Probabilistic Queries in Large-Scale Networks

Fernando Pedone[1,2], Nelson L. Duarte[3], and Mario Goulart[3]

[1] Hewlett-Packard Laboratories
Software Technology Laboratory
Palo Alto, CA 94304, USA
fernando_pedone@hp.com
[2] Ecole Polytechnique Fédérale de Lausanne (EPFL)
Faculté Informatique & Communications
CH-1015 Lausanne, Switzerland
[3] Mathematics Department
Universidade do Rio Grande
Rio Grande, RS 96200, Brazil
dmtnldf@furg.br, mario@proxy.furg.br

Abstract. Resource location is a fundamental problem for large-scale distributed applications. This paper discusses the problem from a probabilistic perspective. Contrary to deterministic approaches, which strive to produce a precise outcome, probabilistic approaches may sometimes expose users with incorrect results. The paper 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. 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 implement probabilistic queries, we introduce PSEARCH, an epidemic-like algorithm that uses basic concepts of Bayesian statistical inference. Among its properties, PSEARCH is able to adapt itself to new system conditions caused, for example, by failures.

1 Introduction

1.1 Motivation and Context

Resource location is a fundamental problem for large-scale distributed applications, and even though finding resources in a network of computers is a problem probably as old as distributed computing itself, different system requirements (e.g., high scalability) and conditions (e.g., unreliable user behavior) of current large-scale applications on the Internet have recently led to a flurry of new approaches to the problem. Good examples of current, and very popular, large-scale applications are peer-to-peer systems such as Gnutella where users find and share information on the Internet (e.g., MP3 music files) [1]. Involving many distributed sites, ability to scale well is a highly-desirable property for such systems. Moreover, since virtually any sites can take part in the system, dealing with unreliable sites is also important.

This paper discusses the problem of finding resources in large distributed systems from a probabilistic perspective. Contrary to deterministic approaches, which strive to produce a precise outcome, this paper discusses an approach in which users may be sometimes presented with incorrect results. Algorithms that provide probabilistic guarantees—also known as *probabilistic algorithms*—have recently been exploited in large-scale distributed systems [11, 13, 6] to improve scalability. To the best of our knowledge, however, this is the first time the approach is used in the context of resource location.

1.2 Probabilistic Queries

We formalize 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 sites returned; 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 predicate does not hold in all, or even in any, of the sites returned. Moreover, a resource-locating protocol should not only find sites where a given predicate is satisfied, but highly-available sites where this is true. For example, a site that concentrates many system resources and where most predicates are satisfied is of little use if it is down too often. Therefore, queries should avoid including such sites in their results.

1.3 Psearch Algorithm

To implement probabilistic queries, we introduce PSEARCH, an epidemic-like algorithm that uses basic concepts of Bayesian statistical inference. Briefly, sites exchange information among each other about the execution of previous queries, and use this information to forward queries to the locations where most likely a queried predicate holds. Sites use a gossip technique to exchange these tables among themselves and update entries according to causal relationships between entries and Bayesian statistical inference. PSEARCH is a robust algorithm, which tolerates site crashes and recoveries, message losses, and network partitions. We evaluate the PSEARCH algorithm using a simple analytical model and a simulation model. Our results show that if some of the system sites contain many resources, an assumption that has been found to hold in some environments [10, 12], the results produced by PSEARCH can be very precise.

1.4 Related Work

Traditionally, locating resources and information in a distributed system has been accomplished using mechanisms such as global indexes. There are two fundamental differences between such kinds of mechanisms and PSEARCH: First,

mechanisms based on indexes perform *search by references,* 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 Internet Domain Name System (DNS) [5], one of the largest name services in use today, but many other systems have also been build based on deterministic mechanisms [2, 8, 9, 14, 19, 20].

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 some chances that this site stores other interesting information of the same kind—conceptually, this is similar to a cache mechanism. The advantage of PSEARCH over deterministic approaches is that the system can easily evolve to adapt itself 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 [1] execute brute-force searches: processes propagate queries to their neighbors in order to find the location of files. The work in [3] builds on the assumption that the number of links connecting processes in large-scale networks follows a powerlaw 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 implicitly assumes that processes with a large number of connections will be also the ones most likely to answer application queries; otherwise, it risks to flood the network with messages. PSEARCH is a more general solution, and if the number of connections is indeed related to the likelihood of successfully resolving queries, PSEARCH will adapt to this situation.

1.5 Summary of Contributions

Summing up, the contributions of the paper are the following:

- We propose a formal definition of *probabilistic queries.* This definition quantifies the result of queries in terms of the probability that they contain useful information and how hard it will be to access this information (i.e., the "quality" of the result).
- We introduce PSEARCH, an epidemic-like probabilistic algorithm. PSEARCH is novel in that it combines standard epidemic techniques with Bayesian statistical inference to adapt to new system conditions (e.g., due to process failures and network partitions).
- We develop an analytical model to reason about PSEARCH, different from traditional epidemic-like analytical models, which would not be appropriate in our context.
- We investigate the performance of PSEARCH by means of simulation, considering various probabilistic data distributions and system failures.

1.6 Roadmap

The remainder of the paper is structured as follows. Section 2 describes our system model. Section 3 formally introduces and discusses the notion of probabilistic queries in the context of large-scale distributed systems. Section 4 presents PSEARCH, an algorithm that solves probabilistic queries using simple concepts from distributed systems and Bayesian statistical inference. Section 5 proposes analytical and simulation models to evaluate the performance of PSEARCH. Section 6 concludes the paper.

2 System Model and Assumptions

2.1 Processes and Failures

We model our distributed system as a set $\Pi = \{p_1, p_2, ...\}$ of processes (or sites) which communicate by message passing. Each process p_i is associated with a unique identifier (e.g., its IP address), and executes a sequence of *steps*, where a step can be a change in the process' local state, sending a message, or receiving a message. A process may crash and subsequently recover. After a process crashes, it does not execute any steps until it recovers. The system is asynchronous, that is, there are no bounds on the time it takes for processes to execute steps nor on the time it takes for messages to be transmitted.

Processes are distinguished between *correct* and *faulty*, according to their behavior with regards to failures. A correct process either is permanently up (i.e., it never crashes) or eventually will be permanently up (i.e., the process crashes and recovers at least once but eventually recovers and no longer crashes). A faulty process may crash and recover an unbounded number of times but eventually crashes and never recovers. All processes, however, whenever up, behave according to their protocol—no Byzantine failures. Furthermore, we do not model processes with a permanent intermittent behavior, that is, processes that keep crashing and recovering forever without ever performing any useful computation. Such processes are problematic from a strict viewpoint since they cannot be satisfactorily distinguished from correct processes that crash and recover an unbounded, but finite, number of times [4].

The correct and faulty abstractions are meant to capture not only real process failures but also the behavior of processes that "join" and "leave" the system spontaneously. This is typically what happens with Internet users with dial-in connections who are online for short periods of time. Moreover, "eventually permanently up" is used to simplify the formal treatment of the problem. In practice, we do not expect correct processes to remain up forever, but "long enough" to perform some useful computation such as participate in the execution of a query. Since it would be complicated to determine how long such processes have to remain up, we simply assume that eventually they remain permanently up (i.e., if they are correct) or down (i.e., if they are faulty).

2.2 Process Timers

Each process is equipped with a *timer*. Timers allow processes to give up waiting for events that may never happen, such as receiving a message sent by a process that has crashed. But timers give no guarantees with respect to processes crashes or message losses—as stated previously, there are no bounds on the time it takes for processes to execute steps nor on the time its takes for messages to be transmitted. For example, if p_i waits for a message from p_j and its timer times out, it can be 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 there is no way p_i can distinguish between these cases. The only guarantee provided is that if p_i sets its timer and does not crash, then eventually p_i's timer times out.

2.3 Communication Links

We assume that communication links, defined by the primitives send(m) and receive(m), can duplicate and lose messages but are *fair*, that is, if p_i sends m to p_j a finite number of times, then p_j receives m a finite number of times (e.g., maybe p_j does not receive m at all), but if p_i sends m to p_j an infinite number of times, and p_j is correct, then p_j receives m an infinite number of times (i.e., p_j receives m at least once); furthermore, p_j only receives m if p_i sent m to p_j (i.e., communication links do not create messages). This definition of communication links captures the intuition that if p_i sends m to p_j and both processes do not crash for a "certain period of time," allowing "enough" re-transmissions of m, p_j eventually receives m.

The network is partially connected. The set of neighbors of p_i is denoted by $neighbors(p_i)$. We further assume that there exists a *fair path* connecting any two correct processes p_i and p_j in the system, that is, there is a path $p_i \rightarrow p_{k_1} \rightarrow p_{k_2} \rightarrow ... \rightarrow p_j$ such that every p_{k_l} is correct and every link in the path is fair. Such assumptions admit temporary network partitions, since a process in a unique path between two processes may be temporarily down. However, a fair path between any two correct processes guarantees that network partitions eventually heal since eventually every correct process is permanently up.[4]

3 Probabilistic Queries

3.1 Informal Definition

In this section, we introduce the concept of probabilistic queries. A probabilistic query is a request to find processes in the system in which some local predicate holds. A predicate can be, for example, an assertion about the resources or data available at the process. The query is probabilistic because there is a probability that the result may not contain processes in which the predicate holds and contain processes in which the predicate does not hold.

[4] This assumption is not necessary to ensure progress of the execution of the algorithms discussed in this paper, but it allows correct processes to have a convergent view of the system.

3.2 Formal Definition

To execute a query for predicate Σ, p_i calls function $Q(\Sigma)$ and waits for its result. $Q(\Sigma)$ returns a set $\pi \subseteq \Pi$ of processes. When p_i returns from the invocation of $Q(\Sigma)$ with set π, we say that p_i executed query $Q(\Sigma)$. Probabilistic queries are formally defined by properties P1 and P2, presented next, and property P3, presented in the end of the section.

Let π be the result of query $Q(\Sigma)$, executed by some process:

- P1. With probability ϕ_1, there is some p_i in π in which Σ holds.
- P2. With probability ϕ_2, if p_i is in π then Σ holds in p_i.

Probability ϕ_1 represents the percentage of queries, in a sequence of query executions, whose results contain at least one process where the queried predicate holds. Probability ϕ_2 represents the percentage of queries, in a sequence of query executions, whose results do not contain processes in which the queried predicate does not hold. Consider the case in which $\phi_1 = 1$ and $\phi_2 = 1$. From P1, the result of any query should contain at least a process where the queried predicate holds;[5] from P2, the result of any query should not contain processes where the queried predicate does not hold.

In the absence of property P2, queries could be trivially optimized by always returning Π as a result. In the absence of property P1, queries could be trivially optimized by always returning the empty set as a result. Although hardly useful, these cases help explain why one would not be interested in property P1, or P2, alone.

A simple algorithm, which does not incur in any communication overhead among processes, can be used to implement probabilistic queries. Such an algorithm simply chooses a random subset of Π as the result of a query. In Section 4 we present an algorithm that improves the values of ϕ_1 and ϕ_2 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.

3.3 Excluding Faulty Processes

Only finding processes where some predicate holds may not entirely capture the intuitive functionality expected from queries. For example, returning sets of faulty processes is not useful if these process are to be contacted. Therefore, queries should avoid, whenever possible, returning faulty processes. Properties P1 and P2 are only concerned with distinguishing between processes in which some predicate holds and in which it does not. To exclude faulty processes, we introduce property P3.

- P3. Eventually no faulty process is returned in the result of a query.

[5] $\phi_1 = 1$ is only possible to achieve if all queried predicates hold in some process in the system, which may not always be the case. When not all queried predicates hold in the system, the maximum ϕ_1 obtainable is smaller than 1.

Actually, one would like to always avoid faulty processes in the result of queries—and not only eventually. It turns out, however, that such a property cannot be achieved satisfactorily [4]. The intuitive reason is that no process can tell in advance whether another process will crash, and if crashed, whether it will recover. Thus, when trying to always avoid faulty processes in the result of queries, processes will inevitably exclude from the query result correct processes that will recover, or include in the query result faulty processes that will crash and never recover, both unwanted results. Nevertheless, by tring to eventually remove faulty processes, queries will always strive to minimize the number of faulty processes returned.

4 The Psearch Algorithm

4.1 Overview of the Algorithm

In this section we introduce PSEARCH, an algorithm that implements probabilistic queries. PSEARCH is a highly-resilient protocol, which guarantees progress in the presence of process crashes, message loses, and network partitions—although such events may affect the results produced by the algorithm (i.e., probabilities ϕ_1 and ϕ_2). In order to achieve this degree of resilience, PSEARCH combines epidemic techniques and basic notions of Bayesian statistical inference.

Queries are executed by a recursive algorithm: Upon receiving a query for execution from the application or from other process, p_i evaluates the queried predicate. If the predicate holds locally, p_i replies immediately to the caller—the application that requested the query or the process it received the query from. If the predicate does not hold locally, p_i forwards the query to other processes and waits for the results; p_i uses the results received from these processes, if any, together with its local estimates about where the queried predicate may hold to reply back to the caller.

To decide where to forward queries, p_i keeps a local list of processes in which p_i believes predicates may hold. This list, denoted *s_table*, has a collection of entries, each one for a different process (including p_i). Each entry contains a *process id*, an estimate of the probability that a predicate holds at this process, denoted *probability of success*, and a *timestamp* associated with the probability of success. Process p_i continuously updates the entry to itself in its *s_table* based on the execution of past queries and periodically propagates its *s_table* to its neighbors; *s_table*s received by p_i from its neighbors are used to update the entries to other processes in p_i's *s_table*.

4.2 The Update Algorithm

The update algorithm (see Algorithm 1) is an epidemic-like protocol where processes periodically send their *s_table*s to their neighbors (according to the network topology). As local *s_table*s are updated with the information received from other processes, data travels the network, from process to process. When p_i sends

its *s_table* to other processes, its entry in the table is more up-to-date, or recent, than any other entries in the table since it is continuously updated by p_i while the other entries have to travel the network, possibly suffering delays.

Algorithm 1 Updating *s_table*s (for process p_i)

1: Initialization:

2: $s\_table_i \leftarrow \emptyset$ {s_table_i *initially empty*}
3: $I \leftarrow 100$ {*determines the precision of the model*}
4: **for** $l = 1..I$ **do** {*build table with probabilities and believes*}
5: $P[B]^l \leftarrow 1/I$ {*initial value of the belief a priori*}
6: $P[S|B]^l \leftarrow (2\,l - 1)/2\,I$ {*tentative probabilities of success*}
7: $P[S|B] \leftarrow P[S|B]^l$ s.t. $P[S|B]^l \in \{P[S|B]^1, ..., P[S|B]^I\}$ {*probability of success*}

8: To update the search table:

9: **periodically do** {*epidemic propagation:*}
10: $new\_tmp \leftarrow$ greatest timestamp in $s\_table_i + 1$ {*determine biggest timestamp*}
11: $s\_table_i \leftarrow s\_table_i \setminus \{[p_i, P[S|B], *]\}$ {*remove own entry, in order to...*}
12: $s\_table_i \leftarrow s\_table_i \cup \{[p_i, P[S|B], new\_tmp]\}$ {*...include updated entry*}
13: **for each** $p_j \in neighbors(p_i)$ **do** {*for each neighbor:*}
14: send s_table_i to p_j {*send* s_table_i}

15: **when** receive s_table_j from p_j {*when receive an* s_table_j*:*}
16: **for each** $[p_k, P[S|B]_k, tmp_k] \in s\_table_j$ **do** {*for each process...*}
17: **if** $[p_k, P[S|B]_{k'}, tmp_{k'}] \in s\_table_i$ **then** {*...in both tables...*}
18: **if** $tmp_{k'} < tmp_k$ **then** {*...take the most up-to-date entry*}
19: $s\_table_i \leftarrow s\_table_i \setminus \{[p_k, P[S|B]_{k'}, tmp_{k'}]\}$ {*remove old entry for* p_k}
20: $s\_table_i \leftarrow s\_table_i \cup \{[p_k, P[S|B]_k, tmp_k]\}$ {*include new entry for* p_k}
21: **else**
22: $s\_table_i \leftarrow s\_table_i \cup \{[p_k, P[S|B]_k, tmp_k]\}$ {*include the new entry*}

23: **while** $|s\_table_i| \geq M$ **do** {*keep table in its right size*}
24: $oldestEntries \leftarrow \{[p_k, *, tmp_k] \mid [p_k, *, tmp_k]$ is the oldest entry in $s\_table_i\}$
25: $s\_table_i \leftarrow s\_table_i \setminus oldestEntries$

Processes assign timestamps to their entries using a mechanism similar to Lamport's timestamps [15]. 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 value bigger than any other timestamps in the table.

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_table*s. For entries in both *s_table*s related to the same process, p_i can safely determine which one is the most recent (i.e., the one with the biggest timestamp). For entries related to different processes, we face two problems. First, since the relation between entries is a partial order [15], two entries may be not related, and it does not make sense to talk about which one is more recent than the other. 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 [16], if the entries refer to different processes.

The epidemic nature of the algorithm leads to a simple way to handle network partitions. In case of a network partition, *s_table*s from processes in on partition will fail to reach processes in the other partition. If the partition lasts a long time, the algorithm will tend to make the system converge to a state where *s_table*s of processes in a partition only contain entries for processes in the same partition. However, since processes keep trying to send their *s_table*s to all their neighbors, once the partition heals, *s_table*s will reach processes in both partition, and the system will tend to converge back to the state prior to the partition.

4.3 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 processes (this mechanism is similar to the one used by Gnutella [1]). 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 processes. The use of a visited set, however, does not completely prevent the reception of duplicated requests.[6]

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.

[6] For example, consider that p_1 forwards $Q(\Sigma)$ 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 has not yet received $Q(\Sigma)$.

To execute query $Q(\Sigma)$, p_i calls function $Q(\Sigma, D, \{p_i\})$ (see Algorithm 2). Function $\max_L(set)$ returns a subset of size L containing those processes in set with the highest probability of success.[7] For each query received, p_i calculates the believes 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.

Algorithm 2 Query execution (for process p_i)

```
1:  function Q(Σ)
2:    return(Q(Σ, D, {p_i}))

3:  function Q(Σ, d, visited)
4:    if Σ holds at p_i then
5:      for l = 1 .. I do P[B]^l ← P[B]^l × P[S|B]^l / Σ_k P[B]^k × P[S|B]^k   {update table after success and...}
6:      result ← {[p_i, 1, −]}                                                  {...returns result}
7:    else
8:      for l = 1 .. I do P[B]^l ← P[B]^l × P[S̄|B]^l / Σ_k P[B]^k × P[S̄|B]^k   {update table after failure}
9:      bestSet ← max_L(s_table_i \ visited)                                    {determine best set of processes}
10:     result ← ∅                                                              {initially no result is known}
11:     if d > 0 then                                                           {if can forward query:}
12:       visited ← visited ∪ bestSet                                           {update current visited processes and...}
13:       for each p_j ∈ bestSet do send Q(Σ, d − 1, visited) to p_j           {...forward query}
14:       set timer                                                             {...to be ready for failures and message losses}
15:       wait until [(∀p_j ∈ bestSet : (receive(response) from p_j)) or timeout]
16:       for each p_j, received response_j from p_j do                         {for each response received:}
17:         result ← max_L(result ∪ response_j)                                 {compute its own response}
18:     else
19:       result ← bestSet                                                      {return its best guess}

20:   P[S|B] ← P[S|B]_l  s.t. P[B]_l is the max in P[B]_1, P[B]_2, ..., P[B]_I  {determine new prob. of success}
21:   s_table_i ← s_table_i \ {[p_i, *, *]}                                     {update s_table_i}
22:   s_table_i ← s_table_i ∪ {[p_i, P[S|B], −]}                                {done!}

23:   return(result)

24:   when receive Q(Σ, d, visited) from p_j
25:     response ← Q(Σ, d, visited)
26:     send response to p_j
```

[7] In line 9 of Algorithm 2, 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, "\" is not the "standard" set operator since s_table and $visited$ sets are not of the same type.

4.4 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 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 $[0, \lambda^1), [\lambda^1, \lambda^2), ..., [\lambda^k, 1]$, where $0 \leq \lambda^1 < \lambda^2 < ... < \lambda^k \leq 1$, and degrees of belief $P[B]^1, P[B]^2, ..., P[B]^{k+1}$ 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.

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

Fig. 1. Initial configuration

	$P[B\|S]^l$ (new $P[B]^l$)	$P[S\|B]^l$
[0.0, 0.2)	0.04	≈ 0.1
[0.2, 0.4)	0.12	≈ 0.3
[0.4, 0.6)	0.20	≈ 0.5
[0.6, 0.8)	0.28	≈ 0.7
[0.8, 1.0]	0.36	≈ 0.9

Fig. 2. Successful query

Bayesian networks are direct acyclic graphs, where the vertices represent random variables and the edges their relationships. Thus, each process maintains a small Bayesian network $b \rightarrow s$, where b is associated to the probability $P[B]^l$ and s is associated to the probability $P[S|B]^l$ [18]. A Bayesian network can be used to make inferences like: "What is the new degree of belief on a probability interval given that the last query was successful?" To compute the new degree of belief on a given interval, $P[B|S]^l$, we use basic conditional probability: $P[B|S]^l \times P[S]^l = P[S|B]^l \times P[B]^l$, and Bayes theorem:

$$P[B|S]^l = \frac{P[S|B]^l \times P[B]^l}{\sum_k P[S|B]^k \times P[B]^k} \ . \tag{1}$$

Equation (1) is used to compute the belief *a posteriori* on $P[S|B]^l$ (denoted $P[B|S]^l$), which will be the new value of $P[B]^l$ after a query executed at the process holds. If the queried predicate does not hold at the process, a similar equation is used, derived from $P[B|\overline{S}]^l \times P[\overline{S}]^l = P[\overline{S}|B]^l \times P[B]^l$, to re-compute

its table. Figure 2 illustrates the new values for $P[B]^l$ when the queried predicate holds. If the heuristic used to determine $P[S|B]$ is effective, then as the execution evolves, $P[S|B]$ becomes the average value of the interval whose belief tends to 1.

4.5 Classes of Resources

We have simplified the discussion about the algorithm by assuming that all possible queried predicates in the system belong to the same "class." Greater accuracy of the results can be obtained by dividing predicates into different classes (e.g., one class could group predicates involving MP3 Bossa Nova music files and another predicates involving MP3 Jazz music files).

While distinguishing between classes will increase the amount of information that processes have to keep locally—each class has to have its own *s_table* and probability of success, we note that not all processes have to keep information about all classes in the system; however, the more information a process has about a class of predicates, the higher the chances that queries executed by this process for predicates in this class will be successful.

Even though a discussion about how predicates can be divided into classes is beyond the scope of this paper (for a discussion, see for example [17]), the propagation and update mechanisms used to execute queries in such contexts are the same ones used for a single class.

5 Psearch Assessment

5.1 Analytical Analysis

In the following, we assess the PSEARCH algorithm by characterizing ϕ_1 and ϕ_2 analytically. We simplify the analysis by considering executions where no processes fail and timers do not time out. Probabilities ϕ_1 and ϕ_2 can be estimated from the local probabilities ϕ_1^i and ϕ_2^i at each p_i:

$$\phi_{1,2} = \sum_i \frac{\text{number of queries executed by } p_i}{\text{total number of queries executed}} \times \phi_{1,2}^i.$$

We model processes and the entries in their *s_table*s with a directed graph $G(\Pi, E)$, where Π is the set of all processes and E the the set of "logical links" between processes: there is a link from p_i to p_j in G if and only if p_j is in p_i's *best_set*. ϕ_1^i is the probability that $Q(\Sigma)$, initiated in p_i with diameter D, can be successfully executed by some process in A_i, the set of all processes that can be returned by the query. We define $C_i^k(d)$, the k-th *simple path*[8] of length d in G with origin in p_i, as $C_i^k(d) = \langle p_{k_0}, p_{k_1}, ..., p_{k_d} \rangle$, where $p_{k_0} = p_i$ and link $\langle p_{k_l}, p_{k_{l+1}} \rangle \in G, 0 \leq l < d$ (see Figure 3). The set $C_i(d)$ of all simple paths of length d in G with origin in p_i is defined as $C_i(d) = \cup_k C_i^k(d)$.

[8] A path is simple if it only contains different processes.

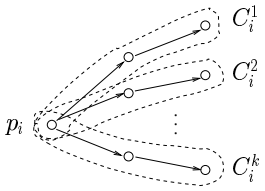

Fig. 3. Paths of length 2

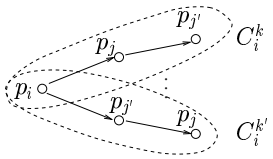

Fig. 4. Paths with the same processes

We calculate A_i by considering all processes in simple paths of length equal to or smaller than $D+1$ and with origin at p_i in G: $A_i = \cup_{d \leq D+1} \cup_{p_j \in C_i(d)} p_j$. ϕ_1^i is the probability that $Q(\Sigma)$ can be solved at some process in A_i:

$$\phi_1^i = 1 - \prod_{p_j \in A_i} P[\overline{S}|B]_j. \tag{2}$$

To calculate ϕ_2^i, the probability that Σ holds at all processes returned as the result of query $Q(\Sigma)$, we determine first the probability that Σ does not hold at any process returned by $Q(\Sigma)$. From the PSEARCH algorithm, processes where Σ does not hold can only be returned by processes at distance D. To see why, consider that some process p_j where Σ does not hold receives $Q(\Sigma)$. If p_j is at distance $d < D$ from p_i, p_j forwards $Q(\Sigma)$ to the processes in its *best_set*; if p_j is at distance D from p_i, p_j returns such processes instead. Thus, if p_l is a process returned by $Q(\Sigma)$ in which Σ does not hold, and $C_i^k(D+1)$ is a simple path of length $D+1$ in G from p_i to p_l, then Σ does not hold in any process in $C_i^k(D+1)$, and we say that Σ does not hold in path $C_i^k(D+1)$.

Therefore, the probability that Σ does not hold at some process returned by $Q(\Sigma)$ is the probability that Σ does not hold in some path $C_i^k(D+1)$. Such a probability would be straightforward to calculate if not for the fact that paths are not independent. Consider for example Figure 4 where both paths C_i^k and $C_i^{k'}$ contain processes p_j and $p_{j'}$. When calculating the probability that Σ does not hold at C_i^k and $C_i^{k'}$, we should consider p_i, p_j, and $p_{j'}$ only once.

To solve this problem, we point out that paths C_i^k and $C_i^{k'}$ correspond to the same *events* in the *space of events* determined by paths in G: both C_i^k and $C_i^{k'}$ correspond to the event that Σ fails in p_i, p_j, and $p_{j'}$.[9] Thus, we initially determine the set $\Omega_i = \{e_i^1, e_i^2, ...\}$ of *events of interest*, that is, a subset of the space of events Ω corresponding to all paths in G starting in p_i of length $D+1$ in which Σ fails in all processes in the path and then calculate the probability that these events happen. The probability that event e_i^j happens, denoted $P(e_i^j)$, is calculated as $P(e_i^j) = \omega_{l_1} \times \omega_{l_2} \times ...$, where ω_{l_k} is the probability of success at

[9] This case follows directly from the commutative property of intersection: $P[A] \times P[B] = P[B] \times P[A]$. Our solution also works in more complex cases, such as "Y-shaped" paths (e.g., $C_i^k = \langle p_i; p_j; p_k \rangle$ and $C_i^{k'} = \langle p_i; p_j; p_{k'} \rangle$), but for brevity we do not further discuss the issue in this paper.

process p_{l_k} or its converse. Finally, probability ϕ_2^i is determined by:

$$\phi_2^i = 1 - \sum_{e_i^j \in \Omega_i} P(e_i^j). \tag{3}$$

To illustrate our analysis, we consider the simple case of a regular graph where each process has the same number $L-1$ of neighbors and the same probability of success α. Queries are executed with a diameter $D = 0$. In this case, the probability that $Q(\Sigma)$ returns a process where predicate Σ holds is:

$$\phi_1^i = 1 - (1-\alpha)^L,$$

which is 1 minus the probability that Σ does not hold in any processes among the L involved (i.e., the process where the query originated plus its $L-1$ neighbors). For the case of a completely connected graph, if $L = n$ then ϕ_1 represents the probability that the "system" can resolve queries. Such a value is the maximum probability achieved by any algorithm—even a deterministic one—when processes have a probability of success equal to α.

The probability that $Q(\Sigma)$ does not return any process where Σ does not hold is:

$$\phi_2^i = 1 - (1-\alpha) \times (1-\alpha^{L-1}),$$

which is 1 minus the probability that processes in which Σ does not hold are returned in all paths considered by $Q(\Sigma)$: $(1-\alpha)$ is the probability that $Q(\Sigma)$ fails at the process where it originated, and $(1-\alpha^{L-1})$ is the probability that $Q(\Sigma)$ fails in at least one of the remaining $L-1$ processes (i.e., $L-1$ paths of length 1).

Figures 5 and 6 show the variations in ϕ_1^i and ϕ_2^i, respectively, with the variation in L and α. As a reference, we have also included a curve with simulation results; details about our simulation are given in the next section. The simulation values shown in Figures 5 and 6 consider a completely connected network and processes with a probability of success equal to 0.1.

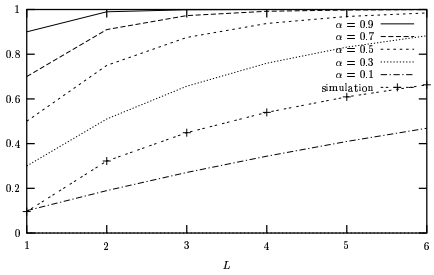

Fig. 5. ϕ_1^i in a fully connected graph

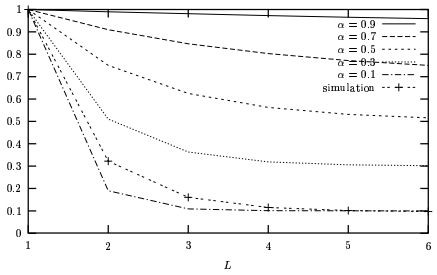

Fig. 6. ϕ_2^i in a fully connected graph

For high values of α (i.e., ≥ 0.5), ϕ_1^i quickly tends to 1. For low values of α, and given enough processes, ϕ_1^i can be reasonable high—notice that the probabilities are assumed to be independent. Conversely, ϕ_2^i is much more susceptible to variations in α than to variations in L. Actually, for $L \geq 3$, ϕ_2^i almost only depends on α. From Figures 5 and 6 we can conclude that PSEARCH is effective in scenarios where predicates hold with high probability in a subset of processes, and provided that the system evolves to identify what these processes are.

5.2 Simulation-Based Analysis

To better understand the behavior of PSEARCH, we have build a simulation model in C++ using the simulation package CSIM [7]. In the beginning of the execution each process is assigned a real probability of success, according to a certain distribution of probability, which determines the chances that a queried predicate holds at the process. This is the probability that processes try to determine using Bayesian statistical inference. Processes generate queries regularly and data starts to be collected once the local probability of success determined by processes become near the real values (i.e., around 5% of difference).

The impact of the probability distribution. Figures 7 and 8 compare different distributions of the real probabilities assigned to processes. To minimize the effects of other parameters, we considered a network completely connected and very large *s_table*s (i.e., able to store 50 processes). We have conducted experiments where all processes have a real probability of 0.1, a uniform distribution of real probabilities, and a powerlaw distribution of real probabilities. In a powerlaw distribution, a very few processes have a very high real probability of success, and most processes have a low real probability of success (i.e., in our experiments, only three processes have a real probability of success greater than 50%).

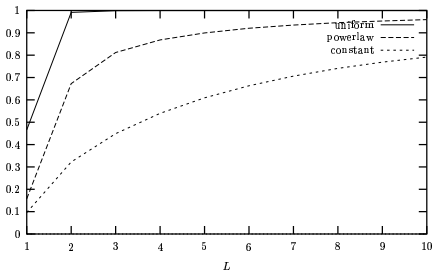

Fig. 7. ϕ_1^i in a fully connected graph

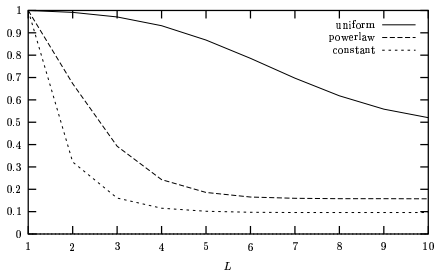

Fig. 8. ϕ_2^i in a fully connected graph

For the uniform and the powerlaw distributions, some processes have a high probability of executing queries with success. Such processes eventually end up in the *s_table*s of all processes, which explains the high values of ϕ_1 and ϕ_2

for these two distributions. Probability ϕ_2 decreases with the increase in the number of processes in the result because there are not so many processes with high probabilities of success, and those with high probability are included first in the result of queries. Therefore, when processes of low probability of success are included ($L \geq 7$) the chances that the queried predicate does not hold in all processes in the result raises.

The effect of failures. In these experiments, we consider a random network: We initially randomly generate links of varying latencies connecting processes and then take the biggest connected component as our network—processes not connected to the main component are discarded as are any connections involving them. To achieve a connected component with 100 processes, we interactively increased the initial number of processes until we reached 100 in the main component. The real probabilities of success are generated according to a powerlaw distribution, and each *s_table* can contain 10 processes. In Figures 9 and 10, θ represents the percentage of faulty processes. Faulty processes proceed in cycles: they execute for a certain period of time, crash and lose all the information they gathered, and recover. In the execution, each faulty process spends half of the simulation time up.

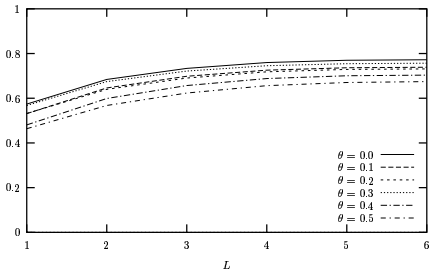

Fig. 9. ϕ_1^i in the presence of failures

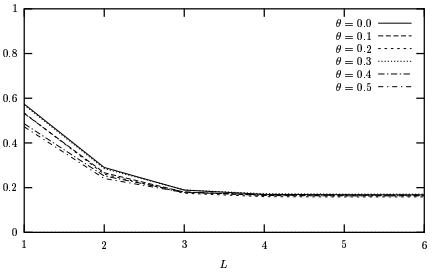

Fig. 10. ϕ_2^i in the presence of failures

Although the effects of process failures is not very significant on the values of ϕ_1 and ϕ_2, even the best case scenario (i.e., no failures) has low values of ϕ_1 and ϕ_2. This happens because with a powerlaw distribution, only very few processes have high real probabilities of success. As all processes periodically forward their *s_table*s to their neighbors, processes with high probability of success never last long in the *s_table*s—notice that in order to be able to get rid of faulty processes, we have to remove processes from *s_table*s based on their timestamps, and not on their probability of success. We discuss in next paragraph a way to improve ϕ_1 and ϕ_2.

Improving ϕ_1 and ϕ_2. To improve ϕ_1 and ϕ_2 and still be able to remove faulty processes, we modified our algorithm as follows: Initially, every process uses the same time interval to forward *s_table*s. After executing a query without success,

this value is increased (until it reaches some maximum threshold); after a query is executed with success by the process, the value is decreased (until it reaches some minimum threshold). Therefore, processes with very high probability will send their *s_table*s more frequently than processes with low probability, and so, will dominate the occupancy of *s_table*s. As depicted in Figures 11 and 12, this technique proved to be very effective. Moreover, the impact of failures on ϕ_1 and ϕ_2 is still small.

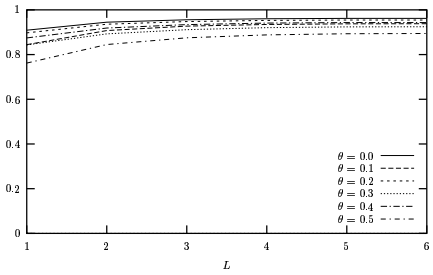

Fig. 11. ϕ_1^i with improved PSEARCH

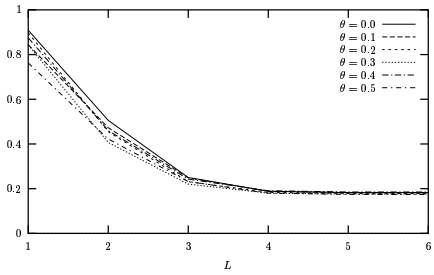

Fig. 12. ϕ_2^i with improved PSEARCH

6 Conclusion

This paper introduced the notion of probabilistic queries, an abstraction used to find resources and information (e.g., data files, processing capabilities) in large-scale systems. Contrary to deterministic solutions to the problem, probabilistic queries admit mistakes. The quality of a probabilistic query algorithm can be measured by two parameters: ϕ_1 and ϕ_2; the former is related to the chances that a query result contains a process of interest and the latter is related to the chances that "useless" processes are returned in the result of the query. Parameters ϕ_1 and ϕ_2 are complementary. Without ϕ_2, optimal queries could return all processes in the system; without ϕ_1, optimal queries could return no processes at all. The paper also presents PSEARCH, an algorithm that implements probabilistic queries using basic concepts of Bayesian statistical inference.

Preliminary results, by analytical and simulation models, show that if the system contains processes that concentrate most of the resources, an assumption that has been observed in some environments [10, 12], PSEARCH can be reasonable precise. PSEARCH is a promising way of dealing with the location problem in distributed system. Part of its power comes from its ability to adapt to system changes, that is, if the patterns of use chance over time, with some resources being more requested than others or some processes being more able to respond to request than others, PSEARCH adapts itself to new demands. We are currently working on a large set of experiments whose goal is to better understand the behavior of PSEARCH under various system loads and network partitions.

References

1. *Peer-to-peer: Harnessing the Benefits of a Disruptive Technology.* O'Reilly & Associates, Inc., 2001.
2. K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *9th International Conference on Cooperative Information Systems*, volume 2172 of *Lecture Notes in Computer Science*. Springer, 2001.
3. L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in power-law networks. Technical report, Hewlett-Packard Laboratories, 2001.
4. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the International Symposium on Distributed Computing (DISC'98)*, pages 231–245, September 1998.
5. P. Albitz and C. Liu. *DNS and BIND.* O'Reilly & Associates, 3rd edition, 1998.
6. K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.
7. *CSIM 18 simulation engine (C++ version).* Mesquite Software, Inc. 3925 W. Braker Lane, Austin, TX 78755-0306.
8. F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *8th IEEE Workshop on Hot Topics in Operating Systems*, May 2001.
9. M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. *Lecture Notes in Computer Science*, 1499, 1998.
10. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *Computer Communication Review*, 29(4), 1999.
11. I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'2001)*, August 2001.
12. B. A. Huberman and L. A. Adamic. Growth dynamics of the World-Wide Web. *Nature*, 401(6749), September 1999.
13. K. P. Birman I. Gupta, R. van Renesse. Scalable fault-tolerant aggregation in large process groups. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2001)*, July 2001.
14. J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, November 2000.
15. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
16. F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms.* Elsevier Science Publishers, 1989.
17. Open Directory Project. http://dmoz.org.
18. D. S. Sivia. *Data Analysis: A Bayesian Tutorial.* Oxford Science Publications, 1996.
19. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S.J. Mullender, J. Jansen, and G. van. Rossum. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12), December 1990.
20. S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. 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*, 2001.