International Journal of Computer Science and Applications

Vol. 9 No. 3, pp. 77 - 91, 2012

DCFMS: Data Handling Distributed System for Achieving High Data Availability

Cosmin Marian Poteras

Software Engineering Department, University of Craiova, Bvd. Decebal 107
Craiova, 200440, Romania
cpoteras@software.ucv.ro

Mihai Mocanu

Software Engineering Department, University of Craiova, Bvd. Decebal 107
Craiova, 200440, Romania
mmocanu@software.ucv.ro

Cristian Marian Mihaescu

Software Engineering Department, University of Craiova, Bvd. Decebal 107
Craiova, 200440, Romania
mihaescu@software.ucv.ro

High data availability is crucial in every high performance environment. Even though network environments have improved a lot, costly data transfers remain the main drawback in achieving high performance in complex distributed applications which need to come up with additional design techniques to overcome this drawback. Continuing our previous research which focused on the design and implementation of a computational steering enabled distributed framework, we have developed a data flow management system that is able to isolate the data handling from the computational environment. The system does not focus on defining new transfer techniques for the physical environment, instead the innovation comes from two main techniques that avoid costly transfers: logical partitioning and data localization. The experimental evaluation of the system is also included.

Keywords: distributed data; data partitioning; distributed file system, high data availability

1. Introduction

Real time visualization and computational steering are key elements when running a category of applications known as distributed (discrete event) simulation [1] [2]. Generally, simulation refers to the numerical evaluation of a model. It is well known that running simulations on distributed high performance environments might become embarrassingly slow if the analyze phase is performed as a post-processing phase. A simulation has to be exhaustively executed for all input data sets and

data can only be analyzed as a post-simulation phase, even if in some cases the process may reveal useless results from the beginning. Running complex applications in todays world is more and more a matter of integration of efficient infrastructures and good computational techniques. Cluster and grid simulation applications that employ parallel computing techniques (i.e. MPI, OPENMP) [1, 2] to simulate real processes are just a common example. Therefore, we focused our recent research towards the need to design a high performance distributed simulation framework [3] [4] [5] whose main goal is to optimize scientific simulations. Our framework uses the concept of state-machines for representing general purpose parallel processing tasks and allows the researcher to visualize, analyze and steer the ongoing simulation avoiding irrelevant areas of the simulation process. The development of distributed simulation and steering frameworks, able to support run-time adjustments and live visualization, has not been an easy task. Extensive surveys of research in this area were carried out in over the last two decades [10] [11], however not many of the projects led to practical tools. Some of the most relevant frameworks for distributed simulation and computational steering, for the scope of this paper, may be considered: COVS[1], RealityGrid, CUMULVS and CSE.

The main performance bottleneck that we dealt with while developing the our computational steering distributed framework, was the data handling itself (acquiring data very fast, dealing with multiple data sources, controlling the network availability, a.o.). Another important aspect that we've noticed was that in many situations it was more efficient to migrate the processing task (state machine) to the host that actually holds the input data than acquiring the data throughout the network. For that we needed a way to query all nodes and find out where the data resides before migrating. It comes naturally that the data flow is a crucial factor for achieving the desired performance. If data flow would be entirely handled at the application level, the entire development process would be significantly slowed down, the application's maintenance would be less flexible, and there would be important doubts on the data transfers efficiency. Obviously this is not a desirable solution for handling data flow in a distributed environment. Instead one could separate the data flow handling into a standalone module whose main role is to acquire, store and provide the data required by the application's processes in the most efficient way.

In this paper we describe the Distributed Chunks Flow Management System (DCFMS) that enables and supports data steering of distributed simulation applications.The system acts like a file system while it adds two new innovative features: logical partitioning and data awareness. Logical partitioning allows the application to define how the files shall be splitted into chunks. This is very important for avoiding unnecessary transfers of the entire file while only a part of it is needed, instead the transfers fit exactly the application's needs. The data awareness allow the application to query information related to data location. Most of the times in distributed environments it is desirable to migrate processes towards data than

the other way around. This feature allow the application to decide whether to send data towards processes or processes towards data.

The rest of the paper is organized as follows. Section 2 discusses briefly some related work. Section 3 introduces the new model of DCFMS, as a distributed file system. Section 4 introduces the main methods behind DCFMS and gives some implementation details for partitioning classes. Preliminary performance results are overviewed in Section 5. Section 6 concludes the paper and outlines the future trends of development.

2. Related Work

In this section we will mention two of the most popular and widely used distributed data transfer systems which have similar components with the ones introduced in this paper: Apache Hadoop - HDFS and BitTorrent Protocol.

BitTorrent Protocol [6] is a file-sharing protocol designed by Bram Cohen used in distributed environments for transferring large amounts of data. The idea behind BitTorrent is to establish peer-to-peer data transfer connections between a group of hosts, allowing them to download and upload data inside the group simultaneously. The torrents systems that implement BitTorrent protocol use a central tracker that is able to provide information about peers holding the data of interest. Once this data reaches the client application, it tries to connect to all peers and retrieve the data of interest. However, it is up to the client to establish the upload and download priorities. Torrents systems might be a good choice for distributed environments, especially for those based on slower networks. However, the main disadvantages of torrent systems are related to the centralized nature of the torrents tracker as well as leaving the entire transfer algorithms and priorities up to the client application which might cause important delays if the transfers trading algorithm chooses to serve a peer that might have a lower priority at the application level. The reliability of the entire system is concentrated around the tracker; if the tracker goes down, the entire system becomes not functional. Torrents are mainly systems that transfer files in distributed environments in raw format without any logical partitioning of the data. Such logical partitioning might often prove to be very important. For example if an imaging application needs a certain rectangle of an image it would have to download the entire file and then extract the rectangle by itself instead of just downloading the rectangular area and avoid transferring unnecessary parts of the file. As a remarkable advantage of torrents systems we could mention self-sustainability [7] due to peer independence and redundancy.

Hadoop Distributed Files System (HDFS) has been designed as part of Apache Hadoop [8] distributed systems framework. Hadoop has been built on top of the Google's Map-Reduce architecture and HDFS. HDFS proved to be scalable, and portable. It uses a TCP/IP layer for internal communication and RPC for client requests. The HDFS has been designed to handle very large files that are sent across hosts in chunks. Data nodes can cooperate with each other in order to provide

data balancing and replication. The file system depends closely on a central node, the name node whose main task is to manage information related to directory namespace. HDFS offers a very important feature for computational load balancing, namely it can provide data location information allowing the application to migrate the processing tasks towards data, than transferring data towards processing task over the network [9]. The main disadvantage of HDFS seems to be the centralized architecture built around the name node. Failure of the name node implies failure of the entire system. Though, there are available replication and recovering techniques of the name node, this might cause unacceptable delays in a high performance application.

3. Conceptual Model

In this section the conceptual model of our distributed file system will be introduced. The system is simple, based on a client-server architecture. The entire model has been built around the key element, data chunk. The data chunk usually represents a file partition but none the less it can be any data object required by the application's processes. Besides the data piece itself, a data chunk also contains meta-information describing the data piece, like: size, location inside source file, the data type, timestamp of latest update or the class that handles chunks of its type. The system is supposed to offer the following features:

- Cost. Better price/performance as long as commodity hardware is used for the component computers.
- Portability. A cross-platform system design that does not require special system priviledges for running
- Extensibility. Easy to do, because of its modular design
- Scalability. Hosts can be added at run-time, and storage capacity can be increased incrementally
- Run-time storage updates, abstract communication API
- Customizable data handling for all data types, etc.

Figure 1 illustrates the systems model. The most important contribution of DCFMS is that it handles chunks of different types in an abstract mode without actually knowing what is inside the chunk, leaving the data partitioning up to the application level. This is very important from the application's perspective as it can define the way files are partitioned into chunks and how they can be put together again to reconstruct the initial file allowing the application to map data chunks to processing tasks in the most appropriate way for efficient processing. No restrictions are imposed by the DCFMS on data partitioning.

The entire model has been built around one key element, the data chunk. It usually represents a file partition but it can also be any data object required by the applications processes. Besides the data piece itself, a data chunk also contains meta-information describing the data piece, like: size, location inside source file, the

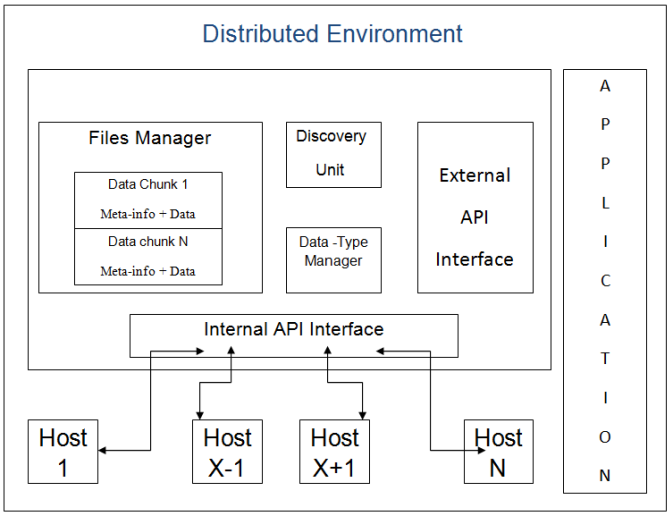

Fig. 1. DCFMS design model

data type, timestamp of latest update or the class that handles chunks of its type. Thus, the most important contribution of DCFMS is the way it handles chunks of different types in an abstract mode without actually knowing what is inside the chunk, leaving the data partitioning up to the application level. No restrictions are imposed by the DCFMS on data partitioning.

The bridge between the abstract representation of data chunks and their actual type is the Type Manager. It is able to make use of external classes (defined at the application level) where all the file type specific functionality can reside. The classes are dynamically loaded whenever the application layer needs partitioning, files reconstruction as well as information related to the collection of chunks (i.e. the number of chunks). It is the applications' developer task to implement the data chunks handler classes. The DCFMS only provides a set of interfaces that help to implement the partitioning logic. For example, one might need to handle two types of files in their distributed application: image files and text files. In case of the image files a data chunk might be represented by a rectangular region of the initial image. Multiple such chunks can cover the entire image. An image can be split into rectangular chunks by dynamically invoking the image partitioning method. In case a node needs an entire file that is spread all across the system, DCFMS can acquire all its chunks from different hosts and recompose the image by dynamically invoking the image reconstruction method. In case of a text file, the chunks can take the form of paragraphs, or pages, or simply and array of characters of a certain size. In a similar way the files can be dynamically partitioned and reconstructed. Later in this paper we will discuss the development effort involved in writing such classes. The proposed DCFMS is able to scale up dynamically at run time without using a

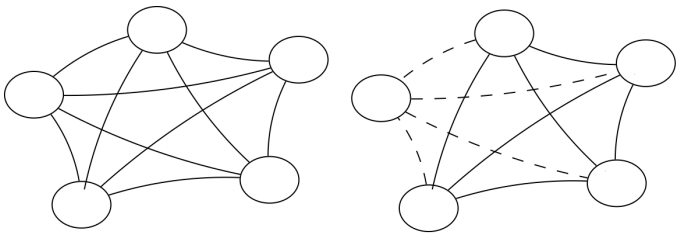

Fig. 2. Synchronization with no event restransmission

central node. This functionality is achieved by the Discovery Unit which broadcasts and listens to discovery messages. There are two API interfaces that allow DCFMS nodes to communicate with each other and also with the client application.

4. Methods and Algorithms

4.0.1. *Data synchronization*

A key feature in any distributed system that handles large amount of data is keeping data synchronized. Spreading data around the network while keeping it up to date uses events. Each node that has updated a data chunk must broadcast to all other nodes that he is aware of about the changes, and event handlers update the timestamp of the affected data. Depending on the nodes connectivity there are two choices:

- No event retransmission the ideal situation when the network bandwidth allows 1 to 1 connections between any two nodes in the system; it is enough to broadcast an update event once to all other nodes in the system. In the left side of figure 2 we have considered a network of 5 nodes. At some moment in time, one of the hosts has to boradcast an event. The event transmission is handled in only one step as can be seen on the right side of the figure.
- Event retransmission when there is at least one node not interconnected with all other nodes in the system. To make sure that node is always notified about update events, retransmission is necessary; to stop infinite loop of update events nodes employ timestamps (whenever an update event time stamped in the past it will be ignored.) In figure 3 a) it is considered a network of 5 hosts. One of the hosts needs to notify the others about an event. At each step b), c), d) every node sends the event to its neighbours.

For a better understanding of the data flow algorithm, we will analyze a concrete scenario. Lets assume DCFMS consists of nodes N0, N1, ..., Nn, and let node N0 be interested in aquiring data chunks C1, C2, ..., Cm. N0 will broadcast a request for C1, ..., Cm to the entire DCFMS. Nodes N1, ..., Nn reply back to N0 with a subset of C1, ..., Cm that they host. As soon as replys arrive, N0 builds a chunks

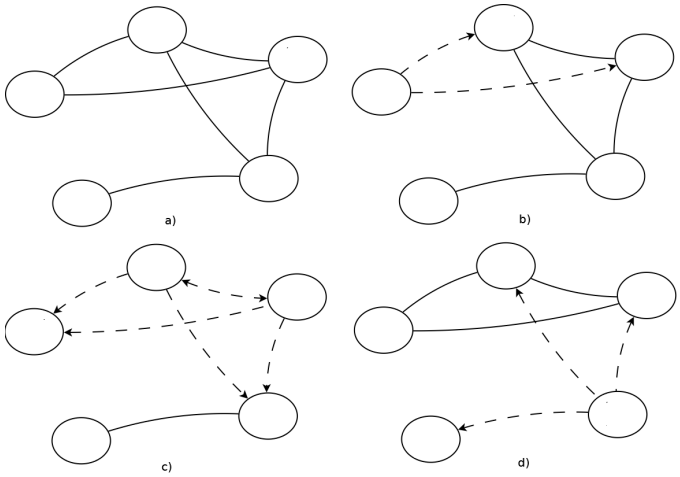

Fig. 3. Synchronization with event retransmission

availability matrix having as rows the nodes N1, ..., Nn and as columns chunks C1, ..., Cm. (Ni,Cj) gets valued 1 if the chunk Cj is available on host Ni, otherwise it gets valued 0. N0's main goal is to establish as many connections as possible, but not more than one connection per serving host (at most n-1 connections at a time). Chunks availability responses are performed in an asynchronous manner so that N0 won't have to wait for all responses before proceeding with transfers. Instead it will establish connections as the responses arrive, overlapping chunks transfer with availability requests. Whenever a chunk transfer completes, the External API will be informed about it and the client application can start processing the newly acquired data. As chunks might spread across DCFMS while N0 transfers it's chunks, the availability matrix will be constantly updated by sending new availability requests whenever a chunk transfer completes and N0 has established less than n-1 connections (free download slots available).

4.1. *Data Partitioning*

The user is able to retrieve exactly the data of interest causing an important reduction of the amount of data that travels through the network. A data chunk is basically any logic unit of data extracted from a data set (usually a file) according to a certain algorithm that reflects the applications needs. The data extraction is based on the most simple principle: request answer. The application places queries against DCFMS, queries are broadcasted in the entire system, each node invokes the appropriate chunker (the one associated with the requests type), the chunker extracts the logical piece of data according to its internal algorithm (custom algorithm designed to serve the application environments needs), and ultimately it

replies back with the data chunk.

4.1.1. *Support for load balancing*

In distributed applications it often happens that the processing of a data chunk requires less time than the transfer of the data itself. For this reason it might be a good practice to migrate the processing task towards the data than transferring data to the processing host. The DCFMS is able to provide through its external API locating information about the data it holds (data aware system). It is the application's task to migrate the processing tasks throughout the nodes in order to reduce or eliminate the data transfer time.

4.1.2. *Application developer's task: Implementing Chunker classes*

Chunker classes define how files or data objects are split into data chunks. A chunker class is nothing else than a class that implements a Chunker interface defining the following methods:

- GetChunk(chunkId)
- IsChunkAvailable(chunkId)
- ReconstructFile(filename)

Chunker classes are dynamically invoked at run time every time chunks or their associated meta-data are being requested. Data chunks are mapped to chunker classes by their meta-data.

4.2. *Integration with other systems*

DCFMS processes are spread accross the distributed environment. Considering the distributed design of DCFMS it is recommended to be run a dedicated instance of DCFMS for each instance of the client processing system. Figure 4 describes the way DCFMS connects to processing systems and provides access to the data it holds. The bridge between the two systems is the DCFMS external API. Through the external API, the processes can perform requests, retrieve location information, download or upload data.

5. Experimental results

We conducted a set of experiments for the preliminary performance evaluation of our DCFMS, as a standalone system, out of the scope of the framework in which it will be finally integrated. To evaluate DCFMS we've made use of two environments:

- A high performance Myrinet network of 4 Gbps bandwidth consisting of 8 identical hosts with Intel Core 2 Duo E5200 processors, 1GB of memory and the hard drive benchmarked at an average read speed of 57MB/s and write speed of 45MB/s.

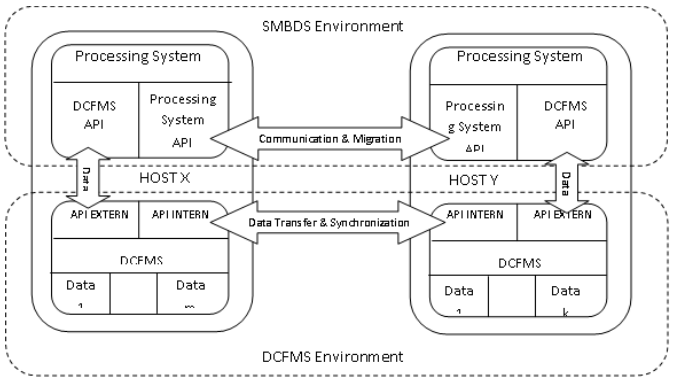

Fig. 4. Integrating DCFMS with third party distributed applications

- A regular Ethernet network of 100Mbps bandwidth consisting of 8 identical hosts 4 hosts with Intel Core 2 Quad processors and 4 GB of memory.

The purpose of the experiments was to determine how fast will perform the DCFMS in one-to-many and many-to-one scenarios. We've picked those two scenarios since they represent the worst and the best traffic demanding scenarios. In one-to-many scenario a certain file was hosted by one node and had to be transferred to all the other nodes starting simultaneous. The reverse work had to be performed in a many-to-one scenario, namely all hosts except one hosted the file, and they had to serve collectively the file to the client host. Each scenario has been run for different chunk sizes. The test cases presented in this paper focus on the transfer speed of DCFMS rather than on the computational performance of a system based on it. Around 50 runs were performed for each case and the results were statistically processed, avoiding singularities. We appreciate that the results can be significantly improved by our DCFMS running in its real design environment instead of a testbed, by using the data awareness capabilities discussed in the previous sections.

5.1. *Myrinet Network results:*

Test case 1: One-to-Many one sender host and 7 receivers that request the same file of 180.6MB simultaneously. Results are presented in Table 1 and Figure 5

Test case 2: Many-to-One 7 senders that will serve one host that requests the same file of 180.6MB from all senders. Results are presented in Table 2 and Figure 6

By examining the results obtained at test cases 1 and 2 we can conclude that the chunk size has an important impact on the file system's performance. The One-to-Many scenario shows good performance on a chunk size between 512KB and 5MB, while the Many-to-One scenario shows best performance on a chunk size

Table 1. Myrinet One-to-Many results

Chunk Size	Min. time 7 hosts (s)	Max. time 7 hosts (s)	Avg. Time 7 hosts (s)
256KB	14.197	16.973	15.730
512KB	9.072	10.291	9.704
1MB	9.110	10.534	9.929
2MB	8.800	9.802	9.404
5MB	9.779	11.926	10.960
10MB	9.221	12.253	10.99
15 MB	9.166	11.529	10.353
20MB	12.444	18.583	15.640
25MB	13.763	18.407	17.066

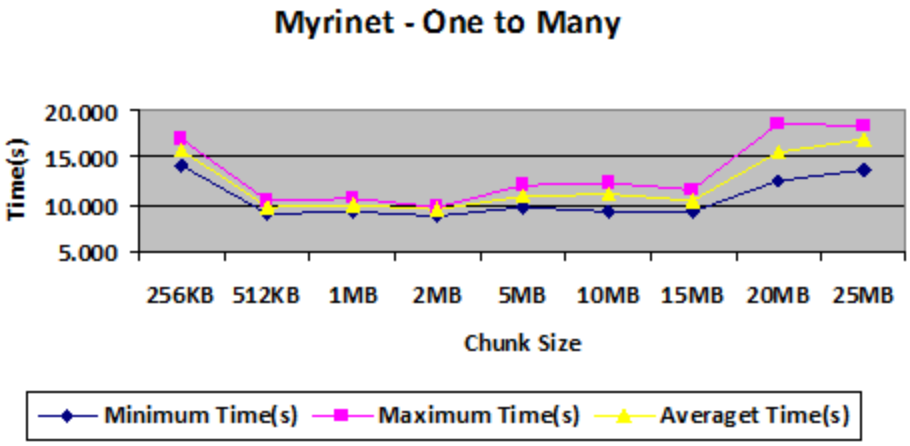

Fig. 5. Myrinet One-to-Many results

between 2MB and 10MB. Therefore, for best timings, the Myrinet-based application developer has to choose a chunk size between 2MB and 5MB.

5.2. *Ethernet Network results:*

As the Ethernet speed is far less than the Myrinet network, we decided to use a small file, namely a 9.8MB file.

Table 2. Myrinet Many-to-One results

Chunk Size	Time (s)
256KB	11.362
512KB	5.693
1MB	5.466
2MB	4.168
5MB	3.692
10MB	4.111
15 MB	4.216
20MB	5.856
25MB	8.378

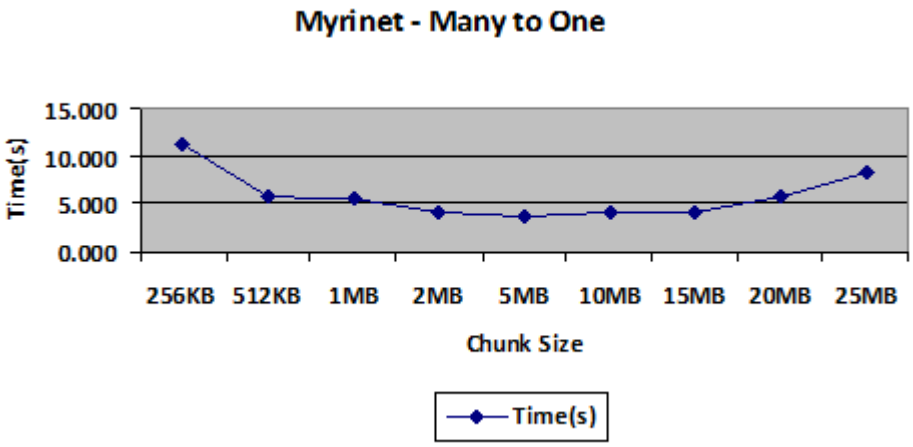

Fig. 6. Myrinet Many-to-One results

Test case 3: One-to-Many one sender host and 7 receivers that request the same file of 9.8MB simultaneously. Results are presented in Table 3 and Figure 7

Test case 4: Many-to-One 7 senders that will serve one host that requests the same file of 9.8MB from all senders. Results are presented in Table 4 and Figure 8

Unlike the Myrinet network, in case of the Ethernet the chunk size doesn't have a big impact on the performance. However the Ethernet network proved not to be the appropriate environment for high performance distributed applications that require high data availability.

Table 3. Ethernet One-to-Many results

Chunk Size	128KB	256KB	512KB	1MB	2MB
Minimum time of the 7 hosts (s)	77.241	80.833	64.011	70.572	57.159
Maximum time of the 7 hosts (s)	83.553	85.908	80.121	82.930	75.547
Average Time of the 7 hosts (s)	80.344	83.550	72.994	77.346	67.021

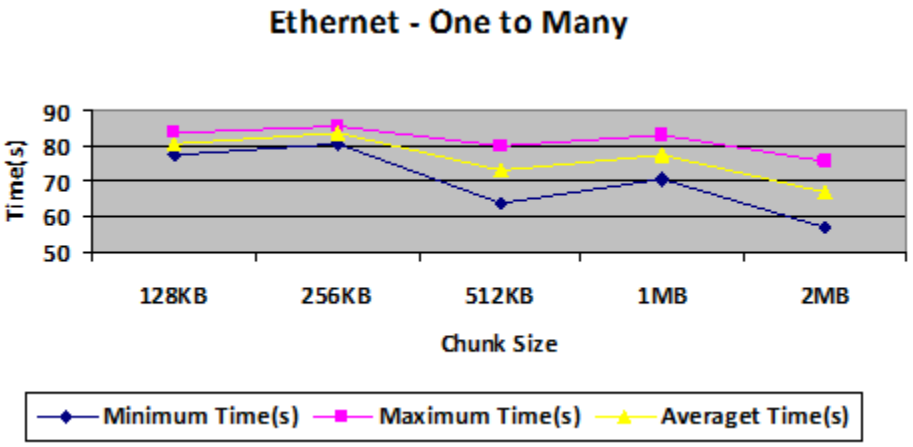

Fig. 7. Ethernet One-to-Many results

Table 4. Ethernet Many-to-One results

Chunk Size	128KB	256KB	512KB	1MB	2MB
TIme (s)	29.697	29.048	28.940	28.726	28.752

6. Conclusions and Future Work

We presented in this paper the main design ideas, together with the preliminary performance evaluation results, for a Distributed Chunk-based Flow Management System which is part of a more complex distributed simulation framework under development. Important contributions of the system relate to: custom logical partitioning defined at the application level (abstractly handling) and load balancing support due to the data awareness (data location information) feature while main-

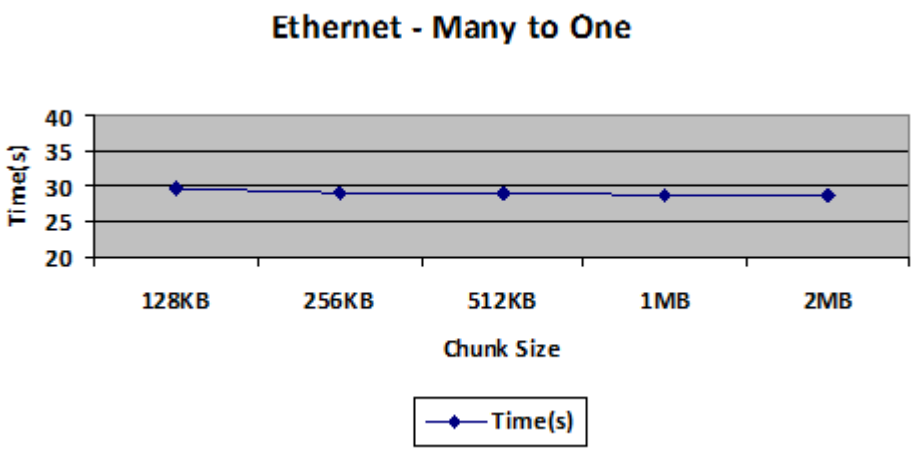

Fig. 8. Ethernet Many-to-One results

taining a high data availability. The system shows good performance in very high speed networks (Myrinet), but it can also be a good choice in Ethernet networks for applications not requiring transfers of high data volumes across the network. It seems that, compared to other existing similar systems, DCFMS is the system requiring the least requirements. Actually the only basic requirement is the permission to create sockets. Configuration needs are very low, and the system can be very easily extended by just plugging in new chunker classes even at run-time. Also, the system can be scaled up as much as necessary at run-time. To conclude, the dynamic discovery feature of the system ensures the scalability of the system, the decentralized architecture improves the reliability, while the dynamic data handling offered by the run-time loaded chunker classes make the system flexible and easier to extend. We have many ideas to adjust the design of our system, for a future version, as well as some hints to improve the performances of the existing system, after a complete evaluation in the distributed simulation framework. Actually, chunk transport is done only when all the chunk related information has been gathered. The solution that allows continuous data transfer over a network may be very different, although its design starts from the same assumption: for certain network configurations and physical performances, there is always an optimal dimension for chunks able to reach their destination in the shortest possible time. This may require the differentiation of logical chunks (used for the same purpose as data partitions for external applications) from physical chunks (smaller, and sized for optimal data transport). This approach must rely on as many as possible simultaneous operations done by the logical chunker; on the server side, there may be fetching of data from local hard-disks, preparing and transmission of physical chunks, while on the client side there are fast gathering of information over the network or the use of

caching techniques according to system needs. Other immediate future directions in the development of the system may also be the hosts speed ranking which could be very significant when deciding the source hosts, or the network traffic monitoring which could help deciding the route that should be followed for a faster download. Being designed as part of a distributed simulation framework [3], as mentioned in the Introduction, DCFMS shall be able to provide support for computational steering. Besides steering the simulation processes the researcher shall be able to also steer data storage, or alter data held by the DCFMS while simulation is running. Apart from other existing systems, like Dryad [12], designed as a general-purpose distributed execution engine for coarse-grain data-parallel applications, our system will try to look for speed at all levels, and use also the fine-grain data parallelism. Some similarities will be maintained, such as those related to an application ability to discover the size and placement of data at run time, scalability, extensibility and ability to reconfigure the computation graph as the computation progresses, to make efficient use of the available resources.

References

[1] R.J. Allan and M. Ashworth. A survey of distributed computing, computational grid, meta-computing and network information tools. Daresbury, Warrington WA4 4AD, UK,2001, pp. 38-42

[2] Esnard, A. Richart, N. , Coulaud, O. - A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations,. Proceedings of DS-RT'06 - 10th IEEE International Symposium on Distributed Simulation and Real-Time Applications, 2006, pp.7-14

[3] Cosmin Poteras, Mihai Mocanu Grid-Enabled Distributed Simulation - A State Machine Based Approach, Proceedings of TELFOR 2010, Belgrade, Serbia, pp. 1323-1326

[4] C. Poteras, M. Mocanu, C. Petrisor - A Distributed Design for Computational Steering with High Availability of Data, International Journal of Systems Engineering, Applications and Development, ISSN: 2074 1308, Issue 1, Volume 6, 2012

[5] M. Mocanu, C. Poteras - Improving Parallel Data Flow Support in a Visualization and Steering Environment, Proceedings of the 2nd International conference on Applied Informatics and Computing Theory (AICT '11) , Recent Researches in Applied Informatics, pp. 226-231, Sept 2011

[6] Bram Cohen- The BitTorrent Protocol Specification

[7] D. Menasche, A. Rocha, E. de Souza e Silva, R. M. Leao, D. Towsley, A. Venkataramani - Estimating Self-Sustainability in Peer-to-Peer Swarming Systems, Journal of Performance Evaluation Volume 67 Issue 11, November, 2010.

[8] http://hadoop.apache.org/

[9] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin - Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters, IPDPSW 2010, Atlanta, pp. 1-9

[10] W. Gu, J. Vetter and K. Schwann. An annotated Bibliography of Interactive Program Steering, SIGPLAN Notices 29 (1994), pp. 140-148 and Technical Report GIT-CC-94-15 (Georgia Institute of Technology)

[11] R.J. Allan and M. Ashworth. A Survey of Distributed Computing, Computational Grid, Meta-computing and Network Information Tools, available from http://www.ukhec.ac.uk/publications/reports/survey.pdf

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, EuroSys - European Conference on Computer Systems, Lisbon, Portugal, March 21-23, 2007, pp. 59-72