On a Graph-theoretic Approach to Scheduling Large-scale Data Transfers

William C. Cheng
Department of Computer Science
University of Maryland at College Park

Cheng-Fu Chou
Department of Computer Science
University of Maryland at College Park

Leana Golubchik
University of Maryland Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland at College Park

Samir Khuller
University of Maryland Institute for Advanced Computer Studies
and Department of Computer Science
University of Maryland at College Park

Yungchun (Justin) Wan
Department of Computer Science
University of Maryland at College Park

Abstract

In this paper we consider the problem of moving a large amount of data from several different hosts to a single destination in a wide-area network. Often, due to congestion conditions, the choice of paths by the network may be poor. By *choosing* an indirect route at the application level, we may be able to obtain substantially higher performance in moving data through the network. We formulate this data transfer (collection) problem as a network flow problem. We show that by using a min-cost flow algorithm on an appropriately defined time-expanded (network) graph, we can obtain a data transfer schedule. We show that such schedules can be an order of magnitude better than schedules obtained by transferring data directly from each host to the destination. In fact, this holds, even though we make no assumptions about knowledge of the topology of the network or the capacity available on individual links of the network. We simply use end-to-end type information and compute a schedule for transferring the data. Finally, we also study the shortcomings of this approach in terms of the gap between the network flow formulation and data transfers in a wide-area network.

1 Introduction

Large-scale data collection problems or *uploads* correspond to a set of important applications. These applications include online submission of income tax forms, submission of papers to conferences, submission of proposals to granting agencies, Internet-based storage, and many more. In the past, much research has focused on downloads or data dissemination applications; in contrast, large-scale wide-area uploads have largely been neglected. However, upload applications are likely to become significant contributors to Internet traffic in the near future, as digital government programs as well as other large scale data transfer applications take advantage of the proliferation of the Internet in society and industry. For instance, consider the online submission of income tax forms. US Congress has mandated that 80% of tax returns to be filed electronically by 2007. With (on the order of) 100 million individual tax returns filed in US yearly, where each return is on the order of 100 KBytes [17], scalability issues are a major concern.

Recently, a scalable and secure *application-level* architecture for wide-area upload applications was proposed [1]. This upload architecture is termed *Bistro*, and hosts which participate in this architecture are termed *bistro*s. Given a large number of clients that need to upload their data by a given deadline to a given destination server, Bistro breaks the upload process into three steps: (a) a timestamp step which ensures that the data is submitted on-time *without* having to actually transfer the data, (b) a data transfer step, where clients *push* data to intermediate hosts (bistros), which ensures fast response time for the clients, and (c) a data collection step, where a destination server (termed destination bistro) *pulls* data from bistros, i.e., the destination server determines how and when the data is transferred from the bistros. We note that during step (b) receipts corresponding to clients' transfers are sent by the (intermediate) bistros to the destination bistro; hence the destination bistro knows where to find all the data which needs to be collected during step (c). (A more detailed description of Bistro is included in the Appendix for completeness). *Performance of the data collection step is the focus of this paper.*

Specifically, we focus on the collection of reasonably large amounts of data, such as in the online tax submission example given above which can easily result in approximately 10 Terabytes of data corresponding to individual tax forms alone (business tax returns can be significantly larger). In such applications, long transfer times between one or more of the hosts (holding this data) and the destination server can significantly prolong the amount of time it takes to complete the data collection process. Such long transfer times can be the result of poor connectivity between a pair of hosts, or it can be due to wide-area network congestion conditions, e.g., due to having to transfer data over one or more (so-called) peering points whose congestion is often cited as cause of delay in wide-area data transfers [21]. Given the current state of IP routing, congestion conditions may not necessarily result in a change of routes between a pair of hosts, even if alternate routes exist.

Thus, we consider *application-level* approaches to improving performance of large-scale data collection. We do this in the context of the Bistro upload framework. However, one could consider other applications where such improvements in data transfer times is an important problem. One example is high-performance computing applications where large amounts of data need to be transferred from one or more data repositories to one or more destinations, where computation on that data is performed [9]. Another example is data mining applications where large amounts of data may need to be transferred to a particular server for analysis purposes.

Consequently, in this paper we consider large-scale data collection from a set of source hosts (bistros) to the destination host (destination bistro) where our *data collection problem* can be stated as follows.

Given

a set of source hosts, the amount of data to be collected from each host, and a common destination host for the data

our goal is to

construct a data transfer schedule which specifies on which path, in what order, and at what time should each "piece" of data be transferred to the destination host

where the objective is to

minimize the time it takes to collect all data from the source hosts, usually referred to as *makespan*.

The data collection problem is a non-trivial one because the issue is not only to avoid congested link(s), but to devise a *coordinated* transfer schedule which would afford maximum possible utilization of available network resources between multiple sources and the destination. We formulate this notion more formally in the remainder of the paper.

We note that the choice of the makespan metric is dictated by the applications stated at the beginning of this section, i.e., there is no clients in the data collection problem and hence metrics that are concerned with interactive response time (such as mean transfer times) are not of as much interest here. Since the above mentioned applications usually process the collected data, the total time it takes to collect it (or some large fraction of it) is of greater significance. We also note that in our case there is no need for a distributed algorithm for the above stated problem since Bistro employs a server pull approach, with all information needed to solve the data collection problem available at the destination server. Also not all hosts participating in the data transfer need to be sources of data; this does not change the formulation of our problem since such hosts can simply be treated as sources with zero amounts of data to send to the destination. In the remainder of the paper we use the terms hosts, bistros, and nodes interchangeably.

There are, of course, simple approaches to solving the data collection problem; for instance:

- transfer the data from all source hosts to the destination host in parallel, or
- transfer the data from the source hosts to the destination host sequentially in some order, or
- transfer the data in parallel from a constant number of source hosts at a time and possibly during a predetermined time slot,

as well as other variants. These methods are all "direct", in the sense that they send data directly from the source hosts to the destination host. In this paper, we show that "indirect" methods which re-route data through other hosts can result in a significant performance improvement as compared to "direct" methods (refer to Section 3 for details of direct methods used for comparison purposes). Consequently, our focus in this work is on development of *algorithms* for *indirect coordinated* transfer methods for the data collection problem.

Since we are focusing on application-level solutions, a path (in the above stated data collection problem) is defined as a sequence of hosts, where the first host on the path is the source of the data, intermediate hosts are other bistros in the system, and the last host on the path is the destination host. The transfer of data between any pair of hosts is performed over TCP/IP, i.e., the path the data takes between any pair of hosts is determined by IP routing.

Given the above stated problem, additional possible constraints include (a) ability to split chunks of data into smaller pieces, (b) ability to merge chunks of data into larger pieces, and (c) storage constraints at the hosts. To focus the discussion, we consider the following constraints. For each chunk of data we allow (a) and (b) to be performed only by the source host of that data and the destination host. We also do not place storage constraints on hosts but rather explore storage requirements as one of the performance metrics in evaluation of indirect methods (refer to Section 7).

We note that a more general problem where there are multiple destination hosts is also of significant importance, e.g., when the same set of bistros is simultaneously used for multiple upload applications or events. For the experiments in this paper, we only consider the single destination case, for ease of exposition. However, by employing multicommodity flow algorithms [2], rather than a single commodity min-cost flow algorithm (refer to Section 5) we can solve this problem as well. In other words, there is nothing about our approach that would fail for the case of multiple destinations for different pieces of data.

The contributions of this work are as follows. We propose novel algorithms, which we term "indirect" methods in contrast to the direct methods mentioned above, for the large-scale data collection problem defined above, intended for an IP-type network. The main benefit of these methods is application-level coordinated re-routing of large-scale data transfers around congestion spots or poor connectivity between a source of data and its final destination. We evaluate the performance of these algorithms in a simulation setting (using ns2 [15]). We show that the indirect methods perform significantly better than direct methods. Specifically we show one to two orders of magnitude improvement under high congestion conditions (without losses in performance under no congestion conditions). These improvements are achieved under low storage requirement overheads and without significant detrimental effects on other network traffic (refer to Section 7 for details).

We note that it is *not* the purpose of this work to propose novel techniques for identifying congestion conditions or determining available capacity or bottleneck link capacity. Rather, our goal is to propose algorithms for constructing data transfer schedules (as defined above) under the assumption that such techniques are (or will become) available (e.g., as in [4, 7]) and show that significant benefits can be gained from such algorithms.

The remainder of the paper is organized as follows. In Section 2 we briefly survey related work. Section 3 gives several simple direct methods for the data collection problem as described above; these are used for comparison purposes. Section 4 gives an overview of our approach for constructing indirect methods for solving this problem, and Sections 5 and 6 give the details of this approach. In Section 7 we give a quantitative evaluation of our approach to the data collection problem. Section 8 gives concluding remarks. (We also include an Appendix at the end of the paper, which briefly describes the Bistro architecture. We note that it is included for reviewers' benefit *only*.)

2 Related Work

As stated in Section 1, in this paper we focus on algorithms for large-scale data transfers over wide-area networks, in the context of upload applications. In [1] an application-level *framework* for large-scale upload applications, termed Bistro, is proposed. To the best of our knowledge no other large-scale *upload* architecture exists to date. Hence, we do this work in the context of Bistro, although as noted above, other types of applications can benefit as well. Specifically, we focus on the performance of the *data collection step* as described in Section 1 where our goal is to construct *algorithms* for *coordinated* data transfers from multiple source hosts to the destination host. Moreover, we focus on methods which re-route data, in a coordinated fashion, through other hosts at the application-level. Hence, below we briefly survey works which consider application-level re-routing issues.

Re-routing at the application level has been used to provide better end-to-end performance or efficient fault detection and recovery for wide-area applications. For instance, in [27] the authors perform a measurement-based study of comparing end-to-end quality of default routing vs alternate path routing (using metrics such as round-trip time, loss rate, and bandwidth). Their results show that in 30% to 80% of the cases considered, there is an alternate path with significantly superior quality. This work provides evidence for existence of alternate paths which can outperform default Internet paths.

Other frameworks or architectures which consider re-routing issues include Detour [26] and RON [3]. The Detour framework [26] is an informed transport protocol. It uses sharing of congestion information between hosts to provide a better "detour path" (via another node) for applications in order to improve the performance of each flow and the overall efficiency of the network. Detour routers are interconnected by using tunnels (i.e., a virtual point-to-point link); hence Detour is an in-kernel IP-in-IP packet encapsulation and routing architecture designed to support alternate-path routing. This work also provides evidence of potential long-term benefits of "detouring" packets via another node by comparing the long-term average properties of detoured paths against Internet-chosen paths.

The Resilient Overlay Network (RON) [3] is an architecture which allows distributed Internet applications to detect failure of paths (and periods of degraded performance) and recover fairly quickly by routing data through other (than source and destination) hosts. It also provides a framework for the implementation of expressive routing policies.

We note that the above mentioned re-routing works, for the most part, focus on architectures, protocols, and mechanisms for accomplishing application-level re-routing through the use of overlay networks. These works also provide evidence that such approaches can result in significant performance benefits. In this work, we consider a similar environment (i.e., application-level techniques in an IP-type wide-area network). However, in contrast we focus on *algorithms* for large-scale transfers when re-routing opportunities exist. Another important distinction here is that the above mentioned works do not consider *coordination* of multiple data transfers. That is, all data transfers are treated independently, and hence each takes the "best" application-level route available. In contrast, our work focuses on coordination of multiple data transfers destined for the same host. (As noted earlier, our approach can also be extended to multiple destination hosts.) As pointed out in Section 1, this additional consideration, which contributes to the difficulty of our data collection problem, is a result of the applications motivating this work.

3 Direct Methods

In this section we give the details of the direct methods used for comparison purposes (as described in Section 1). Specifically, we consider the following direct methods:

- *All-at-once*. Data from all source hosts is transferred simultaneously to the destination server. This continues until all transfers are complete.
- *One-by-one*. The destination server *randomly* selects one source host from a set of hosts which still have data to send; all data from that source host is then transferred to the destination server. Once this transfer completes, the destination server then randomly chooses another source host for transferring its data. This continues until all source hosts have their data transferred to the destination server.
- *Spread-in-time-GT*. The destination server chooses values for two parameters: (1) group size (G) and (2) time slot length (T). At the beginning of each time slot, the destination server *randomly* selects one group (of size G) and then the data from all source hosts in that group is transferred to the destination server; these transfers continue beyond the time slot length T, if necessary. At the end of a time slot (of length T), the destination server selects another group of size G and the transfer of data from that group begins regardless of whether the data transfers from the previous time slot have completed or not. (That is, data transfers which started during different time slots might overlap in time.) This continues until all source hosts have completed their data transfer.
- *Concurrent-G*. The destination server chooses a group size (G). It then *randomly* selects G of the source hosts and begins transfer of data from these hosts. The destination server always maintains a constant number, G, of hosts transferring data, i.e., as soon as one of these hosts completes its transfer, the destination server *randomly* selects another source host and its data transfer begins. This continues until the last source host completes its data transfer.

Clearly, there are a number of other direct methods that could be constructed as well as variations on the above ones. However, this set of direct methods is reasonably representative for us to make comparisons to indirect methods (refer to Section 7).

We note, that each of the above methods has its own shortcomings. For instance, if the bottleneck link is not shared by all connections, then direct methods which explore some form of parallelism in data transfer (such as the all-at-once method) might be able to better utilize existing resources and hence perform better than those that do not exploit parallelism (such as the one-by-one method). On the other hand, methods such as all-at-once might result in worse effects on (perhaps already poor) congestion conditions. Methods such as concurrent and spread-in-time require proper choices of parameters and their performance is sensitive to these choices.

Regardless of the specifics of a direct method, due to their direct nature, none of them are able to take advantage of network resources which are available on routes to the destination server other than the "direct" ones (i.e., those dictated by IP). Taking advantage of such resources can be especially important when the "direct" routes to the destination server are poor or congested. Indirect methods proposed in this paper are able to take advantage of such resources and therefore result in significantly better performance, as illustrated in Section 7.

4 Overview of Our Approach

An overview of our approach to the problem stated in Section 1 and the subsequent evaluation of that approach is as follows:

- *Step 1.* Construct a graph representation of the hosts and the corresponding communication network. This includes specification of nodes, connectivity between nodes, (estimated) capacities of the corresponding connections, and so on.
- *Step 2.* Generate a time-expanded version of the graph constructed in Step 1.
- *Step 3.* Determine a data transfer schedule on the time-expanded graph by optimizing a given objective function under the appropriate set of constraints. In this paper, we focus on the objective of minimizing the total amount of time it takes to collect the data from the source hosts, i.e., *makespan*, and we place some constraints on splitting and merging of data chunks.
- *Step 4.* Convert the solution produced in Step 3 under the graph theoretic formulation to a data transfer schedule for a communication network, taking into consideration the network protocols to be used for the transfers (e.g., TCP/IP). As stated in Section 1, this schedule must specify on what path and in what order should each "piece" of data be transferred to the destination host, where a path is defined as a sequence of hosts, with the first host on the path being the source of the data, intermediate hosts on the path being other hosts, and the last host on the path being the destination host.
- *Step 5.* Execute the data transfer schedule produced in Step 4 using ns2 [15] in order to evaluate the "goodness" of this data transfer schedule (i.e., this step is performed to evaluate our approach).

The details of Steps 1 through 3 are given in Section 5. The details of Step 4 are given in Section 6. Lastly, the details of Step 5 (as well as determination of parameters needed in Step 1) and the corresponding performance evaluation results are given in Section 7.

5 Graph Theoretic Formulation of the Data Collection Problem

We assume that the network topology is specified by a graph $G_N = (V_N, E_N)$. There are two kinds of nodes in the network, namely end-hosts and routers. The sources $S_1, \ldots S_k$ and destination D are a subset of the end-hosts. There is a capacity function c that specifies the capacity on the links in the network. In addition, background traffic exists, which effects the available capacity on the links.

In a wide-area network such as the Internet, we may not be aware of the exact topology of the entire network or the exact capacity function c. We will model the network by an overlay graph consisting of the set of source hosts and the destination host. (For ease of presentation below we discuss our methodology in the context of source hosts and destination host; however, any end-host can be part of the overlay graph, if it is participating in the Bistro architecture. In that case, the node corresponding to this host would simply have zero amount of data to send in the exposition below.) We refer to the overlay graph as $G_H = (V_H, E_H)$. The overlay graph is a directed (complete) graph where $V_H = \{S_1, \ldots, S_k\} \cup \{D\}$. (See Figure 1 for an example where we do not show outgoing edges from D since they are never used.) The capacity function models available capacity c' on each edge and is assigned as the bandwidth that is available for data transfer between end-hosts. (This takes into account the background traffic, but not any traffic that we are injecting into the network for the movement of data from the sources to the destination.) In other words, this is the bandwidth that is available to us on the path that the *network provides us* in the graph G_N, subject to the background traffic. Note that since we may not know the underlying topology or the routes that the paths take, we may not be able to properly model conflicts between flows. In other words, node S_2 may not simultaneously be able to send data at rate 1 to each of D and S_3 since the paths that are provided by the network share a congested link and compete for bandwidth. Such knowledge (if available) could be used to specify a capacity function on *sets* of edges, and one could use Linear Programming [5] to obtain an optimal flow under those constraints.

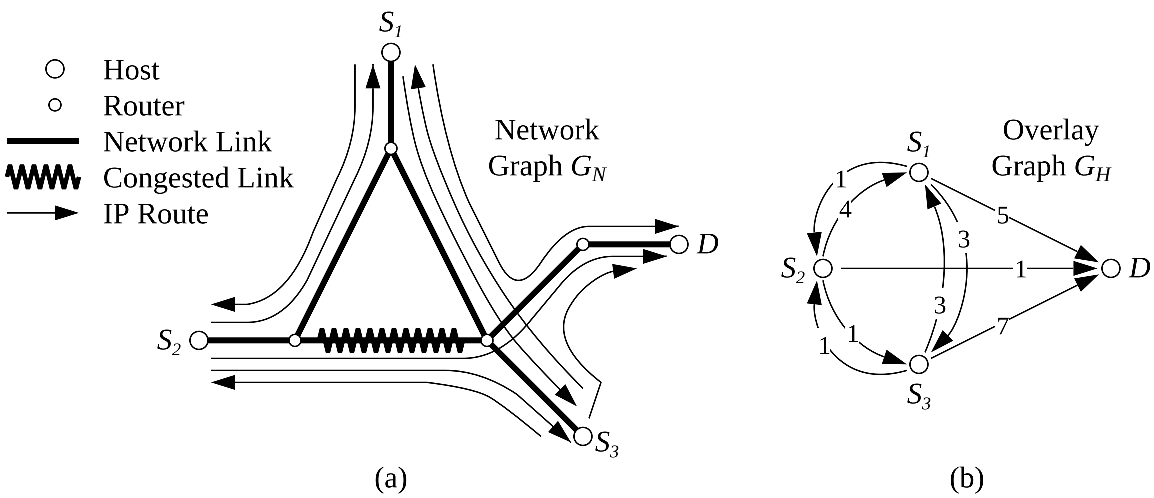

Figure 1: Network topology and the overlay graph.

From the overlay graph G_H we construct the "time-expanded" graph G_T [8, 13] (see Figure 2) which is the graph that our algorithms will use for computing a schedule to route the data from the sources to the destination. Given a non-negative integer T, we construct this graph as follows: for each node u we create a set of $T+1$ vertices $u(i)$ for $i = 0 \ldots T$. We pick a unit of time t (refer to Section 7 for the choice of t) and add edges in G_T from $u(i)$ to $v(i+1)$ with capacity $t \cdot c'(u,v)$.

(For example, suppose we have available capacity between u and v of 20 Kbps and define a unit of time t to be 2 seconds. In this case, we can transfer 40 Kb from u to v in "one unit of time".) We define the capacity of the edge from $u(i)$ to $v(i+1)$ as the amount of data that can be transferred from u to v in one unit of time. In addition, we have edges from $u(0)$ to $u(i)$ which are referred to as the "holdover" edges. This just corresponds to keeping the data at that node without sending it anywhere. We also add edges from $D(i)$ to a virtual destination D'. Each source $S_i(0)$ has a certain amount of flow available at time 0. All the flow has to be shipped to the virtual destination D'. Note that by *disallowing* edges from $u(i)$ to $u(i+1)$ for $i > 0$, we hold flow at the source nodes until it is ready to be shipped. In other words, flow is sent from $S_2(0)$ to $S_2(1)$ and then to $S_1(2)$, rather than from $S_2(0)$ to $S_1(1)$ to $S_1(2)$ (which is not allowed since there is no edge from $S_1(1)$ to $S_1(2)$). This has the advantage that the storage required at the intermediate nodes is lower. Hoppe and Tardos [14] argue that allowing edges of the form $u(i)$ to $u(i+1)$ does not decrease the minimum value of T.

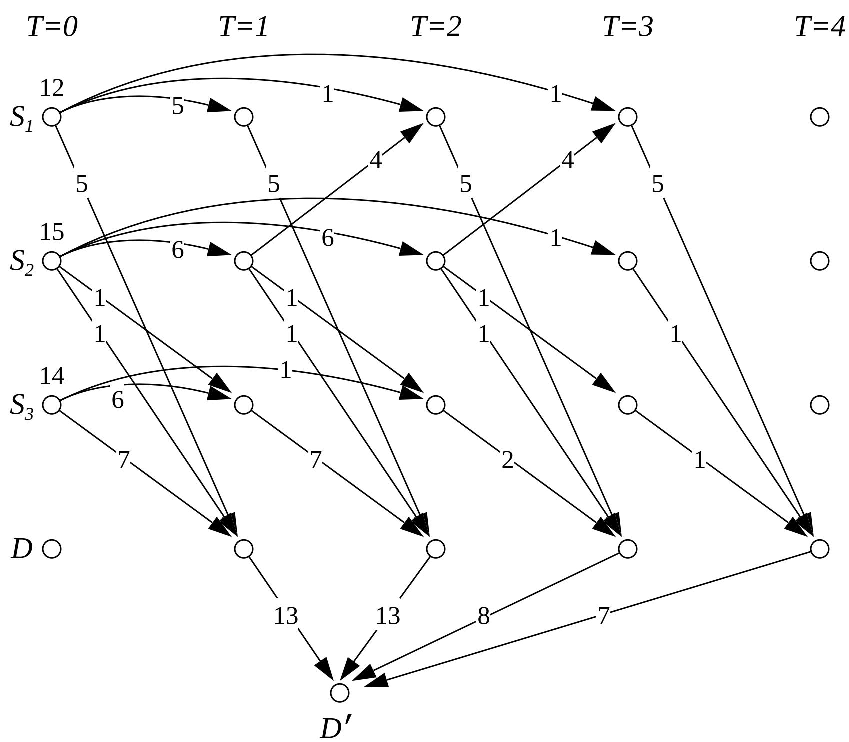

Figure 2: Time-expanded graph.

Our first goal then is to compute the minimum value T such that we can route all the data to the destination D'. We can find T, in $O(\log T)$ time by doing a "doubling" search, followed by a binary search once we find an interval that contains the minimum T for which a feasible solution exists.

Once we find the minimum value T, we find a min-cost flow [2] in the graph as follows. We associate a cost of $C_1 - C_2 \cdot c'(u,v)$ with every transfer edge $u(i)$ to $v(i+1)$, where C_1 and C_2 are constants and $C_1 \gg C_2 \gg 1$. Our solution would prefer sending data over high capacity links if two solutions have the same total number of transfers. This also provides a more regular pattern in the flow solution (which can be useful in the PathMerge algorithm described in Section 6). To every holdover edge $u(0)$ to $u(i)$, we assign a cost of i. This ensures that data is sent as soon as possible. In other words, once we find the minimum T, there could be several feasible flows that route the data. Among such flows we prefer the ones with the property that the data arrives earlier at D'. The cost is 0 for all other edges. The cost function is chosen so as to obtain a feasible flow with certain properties, as opposed to an arbitrary flow. Imposing costs simply guides the solution so that we find a solution that sends the flow earlier rather than later and prefers larger capacity edges to smaller capacity edges. Modifications to this cost function can be made if other properties are desired (e.g., based on network and/or protocol characteristics).

In our problem T is not very large so it is feasible to build the entire time-expanded graph and to run a min-cost flow algorithm on it. If T is extremely large, then one could use other algorithms (see Hoppe and Tardos [13, 14]) which are faster.

The min-cost flow algorithm computes a flow function f_T that specifies the flow in the time expanded graph G_T. (For computing the min-cost flow, we used Goldberg's code [11, 12].) We now need to convert this flow into a schedule for transferring data in the real network.

Remark 1: An alternative to our formulation above is to use the overlay graph, G_H, to compute the "best" path in G_H from each host to the desination, independently. For instance, S_2 may choose the path (S_2, S_1, D) since it is the maximum capacity path to D, and send all of its data along this path. This alternative would correspond to the application level re-routing mechanisms surveyed in Section 2, and hence we refer to it as ALR below. However, note that this option does *not* permit for (a) any coordination between transfers from different source hosts, (b) explicit load balancing as each node makes its own decision as to which route to send the data on, and (c) maximum possible utilization of available network resources between a source and the destination. More formally, in our time-expanded graph, ALR corresponds to a feasible flow in a graph G_{T_i} for some T_i. Note that in fact $T_i \geq T_{\min}$ where $T_{\min}$ is the solution obtained by our algorithm, which allows for sending of data along multiple paths between a source and the destination. In fact, by sending the data along several paths, our algorithm obtains a better solution than ALR (as described above). This difference becomes especially significant, if several good application level routes exist, but ALR strategies send their data along the "best" path, thus causing congestion along this path.

Remark 2: Note that in our formulation, we compute the capacity function once initially (refer to Section 7), to estimate the available capacity between pairs of hosts. Once we do this we will assume this as the available bandwidth for the entire duration of the transfer. Of course, if the transfer is going to take a long time, we cannot assume that the network conditions are static. In this case, we can always compute a new estimate of available bandwidth during the scheduling of the transfer and compute a new transfer schedule for the remaining data. (Our algorithm itself is very fast, and so this does not cause a problem even if the current transfer is stopped, and the schedule is changed.) In fact, the algorithm itself can detect when transfer times are not behaving as predicted and compute a new estimate of capacities.

Also note that we are dealing with the network at the application layer, where we can control the route within the *overlay* network that the data takes to the destination without any change to the network protocols (such as IP or TCP).

Finally, the formulation above is quite robust and we can use it to model situations where data may be available at different sources at different times.

6 Construction of Network Transfer Schedule

What remains is to construct a data transfer schedule, f_N (defined as the goal of our data collection problem in Section 1), from the flow function f_T computed in Section 5, while taking into consideration characteristics of wide-area networks such as the TCP/IP protocol used to transfer the data. This conversion is non-trivial partly due to the discrepancies between the graph theoretic abstraction used in Section 5 and the way a TCP/IP network works. (Below we assume that each data transfer is done using a TCP connection.)

One such discrepancy is the lack of variance in data transfers in the graph theoretic formulation, i.e., a transfer of X units of data always takes a fixed amount of time over a particular link. This is not the case for data transferred over TCP in a wide-area network, partly due to congestion characteristics at the time of transfer and partly due to TCP's congestion avoidance mechanisms (e.g., decreases in sending rate when losses are encountered). Another discrepancy in the graph theoretic formulation is that it does not matter (from the solution's point of view) whether the X units are transferred as a single flow, or as multiple flows in parallel, or as multiple flows in sequence. However, all these factors affect the makespan metric when transferring data over TCP/IP. Again, these distinctions are partly due to TCP's congestion avoidance mechanisms.

Thus, we believe that the following factors should be considered in constructing f_N, given f_T: (a) size of each transfer, (b) parallelism in flows between a pair of hosts, (c) data split and merge constraints, and (d) synchronization of flows. In this paper, we propose several different techniques for constructing f_N from f_T, which differ in how they address issues (a) and (d). We first give a more detailed explanation of these issues and then describe our techniques. Note that, we use the term "transfer" to mean the data transferred between two hosts during a single TCP connection.

Size of each transfer.
If the size of each transfer is "too large" we could unnecessarily increase makespan due to lack of pipelining in transferring the data along the path from source to destination (in other words, increased delay in each stage of the indirect routing). For example, suppose f_T dictates a transfer

of 100 units of data from node S_2 to S_3 to D. S_3 does not start sending data to D until all 100 units of data from S_2 have arrived. If the size of each transfer is 10 units, S_3 can start sending some data to D after the first 10 units of data have arrived. On the other hand, if the size of each data transfer is "too small" then the overheads of establishing a connection and the time spent in TCP's slow start could contribute significantly to makespan.

In this work, we address the "too small" problem in two ways. First, we ensure that each transfer is of a reasonably large size by carefully picking the time unit and data unit size parameters in the graph construction step (refer to Section 7 for details). Second, we provide a mechanism for merging data transfers which are deemed "too small" (details given below in the PathMerge algorithm). The "too large" problem is addressed by a proper choice of the time unit parameter (as described in Section 7).

Parallelism between flows.

One could try to obtain a greater share of a bottleneck link for an application by transferring its data, between a pair of hosts, over multiple parallel TCP connections. However, we do not explore this option here, mainly because it is not as useful (based on our simulation experiments) in illustrating the *difference* between the direct methods and the indirect methods since both types of methods can benefit from this. In fact, we made a comparison between the all-at-once method employing parallel connections and our indirect methods *without* parallel connections, and the result was that indirect methods could still achieve an order of magnitude better performance.

Data split and merge constraints.

The f_T solution of Section 5 allows for arbitrary (although discrete) splitting and merging of data being transferred. However, in a real implementation, such splitting and merging (of data which represents uploads coming from many different clients) can be costly. For instance, in the income tax submission forms example, if we were to arbitrarily split a user's income tax forms along the data transfer path, we would need to include some meta-data which would allow piecing it back together at the destination server. Since there is a cost associated with splitting and merging of data, in this paper we allow it only at the source of that data and the destination, i.e., we do not allow intermediate hosts, through which the data is transferred, to split or merge data. To ensure this constraint is met, the first step in our f_N construction techniques is to decompose f_T into flow paths (see details below).

Evaluation of potential additional benefits of splitting and merging is ongoing work. For instance, if we do not want to allow any splitting of the data, we could consider formulating the problem as an unsplittable flow problem. Unfortunately, unsplittable flow problems are NP-complete [19]. Good heuristics for these have been developed recently, and could be used [6].

Synchronization of flows.

The f_T solution of Section 5 essentially synchronizes all the data transfers on a per time step basis, which leads to proper utilization of link capacities. This synchronization comes for free given our graph theoretic formulation of the data collection problem. However, in a real network, such synchronization will not occur naturally. In general, we could implement some form of synchronization in data transfers at the cost of additional, out-of-band, messages between bistros. Since the Bistro architecture employs a server pull of the data (refer to Section 1), this is a reasonable approach, assuming that some form of synchronization is beneficial. Thus, in this paper we explore the benefits of synchronization.

Splitting the flow into paths.
Given that splitting and merging of data is restricted, we now give details of decomposing f_T into paths, which is the first step in constructing f_N from f_T. To obtain a path from f_T, we traverse the time-expanded graph (based on f_T) and construct a path from the nodes we encounter during the traversal as follows. We start from a source host which has the smallest index number. Consider now all hosts that receive non-zero flows from it. Among those we then choose the one with the smallest index number, and then proceed to consider all hosts that receive non-zero flows from it. We continue in this manner until the virtual destination is reached. The data transfered over the resulting path p is the maximum amount of data that can be sent through p (i.e., the minimum of flow volume over all edges of p). We note that a path specifies how a fixed amount of data is transfered from a source to the destination. For example (see Figure 2), a path can be specified as $(S_2(0), S_2(1), S_1(2), D(3), D')$, which says that a fixed amount of data is transferred from node S_2 to node S_1 at time 1, and then from node S_1 to the destination D at time 2 (and D' is the virtual destination). In fact, for this path the value of the flow is 4.

To split the flow network into paths, we first obtain a path using the procedure described above. We then subtract this path from f_T. We then obtain another path from what remains of f_T and continue in this manner until there are no more flows left in f_T. At the end of this procedure, we have decomposed f_T into a collection of paths. (An example of this flow decomposition is given under the description of the PathSync algorithm below and in Figure 3.)

Imposing Synchronization Constraints.
What remains now is to construct a schedule for transferring the appropriate amounts of data along each path. We propose the following methods for constructing this schedule which differ in how they attempt to preserve the time synchronization information produced by the time-expanded graph solution given in Section 5.

The PathSync Method.
In this method we employ complete synchronization as prescribed by the time-expanded graph solution obtained in Section 5. That is, we first begin all the data transfers which are supposed to start at time step 0. We wait for all transfers belonging to time step 0 to complete before beginning any of the transfers belonging to time step 1. When all transfers from time step 0 complete, we begin all transfers from time step 1. We continue in this manner until all data transfers in the last time step are complete. We term this approach *PathSync100* (meaning that it attempts 100%

synchronization as dictated by f_T).

Recall that the capacity of an edge in the time-expanded graph is the volume of data that can be sent over it during one time unit. Since estimates of available capacity may not be accurate (refer to Section 7), and since we may not know which transfers do or do not share the same bottleneck link (unless, e.g., we employ techniques in [25]), it is possible, that some transfers may take a significantly longer time to finish than dictated by f_T. Given the strict synchronization rules above, one or two slow transfers could greatly affect makespan. An alternative is to synchronize only $X\%$ of the transfers. That is, as long as a certain percentage of the data transfers have completed, we can begin all the transfers corresponding to the next time step, except, of course, those that are waiting for the previous hop on the same path to complete. We term this alternative *PathSyncX* where X indicates the percentage of transfers needed to satisfy the synchronization constraints, e.g., *PathSync90* requires that 90% of transfers from time step i to complete before transfers from time step $i+1$ can begin.

An example of PathSync is depicted in Figure 3 which shows a collection of paths obtained from decomposing f_T. At time step 0, PathSync100 starts the transfer from $S_1(0)$ to $D(1)$, $S_2(0)$ to $S_3(1)$, $S_2(0)$ to $D(1)$, and $S_3(0)$ to $D(1)$, since all these transfers belong to time step 0. When all these transfers have finished, PathSync100 starts the transfers belonging to time step 1, namely $S_1(1)$ to $D(2)$, $S_2(1)$ to $S_1(2)$, $S_2(1)$ to $S_3(2)$, etc.

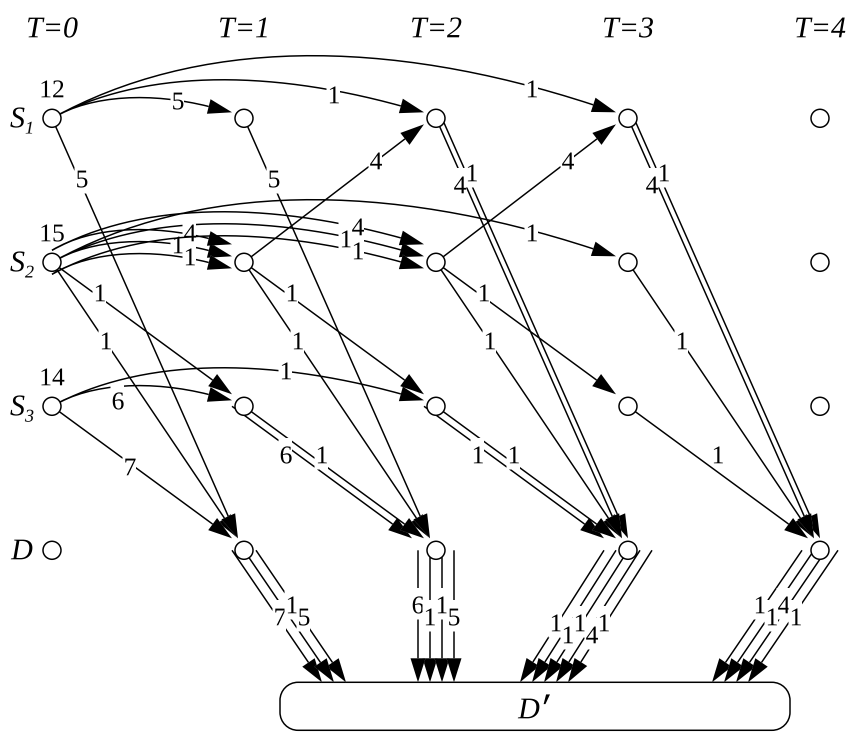

Figure 3: Solution obtained after flow decomposition.

We will show in Section 5 that the PathSync method performs quite well, especially when the percentage of transfers that satisfy the synchronization requirements is a bit lower than 100%. This is an indication that it is worth while to attempt to preserve the timing constraints prescribed by the solution of the time-expanded graph (as long as these benefits are not subsumed by the harmful effects of potentially high variance in the transfers). Since synchronization between bistros is not free in a real implementation, we also consider a method which does not require it.

The PathDelay Method.
In the *PathDelay* method we do not attempt any synchronization between transfers once a transfer along a particular path begins. That is, as long as a particular data transfer along one hop of a path completes, the transfer of that data begins along the next hop of that path. The only synchronization performed in this method is to delay the transfer of that data from the *source* node until an appropriate time, as dictated by f_T. For example, after the decomposition of f_T into paths, there is a path $(S_2(0), S_2(2), S_1(3), D(4), D')$ of size 4 (see Figure 3). Since the data is held at the source S_2 until time step 2 in f_T, we schedule the $S_2(2)$ to $S_1(3)$ transfer at "real" time $2 \cdot t$, where t is our time unit (refer to Section 7).

One could also create variations on PathDelay by expanding or contracting the time unit, used in computing f_T, when constructing f_N, again to account for variance in data transfer in a real network as compared to the graph theoretic formulation. For instance, *PathDelayX* would refer to a variation where the time unit t in f_T is modified to be Xt in f_N.

The PathMerge Method.
We consider one more variant in construction of f_N as compared to *PathDelay*. We first observe that after we split f_T into paths, some paths may visit exactly the same sequence of hosts, but at different time steps. For instance, in Figure 3 we have a path carrying 1 unit of flow from $S_2(0)$ to $S_3(1)$ to $D(2)$, and another path carrying one unit of flow from $S_2(1)$ to $S_3(2)$ to $D(3)$. Since these two paths are transferring the flow along the same path, we could combine them into a single transfer if the amount of data in each one is too small. We call two such paths *consecutive* since the data is traveling on the same route, just shifted in time by 1 unit of time. *PathMerge* merges all consecutive paths before initiating data transfers. After that it behaves just like PathDelay. An example of PathMerge is given in Figure 4. Consider S_1 which sends 5, 5, 1, and 1 units of data to D at time steps 0, 1, 2, and 3, respectively (refer to Figure 3). Since these are consecutive paths, PathMerge merges all of them into a single transfer of size 12, which starts at time 0 from S_1 to D. Moreover, note that path $(S_2(0), S_2(1), S_1(2), D(3), D')$ of size 4 and path $(S_2(0), S_2(2), S_1(3), D(4), D')$ of size 4 are also consecutive (also in Figure 3). PathMerge merges them into a path of size 8, which starts at time step 1, from S_2 to S_1 to D.

We have observed from our simulation experiments that data transfer sizes can effect the makespan metric in a TCP/IP network (e.g., when these sizes are "too small' as described above). In general, one might try to optimize the size of each data transfer, after obtaining f_T, and the PathMerge technique facilitates such optimization. For instance, such an optimization might be

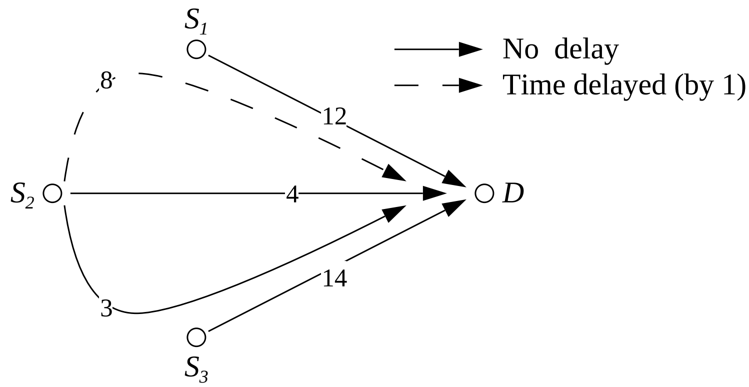

Figure 4: Solution obtained after merging paths.

done by taking into consideration the path taken by the transfer and through the use of TCP throughput equations [24] or possibly by using the pipelining optimization work in [30]. Once an appropriate data size is obtained, the PathMerge technique can be used to merge consecutive paths until a proper data transfer size is reached (i.e., the aggressiveness of PathMerge can be controlled). This is an ongoing effort. Current experiments show that only if the volume of data is too small, or too large, this causes an increase in the makespan since typically, we are re-routing the data through relatively few intermediate hosts.

Note that we do not perform path merging in conjunction with the PathSync technique since merging of paths along the time dimension and synchronization of transfers are essentially opposing goals.

7 Validation and Performance Evaluation

In this section we evaluate the performance of direct and indirect methods to illustrate the benefits of using indirect approaches. (Refer to Section 3 for a detailed description of direct methods; refer to Sections 4 through 6 for a detailed description of our indirect methods.) This evaluation is done through simulation; all results are given with at least $90\% \pm 10\%$ confidence.

Experimental Setup

We use ns2 [15] for all simulation results reported below. In conjunction with ns2, we use the GT-ITM topology generator [16] to generate a transit-stub type graph for our network topology. Specifically, we use GT-ITM to create a transit-stub graph with 152 nodes. The number of transit domains is 2, where each transit domain has, on the average, 4 transit nodes with there being an edge between each pair of nodes with probability of 0.6. Each node in a transit domain has, on the average, 3 stub domains connected to it; there are no additional transit-stub edges and no additional stub-stub edges. Each stub domain has, on the average, 6 nodes with there being an edge between every pair of nodes with probability of 0.2. A subset of our simulation topology (i.e., without stub domain details) is shown in Figure 5. The capacity of a “transit node to transit

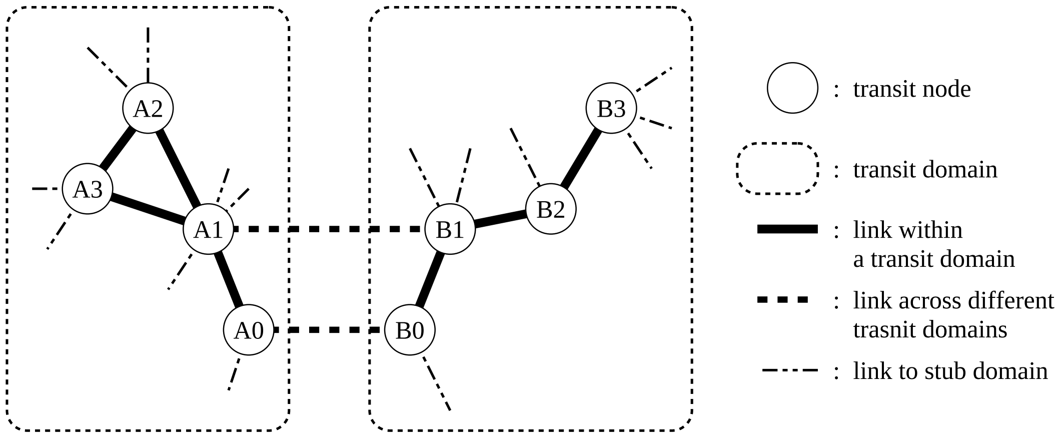

Figure 5: The simulation topology.

node" edge within the same transit domain is 1 Mbps. The capacity of a "transit node to transit node" edge across different transit domains is 512 Kbps. The capacity for a "transit node to stub node" edge or a "stub node to stub node" edge is 256 Kbps. Our motivation for assigning a lower capacity to the "transit node to transit node" edge across different transit domains is to emulate poorer performance conditions that exist at the so-called peering points [21]. Note that, the size and parameters of our network model and the following experimental setup are motivated by what is practical to simulate with ns2 in a reasonable amount of time. However, since our goal is to illustrate benefits of indirect methods, this will suffice.

We locate the destination server in the stub domain connected to $A1$, and we locate 7 other bistros in stub domains connected to other transit nodes. Each bistro holds a total amount of data which is uniformly distributed between 25 MBytes and 75 MBytes with an additional constraint that the total amount of data in all bistros is 350 MBytes. In addition to the upload traffic, we generate background traffic consisting of infinite ftp flows, where the number of such flows is varied from 0 (i.e., no background traffic) to 120. The infinite ftp flows all traverse the $B1$ to $A1$ link. We choose this simple congestion pattern for clarity of illustration. Let x be the number of infinite ftp flows in a particular experiment. Then, the background traffic is generated by randomly choosing x source stub domain nodes (connected to either $B1$, $B2$, or $B3$ transit nodes) as well as x destination stub domain nodes (connected to either $A1$, $A2$, or $A3$ transit nodes) to participate in the infinite ftp flows. To illustrate a reasonably interesting scenario, all nodes participating in background traffic are located in stub domains that are different from those holding the bistros participating in upload traffic. This choice avoids the non-interesting cases (at least for makespan) where a single bistro ends up with an extremely poor available bandwidth to *all* other bistros (including the destination server) and hence dominates the makespan results (regardless of the data transfer method used).

Construction of Corresponding Graph

We now give details of constructing graph G_H of Section 5 from the above network. The eight bistros make up the nodes of G_H, with the destination bistro being the destination node (D) and the remaining bistros being the source nodes (S_i) with corresponding amounts of data to transfer. The link capacities between any pair of nodes in G_H is determined by estimating the end-to-end mean TCP throughput between the corresponding bistros in the network. In our experiments these throughputs are estimated in a separate simulation run, by measuring the TCP throughput between each pair of bistros while sending a 2 MByte file between these bistros. These measurements are performed with background traffic conditions corresponding to a particular experiment of interest but without any upload traffic or measurement traffic corresponding to other bistro pairs. We do this in order to have a reasonably accurate and simple estimate of congestion conditions. However, we note, that it is *not* our intent to advocate particular measurement and available bandwidth estimation techniques. Rather, in a real implementation, we intend to use whatever techniques are available at the time, such as: (a) computing link bandwidth by actively sending probe packets [7, 18, 22, 4], (b) computing link bandwidth through packet pairs and potential bandwidth filtering techniques [20], or (c) obtaining link bandwidth information by keeping some shared passive measurements [29]. Potential sources of such information in the future might also be services such as SONAR [23], Internet Distance Map Service (IDMaps) [10], Network Weather Service (NWS) [31], and so on.

In order to construct G_T from G_H we need to determine the time unit and the data unit size. The bigger the time unit is, the less costly is the computation of the min-cost flow solution but potentially (a) the less accurate is our abstraction of the network (due to discretization effects) and (b) the higher is the potential for large transfer sizes (which in turn contribute to lack of pipelining effects as discussed in Section 6). The smaller the time unit is, the greater is the potential for creating solutions with transfer sizes that are "too small" to be efficient (as discussed in Section 6). Similarly, the data unit size should be chosen large enough to avoid creation of small transfer sizes and small enough to avoid significant errors due to discretization (as discussed in Section 6).

In the experiments presented here we use a time unit which is an order of magnitude larger than the maximum round trip time (RTT) on the longest path, as generated by the GT-ITM topology generator [16] (we note that since we use GT-ITM these delay parameters are *not* under our control). Specifically, in the experiments of this section, this RTT is 6.9 sec, and hence we use a time unit of 69 sec. The data unit size is chosen to ensure that the smallest transfer is large enough to get past the slow start phase and reach maximum available bandwidth without congestion conditions. Since without background traffic a bistro can transmit at a maximum window size of 256 Kbps $\times$ 6.9 sec (on the longest path), we use a data unit size a bit larger than that, specifically 256 KBytes.

Performance Metrics.
The performance metrics used in the remainder of this section are: (a) makespan and makespan$X$, i.e., the time needed to complete transfer of at least X percent of the total amount of data from all bistros, (b) maximum storage requirements averaged over all bistros (not including the destination bistro since it must collect all the data), and (c) mean throughput of background traffic during the

data collection process, i.e., we also consider the effect of upload traffic on other network traffic. We believe that these metrics reflect the quality-of-service characteristics that would be of interest to large-scale data collection applications. (As noted in Section 1, we do not consider the mean bistro transfer times since there are no clients in the data collection problem and hence interactive response time related metrics are not an issue.)

Evaluation Under the Makespan Metric.

We first evaluate the direct methods described in Section 3 using the makespan metric. As illustrated in Figure 6(a) direct methods which take advantage of parallelism in data delivery (such as all-at-once) perform better under our experimental setup. Intuitively, this can be explained as follows. Given the makespan metric, the slowest bistro to destination server transfer dominates the

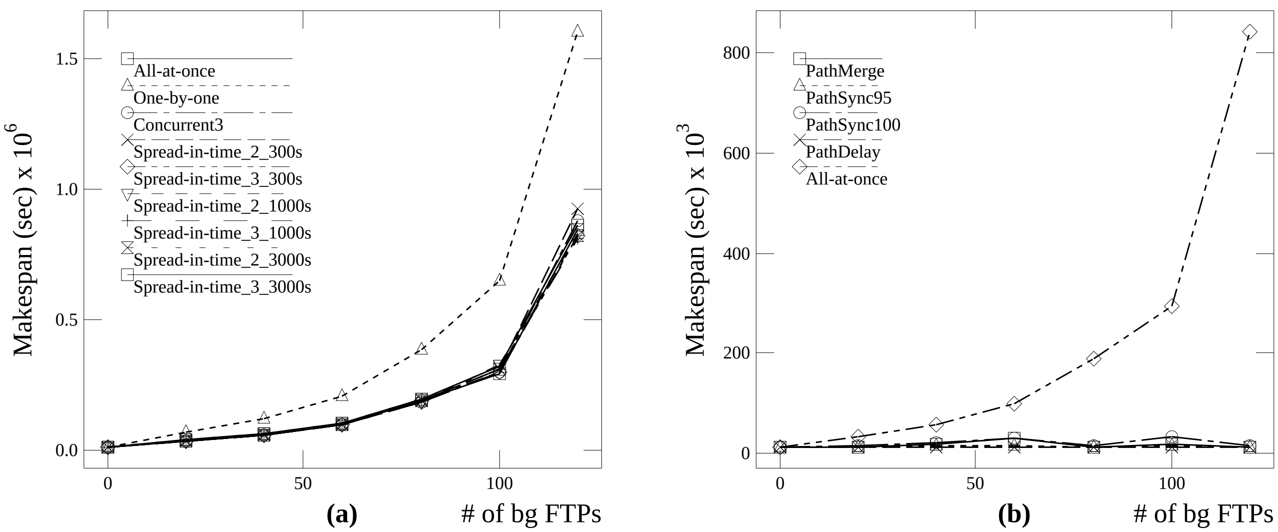

Figure 6: Direct and Indirect Methods under the Makespan Metric.

makespan metric. Since in our case, the bottleneck which determines the *slowest* transfer in direct methods is not shared by all bistros, it makes intuitive sense to transfer as much data as possible, through bottlenecks which are different from the one used by the slowest transfer, in parallel with the slowest transfer.

Since all-at-once is a simple method and it performs better than or as well as any of the other direct methods described in Section 3 under the makespan metric in our experiments, we now compare just the all-at-once method to our indirect methods (as described in Sections 4 through 6). This comparison is illustrated in Figure 6(b) where we can make the following observations. All schemes give comparable performance when there is no other traffic in the network (this makes intuitive sense since the capacity near the server is the limiting resource in this case). When there is congestion in the network and some bistros have significantly better connections to the destination server than others, our indirect methods do result in a significant improvement in performance, especially as this congestion (due to other traffic in the network) increases. For instance, in Figure 6(b) we observe improvements from more than 2 times under 20 background flows as high as 75 times when the background traffic is sufficiently high (in this case at 120 flows).

It is difficult to observe differences between the indirect methods in Figure 6(b) since the performance of the direct methods is so much worse. Hence, we now focus on the indirect methods only, and we note the differences between them under the makespan and the makespan90 metrics. This is illustrated in Figure 7(a) where we make the following observations. Enforcing full

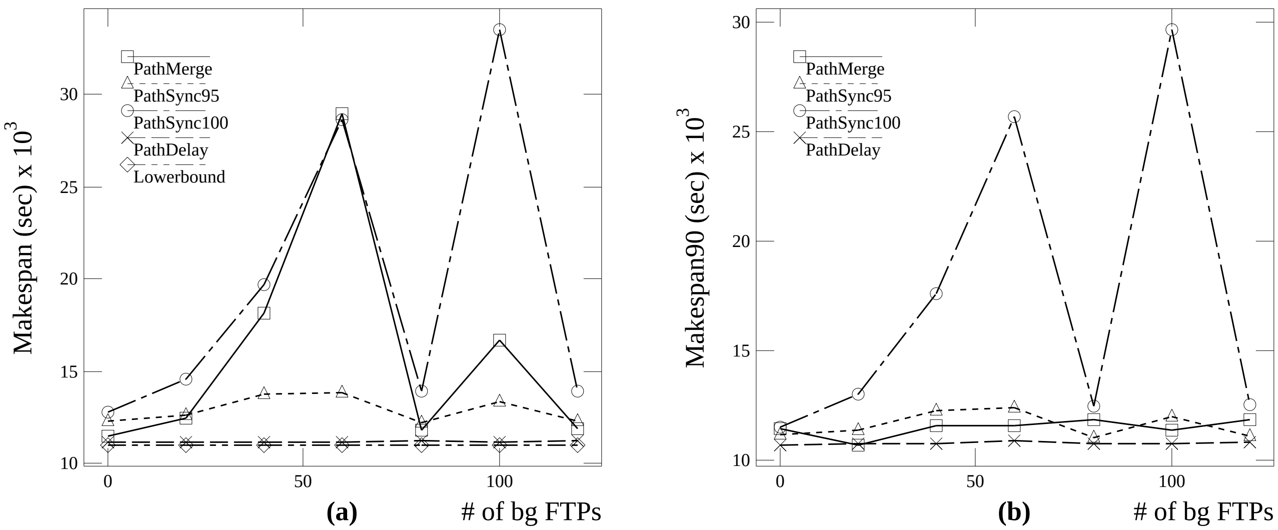

Figure 7: Indirect Methods under the Makespan Metric.

synchronization (as in PathSync100) can be harmful which is not surprising since a single slow stream can lead to (a) significant increases in overall data collection time (although nowhere as significant as the use of direct methods) and (b) increased sensitivity to capacity function estimates and parameter choices in G_H and G_T. We can observe (a), for instance, by comparing the overall performance of PathSync100 and PathSync95 in Figure 7(a). We can observe (b), for instance, by noting the difference in performance of PathSync100 and PathSync95 at 60 vs 80 background flows or 100 vs 120 background flows (in Figure 7(a)). Intuitively, we expect the makespan metric to be monotonically non-decreasing as a function of increasing background flows. What happens here is that due to discretization and capacity estimation errors in construction of G_H and G_T, the highly congested link is set to zero at 80 and 120 flows and it is set to a very small capacity at 60 and 100 flows. Hence, in the latter case, a very small amount of traffic is still routed by the min-cost flow algorithm through the highly congested link. This results in poorer performance of the indirect methods which are more sensitive to variances in data transfers, i.e., such as PathSync100. However, by relaxing the synchronization constraints, e.g., as in PathSync95, one can significantly reduce such sensitivity.

We note that we made small modifications to the background traffic from the time the capacity estimates were done to the time the upload traffic was run (these changes were in sizes of packets used for background traffic). When such changes were not made, PathSync100 performed anywhere from almost identically to $\approx 50\%$ better (although in most cases improvements were more modest);

this is another indication that it is sensitive to capacity function estimates. We also tried modifications to data unit size (during the discretization step in constructing G_H and G_T) and observed similar effects on PathSync100, for reasons similar to those given above. (We do not include these graphs here due to lack of space).

Such sensitivity is exhibited by the PathMerge algorithm as well. We believe that this is due to the fact that in these experiments PathMerge ended up being more aggressive in a few cases in merging paths than was necessary. This is evident, for instance, from observing the difference in PathMerge under the makespan metric in Figure 7(a) and the makespan90 metric in Figure 7(b) as well as by observing the difference between PathMerge and PathDelay. However, other experiments with smaller initial amounts of data per bistro indicate that PathMerge can perform better than PathDelay (we do not include them here due to lack of space). Hence we believe there is a need for more "controlled" path merging; this is an ongoing effort.

Above observations raise another question, which is how much synchronization is really needed in the data collection schedule. By comparing PathDelay with PathSync (and its variants) one might say that ensuring that transfers are initiated at the appropriate times (and then not synchronizing them along the way) is sufficient, since PathDelay performs well in the experiments of Figure 7. However, the experiments in this figure are relatively small scale and hence have relatively few hops in the paths constructed from f_T. Other experiments indicate that as the number of hops on a path (in G_T) increases, PathDelay begins to suffer from getting out of sync with the schedule computed in f_T and performs worse than PathSync95, for instance. (We do not include these figures due to lack of space as well as due to the fact that they are also relatively small scale experiments.)

Another question might be whether the notion of simply assigning time slots (to bistros) during which to transfer data is a reasonable approach, which is essentially the idea behind direct methods such as spread-in-time. We note that the good performance of PathDelay seems to indicate that this idea is on the right track, as long as it is done in the context of *indirect* methods. Of course, this type of a comparison between spread-in-time and PathDelay is not entirely fair since direct vs indirect is not the only difference between them. However, we still believe that this is an indication that clever approaches to spreading load in time without considering the benefits of re-routing that can be obtained from *indirect* methods do not lead to sufficiently good solutions.

In order to estimate how much room is left for improvement, we construct a lower bound on the makespan metric in the context of our experimental setup as follows. We assume that all the data that needs to be collected is located at the "best" bistro, i.e., one with the best connection to the destination server, without background traffic. We then simulate the transfer of all data from the "best" bistro to the destination server *without* any other traffic on the network and observe the resulting performance using the makespan metric. The result of this experiment is depicted in Figure 7(a) along with our indirect methods which illustrates that there is relatively little room left for improvement in this case. We observe less than a 3% difference between the lower bound and the best performing indirect method (at any one point).

However, we note that this is not a general lower bound. Specifically, it works as a lower bound in our case, because we have a fairly symmetric setup and without background traffic all bistros experience a bottleneck at the same place (near the server). Hence, no benefit would be gained from parallelism (i.e., splitting the data between multiple bistros and sending it in parallel) under these conditions (as can also be seen from Figure 6(a) when there is no background traffic). Therefore, in general, there may be room for improvement.

Evaluation Under the Storage Metric.
Next, we evaluate the indirect methods with respect to the storage requirements metric. We note that the direct methods (e.g., as those described in Section 3) do not require additional storage, i.e., beyond what is occupied by the original data itself. In contrast, indirect methods do, in general, require additional storage, since each bistro might have to store not only its own data but also the data being re-routed through it to the destination server.

Figure 8 illustrates the *normalized* maximum per bistro storage requirements, averaged over all bistros (other than the destination), of the indirect methods as a function of increasing congestion conditions. These storage requirements are normalized by those of the direct methods. We use the

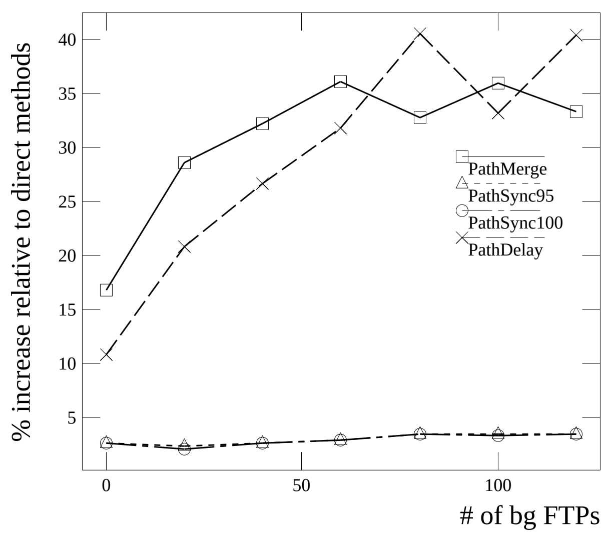

Figure 8: The Storage Metric.

direct methods as a baseline since they represent the inherent storage requirements of the problem as noted above. As can be seen from this figure, the additional storage requirements of our algorithms are small. In all experiments performed by us, storage overheads of all PathSync variations were no more than 4%. PathMerge and PathDelay resulted in storage overheads of no more than 41% (this makes sense since greater storage is needed when less stringent flow synchronization is used). We believe these are reasonable given improvements in overall data collection times as high as one to two orders of magnitude (and, also given the current storage costs).

Evaluation Under the Throughput Metric.

Lastly, we evaluate the indirect methods under the *normalized* mean throughput metric, i.e., how they affect the throughput of the background traffic which represents other traffic in the network. The results are normalized by the throughput achieved by the background traffic *without* presence of the data collection traffic.

We first evaluate the throughput of the direct methods. As illustrated in Figure 9(a), the one-by-one heuristic allows for the highest background traffic throughput. This is not surprising, since

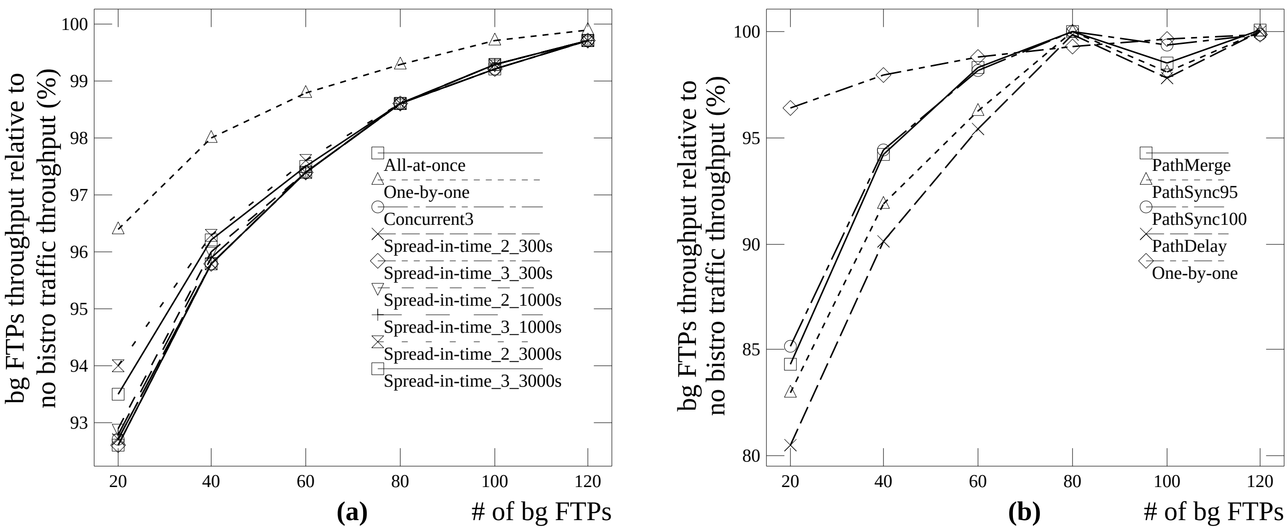

Figure 9: Direct and Indirect Methods under the Throughput Metric.

one-by-one is the most conservative direct method in the sense that it injects the the data collection traffic into the network one flow at a time.

We can now evaluate the throughput due to indirect methods and compare it to that of the one-by-one method. As can be seen from Figure 9(b), the indirect methods result in lower background traffic throughput, but not significantly. The largest difference we observed was no more than 17%. This, of course, is not surprising since the indirect methods are more aggressive than direct methods in taking advantage of bandwidth available in the network. We believe that this is an indication of the fact that they are taking such advantage without significantly adverse effects on other traffic in the network. We also note that the higher (as compared to one-by-one) throughput of background traffic under the indirect methods at, for instance, 80 background flows has to do with indirect methods not using the highly congested link at all at that point (as a result of the discretization and capacity estimation errors in constructing G_H and G_T, as detailed above).

8 Conclusions

The main contribution of this work is to show that *coordinated* indirect routing can be an order of magnitude better than direct routing. A network flow based solution, utilizing a *time-expanded* graph, gives us an optimal routing policy, given the constraints on knowledge of the topology and capacity of the network. Experimentally, we have established that the lack of knowledge of the paths provided by the network to send data, are not a significant barrier. Of course, the more we know about the available capacity and paths chosen by the network, the better our modeling can be. If such information were available, we could use an LP solver to obtain an optimal min-cost flow. In current work, we are exploring how to utilize such information (if it is available), since this may further improve the results to some extent. There are easy extensions to the case of multiple destinations by using multicommodity flow algorithms.

References

[1] ********. *Reference removed for double-blind reviewing.*

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *18th ACM SOSP Conference*, Canada, October 2001. ACM.

[4] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Performance Evaluation*, 27 and 28, 1996.

[5] G. Dantzig. Linear programming and extensions, 1963.

[6] Y. Dinitz, N. Garg, and M. Goemans. On the single source unsplittable flow problem. In *Proceedings of FOCS'98*, pages 290–299, 1998.

[7] A. B. Downey. Using pathchar to estimate internet link characteristics. *ACM SIGCOMM*, 1999.

[8] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, New Jersey, 1962.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.

[10] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. Gryniewicz, and Y. Jin. *Internet Distance Map Service.* http://irl.eecs.umich.edu/jamin/, 1999.

[11] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[12] Andrew V. Goldberg. *Andrew Goldberg's Network Optimization Library.* http://www.star-lab.com/goldberg/soft.html, 2001.

[13] B. Hoppe and E. Tardos. Polynomial time algorithms for some evacuation problems. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 512–521, 1994.

[14] B. Hoppe and E. Tardos. The quickest transshipment problem. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, pages 443–441, 1995.

[15] http://www-mash.cs.berkeley.edu/ns/. *UCB/LBNL/VINT Network Simulator - ns (version 2).*

[16] http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html. *Georgia Tech Internetwork Topology Generator.*

[17] IRS. *Fill-in Forms.* http://www.irs.ustreas.gov/prod/forms_pubs/fillin.html, 2001.

[18] Van Jacobson. *pathchar.* http://ftp.ee.lbl.gov/pathchar/, 1997.

[19] Jon M. Kleinberg. Single-source unsplittable flow. In *IEEE Symposium on Foundations of Computer Science*, pages 68–77, 1996.

[20] Kevin Lai and Mary Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems.* USENIX, March 2001.

[21] T. Leighton. *The Challenges of Delivering Content on the Internet.* Keynote address at the ACM SIGMETRICS 2001 Conference, Cambridge, Massachusetts, June 2001.

[22] Bruce A. Mah. *pchar.* http://www.ca.sandia.gov/ bmah/Software/pchar/, 2000.

[23] K. Moore, J. Cox, and S.Green. Sonar - a network proximity service. *IETF Internet-Draft*, 1996.

[24] Jitedra Padhye, Victor Firoiu, Don Towsley, and Jim Krusoe. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, 1998.

[25] Dan Rubenstein, James F. Kurose, and Donald F. Towsley. Detecting shared congestion of flows via end-to-end measurement. In *Proceedings of 2000 ACM SIGMETRICS Conf.*, pages 145–155, 2000.

[26] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: a case for informed internet routing and transport. In *IEEE Micro*, volume 19, pages 50–59. IEEE, January 1999.

[27] Stefan Savage, Andy Collins, and Eric Hoffman. The end-to-end effects of internet path selection. In *ACM SIGCOMM '99 conference on Applications, technologies, architectures, and protocols for computer communication*, 1999.

[28] B. Schneier. *Applied Cryptography, Second Edition.* Wiley, 1996.

[29] S. Seshanm, M. Stemm, and R.H. Katz. SPAND: Shared passive network performance discovery. In *Proceedings of the First USENIX Symp. on Internet Technologies and Systems*, Dec 1997.

[30] Randolph Wang, Arvind Krishnamurthy, Richard P. Martin, Thomas E. Anderson, and David E. Culler. Modeling communication pipeline latency. In *Proceedings of 1998 ACM SIGMETRICS Conf.*, pages 22–32, 1998.

[31] R. Wolski and M. Swany. *Network Weather Service.* http://nws.cs.utk.edu/, 1999.

Appendix: Background on Bistro

In this appendix we give a more detailed description of the Bistro framework. This is included for the reviewers' benefit only (i.e., it is not intended as part of the paper or its contributions). In this exposition we focus on upload applications with reasonably large data transfers and deadlines for clients to submit their data, e.g., such as the online income tax submission application mentioned in Section 1. However, we note that deadlines are not a factor in this work since the focus of this paper is on the performance of Step 3 below, which would typically be performed after the deadline. Hence, there is no deadline associated with this step, but its performance is still crucial since the data cannot be processed by the application until it is collected.

Briefly, the Bistro upload architecture works as follows (refer to [1] for details). Given a large

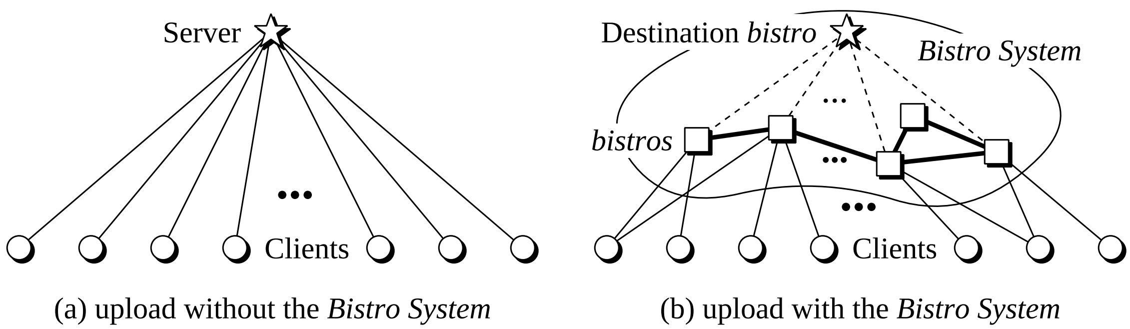

Figure 10: Upload Problem.

number of clients that need to upload their data by a given deadline to a given destination server (refer to Figure 10(a)), the Bistro architecture breaks the upload problem into three steps (as illustrated in Figure 10(b)):

- *Step 1*, the timestamp step, which must be accomplished prior to the deadline for clients to submit their data to the destination server. In this step, each client sends to the server a message digest of their data [28] and in return receives a timestamp ticket from the destination server as a receipt indicating that the client made the deadline for data submission. The purpose of this step is to ensure that the client makes the deadline *without* having to transfer their data which is significantly larger than a message digest and might take a long time to transfer during high loads which are bound to occur around the deadline time. It is also intended to ensure that the client (or an intermediate bistro used in Step 2 below) does not change their data after receiving the timestamp ticket (hence the sending of the message digest to the destination server). All other steps can occur before or after the deadline.
- *Step 2*, the transfer of data from clients to intermediate hosts, termed bistros. This results in a low data transfer response time for clients since (a) the load of many clients is distributed among multiple bistros and (b) a good or near-by bistro can be selected for each client to improve data transfer performance. Since the bistros are not trusted entities (unlike the

destination server), the data is encrypted by the client prior to the transfer.

- *Step 3*, the collection of data by the destination server from the bistros. The destination server determines when and how the data is collected in order to avoid hotspots around the destination server (i.e., the original problem of having many sources transfer their data to the same server around the same time). Once the destination server collects all the data, it can decrypt it, recompute message digests, and verify that no changes were made to a client's data (either by the client or by one of the intermediate bistros) after the timestamp ticket was issued.

A summary of main advantages of this architecture is as follows: (1) hotspots can be eliminated around the server because the transfer of data is decoupled from making of the deadline, (2) clients can receive good performance since they can be dispersed among many bistros and each one can be direct to the "best" bistro for that client, and (3) the destination server can minimize the amount of time it takes to collect all the data since now it is in control of when and how to do it (i.e., Bistro employs a server pull). Algorithms for performance improvement of Step 3 is the focus of this paper.