Join and Leave in Peer-to-Peer Systems: The Steady State Statistics Service Approach

Keno Albrecht, Ruedi Arnold, Michael Gähwiler, Roger Wattenhofer
{kenoa@inf, rarnold@inf, mgaehwil@student, wattenhofer@inf}.ethz.ch
Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

July 2003

Abstract

In this paper we introduce the steady state statistics service as a useful means to collect approximate statistical information about a peer-to-peer system. As an example application we show how this service can be employed for establishing an effective deterministic join algorithm. Through simulation we show that insertion of peers using the service results in a well-balanced system. Moreover, our join algorithm gracefully resolves load imbalances in the system due to unfortunate biased leaves of peers.

1 Introduction

Peer-to-peer (P2P) systems connect ordinary desktop computers throughout the Internet, leveraging their united power beyond the sum of the individual parts. In a P2P system, peers share computer resources and services without a central coordinator. The most prevalent publicly available peer-to-peer systems such as Gnutella or Kazaa focus on the task of sharing multimedia data. In the near future, however, we envision P2P systems designed for more elaborate tasks, such as online collaborations, or transactional database systems.

The growing popularity of real-world P2P systems has spawned a thriving research community. The focus of most research is the development of an efficient search operation (a.k.a. lookup mechanism): Given a search key, a peer responsible for the key must be identified. This operation is related to hashing and is therefore sometimes also known as distributed hashing in conjunction with a distributed hash table (DHT).

Following the seminal work of Plaxton et al. [7], an assortment of variants of P2P systems and distributed hashing algorithms have been proposed, such as CAN [10], Chord [12], PAST [4], and Kademlia [6]. These proposals can be summarized as follows.[1] Each peer is assigned a unique overlay identifier (which is typically a binary bit string, as assumed for the rest of this paper). This ID specifies the "domain space" of the peer; a peer is responsible for storing all keys that are within its domain space. In particular, a key is stored by the peer whose bit string matches the longest prefix of the key. A peer p with bit string $b_1b_2 \dots b_k$ keeps contact with k other peers—its "neighbors." Neighbor p_i ($i = 1, \dots, k$) of peer p features a similar bit string as peer p; in particular all the first $i-1$ bits are the same as the bits of peer p, and the bit i itself is inverted.[2]

The peers are connected in such a way that an efficient logarithmic (in the number of peers, usually denoted by n) search operation is guaranteed, and at the same time each peer only needs to connect to a logarithmic number (that is, $O(\log n)$) of peers.[3]

In essence the logarithmic degree (number of neighbors) and the logarithmic diameter (search time) of P2P systems are required by the dynamics and fluctuation of a P2P system. Peers are highly unreliable—most users owning a peer will join the P2P system only for the time they per-

[1] For reference, see Kademlia [6].

[2] Various systems handle the remaining bits differently; this difference is not of relevance in this paper.

[3] There have also been proposed butterfly-based P2P architectures such as Viceroy [5] that feature only a constant number of neighbors; our contribution translates directly to these approaches as well.

sonally use (e.g. search and download a file) the P2P system, which is on average only an hour [11]. A P2P system does not feature a stable central server (or group of servers) that manages the system. Instead, the P2P system is managed by the peers themselves. Having only a logarithmic number of neighbors helps in the case of a leaving peer, because only a logarithmic number of peers (the neighbors of the peer that left) must search for a new replacement neighbor.

1.1 Joins and Leaves

An important lingering problem, and the primary focus of this paper, is how the overlay ID is assigned to a peer. Since the P2P system is completely decentralized and highly dynamic, present solutions assign the overlay IDs randomly. A newly joining peer connects to an arbitrary peer in the P2P system[4], and chooses a random overlay ID. Similarly to a search operation, the newly joining peer searches for its place (determined by the randomly chosen overlay ID) in the P2P system, and connects to the neighbors. A peer leaving the P2P system (generally without notice) drops all stored keys at once.[5]

It is often argued (see Chord [12] for example) that random overlay ID association will balance the keys well. This is not quite true; in fact, a well-known balls-into-bins analysis [8] will reveal that there is a logarithmic imbalance factor [2]. In other words, with high probability a highly loaded peer stores a factor of $\Theta(\log n)$ more keys than a peer with average load.

There have been a number of recent proposals on how to improve the imbalance. Byers et al. [2] applied the "power of two choices"-paradigm originally developed by Azar et al. [1] to reduce the logarithmic imbalance to $\Theta(\log n / \log \log n)$. Rao et al. [9] adopted hill-climbing techniques introduced by Douceur et al. [3] in a different context to improve the load balancing. Still, these improved solutions are based on randomization.[6] We propose a non-randomized join algorithm deploying a new abstract distributed information service for P2P systems, which leads to balanced P2P systems.

[4]This process is known as "bootstrapping."

[5]In order to allow for fault tolerance, P2P systems store keys at several peers; if one of the peers goes down, there are still enough replicas available.

[6]A more cleverly form of randomization though.

The paper is organized as follows. In Section 2 we introduce our service (dubbed "*4S*") for P2P systems. Section 3 explains our join algorithms based on the *4S*. We compare the balance of our algorithms for the join operation with random solutions in Section 4. In Section 5 we give an implementation of *4S* that does not generate additional traffic. In Section 6 we conclude the paper.

2 Steady State Statistics Service (*4S*)

The steady state statistics service (*4S*) is an abstract decentralized service which provides approximate[7] statistical information about the P2P system.

4S is built on top of the regular P2P structure as sketched in the introduction. The basic idea is as follows: A peer p with bit string $b_1 b_2 \ldots b_k$ is considered to be an "expert" on all the sub domains of all the prefixes of its bit string (that is, for $b_1 b_2 \ldots b_i$, $i = 1, \ldots, k$). The expert knowledge is constructed inductively through information exchange with the neighbor peers. The peer p is by definition an expert about its own sub domain $b_1 b_2 \ldots b_k$. Also, the peer p can deduce the state in sub domain $b_1 b_2 \ldots b_i$ by aggregating its own knowledge on sub domain $b_1 b_2 \ldots b_{i+1}$ (which is available by induction) with the knowledge provided by neighbor peer p_{i+1} about sub domain $b_1 b_2 \ldots \overline{b_{i+1}}$ (peers periodically exchange sub domain information with their neighbors). In the end, peer p can deduce the state of the whole P2P system, which is equivalent to the sub domain of the empty prefix.

For illustration, we give an example: We use *4S* to learn the total number of peers in the P2P system. We assume to have a stable P2P system, as in Figure 1. We describe our example from the perspective of peer p with bit string 001 (see Figure 2). Peers periodically exchange sub domain information with their neighbors. In particular, peer p sends the information that there is one peer in sub domain 001 to neighbor peer p_3 (with ID 000), and in exchange learns that there is one peer in sub domain 000 from neighbor p_3. Literally summing up one and one, peer p deduces that there are 2 peers

[7]The exact up-to-date state of the whole system cannot be known. This would be equivalent to consensus in an asynchronous and dynamic distributed system, which is well known to be impossible.

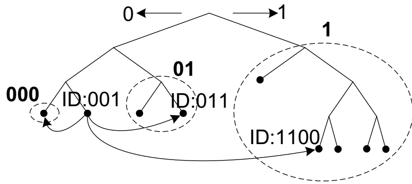

Figure 1: A sample illustration of sub domains. Dashed circles indicate the partitioning of the system into sub domains for peer 001.

with prefix 00. Similarly, on the next higher level, peer p exchanges information with neighbor peer p_2 (ID 011) to learn there are 2 peers with prefix 01. This sums up to a total of 4 peers with prefix 0. In a last step, peer p learns from neighbor peer p_1 (ID 1100) that there are 5 peers with prefix 1. Since there are 4 peers with prefix 0 and 5 peers with prefix 1, peer p knows that there is a total of 9 peers in the P2P system. Note that 4S runs simultaneously at every peer, and therefore provides statistics in a bottom-up aggregated manner at every peer.

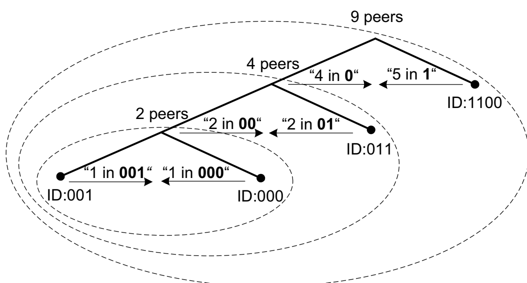

Figure 2: Illustration of the messages exchanged by peer $p = 001$ for the example given in Figure 1.

The accuracy of *4S* depends on the message propagation mechanisms of the implementation. In a stable system, *4S* provides exact statistical information without message overhead. In a dynamic system, more accuracy requires more frequent message exchanges between neighbors. We discuss some of these implementational issues of *4S* in more detail in Section 5.

Besides the example, *4S* can deliver a wide range of information, such as the average up-time of peers, or the total amount of bytes stored in the system.

We emphasize that *4S* is not coupled with any specific P2P system and can be incorporated with all existing proposals we are aware of. We advocate *4S* as a basic building block for P2P systems that can be used for all sorts of steady state statistics of a P2P system.

3 Join Algorithm using *4S*

The insertion of new peers is an essential and challenging operation in a P2P system. In this section we introduce two join algorithms employing *4S* as an example application using information provided by *4S*.

3.1 Random Join (RJ)

Assignment of overlay IDs and insertion of new peers into the P2P system is typically done very similarly in all P2P proposals. As stated in Section 1, joining peers are routed to their destination, which is determined by their randomly assigned overlay ID. We include this *Random Join (RJ)* algorithm for reference in our simulations.

3.2 Number Join (NJ)

Through the *4S* number of peers statistic service presented in Section 2, each peer can deduce which sub domain is sparser (has fewer peers). With a first and simple join approach (*Number Join*, short *NJ*), a new peer (joiner) is first routed through the P2P system to such sparse sub domains and then assigned an overlay ID. At every passing peer, one bit of the bit string of the joiner is fixed. This guarantees termination. If a peer (inserter) cannot route the joiner any further, it becomes responsible for inserting the new peer. The inserter assigns the joiner its own bit string plus a 1, and adds a 0 to its own bit string, thus splitting its domain space in half.

3.3 Depth Join (DJ)

As we will see in Section 4.1, the number of peers in a certain sub domain is not an optimal criterion. Consider Figure 1 again. Newly joining peers are inserted on the left half of the tree, that is with prefix 0, since the sub domain with prefix 0 is sparser. This does not reduce the imbalance in the P2P system, since the most loaded peer (ID 10) remains at depth 2.

For an improved join algorithm we employ the *4S* minimal depth statistic. The *depth* of a peer is defined as the length of its bit string.[8]

[8] Note that we use bit strings of variable length. If the bit strings are of fixed length, the depth of a peer is the length of the so far assigned prefix of its bit string.

The minimal depth statistic of *4S* works as follows (we consider the example given in Figures 1 and 2 again): Peer p with ID 001 wants to know in which sub domain a peer with minimal depth can be found. From its neighbor peer p_3 (ID 000) it knows that its minimal depth is 3, and so deduces that with prefix 00 the minimal depth is 3, since both the sub domain of p_3 and p have the same minimal depth. In the next inductive step, through information exchanged with neighbor p_2 (ID 01) it learns that the minimal depth in the sub domain of p_2 is 3 as well. In a last step, peer p gets to know from neighbor p_1 (ID 1100) that the minimal depth in its sub domain is 2. The overall minimal depth is 2 and *4S* provides peer p with this result.

The minimal depth (*Depth Join*, short *DJ*) algorithm works analogous to the *NJ* algorithm. Instead of using the number of peers, it uses the minimal depth statistic as criterion: joiners are routed toward the sub domain with the smallest minimal depth.

Note that both join approaches can be combined with other load balancing strategies such as load-stealing or load-shedding as described in [2].

As an additional feature, our join algorithms also work against attackers: A malicious adversary might attack a random join system by simply taking out all the peers of a sparse sub domain, making that sub domain even sparser, and raising the load of the remaining peers in the sub domain. Our non-randomized solutions will constantly guide newly joining peers towards the sparse sub domain, filling the gaps of the peers that left.

4 Simulation

We conducted a series of experiments running fine-grained event-driven simulations (factoring in, for example, message delay) of our algorithms. We ran several simulations in order to test the algorithms in different realistic situations, such as balanced and imbalanced P2P systems. The size of the simulated P2P system varies in the range from a couple of hundreds up to tens of thousands of peers. New peer arrivals are modeled as a Poisson process in order to model a real world P2P system. In addition, each simulation consists of 10 runs in order to average all measures and have statistically stronger results.

We use two simple, but reasonable and handy criteria for evaluating the proposed algorithms. The first is the minimal depth D of a peer in the system. We have chosen this quality measure since a low depth stands for a high load. We therefore desire a minimal depth to be as close as possible to the optimal minimal depth.

The second measure, the balance measure B, is more fine-grained and also takes the *number* of peers with small depth into account:

$$B = \sum_{i \in V} 2^{-2d_i} > 0$$

where V is the set of all peers in the system and d_i is the depth of peer i. This way we have a weighted measure in which peers with smaller depth contribute more than peers with larger depth. For expressiveness we normalize the balance B_{Alg} of algorithm *Alg* with the optimal balance B_{Opt}[9]:

$$\rho_{Alg} = \frac{B_{Opt}}{B_{Alg}} > 0$$

By definition, an optimal algorithm *Opt* has a ρ_{Opt} of 1.0.

4.1 Results

Already the naïve *NJ* algorithm leads to good results when the initial setting for insertion of new peers is a balanced P2P system (Figure 3).

[9] Note that in an optimal balanced P2P system all peers are at depth $\lfloor log_2(n) \rfloor$ and $\lceil log_2(n) \rceil$.

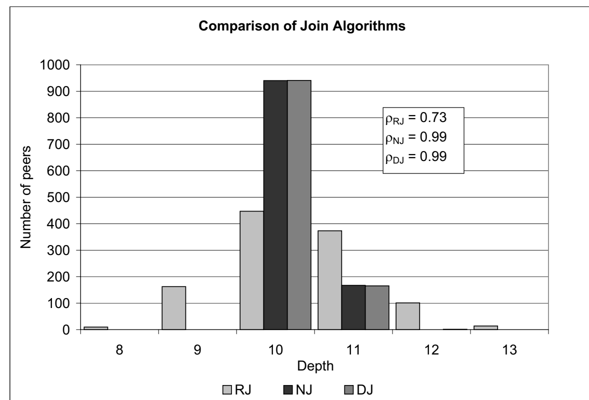

Figure 3: This simulation started with an initial P2P system containing two peers. The graph shows distribution of peers after insertion of 1106 peers with an optimal depth of $\lfloor log_2(1108) \rfloor = 10$.

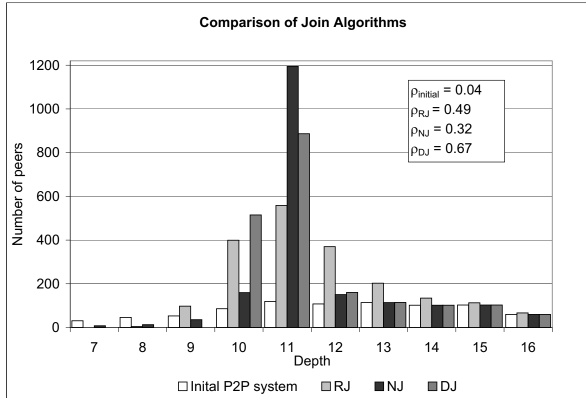

Figure 4: Initially the P2P system is populated with 995 peers in a imbalanced fashion. The graph shows the distribution of peers after inserting 1113 peers to a total of 2108 peers.

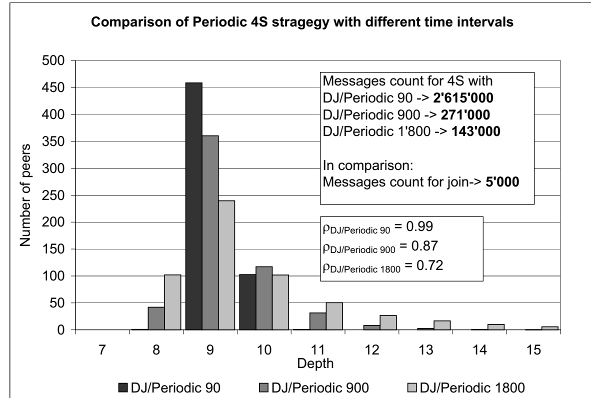

Figure 5: This simulation used the *DJ* algorithm with a *periodic 4S* with different update time interval. The graph shows the distribution of peers after insertion of 561 peers into an empty P2P system.

Unfortunately insertion of peers with the *NJ* algorithm into imbalanced P2P systems does not level out those (Figure 4). The problem is described in Section 3.3. Consequently, the *DJ* algorithm is superior to the others in resolving load imbalances due to biased leaves of peers.

5 Implementation of *4S*

In the implementation of *4S* used in Section 3, every peer periodically sends update messages to its neighbors. The shorter the update interval, the more accurate is the information available for joining peers.

On the other hand, the shorter the interval, the more update messages must be transmitted. For an update interval of 90 time steps, for example, the message load for *4S* is 2.6 million with 563 peers, while the number of messages sent for the join operation is only in the order of thousands (Figure 5). To be practical, the number of messages should be small and scale with the number of peers.

As an improvement, we implement a second approach using an *adaptive* technique. Messages are only sent if there is a change in the *4S* data set. For the *4S* service providing minimal depth information this means: a peer sends an update of the *4S* information to its neighbor if the peer detects a change in the minimal depth. Because changes of minimal depth in an P2P system only take place rarely, this clearly reduces the amount of messages sent. Using the same simulation as above, the message count drops from millions to thousands, as can be seen in Figure 6. Reducing the messages also reduces the quality of B.

Our final and preferred approach to reduce the message overhead employs updating *4S* information while routing messages through the P2P system. For our join application we simply *piggyback* the minimal depth with the "regular" messages. This introduces no additional messages into the P2P system. As illustrated in Figure 6, the quality only reduces scarcely.

Of course, piggybacking does not restrictively

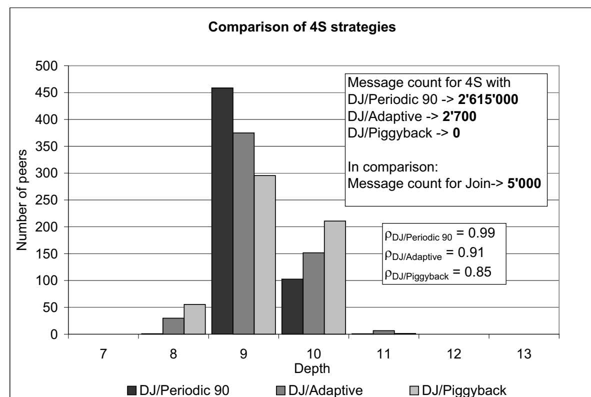

Figure 6: This graph shows the three described approaches—*Periodic 4S*, *Adaptive 4S*, and *Piggyback 4S*—deployed for propagating 4S information through the P2P system. The simulation started with an initial P2P system containing two peers and inserted 561 peers. This leads to an optimal depth of 9.

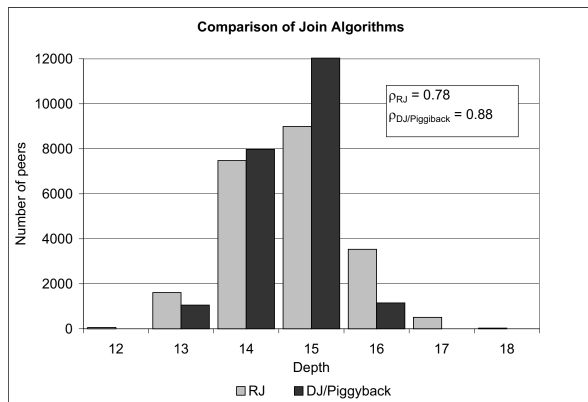

Figure 7: This graph shows distribution of peers after insertion of 22'211 peers with an optimal depth of 14.

apply to the insertion application. Any application using a P2P system can employ piggybacking to spread information within its regular messages. This allows updating of *4S* information at no additional message cost.

Simulations described so far use hundreds of peers. In order to evaluate the scalability of our approach, we simulate our proposed *DJ* algorithm with *piggyback 4S* in a larger setting. Figure 7 shows that our solution performs better with respect to balancing in a P2P system with about 22'000 peers inserted, compared to the commonly used random join, at no additional message cost.

6 Conclusion

We introduced the steady state statistics service as a simple yet useful tool for P2P systems. As a sample application we showed how *4S* is employed for join operations. Our proposed join algorithm is based on minimal depth information provided by *4S*. We show through simulation that it results in better balanced P2P systems than the standard assignment of random overlay IDs, especially in the case where the leaves leave behind an imbalanced P2P system.

As a next step we plan to integrate our join approach using steady state statistics service into an existing P2P system. We intend to conduct experiments in a real-world setting. Besides inserting peers, we believe that *4S* information can be employed by a wide variety of other applications.

References

[1] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proc. 26th Annual ACM Symposium on the Theory of Computing (STOC 94)*, pages 593–602, 1994.

[2] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *IPTPS '03*, 2003.

[3] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of 20th IEEE SRDS*, 2001.

[4] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[5] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings PODC'02*, 2002.

[6] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02, Cambridge, USA*, March 2002.

[7] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[8] M. Raab and A. Steger. "balls into bins" — A simple and tight analysis. *Lecture Notes in Computer Science*, 1518, 1998.

[9] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of IPTPS'03*, Berkeley, CA, February 2003.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[11] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *SIGCOMM*, 2001.