Large-scale Data Collection: a Coordinated Approach

William C. Cheng† bill.cheng@acm.org
Cheng-Fu Chou‡ chengfu@cs.umd.edu
Leana Golubchik§† leana@cs.usc.edu
Samir Khuller‡ samir@cs.umd.edu
Yung-Chun (Justin) Wan‡ ycwan@cs.umd.edu

Abstract— **In this paper we consider the problem of collecting a large amount of data from several different hosts to a single destination in a wide-area network. Often, due to congestion conditions, the paths chosen by the network may be poor. By choosing an alternate route at the application level, we may be able to obtain substantially higher performance. This 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. In this paper we present an approach for computing coordinated data collection schedules, which can result in significant performance improvements. We make no assumptions about knowledge of the topology of the network or the capacity available on individual links of the network, i.e., we only use end-to-end information. Finally, we also study the shortcomings of this approach in terms of the gap between the theoretical formulation and the resulting data transfers in wide-area networks. In general, our approach can be used for solving arbitrary data movement problems over the Internet. We use the Bistro platform to illustrate one application of our techniques.**

Keywords: System design, Simulations, Graph theory.

I. 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 should be filed electronically by 2007. With (on the order of) 100 million individual tax returns (alone) filed in US yearly, where each return is on the order of 100 KBytes [19], scalability issues are a major concern, since the data involved is on the order of terabytes.

Recently, we proposed a scalable and secure *application-level* architecture for wide-area upload applications [3]. 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 to a given destination server, Bistro breaks the upload process into three steps: (1) a timestamp step which provides bit-commitment [**?**] as well as ensures that the data is submitted on-time, for applications with deadlines, all *without* having to actually transfer the data, (2) a data transfer step, where clients *push* their data to intermediate hosts (bistros), which ensures fast response time for the clients, and (3) 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 (2) 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 (3). *Performance of the third step, i.e., large-scale data collection from multiple source hosts to a destination server, is the topic of this paper.* We note that in this context there are no deadline issues, as any deadlines associated with an upload application are taken care of in step (1) above.

Hence, we focus on *application-level* approaches to improving performance of large-scale data collection. We do this in the context of the Bistro upload framework. How-

This work is supported in part by the NSF Digital Government Grant EIA-0091474 and the NSF ITR CCR-0113192 grant.

†This work was done in part while the authors were with the Dept. of Computer Science and UMIACS, U. of Maryland at College Park.

‡Dept. of Computer Science and UMIACS, U. of Maryland at College Park.

§Dept. of Computer Science / ISI / IMSC, USC.

ever, 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 [11]. Another example is data mining applications where large amounts of data may need to be transferred to a particular server for analysis purposes.

Consequently, our *data collection problem* can be stated as:

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*.

Since we are focusing on application-level solutions, a path (above) 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 (hosts) 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.

We note that the choice of the makespan metric is dictated by the applications stated above, i.e., there are 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. Note, however, that our problem formulation (below) is versatile enough that we can optimize for other metrics (if desired), e.g., mean transfer times. We also note that we do not require 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 amount 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: (a) transfer the data from all source hosts to the destination host in parallel, or (b) transfer the data from the source hosts to the destination host sequentially in some order, or (c) transfer the data in parallel from a subset of source hosts at a time and possibly during a predetermined time slot, as well as other variants (refer to Section V for details). We refer to these methods as *direct*, since they send data directly from the source hosts to the destination host.

However, long transfer times between one or more of the hosts (holding the data) and the destination server can significantly prolong the amount of time it takes to complete a large-scale 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 peering points whose congestion is often cited as cause of delay in wide-area data transfers [23]. 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.

An approach to dealing with such congestion problems might be to use application-level re-routing techniques (refer to Section II for details). Most such techniques use "best"-path type rerouting, i.e., data from a source host is transferred over the "best" application level path to the destination host, where the path is determined independently for each source host.

However, we believe that in the case of a large-scale data collection problem, 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 below.) Consequently, our focus in this work is on development of *algorithms* for *coordinated* data transfers in large-scale data collection applications. In contrast, we refer to the application-level re-routing techniques (mentioned above) as *non-coordinated* methods.

Given the above stated data collection 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 related 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.

We note that a more general problem where there are multiple destination hosts is also of importance, e.g., in the income tax forms submission application, IRS might want the data to be delivered to multiple processing centers. Although in this paper we present our methodology in the context of a single destination, for ease of exposition, we can solve the multi-destination problem as well (by employing either multicommodity flow algorithms [1], or a single commodity min-cost flow algorithm, as in Section III, depending on the requirements). In other words, there is *nothing about our approach that would fail* for the case of multiple destinations for different pieces of data.

Our contributions are as follows. We propose *coordinated* data transfer algorithms for the large-scale data collection problem defined above, intended for an IP-type network. We evaluate the performance of these algorithms in a simulation setting (using ns2 [17]). We show that *coordinated* methods can result in *significant performance improvements.* These improvements are achieved under *low storage requirement overheads and without significant detrimental effects on other network traffic.*

II. Related Work

As stated in Section I, in this paper we focus on algorithms for large-scale data transfers over wide-area networks, in the context of upload applications. To the best of our knowledge, Bistro [3] is the only *application-level* framework for large-scale upload applications existing to date. Hence, we do this work in the context of Bistro, although as noted above, other types of applications can benefit as well.

Although some works exist on multipoint-to-point aggregation mechanisms at the IP layer [**?**], [**?**], such solutions have focused on reduction of overheads due to small packets (e.g., ACKs) and usually require the use of an active networks framework which is not currently widely deployed over the public Internet.

Another approach is application-level re-routing, which is used to improve end-to-end performance, or provide efficient fault detection and recovery for wide-area applications. For instance, in [28] the authors perform a measurement-based study of comparing end-to-end round-trip time, loss rate, and bandwidth of default routing vs alternate path routing. Their results show that in 30% to 80% of the cases, there is an alternate path with significantly superior quality. Their 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 [27] and RON [2]. The Detour framework [27] is an informed transport protocol. It uses sharing of congestion information between hosts to provide a better "detour path" (via another node) for applications to improve the performance of each flow and the overall efficiency of the network. Detour routers are interconnected by 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 default Internet paths.

The Resilient Overlay Network (RON) [2] is an architecture allowing 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 implementing expressive routing policies.

The above mentioned re-routing schemes focus on architectures, protocols, and mechanisms for accomplishing application-level re-routing through the use of overlay networks. They provide evidence that such approaches can result in significant performance benefits. We consider a similar environment (i.e., application-level techniques in an IP-type wide-area network). However, an important distinction is that the above mentioned works do not consider *coordination* of multiple data transfers. All data transfers are treated independently, and each takes the "best" application-level route available. We refer to such techniques as "non-coordinated" data transfers. In contrast, our goal is to construct application-level *coordinated* data transfers. In [6] we perform an initial performance study to illustrate potential benefits of coordinated transfers over non-coordinated ones, with respect to performance and robustness under inaccuracies in network bandwidth estimation, i.e., when actual available bandwidth deviates from the its estimate by significant amount (e.g., 70%). We note that, no algorithms were presented in [6]. In this paper we focus on the coordinated data collection *algorithms* and their performance in comparison to a number of direct methods (detailed in Section V) as well as non-coordinated methods (for completeness).

Lastly, we note that it is *not* the purpose of this work to propose novel techniques for identifying congestion conditions or determining available or bottleneck link capacities. Rather, we do this work under the assumption that in the future such information will be provided by other Internet services, e.g., as in [12], [25], [30].

III. Graph Theoretic Formulation

In general, the network topology can be described by a graph $G_N = (V_N, E_N)$, with two types of nodes, namely end-hosts and routers, and a capacity function c which specifies the capacity on the links in the network. The sources $S_1, \dots S_k$ and the destination host D are a subset of the end-hosts. The example of Figure 1(a) illustrates the potential benefits of a coordinated approach. Here, some chunk of data can be be transferred between S_3 and D, while another is transferred, in parallel, between S_2 and S_1 (as staging for the final transfer from S_1 to D). These two transfers would not interfere with each other while attempting to reduce the makespan metric by utilizing different resources in the network in parallel.

Note that, our algorithm (below) *does not know either the topology of the network or the capacity function.* In addition, background traffic exists, which affects the available bandwidth on the links. Hence, we model the network by an *overlay* graph consisting of the set of source hosts and a 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(b) for an example corresponding to Figure 1(a); outgoing edges from D are not shown since they are never used.) The capacity function in G_H 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 which the *network provides us* in the graph G_N, subject to the background traffic. 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 [7] to obtain an optimal flow under those constraints.

From the overlay graph G_H we construct the "time-expanded" graph G_T [10], [15] (see Figure 2) which is the graph that our algorithm 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 VI 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

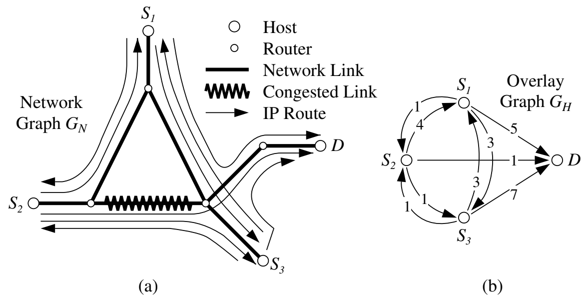

Fig. 1. Network topology and a corresponding overlay graph.

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 [16] argue that allowing edges of the form $u(i)$ to $u(i+1)$ does not decrease the minimum value of T (i.e., makespan).

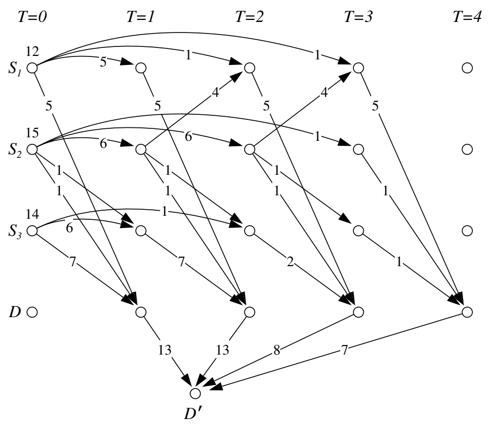

Fig. 2. Time-expanded graph.

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 apply a min-cost flow algorithm [1] in the time-expanded graph to obtain an *optimal* flow solution, e.g., the number on the edge in Figure 2 corresponds to how much data will be sent by that link at that time, 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. 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).

The min-cost flow algorithm computes a flow function f_T that specifies the flow in the time expanded graph G_T; we use Goldberg's code [13], [14] for this computation. Then we convert this flow into a schedule for transferring data in a real network (in Section IV). The min-cost flow algorithm runs in $O(n^2 m \log nC)$, where n, m, and C are the number of vertices, the number of edges, and the maximum capacity of a link, in the network flow graph, respectively. If we have b bistros, then $n = (T+1)b$, $m = (T+1)b(b-1)/2$, and C = (*maximum amount of data that can be sent in one time unit*) / (*data unit size*). Thus, the total running time of our algorithm is $O(T^3 b^4 (\log TbC)(\log T))$. Note that this is the worst case behavior. In our experiments, the entire computation takes on the order of a few seconds on a Pentium III 650 MHz, and typical values of T, b, and C are 150, 8, and 30, respectively.

In our problem T is not large so it is feasible to build the entire time-expanded graph and run a min-cost flow algorithm on it. Otherwise, one could use other algorithms (see Hoppe and Tardos [15], [16]) which run in polynomial time rather than pseudo-polynomial time.

Note that in our formulation, we compute the capacity function once initially (refer to Section VI), to estimate the available capacity between pairs of hosts. We then we 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. Such adaptation to changing network conditions is an ongoing effort. Also note that our algorithm is not complicated to implement since we are working at the application layer, where we only need to control the route within the *overlay* network without any changes to the network protocols (such as IP or TCP). We also note that for ease of exposition, we do not explicitly include I/O bandwidth constraints in our formulation; however, this can easily be included in capacity constraints within the same graph-theoretic formulation. We do not include this in our experiments (below) as currently, I/O bandwidth is not the bottleneck resource in our application. Lastly, 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.

Remark: An alternative approach might be to use the overlay graph, G_H, to compute the "best" path in G_H from each host to the destination, independently, e.g., 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 would correspond to the *non-coordinated* approach, and hence, our *coordinate* approach formulation includes the non-coordinated approach as a special case. However, this option does *not* permit for (a) any coordination between transfers from different source hosts, (b) explicit load balancing as each node makes an independent 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, the non-coordinated method corresponds to a feasible flow in a graph G_{T_i} for some T_i. Note that $T_i \geq T_{\min}$ 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 the non-coordinated method*. This difference becomes especially significant, if several good application level routes exist, but non-coordinated methods send their data along the "best" path, thus causing congestion along this path. In this paper, we show that the coordinated approach performance significantly better (refer to Section VI).

IV. Transfer Schedule Construction

What remains is to construct a data transfer schedule, f_N (defined as the goal of our data collection problem in Section I), from the flow function f_T computed in Section III, 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 III 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 application-level 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 as follows: 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 VI for details). Second, we can provide a mechanism for merging data transfers which are deemed "too small" (we omit this approach due to lack of space; please refer to [**?**]). The "too large" problem is addressed by a proper choice of the time unit parameter (see Section VI).

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 data collection methods since all these methods can benefit from this. In fact, we made a comparison between a direct method employing parallel connections and our coordinated methods *without* parallel connections, and the coordinated methods could still achieve better performance.

Data split and merge constraints.
The f_T solution 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. 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 [21]. Good heuristics for these have been developed recently, and could be used [8].

Synchronization of flows.
The f_T solution 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, 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 transferred 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 transferred from a source to the destination. For example (in 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.

The PathSync Method.

In this method we employ complete synchronization as prescribed by the time-expanded graph solution obtained in Section III. 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, and so on. 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 VI), and since we may not know which transfers do or do not share the same bottleneck link (unless, e.g., we employ techniques in [26]), 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.

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.

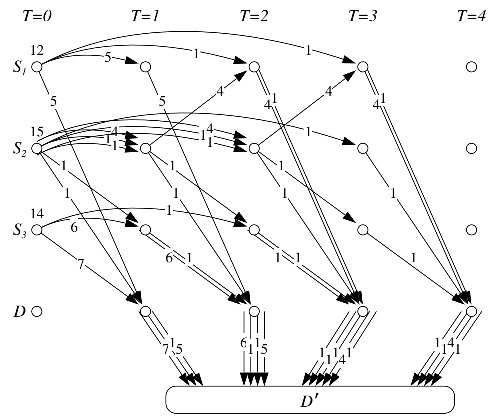

Fig. 3. Solution obtained after flow decomposition.

The PathSync method performs quite well (refer to Section VI), 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 hosts 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 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, i.e., no inter-host synchronization needed. 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 VI).

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. Due to lack of space, we do not explore this here further; refer to [**?**] for details and description of other methods for constructing f_T from f_N.

V. Direct Methods

We now give details of the direct methods used (below) for comparison purposes.

- *All-at-once*. Data from all source hosts is transferred simultaneously to the destination server.
- *One-by-one*. The destination server *randomly* repeatedly 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.
- *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 a 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.
- *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.

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

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. 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 (as dictated by IP). Coordinated methods proposed in this paper are able to take advantage of such resources and therefore result in significantly better performance, as illustrated in Section VI.

VI. Performance Evaluation

In this section we evaluate the performance of the various data transfer methods and illustrate the benefits of using a *coordinated* approach. This evaluation is done through simulation; all results are given with at least $95\% \pm 5\%$ confidence.

Simulation Setup

For all simulation results reported below, we use ns2 [17] in conjunction with the GT-ITM topology generator [18] to generate a transit-stub type graph with 152 nodes for our network topology. 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

4. The capacity of a "transit node to transit node" edge within the same transit domain is 10 Mbps. The capacity of a "transit node to transit node" edge across different transit domains is 5 Mbps. The capacity for a "transit node to stub node" edge or a "stub node to stub node" edge is 2.5 Mbps. 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 peering points [23]. Moreover, we linearly scale the propagation delay between nodes generated by the GT-ITM topology generator [18] such that the maximum round trip propagation delay in our topology is 80 ms. Note that, the size and parameters of our network model and the simulation setup are motivated by what is practical to simulate with ns2 in a reasonable amount of time. However, since our goal is a relative comparison of the methods, this will suffice.

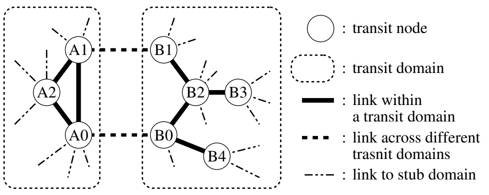

Fig. 4. The simulation topology.

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 and 75 MBytes with an additional constraint that the total amount of data in all bistros is 350 MBytes. In addition to the data collection traffic, we setup 0 to 120 background traffic flows from nodes attached to transit domain B to nodes attached to transit domain A. In our experiment, the ratio of the number of background flows in peering point (B0,A0) to the number of background flows in peering point (B1,A1) is 1:3 (asymmetric). We also investigated how the methods behave under different ratios between the peering points such as 1:1 (symmetric), 2:1, and 1:2; the results indicate similar trends, and we do not include them here due to lack of space. The background traffic pattern is similar to that in [26]. Each background flow is an infinite FTP with a probability of 0.75. Otherwise, it is an on-off CBR UDP flow. The average on-time and off-time is chosen uniformly between 0.2 and 3 seconds. The average rate (including the off periods) is chosen so that the expected total volume of UDP traffic through a peering point takes up 5% of the capacity of that point. (Similar trends persist under different volumes of UDP traffic; we do not include these results due to lack of space.) 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 data collection traffic. This choice avoids the non-interesting cases (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 III 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 are 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 separately while sending a 5 MByte file between these bistros. We repeat the measurement 10 times and take the average to get a better estimation. These measurements are performed with background traffic conditions corresponding to a particular experiment of interest but without any data collection traffic or measurement traffic corresponding to other bistro pairs. Although a number of different measure techniques exist in literature [4], [9], [20], [22], [24], [29], we use the above one in order to have a reasonably accurate and simple estimate of congestion conditions for purposes of *comparison of data collection methods*. However, we note, that it is *not* our intent to advocate particular measurement and available bandwidth estimation techniques. Rather, we expect that in the future such information will be provided by other Internet services, e.g., such as those proposed in SONAR [25], Internet Distance Map Service (IDMaps) [12], Network Weather Service (NWS) [30], and so on. Since these services will be provided for many applications, we do *not consider bandwidth measurement as an overhead* of *our* application but rather something that can be amortized over many applications.

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 IV). 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 IV).

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 IV).

In the experiments presented here we use a time unit which is 100 times bigger than the maximum propagation delay on the longest path, i.e., 8 sec. (This choice is motivated by the fact that in many cases we were not able to run ns simulations with smaller time units as the resulting number of flows was too large; a smaller time unit did *not* present a problem for our theoretical formulation.) 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 2.5 Mbps $\times$ 80 ms (on the longest path), we use a data unit size a bit larger than that, specifically 64 KBytes.

Performance Metrics.

The performance metrics used in the remainder of this section are: (a) makespan, i.e., the time needed to complete transfer of 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 data collection 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 (refer to Section I).

Evaluation Under the Makespan Metric.

We first evaluate the direct methods described in Section V using the makespan metric. As illustrated in Figure 5(a) direct methods which take advantage of parallelism in data delivery (such as all-at-once) perform better under our simulation setup. Intuitively, this can be explained as follows. Given the makespan metric, the slowest bistro to destination server transfer dominates the 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.

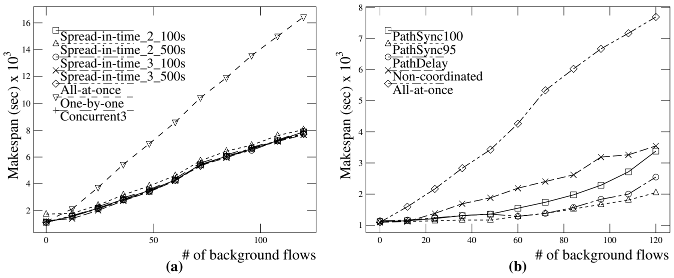

Fig. 5. Direct, Non-coordinated and Coordinated Methods under the Makespan Metric.

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 V under the makespan metric in our experiments, we now compare just the all-at-once method to our coordinated methods. We include non-coordinated methods in this comparison for completeness (refer to [6] for details). This comparison is illustrated in Figure 5(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 coordinated methods result in a significant improvement in performance, especially as this congestion (due to other traffic in the network) increases. For instance, in Figure 5(b), using PathSync95 we observe improvements (compared to direct methods) from 1.9 times under 24 background flows to 3.7 times when the background traffic is sufficiently high (in this case at 120 flows). PathSync95 is 1.7 times better than the non-coordinated method under 120 flows.

As shown in Figure 5(b), enforcing full 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 5(b). Regarding (b), intuitively, overestimating the capacity of a link may result in sending too much data in one time slot in a particular transfer in our schedule, which may delay the whole schedule as we fully synchronize all transfers (refer to [**?**] for details).

We note that if the packet size of the background traffic at the time the capacity estimations were done is different from those at the time the data is collected, PathSync100 performed anywhere from almost identically to two times worse; 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).

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 pretty well in the experiments of Figure 5(b). 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 much worse than PathSync95, for instance. (We do not include these figures due to lack of space.)

Remark: One question might be whether the notion of simply assigning time slots (to bistros) during which to transfer data directly to the destination server is a reasonable approach. Since this is essentially the idea behind direct methods such as spread-in-time, and since they performed *significantly* worse than the *coordinated* methods in the experiments illustrated above, we believe that such methods do not lead to sufficiently good solutions.

Evaluation Under the Storage Metric.

Next, we evaluate the different methods with respect to the storage requirements metric. We note that the direct methods do not require additional storage, i.e., beyond what is occupied by the original data itself. In contrast, non-coordinated and coordinated 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 6 illustrates the *normalized* maximum per bistro storage requirements, averaged over all bistros (other than the destination), of the non-coordinated and coordinated methods as a function of increasing congestion conditions. These storage requirements are normalized by those of the

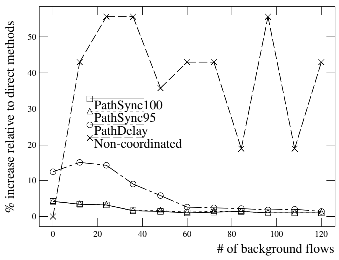

Fig. 6. The Storage Metric.

direct methods. We use the 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 coordinated algorithms are small. In all experiments performed by us, storage overheads of all PathSync variations were no more than 5%. PathDelay resulted in storage overheads of no more than 15% (this makes sense since greater storage is needed when less stringent flow synchronization is used). We believe these are reasonable given the above improvements in overall data collection times (and, also given the current storage costs). Note that the storage requirement of the non-coordinated method is high because multiple data flows from different source hosts may be re-routed to the same intermediate host at the same time.

Evaluation Under the Throughput Metric.

We also evaluate the non-coordinated and coordinated methods under the *normalized* mean throughput metric, i.e., their effect on the throughput of the background traffic which represents other traffic in the network. The results are normalized by the throughput achieved by the background ftp traffic *without* presence of the data collection traffic.

We first evaluate the throughput of the direct methods. As illustrated in Figure 7(a), the one-by-one method allows for the highest background traffic throughput. This is not

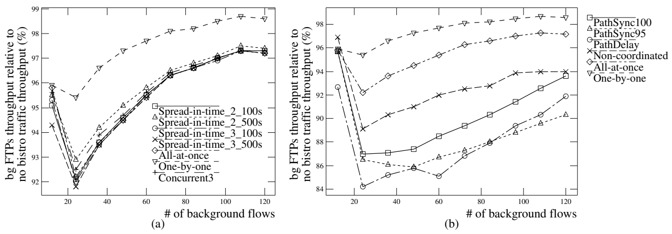

Fig. 7. Direct, Non-coordinated and Coordinated Methods under the Throughput Metric.

surprising, since one-by-one is the most conservative direct method in the sense that it injects the data collection traffic into the network one flow at a time. As can be seen in Figure 7(b), the non-coordinated and coordinated methods result in lower background traffic throughput, but not significantly. The largest difference we observed was no more than 16% (for coordinated and non-coordinated methods). This, of course, is not surprising since the coordinated and non-coordinated methods are more aggressive than direct methods in taking advantage of bandwidth available in the network. We believe that this is an indication that they are taking such advantage without significant adverse effects on other traffic in the network.

VII. Conclusions

In this paper we proposed *coordinated* data transfer algorithms for the large-scale data collection problem and showed that coordinated methods can result in *significant performance improvements* as compared to direct and non-coordinated methods. We used a network flow based solution, utilizing *time-expanded* graphs, which gave us an *optimal* data transfer schedule with respect to the makespan metric, 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 potentially our modeling can be.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *18th ACM SOSP Conference*, Canada, October 2001. ACM.

[3] B. R. Badrinath and P. Sudame. Gathercast: An efficient mechanism for multi-point to point aggregation in ip networks. Technical Report DCS-TR-362, Computer Science Department, Rutgers University, July 1998.

[4] S. Bhattacharjee, W. C. Cheng, C.-F. Chou, L. Golubchik, and S. Khuller. Bistro: a platform for building scalable wide-area upload applications. *ACM SIGMETRICS Performance Evaluation Review (also presented at the Workshop on Performance and Architecture of Web Servers (PAWS) in June 2000)*, 28(2):29–35, September 2000.

[5] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and Implementation of an Active Network Service. *IEEE Trans. on Networking*, 2000.

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

[7] W. C. Cheng, C.-F. Chou, L. Golubchik, S. Khuller, and Y.C. Wan. On a graph-theoretic approach to scheduling large-scale data transfers. Technical Report CS-TR-4322, University of Maryland, Jan 2002.

[8] C.-F. Chou, Y.-C. Wan, W. C. Cheng, L. Golubchik, and S. Khuller. A performance study of a large-scale data collection problem. *To appear in 7th International Workshop on Web Content Caching and Distribution*, August 2002 (7 pages).

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

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

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

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

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

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

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

[16] A. V. Goldberg. *Andrew Goldberg's Network Optimization Library*. http://www.avglab.com/andrew/soft.html, 2001.

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

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

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

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

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

[22] V. Jacobson. *pathchar*. http://ftp.ee.lbl.gov/pathchar/, 1997.

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

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

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

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

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

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

[29] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: a case for informed internet routing and transport. In *IEEE Micro*, volume 19, pages 50–59. IEEE, January 1999.

[30] S. Savage, A. Collins, and E. 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.

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

[32] 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.

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