RAPPORT DE RECHERCHE

L R I

IMPLEMENTATION OF AN ORACLE ON TOP OF A PEER SAMPLING SERVICE

PERES O

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

01/2008

Rapport de Recherche N° 1489

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Implementation of an oracle on top of a peer sampling service

Olivier Peres

February 12, 2008

Abstract

Peer to peer systems are large scale distributed systems in which each node has the same role as any other node. Due to their size, these systems distribute the global knowledge of the whole list of peers onto the system such that no one node has the list of all the other nodes. A peer sampling service links the nodes together, forming a connected topology.

To write distributed algorithms for peer to peer systems, one can rely on an oracle, which abstracts out the peer sampling service. We introduce a method to implement this oracle on top of a peer sampling service using a random walk.

1 Introduction

Distributed algorithms work on systems made of several processes linked by a communication network. It is usually assumed that each process knows the list of its neighbours, i.e. all the processes with which it can communicate. This list is updated whenever a process arrives or leaves the system.

This hypothesis, however, is too strong in a large scale peer to peer system. Such systems include thousands of processes, which makes the cost of maintaining neighbour lists too high. Instead, peer to peer systems rely on peer sampling services [3]. Each process can access the service to learn new process identifiers. An underlying protocol maintains a connected topology throughout the execution by inserting arriving processes and deleting departing processes.

In order to write algorithms for large scale systems, one needs a model [2]. In this model, each process has a theoretical device called *oracle* that abstracts out the peer sampling service. This allows the programmer to rely directly on a device that exhibits desirable properties without having to know how it is implemented.

The implementation of an oracle on top of a peer sampling service remains to be addressed. In this paper, we present a solution based on a Markov chain. We show that our implementation has the properties that an oracle needs and discuss both the theoretical and the practical aspects.

The rest of the paper is structured as follows. We discuss related works in section 2, we formally define our implementation and present its theoretical aspects in section 3, we discuss practical considerations in section 4 and conclude in section 5.

2 Related works

Jelasity, Guerraoui, Kermarrec and van Steen formalised the concept of a peer sampling service. According to their definition, such a service provides its user with a function getPeer that returns a process identifier. They proposed a framework to implement and evaluate peer sampling service, arguing that all of them maintain a topology by a gossip-based protocol. The list of identifiers available on a given node is its *view*. An implementation of a peer sampling service has three main characteristics: *peer selection* is its policy concerning the choice of a neighbour with whom to exchange information, *view propagation* determine who sends information to whom once a peer is selected, and *view selection* is a view truncature function. According to the authors' experimentations, the resulting services do not build a random graph based on uniform randomness, but a small-world network following a power law degree distribution.

The theoretical model [2] based on peer sampling services provides an *oracle* that abstracts out this concept. The oracle, when queried, provides a process identifier. There is no guarantee on any one identifier, for example it could belong to a stopped (crashed) process. However, the collection of all the oracles in the system verifies a global condition : in an execution, if S is the set of processes that query their oracle infinitely often, then any $s \in S$ eventually obtains all the processes of S. This gives a theoretical basis in which one can write algorithms that can run on large scale system.

Gkantsidis, Mihail and Saberi [1] studied the effectiveness of random walks on peer to peer systems. This paper uses some of their results, e.g. the time needed to reach the stationary distribution and the cover time.

3 Theoretical discussion

3.1 The algorithm

It is basically a random walk on the graph induced by the peer sampling service. Below is the formal version.

When first queried, the oracle $\mathcal{O}_p$ of process p sends a Request(p) message to itself. Whenever a process q receives Request(p), it calls getPeer to obtain a process r among its neighbours in the view provided by the peer sampling service. Then, q sends Result(r) to p.

When p receives Result(r), it returns r as its answer. The next time p queries its oracle, p sends Request(p) to r and the random walk resumes where it ended.

3.2 Properties

This protocol uses almost no memory. In a system comprising n processes, it only uses two messages of size $O(\log n)$ per request, since each of them only carries a process identifier.

This implementation of an oracle is probabilitic. This makes sense in a large scale system because a (non)deterministic approach would not be scalable. For example, to enumerate all the possible paths in the graph, each node would need one word per process in the system, which is exactly what we want to avoid.

The behaviour of this algorithm can be deduced from the article of Gkantsidis, Mihail and Saberi [1]. Consider a realistic peer to peer system. Assuming nodes are numbered from 1 to n, all the edges are bidirectional, there are E edges and δ_i is the degree of node i, the stationary distribution of the random walk is $\vec{\pi}$ such that for all $i \in [\![1, n]\!]$, $\pi_i = \frac{\delta_i}{2E}$.

In the context of self-stabilisation, the first steps needed to reach $\vec{\pi}$ can be seen as a convergence time. An upper bound on this time is $\tau = \frac{\log \pi_{\min}^{-1}}{1-\lambda_2}$, where λ_2 is the second eigenvalue of the transition matrix. In realistic systems, $-1 < \lambda_2 \ll 1$.

Another relevant metric is the cover time, i.e. the number of steps C_n needed for the random walk to visit all the nodes at least once. It is $O(\frac{\tau_{\min}^{-1} \log n}{1-\lambda_2})$.

In real systems, $\pi_{\min} = \Omega(\frac{1}{n})$ and λ_2 does not change much over time. Thus, the convergence time is $O(\log n)$ and the cover time is $O(n \log n)$.

Remark 1. The fact that the cover time is bounded means that this implementation eventually verifies the global condition on the oracle with probability 1.

4 Practical aspects

In a real system, the time between a query to an oracle and its answer is very long since several messages need to be sent around the system. A query to an oracle is an internal operation, so its running time should be negligible. Also, all the processes are treated equally. In practice, only some of them are of interest for the user algorithm. As a result, the answers of the oracle will often be irrelevant.

4.1 Running time of a query

To make queries faster, we propose a caching mechanism. Each process has a buffer that holds m results. The Request message takes an integer parameter k, the number of steps that the random walk should take. Each step returns a result to the initiator, the last one (taken with $k = 1$) returns an EndResult from which the random walk is to be restarted when new identifiers are needed. When a new result arrives, it replaces the least recently used result. Each process can thus keep its cache filled, provided m is chosen appropriately.

4.2 Making the results more relevant

Typically, for a given user algorithm, a given process is able to determine whether its own identifier should be returned as an answer. In the revised algorithm, each process p returns its own identifier as a result with probability $\alpha_p \in]0,1[$, so the random walk skips p with probability $1-\alpha_p$. On a regular basis, each process calls a function *candidate*, provided by the oracle, that takes a boolean argument. If this argument is *true*, the oracle increases α_p and if it is false, it lowers α_p.

If a process itself does not know whether it is a good candidate, other process may help in obtaining this information. Whenever a process gets p as an answer from its oracle, it may *vote*, again with a boolean argument, to inform the oracle that this result was, or was not, useful. The oracle of p can then adjust its probability accordingly.

This new algorithm is in the same complexity class as the basic algorithm, but we expect that the constants involved should be lower.

5 Conclusion and future works

We present an implementation of a theoretical oracle on top of a peer sampling service, as found in modern peer to peer systems. This allows to run algorithms designed for scalability in real systems.

We intend to make experimental measurements on real systems to evaluate the performance of this implementation. In particular, it will be interesting to determine what a good value of α would be.

References

[1] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, volume 1, pages 120–130, 2004.

[2] T. Herault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. A model for large scale self-stabilization. In *21st IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.

[3] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.