Adaptive Peer Selection

Daniel S. Bernstein, Zhengzhu Feng, Brian Neil Levine, and Shlomo Zilberstein

Department of Computer Science, University of Massachusetts,
Amherst, Massachusetts 01003
{bern,fengzz,brian,shlomo}@cs.umass.edu

Abstract. In a peer-to-peer file-sharing system, a client desiring a particular file must choose a source from which to download. The problem of selecting a good data source is difficult because some peers may not be encountered more than once, and many peers are on low-bandwidth connections. Despite these facts, information obtained about peers just prior to the download can help guide peer selection. A client can gain additional time savings by aborting bad download attempts until an acceptable peer is discovered. We denote as peer selection the entire process of switching among peers and finally settling on one. Our main contribution is to use the methodology of machine learning for the construction of good peer selection strategies from past experience. Decision tree learning is used for rating peers based on low-cost information, and Markov decision processes are used for deriving a policy for switching among peers. Preliminary results with the Gnutella network demonstrate the promise of this approach.

1 Introduction

In a peer-to-peer file-sharing system, data is replicated among the peers participating in the system. Replicated data, while providing scalability and fault-tolerance, introduces the problem of source selection. After determining the locations of a desired file, a client must decide where to download from in order to receive the file quickly. We assume that only one peer can send data at a time, but our work can be extended to multi-source downloading (see the discussion section).

This problem has been studied mainly in the context of mirrored Web data, where it is called the *server selection* problem. Various solutions have been proposed and validated with experiments on the Internet [1–9]. However, many of the existing techniques rely on assumptions that render them inapplicable in the dynamic setting of peers. For instance, selection strategies based on experience with specific hosts do not apply when hosts are not likely to be encountered more than once. In addition, selection strategies that rely on network-layer assistance are not feasible.

In this paper, we introduce techniques for efficiently obtaining replicated content in peer-to-peer networks. We assume that the client has obtained a list of peers, each possessing a desired file. No assumptions are made about whether

the client has previously encountered any of the peers on the list. The client has access to limited information about its bandwidth to each of the peers. In addition, the client can perform partial downloads from peers before finally settling on one. We define as *peer selection* the problem of switching among peers and finally settling on one, while keeping the total time to a minimum.

The novel aspect of our approach to this problem is that it is based on the machine learning methodology from artificial intelligence. Our approach is *not* to introduce a new strategy for peer selection, but rather to introduce techniques by which a client can automatically derive a selection strategy based on its own experience. The strategies that are eventually produced are actually *adapted* to the client (hence the title).

The two phases of selection strategy execution proceed as follows. First, passively collected information is used to rate each peer on the list in terms of expected transfer time. In the Gnutella network, this information includes attributes from the search response messages regarding each peer's current load, current bandwidth, and past uploading experience. A *decision tree*, learned using data from previous downloads, rates never-before-seen peers based on attribute values. One benefit of using decision trees as a basis for the rating system is that they allow us to rate peers based on combinations of attributes. When individual attributes are unreliable, as is the case in peer-to-peer networks, this becomes important.

In the second phase of peer selection, the client uses the rating system to sort its list of peers, and then executes a policy for performing partial file downloads from the most promising ones and finally settling on a peer. An appropriate framework for deriving such a policy is the *Markov decision process (MDP)* framework. The peer selection process can be modeled as an MDP, the parameters of which are obtained from previous experience. The MDP can be solved for a selection policy that is optimal with respect to the model.

We implemented our ideas using the Gnutella network as our experimental platform. Downloading data was collected from four different client sites. Using this data, a different decision tree was learned for each of the clients. The resulting rating systems turned out to be fairly accurate in their predictions. In addition, we constructed an MDP for each client and solved each MDP for a complete peer selection strategy. Although mostly the same, the resulting policies did show some interesting differences across clients. We are in the process of evaluating the complete selection strategies with respect to other possible strategies.

As we mentioned, by focusing on peer selection, we address issues not dealt with in the classical server selection work. In addition, there been little research on combining attributes to yield better predictions. Two exceptions are [6], in which prior bandwidth and round-trip latency were combined, and [5], in which linear regression was used to combine round-trip latency and current available bandwidth. Finally, our focus on techniques for automatically learning a peer selection strategy from experience is novel.

2 Data Collection

Our machine learning approach requires that we have a set of training data from which to learn. To this end, a period of approximately two weeks was set aside to perform several downloads and record statistics about them. We note that, in practice, this training data could be produced as a byproduct of actual system use. This is left for future work and is discussed briefly in the final section of the paper.

For data collection, we modified a version of Gtk-Gnutella 0.85. The program was run at four client sites: University of Massachusetts in Amherst, University of Maryland in College Park, University of California in San Diego, and on an AT&T Broadband cable modem connection in Boston, Massachusetts. During this time the clients repeatedly attempted to download the first megabyte of randomly selected mp3 files from randomly selected peers. For each peer contacted, the following attributes were recorded from its search response message: an indication of whether all of the peer's upload slots were currently full (busy flag), an indication of whether the peer had successfully uploaded at least one file (uploaded flag), an indication of whether the peer was firewalled (firewall flag), a number representing the connection speed (speed field), and an indication of whether the speed field was measured or set by the user (measured flag). Note that the accuracy of this information is dubious, as different client programs have different policies for providing it. If the client was able to connect to the peer, the connection time was recorded, and if the client was able to download from the peer, the number of bytes received was recorded every 0.5 seconds.

Data was collected separately at each of the four clients. The derivation of each client's selection strategy, as described in the following sections, relied only on that client's data.

3 Rating Peers

In this section, we describe how to use training data to learn a decision tree for rating peers. We begin with a brief introduction to decision trees. A more thorough treatment can be found in [10].

3.1 Decision Trees

The decision trees we consider are used to approximate noisy binary-valued functions. The input to a decision tree consists of a set of attributes. In order to compute the output of a decision tree, one traverses down from the root, following the branches dictated by the attribute values, until a leaf is reached. Each leaf contains an output value. A decision tree can also be viewed as a list of if-then rules, one for each leaf. A decision tree is learned using a dataset of input-output pairs from the function. In our case, the inputs were search message attributes, and the output was an indication of whether the download from the corresponding source was fast (above the median speed) or slow (below

the median speed). Figure 1 shows an artificial dataset along with a decision tree that could have been learned from the dataset.

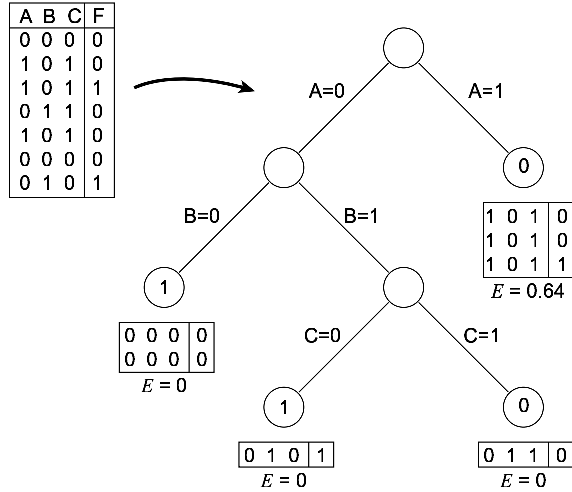

Fig. 1. A dataset along with a decision tree that could have been learned from the dataset.

Given a dataset, one desires a reasonably-sized decision tree that returns outputs with a high degree of certainty. In other words, for a given leaf, one would like most of the training instances associated with that leaf to agree on the output. Notice that in the decision tree in the figure, only one of the leaves contains a disagreement. One quantitative measure of the confidence of a leaf is the *entropy* of the leaf. The entropy of a leaf is defined as

$$E = -p_0 \log p_0 - p_1 \log p_1,$$

where p_0 is the fraction of instances with output 0, and p_1 is the fraction of instances with output 1.

3.2 Application to Rating Peers

Preliminary experiments revealed that the busy flag is strongly correlated with connection success. Thus we did not use the busy flag in the decision tree and decided to assign all instances with the busy flag set the lowest rating, B. Furthermore, we used only training downloads that completed successfully for learning a decision tree with the remaining four attributes.

We used the ITI decision tree algorithm [11] to learn a decision tree for each client site. The leaves of the decision trees are rated in the following way. First the leaves are sorted according to how many instances each contains. The bottom

50% are assigned the rank U (uncertain). These leaves contain too few instances to provide reliable information. Any of the remaining leaves with $E > 0.918$ are assigned rank U because of the high uncertainty inherent in them. For each of the remaining leaves, if the majority of the leaf's instances are above the median speed, and $E > 0.65$, then the leaf is assigned LF (likely fast). A similar rule holds for categorizing leaves as LS (likely slow). For the leaves still remaining, if the majority of the leaf's instances are above the median speed, then the leaf was assigned VLF (very likely fast). A similar rule holds for categorizing leaves as VLS (very likely slow). This completes the rating system (from least to most desirable): B, VLS, LS, U, LF, VLF.

Examining the resulting rating systems, we were able to extract some general rules. For the campus connections, a measured high speed along with a positive firewall flag were indicative of a fast download. The measured high speed makes sense, but there is not a clear explanation of why firewalled hosts would be faster; we speculate that "always on," high-speed hosts tend to employ firewalls more often, but we have no data to back this claim. Our cable modem was behind a firewall, which prevented it from downloading from firewalled peers (the Gnutella protocol does not allow this), and consequently rendered the firewall flag irrelevant. The best indicators of a fast download in this case seem to be previous upload success and a high measured speed. The indicators of a slow download seem to be that the peer has not successfully uploaded before and has a low value in its speed field.

Figure 2 provides evidence of the rating system's utility. It illustrates how rating correlates with download speed on test data (a small part of the dataset not used for training). We see that peers rated as VLF give significantly faster downloads than those with other ratings.

4 Peer Selection

In the preceding section, we showed how to construct a rating system for peers based on low-cost attributes. After sorting its list according to rating, the client can perform a sequence of partial downloads, eventually settling on a peer. Since partial downloads consume time, the client must proceed in an intelligent manner. It basically needs to make a sequence of good decisions, ending with the decision to commit to a peer.

An elegant framework for addressing this kind of sequential decision-making problem is the *Markov decision process (MDP)* framework [12]. An MDP models an agent acting in a stochastic environment with the aim of minimizing expected long-term cost. Our agent is the Gnutella client, and its long-term cost is the total time to obtain the first megabyte of a file, including the time for connection establishment and aborted downloads. To achieve this goal, it has a policy that indicates the situations in which it should abort its current download and start over with the next peer on the list. After describing the MDP framework in more detail, we show how an MDP can be constructed from training data and solved to yield a selection strategy.

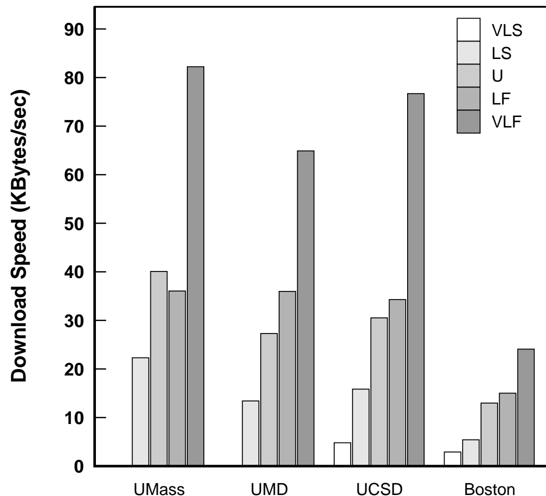

Fig. 2. The average download speed corresponding to the ratings for each client, measured on the test data. Note that the UMass and UMD decision trees had no VLS leaves.

4.1 Markov Decision Processes

We consider a type of MDP in which the agent tries to minimize the expected total time to reach a goal. The process proceeds through a sequence of stages $t = 0, 1, 2, 3, \ldots$ At each stage, the agent perceives the state of the process, $s_t \in S$, and chooses an action, $a_t \in A$. One stage later, the process produces a numerical cost, c_t, and a next state, s_{t+1}. This continues until a zero-cost absorbing state is reached, which indicates the attainment of the goal and the end of an *episode*.

Formally, an MDP is a tuple $\langle S, A, T, C \rangle$, where S is a finite set of states; A is a finite set of actions; $T(s, a, s')$ is a transition function representing the probability of transitioning from state s to state s' under action a; and $C(s, a, s')$ is a cost function giving the cost for executing action a in state s and transitioning to state s'. Actions are chosen according to a *policy*, $\delta : S \to A$. The *cost-to-go function*, $J_\delta(s)$, for a policy δ gives the expected (discounted) sum of future costs upon executing δ from state s. The aim is to find a policy δ that minimizes $J_\delta(s)$ for all states s. MDPs can be solved efficiently using dynamic programming [13].

4.2 Peer Selection as an MDP

We first provide a high-level description of the MDP model that is constructed from the training data. This MDP is an idealized model of the peer selection process, in which the client desires a 1 MB file and has access to an unlimited list of peers possessing the file. At any given time, it interacts with only one peer, denoted the *active peer*. The interaction consists of two phases. First is the

connecting phase, which lasts a maximum of 3 seconds. If no connection is made during that time, a new peer is randomly drawn from the list. If a connection *is* established, the *downloading* phase begins. After 3 seconds of downloading have passed, the client automatically commits to downloading the rest of the file from the currently active peer, and the episode is over. Failures can only occur during the connecting phase. For the first 3 seconds after a connection is made, the download speed may fluctuate, but it remains constant from the 3 second point on. Some aspects of our model may seem unrealistic; however, our objective is not realism but a model that can be solved quickly to yield an effective policy.

We now describe the model in more detail. The action set contains two elements. At each stage, the agent may either continue its download attempt with the active peer or start over with a new peer. In the connecting phase, an action is chosen every 0.5 seconds, and in the downloading phase, an action is chosen every 1.0 seconds. The states of the problem are as follows. The pre-connecting states,

$$P = \{\text{B}, \text{VLS}, \text{LS}, \text{U}, \text{LF}, \text{VLF}\},$$

indicate the rating of the active peer. The connecting states,

$$N = \{\text{B}, \text{VLS}, \text{LS}, \text{U}, \text{LF}, \text{VLF}\} \times \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0\},$$

indicate the rating of the active peer and how much time has passed since the active peer was first contacted. The downloading states,

$$D = \{\text{B}, \text{VLS}, \text{LS}, \text{U}, \text{LF}, \text{VLF}\} \times \{1.0, 2.0, 3.0\} \times \{(0\text{-}1), (1\text{-}2), (2\text{-}4), (4\text{-}8), (8\text{-}16), (16\text{-}32), (32\text{-}64), (64\text{-}128), (128\text{-}\infty)\},$$

indicate the rating of the active peer, the time spent downloading so far, and the (discretized) average speed so far (KB/sec). Finally we have an absorbing state, a, which is entered at the end of the downloading phase. Thus $S = P \cup N \cup D \cup \{a\}$, and $|S| = 205$.

Since the only cost in our problem is time, our cost function is relatively straightforward. A transition into a pre-connecting state incurs no cost. For transitions into connection and download states, the immediate costs are 0.5 and 1.0, respectively. Upon entry into the absorbing state, a final cost is incurred. This cost is the time that it would take to download the rest of a 1 MB file, assuming the speed from the first 3 seconds persists. Given this cost function, the total cost for an episode is equal to the time taken to download a complete 1 MB file, including the overhead for connection establishment and aborted downloads.

The parameters for the transition dynamics are derived using the training data. The initial pre-connecting state is drawn from a distribution that matches the distribution over ratings observed in the data. The rating component of the

state remains fixed until the episode ends or an abort action has been performed. At any stage, an abort action causes a transition to a new pre-connecting state, drawn from the distribution mentioned above. For transitions into connecting states, the probabilities are determined from the distribution over connect times in the training data. If the 3 second point of the connecting phase is reached, an automatic transition to a new pre-connecting state occurs on the next step. The probabilities for transitions into downloading states are determined from the training data. At the end of the downloading phase, there is a deterministic transition into the absorbing state.

4.3 The Resulting Policies

We solved each of the clients' MDPs using a dynamic programming algorithm. Not surprisingly, the policies are similar. In most cases, the policy aborts if no connection has been made in 0.5 seconds, or if a connection has been made but the speed at 1 second is below a threshold. The threshold is usually 32 KB/sec. For the cable modem connection, there are a few ratings for which the threshold is lower. This makes intuitive sense. As connection speed decreases, the client's connection becomes more of a bottleneck, and it makes less sense for the client to be choosy about which peer it downloads from. We conjecture that a modem would have a very low threshold.

There are some exceptions to the aforementioned policy rules. Some are difficult to explain and could be due to modeling assumptions and noise in the data. There is however, one apparently meaningful exception. If a peer is highly rated, policies are sometimes willing to wait longer to establish a connection. This makes sense, as it is worth investing extra time when the potential payoff is high.

We performed some preliminary experiments integrating the complete selection strategies into a client and using the client to obtain popular files. Our strategies are competitive with random strategies and strategies based on round-trip latency because they often quickly find peers with high bandwidth connections. A careful assessment of our strategies under a variety of conditions remains to be done.

5 Discussion

We have presented an approach to peer selection based on the machine learning methodology. Decision trees were used for learning peer rating systems, and our experiments showed the resulting rating systems to be accurate. The MDP framework was used for deriving policies for aborting downloads. These policies decide whether to continue or abort based on the state of the current download, so as to minimize the total time to receive the file.

By adding more information to the client's state, more sophisticated and better-performing policies should be possible. Information about peers other than the active peer should be useful—whether or not it is advantageous to

abort sometimes depends on whether other promising peers are available. In situations where a client will be downloading files of varying sizes, it may also be useful to base decisions on the size of the desired file. As files get larger, it is probably worth investing more time in trying to find a well-performing one. Finally, it may be a good idea to incorporate performance estimators such as hop count and round-trip latency into the MDP.

One natural question that arises from this work is whether the rating system and MDP can be updated online, based on data from actual system use. This is often referred to as *reinforcement learning* [14] in the artificial intelligence community. We see no major obstacles to doing this, and it has advantages over offline training. One advantage is that the time and congestion incurred by gathering large batches of training data are eliminated. In addition, an online approach allows for adaptation to changes in the Gnutella network and in local traffic conditions, which could in turn lead to better overall performance. Also interesting are *model-free* online learning techniques, which learn a policy without first constructing an MDP (see, e.g., [15]). These more direct approaches can alleviate some the burdensome assumptions implicit in an MDP model, but usually require more data.

In our study, we restricted the number of concurrent download attempts to be one. It has been shown, however, that parallel access to multiple sources can lead to lower transfer times [16, 17]. Indeed, many of today's peer-to-peer clients employ multi-source downloads. We believe that our methods can be extended to provide intelligent management of parallel transfers. A set of promising peers can be identified before the start of the download, and peers can be switched into and out of the set during the download. Intelligent parallel transfers should achieve maximal performance with fewer connections than naive parallel transfers, thus reducing congestion in the network. Some evidence of this is provided in [18], in which round-trip latency is used to select a set of servers for a parallel download.

Acknowledgments

We are grateful to Geoff Volker and Stephen Savage at UCSD and Samrat Bhattacharjee at UMD for providing the remote accounts used in our experiments. We also thank Zihui Ge for helping with our data collection. This work was supported in part by the National Science Foundation under grants IIS-9907331, ANI-0133055, and EIA-0080119. Daniel Bernstein was supported by a NASA GSRP Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or NASA.

References

1. Gwertzman, J., Seltzer, M.: The case for geographical push-caching. In: Proceedings of the 1995 Workshop on Hot Operating Systems. (1995) 51–55

2. Guyton, J.D., Schwartz, M.F.: Locating nearby copies of replicated Internet servers. In: Proceedings of SIGCOMM '95, Boston, MA (1995) 288–298
3. Yoshikawa, C., Chun, B., Eastham, P., Vadhat, A., Anderson, T., Culler, D.: Using smart clients to build scalable services. In: Proceedings of the First USENIX Symposium on Internet Technologies and Systems. (1997)
4. Sayal, M., Breitbart, Y., Scheuermann, P., Vigralek, P.: Selection algorithms for replicated web servers. Performance Evaluation Review **26** (1998) 44–50
5. Carter, R.L., Crovella, M.E.: On the network impact of dynamic server selection. Computer Networks **31** (1999) 2529–2558
6. Dykes, S.G., Robbins, K.A., Jeffery, C.L.: An empirical evaluation of client-side server selection algorithms. In: Proceedings of INFOCOM '00. (2000) 1361–1370
7. Stemm, M., Katz, R., Seshan, S.: A network measurement architecture for adaptive applications. In: Proceedings of INFOCOM '00. (2000)
8. Zegura, E.W., Ammar, M.H., Fei, Z., Bhattacharjee, S.: Application-layer anycasting: A server selection architecture and use in a replicated web service. IEEE/ACM Transactions on Networking **8** (2000) 455–466
9. Hanna, K.M., Natarajan, N., Levine, B.N.: Evaluation of a novel two-step server selection metric. In: Proceedings of IEEE International Conference on Network Protocols, Paris, France (2001)
10. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Mateo, CA (1993)
11. Utgoff, P.E., Berkman, N.C., Clouse, J.A.: Decision tree induction based on efficient tree restructuring. Machine Learning **29** (1997) 5–44
12. Puterman, M.L.: Markov Decision Processes. J Wiley & Sons, New York (1994)
13. Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton, NJ (1957)
14. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
15. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning **8** (1992) 229–256
16. Byers, J., Luby, M., Mitzenmacher, M.: Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In: Proceedings of INFOCOM '99. (1999)
17. Rodriguez, P., Biersack, E.W.: Dynamic parallel-access to replicated content in the Internet. IEEE/ACM Transactions on Networking **10** (2002) 455–464
18. Zeitoun, A., Jomjoom, H., El-Gendy, M.: Scalable parallel-access for mirrored servers. In: Proceedings of IASTED International Conference on Applied Informatics, Innsbruck, Austria (2002)