ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Hannes Geissbühler

Implementation eines Auktions-basierten Pricing Mechanismus für Peer-to-Peer Systeme

Student Thesis SA-2004-26
Summer Term 2004

Tutor: David Hausheer

Supervisor:
Prof. Dr. Burkhard Stiller

15.8.2004

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

Over the last few years Peer-to-Peer (P2P) networks got very popular. Mainly file-sharing applications like KaZaA or E-Donkey made the breakthrough in the community. But there exists almost no commercial P2P due to the fact that well working pricing mechanisms on P2P networks hardly exist.

The goal of this thesis is to implement and evaluate PeerMart, the concept of a distributed market-based infrastructure for trading P2P services developed at TIK. PeerMart provides the necessary pricing mechanism and reliable price dissemination which allows also commercial Applications to enter the advantages of Peer-to-Peer network performance and robustness.

The solution of PeerMart bases on a Continuous Double Auction (CDA) infrastructure. In this Continuous Double Auction a cluster of peers, the so called BrokerPeers, is responsible for a given service. To map any service to the corresponding BrokerPeers and route requests from providers and consumers to the appropriate peer, a DHT-based infrastructure is used as underlying layer. To manage malicious peers and P2P-network changes, PeerMart uses redundancy in different variants.

This thesis presents refinements of the concept of PeerMart and mainly an implementation in Java together with FreePastry as the DHT service layer. Further this report contains an evaluation which compares theoretical efficiency and reliability values with results from this implementation.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Peer-to-Peer (P2P) Netzwerke sind dank populären Filesharing Systemen wie Gnutella, KaZaA, E-Donkey und E-Mule in der Öffentlichkeit seit ein paar Jahren weit verbreitet. Auch in der Forschung beschäftigt man sich intensiv mit P2P Systemen, da diese mittels Replikation und Aggregation von Ressourcen eine viel höhere Robustheit und Performance erreichen können als klassische Client/Server Applikationen (vgl. [1][3][5]). Ein wichtiger Aspekt bei P2P-Systemen ist, dass der Aufwand und Nutzen der angebotenen Services gleichmässig auf die verschiedenen Peers verteilt sein sollte. Um dies zu erreichen werden vermehrt Accounting Mechanismen eingesetzt, welche dafür sorgen, dass Peers nur soviel an Services von anderen Peers nutzen können, wie sie selber auch anbieten. Für Beispiele zu solchen Accounting Mechanismen betrachte man BitTorrent's Tit-for-Tat [12] oder Karma [10] oder [4]. Diese Accounting Mechanismen funktioniert heute immer noch nicht ganz wunschgemäss und lassen vor allem nur den Austausch von gleichwertigen Services zu. So ist Beispielweise das Phänomen des free-riding (z.B. user in Filsharing Systemen die nur konsumieren aber selbst keinen Content zur Verfügung stellen) ein bekanntes und ungelöstes Problem. Diese schlecht funktionierenden Accounting Mechanismen haben zur Folge, dass die Performance der P2P Systeme häufig weit unter dem Maximum liegt und dass professionelle und kommerzielle Angebote bis jetzt weitgehend inexistent sind.

Ziel dieser Semesterarbeit war es PeerMart [8], ein am TIK entwickeltes Konzept für einen Pricing Mechanismus für P2P Systeme, zu implementieren und anhand von Simulationen zu evaluieren. Wo nötig waren entsprechende Verfeinerungen und Anpassungen am Konzept und Design anzubringen. PeerMart ermöglicht sowohl eine dynamische Preisfestlegung als auch eine effiziente und verlässliche Preissuche was viele der oben genannten Probleme löst. Dank eines solchen Pricing Mechanismus wird es nun auch möglich verschiedene Services mit unterschiedlichen Preisen über P2P Systeme effizient zu handeln.

Das Konzept von PeerMart basiert auf einer Continous Double Auction (CDA) bei welcher die Auktion für einen bestimmten Service auf verschiedene Peers verteilt wird. Die Kommunikation im Peer-to-Peer Netzwerk wird über eine Distributed Hash Table (DHT) geregelt.

Diese Semesterarbeit zeigt in den folgenden Kapiteln auf, wie das Konzept PeerMart mittels Java implementiert wurde. Dabei werden interessante Aspekte wie etwa das Threading, der zeitliche Ablauf einer Auktion oder wie eine konsistente Datenhaltung über das P2P-Netzwerk erreicht werden kann speziell genau behandelt.

Dieser Bericht ist wie folgt gegliedert: In Kapitel zwei wird genauer auf das Konzept von PeerMart eingegangen, in Kapitel drei wird die Implementation sehr detailliert erläutert, Kapitel vier befasst sich mit der Evaluation und schlussendlich werden in Kapitel fünf offene Punkte angesprochen.

Kapitel 2

Konzept von PeerMart

Wie schon im vorherigen Kapitel erwähnt, stellt PeerMart ein Konzept für einen Pricing Mechanismus dar, mit welchem sowohl dynamische Preisfestlegung als auch eine effiziente und verlässliche Preissuche möglich ist. Die Lösung basiert grundsätzlich auf einer Continuous Double Auction (CDA) Infrastruktur für P2P Content Services. Die CDA wird heute vor allem an der Börse im Wertschriftenhandel eingesetzt. Dabei können sowohl die Verkäufer wie auch die Käufer kontinuierlich ihre Angebote und Nachfragen platzieren, welche laufend mittels einer gewissen Business Strategie abgeglichen werden. Genauere Informationen zur CDA sind unter [2] und [6] zu finden.

Die Grundidee von PeerMart besteht nun darin, dass jeweils ein Cluster von Peers als Broker für einen bestimmten Service verantwortlich ist und die Preisgebote für diesen Service verwaltet und bei Bedarf diese an die Käufer/Verkäufer weiterleitet. Durch die Verteilung der Infrastruktur auf mehrere Peers und die Redundanz innerhalb eines Clusters wird trotz der grundsätzlichen Fehleranfälligkeit von P2P-Netzwerken eine hohe Verlässlichkeit und Vertrauenswürdigkeit erreicht.

Im Weiteren dieses Kapitels wird nun detaillierter auf die einzelnen Mechanismen und Konzepte von PeerMart eingegangen.

2.1 Ablauf einer Auktion

Prinzipiell können Käufer und Verkäufer (Im weiteren häufig Consumer und Provider genannt) entweder eine Preisanfrage stellen oder gleich ein Angebot platzieren. Von nun an wird bei Angeboten von Consumern (Käufern) von Bids, bei Angeboten von Providern (Verkäufern) von Asks gesprochen. Wenn eine Anfrage solche Bid oder Ask enthält, wird diese an die zuständigen Broker weitergeleitet. Ein solcher Broker ist Teil eines Cluster von Peers (dem so genannten BrokerSet), die für einen bestimmten Service zuständig sind. Diese Broker antworten auf die Anfrage in jedem Fall mit dem aktuellen höchsten Bid Preis oder dem tiefsten Ask Preis je nachdem ob die Anfrage ein Bid oder Ask ist. Bei einem Angebot wird zusätzlich mitgeteilt ob das Angebot in die Auktion aufgenommen wurde. Denn bei den Brokern wird nur eine beschränkte Anzahl der besten Bids und Asks in einer Tabelle gespeichert. Sobald die ersten Bids und Asks bei den Brokern eingetroffen sind, wird immer am Ende eines spezifischen, relativ kurzen Zeitintervalls versucht diese Angebote zu matchen, und falls es passende Paare von Bids und Asks gibt, werden diese aus der Tabelle

der Auktion entfernt und die dazugehörigen Provider und Consumer informiert, dass eine erfolgreiche Auktion zustande kam. Dieser Ablauf von ankommenden Bids und Asks und dem dazugehörenden Matching und dem Informieren der Consumer und Provider findet dann weiterhin in regelmässigen Abständen statt.

2.2 DHT als Hilfsschicht

PeerMart versteht sich selbst nicht als eine P2P Anwendung, sonder agiert mehr als eine Middleware für P2P Programme. Als Serviceschicht bietet PeerMart ein paar wenige Methoden an, über welche es mit der wirklichen P2P-Anwendung kommuniziert. Mehr zum API von PeerMart ist im Kapitel 3 zu erfahren. PeerMart selbst baut aber nicht direkt auf einem zufälligen Peer-to-Peer Netzwerk auf sondern stützt sich selbst auf einen Distributed Hash Table (DHT) Layer. Abbildung 2.1 zeigt schematisch mit welchen Layers PeerMart kommuniziert. Dieser DHT-Layer ist zuständig für das Versenden, Routen und Empfangen von allen Nachrichten zwischen Brokern, Consumern und Providern. Auch ist die DHT-Schicht dafür verantwortlich, die verschiedenen Services den Brokern zuzuordnen und die dynamischen Änderungen im P2P-Netzwerk zu verwalten. Eigentlich wird ein Key Based Routing (KBR) Mechanismus verwendet. Das KBR erlaubt es dann eine DHT aufzubauen, weshalb diese beiden Begriffe KBR und DHT häufig equivalent verwendet werden.

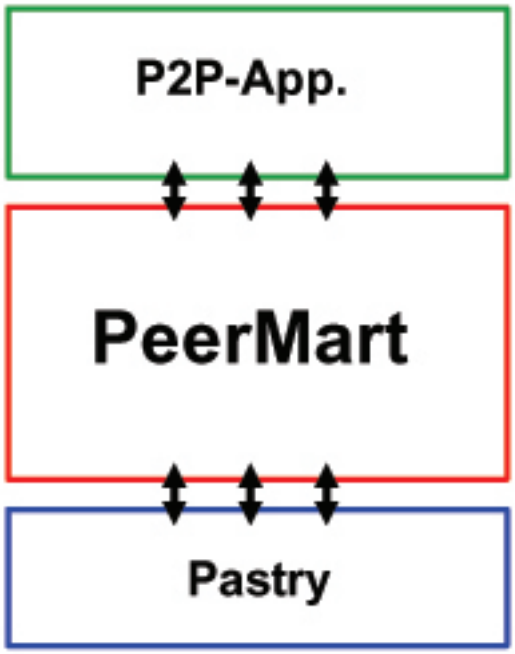

Abbildung 2.1: Layers von PeerMart

Das Konzept von PeerMart sieht zudem vor, dass eine PKI vorhanden ist. Das heisst, dass jeder Peer über ein public/private key pair verfügen sollte, um die Nachrichten, welche über die DHT-Schicht versandt werden, signieren zu können. Eine

mögliche Alternative zur PKI wird in [11] aufgezeigt. Bei dieser Alternative wird anhand eines Rätsels, welches nur der richtige Node einfach lösen kann, ein key generiert.

2.3 System Design

Damit PeerMart funktioniert, muss jeder gehandelte Service global eindeutig über eine ServiceId identifiziert werden können. Diese ServiceId muss allen Peers die über PeerMart handeln bekannt sein. Bei einem Filesharing Service kann die ServiceId eines Files beispielsweise der Hashwert des Files selbst sein. Oder bei einem Wertschriftenhandel die eindeutige Kennung des Wertpapiers. Des weiteren sollte die Währung für den Service bekannt sein.

Diese eindeutige ServiceId wird in PeerMart auf den Adressbereich der darunter liegenden DHT-Schicht abgebildet. Über die DHT-Schicht kann dann auf einfache Weise das BrokerSet für diesen Service bestimmt werden. Das BrokerSet beinhaltet die n Peers dessen Id's im DHT Adressraum den n/2 numerisch nächst grösseren und n/2 numerisch nächst kleineren Adressen zu der ServiceId Adresse entsprechen. Diese n BrokerPeers eines BrokerSets sind von nun an zuständig, die Auktion für den dazugehörigen Service zu verwalten. Da die Id's in Distributed Hashtables über gleichmässig verteilte Hashfunktionen berechnet werden, kann davon ausgegangen werden, dass jeder Peer für ungefähr die gleiche Anzahl von Auktionen verantwortlich ist, die Rechenlast also gleichmässig auf alle Peers verteilt wird. Sobald neue Peers in das P2P System eintreten oder Peers das System verlassen, müssen natürlich die BrokerSets entsprechend angepasst werden.

Wie im vorherigen Abschnitt erwähnt wurde, führt nun jeder BrokerPeer für jeden Service, für den er zuständig ist, eine Tabelle der Grösse m mit, in der die m/2 besten Bids und die m/2 besten Asks gespeichert sind. Dabei ist zu beachten, dass jeder Bid und jeder Ask eine Sequenznummer und eine Gültigkeitsdauer hat. Für jeden Consumer und Provider wird immer nur der aktuellste Bid und Ask in der Tabelle gespeichert und verarbeitet. Sobald ein Bid oder Ask seine Gültigkeitsdauer überschreitet wird er kommentarlos aus der Tabelle gelöscht. Wie genau die Kommunikation zwischen den verschiedenen BrokerPeers eines BrokerSets abläuft ist im Abschnitt 2.5 beschrieben.

2.4 Zuverlässigkeit

Ein wichtiger Aspekt von PeerMart ist, dass trotz möglichen böswilligen Peers und häufigen Leaves und Joins im P2P Netzwerk die Zuverlässigkeit der Auktion gewährleistet bleibt.

Das böswilligste Verhalten, dass ein Peer in PeerMart zeigen kann ist, dass er Bids und Asks nicht weiterleitet oder verarbeitet. Sobald ein Peer auf der Pastry-Ebene nicht mehr antwortet, wird er von der DHT automatisch aus dem P2P Netzwerk ausgeschlossen. Um nun Nodes, die auf der PeerMart-Ebene böswilliges Verhalten zeigen, umgehen zu können, arbeitet PeerMart mit Redundanz. Jedes Angebot wird f-fach parallel an f zufällig ausgewählte Peers aus dem BrokerSet gesendet. Wobei f logischerweise nicht grösser als die BrokerSetgrösse sein kann. Aus den f Antworten, welche im besten Fall zurückkommen sollten, wird dann wiederum ein Mehrheitsentscheid gebildet, damit mögliche falsche Antworten erkannt und ausge-

schlossen werden können. Auch die Auktion selbst wird auf allen BrokerPeers des BrokerSets separat durchgeführt, womit auch dort genug Redundanz gegeben ist, um böswillige Peers, die das Matching nicht korrekt ausführen, auszuschalten.

Falls jemand versuchen sollte mittels IP-Spoofing eine falsche IP vorzutäuschen wird dies von der DHT unterbunden, weil Pastry selbst bemerkt, dass unter der angegeben IP-Adresse kein Pastry Client läuft und somit keine Antwort bekommt.

Peers die laufend Bids und Asks abesenden, diese aber nicht einhalten, wenn es zu einem Match kommt, können von PeerMart nicht direkt erkannt und ausgeschlossen werden. Falls solches vorsätzliches, böswilliges Handeln verhindert werden soll, können Reputationssysteme eingesetzt werden. Ein solches Reputationssystem würde die Reputation eines Peers bei jedem Nichteinhalten hinabsetzten bis die Reputation dieses Peers so tief wäre, dass von diesem Peer gar keine Bids und Asks mehr akzeptiert würden. Als mögliche Reputationssysteme seien hier EigenRep [13] und NICE [14] erwähnt.

2.5 Zeitaspekte und Abwicklung der Auktion

Eines der grössten Probleme bei verteilten Systemen ist eine synchrone Zeit auf allen Peers zu erlangen. So ist auch die synchrone Abwicklung der zeitkritischen Auktionen nicht trivial. Die Auktion ist deshalb zeitkritisch, weil innerhalb einer gewissen Zeit alle Peers für eine bestimmte Auktion die gleichen Aktionen ausführen müssen. Werden diese Aktionen nicht gleichzeitig ausgeführt, hat das inkonsistente Transaktionen zur Folge. Da die Auktion parallel auf jedem BrokerPeer des BrokerSets ausgeführt wird, die Peers ihre Asks und Bids aber nur an f zufällige Peers schicken, müssen die Brokerpeers dafür sorgen, dass jeder BrokerPeer des BrokerSets alle notwendigen Informationen bekommt. Das wiederum bedingt, dass die Auktionsphase in zwei Zeitslots unterteilt wird. Nämlich einen Zeitslot, in dem die neuen Bids und Asks angenommen werden, und einen zweiten Zeitslot, in dem dann die Matches wirklich berechnet werden. So können die Peers am Ende des ersten Zeitslots die relevanten Daten untereinander synchronisieren, sodass im zweiten Zeitabschnitt alle Brokerpeers die gleichen Informationen haben. Am Ende des ersten Zeitslots werden jeweils die provisorischen, am Ende des zweiten Zeitslots die definitiven Matches berechnet. Damit garantiert werden kann, dass am Ende eines Zeitfensters wirklich alle Peers die gleichen Daten haben, muss die Länge eines Zeitfensters mindestens die maximale Roundtripzeit (RTT), die in dem Peer-to-Peer System möglich ist, aufweisen. Sinnvollerweise besitzen die beiden Zeitslots die gleiche Länge.

Jeder BrokerPeer muss sich beim Starten einer Auktion für einen spezifischen Service die Startzeit merken, anhand dessen jeder BrokerPeer jederzeit berechnen kann, in welchem der beiden Zeitslots er sich gerade befindet.

Trifft nun ein Bid oder Ask bei einem BrokerPeer ein, schaut dieser zuerst ob er sich gerade in Zeitslot eins (der Annahmephase) befindet. Falls ja, versucht er den Bid oder Ask in seine Tabelle einzufügen (wie schon gezeigt wurde, wird dies nur geschehen, wenn der eintreffende Bid oder Ask unter die lokal m/2 besten fällt). Falls sich der Peer nicht in der Annahmephase befindet, wird der eintreffende Bid oder Ask in eine lokale Queue gelegt, die erst in der nächsten Annahmephase abgearbeitet wird. Am Ende jeder Annahmephase werden nun die temporären Matches berechnet (dies ist jedoch nur notwendig, wenn im aktuellen Zeitslot neue Bids und Asks eingetroffen sind, weil es ja sonst gar keine Neuerungen und somit auch keine neue Matches geben kann). Diese höchstens m/2 Matches werden dann an alle üb-

rigen BrokerPeers des BrokerSets weitergeleitet. Dabei gibt es noch einen Spezialfall zu betrachten: Falls sich der BrokerPeer das erste Mal am Ende der Annahmephase befindet oder beim letzten definitiven Berechnen Matches gebildet werden konnten und es bei den aktuellen temporären Matches keinen Match gibt, so muss auch der beste Bid und der beste Ask mitgeschickt werden. Dies ist notwendig, weil sonst die Situation auftreten könnte, das matchende Bids und Asks auf verschiedenen BrokerPeers verteilt sind, diese aber nie davon erfahren. Eventuell könnten auch die zwei oder drei besten Bids/Asks weitergeschickt werden. Das ist ein Designparameter, welcher noch genau getunt werden muss.

Sobald nun diese temporären Matches an alle übrigen BrokerPeers versendet wurden, wechselt der Peer in den zweiten Zeitslot, die Synchronisationsphase. Ankommende neue Bids oder Asks werden während dieser Phase, wie schon erwähnt, in die Queue gespeichert. Dafür werden ankommende Synchronisationsnachrichten von den anderen BrokerPeers in die Tabelle eingefügt (natürlich wieder nur, wenn sie zu den m/2 besten gehören). Am Ende dieser Synchronisationsphase besitzen nun alle Brokerpeers die notwenigen Daten, die für das definitive Berechnen der Matches notwenig sind. Diese Berechnung wird dann auch lokal von jedem Peer am Ende der Synchronisationsphase durchgeführt. Wie schon das Berechnen der temporären Matches ist das Berechnen der definitiven Matches nur dann notwendig, wenn entweder während der Annahmephase oder während der Synchronisationsphase neue Bids und/oder Asks eingetroffen sind.

Falls sich nun bei diesem definitiven Berechnen definitive Matches ergeben, wird von jedem Brokerpeer einzeln überprüft, ob entweder der Bid oder der Ask eines Matches ursprünglich von ihm selbst empfangen wurde. Falls dies der Fall ist, werden der Consumer und/oder der Provider von diesem Brokerpeer über den erfolgreichen Match informiert. Schlussendlich werden noch alle Matches aus der Tabelle gelöscht und die nächste Annahmephase kann beginnen. Wie dieses Konzept genau implementiert ist, wird ausführlich im nächsten Kapitel beschrieben.

Abbildung 2.2 fasst den ganzen Ablauf schematisch noch einmal zusammen.

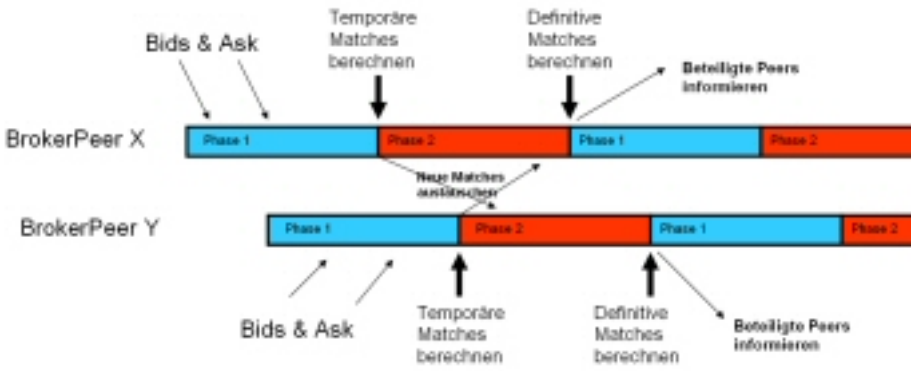

Abbildung 2.2: Zeitliche Abwicklung der Auktion

Kapitel 3

Implementation

Der Hauptfokus dieser Semesterarbeit liegt, wie bereits in der Einleitung erwähnt, in der Implementierung von PeerMart, welche in diesem Kapitel ausführlich behandelt wird. In diesem Kapitel werden alle wichtigen Aspekte und Details der Implementierung von PeerMart detailliert erläutert. Für eine noch genauere Dokumentation sei das direkte Betrachten des relativ gut dokumentierten Quellcode empfohlen.

3.1 Wahl der Programmiersprache

Einer der ersten und zentralsten Punkte war die Wahl der Programmiersprache. Dadurch, dass Pastry als DHT-Layer gewählt wurde, welches in Java geschrieben ist und der Autor dieser Arbeit Java am besten beherrscht, fiel die Wahl relativ eindeutig auf Java. Java hat zudem den grossen Vorteil der Platformunabhängigkeit. Auch ist Java sowohl in der akademischen wie auch in der industriellen Welt weit verbreitet, so dass eine mögliche spätere Erweiterung von praktisch jedermann vorgenommen werden kann.

3.2 Wahl der DHT

Der nächste wichtige Schritt war die Auswahl des Kommunikationslayers, über welchen die einzelnen PeerMart Nodes miteinander kommunizieren. Damit nicht die ganze Kommunikation von Grund auf neu programmiert werden musste, war klar, dass dazu ein frei verfügbares Framework benutzt werden soll. Zur Auswahl standen dabei grundsätzlich allgemeine Peer-to-Peer Frameworks, wie etwa das von Sun geförderte JXTA [15] oder eine Implementation einer Distributed Hash Table. Da in PeerMart vor allem mit Servicelds gearbeitet wird, welche gewissen Brokern, also Nodeld's, zugeordnet werden müssen, fiel die Wahl ziemlich eindeutig auf die DHT, weil diese genau den Zuordnungsmechanismus als ihre Hauptaufgabe zur Verfügung stellt.

Bei der Auswahl einer bestimmten DHT fiel die Entscheidung relativ schnell zugunsten von Pastry /cite9 aus. Zum einen existiert mit FreePastry eine ausgereifte, frei verfügbare Java Implementation von Pastry, zum anderen ist die Besonderheit von Pastry gegenüber anderen DHTs wie beispielsweise Chord, der spezielle Aufbau der Routingtable. Pastry generiert nämlich ein so genanntes Leafset, welches sehr

nützlich ist um das BrokerSet zu erstellen. Aber mehr dazu im folgenden Abschnitt über Pastry.

3.3 Pastry

3.3.1 Was ist Pastry

Pastry [9] ist eine Distributed Hash Table, welche einen zirkulären Adressraum von 0 bis 2^{128} aufspannt. Dabei bekommt jeder Pastrynode eine 128-bit Nodeld zugewiesen. Diese Nodeld wird durch Anwendung einer Hashfunktion auf die IP-Adresse oder den public-key generiert, so dass die Nodeld's gleichmässig über den ganzen Adressraum verteilt sind. Pastry braucht im Durchschnitt nur $\log_{2^4} N$ Schritte um eine Message an einen bestimmten Knoten weiterzuleiten. Sobald die IP Adresse eines Nodes einmal bekannt ist (Pastry einen so genannten NodeHandle besitzt) wird die Nachricht sogar direkt zum Knoten weitergeleitet ohne die Routinginformationen von anderen Pastrynodes in Anspruch zu nehmen. Pastry passt sich völlig automatisch dem sich laufend verändernden Peer-to-Peer-Netzwerk an. Eine Spezialität von Pastry gegenüber anderen DHTs ist, dass jeder PastryNode neben der bei DHTs üblichen Routingtabelle eine zweite kleine Tabelle, das so genannte Leafset, enthält. Das Leafset ist nichts anderes als die zur eigenen Nodeld b/2 numerisch nächst grösseren und die b/2 numerisch nächst kleineren Nodeld's. Bei Pastry ist b standartmässig auf 24 codiert.

3.3.2 API von Pastry

Seit dem Release 1.3 von FreePastry sollte das Package rice.p2p.commonapi als Schnittstelle zu Pastry verwendet werden. In diesem Package ist vor allem das Interface *Application* elementar. Jede Applikation welche Pastry benutzen will muss dieses Interface implementieren. Das Interface rice.p2p.commonapi.Application definiert folgende 3 Methoden:

```
void deliver(Id id, Message message)

boolean forward(RouteMessage message)

void update(NodeHandle handle, boolean joined)
```

Die deliver Methode wird immer dann von Pastry aufgerufen, wenn der lokale Pastrynode eine Message erhalten hat, dessen Id auf dem Pastryadressraum numerisch am nächsten bei der Nodeld dieses Pastrynodes liegt. Alle eingehenden Nachrichten (über Pastry können prinzipiell nur Nachrichten (Messages) versandt werden) gelangen also über diese Methode zu PeerMart.

Die forward() Methode wird von Pastry immer dann aufgerufen, wenn der PastryNode eine Pastrynachricht weiterleiten möchte. Die Applikation hat dabei die Möglichkeit die Nachricht zu verändern oder auch eine neues Ziel festzulegen. Über den booleschen Rückgabewert kann die Applikation sogar bestimmen ob die Nachricht überhaupt weitergeleitet werden soll. PeerMart selbst benutzt diese Möglichkeiten nicht. Einzig wenn man böswillige Peers simulieren möchte, kann man über diesen Rückgabewert bewirken, dass dieser Peer kein Nachrichten mehr weiterleitet.

Die update Methode wird von Pastry aufgerufen, wenn ein Node mit dem Nodehandle *handle* das Leafet des eigenen Pastrynodes betreten oder verlassen hat (je nachdem ob joined auf true oder false ist). Das heisst, diese Methode wird immer dann aufgerufen wenn es Änderungen im P2P Netzwerk in der virtuellen Nachbarschaft gegeben hat. Peers sind virtuelle Nachbarn, falls ihre NodeIds numerisch nebeneinanderliegen. Für PeerMart ist diese Methode sehr wichtig, denn bei jeder Änderung in der Nachbarschaft müssen tendenziell ein oder mehrere BrokerSets angepasst werden.

Damit eine Instanz wirklich auf die Dienste von Pastry zugreifen kann, muss für jeden Peer mittels der richtigen NodeFactory ein PastryNode erzeugt werden. Für PeerMart ist dies eine DistPastryNodeFactory. Alternativ bietet Pastry eine DirectPastryNodeFactory an, welche Nodes produziert, die nur Verbindungen zwischen Nodes, die von der gleichen Factory erzeugt wuerden, zulässt. Diese DirectPastryNodeFactory ist für PeerMart also völlig unbrauchbar. Dieser Factory kann optional ein BootstrapNode übergeben werden, um einem bestehenden Pastry Netzwerk beizutreten, ein Port, auf welchem der Node laufen soll, sowie das gewünschte Protokoll. Danach wird mit Hilfe dieser Factory der eigentlichen PastryNode erzeugt. Nachdem dann bei dieser PastryNode die Applikation, in diesem Fall also die PeerMartApp, registriert wurde, ist der Initialisierungsprozess von Pastry abgeschlossen und es können fortan Messages versandt und empfangen werden. Bei der Registrierung der Applikation wird von Pastry zusätzlich zum PastryNode ein Endpoint zurückgeliefert, über welchen mittels der route() Methode Nachrichten an die übrigen PastryNodes verschicken werden kännen. Falls Pastry bei der angegebenen BootstrapNode-Adresse keinen PastryNode findet, wird eine neuer Pastry Ring erzeugt.

3.3.3 Verwendung von Pastry in PeerMart

Pastry wird in PeerMart für drei verschiedene Aspekte verwendet:

1. Die ganze Kommunikation zwischen den verschiedenen Peers Läuft über Pastry. Das heisst einerseits die Kommunikation zwischen Customers/Providern und Brokerpeers als auch die direkte Kommunikation zwischen den verschiedenen Brokern eines BrokerSets. Das Versenden von Nachrichten gestaltet sich sehr einfach über die Methode

   ```
   void route(Id id, Message message, NodeHandle hint)
   ```

   Dabei ist die *id* die NodeId des Peers, zu welchem die Nachricht geschickt werden soll. *Message* ist die Nachricht, die verschickt wird und der *hint* ist ein optionaler Hinweis falls man schon einen NodeHandle auf den Node besitzt, an den die Nachricht gerichtet ist. Falls man schon einen solchen NodeHandle angeben kann, wird eine direkte IP Verbindung aufgenommen.

2. Zudem wird Pastry benutzt, um einer bestimmten ServiceId das richtige BrokerSet zuzuordnen. Die Methode dazu ist, dass eine ServiceId genau gleich behandelt wird wie eine NodeId. Wenn also ein Peer das erste Mal den Service X beanspruchen will, sendet er einfach eine Nachricht an die ServiceID von Service X. Pastry routet diese Nachricht dann automatisch zum numerisch zu dieser ServiceID nächstgelegenen PastryNode, dem so genannten RootBrokerPeer für diesen Service. Falls dieser RootBrokerPeer das erste Mal mit dem

Service X konfrontiert wird, testet der Node selbst noch einmal, ob er wirklich der nächstgelegene Node zu diesem Service ist (auch wieder über das eigene Leafset), falls ja, übernimmt er die Rolle des BrokerPeers für diesen Service. Als RootBrokerPeer hat dieser PastryNode zusätzlich die Aufgabe allen übrigen BrokerPeers des betreffenden BrokerSets zu informieren, dass sie nun BrokerPeer für diesen Service werden müssen. Dies ist nun aber wiederum ganz einfach, da das BrokerSet einer Teilmenge des Leafsets entspricht. (je nachdem wie die Grösse des BrokerSets gewählt ist). Der RootBrokerPeer sendet dazu allen zukünftigen BrokerPeers das Brokerset und eine Aufforderung BrokerPeer zu werden.

Falls der RootBroker den Service X jedoch schon kennt macht er nichts anderes als dem anfragenden Peer das BrokerSet zurückzusenden.

3. Es wird Pastry überlassen, das sich dynamisch ändernde P2P Netzwerk zu kontrollieren. Jedes Mal wenn es in der virtuellen Nachbarschaft des eigenen Peers eine Veränderung gibt, wird dieser Event über die schon erwähnte Methode update mitgeteilt. Bei jedem Hinzukommen eines neuen Peers muss jeder entsprechende Knoten für sich folgende Punkte überprüfen: Falls die Id des neuen Peers näher bei einer ServiceId liegt, für welchen der prüfende Knoten der RootBrokerPeer ist, muss dem neuen Peer mitgeteilt werden, dass er zum RootBrokerPeer für diesen Service wird. Die Tatsache, ob ein Peer RootBrokerPeer für einen Service ist, kann überprüft werden, da in der PeerMartApp Klasse eine Hashtable mitgeführt wird, welche RootNodeId's zu ServiceId's abbildet. Zudem muss der Peer für alle Services, für welche er der RootBrokerPeer ist, die aktualisierten Brokersets an alle BrokerPeers versenden. Bei jedem Verlassen eines Peers muss jeder entsprechende Knoten für sich selbst überprüfen, ob er selbst nun der numerisch nächste Node für einen Service ist, für welchen der verlassende Peer der RootBrokerPeer war. Falls dies zutrifft muss der Knoten zum neuen RootBrokerPeer werden. Diese Überprüfungen werden durch die beiden Methoden

   ```
   private void handleLeavedNode(NodeHandle handle)

   private void handleJoinedNode(NodeHandle handle)
   ```

   vorgenommen. Die Methoden werden in der update Methode aufgerufen, welche wiederum von Pastry aufgerufen wird, sobald es eine Änderung im Leafset gibt.

3.4 API von PeerMart

In diesem Abschnitt werden die Methoden von PeerMart vorgestellt über welche eine Peer-to-Peer Anwendung den Pricing Mechnismus von PeerMart verwenden kann. Folgende Methoden sind für die Interaktion mit der P2P-Applikationsschicht gedacht:

```
public void getAsk(ServiceID)

public void getBid(ServiceID)
```

Diese beiden Methoden werden gebraucht um den aktuell tiefsten Ask oder den höchsten Bid Preis bei den, für den Service verantwortlichen, BrokerPeers zu erfragen. Als Parameter muss die Serviceld des gewünschten Services angegeben werden.

```
public void notifyAsk(ServiceID, Price)
```

```
public void notifyBid(ServiceID, Price)
```

Diese beiden Methoden bilden die Antwortmethoden zu den beiden vorhergehenden Anfragemethoden. Die getAsk und getBid Methoden haben selbst keinen direkten Rückgabewert, weil diese sonst blockierend wären und somit der ganzen Programmfluss von PeerMart unterbrochen würde, bis die Antwort für die Anfrage eingetroffen wäre, was natürlich in einem verteilten System nicht sinnvoll ist. Wichtig ist, dass notifyAsk *Integer.MAX_VALUE* und notifyBid 0 zurückliefert, falls lokal gar kein Ask oder Bid in der Servicetabelle vorhanden ist.

```
public void sendBid(ServiceID, Price, long)
```

```
public void sendOffer(ServiceID, Price, long)
```

Die beiden send Methoden werden benötigt um den Providern das Senden eines Bids und den Customers das eines eines Asks zu ermöglichen. Dabei müssen als Parameter die Serviceld des Services, der Preis, den man bereit ist zu zahlen (den man verlangt) und die Zeitdauer (in Millisekunden) während der das Angebot gelten soll übergeben werden.

```
public void notifySendBid (ServiceID, Price, Price, boolean)
```

```
public void notifySendOffer(ServiceID, Price, Price, boolean)
```

Die beiden notify Methoden teilen der P2P Applikation mit, dass der Bid oder der Ask erfolgreich an die betreffende Auktion weitergeleitet wurde. Dabei werden die beiden Methoden mit folgenden vier Parametern aufgerufen: Die Serviceld gibt an, um welchen Service es sich handelt, der erste Preis entspricht dem Preis der von diesem Peer selbst geboten (offeriert) wurde, der zweite Preis entspricht dem zurzeit höchsten Bid (tiefsten Ask) der momentan gehandelt wird. Der letzte Parameter schlussendlich gibt an, ob der Bid (Ask) überhaupt in die Servicetabelle der Auktion aufgenommen wurde (also zu den aktuellen m/2 besten gehört). Auch hier sind die Methoden in eine send und eine notify Methode aufgeteilt, um den Programmfluss vollkommen asynchron und nicht blockierend gestalten zu können.

```
public void notifySuccessfulOffer(ServiceID, Price, Id, Price)
```

```
public void notifySucessfulBid(ServiceID, Price, Id, Price)
```

Die beiden letzten Methoden werden von PeerMart benutzt um der darüber liegenden Applikation das Zustandekommen eines Matches mitzuteilen. Dabei wird die Methode mit den vier folgenden Parametern aufgerufen: Der Serviceld des Services, dem Preis der dieser Peer bezahlen (einfordern) wird, die Id des Partners, mit welchem das Geschäft abgewickelt wird und den Preis welchen dieser Partner bezahlen (einfordern) wird.

Damit PeerMart mittels den notify Methoden mit der darüber liegenden P2P Applikation kommunizieren kann, muss entweder in jeder der 6 notify Methoden einzeln ein direkter Methodenaufruf auf eine Methode der Applikation implementiert

werden, oder es wird ähnlich wie bei Pastry ein Application Interface generiert, welches alle 6 notify Methoden beinhaltet und natürlich von der Anwendung dann auch implementiert werden muss. In dieser Arbeit gab es noch keine wirkliche Applikation, die PeerMart benutzte, deshalb wurde der erste Ansatz gewählt. In jeder notify Methode wurden einfach *System.out.println()* eingebaut um eine Rückmeldung zu bekommen.

Abbildung 3.1 zeigt zwei übliche Abläufe von PeerMart-Methoden Aufrufen. Durchgezogene Pfeile bedeuten, dass diese Aufrufe immer stattfinden, gestrichelter Pfeil bedeutet, dass dieser Aufruf nur unter bestimmten Umständen (z.B. ein erfolgreicher Match) getätigt wird.

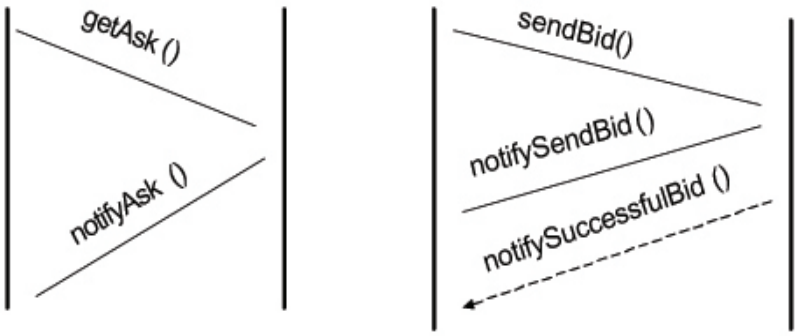

Abbildung 3.1: Aufruf-Ablauf der PeerMart-Methoden

3.5 Aufteilung von PeerMart in Packages

In diesem Kapitel wird die Aufteilung von PeerMart in die einzelnen Packages erläutert. Zu den meisten Klassen wird auch gleich dessen Funktionsweise und Notwendigkeit für PeerMart aufgezeigt. Weil das Threading in PeerMart einen sehr hohen Stellenwert hat, werden die Klassen dieses Packages im eigenen Abschnitt 3.6 Threading erläutert.

3.5.1 Package main

Im main Package befinden sich alle wichtigen Hauptklassen von PeerMart.

PeerMartApp

Die Klasse PeerMartApp ist ohne Zweifel die Hauptklasse von PeerMart. Dort werden alle Aktionen koordiniert. Die Klasse PeerMartApp ist auch diejenige, die das Interface *Application* (vgl. 3.3.2) von Pastry implementiert. Auch alle Methoden die für die Interaktion mit der darüber liegenden P2P Applikation sorgen (vgl. 3.4) befinden sich in dieser Klasse. Nicht zuletzt besitzt diese Klasse den eigentlichen PastryNode und den Endpoint, welcher zum Versenden von Nachrichten gebraucht wird. Neben einer grösseren Anzahl kleiner Hilfsmethoden wie etwa sendBrokerSetToAllBrokers(ServiceID, BrokerSet), um allen Brokern eines BrokerSet das Broker-

Set zu senden oder die elementare Methode sendMessage(Id, Message, NodeHandle) um Nachrichten zu versenden werden auch verschiedenste Listen mitgeführt, die zum Verwalten der Auktionen gebraucht werden. So etwa alle bekannten ServiceIds (also Services zu denen der Peer das BrokerSet kennt), alle eigenen Auctions (also Services für die der Peer selbst ein BrokerPeer ist) und nicht zuletzt eine Liste aller Rootauctions (also Auktionen für die der Peer der RootBrokerPeer ist). Zuletzt wird noch eine Liste von Threads geführt, die alle auf eine Antwort von einem anderen Peer warten.

Die Hauptaufgabe von PeerMartApp ist es, einerseits die Messages, die über Pastry ankommen, zu empfangen, zu dispatchen und je nach Art der Nachricht diese der richtigen, für die Weiterverarbeitung zuständigen Methode zuzuführen. Wobei schlussendlich fast immer ein Thread die Nachricht in Empfang nimmt und weiterverarbeitet. Wie sich genau der Verlauf einer solchen Transaktion abspielt wird im Kapitel 3.6.1 wo es um das Threading geht, genauer erklärt. Andererseits nimmt diese Klasse auch die Anfragen der oberhalb liegenden Applikationsschicht entgegen und startet die notwendigen Transaktionen, um diese Anfrage abwickeln zu können.

PeerMartPastryNodeCreator

Die Klasse PeerMartPastryNodeCreator wird gebraucht um den eigentlichen PastryNode zu erzeugen. Um einen PastryNode erzeugen zu können muss vor allem die richtige NodeFactory mit den spezifischen Parametern für den Port und das Protokoll erzeugt werden. Zudem kann ein BootstrapNode angegeben werden um einem bestehenden Pastrynetzwerk beizutreten. Die Klasse besitzt deshalb vor allem viele verschiedene Konstruktoren, um verschiedenste Nodes erzeugen zu können, und eine wichtige Methode:

```
public PastryNode createNode()
```

welche beim Ausführen den eigentlichen Node zurückliefert. PeerMart benutzt ausschliesslich folgende zwei Konstruktoren:

```
public PeerMartPastryNodeCreator()

public PeerMartPastryNodeCreator(int port, int bootstrapPort,
String bootstrapHost)
```

Ersterer wird benutzt um einen Node in einem neuen Pastry-Ring zu erzeugen. Dafür wird eine DistPastryNodeFactory mit dem Protokoll WIRE (alternativ wäre RMI) auf Port 5009 erzeugt. Beim Protokoll WIRE findet die Kommunikation über TCP/IP statt, bei RMI über Remote Message Invocation (RMI). Der zweite Konstruktor erzeugt einen PastryNode der den PastryRing des ersten Nodes beitritt. Dabei muss geschaut werden, dass als Parameter *port* ein freier Port übergeben wird und als *bootstrapPort* den Port 5009 und als *bootstrapHost* die IP-Adresse der Maschine auf welcher der Root-Node gestartet wurde.

Auction

Die Klasse Auction repräsentiert die Auktion für eine einzelne Serviceld. Die Auction Klasse wird dann erzeugt, wenn der dazugehörige Service das erste Mal gehandelt wird. Jede Auction Klasse kennt ihre Serviceld, ihr BrokerSet und die Auktionsstartzeit, anhand derer jederzeit berechnet werden kann, in welchem Zeitfenster die

Auktion sie sich gerade befindet (vgl. Kapitel 2.5). Jede Auction Klasse besitzt zudem eine Servicetabelle (siehe auch Kapitel 3.5.1) und eine Queue für Bids und Asks, die im falschen Zeitfenster eintrafen.

Falls über Pastry ein Bid, Ask oder eine AuctionMessage eintrifft, wird die entsprechende Einfügemethode (putBid, putAsk oder handleAuctionMessage) auf der Auction Klasse ausgeführt. Die Auction Klasse überprüft dann, ob sie sich im richtigen Zeitslot für das Einfügen befindet. Falls dies zutrifft wird versucht die Bids und Asks in die Servicetabelle einzufügen. Falls sich die Auction Klasse gerade im falschen Zeitslot befinden sollte, werden die Bids und Asks in die Queue gelegt oder im Falle von AuctionMessages weggeworfen. Denn AuctionMessages die im falschen Zeitfenster eintreffen sollten nicht vorkommen.(Sollten AuctionMessages im falschen Zeitfenster eintreffen ist die Länge der Zeitslots zu klein gewählt). Falls dies aber trotztdem einmal eintreffen sollte, ist es zu spät und ein Einfügen könnte die Datenkonsistenz verletzen. Bei ankommenden Bids und Asks werden auch gleich die Antwortnachrichten verfasst und versandt. Falls nun entweder Bids oder Asks erfolgreich in die Tabelle eingefügt wurden, muss die Auction Klasse zudem das provisorische Berechnen der Matches vorbereiten. Wie das genau geschieht, ist in Kapitel 3.6.2 genau erklärt. Falls es beim profisorischen Berechnen Matches gab, ist es die Aufgabe der Auction Klasse diese allen anderen BrokerPeers mitzuteilen. Und falls es Matches gab oder während der Synchronisationsphase AuctionMessages eintrafen muss auch ein definitives Berechnen für diese Auktion geplant werden.

Um Bids und Asks in die Queue zu legen gibt es noch eine innere Klasse QueueElement. Dieses QueueElemente sind nötig, weil zum Ablegen und Weiterverarbeiten der Bids und Asks Objekte erzeugt werden müssen, es aber nicht sinvoll ist gleich die ganze Message zu speichern.

BrokerSet

Das BrokerSet enthält, wie schon einige Male erwähnt, alle BrokerPeers, die für einen bestimmten Service zuständig sind. Die Klasse BrokerSet ist prinzipiell nichts anderes als ein Wrapper um die Leafsetklasse von Pastry und enthält für diesen Zweck je ein Feld für die Serviceld und ein Feld für das Leafset. Damit man BrokerSets über Pastry verschicken kann, muss das MarkerInterface *Serializable* implementiert werden. Das BrokerSet bietet folgende drei wichtige Methoden an:

```
public NodeSet getBrokerNodesWithRoot()

public NodeSet getBrokerNodesWithOutRoot()

public NodeSet getRandomBrokers(int nrOfRandomBroker)
```

Die erste Methode wird von der Auctions Klasse benutzt um temporäre Matches an alle BrokerPeers zu versenden. Die zweite Methode wird von den RootBrokerPeers benutzt, um allen übrigen BrokerPeers das BrokerSet mitzuteilen und die letzte Methode wird von den Customers und Providern benutzt um ihre Anfragen an f *nrOfRandomBroker* verschiedene BrokerPeers zu senden.

ServiceOffer

Die Klasse ServiceOffer repräsentiert die Form wie die Bids und Asks in der Servicetabelle gespeichert werden. Denn neben dem simplen Preis muss zusätzlich noch

gespeichert werden, von wem der Bid/Ask kommt, ob dieser BrokerPeer das Angebot selbst empfangen hat (oder ob über ein AuctionMessage kommuniziert wurde) und auch die Gültigkeitsdauer für das Angebot muss gespeichert sein. Die ServiceOffer Klasse ist eine Klasse die nur aus Feldern und ihren dazugehörenden getter und setter Methoden besteht.

AuctionTable

Die Klasse AuctionTable repräsentiert die Servicetabelle einer Auktion. Ihr Hauptaufgabe ist es die m/2 besten Bids und m/2 besten Asks zu speichern und sowohl die temporären wie auch die definitiven Matches zu berechnen. Um diese Bids und Asks zu speichern werden vier TreeMaps geführt, je zwei pro Ask und Bids. Und zwar braucht es je eine Map in der die Elemente nach Preis sortiert sind und eine, in der die Elemente nach PeerId sortiert sind, weil man sowohl nach Preis wie auch nach PeerId suchen können muss. Letzteres braucht man um schnell überprüfen zu können, ob ein Peer schon ein Angebot in der Tabelle hat oder nicht.

Die zwei offensichtlichsten Methoden, die die Servicetabelle gegen aussen anzubieten hat sind:

```
public Price getHighestBid()

public Price getLowestAsk()
```

welche den aktuell höchsten Bidpreis oder den tiefsten Askpreis zurückliefern.

des weiteren bietet die Servicetabelle die beiden Methoden

```
public boolean putBid(...)

public boolean putAsk(...)
```

an, mit welchen versucht werden kann ein Bid oder Ask in die Tabelle einzufügen. Bei diesen Methoden schaut der AuctionTable zuerst, ob es von dem betreffenden Peer nicht schon einen neueren Eintrag gibt, falls dies nicht zutrifft, wird geschaut ob der Bid oder Ask unter den m/2 Besten ist, falls ja wird dieser dann wirklich eingefügt. Ob er wirklich eingefügt wurde, wird anhand des booleschen Rückgabewert angezeigt. Dabei werden nicht direkt Bids und Asks in die Tabelle eingefügt sondern es werden AuctionTableObjekte (siehe vorheriger Abschnitt) erzeugt und eingefügt.

Die letzten beiden wichtigen Methoden der Servicetabelle sind:

```
public void calculateTempPairs(LinkedList bids, LinkedList asks)

public boolean calculateDefinetlyPairs(PeerMartApp myApp)
```

Die Methode *calculateTempPairs* berechnet die temporären Matches und liefert die dazugehörigen Bids und Asks in den beiden LinkedLists zurück, die als Parameter übergeben werden (dabei werden ServiceObjekte in die Liste eingefügt). Die *calculateDefinetlyPairs* Methode wiederum berechnet die definitiven Matches und im Falle, dass es welche gibt, werden die Peers auch gleich darüber informiert. Die beiden Berechnungsmethoden werden ausschliesslich von der Auktion aufgerufen. Die Referenz *myApp* bei der Methode *calculateDefinetlyPairs* wird benötigt um die neuen NotifyMessageThreads zu erzeugen.

3.5.2 Package messages

Im Package messages befinden sich alle Messageklassen welche PeerMart benutzt. Wie schon im Kapitel 3.3 über Pastry erwähnt wurde, funktioniert die gesamte Kommunikation in Pastry über Nachrichten. Damit eine Nachricht mittels Pastry versendet werden kann muss diese das Interface *rice.p2p.commonapi.Message* implementieren und vollständig serialisierbar sein.

Alle Messages gelangen über die *deliver* Funktion von Pastry zu PeerMart. Dort wird mittels der Methode

```
private void dispatchReceivedMessages(Message arrivedMessage)
```

durch das Anwenden von instanceof geschaut um was für einen Nachrichtentyp es sich handelt und je nach Typ die richtige handle Methode aufgerufen. Es folgt nun eine Übersicht über alle Messagetypen die PeerMart verwendet.

PeerMartMessage

PeerMartMessage ist die abstrakte Basisklasse von welcher alle spezialisierten Messagetypen abgeleitet werden. Der Konstruktor

```
public PeerMartMessage(NodeHandle src, Id tgt, SequenceNr seqNr)
```

verlangt den Source NodeHandle, die Target Id und eine SequenceNr. Mehr zur SequenceNr gibt es unter 3.5.3. Aus der Sourceld und der SequenceNr wird dann die eindeutige Requestld gebildet. Zudem kann über eine Setterfunktion die OriginalRequestId gesetzt werden. Diese wird dann gebraucht, wenn es sich um eine Bestätigungsnachricht auf eine vorhergehende Nachricht handelt.

AuctionMessage

AuctionMessages werden benutzt um die temporären Matches eines BrokerPeers den übrigen BrokerPeers mitzuteilen (vgl. 2.5). Dabei werden die Bids und Asks je in eine LinkedList verpackt. Natürlich muss auch die Serviceld mitgeliefert werden, damit der BrokerPeer weiss, um welchen Service es sich handelt.

BecomeBrokerMessage

BecomeBrokerMessages werden gebraucht um einem Peer mitzuteilen, dass er BrokerPeer für einen bestimmten Service werden muss. Die BecomeBrokerMessage enthält dazu die Serviceld und das BrokerSet für diesen Service.

BrokerMessage

BrokerMesssages werden benutzt, um das BrokerSet einer Auktion einem andern BrokerPeer, Customer oder Provider mitzuteilen. Die Message enthält dafür die Serviceld und das BrokerSet für diesen Service.

ChangeMessage

ChangeMessages werden dann gebraucht, wenn es in der Nachbarschaft eines PastryNodes eine Veränderung gegeben hat und man einem Peer mitteilen muss, dass er entweder das BrokerSet verlassen soll oder er zum neuen RootBrokerPeer eines Services wird. Bei der Aufforderung RootBrokerPeer eines Services zu werden,

überprüft der empfangende Peer selbst noch einmal, ob die Veränderung wirklich stattfand.

FindMessage

Eine FindMessage wird dann von einem Customer/Providern versandt, wenn diese das erste Mal einen Service nutzen wollen, von welchem sie das BrokerSet noch nicht kennen.

NotifyMessage

NotifyMessages werden dann an die Customer/Provider versandt, wenn ein erfolgreicher Match zustande kam. Die Customer/Provider wiederum schicken eine NotifyMessage als Bestätigung für den Empfang der NotifyMessage zurück. Die NotifyMessage enthält neben der ServiceId die NodeId des Auktionspartners und die beiden Preise. Zudem kann ein Flag gesetzt werden, dass es sich um eine Bestätigung handelt.

RequestMessage

Eine RequestMessage wird immer dann von einem Customer/Provider versandt wenn eine sendBid (sendOffer) oder eine getAsk (getBid) Methode im Spiel ist. Das heisst sowohl die Offerten wie auch die Anfragen werden über Requestnachrichten abgewickelt. Um dies zu ermöglichen wird in der Nachricht die ServiceId, der Nachrichtentyp und falls nötig der Preis und die Gültigkeitsdauer für das Gebot gespeichert.

ResponseMessage

Auf jede RequestMessage wird der Brokerpeer mit einer ResponseMessage antworten. Dabei muss dieser Nachrichtentyp folgende Felder beinhalten: Die ServiceId des Services, den Preis des aktuell besten Bids(Asks) den Antworttyp der Nachricht und je nach Messagetyp einen booleschen Wert ob der Bid(Ask) in die ServiceTabelle eingefügt wurde.

3.5.3 Package util

Im Package util befinden sich kleine Helferklassen die aber für das Funktionieren von PeerMart unabdingbar sind. Die Funktionen dieser Klasse werden in den folgenden Unterabschnitten einzeln erklärt.

SequenceNr

Um jede Nachricht und vor allem jeden Bid und Ask eindeutig zeitlich einordnen zu können, braucht es eine SequenceNr. Diese Klasse implementiert diese SequenceNr. Die SequenceNr ist ein long und enthält wichtige Vergleichoperatoren. Wichtig ist die Invariante, dass eine neuere SequenceNr immer grösser ist als eine ältere.

SequenceNrGenerator

Der SequenceNrGenerator implementiert das Design Pattern *Singleton*. Das heisst es gibt immer nur genau eine statische SequenceNrGenerator Instanz. Die Klasse SequenceNrGenerator stellt nur eine einzelne Methode zur Verfügung. Nämlich *getSequenceNr()*, welche eine neue SequenceNr zurückgibt, welche garantiert grösser ist als die vorherige. Der SequenceNrGenerator benutzt als Basiswert System.currentTimeMillis() und erhöht dann kontinuierlich diesen Basiswert.

RequestId

Bei den PeerMartMessages reicht eine SequenceNr alleine nicht aus um eine globale UniqueId zu erlangen, weil die SequenceNr von verschiedenen Peers die gleiche sein kann. Deshalb braucht es die RequestId. Die RequestId ist eine Kombination von SequenceNr und NodeId. Diese beiden Elemente zusammen stellen dann eine PeerMart-weite eindeutige Id dar.

Price

Wie schon im Konzept von PeerMart erwähnt wurde, muss die Währung für einen Service bekannt sein. Die Klasse Price stellt eine Wrapperklasse um die Währung bereit. Innerhalb von PeerMart wird mit der Währung Price gearbeitet. Wie diese genau aussehen soll, kann in dieser Klasse festgelegt werden. In der jetzigen Implementation von PeerMart entspricht der Preis einem double Wert.

PeerMartProperties

In dieser Klasse sind die Properties von PeerMart, also Parameter abgelegt. Dies hat den Vorteil, dass man elementare PeerMart-Grössen wie beispielsweise die BrokerSetgrösse oder die Parameter f (Anzahl paralleler Anfragen) und m (Grösse der Servicetable) einfach und zentral ändern kann. Diese Parameter sind als public final static Felder definiert.

ServiceID

Das Konzept von PeerMart sieht vor, dass jeder Service eindeutig über seine ServiceId identifiziert werden kann. Für PeerMart ist wichtig, dass diese ServiceId eine Erweiterung von *rice.p2p.commonapi.Id* ist, denn so lassen sich die ServiceId's äquivalent zu NodeId's verwenden und können sehr einfach als Destination von Pastry-Messages benutzt werden. Die Klasse ServiceID generiert einen Hashwert aus der ursprünglichen ServiceId oder einem File / einer File Description. Die Ursprüngliche ServiceId kann eine Zahl oder ein String sein.

3.5.4 Package benchmarking

Im Package benchmarking befinden sich die beiden Klassen MaliciousBenchmarkThread und MessageBenchmarkThread welche zum Benchmarken von PeerMart benötigt werden. Die beiden Klassen sind als Threads implementiert, die in ihren run Methoden jeweils das Verhalten eines einzelnen Evaluationspeers implementieren. Dieses Verhalten sieht folgendermassen aus: Zuerst wird der neue Bid oder Ask in Abhängigkeit des aktuellen Preises und der Limite des einzelnen Peers berechnet.

Dann wird der Bid oder Ask versandt und eine gewisse Zeit gewartet. Je nach Testanordnung wird dann ein weiterer Bid oder Ask versandt oder der Peer hört nach einem erfolgreichen Handel auf zu bieten. Mehr dazu auch in Kapitel 4 Evaluation.

3.6 Threading

Threading spielt in PeerMart eine sehr wichtige Rolle. Da es in PeerMart sehr viele asynchrone Methodenaufrufe gibt, die häufig auf Antworten von anderen Peers warten müssen, aber trotzdem die Kontrolle vorhanden sein muss, ob nun beispielsweise die Anfrage bestätigt wurde, gibt es praktisch keine Alternative zu Threads. Alternativ würde ein sehr kompliziertes Eventsystem, welches aber mit sehr grossem Verwaltungsaufwand und grosser Fehleranfälligkeit verbunden wäre, den selben Zweck erfüllen.

3.6.1 Threading der Request

Das eine Anwendungsgebiet von Threads in PeerMart ist das Abhandeln von Transaktionen. Als eine Transaktion wir beispielsweise der ganzen Ablauf des Senden eines Bids bis zum Eintreffen der Bestätigung bezeichnet. Praktisch jede Interaktion zwischen verschiedenen Peers wird in einem eigenen Thread ausgeführt, sobald Verwaltungs- und Kontrollmechanismen ins Spiel kommen. Die nächsten fünf Unterkapitel beschreiben alle MessageThreads welche in PeerMart auftreten.

BasicMessageThread

Die abstrakte Klasse BasicMessageThread implementiert die Grundfunktionalität aller MessageThreads. Jeder MessageThread hat eine eindeutige RequestId mittels welcher Antwortnachrichten dem richtigen Thread zugewiesen werden können. Dann besitzt der BasicMessageThread eine LinkedList, in welche alle Nachrichten eingefügt werden, auf welche dieser Thread wartet. Das eigentliche Einfügen der Nachrichten in die Listen der MessageThreads übernimmt aber die Klasse WaitingThreadList. Wenn ein Thread auf eine Nachricht wartet, trägt sich dieser in diese WaitingThreadList mit seiner RequestId ein. Wenn nun eine entsprechende Nachricht über Pastry eintrifft, übergibt die PeerMartApp Klasse diese Nachricht der WaitingThreadList Klasse, welche dann die Nachricht in die Warteliste der einzelnen Threads einfügt. Sobald die WaitingThreadList eine Nachricht eingefügt hat, wird der dazugehörige Thread über die *interrupt* Methode der Threadklasse informiert, dass er weiterarbeiten kann. Damit sind diese MessageThreads ziemlich effizient implementiert, weil nie ein busy wait stattfindet.

FindMessageThread

Wenn ein Peer eine Anfrage zu einem Service stellt, zu welchem er das BrokerSet noch nicht kennt wird ein FindMessageThread gestartet. Dieser FindMessageThread schickt sofort eine FindMessage an den RootBrokerPeer los. Dann wartet der Thread, bis eine Antwort auf die FindMessage eintrifft oder ein gewisser Timeout überschritten wurde. Beim Überschreiten des Timeouts wird bis zu einer über die Properties einstellbaren Zahl die FindMessage wiederholt. Wenn der Thread das Brokerset erhalten hat, wird die eigentliche Anfrage mittels eines RequestMessageThreads gestartet.

RequestMessageThread

Jedesmal wenn ein Preis angefragt oder ein Bid oder Ask abgegeben wird, wird ein RequestMessageThread gestartet. Dieser Thread sendet als erstes die f parallelen RequestMessages an zufällig gewählte BrokerPeers los. Dann wartet der Thread so lange, bis er von allen f RequestMessages eine Antwort bekommen hat oder das Timeout eintrifft. Auch hier wird beim Überschreiten des Timeouts bis zu einer, über die Properties einstellbaren, Zahl die RequestMessage wiederholt. Hat der Thread nach diesem Warten mindestens eine Antwort erhalten, wird aus allen Antworten der meistgenannte Antwortspreis berechnet und dieser dann mittel der entsprechenden *notify*-Methode der PeerMartApp Klasse der Applikation mitgeteilt. Falls es keinen meistgenannten Preis gibt, wird der jeweils vorteilhafteste Preis zurückgegeben.

NotifyMessageThread

Ein NotifyMessageThread wird immer dann gestartet, wenn es einen definitiven Match gegeben hat und der Customer/Provider darüber informiert wird. Der NotifyMessageThread sendet zu diesem Zweck eine NotifyMessage und wartet auf die Bestätigung. Sollte diese Bestätigung nicht eintreffen wird nach einem gewissen Timeout die NotifyMessage ein weiteres Mal versandt. Die Anzahl der Versuche, diese NotifyMessage zu versenden, wird über einen Parameter in den PeerMartProperties bestimmt.

ChangeMessageThread

Ein ChangeMessageThread wird dann gebraucht, wenn einem anderen Peer mitgeteilt werden muss, dass er entweder das BrokerSet verlassen soll oder aber RootBrokerPeer werden soll. Der ChangeMessageThread sendet eine ChangeMessage und wartet auf die Bestätigung dieser Nachricht. Genau gleich wie der NotifyMessageThread sendet der Thread nach einem gewissen Timeout die Message noch einmal, falls keine Bestätigung eintreffen sollte.

3.6.2 Threading der Auktion

Das zweite wichtige Gebiet von Threads ist das Timing der Auktion. Wie aus Kapitel 2.5 bekannt ist, findet die Auktion in zwei verschiedenen Zeitfenstern statt. Und falls sich in dem jeweiligen Zeitfenster eine Änderung zugetragen hat, muss am Ende des Zeitfensters eine Berechnung durchgeführt werden. Damit diese Berechnung nur dann ausgeführt wird, wenn es wirklich nötig ist und vor allem zum richtigen Zeitpunkt startet, kümmert sich ein Thread, nämlich der AuctionThread, um diese Planung. Eigentlich war das Ziel den AuctionThread als Singleton zu implementieren, dass ist aber nicht möglich, weil mit einer statischen Instanz nicht mehrerer PeerMart Instanzen auf der gleichen JAVA Virtual Machine gleichzeitig laufen können, weil sich die verschiedenen Auktionen der verschiedenen PeerMart Instanzen dann über den gleichen statischen AuctionThread steuern lassen würde, was zu Konflikten führt. Der AuctionThread selbst beinhaltet an Feldern nur ein SortedSet *tasks* mit einem speziellen Comparator. Dieser Comparator ordnet die anfallenden Tasks zeitlich in der richtigen Reihenfolge. Dann gibt es eine *schedule* Methode, über welche neue AuctionTasks in das SortedSet eingefügt werden können. Die run Methode des AuctionThread wartet schliesslich bis zum nächsten

Ausführungszeitpunkt eines Auctiontasks und führt diesen Task dann zum richtigen Zeitpunkt aus.

Der AuktionTask selbst ist eine Klasse, die Informationen besitzt, wann, von welcher Auktion und was für ein Task ausgeführt werden muss (provisorisches oder definitives Berechnen der Matches oder das Einfügen der gequeuten Bids und Asks). Die run Methode dieser AuktionTasks ist wiederum ein einfacher Methodenaufruf auf die Auktions Klasse. Dieses Threading hat erneut zur Folge, dass nur dann etwas berechnet wird, wenn es wirklich nötig ist, das heisst, dass keine busy waits auftreten.

3.6.3 Timertasks

Ein dritter Aspekt bei welchem Threads eine Bedeutung haben sind die Timertasks. Jeder Bid und Ask hat eine vom Customer/Provider gewählte Gültigkeitsdauer. Sobald diese überschritten wird, muss der entsprechende Bid/Ask aus der Tabelle gelöscht werden. Dazu besitzt jede ServiceTabelle einen Timer, dem für jeden eingefügten Bid und Ask ein ValidityTimeOverTask eingefügt wird. Die run Methode dieses ValidityTimeOverTask, welcher den Standart Java TimerTask erweitert, bewirkt dass der entsprechende Bid/Ask aus der ServiceTabelle gelöscht wird.

3.6.4 Alternativer Ansatz: Threadpools

Um die Implementierung der MessageThreads noch effizienter gestalten zu können, bestand zu Beginn der Plan, alle diese Threads über einen Threadpool laufen zu lassen. Das hätte konkret bedeutet, dass immer ein fixe Anzahl von Threads am laufen gewesen wären und die MessageThreads nicht eigentliche Threads gewesen wären, sondern nur Objekte die das Interface *Runnable* implementieren. Sobald dann ein Thread dieses Threadpools frei gewesen wäre, hätte man das Objekt diesem Thread zuweisen können und dieser Thread hätte dann die run Methode dieses Runnable ausgeführt. Mehr Details gibt es unter [16] und [17]. Dadurch wäre der Aufwand erspart geblieben, jedesmal einen neuen Thread zu instanzieren und die maximale Prozessorlast wäre relativ einfach über die fixe Anzahl an Threads zu steuern gewesen. Warum also wurde PeerMart nicht mittels eines Threadpools implementiert? Das Problem ist, dass nur Threads selbst in den *wait* Status gesetzt werden können (Runnables selbst nicht) und auch nur der Thread selbst über notify wieder aufgeweckt werden kann. Das heisst, dass aus der run Methode des Runnable, welches wiederum von der run Methode des Threads aus dem Threadpool ausgeführt wird, keine Kontrolle über den ausführenden Thread selbst erlangt werden kann. Das Problem ist also, dass man in einer run Methode eines Threads nicht ein anderes Runnable ausführen und auch wieder unterbrechen kann. Das bedeutet, dass das Warten auf die Antwortnachrichten als busy wait hätte implementiert werden müssen, was die Performance Vorteile des Threadpools bei weitem überschattet hätte. Die Klasse ThreadPool ist als Alternative jedoch ebenfalls im Package threading enthalten.

3.7 Beispiel einer Transaktion

Bis jetzt wurden die einzelnen Teile von PeerMart detailliert in ihrer Einzelheit betrachtet, so dass eventuell der Gesamtzusammenhang der einzelnen Komponenten

etwas verloren gegangen ist. Dieser Abschnitt hat das Ziel anhand einer relativ komplizierten Transaktion das ganze Zusammenspiel der einzelnen Teile noch einmal vor Augen zu führen.

Als Beispiel wird der Ablauf aufzeigt, wie er stattfindet, falls ein Peer X das erste Mal den Service S1 benutzt und zwar einen Bid für diesen Service schickt welcher mit dem Ask von Peer Y matched.

Als erstes wird die sendBid() Methode der PeerMart Hauptklasse aufgerufen. In dieser Methode wird in der Liste der bekannten Servicelds nachgeschaut, ob Peer X das BrokerSet für diesen Service schon kennt, was nicht der Fall sein wird, weil der Peer X das erste Mal diesen Service benutzt. Deshalb wird ein FindMessage-Thread erzeugt und auch gleich gestartet. Dieser FindMessageThread kreiert eine FindMessage (1), welche über die sendMessage() Methode der Klasse PeerMartApp an die Serviceld des gewünschten Services geschickt wird. Gleichzeitig trägt sich der FindMessageThread in die Liste der auf Nachrichten wartenden Threads ein und geht mittels Thread.wait() für die Länge der Timeoutzeit schlafen. Über Pastry wird die FindMessage automatisch zum RootBrokerPeer geleitet, weil dessen Nodeld numerisch am nächsten bei der Serviceld liegt. Beim RootBrokerPeer wird die Findmessage von Pastry der deliver() Methode abgeliefert. Von dort aus gelangt sie über die dispatchReceivedMessages() Methode zur handleFindMessage() Methode. Dort findet der RootBrokerPeer heraus, dass er bereits Broker für diese Serviceld ist und schickt mittels einer BrokerMessage (2) das BrokerSet für diesen Service zurück. Beim Peer X gelangt dann diese BrokerMessage in die handleBrokerMessage() Methode, wo eine neue Auktion erzeugt wird und die verschiedensten Listen von bekannten Services und die Liste der RootBroker angepasst werden. Zuletzt wird die Message dann der WaitingThreadlist übergeben, welche die Nachricht dem wartenden FindMessageThread in die Liste der angekommenen Nachrichten legt und den Thread aufweckt. Im Thread wird dann die handleArrivedMessage() Methode aufgerufen, welche einen RequestMessageThread erzeugt, diesen startet und sich schlussendlich selbst beendet. Dieser RequestMessageThread sendet als erstes f RequestMessages (3) an zufällig ausgewählte BrokerPeers des dazugehörigen BrokerSets. Der Thread trägt sich wieder in die Liste der wartenden Threads ein und geht dann ebenfalls schlafen. Bei den BrokerPeers treffen dann die RequestMessages ein und landen dort in der handleRequestMessage() Methode, welche die putBid() Methode auf der richtigen Auktion ausführt. Dort wird zuerst geschaut ob sich die Auktion gerade in der richtigen Zeitphase zum Einfügen befindet, wenn ja wird versucht der Bid einzufügen, sonst wird der Bid in die Warteschlange eingefügt und es wird ein AuktionTask erzeugt, der bewirkt, dass sobald der richtige Zeitslot beginnt, der Bid zu diesem späteren Zeitpunkt versucht wird einzufügen. Es wird angenommen, dass der Peer sich im richtigen Zeitslot befindet, also direkt versucht wird, den Bid einzufügen. Dabei wird zuerst geschaut, ob nicht schon ein aktuellerer Bid von Peer X in der Tabelle existiert. Das ist auch nicht der Fall, deshalb wird geschaut, ob der Bid unter die m/2 besten in der Tabelle zu liegen kommt. Das ist der Fall, deshalb wird der Bid in die Tabelle eingefügt und es wird auch gleich ein ValidityTimeOverTask für diesen Bid erstellt, der diesen Bid nach dem Ablauf der Gültigkeitsdauer wieder löschen würde. Die Auktion schickt schlussendlich eine AnswerRequestMessage (4) zurück, welche das Einfügen bestätigt und auch gleich den höchsten Ask mitteilt. Da ein neuer Bid in die ServiceTabelle eingefügt wurde, scheduled die Auktion auf dem BrokerPeer für sich auch gleich ein provisorisches Kalkulieren, sobald der aktuelle Zeitslot beendet ist.

Sobald beim Peer X alle AnswerRequestMessage messages eingetroffen sind (wel-

che auch wieder dem RequestMessageThread mitgeteilt werden) oder das Timeout erreicht wurde, bildet der RequestMessageThread einen Mehrheitsentscheid aus den eingetroffenen AnswerRequestMessages und teilt das Resultat über die notifySendBid() Methode der Applikation oberhalb PeerMart mit. Der RequestMessageThread hat nun seine Arbeit erledigt und beendet sich.

Auf den f verschiedenen BrokerPeers kommt nun das Ende des Einfügezeitabschnitt und der AuctionThread führt nun den AuctionTask aus, welcher auf der Auktion die Methode calculateTempPairs() aufruft. Die Grundannahme war, dass es einen Ask von Peer Y gibt, welcher auf den Bid von Peer X matched. Deshalb wird es auf mindestens einem der f BrokerPeers einen provisorischen Match geben und dieser wird mittels einer AuctionMessage (5) allen übrigen BrokerPeers mitgeteilt. Da nun alle BrokerPeers eine AuctionMessage erhalten, planen auch alle BrokerPeers ein definitives Berechnen der Matches. Sobald nun der Zeitpunkt für das definitive Berechnen der Matches kommt, löst der AuctionThread diese Berechnung wieder aus. Dabei wird der Match von Bid X mit Ask Y auf alle BrokerSets berechnet, wenn es nicht auf einem der Brokersets noch ein besser passendes Angebot gäbe, was nun nicht der Fall sei. Am Ende dieses definitiven Berechnen der Matches werden der Bid X und der Ask Y aus den Servicetabellen gelöscht und diejenigen f BrokerPeers die den Bid von X und den Ask von Y von diesen beiden Peers direkt erhalten haben, starten einen NotifyMessageThread um den betreffenden Peer über den Match zu informieren. Dieser NotifyMessageThread sendet eine NotifyMessage (6) welche bei dem betreffenden Peer das Ausführen der notifySucessfulBid() oder notifySuccessfulOffer() Methode zur Folge hat und auch die NofityMessage rückbestätigt (7). Sobald die Rückbestätigung beim BrokerPeer angekommen ist, hat der NotifyMessageThread seine Aufgabe erledigt und die ganze Transaktion ist abgeschlossen.

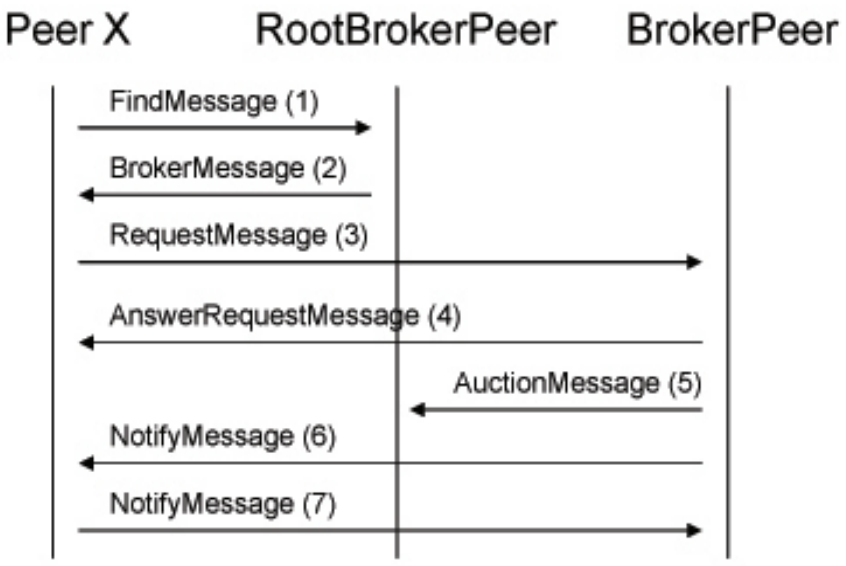

Abbildung 3.2: Ablauf der PeerMart-Nachrichten im Beispiel

Kapitel 4

Evaluation

In diesem Kapitel geht es darum die Implementierung von PeerMart bezüglich Effizienz, Skalierbarkeit und Verlässlichkeit zu evaluieren und vor allem auch mit den theoretischen Berechnungen aus dem PeerMart Paper [8] zu vergleichen.

4.1 Allgemeine Betrachtung

Wie dem PeerMart Paper schon zu entnehmen ist, ist PeerMart sehr schlank aufgebaut. Pro Service, für welchen ein Peer BrokerPeer ist, müssen praktisch nur die m Einträge in der Servicetabelle und das BrokerSet selbst gespeichert werden. Dazu kommten noch einige wenige Listen, in welchen nachgeführt wird, welche Services man selbst kennt. Diese fallen aber kaum ins Gewicht. Vor allem wenn man bedenkt, dass ein heutiger durchschnittlicher Computer doch sehr leistungsfähig ist. Auch die Nachrichten die versandt werden sind alle sehr klein (alle in der Grössenordnung von 1000 Bytes. Die genauen Grössen sind in Abbildung 4.1 ersichtlich) und ausser der AuctionMessage immer von konstanter Grösse. Die AuctionMessage ist je nachdem

Message Type	Size (Byte)
RequestMessage	1143
ResponseMessage	1137
NotifyMessage	1173
FindMessage	1027
ChangeMessage	1064
BrokerMessage	1576
BecomeBrokerMessage	1570
AuctionMessage	1089

Abbildung 4.1: Grössen der PerMart Messages

wie viele Matches darin enthalten sind etwas grösser oder kleiner. Der Unterschied

beträgt 234 Bytes pro zusätzlichem Ask oder Bid, der in der AuctionMessage enthalten ist.

Obwohl PeerMart sehr viele Threads braucht, ist die Anforderung an den Arbeitsspeicher sehr gering. Beim parallelen Starten von 100 PeerMart Instanzen auf einem IBM 1,6 GHz Laptop mit 1 GB Ram wurden unter Windows XP ca. 40 Mb Arbeitsspeicher benoetigt, was für jeden zeitgemässen Computer kein Problem darstellt. Die durschnittliche Prozessorbelastung ist auch bei 100 parallelen PeerMart Instanzen relativ gering und macht durschnittlich 15 - 20% aus. Dabei ist zu beachten, dass die Prozessorbelastung zwischenzeitlich für sehr kurze Zeit bis fast 100 % ansteigt. Der einzige Bottleneck der sich beim Testen herausstellte war die Geschwindigkeit, mit welcher die Pastry Messages weitergeleitet werden. Bei 100 parallelen Instanzen, die alle den Parameter f auf 16 hatten (das heisst jeder Request wird 16-fach parallel verschickt) bekam Pastry Probleme. Numerisch gesehen bedeutet das, dass in relativ kurzer Zeit etwa 1600 Nachrichten verschickt werden. Scheinbar kamen dann die Pastry Anworten auf diese Anfragen nicht genug schnell an, so dass Pastry Fehlermeldungen folgender Art produzierte:

```
Pastry node <0xEAA083..> found <0xC5724A..> to be dead
```

nach einer gewisser Zeit kam dann von Pastry zwar die Entwarnung:

```
Pastry node <0xEAA083..>found <0xC5724A..> to be alive after all
```

aber PeerMart funktionierte nicht mehr zuverlaessig, weil das ganze Timing der Auktion zusammenbrach.

In den nächsten beiden Unterkapiteln wird nun spezifisch auf zwei spezielle Evaluationen eingegangen.

4.2 Overhead Evaluation

Bei dieser Evaluation ging es darum empirisch zu testen, wie viele Nachrichten von PeerMart beim Ablauf der Auktion generiert werden. Die Versuchsanordnung wurde ähnlich gewählt wie bei der theoretischen Simulation im PeerMart Paper Abschnitt 4.1 . Dabei waren folgende Bedinungen gegeben: Es wurden 100 PeerMart Peers lokal auf einer Maschine gestartet wobei 50 davon Consumers und die anderen 50 Providers waren. Es wurde nur ein einzelner Service gehandelt und während dem Handeln gab es keine Veränderungen im P2P Netzwerk. Die Grösse der Servicetabelle war auf 10 (5 Bids und 5 Asks) gesetzt, die Grösse des BrokerSets und die Anzahl parallel gesandter Nachrichten variierte. Gemessen wurde über eine Zeitdauer von 60 Sekunden. Während diesen 60 Sekunden gaben die Provider alle 10 Sekunden einen neuen Ask ab. Die Customer ihrerseits geben nur so lange alle 10 Sekunden einen neuen Bid ab, bis sie einen erfolgreichen Handel zustande brachten. Dabei war die Gebotsstrategie folgendermassen: Conumer wählten ihren Preis folgendermassen: min(ask price + 0.1*(bid limit - ask price), bid limit), Provider hingegen: max(bid price - 0.1*(bid price - offer limit), offer limit). Customer wählten ihr bid limit normalverteilt um den Preis 5, Provider normalverteil um den Preis 6. In Abbildung 4.1 sind die Messresultate zu sehen.

Wenn man nun die Resultate (die genauen Messresultate sind im Appendix unter Abbildung 7.1 zu finden) anschaut fällt auf, dass es doch erhebliche Unterschiede zu den theoretischen Messwerten aus dem Paper gibt. Die Anzahl Nachrichten pro

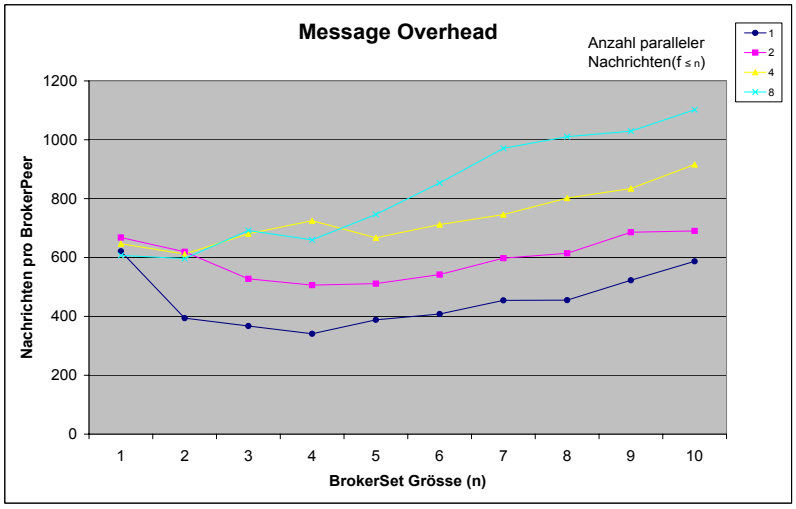

Abbildung 4.2: Message Overhead von PeerMart

Broker nehmen zwar jeweils über einen sehr kleinen Bereich ab, nehmen ab sehr bald mit grösserer BrokerSet Grösse wieder zu. Dies ist damit erklärbar, dass es zwei verschiedene Trends gibt. Einerseits verteilen sich die Nachrichten besser auf die Brokers bei grösserem BrokerSet, andererseits steigt der Verwaltungsaufwand mit zunehmender BrokerSet Grösse. In dem Bereich, wo die Anzahl Nachrichten abnimmt, scheint der abnehmende Trend für kurze Zeit stärker zu wirken. Nun besteht die Frage, woher diese Differenzen herkommen. Erstens sind die beiden Versuche nicht ganz identisch. Bei der theoretischen Simulation hatte es viel mehr Peers und diese gaben jeweils nur ein einzelnes Gebot ab, währendem in der experimentellen Evaluation die Peers über eine gewisse Zeitpunkt immer wieder Gebote abgaben. Zudem konnte die Evaluation nur in einem viel begrenzteren Bereich der BrokerSet Grösse gemacht werden. Auch kommt natürlich eine gewisse Ungenauigkeit durch die zufällige Wahl der bid und ask limits hinzu weil diese über eine Zufallsfunktion für jeden Durchgang neu gewählt werden.

Ein weiterer wichtiger Unterschied lässt sich relativ einfach erklären. Bei der theoretischen Simulation ist der Wert der Nachrichten bei Brokersetgrösse eins für die verschiedenen f's unterschiedlich, in der Evaluation nicht. Das kommt daher, dass PeerMart bei einer Brokersetgrösse von eins und einem Parameter f von vier trotztdem nur eine einzelne parallele Nachricht sendet, weil es keinen Sinn macht dem gleichen BrokerPeer viermal die gleiche Anfrage zu senden.

Eine mögliche Erklärung für das Resultat ist, dass durch die hohe Kadenz der neuen Gebote und den dadurch entstehenden Synchronisationsnachrichten, das Resultat so fest abweicht. Eventuell wurde in der theoretischen Simulation auch der Aufwand vernächlässigt der nötig ist um nach erfolgreichen Matches den jeweils besten Bid und Ask unter den Broker zu synchronisieren.

Die Evaluation zeigt auf alle Fälle dass eine richtige Wahl der beiden Parameter n und f elementar für die Skalierbarkeit ist und dass auf jeden Fall noch weitere Evaluationen gemacht werden müssen. Leider zeigt aber diese Evaluation auch,

dass zumindest bei dieser Testanordnung diese PeerMart Implementation nicht so schön skaliert, wie es die theoretische Berchnung eigentlich vorgezeigt hat.

4.3 Reliability Evaluation

Ziel dieser zweiten Evaluation war es zu testen, wie sich die Zuverlässigkeit von PeerMart verhält, wenn bösartige Peers involviert sind. Auch diese Evaluation ist so gewählt, dass sie eine gewissen Ähnlichkeit mit der theoretischen Simulation aus dem PeerMart Paper Abschnitt 4.2 hat.

In dieser Evaluation wurden böswillige Peers so modelliert, dass diese nicht auf Request- und Auctionnachrichten eingehen. Da im realen System nicht von vorherein klar ist, wer genau BrokerPeer wird, wurden die böswilligen Peers prozentual auf alle Peers verteilt. Die Testanordnung war wieder sehr ähnlich derjenigen der vorhergehenden Evaluation: 100 Peers davon 50 Customer und 50 Providers. Wieder wurde nur ein einzelner Service gehandelt und wieder blieb das P2P Netzwerk über die ganze Zeitdauer der Evaluation gleich. Die Servicetabellengrösse war wieder 10 und die Grösse des BrokerSets war 16. Diese Evaluation lief aber über eine Zeitdauer von 3 Minuten, wobei die Consumers und Providers alle 15 Sekunden ein neues Angebot abgaben. Sobald aber ein Customer oder Provider einen Handel erfolgreich abgeschlossen hat, hört er auf zu bieten. Die Wahl des Preises erfolgte bei diesem Test viel einfacher. Die Customer wählen einen normalverteilten Preis um 6 die Provider einen um 4 aus. Dieses Preiswahl hat zur Folge, dass ohne malicious Peers am Endes des Testlaufs 100% der Peers einen erfolgreichen Handel abgeschlossen haben. In Abbildung 4.2 sind die Resultate dieses Tests zu sehen (die genauen Daten sind im Appendix unter Abbildung 7.2 zu finden).

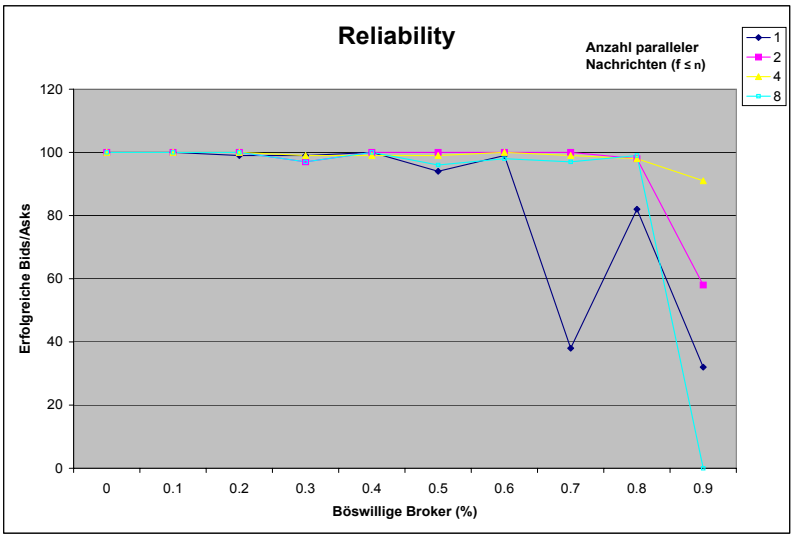

Abbildung 4.3: Zuverlässigkeit von PeerMart

Schon auf den ersten Blick sieht man, dass die Zuverlässigkeit von PeerMart enorm hoch ist, sogar bei nur einer parallelen Nachricht (f = 1). Auch dieser Test-

lauf ist nicht eins zu eins mit dem theoretischen Test aus dem Paper vergleichbar. Erstens werden hier die böswilligen Knoten prozentual auf alle Knoten verteilt, so ist immer etwas Zufall im Spiel, wieviele der Brokerpeers nun wirklich böswillig sind, was jedoch unter realen Bedingungen nicht anders ist. So ist beispielsweise der krasse Abfall bei der Messung mit 8 parallelen Nachrichten und böswilliger NodeRate von 0.9 relativ einfach damit zu erklären, dass alle Brokerpeers böswillig waren und somit keine einzige Anfrage mehr verarbeitet wurde. Und als zweiter grosser Unterschied hatten die Peers bei dieser Anordnung mehrere Chancen doch noch zu einem erfolgreichen Handel zu kommen, weil sie mehrere Male ihre Bids und Asks senden konnten. Gerade dieser zweite Unterschied führt zu der sehr hohen Erfolgsquote auch bei relativ hohen Prozentsätzen der böswilligen Peers.

Aus dieser Evaluation lässt sich ableiten, dass PeerMart im Einsatz eine sehr hohe Zuverlässlichkeit bietet. Es sollte aber sicher noch eine Evaluation gemacht werden, bei welchem jeder Peer genau einen fixen Bid oder Ask hat und diesen nur einmal abgibt und dann die Rate noch einmal ausrechnen.

Kapitel 5

Zusammenfassung und zukünftige Arbeiten

Das grosse Hauptziel, das Konzept von PeerMart zu verfeinern und schliesslich mit Hilfe eines vorhandenen P2P-Frameworks zu implementieren, wurde weitgehend erreicht. Mit dieser Arbeit liegt nun eine lauffähige Java-Version von PeerMart vor welche FreePastry benutzt. Auch wurde die Implementation anhand von zwei Tests evaluiert, wobei diese Evaluation aus Zeitgründen nicht ganz so ausführlich stattfand wie eigentlich geplant gewesen wäre. Dieser Bericht, der als Dokumentation der geleisteten Arbeit zu verstehen ist, rundet diese Semesterarbeit ab.

Als grösste Knacknuss erwies sich der Zeitplan. Aus persönlichen Gründen und dem etwas unterschätzten Zeitaufwand verzögerte sich das Ende der Semesterarbeit um einen Monat. Als enorm zeitaufwendig stellte sich das sehr komplizierte Debuggen von dieser stark verteilten und auf Threads basierenden Applikation heraus. Auch das Evaluieren erwies sich als einiges zeitaufwendiger als geplant, da alleine das Starten von 100 parallelen Instanzen über zwei Minuten in Anspruch nahm.

Als *lessons learned* kann der Umgang mit Threads, der komplizierte Aufbau verteilter Applikationen und auch das Weitervervenden eines Frameworks aus dieser Semesterarbeit mitgenommen werden. Im Grossen und Ganzen hat die Arbeit doch mehrheitlich Spass gemacht und ich konnte dabei sehr viel profitieren. Zuletzt möchte ich mich hiermit bei David Hausheer ganz herzlich für die sehr angenehme und gute Betreuung bedanken.

Diese Semesterarbeit befasste sich hauptäschlich mit der Implementation von PeerMart. Folgende Punkte konnten dabei aus zeitlichen Gründen nicht berücksichtigt werden und sind Teil von zukünftigen Arbeiten:

5.1 Kryptographie

Das Konzept von PeerMart sieht vor, dass jede einzelne Nachricht, die über PeerMart versandt wird signiert wird. Eventuell wäre es angebracht die Nachrichten bei kritischen Applikationen sogar zu verschlüsseln. Ob dabei eine PKI vorausgesetzt wird, oder der interessante Ansatz aus Paper [11] verwendet wird, ist eine andere Frage. Auf alle Fälle wäre es sicher wichtig, dass Signaturen einführt werden, um die Authentität und die Integrität einer Nachricht überprüfen zu können. Dabei darf

natürlich nicht vernachlässigt werden, dass das Einführen von kryptographischen Verfahren zusätzliche Rechenleistung beansprucht.

5.2 Dynamische Grösse des BrokerSet

Wie die theoretischen Berechnungen im PeerMart Paper (vgl. Abschnitt 4.1 in [8]), und auch die Evaluation dieser PeerMart Implementation zeigen, hat die Grösse des BrokerSets einen enorm grossen Einfluss auf die Effizienz und die Skalierbarkeit des ganzen Systems. Die Grösse des BrokerSets ist zwar durch den Paramter n einstellbar, diese kann aber nur einmal, vor dem Starten der Applikation verändert werden und bleibt dann über die ganze Laufzeit für alle Services identisch. Optimal wäre nun, wenn sich die Grösse des BrokerSets dynamisch während dem Laufen der Auktion anpassen würde. Je nachdem wie stark ein gewisser Service gerade gehandelt wird, würde man die BrokerSetgrösse den Umständen entsprechend anpassen. Durch dieses dynamische Anpassen hätte man immer eine optimale Anzahl an Messages und das System würde besser skalieren.

5.3 Single Point of Failure beim Finden des RootBrokers

PeerMart ist darauf ausgelegt, dass trotz möglicher bösartiger Peers und plötzlichen Ausfällen von einzelnen Peers im Peer-to-Peer Netzwerk eine sehr hohe Zuverlässigkeit garantiert wird. Diese hohe Zuverlässigkeit wird vor allem durch Redundanz erreicht. In der jetzigen Implementation gibt es aber einen Single Point of Failure. Wenn ein Peer das erste Mal einen Service handeln will, wird Pastry benutzt um den Peer zu finden, der numerisch am nächsten bei dieser ServiceId liegt (den RoorBrokerPeer). Dies geschieht, indem Pastry als Zieladresse einfach die ServiceId angegeben wird. Falls nun dieser RootBroker aber malicious ist und z.B. nicht auf diese FindMessage reagiert (was durchaus sein kann, weil der Knoten nicht BrokerPeer werden will), gibt es ein Problem. Der Peer, der den Service handeln möchte bemerkt zwar, dass der RootBroker ihm keine Antwort liefert, aber machen kann er dagegen nicht viel. Zusätzlich zu implementieren wäre ein spezieller Fallbackmechanismus, über welchen ein Node kontaktiert werden kann, der numerisch am nächsten beim RootBroker liegt. Über diesen Peer gäbe es die Möglichkeit an das richtige BrokerSet zu gelangen und diesem Peer würde auch mitgeteilt, dass er die Aufgabe des RootBrokerPeers übernehmen muss.

5.4 DHT

Sobald eine Distributed Hash Table einsetzt wird, treten auch gewisse Risiken auf. Denn wenn z.B. genügend böswillige Peers vorhanden sind, ist es möglich eine DHT in zwei oder mehrere voneinander getrennte Teilnetzwerke aufzuspalten. Falls es soweit käme, würden natürlich die Peers des einen Pastry-Netzwerkes nichts von den Angeboten der Peers des anderen Pastry-Netzwerkes erfahren. In den Release Notes von FreePastry 1.2 ist auch eindeutig festgehalten:

```
FreePastry 1.2 limitations:
```

```
Minimal security (no support for insecure networks
or malicious nodes)
```

PeerMart verwendet zwar FreePastry in der Version 1.3 aber im Changelog ist nichts aufgeführt, dass diese Sicherheitslücke entfernt wurde. Verbessern kann man in dieser Hinsicht nicht viel, ausser auf die nächste Version von FreePastry zu warten und diese sobald als möglich einsetzten. Aber die Aussichten sehen nicht schlecht aus, werden doch auf der Homepage von FreePastry folgende Ziele genannt:

```
FreePastry future:

Subsequent releases will support strong security and allow
deployment on insecure networks where nodes may fail
arbitrarily.
```

5.5 (D)DoS Attacken

Wie bei fast allen verteilten Anwendungen sind Denial of Service Attacken ein offenes Problem. Ein oder mehrere Peers können problemlos ununterbrochen neue Anfragen an PeerMart stellen, was zu einer sehr grossen Last bei einzelnen BrokerPeers führt und eventuell sogar den korrekten Verlauf der Auktion behindert. Man müsste also nach Lösungen suchen wie man Denial of Service oder sogar Distributed Denial of Service Attacken erfolgreich ausweichen kann. Es darf aber gesagt werden, dass (D)DoS Attacken für PeerMart nicht so schlimm sind wie für andere P2P-Systeme, da in PeerMart immer nur einige wenige Services davon betroffen wären.

5.6 Reputationssystem

PeerMart ist nur dafür verantwortlich, die abgegebenen Bids und Asks miteinander zu matchen. PeerMart kann aber nicht überprüfen, ob die abgegeben Gebote von den Peers wirklich auch eingehalten werden. An dieser Stelle müsste man ein Reputationssystem wie beispielsweise EigenRep [13] oder NICE [14] einsetzen. Dieses System würde die Reputation von Peers, welche ihre Angebote nicht einhalten kontinuierlich senken. Ab einem gewissen Level würde gar nicht mehr auf die Bids und Asks von Peers eingegangen werden, welche eine zu tiefe Reputation besitzen. Das Einbauen eines Reputationssystems ist darum wichtig, weil nicht ernst gemeinte Gebote das Auktionssystem sehr schnell überlasten können.

5.7 Evaluation

Die Evaluation, welche im Rahmen dieser Semesterarbeit gemacht wurde, zeigt zwar einige interessante Resultate, aber es gibt noch viel mehr zu tun. So sind beispielsweise alle Tests bis jetzt so durchgeführt worden, dass alle PeerMart-Instanzen auf der gleichen CPU liefen, also praktisch keine Delays zwischen den einzelnen Nodes auftraten. Auch sind noch keine Tests gemacht worden, wie sich PeerMart in einem Netzwerk verhält in dem es sehr häufige Joins und Leaves von Nodes gibt. Zudem sollten die theoretischen Tests aus dem PeerMartPaper noch genauer nachgebildet werden, um noch besser vergleichbare Resultate zu erreichen. Auch müssen

noch mehr Tests gemacht werden, um herauszufinden welche Parameter für welches Einsatzgebiet am effizientesten sind.

Kapitel 6

Appendix

In diesem Appendix sind die genauen Messdaten der beiden Evaluationen zu finden, die im Zuge dieser Semesterarbeit gemacht wurden. Abbildung 7.1 zeigt die Messdaten der Message Overhead Evaluation, Abbildung 7.2 die Messdaten der Reliability Evaluation.

BrokerSet Grösse (n)	Anzahl parallele Anfragen (f)			
	1	2	4	8
1	621	668	647	607
2	334	570	569	547
	454	669	654	646
3	327	480	643	670
	329	494	655	643
	446	609	744	762
4	300	478	690	621
	323	481	692	674
	324	497	709	627
	417	569	809	718
5	368	478	631	706
	371	493	653	727
	375	494	657	742
	376	506	667	747
	451	586	727	809
6	378	513	675	820
	385	526	723	827
	388	526	676	847
	391	538	702	852
	394	538	708	855
	513	611	788	920
7	424	571	694	934
	428	574	738	940
	439	580	724	958
	439	583	748	970
	447	592	747	968
	469	601	753	964
	536	684	816	1062
8	432	590	778	981
	437	593	784	993
	443	593	785	997
	445	596	786	1001
	446	604	790	1002
	448	606	795	1003
	448	612	804	1005
	543	720	894	1102
9	500	665	810	998
	501	668	811	1012
	507	671	820	1019
	507	675	824	1023
	515	676	827	1024
	515	676	831	1026
	521	688	835	1033
	526	689	839	1033
	612	765	913	1095
10	565	667	882	1056
	566	684	895	1088
	575	685	904	1097
	576	670	906	1101
	576	666	909	1105
	581	688	911	1096
	582	679	913	1102
	582	685	921	1097
	588	683	928	1103
	679	796	993	1177

Abbildung 6.1: Anzahl Nachrichten pro BrokerPeer in Abhängigkeit der Grösser des BrokerSets (n) und der Anzahl parallel gesendeter Nachrichten (f)

Böswillige Broker Peers (%)	Anzahl parallele Anfragen (f)			
	1	2	4	8
0	100	100	100	100
0.1	100	100	100	100
0.2	99	100	100	100
0.3	99	97	99	97
0.4	100	100	99	100
0.5	94	100	99	96
0.6	99	100	100	98
0.7	38	100	99	97
0.8	82	98	98	99
0.9	32	58	91	0

Abbildung 6.2: Prozentzahlen der erfolgreich gematchten Bids und Asks in Abhängigkeit des Prozentsatzes der böswilligen Peers und der Anzahl parallel gesendeter Nachrichten (f)

Literaturverzeichnis

[1] R. Buyya, S. Vazhkudai: *Compute Power Market: Towards a Market-Oriented Grid* IEEE Session on Global Computing on Perosnal Devices (In conjunction with CCRID 2001), Brisbane, Australia, May 2001

[2] Z. Despotovic, J. Usunier, K. Aberer: *Towards Peer-To-Peer Double Auctioning* In Proceedings of the 37th Hawaii International Conference on System Sciences, Waikoloa, HI, USA, January 2004

[3] R. Dingledine, M.J. Freedman, D.Molnar: *In Peer-To-Peer: Harnessing the Power of Disruptive Technologies* O'Reilly & Associates, Chapter 16, pp. 217 - 340, 1st edition, March 15, 2001

[4] D. Hausheer, N. Liebau, A. Mauthe, R. Steinmetz, B. Stiller: *Token-based Accounting and Distributed Pricing to Introduce Market Mechanisms in a Peer-to-Peer File Sharing Scenario* In Proceedings 3rd IEEE international Confernce on Peer-to-Peer Computing, Linköping, Sweden, September 1-3, 2003

[5] D. Hausheer: *Economics of Peer-to-Peer Systems* Dagstuhl Seminar on Peer-to-Peer Networking & Applications Dagstuhl, Germany, March 2004

[6] E. Ogston, S. Vassiliadis: *A Peer-to-Peer Agent Auction* In Proceedings of the First International Joint Conference on Autonomous Agents an Multi-Agent Systems (AAMAS), Bologna, Italy, July 2002

[7] H. Varian: *Markets for Informations Goods* In Proceedings of Monetary Policy in a World of Knowledge-Based Growth, Quality Change, and Uncertain Measurement, June 1998

[8] D. Hausheer, B. Stiller: *PeerMart: A Distributed Market for P2P Services* Zürich, Switzerland 2004

[9] A.Rowstron and P. Druschel: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems* In Proceedings of IFIP/ACM Middleware 2001, Heidelberg, Germany, November 2001

[10] V. Vishnumurthy, S. Chandrakumar, E. G. Sirer: *KARMA: A Secure Economic Framework for Peer-to-Peer Resource* Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA June 5-6, 2003

[11] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron and Dan S. Wallach: *Secure routing for structured peer-to-peer overlay networks* 5th Usenix Symposium on Operating System Design and Implementation, Bosten, MA, December 2002

[12] B. Cohen: *Incentives Build Robustness in BitTorrent* Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, June 5-6, 2003

[13] S. Kamvar, M.Schlosser and H. Garcia-Molina: *Eigenrep: Reputation management in P2P networks* In Proceedings of 12th Intl. Wordl Wide Web Conference, 2003

[14] S. Lee, R. Sherwood and B. Bhattacharjee: *Cooperative peer groups in nice* IEEE INFOCOM, Apr. 2003

[15] JXTA: *http://www.jxta.org/*

[16] Thread pools and work queues: *http://www-106.ibm.com/developerworks/library/j-jtp0730.html*

[17] Thread-Erzeugung, -Scheduling und -Terminierung: *http://www.ps.uni-sb.de/courses/progsprach-ws97/concurrency.ps*

Abbildungsverzeichnis