An efficient Framework for distributed Computing in heterogeneous Beowulf Clusters and Cluster-Management

Darius Malysiak
Hochschule Ruhr West
Computer Science Institute
Bottrop, Germany
darius.malysiak@hs-ruhrwest.de

Uwe Handmann
Hochschule Ruhr West
Computer Science Institute
Bottrop, Germany
uwe.handmann@hs-ruhrwest.de

***Abstract*—In the context of existing approaches to cluster computing we present a newly developed modular framework 'SimpleHydra' for rapid deployment and management of Beowulf clusters. Instead of focusing only the pure computation tasks on homogeneous clusters (i.e. clusters with identically set up nodes), this framework aims to ease the configuration of heterogeneous clusters and to provide a low-level / high-level object-oriented API for low-latency distributed computing. Our framework does not make any restrictions regarding the hardware and minimizes the use of external libraries to the case of special modules. In addition to that our framework enables the user to develop highly dynamic cluster topologies. We describe the framework's general structure as well as time critical elements, give application examples in the 'Big-Data' context during a research project and briefly discuss additional features. Furthermore we give a thorough theoretical time/space complexity analysis of our implemented methods and general approaches.**

***Keywords*-cluster management, distributed computing, beowulf cluster, efficient distribution, load balancing**

I. Introduction and previous work

There exists a wide variety of different Big-Data problems, be it in scientific research or industrial applications. Developed solutions (algorithms or systems) often benefit from computation clusters, i.e. they are constructed to be paralizable such that they may be distributed among many computation nodes.
Although cluster computing is a very interesting and active field of research, it is difficult to access for many (small) research institutes. Professional high performance systems are often unaffordable, thus universities or research institutes usually decide to use inexpensive Beowulf clusters [1] or related approaches. Examples are [2], [3] or [4], yet most clusters are designed to solve domain specific problems. If they provide a generic API, they often do not include support for cluster management, e.g. adding new nodes or updating/reconfiguring existing nodes. Additionally most Beowulf clusters assume heterogeneous nodes or a static topology, which is a serious restriction for partial system upgrades. Solutions are usually implemented by using plain communication abstractions like PVM [5] or MPI [6], which provide a simple and efficient way for distributed computing, yet these APIs do not address generic concepts of cluster computing (e.g. load balancing strategies, node management). Many (domain specific) extensions for these interfaces exist, e.g. [7](enhanced PVM load balancing) or [8](PVM over ATM lines), which address certain aspects of cluster computation. Often do frameworks include large dependencies to other external libraries, which are not guaranteed to work with future revisions.
The SCMS [9] framework addresses the management problems of Beowulf clusters and provides a practical set of functions. Yet its purpose is solely the management, it does not include inherent support for computation tasks. Building upon SCMS and other frameworks, SCE [10] provides a solution including support for computation tasks by using MPI. Yet, a rigorous analysis of its structure and implementation (e.g. of the low-level network communication with respect to current technologies) is ommited.
We aim to address the problems of Beowulf based computation by providing an integrated but modular framework which not only enables one to rapidly deploy and manage Beowulf clusters but also scales well for huge systems ($>$1000 nodes). Additionally our framework includes support for OpenCL based computation which alongside our support for dynamic heterogeneous topologies provides the basis for a flexible system structure.
Section II will outline the general structure of our system, while the critical aspect of network communication will be addressed in section III. The previously mentioned dynamic topologies will be explained in section IV. We will conclude this paper with the description of the IGOR cluster which was utilized in the APFel research project [11] for distributed and GPU-accelerated people detection. Additionally we will elaborate on the results which were obtained by using SimpleHydra on the aforementioned cluster, thus demonstrating the frameworks potential.

II. A coarse look on the structure

We begin by describing the frameworks modular structure which incorporates the largest modules:

978-1-4799-5338-7/14/$31.00 ©2014 IEEE

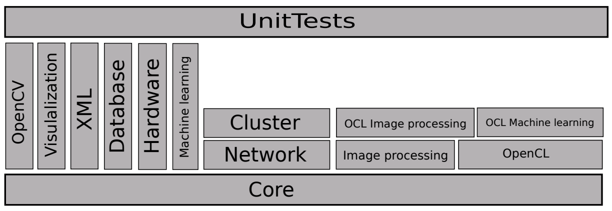

Figure 1. The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

- Core
- Network
- Cluster

The 'Core'-module provides all basic data structures and management functions for the remaining elements, e.g. filesystem support, IPC (inter-process communication) support, a thread management system, time measurement components, serialization facilities and others. All fundamental communication methods are provided by the 'Network'-module, due to its size and complexity we will describe its structure more detailed in section III. The 'Cluster'-module contains high level management routines which enable developers to quickly deploy canonical (i.e. framework provided) management clients and servers for a cluster infrastructure.
In addition to these modules, SimpleHydra (SH) provides a wide functional variety in the areas:

- data exchange (e.g. Matlab interface, XML support for basic XML access and configuration files, MySQL database connectivity)
- image processing (e.g. elementary image manipulation, OpenCL based high performance object detection)
- machine learning (e.g. LIBSVM wrapper, generic and adaptive neural networks with OpenCL support)
- hardware support for video input devices (e.g. V4L devices, AVT cameras)
- data visualization (e.g. video streams, images or functions)

Fig. 1 illustrates the described components and shows their dependencies, i.e. a module requires the components it stands on. The environment requirements in terms of soft- and hardware are very puristic throughout the different modules. Due to efficiency (e.g. threading, time measurement) / cost (e.g. licenses) considerations we decided to implement the framework only for Linux/Unix systems. SH provides interfaces to proprietary libraries, e.g. Matlab, yet this is a requirement solely for the corresponding module (e.g. 'Matlab'-module). Due to the frameworks size, we decided to utilize CMake and bash scripts for the build chain. This allows a fast creation of customized build configurations, e.g. for a small embedded system like a RaspberryPie one could only build the modules 'Core' and 'Network'. The minimal software requirements are a Linux/Unix system, a C++ compiler, the C++ standard library and CMake. There are no hardware restrictions. In order to keep things brief, we just list the external dependencies for each module ('<o>' indicates it as being optional):

- Core(<o>libz, libpthread)
- Network
- Cluster
- Database (libmysql || libmariadbclient, libboost_regex)
- Hardware (libVimbaCPP, libVimbaC)
- OpenCL (libOpenCL)
- Matlab (libmat)
- ImageProcessing (libpng, libjpeg)
- OpenCLImageProcessing
- OpenCLMachineLearning
- MachineLearning (libSVM)
- XML (libxml2)
- Visualization (Qt5, libcustomplot, qwt)
- UnitTests

The reason for such a sparse amount of small external libraries lies in the fact, that SH implements many data structures and elementary control mechanisms from scratch. This is needed to provide system-local thread safety while keeping the data access to primitive data containers (e.g. linked lists) fast. The framework incorporates a build chain which generates release and debug make scripts for static and shared libraries (module wise). In addition to these libraries one can build an executable containing the unit tests.

III. Network protocol and communication facilities

One of the most critical aspects in building a cluster is the communication bandwidth and latency between nodes. It is not only a question of choosing an appropriate physical interface but also the communication protocol. Professional high performance systems often use the Infiniband interface which provides a 2.5GBs link [12] and drive their data with TCP. Infiniband also has a much smaller latency of $\approx 1.7\mu s$ compared to GigE ethernet $\approx 48\mu s$ [13]. Although one might be tempted to use this interface for IPC, it does come with high hardware costs (NICs, switches etc.). Thus for small research institutes GigE (available on almost any modern computer) represents a cost efficient alternative to aforementioned HPC systems.
The concept of Beowulf clusters exists since 1995 [1] and initially described a set of Ethernet-connected workstations, whose communication based on a tocken exchange via UDP. Yet UDP is a connectionless IP based method to transmit data, i.e. one does not have congestion control, receive control, ordering of packet fragments or reachability information about the communication partner. For simple domain specific Beowulf clusters the choice of using UDP might be well founded, e.g. small and sparsely exchanged tokens under the restriction of largely available network /

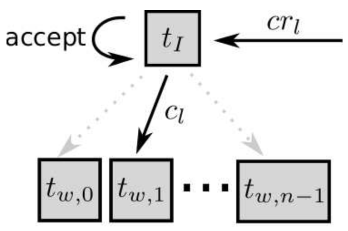

Figure 2. The concept of binned worker threads; one thread t_I handles the incoming connection requests cr_l, creates the connection c_j and assigns it to an appropriate worker thread $t_{w,i}$ (bin)

node capacities. But for a generic approach, e.g. taking the management and control of the cluster into account, with respect to unknown fields of applications as well as heterogeneous hardware configurations, the control requirements for network communication will converge to the feature set of TCP.
Thus we decided to implement the communication via connection-based TCP, the reasons for this choice will become clearer when we discuss the management feature set of SH. It should be mentioned that the SH framework does also utilize UDP for e.g. dynamic cluster topologies and is not restricted to TCP based communication, yet it does not provide high communication facilities with UDP.

A. SH communication

Before discussing the internal mechanism we would like to point out that even though we will carry the described communication facilities throughout the remaining paper, SH provides a generic API which allows developers to change/ implement existing or new communication protocols down to the choice of sockets (e.g. as far as to choose packet sockets).
Communication, in management or computation relevant tasks, uses TCP payloads p of the form

$$p = [h|d] \tag{1}$$

where h is 4 bytes long and contains the size in bytes of the actual data d. Thus the shortest communication beacon will be 4 bytes large. In order to avoid synchronization problems or race conditions during heavy data exchanges, we define each data transmission to be of a request-response form.

B. Worker threads

When it comes to socket communication under Linux/Unix systems the usual naive way of handling incoming connections is to start a single thread for each one of them. This approach is infeasible for large scale servers as it will clog the system with management overhead. Thus in large server applications the concept of binned worker threads is applied, this is depicted in Fig. 2
Our system allocates a thread pool of n worker threads $t_{w,i}$ and starts a single connection handling thread t_I. Every connection request will be processed by t_I and delegated to a fitting thread $t_{w,i}$. The term 'fitting' already indicates that the choice of i is not arbitrary, the simplest strategy would be an even distribution among the workers. Due to time restriction we only implemented this approach, i.e. each single worker can handle up to m connections, in case of nm existing connections every incoming connection will be dropped. A more advanced way would consider the current load of the worker threads and e.g. choose the one with the lowest value.

C. Efficient socket handling

The Linux kernel provides different mechanisms for accessing data in a socket (or checking for available data), namely *select*, *poll* and *epoll*. A call to *select* is the most basic way of checking for available data, it informs the kernel about all file descriptors (i.e. socket descriptors) it would like to check for new events. This approach does not scale well with a growing number of open file descriptors. The same holds for *poll* which differs to *select* only in the number of maximal file descriptors (i.e. it has no fundamental limit compared to the bit mask approach of *select* [14]). The *epoll* function removes this drawback as it only considers the active file descriptors, i.e. it does not require to provide the kernel with a list of desired elements. Both approaches were thoroughly analyzed in [15], who showed that *epoll* exhibits a measurable performance gain of up to 79% for sparse connection activity. Thus we decided to utilize *edge-triggered epoll* in our framework.
SimpleHydra's worker threads use so-called frame assemblers, in order to explain those we must begin with the problem they solve. For the sake of simplicity assume that only a single worker thread $t_{w,0}$ exists and handles m connections. For each of these m connections $t_{w,0}$ will have to assemble the corresponding data streams, as they may arrive in (ordered) fragments. Furthermore $t_{w,0}$ must apply the desired action (e.g. a callback) to the data streams payload. Thus each connection c_j is assigned a single frame assembler a_j which handles the logic behind the assembling (buffer management and construction) as well as the interpretation of the data. The interpretation is done via frame handlers fh_j which are an integral part of each frame assembler (one per assembler).
The process structure of socket management within a worker thread is illustrated through alg. 1. Yet another problem arises in the context of binned worker threads. Let us assume $t_{w,0}$ processes the low-level socket descriptor sd_j of c_j, how does he find a_j efficiently in order to deliver the received data to it? In order to solve this we applied an unordered hash list ($\mathcal{O}(\log(n))$) holding the tuples (sd_j, a_j) with sd_j being the key. One should note that this problem could be solved differently e.g. by including a connection id into the header h, thus the worker thread could up look the frame assembler in a linear array. Yet this

would require a management of available slots in the array. Using the socket descriptor as a key for the linear array itself is infeasible due to its numerical range (4/8 bytes), i.e. this would restrict the array to be continuously growing with each new connection (especially critical for the case of very frequent closed and reopened connections over a long time period), i.e. we would gain $\mathcal{O}(1)$ worst-case lookup time for the cost of a limited system runtime due to a finite memory amount. This dilemma can not be avoided for situations with a variable amount of non-persistent connections.

Algorithm 1 Worker thread $t_{w,i}$ socket management

```
1:  while worker is active do
2:      ($num, event$) =getActiveConnections;    $\rightarrow$ epoll
3:      for i=0; $num$ - 1 do    $\rightarrow$ determine request type
4:          if $event[i].req$=="disconnect" then
5:              find and delete connection from container;
6:          end if
7:          if $event[i].req$=="connect" then
8:              create and add connection to container;
9:          end if
10:         if $event[i].req$=="data" then
11:             find and call assembler $a_j$;
12:         end if
13:     end for
14: end while
```

Each worker thread contains a private epoll system which is used to observe the socket descriptors of all assigned connections. Thus we can summarize the average time complexity T_{sock} of our approach as follows (for the sake of simplicity we chose intuitive index names).

Lemma 1. *Let act_i be the number of active connections in $t_{w,i}$ with $i \in [0, n-1]$ and $act_i \in [1, m]$. Furthermore let D be a data structure, capable of holding integer values, with functions D_{get}, D_{add}, D_{del} and corresponding average complexity sets $\mathcal{O}_{get}(g(k))$, $\mathcal{O}_{add}(a(k))$, $\mathcal{O}_{del}(d(k))$ for k contained elements. Then the average time complexity for a single iteration of $t_{w,i}$ is*

$$T_{sock,i} = \mathcal{O}(\mathbb{E}(act_i) f(k)) \tag{2}$$

with f being a function from the largest of the mentioned complexity sets.
Furthermore the complete average complexity (for a single parallel iteration of all worker threads) is given by

$$T_{sock} = \mathcal{O}(\max_i(\mathbb{E}(act_i)) f(k)) \tag{3}$$

Proof: We have to distinguish two cases, firstly the case of $\mathbb{E}(act_i) = 0$, where the above statements obviously hold. Secondly the more interesting case of $\mathbb{E}(act_i) > 0$. First one has to observe that $\mathbb{E}(act_i)$ can be splitted into $\mathbb{E}(dis_i) + \mathbb{E}(new_i) + \mathbb{E}(get_i)$, where the expectancy values refer to the case of disconnect requests, new connections and existing connections, respectively. Furthermore there exist factors α, β, γ with $\alpha\mathbb{E}(act_i) = \mathbb{E}(dis_i)$ etc. (e.g. $\beta = (\mathbb{E}(del_i) - \mathbb{E}(get_i))/\mathbb{E}(act_i)$). Every worker has to retrieve the active sockets, this can be done in constant time due to preallocated kernel structures (or in $\mathbb{E}(act_i)$ steps from a rigorous point of view). After the descriptors have been retrieved one must process each one of them (i.e. $\mathbb{E}(act_i)$ descriptors), they may inform the program over disconnections, new connections or data for existing connections. For each request type one must execute data structure routines, i.e. D_{del}, D_{get}, D_{add}, respectively. Let $g' \in \mathcal{O}_{get}(g(k)), a' \in \mathcal{O}_{add}(a(k)), d' \in \mathcal{O}_{del}(d(k))$ be arbitrary functions. We can summarize the complexity for the processing of all requests by

$$\mathcal{O}(\mathbb{E}(act_i)(\alpha d' + \beta a' + \gamma g')) \tag{4}$$

which is dominated by the function with the largest asymptotic behaviour, i.e. f. Thus we obtain the complexity for a single iteration and for multiple parallel iterations (as n is constant). ∎

On the basis of lemma 1 it is simple to conduct further runtime analysis depending on the assumed distribution of act_i and the utilized data structure D. The extension for inclusion of high level functions for each request type can be done by adding their complexity to the complexity of the corresponding datastructure routines (i.e. d, a, g). One can also deduct that for a constant time complexity within the described threading concept, all of the datastructures operations must be able to finish in constant average time.

D. Memory management and space efficiency

Apart from the operating systems send and receive buffers, two additional buffers are required in which an assembler a_j can iteratively construct the outgoing and incoming data streams. In our system each a_j constructs an appropriate receive buffer for every incoming datastream, thus we allocate memory only if it is required (depicted on the left image in Fig. 3). Regarding the outgoing data we have chosen a different approach. The worker thread contains a single transmit buffer which is shared among the managed sockets. Each socket will either send all of his queued data or fail, under this restriction we can reduce the amount of required memory significantly (see the right image in Fig.3). Yet this strategy can not be applied for incoming (fragmented) data streams, as we have to store incomplete data streams over time until all fragments have been received.

The required buffer size for an incoming data stream is determined once the first 4 bytes (i.e. the header) have been received. Additionally the send process only considers the existing data in the shared output buffer, .i.e. it does

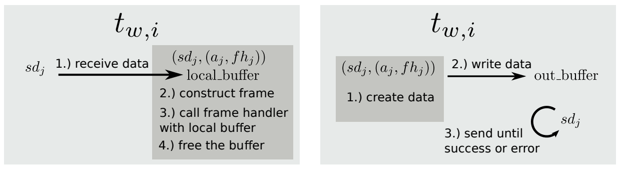

Figure 3. Left side: $t_{w,i}$ receives a data fragment within an iteration and directs them to the appropriate assembler a_j, which stores it in a local buffer and continues with frane reconstruction. Once a frame has been completely received, the framehandler fh_j will commence the interpretation of the payload. Right side: the frame handler fh_j attempts to send data over sd_j, first the data is copied into the worker threads shared output buffer, afterwards the worker thread attempts send all data contained in the buffer (i.e. only the existing payload). The colored rectangles represent different contexts.

not send all allocated buffer bytes. The output buffer size is determined during runtime by analyzing certain system attributes.

As mentioned before our protocol uses a simple request-response scheme, this simplifies the logic behind frame assembling. Through the use of TCP we receive data fragments in correct order, thus the assemble process is a simple concatenation of bytes. The process of frame construction is depicted in alg. 2, each computational step can be done in $\mathcal{O}(1)$. The complexity is mainly determined by the call to fh_j, which can commence arbitrary actions with respect to the received payload.

Thus we can summarize the complexity of our communi-

Algorithm 2 Frame assembling in a_j

```
Require: data fragment $d$
 1: [static init] buffer = $\emptyset$; bytes = 0; payload_size = 0;
 2: if bytes $< 4$ then
 3:     append $d$ to buffer;
 4:     bytes += sizeof($d$);
 5:     return
 6: end if
 7: if bytes $\geq 4 \wedge$ payload_size $==0$ then
 8:     payload_size = $h$    $\rightarrow h$=bytes[0,..,3]
 9:     append $d$ to buffer;
10:     bytes += sizeof($d$);
11:     return
12: end if
13: if payload_size $> 0$ then
14:     append $d$ to buffer;
15:     bytes += sizeof($d$);
16:     if bytes == payload_size then
17:         call $fh_j$ with buffer
18:         buffer = $\emptyset$; bytes = 0; payload_size = 0;
19:     end if
20:     return
21: end if
```

cation protocol with

Theorem 1. *Let Ω be the average complexity set for actions taken by framehandlers fh_j in a given context and $\omega \in \Omega$. The basic SimpleHydra network communication system, with respect to a single (parallel) iteration of all worker threads, exhibits a complexity of*

$$T_{com}(.) = \mathcal{O}(\max_i(\mathbb{E}(act_i))(f(k) + w(.))) \tag{5}$$

Proof: Follows directly from lemma 1 and the corresponding remarks. ∎

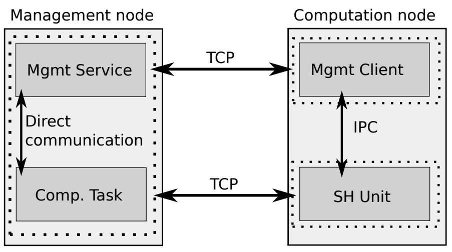

Figure 4. The canonical network service structure of SimpleHydra, the dashed rectangles represent different address spaces. Management of the computation nodes is done via a TCP connection between two corresponding services. These services are independent of the actual computation task but can communicate with it either via direct addressing or IPC. The node interaction during computation tasks is also done via TCP connections.

IV. Dynamic topologies

The communication topology of a Beowulf cluster is star shaped, with a management node in the center which distributes work among the available nodes (including itself). This topology is usually assumed to be static in terms of e.g. node count or communication interface. Additionally it is assumed that the nodes are similar (if not identical) configured. Yet some applications benefit from a dynamical topology, e.g. one which allows the insertion or removal of nodes during runtime, where the nodes may be differently structured (e.g. powerful multi-GPU nodes).

Thus we designed our framework in a way which allows the configuration of such clusters, furthermore we provide a low-level API which allows not only the construction of highly dynamic topologies but also their runtime management.

The general structure of this system is depicted in Fig. 4, where the management node executes two distinct (independent but connected) subprograms; the *management service* and the *computation task*. The management service is capable of e.g. keeping track of each node's available computation ressources, copying data onto nodes or executing arbitrary system commands. The computation task has the responsibility of managing work distribution among the nodes.

Each node runs the corresponding counter parts; the *management client* and the *SH Unit*. We will describe the concept of SH Units in the next section, for now it should suffice to consider an SH Unit as distributed workload. Similar to the management node, the subprograms on a computation node are independent but connected. The motivation for

this design was to keep the cluster stability as high as possible. Even if an SH Unit fails, e.g. enters an endless loop, the management node can still use the connection to the management client to stop the Unit through the node's operating system.
The management service and computation task are being executed in the same system process (but in separate threads). For the sake of simplicity let us assume that each node already runs the management client. First the management service will be started, it will find all available nodes on the network (predefined or dynamic, details in next subsection) and setup a connection to them (i.e. the running clients). Afterwards it will distribute SH Units among them and start the computation task. Each SH Unit will then connect to the computation task and the actual work may commence.
We point out that the computation nodes establish the connection, thus they have to handle only two connections (one for management and one for computation tasks), whereas the management node will handle its connections efficiently via the approach described in section III.

A. *Self-configuring clusters*

We will now detail how the connections between nodes are being established. Within the previously described approach one might assume that the management node carries an initial list of all potentially available nodes. This is not required as we designed SimpleHydra for self-configuring clusters.
The management client contains an UDP Remote Control Service (UDPRCS), one of its functionalities being the ability to reply to home beacons (33 Byte large UDP broadcasts containing information about the management node). First the management node M will send a home beacon, all available nodes n_i may answer to it, M will wait for a defined time and create a list $N = \{n_i\}$ of available nodes. Independent of that, the nodes n_i will connect to the management service at M (the required data is extracted from the home beacon). The management node may use N to verify if all nodes have connected to it. Afterwards M will use these connections to distribute data and instructions to the nodes. Once the nodes have received all initial data they will execute the instructions (e.g. set up a connection to the computation task on M). This scheme is illustrated in Fig. 5, for the sake of understanding we ommited the details of synchronization e.g. the management client will wait until the SH Unit has been successfully deployed. Additionally we left out the details of network communication like response messages.
Using the UDPRCS one can build architectures which allow the online expansion of computational resources. Yet, this approach works only on local subnets and thus SimpleHydra also supports the use of static node lists. These node lists allow the configuration of clusters in wide area networks, i.e. the provide the possibility for grid computing.

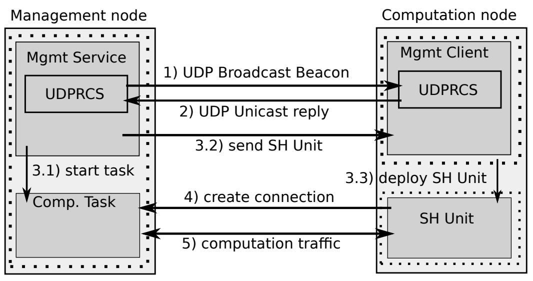

Figure 5. The process of self-configuring clusters with SimpleHydra. The numbers inside the annotations denote the order of execution. First the management node attempts to find all available nodes on the local subnet via a UDP broadcast. The available nodes reply to the beacon and extract the servers connection information (e.g. address, port) fromt it. Using this information they establish a connection to the management server which, once all nodes have connected, starts the computation task and sends an SH Unit to the nodes. Once a node received the Unit, it will deploy and execute it. The actual computation may then begin. For the sake of transparency the synchronization details of communication have been ommited.

V. Cluster Management

One often underestimated point is the management of a cluster, this includes tasks as setting up single nodes, keeping track of available nodes, updating software (e.g. libraries) on nodes and many more. For larger cluster systems (>20 nodes) these tasks create serious overhead for the administrator and introduce downtimes to the cluster. In order to accommodate this we provide a generic function set along with the management service, including e.g.:

- Remote shell (synchronous, asynchronous, parallel on multiple nodes)
- File copy
- Resource querying (CPU load, free HDD space etc.)

A developer can use these functions to create solutions for more complex tasks, e.g. update multiple nodes by copying a script to them and using a parallel shell to execute it.

VI. Workload distribution paradigm

Let us consider the following scenario, an existing cluster with identically configured nodes (i.e. identical soft- and hardware) should be upgraded during runtime with one additional node. Yet this node contains almost completely different hard- and software (still a Linux/Unix system though). If the workload (i.e. the data and program code) would have been distributed as binary data, it would be impossible to use it on the new node. In order to address situations like this we developed the concept of SH Units.

A. *SH units*

An SH Unit is a datastructure $u = (\mathcal{P}, d)$, with $\mathcal{P}$ being an arbitrary payload and d a deployment script. Technically u has the form of a compressed archive which contains C/C++ source files and binary data (i.e. the payload $\mathcal{P}$), the deployment script (i.e. d) usually consists of a single '*.sh' file. Alg. 3 illustrates the process of deployment on the computation node.

Algorithm 3 Deployment of an SH Unit u

```
Require: SH unit $u$
1: unpack archive into temporary folder
2: execute $d$    $\rightarrow$ e.g. compile source files and execute the binary
```

The algorithm is very short as all deployment logic will be included in the deployment script. It can execute arbitrary commands, compiling the source files within the payload is only one of them. It may also gather system information and provide them to the executed binary or copy needed files to specific locations. The only statically defined step in the context of an SH Unit is the execution of the deployment script, which can also be used for administrative tasks. As mentioned in section IV the started SH Unit (i.e. the compiled binary) can communicate with the management service through IPC. Thus, although the Unit is started as a new process the management service still has low-level control over it. In order to enable rapid prototyping we provide SH Unit / computation task templates for e.g. map and pipe skeletons. The developer might also create completely new tasks with ease (as our system provides a transparent and generic class structure).

B. Load balancing

During runtime it is crucial to distribute the workload adequately among the nodes. Factors may be power efficiency [16], memory attributes [17] or domain specific optimizations [18], just to name a few. We do not aim to provide implemented solutions for any of such problems, instead our framework provides the needed infrastructure for implementing the corresponding solutions.
The management service does not have to be completely idle during computation, it may e.g. collect information about the nodes. This can be done by e.g. polling all nodes with a fixed frequency or letting the nodes themself transfer a status report (6). The computation task may utilize this information for an arbitrary scheduling algorithm.

VII. Inner workings of SH

Every connected node is being assigned a unique node id which is stored in a temporary database on the management node. A computation task can access the contained entries (which also include other node metadata) and e.g. decide whether a node is capable of executing a certain operation. The details of collecting metadata is depicted in Fig. 6, the client c connects to M, during this process c gets assigned a node id and additionally sends a small system report R. R contains e.g. the number of OpenCL devices, CUDA devices, CPU information, the amount of RAM, available HDD space and many more. At the end of the registration the report data is saved in a local database. A workload scheduling algorithm can utilize this meta information for a

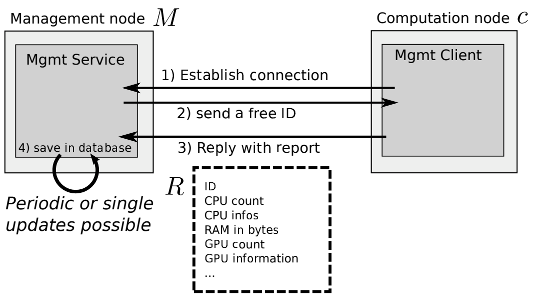

Figure 6. The simplified registration process of a new node c and the management node M. First c establishes the connection to M, which in turn sends a free ID (this number is removed from the pool of available IDs until c disconnects or the registration fails). The computation node then acknowledges this number by sending it back as an attributes of a system report R. The management node then saves the report data in a local database and assigns the ID to the corresponding connection. It is possbile to update the report data periodically (i.e. c sends periodic updates) or only per request (i.e. M requests an update from c).

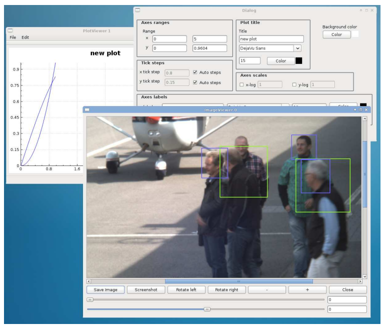

Figure 7. The window system is part of the visualization module, it contains e.g. sophisticated plot features, image viewers with annotation functions, video viewers and cluster management tools like e.g. parallel remote shells.

balanced distribution of tasks. It is also possible to periodically update the report information for a set of nodes (single updates are possible as well). For the sake of understanding we left out any timers (which are used in steps 2 and 3) within this illustration.
The communication between management client and SH Unit on the node side is done via IPC mechanisms. For this and other purposes, SimpleHydra supports both, *System V* and *POSIX* based shared memory IPC. Another application for IPC is the libraries visualization module (see Fig. 7), which uses shared mutexes for synchronization with the Qt based window system. It must be noted that although SimpleHydra provides an object oriented interface to this window system, the system itself provides a low level C language interface, which allows its use in native C programs.
We will now explain how our protocol avoids race conditions and synchronization problems. Let us assume (without loss of generality) three communication parties A, B and C are communicating with each other, A and B are separate threads on a management node while C is a thread on a

computation node. Furthermore let A be receiving a large file from C and B preparing to request a status report from C. The protocol uses a sequential request-response format, i.e. the communication tasks are queued and will be sequentially processed. For our example this means that while A receives data, no other communication will occur on this connection (i.e. B will wait for A to finish). This allows a simple and fast messsage parsing for each connection as the receiver can be sure that he receives a coherent payload stream, i.e. only the data from one communication context (no interleaved fragments). Once A has received all the data, the next enqueued communication will commence, i.e. B will send a request to C, which in turn will respond with a status report. It must be noted that such a connection scheme has its drawbacks, e.g. for a connection which transfers mostly large files, short messages will experience a great latency. Yet in the context of Beowulf cluster computing one usually avoids great amounts of network communication due to the small bandwidth and high latency of ethernet, additionally the exchanged data blocks are often small and in similar size. Due to these reasons we implemented the described sequential approach, yet our framework gives the user the freedom to implement a (situation-)optimized communication scheme.

VIII. SimpleHydra deployment in a cluster

In order to efficiently use our software we also optimized the deployment in an existing infrastructure. A Beowulf cluster makes little to no assumptions about the used hardware ([1] refers to the computation nodes as simple workstations), thus a corresponding framework should be able to deal with a large variety of hardware configurations. The most relevant hardware elements are system memory, CPUs, GPUs, HDDs and network interfaces, which our framework handles in a generic manner. Many frameworks incorporate similar functionality but achieve it through the use of external libraries, which often are restricted to certain hard- or software configurations and can change their APIs anytime (which forces one to adapt the framework).
In order to circumvent these problems and restrictions we designed our framework from scratch with only a minimal amount of external dependencies (the biggest being the Qt framework for the visualization module). This makes deployment in a network environment very easy, for the basic distribution of workloads one only needs to copy a small (prebuilt) client demon to the workstation (if no prebuilt demon is available, one has to compile it either on the workstation or with a fitting crosscompiler). No further configuration is required, especially no external libraries besides the C/C++ standard library. This deployment can be archived via e.g. a small shell script, the client demon itself is very puristic and effectively consumes no system resources. Once this demon has started, the workstation becomes an available computation node. It should be noted that this demon can reside on the workstation after the computation tasks finished, thus the workstation will be made available for e.g. tasks of another management node. The demon also determines the available system features, e.g. OpenCL, CUDA, memory amount etc., yet it is also possible to configure it with a static configuration file which defines the available services. These static configuration files are useful in the case of strongly varying or special environments, which can make it difficult to reliably determine certain system features.

IX. Application

In order to test and demonstrate the potential of SimpleHydra we constructed a Beowulf cluster IGOR (Intelligent GPU based Object Recognizer) and utilized it within the APFel research project. Throughout this section we will describe IGORs system structure, the deployed tasks and discuss the results.

A. The Beowulf cluster IGOR

IGOR consists of 15 nodes in total, which are equipped as follows:

- 1 nodes: Intel i7 4770, 3x Radeon 7990, 64GB RAM, 12TB HDD (3 disks)
- 6 nodes: Intel i7 4770, 1x Radeon 7970, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 1x Radeon R9 290x, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 1x Geforce GTX780-Ti, 16GB RAM, 1TB HDD
- 2 nodes: Intel i7 4770, 2x Radeon R9 290x, 64GB RAM, 8TB HDD (2 disks)
- 1 nodes: Intel i7 4770, 3x Geforce GTX780-Ti, 64GB RAM, 8TB HDD (2 disks)
- 1 node: Intel i7 4770, 16GB RAM, 1TB HDD

The last node was used as a dedicated management node, i.e. it did not participate in the execution of SH Units (Fig. 8 shows the two different types of nodes). Thus in total IGOR features 56 x64 CPU-cores with 3.6GHz, 368GB of system memory and 55872 GPU shaders, with a total of $\approx$ 107 TFlops (synthetic, FP32 GPU) and 2.38 TFlops (synthetic, FP32 CPU). Except for the identical CPU the nodes exhibit different amounts of RAM, different GPU counts and types (this implies different local tool chains and interfaces) and different amounts of HDD storage. The nodes were interconnected via a dedicated GigE switch and used different versions of ArchLinux.

B. The APFel person detection system

Our detection system (Fig 9) is described in [11] with more detail, for this paper we will give a short summary and discuss our testing methods.
Two problem instances have to be distinguished; the classifier training and the case of detection tasks.

Figure 8. The two node types used in IGOR. The left picture shows very compact mini-ITX case with one discrete GPU, 16GB RAM and one HDD of 1TB. The right side depicts a large ATX tower with 64GB RAM, multiple GPUs and HDDs (8TB).

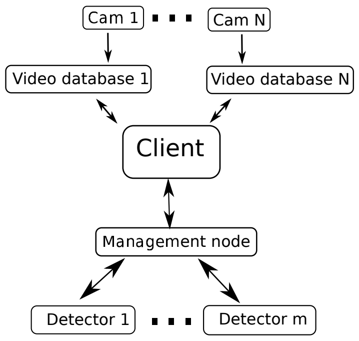

Figure 9. The detection system of the APFel project. The client obtains images from different video databases, he uses the management node of the Beowulf cluster as an abstraction proxy to the collected computation power of all m GPUs (which are distributed over n) nodes. The management node receives detection requests which also contain the corresponding image, it delegates these requests to a node with an available GPU, receives the results and forwards them to the client.

The detection algorithm utilizes a linear GPU-based support vector machine, which was trained with up to 1600 elements of dimension 3565. In order to find the optimal training parameters we used a gridsearch with the crossvalidation error as an objective function. We splitted the search grid into equally sized tiles and distributed them among the nodes which in turn executed the corresponding grid search. The management node collected the results and extracted the best parameters.

Regarding the case of detection tasks we employed a similar strategy. As depicted in Fig. 9, the management node receives detection requests, finds a node capable of handling them and delegates the task accordingly. The system processes multiple parallel videostreams with 10 fps.

C. Results

The most time consuming task during the construction of IGOR was the configuration of the different node types, as they contained either AMD or NVidia GPUs we needed to configure different tool chains. In order to speed up this process we utilized HDD image tools. For a more homogeneous hardware configuration a distributed filesystem might be better suited.

Our highly optimized GPU-based HOG-algorithm [11],[19] executes the detection on a single image in $\approx$ 60ms (processing $\approx$ 14GB during this time). The detection tasks were deployed among the nodes in a first come - first serve strategy, i.e. the task was assigned to the first found node with an unoccupied GPU. As we used our system in an offline mode (i.e. with recorded images), we handled the case of 'no free nodes' by simply letting the task wait until a CPU/GPU was available. The computing performance scaled linearly with the cluster's GPU count, i.e. detection times with videostreams were reduced by a factor of $1/m$ with m GPUs. The distributed gridsearch during SVM training reduced the training time by a factor of 14, as the SVM was trained on 14 identical CPUs in parallel. We measured raw communication latencies between the nodes of $\approx 45\mu$s while transfering 1kB of management payload for a low amount of 30 connections. These latencies have been completely masked by the (in comparison) large computation times of $\approx$60ms (detection process) and $\approx$20s (linear SVM gridsearch step) / $\approx$15m (RBF SVM gridsearch step). During our evaluation we used a total of 8 worker threads on the management node, i.e. 1 worker thread for the management server and 7 for the computation tasks (as we used the management service only for collecting the nodes system load). We successfully tested the insertion and removal of nodes during the clusters live operation, the whole system was capable of handling the gained or lost capacity with ease. Thus our system was able to process $\approx$345fps i.e. $\approx$34 parallel video streams or in terms of data volume 4.8 TB/s.

X. Conclusion

In this paper we present a different approach to frameworks for cluster computing, instead of relying on existing communication systems (e.g. MPI) we developed a new communication structure with respect to massive scaling and low time/space complexity. Furthermore we designed the enveloping framework to be as adaptable as possible, not only to allow its use on heterogeneous hardware configurations but also to allow developers to implement their own (maybe completely different) components in an object-oriented way (e.g. Map&Reduce [20] algorithms). The modular structure and simplistic design enables its use on very basic hardware configurations.

We have shown that our approach to network communication scales according to the utilized datastructure's worst time complexity. Our implementation was tested on a small Beowulf cluster and showed its flexibility as well as efficiency on a Big-Data problem. Furthermore our results showed that with an adequate load balancing policy our framework gives rise to a linear scaling of computation power. Yet in order to truly assess SimpleHydras network efficiency (in terms of latency in e.g. computation tasks with varying communication density and a large number, e.g. >1000, of connections), further testing is required and will be our next step in its development process. Additionally the efficiency relation between worker thread count and active connections must be analyzed more precisely. Even as we only used

a low amount of connections, the system shows promising potential regarding the communication latencies, which were effectively masked by those of the GigE interface of $\approx$ 40-50μs (i.e. they were smaller by at least a magnitude).
Although we provide yet another framework for Beowulf clusters, SimpleHydra is self-adaptive with respect to the utilized hardware configuration and minimizes the use of external libraries as far as possible (i.e. only libpthread in the basic setup for pure CPU based computation). It can be easily deployed in an existing evironment and provides generic management functionality for the cluster. No restrictions have to be made for the cluster topology and even a highly dynamic environment (insertion and removal of nodes during computation) is handled with ease. This is especially useful for coping with malfunctioning nodes and their live replacement. Beyond that we provide a rich toolbox of visualization methods and system tools (e.g. thread pools, time functions or system information gathering).
Even though it is still in development, we plan on making our framework available under the GPL license type. So far we restricted most of our tests to Linux systems, but Unix based systems like FreeBSD will be supported as well (e.g. using kqueue instead of epoll).

References

[1] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *In Proceedings of the 24th International Conference on Parallel Processing*. CRC Press, 1995, pp. 11–14.

[2] J. Dubinski, R. Humble, C. Loken, U.-L. Pen, and P. Martin, "Mckenzie: A teraflops linux beowulf cluster for computational astrophysics," in *Proc. of the 17th Annual International Symposium on High Performance Computing Systems and Applications*, 2003.

[3] J. D. Grant, R. Dunbrack, F. J. Manion, and M. F. Ochs, "Beoblast: distributed blast and psi-blast on a beowulf cluster," *Bioinformatics*, vol. 18, no. 5, pp. 765–766, 2002.

[4] J. Adams and D. Vos, "Small-college supercomputing: Building a beowulf cluster at a comprehensive college," *SIGCSE Bull.*, vol. 34, no. 1, pp. 411–415, Feb. 2002. [Online]. Available: http://doi.acm.org/10.1145/563517.563498

[5] V. S. Sunderam, "Pvm: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, 1990.

[6] "Mpi: A message passing interface," in *Supercomputing '93. Proceedings*, Nov 1993, pp. 878–883.

[7] C. Di Biagio, G. Pennella, E. De Paoli, R. Grandi, and F. Giammarino, "Pvm advanced load balancing in industrial environment," in *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, Feb 2006, pp. 5 pp.–.

[8] C. Di Napoli, M. Giordano, M. Furnari, and F. Vitobello, "Pvm application-level tuning over atm," in *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, 2000, pp. 391–397.

[9] P. Uthayopas, S. Paisitbenchapol, T. Angskun, and J. Maneesilp, "System management framework and tools for beowulf cluster," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol. 2, May 2000, pp. 935–940 vol.2.

[10] P. Uthayopas, S. Phatanapherom, T. Angskun, and S. Sripray-oonsakul, "Sce: A fully integrated software tool for beowulf cluster system," in *Proceedings of Linux Clusters: the HPC Revolution.* Citeseer, 2001, pp. 25–27.

[11] S. Hommel, D. Malysiak, and U. Handmann, "Model of human clothes based on saliency maps," in *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, Nov 2013, pp. 551–556.

[12] H. Jin, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, 1st ed., R. Buyya and T. Cortes, Eds. New York, NY, USA: John Wiley & Sons, Inc., 2001.

[13] H. A. Council, "Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing," HPC Advisory Council, Tech. Rep., 2009.

[14] W. R. Stevens, *UNIX Network Programming: Networking APIs: Sockets and XTI*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[15] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms," in *In Proceedings of 6th Annual Linux Symposium*, 2004.

[16] L. Wang and Y. Lu, "Power-efficient workload distribution for virtualized server clusters," in *High Performance Computing (HiPC), 2010 International Conference on*, Dec 2010, pp. 1–10.

[17] T.-Y. Liang, Y.-T. Liu, C.-K. Shieh, and C.-Y. Wu, "A new approach to distribute program workload on software dsm clusters," in *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, May 2004, pp. 201–206.

[18] L. Cherkasova and M. Karlsson, "Scalable web server cluster design with workload-aware request distribution strategy ward," in *Advanced Issues of E-Commerce and Web-Based Information Systems, WECWIS 2001, Third International Workshop on.*, 2001, pp. 212–221.

[19] D. Malysiak and U. Handmann, "An algorithmic skeleton for massively parallelized mean shift computation with applications to gpu architectures," in *Computational Intelligence and Informatics (CINTI), 2014 IEEE 15th International Symposium on*, Nov 2014.

[20] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.