Peer-to-Peer Netw Appl
DOI 10.1007/s12083-009-0062-6

Decentralising a service-oriented architecture

Jan Sacha · Bartosz Biskupski · Dominik Dahlem · Raymond Cunningham · René Meier · Jim Dowling · Mads Haahr

Received: 7 January 2009 / Accepted: 22 September 2009

Abstract Service-oriented computing is becoming an increasingly popular paradigm for modelling and building distributed systems in open and heterogeneous environments. However, proposed service-oriented architectures are typically based on centralised components, such as service registries or service brokers, that introduce reliability, management, and performance issues. This paper describes an approach to fully decentralise a service-oriented architecture using a self-organising peer-to-peer network maintained by service providers and consumers. The design is based on a gradient peer-to-peer topology, which allows the system to replicate a service registry using a limited number of the most stable and best performing peers. The paper evaluates the proposed approach through extensive simulation experiments and shows that the decentralised registry and the underlying peer-to-peer infrastructure scale to a large number of peers and can successfully manage high peer churn rates.

Keywords Gradient topology · Service-oriented architecture · Super-peer election · Utility · Aggregation

J. Sacha (✉)
Vrije Universiteit, Amsterdam,
The Netherlands
e-mail: jsacha@cs.vu.nl

B. Biskupski · D. Dahlem · R. Cunningham · R. Meier · M. Haahr
Trinity College, Dublin, Ireland

J. Dowling
Swedish Institute of Computer Science, Kista, Sweden

1 Introduction

Service-Oriented Computing (SOC) is a paradigm where software applications are modelled as collections of loosely-coupled, interacting services that communicate using standardised interfaces, data formats, and access protocols. The main advantage of SOC is that it enables interoperability between different software applications running on a variety of platforms and frameworks, potentially across administrative boundaries [1, 2]. Moreover, SOC facilitates software reuse and automatic composition and fosters rapid, low-cost development of distributed applications in decentralised and heterogeneous environments.

A Service Oriented Architecture (SOA) usually consists of three elements: service providers that publish and maintain services, service consumers that use services, and a service registry that allows service discovery by prospective consumers [1, 3]. In many proposed SOAs, the service registry is a centralised component, known to both publishers and consumers, and is often based on the Universal Description Discovery and Integration (UDDI) protocol.[1] Moreover, many existing SOAs rely on other centralised facilities that provide, for example, support for business transactions, service ratings or service certification [3].

However, each centralised component in a SOA constitutes a single point of failure that introduces security and reliability risks, and may limit a system's scalability and performance.

This paper describes an approach to decentralise a service-oriented architecture using a self-organising

[1] http://uddi.xml.org/

Published online: 08 October 2009

Peer-to-Peer (P2P) infrastructure maintained by service providers and consumers. A P2P infrastructure is an application-level overlay network, built on top of the Internet, where nodes share resources and provide services to each other. The main advantages of P2P systems are their very high robustness and scalability, due to inherent decentralisation and redundancy, and the ability to utilise large amounts of resources available on machines connected to the Internet. While the service provision and consumption in a SOA are inherently decentralised, as they are usually based on direct interactions between service providers and consumers, a P2P infrastructure enables the distribution of a service registry, and potentially other SOA facilities, across sites available in the system.

However, the construction of P2P applications poses a number of challenges. Measurements on deployed P2P systems show that the distributions of peer characteristics, such as peer session time, available bandwidth or storage space, are highly skewed, and often heavy-tailed or scale-free [4–7]. A relatively small fraction of peers possess a significant share of the system resources, and a large fraction of peers suffer from poor availability or poor performance. The usage of these low stability or low performance peers for providing system services (e.g., routing messages on behalf of other peers) can lead to a poor performance of the entire system [8]. Furthermore, many distributed algorithms, such as decentralised keyword search [9], become very expensive as the system grows in size due to the required communication overhead.

As a consequence, P2P system designers attempt to find a trade-off between the robustness of fully decentralised P2P systems and the performance advantage and manageability of partially centralised systems. A common approach is to introduce a two-level hierarchy of peers, where so called super-peers maintain system data and provide core functionality, while ordinary peers act as clients to the super-peers.

In this paper, a service registry is distributed between service providers and consumers in the system using a gradient-based peer-to-peer topology. The application of a gradient topology allows the system to place the SOA registry on a limited number of the most reliable and best performing peers in order to improve both the stability of the service registry and the cost of searching using this registry. Furthermore, the gradient topology allows peers to update and optimise the set of registry replicas as the system and its environment evolve, and to limit the number of replica migrations in order to reduce the associated overhead. Analogously, the gradient topology can be used to decentralise additional facilities in a SOA, such as a transaction service or a certificate repository.

The proposed approach has been evaluated in a number of usage scenarios through extensive simulation experiments. Obtained results show that the decentralised registry, and the underlying algorithms that maintain the gradient topology, are scalable and resilient to high peer churn rates and random failures.

The remainder of this paper is organised as follows. Section 2 describes the design of a gradient P2P topology and shows how this topology is used to support a decentralised service registry. Section 3 contains an extensive evaluation of a decentralised service registry built top of a gradient topology. Section 4 describes the review of related work, and Section 5 concludes the paper.

2 Gradient topology

The gradient topology is a P2P overlay topology, where the position of each peer in the system is determined by the peer's utility. The highest utility peers are clustered in the centre of the topology (the so called *core*) while peers with lower utility are found at gradually increasing distance from the centre. Peer utility is a metric that reflects the ability of a peer to contribute resources and maintain system infrastructural services, such as a SOA registry.

The gradient topology has two fundamental properties. Firstly, all peers in the system with utility above a given value are connected and form a gradient subtopology. Such high utility peers can be exploited by the system for providing services to other peers or hosting system data. Secondly, the structure of the topology enables an efficient search algorithm, called *gradient search*, that routes messages from low utility peers towards high utility peers and allows peers to discover services or data in the system. These two properties contribute to an efficient implementation of a decentralised SOA registry.

The SOA registry is distributed between a number of peers in the system for reliability and performance reasons. Hence, there are two types of peers: super-peers that host registry replicas, and ordinary peers that do not maintain any replicas. A utility *threshold* is defined as a criteria for the registry replica placement, i.e., all peers with utility above a selected threshold host replicas of the registry. Finally, gradient search is used by ordinary peers to discover high utility peers that maintain the registry. Figure 1 shows a sample P2P gradient topology, where the

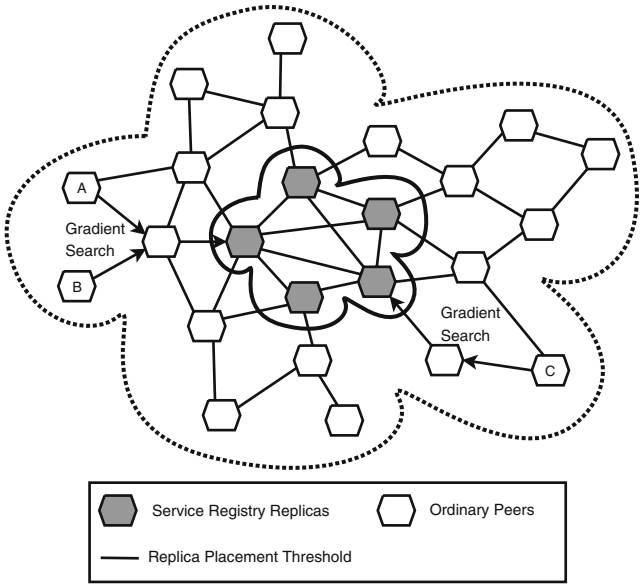

Fig. 1 Registry replication and discovery in the gradient topology. Peers A, B, and C access registry replicas, hosted by peers in the core, using gradient search

service registry is located at the core peers determined by the replica placement threshold.

The following subsections describe in more detail the main components of the gradient topology: utility metrics that capture individual peer capabilities; a neighbour selection algorithm that generates the gradient topology; a super-peer election algorithm for registry replica placement; an aggregation algorithm, required by the super-peer election, that approximates global system properties; a gradient search heuristic that enables the discovery of registry replicas; and finally, the registry replica synchronisation algorithms.

2.1 Characterising peers

In order to determine peers with the most desired characteristics for the maintenance of a decentralised service, such as the SOA registry, a metric is defined that describes the utility of peers in the system. Peer utility, denoted $U(p)$ for peer p, is a function of local peer properties, such as the processing performance, storage space, bandwidth, and availability. Most of these parameters can be measured, or obtained from the operating system, by each peer in a straight-forward way. In the case of dynamically changing parameters, a peer can calculate a running average.

Network characteristics, such as bandwidth, latency, and firewall status, are more challenging to estimate due to the decentralised and complex nature of wide-area networks. Moreover, many network properties, including bandwidth and latency, are properties of pairs of peers, i.e., connections between two peers, rather than individual peers. Nevertheless, a peer can estimate the average latency and bandwidth of all its connections over time and use the average value as a general indication of its network connectivity and overall utility for the system. Furthermore, it has been shown that the bottleneck bandwidth of a connection between a peer and another machine on the Internet is often determined by the upstream bandwidth of the peer's direct link to the Internet [10]. Thus, available bandwidth can be treated as a property of single peers.

Peer stability is amongst the most important peer characteristics, since in typical P2P systems the session times vary by orders of magnitude between peers, and only a relatively small fraction of peers stay in the system for a long time [8]. One way of measuring the peer stability is to estimate the expected peer session duration using the history of previous peer session times. Stutzbach et al. [7] show that subsequent session times of a peer are highly correlated and the duration of a previous peer session is a good estimate for the following session duration. However, the information about previous peer session durations may not always be available, for example for new peers that are joining the system for the first time. Another approach is to estimate the remaining peer session time using the current peer uptime. Stutzbach et al. [7] show that current uptime is on average a good indicator of remaining uptime, although it exhibits high variance. For example, in systems where the peer session times follow the power-law (or Pareto) distribution, the expected remaining session time of a peer is proportional to the current peer uptime. Similar properties can be derived for other session time distributions, such as the Weibull or log-normal distributions, used in P2P system modelling.

Formally, if the peer session times in a system follow the Pareto distribution, the probability that a peer session duration, X, is greater than some value x is given by $P(X > x) = (\frac{m}{x})^k$, where m is the minimum session duration and k is a system constant such that $k > 1$. The expected peer session duration is $E(X) = \mu = \frac{k \cdot m}{k-1}$. However, if a peer's current uptime is u, where $u > m$, the expected session duration follows the Pareto distribution with the minimum value of u, i.e., $P(X > x) = (\frac{u}{x})^k$, and hence, the expected session duration time is $\frac{k \cdot u}{k-1}$. From this we can derive the expected remaining uptime as $\frac{k \cdot u}{k-1} - u = \frac{u}{k-1}$.

2.2 Utility metric properties

The choice of the utility metric has a strong impact on the gradient topology. A utility metric where peers frequently changes their utility values puts more stress on the neighbour selection algorithm and may destabilise the topology. It may also cause frequent switches between super-peers and ordinary peers, which may be expensive and undesired.

However, if peer utility grows or decreases monotonically, peers can cross the super-peer election threshold only once, assuming a constant threshold. Additionally, if the utility changes are predictable, each peer is able to accurately estimate its own utility and the utility of its neighbours at any given time.

For example, if peer p defines its utility as the expected session duration, $Ses(p)$, and estimates it based on the history of its previous sessions, utility $U(p)$ is constant during each peer session. When p is elected a super-peer, it is not demoted to a client unless the super-peer election threshold increases above $U(p)$.

If the utility of p is defined as p's current uptime, denoted $Up(p)$, peer utility increases monotonically with time. Again, when p is elected a super-peer, it is not demoted unless the election threshold rises above $Up(p)$. More importantly, the utility function is fully predictable. Any peer q, at any time t, can compute the utility of p, given q has a knowledge of p's *birth time*, i.e., the time t_p when peer p entered the system. Peer utility is simply equal to

$$U(p) = t - t_p. \tag{1}$$

Clocks do not need to be synchronised between peers, and q can estimate the birth time of p using its own clock. At time t, when q receives the current uptime $Up(p)$ from p, it assumes that $t_p = t - Up(p)$.

For capacity metrics, such as the storage, bandwidth, or processing capacity, there are two general approaches to define peer utility. One approach is to calculate peer utility based on the currently *available* peer capacity. However, this has the drawback that peer utility may change over time, and these changes may be unpredictable to other peers. A better approach is to define peer utility based on the *total* peer capacity, which is usually static. Such utility functions are addressed later in the super-peer election Section 2.4.

Finally, certain algorithms described in this article assume that peer utility values are unique, i.e., $U(p) \neq U(q)$ for any peers $p \neq q$. This property may not hold for some utility definition, particularly if peer utility is based on hardware parameters such as CPU clock speed and amount of RAM. If the utility function is significantly coarse-grained, the construction of a gradient topology may become impossible. In order to address this problem, each peer can add a relatively small random number to its utility value to break the symmetry with other peers.

Table 1 summarises the utility metric properties.

2.3 Generating a gradient peer-to-peer topology

In P2P systems, each peer is connected to a limited number of neighbours and the system topology is determined by the neighbourhood relation between peers.

There are two general approaches to modelling and implementing the neighbourhood relation between peers. In the first approach, a peer stores addresses of its neighbours, which allows the peer to send messages directly to each neighbour, and the neighbourhood relation is asymmetric. This strategy is relatively straightforward to implement, but it has the drawback that peers may store stale addresses of peers that have left the system. This is especially likely in the presence of heavy churn in the system. Moreover, such dangling references can be disseminated between peers unless an additional mechanism is imposed that eliminates them from the system, such as timestamps [11, 12].

In the second approach, the neighbourhood relation between peers is symmetric. This can be simply implemented by maintaining a direct, duplex connection (e.g., TCP) between each pair of neighbouring peers. If a peer is not able to maintain connections with all its neighbours, for example due to the operating system limits, neighbouring peers store the addresses of each other. This has the advantage that peers can notify each other when changing their neighbourhoods or leaving the system, which helps to keep the neighbourhood sets up to date. Furthermore, outdated neighbour entries are not propagated between peers in the system, as each peer verifies a reference received from other peers by establishing a direct connection with each new neighbour. In the case of neighbours crashing, or leaving without notice, broken connections can be detected either by the operating system (e.g., using TCP keep alive protocol) or through periodic polling of neighbours at the application level. In the remaining

Table 1 Utility metric properties

Utility metric	Constant	Monotonic	Predictable
Total capacity	Yes	Constant	Yes
Available capacity	No	No	No
Session length	Yes	Constant	Yes
Uptime	No	Increasing	Yes

part of this paper, it is assumed that the neighbourhood relation between peers is symmetric.

The gradient topology is generated by a periodic neighbour selection algorithm executed at every peer. Periodic neighbour selection algorithms generally perform better than reactive algorithms in heavy churn conditions, as they have bounded communication cost. It has been observed that in systems with reactive neighbour exchange, peers generate bursts of messages in response to local failures, which congest local connections and result in a chain-reaction of other peers sending more messages, which may lead to a major system failure [8].

The structure of the algorithm, shown in Fig. 2, is similar to the T-Man framework [13], however, due to the different neighbourhood models, the two algorithms are not directly comparable.

The algorithm relies on a *preference* function defined for each peer p over its neighbourhood set S_p, such that $maxS_p$ is the most preferred neighbour for p and $minS_p$ is the least preferred neighbour for p. Peer p attempts to connect to a new neighbour when the size of S_p is below the desired neighbourhood set size s^*, and a peer disconnects a neighbour when the size of S_p is above s^*.

New neighbours are obtained through gossipping with high preference neighbours, $maxS_p$ in particular, which is based on the assumption that high preference neighbours of peer p are logically close to each other in the gradient structure. However, greedy selection of $maxS_p$ for gossipping has the drawback that p is likely to obtain the same neighbour candidates from $maxS_p$ in subsequent rounds of the algorithm. The algorithm can potentially achieve better performance if p selects neighbours for gossipping probabilistically with a bias towards higher preference peers.

```
1  if |S_p| > s* then
2  |  disconnect(min(S_p))
3  end
4  else
5  |  n ← max(S_p)
6  |  S' ← (S_n \ S_p) \ {p}
7  |  if |S_p| < s* then
8  |  |  connect(max(S'))
9  |  end
10 |  else
11 |  |  if max(S') > min(S_p) then
12 |  |  |  disconnect(min(S_p))
13 |  |  |  connect(max(S'))
14 |  |  end
15 |  end
16 end
```

Fig. 2 Neighbour selection at peer p

In the gradient topology, a peer p maintains two independent neighbourhood sets: a similarity set S_p and a random set R_p. The similarity set clusters peers with similar utility characteristics and generates the gradient structure of the topology, while the random set decreases the peer's clustering coefficient, significantly reducing the probability of the network partitioning as well as decreasing the network diameter. Random sets are also used by the aggregation algorithm described below.

For static and predictable utility metrics, each peer is able to accurately estimate its neighbours' utility. In case of non-predictable utility metrics, each peer p needs to maintain a cache that contains the most recent utility value, $U_p(q)$, for each neighbour q. Every entry $U_p(q)$ in the cache is associated with a timestamp created by q when the utility of q is calculated. Neighbouring peers exchange and merge their caches every time their neighbour selection algorithms exchange messages, preserving the most recent entries in the caches. Clocks do not need to be synchronised between peers since all utility values for a peer q are timestamped by q.

For the random set, the preference function is uniformly random, i.e., the relationship between any two peers is determined using a pseudo-random number generator each time two peers are compared. The topology generated by such a preference function has small-world properties, including very low diameter, extremely low probability of partitioning, and higher clustering coefficient compared to random graphs. Similar topologies can be generated by other randomised gossip-based neighbour exchange algorithms, such as those described in [11, 12].

For the similarity-based set, the preference function is based on the utility metric U. Peers aim at selecting neighbours with similar but slightly higher utility. Formally, peer p prefers neighbour a over neighbour b, i.e., $a > b$, if and only if

$$U_p(a) > U(p) \;\; and \;\; U_p(b) < U(p) \tag{2}$$

or

$$\left|U_p(a) - U(p)\right| < \left|U_p(b) - U(p)\right| \tag{3}$$

for $U_p(a), U_p(b) > U(p)$ and $U_p(a), U_p(b) < U(p)$. Moreover, peer p selects potential entries to S_p from both S_q and R_q of a neighbour q.

A simpler strategy, where peers prefer neighbours with the closest possible utility, i.e., $a > b$ if $\left|U_p(a) - U(p)\right| < \left|U_p(b) - U(p)\right|$, does not work well

in systems with skewed utility distributions, as it may produce disconnected topologies consisting of clusters of similar utility peers. For example, in systems with heavy-tailed utility distributions, peers do not connect to the few highest utility peers, as they have closer lower-utility neighbours. This problem is alleviated if peers connect to similar, but preferably higher utility, neighbours.

The random set, R_p, never reaches a stable state, as peers constantly add and remove random neighbours. This is desired, since random connections provide a means for the exploration of the system. However, for the similarity sets, S_p, instability or thrashing of connections are harmful as reconfiguring of neighbour connections increases system overhead. Such connection thrashing may occur when p selects q as the best available neighbour, while q consistently disconnects p as a non-desired neighbour. In order to avoid such cases, each peer distinguishes between connections initialised by itself and connections initialised by other peers. In the absence of failure, a peer closes only those connections that it has initialised. By doing so, peers agree on which connections can be closed, improving topology stability.

The performance of the algorithm can be further improved by introducing "age bias" [14]. With this technique, a peer p does not initiate gossip exchange with low-uptime neighbours, because such neighbours have not had enough time to optimise their neighbourhood sets according to the preference function, and therefore are not likely to provide good neighbours for p.

The described neighbour selection algorithm continuously strives to cluster peers with similar utility. However, due to the system scale and dynamism, only the highest utility peers, with sufficiently long life span and high amount of resources, are able to discover globally similar neighbours, while lower utility peers, due to their instability, have mostly random neighbours. As a consequence, a stable core of the highest utility peers emerges in the system, where the connections between peers are stable, and the core is surrounded by a swarm of lower utility peers, where the topology structure is more dynamic and ad-hoc. As shown later in the evaluation section, the neighbour selection algorithm generates a gradient topology in a number of different P2P system configurations.

2.4 Electing super-peers

The super-peer election algorithm, executed locally by each peer in the system, classifies each peer as either a super-peer hosting a registry replica or an ordinary peer that hosts no replicas. The algorithm has the property that it elects super-peers with globally highest utility, and it maintains this highest utility set as the system evolves. Furthermore, the algorithm limits the frequency of switches between ordinary peers and super-peers in order to reduce the associated overhead.

The election algorithm is based on adaptive utility thresholds. Peers periodically calculate a super-peer election threshold, compare it with their own utility, and become super-peers if their utility is above the threshold. Eventually, all peers with utility above the current threshold become super-peers.

The *top-K threshold* is defined as a utility value, t_K, such that the K highest utility peers in the system have their utility above or equal to t_K, while all other peers have utilities below t_K. Given the cumulative peer utility distribution in the system, D, where

$$D(u) = \left|\left\{p \mid U(p) \geq u\right\}\right| \tag{4}$$

the top-K threshold is described by the equation

$$D(t_K) = K. \tag{5}$$

In large-scale dynamic P2P systems, the utility distribution function is not known a priori by peers, as it is a dynamic system property, however, peers can use decentralised aggregation techniques, described in the next section, to continuously approximate the utility distribution by generating utility *histogram*s. The cumulative utility histogram, H, consisting of B bins of width λ can be represented as a B-dimensional vector such that

$$H(i) = \left|\left\{p \mid U(p) \geq i \cdot \lambda\right\}\right| \tag{6}$$

for $i \in \{1, ..., B\}$. The histogram is a discrete approximation of the utility distribution function in B points in the sense that $H(i) = D(i \cdot \lambda)$ for $i \in \{1, ..., B\}$. The top-K threshold can be then estimated using a utility histogram with the following formula

$$t_K = D^{-1}(K) \approx \lambda \cdot \arg\max_{1 \leq i \leq B} \left(H(i) \geq K\right) \tag{7}$$

where the accuracy of the threshold approximation increases with the number of bins in the histogram. The approximation accuracy can be further improved if bin widths in the histogram are non-uniform and are adjusted in such a way that bins closest to the threshold are narrow while bins farther from the threshold are gradually wider.

A top-K threshold allows a precise restriction on the number of super-peers in a dynamic system, where peers are continuously joining and leaving, since it has the property that exactly K peers in the system are above this threshold. Similarly, a *proportional threshold*

is defined as a utility value, t_Q, such that a fixed fraction Q of peers in the system have utility values greater than or equal to t_Q and all other peers have utility lower than t_Q. In a system with N peers, this is described by the following equation

$$D(t_Q) = Q \cdot N. \tag{8}$$

The proportional threshold can be approximated using a utility histogram, since

$$t_Q = D^{-1}(Q \cdot N) \approx \lambda \cdot \arg\max_{1 \le i \le B} \big(H(i) \ge Q \cdot N\big). \tag{9}$$

where the utility histogram, H, and the number of peers in the system, N, are again determined using the aggregation algorithm.

As the system grows or shrinks in size, the proportional threshold increases or decreases the number of super-peers in the system and the ratio of super-peers to ordinary peers remains constant. As such it is more adaptive than the top-K threshold algorithm. However, setting an appropriate number K, or ratio Q, of super-peers in the system using the top-K threshold or proportional thresholds requires domain-specific or application-specific knowledge about system behaviour. A self-managing approach is preferable where the size of the super-peer set adapts to the current demand or load in the system.

It can be assumed that each peer p has some total capacity $C(p)$, which determines the maximum number of client requests that this peer can handle at a time if elected super-peer, and each peer has a current load, $L(p)$, which represents the number of client requests currently being processed by peer p, where $L(p) < C(p)$. One approach is to define peer utility as a function of the peer's *available* capacity (i.e., $C(p) - L(p)$) and to elect super-peers with maximum available capacity. However, this has the drawback that the utility of super-peers decreases as they receive requests, and increases as they fall below the super-peer election threshold and stop serving requests, which may generate fluctuations of high utility peers in the core. Depending on the application, frequent switches between ordinary peers and super-peers may introduce significant overhead, and may destabilise the overlay.

A better approach is to define the peer utility as a function of the *total* peer capacity, $C(p)$, and to adjust the super-peer election threshold based on the load in the system. This way, peer utility, and hence the system topology, remains stable, while the super-peer set grows and shrinks as the total system load increases and decreases.

The utilisation of peer p is the ratio of peer's current load to the peer's maximum capacity, $\frac{L(p)}{C(p)}$. For a set SP of super-peers in the system, the average super-peer utilisation is given by

$$\frac{\sum_{p \in SP} L(p)}{\sum_{p \in SP} C(p)}. \tag{10}$$

In order to maintain the average super-peer utilisation at a fixed level, W, where $0 \le W \le 1$, and to adapt the number of super-peers to the current load, the *adaptive threshold* t_W is defined such that

$$\frac{\sum_{p} L(p)}{\sum_{U(p) > t_W} C(p)} = W. \tag{11}$$

Peers can estimate the adaptive threshold by approximating the average peer load in the system, L, the total number of peers in the system, N, and the capacity histogram, H^c, defined as

$$H^c(i) = \sum_{U(p) \ge i \cdot \lambda} C(p) \tag{12}$$

where $i \in \{1, ..., B\}$. The total system load is given then by $N \cdot L$, and the adaptive threshold can be estimated using the following formula

$$t_W \approx \lambda \cdot \arg\max_{1 \le i \le B} \left(H^c(i) \ge \frac{N \cdot L}{W} \right). \tag{13}$$

In a dynamic system, the super-peer election threshold constantly changes over time due to peer arrivals and departures, utility changes of individual peers, statistical error in the approximation of system properties, and system load variability. Hence, peers need to periodically recompute the threshold and their own utility in order to update the super-peer set. However, frequent switches between super-peers and ordinary peers increase the system overhead, for example due to data migration and synchronisation between super-peers. In order to avoid peers frequently switching roles between super-peer and ordinary peer, the system uses two separate thresholds for the super-peer election, an *upper* threshold, t_u, and a *lower* threshold, t_l, where $t_u > t_l$ (see Fig. 3). An ordinary peer becomes a super-peer when its utility rises above t_u, while a peer stops to be super-peer when its utility drops below t_l. This way, the system exhibits the property of hysteresis, as peers between the higher and lower utility thresholds do not switch their status, and the minimum utility change required for a peer to switch its status is $\triangle = t_u - t_l$. Figure 4 shows the skeleton of the super-peer election algorithm.

2.5 Estimating system properties

The aggregation algorithm, described in this section, allows peers to estimate global system properties

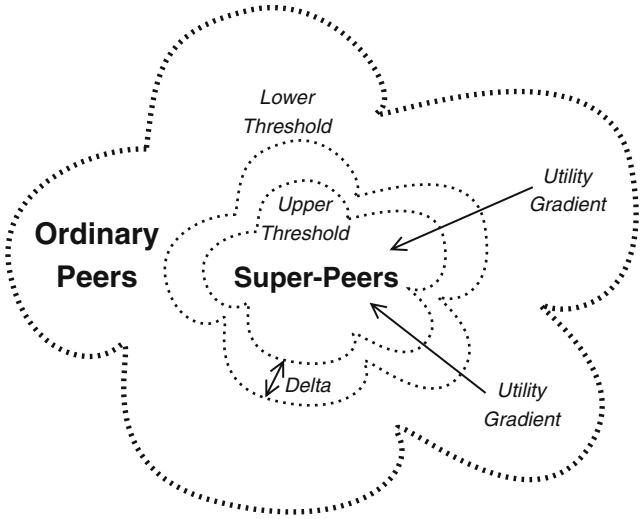

Fig. 3 Super-peer election with two utility thresholds on the gradient topology

required for the calculation of the super-peer election thresholds. The algorithm approximates the current number of peers in the system, N, the maximum peer utility in the system, Max, the average peer load in the system, L, a cumulative utility histogram, H, and a cumulative capacity histogram, H^c. Depending on the super-peer election method, peers may only need a subset of these system properties.

The aggregation algorithm is based on periodic gossipping. Each peer p maintains its own estimates of N, Max, L, H, and H^c, denoted N_p, Max_p, L_p, H_p, and H^c_p, respectively, and stores a set, $\mathscr{T}_p$, that contains the currently executing aggregation instances.

Each peer runs an active and a passive thread, where the active thread initiates one gossip exchange per time step and the passive thread responds to all gossip requests received from neighbours. On average, a peer sends and receives two aggregation messages per time step. When initiating a gossip exchange at each time step, peer p selects a random neighbour, q, and sends $\mathscr{T}_p$ to q. Peer q responds immediately by sending $\mathscr{T}_q$ to p. Upon receiving their sets, both peers merge them using an *update*() operation described later. The general structure of the algorithm is based on Jelasity's push-pull epidemic aggregation [15].

```
1  while true do
2      if super-peer then
3          threshold ← calculateLowerThreshold()
4          if U(p) < threshold then
5              becomeOrdinaryPeer()
6          end
7      end
8      else
9          threshold ← calculateUpperThreshold()
10         if U(p) > threshold then
11             becomeSuperPeer()
12         end
13     end
14 end
```

Fig. 4 Super-peer election algorithm at peer p

The aggregation algorithm can be intuitively explained using the concept of aggregation *instances*. An aggregation instance is a computation that generates a new approximation of N, Max, L, H, and H^c for all peers in the system. Aggregation instances may overlap in time and each instance is associated with a unique identifier *id*. Potentially any peer can start a new aggregation instance by generating a new *id* and creating a new entry in $\mathscr{T}_p$. As the new entry is propagated throughout the system, other peers join the instance by creating corresponding entries with the same *id*. Thus, each entry stored by a peer corresponds to one aggregation instance that this peer is participating in. Eventually, the instance is propagated to all peers in the system. Every instance also has a finite time-to-live, and when an instance ends, all participating peers remove the corresponding entries and generate new approximations of N, Max, L, H, and H^c.

Formally, each entry, T_p, in $\mathscr{T}_p$ of peer p is a tuple consisting of eight values,

$$(id,\ ttl,\ w,\ m,\ l,\ \lambda,\ h,\ h^c) \tag{14}$$

where *id* is the unique aggregation instance identifier, *ttl* is the *time-to-live* for the instance, w is the weight of the tuple (used to estimate N), m is the current estimation of Max, l is the current estimation of L, λ is the histogram width used in this aggregation instance, while h and h^c are two B-dimensional vectors used in the estimation of H and H^c, respectively.

At each time step, each peer p starts a new aggregation instance with probability P_s by creating a local tuple

$$\left(id,\ TTL,\ 1,\ U(p),\ L(p),\ \lambda,\ I_p,\ I^c_p\right) \tag{15}$$

where *id* is chosen randomly, TTL is a system constant, I_p is a utility histogram containing one peer p

$$I_p(i) = \begin{cases} 0 & if\ \ U(p) < i \cdot \lambda \\ 1 & if\ \ U(p) \geq i \cdot \lambda \end{cases} \tag{16}$$

and I^c_p is a capacity histogram initialised by p

$$I^c_p(i) = \begin{cases} 0 & if\ \ U(p) < i \cdot \lambda \\ C(p) & if\ \ U(p) \geq i \cdot \lambda \end{cases}. \tag{17}$$

The bin width λ is set to $\frac{Max_p}{B}$, where B is the number of bins in the histograms H and H^c. Probability P_s is calculated as $\frac{1}{N_p \cdot F}$, where F is a system constant that regulates the frequency of peers' starting aggregation instances. In a stable state, with a steady number of peers in the system, a new aggregation instance is created on average with frequency $\frac{1}{F}$. Furthermore, since an aggregation instance lasts TTL time steps, a peer participates on average in less than $\frac{TTL}{F}$ aggregation instances, and hence, stores less than $\frac{TTL}{F}$ tuples.

As the initial tuple is disseminated by gossipping, peers join the new aggregation instance. It can be shown that in push-pull epidemic protocols, the dissemination speed is super-exponential, and with a very high probability, every peer in the system joins an aggregation instance within just several time steps [15].

The tuple merge procedure, $update(\mathscr{T}_p, \mathscr{T}_q)$, consists of the following steps. First, for each individual tuple $T_q = (id, ttl_q, w_q, m_q, l_q, \lambda_q, h_q, h_q^c) \in \mathscr{T}_q$ received by p from q, if $\mathscr{T}_p$ does not contain a local tuple identified by id, and $ttl_q \geq \frac{TTL}{2}$, peer p creates a local tuple

$$\left(id,\ ttl_q,\ 0,\ U(p),\ L(p),\ \lambda_q,\ I_p,\ I_p^c\right) \tag{18}$$

and adds it to $\mathscr{T}_p$. This way, peer p joins a new aggregation instance id and introduces its own values of $U(p)$, $C(p)$, and $L(p)$ to the computation. However, if $ttl_q < \frac{TTL}{2}$, peer p should not join the aggregation, as there is not enough time before the end of the aggregation instance to disseminate the information about p and to calculate accurate aggregates. This usually happens if p has just joined the P2P overlay and receives an aggregation message that belongs to an already running aggregation instance. In this case, the update operation is aborted by p.

In the next step, for each tuple $T_q = (id, ttl_q, w_q, m_q, l_q, \lambda_q, h_q, h_q^c) \in \mathscr{T}_q$, peer p replaces its own tuple $T_p = (id, ttl_p, w_p, m_p, l_p, \lambda_p, h_p, h_p^c) \in \mathscr{T}_p$ with a new tuple $T_n = (id, ttl_n, w_n, m_n, l_n, \lambda_n, h_n, h_n^c)$ such that

$$ttl_n = \frac{ttl_p + ttl_q}{2} - 1, \quad w_n = \frac{w_p + w_q}{2}, \quad l_n = \frac{l_p + l_q}{2} \tag{19}$$

$m_n = \max(m_p, m_q)$, $\lambda_n = \lambda_p = \lambda_q$, and h_n and h_n^c are new histograms such that

$$h_n(i) = \frac{h_p(i) + h_q(i)}{2}, \quad h_n^c(i) = \frac{h_p^c(i) + h_q^c(i)}{2} \tag{20}$$

for each $i \in \{1, ..., B\}$. Thus, peer p merges its local tuples with the tuples received from q, contributing to the aggregate calculation.

Finally, for each tuple $T_p = (id, ttl_p, w_p, m_p, l_p, \lambda_p, h_p, h_p^c) \in \mathscr{T}_p$, such that $ttl_p \leq 0$, peer p removes T_p from $\mathscr{T}_p$ and updates the current estimates in the following way: $N_p = \frac{1}{w_p}$, $Max_p = m_p$, $L_p = l_p$, $\lambda = \lambda_p$, and for each $i \in \{1, ..., B\}$

$$H_p(i) = \frac{h_p(i)}{w_p}, \quad H_p^c(i) = \frac{h_p^c(i)}{w_p}. \tag{21}$$

The algorithm has the following invariant. For each aggregation instance id, the weights of all tuples in the system associated with id sum up to 1, with $\frac{1}{w}$ estimating the number of peers participating in this aggregation instance.

Peers joining the P2P overlay obtain the current values of N, Max, L, H, and H^c from one of their initial neighbours. Peers leaving, if they do not crash, perform a leave procedure that reduces the aggregation error caused by peer departures, where they send all currently stored tuples to a randomly chosen neighbour. The receiving neighbour adds the weights of the received tuples to its own tuples in order to preserve the weight invariant. Similarly as when joining an aggregation instance, peers do not perform the leave procedure for tuples with the time-to-live value below $\frac{TTL}{2}$, as there is not enough time left in the aggregation instance to propagate the weight from these tuples between peers and to obtain accurate aggregation results.

It can be shown, as in [15], that the values N_p, Max_p, L_p, H_p, and H_p^c generated by the algorithm at the end of an aggregation instance at each peer p approximate the true system properties N, Max, L, H, and H^c, with the average error, or variance, decreasing exponentially with TTL.

In order to calculate the super-peer election thresholds, peers need to complete two aggregation instances, which requires $2 \cdot TTL$ time steps. In the first instance, peers estimate the maximum peer utility (Max) and determine the histogram bin width ($\lambda = \frac{Max}{B}$). In the following instance, peers generate utility histograms (H or H^c), estimate the system size (N) and load (L), and calculate appropriate thresholds, as defined in Section 2.4.

2.6 Discovering high utility peers

The gradient structure of the topology allows an efficient search heuristic, called gradient search, that enables the discovery of high utility peers in the system. Gradient search is a multi-hop message passing algorithm, that routes messages from potentially any peer in the system to high utility peers in the core, i.e., peers with utility above the super-peer election threshold.

In gradient search, a peer p greedily forwards each message that it currently holds to its highest utility

neighbour, i.e., to a neighbour q whose utility is equal to

$$\max_{x \in S_p \cup R_p} \big(U_p(x)\big). \tag{22}$$

Thus, messages are forwarded along the utility gradient, as in hill climbing and similar techniques.

Local maxima should not occur in an idealised gradient topology, however, every P2P system is under constant churn and the gradient topology may undergo local perturbations from the idealised structure. In order to prevent message looping in the presence of such local maxima, a list of visited peers is appended to each search message, and a constraint is imposed that forbids message forwarding to previously visited peers.

The algorithm exploits the information contained in the topology for routing messages and achieves a significantly better performance than general-purpose search techniques for unstructured P2P networks, such as flooding or random walking, that require the communication with a large number of peers in the system [16]. Gradient search also reduces message loss rate by preferentially forwarding messages to high utility, and hence more stable, peers.

However, greedy message routing to the highest utility neighbours has the drawback that messages are always forwarded along the same paths, unless the topology changes, which may lead to a significant imbalance between high utility peers in the core. This is especially probable in the presence of "heavy hitters", i.e., peers generating large amounts of traffic, as commonly seen in P2P systems [4]. Load balancing can be improved in the gradient topology by randomising the routing, for example, if a peer, p, selects the next-hop destination, q, for a message with probability, $P_p(q)$, given by the Boltzmann exploration formula [17]

$$P_p(q) = \frac{e^{(U_p(q)/Temp)}}{\sum_{i \in S_p \cup R_p} e^{(U_p(i)/Temp)}} \tag{23}$$

where $Temp$ is a parameter of the algorithm called the temperature that determines the "greediness" of the algorithm. Setting $Temp$ close to zero causes the algorithm to be more greedy and deterministic, as in gradient search, while if $Temp$ grows to infinity, all neighbours are selected with equal probability as in random walking. Thus, the temperature enables a trade-off between exploitative (and deterministic) routing of messages towards the core, and random exploration that spreads the load more equally between peers. The impact of the temperature on the performance of Boltzmann search has been studied in [16].

2.7 Supporting the decentralised registry service

The registry stores information about services available in the system. For each registered service, it stores a record that consists of the service address, text description, interface, attributes, etc. The registry allows each peer to register a new service, update a service record, delete a record, and search for records that satisfy certain criteria. Each record can be updated or deleted only by its *owner*, that is the peer that created it.

For fault-tolerance and performance reasons, the registry service is replicated between a limited number of high-utility super-peers. Each peer periodically runs the aggregation algorithm, calculates the super-peer election thresholds, and potentially become a super-peer if needed.

It is assumed that the average size of a service record is relatively small (order of kilobytes), and hence, each super-peer has enough storage space to host a full registry replica, i.e., a copy of all service records. Due to this replication scheme, every super-peer can independently handle any search query without communicating with other super-peers. This is important, since complex search, for example based on attributes, keywords, or range queries, is known to be expensive in distributed systems [9, 18]. It is also assumed that search operations are significantly more frequent than update operations, and hence, the registry is optimised for handling search.

In order to perform a search on the registry, a peer generates a query and routes it using gradient search to the closest super-peer. If the super-peer is heavily-loaded, it may forward the query to another super-peer which has enough capacity to handle it. The super-peer processes the query and returns the search results directly to the originating peer. Optionally, clients may cache super-peer addresses and contact super-peers directly in order to reduce the routing overhead.

In order to create, delete, or update a record in the registry, a peer generates an update request and routes it to the closest super-peer using gradient search. The update is then gradually disseminated to all super-peers using a probabilistic gossip protocol. Every record in the registry is associated with a timestamp of the most recent update operation on this record. The timestamps are issued by the records' owners. Super-peers periodically gossip with each other and synchronise their registry replicas, as in [19]. Each super-peer periodically initiates a replica synchronisation with a randomly chosen super-peer neighbour, and exchanges with this neighbour all updates that it has received since the last time the two super-peers gossipped with each other.

Conflicts between concurrent updates are resolved based on the update timestamps. Every record can be updated only by its owner, and it is assumed that the owner is responsible for assigning consistent timestamps for its own update operations. Moreover, super-peers do not need to maintain a membership list of all replicas in the system. Due to the properties of the gradient topology, all super-peers are located within a connected component, and hence, every super-peer eventually receives every update.

Super-peers are elected using a load-based utility threshold. Each peer defines its capacity as the maximum number of queries it can handle at one time. The load at a peer is defined as the number of queries the peer is currently processing. The super-peer election threshold is calculated in such a way that the super-peers have sufficient capacity to handle all queries issued in the system. When the load in the system grows, new replicas are automatically created.

2.8 Supporting additional SOA facilities

Apart from the service registry, which needs to be present in a service-oriented architecture, many SOAs rely on other infrastructural facilities, such as business transaction services, or ranking systems, that are often implemented in a centralised fashion. This section shows an approach to decentralise such facilities using the gradient topology.

Assuming two applications, A and B, where each application has different peer utility requirements that can be encapsulated in two utility functions, U_A and U_B, respectively, each application defines its utility threshold, t_A and t_B, and the goal of the system is to elect and exploit super-peers p such that either $U_A(p) > t_A$ or $U_B(p) > t_B$.

A naive approach is to generate two independent gradient overlays, using the two utility functions and the algorithms described in the previous sections. However, this would double the system overhead. A better approach is to combine the two utility functions into one general utility function U and to generate one gradient overlay shared by both applications. A convenient way of defining such a common utility function is

$$U(p) = \max\big(U_A(p),\ U_B(p)\big). \tag{24}$$

This has the advantage that both, peers with high value of U_A and peers with high value of U_B, have high utility U, and hence are located in the core and can be discovered using gradient search. The only change required in the routing algorithm is that a search message, once delivered to a high utility peer p in the core, may have to be forwarded to a different peer in the core, since p either has a high value U_A or U_B. This last step, however, with a high probability can be achieved in one hop, since peers in the core are well-connected.

The super-peer election thresholds, t_A and t_B, are estimated using the same aggregation algorithm, where the histograms for both U_A and U_B are generated through the same aggregation instance in order to reduce the algorithm overhead. However, a potential problem may appear if the two utility functions, U_A and U_B, have significantly different value ranges, since the composed utility U may be dominated by one of the utility functions. For example, if U_A has values within range [0..1] and U_B has values in range [1..100], then U is essentially equal to U_B, and searching for peers with high U_A becomes inefficient.

One way to mitigate this problem is to define the two utility functions in such a way that both have the same value ranges, e.g., [0..1]. However, this requires system-wide knowledge about peers. Simple transformations or projections onto a fixed interval, for example using a sigmoid function, do not fix the problem, since if one function has higher values than the other function, the same relation holds when the transformation has been applied. A better approach is to scale one of the two utility functions using the current values of the super-peer election thresholds, for example in the following way

$$U(p) = \max\left(U_A(p),\ \frac{t_A}{t_B}U_B(p)\right). \tag{25}$$

This has the advantage that the core of the gradient topology, determined by the threshold t_A, contains peers with U_A above t_A and peers with U_B above t_B, since if $U(p) > t_A$ for a peer p then either $U_A(p) > t_A$ or $U_B(p) > t_B$.

Similarly, in the general case, where a gradient topology supports more than two applications, all utility functions are scaled by their respective thresholds

$$U(p) = \max\left(U_A(p),\ \frac{t_A}{t_B}U_B(p),\ \frac{t_A}{t_C}U_C(p),\ \ldots\right). \tag{26}$$

This way, all peers required by the higher-level applications (i.e., each peer p such that $U_A(p) > t_A$ or $U_B(p) > t_B$ or $U_C(p) > t_C$ and so on) have utility $U(p)$ above t_A, and can be elected super-peers using the single utility threshold t_A.

Figure 5 shows a sample gradient topology that supports two different applications, A and B. Ordinary

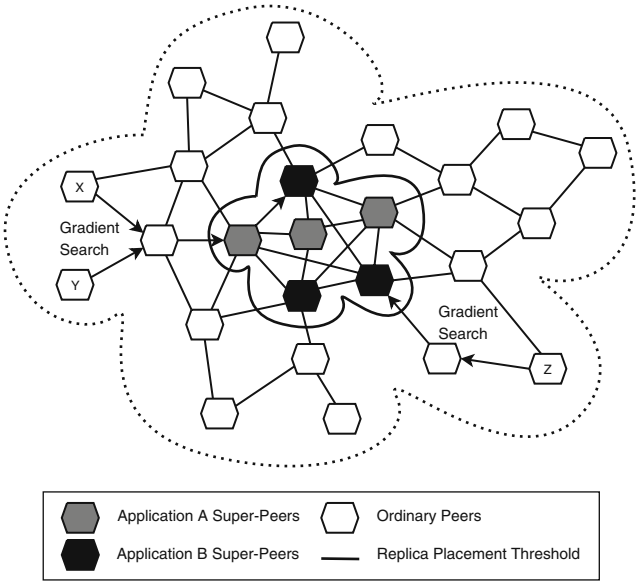

Fig. 5 Super-peer election and discovery in a gradient topology supporting two different applications A and B

peers perform gradient search to discover application *B* super-peers. Peers *X* and *Y* locate an "*A*-type" super-peer in the core and their request is forwarded to a "*B*-type" super-peer. Peer *Z* discovers a "*B*-type" super-peer directly.

2.9 Peer bootstrap

Bootstrap is a process in which a peer obtains an initial configuration in order to join the system. In P2P systems, this primarily involves obtaining addresses of initial neighbours. Once a peer connects to at least one neighbour, it can receive from this neighbour the addresses of other peers in the system as well as other initialisation data, such as the current values of aggregates.

However, initial neighbour discovery is challenging in wide-area networks, such as the Internet, since a broadcast facility is not widely available. In particular, the IP multicast protocol has not been commonly adopted by Internet service providers due to design and deployment difficulties [20]. Most existing P2P systems rely on centralised bootstrap servers that maintain lists of peer addresses.

This section describes a bootstrap procedure that consists of two stages. In the first stage, a peer attempts to obtain initial neighbour addresses from a local cache saved during the previous session, for example on a local disk. This can be very effective; Stutzbach et al. [7] analyse statistical properties of peer session times in a number of deployed P2P systems and show that if a peer caches the addresses of several high-uptime neighbours, there is a high probability that some of these high-uptime neighbours will be on-line during the peer's subsequent session. Furthermore, such a bootstrap strategy is fully decentralised, as it does not require any fixed infrastructure, and it scales with the system size.

However, if all addresses in the cache are unavailable or the cache is empty, for example if the peer is joining the system for the first time, the peer needs to have an alternative bootstrap mechanism. In the second stage, peers obtain initial neighbour addresses from a bootstrap node. The IP addresses of the bootstrap nodes are either hard-coded in the application, or preferably, are obtained by resolving well known domain names. This latter approach allows greater flexibility, as bootstrap nodes can be added or removed over the course of the system's lifetime. Moreover, the domain name may resolve to a number of bootstrap node addresses, for example selected using a round-robin strategy, in order to balance the load between bootstrap nodes.

Each bootstrap node is independent and maintains its own cache containing peer addresses. The cache size and the update strategy are critical in a P2P system, as the bootstrap process may have a strong impact on the system topology, particularly in the case of high churn rates. If the cache is too small, subsequently joining peers have similar initial neighbours, and in consequence, the system topology may become highly clustered or even disconnected. On the other hand, a large cache is more difficult to keep up to date and may contain addresses of peers that have already left the system.

A simple cache update strategy is to add the addresses of currently bootstrapped peers and to remove addresses in a FIFO order. However, this strategy has the drawback that it generates a topology where joining peers are highly connected with each other, which again leads to a highly-clustered topology and system partitioning. A better approach is to continuously "crawl" the P2P network and "harvest" available peer addresses. In this case, the bootstrap node periodically selects a random peer from the cache, obtains the peer's current neighbours, adds their addresses to the cache, and removes the oldest entries in the cache. This has the advantage that the addresses stored in the cache are close to a random sample from all peers in the system.

3 Evaluation

Evaluation is especially important when designing a novel P2P topology, such as the gradient topology,

since P2P systems usually exhibit complex, dynamic behaviour that is difficult to predict a priori. Theoretical system analysis is difficult, and often infeasible in practice, due to the system complexity. At the same time, a full implementation and deployment of a P2P system on a realistic scale requires extremely large amounts of resources, such as machines and users, that are prohibitive in most circumstances. Consequently, the approach followed in this paper is simulation.

However, designing P2P simulations is also challenging. The designer has to decide upon numerous system assumptions and parameters, where the appropriate choices or parameter values are non-trivial to determine. Furthermore, dependencies between different elements of a complex system are often non-linear, and a relatively small change of one parameter may result in a dramatic change in the system behaviour.

Moreover, due to the large scale and complexity, P2P systems are not amenable to visualisation techniques, as a display millions of peers, connections, and messages is not human-readable. P2P simulations must continuously collect and aggregate statistical information about the system in order to, detect topology partitions, identify bottlenecks, measure global system properties, etc. Such frequent and extensive measurements are often computationally expensive, which adds further challenges to analysing P2P systems.

3.1 Evaluation goals

In order to evaluate the gradient topology and its usage in the SOA, the behaviour of the three main algorithms are studied: the neighbour selection algorithm, super-peer election (i.e., registry replica placement), and request routing.

The neighbour selection algorithm is evaluated through an analysis of the generated topology, where the analysed properties include the average peer degree (i.e., number of neighbours), clustering coefficient, average path length in the topology, and the average percentage of globally optimal neighbours in a peer's neighbourhood set. The super-peer election algorithm, and indirectly the aggregation algorithm, are evaluated in a simulation run by measuring the average difference between the desired and the observed numbers of super-peers in the system, the average number of switches between super-peers and ordinary peers, and the total capacity, utilisation and load of super-peers. Finally, the performance of the routing algorithms on the gradient topology is studied by measuring the average request hop count and average failure rate (i.e., percentage of request messages that are lost) in a simulation run.

The algorithms are run in a number of different experiments that examine the impact of relevant system parameters on the system performance, such as the number of peers, churn rate, average load, and super-peer thresholds. The evaluation shows that the gradient topology scales to a large number of peers and is resilient to high peer churn rates.

For the interested reader, a further, more comprehensive evaluation of the gradient topology can be found in [21]. In particular, [21] compares a number of state-of-the-art super-peer election techniques, and shows that the aggregation-based election used in this paper generates higher-quality super-peer sets, according to a number of different metrics, at a similar cost, compared to the other known super-peer election algorithms.

3.2 System model

The gradient topology has been evaluated in a discrete event simulator. The system consists of a set of peers, connections between peers, and messages passed between peers. It is assumed that all peers are mutually reachable and any pair of peers can potentially establish a connection. The neighbourhood model is symmetric, as discussed earlier in Section 2.3. The maximum number of neighbours for a peer at any given time is limited to 26, however, as shown later, peers rarely approach this limit, as the desired number of peer neighbours is set to 13 (7 for the random set and 6 for the similar set).

The P2P network is under constant churn, with peer session times determined probabilistically, following a Pareto distribution. While the paper describes a peer leave procedure, it is hard to estimate how many peers in a real-world system would perform the procedure when leaving. For that reason, the worst case scenario is assumed in the experiments, where no peers perform the leave procedure. Joining peers are bootstrapped by a centralised server, which provides addresses of initial neighbours. The server obtains these addresses by "crawling" the P2P network and maintaining a FIFO buffer with 1,000 entries. The bootstrap server is also used for initiating aggregation instances.

The peer churn rate in the experiments is carefully calculated. In a number of independent measurements [4, 5], median peer session time has been reported as being between 1 minute and 1 hour. A good summary of median session durations in P2P system is given in [8]. In a more recent report [7], mean session times range from about 30 min in Gnutella, through approximately 20 min in Kademlia, to about 2–5 min in BitTorrent. In order to be consistent with these real-world measurements, the mean peer session time in the

experiments in this paper is set to 10 min. Assuming a time step of 6 seconds, this corresponds to a mean session time of 100 time steps and a churn rate of 0.7% peers per time step (0.11% peers per second).

While session time distributions are highly-skewed in existing P2P systems, there is no general consensus whether these distributions are heavy-tailed and which mathematical model best fits the empirically observed peer session times. Sen and Wong [4] observe a heavy-tailed distribution of the peer session time, however, Chu et al. [22] suggest a log-quadratic peer session time distribution, while Stutzbach and Rejaie [7] suggest the Weibull distribution. Moreover, Stutzbach and Rejaie discovered that the best power-law fit for the peer session times in a number of BitTorrent overlays has an exponent whose value is between 2.1 and 2.7, and therefore the distributions are not heavy-tailed. In the experiments in this paper, the peer session times are set according to the Pareto distribution with a median of 10 min and exponent 2.0 (which is border case between heavy-tailed and non-heavy-tailed distributions).

3.3 Service registry simulation

The service registry is maintained by super-peers elected using the adaptive thresholds. The capacity value $C(p)$ determines the maximum number of requests a peer p can simultaneously handle if elected super-peer and hosting a registry replica. The load at peer p, denoted $L(p)$, is defined as the number of requests currently being processed at peer p. The capacity values are assigned to peers according to the Pareto distribution with exponent of 2 and average value of 1, which models peer resource heterogeneity in the system. Moreover, peer utility is defined as

$$U(p) = C(p) \cdot \log\big(Up(p)\big) \tag{27}$$

where the capacity is weighted by the peer's current uptime in order to promote stable peers. As discussed in Section 2.2, this utility metric is fully predictable.

At every step, each peer p in the system emits a search request with probability $P_{req}(p)$. Probability $P_{req}(p)$ follows the Pareto distribution between peers with exponent 2 and average value $P_{req} = 0.01$. Peers that generate more traffic correspond to the so called "heavy hitters" in the P2P system.

Request routing is performed in two stages. First, a newly generated request is routed using Boltzmann search with low temperature $T = 0.5$ steeply to the core until it is delivered to a super-peer. In the second stage, the request is forwarded between super-peers in the core until it is delivered to super-peer s that has enough free capacity to handle the request (i.e., $C(s) - L(s) \geq 1$). Once super-peer s accepts the requests and starts handling it, its load is increased by one. When the request processing finishes, the load at the super-peer is reduced by one.

Forwarding between super-peers is probabilistic. A super-peer p forwards the request to one of its neighbours, q, such that $U_p(q) > t$, where t is the current super-peer election threshold, with probability $P_p^{'}(q)$ proportional to q's capacity

$$P_p^{'}(q) = \frac{C(q)}{\sum_{U_p(x)>t} C(x)}. \tag{28}$$

The bias towards high capacity neighbours improves the load balancing property of the routing algorithm. If no neighbour q exists such that $U_p(q) > t$, the request is routed randomly. Every request has a time-to-live value, initialised to $TTL_{req} = 30$, and decremented each time a request is forwarded between peers. Thus, a request message can be lost when its time-to-live value drops to zero or when the peer that is currently transmitting it leaves the system.

3.4 Maintenance cost

At every time step, each peer executes the neighbour selection, aggregation, super-peer election, and message routing algorithms. A peer sends on average 4 neighbour selection messages per time step (a request and response for S_p and similarly a request and response for R_p) and less than 4 aggregation messages per time step (2 request messages and 2 response messages, since for $F = 25$ and $TTL = 50$ a peer participates on average in less than 2 aggregation instances, as explained in Section 2.5). The election algorithm does not generate any messages. It can be shown that the size of both the neighbour selection and aggregation messages is below 1KB, and therefore, for the basic topology maintenance, a peer sends less than 8KB of data per time step. Given a time step of 6 seconds, this corresponds to an average traffic rate of 1.25KB/s. Moreover, this cost is independent of the system size and the churn rate, since the aggregation and neighbour selection algorithms are executed at a fixed periodicity and always generate the same number of messages per time step. However, the cost associated with gradient search depends on the rate of requests and the size of request messages, and hence is application-specific.

3.5 Topology structure

This section evaluates the neighbour selection algorithm by analysing the generated overlay topology. The evaluation is based on a set of experiments. Each

experiment begins with a network consisting of a single peer, and the network size is increased exponentially by adding a fixed percentage of peers at each time step until the size grows to 100,000 peers. At the following time steps, the system is under continuous peer churn, however, the rate of arrivals is equal to the rate of departures and the system size remains constant.

The following notation and metrics are used. The system topology T is a graph (V, E), where V is the set of peers in the system, and E is the set of edges between peers determined by the neighbourhood sets: $(p, q) \in E$ if $q \in S_p \cup R_p$. The graph is undirected, since the neighbourhood relation is symmetric and if $q \in S_p \cup R_p$ then $p \in S_q \cup R_q$. Similarly, sub-topologies $T_S = (V, E_S)$ and $T_R = (V, E_R)$ are defined based on the similarity and random neighbourhood sets, S_p and R_p, accordingly, where $(p, q) \in E_S$ if $q \in S_p$, and $(p, q) \in E_R$ if $q \in R_p$.

Figure 6 shows the average peer degree distribution in four systems with 100,000 peers and different churn rates, where each plotted point represents the total number of peers in the system with a given neighbourhood size. The graph has been obtained by running four experiments with different churn rates, each for 2,000 time steps, generating peer degree distributions every 40 time steps, and averaging the sample sets at the end of each experiment in order to reduce the statistical noise. The same procedure has been applied to generate all the remaining graphs in this subsection.

The obtained degree distributions resemble a normal distribution, where majority of peers have approximately 13 neighbours, as desired. Moreover, the distributions are nearly identical for all churn rates, suggesting good resilience of the neighbour selection algorithm to peer churn.

V_r is defined as a subset of peers in the system, $V_r \subset V$, that contains r highest utility peers. Formally,

$$V_r = \{p \in V \mid U(p) \geq U(p_r)\} \tag{29}$$

where p_r is the rth highest utility peer in the system. In order to investigate the correlation between peer degree and peer utility, the average peer degree is calculated for a number of V_r sets in T, T_S and T_R. Figure 7 shows the results of this experiment. The plots are nearly flat, indicating that the average number of neighbours is independent from the peer utility, and in particular, the highest utility peers are not overloaded by excessive connections from lower utility peers. The slight increase in the degree of the 12 highest utility peers is caused by the fact that these peers cannot find any higher utility neighbours, and hence, connect to lower utility peers, generating a locally higher average degree.

Similarly, Fig. 8 shows the clustering coefficient in topologies T, T_S and T_R for a number of V_r set with increasing utility rank r. In the T_S topology, the coefficient gradually grows as peer utility increases, almost reaching the value of 0.8 for $r = 1$, which indicates that the highest utility peers in the system are highly connected with each other and constitute a "core" in the network. At the same time, the coefficient is nearly constant in T_R, since the preference function for the random sets is independent of peer utility.

Given global knowledge about the system in a P2P simulator, the optimal neighbourhood set S_p^* for each peer p can be determined using the neighbour preference function defined by formulas (2) and (3) in Section 2.3. Formally, S_p^* is a subset of all peers in the system, $S_p^* \subset V$, such that $min(S_p^*) \geq max(V \backslash S_p^*)$, where the $\geq$ relation is defined in formulas (2) and (3).

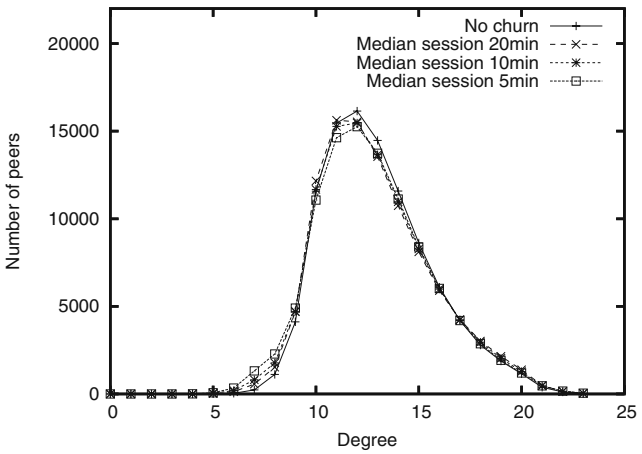

Fig. 6 Peer degree distribution in four systems with different churn rates

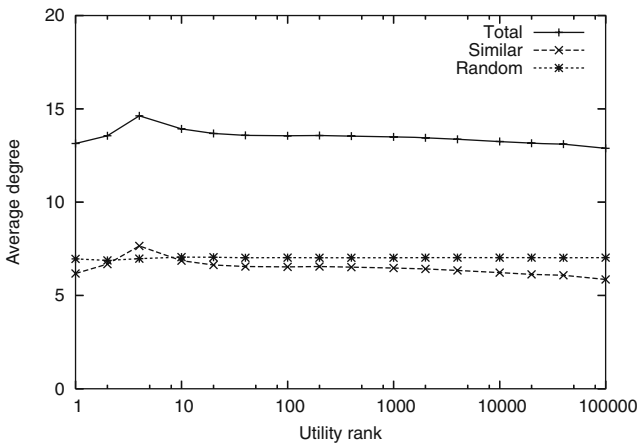

Fig. 7 Average sizes of neighbourhood sets for highest utility peers

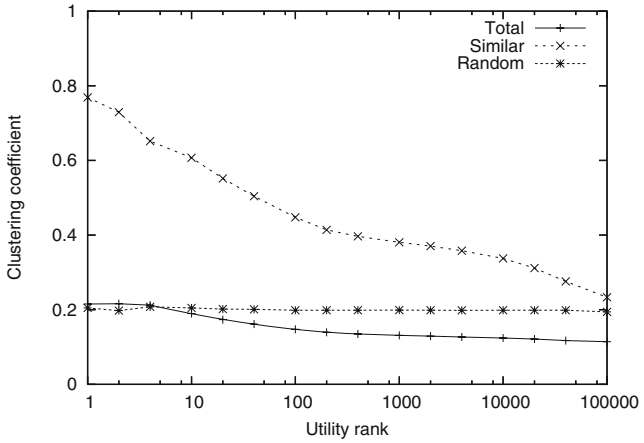

Fig. 8 Clustering coefficients of highest utility peers in sub-topologies determined by similarity and random neighbourhood sets

The quality of a peer neighbourhood set S_p can be then estimated using $Opt(p)$ metric defined as the portion of optimal entries in S_p

$$Opt(p) = \frac{|S_p \cap S_p^*|}{|S_p^*|}. \tag{30}$$

Consequently, $Opt_{avg}(V_r)$ is the average value of $Opt(p)$ for the r highest utility peers in the system.

Figure 9 shows the value of $Opt_{avg}(V_r)$ as a function of the utility rank r in four experiments with different churn rates. The graph shows that while the average value of $Opt()$ in the entire system is very low, as it is relatively unlikely for peers to discover their globally optimal neighbours in a large-scale dynamic system, $Opt(p)$ grows with peer utility and reaches its maximum value of 1 for the highest utility peers. This confirms that the highest utility peers are stable enough

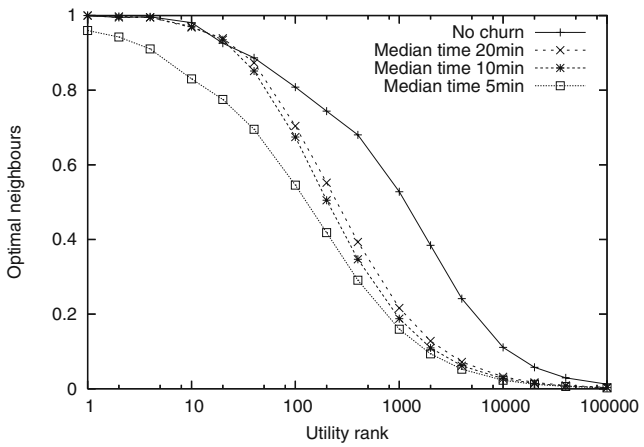

Fig. 9 Average fraction of globally optimal neighbours for peers of different utility ranking

and have long enough session times to fully optimise their neighbourhood sets. Thus, the topology consists of a stable "core" of the highest utility peers, which maintain the topology structure, and lower utility peers that are subject to heavy churn and have a reduced ability to optimise their neighbourhood sets.

Furthermore, the highest utility peers manage to maintain close-to-optimal neighbourhood sets in all experiments with median peer session times ranging from infinity (no churn) to 5min.

In order to get more insight into the structure of the gradient topology, the subsequent experiments measure the average path length between highest utility peers in the system. $D(p, q)$ is defined as the shortest path length between peers p and q in the system topology T, and analogously, $D_S(p, q)$ and $D_R(p, q)$ are defined for T_S and T_R. The average path length in T, denoted $Apl(V)$, is the average value of $D(p, q)$ over all possible pairs of peers (p, q) in the system

$$Apl(V) = \frac{\sum_{p,q \in V} D(p, q)}{|V|^2}. \tag{31}$$

Given a utility rank r, the average path length between the r highest utility peers is $Apl(V_r)$. Furthermore, Apl_S and Apl_R are again defined as the average path lengths in T_S and T_R, respectively.

The average path length $Apl(V)$ can be calculated using the Dijkstra shortest path algorithm at $O(|V|^2 d)$ cost, where d is the average peer degree in V. However, in the system described in this paper, with $|V| = 100,000$ and $d = 13$, this would require performing over $100,000,000,000$ basic operations. This cost can be reduced by selecting a random subset V' from V and approximating $Apl(V)$ with

$$Apl'(V) = \frac{\sum_{p \in V'} \sum_{q \in V} D(p, q)}{|V'| \cdot |V|}. \tag{32}$$

Such approximation requires running the Dijkstra algorithm for $|V'|$ peers, and hence, incurs the computational cost of $O(|V'||V|d)$ operations. In practice, $|V'| = 100$ generates accurate results.

In the unlikely case where two peers p and q are not connected in the system topology, the distance $D(p, q)$ is not defined and the (p, q) pair is omitted in the calculation of Apl'. The number of such pairs is extremely low in the reported experiments and such pairs only occur when a peer becomes isolated and needs to be re-bootstrapped. With the exception of isolated peers, topology partitions were never observed in any of the experiments described in this paper.

Figure 10 shows the average path length between the highest utility peers in the system. Each point plotted in the graph represents the value of $Apl'(V_r)$ for a

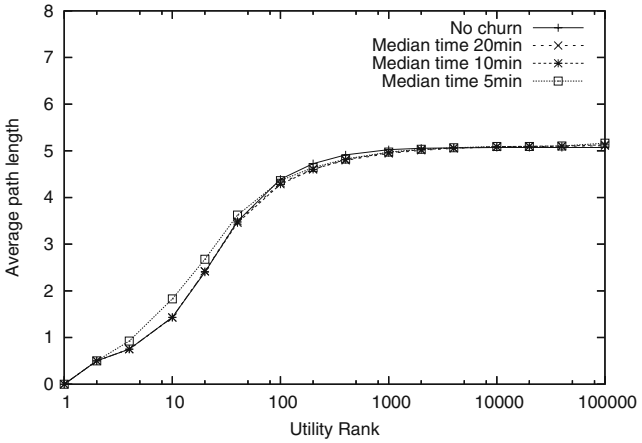

Fig. 10 Average path length between peers of different utility ranking

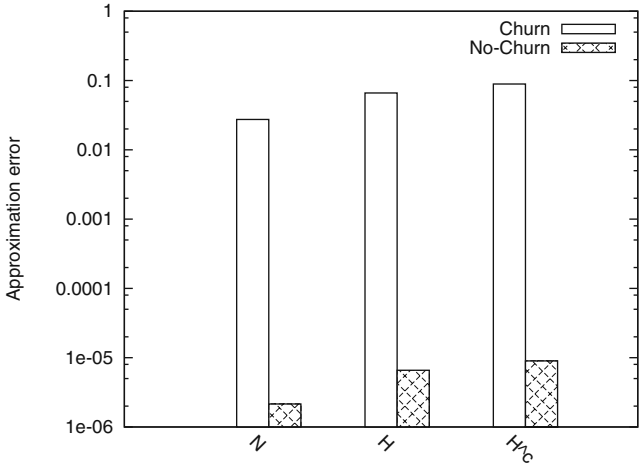

Fig. 11 Influence of churn on aggregation error

given utility rank r. For all churn rates, the average path length gradually converges to zero when decreasing r. This confirms the emergence of a gradient structure in the system topology, where high utility peers, determined by a utility threshold, are closely connected.

3.6 Aggregation

This section evaluates the accuracy of the aggregation algorithm. The following notation and metrics are used. Variables $N_{p,t}$, $H_{p,t}$ and $H^c_{p,t}$ denote the current estimations at peer p of the current system size, N, utility histogram, H_t, and capacity histogram H^c_t, respectively, at time step t. The average relative error in the system size approximation, calculated over all time steps and peers in the system, is defined as

$$Err_N = \frac{1}{Time} \sum_{t=1}^{Time} \frac{1}{N} \sum_{p} \frac{|N_{p,t} - N|}{N}. \tag{33}$$

where $Time$ is the experiment duration. Similarly, the average error in utility histogram estimation, Err_H, is defined as

$$Err_H = \frac{1}{Time} \sum_{t=1}^{Time} \frac{1}{N} \sum_{p} d(H_t, H_{p,t}) \tag{34}$$

where d is a histogram distance function defined as

$$d(H_t, H_{p,t}) = \frac{1}{B} \sum_{i=0}^{B-1} \frac{|H_t(i) - H_{p,t}(i)|}{H_t(i)}. \tag{35}$$

Analogously, Err_{H^c} is defined as the average error in the capacity histogram estimation.

Figure 11 shows the values for Err_N, Err_H and Err_{H^c} in two sets of experiments. In the first set, labelled "Churn", nodes are allowed to join and leave the overlay, as explained in Section 3.2. In the second set of experiments, labelled "No-Churn", the population of nodes is static.

In the absence of churn, the aggregation algorithm produces almost perfectly accurate system property approximations, with the average error below 0.001%. This behaviour is consistent with the theoretical and experimental analysis described in [15]. In the presence churn, the observed error is non-negligible, and is approximately equal to 3% for N and 10% for the histograms. The next section evaluates the influence of churn and aggregation error on the super-peer election.

There are two parameters that control the cost and accuracy of aggregation, which are the frequency of instance initiation, F, and an instance time-to-live, TTL. Additionally, the histogram resolution, B, impacts on the accuracy of utility distribution approximation.

When F is decreased, peers perform aggregation more frequently, and have more up-to-date estimations of the system properties. However, in the described experiments, the system size and the probability distributions of peer utility and capacity are constant, and hence, running aggregation more often does not affect the results. At the same time, when F is decreased, the average message size increases, since peers participate in a higher number of aggregation instances.

Similarly, when the TTL parameter is increased, aggregation instances last longer, peers store more local tuples, and aggregation messages become larger. However, as shown in Fig. 12, if the TTL parameter is too low (e.g., equal to 30), the aggregation instances are too short to average out the tuples stored by peers and the results have a high error. Conversely, when aggregation instances run longer, they suffer more from

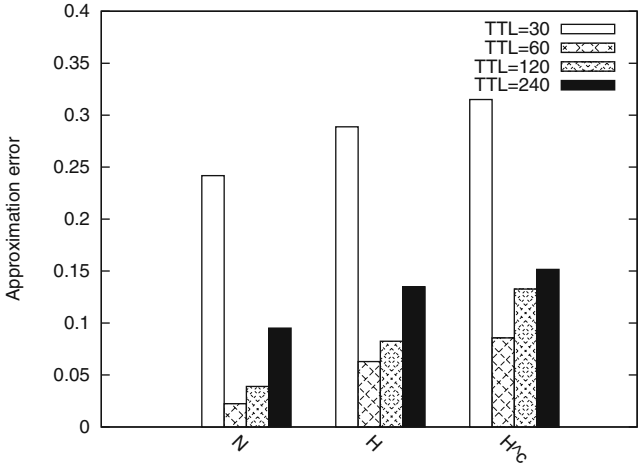

Fig. 12 Aggregation error versus instance TTL

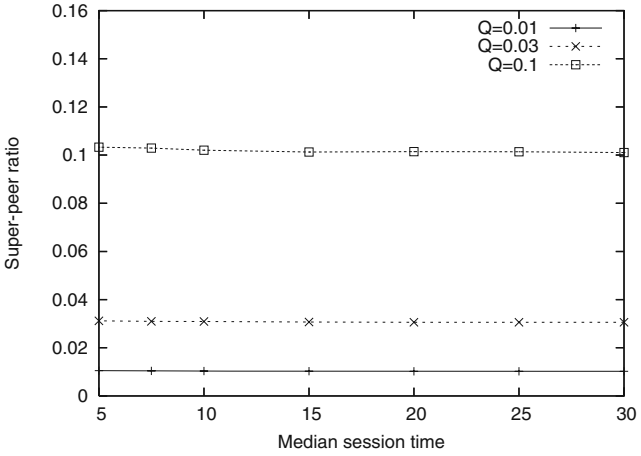

Fig. 13 Average super-peer ratio observed in the system

churn (more tuples are lost during an instance) and the quality of results gradually deteriorates. It appears that optimum performance is achieved for $TTL \approx 50$, and this value for TTL is used in the experiments described in this article.

Finally, the accuracy of utility distribution approximation can be improved by increasing the histogram resolution, B. Clearly, the message size grows linearly with the number of histogram bins. The actual accuracy improvement depends on the shape of the distribution function and the histogram interpolation method. In this article, linear interpolation is used and histograms have 100 bins. This way, aggregation messages have below 1kB, and would fit well into UDP packets, assuming this protocol was used in the implementation.

3.7 Super-peer election

The following section evaluates the super-peer election algorithm for the registry replica placement. A number of experiments is performed. In each experiment, the P2P network initially consists of one peer and is gradually expanded until it grows to N peers, as in the previous section. Super-peers are elected using two proportional thresholds, an upper threshold t_u and a lower threshold t_l, such that $t_u = t_Q$, where Q is the desired super-peer ratio in the system, and $t_l = t_{Q+\Delta}$, where Δ determines the distance between upper and lower thresholds. At any time t, M_t denotes the current number of super-peers in the system, $\frac{M_t}{N}$ is the current super-peer ratio, and $Err_t = |M_t - QN|$, called algorithm error, is the difference between the elected and the desired numbers of super-peers in the system, which reflects the election algorithm accuracy. Similarly, $RErr_t = \frac{Err_t}{QN}$ is the relative election error.

As the network grows to N peers, the system is run for 2,000 time steps and results are aggregated. M denotes the average number of super-peers in the system over all time steps, Err is the average error, and $RErr$ is the relative algorithm error.

The first set of experiments investigate the impact of churn on the super-peer election. Figure 13 shows the average super-peer ratio $\frac{M}{N}$ in systems with 50,000 peers, where Q is set to 0.01, 0.03 and 0.1, and the median peer session duration is ranging between 5min and 30min. Figure 14 shows the average error $RErr$ in the same experiment. As expected, the accuracy of the election algorithm degrades when the churn rate increases (i.e., for shorter peer sessions), since churn affects the aggregation algorithm, causing larger error in the generated aggregates and in the threshold calculation. However, in all cases, the observed super-peer

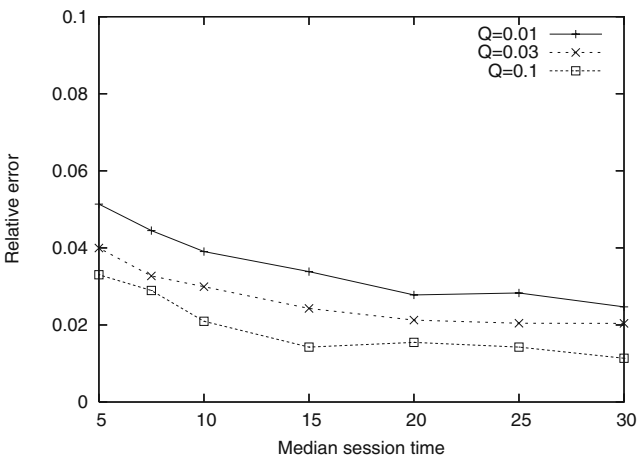

Fig. 14 Average error in the number of super-peers elected as a function of churn rate

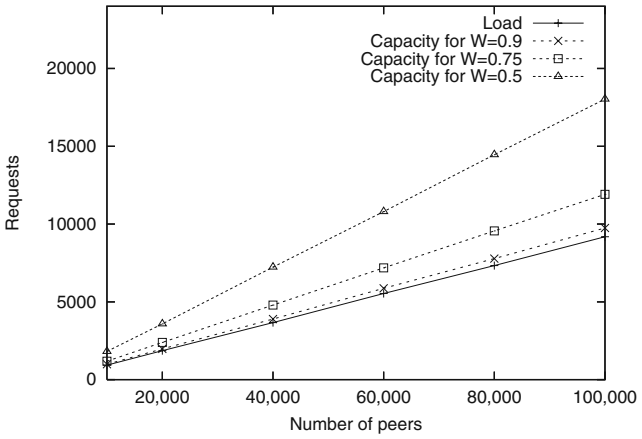

Fig. 15 Total load and super-peer capacity in systems with varied sizes and adaptive thresholds

ratio is close to Q, and the average error is bounded during simulation at 5%, which shows that peers' local estimations of the super-peer election thresholds are close to the desired values.

Moreover, it should be noted that the calculated peer session durations and the churn rates are based on the assumption that the time step is 6 seconds long. The system can achieve better churn tolerance either when peers run periodic algorithms more frequently and exchange more messages (i.e., the time step is shortened).

In the next experiments, super-peers are elected using adaptive thresholds t_W, where W is the desired super-peer utilisation, with the upper threshold $t_u = t_W$ and lower threshold $t_l = t_{W-\Delta}$. Figures 15 and 16 show the total system load, super-peer capacity, and the average super-peer utilisation in a number of experiments with the system size N ranging from 10,000 to 100,000 and W set to 0.9, 0.75, and 0.5. The median peer session length is fixed at 10min and $\Delta = 0.01$. It can be seen that the systems exhibit stable behaviour, with the total super-peer capacity growing linearly with the system size and proportionally to the system load, while the average super-peer utilisation remains at a constant level, relatively close to W.

3.8 Dynamic peer utility

In the previous sections, it has been assumed that peer capacity is constant. However, this assumption may not always be realistic, as resources such as storage capacity, network bandwidth, processor cycles and memory, can be consumed by external applications, reducing the capacity *perceived* by the peer. Furthermore, a peer may be unable to determine its capacity precisely, and may have to rely on local measurements or heuristics that incur an estimation error.

In the following experiments, each peer p has a constant maximum capacity value, $C^*(p)$, and a current capacity value, $C(p)$, determined at each time step by formula $C(p) = C^*(p) \cdot (1 - \varepsilon)$, where ε is randomly chosen between 0 and ε_{max}. Thus, the ε parameter can be seen either as the peer capacity estimation error or the interference of external applications.

Each experiment is set up with three parameters: the capacity change amplitude, ε_{max}, labelled "Epsilon" on the graphs, the desired super-peer utilisation, W, and the difference between the upper and lower thresholds, Δ. In order to prevent super-peers close to the election threshold from frequently switching their status to ordinary peers and conversely, super-peers are elected using two utility thresholds, where again $t_u = t_W$ and $t_l = t_{W-\Delta}$.

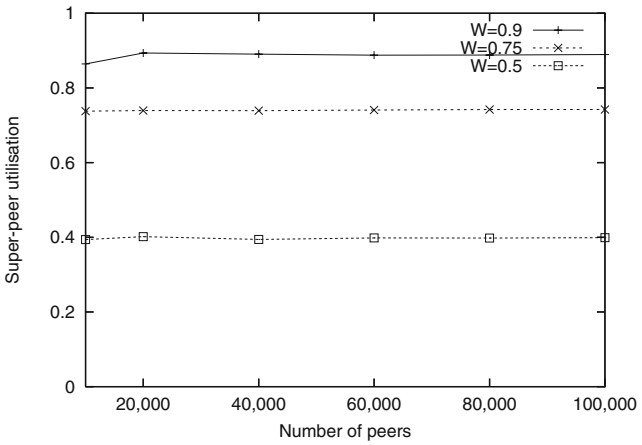

Fig. 16 Average super-peer utilisation as a function of system size

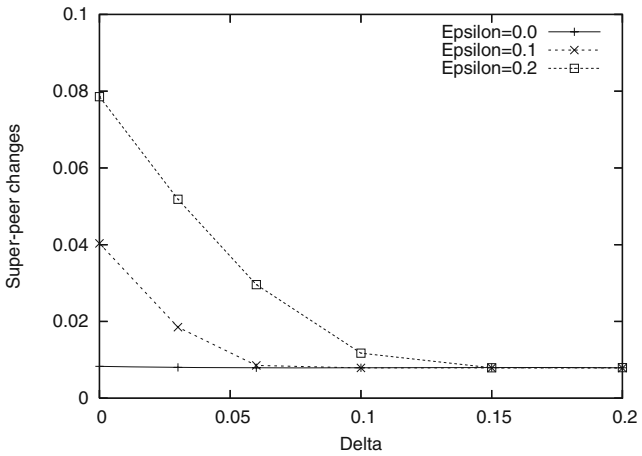

Fig. 17 Relative number of super-peer changes per time step as a function of Δ

Figure 17 shows the average number of super-peer changes as a function of Δ. The experiment demonstrates that the number of changes sharply decreases as Δ is increased. However, it does not converge to zero, but rather to a constant positive value. This is caused by the fact that some super-peers always leave the system, due to churn, and ordinary peers must continuously switch to super-peers in order to maintain enough capacity in the core.

Hence, super-peer changes are due to two reasons. First, as super-peers leave the system, ordinary peers need to replace them and switch their status to super-peers. The number of such switches can be simply determined by counting super-peers leaving the system, and is labelled "Churn" in the graphs. Secondly, both the utility of individual peers and the utility threshold constantly fluctuate, due to changes in the system load, peer departures and arrivals, errors in the aggregation algorithm, etc., which causes peers with utility close to the election threshold to occasionally change their status. The latter category of changes is labelled "Threshold" in the graphs.

Figure 18 shows the number of super-peer changes divided between the two categories in a system with $\varepsilon_{max} = 0.1$. The experiment demonstrates that the number of super-peer changes caused by utility and threshold fluctuations can be reduced to a negligible level by using an appropriate Δ.

Figure 19 shows the impact of Δ on the super-peer election error. As expected, the error grows together with Δ, since a larger gap between t_u and t_l relaxes the constraints on the number of super-peers in the system. The less precise restriction of the number of super-peers in the system is the price for the reduction of the super-peer switches.

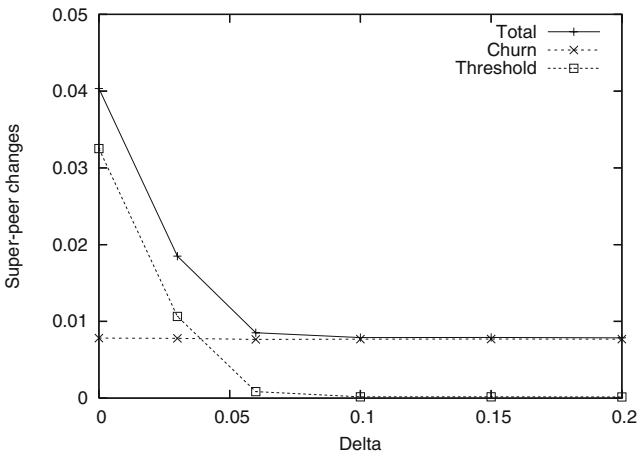

Fig. 18 Relative number of super-peer changes due to super-peer departures and threshold fluctuations

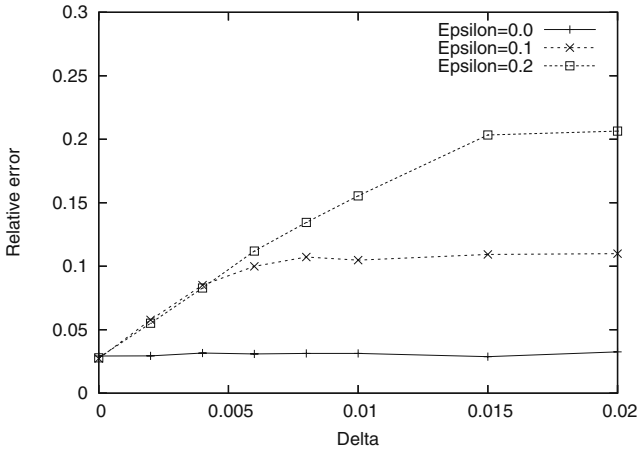

Fig. 19 Relative error in the number of super-peers elected in the system as a function of Δ

3.9 Routing performance

The following section evaluates the routing algorithm used by peers to access SOA registry replicas. As described previously, requests are generated by peers with average probability P_{req} and are routed to available registry replicas hosted by super-peers. In a number of experiments, two properties are measured: the average request hop count and the average request failure rate. Furthermore, these two parameters, hop counts and failure rates, are calculated for requests routed between ordinary peers, before delivered to a super-peer in the core (labelled "Outside core" on the graphs), and for requests routed in the core, when searching for a super-peer with available capacity (labelled "Inside core" on the graphs).

Figure 20 shows the average request hop count as a function of median peer session time. The hop count

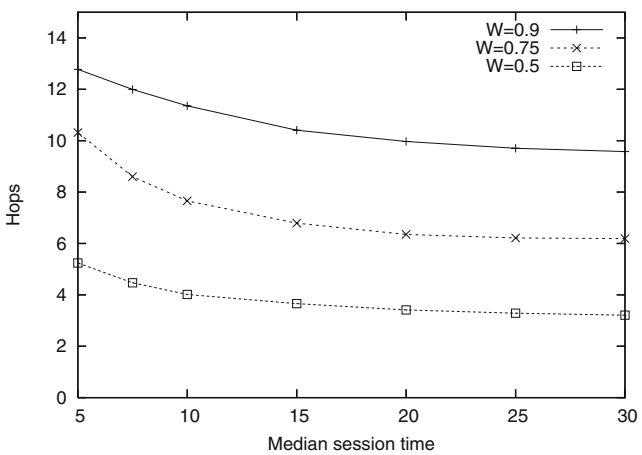

Fig. 20 Average request hop count as a function of churn rate

increases when peer sessions are shorter, which indicates that the topology structure degrades when churn rate increases, reducing the routing performance. Furthermore, the request hop count is significantly higher for $W = 0.9$ than for $W = 0.5$ and $W = 0.75$, which is due to two reasons. First, in systems with higher super-peer utilisation W, fewer super-peers are elected and hence it is harder for ordinary peers to discover the super-peers. Secondly, in systems with higher super-peer utilisation, requests are forwarded more times between super-peers in the core, as it is less likely to discover a super-peer with spare capacity.

This observation suggests that routing is generally more efficient in systems with lower super-peer utilisation. However, with lower W, a larger number of super-peers are elected, which increases the replica maintenance cost, since more data needs to be migrated over the network in order to create and synchronise the replicas. Consequently, the adaptive threshold enables a trade-off between the replica discovery cost and the replica maintenance cost.

Figure 21 shows the impact of churn on the request failure rate. As expected, the number of failures grows when the churn rate is increased, and similarly as the hop count, the failure rate is significantly higher for $W = 0.9$ than for $W = 0.75$ and $W = 0.5$.

Figure 22 shows the average request hop count as a function of the system size. The results show that the hop count does not grow significantly within the investigated range of 10, 000–100, 000 peers, which can be explained by the fact that the hop count depends mainly on the super-peer ratio in the system, which is constant with the system size, and is determined by W. Figure 23 shows the average request failure rate in the same experiment. The results indicate good system scalability, as both the hop count and failure rate are constant with the number of peers in the system.

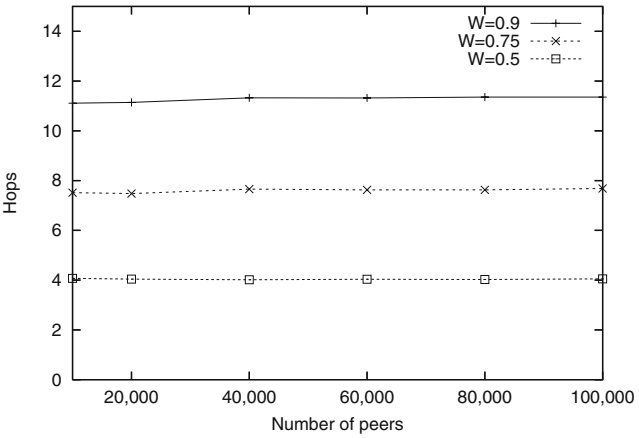

Fig. 22 Average request hop count as a function of system size

3.10 Impact of Boltzmann temperature

The following set of experiments investigates the impact of the Boltzmann temperature $Temp$ on the performance of request routing and the distribution of requests between super-peers. Figure 24 shows the average request hop count outside the core (before a request is delivered to a super-peer) as a function of the temperature $Temp$. The $Temp = 0$ case represents gradient search. The hop count grows steadily with the temperature, and the best routing performance is achieved with the lowest temperature. This justifies the usage of greedy routing (i.e., gradient routing) outside the core.

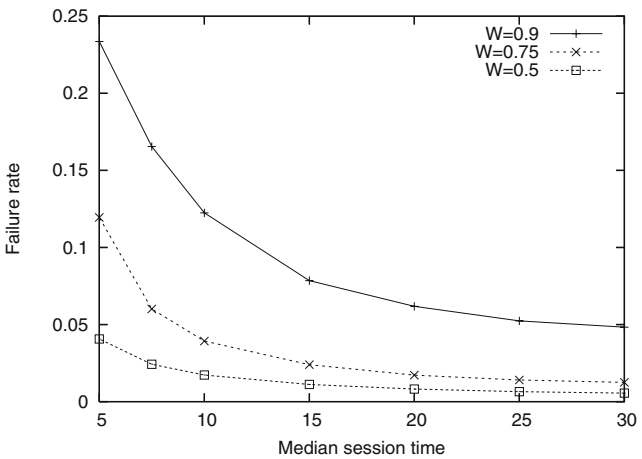

Fig. 21 Request failure rate as a function of peer churn rate

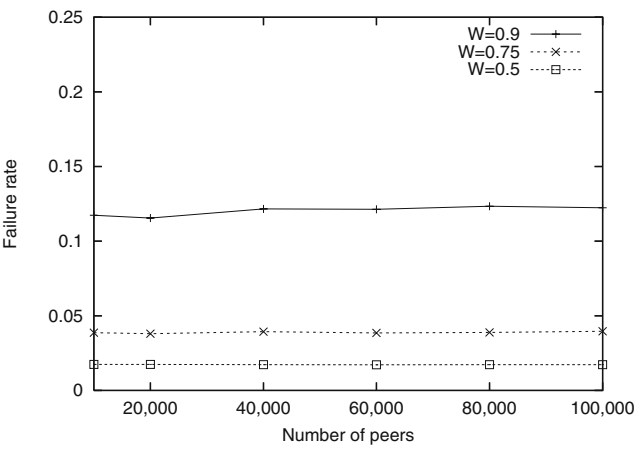

Fig. 23 Request failure rate as a function of system size

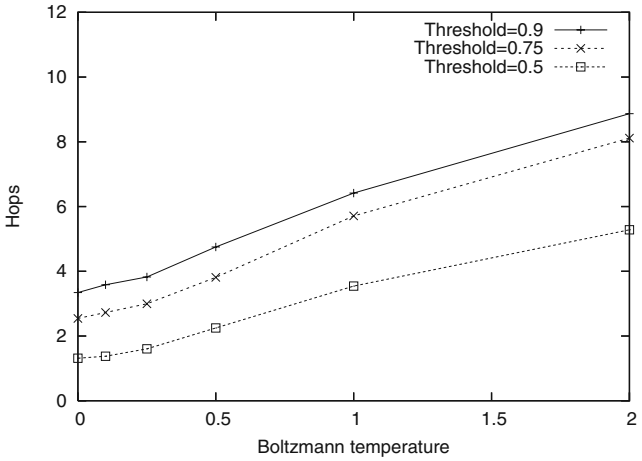

Fig. 24 Average number of request hops outside core as a function of Boltzmann temperature for three different super-peer election thresholds

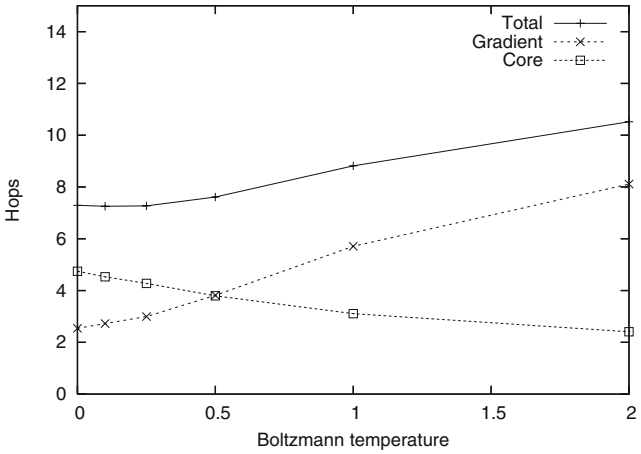

Fig. 26 Average number of request hops inside and outside core as functions of Boltzmann temperature

Figure 25 shows the average number of request hops inside the core as a function of *Temp*. Unlike in the previous experiment, better performance is achieved for higher temperatures. It should be noted that while requests are forwarded between super-peers using formula (28), which is independent of *Temp*, Boltzmann temperature impacts on the delivery of requests to super-peers, and hence may affect routing in the core. This is confirmed by the experimental results; the average request hop count in the core decreases when *Temp* grows, which indicates that the load is distributed more equally between super-peers for higher *Temp*.

Thus, the temperature parameter enables a trade-off between greedy routing, which delivers request quickly to the core, and randomised routing that improves load balancing. This is further illustrated in Fig. 26, which compares the request hop count inside the core and outside the core for $W = 0.75$ and $0 \leq Temp \leq 2$, and in Fig. 27, which shows the average request failure rate in the same experiment. Both figures suggest that the optimal temperature for routing is close to 0.5.

3.11 Impact of system load

The final set of experiments investigates the performance of the system under variable load conditions. Figure 28 shows the relationship between the request probability P_{req} and the total number of super-peers in the system. It can be seen that the super-peer set adapts to the increasing load in the system. The super-peer ratio initially grows slowly, as high capacity super-peers

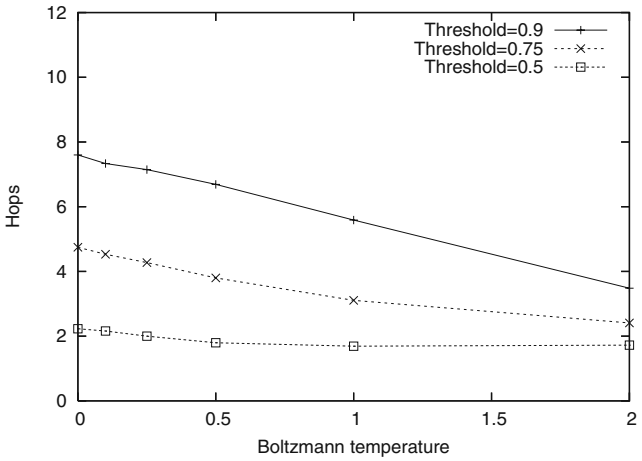

Fig. 25 Average number of request hops inside core as a function of Boltzmann temperature or three different super-peer election thresholds

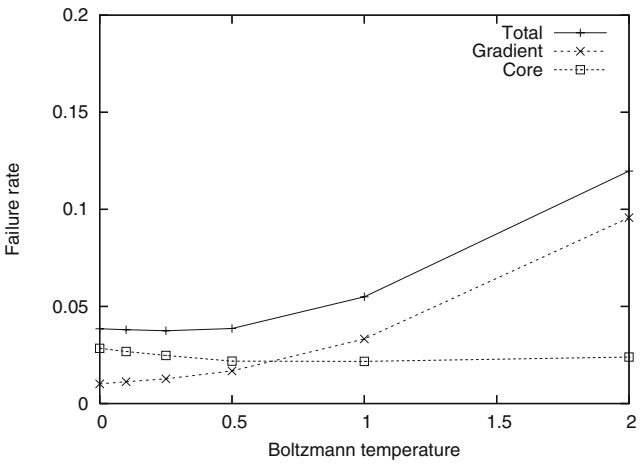

Fig. 27 Average request failure rate inside and outside core as functions of Boltzmann temperature

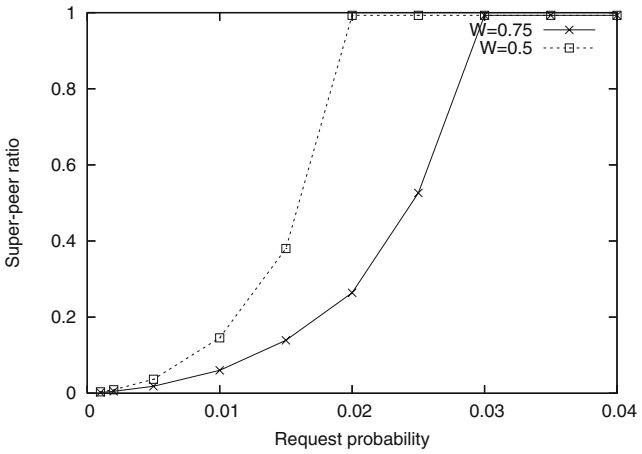

Fig. 28 Super-peers ratio as a function of request probability

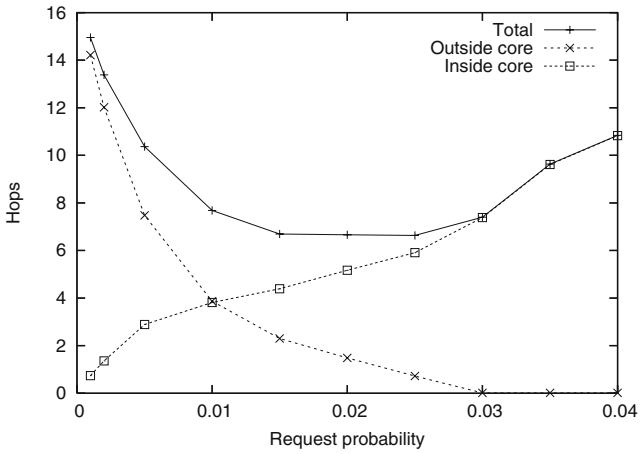

Fig. 30 Average request hop count as a function of system load

are available, but the growth rate quickly increases for higher P_{req}, and eventually all peers in the system become super-peers.

Figure 29 shows the average super-peer capacity and the total system load in the same experiment. The figure demonstrates that the super-peer capacity scales linearly with the load in the system, which is proportional to P_{req}, until all peers in the system are fully utilised.

Figure 30 shows the average request hop count as a function of the request probability P_{req}. Remarkably, the hop count is high for both low P_{req} and high P_{req}, while achieving its minimum for $P_{req} \approx 0.02$. For low P_{req}, the high number of hops is caused by the fact that very few (potentially zero) super-peers are elected when the system load is very low, and hence, it is hard for peers to discover the super-peers. On the contrary, when the load is very high, a large number of peers (potentially all peers in the system) are elected super-peers, and the task of load balancing between super-peers becomes very hard. In the latter case, the performance of routing is mainly determined by the load balancing algorithm. A similar effect can be observed when measuring the request failure rate, as shown in Fig. 31. A high percentage of requests are lost when P_{req} is very low or very high, while the lowest failure rate is reached for P_{req} close to 0.015.

An important conclusion from these experiments is that the system should always maintain a minimum number of super-peers, even in the presence of no load, in order to reduce the request hop count and failure rate. This can be accomplished by combining the adaptive threshold with the top-K or proportional threshold.

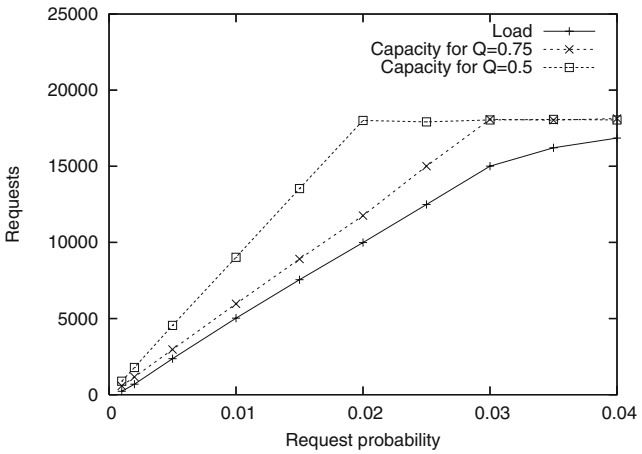

Fig. 29 System load and super-peer capacity as functions of request probability

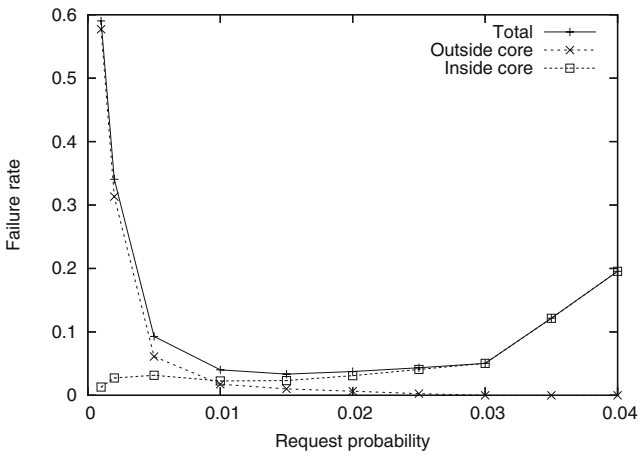

Fig. 31 Request failure rate as a function of system load

4 Related work

An approach to web service discovery that uses a decentralised search engine, based on a P2P network, is described in [23]. In this approach, services are characterised by keywords and positioned in a multi-dimensional space that is mapped onto a DHT and partitioned between peers. A similar approach, described in [24] and [25], partitions the P2P system into a set of ontological clusters, using a P2P topology based on hypercubes, in order to efficiently support complex RDF-based search queries. However, both these approaches are based on P2P networks that do not reflect peer heterogeneity in the system, unlike the gradient topology, and do not address the problem of high utility peer discovery in a decentralised P2P environment.

A number of general search techniques have been developed for unstructured P2P systems (e.g., [26] and [27]), however, these techniques do not exploit any information contained in the underlying P2P topology, and hence achieve lower search performance than the gradient heuristic that takes advantage of the gradient topology structure [16]. Morselli et al. [28] proposed a routing algorithm for unstructured P2P networks that is similar to gradient searching, however, they address the problem of routing between any pair of peers rather than searching for reliable peers or services.

In traditional super-peer topologies, the super-peers form their own overlay within the existing P2P system, while ordinary peers are connected to one or more super-peers. Kazaa [29], Gnutella [30], and Skype [31] are examples of such systems deployed over the Internet. Yang and Garcia-Molina [32] give general principles for designing such super-peer networks. However, nearly all known P2P systems lack an efficient, decentralised super-peer election algorithm. Traditional election algorithms, such as the Bully algorithm [33], and other classical approaches based on group communication [34], cannot be applied to large-scale P2P systems, as they usually require agreement and message passing between all peers in the system. In many P2P systems, super-peers are selected manually, through some out-of-band or domain-specific mechanism. Often, the super-peer set is managed centrally, for example by the global system administrator or designer, and often statically configured (hard-coded) into the system. In other cases, super-peers are elected locally using simple heuristics. These approaches, both centralised and decentralised, often select a suboptimal set of super-peers due to the lack of system-wide knowledge of peer characteristics [21]. This paper describes a more elaborate approach, where the super-peer election is based on the aggregation of system-wide peer utility characteristics.

Xiao and Liu [35] propose a decentralised super-peer management architecture, similar to the one described in this paper, that focuses on three fundamental questions: what is the optimal super-peer ratio in the system; which peer should be promoted to super-peers; and how to maintain an optimal super-peer set in a dynamic system. To this end, they introduce the peer capacity and session time metrics, similarly as in the gradient topology, and they aim to elect super-peers with globally highest capacity and stability in the system. However, their approach uses relatively simple, localised heuristics at each peer in order to estimate system-wide peer characteristics, in contrast to the aggregation algorithms used in this paper. Furthermore, their architecture does not use double election thresholds that reduce the number of swappings between super-peers and ordinary peers, and they do not address varying load and peer capacity.

Montresor [36] proposes a self-organising protocol for super-peer overlay generation that maintains a binary distinction between super-peers and client peers. The algorithm attempts to elect a minimum-size super-peer set with sufficient capacity to handle all client peers in the system. This approach has been further extended in [37], where the super-peer election algorithm not only attempts to minimise the total number of super-peers in the system, but also imposes a limit on the maximum latency between super-peers and their client peers. In contrast, the gradient topology introduces a continuous peer utility spectrum and a gradient structure that enables super-peer election based on adaptive utility thresholds. Furthermore, the gradient topology allows the partitioning of peers into a configurable hierarchy, where each level of the hierarchy consists of peers whose utility values fall within the same utility range, as in [14].

The task of data aggregation, or synopsis construction, has been well-studied in the past in the areas of sensor networks [38, 39] and distributed databases [40, 41]. Most of the proposed algorithms rely on dissemination trees, where the aggregated data is sent to a single node. However, in the architecture described in this paper, all nodes need to estimate global system properties in order to decide on the super-peer election.

Kempe et al. [42] describe a push-based epidemic algorithm for the computations of sums, averages, random samples, and quantiles, and provide a theoretical analysis of the algorithm. Their algorithm has been used for the histogram and utility thresholds calculation in [43]. However, Montresor et al. [15] introduce

a push-pull aggregation algorithm that offers better performance, compared to push-based approaches, in systems with high churn rates. This paper extends the push-pull aggregation algorithm by enabling the calculation of utility and capacity histograms and by adding a peer leave procedure that further improves the behaviour of the algorithm in the face of peer churn.

The approach to decentralise a service-oriented architecture, described in this paper, has been initially proposed in [44]. The gradient search and Boltzmann search heuristics have been first proposed in [16], and the super-peer election thresholds have been introduced in [43]. However, compared with [44], [16] and [43], the algorithms presented in this paper have been substantially elaborated and improved. In particular, the neighbour selection algorithm has been extended, the push-based aggregation algorithm has been replaced by a push-pull algorithm, a new super-peer election approach based on system load and adaptive thresholds has been introduced, an approach to multiple utility functions support has been added, a bootstrap mechanism has been described, and most importantly, a substantially more elaborate evaluation has been performed.

5 Conclusions

This paper describes an approach to fully decentralise a service-oriented architecture using a self-organising peer-to-peer network maintained by service providers and consumers. While the service provision and consumption are inherently decentralised, as they usually involve direct interactions between service providers and consumers, the P2P infrastructure enables the distribution of a service registry, and potentially other SOA facilities, across a number of sites available in the system.

The most interesting element of the presented approach is the gradient topology, which pushes the state of the art of super-peer election algorithms by using aggregation techniques to estimate system-wide peer properties. The gradient topology allows peers to control and dynamically refine and optimise the super-peer set by adjusting the super-peer election threshold; this is, as the authors believe, an important property for super-peer systems in dynamic environments. Furthermore, the approach allows the margin around the super-peer threshold to be configurable, which reduces the impact of random utility fluctuations on super-peer stability. This decreases the system overhead associated with creating or migrating super-peers.

The experimental evaluation of the gradient topology shows that a system consisting of 100,000 peers maintains the desired structure in the presence of heavy churn. Furthermore, peers successfully elect and update a set of highest utility super-peers, maintaining a total super-peer capacity proportional to the system load. The election algorithm can also reduce the frequency of switches between super-peers and ordinary peers, in case of fluctuating peer utility, by applying upper and lower thresholds and relaxing the super-peer utility requirements. Finally, the presented routing algorithms are robust to churn and scale to large numbers of peers, enabling efficient super-peer discovery. Load balancing can be achieved by a Boltzmann heuristic at the cost of routing performance.

Acknowledgements The work described in this paper was partly funded by the EU FP6 Digital Business Ecosystem project (DBE), Microsoft Research Cambridge, and the Irish Research Council for Science Engineering and Technology (IRCSET).

References

1. Huhns MN, Singh MP (2005) Service-oriented computing: key concepts and principles. IEEE Internet Computing 9(1):75–81
2. Jammes F, Smit H (2005) Service-oriented paradigms in industrial automation. IEEE Trans Ind Inf 1:62–70
3. Papazoglou MP, Georgakopoulos D (2003) Service-oriented computing. Commun ACM 46:24–28
4. Sen S, Wang J (2004) Analyzing peer-to-peer traffic across large networks. IEEE/ACM Trans Netw 12:219–232
5. Gummadi KP, Dunn RJ, Saroiu S, Gribble SD, Levy HM, Zahorjan J (2003) Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: Proceedings of symposium on operating systems principles, pp 314–329
6. Saroiu S, Gummadi PK, Gribble SD (2003) Measuring and analyzing the characteristics of napster and gnutella hosts. Multimedia Syst 9(1):170–184
7. Stutzbach D, Rejaie R (2006) Understanding churn in peer-to-peer networks. In: Proceedings of the 6th ACM SIGCOMM conference on internet measurement. ACM, New York, pp 189–202
8. Rhea S, Geels D, Roscoe T, Kubiatowicz J (2004) Handling churn in a DHT. In: Proceedings of the USENIX annual technical conference. USENIX, El Cerrito, pp 127–140

9. Li J, Loo BT, Hellerstein JM, Kaashoek MF, Karger DR, Morris R (2003) On the feasibility of peer-to-peer web indexing and search. In: Proceedings of the 2nd international workshop on peer-to-peer systems. Springer, New York, pp 207–215, LNCS 2735
10. Lakshminarayanan K, Padmanabhan VN (2003) Some findings on the network performance of broadband hosts. In: Proceedings of the 3rd ACM SIGCOMM conference on internet measurement. ACM, New York, pp 45–50
11. Voulgaris S, Gavidia D, van Steen M (2005) CYCLON: inexpensive membership management for unstructured P2P overlays. J Netw Syst Manag 13(2):197–217
12. Jelasity M, Guerraoui R, Kermarrec A-M, van Steen M (2004) The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In: Middleware. Springer, New York, pp 79–98, LNCS 3231
13. Jelasity M, Babaoglu Ö (2006) T-man: gossip-based overlay topology management. In: Proceedings of the 3rd international workshop on engineering self-organising systems. Springer, New York, pp 1–15, LNCS 3910
14. Jelasity M, Kermarrec A-M (2006) Ordered slicing of very large-scale overlay networks. In: Montresor A, Wierzbicki A, Shahmehri N (eds) Proceedings of the 6th IEEE international conference on peer-to-peer computing. IEEE Computer Society, Piscataway, pp 117–124
15. Jelasity M, Montresor A, Babaoglu O (2005) Gossip-based aggregation in large dynamic networks. ACM Trans Comput Syst 23:219–252
16. Sacha J, Dowling J, Cunningham R, Meier R (2006) Discovery of stable peers in a self-organising peer-to-peer gradient topology. In: Proceedings of the 6th IFIP international conference on distributed applications and interoperable systems. Springer, New York, pp 70–83, LNCS 4025
17. Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. MIT, Cambridge
18. Patrick Reynolds AV (2003) Efficient peer-to-peer keyword searching. In: Middleware, ser. LNCS, vol 2672. Springer, New York, pp 21–40
19. Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D (1987) Epidemic algorithms for replicated database maintenance. In: Proceedings of the 6th ACM symposium on principles of distributed computing. ACM, New York, pp 1–12
20. Diot C, Levine BN, Lyles B, Kassem H, Balensiefen D (2000) Deployment issues for the IP multicast service and architecture. IEEE Netw 14(1):78–88
21. Sacha J (2009) Exploiting heterogeneity in peer-to-peer systems using gradient topologies. Ph.D. dissertation, Trinity College Dublin
22. Chu J, Labonte K, Levine BN (2002) Availability and locality measurements of peer-to-peer file systems. In: Proceedings of ITCom: scalability and traffic control in IP networks, vol 4868, pp 310–321
23. Schmidt C, Parashar M (2004) A peer-to-peer approach to web service discovery. World Wide Web 7(2):211–229
24. Schlosser M, Sintek M, Decker S, Nejdl W (2002) A scalable and ontology-based p2p infrastructure for semantic web services. In: Proceedings of the 2nd international conference on peer-to-peer computing, pp 104–111
25. Nejdl W, Wolpers M, Siberski W, Schmitz C, Schlosser M, Brunkhorst I, Löser A (2003) Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In: Proceedings of the 12th international conference on world wide web. ACM, New York, pp 536–543
26. Yang B, Garcia-Molina H (2002) Improving search in peer-to-peer networks. In: Proceedings of the 22nd international conference on distributed computing systems. IEEE, Piscataway, pp 5–14
27. Lv Q, Cao P, Cohen E, Li K, Shenker S (2002) Search and replication in unstructured peer-to-peer networks. In: Proceedings of the 16th international conference on supercomputing. ACM, New York, pp 84–95
28. Morselli R, Bhattacharjee B, Srinivasan A, Marsh MA (2005) Efficient lookup on unstructured topologies. In: Proceedings of 24th ACM symposium on principles of distributed computing, pp 77–86
29. Leibowitz N, Ripeanu M, Wierzbicki A (2003) Deconstructing the Kazaa network. In: Proceedings of the 3rd international workshop on internet applications. IEEE Computer Society, Piscataway, pp 112–120
30. Singla A, Rohrs C (2002) Ultrapeers: another step towards gnutella scalability, version 1.0. Lime Wire LLC, Tech. Rep.
31. Guha S, Daswani N, Jain R (2006) An experimental study of the Skype peer-to-peer VoIP system. In: Proceedings of the 5th international workshop on peer-to-peer systems, pp 1–6
32. Yang B, Garcia-Molina H (2003) Designing a super-peer network. In: Proceedings of the 19th international conference on data engineering. IEEE Computer Society, Bangalore, pp 49–60
33. Garcia-Molina H (1982) Elections in a distributed computing system. IEEE Trans Comput 31(1):48–59
34. van Renesse KPBR, Maffeis S (1996) Horus, a flexible group communication system. Commun ACM 39(4):76–83
35. Xiao L, Zhuang Z, Liu Y (2005) Dynamic layer management in superpeer architectures. IEEE Trans Parallel Distrib Syst 16:1078–1091
36. Montresor A (2004) A robust protocol for building superpeer overlay topologies. In: Proceedings of the 4th international conference on peer-to-peer computing. IEEE Computer Society, Piscataway, pp 202–209
37. Jesi GP, Montresor A, Babaoglu Ö (2006) Proximity-aware superpeer overlay topologies. In: Keller A, Martin-Flatin J-P (eds) Proceedings of the 2nd IEEE international workshop on self-managed networks, systems, and services. Springer, New York, pp 43–57, LNCS 3996
38. Aggarwal CC, Yu PS (2006) A survey of synopsis construction in data streams, ch. 9. Springer, New York
39. Nath S, Gibbons PB, Seshan S, Anderson ZR (2008) Synopsis diffusion for robust aggregation in sensor networks. ACM Trans Sens Netw 4(2)
40. Arai B, Das G, Gunopulos D, Kalogeraki V (2007) Efficient approximate query processing in peer-to-peer networks. IEEE Trans Knowl Data Eng 19(7):919–933
41. Renesse RV, Birman KP, Vogels W (2003) Astrolabe: a robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans Comput Syst 21(2):164–206
42. Kempe D, Dobra A, Gehrke J (2003) Gossip-based computation of aggregate information. In: Proceedings of the 44th IEEE symposium on foundations of computer science, pp 482–491
43. Sacha J, Dowling J, Cunningham R, Meier R (2006) Using aggregation for adaptive super-peer discovery on the gradient topology. In: Proceedings of the 2nd IEEE international workshop on self-managed networks, systems & services (SelfMan). Springer, New York, pp 77–90, LNCS 3996
44. Sacha J, Biskupski B, Dahlem D, Cunningham R, Dowling J, Meier R (2007) A service-oriented peer-to-peer architecture for a digital ecosystem. In: Proceedings of the 1st IEEE international conference on digital ecosystems and technologies. IEEE, Piscataway, pp 205–210

Jan Sacha is a postdoctoral researcher in the Computer Systems Group at Vrije Universiteit Amsterdam. He holds a Ph.D. degree from Trinity College Dublin and a M.Sc. degree from both Warsaw University and Vrije Universiteit Amsterdam. His main research interests include peer-to-peer systems, grid systems, and self-organising systems.

Bartosz Biskupski holds a Ph.D. in computer science from Trinity College Dublin in Ireland and M.Sc. in computer science from Vrije Universiteit Amsterdam in the Netherlands and Warsaw University in Poland. His research interests include peer-to-peer systems, media streaming and self-organisation in distributed systems. He is currently starting up his own technology company.

Dominik Dahlem received a Diplom Engineer in Computer Science from the University of Applied Sciences in Wiesbaden, Germany, and an M.Sc. by research from Trinity College Dublin, Ireland. He is currently working on his Ph.D. at Trinity College Dublin. His research interests include multi-agent reinforcement learning, social network analysis, kriging metamodelling of computer experiments, and simulation technologies on high-performance computing infrastructures.

Raymond Cunningham is the founder of a startup company. Previously, he was a Research Fellow at the Department of Computer Science, Trinity College Dublin. He holds a B.A. degree in Mathematics and M.Sc. and Ph.D. degrees in Computer Science, all from Trinity College Dublin. His research interests covered the area of mobile distributed systems, distributed systems optimisation techniques and adaptive middleware.

René Meier is a lecturer in the School of Computer Science and Statistics at Trinty College Dublin. He holds Ph.D. and M.Sc. degrees from Trinity College Dublin. His research interests include programming models and middleware for very large-scale, context-aware mobile and pervasive computing systems as well as for self-organising (peer-to-peer) systems.

Jim Dowling received the B.A. and Ph.D. degrees in computer science from Trinity College, Dublin, Ireland. He is a researcher at the Swedish Institute of Computer Science in Stockholm, and a former Marie Curie Intra-European scholar. He has managed both national and EU research projects in Ireland and Sweden. His research interests are primarily in the areas of distributed systems, autonomic computing, and middleware.

Mads Haahr is a Lecturer in Computer Science at Trinity College Dublin. He holds BSc and MSc degrees from the University of Copenhagen and a PhD from Trinity College Dublin. He is Editor-in-Chief of Crossings: Electronic Journal of Art and Technology and also built and operates RG. His current research interests are in large-scale self-organising distributed and mobile systems, in sensor-augmented artefacts and in true random number generation.