# The Internet Backplane Protocol Data Mover Module

*Erika Fuentes {efuentes@cs.utk.edu}*
Logistical Computing and Internetworking Laboratory
Computer Science Department, University of Tennessee

***Abstract***

*The Internet Backplane Protocol (IBP) is a mechanism that implements storage of data in the network using shared physical resources to support Logistical Networking in large scale; it provides a series of functions to handle the data. This data is stored in what is known as IBP depots, which also provide data management services. The Data Mover Module is a plug-in tool designed to enable advanced data transfers (high performance/reliable) in IBP and to provide point-to-multipoint data transfers (as a form of Overlay Networking, hiding data processing and performance issues from the user, and allowing developers to add new data movers with flexibility depending on the requirements of the applications.*

## 1. Introduction

### 1.1 The Internet Backplane Protocol

The Internet Backplane Protocol (IBP) is middleware for managing and using remote storage [1]. It acquired its name because it was designed to enable applications to treat the Internet as if it were a processor back plane. IBP is composed of servers (depots) and clients. The IBP Client side is composed of a series of functions that conform the API that can perform a variety of operations over the data located in the depots (IBP allocations), some examples of these, are implemented by functions such as IBP_load, IBP_store, IBP_manage, IBP_copy, and IBP_mcopy. The IBP Server or depot contains the handlers for the requests. IBP supports Logistical Networking, which combines transmission with storage resources to provide scalability and QoS [2].

The IBP_copy call can be used to copy data from one source depot to a single target, this function essentially offers access to a simple data transfer using a single TCP stream; it is built into the IBP depot writes the data to the local file system.

In order to move the data to more than one target using IBP_copy it is necessary to implement an external mechanism at user level using multiple calls to this function (i.e. multithread copy), or even using only IBP_load and IBP_store, but this could raise trust and performance concerns. In here lies the necessity for an internal function that gives ability to enable advanced data transfers providing point-to-multipoint copies, hiding the implementation details and giving the user access to a variety of methodologies to perform this task.

### 1.2 The Data Mover Module

Data mover is a plug-in tool designed to perform a point-to-multipoint data transfer as a form of overlay networking; it is an external module that can access the IBP allocations using standard IBP API functions (IBP_store and IBP_load) implementing the IBP_mcopy service in the IBP depot and allows handling point-to-multipoint copy requests from the client (see Fig.1).

This program runs independently from IBP; it is created as a separate process from the original IBP server, it is self-contained and flexible and can provide access to a variety of operations and protocols to perform the transfers. The options currently implemented work over TCP and UDP protocols. The TCP option supports sequential and simultaneous modalities, and the UDP option uses the "Blaster" algorithm and currently supports only point-to-point and simultaneous point-to-multipoint versions. The Data Mover module has been integrated to be part of the IBP services, but remaining as an independent process, which hides all the data processing and performance details from the user, but allows the developer to add new types of data movers with flexibility depending on the requirements of the environment or application.

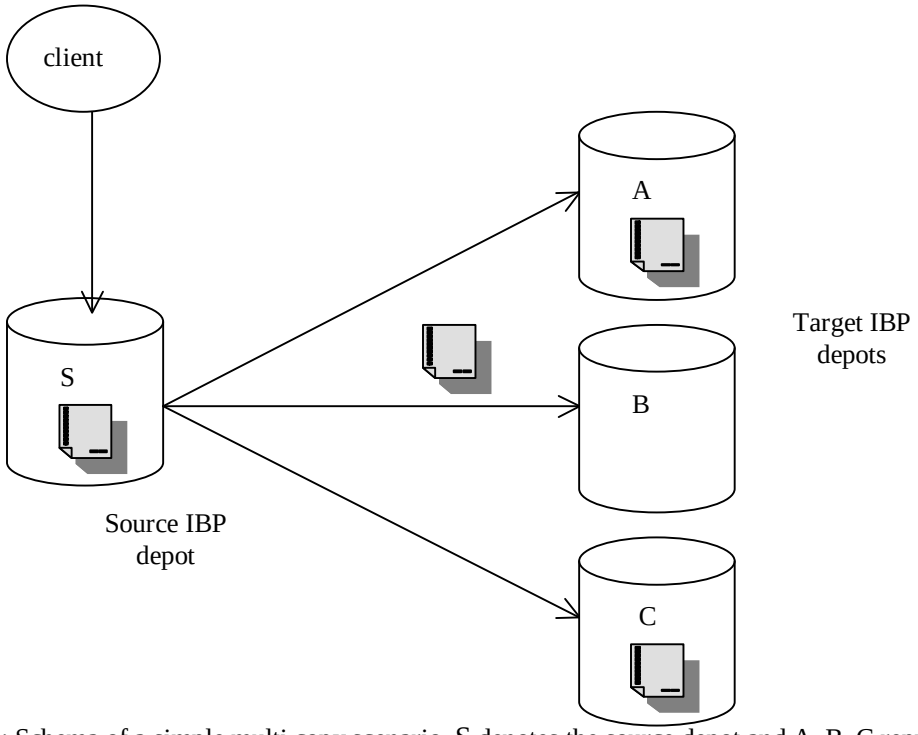


Figure 1: Schema of a simple multi copy scenario, S denotes the source depot and A, B, C represent the target depots.

## 2. Design

Four main design parts can be identified in the Data Mover scheme; the first one is the client (user) API that is part of the IBP Client library. The second one is the interface between the plug-in and the IBP Server code, the third part involves all the main Data Mover plug-in functions and management, and the last part contains the data movers. Each of these parts is defined in the following subsections.

### 2.1 Client API: IBP_mcopy

This function operates in the client side (fig 6), and is part of the IBP client library; through its parameters, the user can specify which kind of data mover (protocol and method) wants to use to carry out the data transfer each target with specific options. This interface also specifies which are the source and target IBP capabilities and associated depots, as well as the data size and starting offset for the data to copy and the number of targets desired. This API will be described in detail in the implementation section, including the data types return values and specific behavior and parameters for the existent data movers.

### 2.2 Server Interface: handle_mcopy

This function is located within the IBP depot (fig 7), is the counterpart of IBP_mcopy, it receives and serves the request in two parts: the first one indicates the size of the second request depending on how many targets will be involved in the copy, and the second one contains all the data necessary to perform the copy to each of the target depots. This function then spawns a new process that executes the Data Mover control that takes charge the request, waiting for the return of the data mover and reports the outcome to the client.

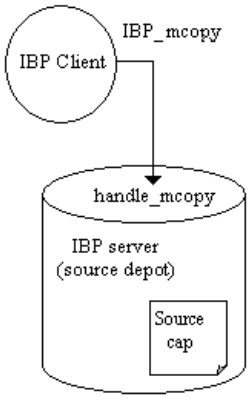


Figure 6. IBP_mcopy call is located in the client side

### 2.3 Data Mover Control (DMC)

This is the core of the Data Movers, it can be considered as a manager of the functionality; it receives the instructions from the depot through handle_mcopy, parses the arguments and parameters, processes them and chooses and executes the type of data mover required as a separate process with the respective parameters.

The Data Mover Control works both in the source depot and in each of the target depots, and the data transfer is done based in the typical client-server paradigm; the DMC has two modalities depending where it has been originated, either in the source or in a target depot and needs to perform several activities: spawn a local Data Mover process in the source depot that behaves as a client, then it sends a message to each of the targets with the type of data mover and parameters to indicate that they have to invoke a symmetric DMC server process, this is done through an internal request function known as IBP_datamover with parameters such as target capabilities, type of data mover and options, finally it executes the desired data mover.

After both DMC's have been created, the client reads the data (associated with an IBP capability) from the local storage and sends it through the network; the servers receive it on the other side and write it as an IBP capability to their local storage system. This behavior is illustrated in figure 2. After the transfer is done, the data mover returns success or failure to the DMC process, which passes this result to the IBP depot process and this back to the IBP Client (user).

**2.4 Data Mover programs**

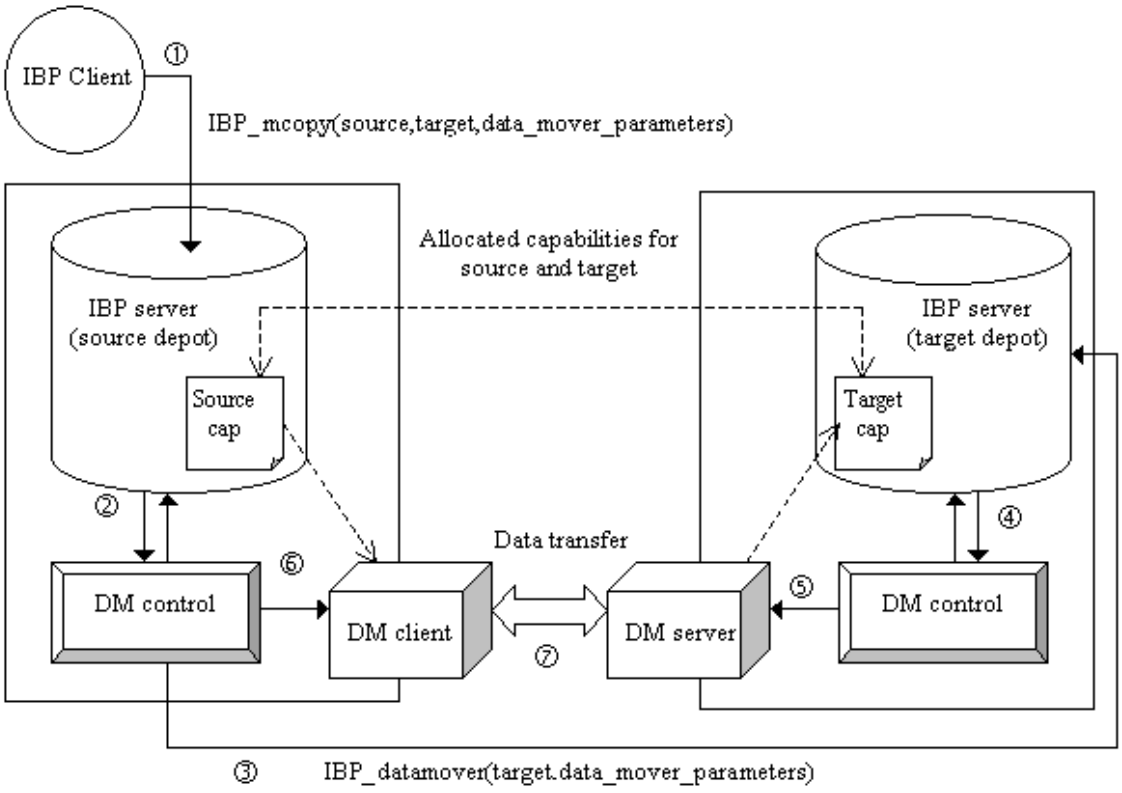


Fig.2 The Data Mover general scheme design: this picture shows the three main design components and how they relate to each other (arrows). Dotted lines indicate IBP calls implemented independently from the data movers, black lines indicate data flow within the data mover modules.

# 3. Implementation and Integration with IBP

For the implementation we can identify three different layers extracted from the design described in the previous section, these are: integration with IBP, Data Mover Control, and Data Mover programs or plug-ins.

**3.1 Integration with IBP**

This is the uppermost layer and it contemplates the **`IBP_mcopy`** in the IBP client side and the **`handle_mcopy`** function in the IBP server-depot side, as described in the previous sections, this code's purpose is basically to create an interface between the IBP code and the data movers, translate the parameters of one into the other's, create the DMC process and report return values

from it to the user. **IBP_mcopy()** is a blocking call that returns only when all required data is successfully copied from source depot to each of the targets. Table 1 presents the IBP_mcopy API in detail

| | Variable name | Variable type |
|---|---|---|
| Parameters | pc_SourceCap | IBP_cap |
| | pc_TargetCap[] | IBP_cap |
| | pi_CapCnt | unsigned int |
| | ps_src_timeout | IBP_timer |
| | ps_tgt_timeout | IBP_timer |
| | pl_size | unsigned long int |
| | pl_offset | unsigned long int |
| | dm_type[] | int |
| | dm_option[] | int |
| | dm_service | int |
| Return value | | unsigned long int |

Table 1. IBP_mcopy API: IBP_cap data type defines a capability, and IBP_timer holds the timeout for the depot and the sync. IBP_mcopy() copies up to size bytes, starting at offset, from the storage area accessed through the IBP read capability source, and writes them to the storage area(s) accessed through the IBP write capability(ies) target[]. The number of target depots is CapCount. For this call to succeed, source must be a readcap returned by an earlier call to IBP_allocate(), target must be a writecap returned by a similar call.

### 3.2 Data Mover Control

It is implemented as a plug-in module to the IBP depot, it is activated by forking a new DMC process whenever an IBP_mcopy or IBP_datamover command is detected; the new process takes over control, it parses the command line, identifies the kind of service, the type of data mover requested and executes the transfer. The first Data Mover Control is generated in the source depot, after parsing the argument list creates a series of IBP_datamover commands and sends one to each of the target depots. In each of the targets, the IBP depot receives the command and creates a local Data Mover process that act as servers.

After a Data Mover process is created, it runs as an independent process and it has two main parts:

- Control process: it parses the command line passed from the IBP depot, strips the parts of the command (capabilities, targets, ports and data mover options), identifies the targets and data mover options and operation mode (client or server), and creates the arguments required by the second part with this data; it is also responsible for recognizing and choosing the data mover program to be executed.
- In the second part a new process is generated using `fork` and `exec`, the new process the one that transfers the data from the source to the target(s). There exist several implementations that run over different protocols (TCP and UDP) and use different methodologies (sequential, round-robin, optimized threaded); the new process is also independent from the Data Mover main calling process and from the IBP depot.

Figure 3 shows the general data flow followed for the implementation of this part of the data mover module.

**3.3 Data Mover Plug-ins**

The data moving process is at the lowest level (see fig.2), is here where the protocols and transfer methods are implemented. All the data movers are composed of two pieces of code, one that implements the client side, and one for the server side. The client side is the more complex piece since it must handle transfers to multiple targets with different cases like multiple transfers sequentially, simultaneous or threaded. These client implementations at this time need to receive the same uniform parameters, which are:

- File descriptor of the associated source capability in the IBP depot
- Full string containing the actual capability (file name) in the source depot
- Size of the file to transfer
- Offset in the file where to start to read the data to be transferred
- Full strings containing the capabilities in each of the target depots
- Options: e.g. port numbers where the server part will be running in the target depots
- Type of data mover used for server side
- Type of service for client side

As for the server side, there are currently only two kinds of servers, for TCP and UDP because they only need to implement a basic functionality in one host, which is one of the targets, by reading from the socket established with the client and writing to the local storage system the data received, and this is generic to all the clients of its same protocol.

Table 2 shows how the choices of data movers are defined in the API using the parameters dm_types[] and dm_service, the first one specifies the data mover type for the target depots, and the second specifies the corresponding type for the source depot.

| | Targets | Source | Description |
|---|---|---|---|
| TCP | DM_UNI | DM_SMULTI<br>DM_PMULTI | Sequential transfers<br>Simultaneous transfers |
| UDP | DM_BLAST | DM_MBLAST | UDP based transfers |

Table 2. API options for the existing data mover plug-ins.

## 4. Experimental results

IBP_mcopy is a more general facility than IBP_copy, designed to provide access to operations that range from simple variants of basic TCP-based data transfer to highly complex protocols using multicast and real-time adaptation of the transmission protocol, depending of the nature of the underlying backbone and of traffic concerns. In all cases, the caller has the capacity to determine the appropriateness of the operation to the depot's network environment, and to select what he believes the best data transfer strategy.

The experiments completed concentrate in the following operations that the Data Mover software can support:

- Point-to-multipoint through simple iterated TCP unicast transfers
- Point-to-multipoint through simultaneous threaded TCP unicast transfers.
- Fast, reliable UDP data transfer over private network links [3]

The design goal of the Data Mover plug-in and the function IBP_mcopy is to provide an optimized point to multipoint transfer tool, as well as a support for different protocols and

methods for data transfer. In order to visualize and compare the behavior of the different methods to perform the data movement, three main experiments were completed under a specific environment where each of the nodes involved were interconnected with a stable, fast link within a local area network (LAN). The subsections 4.1.1 and 4.1.2 describe these experiments and their corresponding results, as well as a comparison of their performance and possible optimizations, using the Data Mover module, with the commonly used methods, such as IBP_copy and TCP respectively.

### 4.1 TCP Data Movers performance

This experiment concentrates in the comparison of the three variants of TCP data movers currently implemented, and the optimizations that they characterize.

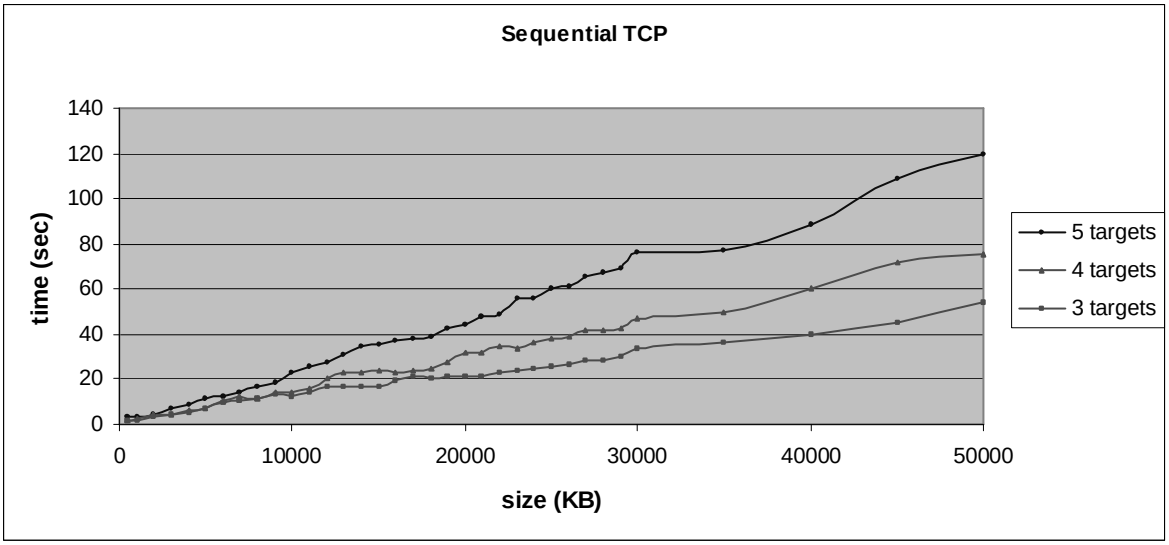


Figure 3.a

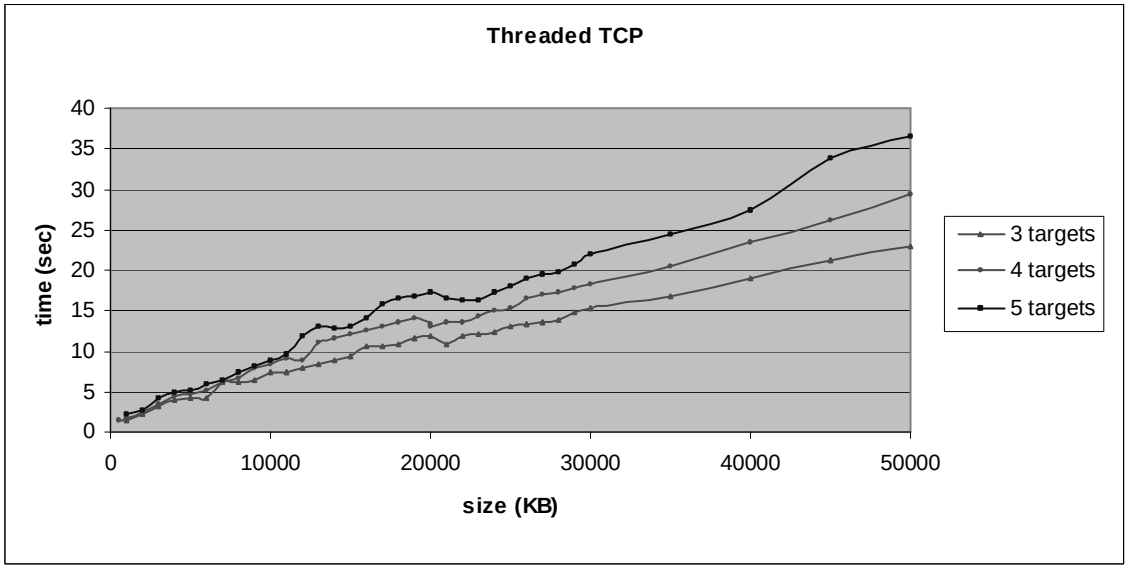


Figure 3.b

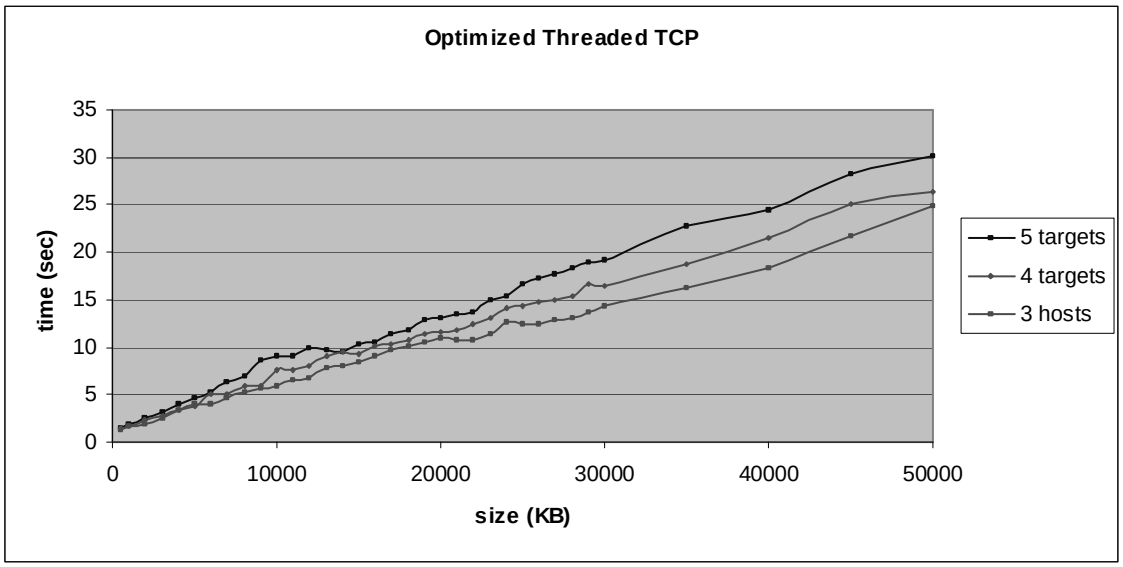


Figure 3.c

As we can observe by comparing the three figures above the transfer time increases proportionally to the size. Figure 3.a is the most conservative method, it transfers the data sequentially to each host, one after the previous one has been completed, as the chart shows the time increases significantly as does the number of targets. Figure 3.b shows a first implementation to perform simultaneous TCP transfers using threads, is more efficient but can be improved (note gaps between curves - same data is read by as many threads as copies are needed); the time increase is still proportional to the number of hosts but not as dramatic as the first case. Figure 3.c shows a simultaneous transfer using "optimized" threads, the implementation improves the performance to some extent by performing less read accesses to the source data than the previous one, in here the time variation for the different number of targets is relatively small (note gaps between curves – only one read).

In figure 4, we can observe and compare the rates in MB/s for each of the TCP data movers described above for a constant number of targets (5), as the size increase, the rate tends to stabilize, however, the difference between the rates obtained by the different methods is noticeable, mostly compared to the Sequential approach; as for the simultaneous and threaded data movers the rates are more similar, but the thread approach has some performance gain.

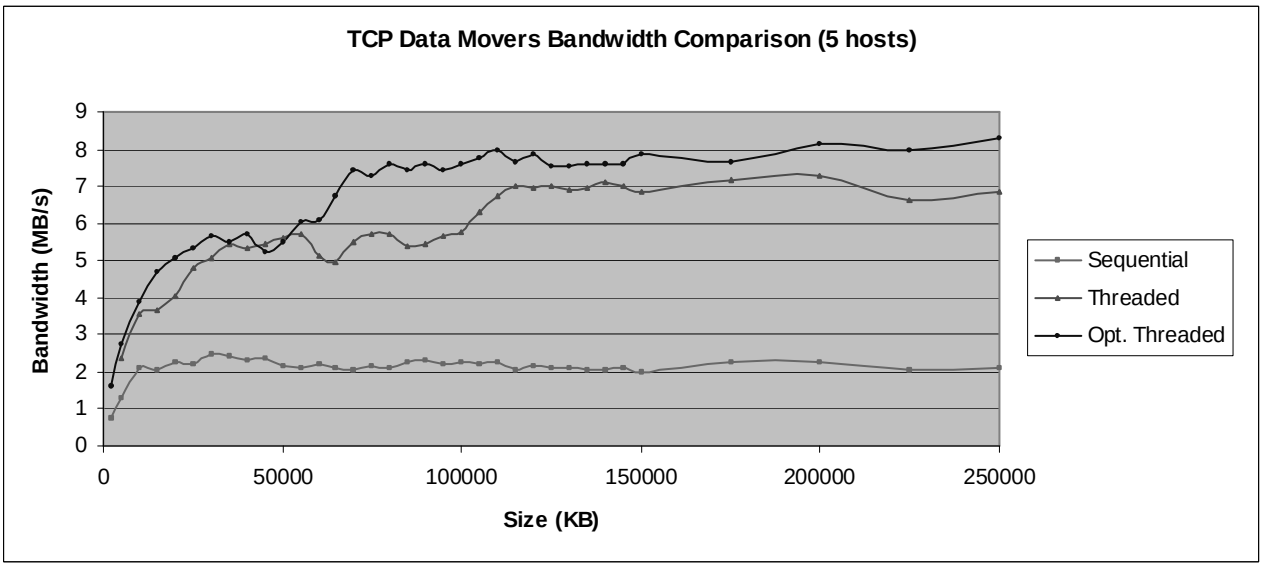


Figure 4: general bandwidth comparison between the three TCP data movers addressed

### 4.2 Point to Multipoint TCP

This subsection concentrates in comparing the transfer of different amounts of data using multiple simultaneous point to point TCP data transfers implemented at user level using threads, and using a single point to multipoint implementation of the Data Mover to transfer the same amounts of data. As figure 2 reveals the latter approach shows an improvement in the overall transfer time. This experiment consisted of transferring several pieces of data of different sizes from one to various numbers of nodes.

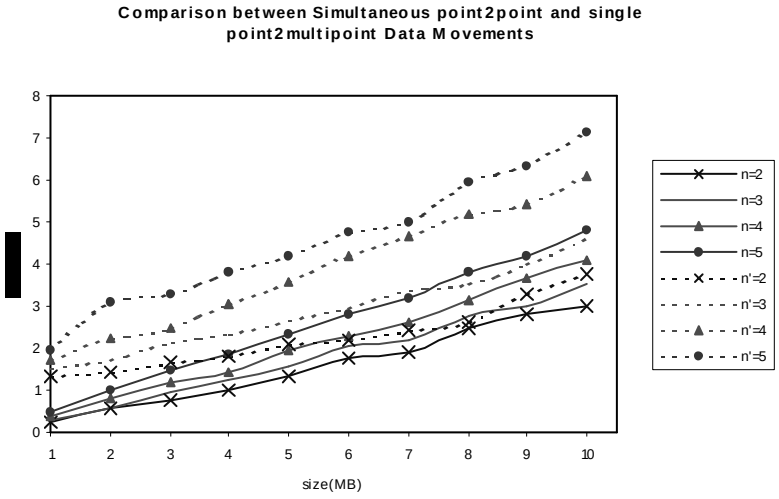


Fig. 2. Dotted lines represent the multiple simultaneous point to point TCP transfers the number of hosts used in different tests is given by n. The continuous lines represent point to multipoint approach, and number of hosts for this cases is given by n.

### 4.3 Point-to-Point UDP versus TCP

The experiment described in this subsection consists of a comparison between the transfer times using TCP and UDPBlaster point-to-point Data Movers. Figure 3 shows the improvement achieved by using UDPBlaster. The Data Mover plug-in could support a variety of protocols and methods; however, for the purpose of this experiment we concentrate on the comparison of TCP versus UDP, to show how the improvement can be achieved within the same data mover using different protocol depending on the characteristics of the backbone being used. It is important to note that since this protocol is still in the experimental phase may behave differently in diverse test environments under different circumstances.

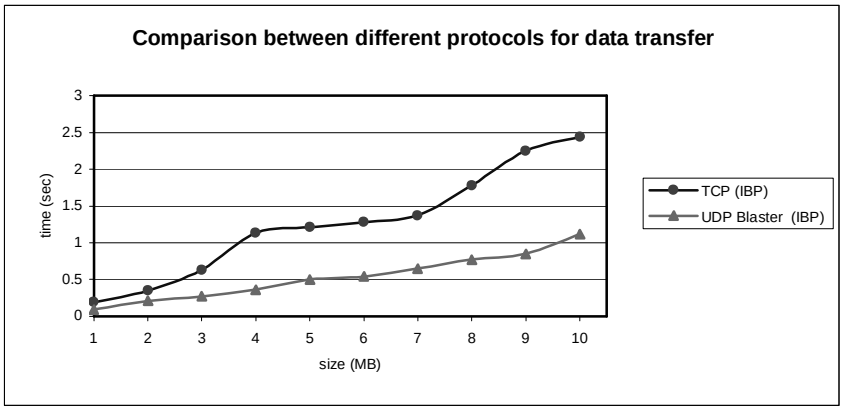


**Fig. 3**. Improvement seen by using UDPBlaster

## 5. Conclusion

The design goals of the Data Mover plug-in and the function **IBP_mcopy()** are:

- To provide a flexible point-to-multipoint transfer tool that supports different protocols and transfer methods
- To allow access to a variety of operations depending on the nature of the underlying backbone and of traffic concerns
- To permit a simple implementation that avoids trust and performance concerns using an *encapsulated* approach, even if software architectures based on external data movement operations are still of great interest.

At all times t he user has the capacity to determine the appropriateness of the operation to the depot's network environment, and to select what he believes is the best data transfer strategy.

## 6. References

[1] Plank, J., Beck, M., Elwasif, W., Moore, T., Swany, M., Wolski, R.; *"The Internet Backplane Protocol: Storage in the Network"*; in NetStore99: The Network Storage Symposium, (Seattle, WA, 1999)

[2] Beck, M., Arnold, D., Bassi, A., Berman, F., Casanova, H., Dongarra, J., Moore, T., Obertelli, G., Plank, J. Swany, M., Vadhiyar, S., and Wolski, R.; *"Logistical Computing and*

*Internetworking: Middleware for the Use of Storage in Communication*"; in the 3rd International Workshop on Active Middleware Services, San Francisco, August, 2001.

[3] Beck, M., Fuentes, E.; *"A UDP-Based Protocol for Fast File Transfer"*, Department of Computer Science, University of Tennessee, Knoxville, TN, CS Technical Report, ut-cs-01-456, June, 2001, http://www.cs.utk.edu/~library/2001.html.