The File Mover: High-Performance Data Transfer for the Grid *

Cosimo Anglano, Massimo Canonico
Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria (Italy)
Via Bellini 25/g - 15100 Alessandria, Italy
email:{cosimo.anglano,massimo.canonico}@unipmn.it

Abstract

The exploration in many scientific disciplines (e.g., High-Energy Physics, Climate Modeling, and Life Sciences) involves the production and the analysis of massive data collections, whose archival, retrieval, and analysis require the coordinated usage of high capacity computing, network, and storage resources. To obtain satisfactory performance, these applications require the availability of a high-performance, reliable data transfer mechanisms, able to minimize the data transfer time that often dominates their execution time. In this paper we present the *File Mover*, an efficient data transfer system specifically tailored to the needs of data-intensive applications, that exploits the *overlay networks* paradigm to provide superior performance with respect to conventional file transfer systems. An extensive experimental evaluation, carried out by means of a proof-of-concept implementation of the File Mover for a variety of network scenarions, shows the ability of the File Mover to outperform alternative data transfer systems.

1 Introduction

The scientific exploration in many disciplines, like High-Energy Physics, Climate Modeling, and Life Sciences, more and more involves the production and the analysis of massive data collections, whose size is in the order of tens to hundreds of Megabytes (and sometimes even Petabytes) [28]. The archival, retrieval, and analysis of such datasets, that are usually disseminated among scientists located over a wide geographic area, require the coordinated usage of high capacity computing, network, and storage resources. *Data Grids* [47], providing infrastructure and services for distributed data-intensive applications, have been proposed as a viable solution to the needs of these applications. Data Grids provide services that help users discover, transfer, and manipulate large datasets stored in distributed repositories, and also create and manage copies of these datasets. At minimum, a Data Grid must provide two basic functionalities [12]: a high-performance, reliable data transfer mechanism, and a scalable replica discovery and management mechanism. This paper focuses on the former issue, and assumes that the latter one is solved by using one of the various available replica management system (see [47] for an exhaustive coverage of these systems).

The need for a high-performance data transfer mechanism arises from the observation that the completion time of typical data-intensive applications, given by the sum of their execution time and of the time taken to transfer the data they need [37], is often dominated by the data transfer time. In response to this need, several data transfer mechanisms, having as common goal that of reducing as much as possible the time taken to transfer data across Grid resources, have been developed in the recent past [5, 16, 19, 20, 22, 45, 48]. These systems typically rely on the TCP or UDP protocol to move data among machines, and use various optimization techniques [8, 33, 41] to efficiently exploit the available network bandwidth available in order to minimize transfer times.

However, despite the optimizations they use, the performance that can be attained by these tools is limited by the the bandwidth available on the network path used to carry out the transfer. Therefore, the choice of a good network path in terms of available bandwidth is crucial for the achievement of satisfactory data transfer times. Unfortunately, in the TCP/IP architecture the choice of the network path used to transfer data between pairs of machines is determined by the IP routing protocols and algorithms, that base their decisions solely on connectivity information and not on performance considerations. Consequently, the routes they find are often suboptimal in terms of performance, i.e., network paths

*This work has been supported in part by the Italian MIUR grant no. RBNE01WEJT (FIRB "WebMINDS" project).

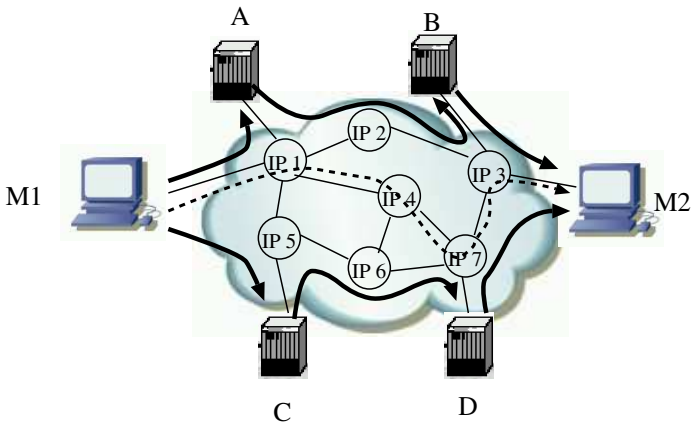

Figure 1: An example data transfer overlay network. Circles, thin lines, thick lines, and dotted lines represent IP routers, physical links, virtual link, and network paths, respectively.

connecting the same pair of hosts and exhibiting better performance exist, as observed in various experimental studies [39, 40, 44]. For instance, Savage *et al.* observe [40] that for 30 to 80 percent of the network paths chosen by the IP routing algorithms between pairs of Internet hosts, taken from a relatively large set of machines, it was possible to find alternative paths with better performance characteristics.

Another performance-limiting factor of IP routing protocols is the potentially very high amount of time required to recover from link failures, with the consequence that transfers along faulty paths may take an inordinate amount of time. As a matter of fact, the fault recovery mechanisms used by typical Internet routing protocols sometimes take many minutes to converge to a consistent form [24], and there are times when path outages lead to significant disruptions in communication lasting even tens of minutes or more [14, 29, 30].

In order to overcome the above limitations, we have developed the *File Mover*, a data transfer system that exploits an *overlay network* [15, 17] architecture to avoid using the suboptimal routes imposed by the IP routing algorithms and to quickly restart a transfer after a path failure has occurred. An overlay network is a virtual network, layered on top of a physical one (e.g., the Internet), whose member nodes are placed at the edges of the underlying physical network, communicate by means of a transport-level protocol (e.g, TCP or UDP), and agree to relay each other's traffic until the destination host is reached.

The advantages of using an overlay network to perform file transfers rather than directly transferring them over the IP network paths are manifold. First, network paths resulting in better throughput than the one chosen by the IP routing algorithms may be exploited by interposing, between the file source and the file destination, a suitable set of overlay nodes. For example, consider the overlay network, depicted in Figure 1, that comprises four overlay nodes (A,B,C, and D), and assume that a file must be transferred from host M1 to host M2. Suppose now that the IP network path connecting these two hosts traverses routers IP 1, IP 4, IP 7, and IP 3, but that the path traversing routers IP1, IP2, and IP3 has a higher available bandwidth. While any tool directly transferring the data would be forced to use the less efficient network path, one using the overlay network can exploit the more efficient path by sending data to node A first, and then to B and, finally, to M2.

Second, if more paths are available between a source and a destination machine, an overlay-based system can use them simultaneously to transfer different parts of the same file in parallel, so that download time is reduced with respect to using a single path only. For instance, in the situation depicted in Figure 1, it is possible to transfer a portion of the file on the path IP 1, IP 2, IP 3 (by using nodes A and B) and another portion on the path IP 1, IP 5, IP 6, IP 7, IP 3 (by using C and D). Multipath transfers may also be used to increase the throughput between a source and a destination machine by transferring different files at the same time when a batch of files is requested to the same server. In contrast, the parallel download of a batch of files with a conventional file transfer system requires the usage of multiple simultaneous transfer sessions between the same pair of hosts. These sessions, however, must use the same network path, so the aggregate throughput that can be achieved is limited by the path throughput.

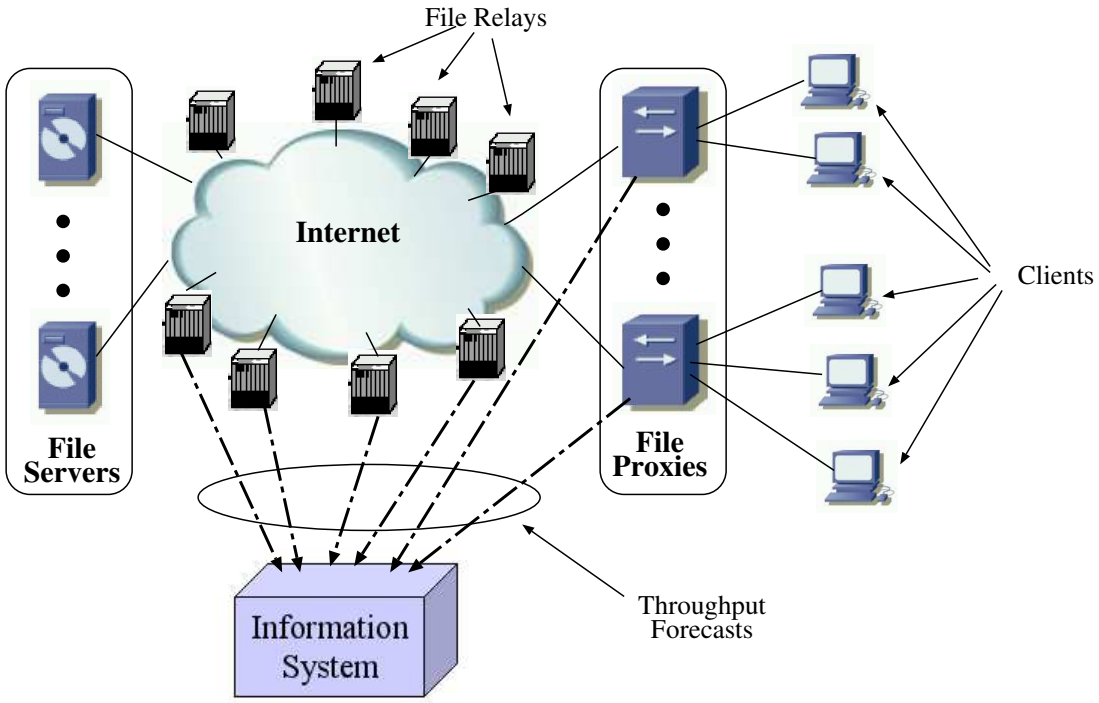

Figure 2: The File Mover architecture

Third, in case of a link failure, a shorter recovery time may be obtained by using a new sequence of overlay nodes such that the data movement does not include faulty physical links or routers. In contrast, as already mentioned, systems based on IP routing must wait until the IP routing protocols converge to a consistent configuration, and this may take a large amount of time.

The remainder of this paper is organized as follows. In Section 2, we describe the architecture of the File Mover, its behavior and operations. In order to demonstrate the viability of our approach, we have developed an implementation of the File Mover, and we have used it to carry out an extensive experimentation aimed at comparing its performance against those attained by the state-of-the-art file transfer tools using direct transfers. These experiments were performed by using a network testbed built to accurately reproduce the operational conditions of realistic production networks (like the Internet) both in terms of path capacities and traffic flows. The results obtained in our experiments, that clearly demonstrate that the File Mover maintains its promises for a variety of network scenarios and operational conditions, are discussed in Section 3. In Section 4, we place our work in the context of the related research. Finally, Section 5 concludes the paper and outlines future research work.

2 The File Mover Architecture

The File Mover is a software system, whose architecture (schematically depicted in Figure 2) is based on the overlay networks paradigm, providing a data transfer service between a *File Server* and a *Client* by using a suitable set of *File Relays* as intermediate nodes. The overlay network is fully connected, that is each File Relay may communicate directly with any other File Relay via transport-level connections (henceforth denoted as *virtual links*). Virtual links are unidirectional, since IP routing is in general asymmetric (that is the path taken by packets sent from host A to host B may be different than the one followed by packets sent from B to A).

File Proxies represent the interface between clients and the File Mover overlay network, that is when a client needs a file, it contacts one of the File Proxies, that in turn requests the file to a server holding one of its copies, and notifies

the requesting client when the transfer has been completed. [1]

Upon receiving a request, a Server determines the *virtual path* (i.e., the sequence of virtual links) connecting itself to the requesting Proxy that provides the highest available bandwidth (and, hence, in the best performance) among all the possible virtual paths among them. Since the size of the transferred files can be very large, the transfers can last a potentially very long time interval, during which the network conditions may vary so much to determine a dramatic change of the available bandwidth on the chosen virtual path. In order to avoid as much as possible these situations, the File Mover computes mid-term forecasts of the available bandwidth of each virtual link, and uses these forecasts to compute the virtual path exhibiting the highest expected available bandwidth. These forecasts are computed by measuring, at regular time instants, the available bandwidth of each virtual link of the File Mover, and by feeding these measurements to the statistical forecasting algorithms of the *Network Weather Service* (NWS) [50]. After having been computed, forecasts are stored into the *Information System* for future use. Measurements are carried out on an individual basis, that is every Server and Relay periodically measures the available bandwidth for all its outgoing virtual link (special care is taken to avoid interference among concurrent measurements). Moreover, in order to take into account also the slowdown due to the traversal of all the Relays laying on the virtual path chosen to carry out the transfer, each Relay measures also its *traversal bandwidth*, i.e. the speed (expressed in bit/sec.) at which it can move data from an input to an output virtual link. These measurement are in turn used to compute forecasts that are used, as discussed later, to compute the best virtual path connecting the Server to a requesting Proxy.

In the current implementation of the File Mover, available bandwidth measurement are taken by using UDP probes with *Iperf* [1] (other measurement tool can be however easily integrated into the File Mover), while the traversal bandwidth is measured by instrumentation of the Relay source code. It is worth to point out that the need of providing accurate forecasts is the main reason behind the introduction of File Proxies. As a matter of fact, in order to increase forecast accuracy, a suitable number of measurement must be taken for each virtual link, including those terminating into Clients. These last links, however, represent a potential problem in situations where Clients come and go, since the number of collected measurements for them might not be enough to result in a satisfactory forecast accuracy. Conversely, Proxies are a permanent component of the infrastructure, so do not pose this problem.

Let us now discuss in detail the behavior of the File Mover by describing how file transfers are carried out by its components.

2.1 Virtual Path Computation

As already discussed, the first step performed by the Server is the computation of the virtual path that will be used to carry out the transfer. The best virtual path for a given transfer is the one characterized by the highest expected available bandwidth, among all the potential virtual paths between the Server and the requesting Proxy. Recalling that data transfers are pipelined across the various virtual links of the chosen path, the available bandwidth of a given virtual path correspond to that of its slowest component, that may be either a virtual link or an intermediate Relay.

Virtual path computation is carried out by means of a variant of the Dijkstra's *single-source shortest path* algorithm [13], listed in Fig. 3, that we have modified in order to make it able to compute a path maximizing the available bandwidth (rather than the sum of individual link costs, as in the original version). This algorithm models the overlay network as a directed graph where edges correspond to virtual links and nodes to Servers, Relays, and Proxies. Edges and nodes are labeled with a real number, representing the available bandwidth (expressed in Megabits/sec., or Mbps for brevity) of the corresponding virtual link or Relay.

Before describing the algorithm, let us introduce some notation. Given a node N_i, we denote as $HighestBW(N_i)$ the available bandwidth of the best path connecting the Server to N_i, and as $Pred(N_i)$ the *predecessor of* N_i on this path. Furthermore, we denote as $BW(N_i, N_j)$ the bandwidth of the virtual link connecting node N_i to node N_j, while $BW(N_i)$ denotes the traversal bandwidth of Relay (N_i). Finally, V denotes the set of nodes in the graph.

The algorithm inspects all the nodes in the graph (the order of the inspection is immaterial) starting from the Server, and builds the best virtual path by progressively updating, for each considered node N_i, its predecessor node $Pred(N_i)$ until all the nodes have been visited. The predecessor of a node is updated only if a better predecessor (i.e., a predecessor laying on a better path) is found. At the end of the execution of the algorithm, the best path from the Server to each other node in the overlay can be built by following the chain of predecessors in a backward fashion.

[1] We assume that the identity of the Server holding a copy of the requested file is determined by the Client by means of a data location service external to the File Mover, as for instance the *Replica Location Service* [10].

1: $S = \{V\}$-Server; /* S is the set of nodes that still have to be visited */
2: $HighestBW(N_i) = 0, \ \forall N_i \ \in V$;
3: $CurNode =$ Server;
4: **while** (S is not empty) **do**
5: **for all** ($N_i \in Neighbors(CurNode)$) **do**
6: $BW1 = \min(BW(CurNode, N_i), BW(N_i))$;
7: **if** ($HighestBW(N_i) > 0$) **then**
8: $NewBW = \min(HighestBW(N_i), BW1)$;
9: **else**
10: $NewBW = HighestBW(N_i) = BW1$;
11: **end if**
12: **if** ($NewBW \geq HighestBW(N_i)$) **then**
13: $HighestBW(N_i) = NewBW$;
14: $Pred(N_i) = CurNode$;
15: **end if**
16: **end for**
17: $CurNode = Pop(S)$; /* Pop() removes an element of a set */
18: **end while**

Figure 3: The modified Dijkstra's single-source shortest path algorithm.

Furthermore, the algorithm provides also an estimate $AvailBW$ of the bandwidth that will be available on the chosen virtual path, that is used by the Server to estimate the expected transfer time as $AvailBW/Sz$, where Sz is the size in bytes of the transferred file. This feature of the File Mover may be particularly useful in many situations. For instance, when multiple copies of the same file exist in different locations, better application performance may be obtained by transferring it from the replica yielding the smallest transfer time. Similarly, if the computational resources must be reserved in advance, and users are charged also for reserved but unused time, making reservations at the right time (and not too early) may result in better resource utilization and lower resource costs.

To illustrate how the algorithm works, in Fig. 5 we report the sequence of updates to the $Pred()$ and $HighestBW()$ values for the nodes and edges in the graph shown in Figure 4, representing a scenario in which three File Relays are available to transfer a file between the *Server* and the *Proxy*. In the above figure, circles represent Relays and are labeled with the corresponding traversal bandwidth value, while arcs correspond to virtual link and are labeled with the available bandwidth on the corresponding network path. For the graph of Fig. 4, the algorithm works as follows:

1. *Iteration 1 –* $CurNode =$ *Server*: in the first iteration, for each node N_i in the graph, the algorithm sets the Server node as the predecessor node and consequently sets $HighestBW(N_i) = min(BW(S, N_i), BW(N_i))$, that is the minimum between the bandwidth of the virtual link from the Server to the node N_i and the traversal bandwidth of the node N_i, as shown in Fig. 5(a).

2. *Iteration 2 –* $CurNode = R1$: only the variables concerning node $Proxy$ are updated, since the available bandwidth of the $Server \rightarrow R1 \rightarrow Proxy$ path (9 Mbps) is higher than that of the $Server \rightarrow Proxy$) path (2 Mbps), while no changes are made for the other nodes (see Fig. 5(b));

3. *Iteration 3 –* $CurNode = R2$: only the information concerning node $R3$ is updated, since the available bandwidth of the path $Server \rightarrow R2 \rightarrow R3$ (4 Mbps) is higher than that of the path $Server \rightarrow R3$ (2 Mbps), while no changes are made for the other nodes (see Fig. 5(c)).

4. *Iteration 4 –* $CurNode = R3$: no updates are performed with respect to the previous iteration, since no improvements can be obtained for any of the other nodes by passing through R3 (see Fig. 5(d)).

5. *Iteration 5 –* $CurNode = Proxy$: this node has no neighbors, since it has no outgoing links, so no changes are made for all the nodes in the graph.

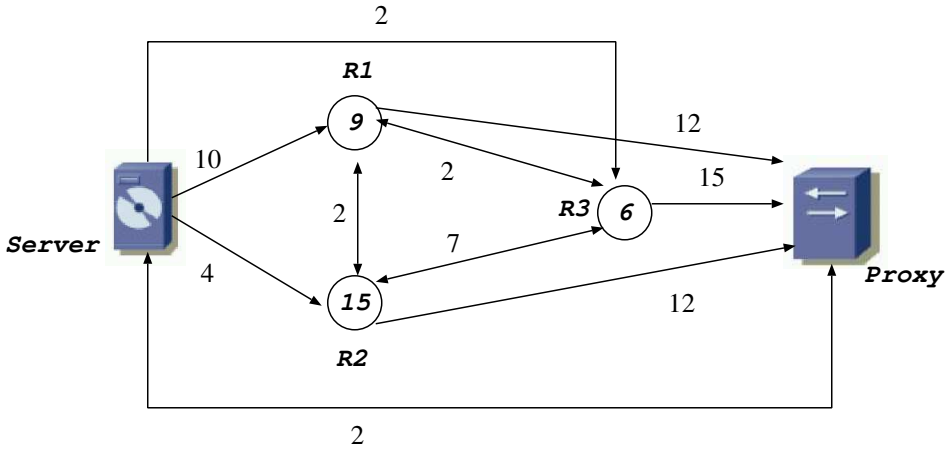

Figure 4: File Mover configuration used to illustrate the computation of the best virtual path. For readability purposes, we have assumed that the virtual links connecting the same pair of Relays have the same throughput, so that the corresponding arcs are drawn as a single bidirectional arc.

At the end of the algorithm, by looking at the Proxy row of Fig. 5(d), we see that the best virtual path has an (expected) available bandwidth of 9 Mbps, and that the predecessor of the Proxy is node R1, whose predecessor is the Server node, meaning that the best virtual path goes through relay R1 first, then through R2, and finally gets to the Proxy.

In order to avoid that ongoing transfers are slowed down by new ones, as well as to enhance the accuracy of predictions, we exclude from the computation of the virtual path for a new transfers all virtual links already in use, or that share at least a physical link with a currently used one. Note that two virtual links that share the same destination File Relay, share at least a physical link (the one connecting the File Relay to its default router). Therefore, a File Relay already involved in another transfer is not included in the graph, so it cannot be involved in more than one transfer at a time. We are aware that this is a rough simplification, as two transfers using the same virtual link would slow down each other only if the virtual link represents a bottleneck for both of them. However, the identification of shared bottleneck among network paths is an open research problem for which there are only partial solutions [23, 38], so we decided to adopt the more conservative approach outlined before.

2.2 Virtual Path Creation

Before a virtual path can be used to carry out a file transfer, it must be established, that is each virtual link must be "connected" to the next one. This is accomplished by having each File Relay in the path "connect" its input and output virtual links, so that the data sent by its preceding Relay can be forwarded to the next one along the path. As shown in Figure 6, the File Relay connects these virtual links by creating an *Input* thread, that removes the *file blocks* (see the next subsection for the definition of file block) from the input virtual link and stores them into a buffer, and an *Output* thread that extracts the file blocks from the buffer and sends them on the output virtual link.

Virtual path setup is cooperatively carried out by all the involved File Relays by means of the *virtual path allocation protocol*, that is discussed in the remainder of this section by referring to Figure 7 (showing the creation of a virtual path including three File Relays). Virtual path creation is started by the source File Server, that creates a *Transfer Manager* thread in charge of handling the transfer. This thread prepares a request message containing the addresses of all the File Relays in the virtual path and sends it to the first one (step (1)). When this File Relay receives the allocation message, it creates a local *Transfer Manager* thread (step (2)), in charge of performing some control functions before and during the transfer, that strips its address from the set of the involved File Relays, and forwards the allocation message to the second File Relay (step (3)) in the virtual path, that in turn performs the same actions. Eventually, the last File Relay in the virtual path will forward the allocation message to the File Proxy (step (7)). Upon receipt of the allocation message, the File Proxy creates its local *Transfer Manager* thread (step (8)), that in turn creates (step (9)) a *Transfer thread* (in charge of receiving and reassembling the file blocks), and sends to the last File Relay an *OK* message (step

(a) Visit to Server

Node	HighestBW	Pred
R1	9	Server
R2	4	Server
R3	2	Server
Proxy	2	Server

(b) Visit to R1

Node	HighestBW	Pred
R1	9	Server
R2	4	Server
R3	2	Server
Proxy	9	R1

(c) Visit to R2

Node	HighestBW	Pred
R1	9	Server
R2	4	Server
R3	4	R2
Proxy	9	R1

(d) Visit to R3

Node	HighestBW	Pred
R1	9	Server
R2	4	Server
R3	4	R2
Proxy	9	R1

Figure 5: Execution of the best path algorithm on the graph of Fig. 4.

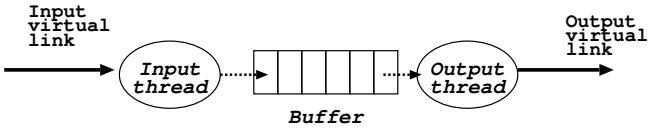

Figure 6: Connecting the input with the output virtual links of a File Relay.

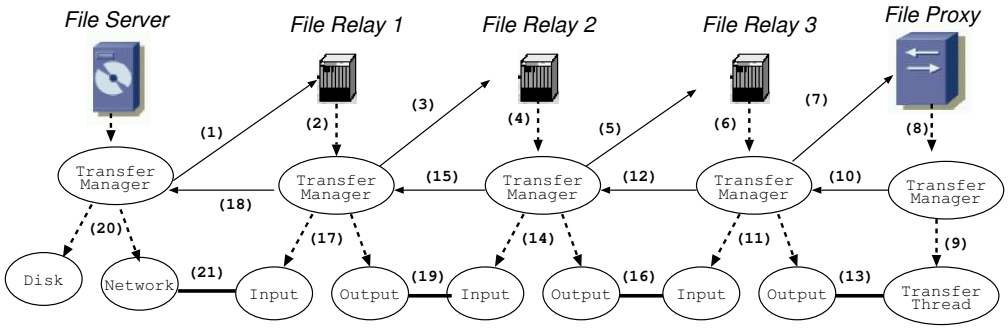

Figure 7: The Virtual Path Allocation Protocol. Continuous arrows, dotted arrows, and thick lines correspond respectively to messages, thread creation operations, and TCP connections. The numbers between parenthesis denote the step of the protocol in which the various actions are carried out.

(10)) carrying the information of the *UDT port* where the Transfer thread is waiting for the file blocks (as discussed in the next section, transfers on individual virtual links are carried out by means of the UDT protocol [20]). When the File Relay receives this message, it creates the Input and Output threads (step (11)), and sends an *OK* message to the preceding File Relay (step (12)). The Output thread then creates a UDT connection with the Transfer thread of the Proxy (step (13)), while the Input thread waits for an incoming UDT connection creation request from the Output thread of its preceding File Relay. The previous File Relay performs in turn the same actions, but this time its Output thread creates am UDT connection with the Input thread of the next File Relay (step (16)). Eventually, the *OK* message gets to the Transfer Manager of the File Server (step (18)) that, after updating the information in the Information Service in order to mark all the File Relays and virtual links used for the transfer, starts the file transfer. In particular, it creates (step (20)) a *Disk* thread, that will read the file blocks from disk, and a *Network* thread, that will send these blocks to the first Input thread after am UDT connection has been created (step (21)).

Conversely, if one of the File Relays does not respond withing a given amount of time, or if it cannot participate to the file transfer for some reasons, it does not create the corresponding Transfer Manager but replies with a *FAIL* message to the sending Transfer Manager, that will forward this message to the previous one until the File Server is informed of the allocation failure, so that it can use a different virtual path.

2.3 File Transfer

After the virtual path connecting the File Server to the destination File Proxy has been set up as discussed in the previous section, the transfer is started. In order to achieve the maximum possible performance, two issues have to be tackled, namely the minimization of the forwarding delay that each File Relay introduces when moving the file block from the input to the output virtual link, and the maximization of the throughput on the various virtual links used to transmit data among File Relays.

The minimization of the forwarding delay is achieved by pipelining the data transfers, that is the file is split into a sequence of *blocks* of suitable size, that are transmitted independently from each other. That is, at any given time, different blocks of the file are simultaneously transmitted across different virtual links, so that the transfer time is reduced with respect to a "store-and-forward" technique where the whole file is stored at each File Relays before being forwarded. Moreover, pipelining lowers the memory requirements of File Relays, as only a portion of the file needs to be stored locally at any given time. Pipelined data transfers are implemented as follows. In order to maximize the rate at which data blocks are injected in the network, the Disk thread of the File Server continuously reads blocks from disk and stores them into a buffer shared with the Network thread. The Network thread, instead, repeatedly reads a block from the buffer and sends it on the virtual link connecting it to the first File Relay. On each File Relay in the virtual path, the Input thread receives file blocks on the input virtual links and stores them into a buffer (see Figure 6), while the Output thread reads blocks from the buffer and sends them to the next File Relay. If the output link is slower than the input link, the blocks waiting to be transmitted remain in the buffer until they are transmitted on the next virtual link.

The maximization of the throughput on each virtual link is achieved by using the *UDP-based Data Transfer* (*UDT*) protocol [20] rather than plain UDP or TCP. UDT is an application-level protocol specifically designed to maximize file transfer performance over wide-area networks, that uses UDP to transfer data between pairs of hosts, and combines rate, window and delay-based control mechanisms to deliver high-throughput and low-loss data transmission. The choice of UDT is motivated by the observation that, as shown by recent work [7], it is the most effective data transfer protocol among those available today, and can be easily integrated into a working prototype thanks to the availability of libraries for various operating systems [6].

3 Experimental Evaluation

In order to show that the File Mover is able to obtain performance higher than alternative file transfer systems, we developed a C++ implementation that relies on the *UDT* libraries version 3.0, and have used it to perform an extensive experimentation in which its performance have been compared with those attained by *UDT-sendfile*, an UDT-based file transfer tool distributed as part of the main UDT software distribution. We restricted our comparison to UDT-based tools since, as shown in [7], it was the best performer available at the moment of this writing.

In order to obtain reproducible and at the same time realistic results, we ran our experiments on a local testbed built in such a way to mimic as realistically as possible the characteristics and dynamics of real networks. On this testbed, in addition to the Information System (a machine running a MySQL DBMS), we deployed 5 Relays, 1 Server, 1 Proxy, and 1 Client. We decided to use our own, locally controlled testbed rather than publicly available ones (e.g., PlanetLab [4] or Emulab [49]) in order to maintain control over the workload ran on the composing machines and the traffic load injected into the network. In this way, we could ensure experiment repeatability, tool comparison in the same operational conditions, and accuracy of result interpretation, while at the same time maintaining the realism of the environment in which experiments were executed.

We performed a large set of experiments in which various overlay configurations (differing in the capacities of the virtual links interconnecting the various File Mover components) and background traffic scenarios have been considered. As discussed later in this section, the results we obtained clearly show that the File Mover outperformed *UDT-sendfile* for all the workloads and in all the network scenarios we considered.

In the remainder of this section we describe (a) the characteristics of our network testbed (Sec. 3.1), (b) the scenarios in which the comparisons were performed (Sec. 3.2), and (c) the results we obtained (Sec. 3.3).

3.1 The Network Testbed

For our testbed, we used a cluster of PCs running the Linux operating system, each one equipped with an AMD 1.6 GHz Athlon, 724 MB of main memory, a 60 GB hard disk, and two Fast Ethernet network interfaces. The machines were dedicated to our testbed (i.e., no other applications were executed during the experiments), and did not have any external network connections, so no external traffic could enter the testbed network.

In order to reproduce the behavior of a real network, we equipped our testbed with suitable mechanisms for the specification and enforcement of the capacities of virtual links, as well as for the injection of background traffic streams (i.e., traffic streams due to other applications using the network) on them.

Virtual link capacity is set and enforced by means of the *Traffic Control* (TC) mechanisms available in the Linux kernel [2]. TC works by associating with each network interface a set of queues, each one representing a virtual (unidirectional) link towards a specific host, and by enforcing a user-defined transmission rate for each of them. In our testbed, on each machine a queue was created for each virtual link directed towards the other File Mover components. We chose TC rather than alternative possibilities (e.g., NistNET [3]) since this resulted in maximal stability and minimum overhead.

We developed two different mechanisms for the injection of background traffic. The first one generates real traffic (i.e., real packets are transmitted across virtual links), and relies on the *Distributed Internet Traffic Generator* (D-ITG) [31]. D-ITG is a platform capable of producing packet-level traffic whose characteristics (inter-packet departure time and packet size) can be specified as random variables having various probability distributions, and of generating traffic at the network, transport, and application level. Each machine of the testbed runs a generator process, that is in charge of generating the background traffic for all the virtual links departing from that machine.

While this mechanism generates a very realistic workload, it places a rather heavy demand on the CPU of each machine (we measured an occupation of up to 20% of the CPU for each generated stream), that greatly interferes with the File Mover. In order to avoid such interference, we have developed an alternative, trace-driven mechanism that reproduces the effects of the background traffic (namely, the reduction of the bandwidth available on each virtual link) without actually generating and sending the packets. This mechanism uses a trace containing samples of the available bandwidth for each virtual link in the overlay, collected at regular intervals. The background load generator traverses the trace, sets the capacity of the virtual link to the next value contained in the trace by using TC, and then waits until the next time instant in the trace to repeat the same operation. For example, if the trace file reports that at time $t = 10$ the available bandwidth on a virtual link is 30 Mbps, while at time $t = 20$ it drops to 10 Mbps, the generator (a) at time $t = 10$ sets the capacity of the virtual link to 30 Mbps, (b) suspends itself until time $t = 20$, and (c) wakes up at time $t = 20$ and sets the capacity of the virtual link to 10 Mbps. The overhead of this method is negligible and, as we verified experimentally by comparing the file transfer throughput obtained with the emulated and the real traffic, results in realistic effects on file transfers.

In our experiments, the traces were collected by running D-ITG (in isolation) on all the machines of the testbed, with the traffic parameters discussed in Section 3.2.2, and by sampling the available bandwidth of each virtual link every 10 seconds.

Table 1: Distribution of virtual link capacities

Capacity (Mbps)	Fraction (%)
Uniform(0,20)	30
Uniform(20,50)	9
Uniform(50,80)	6
Uniform(80,100)	55

3.2 Experimental Scenarios

Let us describe now the scenarios in which our experiments have been performed. We considered three different overlay configurations, obtained by varying the capacity of the various virtual links (see Sec. 3.2.1) interconnecting the Server, the Proxy, and the Relays, and three different background traffic workloads (see Sec. 3.2.2), for a total of 9 scenarios.

3.2.1 Overlay Configurations

In order compare the File Mover and *UDT-sendfile* in different operational conditions, we considered three different overlay configurations in which we varied the ratio α_{cap} of the capacity [2] $Cap(S,P)$ of the direct virtual link interconnecting the Server (S) and the Proxy (P) over the capacity $HighestCap(S,P)$ of the best virtual path among them that includes at least two virtual links, that is:

$$\alpha_{cap} = \frac{Cap(S,P)}{HighestCap(S,P)} \tag{1}$$

Intuitively, the higher α_{cap}, the higher the advantage of using the direct Server-Proxy virtual link to carry out the transfer. For our experiments, we chose the following three configurations of the overlay:

- **Configuration A**: $\alpha_{cap} = 1.03$ (i.e., the throughput of the direct link is 3% higher than that of the highest-capacity overlay path), corresponding to a situation in which that the direct transfer is usually the best choice, regardless of the background traffic load;
- **Configuration B**: $\alpha_{cap} = 0.88$ (i.e., the throughput of the direct link is 12% lower than that of the best-capacity overlay path), corresponding to a situation is which either the direct or the overlay-based transfer may be the best solution, depending on the background traffic load;
- **Configuration C**: $\alpha_{cap} = 0.63$ (i.e., the throughput of the direct link is 37% lower than that of the best-capacity overlay path), corresponding to a situation where there is a clear advantage in using the overlay, regardless of the background traffic load.

The above characterization specifies only the ratio of the capacities of a subset of the virtual links of the overlay, but nothing has been said yet about the actual capacities of all the virtual links. These capacities are set in such a way to closely reproduce the characteristics of a realistic network setting by using the results presented in [25] (summarized in Table 1), where the capacity distribution of the network paths among PlanetLab nodes has been reported. As can be seen from the above table, the capacity of about one third (30%) of the virtual links is uniformly distributed between 0 and 20 Mbps, of more than half (55%) is uniformly distributed between 80 and 100 Mbps, while the rest of them have intermediate capacities.

The three configurations used for our experiments have been generated by repeatedly generating in a random fashion the capacities of the individual virtual links, according to the distribution of Table 1, until a capacity assignment resulting in the corresponding α_{cap} value has been generated. The resulting three configurations are depicted in Figs. 8, 9, and 10 (where the small boxes labeled with integer numbers correspond to File Relays), while the corresponding values for $Cap(S,P)$ and $HighestCap(S,P)$ are reported in Table 2. For the sake of readability, in Figs. 8, 9, and

[2] In this paper we define the capacity of a network path as the maximum throughput that the path can provide to a flow when there is no cross traffic [34].

Table 2: $CAP(S, P)$ and $HighestCap(S, P)$ values for the overlay configurations (in Mbps)

Topology	$Cap(S, P)$	$HighestCap(S, P)$	α_{cap}
A	92	89	1.03
B	85	96	0.88
C	59	93	0.63

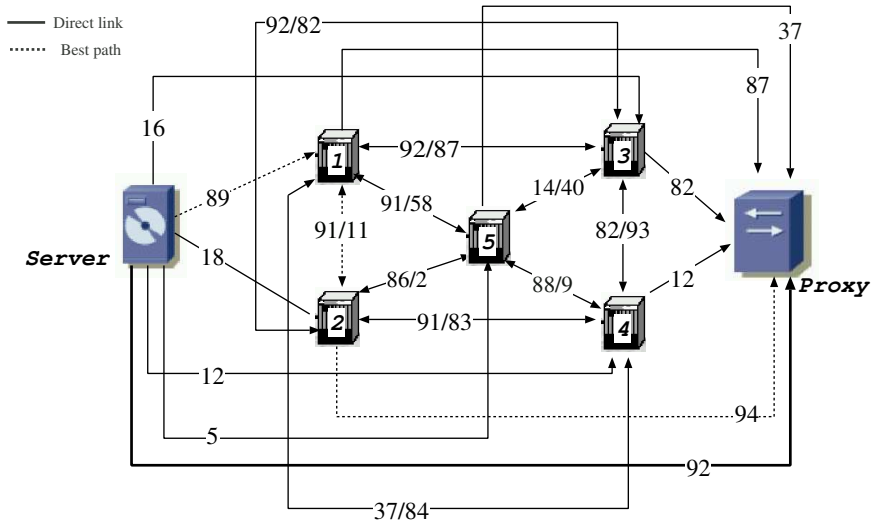

Figure 8: Configuration A: $\alpha_{cap} = 1.03$

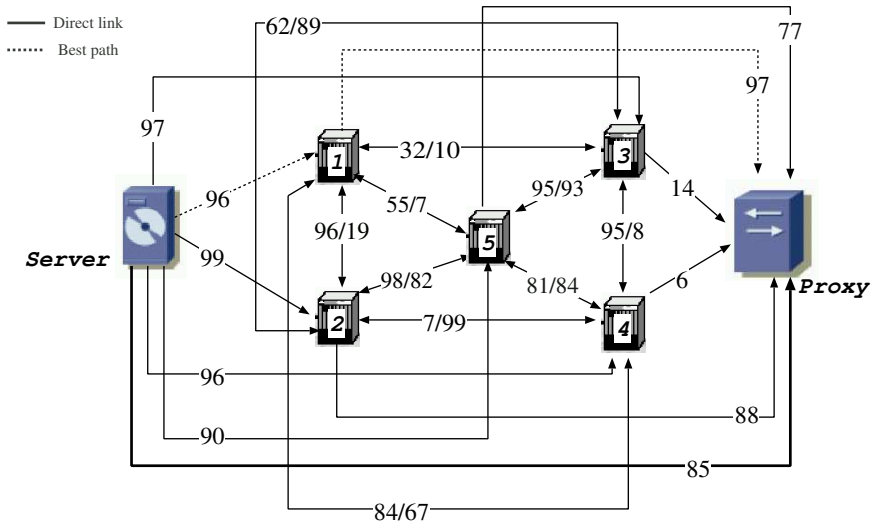

Figure 9: Configuration B: $\alpha_{cap} = 0.88$

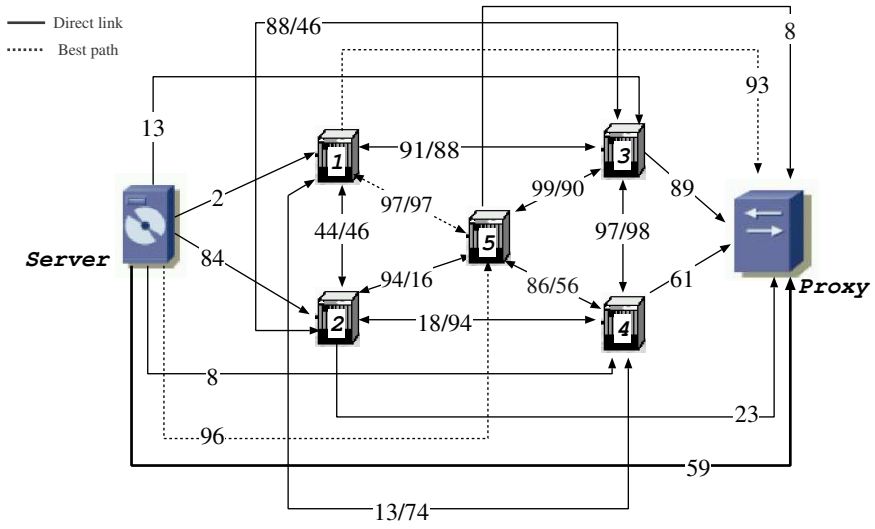

Figure 10: Configuration C: $\alpha_{cap} = 0.63$

10 the two virtual links connecting the same pair of Relays (recall that, as discussed before, IP routing is in general asymmetric so virtual links are unidirectional) are shown as a single, bi-directional edge labeled with two capacity values: the first values corresponds to capacity of the virtual links connecting the lower index Relay with the higher index one (e.g., Relay 3 to Relay 4), while the second one is associated with the virtual link connecting the higher index to the lower index Relay (e.g, Relay 4 to Relay 3). For instance, in Fig. 8 the capacity of the virtual link connecting Relay 3 to Relay 4 is 82 Mbps, while that of the virtual link connecting Relay 4 to Relay 3 is 93 Mbps. Furthermore, the highest-capacity path has been highlighted by drawing the corresponding virtual links as dotted lines, while the direct link between the Server and the Proxy has been drawn as a bold, continuous line.

3.2.2 Background Traffic Load

As already discussed, in order to mimic the behavior of a realistic production network, we injected background traffic on all the virtual links of our overlay testbed. This traffic consists in a set of data streams transmitted between pairs of testbed nodes, each one characterized by a set of parameters related to its *intensity* (i.e., how much resources it uses) and *dynamism* (i.e., how it uses these resources). The parameters of a stream related to its dynamism are its lifetime, the protocol used to transport data, and the distribution of packet size, while intensity is expressed in terms of the time elapsing between the generation of two consecutive packets (the *inter-packet time*).

For the sake of realism, for our experiments the characteristics of these streams have been set according to the latest research results in the field of Internet traffic characterization. Although an analysis of the relevant literature reveals that no single traffic model exists that takes into consideration all the parameters mentioned above, there are studies that address each of the above characteristics separately. We have therefore developed our own traffic model, whose parameters are summarized in Table 3, that puts together the results of these studies by setting the various parameters as follows:

- *Lifetime distribution*: according to the results in [9], we assume that (a) about 45% of the streams last 2 seconds or less, (b) about 53% of the streams last less than 15 minutes, and (c) about 2% of the streams last up to several hours or days (in our experiments, the maximum stream lifetime was set to one day).

- *Protocol type distribution*: we use the protocol distribution reported in [18], where a longitudinal study of various academic, research and commercial sites lasting more than four and a half years (1998-2003) revealed that as far as the number of transferred bytes is concerned, 80% of the traffic is due to TCP streams, almost 20% is due to UDP, while all the other protocols sum up to less than 1%. Consequently, in our experiments we considered only TCP and UDP streams with proportion of 80% and 20%, respectively, of the total.

Table 3: Common background traffic parameters

Stream parameter	**Value**
Lifetime distribution	45% less than 2 sec 53% less than 15 minutes 2% up to 2 hours
Protocol type distribution	80% TCP 20% UDP
Packet size distribution	25% Uniform(40,44) 50% Uniform(45,576) 25% Uniform(577,1500)
Inter-packet time distribution	Exponential (parameter set as indicated in Table 4)

- *Packet size distribution*: the distribution of packet sizes has been set according to [27], where a study lasting more than 10 months revealed that the size of about 25% of the packets exchanged between two network endpoints was uniformly distributed between 40 and 44 byte (the minimum packet size for TCP), of about 50% of the packets was uniformly distributed between 46 and 576 bytes, while for the the remaining 25% was uniformly distributed between 577 and 1500 byte (the maximum Ethernet payload size).
- *Inter-packet time distribution*: as indicated by recent work [43], the distribution of the time elapsing between the generation of two consecutive packets can be closely approximated by an exponential, so we adopted this choice. The parameter of the exponential, corresponding to the average number PPS of *packets sent per time unit*, is set (as discussed later in this section) in such a way to obtain different background load intensities.

In order to perform our comparison in various operational conditions, we defined three different background traffic workloads corresponding to situations of low, intermediate, and high intensity. The intensity of background traffic is defined as the fraction of the *average overlay capacity* (*AvgCap*) that it consumes, that is in turn defined as the arithmetic average of the capacities of the individual virtual links. The three background traffic workloads considered in our study, denoted in the rest of the paper as *Low*, *Medium*, and *High*, correspond to intensities of $20\%, 50\%$, and 80%, respectively. The parameter PPS of the inter-packet time distribution for the background traffic streams transmitted over the various virtual links is computed as

$$PPS = \lfloor \frac{AvgCap * BgTrafIntensity}{MaxPacketSize} \rfloor \quad (2)$$

where $BgTrafIntensity$ is the intensity of the background traffic, and $MaxPacketSize$ is the maximum packet length. As an example, for *Configuration B* and the *Low* intensity value, recalling that $AvgCap = 65$ Mbps and that $MaxPacketSize = 1500$ bytes, we have that:

$$PPS = \lfloor \frac{65 * 10^6 * 0.2}{1500 * 8} \rfloor = 1083 \text{ packets/sec.}$$

The average capacity and the average number of packets transmitted per second for all the overlay configurations and the background load intensities considered in our experiments are reported in Table 4.

3.3 Experimental Results

To assess whether the File Mover is able to provide advantages with respect to traditional file transfer solutions based on IP routing, we performed a set of experiments consisting in the transfer of files of various size for all the scenarios discussed above. Our experiments were aimed at evaluating the following three features of the File Mover:

Table 4: Average number of packets per second used in the experiments

Overlay configuration	**Average overlay capacity (Mbps)**	PPS		
		Low	Medium	High
A	58	967	2417	3867
B	65	1083	2708	4333
C	63	1050	2625	4200

- the *performance gain* that it provides with respect to UDT-sendfile. The performance metric used to carry out this comparison was the *average effective throughput*, computed as the arithmetic average of the throughput obtained in a set of five transfers of the same file (this latter throughput value was computed as the ratio of the file size over the time taken to transfer it);
- the *accuracy* of the transfer time predictions it computes, quantified by the *relative error* between the average predicted and measured throughput (note that the transfer time can be directly computed from the throughput);
- its *adaptability* to variations of the network conditions due to background traffic, measured by the number of different virtual paths used by the File Mover to carry out all the transfers of the same file (two paths are considered different if they differ in at least one Relay).

For our experiments, we considered two different file transfer workloads. The first one consisted in a set of transfers of a single file, and was aimed at showing that the File Mover is able to identify and use a virtual path providing a throughput higher than the virtual link connecting Server and the Proxy. The second transfer workload, consisting in the simultaneous transfers of two files between the same Server-Proxy pair, was aimed at demonstrating the ability of the File Mover to simultaneously exploit several virtual paths when more file transfers are requested simultaneously.

In order to compare the File Mover and *UDT-sendfile* in the same operational conditions, the measurement experiments were organized into a set of rounds, each one consisting in a transfer carried out with the File Mover, immediately followed by one performed with *UDT-sendfile*. In this way, both transfers were performed in very similar conditions. The time elapsing between two consecutive rounds was a random variable uniformly distributed between 5 and 15 minutes. Of course, by using a testbed totally under our control, we could have obtained the same network conditions by running all the File Mover experiments first, restarting the testbed, and performing the *UDT-sendfile* transfers exactly at the same time instant from the beginning of the injection of background traffic. However, we decided to adopt the organization of experiments in rounds since (a) the batch of experiments could be completed in a much shorter time, and (b) a validation consisting in a few transfers executed by separating the experiments performed using the File Mover from those using *UDT-sendfile* showed practically identical results.

The results obtained in our experiments, discussed in detail in the following subsections, demonstrate that the File Mover achieves performance very similar (in the most unfavorable scenarios we considered), better (in the intermediate scenarios), and much better (in the most favorable scenario) than *UDT-sendfile*.

3.3.1 Workload 1: Single File Transfers

For this workload, we performed experiments for three different sizes of the transferred files, namely $100, 500$, and 1000 MegaBytes (MB).

The first set of results, reported in Table 5, corresponds to the three scenarios resulting from the injection of the different background traffic intensities (*Low, Medium*, and *High*) on *Configuration A* of the overlay. For this particular overlay configuration, we report only the results for the 500 MB case, since both the File Mover and *UDT-sendfile* achieved practically identical performance for all the file sizes. This depends from the fact that, for this particular configuration of the overlay, the highest available bandwidth was always exhibited by the direct virtual link between the Server and the Proxy, regardless of the background traffic intensity, so this link was used by the File Mover in all

Table 5: Workload 1: results for Configuration A, 500 MB transfers.

Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	83.58	83.58	0 %	87.0	4.34%	1
Medium	83.07	82.61	0.55%	86.24	3.81%	1
High	83.56	83.56	0%	80.81	-3.28 %	1

Table 6: Workload 1: results for Configuration B.

100 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	81.19	78.52	3.4%	91.43	12.6%	1
Medium	77.16	76.63	0.69%	82.05	6.3%	1
High	74.58	73.77	1.1%	84.77	13.6%	1

500 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	85.95	77.21	11.3%	89.83	4.5%	2
Med	77.75	72.78	6.8%	83.93	7.9%	2
High	73.75	73.11	0.87%	79.93	8.37%	3

1000 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	80.83	77.69	4.04%	91.06	12.6%	2
Med	71.42	69.12	3.37%	85.73	20%	2
High	73.95	71.81	2.98%	85.62	15.7%	2

the experiments. Since the File Mover uses UDT to transfer data across individual virtual links, its performance were identical to those attained by *UDT-sendfile*.

Besides the practically identical values for the average throughput (columns *Average throughput* of Table 5), the results we obtained show that the average throughput predicted by the File Mover for the various transfers (column *Est. thr.*) is very close to the value actually obtained, so the resulting relative error (column *Error*) is very low (4.34% in the worst case). As a final consideration, we note that the number of different paths chosen by the File Mover (column *Num. paths*) was always 1, and corresponded to the direct Server-Proxy virtual link.

Let us consider now the experiments performed on the scenarios corresponding to *Configuration B* of the overlay, whose results are reported in Table 6. As can be observed from the average throughput values, the File Mover constantly outperformed *UDT-sendfile*, although the performance gain was relatively modest in all but one case (*Low* background traffic intensity and 500 MB file). However, it is important to observe that even in this overlay configuration, where the lower capacity of the direct Server-Proxy virtual link (that was 88% of that of the best virtual path) was often compensated by the effects of the background traffic on the other virtual links, the File Mover performed better than *UDT-sendfile*. This was the result of the ability of the File Mover to adapt to changes in the available bandwidth on the various virtual links caused by the presence of dynamically-varying competing background traffic. The adaptability of the File Mover is demonstrated by the fact that in all the scenarios for *Configuration B* the number of different virtual paths (columns *Num. paths* in Table 6) was larger than 1 (except for the 100 MB transfers). Furthermore, while the

Table 7: Workload 1: results for Configuration C.

100 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	78.20	53.89	45.11%	85.27	9.04%	3
Medium	76.24	47.20	61.52%	83.35	9.32%	2
High	73.17	52.37	39.71%	84.51	15.49%	2

500 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	72.76	54.28	34.04%	86.40	18.7%	2
Medium	77.62	48.35	60.53%	80.83	4.13%	3
High	75.12	47.83	57.05%	82.30	9.55%	2

1000 MB transfers						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	80.34	53.62	49.83%	84.83	5.58%	2
Medium	72.01	48.50	48.47%	81.66	13.4%	3
High	70.32	47.77	47.20%	82.66	17.5%	2

background traffic intensity had an obvious influence on performance, the file size did not have any significant impact, as can be seen by comparing the throughput values obtained with files of different sizes for the same background traffic intensity.

Another observation that can be made by looking at Table 6 is that prediction accuracy is lower than in the previous scenario. This is due to the fact that, unlike the previous case, the virtual paths used for the transfers are composed by two or more virtual links, so the prediction errors on individual links accumulate and result in a higher cumulative error.

Let us now consider the experiments performed with the scenarios based on *Configuration C* of the overlay where, as discussed in Sec. 3.2.1, the direct Server-Proxy virtual link is characterized by a capacity 37% lower than the best virtual path. The results we obtained in these experiments, reported in Table 7, show that the File Mover achieves performance significantly better than *UDT-sendfile* (with gains ranging from 34% to 61%), for all the background traffic intensities and file sizes. Of course, this is direct consequence of the fact that, even when the background traffic intensity reaches 80% of the average capacity of the overlay (for the *High* case), the File Mover is able to identify and use a virtual path exhibiting a higher available bandwidth than the direct Server-Proxy virtual link (if such a virtual path exists). As for the previous overlay configurations, also the adaptability of the File Mover to variations of the network conditions is rather good, as indicated by the number of different virtual paths used to carry out the various transfers. Finally, also in these scenarios the prediction accuracy is lower than the first scenario for the same reasons mentioned before (in particular, in this last case, the prediction error ranges between 4% and 18%).

3.3.2 Workload 2: Batch File Transfers

In the previous set of experiments, the Proxy requested to the Server the transfer of a single file, that resulted in the usage of a single virtual path. However, in many cases it may happen that the Proxy simultaneously requests a set of files to the same Server. In these situations, the Server may exploit the availability of various virtual paths connecting the same Server-Proxy pair to simultaneously transfer all these files (by properly scheduling the transfers if the number of files exceeds that of available virtual paths). In order to demonstrate that the File Mover is able to take advantage of this opportunity, we performed a set of experiments for all the scenarios previously discussed in which two files were

Table 8: Workload 2: results for all the overlay configurations.

Configuration A						
Background	**Average throughput (Mbps)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	100.74	84.33	19.46%	85.60	-15.03%	1
Medium	82.86	77.51	6.9%	77.50	-6.46%	2
High	77.26	84.21	-8.82%	71.08	-7.99%	2

Configuration B						
Background	**Average transfer time (sec.)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	77.37	74.92	3.27%	83.90	8.44%	2
Medium	91.14	78.00	16.85%	66.31	37.4%	2
High	74.58	72.71	2.84%	74.67	0.12%	3

Configuration C						
Background	**Average transfer time (sec.)**			**Prediction accuracy**		**Num.**
Traffic Load	*File Mover*	*UDT*	*Gain*	*Est. thr.*	*Error*	**paths**
Low	84.23	84.16	0.08%	87.69	4.10%	2
Medium	95.92	81.92	17.09%	82.22	-14.28%	1
High	80.02	69.82	14.61%	69.42	-13.24%	3

simultaneously requested to the Server. The size of the first file was set to 1 GB, while that of the second one was 500 MB. For these experiments, the Server allocated the best virtual path to the larger file, and the second best virtual path (not including any Relay belonging to the first virtual path) to the 500 MB file, since this allocation minimizes the time taken to complete the transfer of both files. The metrics chosen for the comparison between the File Mover and *UDT-sendfile* was the same used in the previous set of experiments, and were computed from the average throughput of the various transfers, that in this case was computed as the ratio of the total amount of transferred data (1.5 GB) over the time taken to complete the transfer of both files. The results obtained in these experiments, reported in Table 8, clearly show that the File Mover achieves performance better than *UDT-sendfile* (for all but one case – *High* background traffic intensity on *Configuration A* – where *UDT-sendfile* obtained better performance), with gains ranging from about 1% to more than 19%. The prediction accuracy also in this case was acceptable, with absolute errors below 15% (in all but one case – *Medium* background traffic intensity on *Configuration B*). Finally, the adaptability of the File Mover to variations in the network conditions is also clearly evident also in this case, as shown by the results reported in column *Num. paths* of Table 8.

4 Related Work

The problem of devising new mechanisms for efficient data transfer across Grid infrastructures has received a great attention in the recent past, and many file transfer systems have been proposed in the literature. These systems can be broadly classified in two classes.

The first class includes systems based on direct transfers between the source and destination machines, and can be further divided into systems that rely on TCP, and systems that use UDP. TCP-based tools (e.g., GridFTP [45], BBFTP [5], and bbcp [22]) try to overcome some intrinsic limitations of the TCP protocol by using suitable optimization techniques [26, 46] (e.g., proper setting of the TCP window size and usage of parallel TCP streams). Conversely, UDP-based tools (e.g., FOBS [16], Tsunami [48], SABUL [19], and UDT [20]), use UDP to move data, and employ rate-based control algorithms to avoid to saturate the network path used to move the data. However, in spite of these optimizations, the reliance of these systems on IP routing limits, as already discussed, the performance that they can

attain. Conversely, the File Mover is able to exploit network paths exhibiting better performance than those selected by the IP layer. Furthermore, the File Mover takes advantage of the superior performance offered by UDT to speed-up transmissions over individual virtual links.

The second class of file transfer systems are based, as the File Mover, on the overlay network paradigm. Systems in this class include the Logistical Session Layer [42], two overlay-TCP systems proposed by different authors [35, 21], RON [15], and Detour [39]. The File Mover differs from these systems in three important regards. First of all, these systems are not specifically conceived to improve file transfer performance, but rather they are aimed at improving TCP performance with respect to "vanilla" TCP relying on IP-layer routing. In contrast, the File Mover adopts a series of optimizations and techniques aimed at improving file transfer performance, such as the UDT protocol to move data across individual end-to-end connections, and *selfish routing* [36] for different file transfers occurring between the same pair of hosts (i.e., different virtual paths may be used for simultaneous transfers between the same Server-Proxy pair). Second, all these systems route data on the paths that provide the best *current* performance, thus in case of changes in network performance, already-established TCP connections are forced to traverse a possibly suboptimal network path. Conversely, the File Mover chooses the network paths that are expected to yield the best performance in the medium-term future, so that performance degradations due to changes in network conditions are less likely to occur during a file transfer. Third, the File Mover seeks to achieve predictability in the transfer times, in order to support effective resource scheduling, while this feature is totally absent in the alternative overlay-based systems.

5 Conclusions and Future Work

In this paper, we have described the File Mover, a file transfer system based on the overlay network paradigm that is able to exploit network paths that cannot be used by classical "end-to-end" file transfer approaches. We have developed a proof-of-concept implementation of the File Mover, and used it to carry out an extensive experimentation for various network operational scenarios. Our results clearly indicate that the File Mover is able to (a) outperform alternative approaches, (b) adapt itself to changes in the network conditions, and (c) provide reasonably accurate predictions of file transfer times.

There are several avenues of investigations that we plan to explore in the future. First of all, we plan to study techniques aimed at increasing system scalability, such as distributed databases for the Information Service and measurement techniques requiring less than N^2 probes for N hosts (e.g., [11]). Second, we plan to study techniques aimed at enhancing file transfer performance. The first of these techniques consists in creating local caches on individual Relays, so that subsequent transfers for the same file (or portions of it) can be started from the replica yielding the best performance. The availability of multiple copies (complete, or even partial) of the same file would enable us to exploit striped transfers, in which different portions of the same file are simultaneous transferred from different Relays, as for instance done in [32]. Another optimization technique we plan to investigate consists in using dynamic reconfiguration strategies enabling a Relay to change a portion of the virtual path on which it lays while engaged into a transfer when changes in network performance occur. Finally, we plan to study better resource management policies in which Servers, when computing the best virtual path, take into considerations also Relays already engaged into ongoing transfers, since sometimes it might be more profitable to wait until a busy Relay becomes available, rather than using only the available ones.

References

[1] Iperf: The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects/Iperf.

[2] Linux advanced routing & traffic control. http://lartc.org/.

[3] The nistnet project. http://snad.ncsl.nist.gov/nistnet/.

[4] The planetlab project. http://www.planet-lab.org/.

[5] The bbFTP – Large Files Transfer Protocols Web Site. http://doc.in2p3.fr/bbftp.

[6] The udt web site. Visited on Aug. 26th, 2006.

[7] Cosimo Anglano and Massimo Canonico. Performance analysis of high-performance file-transfer systems for grid applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(8):807–816, 2006.

[8] B. Tierney. TCP Tuning Guide. http://www-didc.lbl.gov/TCP-tuning/.

[9] N. Brownlee and K. Claffy. Understanding internet traffic streams: Dragonflies and tortoises. *IEEE Communications Magazine*, 40(10):110–117.

[10] M. Cai, A. Chervenak, and M. Frank. A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table. In *Proc. of Supercomputing 2004*, Pittsburgh, PA, USA, Nov. 2004. IEEE Press.

[11] Yan Chen, David Bindel, and Randy H. Katz. Tomography-based overlay network monitoring. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 216–231, New York, NY, USA, 2003. ACM Press.

[12] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[13] T.H. Cormen, C.L. Leiserson, and R.L. Rivest. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.

[14] Michael Dahlin, Bharat Baddepudi V. Chandra, Lei Gao, and Amol Nayate. End-to-end wan service availability. *IEEE/ACM Trans. Netw.*, 11(2):300–313, 2003.

[15] D.G. Andersen and H. Balakrishnan and M.F. Kaashoek and R.Morris. Resilient Overlay Networks. In *Proc. of 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[16] P.M. Dickens. FOBS: A Lightweight Communication Protocol for Grid Computing. In *Proc. of Europar 2003*, Klagenfurt, Austria, August 2003.

[17] D.D. Doval and D. O'Mahony. Overlay Networks: A Scalable Alternative for P2P. *IEEE Internet Computing*, pages 79–82, July–August 2003.

[18] Marina Fomenkov, Ken Keys, David Moore, and K Claffy. Longitudinal study of internet traffic in 1998-2003. In *WISICT '04: Proceedings of the winter international synposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004.

[19] R. Grossman, M. Mazzucco, H. Sivakumar, Y. Pan, and Q. Zhang. Simple Available Bandwidth Utilization Library for High-Speed Wide Area Networks. *The Journal of Supercomputing*, 34(3), December 2005.

[20] Yunhong Gu and Robert L. Grossman. Optimizing udp-based protocol implementation. In *Proc. of the Third International Workshop on Protocols for Fast Long-Distance Networks PFLDnet*, 2005.

[21] H. Han, S. Shakkottai, C.V. Hollot, R. Srikant, and D. Towsley. Overlay TCP for Multi-Path Routing and Congestion Control. In *IMA Workshop on Measurements and Modeling of the Internet*, Jan. 2004.

[22] A. Hanushevsky, A. Trunov, and L. Cottrell. Peer-to-Peer Computing for Secure High Performance Data Copying. In *Proc. of the 2001 Int. Conf. on Computing in High Energy and Nuclear Physics (CHEP 2001)*, Beijng, China, September 2001.

[23] D. Katabi, I. Bazzi, and X. Yang. A Passive Approach for Detecting Shared Bottlenecks. In *Proc. of IEEE Int. Conf. on Computer Communications and Networks*, Arizona,USA, 2001.

[24] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. In *Proc. of ACM SIGCOMM*, pages 175–187, Stockolm, Sweden, September 2000.

[25] Sung-Ju Lee, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Rodrigo Fonseca. Measuring bandwidth between planetlab nodes. In *Proc. of the Passive and Active Measurement Workshop (PAM 2005)*, pages 292–305, 2005.

[26] M. Mathis and R. Reddy. Enabling High Performance Data Transfers. http://www.psc.edu/networking/perf_tune.html.

[27] S. McCreary and K. Claffy. Trends in wide area IP traffic patterns - A view from ames Internet exchange. *Proceedings of the 13th ITC Specialist Seminar on Internet Traffic Measurement and Modelling, Monterey, CA*, 2000.

[28] H.B. Newman, M.H. Ellisman, and J.H. Orcutt. Data-Intensive E-Science Frontier Research. *Communications of the ACM*, 46(11), Nov. 2003.

[29] V. Paxson. End-to-End Routing Behavior in the Internet. In *Proc. of ACM SIGCOMM*, pages 25–38, Stanford, CA, August 1996.

[30] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM*, pages 139–152, Cannes, France, September 1997.

[31] A. Pescape, S. Avallone, and G. Ventre. Analysis and experimentation of internet traffic generator, 2004.

[32] J. Plank, S. Atcheley, Y. Ding, and M. Beck. Algorithms for High Performance, Wide-Area, Distributed File Downloads. *Parallel Processing Letters*, 13(2), June 2003.

[33] R.S. Prasad, M. Jain, and C. Dovrolis. Socket Buffer Auto-Sizing for High-Performance Data Transfers. *Journal of Grid Computing*, 1(4), December 2003.

[34] R.S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth estimation: metrics, measurements techniques, and tools. *IEEE Network*, Dec. 2003.

[35] H. Pucha and Y.C. Hu. Overlay TCP: Ending End-to-End Transport for Higher Throughput. In *Proc. ACM SIGCOMM*, Philadelphia, PA, USA, Aug. 2005. ACM Press.

[36] L. Qiu, Y.R. Yang, Y. Zhang, and S. Shenker. On Selfish Routing in Internet-Like Environments. In *Proc. of ACM SIGCOMM '03*, Karlsruhe, Germany, Aug. 2003. ACM Press.

[37] K. Ranganathan and I. Foster. Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids. *Journal of Grid Computing*, 1(1):53–62, 2003.

[38] D. Rubenstein, J. Kurose, and D. Towsley. Detecting Shared Congestion of Flows Via End-to-end Measurement. *IEEE/ACM Transactions on Networking*, 10(3), June 2002.

[39] Stefan Savage, Thomas Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: A Case for Informed Internet Routing and Transport. *IEEE Micro*, 19(1):50–59, Jan. 1999.

[40] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas Anderson. The end-to-end effects of internet path selection. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 289–299. ACM Press, 1999.

[41] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *Computer Communications Review*, 28(4), October 1998.

[42] M. Swany and R. Wolski. Data logistics in network computing: The Logistical Session Layer. In *IEEE International Symposium on Network Computing and Applications (NCA '01)*, Cambridge, MA, USA, Oct. 2001. IEEE Press.

[43] M Molle T Karagiannis and M Faloutsos. Long-range dependence - ten years of internet traffic modeling. *IEEE Internet Computing*, 8(5):57–64, 2004.

[44] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Proc. IEEE Infocom '01*, Anchorage, Alaska, April 2001.

[45] The Globus Team. GridFTP: Universal Data Transfer for the Grid. http://www.globus.org. White Paper.

[46] B. Tierney. TCP tuning Guide for Distributed Applications on Wide Area Networks. *Usenix ;login Journal*, page 33, Feb. 2001.

[47] S. Venugopal, R. Buyya, and K. Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing. *ACM Computing Surveys*, 38(1), June 2006.

[48] S. Wallace. Tsunami File Transfer Protocol. In *Proc. of First Int. Workshop on Protocols for Fast Long-Distance Networks*, CERN, Geneva, Switzerland, February 2003.

[49] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, 2002.

[50] R. Wolski. Dynamically Forecasting Network Performance Using the Network Weather Service. *Cluster Computing*, 1(1):119–132, Jan. 1998.