gleichzeitig Koordinationsserver
  - Intel Pentium Dual T2390, 1.86 GHz
  - 3 GB RAM
  - Windows Vista, 32 Bit
- Spieleserver
  - Core i7-620M, 2.67 GHz
  - 8GB Ram
  - Windows 7, 64 Bit

Getestet wird eine Simulation, bei der 8 Spieler in einem Spiel miteinander kommunizieren. Während eines einzigen Durchlaufs eines Performance-Tests wird für 10 Sekunden Last erzeugt. Die Last-Test-Simulation selbst wird 30 Sekunden lang durchgeführt. Alle Ergebnisse des Last-Tests finden sich im Anhang in Abschnitt A.

3.1. Auswertung

Abbildung 6 zeigt deutlich, wie sich zuerst Client A verbindet und einen Performance-Test durchführt, dann Client B ebenso einen Performance-Test ausführt und anschließend beide gemeinsam den Last-Test abwickeln. Die Tabelle 4 enthält die Ergebnisse der Performance-Tests der beiden Clients sowie die durchschnittliche Anzahl Operations / s. Diese Kennzahl wurde aus der Konsole abgelesen. Insgesamt werden also während des Last-Tests 640 Spieler simuliert. Rein rechnerisch sollten diese Spieler eine Last von 5.376.000 Bytes erzeugen. In Abbildung 5 ist jedoch während des Last-Tests nur eine maximale Gesamtauslasung der

Netzwerkschnittstelle von ca. 3.500.000 Bytes festzustellen. Dabei dauert diese Auslastungsphase auch nur, die für einen Last-Test vorgeschriebenen 30 Sekunden an. Zusätzlich zeigt Abbildung 4, dass die CPU-Auslastung des Servers bei maximal 40% liegt. Daher scheint es unwahrscheinlich, dass die reduzierte Netzwerk-Last pro Sekunde auf eine verzögerte Auslieferungszeit zurückzuführen ist. Möglicherweise ist dieses Phänomen auf den Umstand zurückzuführen, dass eine Spielsimulation von einem einzelnen Client durchgeführt wird. Dieser Umstand soll in Zukunft näher untersucht werden.

Bemerkenswert ist, dass die benötigten Speicherbereiche von Client A (Abbildung 8), Client B (Abbildung 11) und des Servers (Abbildung 7) unabhängig von der Last nicht erweitert werden müssen.

Vergleicht man die CPU Auslastung von Client A und Client B (siehe Abbildung 8 und 10), tritt deren unterschiedliche Hardware deutlich zu Tage.

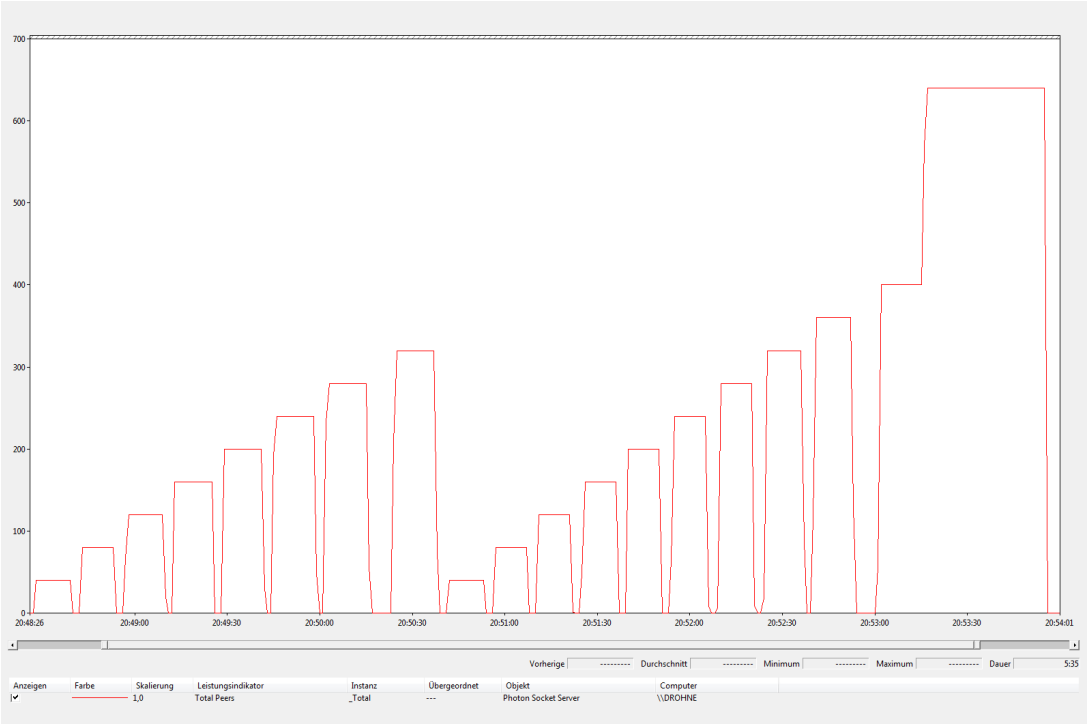

Abbildung 3: Server: Spieler

	Client A	Client B
Anzahl Spiele mit 8 Spielern	45	35
Anzahl Spieler	360	280
Durchschnittliche Anzahl Operations / s	8,169245	8,124778

Tabelle 4: Ergebnis des Performance-Test von Client A und Client B

4. Fazit

Zu Anfang der Ausarbeitung wurde der Sinn und Zweck der Umsetzung eines Last-Test-Servers auf der Grundlage des Photon-Frameworks beschrieben. Anschließend wurde ein Szenario aufgestellt und dieses analysiert. Die sich daraus ergebenden Erkentnisse wurden für die Entwicklung des LoadClients eingesetzt. Ebenso wurden die genutzten Technologien beschrieben. Im Abschnitt 3 wurde probeweise ein Last-Test durchgeführt und die Ergebnisse diskutiert. Die Anomalie der zu geringen Netzwerklast soll zukünftig untersucht werden. Die Umsetzung der Last-Test-Anwendung kann daher als erfolgreich bewertet werden. Ein Einsatz des Photon-Frameworks im Rahmen eines Vergleiches von Interest Management Algorithmen scheint erfolgsversprechend.

Literatur

[Agha 1990] AGHA, Gul: Concurrent object-oriented programming. In: *Commun. ACM* 33 (1990), Nr. 9, S. 125–141. – ISSN 0001-0782

[cmune 2010] CMUNE: *Paradise Paintball Passes the 500,000 Monthly Users Mark*. 2010. – URL http://www.cmune.com/index.php/2010/05/30/paradise-paintball-passes-the-500000-monthly-users-mark/

[Games 2010] GAMES, Exit: *Official Photon-Site*. 08 2010. – URL http://www.exitgames.com/Photon

[Morse u. a. 2000] MORSE, Katherine L. ; BIC, Lubomir ; DILLENCOURT, Michael: Interest Management in Large-Scale Virtual Environments. In: *Presence: Teleoper. Virtual Environ.* 9 (2000), Nr. 1, S. 52–68. – ISSN 1054-7460

[Rettig und Nash 2010] RETTIG, Mike ; NASH, Graham: *Retlang Project Home*. 2010. – URL http://code.google.com/p/retlang/

[Salzman 2010] SALZMAN, Lee: *ENet*. 2010. – URL http://enet.bespin.org

[Singhal und Zyda 1999] SINGHAL, Sandeep ; ZYDA, Michael: *Networked virtual environments: design and implementation*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1999. – ISBN 0-201-32557-8

A. Anhang

A.1. Resultat Last-Test

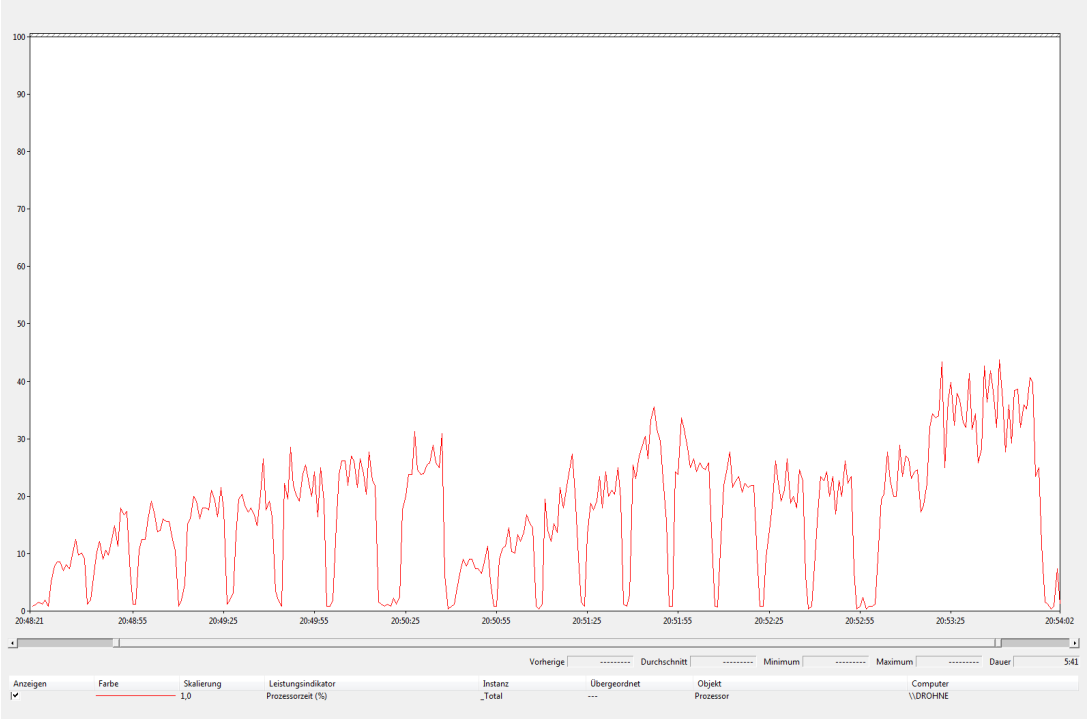

Abbildung 4: Server: genutzte CPU in %

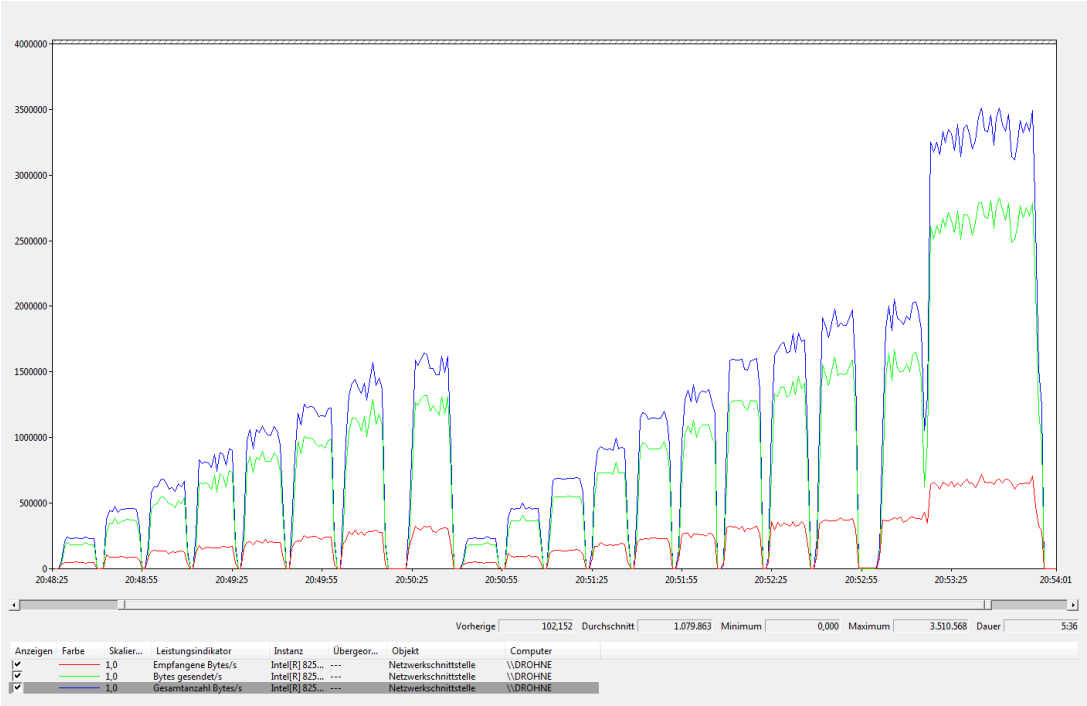

Abbildung 5: Server: Netzwerkschnittstelle

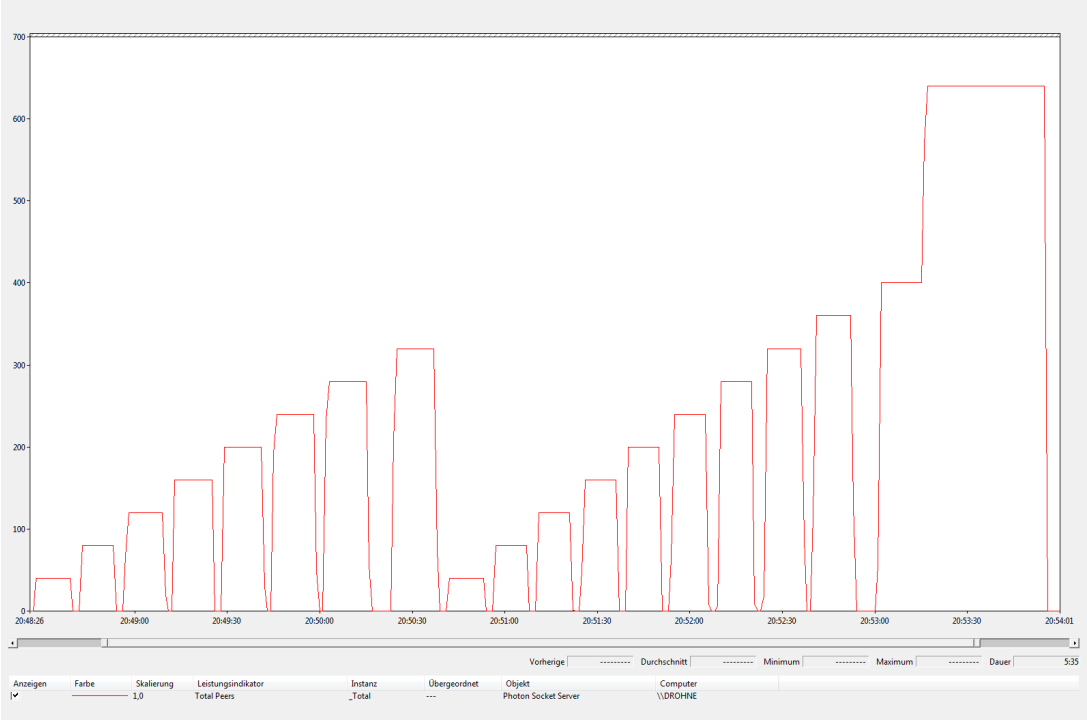

Abbildung 6: Server: Spieler

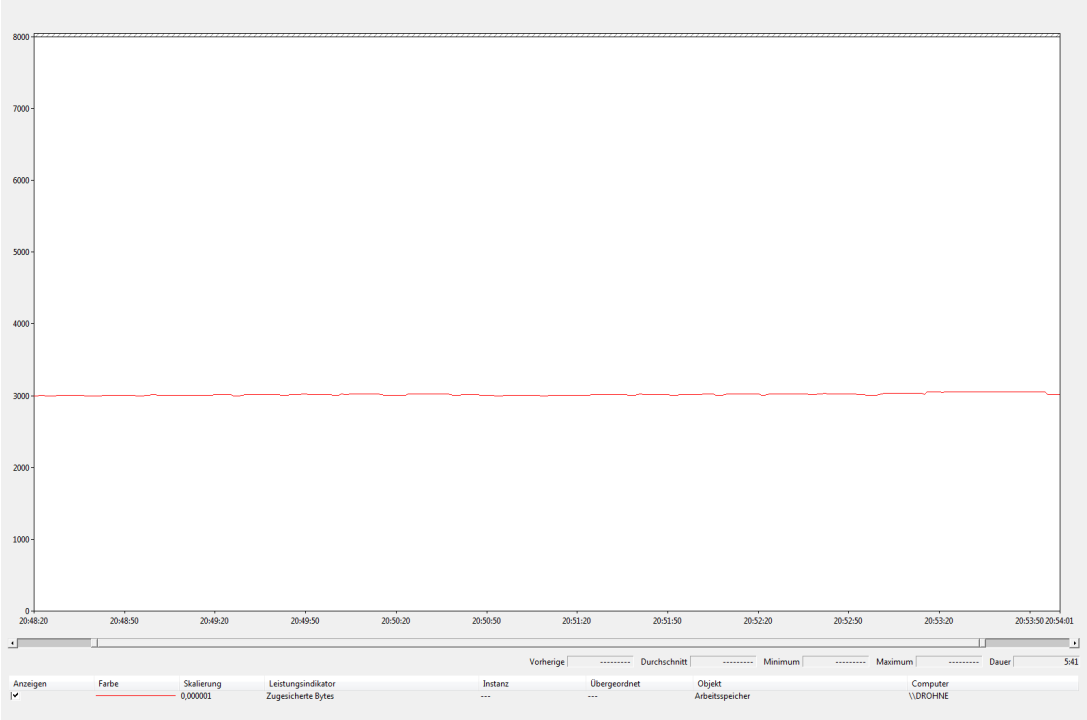

Abbildung 7: Server: genutzter Speicher in Bytes

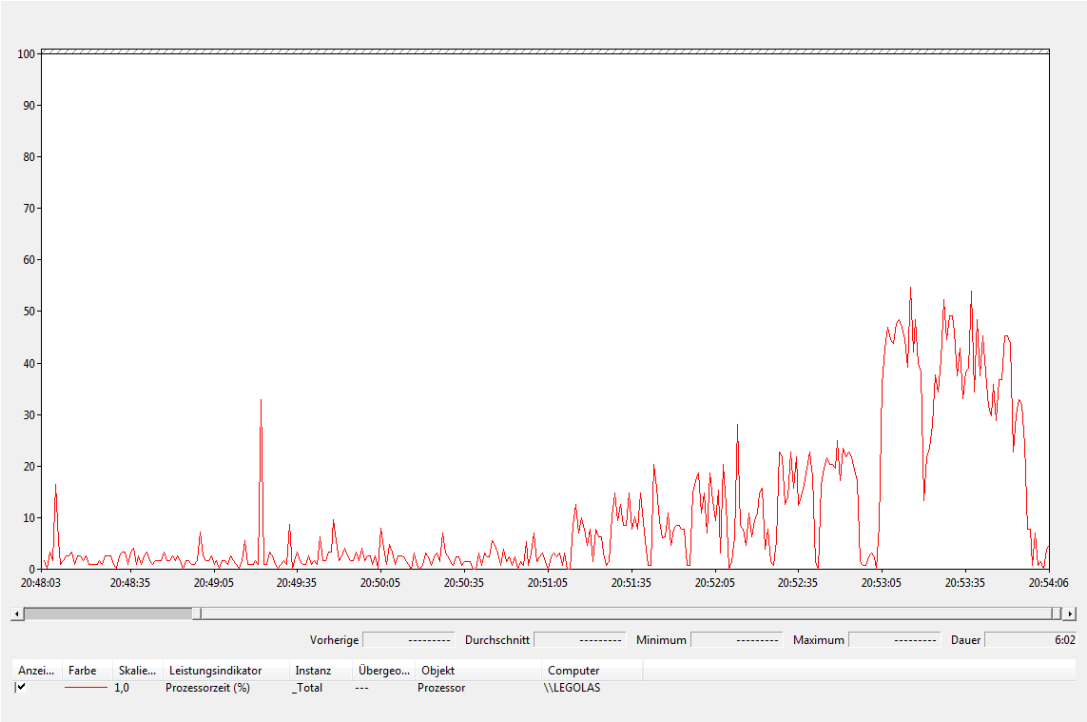

Abbildung 8: Client A: genutzte CPU in %

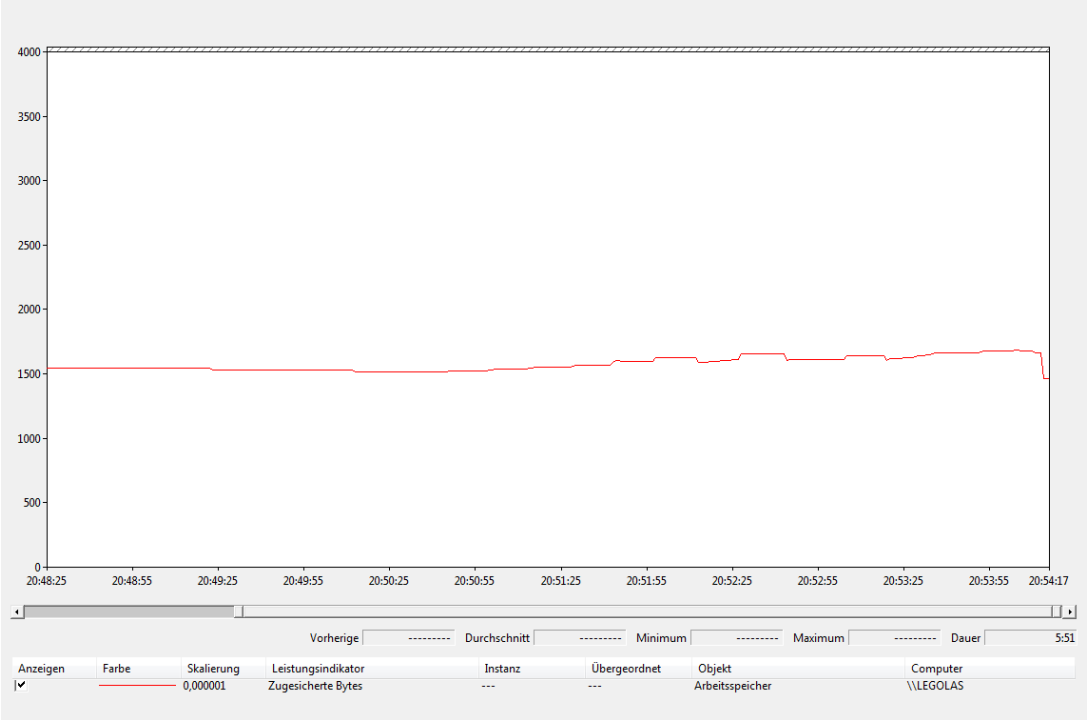

Abbildung 9: Client A: genutzter Speicher in Bytes

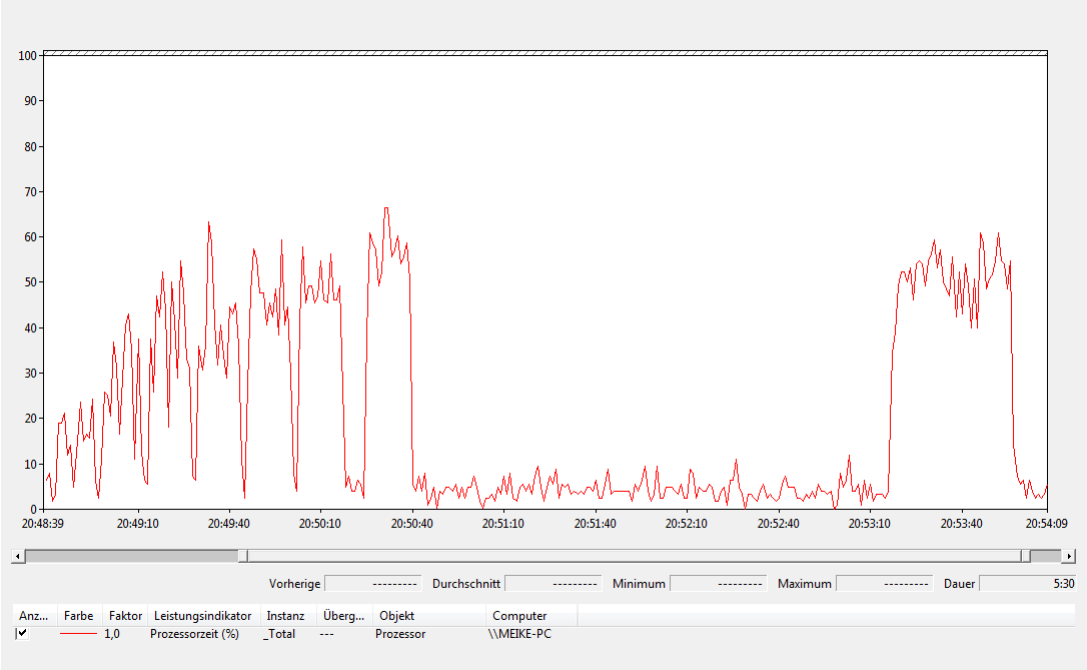

Abbildung 10: Client B: genutzte CPU in %

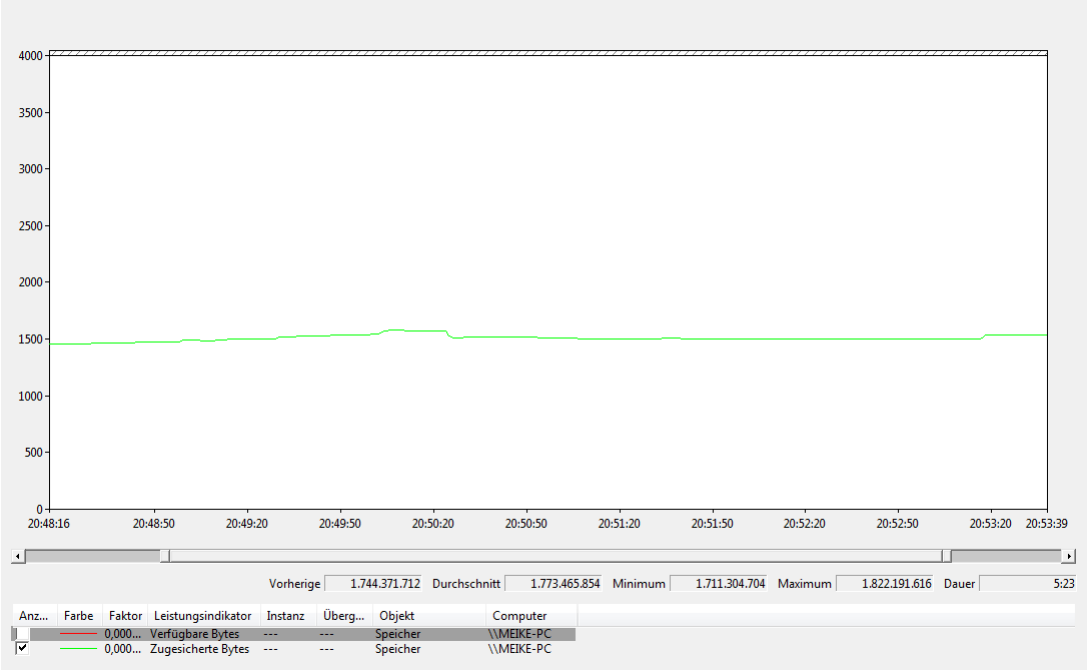

Abbildung 11: Client B: genutzter Speicher in Bytes