A Distributed Design for Computational Steering with High Availability of Data

Cosmin Marian Poteras, Mihai Mocanu and Constantin Petrişor

***Abstract*—** This paper introduces a new approach, based on state machines, for distributed frameworks, that is able to support both distributed simulation and computational steering. The framework makes use of a Distributed Chunk-based Flow Management System (DCFMS) having as main benefits the logical partitioning and data localization information. The architectures and implementation details of the two systems as well as integrative experimental results are briefly discussed.

***Keywords*—** distributed framework, chunk based data flow, state machines

I. Introduction

RUNNING complex applications in today's 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. Modeling and simulation have become key phases for a wide spectrum of applications in modern research. In contrast with the study of a real system, whose advantage is the accuracy of the evaluation, but who might become destructive, dangerous, and expensive, the study of a model is easier, safer and cheaper. Modeling, as a general term, denotes the process that offers an abstract representation of a system, which allows, in turn, through its study, the formulation of valid conclusions on the real system. Simulation generally refers to the numerical evaluation of a model. When dealing with a complex system whose analytical solution is hard to be determined, simulating the system's behavior on a model could be the only solution left. The outcome of a simulation may be analyzed in a separate post-processing step (for instance by viewing the results in a separate visualization application), and, based on intermediate results, a decision can be made to change simulation parameters for another computational period. In order to increase the efficiency, new techniques for live visualization and steering have emerged allowing simulation and visualization to be performed simultaneously. If online visualization refers to the ability to immediately observe the processing steps during the simulation, this in turn allows for computational steering to influence the computation of the simulation during runtime on a cluster, grid and even on a supercomputer. They are meant together to dynamically steer the parameters of a parallel simulation, increasing not only the interactivity but also the efficiency of the overall process. Visualization is the process of exploring, transforming and rendering of data through images, with the goal to offer a thorough and deep understanding of data. It is a complex field of study in our days, including elements of computer graphics, digital image processing, computational geometry, numerical analysis, statistics, and studies on human perception. Computational steering is a process of manual intervention on an autonomous computational system, with the goal to analyze and modify outputs. It is a very common technique in numerical evaluation, used to guide a computational process towards regions of interest. Apart from this pure applicative perspective, computational steering can be examined from a broader technical perspective; for instance, we may consider the modification of memory amount available for a process, with the goal to observe and influence the effects over the execution time. This paper deals with the concept especially in the latter, broader sense. The taxonomy of the concept also includes: *program steering*, which has been defined as the capability to control the execution of resource-intensive, long running programs (this may imply modifications of program state, starting and stalling program execution, etc.), *data steering* (which implies the management of data output, alteration of resource allocations etc.), and *dynamic steering* (which requires the user to monitor program or system state and have the ability to make changes, through "add-ons" routine calls or data structures interaction in the code). Interactive simulation combined with visualization has undergone a major development and it is now widely used. Up to the late 80s, simulation had been considered a tedious and time consuming process, mainly due to the lack of interactivity with the ongoing simulation process: the researcher had to exhaustively execute the simulation for all input data sets and he could only analyze data as a post-simulation phase, even if in some cases the simulation process reveals useless results from the beginning. When the need of interactivity became obvious, research also concentrated on developing simulation frameworks with visualization and steering capabilities, so that an ongoing simulation could be immediately observed and guided. The development of distributed simulation and

Dr. Mihai Mocanu, Cosmin Poteraş and Constantin Petrişor are with the Department of Software Engineering at the University of Craiova, Romania. Contact e-mail: {mmocanu, cpoteras}@software.ucv.ro

Additional financial support for this work has been provided to Cosmin Poteras by AMPOSDRU grant 109/25.09.2008

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 [3, 4], 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. COVS[7] (Collaborative Onlline Visualization and Communication) is a framework that encapsulates common visualization frameworks (VTK, AVS/Express), steering technologies (VISIT, gViz, ICENI) as well as communication libraries (VISIT, PV3) that carry out the data transportation and steering commands. This multi-framework integration allows COVS to run simulations independently from visualization and communication tasks. RealityGrid [8, 9] is an API library consisting mainly from two modules. The former is responsible for offering steering capabilities and the latter provides tools for dedicated client applications. RealityGrid uses check-pointing techniques for supporting steering commands. CUMULVS (Collaborative User Migration, User Library for Visualization and Steering) [10, 11] has been developed at Oak Ridge National Laboratory and has been designed for the development of collaborative on-line and interactive simulation and visualization. The power of this platform consists in the advanced recovery techniques, the tasks migration support and check-pointing. CSE (Computational Steering Environment) [12, 13] has been developed at the Center for Mathematics and Computer Science, in Amsterdam. It uses a centralized architecture around a replicated Data Manager that is able to carry out steering commands and coordinate the simulation tasks. The Data Manager from CSE leads us to an important problem in the analysis of these efforts: *data availability*. The computations may be dramatically slowed down by the acquiring of data. *Dataflow* processing is at the same time the most appropriate model of *programming* and a crucial factor for achieving the desired *performance*. Existing systems like BitTorrent and Apache Hadoop Distributed File System implement a *parallel dataflow* style of *programming* which provide the data required by a distributed application's processes in the most efficient way. The BitTorrent Protocol [14] establishes 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 centralized nature of the tracker concentrates the reliability 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 The Hadoop Distributed File System has been designed as part of Apache Hadoop [15] distributed systems framework. Hadoop has been built upon the Google's Map-Reduce architecture as well as HDFS file system. 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 [4]. The main drawback of HDFS seems to be the centralized architecture built around the *name* node. Failure of the *name* node implies failure of the entire system. There are still available techniques for replication and recovering of the *name* node, but this might cause unacceptable delays in a high performance application. Due to the well known diversity and complexity of distributed models, choosing the appropriate design for a system like ours is not an easy task. Besides of the usual requirements imposed to a distributed system, like scalability, flexibility, extensibility, portability, we added support for load balancing and tasks migration, and safety features. For this we concluded to a design that merges together parallel mapping of tasks in the form of state machines, able to be deployed in a robust way over a network, and parallel dataflow handling, separated 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. We will describe in this paper only the design of the two modules.

The rest of the paper is organized as follows. Section II describes the design and implementation details of the State Machines Based Distributed System (SMBDS) [5]. Section III introduces the conceptual model for Distributed Chunks Flow Management System (DCFMS) [6] while section IV describes the main implementation details of DCFMS. Section V explains how the two systems can work together and presents integrative experimental results. Section VI concludes the paper and presents our future thoughts.

II. State-Machines Framework Design and Implementation

It is very common that many domains impose very strict requirements for software. For example, medicine requires very high safety standards as well as high performance environments due to the incompatibility of this field with errors and instability. Imagine the dramatic effects of an error occurred in a software application that assists a surgery. That could become fatal. To improve the reliability and safety, one has to make sure that at any moment the software is in a consistent state. A good practice would be to analyze all possible states prior to the system development and by ensuring the system's reaction is appropriate in any state. All these constraints lead us to the idea of representing tasks as finite state machines.

State machines can provide code safety, robustness, traceability, excludes erroneous states and inconsistencies while providing a simple and well structured "package" for representing complex tasks. Being represented as "packages", tasks are encapsulated and can easily migrate in distributed environments. Tasks migration together with live monitoring of the distributed environment reveals new possibilities for defining dynamic load balancing algorithms.

We propose in this paper a new design model for distributed simulation environments whose architecture is illustrated in fig.1. The model has been implemented as a class library that reduces considerably the applications development time. Our model consists of five main modules: Simulation Module, Control and Communication Module, Visualization Module, Shared Memory Module and Client Application.

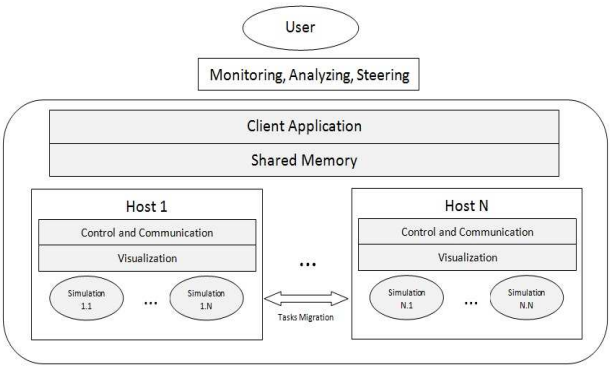

Fig. 1. The structure of the proposed distributed system.

The processing is being performed by the simulation processes. They are represented as state machines, and there can be run as many processes as each host can handle efficiently. The shared memory module can comply either to a distributed form or a centralized one. Its main goal is to store the system's parameters which usually realize the computational steering. The control and communication module handles data flow as well as monitoring and migration jobs. It is responsible for acquiring input data, forwarding output data to the visualization filters, synchronizing access to the shared memory while monitoring the system's resources and loads and realizing machines' migration whenever necessary. The control and communication module is able to rise the computational steering to a new level by allowing the user to manually specify simulation processes migration. There will be only one instance of the control and communication module on each host. The visualization module is responsible for translating simulation's output which usually is in a raw format into a more appropriate format for visualization. The client application initializes, monitors, controls (steers) and analyzes the simulation.

The architecture is based on the theoretical model of a state machine: a state machine is a quintuple $M = (\Sigma, S, s_0, \delta, F)$ where Σ is the set of input parameters (input alphabet, finite, non empty), S is the set of states, s_0 is the initial state, δ is the states transition function $\delta: \Sigma \times S \rightarrow S$ and F is the set of final states. The architecture has ensures the separation between machine code and machine data

The library consists of a set of abstract classes and interfaces that allow the developer to define the machine's algorithm by extending/implementing the proper methods. The library's engine automatically manages the state machines and their migration.

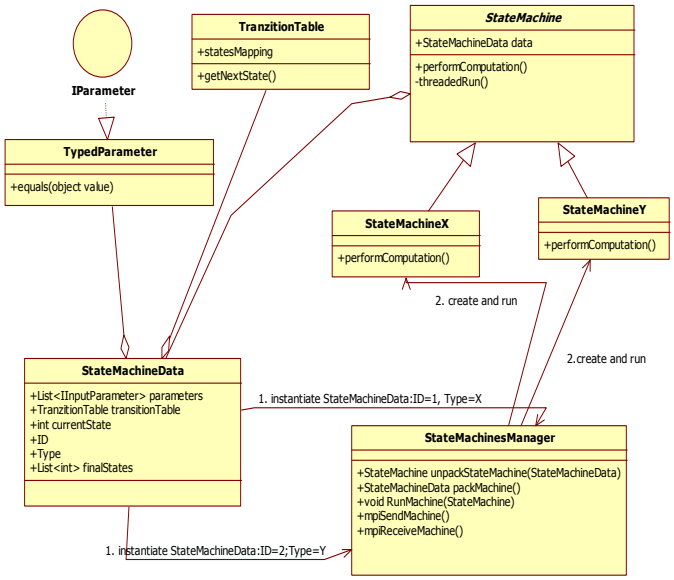

Fig. 2. The Library's architecture and workflow.

The main class of the platform is the StateMachine class. It is an abstract class which serves as base class for every type of state machine required by the application (StateMachineX, StateMachineY). It handles the states succession and computations by employing the performComputation method together with the states transition table. The performComputation method will be overridden by the derived types and it will hold all custom algorithms specific to each

state. StateMachine class starts computations by invoking the method passing as parameters the initial state, performs computations associated with this state and retrieves the output. The states transition table is being checked for the next state and the process continues in the same manner.

Data is being separated from code by using StateMachineData objects. StateMachineData holds all relevant information about the machine: parameters, current state, transition table, machine identifier – unique in the entire environment, the machine type (StateMachineX, StateMachineY), final states, etc. All these can be extended by deriving the StateMachineData class.

The transition table (TransitionTable) its represented as a mapping between pairs <parameters, state> and future state. The transition is performed by method getNextState which retrieves the next state based on the current state and the output values of the parameters from the current state.

For flexibility reasons the parameters have been interfaced by the IParameter interface leaving its implementation up to the developer. IParameter offers getter and setter methods as well as parameters matching methods.

The state machines' management is ensured by the "brain" class, which is StateMachinesManager. Its role is to manage all the machines running on a host. It is able to monitor the system, to ensure data availability, to create, run and migrate machines to and from other hosts. The most important tasks performed by the StateMachinesManager are related to tasks migration and load balancing. These tasks are performed by the following methods: packMachine() – prepares the machine for migration, unpackMachine() – prepares machine for resuming the processing on the new host, mpiSendMachine() – sends the machine to other host, and mpiReceiveMachine() – receives the machine from another host, and RunMachine() – which resumes the processing. Each host in the distributed environment will run one instance of the state machines manager.

Considering the above implementation details we can enumerate the steps needed for implementing distributed simulation applications on top of the framework.

- Defining the parameters of the system (implementing IParameter)
- Defining all types of machines needed. For each type, a new derived class will be created inheriting the class StateMachines. The method performComputation will hold the processing algorithms.
- TransitionTable class will be instantiated and populated with mappings of type <<parameter, current state>, future state>

StateMachinesManager will be instantiated and run on each host.

III. Distributed Chunks Flow Management System. Conceptual Model

The proposed DCFMS, whose model is illustrated in Figure 3, 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.

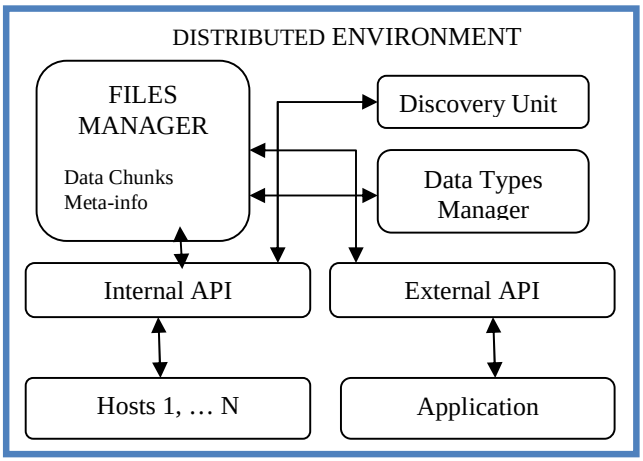

Fig. 3 *The DCFMS design model*

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 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. 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. This is very important from an application perspective, allowing it to map data chunks to processing tasks very efficiently. 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 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.

IV. DCFMS - METHODS AND ALGORITHMS

A. *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.

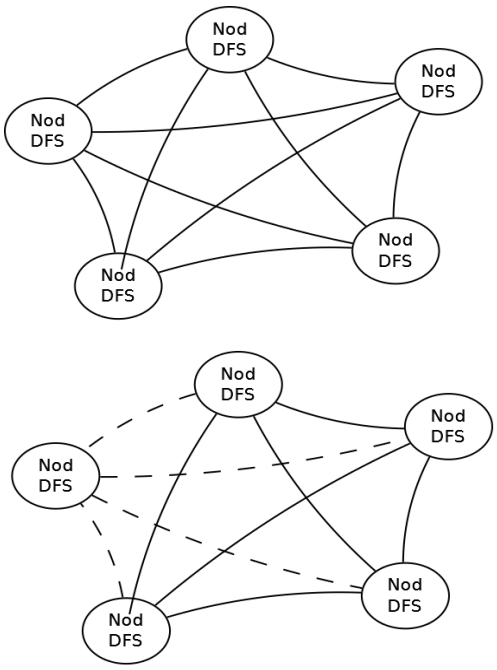

Fig. 4 *Retransmission not needed for event propagation.*

- 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)

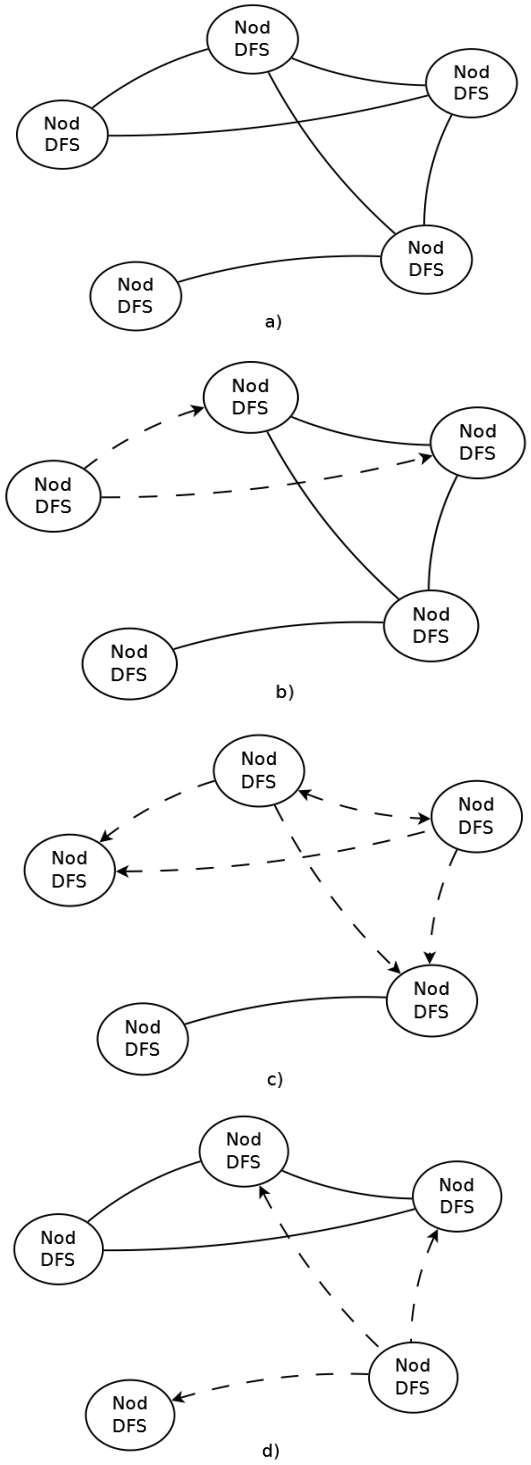

Fig.5.Events retransmission

B. *Data flow*

The flow algorithm is based on availability tables. We will analyze a concrete scenario. Lets assume DCFMS consists of nodes N_0, N_1, … N_n, and let node N_0 be interested in aquiring data chunks C_1, C_2, … C_m. N_0 will broadcast a request for C_1,...C_m to the entire DCFMS. Nodes N_1...N_n reply back to N_0 with a subset of C_1,....C_m that they host. As soon as replys arrive, N_0 builds a chunks availability matrix having as rows the nodes N_1....N_n and as columns chunks C_1...C_m. (N_i,C_j) gets valued 1 if the chunk C_j is available on host N_i, otherwise it gets valued 0. N_0'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 N_0 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 N_0 transfers its chunks, the availability matrix will be constantly updated by sending new availability requests whenever a chunk transfer completes and N_0 has established less than n-1 connections (free download slots available).

C. *Data partitioning*

As previously mentioned, 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 application's 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 request's type), the chunker extracts the logical piece of data according to its internal algorithm (custom algorithm designed to serve the application environment's needs), and ultimately it replies back with the data chunk.

D. *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. 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 query the system for data location information and migrate the processing tasks throughout the nodes in order to reduce or eliminate the data transfer time.

E. *Developer's task: Implementing Chunker classes*

Chunker classes define how files or data objects are split into data chunks. A chunker class is a class that implements a Chunker interface defining the following content:

a. Requests structures
b. Response structures
c. Data Requests handlers
d. Meta-Data Requests handlers (ensuring data-awareness)

An important feature of the system is that not all nodes need to hold all chunker classes known by the system. They only need the chunkers associated with the data they serve. If unknown type of data is requeste the node can dynamically load its associated chunker ar run-time.

V. INTEGRATIVE RESULTS

Figure 6 illustrates the integration of the two modules: SMBDS and DCFMS.

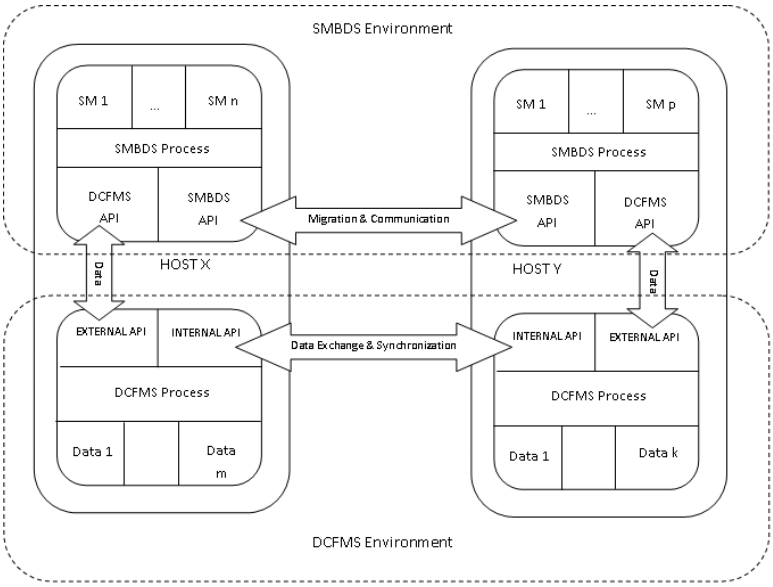

Fig. 6. SMBDS and DCFMS integration

Ideally each host would run one instance of each system. However this is not a constraint. For maximum performance it is recommended that each SMBDS process should be bound to a local DCFMS process. In what follows we will consider that this condition is being satisfied on all hosts. SMBDS will manage the execution of state machines acquiring data for their execution via the external API interface of DCFMS.

Once integrated, we are interested to evaluate the real performance and efficiency of the two modules. Recall that the main benefits of the two integrated modules are: load balancing by migrating tasks, logical partitioning of the data, and data awareness (localization) feature.

For obtaining an overview of these features, there will be compared the performance evaluations for two types of execution: the former execution which makes use of the features above and the latter which would instead use an uniform distribution of the work load involved without making

use of any of the above features.

It has been developed an image processing application whose main goal was to give an overview on the gain introduced by the features above. The processing algorithm applies gaussian filters on images, but this is irrelevant in our study. We are only interested on the improvement of the performance indicators following the two execution types.

The load balancing in our experiments made use of a very simple algorithm based on our observation that the transfer time of a data chunk between two hosts is N times longer than its processing time. There will be associated costs for each state machine. For example if a state machine needs to process a chunk of data that is not locally available, it will be associated a cost of N+1 (N for the transfer, 1 for the processing). The load balancing algorithm tries to keep hosts' costs as balanced as possible.

In our experiments each host will run in parallel five state machines, each machine having as job the processing of 9.2MB of image data.

As experimental environment it has been used a high speed Myrinet network with a bandwidth of 4 Gbps and 7 identical hosts (Intel Core 2 Duo E5200, 1GB of RAM memory).

We have observed that on this environment, the transfer time of a data chunk is about twice longer that the processing time of the same data chunk. That means we have used a cost factor of 3 for a state machine that needs to transfer its data, and a cost of 1 for a state machine that needs to process locally available data.

For evaluating the system we have considered four scenarios, as follows:

Scenario 1: All 7 hosts hold the entire data collection. In this scenario all machines will have a cost of 1 since they can access their associated data set locally.

Scenario 2: Five hosts hold the entire data collection. In this scenario two hosts will try to migrate a part of their tasks while transferring data for the rest of their machines.

Scenario 3: Three hosts hold the entire data collection. In this scenario four hosts will try to migrate a part of their tasks towards the data holding hosts while transferring the data for the rest of their machines.

Scenario 4: Only one host will hold the entire data set. This is the worst scenario as six hosts will have to acquire data for most of their machines. Only few machines will be migrated on the data holding host.

For each scenario there will be measured the data transfer time and the processing time for each host, as well as the total execution time.

Lets examine the results for each scenario.

A. *Scenario 1: All hosts hold the entire data collection*

In table 1 we can examine the results. In this scenario the state machines migration is useless as data can be accessed locally by all machines and the processing jobs are equally distributed on all hosts.

TABLE 1. RESULTS FOR SCENARIO 1.

Host	State Machines sent	State Machines received	Data transfers time(ms)	Processing Time (ms)	Execution time (ms)
1	0	0	0	151	151
2	0	0	0	149	149
3	0	0	0	145	145
4	0	0	0	152	152
5	0	0	0	157	157
6	0	0	0	159	159
7	0	0	0	152	162

Total execution time: 162 ms
Data transferred: 0 MB

B. *Scenario 2: Five hosts hold the entire data collection*

In this scenario each of the two hosts not holding any data have migrated 3 of their tasks towards the data holding hosts.

Table 2 presents the results for this scenario while table 3 presents the results for a statical run (not involving tasks migration) while figure 7 and 8 give a graphical overview of table 2 and 3.

TABLE 2. RESULTS FOR SCENARIO 2 – TASKS MIGRATION

Host	State Machines sent	State Machines received	Data transfers time(ms)	Processing Time (ms)	Execution time (ms)
1	3	0	169	68	237
2	3	0	178	74	252
3	0	2	0	191	191
4	0	1	0	184	184
5	0	1	0	180	180
6	0	1	0	202	202
7	0	1	0	198	198

Total execution time: 252 ms
Data transferred: 36.8 MB

TABLE 3. RESULTS FOR SCENARIO 2 – NO TASKS MIGRATION

Host	Transfer time (ms)	Processing time (ms)	Execution time (ms)
1	209	143	352
2	221	137	358
3	0	163	163
4	0	142	142
5	0	148	148
6	0	161	161
7	0	138	138

Total execution time: 358 ms

Data transferred: 92 MB

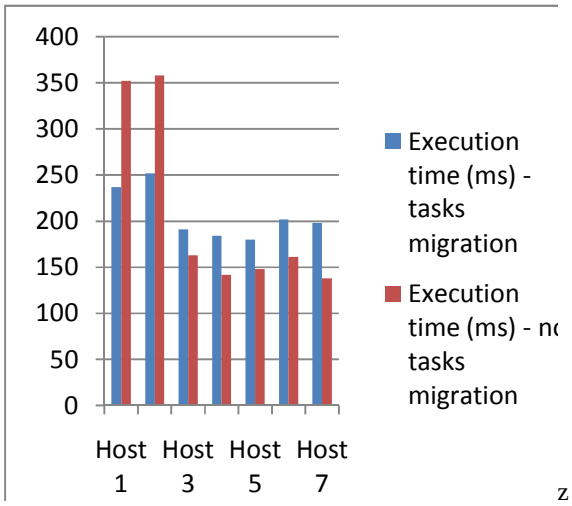

Figure 7. Scenario 2 processing results

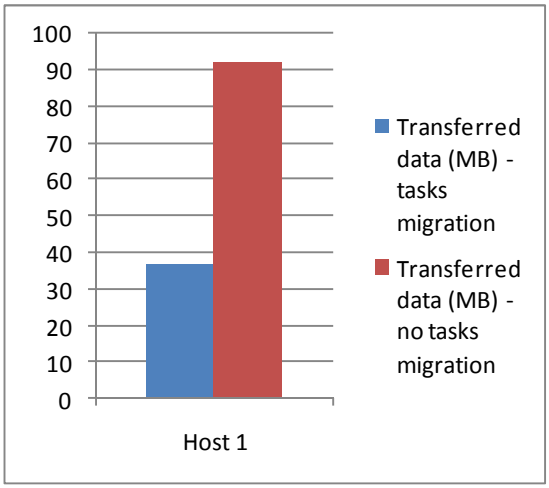

Figure 8. Scenario 2 data transfers.

We can observe that in this scenario there is a 29% gain in terms of processing times and a 60% reduction of the data transferred trough the network.

C. *Scenarion 3. Three hosts hold the entire data collection*

In this scenario each of the four hosts not holding any data have migrated 2 of their tasks towards the data holding hosts.

Table 4 presents the results for this scenario while table 5 presents the results for a statical run (not involving tasks migration) while figure 9 and 10 give a graphical overview of table 4 and 5.

TABLE 4. RESULTS FOR SCENARIO 3 – TASKS MIGRATION

Host	State Machines sent	State Machines received	Data transfers time(ms)	Processing Time (ms)	Execution time (ms)
1	2	0	202	84	286
2	2	0	213	81	294
3	2	0	198	83	281
4	2	0	207	86	293
5	0	3	0	245	245
6	0	2	0	211	211
7	0	3	0	239	239

Total execution time: 294 ms
Data transferred: 110.4 MB

TABLE 5. RESULTS FOR SCENARIO 3 – NO TASKS MIGRATION

Host	Transfer time (ms)	Processing time (ms)	Execution time (ms)
1	214	149	363
2	237	157	394
3	188	134	322
4	225	156	381
5	0	150	150
6	0	154	154
7	0	163	163

Total execution time: 394 ms
Data transferred: 184 MB

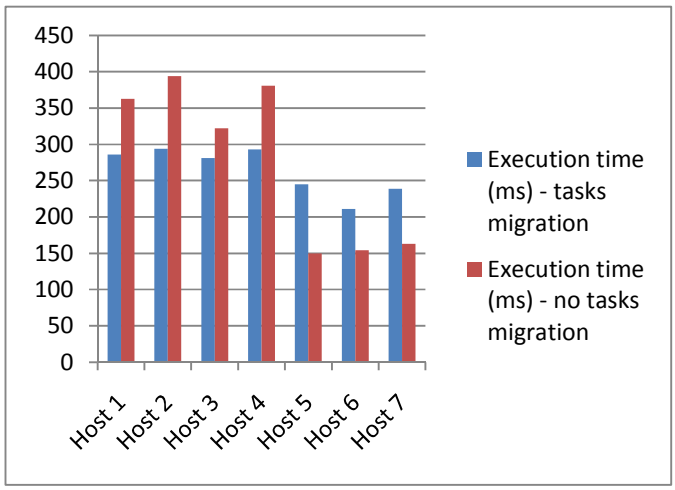

Figure 9. Scenario 3 processing results.

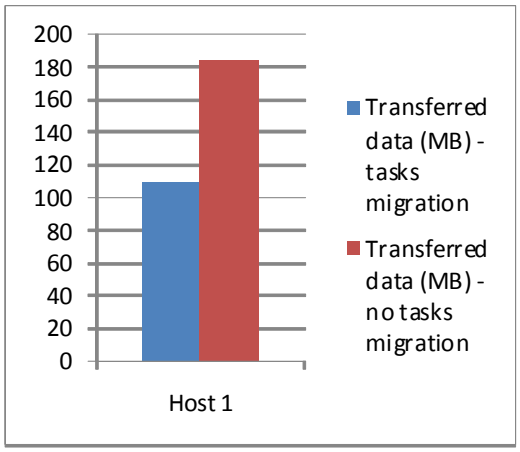

Figure 10. Scenario 3 data transfers.

We can observe that in this scenario there is a 25% gain in terms of processing times and a 40% reduction of the data transferred trough the network.

D. Scenario 4. One host holds the entire data collection

In this scenario each of the six hosts not holding any data have migrated one of their tasks towards the data holding hosts.

Table 6 presents the results for this scenario while table 7 presents the results for a statical run (not involving tasks migration) while figure 11 and 12 give a graphical overview of table 6 and 7.

TABLE 6.RESULTS FOR SCENARIO 4 – TASKS MIGRATION

Host	State Machines sent	State Machines received	Data transfers time(ms)	Processing Time (ms)	Execution time (ms)
1	1	0	238	115	353
2	1	0	260	124	384
3	1	0	242	121	363
4	1	0	263	132	395
5	1	0	267	139	406
6	1	0	235	126	361
7	0	6	0	314	314

Total execution time 406 ms
Data transferred: 220.8 MB

TABLE 6.RESULTS FOR SCENARIO 4 – NO TASKS MIGRATION

Host	Transfer time (ms)	Processing time (ms)	Execution time (ms)
1	223	121	344
2	243	132	375
3	246	137	383
4	253	157	410
5	263	129	392
6	248	119	367
7	0	155	155

Total execution time 410
Data transferred: 276 MB

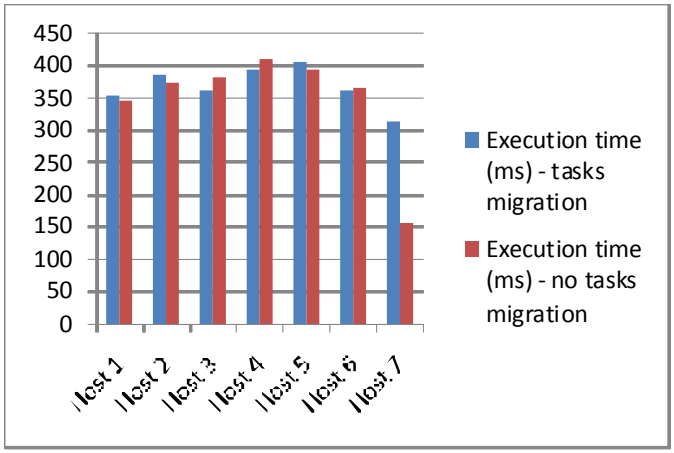

Figure 11. Scenario 4 processing results.

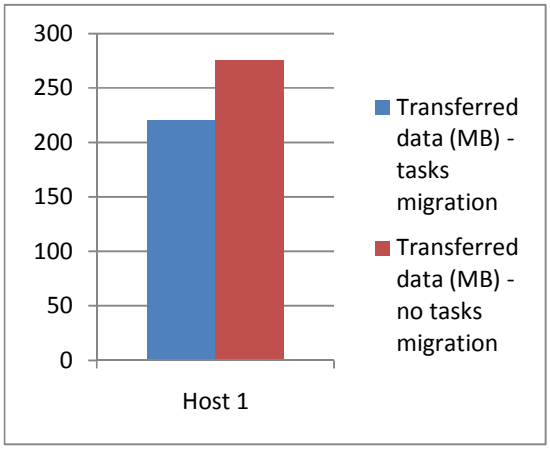

Figure 12. Scenario 4 data transfers

In this scenario one could notice an insignificant improvement in terms of processing time. Instead there is a 20% gain in terms of data transferred trough the network.

VI. Conclusions and Future Work

This paper introduces a new distributed architecture based on state machines concept. The framework has been designed especially for applications requiring high data availability.

The platform is composed by two independent modules SMBDS and DCFMS interconnected via an API interface.

The architecture proved to be scalable, flexible and reduces the development time considerably by providing an engine for state machines management and migration.

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 maintaining a high data availability.

The system has been experimentally evaluated in a very high speed network (Myrinet). Two types of executions have been considered: the former type employed the migration, data localization and logical partitioning features, while the latter didn't. Results for both execution types have been compared in four different scenarios.

The features introduced by the system improved the execution time with up to 30% while reducing data transfers over the network by up to 60%.

Future directions in the development of the system may include the hosts' speed ranking which could become 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.

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 [16], 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] Tadashu Watanabe - Numerical Simulation of Flow Field in and around a Droplet in an Acoustic Standing Wave, Recent Advances in Fluid Mechanics and Heat & Mass Transfer, pp170-175, ISBN 978-1-61804-026-8, WSEAS, Florence, Italy, August 23-25, 2011

[2] Hong Chen, Weimin Zheng, Aiqing Zhang - Parallelization and I/O Optimization for a 3-D Plasma Simulation Code, Recent Advances in Computers, Proceedings of the 13th WSEAS International Conference on COMPUTERS (part of the 13th WSEAS CSCC Multiconference), ISBN 978-960-474-099-4, Rodos, Greece, July 23-25, 2009

[3] 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)

[4] 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

[5] Cosmin M. Poteras, Mihai L. Mocanu - A State Machine-Based Parallel Paradigm Applied in the Design of a Visualization and Steering Framework, Recent Researches in Applied Informatics, Proceedings of the 2nd International conference on Applied Informatics and Computing Theory (AICT '11), ISBN : 978-1-61804-034-3, pp232-236, WSEAS, Prague, Czech Republic, September 26-28, 2011

[6] Mihai L. Mocanu, Cosmin M. Poteras, Constantin S. Petrisor - Improving Parallel Data Flow Support in a Visualization and Steering Environment, Recent Researches in Applied Informatics, Proceedings of the 2nd International conference on Applied Informatics and Computing Theory (AICT '11), ISBN : 978-1-61804-034-3, pp226-231, WSEAS, Prague, Czech Republic, September 26-28, 2011

[7] Morris Riedel, Wolfgang Frings, Sonja Habbinga, Thomas Eickermann, Daniel Mallmann, Achim Streit, Felix Wolf, Thomas Lippert, Andreas Ernst, Rainer Spurzem: Extending the collaborative online visualization and steering framework for computational Grids with attribute-based authorization. GRID 2008: 104-111

[8] S. Jha, S. Pickles, and A. Porter. A Computational Steering API for Scientific Grid Applications: Design, Implementation and Lessons. In Workshop on Grid Application Programming Interfaces, Brussels, Belgium, Sept. 2004.

[9] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning and A. R. Porter, Computational Steering in RealityGrid, Proceedings of the UK e-Science All Hands Meeting, September 2-4, 2003

[10] J. A. Kohl and P. M. Papadopoulos. Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. In 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, OR, Aug. 1998.

[11] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. Intl. Journal of High Performance Computing Applications, 11(3):224-236, Aug. 1997.

[12] J.J. van Wijk and R. van Liere. An environment for computational steering. In G.M. Nielson, H. M¨uller, and H. Hagen, editors, Scientific Visualization: Overviews, Methodologies, and Techniques, pages 89–110. Computer Society Press, 1997.

[13] R. van Liere, J.D. Mulder, and J.J. van Wijk. Computational steering. Future Generation Computer Systems, 12(5):441–450, April 1997.

[14] B. Cohen. The BitTorrent Protocol Specification, BitTorrent.org. (10-Jan-2008)

[15] The Apache™ Hadoop™ project, http://hadoop.apache.org/

[16] *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