Confluence: A System for Lossless Multi-Source Single-Sink Data Collection*

Jay A. Patel, Brian Cho, and Indranil Gupta

Department of Computer Science
University of Illinois at Urbana-Champaign
{jaypatel,bcho2,indy}@cs.uiuc.edu

Abstract. Distributed environments often require collection of large amounts of critical and raw data from multiple locations to a central clearinghouse, e.g., task results or large datasets from multiple clouds, logs from multiple PlanetLab nodes, video transcripts in tele-immersive settings, etc. We present the design, implementation and evaluation of Confluence, a system for rapid and lossless transfer of unique files from multiple source nodes to a single sink node. First, we formally model the multi-source single-sink data collection problem for a static network and present an optimal solution in terms of total transfer time. Second, we build in mechanisms to make the system workable in dynamic networks. The resulting Confluence system builds an adaptive source-2-source (s2s) overlay amongst participating nodes, which exploits spatial as well as temporal heterogeneity of available bandwidth. We conduct an evaluation of Confluence on PlanetLab traces in ns-2. Results show that Confluence can improve total transfer time by as much as 40% (with up to 50 sources).

1 Introduction

Several distributed environments perform central collection of critical and raw data from a small number of source nodes. For instance, a scientist running her data-intensive computation across multiple cloud or grid computing sites, would want to collect the final computation results from each of the site gateways, and have these available on her local server. Another example is a multi-site multimedia tele-immersive setup (e.g., [22]) which typically involves fewer than 10 sites. Each site gateway maintains a video transcript. After the teleconference, a site may collect all the transcripts for archiving and replaying videos. A final example is researchers who deploy and debug prototypes of their distributed systems within small clusters (e.g., a small PlanetLab slice) before moving it to large-scale deployment. They need to periodically collect event logs generated at these hosts to a single sink node, for offline analysis such as debugging and profiling.

All the above settings are characterized by the small number of *source nodes* involved, each with its unique file, and the *single sink node* to which these files need to be downloaded. Another common characteristic is the periodic collection of new data

* This work was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR grant CMS-0427089.

logs that are continuously produced at one or more nodes, i.e., for an always-on service or after execution of another event. In this paper, our goal is to minimize the total time required to transfer the necessary files from the source nodes to the sink node. From here on, we refer to this as the "multi-source single-sink data collection problem."

Currently, researchers commonly use the "Direct Transfer" strategy of initiating direct and simultaneous transfers from each source to the sink. While Direct Transfer offers good performance, the data flows on slow connections, i.e., source nodes with the least amount of available bandwidth to the sink node, lag behind the other, faster data flows. As such, a select few lagged flows prolong the transfer process.

Our solution is based on the key observation that the transfer process can be sped up by routing data via intermediate nodes. The diversity of connections amongst Internet hosts has been widely observed [2, 3], and falls into two categories – spatial and temporal. Spatial diversity refers to the fact that different links have different bandwidth availabilities, whereas temporal diversity refers to the variation over time of the available bandwidth at a single link. For instance, by randomly sampling sets of three nodes from the PlanetLab snapshot provided (on April 8, 2008) by S^3 [23], we observed that 37% of links can achieve better connectivity by leveraging indirection via a third node.

Motivated by the above observation, we designed a new system called *Confluence* that tackles the multi-source single-sink data collection problem. Confluence uses an *adaptive* source-2-source (s2s) overlay in order to speed up the transfer of file blocks towards the sink. Intuitively, the s2s overlay facilitates a source node (with a congested path to the sink) to utilize other source nodes as intermediaries for routing file blocks to the sink. Concretely, our approach first poses the problem as a variant of flow optimization among the source nodes. This captures the spatial diversity in bandwidth. We provide a theoretically optimal solution to this problem. Next, we augment this static solution with on-the-fly recomputation. This helps us exploit temporal diversity in bandwidth.

We present an evaluation of Confluence using an ns-2 [15] implementation driven by PlanetLab traces, while we are continuing our efforts to deploy Confluence as a PlanetLab service. Our experiments find that at small scales (up to 50 source nodes), Confluence outperforms Direct Transfer by up to 40%. However, Confluence is not a panacea - Direct Transfer may be equally preferable at larger scales (with 100 or more sources) or at highly congested sink nodes, due to the continuous saturation of available bandwidth at the sink node. An interesting offshoot of our effort is thus the somewhat counter-intuitive finding that a peer-to-peer solution can do better than a centralized solution at smaller scales (rather than at larger scales). These small scales are reasonable and realistic for many applications, as pointed out previously.

Motivating Example Before delving into the details of Confluence, we use an example to illustrate the benefits of exploiting spatial diversity of bandwidth via an s2s overlay. As in Figure 1, consider a network with two sources x and y, and one sink t, in which the capacity of x-t link is 1 MBps, the capacity of y-t link is 5 MBps, and the capacity of $x-y$ link is 2 MBps. For sake of simplicity, we assume that all links are symmetrical in uplink and downlink connectivity. Further suppose x and y each hold a 1000 MB file. Our problem entails transferring both these files to sink node t as rapidly as possible.

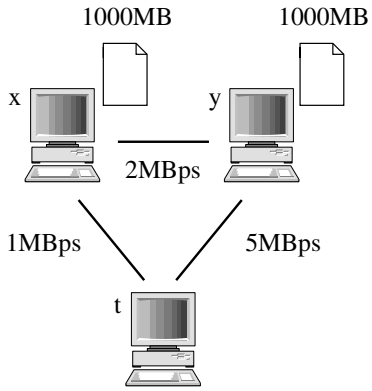


Fig. 1. A motivating example.

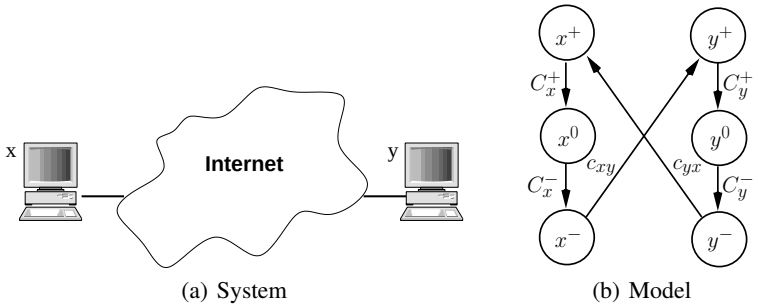
Direct transfers from x and y individually to t, assuming that they are fully utilized (i.e., using both links simultaneously do not induce congestion at t), would take: $\max\left(\frac{1000MB}{5MBps}, \frac{1000MB}{1MBps}\right) = 1000$ seconds. In comparison, if x and y collaborated with one another, x may transfer its file to t via y. Using a sequential transfer process, where y transfers its own file to t and then acts as an intermediary for x's file, would take only $\frac{1000MB}{5MBps} + \frac{1000MB}{\min(5MBps,2MBps)} = 700$ seconds. The completion time can be further reduced by file splitting and pipelining. Using file splitting, x can transfer part of the file directly to t, while the rest of x's file can be transferred to t via y. Pipelining allows y to start receiving data from x, while it transfers it's own file to t. Confluence generalizes these observations to scenarios involving several sources.

Note that our approach is different from well-studied aggregation systems [5, 11] because we cannot use in-network aggregation – the raw data is required by the sink node. However, files can be compressed at source nodes a priori, orthogonal to our file transfer mechanism.

Paper Organization The rest of the paper is organized as follows: Section 2 introduces our problem model and theoretical solution, Section 3 presents our system design, and Section 4 describes our implementation and experimental evaluation. We cover related work in Section 5, and lastly, we conclude in Section 6.

2 Theoretical Formulation and Solution

In this section we formally model a time-invariant (i.e., static) network that captures the spatial diversity of available bandwidth, and describe a theoretical solution for the multi-source single-sink data collection problem. We also discuss the optimality and complexity of our solution.



(a) System (b) Model

Fig. 2. A networked system of two nodes.

2.1 Graph Model

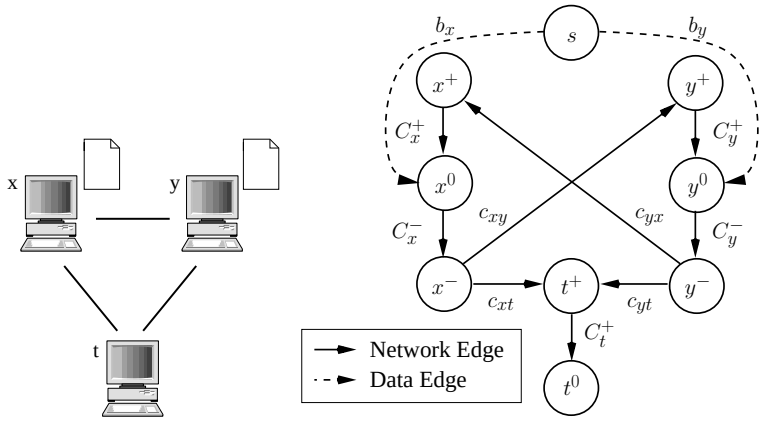
We model the networked system as a directed graph $G = (V, E)$, where V represents the set of end-nodes (derived from all the source hosts and the sink host) and E represents (a subset of the) network paths.

A system with two hosts is modeled as shown in Figure 2. A host x is represented by three vertices x^+, x^0, and x^-. Vertex x^0 represents the physical host itself, whereas vertices x^+ and x^- model the host's ISP. To support asymmetric ISP connectivity, edge (x^+, x^0) models node x's downlink capacity C_x^+, and similarly edge (x^0, x^-) models node x's uplink capacity C_x^-. This model is motivated by previous work that reports packet losses and queuing delays within a backbone ISP are very low [16]. As such, this provides a good balance between the complexity of modeling the underlying IP topology and the realities of network conditions present at end-hosts. For any pair of nodes x, y, the network connection from x to y is modeled as an edge (x^-, y^+) with capacity c_{xy}, and the connection from y to x is represented as edge (y^-, x^+) with capacity c_{yx}. All the edges that describe network capacity are collectively called *network edges*. The capacities are deduced via a combination of "blasting" and lightweight probing [19] (see Section 3.2).

The model generalizes to multi-homed hosts. For each ISP-i that a host x is connected to, we add two vertices x^{i+} and x^{i-}. The incoming and outgoing network connections via ISP-i respectively terminate at x^{i+} and originate from x^{i-}. For example, a node x that is multi-homed via two ISPs can be modeled using five vertices: x^0, x^{1+}, x^{1-}, x^{2+}, and x^{2-}. Four edges are added - for ISP-1: (x^{1+}, x^0) with capacity C_x^{1+}, and (x^0, x^{1-}) with capacity C_x^{1-}; for ISP-2: (x^{2+}, x^0) with capacity C_x^{2+}, and (x^0, x^{2-}) with capacity C_x^{2-}.

2.2 Solution

Given the static network model, we convert the multi-source single-sink data collection problem into a series of maximum flow problems [7]. Informally, the maximum flow problem entails finding the largest feasible flow in the network from a given source to given sink. The output of this centralized algorithm is a flow graph f^* that denotes the rate at which data must be transferred across network links, i.e., the optimal transfer



(a) System (b) Model

Fig. 3. The network graph G for a system of three nodes.

plan. The process of calculating the transfer plan requires the following steps (also see Figure 3):

1. Firstly, all the source nodes are linked to a new vertex s called the *super-source* (see Figure 3(b)). The super-source is a conceptual node from which all data ("source files") originates. A source file at node x consisting of b_x blocks is modeled by adding an edge (s, x^0) with capacity b_x (also see Figure 3(b)). We call such edges *data edges*. Using blocks rather than bytes as the atomic unit helps identify, i.e., name and order, data efficiently. For consistency, the capacities of the network edges are measured in blocks per second. Note that the total number of blocks originating from the super-source is $B = \sum_i b_i$.
2. Secondly, we apply the maximum flow algorithm to find the largest feasible flow from the super-source vertex s to a designated sink vertex t^0 within an arbitrary timespan T. This is done by translating graph G into a graph G^T. The graph translation entails multiplying the capacity of network edges (but not the data edges) by T – signifying the total amount of flow possible through a network edge within time T. For example, the network edge with capacity c_{xy} (in G) becomes $T \cdot c_{xy}$ (in G^T). If the maximum $s \rightarrow t^0$ flow value equals $B = \sum_i b_i$, then the multi-source single-sink data collection can be completed within time T. The resulting flow graph is denoted as f^T. The time complexity of solving the maximum flow problem using the push-relabel algorithm [8] is $O(|V| \cdot |E| \cdot \log(\frac{|V|^2}{|E|}))$.
3. Next, we find the smallest integer value of T for which the maximum $s \rightarrow t^0$ flow value is $B = \sum_i b_i$ blocks. We denote this value as T^* (and its corresponding flow as f^{T^*}). In [7], the theoretical upper bound on T is calculated as $|V| \cdot B \cdot C$, where C is the largest network edge capacity. Hence T^* can be found using a binary search on the range $T \in [0, |V| \cdot B \cdot C]$ and computing the maximum $s \rightarrow t^0$ flow in G^T. Hence, the total time complexity of the multi-source single-sink data collection problem is $O(\log(|V| \cdot B \cdot C) \cdot |V| \cdot |E| \cdot \log(\frac{|V|^2}{|E|}))$, where the first part

is the complexity of the binary search and the second part is the complexity of a single maximum flow computation.

4. Lastly, from f^{T^*}, we obtain the optimal transfer plan f^* with transfer rates f^*_{xy}. Let $f^{T^*}_{xy}$ be the value assigned to network edge (x^-, y^+) by the optimal maximum flow solution f^{T^*}. This is the total number of blocks that must be sent from node x to node y within timespan T^*. As such, the optimal transfer rate is $f^*_{xy} = \frac{f^{T^*}_{xy}}{T^*}$.

A reader may wonder why the graph translation (second step above) is required, when a possible alternative is to simply calculate the number of blocks that can be transferred from the super source to the sink node in a single time unit (G^1), and then repeatedly use that solution until total number of blocks B are transferred from the super source to the sink node. Such a solution would work if the amount of data at source nodes was infinite (e.g. a continuous stream of data) and our goal was simply transferring as much data as possible.

However, this strategy does not solve the problem of transferring files of finite size. More concretely, by looking back at our motivating example (Figure 1), we illustrate a scenario where this strategy does not work. In G^1, we can transfer 1 MB from node x and 5 MB from node y, for a total of 6 MB to the sink node t. As we need to transfer a total of 2000 MBs, based on G^1, one may incorrectly extrapolate that the entire process can be completed in 333.33 seconds at a sustained transfer rate of 6 MBps. However, this is not the case: at $t = 200$ seconds, node y will have finished transferring its file contents to sink node t, and the transfer rate of 6 MBps can no longer be sustained.

Lastly, we would like to point out that the empirical cost to solve this problem on a modern machine (2.8 GHz Intel Xeon processor) is low – it is under 1 second with 500 participating nodes on a complete graph, i.e., modeling link capacities for any given node pair. With 100 participating nodes, the computation completes in under 0.1 second on the same machine.

3 System Design

The Confluence system is built atop the theoretical solution described in Section 2. We first present the system assumptions in Section 3.1. Next, we detail the design of Confluence. In order to address the temporal variation of bandwidth, Confluence uses three mechanisms: (i) it periodically estimates bandwidth capacities to maintain the network graph (Section 3.2); (ii) it creates an efficient transfer plan based on these measurements and the theoretical solution (Section 3.3); and (iii) it adapts the transfer plan with changing network conditions, including leveraging partial replicas of files that are created during the transfer (Section 3.4). For reference, Table 1 summarizes important notations we use in the sections below.

3.1 System Assumptions

Firstly, we assume that all files may be subdivided into blocks. This assumption allows us to split a file into multiple pieces and send them towards the sink via different paths. Secondly, all files (and hence file blocks) are unique and need to be collected at the

Symbol	Meaning	Defined
C_x^-	ISP limit for egress traffic from node x	§ 2.1
C_x^+	ISP limit for ingress traffic to node x	§ 2.1
c_{xy}	Available bandwidth from $x \rightarrow y$	§ 2.1
b_x	Number of file blocks held at node x	§ 2.2
T^*	Optimal transfer completion time	§ 2.2
f^*	Optimal transfer plan	§ 2.2
f^*_{xy}	Optimal transfer rate from $x \rightarrow y$	§ 2.2
l_{xy}	Number of scheduled blocks (left) to be transferred from $x \rightarrow y$	§ 3.3
r_{xy}	Measured transfer rate from $x \rightarrow y$	§ 3.4
b_x^y	Number of blocks held at node x that originated from node y	§ 3.4

Table 1. A summary of important notations used in this paper.

sink node losslessly. Thirdly, we assume that failures do not occur. If a source node fails, Confluence provides no resiliency guarantees on the file blocks originating at that source node. This is acceptable as the same problem exists with Direct Transfer.

3.2 Maintaining the Network Graph

The transfer plan is calculated and updated at a node called the coordinator. The coordinator need not be a dedicated host – any one among the source nodes or the sink node can act as the coordinator. The coordinator maintains the latest network graph G based on reports from the end-nodes.

Each node in the system independently and periodically conducts measurements of the available end-to-end bandwidth to other nodes in the system. It should be noted that maintaining the state of all links, i.e., the complete graph G, is the most favorable scenario, however, the following two factors need to be considered:

- Staleness: Available bandwidth is a temporal and always-changing property of the network. Hence, repeated measurements are required.
- Cost: Actively measuring the available bandwidth expends some of the available bandwidth. Hence, the number of measurements should be minimized.

We adopt two design decisions that address both factors simultaneously. Firstly, we use pathChirp [19] to measure available end-to-end bandwidth between nodes as it provides a good balance between accuracy and measurement cost. Secondly, each node probes a small set of k nodes where $k \ll n$ (the number of nodes in the system).

By limiting the size of k, we can keep the measurements more frequent (avoiding staleness), without requiring extra bandwidth (cost of measurement). For example, consider a system with 50 nodes where bandwidth constrains a node to conducting a measurement every 180 seconds. By using $k = 49$ and performing a round-robin measurement, each link will be measured only once every 8820 seconds. However, if $k = 10$ the frequency of measurement for each link is reduced to 180 seconds. Another beneficial side effect is that only the k probed connections are used to calculate the optimal transfer plan f^*, thereby reducing the computational complexity of the algorithm.

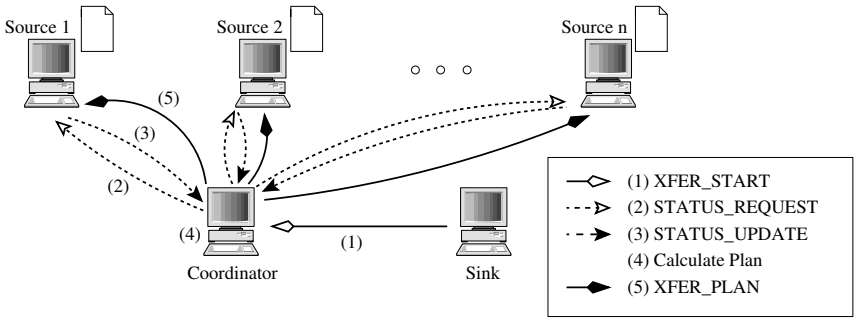


Fig. 4. An overview of the protocol used by Confluence. Note that steps (2)-(5) are repeated periodically (see Section 3.4).

However, we cannot arbitrarily reduce k – with a limited number of peers, the available bandwidth of a well-connected host may not be fully utilized. The value of k is acceptable only as long as the k peers are able to saturate the downlink capacity of the bottleneck node in the system, which is generally the sink node. In our experiments (detailed in Section 4.3), we find that $k = 10$ provides the same performance as $k = n - 1$ for a vast majority of cases for a PlanetLab type network with up to 100 sources.

The coordinator maintains a global membership graph by assigning each node k random peers, where the peer relationships are asymmetric. A given node periodically probes the available bandwidth to each of its k peers in a round-robin manner. After each round of measurements, the node reports the updated measurements to the coordinator. Upon receiving new measurements, the coordinator updates the network graph G.

Measuring ISP Connectivity A node's connectivity to its ISP is unlikely to change significantly unless it is upgraded or downgraded. As a result, this can be measured infrequently, e.g., once a day. Infrequent measurement is further supported by the fact that a node can easily monitor and update its ISP connectivity estimates during actual file transfer. As a result, recomputation of the transfer plan will quickly alleviate any suboptimalities (details are presented in Section 3.4). We use an intuitive "blasting" technique to measure C_x^+ – a host's downlink capacity will be saturated if simultaneously blasted with a continuous stream of data by numerous other hosts. Concretely, each node x independently (at random times, during periods of system idleness) requests its peers to simultaneously blast it via a TCP stream for 30 seconds. The value of C_x^+ is these blasts' peak aggregate (averaged over 5 seconds). Likewise, the node's egress capacity C_x^- can be gauged when the node simultaneously blasts all of its peers.

Note that if a node is multi-homed, the connectivity provided by an ISP can be measured via blasts to and from the subset of peers connected through that ISP. The `traceroute` utility can help deduce the list of peers connected via a given ISP.

3.3 Transfer Plan Execution

Figure 4 provides an overview of the transfer protocol used by Confluence. Any node can become the designated sink when it wishes to retrieve files. It contacts the coor-

dinator (XFER_START) with a list of source nodes. In turn, the coordinator sends a STATUS_REQUEST to all the nodes in the network to collect the latest state of the network graph G. The source nodes piggyback the size of files (in blocks) held at those nodes atop the STATUS_UPDATE response. Using the network graph G as input to the algorithm described in Section 2.2, the coordinator calculates the optimal transfer plan f^*. Based on this calculation, the coordinator sends specific transfer plan *directives* to nodes (XFER_PLAN). The directive for a node x contains the number of blocks node x must send to each peer node y. We use l_{xy} to denote this quantity.

The transfer plan directives are carried out via a push protocol (not shown in Figure 4): data is pushed from a node to all of its receivers simultaneously (in parallel). The value of l_{xy} is decremented locally at node x on each successive block transmission to node y. When l_{xy} reaches 0, node x ceases to send blocks to node y. A *source* node can start pushing the blocks originating from it as soon as it receives its directives. However, a few nodes may additionally act as *intermediate nodes* (i.e., when $\sum_i l_{xi} > b_x$), either to provide a faster transfer route to the sink or because a source node may not have direct overlay connectivity to the sink (due to having only k peers). As such, intermediate nodes need to wait for blocks to "trickle in" from their senders before they can forward such blocks to their receivers. A newly arriving block is pushed out to a receiver selected with probability equal to its share of the total number of blocks remaining, i.e., $Pr[y] = \frac{l_{xy}}{\sum_i l_{xi}}$. When the sink has received all $B = \sum_i b_i$ blocks, the transfer process is deemed complete.

3.4 Dynamic Adaptation

Both inter-flow competition and temporal variation in available bandwidth can adversely affect the transfer plan. For example, if a flow terminating at the sink achieves a better actual transfer rate than its designated optimal transfer rate, it may hog bandwidth away from other flows also terminating at the sink, leading to an increase in total transfer time. There are two approaches to combat this problem – either control the transfer rates, or adapt the transfer plan to the changing network conditions.

We adopt the latter approach of periodically adapting the transfer plan. This is more pragmatic since it has the ability to address both inter-flow competition and temporal variation in network conditions. In fact, our initial take on the problem used the first approach – flow control. Specifically, we maintained transfer rates using the cross-layered TCP Flow Control System (FCS) [14], which adjusts advertised TCP window of receivers to maintain the desired transfer rate. While FCS does better than unadulterated TCP, we unfortunately found that it still degenerates away from the optimal transfer plan for numerous scenarios due to its inability to adapt to changing network conditions. Comparing the two approaches is left as a task for future work.

Periodic Recomputation Periodic recomputation is the process of calculating the transfer plan with an updated network graph G. This process is repeated periodically (every p seconds) until the data collection task is completed. An added bonus of periodic recomputation is that it allows the system to start with weaker estimates of the

network graph G. This further justifies using a cost effective (but sometimes inaccurate) tool such as pathChirp [19] to measure available bandwidth.

During an ongoing data collection, each node x continuously monitors the transfer rate to each of its receivers (details presented in Section 4.1). We call this the measured rate r_{xy}. Every recomputation period, the coordinator sends a `STATUS_REQUEST` message to each node x. Node x responds with the number of blocks it currently holds (b_x) and the measured transfer rate (r_{xy}) for all its peers y (recall from Section 2.2 that b_x is initially the size of the file at node x). As the coordinator receives the responses from the nodes, it updates graph G's data edges with the new b_x values. It also updates G's network edges based on the network conditions. Concretely, if $r_{xy} \geq f^*_{xy} \cdot (1 - \texttt{slack})$, then $c_{xy} = \max(c_{xy}, r_{xy})$ else $c_{xy} = \max(\frac{c_{xy}}{2}, r_{xy})$. In other words, if the measured rate r_{xy} is greater than the optimal rate f^*_{xy} (given some slack), available bandwidth capacity c_{xy} is updated if it improves upon the previous estimate; otherwise, c_{xy} is reduced by up to one-half to match the recently measured r_{xy}. Given that the network conditions are always changing, the slack is necessary to avoid aggressively changing c_{xy}. Our implementation uses a `slack` value of 5%. The else clause limits the reduction of c_{xy} to mitigate the effects of a one-time network event.

After the coordinator receives all the `STATUS_REQUEST` responses, it calculates a new transfer plan (see Section 3.3). Note that the structure of the overlay remains the same (i.e., the same k peers are maintained), however, recomputation adapts the overlay workload to meet the latest network observations.

State Inconsistency In this section, we describe the two interesting issues that arise because the recomputed transfer plan is based on an inconsistent view of the network state. This inconsistency arises due to several reasons: (i) each node's status is reported at a potentially different time, since it is based on the time at which it received the `STATUS_REQUEST` message; (ii) the number of blocks b_x reported to the coordinator includes neither the blocks still pending in the outgoing TCP buffers, nor the packets that are in flight towards receivers, i.e., on the network link; (iii) the network state continues to change while the transfer plan is being calculated at the coordinator; (iv) the latency to deliver the new transfer plan to the nodes.

We tackle this inconsistency by separately handling the two issues it results in. The first issue is that the new transfer plan directive may overstate the number of blocks a node x has. This is the common case as nodes continue to transfer blocks to their peers while the new transfer plan is being computed. Thus, when a node x has transferred all its blocks, it will needlessly wait for more blocks to trickle in. To avoid this, the sink sends an explicit `XFER_COMPLETE` message to all nodes when it has received all $B = \sum_i b_i$ source blocks. Note that this issue (and its solution) does not slow down the transfer plan.

The second, less frequent issue is that the new transfer plan directive may understate the number of blocks a node x has. This occurs when node x receives a large number of packets right after it reported its status, i.e., due to TCP recovery of out-of-order packets or due to a sudden increase in the incoming bandwidth. In this case, node x stops forwarding packets when l_{xy} reaches 0 for all peers y. The remaining blocks will effectively be stranded until the coordinator learns of them and devises a transfer plan

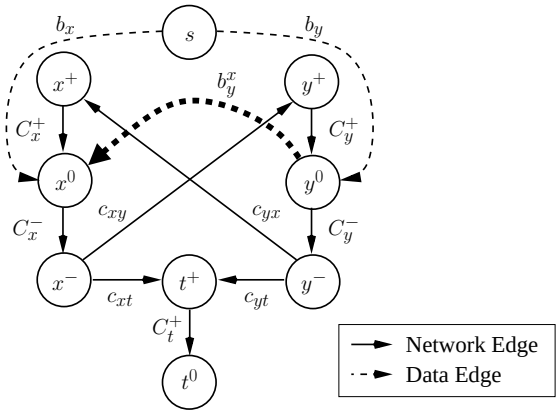


Fig. 5. A file edge with weight equal to b_y^x is added during recomputation for blocks held by node y that originated at node x.

that includes them. This will not happen until the next recomputation (at most p seconds away). To prevent needless elongation of the transfer process, the coordinator sends a special Boolean flag `final_computation` along with the new transfer directives whenever the optimal transfer time $T^* \leq p$; signaling nodes to send any stranded blocks directly to the sink.

Recomputation with Block Replication The reader may notice that a natural artifact of Confluence's transfer process is that an intermediate node x temporarily stores blocks originating from other source nodes. Our implementation of Confluence uses a conservative *purge-immediately* policy at intermediate nodes: blocks are purged as soon as they are forwarded to and acknowledged by the designated receiver. As a result, once a block leaves the origin node but before it reaches the sink, there are exactly two copies of the block in the network. During recomputation, we can use this naturally occurring replication to our advantage – by optimally choosing which of the two replica-holding nodes should forward a given file block to the sink node.

Exploiting replication requires that each file block be tracked. Each block must be tagged with a unique 2-tuple: the origination node and a sequence number (calculated locally by the origin node). This allows node y to count the number of blocks originating from node x that it currently holds. Let b_y^x be the number of blocks held by node y originating from source node x. Note that $\sum_i b_y^i = b_y$ at any given time. The list of origin nodes (and the associated b_y^x) is reported to the coordinator as part of the `STATUS_UPDATE` response.

At the coordinator, during recomputation, for each reported b_y^x, the coordinator adds a data-edge from node y^0 to node x^0 with weight equal to b_y^x (see Figure 5). This step allows the max-flow calculation to calculate the optimal solution with the option of routing up to b_y^x blocks from either node x^0 or node y^0 to the sink, because adding the (y^0, x^0) data-edge does not effect the number of blocks held at the super source node s. If the new recomputation flow f^{T^*} uses the (y^0, x^0) data-edge with capacity $\alpha \leq b_y^x$, it

implies that the new transfer plan now involves originating node x resending α blocks to the sink (via some other route) that are also currently held at intermediate node y. This may happen if the network conditions favor a route starting at node x instead of node y. To support this re-routing, node x needs to know which of its blocks are held at y. Notice that node y may have less than α blocks originating from node x by the time the new transfer plan directive arrives at node x, i.e., $b_y^x < \alpha$. As such, node y iterates through its blocks and finds the first $\min(b_y^x, \alpha)$ blocks originating from node x. Next, node y sends the sequence identifiers of these blocks to node x in a `REPLICATED_BLOCKS` message. Node x now is responsible for transferring these blocks to its receivers.

Please note that while our implementation uses the conservative purge-immediately policy to minimize storage requirements at intermediate nodes, another implementation may have intermediate nodes only lazily delete blocks based on storage needs. The latter choice may provide opportunities for increased replication, and as a result, provide better performance (at the expense of increased storage).

3.5 Overheads of Confluence

The design decisions of Confluence result in a few overheads not present with the Direct Transfer strategy. Many of these overheads have already been discussed previously, and the reader may have observed others. However, for completeness, we now present the list of the major overheads of Confluence.

Firstly, the network state represented by network graph G may be inaccurate, stale, or both. This could be due to both inaccuracies in the underlying measurement tool, and the temporal diversity in available bandwidth. Secondly, the k peers of a node may not be able to saturate capacity of the node. This may lead to suboptimal results, especially if the sink's downlink capacity is not being fully saturated by its k peers. Thirdly, Confluence suffers a delayed start in contrast with Direct Transfer. Metadata about the network graph G must be collected by the coordinator, the solution calculated, and the transfer plan directives sent out to participating nodes before the process can start. Fourthly, due to state inconsistency caused by periodic recomputation, the final set of blocks sent directly to the coordinator may delay the finish, especially if there is abnormally high inconsistency. Lastly, as Confluence needs to track each block to support replication, a small protocol overhead is added to for each data block transferred.

4 Experimental Evaluation

In this section, we present a thorough evaluation of Confluence via trace-driven simulations. We first describe our implementation and experimental methodology in Section 4.1. Recall that the goal of Confluence is to reduce the time it takes to fetch files from multiple sources to a single sink. For comparison, in Section 4.2, we discuss the performance of the Direct Transfer strategy, the most commonly used approach to transferring files from multiple sources to a single sink. We then explore the parameter space of Confluence and choose the default parameters for our experiments in Section 4.3. Finally, in Section 4.4, we compare the performance of Confluence with Direct Transfer under various scenarios. Our results show that Confluence is able to reduce trans-

fer times under many scenarios. For instance, Confluence is able to outperform Direct Transfer by up to 80%, in experiments involving 25 participating nodes.

4.1 Implementation and Experimental Methodology

Implementation In order to accurately model network bandwidth, we implement Confluence using the ns2 [15] network simulator. Our implementation of Confluence resides entirely in the application layer and uses TCP CUBIC [18] as the transport protocol.

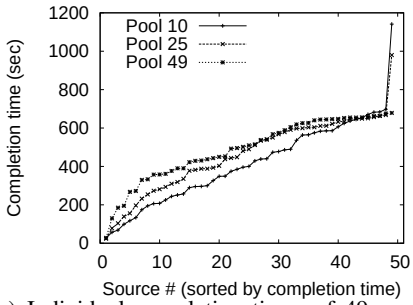
To maximally utilize network capacity, nodes must aggressively send blocks to each of their receivers. However, sending packets aggressively via TCP takes control away from the Confluence application; it cannot efficiently reroute the blocks based on a new transfer plan directive without the wasteful tear down of TCP connections. To address this issue, a Confluence node x buffers out (to the TCP stack) only one second worth of data (based on optimal transfer rate f^*_{xy}) to node y. The application buffers out another block to TCP only upon reception of an explicit ACK from a receiver. As 1 second is an order of magnitude higher than the median delays experienced on Internet routes, our TCP buffer will generously saturate f^*_{xy}.

The measured transfer rate r_{xy} is calculated by node x based on the number of ACKs received. The measured rate is kept as a running average of the last 5 seconds. If b_x becomes 0 (it can't send any more blocks because it is waiting for them to trickle in) or if l_{xy} becomes 0 (all blocks are already sent to node y), r_{xy} is not updated until the next recomputation interval. These stipulations are put in place to not penalize r_{xy} (and in turn, c_{xy} – see Section 3.4) when the sender is unable to send blocks.

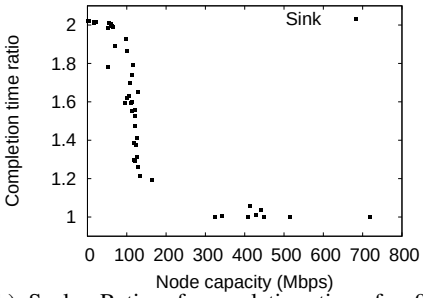
Experimental Methodology We constructed the experimental topologies based on PlanetLab traces collected by S^3 [23] on April 8, 2008. The traces include the two necessary end-to-end network measurements: available bandwidth and latency. However, there is a limitation of this data set: the information about the properties of many links is missing. We construct our experimental topologies by avoiding such links, in the following manner: starting with a random node, we iteratively constructed a node list. A new node (selected at random) was only added to the list if the links connecting the given node to all previous nodes on the list were not missing any information.

For each PlanetLab node, we create an additional ISP node. The IP link between a node and its ISP has a bandwidth capacity equal to the highest end-to-end available bandwidth observed at the corresponding PlanetLab node. Second, each possible pair of ISP nodes is connected with an IP link whose bandwidth and latency characteristics are equal to that of the measurements observed between their associated end-nodes. Note that multiple nodes from the same (DNS) domain share the same ISP. As such, our simplified reconstruction of the IP topology may be problematic. To mitigate this dilemma, we pruned all but one node (selected randomly) from each domain. In a real deployment of Confluence, with multiple hosts per domain, the selected host can act as the gateway for all other nodes in the domain. This design choice works based on the assumption that the intra-domain connections (i.e., hosts across a LAN) have greater available bandwidth than inter-domain connections (i.e., hosts across a WAN).

We are unaware of any existing systems built specifically for the n-to-1 file transfer problem that Confluence targets. Therefore we compare with a simple, but surprisingly



(a) Individual completion times of 49 source nodes



(b) Scale: Ratio of completion time for 99 sources vs. 49 sources

Fig. 6. Direct Transfer performs well with a large number of simultaneous connections.

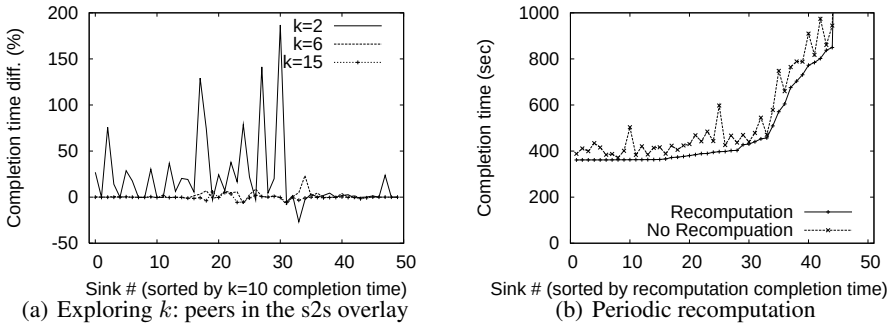
strong, Direct Transfer strategy. In Direct Transfer, the sink node downloads the files directly and simultaneously from the source nodes, using a running pool of m connections. When a download completes from a node in this pool, another source is added to the pool.

For Confluence, we use only the participating source nodes and the sink node to calculate the optimal transfer plan f^*. We believe that adding dedicated intermediary nodes will improve upon the results, however we do not explore this option in order to make the comparison with Direct Transfer a fair one.

For all of our experiments, the sink node downloads unique files of 100MB from all source nodes. The Confluence s2s overlay uses $k = 10$ outgoing peers. Further, the recomputation interval p is 15 seconds. Both of these values were selected based on experimental findings (further discussed in Section 4.3).

4.2 Direct Transfer

We found that the transfer completion time for Direct Transfer improves with increasing pool size. Figure 6(a) shows the time of completion for transfers from $n = 49$ source nodes to a sink node (selected randomly) for different values of m – 10, 25, and 49. For the Direct Transfer experiments, the source nodes were ordered randomly. As a consequence, the sink node fetches files from the first m nodes initially. Once a file is fetched completely from a source node, the sink node begins fetching from the next source node (if any remaining). The x-axis represents the source nodes, sorted by the time they completed their file transfer to the sink node. The y-axis is the transfer time (in seconds). Transfers complete faster initially for lower values of m because there is more bandwidth available per transfer. However, the total completion time is longer for lower values of m because the last few transfers lag behind. With $m = n$, the lagged flows may be just as slow, however they start at time $t = 0$ and have a longer time to complete. Put another way, the probability that a lagged flow starts after $t = 0$ increases with decreasing values of m. Hence, Direct Transfer with $m = n$ has the fastest completion time, and we use this value in all subsequent experiments.



(a) Exploring k: peers in the s2s overlay (b) Periodic recomputation

Fig. 7. Exploring the design parameters of Confluence.

Direct Transfer scales well when downloading from large numbers of sources. Figure 6(b) compares the results of two different experiments. In the first experiment, files are downloaded from 49 source nodes to a sink node. For the next experiment, we required the sink node to fetch files from 99 source nodes. To enable us to compare the two experiments, the first 50 nodes are the same in both topologies. We measure the Direct Transfer time under both scenarios via 50 different experimental runs: in a given run, one of the first 50 nodes act as the sink and fetch files from the all other remaining nodes, which act as source nodes. The x-axis represents the first 50 source nodes, sorted by the the capacity of their network connectivity. The y-axis is the completion time ratio between transferring 99 source files to transferring 49 source files. While the total data transferred in the second experiment is roughly double the first experiment (99 source files vs. 49 source files), the completion time is usually less than twice as long. In fact, when the network capacity of the sink node is large, the completion times are nearly equivalent. This is because very well-connected sink nodes have enough excess capacity to support a greater number of concurrent connections. In contrast, a sink node with lower network capacity ends up being itself the bottleneck, and cannot complete transfers any faster, even with a larger number of concurrent connections.

4.3 Exploring Confluence Parameter Space

In Section 3.2, we discussed the trade-offs between the benefits of maintaining up-to-date information on the network state vs. the cost of measuring this information. For our next experiment, we explore varying the value of k (the number of neighbors in the s2s overlay) in a system of 50 nodes. Via experimentation, we were able to determine that a small set of $k = 10$ peers is sufficient to get fast completion times. Figure 7(a) shows the total completion time for each node acting as the sink and downloading from the other 49 nodes (i.e., there is a separate run for each node acting as a sink). The x-axis represents the sink nodes, sorted by the performance of Confluence with $k = 10$. More concretely, the first few points (on the left) represent the most well-connected nodes, and the last few points (towards the right) are the least well-connected nodes. The y-axis is the relative difference in performance between $k = 10$ and other values of k,

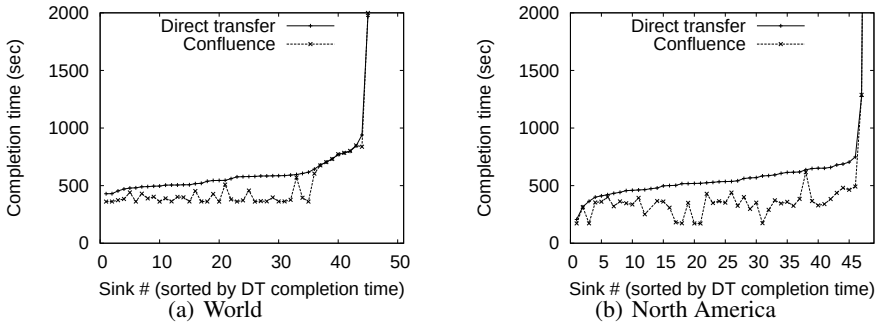


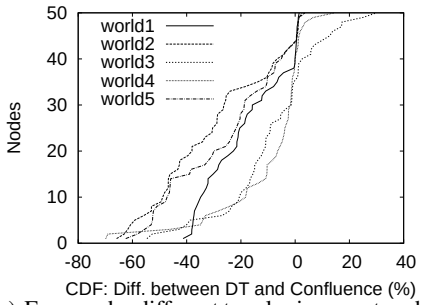
Fig. 8. Confluence outperforms Direct Transfer on both a planetary and a continental scale topology. Note that the results with long completion times are omitted. At best, Confluence finishes 70% faster, and at worst Confluence is only 2% slower (inclusive of omitted results).

with $k = 10$ value acting as the baseline. A negative value implies better performance for other values of k, a positive value implies better performance for $k = 10$.

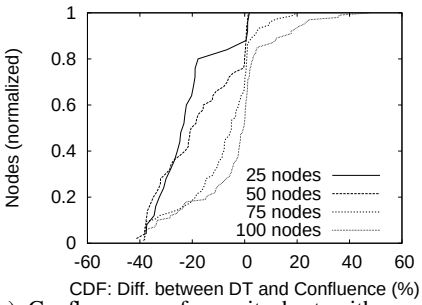
We observe that $k = 2$ performs the worst of the lot – with many results taking twice as much time as $k = 10$. It should be noted that for one special case (sink #33), $k = 2$ actually performs better than $k = 10$. This is the case because the sink node's capacity is being saturated with simply two peers, and having more peers only leads to inter-flow congestion. It shows that there may be some benefit to having a different value of k per node, especially when a node is the sink. For $k = 15$, we see that performance is almost identical to $k = 10$. Thus, we set $k = 10$ as the default for our experiments.

Figure 7(b) shows that recomputation consistently reduces the completion time. We use a recomputation interval of $p = 15$ seconds for this experiment. (The recomputation interval p is explored in the next experiment.) The plot shows the total completion time for each of the 50 nodes acting as the sink and downloading from the other 49 nodes (i.e., there is a separate run for each node acting as a sink). The x-axis represents the sink nodes, sorted by the performance of Confluence with recomputation enabled. The y-axis is the absolute completion time. We see that recomputation consistently improves performance, and in some cases, the improvement is nearly 50%. Thus, Confluence enables recomputation by default.

The last parameter we investigate is the recomputation interval p (results not shown for brevity). Like the previous experiment, we experiment with 50 nodes, with each node acting as a sink node (and all other nodes as source nodes) using 50 different runs. We find a negligible difference in performance of Confluence with a value of $p = 15$ seconds, a more aggressive recomputation value of $p = 10$ seconds, and a less aggressive recomputation of $p = 60$ seconds. However, if network conditions change, a shorter period of recomputation can adjust quicker. Thus, we pick an intermediate default value of $p = 15$ seconds.



(a) Even under different topologies, most nodes see at least some benefit by using Confluence over Direct Transfer. The performance improvement is as high as 75%.



(b) Confluence performs its best with small groups. For example, with $n = 25$, 80% of nodes see a reduction in transfer time of at least 20% over Direct Transfer.

Fig. 9. The performance of Confluence under different topologies and different sized networks.

4.4 Confluence vs. Direct Transfer

We compare Confluence and Direct Transfer using two different topologies of 50 randomly selected nodes: in the first topology, nodes were selected without restriction (i.e., world wide) and the second topology was limited to nodes within North America. For both topologies, we perform $n = 50$ simulations; each simulation had a different node act as the sink node (the remaining 49 nodes were the source nodes). Figure 8(a) shows the results from the the first topology (the results are sorted by the transfer time for Direct Transfer). We observe that Confluence outperforms Direct Transfer (with transfer time reductions of up to 40%), especially for the 35 best-connected nodes. The remaining 15 poorly-connected nodes yield similar results for Confluence and Direct Transfer as both are able to continuously saturate the available bandwidth at the sink. Figure 8(b) shows the results when the topology is constrained to North American nodes. We observe that Confluence reduces transfer times by up to 70%. These experiments demonstrate that Confluence is able to outperform Direct Transfer, due its ability to exploit both spatial and temporal bandwidth, for systems with $n = 50$ on both planetary and continental scale topologies.

Next, we show that Confluence behaves similarly given different PlanetLab topologies. Figure 9(a) shows the CDF of the difference in completion time between Direct Transfer and Confluence in a system with 50 nodes (with each node acting as the sink in separate runs). A negative x value implies that Confluence finishes $x\%$ faster than Direct Transfer with that node as the sink. For all topologies, Confluence reduces the transfer time for most nodes (as sink) – with improvement of up to 75%.

As mentioned in Section 4.2, Direct Transfer works well with a large set of source nodes. In Figure 9(b), we see that the transfer time reduced by Confluence instead of Direct Transfer decreases as the network size increases. The plot shows the CDF of the difference in completion time between Direct Transfer and Confluence in a system with a varying number of nodes (with each node acting as the sink in separate runs). On the x-axis, a negative value implies that Confluence finishes $x\%$ faster than Direct

Transfer for a given node as the sink. With $n = 25$ nodes, 80% of the nodes see an improvement of at least 20%. With $n = 50$ nodes, 70% of nodes see at least some benefit with Confluence. Thus, we conclude that Confluence is most useful when downloading files from a small set of nodes ($n \leq 50$), an appropriate setting for debugging various PlanetLab prototypes and applications (and for meshes of clouds and data-centers).

5 Related Work

Current solutions fail to efficiently address the multi-source single-sink data collection problem.

Distributing a popular file, e.g., a CD/DVD image of a recently released Linux distribution or a trailer to an upcoming Hollywood movie, to multiple hosts in a wide area network is a fairly common content distribution problem. This file transfer problem is diametrically opposite to the problem solved by Confluence, as a file is transferred from one source site ("the content provider") to multiple sinks ("end users"). This is a well-studied problem, and a plethora of solutions [10] have been proposed to efficiently complete this process. Relatedly, content distribution networks [1] efficiently provide static content to a large number of users by moving data closer to the edge of the network [20]. The popular peer-to-peer file sharing system, BitTorrent [4], can quickly disseminate popular files to multiple sinks, starting from a single source. Other peer-to-peer systems (e.g., [21]) utilize tree or mesh structures to allow users to enjoy near real-time multimedia streams. All of the mentioned approaches increase efficiency by replicating content throughout the system. In the multi-source single-sink data collection problem, explicit replication is not as directly advantageous because only a single copy of the data needs to be collected at the sink node.

CoBlitz [17] successfully leverages close-by PlanetLab nodes as intermediaries to provide speedier downloads of large files from a single source to a single sink. Confluence solves the more general problem of downloading from multiple source nodes, using an approach firmly grounded in theoretical formulation.

Within sensor networks, numerous in-network data aggregation techniques have been proposed to reduce the cost of communication [5, 11]. Data aggregation is also performed at data centers to periodically monitor cluster-wide characteristics [12]. However such data aggregation techniques may be lossy and cannot be used for on-demand lossless data collection.

Many systems have been developed to boost data transfers over Long Fat Networks (LFNs). Some approaches use multiple TCP connections per source-sink pair. For instance, GridFTP uses parallel TCP connections [9] to speed up transfer of large files across node pairs. Others have developed specialized TCP variants that excel atop LFNs, e.g., TCP CUBIC [18], TCP-Illinois [13]. Such systems reside at the transport layer and are orthogonal to any optimization techniques performed at the application layer. As such, Confluence can leverage these and newer findings in this area with minimal changes.

Lastly, it should be noted that the primary premise of Confluence is based on the key observation that the transfer process can be sped up by routing data via intermediate

nodes. Previous work [2, 3, 6] has shown that exploring multiple routes can improve connectivity and mitigate outages on the Internet.

6 Conclusion

We have presented the design, implementation, and evaluation of Confluence. Confluence builds an adaptive source-2-source (s2s) overlay to optimize simultaneous download of unique files from multiple source nodes to a single sink node. Confluence exploits spatial heterogeneity of bandwidth in a wide-area setting via a theoretically optimal solution to a flow optimization problem, and exploits temporal heterogeneity by periodically recomputing this solution. Our system implementation is built on these principles, while incorporating a series of practical design decisions.

Experiments using measurement traces from PlanetLab show that Confluence performs better than direct source to sink transfers, especially when the number of sources is in the few tens and the sink is not bottlenecked. For instance, Confluence improves total transfer time by as much as 80% (with 25 sources) compared to direct source to sink downloads. Further, we showed that Confluence performs better than Direct Transfer on both a planetary and a continental scale topology, with up to 50 nodes. As such, Confluence can be useful in many common wide-area networks, for example, across cloud computing sites and data-centers, and for prototyping services on PlanetLab.

References

[1] Akamai. Website. `http://www.akamai.com`.

[2] Aditya Akella, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. On the performance benefits of multihoming route control. *IEEE/ACM Transactions on Networking*, 16(1):91–104, 2008.

[3] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proc. of ACM SOSP*, pages 131–145, 2001.

[4] BitTorrent. Website. `http://www.bittorrent.com/`.

[5] Elena Fasoloy, Michele Rossiy, Jorg Widmer, and Michele Zorziy. In-network aggregation techniques for wireless sensor networks: A survey. *IEEE Wireless Communications*, 14(2):70–87, 2007.

[6] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of internet path faults on reactive routing. In *Proc. of ACM SIGMETRICS*, pages 126–137, 2003.

[7] Lisa Fleischer. Faster algorithms for the quickest transshipment problem with zero transit times. In *Proc. of ACM-SIAM SODA*, pages 147–156, 1998.

[8] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proc. of ACM SOTC*, pages 136–146, 1986.

[9] GridFTP FAQ: How do I choose a value for the parallelism (-p) option? Website. `http://www.globus.org/toolkit/docs/4.0/data/gridftp/rn01re01.html#parallelismvalue`.

[10] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Nicolas D. Georganas. A survey of application-layer multicast protocols. *IEEE Communications Surveys and Tutorials*, 9(3):58–74, 2007.

[11] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of ACM MobiCom*, pages 56–67, 2000.

[12] Steven Y. Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. Moara: Flexible and scalable group-based querying system. In *Proc. of ACM/IFIP/USENIX Middleware*, pages 408–428, 2008.

[13] Shao Liu, Tamer Başar, and Ravi Srikant. TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks. In *Proc. of Valuetools*, page 55, 2006.

[14] Puneet Mehra, Avideh Zakhor, and Christophe De Vleeschouwer. Receiver-driven bandwidth sharing for TCP. In *Proc. of IEEE INFOCOM*, pages 1145–1155, 2003.

[15] The network simulator: ns-2. Website. `http://www.isi.edu/nsnam/ns/`.

[16] Konstantina Papagiannaki, Sue B. Moon, Chuck Fraleigh, Patrick Thiran, Fouad A. Tobagi, and Christophe Diot. Analysis of measured single-hop delay from an operational backbone network. In *Proc. of IEEE INFOCOM*, pages 535–544, 2002.

[17] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. of USENIX NSDI*, pages 29–44, 2006.

[18] Injong Rhee and Lisong Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In *Proc. of Workshop on Protocols for Fast Long-Distance Networks*, 2005.

[19] Vinay J. Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient available bandwidth estimation for network paths. In *Proc. of PAM*, 2003.

[20] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. In *Proc. of USENIX OSDI*, pages 315–327, 2002.

[21] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Proc. of IEEE ICNP*, pages 2–11, 2006.

[22] Wanmin Wu, Zhenyu Yang, Klara Nahrstedt, Gregorij Kurillo, and Ruzena Bajcsy. Towards multi-site collaboration in tele-immersive environments. In *Proc. of ACM Multimedia*, pages 767–770, 2007.

[23] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sung-Ju.Lee, and Sujoy Basu. S^3: A scalable sensing service for monitoring large networked systems. In *Proc. of ACM SIGCOMM Workshop on Internet Network Management*, 2006.