SOSPNet, 2Fast, et al.

Przegląd zagadnień i problemów w systemach typu peer-to-peer

Karol Strzelecki

Plan prezentacji

- Wprowadzenie
- ‘superwęzły’ – podejście pierwsze
- SOSPNet, czyli podejście „dużo bardziej kompleksowe”
- 2Fast – szybkie ściąganie (danych)

Wprowadzenie

- Popularność P2P zapoczątkowana przez Napstera (1999)
- Główne badania to już przeszłość (2000-2007)
- Rozwój architektury

Wprowadzenie

Czego oczekujemy?

- Wyszukiwanie plików użytkowników

Prosta architektura systemu peer-to-peer?

- Informacje o plikach i użytkownikach partycypujących trzymane centralnie

Lub

- Te same informacje trzymane ‘rozproszenie’

Problemy

- Podejście zcentralizowane – za mało rozproszone (ale szybkie szukanie)
- Podejście rozproszone – za wolne (algorytmy DHT szukają ponoć w log n)

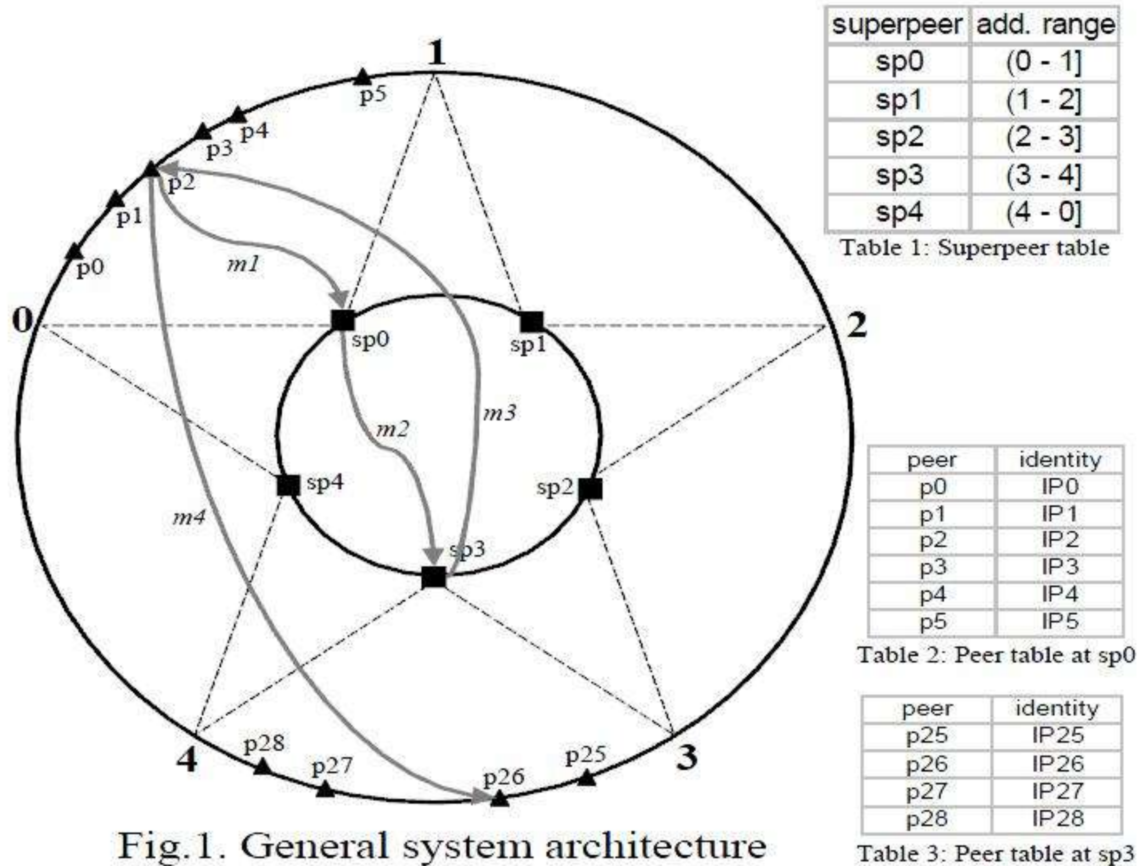
Potrzebujemy rozwiązania, które będzie i szybkie, i bezpieczne

Rozwiązanie?

- Zastosujmy podejście mieszane – trochę rozproszymy, trochę zcentralizujemy
- Leitmotiv – podział węzłów na 2 grupy odpowiedzialności
- Pewien pomysł podany w pracy

„Structured Superpeers: Leveraging Heterogeneity to Provide Constant-Time Lookup”, autorzy to Alper Tugay Mızrak, Yuchung Cheng, Vineet Kumar oraz Stefan Savage

Satan



superpeer	add. range
sp0	(0 - 1]
sp1	(1 - 2]
sp2	(2 - 3]
sp3	(3 - 4]
sp4	(4 - 0]

Table 1: Superpeer table

peer	identity
p0	IP0
p1	IP1
p2	IP2
p3	IP3
p4	IP4
p5	IP5

Table 2: Peer table at sp0

peer	identity
p25	IP25
p26	IP26
p27	IP27
p28	IP28

Table 3: Peer table at sp3

Fig.1. General system architecture

Architektura

- 2 rodzaje peer’ów – normalne i super
- Logiczna organizacja – superpeer’y połączone w wewnętrznym okręgu, zwykłe – w zewnętrznym
- Zakłada się, że jest porządek (w dodatku gęsty!)

Architektura cd.

O ‘superpeerze’

- Ma przydzielony łuk
- Inne superpeery i ich „zakresy”
- Zna swoich podwładnych

O ‘peerze’

- Zna swoje miejsce na łuku
- Zna swojego superpeera

Szukanie pliku

Pamiętajmy, że id plików i węzłów zewnętrznego kręgu są uporządkowane

- Peer wysyła żądanie do swego superpeera
- Jeśli żądany plik jest na „dobrym” łuku, to superpeer szybko zwraca odpowiedź
- Jeśli nie – szybko znajduje innego superpeer’a zarządzającego „dobrym” łukiem

Co jeszcze?

Do dopowiedzenia:

- Jak skalować?
- Jak leczyć po awarii?

Niedopowiedziane:

- Jak przydzielać numerki?

Tolerowanie awarii

- Peer’y na zewnętrznym okręgu czasowo odpytują swoich sąsiadów (zabezpieczenie na wypadek, gdy peer opuszczający system jest „niegrzeczny”)
- Gdy taka sytuacja zaistnieje, superpeer jest powiadamiany i aktualizuje swoje metadane

Awarie superpeer'ów

- Każdy superpeer pamięta metadane trzymane przez pewną liczbę swoich sąsiadów
- Odpytują się nawzajem co pewien czas
- W przypadku awarii wstawiany jest nowy węzeł, wybrany z puli „ochotników”
- Następnie trzeba odświeżyć metadane wszystkich członków systemu…

Skalowalność

Propozycja heurezy, która dla zdefiniowanych twardych i miękkich limitów na przepustowość węzła podejmuje odpowiednie akcje.

Np. jeżeli obciążenie przekroczy limit miękki dla 3 sąsiadujących superpeer'ów

Nowy peer musi znać jednego superpeer'a

Ewaluacja

Przeprowadzona za pomocą narzędzia Chord Simulator

Trochę danych liczbowych:

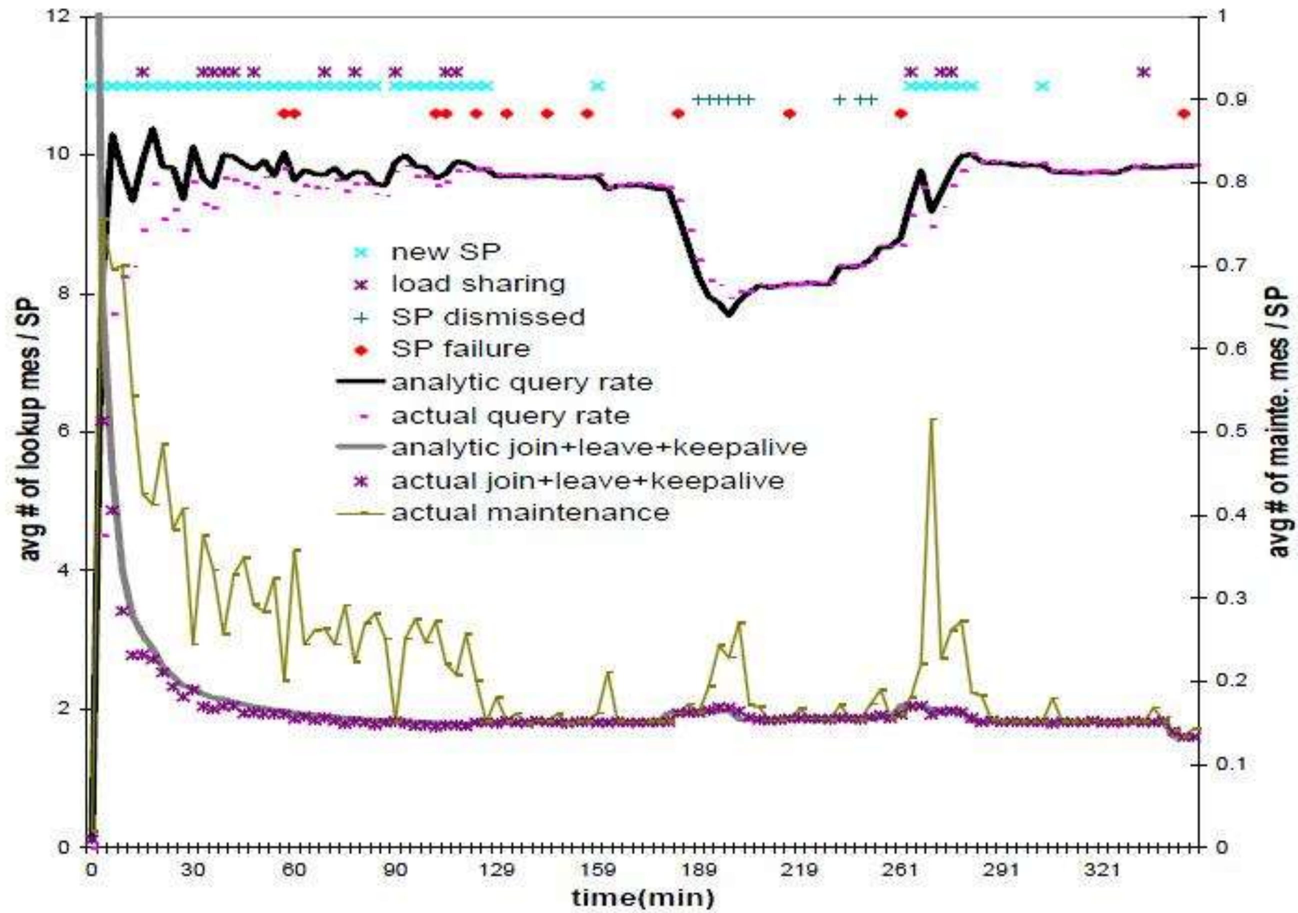
- Prawd. awarii superwęzła – 0.036
- Okres odpytywania (keepalive) – 30 s
- 0.05 zapytania/s

Ewaluacja cd.

Phase	Duration (min)	Join rate (peer/sec)	Leave rate (peer/sec)
1	120	1.5	0
2	60	1.0	1.0
3	20	0.3	3.0
4	60	1.0	1.0
5	20	3.0	0.3
6	60	1.0	1.0
7	10	0	0

Figure 3. Simulation Phases

Wyniki



Wyniki cd.

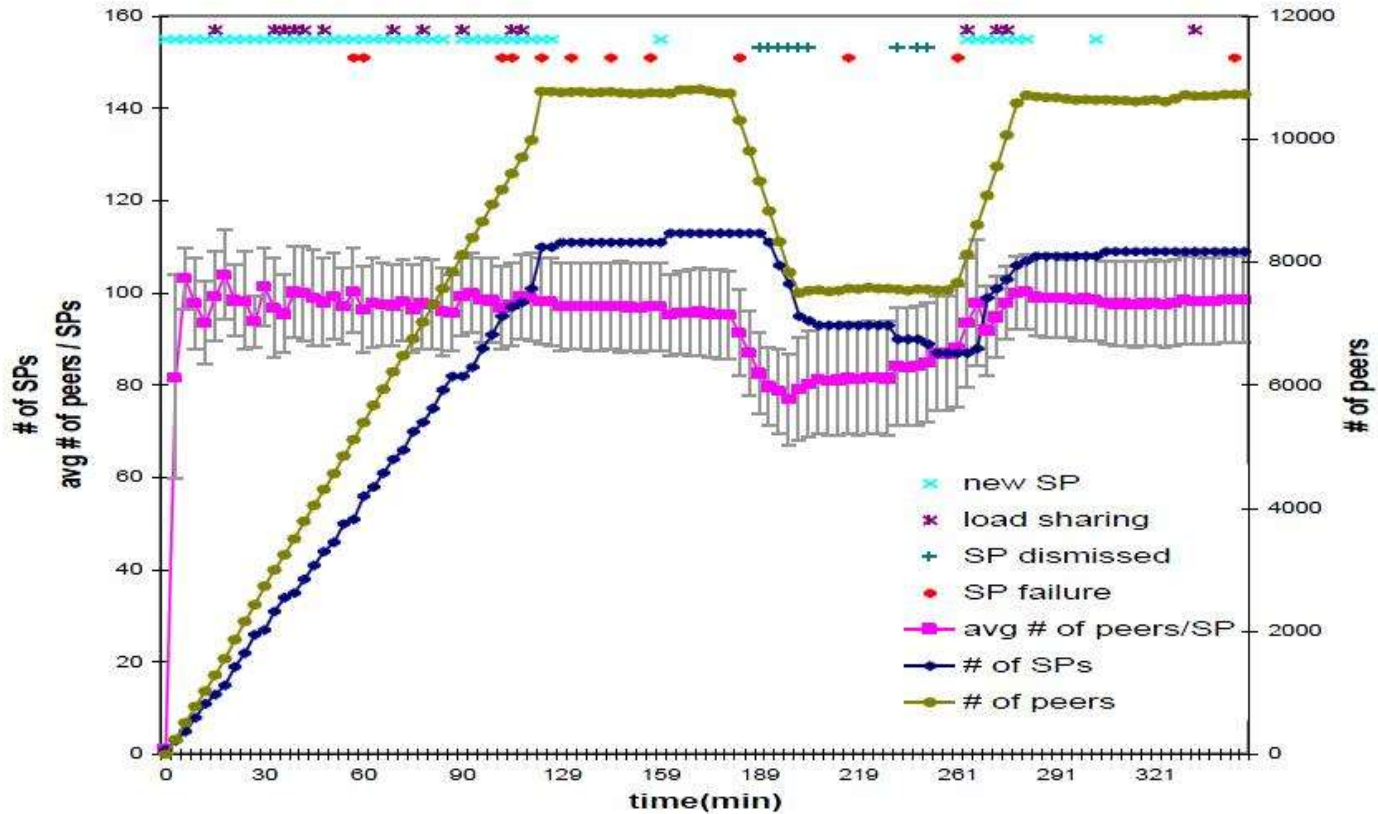


Figure 4. The # of peers, superpeers and average load/superpeer

Wnioski

- Węzły dobrze się organizują
- Komunikaty „keepalive”, itp. nie zajmują zbyt dużo czasu
- Autorzy koncentrują swoją uwagę na zapytaniach o pliki, a nie na ich ściąganiu
- Nie bardzo wiadomo co robić aby przyspieszyć samo pobieranie
- Podobnie w przypadku slashdot’ów

Kolejny krok

SOSPNet – bardziej skomplikowana architektura zaproponowana w pracy „Self-Organizing Super-Peer Networks”, autorzy to Paweł Garbacki, Dick H. J. Epema, i Maarten van Steen

Również zakłada podział na 2 rodzaje węzłów

Założenia SOSPNet'u

- Węzły, które interesują się podobnymi plikami, mają się grupować i wspierać
- Węzeł, który dołącza do sieci powinien szybko zidentyfikować „swoją" grupę
- Elastyczność powiązania peer-superpeer
- Ww. powiązanie powinno być uzasadnione oraz „modyfikowalne"

Architektura - schemat

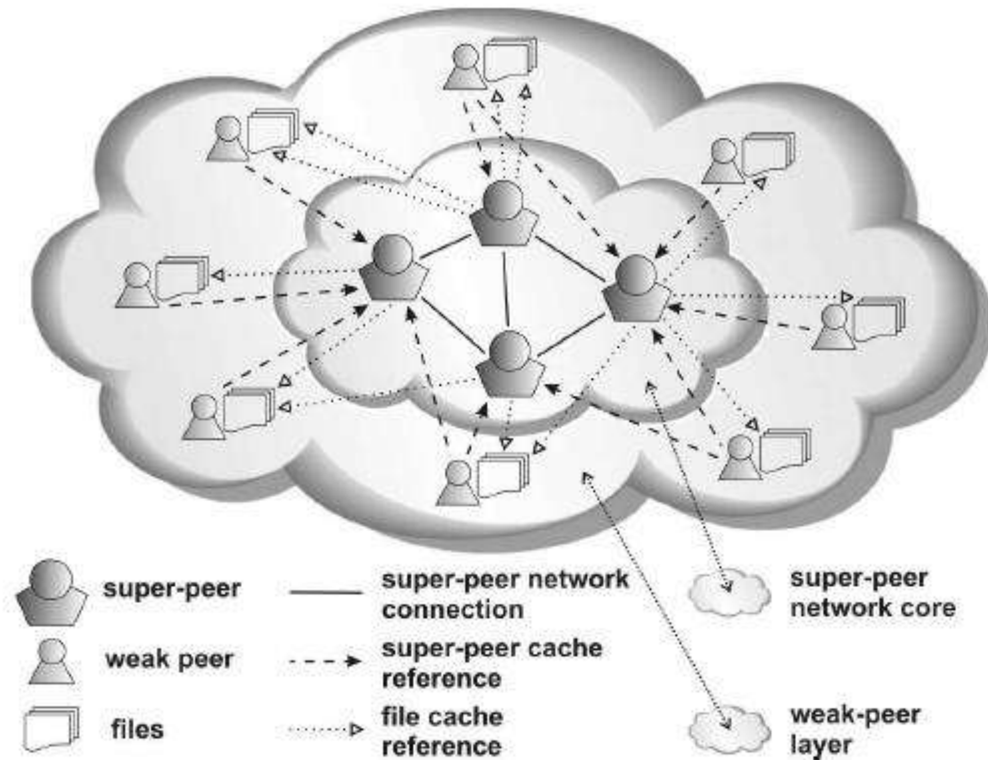


Fig. 1
THE STRUCTURE OF SOSPNET.

Architektura

- „słabe” węzły mają w pamięci podręcznej referencje do superwęzłów wraz z priorytetem
- Superwęzły – referencje do plików w słabych węzłach, również z priorytetem
- Zakładamy, że superwęzłom możemy statycznie przypisać wydajność oraz badać ich obciążenie

Dwupoziomowa pamięć podręczna

- Priorytet uchwytu do superpeer'a jest zwiększany przy komunikacji zakończonej sukcesem
- Pamięcią podręczną weakpeer'a zarządza się wg. schematu LFU, co daje kolejkę priorytetową
- Dla superpeer'ów – podejście mieszane: LRU i LFU (nowy element pamięci podręcznej dostaje maksymalny priorytet)

Jak to działa?

Szukanie pliku

- Weakpeer przechodzi swoją kolejkę priorytetową superpeer'ów pytając ich, czy mają dany plik w pamięci podręcznej
- Jeśli szukanie się nie powiedzie, prosi losowego superpeer'a o wyszukanie pliku
- Na końcu aktualizuje priorytety

Odrobina kodu dla ilustracji działania

```
1  peer_search(p : peer, f : file_name):
2      for s in p.S ordered according to decreasing priorities do
3          q ← super-peer_local_search(s, f)
4          if super-peer_local_search succeeded then
5              t ← s
6              break
7      if f was not found until now then
8          s ← super-peer in p.S selected randomly with
   probability proportional to its priority in p.S
9          < q, t > ← super-peer_search(s, f)
10         if super-peer_search did not succeed then
11             return ERROR "File f not found"
12     if p.S contains t then
13         increase the priority of t in p.S
14     else
15         insert t into p.S
16     merge_super-peer_caches(p, q);
17     return q
```

Kod cd.

```
18 super-peer_local_search(s : super-peer, f : file_name):
19     if an entry < f, q > exists in cache s.F then
20         increase the priority of < f, q > in s.F
21         return q
22     else
23         return ERROR "File f not found"

24 super-peer_search(s, f):
25     perform a search in the super-peer network to locate a
   super-peer t which has an entry < f, q > in its cache
26     if search succeeded then
27         insert < f, q > into s.F
28         return < q, t >
29     else
30         return ERROR "File f not found"

31 merge_super-peer_caches(p : peer, q : peer):
32     for s in q.S do
33         if p.S contains s then
34             increase priority of s in p.S
35         else
36             insert s into p.S
```

Jeszcze wstawianie pliku…

- Weakpeer co jakiś czas wysyła listę swoich plików do któregoś (losowego) superpeer’a
- Ww. superpeer jest wybierany losowo tak, jak w kawałkach kodu prezentowanych wyżej

Skalowalność

Wymagania:

- Nie chcemy przekraczać wydajności superpeer'a
- Idealnie sytuacja - gdy obciążenie proporcjonalne do wydajności
- … oraz niski narzut

Ale superpeer nie zna swoich weakpeer'ów

Skalowalność

Pomysł:

- Superpeer może sterować priorytetami!
- Jeśli jest za bardzo przeciążony, odrzuca żądania weakpeer'ów zmniejszając przez to swój priorytet w ich pamięci podręcznej
- Potrzebujemy jeszcze tylko informacji o obciążeniu innych superpeer'ów

Skalowalność – trochę kodu

```
18   super-peer_local_search(s : super-peer, f : file_name):
18.1     r ← random value from range (0, 1)
18.2     if r > s.accepted_load then
18.3         return ERROR "Super-peer s overloaded"
18.4     add request timestamp to request history window s.W
19       if an entry < f, q > exists in cache s.F then
         ...

24   super-peer_search(s, f):
         ...
26       if search succeeded then
26.1         update_accepted_load(s, t)
27           insert < f, q > into s.F
         ...
```

Jeszcze trochę…

```
37 update_accepted_load(s : super-peer, t : super-peer):
38     s.requests ← number of requests in window s.W
39     t.requests ← number of requests in window t.W
40     s.effective_load ← s.requests/s.capacity
41     t.effective_load ← t.requests/t.capacity
42     Δ ← (t.effective_load − s.effective_load)/(t.effective_load
   + s.effective_load)
43     new_accepted_load ← s.accepted_load + Δ
44     if new_accepted_load > 1 then
45         new_accepted_load ← 1
46     if new_accepted_load < 0 then
47         new_accepted_load ← 0
48     s.accepted_load ←
   β · s.accepted_load + (1 − β) · new_accepted_load
```

Zestawienie z poprzednim projektem

- Mniej niedopowiedzeń
- Większa dynamika
- „Sensowne” wiązanie weakpeer’ów z superpeer’ami
- Mniej pracy przy awariach

Ewaluacja

- Dwa zbiory danych – wygenerowany i „prawdziwy”

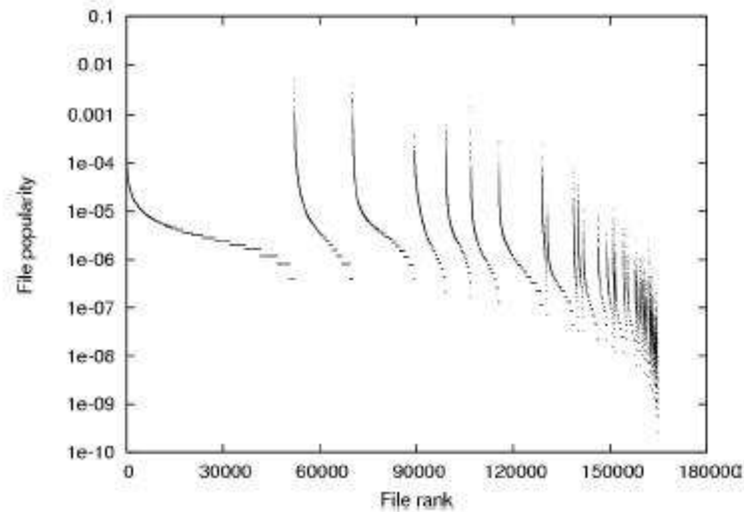
Kilka oznaczeń prawdopodobieństw:

p(m) – losowy węzeł poprosi o jakiś plik z tagiem ‘m’

p(m,k) – losowy węzeł poprosi o konkretny plik ‘k’ z tagiem ‘m’

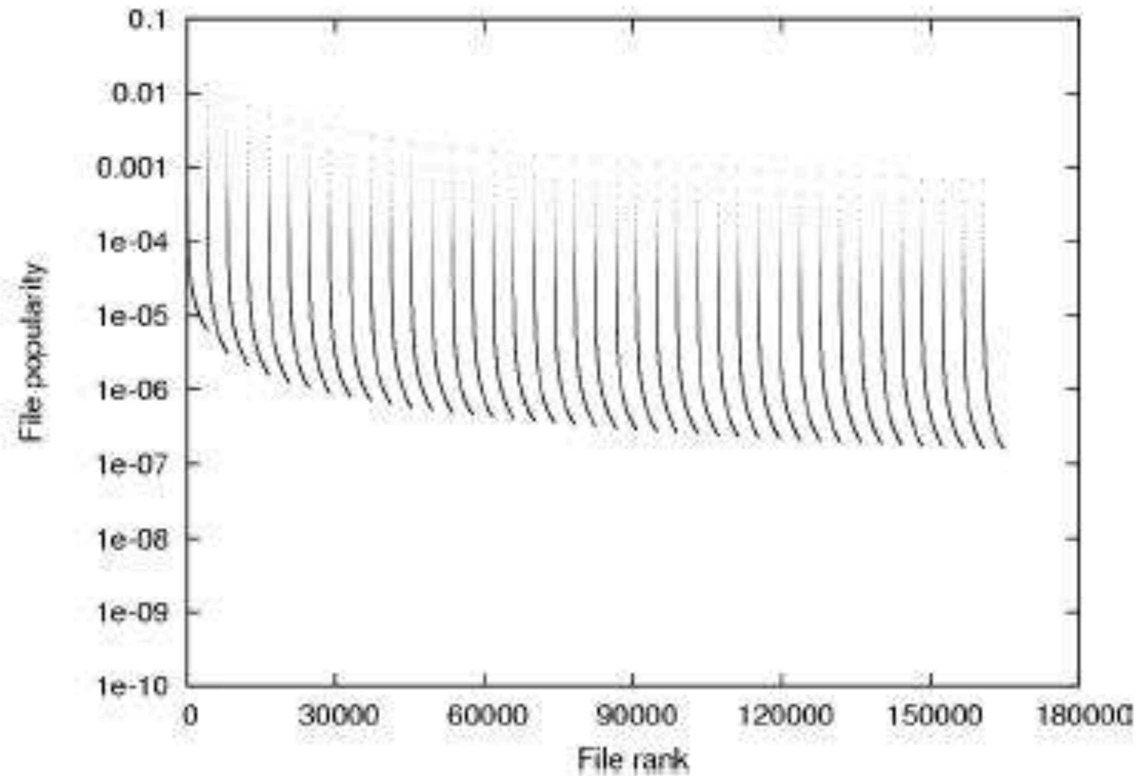
pn(m,k) – jw., ale węzeł też ma swój tag – ‘n’

Wizualizacja danych



(b) piratebay.org

Wizualizacja danych cd.



(d) piratebay_syn

Testy

Znów kilka liczb:

- 100k weakpeer’ów
- Stosunek słabych to silnych 1000:1
- Miejsce w pamięci podręcznej weakpeer’ów – 10
- Superpeer’ów – 1000

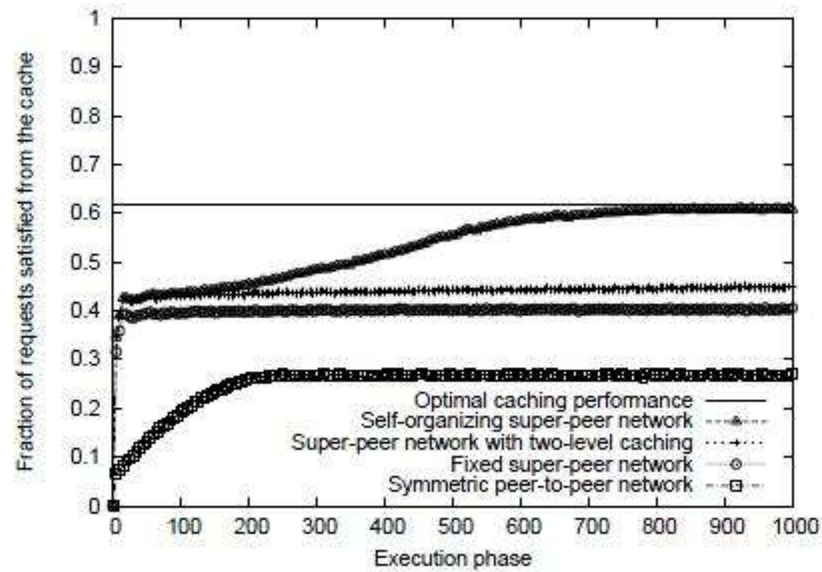
Dodatkowo bierze się pod uwagę preferencje peer’ów

Wydajność pamięci podręcznych

Porównanie z 3 innymi podobnymi systemami:

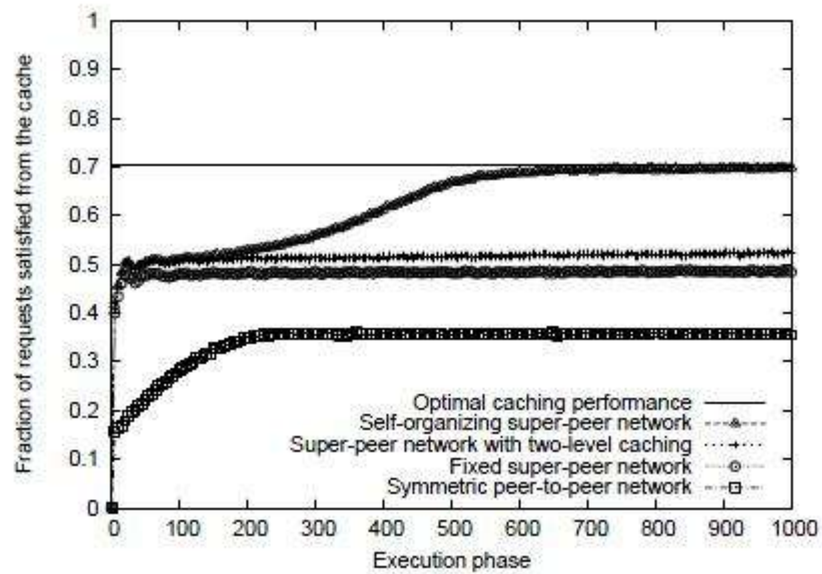
- System bez superwęzłów, z polityką cache'owania jak w SOSPNet
- System ze stałym przyporządkowaniem SP-WP, cache'owaniem jak w SOSPNet
- System jak SOSPNet, ale bez wymiany statystyk między weakpeer'ami

Efekt



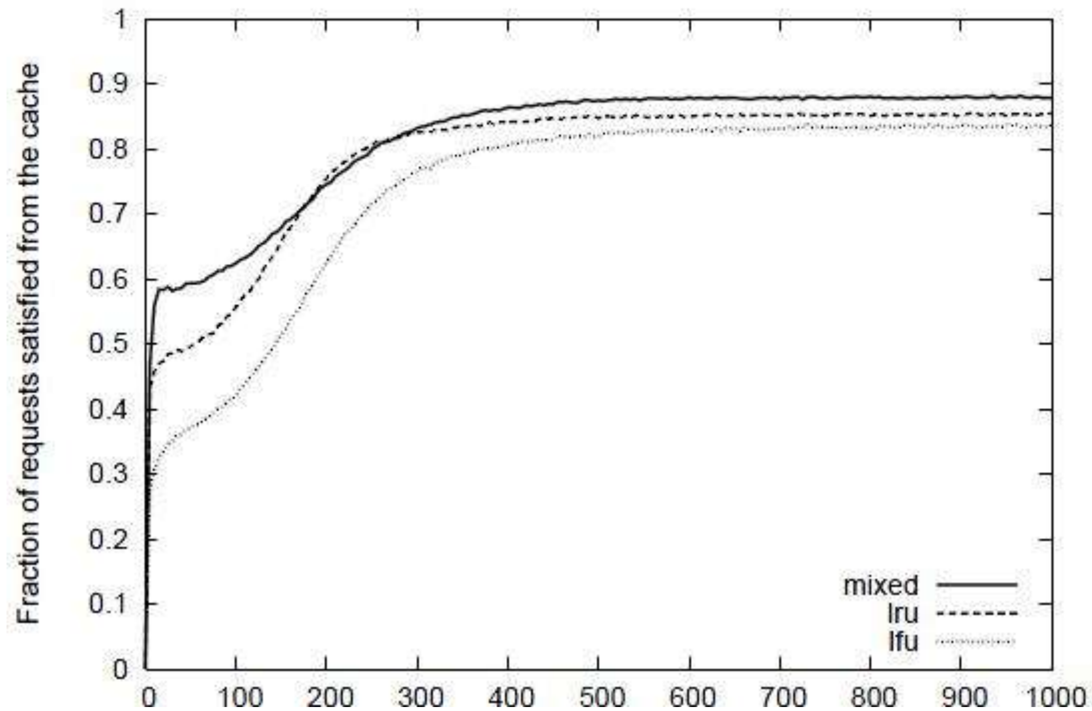
(b) piratebay.org

Efekt cd.

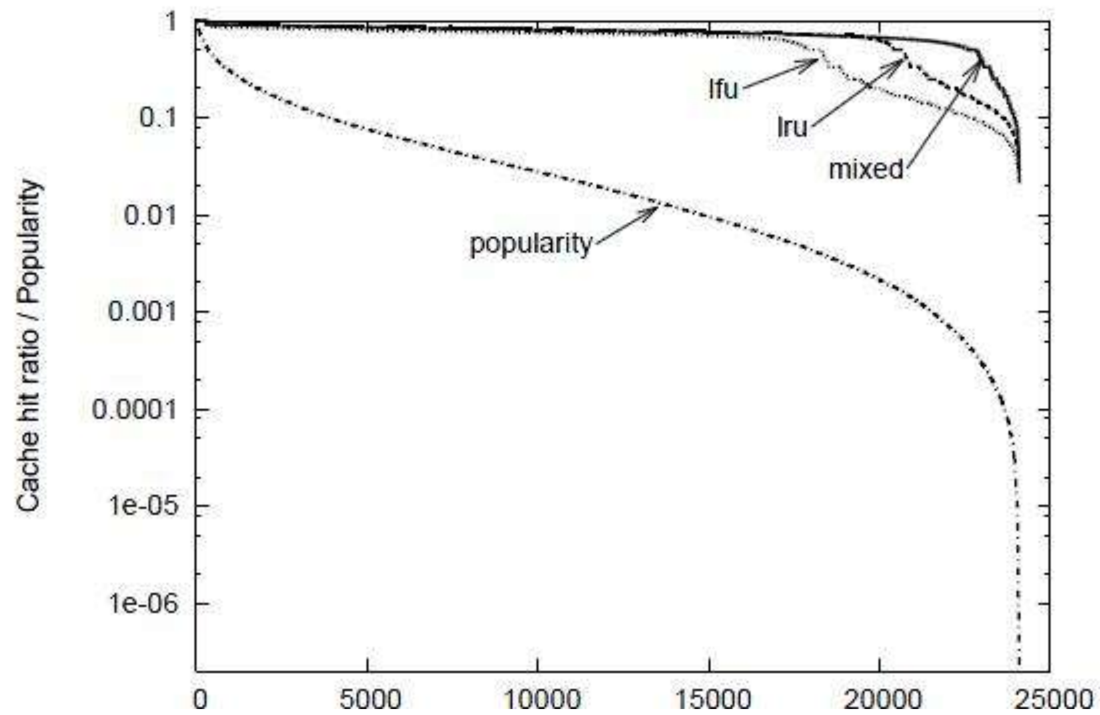


(d) piratebay_syn

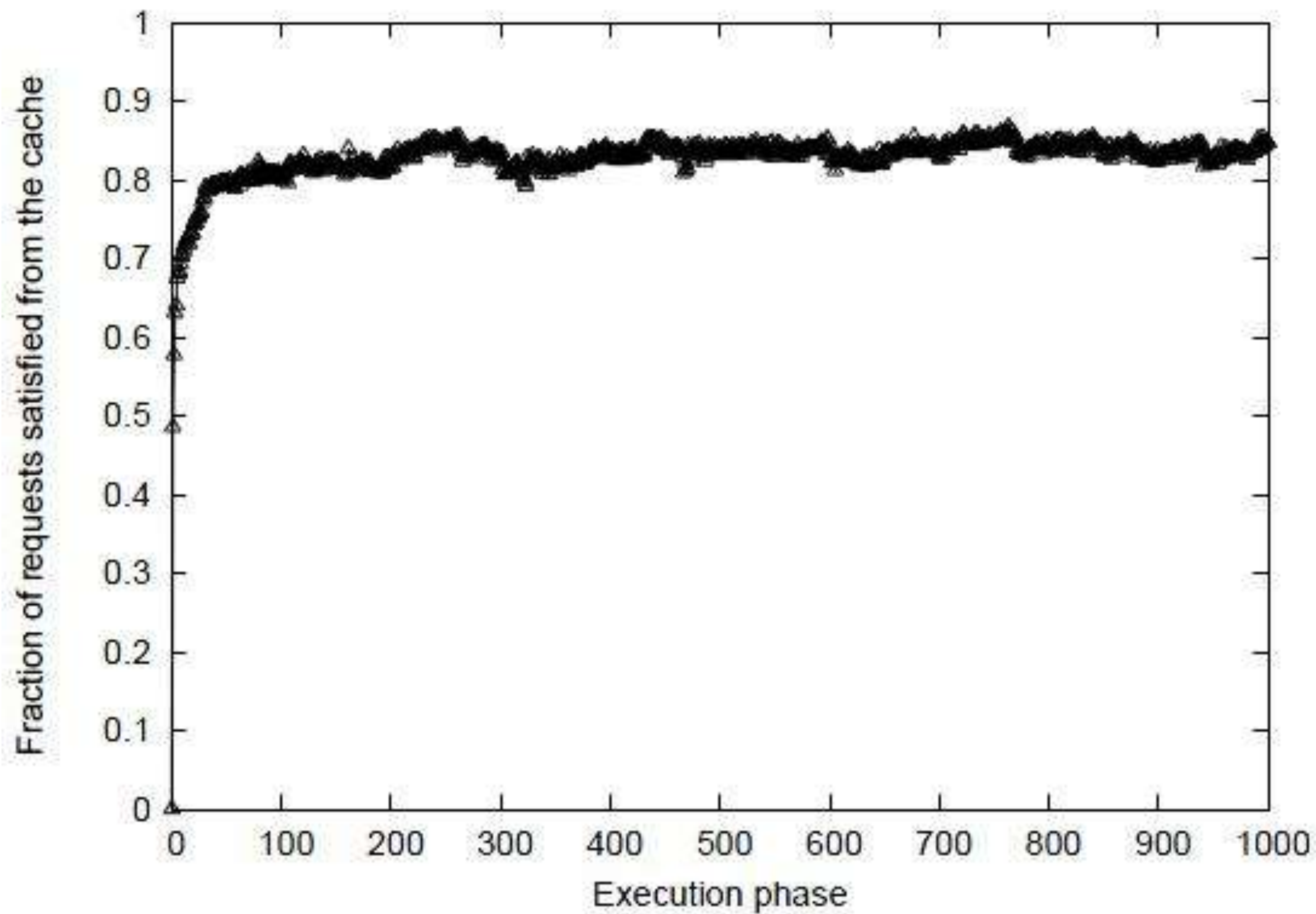
Ewaluacja ‘polityki’ pamięci podręcznej



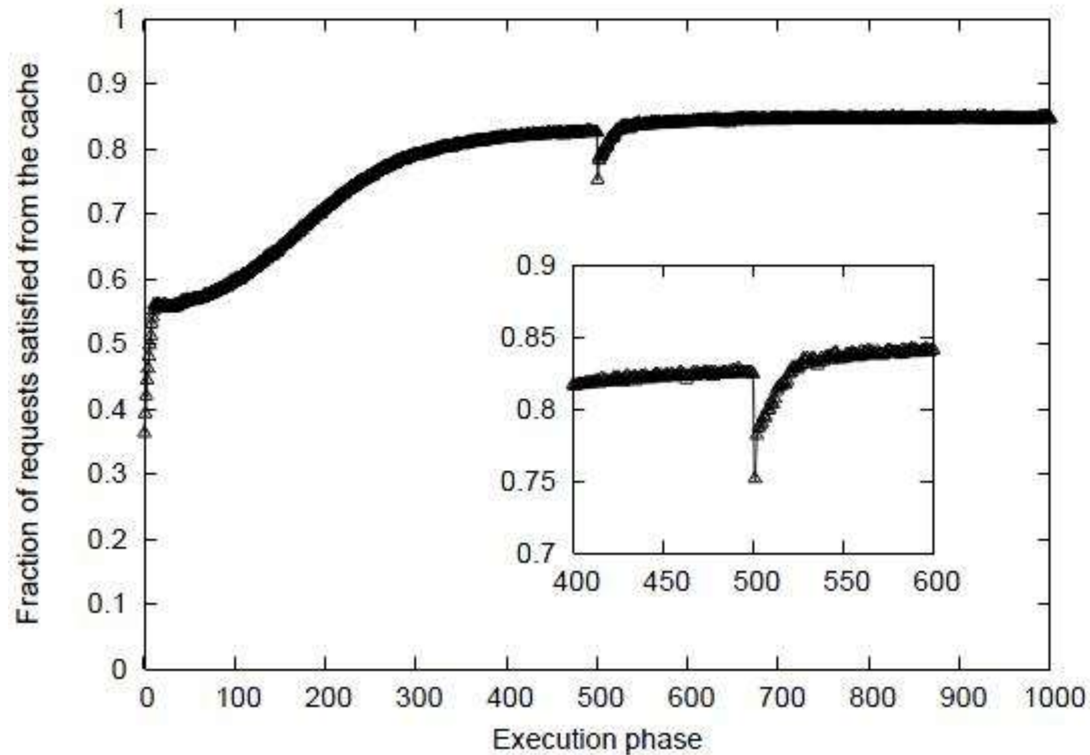
cd.



Dołączanie węzłów



Odporność na awarie



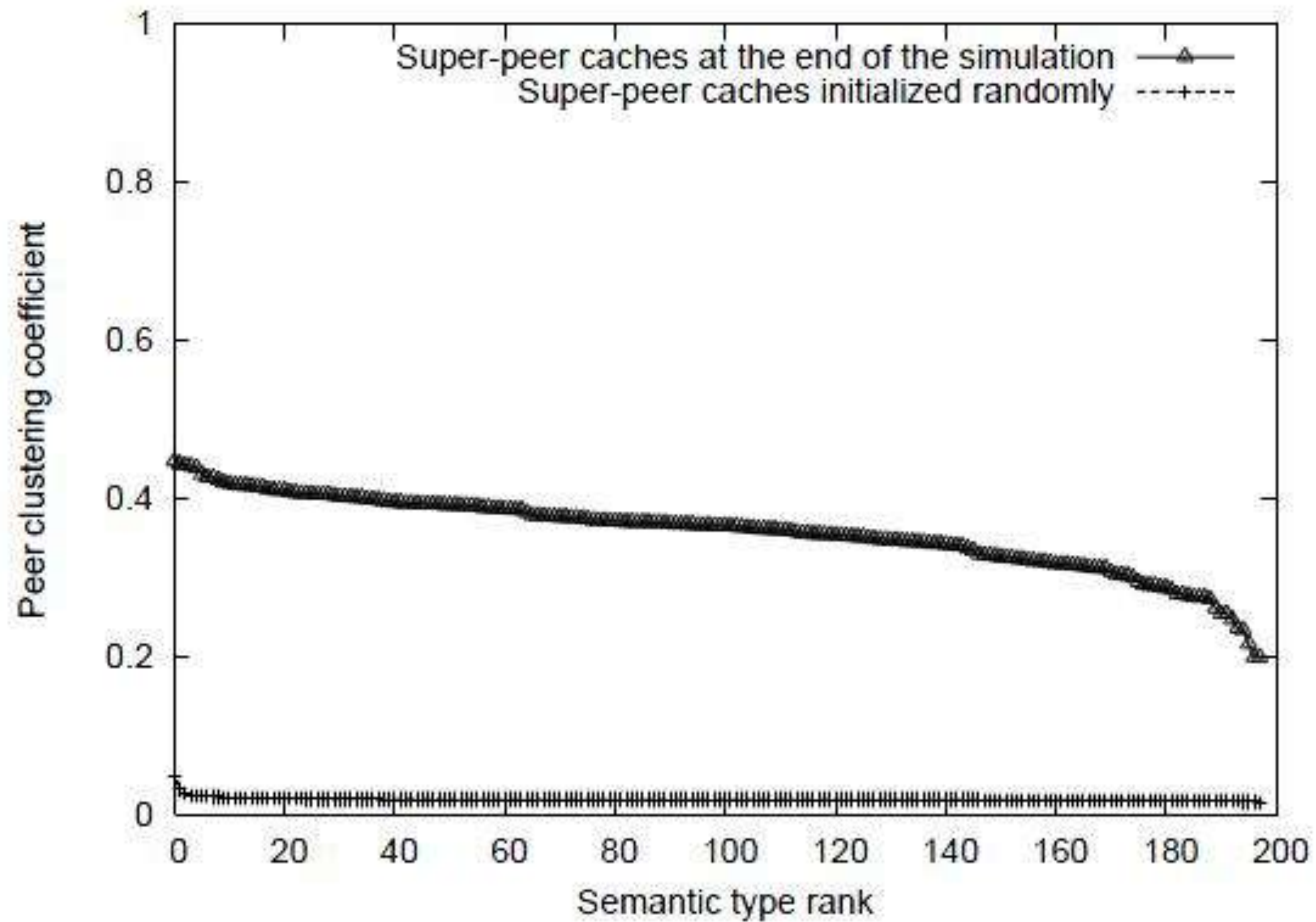
Grupowanie

Przypomnienie: nie tylko pliki, ale również węzły mają tagi.

Teza: węzły o takich samych tagach powinny trzymać w pamięci podręcznej podobne superwęzły.

Miara podobieństwa peer'ów: dla węzłów otagowanych tak samo to średnia liczba takich samych referencji w cache'u / rozmiar cache'u

Grupowanie - wynik



Grupowanie plików

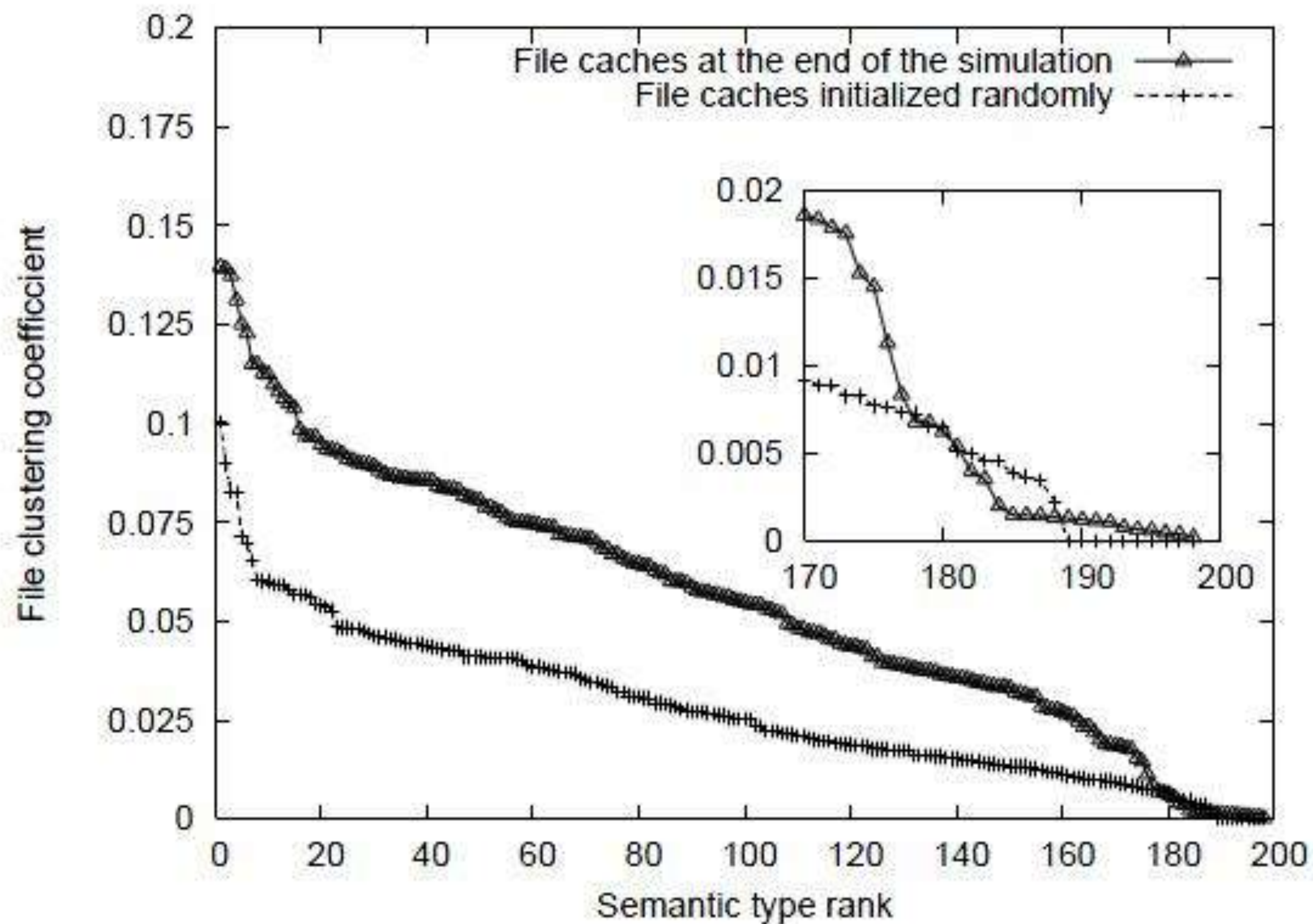
Podobnie superwęzły powinny trzymać w cache'u referencje do podobny plików

Miara podobieństwa: średnia z J(f1,f2) dla każdych dwóch plików otagowanych tak samo

$J(f1,f2) = |Q(f1) \cap Q(f2)|/(|Q(f1)| + |Q(f2)|)$,

Gdzie Q(f) – zbiór superpeer'ów, które mają wskaźnik na plik f

Grupowanie plików - wyniki



Równoważne obciążenie

Super-peer capacity	Number of super-peers	Average effective load	Standard deviation	Cache hit ratio
0.25	242	172.516	30.212	0.843
0.5	252	163.754	20.864	0.835
0.75	243	148.226	19.185	0.847
1	263	159.374	21.549	0.851

Inne problemy

- Wiemy już jak szybko szukać plików
- Ale jak je szybko pobierać?
- Proste podejście – ile wyślesz, tyle odbierzesz ogranicza
- Brak obostrzeń też nie jest dobry

Tragedia wspólnego pastwiska

2Fast (wzmianka)

- Można połączyć użytkowników w grupy przyjaciół
- Nasz przyjaciel użycza trochę transferu
- Pobiera od innych część pliku i udostępnia nam „gratis”
- W zamian my też tak kiedyś zrobimy

Schemat

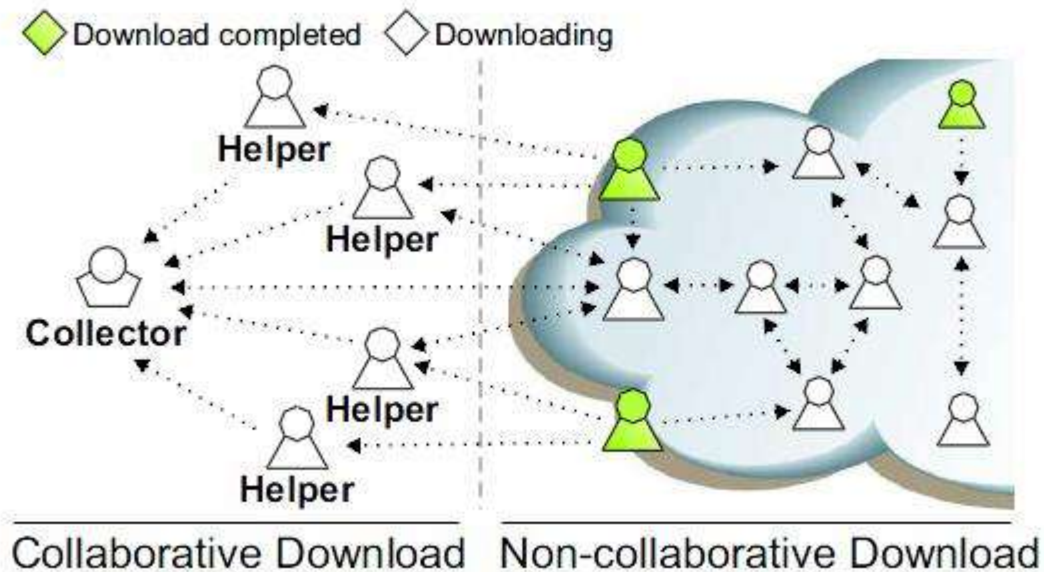


Figure 1. Collector and helpers.

Efekty?

„Socjalne" podejście zapewnia średnio 3.5 wzrost prędkości pobierania

Dziękuję za uwagę

Karol Strzelecki