CARL VON OSSIETZKY universität OLDENBURG

Ausarbeitung für den praktischen Teil des Moduls
"Software System Engineering"

Vergleich von Peer-to-Peer-Architekturstilen im Rahmen von Simulationsstudien

Nina Marwede
11. Semester Diplom-Informatik

13. Februar 2006

Abstract

Dieses Dokument beschreibt Untersuchungen an Peer-to-Peer-Architekturen. Mit dem 'yEd Graph Editor' erzeugte Graphen bilden die Basis für mit dem Simulationswerkzeug eaSim erzeugte Topologien. Auch dazu passende Anfragen nach Peers oder deren Ressourcen kann eaSim erzeugen – dem Benutzer bleibt die Auswahl oder Implementierung einer geeigneten Suchstrategie überlassen. Schwerpunkt der vorliegenden Arbeit sind Varianten der Breitensuche (BFS), wobei primär Wert gelegt wird auf die Qualität der Suchergebnisse, und nur sekundär auf den Verbrauch von Zeit und Transportkosten.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Festlegung der Ziele / Planung

Untersuchungsschwerpunkt soll die Qualität der Suche sein: Eine Suchanfrage soll möglichst viele Treffer liefern, wobei der Verbrauch von Zeit und (Netzwerk-)Ressourcen zweitrangig ist.

Von den in eaSim[1] zur Verfügung stehenden **Metriken** eignet sich dazu "Query success rate" am besten – hierbei wird das Verhältnis von Antworten zu den zugehörigen Anfragen gemessen. Dabei ist primär die Höhe des Wertes von Belang und sekundär der Zeitpunkt, an dem die 100%-Grenze erreicht wird.
Die Metriken "Answers per message" und "Query answer relation" sind interessant, weil identische Ressourcen an mehreren Peers vorhanden sein können.
Die Metrik "Hops per query until answer" lässt erkennen, ob Nachrichten wegen zu niedrigem Wert Hops-to-live verloren gehen: in diesem Fall wird genau diese obere Schranke erreicht, aber nicht überschritten.
Die verbleibenden beiden Metriken sind nur mäßig interessant für die Fragestellung, weil es dabei hauptsächlich um die Auslastung der Ressourcen geht, die zwar im Allgemeinen möglichst niedrig ausfallen sollte, jedoch für den gewählten Untersuchungsschwerpunkt zunächst keine große Rolle spielen soll. Geringere Werte sind also zu bevorzugen, solange die Qualität der Ergebnisse nicht darunter leidet.

Als **Architekturstil** entscheide ich mich für "reines Peer-to-Peer" mit relativ hohem Grad an Vermaschung. Die Größe des Netzes ist nur beschränkt durch die Dauer der Simulation – daher wähle ich zunächst einen Graphen mit 500 Knoten und 1000 Verbindungen. Später könnten beide Parameter variiert werden.

Die Peers sollen zufällig verteilte **Ressourcen** bereitstellen, nach denen gesucht wird. Die Konstruktion übernimmt der "Virtual topology generator" von eaSim. Zuerst dachte ich dabei an physische Geräte wie Drucker oder Scanner, doch schließlich erschien mir die Simulation von Filesharing-ähnlichen Strukturen interessanter, daher gebe ich als Minimum 0 und als Maximum 100 Ressourcen vor – bei 500 Knoten sollten also insgesamt etwa 25.000 Ressourcen zur Verfügung stehen.

Die **Anfragen** nach den Ressourcen sollen ebenfalls zufällig verteilt sein, sowohl den Zeitverlauf als auch den Ausgangsort betreffend. Der "Query Generator" von eaSim ist hinreichend zur Erstellung und bietet nur zwei Optionen. Zur Simulation einer mittelgroßen Last wähle ich zunächst (durchschnittlich) 5 Anfragen pro Zeitschritt, und zwar über eine Dauer von 100 Zeitschritten – in der Summe also etwa 500 Anfragen.
Zum Debuggen des Suchalgorithmus' bietet sich im Verlauf der Untersuchungen eine kleinere Anzahl an, für Belastungstests eine größere.

[1] `http://sourceforge.net/projects/easim/`

Die **Suchstrategie** soll zunächst einfach sein, damit sie überhaupt erstmal funktionieren kann, denn die Quelltexte von eaSim scheinen auf den ersten Blick nicht gut verständlich und kaum dokumentiert. Es ist manchmal schwer zu erkennen, was die vorhandenen Algorithmen tun, ob sie es richtig tun, außerdem sind die Beschreibungen leider oft unverständlich und zum Teil irreführend. Ich entscheide mich für eine Breitensuche (BFS) mit ein paar eigenen Optimierungen für Qualität (primär) und Zeit (sekundär).

Darstellung in der GQM-Methode[2]

Goal	Ziel/Zweck Sachverhalt Objekt/Prozess Standpunkt	Vergleich der Qualität von gerichteter Breitensuche nach Ressourcen Entwicklersicht
Question		Wie viele Suchanfragen werden beantwortet?
Metrics		Query answer relation Answers per message
Question		Wie hoch ist die Erfolgsrate?
Metrics		Query answer relation Query success rate
Question		Ist Hops-to-live groß genug?
Metrics		Hops per query until answer

[2]Goal, Question, Metric – vorgestellt in der Großen Übung zur VL am 10.01.2006

2 Topologien

Ich erzeuge mit dem “yEd Graph Editor”[3] mittels des Werkzeugs “Planarer Graph” eine Topologie bestehend aus zunächst 500 Knoten und 1000 Kanten. Als sinnvolle Anordnung bietet sich das Layout “Baumartig” mit dem Stil “Gerichtet/Geradlinig” an; für kleinere Graphen “Orthogonal”.
Der Topologie-Generator von eaSim erzeugt daraus ein Peer-to-Peer-Netz (Abbildung 1) mit (wie in der Planung festgelegt) jeweils 0 bis 100 Ressourcen pro Peer.

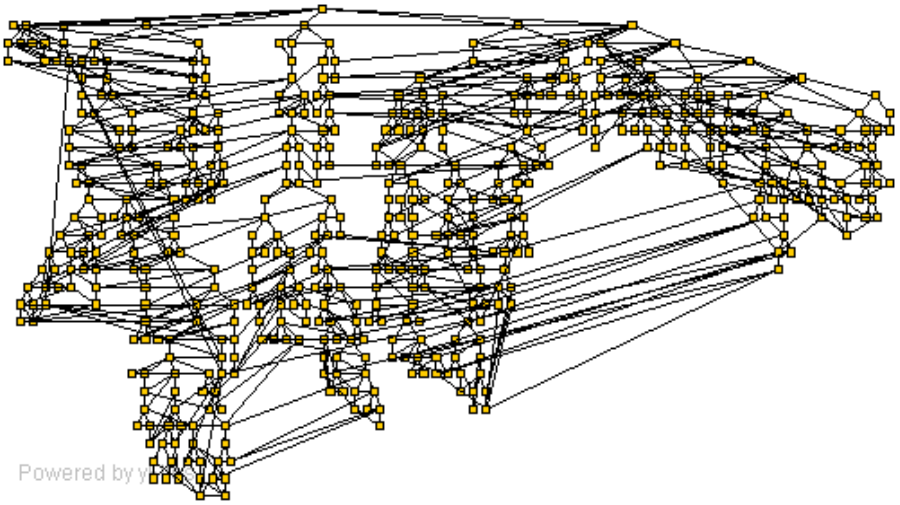

Abbildung 1: Peer-to-Peer-Netz, 500 Knoten, 1000 Kanten

Zum Vergleich bieten sich folgende Varianten an, nach dem gleichen Prinzip erzeugt:

- 50 Knoten und 100 Kanten – ein relativ kleines Netz – Abbildung 2
- 500 Knoten und 500 Kanten – lose zusammenhängend – Abbildung 3
- 2500 Knoten und 5000 Kanten – ein relativ großes Netz – Abbildung 4

Weiterhin sind Veränderungen der Ressourcenverteilung bei wieder ursprünglicher (500/1000) Netzgröße denkbar:

- 0 bis 5 – sehr wenige Ressourcen pro Peer
- 25 bis 50 – mäßig viele Ressourcen pro Peer

[3] `http://www.yworks.com/en/products_yed_about.htm`

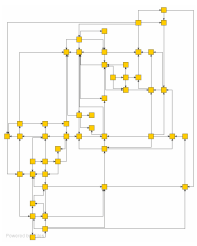

Abbildung 2: Peer-to-Peer-Netz, 50 Knoten, 100 Kanten

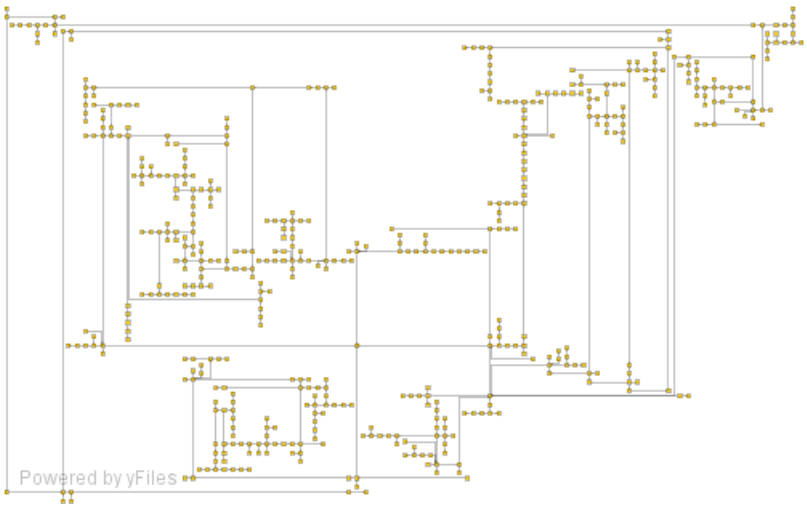

Abbildung 3: Peer-to-Peer-Netz, 500 Knoten, 500 Kanten

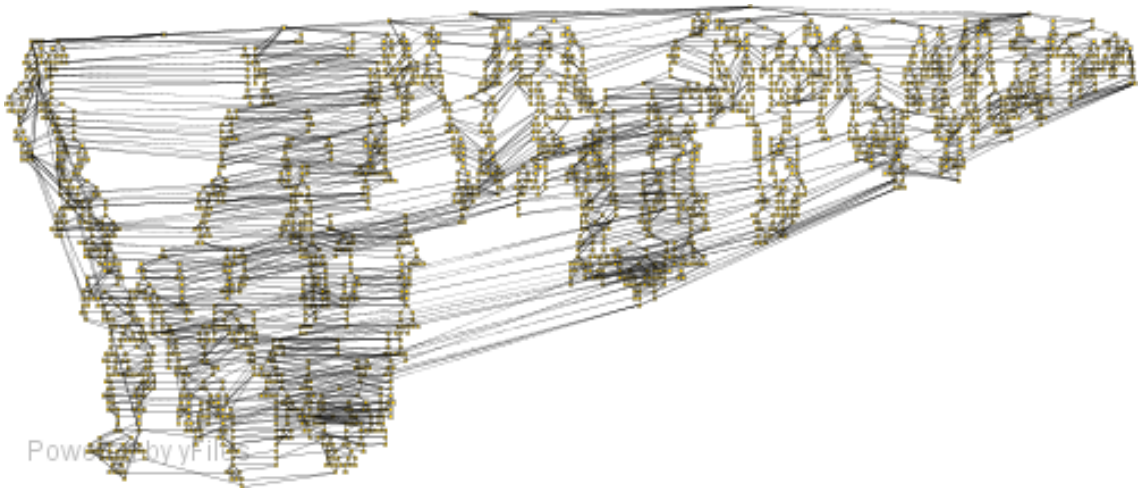

Abbildung 4: Peer-to-Peer-Netz, 2500 Knoten, 5000 Kanten

3 Anfragen

Für die ersten Tests erzeuge ich mittels des Query Generators von eaSim zwecks Übersichtlichkeit nur wenige Anfragen für die virtuelle Ebene, so dass die println-Ausgaben auf der Konsole gut verfolgt werden können. Mit den Parametern “2 Nachrichten pro Simulationsschritt” und “letzte Nachricht nach 20 Schritten” werden 28 Anfragen erstellt.

Sobald der Suchalgorithmus zufriedenstellende Ergebnisse liefert, erhöhe ich die Parameter auf “5 Nachrichten pro Simulationsschritt” und “letzte Nachricht nach 100 Schritten” – es werden 432 Anfragen erstellt.

Auch für die Untersuchungen an den weiteren Topologien benutze ich zur besseren Vergleichbarkeit die selben Werte, sodass mit 460, 425 bzw. 448 ähnliche Anzahlen an Anfragen entstehen.

4 Suchstrategien

Meine Strategie zur Breitensuche orientiert sich zunächst an der Klasse `SuperPeerStrategy_Exercise_1`. Das Grundprinzip ist die möglichst weite Verbreitung der Anfrage, daher werden keinerlei willkürliche Einschränkungen hinsichtlich der Empfänger gemacht.
Der einzige Parameter ist der Wert "Hops-to-live" – dies ist die Anzahl der Peers, die eine Nachricht maximal passieren darf, und somit die maximale Distanz (gemessen in Netzknoten), die eine gesuchte Ressource von ihrem suchenden Peer haben darf. Für eine hohe Qualität der Suche, wie zu Beginn festgelegt, sollten also die Hops-to-live hinreichend groß gewählt werden, damit eine Suche nicht frühzeitig abgebrochen wird. In Distanz ausgedrückt: der Suchradius sollte hinreichend groß sein.

Der Basis-Algorithmus beschreibt sich wie folgt:

- Abbruch der lokalen Suche bei Erreichen des Wertes Hops-to-live.
- Beschränkung auf die Suche nach Ressourcen.
- Prüfung auf lokale Verfügbarkeit der gesuchten Ressource.
- Falls verfügbar, Antwort an den ursprünglichen Fragesteller.
- Ansonsten Weiterleitung der Anfrage an alle Nachbarn.

```
1  public void search( SimulationElement element, Query query ){
2      if( query.maxHopsReached( htl.intValue() ) ){ return; }
3      if( !((VirtualQuery)query).getSearchType().equals( "ressource" ) ){
4          return; }
5      String searchCriteria = query.getSearchCriteria();
6      if( element.getElementData( "ressource" ).contains( searchCriteria ) ){
7          Answer answer = new Answer(
8              element.currentTime().getTimeValue(), element, query,
9              element.getName() + " provides ressource '" +searchCriteria+ "' ");
10         query.addAnswer( element, answer ); // statistics
11         sendMessage( element, query.getInitiator(), answer );
12         return;
13     }
14     SuperPeer[] neighbors = ((SuperPeer)element).getNeighborhood().
           getNeighbors();
15     boolean original_forwarded = false;
16     for( int i=0; i<neighbors.length; i++ ){
17         if( original_forwarded ){
18             sendMessage( element, neighbors[i], query.getCopy() );
19         }else{
20             sendMessage( element, neighbors[i], query );
```

```
21              original_forwarded = true;
22          }
23      }
24  }
```

Hauptproblem dabei ist das fehlende Gedächtnis: Nirgendwo wird gespeichert, welche Nachricht bereits bei welchem Peer verarbeitet worden ist. So steigt zwar die "Query success rate" beim Experiment mit eaSim relativ schnell in akzeptable Bereiche, allerdings werden Anfragen mehrfach von den gleichen Peers beantwortet (in den Debug-Ausgaben beobachtet), und die Anzahl der Antworten steigt signifikant über die Zahl der Anfragen (Metrik "Query answer relation"). Vor allem wird jedoch die Performanz des Simulators massiv durch die exponential steigende Gesamtanzahl der Nachrichten (Metrik "Virtual messages in system") beeinträchtigt.

Als Erweiterung, um dem Algorithmus ein Gedächtnis zu geben, von welchen Peers eine Nachricht bereits verarbeitet worden ist, werden zwei Blöcke eingefügt:

- Am Anfang der Funktion:
  `query.getHistory().addElementToHistory( element );`
- Am Anfang der For-Schleife:
  `if(query.getHistory().historyContainsElement(neighbors[i])){ continue; }`

Dabei kann es jedoch aufgrund der zeitdiskreten Verarbeitung im Simulator immer noch passieren, dass eine Anfrage an einen Peer geschickt wird, während dieser gerade genau diese Anfrage schon verarbeitet. Dies kann mit den vorhandenen Mitteln nicht ohne weiteres verhindert werden und bewirkt, dass in der Statistik bei hinreichend großen Simulationszeiten sowie "hops to live"-Werten deutlich mehr Antworten als Anfragen existieren.
Allerdings kann verhindert werden, dass ein Peer eine bestimmte Anfrage nochmals verarbeitet, die ihn bereits im vorherigen Zeitschritt erreicht hat – dazu wird eine weitere Zeile am Anfang der Funktion eingefügt:

- `if( query.getHistory().historyContainsElement( element ) ){ return; }`

Da der Initiator einer Anfrage zu Beginn nicht in der History enthalten ist, wird in einer weiteren Optimierung am Anfang der For-Schleife verhindert, dass er seine eigene Anfrage bekommt.

- `if( neighbors[i].equals( query.getInitiator() ) ){ continue; }"`

Die letzte Version habe ich der besseren Lesbarkeit halber in Unterfunktionen zerlegt. Im folgenden Listing sind zwecks Übersichtlichkeit die Debug-Ausgaben entfernt – siehe beiliegenden Quelltext `SuperPeerStrategy_BFS_2.java`.

```
1 public void search( SimulationElement element, Query query ){
2     if( query.maxHopsReached( hopsToLive.intValue() ) ){
3         return; }
4     if( query.getHistory().historyContainsElement( element ) ){
5         return; }
6     if( !((VirtualQuery)query).getSearchType().equals( "ressource" ) ){
7         return; }
8     query.getHistory().addElementToHistory( element );
9     if( !sentAnswerToInitiator( element, query, query.getSearchCriteria() )){
10        forwardQueryToNeighbors( element, query,
11            ((SuperPeer)element).getNeighborhood().getNeighbors() );
12    }
13 }
14
15 private boolean sentAnswerToInitiator( SimulationElement element,
16        Query query, String searchCriteria ){
17    if( element.getElementData( "ressource" ).contains( searchCriteria ) ){
18        Answer answer = new Answer(
19            element.currentTime().getTimeValue(), element, query,
20            element.getName() + " provides ressource '" + searchCriteria + "' "
                  );
21        query.addAnswer( element, answer ); // statistics
22        sendMessage( element, query.getInitiator(), answer );
23        return true;
24    }
25    return false;
26 }
27
28 private void forwardQueryToNeighbors( SimulationElement element, Query query,
29        SuperPeer[] neighbors ){
30    boolean original_forwarded = false;    // forward only ONE original
31    for( int i=0; i<neighbors.length; i++ ){
32        if( query.getHistory().historyContainsElement( neighbors[i] ) ){
33            continue; }
34        if( neighbors[i].equals( query.getInitiator() ) ){
35            continue; }
36        if( original_forwarded ){
37            sendMessage( element, neighbors[i], query.getCopy() );
38        }else{
39            sendMessage( element, neighbors[i], query );
40            original_forwarded = true;
41        }
42    }
43 }
```

5 Auswertung der Daten

5.1 Experiment 1 – Test des Suchalgorithmus'

Die Grafiken der Testläufe in der einfachsten Topologie (50 Knoten, 100 Kanten) mit dem kleinsten Anfragenpaket und verschiedenen Werten für "Hops-to-live" (HTL) sowie ausführlichen Debug-Ausgaben auf der Konsole lassen darauf schließen, dass der Suchalgorithmus wie gewünscht funktioniert.

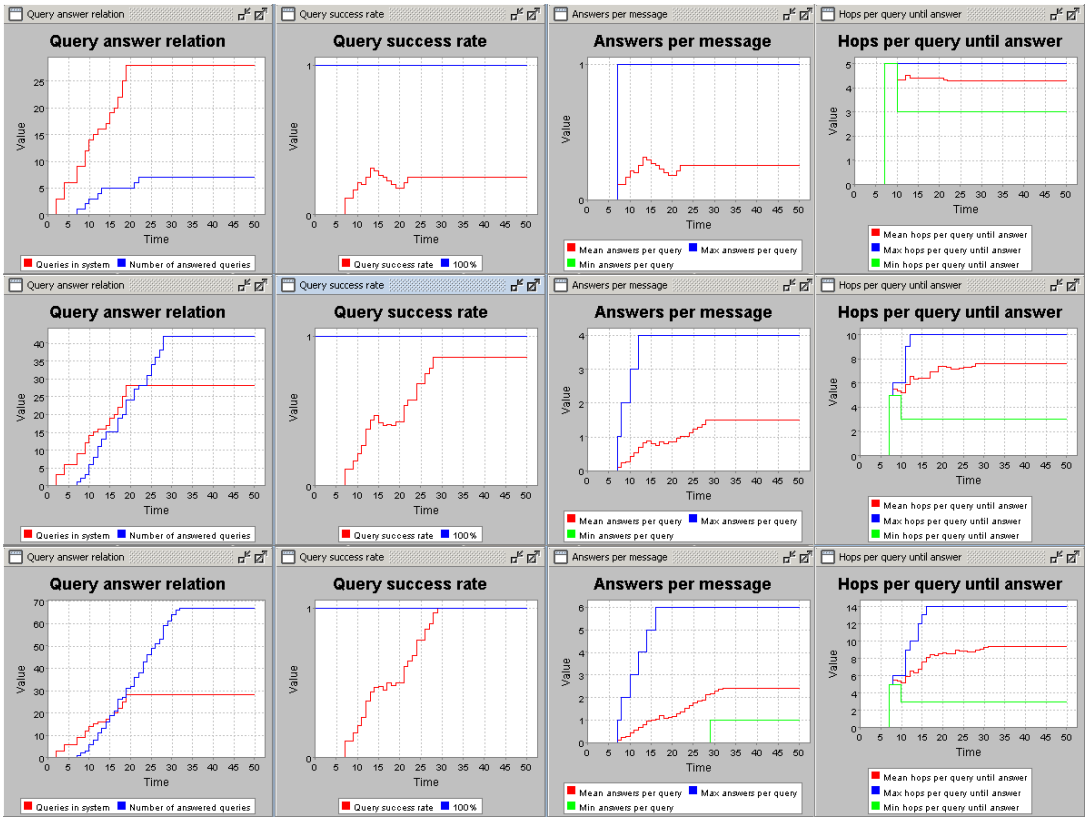

Abbildung 5: Experiment 1 – 3 Durchläufe, 4 Metriken, HTL 5/10/15

Beobachtungen:

- Bei HTL=5 ist das Suchergebnis unbefriedigend – es befinden sich wesentlich weniger Antworten als Anfragen im System.
- Bei HTL=10 werden mehr Antworten erzielt, aber nicht jede Anfrage wird beantwortet.
- Bei HTL=15 erreicht die 'Query success rate' die 100%-Marke, und der Wert 'Max hops per query until answer' bleibt mit 14 knapp unterhalb der HTL.

5.2 Experiment 2 – mittelgroße Topologie

Die Übertragung auf eine wesentlich komplexere Topologie (500 Knoten, 1000 Kanten) bei angepasstem Anfragenpaket (432 Anfragen) ergibt im Prinzip das gleiche.

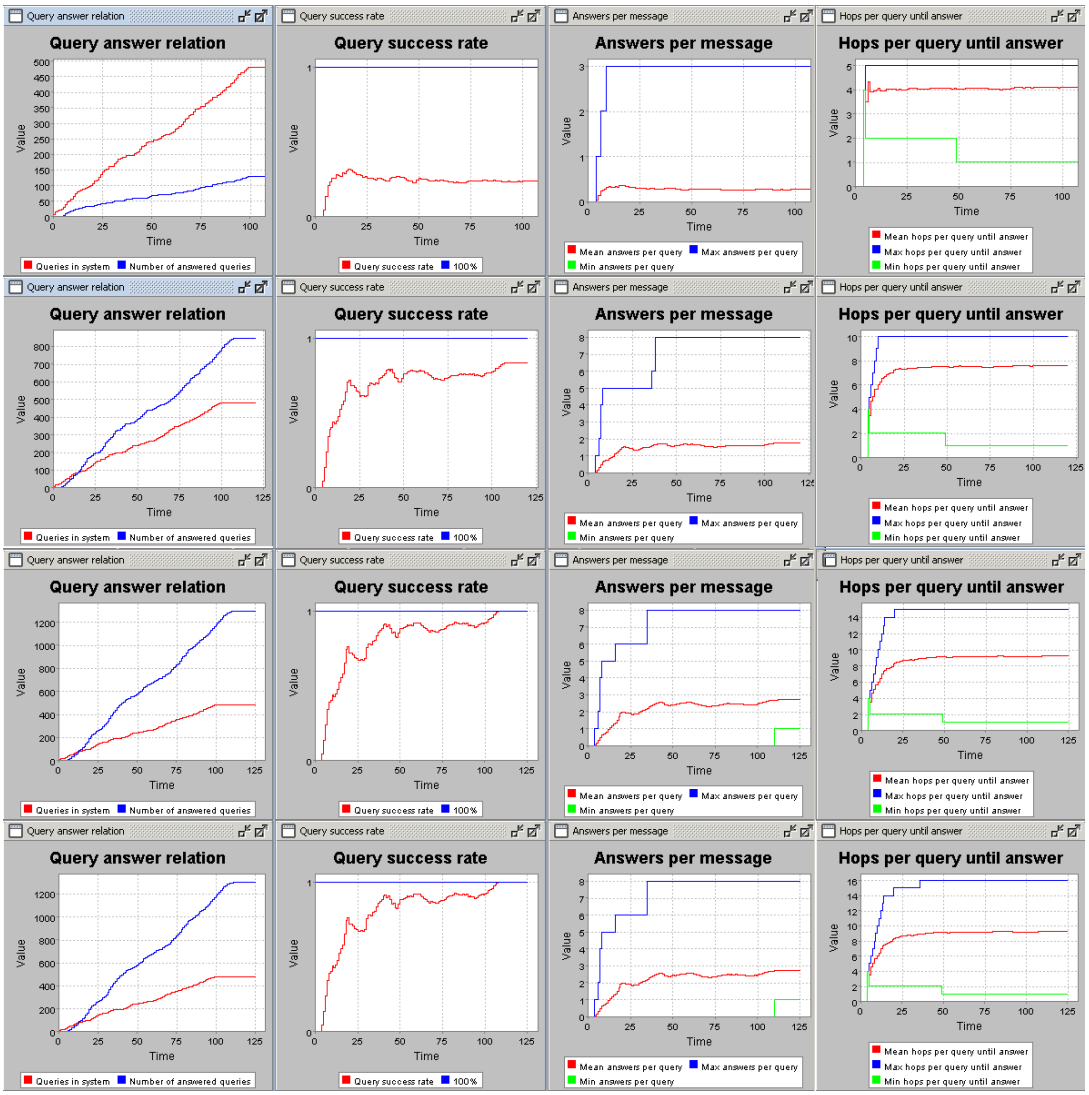

Abbildung 6: Experiment 2 – 4 Durchläufe, 4 Metriken, HTL 5/10/15/20

Beobachtungen:

- Bei HTL=5, 10 und 15 gilt das gleiche wie für Experiment 1: Die Steigerung der HTL wirkt sich positiv auf die Qualität der Suche aus.
- Bei HTL=20 sind nur minimale Unterschiede zu 15 zu erkennen, Werte darüber führen zu exakt identischen Ergebnissen.

5.3 Experiment 3 – kleines Netz

Auch bei der kleineren Topologie (50 Knoten, 100 Kanten) bei 460 Anfragen funktioniert der Suchalgorithmus erwartungsgemäß.

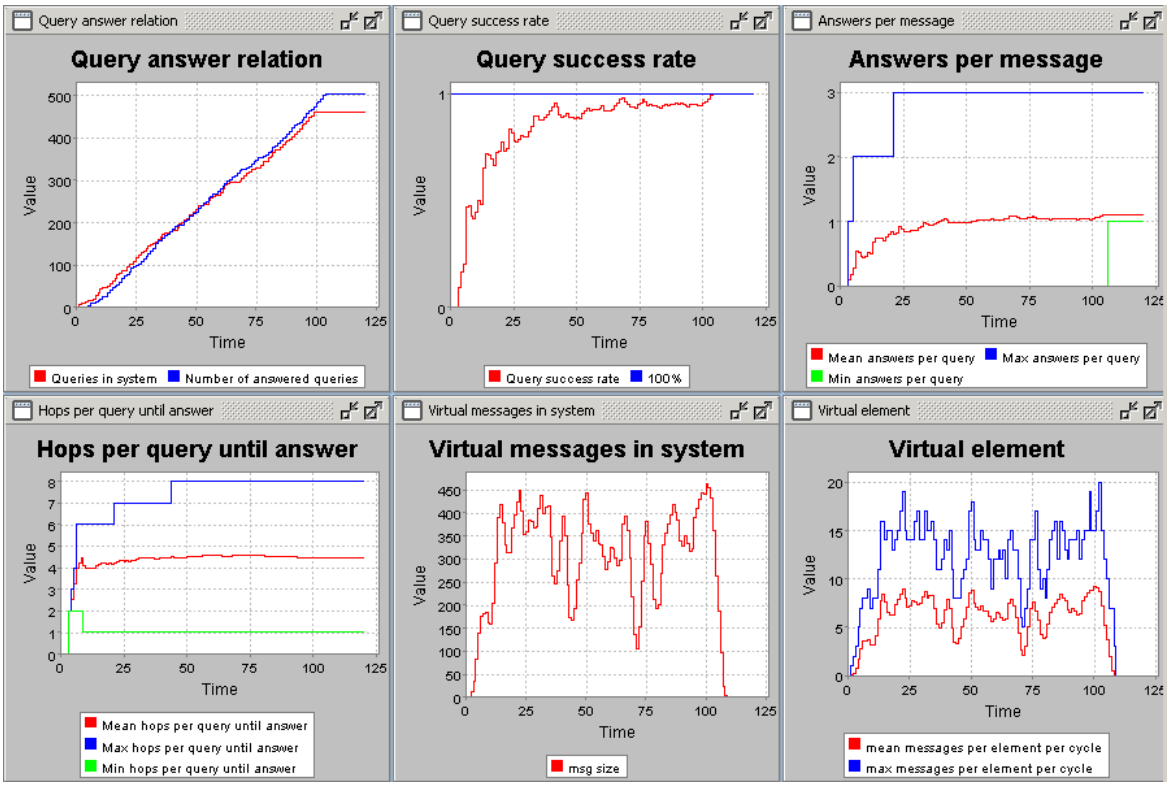

Abbildung 7: Experiment 3 – 1 Durchlauf, 6 Metriken, HTL 15

Beobachtungen:

- Bei HTL=15 wird die 100%-Marke der ‘Query success rate’ gerade so erreicht, während die beiden Werte ‘Query answer relation’ fast parallel und linear verlaufen.

5.4 Experiment 4 – schwache Vermaschung

Die weiten Wege bei der lose zusammenhängenden Topologie (500 Knoten, 500 Kanten) machen sich deutlich bemerkbar, indem für akzeptable Resultate signifikant höhere HTL-Werte nötig sind.

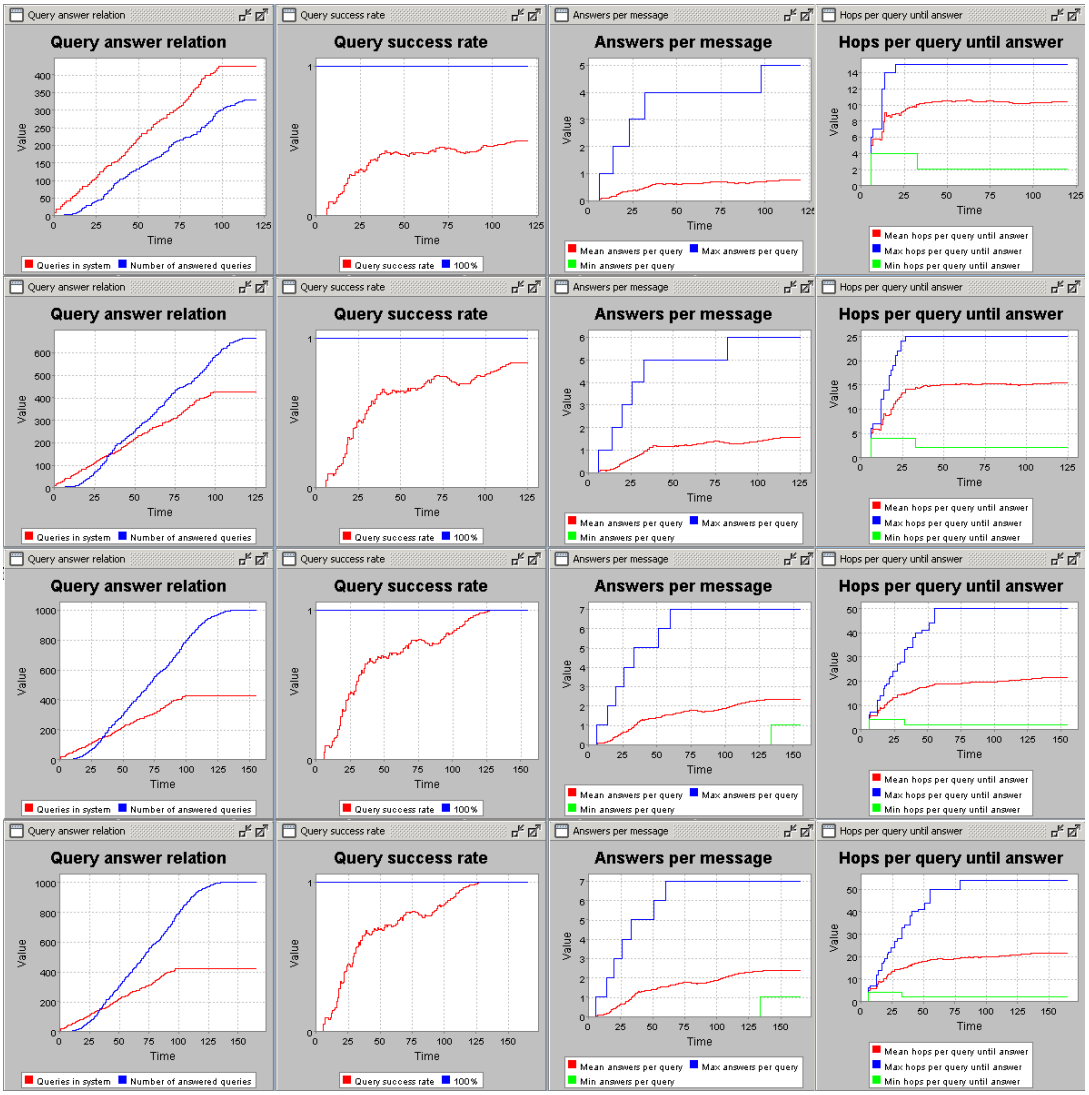

Abbildung 8: Experiment 4 – 4 Durchläufe, 4 Metriken, HTL 15/25/50/60

Beobachtungen:

- Erst bei HTL nahe 50 wird die 100%-Marke erreicht.
- Erst bei HTL nahe 60 ist der Wert ‘gesättigt’, d.h. höhere Werte bringen keinen Vorteil.

5.5 Experiment 5 – großes Netz

Erwartungsgemäß gibt es auch bei dieser Variante (2500 Knoten, 5000 Kanten) mit keine Probleme, lediglich die Gesamtzahl der Nachrichten bzw. deren Verarbeitung in den Peers macht sich ein wenig bei der Dauer der Simulation bemerkbar.

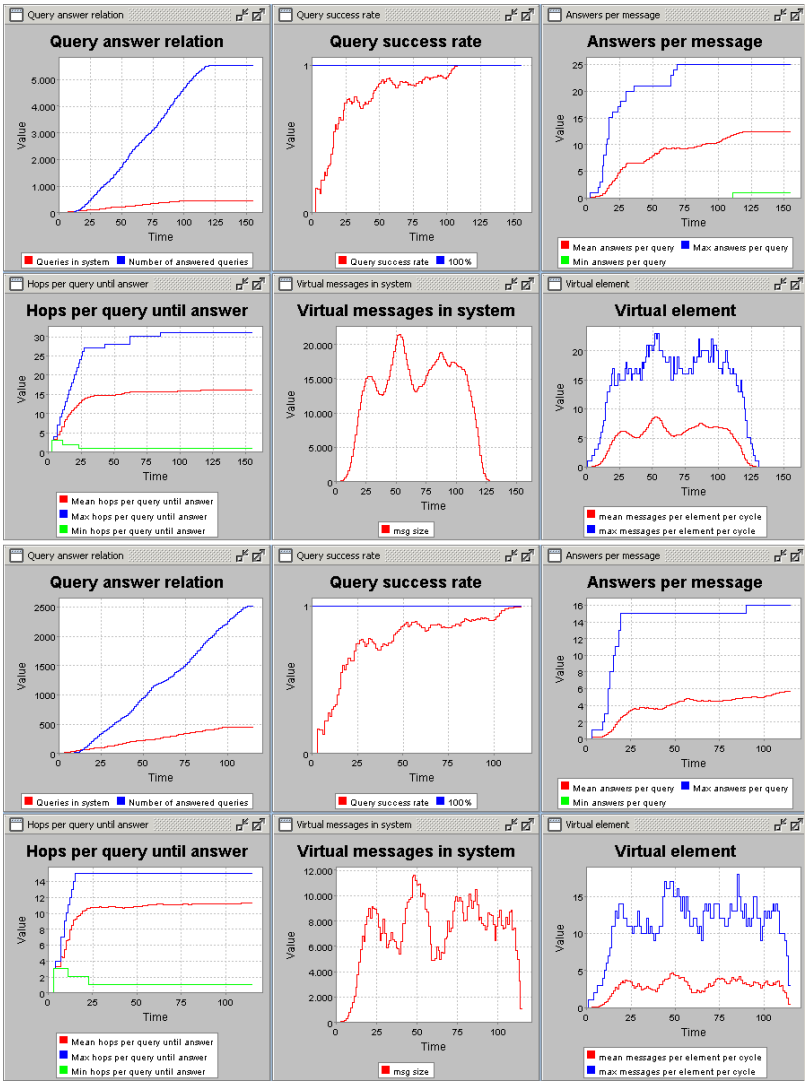

Abbildung 9: Experiment 5 – 2 Durchläufe, 6 Metriken, HTL 15/50

Beobachtungen:

- Der Grenzwert von 15 bei den ‘Max hops per query until answer’ wird schnell erreicht.
- Dennoch (oder deswegen) wird die 100%-Marke der ‘Query success rate’ knapp verfehlt.
- Bei HTL=50 ist zunächst ein fast linearer Anstieg der ‘Max hops per query until answer’ zu verzeichnen, gefolgt von einem sehr flachen Verlauf ab Simulationsschritt 27.
- Der hinreichende Wert für die HTL scheint bei 31 zu liegen.

6 Fazit und Ausblick

Das für mich interessanteste Ergebnis ist, dass es offenbar immer eine obere Schranke gibt, ab der sich eine Erhöhung der HTL nicht mehr positiv auf die Qualität der Suche auswirkt, allerdings auch nicht negativ auf die Performanz des Gesamtsystems, da alle Nachrichten ihr Ziel in weniger als den in der HTL angegebenen Sprüngen erreichen. Leider kann hieraus keine allgemein brauchbare Formel abgeleitet werden, da die Netze in der Praxis überaus unterschiedlich strukturiert sind.

Der Wert von HTL=15 hat sich bei den meisten Topologien als günstiger Startwert herausgestellt, muss jedoch bei schwach vermaschten Netzen aufgrund der größeren 'Entfernungen' (gezählt in Hops) deutlich höher liegen.

Es folgen einige mögliche Erweiterungen des Suchalgorithmus. Alle beziehen sich auf die Optimierung der Netzbelastung, da die maximale Qualität bereits durch eine hohe HTL ausgeschöpft ist.

- Die Entscheidung des Anfragenden, ob nur eine einzelne Ressource gefunden werden soll oder möglichst viele.
- Verallgemeinerung: Die Anzahl der zu findenden Ressourcen sei begrenzt – die Peers fragen regelmäßig (z.B. bei halber HTL) beim Auslöser an, ob eine weitere Suche gewünscht wird.
- Eine Variante, die ich so ähnlich auch irgendwo in [Wil05] gesehen habe: Die HTL wird iterativ erhöht (z.B. verdoppelt), und Knoten, die die gleiche Main-ID der Anfrage noch in der History haben, leiten die Anfrage (mit Sub-ID) einfach durch (Resend).
- Eine Erweiterung, die ich bisher nirgendwo in dieser Form gefunden habe: Jeder Peer sendet regelmäßig eine zufällige Auswahl seiner am häufigsten angefragten Ressourcen an zufällig ausgewählte Nachbarn. Somit wird langfristig ein globaler Index aufgebaut, sodass bei vielen Anfragen Verbindungen mit weniger Hops aufgebaut werden können. Es wird also die Leistung der Suche verbesser, allerdings auf Kosten des (präventiven) Netzverkehrs. Dadurch wäre diese Erweiterung dediziert für schmalbandige Netze, indem der Aufwand für eine Suche über die Zeit verteilt wird. Dabei sollte natürlich darauf geachtet werden, das Netz nicht völlig auszulasten, etwa mittels einer niedrigen Priorität bei Verwendung einer "Quality of Service"-Option.

Literatur

[Wil05] Stefan Willer: Suchverfahren für organisations-orientierte Super-Peer-Architekturen, Diplomarbeit; Universität Oldenburg, 2005.