Performance Aspects of a Probabilistic Resource Location Algorithm

Thiago P. Damas[1] **Nelson Duarte Filho** [2] **Mario D. Goulart** [1]
Ingrid Jansch-Pôrto [1] **Fernando Pedone** [3]

[1]Programa de Pós-Graduação em Computação, Instituto de Informática - UFRGS
Porto Alegre, RS, Brazil
e-mail: {tpdamas, mario, ingrid}@inf.ufrgs.br

[2]Departamento de Matemática, Fundação Universidade Federal do Rio Grande
Rio Grande, RS, Brazil
e-mail: dmtnldf@furg.br

[3]Computer Networking Laboratory, École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015, Lausanne, Switzerland
e-mail: fernando.pedone@epfl.ch

Abstract. *Resource location is a fundamental problem for large-scale distributed applications. In order to deal with this problem we have developed* PSEARCH*, a probabilistic protocol for large-scale networks.* PSEARCH *is an epidemic algorithm that uses basic concepts of Bayesian statistical inference to execute probabilistic queries. A probabilistic query has a predicate as parameter and returns a set of sites where the predicate is believed to hold. The query is probabilistic because there are some chances that the predicate does not hold in all, or even in any, of the sites returned. To evaluate the performance of* PSEARCH*, we have developed a simulation model and a prototype of the algorithm. This paper discusses the performance of* PSEARCH *and presents results based on a comparison with a flooding algorithm.*

1. Introduction

Searching for resources is a fundamental problem for large-scale distributed applications. Recently, the problem has re-emerged in the context of peer-to-peer systems such as Gnutella where users find and share information on the Internet (e.g., MP3 music files) [O'Reilly & Associates, 2001] [Clark, 2001].

PSEARCH is a probabilistic search protocol for large distributed systems introduced in [Pedone et al., 2002]. Contrary to deterministic search mechanisms, which strive to compute a precise result, PSEARCH may sometimes provide applications with incorrect information. As we show in this paper, the probabilistic nature of PSEARCH allows for an efficient search mechanism. Probabilistic algorithms have been largely exploited in large-scale distributed systems to improve scalability [Gupta et al., 2001] [Birman et al., 2001] [Birman et al., 1999], but their use as search mechanisms has been more limited.

PSEARCH formalizes the probabilistic resource-location problem with the notion of *probabilistic queries*. A probabilistic query has a predicate as parameter and returns a set of sites where the predicate is believed to hold. Predicates are application dependent: a predicate could be, for example, "the site stores some music file X," or "the site is equipped with a high-performance CPU." After receiving the result of a query, an application would, in the first case, request file X from one of the returned sites; and, in the second case, send a CPU-bound task for execution to one of such sites. The query is probabilistic because there are some chances that the returned result is not correct, that is, the queried predicate does not hold in all, or even in any, of the sites returned.

PSEARCH is an epidemic-like algorithm that uses basic concepts of Bayesian statistical inference. Sites periodically exchange tables containing information about the execution of previous queries. The information received is used to forward queries to the locations where most likely a queried predicate holds. Sites use a gossip technique to exchange these tables. Table entries are updated according to causal relationships between them and information is determined using basic Bayesian statistical inference.

This paper presents a performance evaluation of the PSEARCH protocol using a simulation model and a system prototype. Our experiments aim at investigating the convergence, scalability, and adaptation to changes. As a reference, we use a deterministic, flooding-based algorithm.

Traditionally, the location of resources and information in a distributed system has been accomplished using mechanisms such as global indexes. There are two fundamental differences between those mechanisms and PSEARCH: first, mechanisms based on indexes perform *search by reference,* while PSEARCH performs *search by content;* second, index-based mechanisms are normally *deterministic*, while PSEARCH is *probabilistic.* The best example of search by reference is the DNS service [P. Albitz, 1998], Internet Domain Name System, largely used today. As many other systems, it has also been designed based on deterministic mechanisms [Dabek et al., 2001] [Clarke et al., 2001] [Kubiatowicz et al., 2000] [Demmer and Herlihy, 1998] [Tanenbaum et al., 1990] [Zhuang et al., 2001].

Differently from deterministic mechanisms, PSEARCH tries to locate information based on patterns of use: if a certain information can be found at some site, there are good chances that this site stores other interesting information of the same kind—conceptually, this is similar to a cache mechanism. Such a pattern can be the result of the way resources are distributed in the network (e.g., "super-site" may store much of the available information) or the patterns of accesses during certain periods of time (e.g., a new music released is likely to increase searches for the corresponding singer). Figure 1 illustrates the latter phenomenon. The data shows that most accesses in the Gnutella network during a 50-minutes sample are for a few data items. Thus, during this time interval, even if a site contains only a few resources, it will satisfy most queries. Besides exploiting such characteristics, PSEARCH is able to adapt to changes in the patterns of use and system failures.

In the context of peer-to-peer networks, some alternative ways of locating resources by content in large-scale networks have emerged. Systems like Gnutella [O'Reilly & Associates, 2001] execute brute-force searches: processes propagate queries to their neighbors in order to find the location of files. The work in [Adamic et al., 2001] builds on the assumption that the number of links connecting processes in large-scale net-

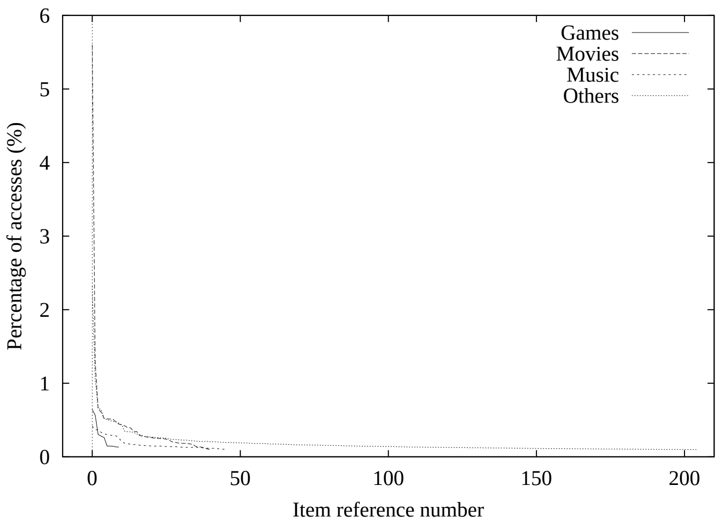

Figure 1: Frequency of searched-for terms during 50 minutes on Gnutella

works follows a power-law distribution, that is, very few processes are connected to most processes in the system and most processes are connected to a few processes. Based on this observation, decentralized algorithms are proposed which strive to visit first processes with a high number of connections. This approach aims at improving the time to reach all sites in the network and consequently floods the system with messages. On the contrary, PSEARCH tries to reduce the number of messages injected in the network by performing a more specialized search and reaching first sites that have higher changes of satisfying the queried predicate.

2. System Model and Assumptions

2.1. Processes and Communication

The model may be generally defined as a set $\Pi = p_1, p_2, \ldots$ of processes (or sites) which communicate by message passing. The system is asynchronous, that is, we make no assumption on the time for process execution and communication[1].

Processes may crash and subsequently recover. Until it crashes, a process follows its protocol—there are no Byzantine failures. When a process crashes it loses all its memory contents. The notions of crash and recover also capture, although conservatively, the behavior of processes that "join" and "leave" the system spontaneously. This is typically what happens with Internet users on dial-in connections who are online for short periods of time. There is no guarantee of stable memory for process storage: after a crash, or a disconnection, a process may have lost, or not, its memory contents.

Processes can be *correct* or *faulty*, according to their behavior with regard to failures. A correct process will be eventually permanently up. A faulty process may crash and recover an unbounded number of times until it crashes and never recovers. Obviously, we do not expect processes to be operational forever. The "eventually permanently up"

[1]This generality of the model, although previewed by the algorithm, is not completely explored by our experiments, for simplicity.

notion is a theoretical abstraction to simplify the treatment of the problem. In practice it means that the process remains up for a "reasonable" amount of time (e.g., enough to execute a few hundred queries).

We assume that communication links, defined by the primitives send(m) and receive(m), can lose messages but are fair, that is, if p_i retransmits a message m to p_j continuously, p_j will eventually receive m. The network is partially connected. The set of neighbors of p_i is denoted by $neighbors(p_i)$.

Each process is equipped with a *timer*. Timers allow processes to give up waiting for events that may never happen, such as receiving for a message from a crashed process. But timers give no guarantee with respect to processes crashes or messages losses. For example, the occurrence of a timeout when p_i is waiting for a message from p_j may mean that p_j has crashed, the message has been lost, the communication link is too slow, or p_i's timer is too fast, and p_i is unable to distinguish among these cases.

2.2. Psearch Specification

PSEARCH implements probabilistic queries. A probabilistic query for predicate Σ is defined by function $Q(\Sigma)$, which returns a set of processes. Probabilistic queries are defined by two properties:

- P1: With probability ϕ_1, query $Q(\Sigma)$ returns some p_i in π in which Σ holds
- P2: Eventually no faulty process is returned in the result of a query

Probability ϕ_1 represents the percentage of executed queries that return at least one process in which the predicate holds. A random choice of any subset of processes is a simple, but inefficient, way to implement probabilistic queries. For example, consider a system of 100 processes where every predicate holds in exactly three of them and queries always return two processes randomly. By applying simple probability theory, we calculate $\phi_1 = 1 - 97/100 \times 96/99 = 0.0594$. PSEARCH improves the values of ϕ_1 by selectively including processes in the result set of a query, trying to avoid processes where the queried predicate does not hold. Such an algorithm, however, requires processes to exchange local information with each other.

Property P2 ensures that eventually only "useful" processes are returned as the result of a query. Faulty processes should be avoided because usually the processes returned in the result of a query will be subsequently contacted, for example, so that the searched object can be reached. Moreover, by trying to "eventually" remove faulty processes, queries will always strive to minimize the number of faulty processes returned.

3. The Psearch Protocol

PSEARCH combines epidemic techniques to propagate local information along the system and basic notions of Bayesian statistical inference to provide processes a way of learning about their abilities to respond successfully to queries. The algorithm presented next is based on the one in [Pedone et al., 2002] with some improvements, as described next.

3.1. Executing queries

To execute query $Q(\Sigma)$ upon request from a local application or from other process, p_i evaluates Σ and depending on the outcome, either replies back to the caller or forwards the query to other processes. Each message received from a process with a query also contains a set of *visited processes*. The visited set aims to reduce the chances that the same query will be received more than once by the same process. To forward a query, p_i chooses those processes in its *s_table* with the highest probability of success that are not in the visited set. Before p_i forwards the query, if it decides to do so, it updates the visited set with such process. The use of a visited set, however, does not completely prevent the reception of duplicated requests. For example, consider that p_1 forwards query $Q(\Sigma)$ to p_2 and p_3. Even though p_2 and p_3 will not forward $Q(\Sigma)$ to processes that already received it through p_1, they may both decide to forward $Q(\Sigma)$ to the same process that not yet received $Q(\Sigma)$.

To limit the diameter of $Q(\Sigma)$, that is, the maximum number of times D—a parameter of the algorithm—that $Q(\Sigma)$ can be forwarded to other processes, a message containing $Q(\Sigma)$ also carries a diameter counter, decremented each time the query is forwarded. If the counter reaches zero at some process p_j and Σ does not hold at p_j, instead of forwarding $Q(\Sigma)$ to another process, p_j returns to the caller a subset of size L—a parameter of the algorithm—of its *s_table* with the processes in which most probably Σ holds. Once p_i receives the response back from the processes it sent the query to, it determines its own response, based on the probability of success of the entries in its *s_table* and the probability of success of the results received from other processes.

To execute query $Q(\Sigma)$, p_i calls function $Q(\Sigma, d, p_i, p_{orig})$ (see Algorithm 1). Function $\max_L(set)$ returns a subset of size L containing those processes in set with the highest probability of success. For each query received, p_i calculates the beliefs a posteriori of each probability of success interval, explained in the next section. The probability of success is taken as the average value of the interval with highest degree of belief.

The query execution algorithm presented in this section differs slightly from the one in the previous work [Pedone et al., 2002]: In order to minimize the effects of timeouts, only the process that originated a query waits for its results. Intermediate processes only forward the query and do not wait for responses.

3.2. Updating *s_tables*

The update algorithm (see Algorithm 2) is an epidemic-like protocol where processes periodically send their *s_tables* to their neighbors. The concept of *neighborhood* used by PSEARCH is based on the communication costs. A neighbor of p_i is a process which is "near" p_i, that is, a process which connects to p_i at a low latency. As local *s_tables* are updated with the information received from other processes, data travels through the network indirectly from process to process. When p_i sends its *s_table* to other processes, its entry in the table is more recent than any other entries in the table since it is continuously updated by p_i while the other entries possibly suffer delays when they are propagated in the network.

For simplicity, in the original version of the algorithm, processes assigned timestamps to their entries using a mechanism similar to Lamport's timestamps [Lamport, 1978]. If an entry e is more recent than an entry e' in p_i's *s_table*, the timestamp assigned to e is greater than the timestamp assigned to e' (the converse is not necessarily true). Thus, before sending the *s_table* to its neighbors, p_i updates the timestamp of its entry with a bigger value than any other timestamps in the table. As pointed by in [Mattern, 1989],

Algorithm 1 Query execution (for process p_i)

```
function $Q(\Sigma)$
    return(Q($\Sigma$, $D$, $\{p_i\}$, $p_i$))

function $Q(\Sigma, d, visited, p_{orig})$
    if $\Sigma$ holds at $p_i$
        for $l = 1..I$ do $P[B]^l \leftarrow \frac{P[B]^l \times P[S|B]^l}{\sum_k P[B]^k \times P[S|B]^k}$

        send($[p_i, 1, -], p_{orig}$)
    else
        for $l = 1..I$ do $P[B]^l \leftarrow \frac{P[B]^l \times P[\bar{S}|B]^l}{\sum_k P[B]^k \times P[\bar{S}|B]^k}$

        $bestSet \leftarrow \max_L(s\_table_i \setminus visited)$
        $result \leftarrow bestSet$
        if $d > 0$
            $visited \leftarrow visited \cup bestSet$
            for each $p_j \in bestSet$
                send(Q($\Sigma, d-1, visited, p_{orig}), p_j$)
                if $p_i = p_{orig}$
                    set timer
                    wait until (receive($response$) | $response = success$) or timeout
                    for each received $response$
                        $result \leftarrow \max_L(result \cup response)$

                else
                    $result \leftarrow bestSet$
    $P[S|B] \leftarrow P[S|B]_l$ s.t $P[B]_l$ is the max in $P[B]_1, P[B]_2, \ldots, P[B]_I$
    $s\_table_i \leftarrow s\_table_i \setminus \{[p_i, *, *]\}$
    $s\_table_i \leftarrow s\_table_i \cup \{[p_i, P[S|B], -]\}$
    send($result, p_{orig}$)

    when receive $Q(\Sigma, d, visited, p_{orig})$ from $p_j$
        $Q(\Sigma, d, visited, p_{orig})$
```

Note: In line 12 we simplify the notation, denoting the set of processes in entries in *s_table* that are not in $visited$ by *s_table* $\setminus visited$. Thus, "$\setminus$" is not the "standard" set operator since *s_table* and $visited$ sets are not of the same type.

solutions using vector clocks, extend Lamport's clocks on static environments.

When p_i receives *s_table*$_j$ from p_j, it updates its *s_table* using the entries in *s_table*$_j$ and taking into account the timestamps associated with the entries. The idea is to try to keep only the most recent entries from both *s_tables*. For entries in both *s_tables* related to the same process, p_i can safely determine the most recent one by looking at their timestamps. For entries related to different processes, choosing the one with the greater timestamp does not ensure that the most up to date one is selected. This happens for two reasons: First, since the relation between entries is a partial order [Lamport, 1978], two entries may not be related (i.e., the order relationship does not apply to the). Second, from the way timestamps are created, an entry with a timestamp bigger than the timestamp of another entry does not necessarily mean that it is the most recent one [Mattern, 1989], if the entries refer to different processes.

In order to make the "best" processes the most known by the other processes, the ones with higher probability of success propagate their *s_tables* more frequently than those with lower probability of success. We implemented this as follows: initially, every process forwards *s_tables* with the same frequency. After responding successfully to a query, the frequency is increased (until it reaches some maximum threshold); after responding unsuccessfully to a query, the frequency is decreased (until it reaches some minimum threshold).

Algorithm 2 Updating *s_table* (for process p_i)

```
Initialization:

    $s\_table_i \leftarrow \emptyset$
    $I \leftarrow 100$
    for $l = 1..I$
        $P[B]^l \leftarrow 1/I$
        $P[S|B]^l \leftarrow (2l-1)/2I$
    $P[S|B] \leftarrow P[S|B]^l$ such that $P[S|B]^l \in P[S|B]^1, P[S|B]^2, \ldots, P[S|B]^I$

To update the search table:

    periodically do
        $new\_tmp \leftarrow$ biggest $timestamp$ in $s\_table_i + 1$
        $s\_table_i \leftarrow s\_table_i \setminus \{[p_i, P[S|B], *]\}$
        $s\_table_i \leftarrow s\_table_i \cup \{[p_i, P[S|B], new\_tmp]\}$
        for each $p_j \in neighbors(p_i)$
            send($s\_table_i, p_j$)

    when receive $s\_table_i$ from $p_j$
        for each $[p_k, P[S|B]_k, tmp_k] \in s\_table_j$
            if $[p_k, P[S|B]_{k`}, tmp_{k`}] \in s\_table_j$
                if $tmp_{k`} < tmp_k$
                    $s\_table_i \leftarrow s\_table_1 \setminus \{[p_k, P[S|B]_{k`}, tmp_{k`}]\}$
                    $s\_table_i \leftarrow s\_table_i \cup \{[p_k, P[S|B]_k, tmp_k]\}$
            else
                $s\_table_i \leftarrow s\_table_i \cup \{[p_k, P[S|B]_k, tmp_k]\}$

        while $|s\_table_i| \geq M$
            $oldestEntries \leftarrow \{[p_k, *, tmp_k] | [p_k, *, tmp_k]$ is the oldest entry in $s\_table_i\}$
            $s\_table_i \leftarrow s\_table_i \setminus oldestEntries$
```

3.3. The probability of success

The probability of success of a process is a local estimate of the likelihood that the next queried predicate received by the process will hold. It is an estimate because the process never knows what the real chances of success are. Processes permanently re-calculate their probabilities of success after executing a query using some heuristics. In PSEARCH, processes use the relation between past successes with respect to the total number of queries locally executed, which roughly means that the more queries the process is able to successfully execute, the higher are the chances that future queries will also be successful.

To determine its local probability of success, $P[S|B]$, each process keeps a list of probabilities of success intervals $[0, prob^1), [prob^1, prob^2), \ldots, [prob^k, 1]$, where $0 \leq prob^1 < prob^2 < \ldots < prob^k \leq 1$, and degrees of belief $P[B]^1, P[B]^2, \ldots, P[B]^{k+1}$ that $P[S|B]$ lies within each one of these intervals—notice that $\sum_l P[B]^l = 1$. Each interval has an approximate probability of success, $P[S|B]^l$, equal to the average of the values in the interval. Probability $P[S|B]$ is taken as the $P[S|B]^l$ with the highest degree of belief. Figure 1 illustrates an initial configuration with 5 intervals. Since all entries have the same degree of belief, $P[S|B]$ can be any value among 0.1, 0.3, 0.5, 0.7 and 0.9. The length of this list determines the maximum accuracy for process to infer their probability of success. The updating is done according to the Bayesian statistical inference. After responding successfully to a query, processes compute equation 1; after responding unsuccessfully to a query, process compute equation 2.

	$P[B]^l$	$P[S\|B]^l$	$P[\bar{S}\|B]^l$
[0.0, 0.2)	0.2	$\approx$ 0.1	$\approx$ 0.9
[0.2, 0.4)	0.2	$\approx$ 0.3	$\approx$ 0.7
[0.4, 0.6)	0.2	$\approx$ 0.5	$\approx$ 0.5
[0.6, 0.8)	0.2	$\approx$ 0.7	$\approx$ 0.3
[0.8, 1.0)	0.2	$\approx$ 0.9	$\approx$ 0.1

Figure 2: Initial configuration

	$P[B]^l$	$P[S\|B]^l$	$P[\bar{S}\|B]^l$
[0.0, 0.2)	0.2	$\approx$ 0.04	$\approx$ 0.96
[0.2, 0.4)	0.2	$\approx$ 0.12	$\approx$ 0.88
[0.4, 0.6)	0.2	$\approx$ 0.20	$\approx$ 0.80
[0.6, 0.8)	0.2	$\approx$ 0.28	$\approx$ 0.72
[0.8, 1.0)	0.2	$\approx$ 0.36	$\approx$ 0.64

Figure 3: Successful query

$$P[B|S]^l = \frac{P[B]^l \times P[S|B]^l}{\sum_l P[B]^l \times P[S|B]^l}, \tag{1}$$

$$P[B|\bar{S}]^l = \frac{P[B]^l \times P[\bar{S}|B]^l}{\sum_l P[B]^l \times P[\bar{S}|B]^l}, \tag{2}$$

4. Psearch Assessment

In the following we consider the performance of PSEARCH under various conditions. The main goals of the experiments were to evaluate (a) the ability of the algorithm to converge to the real probabilities of success, (b) the scalability of PSEARCH and (c) the algorithm's adaptability to changes in these probabilities.

4.1. Reference Algorithm

As a reference, we compare our results to a basic flooding algorithm (Algorithm 3). In the flooding algorithm, when a process executes a query $Q(\Sigma)$ and gets a negative response, it forwards the query to all its neighbors. As in the PSEARCH algorithm, the maximum diameter of a query is determined by D. Processes in the flooding algorithm keep a list of all received queries, so they can discard duplicated queries.

Algorithm 3 Flooding algorithm (for process p_i)

```
function Q(Σ)
    return(Q(Σ, $D, p_i$))

function Q(Σ, $d, p_{orig}$)
    if Σ holds at $p_i$
        send($[p_i, true], p_{orig}$)
    else
        if $d > 0$
            for each $p_j \in neighbors(p_i)$
                send(Q(Σ, $d - 1, p_{orig}$), $p_j$)
            if $p_i = p_{orig}$
                set timer
                wait until receive($response$) or timeout
        else
            send($[p_i, false], p_{orig}$)

    when receive $Q(\Sigma, d, p_{orig})$ from $p_j$
        $Q(\Sigma, d, p_{orig})$
```

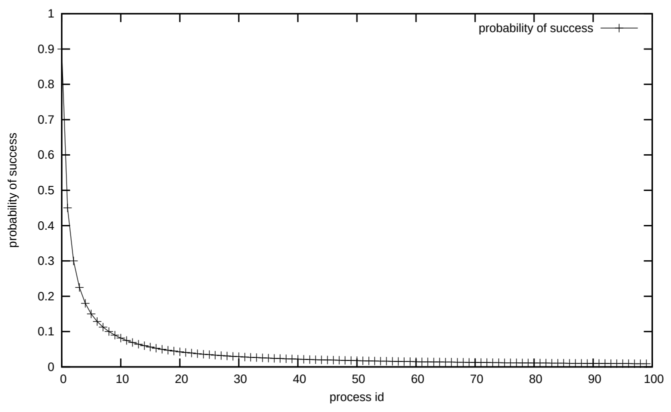

Figure 4: Probabilities of success for 100 processes.

4.2. System's Setup

In order to evaluate the performance of PSEARCH, we developed a simulation model and a prototype of the algorithm. The event-based simulation model was developed in C++, using the CSIM [Mesquite Software,] simulation package and all experiments were executed in a single-processor machine. The prototype is composed of Java processes running in a PC cluster with 16 nodes. In order to get a higher number of processes in the system under evaluation, we used multiple processes per machine.

The behavior of the processes in the simulation model and in the prototype consists in generating queries, answering to queries and, in the case of Psearch, updating and propagating local information (*s_tables*). In both cases, the experiments start by assigning processes real probabilities of success. This is how we represent the "quality" of the processes, that is, the amount of resources that each process has. If we say that a process has a real probability of success equal to 0.3, it means that this process will answer successfully to 30% of all queries. The scenario we used was composed of processes whose probability of success follow a *power-law* distribution, as shown in Eq. (3). Figure 4 shows the probabilities of success assigned to each process in the experiments (for 100 processes).

$$\textit{probability of success} = \frac{0.9}{pid^{-1.25}} \tag{3}$$

In the simulations, the network has a random topology, with an average connectivity of four links per process. It is generated by assigning links randomly between pairs of processes and then calculating the shortest paths between all processes to connect the network. Pairs of processes connected by the random links generated at the first step of the network construction are 1-hop distant. The distances assigned to the other pairs of processes, which are largest than one hop, are obtained by applying the result of the shortest path function. Processes in the PSEARCH system propagate their *s_tables* to 1-hop distant processes. PSEARCH processes have *s_tables* with 10 entries and *best sets* with 3 entries.

4.3. The Results

We present next the results concerning PSEARCH's convergence, scalability, and adaptation. Convergence is related to the time and cost needed by PSEARCH to stabilize (i.e., reach the desired probabilities) starting from some initial configuration ; scalability shows how the algorithm behaves when processes are added to the system; adaptation illustrates the algorithm's capacity to reorganize its data structures as the real probabilities of success change during the execution.

Convergence. To evaluate PSEARCH's convergence, we performed experiments with 20000 queries (about 200 per process). Experiments were conducted for different search diameters, varying from 0 to 3 hops, and data was collected after processes executed 50, 100, and 200 queries. The results (Figure 5 for the simulation and Figure 6 for the prototype) are presented in terms of the mean number of messages exchanged by the protocols per query, and the percentage of hits. From the graphs, for all search diameters, PSEARCH converges faster than our reference algorithm. Moreover, with about 13 messages per query, PSEARCH reaches a plateau, indicating that deeper searches will not significatively improve the number of hits of the algorithm. For all search diameters, the flooding algorithm does not reach the same hit ratio of PSEARCH, and in all cases, more messages are sent per query.

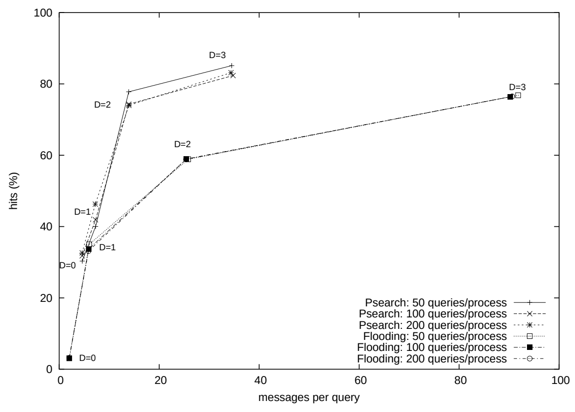

Figure 5: Convergence (simulator data)

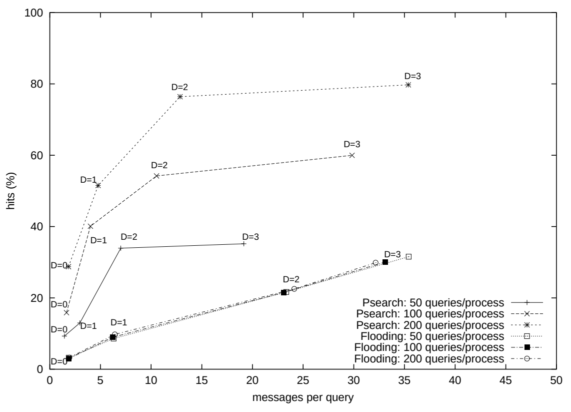

Figure 6: Convergence (prototype data)

Scalability. Experiments were also conducted to study the behavior of PSEARCH as the number of processes increases. In the simulated experiments, processes were connected in the average to four other processes (Figures 7 and 8 for the simulation and the prototype data, respectively). Even when the number of processes is doubled, the number of messages per query remains the same for diameters of 0 and 1 for the results found during simulation and execution of our prototype. For diameters of 2 and 3 hops, the increase in the number of messages is at most 3 messages per query. Performance degradation is more significant in the percentage of hits. While simulation and prototype data are similar for diameters of 2 and 3, the results for diameters of 0 and 1 present different trends: simulation data suggests that there is a significant variation for $D = 0$ and almost no variation for $D = 1$, and prototype results produced opposite results. In the worst case, however, when $D = 0$, the reduction on the number of hits is about 10%.

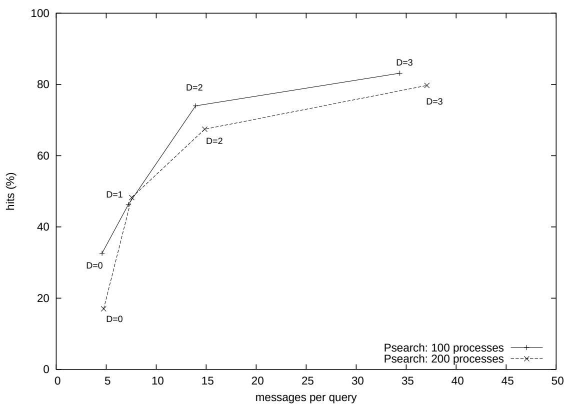

Figure 7: Scalability (simulator data)

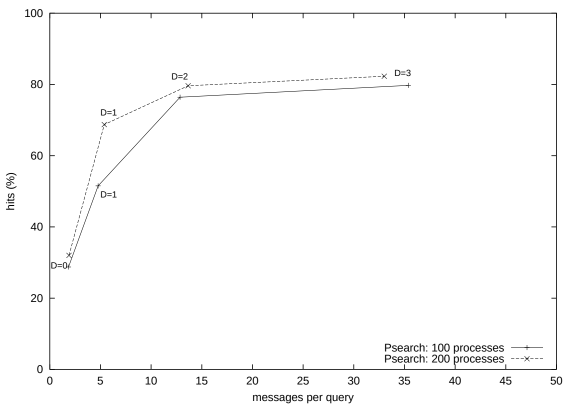

Figure 8: Scalability (prototype data)

Adaptation. To evaluate PSEARCH's adaptation, we performed the following experiments. Processes start the execution with a distribution of real probabilities of success, as done before. Once the system converges, that is, these probabilities become known by the processes, we varied them—this happened after 20000 queries had been executed. Our intent was to fully stress the system, and so, we reversed the order of probabilities of success, that is, the process that had the highest probability of success was assigned the worst one, the process with the second highest probability of success was assigned the second worst, and so forth. The results found (Figures 9 and 10) show that PSEARCH reaches similar convergence values even after the probabilities of success are reversed.

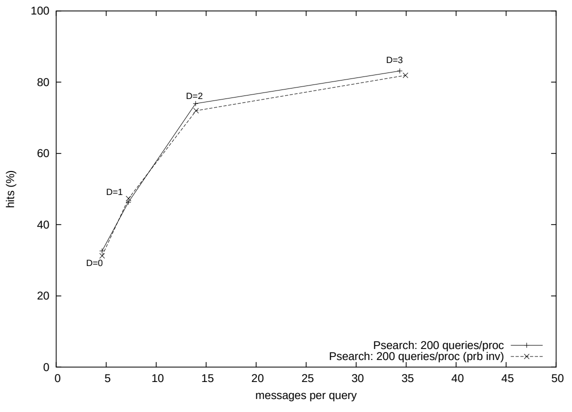

Figure 9: Adaptation (simulator data)

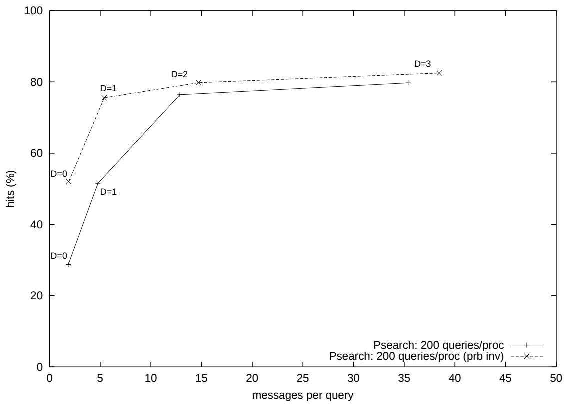

Figure 10: Adaptation (prototype data)

5. Conclusion

The main goal of this research was to better understand the behavior of a probabilistic algorithm from different perspectives. The current study has concentrated on convergence, scalability, and adaptation. As shown in this paper, our results are quite promising.

Convergence, which expresses the ability of the algorithm to stabilize, showed that PSEARCH becomes stable in about half the time needed by a typical flooding algorithm, our protocol of reference. Our scalability study showed that PSEARCH can be nicely employed in large networks. More precisely, we observed that even doubling the number of sites, the efforts necessary to converge did not change significantly; PSEARCH stayed almost stable even with 40 messages per query.

Adaptation, which reflects the behavior of the algorithm when sites change their probabilities of success, demonstrated the responsiveness of the algorithm to dynamic changes. These results are also promising concerning fault-tolerant behavior: a failed site can be thought of as one with very low probability of success. Since PSEARCH can adapt to varying probabilities, we expect it to adapt to the failure of sites as well.

We intend to invest more efforts on the design, implementation, and evaluation of PSEARCH. Further design issues will consider highly-dynamic environments and more sophisticated timestamp mechanisms. We are currently extending our experiments to directly cover faulty processes and links. Finally, other improvements such as the learning ability of joining and recovering processes are also being investigated.

References

Adamic, L. A., Lukose, R. M., Puniyani, A. R., and Huberman, B. A. (2001). Search in power-law networks. Technical report, Hewlett-Packard, Hewlett-Packard Laboratories.

Birman, K., Gupta, I., and van Renesse, R. (2001). Scalabel fault-tolerant aggregation in large process groups.

Birman, K., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. (1999). Bimodal multicast. *ACM Transactions on Computer Systems*, 88.

Clark, D. (2001). Face-to-face with peer-to-peer networking. *IEEE Computer*, 34(1):18–21.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001). Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009.

Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I., and Balakrishnan, H. (2001). Building peer-to-peer systems with Chord, a distributed lookup service. In IEEE, editor, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany*, pages 81–86, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA. IEEE Computer Society Press.

Demmer, M. J. and Herlihy, M. P. (1998). The arrow distributed directory protocol. *Lecture Notes in Computer Science*, 1499.

Gupta, I., Chandra, T. D., and Goldszmidt, G. S. (2001). On scalable and efficient distributed failure detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'2001)*.

Kubiatowicz, J., Bindel, D., Eaton, P., Chen, Y., Geels, D., Gummadi, R., Rhea, S., Weimer, W., Wells, C., Weatherspoon, H., and Zhao, B. (2000). Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 201.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Mattern, F. (1989). Virtual time and global states of distributed systems. In et al., M. C., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Elsevier Science Publishers.

Mesquite Software. CSIM 18 simulation engine (C++ version).

O'Reilly & Associates, I. (2001). *Peer-to-peer: Harnessing the Benefits of a Disruptive Technology*. O'Reilly & Associates.

P. Albitz, C. L. (1998). *DNS and BIND*. O'Reilly & Associates, 3 edition.

Pedone, F., Duarte, N., and Goulart, M. (2002). Probabilistic queries in large-scale networks. In *Proceedings of the 4th European Dependable Computing Conference (EDCC)*. LNCS.

Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J., and van Rossun, G. (1990). Experiences with the amoeba distributed operating system. *Communications of the ACM*.

Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D. (2001). Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *International Workshop on Network and Operating System Support for Digital Audio and Video*.