HOCHSCHULE RUHR WEST
UNIVERSITY OF APPLIED SCIENCES

INSTITUT **INFORMATIK**

Technical Report 16-01

SimpleHydra

Darius Malysiak

Angela Lausberg

Uwe Handmann

SimpleHydra

HRW Technical Report

Darius Malysiak

Angela Lausberg

Uwe Handmann

Contents

Chapter

Abstract

Abstract

This internal report discusses the theoretical and practical aspects of the cluster management framework *SimpleHydra*, which was developed in order to allow researchers the quick setup of classical small to mid-scale computation clusters while being as lightweight and platform independent as possible. We motivate crucial design choices with a theoretical analysis in the aspect of time and space complexity, furthermore we give a comprehensive introduction regarding the frameworks usage (which includes examples and detailed description of fundamental concepts as well as data structures). In addition to that we illustrate application scenarios with complete source code examples. Furthermore we hope that this document proves valuable not only as a development report but also as a practical manual for SimpleHydra.

I. Introduction

Observing the development of CPUs as well as algorithms over the last 10 years one can clearly see the trend towards parallel structures. Modern CPUs feature up to 24 parallel and independent computation units (cores) whereas GPUs exhibit several thousand of primitive and co-dependent units (shaders). This change in architecture brings up the challenge of harnessing the provided computation power, which might require only subtle changes in already existing algorithms or a complete redesign if not the development of new entirely algorithms. Nevertheless the induced problems sparked a new fire on many fundamental algorithmic concepts, an example would be the sheer amount of published research regarding the parallelization of (training and evaluating) artificial neural networks. Another interesting research topic is the utilization of multiple system-local devices, e.g. deploying an already GPU-optimized algorithm among multiple GPUs (which for example presents the problem of high communication latencies and low memory bandwidth). Yet there is another level of abstraction above the system local parallelization, although certain system components provide an overwhelming amount of computation power, which also increases with future hardware revisions, one has to realize the incorporated limits. A single computation unit, let it be a high performance multi-GPU system, could provide about 8-16 GPUs and 96 CPU cores, which in combination with optimized algorithms would exhibit a splendid performance compared to the classical one-CPU-core approach. The next intuitive way of increasing the systems processing power would be to deploy several of such computation units. The researcher will face a new but similar set of problems when it comes to the efficient distribution of work among the available units, furthermore the aspect of fault management steps much further into the foreground, since the only way of obtaining a systems state is by incorporating monitoring capabilities into the network communication protocol. These problems will increase further in the case of heterogeneous units as one has to distribute adequate workloads.
It is hard to deny the temptation of approaching the field of distributed multi-GPU cluster systems in a scientific context, yet the researcher is faced with the challenge of setting up the infrastructure, this does not only include the acquiring of hardware but also the development of the corresponding software. In most cases a very problem-specific solution will be developed which allows e.g. a research group to commence its work, yet makes it difficult if not impossible for other researcher to utilize the groups infrastructure in a different context. The authors of this work faced this challenge within the scope of the research project “APFel” [MG14], in which a heterogeneous multi-GPU cluster has been developed

for the purpose of efficient object detection in a large multitude of videostreams. This report describes the correspondingly developed framework *SimpleHydra*, which represents a generic tool that enables its user to quickly deploy small- to mid-scale heterogeneous cluster systems with arbitrary topologies. Furthermore it provides functions with a slim interface for e.g. database management and GPU computation.
This report is segmented into three major parts, first we will describe previous work and motivate our approach, design decisions will be explained along with a theoretical analysis of the frameworks structure. The second part will introduce the reader to the frameworks elements as e.g. data structures or module structure, every concept will be illustrated with source code listings. The last part presents the explanation of how to use the framework to setup a cluster and deploy workloads within it. Future revisions of this report will include chapters and sections for other parts of SimpleHydra, yet currently we will describe only the frameworks elements which are required for a successful application in the context of cluster system,
We hope that other researchers in the field of multi-GPU clusters will benefit from our work or even build upon it.

Labor omnia vincit

II. Theory

In this section we will discuss the frameworks structure, motivate our design ideas with with previous works and a detailed analysis in terms of space and time complexity.

II.1. Motivation 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 parallelizable 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 [SBS+95] or related approaches. Examples are [DHL+03], [GDMO02] or [AV02], 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 [Sun90] or MPI [12693], 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. [DBPDP+06](enhanced PVM load balancing) or [DNGFV00](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 [UPAM00] 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 [UPAS01] 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 omitted.
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.2 will outline the general structure of our system, while the critical aspect of network communication will be addressed in section II.3. The previously mentioned dynamic topologies will be explained in section II.4. We will conclude this paper with the description of the IGOR cluster which was utilized in the APFel research project [HMH13] 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.2. A coarse look on the structure

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

- Core
- Network
- Cluster

The 'Core'-module provides all basic data structures and management functions for the remaining elements, e.g. file system 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 II.3. 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)

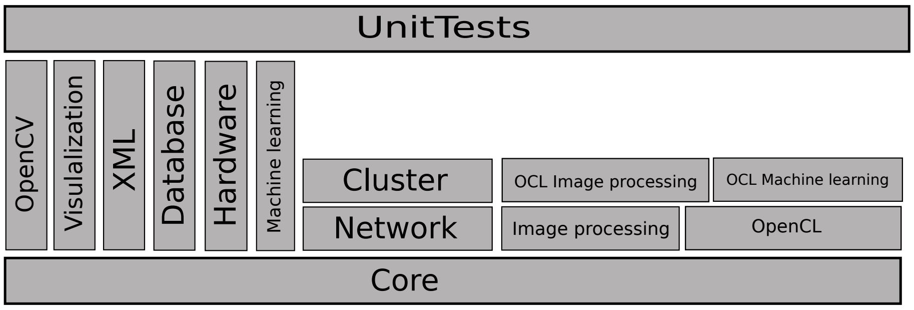


Figure II.1.: The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

- hardware support for video input devices (e.g. V4L devices, AVT cameras)
- data visualization (e.g. video streams, images or functions)

Fig. II.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 RaspberryPi 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,<o>libX11, 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.

II.3. 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 [Jin01] and drive their data with TCP. Infiniband also has a much smaller latency of $\approx 1.7\mu s$ compared to GigE ethernet $\approx 48\mu s$ [Cou09]. 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 [SBS+95] and initially described a set of Ethernet-connected workstations, whose communication based on a token exchange via UDP. Yet UDP is a stateless 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 / 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.

II.3.1. 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

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

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.

II.3.2. 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 unfeasible 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. II.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.

II.3.3. 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

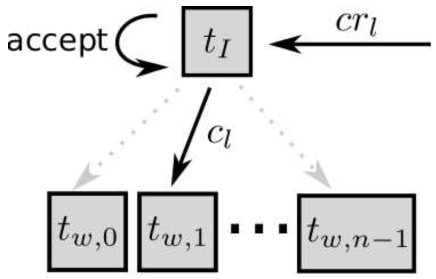


Figure II.2.: The concept of binned worker threads; one thread t_I handles the incoming connection requests cr_l, creates the connection c_j and assigns it to an appropriate worker thread $t_{w,i}$ (bin)

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* [Ste97]). 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 [GBSP04], 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. 4. 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 unfeasible 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*

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

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

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

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 pre-allocated 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

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

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

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 data structure routines (i.e. d, a, g). One can also deduct that for a constant time complexity within the described threading concept, all of the data structures operations must be able to finish in constant average time.

II.3.4. 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 data stream, thus we allocate memory only if it is required (depicted on the left image in Fig. II.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.II.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

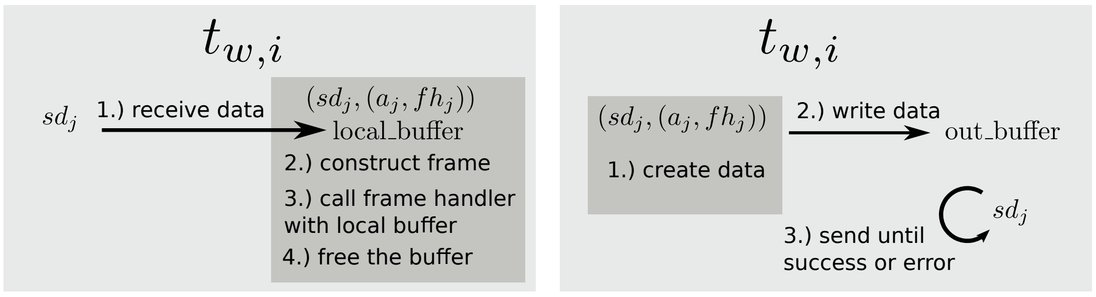


Figure II.3.: Left side: $t_{w,i}$ receives a data fragment within an iteration and directs them to the appropriate assembler a_j, which stores it in a local buffer and continues with frame reconstruction. Once a frame has been completely received, the framehandler fh_j will commence the interpretation 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.

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 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 communication 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*

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

Proof. Follows directly from lemma 1 and the corresponding remarks. □

II.4. 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).

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
```

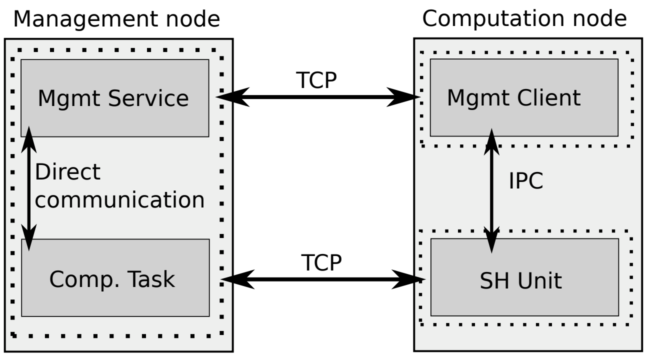


Figure II.4.: The canonical network service structure of SimpleHydra, the dashed rectangles represent different address spaces. Management of the computation nodes is done via a TCP connection between two corresponding services. These services are independent of the actual computation task but can communicate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

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. II.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 resources, 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 II.3.

II.4.1. 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. II.5, for the sake of understanding we omitted 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.

II.5. 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

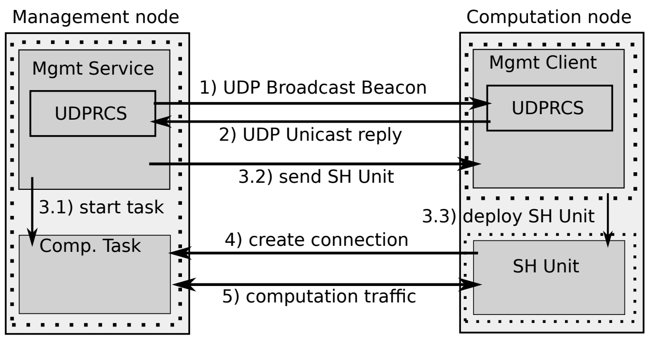


Figure II.5.: The process of self-configuring clusters with SimpleHydra. The numbers inside the annotations denote the order of execution. First the management node attempts to find all available nodes on the local subnet via a UDP broadcast. The available nodes reply to the beacon and extract the servers 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 omitted.

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.

II.6. 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.

II.6.1. SH units

An SH Unit is a data structure $u = (\mathcal{P}, d)$, 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 $d$     $\to$ 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 II.4 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).

II.6.2. Load balancing

During runtime it is crucial to distribute the workload adequately among the nodes. Factors may be power efficiency [WL10], memory attributes [LLSW04] or domain specific optimizations [CK01], 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 (II.6). The computation task may utilize this information for an arbitrary scheduling algorithm.

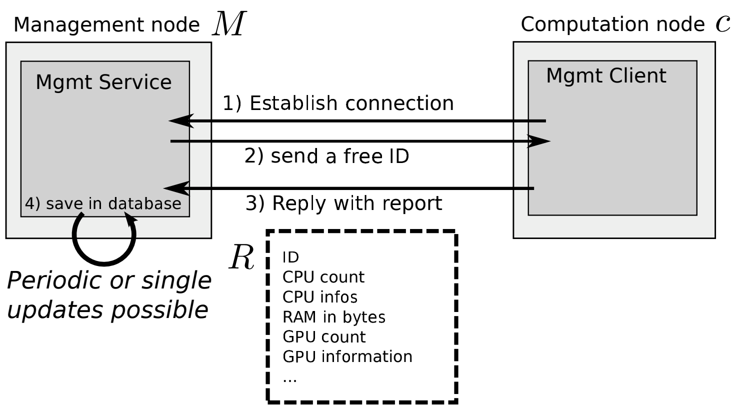


Figure II.6.: The simplified registration process of a new node c and the management node M. First c establishes the connection to M, which in turn sends a free ID (this number is removed from the pool of available IDs until c disconnects or the registration fails). The computation node then acknowledges 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 possible to update the report data periodically (i.e. c sends periodic updates) or only per request (i.e. M requests an update from c).

II.7. 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 meta data) and e.g. decide whether a node is capable of executing a certain operation. The details of collecting meta data is depicted in Fig. II.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 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. II.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

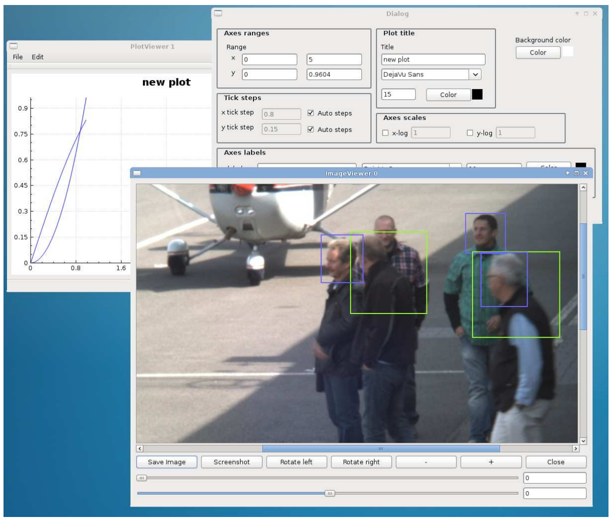


Figure II.7.: The window system is part of the visualization module, it contains e.g. sophisticated plot features, image viewers with annotation functions, video viewers and cluster management tools like e.g. parallel remote shells.

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 message 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.

II.8. 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 ([SBS$^+$95] 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 (pre-built) client demon to the workstation (if no pre-built demon is available, one has to compile it either on the workstation or with a fitting cross-compiler). 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.

II.9. Application Example and Performance Indications

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.

II.9.1. The Beowulf cluster IGOR

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

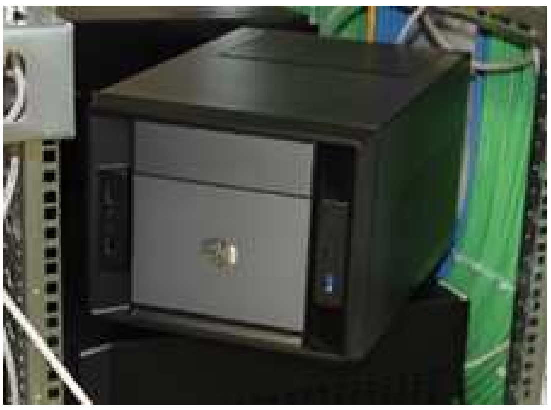
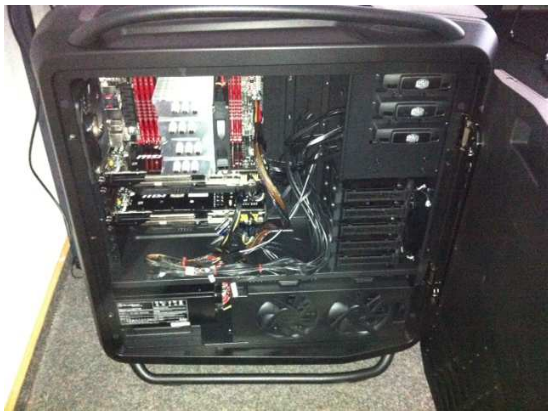



Figure II.8.: The two node types used in IGOR. The left picture shows very compact mini-ITX case with one discrete GPU, 16GB RAM and one HDD of 1TB. The right side depicts a large ATX tower with 64GB RAM, multiple GPUs and HDDs (8TB).

- 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. II.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.

II.9.2. The APFel person detection system

Our detection system (Fig II.9) is described in [HMH13] with more detail, for this paper we will give a short summary and discuss our testing methods.

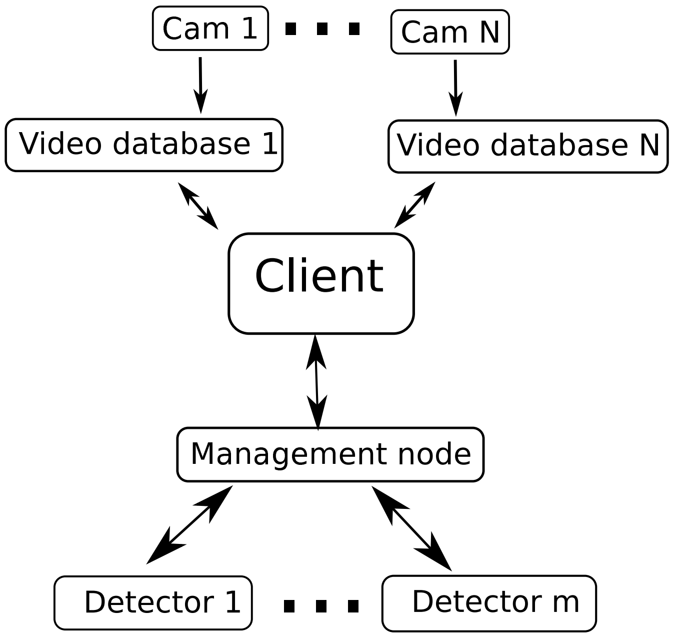


Figure II.9.: The detection system of the APFel project. The client obtains images from different video databases, he uses the management node of the Beowulf cluster as an abstraction proxy to the collected computation power of all m GPUs (which are distributed over n) nodes. The management node receives 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.

Two problem instances have to be distinguished; the classifier training and the case of detection tasks.
The detection algorithm utilizes a linear GPU-based support vector machine (SVM), 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. II.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.

II.9.3. 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 [HMH13],[MH14] 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 $\approx$ 45μs while transferring 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.
The systems structure (i.e. the distributed camera system from Fig. II.9) also illustrates SimpleHydra's adaptation potential, since due to the decoupling between the management system and the distributed workload one can easily develop custom solutions and control their deployment.

III. Data structures

In order to use the SH framework one has to be familiar with the provided data structures, the following sections will provide a coarse introduction, for further details one should refer to the API documentation. All data structures reside in the namespace "SHCore".

III.1. Buffers

Buffers represent the most primitive form of containers, they are essentially augmented wrapper classes for unsigned-char arrays. There are a total of three buffer classes; BasicBuffer, PrimitiveBuffer and Buffer. The class BasicBuffer represents the parent class for all other buffer classes, it never gives upownership over the internal char-array, i.e. it manages the memory internally.

```
//an empty buffer object, its size can be changed through 'resize()'
SHCore::BasicBuffer b1;
SHCore::BasicBuffer b2(40);//a buffer of 40 bytes
```

A buffer object does not care about the actual data types within the internal char-array, a pointer to the actual data array can be obtained via

```
unsigned char* data = b2.getmp_data();
```

This pointer should never be freed externally, its main use is to copy data into/from the buffer object. Once the object has been destroyed any allocated memory will be freed as well. The objects data can be written to or read from file via

```
void writeBufferToBinaryFile(const std::string& filename) const;
void readBufferFromBinaryFile(const std::string& filename);
void writeBufferToTextFile(const std::string& filename) const;
void readBufferFromTextFile(const std::string& filename);
```

The binary methods will write the char array as is into the given file, any previous content will be deleted. The text methods on the other hand will write the data into a text file. If data is read from a file (binary or text) any previously allocated internal data will be freed and replaced with the files content. A buffer file does not contain any header information. In some cases it might come in handy that an existing char-array can be wrapped into a buffer object in order to prevent unnecessary memcpy calls. For such situations one should use the class “PrimitiveBuffer”, which is identical to the “BasicBuffer” with the subtle difference that it can be assigned an existing array of data.

```
unsigned char a[40];
SHCore::PrimitiveBuffer b1(a,40);//a buffer of 40 bytes
```

The assigned data will never be owned by the object, i.e. once the object is destroyed the data will be left untouched. This way one might quickly provide data to certain methods without giving up control over the object, an example would be the cryptographic methods in the Core module. Yet this is only partially true, the assigned char-array can be manipulated via the parent class’s methods (i.e. “resize()”) will delete the assigned memory (this would be a fatal for the stack allocated data in the previous listing).
The class “Buffer” extends the PrimitiveBuffer by a method to change the assigned data anytime.

```
unsigned char* a = new unsigned char[40];
SHCore::Buffer b1(30);//a buffer of 30 bytes
SHCore::Buffer b2(a,40,true);//deep copy of a
b1.set_mp_data(a,40);//takes ownership
```

It must be pointed out that the Buffer will always take ownership over the data, furthermore in the case of a deep copy one does not give up ownership but the copy operation might be unnecessary.

III.2. Lists

SH provides several templated types of lists; non-cyclic singly linked lists (NCSLists) and non-cyclic doubly linked ones (NCDLists). The prefix “Fast” indicates a non-thread safe class, e.g. “FastNCSList”. In this section we will discuss the principles behind each list type, for further information one should refer to the API documentation.

III.2.1. NCSList

A (thread-safe) non-cyclic linked list is composed of a chain of “GenNCSListElement<T>” which have the following (simplified) form

```
GenNCSListElement(unsigned long id = 0);
virtual ~GenNCSListElement();
inline void setm_data(const T& data);
inline T getm_data() const;
inline T& getm_data_ref();
inline unsigned long getID();

private:
```

```
unsigned long m_id;
T m_data;
GenNCSListElement<T>* mp_neighbor;
```

The instances of elements will be managed by the corresponding List object, i.e. when the list is destroyed all element instances will be destroyed as well. This however does not necessarily apply for the each elements' assigned data. If the element contains non-pointer types <T> the data will be deleted with the list element. For pointer types the data will be left untouched. Yet each list class provides two memory management methods

```
/*clears the list and deletes all pointed to objects
in case of pointers*/
void destroy();
/*clears the list and deletes all pointed to array objects
in case of pointers*/
void destroyArrays();
/*just clears the list*/
void clear();
```

which can be used to free the list, in case of pointers the corresponding data will be destroyed as well. If a "destroy" method is called in case of non-pointer data, the methods behavior is identical to just calling "clear()". A final remark regarding the memory management of a list; if a list object is copied, all list elements will be copied sequentially, which can introduce significant overhead for large lists. An example for instantiating a list object is

```
SHCore::NCSList<int> l;
l.appendLast(5);//add a number
//print it
printf("%d\n",l.getListElementAtPos(0));
```

The attribute "mp_neighbor" points to the next element in the list and is managed by the list object, the last list element will point to NULL. Accessing list elements data (i.e. the data of type T contained within the list elements) can be done by calling "getListElementAtPos" or through iterators (see the next sections). The call to "getListElementAtPos" involves relative high overhead as it internally traverses the list up to the desired position (the same holds for its operator "[]", which will return a *reference to the data*). By reading the simplified view one might have observed that the list elements contain an ID attribute. This integer is a number unique among the list elements, the standard enumeration algorithm uses a sequential numbering of each added list element, i.e. it does not analyze for usable ID gaps among the existing list elements. This approach is fast yet has the significant drawback of possible duplicate IDs if to many remove and append operations have been commenced. The lists behavior is unaffected by duplicated IDs! Yet for external uses one might need a more reliable ID management, thus every list class provides a collision free ID management, where collision-free random IDs will be assigned to each object. If a collision can't be avoided, the list element will receive the ID 0. An unavoidable collision is defined if after 500 attempts no free random ID could be found, one should note that this approach might involve increasing overhead for data insertion as the list grows (thus the default is to use simple IDs). The list constructor expects a boolean value for indicating the ID generation algorithm ("true" for simple IDs, "false" for complex ones).

```
NCSList(bool simple_ids = true);
NCSList(const NCSList<T>& list ,bool simple_ids=true);
```

If a user doesn't require any type of identification for list elements, he can safely ignore the ID generation and leave the boolean parameter at the default value.
As mentioned in the beginning of this section every list without a "Fast" prefix is implicitly thread-safe. It uses internally a read/write lock, i.e. there can be parallel read accesses but only one write access, which also prevents any concurrent read attempt. Due to the unidirectional links within the list, it is only possible to remove the first element, which can be done via

```
void removeFirst();
```

III.2.2. NCDList

The major drawback of an singly linked list is the relatively time consuming task of removing elements at positions "within" the list. Even if one has access to the designated element it can not simply be removed as its predecessor is unknown. The non-cyclic doubly linked list (NCDList) addresses this point by managing list elements with an additional pointer to their previous list element.

```
GenNCDListElement(unsigned long id = 0);
virtual ~GenNCDListElement();
inline void setm_data(const T& data);
inline T getm_data() const;
inline T& getm_data_ref();
inline unsigned long getID();

private:
unsigned long m_id;
T m_data;
GenNCDListElement<T>* mp_successor;
GenNCDListElement<T>* mp_predecessor;
```

The interface is an overset of NCSList, new methods are

```
void removeLast();
enum GenNCDList<T>::NCD_ERROR_CODE
```

```
  removeElement(GenNCDListElement<T>* element);
enum GenNCDList<T>::NCD_ERROR_CODE
  removeElementByPosition(unsigned long position);
```

which enable one to remove elements at arbitrary positions. *removeElementByPosition* uses a sequential search as described in the case of NCSList, while *removeElement* directly removes the element. The last method requires the element to be known/available in advance, obtaining elements can be done via iterators, a manual sequential search after obtaining a NCSLists first or last element via

```
GenNCDListElement<T>* getFirst();
GenNCDListElement<T>* getLast();
```

or simply by storing/managing references to the list elements externally.

III.2.3. List Iterators

In order to iterate over a lists elements and thus over its data one can use iterators, e.g.

```
SHCore::FastNCSList<int> data;
data.appendLast(50);

for(SHCore::FastNCSList<int>::Iterator it = data.getStart();
    it != data.getEnd(); ++it)
{
  GenNCSListElement<int>* element = it.getElement();
  int value = element->getm_data();
  printf("value %d\n",value);
}
```

the same holds for NCDLists which also allow reverse traversing

```
SHCore::FastNCDList<int> data;
data.appendLast(50);

for(SHCore::FastNCSList<int>::ReverseIterator it = data.getEnd();
    it != data.getStart(); --it)
{
  GenNCDListElement* element = it.getElement();
  int value = element->getm_data();
  printf("value %d\n",value);
}
```

III.3. KTuples

A *KTuple* is a data structure very similar to std::vector, yet it provides several new features. A KTuple is thread safe and applies the same read/write paradigm as the list structures, it features control over the expansion factor in case “stack-usage” and methods for control over freeing heap allocated data. A simple example is

```
/*a tuple with 50 allocated slots*/
SHCore::KTuple<int> dataArray(50);

/*increases the tuple by current_size*growth_factor and puts
'-1' into the 51 slot*/
dataArray.push(-1);

/*update the 17th element*/
dataArray.setElement(16,-5);

/*update the 16th element*/
dataArray[15] = -10;

/*prints '0'!*/
printf("data %d\n",dataArray.getElement(0));

/*prints '-1'!*/
printf("data %d\n",dataArray.getElement(50));
```

One very important fact about KTuples is that the internal data array will be initialized via the data types (i.e. T’s) default “constructor”! In case of primitive data types it will default to ’0’ while for complex data types it will always call the default constructor. Iterating over a KTuple is straight forward

```
/*a tuple with 50 allocated slots*/
SHCore::KTuple<int> dataArray(50);

for(long long i = 0;i<dataArray.getm_size();++i)
{
  /*prints '0'!*/
  printf("data %d\n",dataArray.getElement(i));
}
```

After the discussion of data management in the context of lists, the semantics of following methods should be clear by their name

```
/*will resize the tuple to size if 'size!=-1'*/
void clear(long long int size = -1);
void eraseWithDestructor();
void eraseArrayWithDestructor();
```

Similar to the lists a KTuple returns references to the data via the "[]" operator. If a KTuple is initialized with a size $i > 0$, the stack pointer (i.e. the pointer which indicates the position where after a call to "push" a new element will be inserted) will point onto the first position behind the i-th element. Only calls to "push" allow an expansion of the tuple size.

III.4. Vectors

Despite a similar name, SHCore::VectorN does not have much in common with std::vector. The suffix 'N' stands for a wildcard $N \in \{1, ..., 8\}$ or in other words; there exist a total of 8 Vector and FastVector classes, e.g. Vector1, Vector5 etc. A Vector class is a template container with N slots of independent data types $T_1...T_8$. An example would be

```
SHCore::Vector3<int,float,std::string> v;
v.setElement1(4);
v.setElement2(7.89f);
v.setElement3("testing");

printf("%d %f %s\n",v.getElement1(),
  v.getElement2(), v.getElement3().c_str());
```

As before the thread safety is created via read/write locks, FastVectorN classes do not use getter or setter methods and expose their members for public access. Due to thread safety considerations all returned values are copies of the original data. For applications in linear algebra one should use FastVectorN classes, which provide template specializations for int, float and double. These specializations allow for e.g. int

```
/*subtract two vectors of same type*/
FastVectorN<int,...,int> operator +(FastVectorN<int,...,int>& b)
/*add b to each element of the vector*/
FastVectorN<int,...,int> operator +(int b)
/*subtract to vectors of same type*/
FastVectorN<int,...,int> operator -(FastVectorN<int,...,int>& b)
/*subtract b from each element of the vector*/
FastVectorN<int,...,int> operator -(int b)
/*get the scalar product of both vectors*/
int operator *(FastVectorN<int,...,int>& b)
/*multiply each vector element with b*/
FastVectorN<int,...,int> operator *(int b)
```

For the sake of completion we show an example for FastVectorN

```
SHCore::Vector3<int,float,std::string> v;
v.m_data1 = 4;
v.m_data2 = 7.89f;
v.m_data3 = "testing;
```

```

printf("%d %f %s\n",v.m_data1,
  v.m_data2, v.m_data3.c_str());
```

III.5. Queues

The *Queue* class in SH provides a versatile thread safe queue with a slim interface, its design is best explained through a simple listing

```
Queue<int> queue(50);

for(unsigned int i=0;i<40;i++)
{
  queue->enqueue(i);
}

//get the data
for(unsigned int i=0;i<40+5;i++)
{
  printf("%d ",queue->dequeue());
}
printf("\n");
```

The constructor parameter represents the maximal capacity of the queue i.e. one can not enqueue more elements than the capacity allows. It usually poses no problem to call *dequeue* on an empty queue since this method will return a default value of template type T. Yet it is often desired to ensure that a valid (i.e. previously enqueued) element is obtained from the queue, for this purpose one can refer to the following member functions

```
T dequeueBlockingOnEmpty();
T dequeue(int& res);
```

The first method will wait if the queue is empty, once an element becomes available it **migh** (in case of multiple consumers) return this element. It is important to understand that in case of multiple threads which are all calling *dequeueBlockingOnEmpty*, the thread which successfully dequeues the element will be determined randomly. But even this approach might not be suitable for all scenarios, let us consider the situation in which two threads operate on the same queue, one enqueues elements while the other thread consumes these elements. If the consumer also needs to process other things besides the elements from the queue it would be a waste of time to wait until a new queue item becomes available. A simple strategy to solve this problem is to check if the value which is returned from a dequeue call actually represents a previously enqueued element. The method *dequeue(int& res)* provides this functionality, the integer *res* will contain 0 if the value was previously enqueued or -1 otherwise. Once a queue is destroyed it will also delete all enqueued elements, in case of pointers the corresponding objects will be left untouched.

III.6. Matrices

SimpleHydra provides slim and efficient 2D/3D matrix classes with automatic type recognition, row- and column-major alignment, pinned memory support and support for CSV/binary export. As the matrix types play a very import role in OpenCL, CUDA, cluster computation, linear algebra and many more fields, we will discuss its features in more detail.

```
/*or MATRIX_DATA_ALIGNMENT_COLUMN_MAJOR*/
SHCore::Matrix2<float>
m(SHCore::Matrix::MATRIX_DATA_ALIGNMENT_ROW_MAJOR,
   false);
```

allocates a 2 dimensional matrix in column-major format and no pinned memory, the shown values are identical with the default ones, i.e. the constructor call from above is equivalent with

```
SHCore::Matrix2<float> m;
```

If one instantiates a matrix object no memory will be allocated, this is done via

```
m.initMatrix(rows,columns);
```

In case of multiple “initMatrix” calls, any previously allocated memory will be deleted (this does not delete objects in case of pointers, only the internal matrix space which contains the pointers). Any copy constructor or assignment operator will perform a sequential copy of all matrix elements. After initializing a matrix one can start working with the matrix object, yet although memory has been allocated now, the matrix elements (or slots) themself are undefined. A call to *resetMatrix* will set all internal memory bytewise to zero. If one desires a specific value, the method *globalSetMatrix(unsigned char val)* can be used, which essentially does the same as *resetMatrix* but with a different value. For complex or general data types T one can use *globalSetMatrixVal(T val)* which also does the same as the previous methods but for complex values. In case of debug situation it can be beneficial to peek into a (small) matrix, which is accomplished through the method *printData*, as it prints the matrix data on the CLI (complex data types will be identified by the static output string ’ABSTRACT’). Setting the actual matrix data can be done on different ways

```
/*update entry in row 1 and column 3*/
m.setData(0,2,74.123f);
```

```
/*update entry in row 2 and column 3*/
m(1,2) = 74.123f;
/*update entry in row 3 and column 3*/
*(m.getDataRef(2,2)) = 74.123f;
/*update entry in row 12 and column 3*/
m.getRawDataRow(11)[2] = 74.123f;
/*update entry in row 13 and column 3.
THIS ONLY WORKS IN COLUMN MAJOR FORMAT!
*/
m.getRawDataColumn(2)[12] = 74.123f;
/*update entry in row 15 and column 3*/
m.getDataPtr[ ELEMENT_INDEX ] = 74.123f;
```

The matrix size can be obtained via *getRowCount* and *getColumnCount*, it must be noted that the matrix classes are not thread safe!
Matrices can be saved to text or binary files via

```
/*CSV export*/
void writeMatrixToFile(const std::string& filename);
/*Binary export*/
void writeMatrixToBinaryFile(const std::string& filename);
/*CSV export without header*/
void exportMatrix(const std::string& filename);
/*Binary export without header*/
void exportMatrixBinary(const std::string& filename);
```

For reasons of numerical precision one should always consider saving the matrix in a binary format. The CSV format is as follows: in case of a header the first line will contain the string '#rows,#columns' followed by one line per row or column in case of row-major or column-major, respectively. The value will be saved as "double-strings" (this must be and will be adapted in the future to be simple type specific). All stringified double values are saved according to the standard "C" locale. Binary files follow a similar concept, i.e. one header line as before, followed by binary data in the next line. Calling a CSV export method in case of abstract data types will result in an error message with no file written, only binary export methods can be used in this case. The following routines can be used to read a matrix from a CSV/binary file

```
/*CSV import*/
void readMatrixFromFile(const std::string& filename);
/*Binary import*/
void readMatrixFromBinaryFile(const std::string& filename);
```

The correct datatype has to be set a-priori during object construction. The data alignment can be queried via

```
enum MATRIX_DATA_ALIGNMENT getDataAlignment() const;
```

and the data type with

```
enum Toolbox::SH_DATA_TYPE getDataType() const;
```

A 3 dimensional matrix is nearly identical with a 2 dimensional, except for the one additional dimension. The resulting differences are a third index k for all addressing operations, which is interpreted as the index of a matrix slice / matrix level, each slice in turn is handled as a previously discussed 2 dimensional matrix. A 3 dimensional matrix is saved slice by slice, i.e. a series of 2 dimensional matrices. The header contains '#rows,#columns,#levels' and is followed in the next line by concatenated 2 dimensional matrix slices.

III.7. Images

The SH framework supports a convenient and thread safe image structure in form of a class called "Image". Yet in case of image processing one might require fast manipulation methods for e.g. pixel values, instead of providing a non-thread safe class e.g. "FastImage" we provide alternative "Fast" methods within the class. An image can be created via

```
 Image<unsigned char>* im =
  new Image<unsigned char>(320,240,
     Image<unsigned char>::IMAGE_DATA_ALIGNMENT_RGB);
```

Where the first two constructor parameters determine the image dimensions while the third parameter determines the data alignment. Currently SH only supports interleaved alignments; RGB(A), BGR(A) and GRAY(A). The template value determines the datatype for each channel, in the example above we create an image with three channels, each of type *unsigned char*. The image class features a rather large set of methods, in this section we will only focus on the most common tasks when it comes to image objects, the reader should refer to the Doxygen documentation for more information.
Pixels in an image can be accessed through channel tuples

```
KTuple<unsigned char> color(3);
color.setElement(0,255);
color.setElement(1,0);
color.setElement(2,255);
im->fillImage(&color);
```

or the corresponding non thread safe method

```
FastKTuple<unsigned char> color(3);
color.setElement(0,255);
color.setElement(1,0);
color.setElement(2,255);
im->fillImageFast(&color);
```

For highly frequent pixel operations, e.g. setting a subset of pixels with a loop, one should use the non thread safe methods as they perform significantly faster. The previous example showed how to fill an image with a single color, it is of course possible to set individual pixels or even manipulate the raw data

```
FastKTuple<unsigned char> color(3);
for(unsigned int i=0;i< im->getm_width();i++)
{
  for(unsigned int j=0;j < im->getm_height();j++)
  {
      im->getPixelFast(&color,i,j);
      color->setElement(1,0);//delete green info
      im->setPixelFast(&color,i,j);
  }
}
//obtain a reference to the raw data
unsigned char* data = im->getmp_image_data();
```

Reading and writing images is done through functions provided by the image processing module, SH currently supports the image formats JPEG and PNG, future versions will also support BMP files. As this report does not feature a description of the image processing module, we will briefly show how to read and write image files in case the mentioned module has been compiled.
If the *ImageProcessing* module is available, it is possible to read images via reader/writer objects, i.e. a single class allows reading and writing of certain image file formats. This shows how to read a JPEG image,

```
JPEGReaderWriter<unsigned char>* jpeg_reader =
  new JPEGReaderWriter<unsigned char>();

jpeg_reader->readImage("image.jpeg",
  SHCore::Image<unsigned char>::IMAGE_DATA_ALIGNMENT_RGBA);

SHCore::Image<unsigned char>* image = jpeg_reader->getmp_image();
```

the call to *getmp_image* returns a reference to the internal image object, yet this does not transfer ownership, the image is still owner by the reader. In order to take ownership one has to use *obtain_mp_image*. Besides the image filename, the readImage method also accepts the definition of data alignment for the read image (the default is RGB). The general rule for allowed alignments is that no channel reduction may occur, e.g. if an image contains three channels one can not request the GRAY alignment, the read method will abort the reading in such cases. Yet it is possible to increase the channel count, e.g. a single channel image can be read as an RGB image.
Writing an image can be done by calling the writeImage method on the reader/writer object

```
jpeg_reader->setmp_image(some_image);
jpeg_reader->writeImage("image2.jpeg",10);
```

This will update the internal image object (transferring ownership) and save the data as “image2.jpeg” with a quality of 10 (highest possible value for JPEG).

IV. Core module

IV.1. Serialization

We will distinguish two kinds of serialization procedures; serializing a serializable object and λ serialization.

IV.1.1. Serializable Objects

In order for an object to be serializable it has to extend the class *Serializable* and implement the methods *serialize* as well as *deserialize*. Our following example features a very small data class

```
class MyData : public Serializable
{
  MyData(){ m_a=0; m_b=0; m_s="Hello"; }
  virtual ~MyData();

  void serialize(){

  //calculate the required size of the buffer
  unsigned long total_serialized_data_size =
    getRequiredStringBufferSize(m_s)
    +Serializable::INT_SIZE
    +Serializable::DOUBLE_SIZE
    ;

  //allocate the serialization buffer
  DELETE_NULL_CHECKING(mp_serialized_data_buffer);
  resetSerializationOffset();
  this->mp_serialized_data_buffer =
    new SHCore::Buffer(total_serialized_data_size);

  //insert the data
  addString(m_s);
  addInt(m_a);
  addDouble(m_b);

  }

  void deserialize(){

  initSerializationOffsetForDeserialization();

  //get the data
  m_b = getDouble();
  m_a = getInt();
  m_s = getString();

  //free the serialization buffer
  DELETE_NULL_CHECKING(mp_serialized_data_buffer);

  }

  int m_a;
  double m_b;
  std::string m_s;
}
```

Although it is mostly self-explanatory we will step through the process. As a first remark, it is mandatory to call *resetSerializationOffset* before starting the serialization. for reasons of safety one should also delete any existing serialization buffer. Before allocating a new serialization buffer we need to calculate the required size, for primitive types this can be easily done via a provided set of static methods from *Serializable*, e.g.*getRequiredStringBufferSize()*. Once done we allocate the buffer and update the class member *mp_serialized_data_buffer*. Regarding primitive data types it is again easy to copy the data into the data buffer (e.g. using *addString()*). We strongly advise not to use "memcpy" in combination with these helper methods (use it only in case you have enough knowledge about the stack pointer within *Serializable*)! In order to copy complex data types into the buffer one should use *addCharA()* and provide a char pointer to the data.

Derserialization is done in a similar manner, it starts with a mandatory call to *initSerializationOffsetForDeserialization*. Afterwards we simply copy the data from the **stack** (in reversed order) with helper methods like e.g. *getDouble*. It is not required to delete the serialization buffer, yet for reasons of safety we do it anyway.

For very complex objects one has to account for the serialization of all subcontainers, i.e. one has to serialize them and copy the linearized data into the main objects serialization buffer.

The serialization buffer can be accessed via *get_mp_serialized_data_buffer*, keep in mind that the buffer belongs to the serialized object, never attempt to delete it externally! The usual serialization process is similar to

```
MyData md;
md.a = 20;
md.b = 3.156;
md.s = "World";

md.serialize;

 Buffer* data = md.get_mp_serialized_data_buffer();

 //hand the data to other elements
```

while deserialization follows

```
 //acquire the serialized buffer

 MyData md;

md.set_mp_serialized_data_buffer(buffer);
md.deserialize;

 printf("Members %d %F %s\n",md.a,md.b,md.s);
```

An even shorter way would consists of

```
class MyData : public Serializable
{
  MyData(){ m_a=0; m_b=0; m_s="Hello"; }
  virtual ~MyData();

  void serialize(){

  registerString(m_s);
  registerElement<int>(1);
  registerElement<double>(1);

  allocateBuffer();

  //insert the data
  addString(m_s);
  addInt(m_a);
  addDouble(m_b);

  }

  void deserialize(){

  //get the data
  m_b = getDouble();
  m_a = getInt();
  m_s = getString();

```

```
    //free the serialization buffer
    resetBuffer();

    }

    int m_a;
    double m_b;
    std::string m_s;
}
```

this merely represents a new revision of the serialization interface, the underlying mechanics remain unchanged.

IV.1.2. λ Serialization

In certain situations one might encounter the need to serialize an existing class instance with the restriction of avoiding to extend it via *Serializable*. This is very common in network communication where a set of primitive data types must be added to a serialized data stream. For this reason SimpleHydra features the so called λ serialization or anonymous serialization, which allows the serialization of data without a wrapper class. Let us study an example;

```
    unsigned int a = 40;
    double b = 3.14;
    std::string s = "Hello";
```

Our goal is to create a buffer large enough to contain the these three variables and copy them into the same. Just as described in the previous section one could create a wrapper class which extends *Serializable*, yet if this situation only occurs a few types this approach might overkill.

```
    unsigned int a = 40;
    double b = 3.14;
    std::string s = "Hello";

    SerializationStack data;
    data.registerElement<unsigned int>(1); //just one uint
    data.registerElement<double>(1); //just one double
    data.registerString(s); //and one string
    data.sealStack();

    data.addUInt(a);
    data.addDouble(b);
    data.addString(s);

    Buffer* buffer = data.getStackBuffer();
    //do something with 'buffer'
```

The class *SerializationStack*, just as the name indicates, represents a stack whose size is calculated via calls to e.g. *registerElement*. Every such call increments the required buffer size, yet it does not allocate any memory until the call to *sealStack*. Afterwards we can add the actual data in a similar fashion as before.
Deserialization is also much alike

```
//get the serialized data

unsigned int a = 0;
double b = 0;
std::string s;

SerializationStack data;
data.setStack(buffer);

s = data.getString();
b = data.getDouble();
a = data.getUInt();
```

Please note the existence of *registerString*, *registerCString*, *registerCharA* and *registerUCharA*, which are methods of *SerializationStack* that should never be mixed. The only thing they all have in common is the fact that each method will reserve an unsigned integer slot on the stack in order to store the corresponding data types length. Never attempt to mix them, *registerString* should only be used for std::string, *registerCString* only for plain 0 terminated C-string, *registerCharA* for signed char arrays and *registerUCharA* for unsigned char arrays.

IV.2. Timing

Currently SH provides an interface to measuring time and scheduling events through system based timers. Let us first discuss the facilities for measuring time. For the most simple cases in which one desires to measure the time between two events and receive the output on the CLI, it will most likely be sufficient to use the *TIC* and *TOC* macros

```
TIC
for(int i=0;i<20;++i)
{
Toolbox::sleep_ms(500);
}
TOC
```

This example also introduces the set of sleep functions, namely *sleep_ms, sleep_us, sleep_ns* , contained within the class *Toolbox*. Due theit nature as macros, *TIC* and *TOC* can be used only once in a given context, the expansion further clarifies on that

```
//TIC
struct timespec* a = SHCore::TimeToolbox::createTick();
struct timespec* b = SHCore::TimeToolbox::createTick();
struct timespec* diff = SHCore::TimeToolbox::createTick();
clock_gettime(0, a);

//TOC
clock_gettime(0, b);
SHCore::TimeToolbox::tickDiff(a,b,diff);
printf("Process took = %.9Lf s\n",
SHCore::TimeToolbox::getTime(diff));
SHCore::TimeToolbox::freeTicks(&a,&b);
{if(diff!=0){free(diff);diff=0;}};
```

It is obvious that an attempt to use *TIC* again would be futile, the compiler would complain about variable re-declaration. In order to reuse these macros one has to augment them with *TIC_* and *TOC_*

```
TIC
for(int i=0;i<20;++i)
{
   Toolbox::sleep_ms(500);
}
TOC_
TIC_
Toolbox::sleep_ms(1100);
TOC
```

Keep in mind that a TOC is required at the end in order to free the allocated variables. Seeing the time difference in a console is a neat thing, what about getting the numerical result for further processing? This can be done equally easy with

```
 TIC
 //....
 TOC_
 long double timespan = SHCore::TimeToolbox::getTime(diff);
 TOC
```

The same holds for measuring the average time over multiple iterations

```
INIT_TIME_AVG
for(unsigned int i=0;i<1000;i++)
{
   TIC_
   //......
   AVG_TOC
}
SUM_AVG_TIME_UP
```

which also outputs the average time on the CLI. Multiple averaging in the same context is done via *INIT_TIME_AVG_* whereas the numerical value is obtained by simply dividing the summed up time through the amount of iterations

```
INIT_TIME_AVG

for(unsigned int k=0; k < eval_count; ++k)
{
  TIC_
  for(int i=0; i < size; ++i)
  {
    for(int j=0; j < size; ++j)
    {
            array[i][j] = 0;
    }
  }
  AVG_TOC
}

SUM_AVG_TIME_UP

//get the numerical value
long double result = avg_time / (long double)step_counter___;

INIT_TIME_AVG_

for(unsigned int k=0; k < eval_count; ++k)
{
  TIC_
  for(int i=0; i < size; ++i)
  {
    for(int j=0; j < size; ++j)
    {
            array[j][i] = 0;
    }
  }
  AVG_TOC
}

SUM_AVG_TIME_UP
```

In addition to local time measuring SH features a global counterpart. The corresponding functions are semantically identical to the previous ones, yet they expect an integer parameter which specifies the global timer. An example

```
GLOBAL_TIC(0)
for(int i=0;i<20;++i)
{
```

```
Toolbox::sleep_ms(500);
}
GLOBAL_TOC(0)
```

The macros for global timing start with the prefix "GLOBAL_" while following part is identical to the previous functions. Every local timing function has a global counterpart, thus we won't describe the functions in detail. Yet it is important to understand what goes on in the background, instead of using local *timespec* structs the functions depend on global ones. These are named "__GLOBAL_TIMERx_TICK_A" and "__GLOBAL_TIMERx_TICK_B" where x stands for a number from 0 to 7, thus one can utilize a total of 8 global timers. Regarding the measuring of average times SH uses global variables for each timer, these are named "__avg_timex" "__step_counterx___" with x being in the same range as before. The difference between two times is stored in global *timespec* variables named "__GLOBAL_TIMERx_DIFF".

IV.2.1. Timers

The second large area of time facilities is occupied by synchronous and asynchronous timer classes. A synchronous timer is always considered to be blocking and non-periodic, in other words once a thread starts a synchronous timer, the thread will block at the call until the timer expires, furthermore it will not restart itself. An asynchronous timer on the other hand is always non-blocking and >can< be periodic. Let us first take a look at synchronous timers

```
SHCore::SynchronousTimer sync_timer;
sync_timer.setOneShotTime(2,0);
```

This creates a synchronous timer, sets it for a single shot 2 seconds and 0 nano seconds after the start. Of course the corresponding event must be registered before the timer is started

```
SHCore::FunctionTask<void,
   SHCore::NO_STATIC_THREAD_REF___>* ft2 =
      new SHCore::FunctionTask<void,
         SHCore::NO_STATIC_THREAD_REF___>();

//prepare the notifier callback
//create the callback container
SHCore::ThreadObjectCallback<void,
   SHCore::Testcallee,void*>* o1 =
         new SHCore::ThreadObjectCallback<void,
            SHCore::Testcallee,void*>();

//assign the callee objects to the containers
o1->set_mp_callee_object(tc);
```

```

  //assign the functions to the containers
  o1->set_mp_object_method(&SHCore::Testcallee::p);

  //assign the parameters for the function
  o1->set_mp_parameter(NULL);

  //create the queue and enqueue the objects
  SHCore::PersistentCallbackQueue* c_queue =
      new SHCore::PersistentCallbackQueue();
  c_queue->enqueueCallback(o1);

  //register the queue at the function task
  ft2->registerCallbackQueue(c_queue);

  //****** register it
  sync_timer.setTask(ft2);

  //******** start it
  sync_timer.startTimer();
```

Any reader unfamiliar with SH function task should briefly skip to section IV.8 and read up on the topic! Done? Perfect! The listed code will not only assign the event but also start the timer, which in turn blocks the calling thread until the trigger time of 2 seconds elapsed and the event has been processed. In other words, the blocking time can be greatly influenced by the duration to complete the registered action. Although a synchronous timer will trigger an event only once, it can be restarted any amount of times, the trigger time can be updated between calls to *startTimer*. Keep in mind that although the function task remains registered it might need to be primed again e.g. by filling the callback queue (the above example circumvents that by using a persistent queue).
An asynchronous timer is very similar, yet besides the fact of running in a separate thread it also differs in the choice of periodic and non-periodic operation

```
SHCore::AsynchronousTimer
  async_timer(SHCore::Timer::TIMER_TYPE_SINGLE_SHOT);

async_timer.setOneShotTime(5,0);

SHCore::Testcallee* tc = new SHCore::Testcallee(1);

//create the task and register it
SHCore::FunctionTask<void,
  SHCore::NO_STATIC_THREAD_REF___>* ft1 =
    SHCore::quickDeployCallbackTaskDRef<void,
      SHCore::Testcallee,void*>( tc, &SHCore::Testcallee::p,
        NULL);

```

```
async_timer.setTask(ft1);

//start it
async_timer.startTimer();

//wait some time for the execution of the timer
SHCore::Toolbox::sleep_ms(10000);

/*thread should have stopped by now, but we try
it anyway: it should work*/
async_timer.stopTimer();
```

This creates a non-periodic timer, assigns an event to it, sets the trigger time to 5 seconds, starts it and waits 10 seconds for it to have finished. The call to *stopTimer* will cause the calling thread to join the terminating timer thread. Just as with synchronous timers any asynchronous timer can be restarted any number of times. One also has to note that the total time between trigger events is actually at least *set time* + *event time*. The periodic case can be achieved through

```
SHCore::AsynchronousTimer
   async_timer2(SHCore::Timer::TIMER_TYPE_PERIODIC);
async_timer2.setPeriodicTime(0,200000000);//200ms
async_timer2.setTask(ft1);
async_timer2.startTimer();
```

Where the time between triggered events still remains unchanged, the only difference is that all events will happen concurrently in another thread. A call to *stopTimer* will gracefully stop (and join) the thread, i.e. a the thread will terminate once the last triggered event has finished.

IV.3. Virtual Disk Operating Systems

In order to provide the developer with a quick and easy way to write NCurses-based interfaces, SimpleHydra provides the concept of virtual disk operating systems (VDOS). The complete VDOS interface is provided by only two classes, namely *VDOSCLI* and *VDOS*. The first class represents a monolithic NCurses system without any logic besides the management of CLI I/O and a multi window terminal. All operating system logic has to be implemented by extending *VDOSCLI*. Let us take a look at the following "Hello World OS"

```
//3 terminal windows and in a system called "HMI"
VDOSCLI cli(3,"HMI");
HelloOS helloOS;

vcli.setVDOS(&helloOS);
```

```
helloOS.setCLI(&clli);

vcli.startCLI();
```

with following implementation

```
class HelloOS : public VDOS
{
HelloOS(){}
virtual ~HelloOS(){}

virtual void processInput(
    unsigned int window, std::string in)
{
    std::string s = std::string("Hello World ")
      + in.substr(0,in.size()-1);
    m_cli->printResult(window,s);

    m_cli->askQuestion(window, "Satiesfied?", "Y/N")
}

virtual void processAnswer(
    unsigned int window, std::string in)
{
   std::string s = in.substr(0,in.size()-1);
   if(s.compare("Y")==0)
    m_cli->printResult(window,"Nice!");
   else
    m_cli->printResult(window,":-(");
}
}
```

Now to the details! A call to *startCLI* will start a multi window terminal in the current CLI. the user can switch between the windows via the F4 key, while pressing the F3 key will terminate the window system and the calling thread will proceed beyond *startCLI*. This does not imply that any actions executed by our VDOS *HelloOS* will immediately terminate! Any user input within a window, terminated by a press on RETURN, will result in a call to *processInput*. The method parameters are the user input (including the line break) and the associated window number. In our example the input string is concatenated with "Hello World" and returned without the trailing line break. This is done via a call to *printResult*, keep in mind that the output can also occur in a different window (see the method signature). *processAnswer* is used for HMI dialogues, after printing the described string, the input process method will ask the user a question (with hinting the desired answer types, e.g. "Y" or "N"). The user will be prompted for an answer, which in turn will be processed by *processAnswer*.
The VDOS cli features more features, which are going to be extended with future SH

versions. Currently on can only modify the prompt string via *updateCLIPrefix*.

IV.4. Directories

A common use-case is the parsing of directories and the extraction of file meta-data. In order to ease such task SH provides the classes *Directory* and *DirectoryEntry*. The rationale is that a directory instance (non-recursively) reads the file list of a given directory, for every file ("everything is file") a single DirectoryEntry object will be created. A simple example would be

```
Directory* dir;

dir.readDirectoryFilesAndDirectories("/tmp");

DirectoryEntry* dir_entry = dir.getNextFile();

while(dir_entry != NULL)
{
  dir_entry->printInfo();
  printf("\n");

  dir_entry = dir->getNextFile();
}
```

Keep in mind that this approach might lead to a serious memory overhead for large amounts of files. If only either directories or files are desired one should use the methods *readDirectoryDirectories* or *readDirectoryFiles*, respectively. One can access the files meta-data through *DirectoryEntry*s public attributes. In order to use the object for other directories or parsing tasks one can simply call a read method, any information about previous directories will then be discarded.
Another useful class is *File*, which represents an abstraction for regular files. It provides methods for reading a file's contents as well as mapping a file into local address space. Let's start with the first case

```
File f(std::string(SimpleHydra_Install_Path) +
    std::string("/UnitTests/Res/TestFile1.txt"));

f.readFile();

PrimitiveBuffer data(f.getData(),f.getSize());

data.printfBuffer(BasicBuffer::BUFFER_DATA_TYPE_UCHAR);
```

This also illustrates the use of a global and static variable called *SimpleHydra_Install_Path* which contains the installation path for SH (e.g. "/usr/local/SH"). Furthermore one

can see the use of unit-test resources which are located under "UnitTests/Res" in the installation path. In itself the listing shows how to use the File class in order to read a file's content, keep in mind that *File* internally uses an unsigned char array as a buffer. Yet *readFile* also accepts two parameters, the first is an offset from which on the file's content should be read, the second one represents the amount of bytes to read. In order to write a file one has to distinguish two cases, first the situation in which the desired file already exists and the other in which the file has not been created yet. Both cases require the use of the *writeFile* function which will overwrite the files previous content. It is also possible to assign new data to the object via a call to *assignData*, this is mainly a virtual operation as the data will not be written to the file until *writeFile* is called. A new and 0-filled file with the provided name will be created in case that the desired file doesn't exist. Its size will equal the sum of the specified offset and size.
One might need to share a files contents with e.g. another process, this can be done through a *shared* memory mapping, in which any change to the files data will be written to the corresponding file location. The File class also supports the *private* mapping, in which any change to a files content will be kept local to the process, i.e. no other process will see these changes. A mapped file will cause the writeFile method to fail. This behavior is well defined as if a file is mapped as a shared object, any change to the data will be immediately written to the corresponding disk content (there is no need for a write method). Should a file be mapped in a private manner one has to synchronize its contents with the file location through a call to *syncMap*. The File classes destructor will unmap any mapped data, in case of privately mapped data no synchronization will be done. Just as in the case of reading a file, should the desired file not exist a new one will be created and mapped. Another restriction for mapping a file lies in the nature of memory mapping, a mapped file can not be increased in size, one can only work in the existing area.
Keep in mind that reading and mapping are complementary operations, should a file object currently be mapping a file any call to *readFile* will unmap the data (in case of a private mapping no synchronization will occur), the same holds vice versa (any loaded and modified data will be discarded without disk synchronization).

IV.5. Loading Libraries during Runtime

Another use-case is the dynamic loading of shared libraries, under Linux this is done via the system calls *dlopen* etc. SH features a wrapper class called *DLLoader* which provides a reference counting for implicit unloading. This can be explained best with an example

```
typedef void ( *EXT_FUNCTION_INT ) ( int );
...
EXT_FUNCTION_INT p_function = NULL;
DLLoader loader("libz.so");
```

```
loader.load();
p_function = (EXT_FUNCTION_INT)
    loader.getElementAddress("functionXYZ");
...
loader.unload();
```

In this listing we have created a unction pointer of type *EXT_FUNCTION_INT*, the shared object loader is associated with the libz library. Each call to *load* will actually load the library (i.e. map the library into the processes address space), the function will return immediately should the library already be mapped. The address of an element within the library can be obtained via a call to *getElementAddress* (be carefull as it might return NULL), in out example we query the object for the address of *functionXYZ*. At the end a call to *unload* indicates that we are done with the library in this context. This is also the point where reference counting comes into play, each call to *load* increments an object internal counter by one, whereas a call to *unload* decrements the counter by one. Should the objects internal counter reach zero during a call of *unload* then it will unmap the library from the processes memory. The rationale is that one loader object is shared among multiple context which at the end call the unload method, thus the loaded library will be unmapped only when all contexts have finished their work with it.
One last and very important note: There is no implicit unmapping/unloading when a loader object is destroyed, thus one has ensure that a every *load* call is fitted with a corresponding *unload* call!

IV.6. Database Support

SimpleHydra features support for MySQL (external mysqlclient library is required) as well as SQLite (included as amalgamation version of SQLite), in the following sections we will talk about the individual interfaces (which are relatively short).

IV.6.1. MySQL

The MySQL interface is only available through the database module. Currently mainly two classes exist, *MySQLWrapper* and *MySQLToolbox*, future releases of SH may very likely expand the MySQL support. The usage is depicted in the following listing

```
SHDatabase::MySQLWrapper sql_wrapper;
sql_wrapper.connect("127.0.0.1","root","pw","imageDB");

SHDatabase::MySQLToolbox sql_toolbox;
FastKTuple<std::string*> attributes(3);
FastKTuple<std::string*> values(3);

```

```
attributes.setElement(0, new std::string("methodID"));
attributes.setElement(1, new std::string("cameraID"));
attributes.setElement(2, new std::string("transactionID"));

values.setElement(0, new std::string("1"));
values.setElement(1, new std::string("2"));
values.setElement(2, new std::string("3"));

std::string* s = sql_toolbox.createInsertQuery(&attributes,
  &values,"imageTable");

attributes.eraseWithDestructor();
values.eraseWithDestructor();

sql_wrapper.query( *s );

delete s;
```

The database connection is established through the wrapper class while the SQL toolbox is utilized to create the query string. One should note that the input tuples contain string pointers, this design choice is motivated by overhead reduction, especially in case of long strings one can avoid the time for string copying. From the above listing it becomes obvious that one drawback of this design choice is the high risk of memory leaks (the string objects must be explicitly deleted). Due to time constraints the described interface it all of SHs MySQL support, yet as stated before this will likely change in future SH releases.

IV.6.2. SQLite

Although mentioned before we repeat that SH contains the entire SQLite implementation, thus one can't use an external SQLite library if SH is being used. This may pose a serious drawback which can be circumvented by rebuilding SH with the option "noInternalSQLite" set to true. Yet in both cases the SQLite interface of SH will be available. We will briefly introduce the three main interface classes, *SQLiteToolbox*, *SystemDB*, *SystemDBEntry*, and conclude the section with some application examples.
If one desires to create a class for a specific database, the class should extend *SystemDB*, usually one will also create classes which can represent entries from certain tables in the database, these classes should extend *SystemDBEntry*. In order to illustrate this we will assume the following scenario; we attempt to create an object oriented wrapping for the database 'TestDB' which contains only one table named 'Table1'.

```
class Table1Entry: public SystemDBEntry {
public:
  Table1Entry();
  virtual ~Table1Entry();

  virtual void abandonData();

  int m_IDTable1;
  std::string m_string;
  long long int m_int;
  Buffer* mp_blob;
};

class TestDB: public SystemDB {
public:
  TestDB(const std::string& path);
  virtual ~TestDB();

  Table1Entry* getEntry(const std::string& s);
  int addEntry(Table1Entry* entry);
  int updateEntry(Table1Entry* entry);
  int deleteEntry(int id);

};
```

The tables structure is depicted through its public attributes, the attribute 'string' is assumed to be unique and 'IDTable1' is auto-incremented. The database class will provide a method to query an existing entry, to delete a single entry, to add a new entry and to alter an existing one. The entry classes methods are defined as

```
Table1Entry::Table1Entry
{
  m_IDTable1 = 0;
  m_int = 0;
  mp_blob = NULL;
}

Table1Entry::~Table1Entry
{
  DELETE_NULL_CHECKING(mp_blob);
}

void Table1Entry::abandonData
{
  mp_blob = NULL;
}
```

A call to *abandonData* will make the object abandon any associated data. This is useful in the following case: the objects pointer type members reference external data, yet once the object is destroyed it will attempt to delete the pointed to objects, in order to prevent any disaster one should call *abandonData* and simply set the internal pointers to NULL.

We can now take a look on the database class.

```
TestDB::TestDB(const std::string& path) : SystemDB(path)
{}

int TestDB::deleteEntry(int id)
{
//--------------- open a connection
if(this->connect() == -1)
{
        return -1;
}

std::string delete_query =
  std::string("DELETE FROM `Table1` WHERE `IDTable1`= \"")
  + Toolbox::intToString(id) +  ("\";");
sqlite3_stmt* delete_query_statement = NULL;
int res = sqlite3_prepare_v2(
  mp_handle,
  delete_query.c_str(),
  strlen(delete_query.c_str()),
  &delete_query_statement,
  NULL
);

if(res != SQLITE_OK)
{
  printf("ERROR: could not create delete statement %d\n",res);
  return -1;
}

//execute the statement
res = sqlite3_step( delete_query_statement );

//we expect only one result
if(res != SQLITE_DONE)
{
  printf("ERROR: could not execute delete statement %d\n",res);

  //clean up
  res = sqlite3_finalize(delete_query_statement);
  if(res != SQLITE_OK)
  {
    printf("ERROR: could not free delete statement %d\n",res);
  }

  return -1;
}

```

```
//clean up
res = sqlite3_finalize(delete_query_statement);
if(res != SQLITE_OK)
{
  printf("ERROR: could not free delete statement %d\n",res);
}

return 0;
}

Table1Entry* TestDB::getEntry(const std::string& s)
{
Table1Entry* entry = NULL;

//---------------------------------- open a connection
if(this->connect() == -1)
{
  return entry;
}

//-------------------------- get the rows of the table
std::string table1_query =
  std::string("SELECT * FROM `Table1` WHERE `string`= \"") +
  s + ("\" ORDER BY `rowid` ASC LIMIT 0, 50000;");
sqlite3_stmt* table1_query_statement = NULL;
int res = sqlite3_prepare_v2(
  mp_handle,
  table1_query.c_str(),
  strlen(table1_query.c_str()),
  &table1_query_statement,
  NULL
);

if(res != SQLITE_OK)
{
  printf("ERROR: could not create statement %d\n",res);
  return entry;
}


//execute the statement
res = sqlite3_step( table1_query_statement );

//we expect only one result as 'string' is unique
if(res != SQLITE_DONE)
{
  printf("ERROR: could not execute statement %d\n",res);
  printf("%s\n",table1_query.c_str());

  //clean up
  res = sqlite3_finalize(table1_query_statement);
  if(res != SQLITE_OK)
  {
    printf("ERROR: could not free statement %d\n",res);
  }

  return entry;
}

//wrap up the result
entry = new Table1Entry();
unsigned char* data__ = NULL;

entry->m_IDTable1 =
  sqlite3_column_int(table1_query_statement, 0);
entry->m_string=
  std::string( (const char*)
  sqlite3_column_text(table1_query_statement, 1) );
entry->m_int=
  sqlite3_column_int64(table1_query_statement, 2);

data__ = (unsigned char*)
  sqlite3_column_blob(table1_query_statement, 3);

if(data__ != NULL)
{
  entry->mp_blob =
    new Buffer(sqlite3_column_bytes(table1_query_statement, 3));
  memcpy(entry->mp_blob->get_mp_data(),
    data__,entry->mp_blob->get_m_length());
}

//clean up
res = sqlite3_finalize(table1_query_statement);
if(res != SQLITE_OK)
{
  printf("ERROR: could not free statement %d\n",res);
}

return entry;
}

int TestDB::addEntry(Table1Entry* entry)
{
//open a connection
if(this->connect() == -1)
```

```
{
        return -1;
}

//---------------- insert data
std::string add_query("INSERT INTO "\
  "`Table1`(string,int,blob) VALUES(?,?,?);");

sqlite3_stmt* add_query_statement = NULL;
int res = sqlite3_prepare_v2(
  mp_handle,
  add_query.c_str(),
  strlen(add_query.c_str()),
  &add_query_statement,
  NULL
);

if(res != SQLITE_OK)
{
  printf("ERROR: could not create insert statement %d\n",res);
  return -1;
}

sqlite3_bind_text(add_query_statement, 1,
  entry->m_string.c_str(), -1, SQLITE_STATIC);
sqlite3_bind_int64(add_query_statement, 2,
  entry->m_int);
sqlite3_bind_blob(add_query_statement, 3,
  entry->mp_blob->get_mp_data(),
  entry->mp_blob->get_m_length(), SQLITE_STATIC);

//execute the statement
res = sqlite3_step( add_query_statement );

//we only expect one row
if(res != SQLITE_DONE)
{
  printf("ERROR: could not execute insert statement %d\n",res);

  //clean up
  res = sqlite3_finalize(add_query_statement);
  if(res != SQLITE_OK)
  {
    printf("ERROR: could not free insert statement %d\n",res);
  }

  return -1;
}
```

```

//clean up
res = sqlite3_finalize(add_query_statement);
if(res != SQLITE_OK)
{
  printf("ERROR: could not free insert statement %d\n",res);
  return -1;
}

return 0;
}


int TestDB::updateEntry(Table1Entry* entry)
{
//open a connection
if(this->connect() == -1)
{
  return -1;
}

//---------------- insert data
std::string update_query = std::string("UPDATE `Table1` "\
  "set string=?, int=?, blob=?, "\
  " WHERE `string`=\"") + entry->m_string + std::string("\";");

sqlite3_stmt* update_query_statement = NULL;
int res = sqlite3_prepare_v2(
  mp_handle,
  update_query.c_str(),
  strlen(update_query.c_str()),
  &update_query_statement,
  NULL
);

if(res != SQLITE_OK)
{
  printf("ERROR: could not create update statement %d\n",res);
  return -1;
}

sqlite3_bind_text(update_query_statement, 1,
  entry->m_string.c_str(), -1, SQLITE_STATIC);
sqlite3_bind_int64(update_query_statement, 2,
  entry->m_int);
sqlite3_bind_blob(update_query_statement, 3,
  entry->mp_blob->get_mp_data(),
  entry->mp_blob->get_m_length(), SQLITE_STATIC);
```

```

//execute the statement
res = sqlite3_step( update_query_statement );

//we only expect one row
if(res != SQLITE_DONE)
{
  printf("ERROR: could not execute update statement %d\n",res);

  //clean up
  res = sqlite3_finalize(update_query_statement);
  if(res != SQLITE_OK)
  {
    printf("ERROR: could not free update statement %d\n",res);
  }

  return -1;
}

//clean up
res = sqlite3_finalize(update_query_statement);
if(res != SQLITE_OK)
{
  printf("ERROR: could not free update statement %d\n",res);
  return -1;
}

return 0;
}
```

SimpleHydra does not provide much wrapping for the vast SQLite API, yet if one does not need to handle binary data (i.e. blobs) it is possible to write much more readable functions with the *SQLiteToolbox* class. Let's assume we want to update an entry in Table1 but without altering the corresponding BLOB attribute, the following function would achieve this

```
int TestDB::updateEntryNOBLOB(Table1Entry* entry)
{
//open a connection
if(this->connect() == -1)
{
        return -1;
}

//------------------------------ insert data
std::string update_query(UPDATE `CLKernel` set string= );

update_query = update_query +
  entry->m_string + std::string(", int= ");
update_query = update_query +
  Toolbox::longLongToString( entry->m_int );

update_query =  update_query + "WHERE string'=\"") +
  entry->m_string + std::string("\";");

int res =
  SQLiteToolbox::executeStaticSQLiteStatement(
    mp_handle, update_query);

return res;
}
```

The *connect* method of *SystemDB* is a protected element, thus it can only be used by the classes themself, every method should call the connect function (multiple calls pose no threat). The rationale behind is that once the connect method was called the object internal database connection will be active until the object is destroyed, thus there is no explicit disconnect method.
More examples can be obtained by studying the SH source code in "/Core/Database" for the various SH databases.

IV.7. Threading

One major aspect of SimpleHydra is its threading system, besides an obligatory thread class it also features a scheduling system and thread pools. We will begin with writing a basic thread class

```
class ExThread : public Thread
{
  ExThread(){}
  virtual ~ExThread(){}

  public void preFlag()
  {

  }

  public void postFlag()
  {

  }

  public void run()
  {
    while(m_shutdown_flag == false)
    {
      printf("Thread (%u/%ull) wrote to stdout\n",
        m_local_thread_id, m_system_thread_id);

      Toolbox::sleep_ms(500);
    }
  }
}
```

The threads logic is implemented inside the *run* method, which in our example contains the most common structure; a while loop with a sentinel called *m_shutdown_flag* (a member of *Thread*). In order to run the thread we write

```
ExThread t;
t.run();

Toolbox::sleep_ms(5000);

/*The thread will automatically be
stopped once the object gets destroyed.*/
t.stop();
```

The thread can be stopped explicitly with a call to *stop*, if a running thread object gets destroyed, the corresponding system thread will also be stopped. This promise of thread control can only be guaranteed iff setting the sentinel value true will induce an exit of the (currently executed) run method. Yet often this is not the case, observe the following situation

```
class ExThread : public Thread
{
  ExThread(){m_a=true;}
  virtual ~ExThread(){}

  public void run()
  {
    while(m_shutdown_flag == false)
    {
      while(m_a == true)
      {
        Toolbox::sleep_ms(500);
      }
      m_a = true;
    }
  }

  private:
  bool m_a;
}
```

This run method will never exit once it started, furthermore it can't be stopped by the sentinel value alone. In order to deal with such situations the thread class provides two methods; *preFlag* and *postFlag*. Their purpose is to ensure that all conditions are met in order for the sentinel to control the run method. *preFlag* should contain all logic which is required before setting the sentinel to false, where as *postFlag* deals with all logic after the sentinel has changed to false. Internally the thread classes stop method will call *preFlag*, set *m_shutdown_flag* to false and call *postFlag* before attempting to join the corresponding system thread. In our example

```
class ExThread : public Thread
{
  ExThread(){m_a=true;}
  virtual ~ExThread(){}

  public void postFlag()
  {
    m_a = false;
  }

  public void run()
  {
    while(m_shutdown_flag == false)
    {
      while(m_a == true)
      {
        Toolbox::sleep_ms(500);
      }
      m_a = true;
    }
  }
  private:
  bool m_a;
}
```

The rationale is that *m_a* will be set to false, this will enable the sentinel to fully control the run method as the inner while loop won't be entered again. It would lead to dead locks if one would use the preFlag method instead, as the inner while loop would exit yet *m_a* could set true again and the inner while loop would be entered again before the sentinel would be evaluated! Let us take a look at the following brain teaser

```
class ExThread : public Thread
{
  ExThread(bool* external_a, bool internal_a){
    m_a = internal_a;
    mp_a = external_a;

    if(external_a == NULL){
    mp_t = new ExThread(&m_a, false);
    mp_t->run();
    }
    else
    mp_t = NULL;
  }
  virtual ~ExThread()
  {
    if(mp_a != NULL)
    *mp_a = false;

    delete mp_t;
  }

  public void postFlag()
  {
    m_a = false;
  }

  public void run()
  {
    while(m_shutdown_flag == false)
    {
      if(mp_a != NULL)
      while(m_a == true)
      {
        Toolbox::sleep_ms(5);
        *mp_a = true;
      }
      else
      while(m_a == true)
      {
        Toolbox::sleep_ms(500);
      }

      m_a = true;
    }
  }

  private:
  bool m_a;
  bool* mp_a;
  ExThread* mp_t;
}
```

If we use it in the following way

```
ExThread t(NULL,true);
t.run();
Toolbox::sleep_ms(5000);
t.stop();
```

our program will most likely hang! Why? First we note that if we provide a null pointer in the constructor, the objects run behavior will be left unchanged to our previous example. If the thread object is provided with a real pointer, the run method will start to continuously set the corresponding boolean variable true. This poses a problem as our thread (T_1) object will start a thread T_2 itself and provide it with a pointer to its *m_a* member. In order to controllably stop T_1 we need to ensure that T_2 won't interfere with our strategy of setting T_1s *m_a* to false. A reasonable solution would be to stop T_2 before the call to *postFlag*. This is an example where the *preFlag* method comes in

```
class ExThread : public Thread
{
  ExThread(bool* external_a, bool internal_a){
    m_a = internal_a;
    mp_a = external_a;

    if(external_a == NULL){
    mp_t = new ExThread(&m_a,false);
    mp_t->run();
    }
    else
    mp_t = NULL;
  }
  virtual ~ExThread()
  {
    if(mp_a != NULL)
    *mp_a = false;

    delete mp_t;
  }

  public void postFlag()
  {
    m_a = false;
  }

  public void preFlag()
  {
    if(mp_t != NULL)
    mp_t->stop();
  }

  public void run()
  {
    while(m_shutdown_flag == false)
    {
      if(mp_a != NULL)
      while(m_a == true)
      {
        Toolbox::sleep_ms(5);
        *mp_a = true;
      }
      else
      while(m_a == true)
      {
        Toolbox::sleep_ms(500);
      }

      m_a = true;
    }
  }

  private:
  bool m_a;
  bool* mp_a;
  ExThread* mp_t;
}
```

This approach will take care of of the problem as it ensures that T_2 will be stopped. As long as one ensures that the sentinel has full control over the run method, the thread will always be stoppable via *stop* or by deleting the object.

IV.8. Thread Pools

Thread pools are essentially a set of running threads which reside in a (mutex-based) waiting state until a task is enqueued in the pool. We will first discuss the general task architecture and afterwards show how to apply it in thread pools. All tasks are extensions of the class *Task*, SimpleHydra provides one such extension namely the class *FunctionTask*, which provides a generic way to schedule asynchronous work. Every child class of *Task* must implement the method *runTask*, which in the case of *FunctionTask* looks like

```
void runTask(Thread* executing_thread = NULL)
{
  //execute the function only if it exists
  if(mp_execution_function != NULL)
  {
    //execute the desired function
    mp_execution_function(executing_thread,
      mp_function_parameter);
  }

  //run the callback
  callback(executing_thread);
}
```

Before considering the details, once a task is scheduled in a thread pool, an arbitrary thread will claim the task object, execute its runTask method and delete it afterwards. The general idea within *FunctionTask* is to first call a static function via a member function pointer *mp_execution_function*, this pointer is set via *setmp_function*. If the function pointer is set to NULL, the call will be omitted. An important note about the requirements for such a functions signature; the parameter list can either be *(Thread*,U)* or *Thread**. In other words, the function must always accept a thread pointer and can at most have another parameter of template type *U*. An example for both cases would be

```
int f(Thread* t, double a)
{
 if(t != NULL)
 printf("f called by thread %ull \n",
   t->get_m_system_thread_id());
 return (int)a;
}

std::string g(Thread* t)
{
 if(t != NULL)
 printf("g called by thread %ull \n",
   t->get_m_system_thread_id());
 return std::string("Hello");
}
```

Thus the return type is arbitrary and any returned value will be discarded within the function task. It is also important to check *t* for NULL as it is the parameters default value (i.e. SH does not enforce a "legit" pointer value). Yet the call of *callback* also needs to be clarified, the parent class *Task* provides the ability to register a so called callback queue (*CallbackQueue*). A callback queue itself may contain callback objects (*Callback*) which provide a very generic interface for delegating tasks. Every class which extends *Callback* must implement *callbackMethod*, which is the method that gets called once an object is dequeued from the callback queue. Time for another illustrative example

```
class ExCallback : Callback
{
 ExCallback(){}
 virtual ~ExCallback(){}

 void callbackMethod()
 {
   printf("hello world\n");
 }
}
```

The class *ObjectCallback* provides the feature to register non-static methods for execution, just as with static functions, any registered member function must provide a signature with either *(Thread*,V)* or *Thread**. Let us assume the functions from before reside in a non static context of a class *ExFoo*, the usage in a object callback would be

```
ObjectCallback<int,ExFoo,double> oc_f;
ObjectCallback<std::string,ExFoo> oc_g;

ExFoo foo;

oc_f.set_mp_callee_object(&foo);
oc_g.set_mp_callee_object(&foo);

oc_f.set_mp_object_method(&(ExFoo::f));
oc_g.set_mp_object_method(&(ExFoo::g));

oc_f.set_mp_parameter(36);//a double value

//explicit calls
oc_f.callbackMethod();
oc_g.callbackMethod();
```

We added the explicit calls at the end to illustrate the rationale, usually *oc_f* and *oc_g* would be registered in a callback queue which in turn would be handled by a thread pool. For the sake of completion we show the registration

```
CallbackQueue* q = new CallbackQueue();

q->enqueueCallback(oc_f);
q->enqueueCallback(oc_g);
```

Thus a call to

```
q->runElements();
```

would be tantamount to

```
oc_f.callbackMethod();
oc_g.callbackMethod();
```

An important note about the *ObjectCallback* constructor; the boolean parameter determines if the registered object should be deleted once its callback method was called (in any case it will be removed from the queue). In order to follow our red line we register this queue in a function task object

```
FunctionTask<void,void>* ft = new FunctionTask<void,void>();

ft->registerCallbackQueue(q);
```

The callback queue must be a heap object as the function task takes ownership during the assignment. Let us now venture to finally create a thread pool.

```
/*create a TP with 10 threads and an TP ID of 1*/
ThreadPool tp(10,1);

tp.scheduleTask(ft);
```

Again we created the function task as a heap object as *scheduleTask* will claim ownership and delete the function task after a thread has executed it. A possibly trivial note; although the callback queue contains two callback objects, they will all be executed by one and the same thread from the pool, parallelism happens on a task-object-level, i.e. tasks will be executed concurrently and not the actions contained in the task. Once a thread pool is no longer needed one can stop it by simply deleting the object. in our example we enqueued function tasks without any reference to static functions, only the callback objects will execute some (more or less useful) logic.
For the sake of brevity; there is a way of defining function tasks for static functions without any parameters at all (i.e. not even a Thread* parameter)

```
    SHCore::FunctionTask<void,
     SHCore::NO_STATIC_THREAD_REF__>* ft =
       new SHCore::FunctionTask<void,
          SHCore::NO_STATIC_THREAD_REF__>();
```

This will create a function task for functions returning *void* and expecting no parameters at all, e.g.

```
void f(){};
```

Another feature are persistent callback queues, which differ from ordinary callback queues in the fact that any enqueued callback objects will not deleted but also re-enqueued after execution

```
//create the queue and enqueue the objects
 SHCore::PersistentCallbackQueue* c_queue =
      new SHCore::PersistentCallbackQueue();
 /*o1 will always exist in the queue until
 it's explicitly removed by the queue itself.*/
 c_queue->enqueueCallback(o1);
```

The enqueued callback objects are owned by the queue and will only be deleted upon the queues destruction (of course without calling them).

IV.9. λ Threads

No threading system would be complete without the possibility to use anonymous threads (λ threads). Starting an anonymous thread is done by utilizing thread pools with a single thread.

```
struct lamda_t_s{
    std::atomic_bool done;
    int external_int;
    ThreadPool* tpool;
};

/*lambda functions as wrappers for the tasks
("mutable -> void" ensures non-constness
of captured-by-value value)*/
auto func1 = [] (Thread*, void* var) -> void
{
  struct lambda_t_s* param =
    (struct lambda_t_s*)var;

  int external_int = param->external_int;
  ThreadPool* tpool = param->tpool;

  printf("Thread pool %d and parameter %d\n",
    tpool->get_m_threadpool_id(), external_int);

  param->done = true;
};
```

A bunch of 5 threads can be started with

```
struct lambda_t_s param[5];
ThreadPool** threadpools = new ThreadPool*[5];

for(unsigned int i=0;i<5;++i)
{
  threadpools[i] = new ThreadPool(1,i);

  param[i].done = false;
  param[i].external_int = i+5;
  param[i].tpool = threadpools[i];

  FunctionTask<void,void*>* ft = new FunctionTask<void,void*>();
  ft1->setmp_function(func1);
  ft1->setmp_function_parameter(param+i);

  threadpools[i]->scheduleTask(ft1);
}
```

and synchronized via

```
//wait until all functions have finished
bool finished = false;

while(finished == false )
{
SHCore::Toolbox::sleep_ms(10);

finished = true;
for(unsigned int i=0;i<5;++i)
{
  //as long as one thread (pool) hasn't finished we will continue
  if(param[i].done == false)
  {
    finished = false;
    break;
  }
}
}
```

don't forget the clean up

```
//clean up
for(unsigned int i=0;i<devices;++i)
{
  delete threadpools[i];
}
delete[] threadpools;
```

In fact there exists another way of starting anonymous threads, yet it represents a rather "dirty" approach of getting the job done. Although the following code looks more inviting through its compactness

```
QUICK_THREAD_START(unique1)

for(int i=0;i<20;++i){
printf("hello\n");
Toolbox::sleep_ms(500);
}

QUICK_THREAD_END(unique1,NULL)
QUICK_THREAD_JOIN(unique1)
```

it induces some code overhead (one new structure declaration for a context local thread). This becomes obvious in the expansion of the above macros

```
struct unique1_struct{
  static void* unique1_function(void* parameter__) {

```

```
  for(int i=0;i<20;++i){
  printf("hello\n");
  Toolbox::sleep_ms(500);
  }

  }
};
pthread_attr_t unique1_thread_attr__;
pthread_t unique1_system_thread_id__;
pthread_attr_init(&unique1_thread_attr__);
pthread_attr_setdetachstate(&unique1_thread_attr__,
  PTHREAD_CREATE_JOINABLE);
pthread_create(&unique1_system_thread_id__,
  &unique1_thread_attr__,
  &unique1_struct::unique1_function, 0);

pthread_join(unique1_system_thread_id__, 0); \
pthread_attr_destroy(&unique1_thread_attr__);
```

The expansion also indicates the way of passing external parameters to logic enveloped by *QUICK_THREAD_START* and *QUICK_THREAD_END*, through the use of the static parameter *parameter___*. In order to keep the anonymous threads referencable an unique identifier must be specified, in the example above we have chosen *unique1* as an identifier. Of course it is not mandatory to join the started thread, i.e. the macro *QUICK_THREAD_JOIN* may be omitted.

IV.10. BlitzView

For systems or environments which attempt to keep a small footprint regarding external libraries it would be a burden to install SHs visualization module as it requires Qt which in turn exhibits its own external dependencies. For such cases (especially for the case of a minimal SH installation) one can turn to the class *BlitzView* (BV) which enables one to display images without additional modules, the only requirement is the X11 library (which should be present anyway for system that attempt to visualize data). We will explain it with a short example

```
SHCore::Image<unsigned char>* image = //...

//create and start the blitzview
SHCore::BlitzView bv(640,480,"Image",30);
bv.start();

//wait a moment for it to start
bv.waitForStart();

bv.showImage(image);
SHCore::Toolbox::sleep_ms(5000);

bv.clearWindow(255,0,0);
SHCore::Toolbox::sleep_ms(5000);

bv.showImage(image);
SHCore::Toolbox::sleep_ms(5000);

bv.setWindowSize(800,600);
SHCore::Toolbox::sleep_ms(5000);

for(int i = 0;i<300;++i)
{
    bv.setPixel(10+i,10,0,0,0);
    SHCore::Toolbox::sleep_ms(100);
}

//stop the blitzview
bv.shutdownJoin();
```

The last call already indicates the thread nature of BV, we begin by crating a BV instance of size 640x480, the value 30 defines the responsivity time (in ms) for an internal keep-alive loop (one can safely ignore this parameter and leave it at the default valu of 30). Once the BV has been started we can issue commands like *showImage*, the follow up commands should be self explanatory (otherwise consult the Doxygen documentation). BV exhibits both, thread- safe and non thread-safe functions, similarly to the data structures the non thread-safe methods are indicated by the prefix ”fast”. A final note about closing windows, once a window was closed its content is lost i.e. there is no hide/show mechanism for BV windows.

IV.11. Compression

SimpleHydra also features a pragmatic interface to zlibs compression routines, which are wrapped in a slim class called *compressionBox*. The following listing shows how to compress a buffer object

```
unsigned int size = 5000;

Buffer testInput(size);
CompressionBox box;

box.setCompressionLevel(5);

//fill the buffer 'testInput' with some data
```

```
//...

Buffer* compressed = box.compressNoStream(&testInput);
```

This example compresses a buffer of 5000 bytes into a new buffer called *compressed*, note that we are working with memory resident elements, i.e. the input and output will reside in the process's memory (as indicated by the methods suffix 'NoStream'). Currently SH only supports that form of compression, later versions might include support for stream based compression, i.e. the possibility for feeding data incrementally to the object. The compression level lies in the range between 0 (no compression) and 9 (maximal compression), any compressed data can be decompressed through e.g.

```
 Buffer* decompressed = box.decompressNoStream(compressed, size);
```

where *size* is the decompressed size, this parameter does not have to be set, yet if the original datas size is not specified, the decompression will be mucht more inefficient. Note that the usage of this parameter is boolean, either set the size or leave it at the default value of 0! If one desires to work with files instead of buffers the following methods can be used instead

```
void compressFile(const std::string& input_file,
  const std::string& output_file);
void decompressFile(const std::string& input_file,
  const std::string& output_file);
```

which should be self explanatory (the compression method will use the previously set compression level). Keep in mind that every zlib compressed data can be handled by these methods, no matter where it resides be it buffers or files, thus even externally compressed (headerless !) data can be handled by this class. The following methods allow for some performance tweaking

```
void setDecompressNoStreamSegmentSize(unsigned int size);
void setFileReadChunkSize(unsigned int size);
```

where the first one specifies the read ahead step for buffer decompression in case of an unknown original size, where the second method does the same for the case of files. Note that for file decompression there is no parameter regarding the original size, which is due to the fact that files are considered to be a streaming data source. Later versions of SH might add support for non-streaming decompression. For the sake of completion we conclude this section with an example of file de-/compression

```
box.setCompressionLevel(9);
box.compressFile("input.txt", "output.txt");
box.decompressFile("output.txt", "input2.txt");
```

IV.12. Processes

Our framework also provides access to deeper system elements like e.g. processes, in this section we will discuss the corresponding classes *Process*, *ProcessHelper*, *ProcessToolbox* and *UsageLimiter*. Regarding the attributes and behavior of Linux system processes we defer the user to the corresponding MAN pages and literature, in the following we will assume the reader to be familiar with the basic mechanics of system processes. An example can often explain things better than a thousand words

```
Process p(22693);
p.printProcInfo();
for(unsigned int i=0;i<20;++i)
{
        p.updateTimeValues();
        p.printCPUTime();

        SHCore::Toolbox::sleep_ms(1000);
}
```

An instance of *Process* represents and object oriented access to a system process, in this case we create the object with a PID 22693 as constructor parameter. The PID specifies the system process to which this instance will correspond, if no PID is specified the object will be considered unprimed, which results in errors on most of its methods. In our example we first call *updateTimeValues* which will print all available process attributes on the CLI, this is followed by a for-loop which updates the processes CPU times and print them on the CLI which is followed by a pause of 1 second. If the object was constructed with no PID it can be primed by using *updatePID*, the second parameter in the classes constructor and *updatePID* specifies if the CPU times of the processes children should be summed up in the processes CPU times, the default value *TIME_DETAIL_WITHOUT_CHILDREN* sets this to be false. The following methods require the calling process to have kill-rights to the mapped process, e.g. by running under root or the same user id as the mapped process

```
int killProcess();
int stopProcess();
int resumeProcess();
int interruptProcess();
```

These methods should be self explanatory, the following methods require even root rights

```
int setAffinity(FastKTuple<unsigned int> &cpus);
int setPriority(int prio);
```

The mapping between a system process and the corresponding object is a loose one, i.e. the system process can vanish without informing the object. In order to address such situations one can use the method *doesExist* which reliably checks if the process still exists. Technically it checks if a process with the specified PID exists and if this process has the right start time (upon mapping/priming the object records the processes start time). The class contains one static method called *getSelf* which returns a process instance to the calling process, this is useful for e.g. enabling a process to control itself by checking its consumption of CPU cycles. Additionally the class contains a vast amount of public member variables which correspond to the processes system attributes, we will only elaborate on a few of them as they are linked to certain class methods. The member *m_cpu_usage_perc* contains the processes average CPU usage (in percent) at the last call of *updateTimeValues*, this average value is calculated as follows

$$t_{average} = (1.0 - \alpha) * t_{average} + \alpha * \Delta J/\Delta t; \tag{IV.1}$$

which is a linear interpolation between the last percentual usage and the new one, ΔJ represents the jiffie difference and Δt the time difference between the last and current call to *updateTimeValues*. One can view this calculation as an averaging over all updates with an exponential decay

Theorem 2. *Let $\tilde{t}$ be the value for $t_{av,0}$ which is the initial average time. The averaging rule*

$$t_{av,i} = (1.0 - \alpha) * t_{av,i-1} + \alpha * \tilde{t}_{i-1} \tag{IV.2}$$

can be expressed as

$$t_{av,n} = (1 - \alpha)^n \tilde{t} + \alpha \sum_{i=0}^{n-1} (1 - \alpha)^{n-1-i} \tilde{t}_i \tag{IV.3}$$

Where $\tilde{t}_i = (\Delta J/\Delta t)_i$ represents the usage in percent between update i and $i - 1$.

Proof. Let us first note that

$$\begin{aligned} t_{av,0} &= \tilde{t} && \text{(IV.4)} \\ t_{av,1} &= \tilde{t} - \alpha\tilde{t} + \alpha\tilde{t}_0 && \text{(IV.5)} \\ t_{av,2} &= (1-\alpha)^2\tilde{t} + \alpha(1-\alpha)\tilde{t}_0 + \alpha\tilde{t}_1 && \text{(IV.6)} \end{aligned}$$

which in turn leads to

$$t_{av,3} = (1-\alpha)^3\tilde{t} + \alpha(1-\alpha)^2\tilde{t}_0 + \alpha(1-\alpha)\tilde{t}_1 + \alpha\tilde{t}_2 \tag{IV.7}$$

This allows us to conclude with a complete induction over n. The statement holds for $n = 0$, let the statement now be valid for $n - 1$ i.e.

$$t_{av,n-1} = (1-\alpha)^{n-1}\tilde{t} + \alpha \sum_{i=0}^{n-2} (1-\alpha)^{n-2-i} \tilde{t}_i \tag{IV.8}$$

by simply inserting this equation into that of $t_{av,n}$ we obtain the desired expression in the same manner as for $t_{av,1}, t_{av,2}, t_{av,3}$.

$$\begin{aligned}
t_{av,n} &= (1-\alpha)\ [(1-\alpha)^{n-1}\tilde{t} + \alpha\sum_{i=0}^{n-2}(1-\alpha)^{n-2-i}\tilde{t}_i] + \alpha\tilde{t}_{n-1} && \text{(IV.9)}\\
&= (1-\alpha)^n\tilde{t} + \alpha\sum_{i=0}^{n-2}(1-\alpha)^{n-1-i}\tilde{t}_i + \alpha\tilde{t}_{n-1} && \text{(IV.10)}\\
&= (1-\alpha)^n\tilde{t} + \alpha\sum_{i=0}^{n-2}(1-\alpha)^{n-1-i}\tilde{t}_i + \alpha(1-\alpha)^{(n-1)-(n-1)}\tilde{t}_{n-1} && \text{(IV.11)}\\
&= (1-\alpha)^n\tilde{t} + \alpha\sum_{i=0}^{n-1}(1-\alpha)^{n-1-i}\tilde{t}_i && \text{(IV.12)}
\end{aligned}$$

$\square$

The proof (see $t_{av,1}, t_{av,2}, t_{av,3}$) also shows the dampening effect for previous utilization values, one should also note that setting $\alpha = 1$ completely removes any predecessing utilization value. This approach is useful for situations where the utilization has sporadic high/low peaks, which wouldn't be recorded in a naive approach (i.e. for $\alpha = 1$). With $\alpha < 1$ one can predictavely smooth out any occuring fluctuations for the average value. A last remark regarding the situation in which the underlying system process ceases to exist, any call to *updateTimeValues* will record a new jiffie delta of 0, i.e. no additional CPU usage will be summed up and the average consumption will continously decrease with any following call to the mentioned method.
Let us now discuss the remaining classes, *ProcessHelper* currently provides only the static method *getAllProcesses*, which in as the name suggests returns a list of all currently existing processes in the system. The class *ProcessToolbox* also contains only one static method called *spawnProcess* which is utilized for e.g. deploying Java based SHU server applications. Let us take a brief view on how this is done

```
FastKTuple<std::string> params(3);

params.setElement(0, "-cp");
params.setElement(1, Toolbox::getCurrentDirectory());
params.setElement(2, "Main");

Process* java_proc =
  ProcessToolbox::spawnProcess("/usr/bin/java",&params);
```

This shows how to spawn a new process with several parameters and obtain an object mapping to it.

The class *UsageLimiter* is another practical tool, it allows to register several processes in it and limit the amount of consumed CPU cycles. Note that in order to utilize this functionality it is mandatory to have kill-privileges for all these processes! An usage example would be

```
UsageLimiter ul(0.9);
ul.setTimeSlotSize(500000);//500us
ul.setPerc(0.5);

ul.registerProcess(5036);
ul.registerProcess(10436);

ul.start();
Toolbox::sleep_ms(5000)//5s
ul.stop();
```

Now this requires some explanation; The limiter operates on a specified time slot, in our case we set this slot to be an interval of 500us. The processes will be limited for that specific amount of time, here we define that all registered processes can consume at most 50% of these 500us, i.e. both registered processes can consume a combined amount of 250us CPU time. Furthermore the start and stop methods already indicate the limiters thread nature, i.e. the limiter will operate from a new thread. Setting a too low value for the time slot might induce a serious overhead since after each passed 'time slot' amount of time the average time for all registered process objects will be updated, i.e. the corresponding files in /proc will be parsed. Yet on the other hand a too high values will make the processes less responsive. The optimal default values must be obtained through evaluation on the particular systems. The limiting system provides an interesting adaptation over time which we will briefly explain through the following pseudo code

Algorithm 4 Limiting algorithm for processes

```
 1: process_set_cpu_time = 0;
 2: work_time = 0;
 3: sleep_time = 0;
 4: work_rate = m_perc;    → slot percentage
 5: while limiter active do
 6:     resume all processes;
 7:     work_time = work_rate*m_time_slot;
 8:     sleep_time = m_time_slot - work_time;
 9:     sleep_ns(work_time);
10:     stop all processes;
11:     sleep_ns(sleep_time);
12:     update average CPU times for all processes;
13:         → sum up the percentual average CPU usage of each process
14:     process_set_cpu_time = $\sum_{i=0}^{pCount} t_{av}(i)$;
15:     work_rate = min(work_rate / process_set_cpu_time * m_perc, m_perc);
16: end while
```

The actual implementation differs in some minor aspects, yet follows the outlined design. During the first iteration the processes will work for $work\_rate * time\_slot$, e.g. in our example 250us, the sleep time will equal 0 during this iteration thus the for loop will swiftly jump to the beginning. At the end the loops end the work rate will be recalculated, now two different situations may occur:

1. The processes consumed less than (or equal) the allowed time in the time slot, i.e. $work\_rate/process\_set\_cpu\_time \geq 1$ which implies $work\_rate/process\_set\_cpu\_time * m\_perc \geq m\_perc$. In this case the next iteration will be identical to the first one since the minimum function return m_perc (i.e. $work\_rate = m\_perc$ as before).

2. The processes consumed more than the allowed time, i.e. $work\_rate/process\_set\_cpu\_time < 1$ which implies $work\_rate/process\_set\_cpu\_time * m\_perc < m\_perc$. Now the work rate will be reduced for the next iteration, i.e. the processes will work for a shorter time and sleep for a longer time in order to compensate for this large consumption. In other words; the process will have less time to execute in the time slot which will hopefully reduce the amount of consumed CPU cycles to the desired maximum of 250us within 500us.

If a registered system process should die within an active usage limiter, the processes time will not be considered any more. A usage limiter can be restarted an arbitrary amount of times.

IV.13. Interprocess Communication

Interprocess communication (IPC) plays a crucial role in SHs cluster functions, we will not cover SysV or POSIX shared memory in detail and assume the reader to be familiar with the fundamental concepts. SimpleHydra provides wrapper classes for both interfaces, SysV and POSIX, please note at this point that due to the nature of IPC the following classes should be seen as husks which are primed with the actual memory objects.
The general interface to shared memory is provided through the class *SharedMemory*

```
void* getAttachedMemory();
size_t getSize();
int markDisposable();
int detachMemory();
```

The rationale behind this design is as follows. Shared memory is usually disposed by the operating system once all involved processes have explicitly declared not to be interested in it anymore **and** the memory was marked as disposable by one process, a call to *markDisposable* will issue a corresponding system call, the actual memory will not be deleted until all processes which have mapped the memory call *detachMemory*. Note that if a process exits it will automatically unmap any shared memory, yet it will not automatically mark the shared memory as disposable. The method *getAttachedMemory* returns a void pointer to the beginning of the corresponding shared memory whereas *getSize* will return the size of the shared memory. In the following sections we will explain the system interface specific class API and show how to actually allocate shared memory and prime wrapper objects. The reader should also refer to the Doxygen documentation in order gain a more detailed understanding of a classes behavior.

IV.13.1. SysV Shared Memory

Let us begin with a short example of how to create a shared SysV memory segment inside a program p_A

```
SHCore::SharedMemorySysV shared_memory(0777);

//create a SysV shared memory block of at least
//1024 bytes under the ID 2004
shared_memory.allocateMemory(2004,1024);

//...use the memory

//detach the memory
shared_memory.detachMemory();

//mark it for deletion
shared_memory.markDisposable();
```

Another program p_B would be able to use this memory as well by simply attaching it through the corresponding ID

```
SHCore::SharedMemorySysV shared_memory(0777);
shared_memory.attachMemory(2004,1024);

//...use the memory

//detach the memory
shared_memory.detachMemory();
```

In this example p_A is responsible for freeing the memory, one has to keep in mind that shared memory will only be freed if it has been marked as disposable and all processes have detached it (detachment occurs implicitly once the process exits, yet marking it as disposable does not happen automatically). One also needs the correct access rights to the memory segment otherwise the attachment process will fail.

IV.13.2. POSIX Shared Memory

Shared memory according to the POSIX standard has the benefit of being indexed through arbitrary string instead of numbers, which eliminates the need for e.g. hash functions to obtain unique memory IDs. The remaining elements are identical to SimpleHydras SysV wrapper, i.e. the memory is created via

```
SHCore::SharedMemoryPOSIX shared_memory(0777);
shared_memory.allocateMemory("memTEST",1024);

//.. do something with the shared memory

//detach the memory
shared_memory.detachMemory();

//mark it for deletion
shared_memory.markDisposable();
```

and indexed via *memTest* afterwards p_B could use it via

```
SHCore::SharedMemoryPOSIX shared_memory;
shared_memory.attachMemory("memTEST",1024);

//.. do something with the shared memory

//detach the memory
shared_memory.detachMemory();
```

IV.13.3. Shared Mutexes

A shared mutex is essentially an ordinary mutex memory block which resides in a shared memory area, the required size for storing a mutex can be obtained via

```
SHCore::SharedMutex::getRequiredSMSize();
```

Once the shared memory (*shared_memory*) has been created one can instantiate a shared mutex in p_A as follows

```
SHCore::SharedMutex s_mutex(&shared_memory);

//create the mutex
s_mutex.createMutex();
```

Afterwards the mutex can be used just like any ordinary mutex

```
//wait 10s
printf("Shared mutex ready\n");
SHCore::Toolbox::sleep_ms(10000);

s_mutex.lockMutex();
for(unsigned int i=0;i<30;++i)
{
     printf("PID 1 %d\n",i);
     SHCore::Toolbox::sleep_ms(1000);
}
s_mutex.unlockMutex();

s_mutex.destroyMutex();
```

The program p_B could then use this mutex and synchronize its operation with p_A through

```
SHCore::SharedMemorySysV shared_memory(0777);
shared_memory.attachMemory(2004,
   SHCore::SharedMutex::getRequiredSMSize());

SHCore::SharedMutex s_mutex(&shared_memory);

//load the existing mutex
s_mutex.loadMutex();

//commence work
s_mutex.lockMutex();
for(unsigned int i=0;i<30;++i)
{
            printf("PID 2 %d\n",i);
            SHCore::Toolbox::sleep_ms(1000);
}
s_mutex.unlockMutex();
```

```

//detach the memory
shared_memory.detachMemory();
```

A call to *destroyMutex* will destroy the mutex which resides in the shared memory, yet this is a very critical operation since there is no guarantee that no other process currently holds a lock on that mutex. SimpleHydra currently provides no means of ensure a safe mutex destruction, yet the idea to solve this problem is to allocate an array of shared mutexes and manage the access to this array via a central and shared r/w mutex (i.e. *sharedRWMutex*, which is identical to an ordinary shared mutex except to the read and write lock operations according to *pthread_rwlock_t*) as follows:
p_B obtains the lock through

```
rw_mutex.lockMutexR();
if(s_mem[0]!=0)
  s_mutex.lockMutex();
rw_mutex.unlockMutex();
```

while p_A uses

```
rw_mutex.lockMutexW();
s_mutex.lockMutex();
//indicate the destroyed mutex
s_mem[2]=0;
rw_mutex.unlockMutex();

s_mutex.destroy();
```

in order to destroy the shared mutex once it is obtained (*s_mem* is a IPC shared char array in which each char element represents an available mutex, i.e. 0 for unavailable and 1 for available, in our example we enumerated the mutexes in a way that the third char element reflects the existence of *s_mutex*). Keep in mind that the same problem occurs at the point of a mutexes creation, yet it can be solved analogously the previous discussion.

IV.13.4. Shared Semaphores

Shared semaphores are very similar to mutexes (only the names of the corresponding functions are different), thus we will only show a brief example on how to instantiate one.

```
SHCore::SharedSemaphore s_sem(&shared_memory);

//create the semaphore with initial value 1
s_sem.createSemaphore(1);
```

The synchronization problems regarding the creation and destruction of shared semaphores persist in the same way as previously in the context of shared mutexes, yet this also implies they can be solved in the same manner.

IV.14. Cryptography

The cryptography section of the core module features a huge set of wrapper classes to methods within LibTomCrypt. This includes access to hashing functions, block ciphers in various chaining modes, public key cryptography and PRNGs. Due to the large amount of supported ciphers we can't possibly discuss every corresponding class, instead we will focus only on one representive class of each part (as the others are mostly identical regarding their structure).

IV.14.1. Block Ciphers

Each cipher exists as an extension of the class *CipherBox* which provides the following interface

```
int encryptFile(std::string input_file,
  std::string output_file);
int decryptFile(std::string input_file,
  std::string output_file);

BasicBuffer* encryptBuffer(BasicBuffer* buffer);
BasicBuffer* decryptBuffer(BasicBuffer* buffer);

int encryptBufferRaw(BasicBuffer* input_buffer,
  BasicBuffer* output_buffer);
int decryptBufferRaw(BasicBuffer* input_buffer,
  BasicBuffer* output_buffer);

int encryptBuffer(BasicBuffer* input_buffer,
  BasicBuffer* output_buffer);
int decryptBuffer(BasicBuffer* input_buffer,
BasicBuffer* output_buffer);

int encryptFileRaw(std::string input_file,
  std::string output_file);
int decryptFileRaw(std::string input_file,
  std::string output_file);

int encryptFileInPlaceRaw(std::string input_file);
int decryptFileInPlaceRaw(std::string input_file);
```

The details for each function can be found in the Doxygen documentation, we will just briefly outline the already self explanatory methods. The first two methods should primary be used for any kind of chaining which does not allow arbitrarily sized blocks (e.g. CBC). The encrypted files will contain a small header which usually contains only the original data size. Methods 3 and 4 do the same but for data within buffers, whereas method

5/6 should only be used for chaining modes which support arbitrarily sized blocks (e.g. counter mode CTR) as they won't add any header information. Methods 7/8 are the counterparts for the first two, these should also only be used with chaining modes which support variably sized blocks (e.g. CTR). The last two member functions are special cases of the previous two, these should be used for very large files since all the previous file encryption methods will load the input file into memory and encrypt them into memory as well. The **cryptFileInPlaceRaw* methods will only preload small chunks of the files and write the resulting cipher text immediately back to the disk. In order to further explain the usage we will discuss the class *AESCipherText* with an example

```
unsigned char key[32] = {0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

unsigned char iv[16] = {0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

SHCore::AESBox aes_box(key, iv, 32, CipherBox::CHAINING_MODE_CTR);
{
  //one full block of 16 bytes

  unsigned char pt[16] = {0x01, 0x02, 0x04, 0x08,
    0x10, 0x02, 0x01, 0x03,
    0x05, 0x06,0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

  SHCore::Buffer pt_buffer(pt,16,true);

  pt_buffer.printfBuffer(SHCore::Buffer::BUFFER_DATA_TYPE_UCHAR);

  SHCore::BasicBuffer* ct_buffer =
    aes_box.encryptBuffer(&pt_buffer);

  ct_buffer->printfBuffer(SHCore::Buffer::BUFFER_DATA_TYPE_UCHAR);

  SHCore::BasicBuffer* pt_buffer2 =
    aes_box.decryptBuffer(ct_buffer);

  pt_buffer2->printfBuffer(SHCore::Buffer::BUFFER_DATA_TYPE_UCHAR);
}
```

Here we have used AES with a 32 bytes key, overlooking the fact that it is set to zero we note that one could also specify a smaller key, the accepted sizes are listed in the Doxygen documentation for each cipher class (the same holds for any other parameter).

Once the class was instantiated with an initialization vector (IV), key and chaining type we can continue to the en-/decryption of some dummy data. Since we have chosen the counter mode chaining out input data does not have to be aligned to the ciphers native block size. The Interface usage of *BasicBuffer* instead of *Buffer*, as data often exists in a plain array we can use *PrimitiveBuffer* (a subclass of *BasicBuffer*) instances to carry the array into the cryptographic methods (a primitive buffer will never take ownership of the actual data). To be technically correct we have to mention the existence of a subclass between *CipherBox* and the actual cipher class, which is called *TomCryptWrapper*. Yet this is merely an intermediate segmentation step which provides no additional interface functions, thus the user might safely ignore this class. Note that once set the chaining mode can not be changed, the same holds for the key and IV.

IV.14.2. Hash Functions

Each available hash function is a subclass of *HashFunction*, which exhibits the following interface

```
//-------------------------- Single element hashing
BasicBuffer* getSingleHash(BasicBuffer* input);
int getSingleHash(BasicBuffer* input, BasicBuffer* output);
BasicBuffer* getSingleFileHash(std::string input_file);
int getSingleFileHash(std::string input_file, BasicBuffer* output);

//-------------------------- Continuous hashing
int process(BasicBuffer* input);
BasicBuffer* getHash();
int getHash(BasicBuffer* output);

virtual void reset();
```

The details can again be found in the Doxygen documentation. Every hash function can operate either as a hasher for single buffers or as a data accumulator until the actual hash of all accumulated data is retrieved. Do never mix these operation modes! The first mode is used via the first 4 methods which are already explained mostly by their name. The last three methods correspond to the second operating mode, the object is fed with data via calls to *process* until the hash is retrieved via *getHash*. Of course no input data is buffered via *process* only the internal hash functions state is updated. Again an example for illustration

```
MD5HashFunction md5;

//file hash test
SHCore::BasicBuffer* output = md5.getSingleFileHash("test.txt");
output->printfRawBufferHex();
delete output;
```

```

//——— state test

unsigned int size = 1;
unsigned char* pt =
  DataGenerator<unsigned char>::getRandomNumberArray(0,255,size);

SHCore::Buffer input(pt,size,false);//will also delete the array

//two iterations

md5.process(&input);

md5.process(&input);

output = md5.getHash();

output->printfRawBufferHex();

delete output;
```

Here we get the MD5 hash for a file called *test.txt* and later use the same object as an incremental hasher which is fed twice the same input. Note that after a call to any "get method" the usage of the hash object can safely be switched to the other mode of operation, e.g. after the last call to *getHash* in our example we can use the object for instant hashing via e.g. *getSingleHash*.

IV.14.3. Pseudorandom Number Generators

Pseudorandom number generators (PRNGs) play an important role in cryptographic systems (as we will see in the case of public key ciphers), SimpleHydra provides two kinds of PRNGs; a set of algorithmic PRGNs (e.g. Fortuna, RC4 or Yarrow) and one secure PRNG (which extracts random data from /dev/random). All PRNGs extend the class *PRNG* which provides the following interface

```
int addEntropy(BasicBuffer* entropy_buffer);

int sealEntropy();

int resetPRNG();

BasicBuffer* retrieveEntropyState();

int setEntropyState(BasicBuffer* entropy);

BasicBuffer* getRN(unsigned long length);
```

```

int getRN(BasicBuffer* output);

prng_state* getState();

int getIdx();
```

Every PRNG must first be seeded with some entropy, this is done through *addEntropy* (multiple calls possible, each will add more entropy), afterwards the PRNGs accumulated entropy must be sealed through *sealEntropy*. A PRNG with a sealed entropy pool can be used to obtain pseudorandom numbers by calling *getRN*, in order to update the PRNG with fresh entropy one has to call *resetPRNG* and repeat the previously described steps of adding entropy. The member functions *retrieveEntropyState* and *setEntropyState* provide a way of retrieving the underlying PRNGs state, this can later be used to re-prime a PRNG to the previous state, a call to *setEntropyState* will replace the PRNGs current state (i.e. replace the current entropy but >won't< seal the entropy pool). A similar description applies to the methods *getState* and *getIdx* which also access internal PRNG elements, we won't explain them at this point and defer the reader to the LibTomCrypt documentation and SH source code. Let us view an example of applying an algorithmic PRNG

```
FortunaPRNG prng;
const char* seed = "hello world";
PrimitiveBuffer buffer((unsigned char*)seed,sizeof(seed));

prng.addEntropy(&buffer);
prng.sealEntropy();

for(int i=0;i<15;++i)
{
  BasicBuffer* output = prng.getRN(40);

  output->printfRawBufferHex();

  delete output;
}
```

Here we obtain 15 40 byte sequences via the Yarrow PRNG which is seeded with 11 bytes from the string "hello world". This also demonstrate the usage of *PrimitiveBuffer* as a husk for plain arrays.
The class *SecurePRNG* behaves quite different as it merely represents an interface to /dev/random. This implies that *addEntropy*, *sealEntropy*, *resetPRNG*, *retrieveEntropyState* and *setEntropyState* have become meaningless. Yet a new method has been added to the PRNG interface

```
int kickstartPRNG(PRNG* prng, int entropy_bits);
```

This one allows to prime any PRNG instance through /dev/random i.e. the PRNG will be augmented with *entropy_bits* bits from /dev/random and its entropy pool will be sealed afterwards. It is important to note that *getState* will return NULL for the secure PRNG.

IV.14.4. Public Key Cryptography

Currently SimpleHydra only provides access to RSA encryption and Diffie Hellman key exchange, later versions will also provide access to ECC. Le us first discuss the RSA classes *RSA* and *RSAKey*, an RSA key can be constructed with the help of the RSA class and a PRNG

```
SHCore::RSA rsa;

SHCore::FortunaPRNG prng;
const char* seed = "hello world";
SHCore::PrimitiveBuffer buffer((unsigned char*)seed,sizeof(seed));
prng.addEntropy(&buffer);
prng.sealEntropy();

rsa.makeKey(&prng,1024);

rsa.printKey();

SHCore::RSAKey* key = rsa.getKey();

key->printKey();
```

Here we create an RSA key of length 1024(bits), all the keys elements can be accessed via *RSAKey*s public attributes

```
bool m_private;

unsigned char* mp_e;
unsigned char* mp_d;
unsigned char* mp_N;
unsigned char* mp_p; //p factor of N
unsigned char* mp_q; //q factor of N
unsigned char* mp_qP; // 1/q mod p CRT param
unsigned char* mp_dP; //d mod (p - 1) CRT param
unsigned char* mp_dQ; //d mod (q - 1) CRT param

//length in bytes
long long m_e_length;
long long m_d_length;
long long m_N_length;
long long m_p_length;
long long m_q_length;
long long m_qP_length;
long long m_dP_length;
long long m_dQ_length;
```

Each element is saved as binary data in the corresponding character array (the first array element is the most significant byte of the integer), all RSA attributes are unsigned thus there is no indicator for the sign i.e. all attributes are plain (i.e. not encoded) positive integers whose bits are stored in each character array. The parameter *m_private* indicates whether the key is a private key i.e. if it contains d and e (and corresponding intermediate values). When we created our RSA key in the example from above we created a private key, keep in mind that each method of *RSA* which expects a PRNG as parameter assumes the PRNG to be ready (i.e. have a sealed entropy pool). The complete class interface consists of

```
int makeKey(PRNG* prng, int bit_length = 1024,
  long e = 65537);

BasicBuffer* encryptKey(PRNG* prng, HashFunction* hf,
  std::string tag, BasicBuffer* key);

BasicBuffer* decryptKey(HashFunction* hf,
  std::string tag, BasicBuffer* cipher_t);

BasicBuffer* getSignature(PRNG* prng, HashFunction* hf,
  BasicBuffer* message, unsigned long salt_size = 8);

int verifySignature( HashFunction* hf, BasicBuffer* signature,
  BasicBuffer* message, unsigned long salt_size=8);

int exportKey(BasicBuffer** buffer, bool export_as_private = true);

int importKey(BasicBuffer* buffer);

RSAKey* getKey();

void printKey();
```

We already met the first method, the next two member functions allow the encryption of keys (e.g. for symmetric ciphers), note the requirement for a hash function which is used to hash the given tag (which can be an empty string). In order to decrypt the key again one needs to provide not only the right decryption key but also the previously used tag. Every key will be encrypted according to PKCS_1_OAEP. Through *getSignature* one can obtain the signature for a given message, verification is done with *verifySignature*. Signatures will be created acording to PKCS_1_PSS. Once a key has been generated it can be exported into a buffer via *exportKey*, the second function parameter specifies if the full key should be exported as such or if only the public key part should be inserted into

the buffer (this method will not change the current internal key). A call to *importKey* will import a previously exported key, this will replace any active internal key.
We conclude the section and chapter with a brief discussion about SHs support for the Diffie Hellman key exchange.

```
DHKeyExchange dh1;
DHKeyExchange dh2;
FortunaPRNG prng;
FortunaPRNG prng2;

//create the private keys
dh1.createKey(&prng,128);
dh2.createKey(&prng2,128);

//export the public keys
BasicBuffer* dh1_pk = dh1.exportKey();
BasicBuffer* dh2_pk = dh2.exportKey();

BasicBuffer* secret1 = dh1.createSharedSecret(dh2_pk);
BasicBuffer* secret2 = dh2.createSharedSecret(dh1_pk);

//compare the secrets
if(secret1->get_m_length() != secret2->get_m_length())
{
   printf("ERROR: secrets differ in size\n");
}
else
{
  for(unsigned int i=0;i<secret1->get_m_length();++i)
  {
    if(secret1->get_mp_data()[i] != secret2->get_mp_data()[i])
    printf("ERROR: secrets differ %u\n",i);
    printf("%02x ",secret1->get_mp_data()[i]);
  }printf("\n");
}

//free memory
delete dh1_pk;
delete dh2_pk;
delete secret1;
delete secret2;
```

It is crucial to consider the Doxygen documentation before using this class as it provides some important information about the member functions. In this example we have first created the private and public keys via *createKey*, **the PRNG is expected to be uninitialized** otherwise it will be reinitialized through a secure PRNG. We created two objects *dh1_pk* and *dh2_pk* which correspond to each side in the DH key exchange protocol, afterwards the public key of each object is obtained by *exportKey*. In a practical implementation both sides would exchange these keys and later call *createSharedSecret* in order to calculate the common key, the example ends by comparing the generated common key of both communication parties.

IV.15. System Elements

IV.15.1. CPU

The class *SystemCPU* provides a simple way of querying the systems CPU information (which is obtained via the file '/proc/cpuinfo'). SH considers any logical CPU as a system CPU thus on a system with e.g. two physical CPUs, each with eight core one could access their description through

```
FastKTuple<SystemCPU*> cpus;

for(int i=0;i<16;++i)
{
  cpus.add(new SystemCPU(i));
  cpus[i].printCPUInfo();
}

cpus.eraseWithDestructor();
```

All CPU attributes are accessible through the classes public members, a call to *loadInfo* will update the CPU information, this is useful for e.g. situations in which the clock rate has changed.
During the process of deserialization an empty object is required, i.e. a SystemCPU instance which is not associated with any CPU, such an object can be obtained by using the default constructor.

IV.15.2. System Disks

The following two lines are enough to access a system disks attributes

```
SHCore::SystemDisk sda("/dev/sda");
sda.printInfo();
```

just as in case of *SystemCPU* all disk attributes are accessible through the classes public members. Yet there is a subtle difference when it comes to the contained partitions as each of them is described by an instance of *DiskPartition*. Since these classes use the *blklib* library one may need root privileges to access the disk information, furthermore any information about mount points for partitions will only be obtained if the partition

is actually mounted.
During the process of deserialization an empty object is required, i.e. a SystemDisk instance which is not associated with any disk, such an object can be obtained by using the default constructor.

IV.15.3. Network Interfaces

With the help of the class *InterfaceToolbox* one can easily access all existing NICs in a system. *InterfaceToolbox* provides a set of static functions

```
static FastNCSList<NetworkInterface*>* getAllNICs();
static FastNCSList<NetworkInterface*>* getAllRealNICs();
static NetworkInterface* getFirstRealNIC();
static NetworkInterface* getNIC(const std::string& nic_name);
```

As the names already give a hint about the underlying function we will only briefly elaborate on them. A "real" NIC is any NIC besides the systems loopback device (i.e. "lo"), yet one has to be careful when it comes to system which provide a different naming scheme, SH assumes that "lo" is the loopback device. For details we defer the reader to the Doxygen documentation.

IV.15.4. System Users

As before in the case of system disks we begin with two lines of code

```
SystemUser user("root");
user.printInfo();
```

which also might require root privileges in order to retrieve the information for all users. Instead of using a user name one might also use the corresponding UID. If a list of all existing users is desired

```
FastNCSList<SystemUser*>* users
  = SystemUser::getAllSystemUsers();
for(auto it = users->getStart();
     it != users->getEnd();++it)
{
  it.getElement()->getm_data()->printInfo();
  printf("----------\n");
}
users->destroy();
delete users;
```

will get the job done. The user password is obtained from the shadow file, this implies that no password will be obtained on system which still write their users password into the passwd file. Since there are different ways of hashing the password (e.g. MD5, SHA256, SHA512) the static function *getEncryptedPassword* expects an ID string parameter ("1"=MD5, "5"=SHA256, "6"=SHA512) in order to return the right hash value. Also keep in mind that each hash value is of the form "$id$salt$hash".

IV.15.5. System Groups

In order to keep it brief, information about system groups is obtained very very similar to the case of system users

```
FastNCSList<SystemGroup*>* groups
  = SystemGroup::getAllSystemGroups();
for(auto it = groups->getStart();
     it != groups->getEnd();++it)
{
  it.getElement()->getm_data()->printInfo();
  printf("-----------\n");
}
groups->destroy();
delete groups;
```

Yet there is one major difference and possible source for serious memory overhead, each group object will carry a list of corresponding user objects, for systems with many large groups this might induce problems if many group objects are instantiated.

IV.15.6. System Information

The class *SystemInformation* provides the means to obtain a summary of many system specs, e.g. all system CPUs, meta data for a list of sytem disks, the system log (i.e. the all systemd kernel messages) and currently used memory. Be aware that in case of a large system log an object may exhibit a very large memory footprint.

```
FastKTuple<std::string> disks;
disks.push("/dev/sda");

SystemInformation info(&disks);
info.printSystemInformation();
```

During the process of deserialization an empty object is required, i.e. a SystemInformation instance which is not associated with any disk, such an object can be obtained by using the default constructor.

V. Network module

The network module contains all of SimpleHydras core components for network communication, just as with the core module it follows the paradigm of providing practical wrapper classes while still allowing access to the underlying operating system components. The chapters first section introduces the fundamental socket architecture upon which more complex system as e.g. TCP servers can be built. Thus the following sections will illustrate the design of TCP/UDP servers and corresponding helper classes.

V.1. Sockets

All TCP and UDP socket classes are extensions of *Socket* which represents an abstract wrapper class for file descriptors (i.e. socket file descriptors) regarding network communication. Let us create a UDP socket, send some data and receive a response

```
UDPSocket udpSocket("10.2.129.203",5000,"eno1",
  ADDRESS_TYPE_IPV4);
```

Before venturing into sending and receiving let us briefly discuss the constructor parameters, an UDP socket will be associated with an IP address, keep in mind that this isn't binding, the IP resides as meta information within the object until e.g. a call to "bind". The second parameter represents the local socket port and is followed by the specification of the interface which should be used, the last parameter specifies whether IPv4 or IPv6 should be used. Both last parameters are optional and default to the first found real NIC (i.e. not the loopback device) and IPv4, respectively. Yet even with two parameters the first one can pose a serious challenge as one must know the desired source IP. In order to ease the instantiation of sockets (TCP, UDP and RAW) the class *SocketFactory* provides static helper methods. One could use e.g.

```
UDPSocket* udpSocket =
  SocketFactory::createUDPSocketFirstNICIPv4(port);
```

which creates an UDP socket on the first found real NIC. The corresponding operating system file descriptor will be created along with the object, furthermore a poll system descriptor will be created in which the socket descriptor will be registered. In order to receive or send data the socket will need associated user space buffers, which will be created with the object as well, the constructor will attempt to instantiate buffers of a default size of 65000 bytes, should the underlying socket use larger OS internal buffers then these larger values will be used for the object internal buffer. This ensures that a call to the receive and send methods will always be able to obtain the complete OS socket data. Once a socket has been created it can be bound to the specified IP/Port information through a call to *bindSocket()*, this ensures that all data sent over this socket will originate from the mentioned address tuple. Now finally to the send/receive operations

```
Buffer* receiveBuff = udpSocket.getRecvBuffer();
Buffer* sendBuff = udpSocket.getSendBuffer();

udpSocket.m_remote_address = "10.2.129.200";
udpSocket.m_remote_port = 5000;

//send some (junk) data to port 5000 of the remote address
for(unsigned int i=0;i<sendBuffer->get_m_length;++i)
{
  sendBuff->get_mp_data[i]=i;
}
udpSocket.send(0);
```

This illustrates an important concept of the class, the socket also includes information about the remote side (IP and port); if both values are specified before the call to *send* it suffices to use “send(0)”. If a different port is desired one can simply call “send(PORT)”, should a smaller fragment of the send buffer be sent one can specify the portion by issuing “send(0,SIZE)”, in case of a different IP one must use “send(0,SIZE,IP)”. In order to receive data one can use

```
itn bytes = udpSocket.receiveFromWho();
```

The method accepts a timeout value, by default it is set to 0ms (i.e. an instantaneous check for available data), a value of -1 corresponds a blocking call. Should some data have been received one can query the sender address through the variables *m_remote_address* and *m_remote_port*, yes, the target attributes of the UDP socket get updated every time when something was received via this method. This function is especially useful when it comes to receiving broadcasts and determining their source. The function returns the amount of received bytes. Yet in high speed communication it might be undesired/unnecessary to execute these updates (string allocations would waste CPU time), for such cases the class provides a simpler version called *send* which is identical to *receiveFromWho* except that it does not process any source address information.
In order to send a broadcast the UDP socket class provides the method *sendBroadcast* which exposes the same interface as the send method, the only difference is the target address which must be a valid subnet broadcast address. The socket must also be bound before attempting to broadcast packets.
TCP sockets are somewhat similar to the UDP variant yet they differ by being connection

oriented or stateful. Creating a TCP socket on e.g. port 5000 is done via

```
TCPSocket* tcpSocket =
  SocketFactory::createTCPSocketFirstNICIPv4(5000);
```

which is identical to the UDP variant. Yet before sending or receiving data one has to call *connectIPv4* which will attempt to establish a TCP connection

```
tcpSocket.connect("10.2.111.20",5900);
```

any existing TCP connection over this socket will be closed. On close inspection it becomes obvious that *TCPSocket* provides no methods for sending or receiving data, these funtions are contained within the "Connection"-class family which will be discussed in the next section.
Checking if connection still active.

V.2. TCP Connections

The family of "Connection" classes is grouped as follows; *Connection* and *ConnectionPoll* are extensions of *GenericConnection*, the child classes differ among each other by the underlying polling strategy for sockets, while *Connection* uses epoll *ConnectionPoll* utilizes the classic poll system. These classes represent the two fundamental branches in the family tree; *TCPConnection* builds upon *Connection* and *TCPConnectionPoll* upon *ConnectionPoll*. The behavior of both branches is nearly identical thus we will state the example only for the epoll variant which exposes a special set of parameters for the underlying poll system (this is the only difference between both branches in the TCP context). An object can be instantiated via 3 constructor types

1. *No external epoll registration*:

```
TCPConnection(unsigned int connection_id,
  unsigned int receive_buffer_size =
    DEFAULT_RECEIVE_BUFFER_SIZE,
  unsigned int send_buffer_size =
    DEFAULT_SEND_BUFFER_SIZE,
  int epoll_in_events=MAX_EPOLL_EVENTS,
  int epoll_out_events=2)
```

2. *Registration within an external epoll-in system*:

```
TCPConnection(unsigned int connection_id,
    int epoll_in_fd,
    struct epoll_event* active_in_events,
    unsigned int receive_buffer_size =
      DEFAULT_RECEIVE_BUFFER_SIZE,
    unsigned int send_buffer_size =
      DEFAULT_SEND_BUFFER_SIZE,
    int epoll_in_events=MAX_EPOLL_EVENTS,
    int epoll_out_events=2)
```

3. *Registration within an external epoll-in and epoll-out system*:

```
TCPConnection(unsigned int connection_id,
    int epoll_in_fd, int epoll_out_fd,
    struct epoll_event* active_in_events,
    struct epoll_event* active_out_events,
    unsigned int receive_buffer_size =
      DEFAULT_RECEIVE_BUFFER_SIZE,
    unsigned int send_buffer_size =
      DEFAULT_SEND_BUFFER_SIZE,
    int epoll_in_events=MAX_EPOLL_EVENTS,
    int epoll_out_events=2)
```

The choice of a constructor is a final decision which can not be reverted during an objects lifetime. An example for choosing the second variant would be the following situation; a worker thread (among many others) manages an internal set of TCP connections (i.e. it polls the ready-to-read descriptors within an epoll registry), thus if the connection objects would carry an own epoll descriptor registry it would induce two sources of unnecessary overhead: 1) the worker thread would need to iterate over many registries (CPU time) and 2) each registry would contain only a single socket descriptor (memory). Let us assume the object has been constructed via one of the listed methods, a connection can then be established through

```
int connect(unsigned int remote_port,
  const std::string remote_ip,
  const std::string& local_ip="",
  unsigned int local_port=0,
  const std::string& local_device="")
```

The information about target IP and port is mandatory all other parameters are optional. Specifying the local IP will induce that the connection will be created through this address (i.e. the underlying socket will be bound), this holds analogously for the port and device parameter. In case of an already existing connection a call to this method will close it and attempt to connect to the given target, independent of the attempts success, the previous connection will be closed. Each constructor variant will create internal send/receive buffers of the specified (or default) size, yet one has the option to replace these buffers with external ones via

```
 setExternalBuffer(SHCore::Buffer* send, SHCore::Buffer* recv);
```

The exchange can not be reverted! At this point we are also able to close the gap between TCP sockets and TCP connections. If a TCP socket has successfully connected to a target one can assign this socket to an existing TCP connection via

```
void set_mp_socket(Socket* socket);
```

This method will also adapt the *internal* buffers size if it's too small in order to hold the sockets waiting data. Any existing socket e.g. obtained through a *connect* call will be deleted and replaced (the connection will take ownership over the socket object). Communication is handled by a set of three methods

```
int send_data(unsigned int data_length,int timeout=-1);
int wait_for_data(unsigned int expected_data,int timeout=-1);
int receive(int timeout = -1);
```

which indicate their function through the naming, as usual a timeout value of -1 is associated with *no timeout* whereas the time is measured in milliseconds.

V.3. TCP / UDP Servers

The creation of TCP and UDP servers is explained in great detail within chapter X, thus we omit it in this context.

V.4. Wake On LAN

A very practical feature is provided by the *WOLFacility* class as it enables one to utilize a NICs Wake On LAN (WOL) feature. Its usage is as simple as

```
SHNetwork::WOLFacility wol;
wol.sendWOLPacketUDP("60:a4:4c:b5:c2:f7");
```

V.5. CURL

The singleton class *CURLWrapper* provides wrapper methods for elemental CURL features

- Get an HTTP(s) ressource with or without HTTP header
- Get an FTP(s) ressource with or without FTP header
- Get an SCP ressource
- Put an FTP(s) ressource

- Put an SCP ressource
- Send an email via STMP TLS.

Two short usage examples for HTTP

```
SHNetwork::CURLWrapper* wrapper = SHNetwork::CURLWrapper::getInstance();
SHCore::BasicBuffer* data = wrapper->getHTTP("http://google.de");
data->printfBufferChar();

delete data;
```

and FTP

```
SHCore::BasicBuffer* header;
wrapper->getFTPwHeader("ftp://ftp.u-tx.net/archlinux/",&data,&header);
data->printfBufferChar();
header->printfBufferChar();
wrapper->putFTP("ftp://x:x@localhost/test.txt",data);

delete data;
delete header;
```

VI. OpenCL

SimpleHydra provides abstraction/wrapper classes for most OpenCL structures, we distinguish between wrapper and helper classes. The set of wrappers consists of *OpenCL-Program*, *OpenCLKernel*, *OpenCLDeviceCapabilities*, *OpenCLContext* and *OpenCLCommandQueue*. These classes feature many (but not all) attributes of the corresponding context. With coming SH revisions these classes will be continuously expanded. For all details we defer the reader to the self explanatory header files as well as the Doxygen documentation (further details can also be obtained from the sources). Just as in the previous chapters we will discuss the usage of the module and show common pitfalls.
The main logic is implemented in the helper classes which are explained in the following sections.

VI.1. Creating the context

An OpenCL context can be created very comfortably via

```
OpenCLContextHelper helper;
helper.setupPlatform(platform_id, false);
```

An *OpenCLContextHelper* instance can handle exactly one OpenCL context (i.e. a device subset of one platform). The call to *setupPlatform* is mandatory, it defines the desired platform for further operations (the second parameter determines if information about all found platforms should be printed to stdout). Once the platform has been chosen, one continues with the selection of devices through following methods

```
helper.initCLContext(std::vector<unsigned int> device_list);
```

which expects a list of device indices within the chosen platform,

```
helper.initCLContextAllGPU();
```

which selects all GPUs in the platform,

```
helper.initCLContextAllCPU();
```

which selects all CPUs in the platform,

```
helper.initCLContextAllDevices();
```

which simply selects all available platform devices. Every of these methods will not only create an OpenCL context for the selected devices, it will also create corresponding command queues and capability objects, which can be obtained through the following functions

```
OpenCLDeviceCapabilities* getDeviceCaps(unsigned int device);
FastKTuple<OpenCLDeviceCapabilities*>* getAllDeviceCaps();
OpenCLPlatform* getPlatformInfo();
OpenCLContext* getCLContext();
OpenCLCommandQueue* getCommandQueue(unsigned int device);
FastKTuple<OpenCLCommandQueue*>* getAllCommandQueues();
```

Keep in mind that all indices are relative to the actually selected devices and not to the devices available on the platform. Additionally one has to respect that once a wrapper class is assigned a native OpenCL element, it will claim ownership over it. Once the context has been created one can use it to execute kernels or manage device memory. Let us create a context for a single GPU, allocate some memory, and execute a simple kernel

```
 OpenCLHelper cl_helper;
 cl_helper.quickSetupSingleDevice(platform,device);

 //system for the first device in the context
 FastVector3<OpenCLDeviceCapabilities*,
          OpenCLCommandQueue*,
          OpenCLContext*> cl_system;

 cl_system = cl_helper.getContextHelper()->getSingleDeviceSystem(0);

 //print device caps
 cl_system.m_data1->printCapabilities();
```

We utilized the *OpenCLHelper* class in order to create the context and request an OpenCLSystem, i.e. a triplet of capabilities, command queue and context, for the first (and only) context device. This is followed by a debug output of the capabilities and the creation of a kernel helper

```
std::string cl_source("\
      "__kernel void testKernel(__global float* input, "\
      "__global float* output) "\
"{" \
" unsigned int globalIdx = get_global_id(0); " \
" output[globalIdx] = 3.0f * input[globalIdx];"
"}"
);

SHOpenCL::OpenCLKernelHelper kernel_helper;
kernel_helper.createKernelFromSourceString(cl_source,
            std::string("testKernel"),
            cl_system.m_data1->getm_id(),
            cl_system.m_data3->getm_context(),
            std::string(""));
kernel_helper.printLastBuildLog();
```

which is used to create and build the OpenCL program and additionally create an OpenCL kernel. This is a rather interesting helper class, as it solely provides methods for creating a ready-to-tun kernel object. In order to create an intermediate form, i.e. an OpenCL program, one has to replace or augment the kernel helper with *OpenCLProgram* (see the corresponding section). We will now allocate the desired amount of device memory for our kernel and copy data into it.

```
OpenCLMemoryHelper<float> memory_helper(context->getm_context(),
        context_helper->getCommandQueue(0)->getm_queue());

cl_mem gpu_data_a = memory_helper.createReadWriteMemory(1024);
cl_mem gpu_data_b = memory_helper.createReadWriteMemory(1024);

float buffer[1024];
for(unsigned int i=0;i<1024;++i)
{
        buffer[i] = i;
}

memory_helper.writeBufferBlocking(gpu_data_a, buffer, 1024);
```

The listing should be self explanatory, a (template) memory helper instance is used to allocate a buffer for 1024 float values. A corresponding host buffer is created, filled with values and copied to the device. Now the kernel parameters must be set as well as the grid dimensions.

```
cl_helper->setup1DDefaultDims(256,4,0);

//set the kernel parameter
kernel_helper->setKernelParameter(0,sizeof(cl_mem),&gpu_data_a);
kernel_helper->setKernelParameter(1,sizeof(cl_mem),&gpu_data_b);

INIT_TIME_AVG
for(unsigned int i=0;i<10000;i++)
{
  TIC_
  cl_helper->launchKernelBlocking(kernel_helper->getKernel(),0);
  AVG_TOC
}
SUM_AVG_TIME_UP
```

We setup a one dimensional thread grid of 4 OpenCL blocks each containing 256 threads (i.e. 1024 threads). We then set the kernel parameters and execute the kernel 10000 times in order to measure the average execution time. Afterwards we verify the data and do the required clean up.

```
float return_buffer[1024];
memory_helper.readBufferBlocking(gpu_data_b, return_buffer, 1024);

for(unsigned int i=0;i<1024;++i)
{
  if(return_buffer[i] != 3*buffer[])
    printf("Error in buffer[%u] %f != %f \n",
      i,buffer[i],return_buffer[i] );
}

delete cl_helper;
delete kernel_helper;
clReleaseMemObject(gpu_data_a);
clReleaseMemObject(gpu_data_b);
```

VI.2. OpenCL Programs

On might encounter situations in which an intermediate OpenCl program representation is required. For this purpose one can refer to the *OpenCLProgram* class. Besides being a wrapper for an OpenCL program, it also features management options as e.g. saving a program object to a file.

```
//create an empty husk
OpenCLProgram clprogram;
```

Once the class has been instantiated we can assign it an OpenCL program through the following ways

1. Assign an already existing OpenCL program via *setCLProgram*

2. Load the programs source from a file, create the OpenCL program without building it; *createProgramFromSourceString*

3. Load the programs source code from a file, create and build the OpenCL program for a single device; *createAndBuildProgramFromSourceString*

4. Build an assigned (method 1 or 2) OpenCL program for a single device; *buildActiveProgram*

5. Load a binary for a single context device or multiple context device; *loadProgram* or *loadProgramMultiDev*

An important note, just as mentioned earlier the wrapper classes take ownership of the OpenCL object, thus it is a bad idea to e.g. use an existing kernel wrapper object, obtain its OpenCL program and assign it to an *OpenCLProgram* instance.

VI.3. Higher OpenCL wrapper

As most OpenCL development utilizes work flows very similar to the one in the previous section, SimpleHydra also features more abstract wrappers for speeding up the development without discarding access to any lower system elements. The previous kernel example can be expressed in the following more compact way

```
std::string cl_source("\
      "__kernel void testKernel(__global float* input,"\
      "__global float* output)"\
 "{" \
 " unsigned int globalIdx = get_global_id(0); " \
 " output[globalIdx] = 3.0f * input[globalIdx];"
 "}"
 );

 OpenCLHelper cl_helper;
 cl_helper.quickSetupSingleDevice(platform,device);

 //system for the first device in the context
 FastVector3<OpenCLDeviceCapabilities*,
             OpenCLCommandQueue*,
             OpenCLContext*> cl_system;

 cl_system = cl_helper.getContextHelper()->getSingleDeviceSystem(0);

 //print device caps
 cl_system.m_data1->printCapabilities();

 //load a kernel from a file
 OpenCLKernelHelper kernel_helper;
 kernel_helper.createKernelFromSourceString(cl_source,
       std::string("testKernel"),
       cl_system.m_data1->getm_id(),
       cl_system.m_data3->getm_context(),
       std::string(""));
 kernel_helper.printLastBuildLog();

 OpenCLKernel* cl_kernel = kernel_helper.getKernel();
 cl_kernel->printKernelInfo();

 //our work data
```

```
unsigned int size = 1024*sizeof(float);
OCLBuffer input(size);
OCLBuffer output(size);

for(unsigned int i=0;i<size;++i)
{
        ((float*)input.get_mp_data())[i] = i;
}

input.setOCLSystem(cl_system.m_data2, cl_system.m_data3);
output.setOCLSystem(cl_system.m_data2, cl_system.m_data3);

input.createDeviceData();
output.createDeviceData();

input.uploadData();

//setup grid
cl_helper.setup1DDefaultDims(256,size/256,0);

//set the kernel parameter
cl_mem a_ = input.getDeviceDataPtr();
cl_mem b_ = output.getDeviceDataPtr();
kernel_helper.setKernelParameter(0,sizeof(cl_mem), &a_);
kernel_helper.setKernelParameter(1,sizeof(cl_mem), &b_);

INIT_TIME_AVG
for(unsigned int i=0;i<10000;i++)
{
        TIC_
        cl_helper.launchKernelBlocking(cl_kernel,0);
        AVG_TOC
}
SUM_AVG_TIME_UP

//read the data back from the device
output.downloadData();

for(unsigned int i=0;i<1024;++i)
{
   if(return_buffer[i] != 3*buffer[])
      printf("Error in buffer[%u] %f != %f \n",
         i,buffer[i],return_buffer[i] );
}
```

Besides allowing for the use of more than 1024 values, the example avoids the usage of any memory helpers. In fact SimpleHydra provides various wrappers for commonly used data structures;

- *OCLBuffer* for one dimensional arrays
- *OCLMatrix2* and *OpenCLMatrix3* as extension of *Matrix2* and *Matrix3*, respectively.
- *OCL2DReadImage*, *OCL2DWriteImage* for read-only and write-only OpenCL 2D images, respectively.

VII. Machine Learning

VII.1. Neural Networks

SimpleHydra features a flexible way of training neural network, more specially it provides an interface for injection of different heuristics at significant locations. The following listing shows the initialization of a neural network with 2 hidden layers, each with 25 neurons and a tanh activation function

```
unsigned int hidden_neurons = 25;

unsigned int layer_count = 2;

SHCore::FastKTuple<unsigned int> layer_neurons(layer_count);

layer_neurons.setElement(0,2);
layer_neurons.setElement(1,hidden_neurons);
layer_neurons.setElement(2,hidden_neurons);
layer_neurons.setElement(3,1);

SHML::NNStructureFullyConnectedFF nn_structure(layer_count,layer_neurons);

nn_structure.setActivationFunction(0,NN_ACTIVATION_FUNCTION_UNIT);
nn_structure.setActivationFunction(1,NN_ACTIVATION_FUNCTION_TANH);
nn_structure.setActivationFunction(2,NN_ACTIVATION_FUNCTION_TANH);
nn_structure.setActivationFunction(3,NN_ACTIVATION_FUNCTION_UNIT);

nn_structure.setSumFunction(0,NN_SUM_FUNCTION_CANONICAL_SUM);
nn_structure.setSumFunction(1,NN_SUM_FUNCTION_CANONICAL_SUM);
nn_structure.setSumFunction(2,NN_SUM_FUNCTION_CANONICAL_SUM);
nn_structure.setSumFunction(3,NN_SUM_FUNCTION_CANONICAL_SUM);

SHML::NNRepresentationCanonicalMatrix* rep =
     (SHML::NNRepresentationCanonicalMatrix*)
     nn_structure.synthesize(NN_REPRESENTATION_TYPE_CANONICAL_MATRIX);
```

Thus the network contains a total of 4 layers (one input layer and one output layer in addition to the hidden layers), note the unit functions for in- and output. The input layer (i.e. first layer) consists of 2 neurons while the output layer contains 1 neuron. The rationale behind the listing is a strict separation of network structure and its synthesized form. In this case the synthesized form *SHML::NNRepresentationCanonicalMatrix* consists of a set of matrices which in turn contain the weights for each layer. The training algorithms work entirely on this synthesized form, yet before one can start a training the corresponding training data must be set via

```cpp
unsigned int sample_count = 20;
SHML::NNDataModelNative model(sample_count*sample_count,2,1);

//sample the sine function
double x1_start = -1.14;
double x1_end = 1.14;
double x2_start = -1.14;
double x2_end = 1.14;
double x1_step = (x1_end-x1_start)/(double)(sample_count);
double x2_step = (x2_end-x2_start)/(double)sample_count;

//these are the initialization parameters
unsigned int x_dim = 2;
unsigned int y_dim = 1;

double sampleInc = x1_start;

for(unsigned int i=0; i < sample_count*sample_count; ++i)
{
    SHCore::FastKTuple<double> x(x_dim);
    SHCore::FastKTuple<double> y(y_dim);

    if(i % sample_count == 0)
    {
        sampleInc += x1_step;
    }

    x[0] = sampleInc;
    x[1] = x2_start + x2_step * (double)(i%sample_count);

    y[0] = sin( std::sqrt(x[0]*x[0] + x[1]*x[1]) );

    model.setData(i,x,y);
}
```

The listing shows the subsampling (20×20 grid starting at $(-1.14, -1.14)$) of the function $\sin(|x|)$ with $x \in \mathbb{R}^2$. The structure *SHML::NNDataModelNative* represents a wrapper which carries the training data into the training algorithm. The interesting part is the training interface, which we explore line by line

```cpp
double eta = 0.1;
double alpha = 0.9;
unsigned int max_iterations = 100000000;
```

```
double error_threshold = 0.01;

SHML::NNTrainerBackProp bp_trainer;

SHML::NNBackPropHeuristicSlowDown slowDown;

bp_trainer.addPreEpochHeuristic(&slowDown);

bp_trainer.setParameters(alpha, eta, max_iterations, error_threshold,
  sample_count*sample_count, true);

bp_trainer.setTrainingData( &model );

bp_trainer.setRepresentation( rep );

bp_trainer.prime();

bp_trainer.initWeightsRandom(-0.001,0.001);
/*bp_trainer.initWeightsRandomFanIn(-0.001,0.001);*/

bp_trainer.initBias(1);

bp_trainer.useFullMomentum(false);

bp_trainer.useShuffling(true);

bp_trainer.trainPar(SHML::NNTrainerBackProp::PARALLEL_MODE_TYPE_SPLIT_SAMPLES);
```

SHML::NNTrainerBackProp is the actual training algorithm, vanilla backpropagation in this case, *SHML::NNBackPropHeuristicSlowDown* represents a so called heuristic, this one is predefined in SimpleHydra. *NNBackPropHeuristicSlowDown* will slow the gradient descent down in case of many sequentially successful gradient steps, its parameters consist of public attributes

```
unsigned int m_epoch_error_counter;
unsigned int m_interval;
double m_inc;
double m_dec;
```

m_epoch_error_counter will contain the number of how often the absolute batch error has increased during *m_interval* epochs, i.e. the number of error regressions in a given epoch number. *m_inc* and *m_dec* represent the number of how much the learning rate should be increased or decreased, after *m_interval* epochs without or with error regressions, respectively. A call to *prime* will initialize the training algorithm with all heuristics, parameters and given training data. Afterwards one can set the range of weight and bias initialization; *initWeightsRandom* will init the weights with uniformaly distributed numbers in the given range, two alternatives exist; *initWeightsRandomFanIn(a,b)* which initializes the weights in the range of $[-a\sqrt{6/f_i}, b\sqrt{6/f_i}]$ i.e. with respect to each neurons fan-in f_i (i.e. the number of the neurons incoming connections) and *initWeightsRandomFanInOut* which also includes the fan-out through $[-a\sqrt{6/f_i}, b\sqrt{6/f_i}]$. The method *useFullMomentum* defines if the momentum approach (and thus the momentum coefficient) should also be applied to the bias weights. A heuristic can be added via three methods, each representing a specific point during the backpropagation training.

- *addPreEpochHeuristic* adds a heuristic which is applied before a batch iteration.
- *addSampleHeuristic* adds a heuristic which is applied between loading the sample from the training set and using it for gradient calculation.
- *addPostEpochHeuristic* adds a heuristic which is applied after a batch iteration.

The basic idea is to write child classes for the corresponding parameter types which implement the *process* method. The training algorithm, be it Quickprop or iRProp, will always call this method of its asociated heuristic objects. We won't go into detail of the methods signatures, instead we only mention that all provided parameters can be used to enhance / modify the actual training algorithm (see vanilla backprop above). Finally one can commence the training through a call to *trainPar* (multicore support) or *train* (single core).
SimpleHydras machine learning module also provides iRProp+/-, Quickprop, RProp+/- (which is available by setting *useIRProp(false)* for the *NNTrainerIRProp* object). In order to decide between + and - one has to call *useBacktracking(true)* or *useBacktracking(false)*, respectively.
A neural networks representation can be saved via *saveNN* and loaded through *loadNN*. Once the network has been loaded it can be provided to an instance of *NNEvaluator* via *setRepresentation* which enables one to evaluate the network. For the sake of brevity we list a small example

```
rep->saveNN("nn.bin");
delete rep;
rep = new NNRepresentationCanonicalMatrix();
rep->loadNN("nn.bin");
NNEvaluator eval;
eval.setRepresentation(rep);
eval.prime();

/*load n data samples as row vectors into matrix x*/
/*allocate a corresponding matrix y with n rows and k columns*/

eval.evaluateNetwork(x,y);

```

```
/* //parallel evaluation
evaluateNetworkParallel(PARALLEL_MODE_TYPE_SPLIT_SAMPLES,x,y);
*/
```

VII.2. PCA

Regarding data preprocessing SH provides classes for principal component analysis (PCA) and kernel principal component analysis (KPCA). It can be simply described with the following listing

```
SHCore::Matrix2<double>* circleData =
  SHCore::MultiDimDataGenerator<double>::generateCircleData(10,1.0,-0.2,0.2);

SHML::PCA<double>::calculatePCATransformation(*circleData,*circleData,1);

circleData->printData();
```

The first parameter is the actual data from which the covariance matrix is calculated and the second parameter represents the data which is to be transformed. Note that the provided data will be modified, yet as shown in the listing above, it poses no problem if the covariance data is identical to the transformation data.

VII.3. Kernel PCA

In addition to the standard PCA SH provides the kernel PCA which exhibits an interface similar to the previous one

```
SHCore::Matrix2<double>* circleData =
  SHCore::MultiDimDataGenerator<double>::generateCircleData(10,1.0,-0.2,0.2);

struct SHCore::DataStatistics<double>::KernelParams params;
params.m_type = SHCore::DataStatistics<double>::KERNEL_COVARIANCE_TYPE_RBF;
params.m_rbf_sigma = 0.1;

SHML::KernelPCA<double>::calculatePCATransformation(*circleData,*circleData,
        1,params);

circleData->printData();
```

The main difference lies in the kernel parameter which must be provided in the transformation call, the attributes of KernelParams should be self explanatory by their names.

VII.4. Genetic Algorithms

Since the support for genetic optimization currently can be considered to be in an pre-alpha stage we mainly mention SHs support for it. A currently supported (yet trivial) example is shown in the following listing

```
PopulationHelper  helper(4,0.6,0.01);
helper.createInitialPopulation(150);
helper.reportCurrentPopulation();

for(unsigned int i=0;i<550;++i)
{
        helper.iterateOneGen();
}

helper.reportCurrentPopulation();
```

A population helper is responsible for the creation of entities, its parameters are chromosome length in bits, propability for crossover and propability for mutation in each generation, respectively. A call to *createInitialPopulation* will create a starting population with the specified amount of individuals, *reportCurrentPopulation* will print a short summary of the current population, *iterateOneGen* will iterate one generation. Future revisions will extend this approach to enable a flexible “blackbox optimization”.

VIII. XML

SimpleHydra provides a general access layer to XML files and a set of reader/writer classes for XML-based configuration files. The following two sections will explain the manipulation of general XML files and the handling of configuration files, respectively.

VIII.1. Reading / Writing XML Files

This framework features two powerful classes in order to create and manipulate XML files, these are *SimpleXML* and *SimpleXMLExtended*. All provided methods are thread safe! Lets assume we have the following XML file/structure saved in *data.xml*

```
<main>
<tag1 attrib1="50"></tag1>
<tag1 attrib1="60">Some data</tag1>
<tag3 attrib88="50">
  <tag1 attrib1="50">DATA!</tag1>
  <tag4>
   something?:-)
     <tag4>
      yes!;-)
     </tag4>
  </tag4>
</tag3>
</main>
```

At first we will attempt to write a parser capable of reading and manipulating these structures. We begin with *SimpleXML* as we are not going to require any methods of the other (more generic) class.

```
SimpleXML sxml;
sxml.parseFile("data.xml");
```

This will result in parsing the given file and saving it as a tree in memory, once done we can access all nodes and their corresponding attributes as well as data. Let us clarify these terms in detail; a node is a representation of an XML tag, each node may contain attributes or enclosed data. Regarding our example we will obtain at least two "tag1" nodes, none of both will contain any data but one will contain an attribute (namely "attrib1"). Let us now access the first "tag1" tag

```
std::vector<std::string> path;
path.push_back("main");
std::string attrib;
int res = sxml.readNodeAttribute(path, "tag1", "attrib1", attrib);
if(res!=0)
  printf("ERROR reading attribute\n");
else
  printf("attribute value %s\n", attrib.c_str());
```

This will result in obtaining the attribute of the first "tag1" node, sadly we have two equally named "tag1" nodes. In order access the second one we need to specify which sibling on this tree level we desire.

```
//get the first sibling
res = sxml.readNodeAttribute(path, "tag1", "attrib1", attrib ,1);
if(res!=0)
  printf("ERROR reading attribute\n");
else
  printf("attribute value %s\n", attrib.c_str());
```

Note that the vector *path* specifies the actual level within the XML tree. This will be more important now as we attempt to read the content enclosed in the "tag1" tag within the only "tag3".

```
std::string content;
//path=main->tag3
path.push_back("tag3");
res = sxml.readNodeContent(path, "tag1", content, 0);
if(res!=0)
  printf("ERROR reading node content\n");
else
  printf("node content %s\n", content.c_str());
```

Note the 0 as the last parameter which is merely an explicit specification which sibling we desire, in this case we could ommit it (i.e. use the default value of 0). Now let us add a new attribute to this node

```
res = sxml.addNodeAttribute(path, "tag1", "newAttrib", "randomVal123", 0);
if(res!=0)
  printf("ERROR adding node attribute\n");
else
  printf("node attribute added\n");
```

Manipulating node content can be done in a very similar way, we defer the reader to the API documentation for the corresponding methods. The class *SimpleXMLExtended* provides essentially vectorized versions of *SimpleXML*'s methods, thus we won't discuss it at this point but instead point to the mentioned documentation. There is a small subtlety when it comes to adding new nodes, all discussed methods required a path under which the desired node was located, the same holds for *addNode*. This method will add a new node at the specified tree level, which poses no problem as long as we don't try to add a node at the top most layer, since there are no layers above and the method requires a path(/node) under which to add the new node. In order to solve this problem one has to use *addNodeSibling* instead, which will add the node as a sibling instead of as a child in the given path(/node). Once we are done with everything it could be wise to save the file

```
sxml.saveXMLTree("data2.xml");
```

The described mechanics favor a very iterative style of parsing the tree, they especially prevent any direct access to the XML tree. If one requires access to the tree, it can be done by obtaining the root node through *getRootElement*. Since *SimpleXML* is nothing more than a smart wrapper for libxml2 everything beyond its functionality must also be implemented by using the libxml2 API.
After reading, manipulating and saving an existing XML file let us create a new file from scratch. All it requires is call to

```
int res = createEmptyTree("root");
if(res!=0)
  printf("ERROR creating empty XML tree\n");
else
  printf("XML tree created\n");
```

This creates an empty XML tree and inserts a first node called "root", if saved at this point we would get an XML file with

```
<root>
</root>
```

In order to add another node on the tree root level we again use *addNodeSibling*, everyhing else is identical to the case of starting with an existing file.
We conclude this section with discussing a small subset of methods within *SimpleXML*.

```
int nextNode();
int childNode();
void listNodeAttributes(xmlNodePtr node);
void printXMLTree(xmlNodePtr node);
xmlNodePtr get_m_current_node();
void reset_current_node();
```

The SimpleXML class contains an internal tracking mechanism for iterating node-wise into the XML tree, one can think of this functionality as a little helper approach for handling libxml2 nodes during a custom search in the loaded XML tree. After calling *parseFile* the internal tracker starts at the root node, by calling *nextNode* or *childNode* the tracker will advance to the respective node. In the latter case it will position itself on the first child node of the previous node. The methods *listNodeAttributes* and *printXMLTree* will list the attributes of the given node or print the XML tree from the given start node, respectively. The currently tracked node can be obtained via *get_m_current_node*, while a call to *reset_current_node* will reset the tracker.

VIII.2. Reading / Writing Config Files

SimpleHydra will feature four manager classes for reading and writing config files, namely *ConfigReaderMarkI*, *ConfigReaderMarkII*, *ConfigReaderMarkIII* and *ConfigReaderMarkIV*. Currently only MarkI has been implemented, it exhibits the restrictions of always reading the whole xml file into memory (which might be inefficient for huge (>50MB) XML files). Furthermore it only supports flat (i.e. non-recursive) config files. Before discussing the MarkI class itself we briefly explain the supported XML format.

```xml
<!-- An example file for the MarkI reader -->
<config>
<!-- Options can be grouped into sections -->
<section name="first">
<!-- Each parameter in a section has a >>>unique<<<
name and a type according to the class enums -->
<!-- scalars -->
<parameter name="param1" type="STRING">
This is a test string
</parameter>
<parameter name="param2" type="DOUBLE">
3.141592653589793238462643383279502884
</parameter>
<parameter name="param3" type="FLOAT">
3.1415926
</parameter>
<parameter name="param4" type="INT">
-2147483647
</parameter>
<parameter name="param5" type="LONG">
-9223372036854755808
</parameter>
<parameter name="param6" type="UINT">
4147483647
</parameter>
<parameter name="param7" type="ULONG">
18446744073709551615
</parameter>

<!-- vectors: the numbers are separated by commata
while the strings are splitted via subtags -->
```

```xml
<parameter name="param14" type="STRING_ARRAY">
<string>This is a test string1</string>
<string>This is a test string2</string>
</parameter>
<parameter name="param8" type="DOUBLE_ARRAY">
3.141592653589793238462643383279502884,6.141592653589793238462643383279502884
</parameter>
<parameter name="param9" type="FLOAT_ARRAY">
3.1415926,6.1415926,9.1415926
</parameter>
<parameter name="param10" type="INT_ARRAY">
-2147483647,-1147483647
</parameter>
<parameter name="param11" type="LONG_ARRAY">
-9223372036854755808,-223372036854755808
</parameter>
<parameter name="param12" type="UINT_ARRAY">
4147483647,1147483647
</parameter>
<parameter name="param13" type="ULONG_ARRAY">
18446744073709551615,9446744073709551615,1446744073709551615
</parameter>

</section>


<!-- And another section -->
<section name="second">
<parameter name="param1" type="STRING">
This is a test string again
</parameter>

<parameter name="param2" type="DOUBLE">
2.141592653589793238462643383279502884
</parameter>
<parameter name="param3" type="FLOAT">
2.1415926
</parameter>
<parameter name="param4" type="INT">
-1147483647
</parameter>
<parameter name="param5" type="LONG">
-5223372036854755808
</parameter>
<parameter name="param6" type="UINT">
2147483647
</parameter>
<parameter name="param7" type="ULONG">
```

```
11446744073709551615
</parameter>

<parameter name=”param14” type=”STRING_ARRAY”>
<string>This is a test string1</string>
<string>This is a test string2</string>
</parameter>
<parameter name=”param8” type=”DOUBLE_ARRAY”>
2.141592653589793238462643383279502884,1.141592653589793238462643383279502884
</parameter>
<parameter name=”param9” type=”FLOAT_ARRAY”>
1.1415926,3.1415926,6.1415926
</parameter>
<parameter name=”param10” type=”INT_ARRAY”>
−1147483647,−147483647
</parameter>
<parameter name=”param11” type=”LONG_ARRAY”>
−972036854755808,−23372036854755808
</parameter>
<parameter name=”param12” type=”UINT_ARRAY”>
83647,47483647
</parameter>
<parameter name=”param13” type=”ULONG_ARRAY”>
46744073709551615,996744073709551615,886744073709551615
</parameter>

</section>
</config>
```

All parameters are enclosed in the root section *¡config¿*, which has can provide multiple option groups called *¡section¿*. In case of multiple sections each must have a unique name! The actual options regarding their data type are depicted in the upper listing. Each options name must be unique within the enveloping section. Now lets take a look at how to read and write such files, let our config file contain

```
<config>

<section name=”Section1”>

<parameter name=”Param1” type=”UINT”>
4147483647
</parameter>

<parameter name=”Pram2” type=”UINT_ARRAY”>
4147483647,1147483647
</parameter>

</section>
```

```

<section name="Section2">

<parameter name="Param1" type="STRING">
This is a test string
</parameter>

</section>

</config>
```

Reading this file can be done via

```
//open the file and read all sections
ConfigReaderMarkI conf_reader("test.xml");
conf_reader.readAllSections();

//iterate over all sections and read the params
FastNCSList<XMLConfigMarkISection*>* sections
   = conf_reader.getSections();

FastKTuple<unsigned int> ids;
std::string s;
unsigned int some_int;

for(FastNCSList<XMLConfigMarkISection*>::Iterator it
   = sections->getStart();
    it != sections->getEnd();
   ++it)
{
 std::string section_type = it.getElement()->getm_data()->getName();

 if(section_type.compare("Section1") == 0)
 {
   some_int = XMARK1_UINT_CAST(Param1,it);

   unsigned int* int_array = XMARK1_UINTARRAY_CAST(Param2,it)->getParam();
   for(unsigned int i=0;i<XMARK1_UINTARRAY_CAST(Param2,it)->getSize();++i)
   {
     ids.push( int_array[i] );
   }
 }

 //Node information
 if(section_type.compare("Section2") == 0)
 {
   s = XMARK1_STRING_CAST(Param1,it);
 }
}
```

The listing should be self explanatory for the most part. One should note that e.g. *XMARK1_UINT_CAST* represents a parametrized macro for stack allocated iterators, for heap based iterators (i.e. iterator pointers) one should use *XMARK1_UINT_CAST_H*. Let us take a look at writing a config file

```
//create the section
XMLConfigMarkISection section;

//the parameter
XMLConfigMarkIUInt parameter("param1",45333);

section.addParameter(parameter);

ConfigWriterMarkI conf_writer;
conf_writer.addSection( it.getElement()->getm_data() );


conf_writer.writeConfig("output.xml");
```

IX. Visualization

Before introducing any code examples we must briefly explain the technical approach behind our visualization system. The object which represents an interface to the GUI system follows the singleton pattern. The GUI system provides a fixed amount of image viewers, video viewers and plot windows, once started it also provides various widgets for data manipulation e.g. an editor for matrices. Due to technical reasons/limitations of the Qt framework all draw events are executed in a separate thread, this situation in turn requires the use of a command queue for communication between the GUI thread and a user context. We mention this as SH provides synchronous and asynchronous calls to the GUI system, a synchronous call will wait until the draw event actually finished, whereas any asynchronous method will only send the command into the aforementioned queue and return immediately afterwards. Both approaches have benefits and drawbacks; synchronous calls ensure that e.g. a certain image has been displayed before returning control to program, yet in certain situations this may be undesired e.g. if one plans on displaying exactly one image. In the latter case it is simply unnecessary to wait for the draw event to complete, which incorporates a relatively high latency due to the frequency at which the command queue is processed. Sending commands to the GUI system is in general thread safe, i.e. the requests of multiple threads will be enqueued in the order of their arrival and will be processed sequentially. The use of asynchronous calls may not be equally adequate in the case of highly frequent draw calls; e.g. a thread may enqueue 20 draw calls which will, very likely, be processed with a certain delay and in a speed so high that only the last enqueued image becomes the visible item on the screen.

IX.1. Displaying Images

First the visualization system has to be initialized

```
WindowSystem* ws = WindowSystem::getInstance();
ws->startMainSystem(4,0,0,NULL,NULL);
```

The first three integers in the methods signature represent the amount of image viewers, video viewers and function plotters, the NULL pointers at the end will be discussed later in the context of video viewers. Let us assume we want to display an image, with an active GUI system this can be done rather easily via

```
SHCore::Image<unsigned char>* im = ... //get image pointer
ws->updateImageInImageViewerSync(0,im);
ws->showImageViewerSync(0);
```

This listing will show the image *im* in the first (of four) image viewer, furthermore we use a synchronous call to draw the image. Yet this alone is not enough in order to show the image, the mentioned image viewer must be set visible through *showImageViewerSync* (again a synchronous call).
The amount of available image viewers can not be adjusted after a call to *startMainSystem*, the window system must be stopped in order to be reinitialized through a call to *startMainSystem*, multiple calls to this method will return without problems or changes to the system.

IX.2. Plotting Data

The third parameter in *startMainSystem* determines the amount of available plot viewers. The set of plot functions is rather huge compared the amount of image viewer calls, thus we will focus only on the basic aspects and defer the reader to the Doxygen documentation for more information. We begin with the case of plotting 2-dimensional data points

```
WindowSystem* ws = WindowSystem::getInstance();
ws->startMainSystem(0,0,2,NULL,NULL);
ws->resizePlotViewerSync(640,480,0);
ws->showPlotViewerSync(0);
```

Now we create the actual data

```
FastKTuple<double> x(50);
FastKTuple<double> y(50);

for(unsigned int i=0;i<50;++i)
{
        x[i] = (double)i/50.0;
        y[i] = x[i]*x[i];
}
```

In order to plot these arrays we have to insert a graph into the corresponding viewer before adding the data

```
//add graph to first viewer
ws->addGraphSync(0);
//add data to first graph in first viewer
ws->addDataSync(&x,&y,0,0);
```

In order to plot a curve, i.e. a set of connected and formatable points we just have to exchange a single function call

```
SHCore::FastKTuple<double> x_curve(50);
SHCore::FastKTuple<double> y_curve(50);

for(unsigned int i=0;i<50;++i)
{
    x_curve[i] = (double)i/50.0;
    y_curve[i] = sin(x_curve[i]);
}

ws->showPlotViewerSync(1);
ws->addGraphSync(1);
ws->addCurveSync(&x_curve,&y_curve,1);
```

The example also indicates the availability of synchronous and asynchronous functions. The plot viewer interface also features the possibility of updating existing graphs

```
for(unsigned int i=0;i<5000;++i)
{
  double x = (double)i/5000.0;
  double y = x*x;

  ws->addDataRealTimeSync(x,y,0,0);

  Toolbox::sleep_ms(50);
}
```

This will continously add points to an existing graph, currently this can not be applied to curves (this limitation will be lifted in future versions of the framework). The adding of more points into a graph raises the question of how many points remain visible after some time, let alone the choice of visible points. The graph will start to scroll in order to display the last N added data points, the scroll width is implicitly determined by a member attribute called *realTimeWidth* which in turn can be adjusted through *setRealTimeWidth* (the default value is 0.1), the value determines the visible segment size along the x-axis. The formatting of plotted data, be it simple graphs or curves, can be done visually in the plot viewer.

IX.3. Viewing Video Streams

The facilities for displaying video streams are currently in a kind of infant stage, although it is possible to display such streams, it is rather rudimentary in its technical approach. This is very likely to change in future SH versions and for the sake of completion we will briefly explain the current interface and technical background. The second parameter in *startMainSystem* determines the amount video viewers

```
SHVisualization::WindowSystem* ws = SHVisualization::WindowSystem::getInstance();

SHCore::KTuple<unsigned int>* video_sizes = new SHCore::KTuple<unsigned int>(2);
video_sizes->setElement(0,1600);
video_sizes->setElement(1,1200);

SHCore::CommunicationHeap* c_heap = new SHCore::CommunicationHeap();

//start the window system
printf("starting the window system\n");
ws->startMainSystem(0,1,0,c_heap,video_sizes);
ws->showVideoViewer(0);
```

whereas the last two parameters determine the video source and video size. The definition of the video size should be self explanatory, the tuple contains two numbers (width and height) for each video source. In our case we have only a single video source, represented by a communication heap. The display strategy is very simple, each video viewer will poll the communication heap with a certain frequency and try to obtain an available image object.

```
for(unsigned int i=0;i<60;i++)
{
  //get an image
  im = ...
  //wrap it into a com heap object
  CommunicationHeapElement_Image heap_image =
    new CommunicationHeapElement_Image<unsigned char>(im);

  //dump it on the heap
  c_heap->dumpOnHeap_callingSerialize(heap_image);

  //wait some time
  Toolbox::sleep_ms(100);
}
//clean up
ws->stopWindowSystem();
delete video_sizes;
delete c_heap;
```

This for-loop will dump an image onto the communication heap, the method *dumpOnHeap_callingSerialize* will call the given objects serialize method before obtaining its serialization buffer. The given object will be claimed by the heap, any existing object on the heap will be deleted. This is the default behavior of a communication heap, in order to change that one has to extend the communication heap class and overwrite the "dump" methods. Just mentioned at the beginning of the section this approach is very naive, by default the communication heap features no (ring)buffer for storing images which haven't

been displayed so far.
What about the case in which multiple video sources exist? The method *startMainSystem* actually expects an array of communication heap objects, the amount of video viewers determines the amount of expected objects in this array. Each video viewer will start in a separate thread.

X. Building a Multithreaded TCP Server

Writing a high performance multithreaded TCP server is everything but easy! In this chapter we will not only describe how to implement a working such a server, we will also introduce the problems which naturally occur when attempting such an endeavor. Firstly it is important to understand the class hierarchy of a TCPServer, in order to facilitate that, this chapter provides a Top-Down explanation of the servers design. Furthermore we assume the reader to be familiar with basic network programming and shorten our listings by omitting namespaces and long comments. Most variable names already give a strong hint about the variables functional purpose. The first section will introduce the internal mechanisms of a TCP server while the second section introduces introduces a comfortable wrapper concept for the previously described server concept.

X.1. General Concept of a TCP server

The class *TCPServer* extends the thread class and already implements all required methods in order to be run as a thread. It provides methods to register callbacks, *setConnectCallback* and *setDisconnectCallback*, for the event of client connects and disconnects, respectively. Yet it provides no logic for connection registration and connection handling. In order to fully understand the more abstract classes which build upon *TCPServer*, we will now explain how to write the required logic. Let us assume we extend *TCPServer* with a class *ExtTCPServer*, which implements the methods *listener_method* and *createWorkerThreads*. The body of *listener_method* will handle all incoming connections

```
void ExTCPServer::listener_method(){
int socket_fd;
struct sockaddr_in connection_data;
unsigned int size = sizeof(sockaddr_in);

m_is_active = true;

//listen for new connections
if(listen(mp_socket->get_m_socket_fd(),
   ClusterLibServer::LISTENER_QUEUE_SIZE)==-1)
{
        printf("ERROR: could not setup a listener socket\n");
        exit(1);
}

while(m_shutdown_flag == false)
{

//wait until connections arrive (event based)
int waiting_event_count = epoll_wait(m_epoll_in_fd,
  mp_active_in_events,
  ClusterLibServer::MAX_EPOLL_EVENTS, -1);

//process each waiting event
for(int i=0;i<waiting_event_count;i++)
{
//in case of the listener socket
if(mp_active_in_events[i].data.fd ==
  this->mp_socket->get_m_socket_fd())
{
//check if socket has been closed
if ( (mp_active_in_events[i].events &
     (EPOLLRDHUP | EPOLLHUP)) != 0)
{
        printf("ERROR: listener socket is closed -" \
           "shutting down server\n");
        m_shutdown_flag = true;
        m_thread_self_terminated = true;
        m_is_active = false;
        return;
}

size = sizeof(connection_data);
socket_fd = accept(mp_socket->get_m_socket_fd(),
  (struct sockaddr*)&connection_data,&size);

if(socket_fd == -1)
{
        printf("stopping listening: error on accept" \
          "or socket closed\n");
        continue;
}

//printf("connection accepted with FD
//create a (non-autonomous) connection from socket data
//TCPConnection* connection = new TCPConnection(0,false);
TCPSocket* tcpsocket = new TCPSocket(socket_fd);

//fill the socket information into the socket object
char* c_strbuffer = new char[ 500 ];
inet_ntop(AF_INET,&connection_data.sin_addr,c_strbuffer,500);

tcpsocket->m_address_type = Socket::ADDRESS_TYPE_IPV4;
tcpsocket->m_local_address = m_local_address;
tcpsocket->m_local_port = m_local_port;
tcpsocket->m_remote_address = std::string(c_strbuffer);
tcpsocket->m_remote_port = ntohs(connection_data.sin_port);

//free the cstring buffer
delete[] c_strbuffer;

//********** hand the connection over to a worker thread ***********
long int t_index = findNextBestWorkerThread();
if(t_index != -1)
{
        mp_worker_threads[t_index]->addConnection(tcpsocket);
}
else
{
        tcpsocket->closeSocket();
        delete tcpsocket;
}

}

if(mp_active_in_events[i].data.fd == m_epoll_read_poker)
{
        //close the main socket
        mp_socket->closeSocket();

        m_shutdown_flag = true;
        m_thread_self_terminated = true;
        m_is_active = false;
        return;
}
}
} }
```

It becomes clear that the server incorporates an epoll system which contains the listener sockets file descriptor. The listener method is called by the starting thread, it exits only once the listener sockets stops functioning or in case of a triggered stop. Regarding the latter situation; the server stops controlled by "pocking" himself (i.e. the epoll system) awake after setting the boolean variables to their shutdown values. Yet let us discuss the details, the listener socket is set to non-blocking mode, thus a call to *listen* will return immediately. Inside the while loop *epoll_wait* ensures that only in case of waiting incoming connections the while loop will actually proceed. If one or more events are waiting the

inner for loop will handle them; as *accept* returns a new socket descriptor, it is swiftly used to instantiate a TCPSocket which in turn is handed over to the next best worker thread, via the worker threads *addConnection* method. The listings last fragment shows the "poker" passage, the epoll system contains an internal "dummy" descriptor, to which the method *stopServer* writes, this will induce the servers shutdown.
So far we haven't discussed the creation of worker threads, which we will now catch up to. The following listing shows an example of how to implement *createWorkerThreads*

```
void ExTCPServer::createWorkerThreads() {
for(unsigned int i=0;i<m_worker_thread_count;i++)
{
//Create worker thread, update the pokers and start the thread
mp_worker_threads[i] = new
  ExWorkerThread(
        m_max_connections / m_worker_thread_count,
        (ExComFacility*)mp_cn_facility);

mp_worker_threads[i]->setDisconnectCallback(
          this->mp_disconnect_callback );
mp_worker_threads[i]->setConnectCallback(
          this->mp_connect_callback );
mp_worker_threads[i]->setComFacility(
          mp_cn_node_facility);
mp_worker_threads[i]->start(i);
} }
```

Although very short this listing depicts some very important facts. The class *TCPServer* has an array of *TCPServerWorkerThread* pointers, which yet does not point to any object, the servers *createWorkerThreads* class has to to execute this job. Using this strategy one can provide custom worker threads to his server. On closer inspection one notices that the servers connect/disconnect callbacks are handed over to the worker thread, this is necessary as the worker thread is the responsible connection handler (i.e. it must ensure that the methods get called in the respective situations, but depending on ones server design this might not be required at all). Another curious element emerges in the upper listing, a class called *ExComFacility*, which plays a very crucial role in the servers functionality. Yet, first things first, we still have to actually implement a worker thread class called *ExWorkerThread*.
The class *ExWorkerThread* will extend *TCPServerWorkerThread*, which in turn is an extension of *Thread* (i.e. it will feature another asynchronous context). In order to instantiate objects one has to implement the method *addConnection*, which might look like

```
void ExWorkerThread::addConnection(TCPSocket* tcpsocket) {
if(m_connection_count > m_max_connections)
{
```

```
printf("ERROR: maximal amount of connections already "\
 "present in worker\n");
tcpsocket->closeConnection();
DELETE_NULL_CHECKING(tcpsocket);
}

m_socket_container.externalWRLock();

/*abort if locked:
*required only if another thread calls
* TCPServerWorkerThread::killAllConnections(),
* in order to prevent new connections from being accepted*/
if(m_socket_container.m_lock_list == true)
{
tcpsocket->closeConnection();
DELETE_NULL_CHECKING(tcpsocket);
m_socket_container.externalWRUnlock();
return;
}

//register socket in local epoll system
/*we don't need a special mutex for the following operations,
as epoll is thread safe*/
struct epoll_event epoll_event;
epoll_event.events = EPOLLIN | EPOLLET |
  EPOLLRDHUP | EPOLLERR;
epoll_event.data.fd = tcpsocket->get_m_socket_fd();

int error = epoll_ctl(m_epoll_in_fd, EPOLL_CTL_ADD,
     tcpsocket->get_m_socket_fd(), &epoll_event);
if (error == -1)
{
printf("ERROR: could not register poker "\
     "within in-epoll of worker thread %d\n",
     m_local_thread_id);
exit(1);
}

//create a tcp connection
TCPConnection* connection = new TCPConnection(0,
     m_epoll_in_fd, mp_active_in_events,
     DEFAULT_RECEIVE_BUFFER_SIZE,
     DEFAULT_SEND_BUFFER_SIZE,
     CallbackServerWorkerThread::MAX_EPOLL_EVENTS, 2);

//assign the socket the new connection
connection->set_mp_socket(tcpsocket);

```

```
//assign the shared buffer
connection->setExternalBuffer(mp_send_buffer,
      mp_receive_buffer);

m_connection_count++;

//create the frame assembler
ExFrameHandler* frame_handler =
        new ExFrameHandler(connection,
            mp_callback_facility,mp_callback,this);
ClusterLibFrameAssembler* assembler =
        new ClusterLibFrameAssembler(connection,
        frame_handler);

//register the buffers
SHCore::FastVector4<SHCore::Buffer*,SHCore::Buffer*,
      FrameAssembler*,TCPConnection*>* container =
        new SHCore::FastVector4<SHCore::Buffer*,
            SHCore::Buffer*,FrameAssembler*,TCPConnection*>();

container->m_data1=connection->get_mp_receive_buffer();
container->m_data2=connection->get_mp_send_buffer();
container->m_data3=assembler;
container->m_data4=connection;

(*(m_socket_container.getMap()))
  [connection->get_mp_socket()->get_m_socket_fd()] = container;

m_socket_container.externalWRUnlock();
}
```

This is even more information to digest, let us take the above listing apart! Just in the beginning the method checks if the worker thread can even handle another connection, if not the socket is closed. The variable *m_socket_container* is a member of *TCPServer-WorkerThread*, this is the data structure *D* from the theoretical analysis in chapter II (i.e. the central limiting element in our servers connection handling)! It indexes a quadruple of send buffer, receive buffer, frame assembler and connection. technically it is a thread safe version of an STL unordered map. In this example we use the already existing frame assembler *ClusterLibFrameAssembler* which is used for communication in cluster services (we dedicated a whole chapter on frame assembling, i.e. II.3.3). The only new class is *ExFrameHandler*, which holds all server logic! Once a connection can be accepted by the worker thread, he will register the connection in his container member and assign a dedicated frame assembler to it. The assembler itself gets assigned a frame handler, whose *handlePayload* method will be called by the assembler each time a complete frame was assembled. The created TCP connection object features shared buffers as well as a shared

epoll-in system. The shared buffers should be no surprise after studying chapter II, yet the shared epoll-in system seems a bit puzzling. The reason lies in the servers structure; each worker thread handles a certain amount of connections, he will iterate through all descriptors which have available data, this data will be forwarded to the corresponding assemblers, once an assembler calls the associated frame handler it will process the data. The key point is that a frame handler is forbidden to explicitly receive data (i.e. call a receive method for the connection)! Because of that, a TCP connection does not require a dedicated epoll-in system. Keep in mind that no mechanisms are in place which prevent the programmer to call a receive method in the frame handler, if he does IT WILL RESULT IN CHAOS (never call a receive method in a frame handler, i.e. in the context of a worker thread)! The external lock calls to the container
Before venturing down any further, let us briefly recap what we have done so far. In order to create a TCP server we have extended the class *TCPServer*, which requires us to create worker thread objects that can handle the incoming TCP connection. In order to achieve this we implemented a new class *ExWorkerThread* as an extension of *TCPWorkerThread*. As its parent class is abstract we must implement *addConnection*, which in a nutshell, registers incoming connection.
We are still missing the frame handler class *ExFrameHandler*, which as an extension of *FrameHandler* requires an implemention of *handlePayload*:

```
ExFrameHandler::ExFrameHandler(
  GenericConnection*  connection,
  ClusterNodeComFacility* com_facility)
  : AssembledFrameHandler(connection,com_facility){

  /*create a frame for sending packages (this frame object
  should be used for communication only together with the
  com-facility, i.e. check if a communication is already
  in progress before sending data )*/
  mp_frame = new ClusterLibFrame(connection);

  //register the external active node list
  mp_node_com_facility = com_facility;

  //currently serving no node
  m_serving_node_id = 0;
}

int ExFrameHandler::handlePayload(Buffer* payload)
{
  payload->printfRawBuffer();
  delete payload;

  //send response
  SerializationStack data;
  data.registerElement<unsigned int>(2); //just two numbers
  data.sealStack();

  data.addUInt(3);
  data.addUInt(123);

  Buffer* buffer = data.getStackBuffer();

  mp_frame->sendFrame(buffer);

  return 0;
}
```

This is rather unspectacular but illustrates one important fact, the method *handlePayload* must dispose of the buffer! Furthermore the constructor creates a *ClusterLibFrame* for the connection, this object will provide a convenient way to send data (see *handlePayload*). The final element in our TCP server is a communication facility, which in targets a specific situation. A TCP server class usually provides functions to communicate with the connected clients. Yet in our current design there is no way for the server to access the client connections, as only the worker threads actually create connection objects. A very difficult situation arises now as we are about to handle situations with at least 3 concurrent threads; T_1 calls a method of the TCP server s_1 in order to send data to a client, T_2 waits for incoming TCP connections in data context of s_1 and T_3 (a worker thread) handles the communication with the clients. This is where the *CommunicationFacility* comes in, it provides methods which T_1 can use for client communication. The idea is to use a com facility as an abstract interface for the worker threads data structure, thus the class *TCPServer* already contains a member

```
CommunicationFacility* mp_cn_node_facility;
```

which can be used for that purpose. We will now implement a basic example called *ExComFacility* in order further explain the idea

```
class ExComFacility: public CommunicationFacility {
public:
ExComFacility(){
mp_active_nodes = new
  SHCore::ThreadSafeUnorderedMap<unsigned int, ClusterNode*>();
pthread_mutex_init(&m_mutex,NULL);
}
virtual ~ExComFacility(){
pthread_mutex_destroy(&m_mutex);
/*delete the list of node objects (the nodes are owned by the frame
handlers, thus they are not destroyed)*/
delete mp_active_nodes;
}

void addNode(ExNode* node)
{
mp_active_nodes->insertElement(node->m_node_id,node);
}

virtual int sendRequestToNode(unsigned int node_id,
  SHCore::Buffer* request, SynchronizationCallback* callback)
{
int result = 0;

//get the node
ClusterNode* node = NULL;

try
{
node = mp_active_nodes->getElement(node_id);
}
catch(const std::out_of_range &oe)
{
return -1;
}

if(node == NULL)
{
printf("ERROR: could not send data to node with id %d\n", node_id);
return -1;
}

//Wait until any active connection ceased. Afterwards lock the node for other communication.
node->lockMutex();

//Set the callback
node->setCallback(callback);

//Send data
result = node->get_mp_frame()->sendFrame(request);

//wait for result
callback->lock(node_id);

return result;
}

protected:
SHCore::ThreadSafeUnorderedMap<unsigned int,ClusterNode*>* mp_active_nodes;
pthread_mutex_t m_mutex;
}
```

An object of this class will be instantiated within the constructor of *ExTCPServer*, the com object will be forwarded up to the frame handler, which will create an internal instance of *ExNode* and insert it into the com object via *insertElement*. We haven't shown this but the strategy should be much clearer now, T_1 will indirectly access the com object through the TCP servers interface. In our example we use an unordered map to store the node pointers by their corresponding node id (i.e. an unique attribute among all nodes in all frame handlers). The call to *node->lockMutex()* will simply attempt to lock a mutex that resides in the node object, once the mutex has been acquired every attempt by another thread to communicate with the attempt will halt (this assumes that every other thread also tries to lock the mutex). Afterwards a callback object will be handed to the node, whose purpose is to provide the node with a mechanism to toggle a (callback internal) sentinel, which in turn will allow T_1 to escape from *callback->lock(node_id)*. The internal gears may look like

```
void ExSynchronizationCallback::lock(unsigned int node_id)
{
/*Now wait until the sequence ended, this will be indicated by
'callback.m_done == true' which will be set by some
framehandler. A spinlock may seem to be a wasteful approach,
yet the average waiting time is expected to be very short.
This method will introduce a 50us latency! m_done is atomic
*/
while(m_done == false)
{
  SHCore::Toolbox::sleep_us(50);
}
}

void ExSynchronizationCallback::unlock(unsigned int node_id)
{
//unlock via exiting the spin lock
m_done = true;
}
```

As described in the comment, the *unlock* function will be called by a frame handler (i.e. we must ensure that our frame handler calls it). Thus we again delegate all logic into the frame handler! Let us assume that a “Hello” dialogue consists of two messages m_1, m_2; where m_1 is sent to the client and contains the string 'Hello', while m_2 represents the clients response to the server (contains only the string 'World'). The TCP server will provide a method

```
void ExTCPServer::sendHelloToNode(unsigned int node_id)
{
//the callback
ExSynchronizationCallback comCallback;

```

```
 //send 'hello'
SerializationStack data;
data.registerString("Hello");
data.sealStack();

data.addString("hello");

Buffer* buffer = data.getStackBuffer();

//send data via facility
mp_cn_node_facility->sendRequestToNode(node_id, buffer,&comCallback);
}
```

By using the com facility T_1 will wait until any active dialogue with node *node_id* has ended. Afterwards it will lock the node and begin the communication. The logic in the frame handler will be (we simply adapt our previous code)

```
 int ExFrameHandler::handlePayload(Buffer* payload)
{
  //extract the reply
  SerializationStack data;
  data.setStackCopy(message);
  std::string s = data.getString();//World??
  delete payload;

  printf("%s\n",s.c_str());

  //the sequence is over
  mp_node->unlockMutex();

  SynchronizationCallback* sync_callback =
    mp_serving_node->getCallback();

  //this will release the lock
  sync_callback->unlock(m_serving_node_id);

  return 0;
}
```

This shows the previously mentioned node object within the frame handler, which is available to T_1 via the com facility in the TCP server. Before the message has been sent to the client, the node object received the callback object, which in turn is used by the frame handler to unblock T_1 at the dialogues end.

X.2. A TCP Callback Server

So far we have only provided a rough outline on how to write a TCP server and we haven't even discussed the client. it is easy to understand that even the most careful and experienced developer will likely spend a lot of his time with debugging the system. In order to avoid a reinvention of the wheel, SimpleHydra provides a TCP server husk called "Callback Servers". A callback server is essentially an implementation of the previously described TCP server, the only element which must be provided is the logic in the frame handler. The following example shows how to create a callback server

```
 ExCallbackServer server_callback;

 SHNetwork::CallbackServer* callback_server =
     SHNetwork::ServerFactory::getCallbackServerInstanceIPv4(
        50000/*port*/,
        1/*worker threads*/,
        100/*max connections*/,
       &server_callback);

 computation_server->startServer();


//sendHelloToNode
SHCore::SerializationStack data;
data.registerString("Hello");
data.sealStack();

data.addString("hello");

SHCore::Buffer* buffer = data.getStackBuffer();

/* This will pack 0|buffer into the send buffer and send it*/
callback_server->sendCommandSync(node_id,0,buffer);
}
```

Which without a doubt is more favorable than a development from scratch. Yet in order to understand the class *ExCallbackServer* one has to read the previous section (we assume this has been done).

```
#include <Network/GenericCallback.h>

class ExCallbackServer: public SHNetwork::GenericCallback {

ExCallbackServer(){}
~ExCallbackServer(){}

int call(SHCore::Thread* t, void* var)
{
SHNetwork::CallbackData* data = (SHNetwork::CallbackData*)var;
SHCore::Buffer* payload = data->mp_message;

//extract the reply
SHCore::SerializationStack data;
data.setStackCopy(message);
std::string s = data.getString();//World??
delete payload;

printf("%s\n",s.c_str());

//the sequence is over
mp_node->unlockMutex();

SynchronizationCallback* sync_callback =
  mp_serving_node->getCallback();

//this will release the lock
sync_callback->unlock(m_serving_node_id);

return 0;
}

}
```

This should take care of the server side. Now to the callback client

```
ExCallbackClient client_callback;

SHNetwork::CallbackClient* callback_client =
  new SHNetwork::CallbackClient(ip_of_server,port_of_server,
        node_id,&client_callback);

callback_client->run();

//wait some time
SHCore::Toolbox::sleep_ms(50000000);

callback_client->shutdown();
```

The clients constructor will attempt to connect with the provided server, afterwards the client starts his work in a separate thread. The callback is

```
#include <Network/GenericCallback.h>

class ExCallbackClient: public SHNetwork::GenericCallback {

ExCallbackClient(){}
~ExCallbackClient(){}

int call(SHCore::Thread* t, void* var)
{
SHNetwork::CallbackData* data = (SHNetwork::CallbackData*)var;
SHCore::Buffer* payload = data->mp_message;

//extract the message
SHCore::SerializationStack data;
data.setStackCopy(message);
std::string s = data.getString(); //Hello??
unsigned int type = data.getUInt(); //0??

printf("(%u):%s\n",type,s.c_str());

//send a reply
data.reset();
data.registerString("World");
data.sealStack();

data.addString("World");

Buffer* buffer = data.getStackBuffer();

mp_serving_node->sendFrame();

return 0;
}
}
```

In a nutshell a callback server requires callback objects which provide a *call()* function, this function is called once the frame handler has received a payload which contains no management information (i.e. control messages for the callback server). The same holds for the callback client. Keep in mind that the payload should not be deleted within *call*. Yet what if the roles are reversed, i.e. the client sends the 'Hello' and the server has to answer? One can simply reverse the scene and use

```
ClusterLibFrame* frame = ExCallbackClient->get_mp_frame();
```

to obtain the frame clients frame object in order to send data to the server. A strong word of advice; never mix server and client roles while using the callback classes! Always fix a communication paradigm, otherwise write your own TCP server!

XI. Setting up the framework

The process of setting up SimpleHydra is easily done via CMake. Even the most basic configuration requires several prerequisites must be fulfilled; libtomcrypt and libtomfastmath. Yet several scripts simplify the correspondig installations, we assume the sources have already been checked out into the folder "SimpleHydra". In order to install the following libraries it might be required to add the user group 'wheel' to the system, which can be done by issuing *sudo groupadd wheel*. Change into the "3rdParty" subfolder via *cd 3rdparty*, where you will find all required libraries. For the sake of completion we will explain all available libraries and the modules that depend on them. The most basic version consist only of "SHCore"; change into "libtomfastmath-master" via *cd* and issue the commands *./buildx64*, *sudo make install*. Afterwards issue *cd ..*, *cd libtomcrypt-develop*. *./buildx64* and *sudo make install*. Please ensure that "libpthread", "libX11", "libcurses", "libpanel" and "libz" are available on the system (note that only "libpthread" is mandatory, yet for convenience we will compile SH with the other libraries)! We are now ready to compile a basic version of SH; issue *cd ..*, *cd ..*, *cmake .* and *cmake-gui ..* Press "configure" in the open CMake window, make sure that:

- Only the module "Core" is selected
- BLAS_TYPE is set to "NOBLAS"
- CLMATH_TYPE is set to "NOCLMATH"
- LAPACK_TYPE is set to "NOLAPACK"
- LMSENSORS_TYPE is set to "NO_LM_SENSORS"
- MAGMA_TYPE is set to "NOMAGMA"
- VIMBA_TYPE is set to "NO_VIMBA"

Press twice on "Configure" followed by a click on "Generate" and close the window. The build system features the following build methods: "make SH" (compile SH and the embedded QT window system) and "make SHNOQT" (compile only the SH libraries and use an already compiled window system). As we are about to commence the first build we will compile the target "SH" (although it will skip the build of the window system due to the slim selection of modules). The complete list of build targets is:

- SH (includes the next 4 targets)
- SH_shared_release
- SH_static_release
- SH_shared_debug
- SH_static_debug
- SH_NOQT (includes the next 4 targets)
- SH_NOQT_shared_release
- SH_NOQT_static_release
- SH_NOQT_shared_debug
- SH_NOQT_static_debug

Once the libraries have been built we install them via *sudo make install_all*, this will install the header files into "/usr/include/SH" and the libraries into "/usr/lib/SH" (yet this can be changed in the CMake GUI). Additional installation targets are:

- install_shared_release
- install_shared_debug
- install_static_release
- install_static_debug

As mentioned above we will now briefly list the 3rd party libraries and the SH modules which depend on them:

- Cluster: Network module, Core module
- SDL: Core module, all sub-libraries within SDL2
- Visualization: Core module, QCustomPlot, QT5
- Database: Core module, libmysqlclient, boostregex
- Hardware: Core module, liblmsensors, libv4l2, VimbaSDK
- ImageProcessing: Core module, libjpeg, libpng

- MachineLearning: libSVM
- Matlab: Core module, libmat
- Network: Core module, XML module
- OpenCL: Core module, libOpenCL
- OpenCLImageProcessing: Core module, ImageProcessing module, OpenCL module
- OpenCV: Core module, Visualization module, OpenCV >=4.8
- XML: Core module, libxml2

We strongly advise to use the libraries from the “3rdParty” folder if they are available in it.

XII. Setting up a Cluster

In this chapter we will discuss how to actually use SimpleHydra for its main purpose; distributed computation, cluster computation, cluster management etc. First we will discuss how to set up the involved nodes, this is followed by a tutorial on how to create a set of distributed processes with an example of calculating a good old mandelbrot fractal!

XII.1. Preparing the Nodes

SimpleHydra needs to be installed on all nodes, note that a node can be an arbitrary Linux based system e.g. an existing workstation or a dedicated cluster node, these different node types can also be mixed together. The framework itself is merely a set of passive components which are utilized by a background daemon called *SHClientDaemon*, this daemon will handle the incoming management connections and deploy the local processes.
Let us assume that the daemon sources have been checked out into a folder called *SHClientDaemon*, the installation can be completed by simply issuing *cmake .* followed by *make* and *make install_shared_release*. The make file also features the targets *install_shared_release*, *install_static_release* and *install_static_debug*, yet these are most likely interesting only for developers. it is crucial to understand the mechanics behind the daemon, the installation will do the following things

- copy the daemon executable into /usr/bin
- copy a default configuration file into /etc/SHClient
- create a systemd profile called *SHClientDaemon*
- activate and start the systemd profile

It is wise to configure the daemon after the installation, which can be done through the file /etc/SHClient/clientConfig.xml, let us take a look at the file structure

```
<!-- An example configuration file for the cluster node management client -->
<config>

<!-- Everything which is needed to start the mgmt client -->
```

```
<section name="General">
<parameter name="RCSClientPort" type="UINT">
16001
</parameter>
<parameter name="TCPRCSServerPort" type="UINT">
16005
</parameter>
<parameter name="TCPRCSServerFrameTimeout" type="UINT">
250
</parameter>
<parameter name="TCPRCSServerListenTimeout" type="UINT">
250
</parameter>
<!-- Determines the type of used RCS services;
0 = UDP RCS Only, 1 = TCP RCS only, 2 = UDP & TCP RCS -->
<parameter name="RCSType" type="UINT">
2
</parameter>
<!-- A list of client group IDs to which this RCS client will reply -->
<!-- Predefined groups are ALL=0, ALL_OCL=1, ALL_CUDA=2, ALL_CPU_ONLY=3 -->
<parameter name="RCSClientGroups" type="UINT_ARRAY">
0,1,2,3
</parameter>
<!-- The time (in ms) a client will wait before starting
the next connection attempt to the mgmt server-->
<parameter name="MgmtClientConnectionInterval" type="UINT">
500
</parameter>
<!-- The amount of connection attempts to the mgmt server-->
<parameter name="MgmtClientConnectionAttempts" type="UINT">
500
</parameter>
<!-- This determines if the client should also use TCP RCS-->
<parameter name="UseTCPRCS" type="UINT">
0
</parameter>
<!-- The port on which a TCP RCS client will listen for connections-->
<parameter name="TCPRCSPort" type="UINT">
30000
</parameter>
</section>

<!-- The node attributes -->
<section name="System">
<!-- An unique client identifier -->
<parameter name="CNMClientName" type="STRING">
An unnamed management client
</parameter>
```

```
<!-- The desired node ID, i.e. the ID that the client wishes to obtain-->
<parameter name="DesiredNodeID" type="UINT">
2
</parameter>
<parameter name="OSName" type="STRING">
Linux
</parameter>
<parameter name="Architecture" type="STRING">
x64
</parameter>
<parameter name="SHUDeploymentPath" type="STRING">
/tmp
</parameter>
<!-- The full path to the systems checkout folder, i.e. the
folder to which, during an update, the SH sources will be checked out to. -->
<parameter name="SHCheckoutPath" type="STRING">
/tmp
</parameter>
<!-- The full path to the system database, this must not exceed
511 chars! If it doesn't exist, it will be created on runtime. -->
<parameter name="SystemDB" type="STRING">
/home/lowwang/SHDB.db
</parameter>
<!-- A list of system capabilities-->
<parameter name="SystemCapabilities" type="STRING_ARRAY">
<string>FullSH</string>
</parameter>
<!-- The amount of physical CPUs -->
<parameter name="PhysicalCPUCount" type="UINT">
1
</parameter>
<!-- The amount of cores per CPUs -->
<parameter name="CoresPerCPU" type="UINT">
8
</parameter>
<!-- The amount of HDDDevs in the next parameter -->
<parameter name="HDDCount" type="UINT">
1
</parameter>
<!-- A list of 'HDDCount' HDDDevices -->
<parameter name="HDDDevs" type="STRING_ARRAY">
<string>/dev/sda</string>
</parameter>

<!-- The amount of OpenCL devices in the client -->
<parameter name="OCLDevCount" type="UINT">
1
</parameter>
```

```
<!-- A list of OCL platform indices, one for each device, a total of 'OCLDevCount' -->
<parameter name="CLPlatforms" type="UINT_ARRAY">
0
</parameter>
<!-- A list of OCL device indices, one for each device, a total of 'OCLDevCount' -->
<parameter name="CLDevices" type="UINT_ARRAY">
0
</parameter>
<!-- A list of typenames for each OCL device -->
<parameter name="OCLTypes" type="STRING_ARRAY">
<string>7970</string>
</parameter>

<!-- The amount of CUDA devices in the client -->
<parameter name="CUDADevCount" type="UINT">
0
</parameter>
<!-- A list of typenames for each CUDA device -->
<parameter name="CUDATypes" type="STRING_ARRAY">
<string>780GTX</string>
</parameter>

</section>

</config>
```

Well that's a lot of options which shall be explained in the following sections. Before discussing them we would like to note a few things regarding control over the daemon. The daemon can be controlled through the systemd interface; start it with *sudo systemctl start SHClientDaemon*, restart via *sudo systemctl estart SHClientDaemon*, stop it with *sudo systemctl stop SHClientDaemon*, enable with *sudo systemctl enable SHClient-Daemon* and disable it with *sudo systemctl disable SHClientDaemon*. The daemon can also be started manually; stop the systemd process and call */usr/bin/SHClientDaemon /etc/SHClient/clientConfig.xml*, in case of debug versions one should start it via GDB.

XII.1.1. Starting the daemon as normal user

By default the daemon will be running with root privileges, if one desires to run it as a normal user it can be done via a small change in the file */etc/systemd/system/SH-ClientDaemon.service*. One has to change the option *User=root* to the desired value, but beware that several features as e.g. remote updating of the installed SimpleHydra version can only be executed if the daemon has root priveleges. It might also require to chose ports above 10000 in order to enable the daemon to open sockets. After each edit to the mentioned config file one has to issue *sudo systemctl reload*, otherwise the changes will not be updated.

XII.1.2. Option: RCSClientPort

When it comes to dynamic network topologies one has to rely onto the UDP-based RCS (Remote Configuration Services), SimpleHydra also features TCP-based RCS yet these will be discussed later. In a nutshell RCS are used to acquire meta information of the nodes before the main TCP management connection is established. An example: let there be 10 nodes with unknown IP addresses, the management node desires to establish a TCP connection to each of them, yet this requires the knowledge of all IP addresses (the management node might not even know how many nodes are available in the network)! The management node can use RCS for that, it will repeatedly broadcast an unfragmentable UDP frame into the local network, hopefully every node will receive this frame and reply, the node will reply with a UDP unicast to the management node, thus informing it about its IP and availability. The option *RCSClientPort* defines on which port the client will listen for these UDP broadcasts, every number below 10000 requires root privileges.

XII.1.3. Option: TCPRCSServerPort

TCP-based RCS are very similar to the UDP variant, yet they focus on another use case since there are no TCP broadcasts. Let us assume there are again 10 nodes available but the management node knows about their existence and their IP addresses. Their existence does not imply their availability for cluster computation, this is where TCP RCS come into play. The management node can attempt to connect to a TCP RCS **server** on the client, if e.g. no connection could be established it can assume that the node is unavailable. The bold faced "server" indicates the change of roles in the case of TCP RCS, UDP RCS does not use connections and the central information sink is the management node, thus the management node is called an UDP RCS server, whereas in the TCP case the client provides services and waits for connections from a management node, thus the client now assumes the role of an RCS server. Since TCP RCS provides a much more reliable way of communication it is also used for more than simple exchanges of meta data. One example is the capability of updating the SimpleHydra framework on the node. The client daemon will provide both variants, UDP and TCP RCS, thus it also requires information on which port the daemon should listen for incoming RCS TCP connections. The option *TCPRCSPort* defines that, just as with the UDP counterpart, every number below 10000 requires root priveleges for the daemon.

XII.1.4. Option: TCPRCSServerFrameTimeout

In case of TCP RCS the server will wait for the specified amount of ms for an incoming frame. The value must be greater than 0 in order to keep the server responsive and stoppable.

XII.1.5. Option: TCPRCSServerListenTimeout

This timeout value determines how long the server attempts to listen for an incoming connection, as before, in order to keep the server responsive and especially stoppable one has to set a value greater than 0.

XII.1.6. Option: RCSType

This option determines which kind of service should actually be used, valid values are 0 = UDP RCS Only, 1 = TCP RCS only, 2 = UDP & TCP RCS. Independent of the option value the daemon will start both services, the specified value determines which of both active RCS variants shall react to incoming data (UDP) or connections (TCP).

XII.1.7. Option: RCSClientGroups

The RCS client group is a very important concept in SH, each first incoming RCS frame (UDP and TCP) specifies a client group, the daemon will check if this client group is contained within the set *RCSClientGroups*. If the received client group is not contained in it; the daemon won't answer (UDP) or send a negative response (TCP). In any case the management node won't register the client as available! Predefined groups are ALL=0, ALL_OCL=1, ALL_CUDA=2, ALL_CPU_ONLY=3, which in turn should be self explanatory.

XII.1.8. Option: MgmtClientConnectionInterval

With this option we are leaving the RCS area, once the node has received a positive RCS advertisement (i.e. with a valid group ID), it will attempt to connect to the management server which resides on the management node. This option specifies the delay (ms) between the connection attempts to the server.

XII.1.9. Option: MgmtClientConnectionAttempts

In addition to *MgmtClientConnectionInterval* this option specifies how many connection attempts should be made.

XII.1.10. Option: UseTCPRCS

By default the TCP RCS are used in parallel with the UDP variant, yet if for some reason TCP RCS should not be used, it can be deactivated by setting this option to 0.

XII.1.11. Option: TCPRCSPort

The port under which the TCP RCS server will be started.

XII.1.12. Client metadata

So far we have described daemon related options, yet the client configuration also specifies a large amount of meta data about the client system. We will not discuss the corresponding data types as they are outlined in the default configuration file.

- CNMClientName: The name of the node (does not have to be unique)
- DesiredNodeID: Once the management client connects to the server it will send this node ID. If the server allows it, i.e. if it hasn't been used for another node, the node will be allowed to carry this id. If the server denies the clients wish, it will assign a valid ID to the client. Thus regarding the configuration one should try to distribute unique IDs yet it is technically not required to do so.
- OSName: An identifier for the local operating system.
- Architecture: A string which should indicate the system hardware architecture.
- SHUDeploymentPath: The folder into which a SHU will temporarily be distributed, e.g. /tmp.
- SHCheckoutPath: The folder into which the SH sources will be checked out in case of a remote update.
- SystemDB: The full path to the SimpleHydra database file, if it does not exist an empty database will be created instead.
- SystemCapabilities: A list of available SH modules/features, there are no predefined standard values.
- PhysicalCPUCount: The count of physical CPUs in the system, i.e. real CPUs and not logical ones (especially not cores).
- CoresPerCPU: the amount of cores or logical CPUs per physical one.
- HDDCount: The amount of available HDDs.
- HDDDevs: A list of HDD device files (full path), the number of entries must correspond to the value of *HDDCount*.
- OCLDevCount: The amount of OpenCL capable devices.

- CLPlatforms: An integer list of OpenCL platform indices, one entry for each OpenCL device
- CLDevices: The corresponding list of device indices for each OpenCL device.
- OCLTypes: A string list of OpenCL device names, the strings can be arbitrary descriptions. One entry per OpenCL device.
- CUDADevCount: The amount of CUDA capable devices in the system.
- CUDATypes: The NVidia analogon to *OCLTypes*.

XII.2. Creating SimpleHydra Units

Although it induces a step learning curve we decided to explain the usage of SimpleHydra with a real world example, i.e. we don't play the "hello world" game in this chapter. In order to understand the following sections is absolutely mandatory to understand callback servers/clients and the SHU concept (both topics have been explained before, yet we will deeply elaborate on SHUs). How should the following sections be read and how are they outlined?
First things first! As we venture into our example we will greatly expand on the involved subconcepts, each section marked with "*" is considered as an additional information source which is not necessarily required to understand the following topics. It is highly advisable to read the following sections multiple times and experiment with the provided example.
In this example we will compute the classical Mandelbrot fractal on multiple CPUs, the reader should obtain the example projects called *SHMandelbrotDemoUnit* (SHU) and *SHMandelbrotDemoApp* (application for the management node). We assume the projects have been put into equally named folders, thus let's change into *SHMandelbrotDemoUnit* and discuss the folders structure.

XII.2.1. The Mandelbrot SHU

The subfolders *data*, *DeploymentScripts*, *SHUnits* and *src* are typical for SHUnit projects and must exist in any case! The data folder should be used for small amounts of static data as it will be distributed with the SHU file, the term small is very relative and should be quantified with respect to the SHU processing (see the later chapter for a detailed description of SHU deployment). The directory *DeploymentScripts* contains bash scripts which manage the deployment process on the node, the compiled SHU files will be put into the folder *SHUnits* while the actual source files reside in *src*.

For now let's focus on the SHUs source files, within *src* and besides the CMake script we find the C++ source file *Main.cpp* which contains

```
int main(int argc, char* argv[])
{
  SHU_INIT

  //----------------- start the computation client
  //the server logic is contained within this callback object
  WorkCallbackClient client_callback;

  printf("Going to connect to %s : %u\n",shu_server_ip.c_str(),shu_server_port);
  SHNetwork::CallbackClient* callback_client =
      new SHNetwork::CallbackClient(shu_server_ip,shu_server_port,
        shu_com_struct->m_node_id,&client_callback);

  callback_client->run();

  //--------------------
  SHU_CLOSE

  return 0;
}
```

One should note that we do not omit the namespaces in this and following listings, as we are discussing a real world application. Each SHU's entry point, i.e. the main function, must start with a call of the macro *SHU_INIT* and end with a call to *SHU_CLOSE*! All program logic must fully reside between these two calls. The Mandelbrot SHU uses a standard callback server/client paradigm for communication, the client logic is fully contained within *WorkCallbackClient* which resides in the file *NetworkCommunication/-WorkCallbackClient*. A corresponding object called *client_callback* is created and passed to the instance of the callback client. The first interesting thing are the additionaly passed parameters *shu_server* and *shu_server_port*, since they apparently haven't been declared in the main function. The declaration is hidden in the *SHU_INIT* macro which features even more useful variables (see the corresponding section)! The variables content is indicated by their name, *shu_server_ip* provides the IP address of the management server while *shu_server_port* provides the corresponding port. The variables are filled "within" the macro, we won't bother with the technical details in this section. We remind the reader that a callback client is designed to be run as a thread, yet in this example we do not call the classes start method, instead we directly call "run()", i.e. no additional thread is started! The rationale behind this approach is that once the client's state machine exits, the SHU will commence a graceful shutdown.
Before continuing to the callback logic let us summarize the targeted idea. The SH management daemon on each computation node establishes a connection with the man-

agement node, the management node will send a SHU file to each connected node, once the a node receives the the complete SHU it will commence the deployment. The deployment consists of compiling the SHU content into an executable and its start as a new process which is fully controlled by the management daemon. In our example this process features the logic indicated by the previous listing.
We now venture into the callback clients logic, the idea can be quickly summarized; once the connection has been established the client will receive a parameter set of which region of the fractal he should compute, the computation itself is splitted into so called *tiles* i.e. rectangular and disjoint areas of the designated fractal space. This can be further expressed by concrete numbers, let's assume we attempt to calculate the fractal in an image area of 4096x2160, each of 4 nodes will calculate a fractal space with dimension 1024x540 which in turn shall be splitted into tiles of size 30x30 (we allow fragmented tiles!). Once a node has computed a tile it will transmit this result to the management node, which in turn will acknowledge the received data, once the client received the acknowledgment it will begin to compute the next tile. On the side of the management node the received tiles will be placed in a queue and periodically dequeued for live printing into an image viewer.
The NetworkCommunication folder also contains the callback servers header file, as it provides the definition of message types

```
enum WORK_COM_REQUEST_TYPE{WORK_COM_REQUEST_TYPE_INCOMING_DATA_UNIT=
  SHNetwork::CallbackServer::RESERVED_TYPE_SIZE,
  WORK_COM_REQUEST_TYPE_START_PROCESSING,
  WORK_COM_REQUEST_TYPE_INCOMING_RESULT_UNIT, WORK_COM_REQUEST_TYPE_OK,
  WORK_COM_REQUEST_TYPE_SET_REGION};
```

Note the definition of the enums start value, we remind the reader that this is mandatory for a successful message parsing (see chapter X). It also declares the communication structs

```
struct MandelFragment
{
  unsigned int m_global_pos_x;
  unsigned int m_global_pos_y;
};

struct MandelFragmentQueueElement
{
  unsigned int m_global_pos_x;
  unsigned int m_global_pos_y;
  SHCore::Image<unsigned char>* mp_image;
};
```

The usage of these structures will be explained alongside the following listing, at this point it suffices to say that the first struct is directly used in the communication while the second one is only used with the management nodes queue.

The callback client features a set of internal variables which we will not explain in detail, it uses a method called *createPalette* in order to compute the color palette for the fractal and a method *calculatePatch* to actually calculate an image tile. We will not elaborate on the computation itself and consider these methods as black boxes, our main focus lies on the method *call*. It starts with the canonical message extraction

```
SHNetwork::CallbackData* data = (SHNetwork::CallbackData*)var;

unsigned int type_size =
  sizeof(enum WorkCallbackServer::WORK_COM_REQUEST_TYPE);
enum WorkCallbackServer::WORK_COM_REQUEST_TYPE type =
  (enum WorkCallbackServer::WORK_COM_REQUEST_TYPE)
     SHNetwork::CallbackServer::getCRequestType(data->mp_message, type_size);
```

Afterwards the method enters a long switch statement which itself only distinguishes between two options, the chronologically first being *WorkCallbackServer::WORK_COM_REQUEST_TYPE*
The first message the client will receive contains the parameters for the upcoming computation task e.g. fractal area and tile size. The corresponding case-block begins with a rather unspectacular data extraction which is followed by a reply to the server (type *SHNetwork::CallbackServer::CALLBACK_SERVER_REQUEST_TYPE_OK*), additionally the color palette is created after the hopefully successful response.

```
//unpack the data
SHCore::Buffer* data_unit = new SHCore::Buffer(
  data->mp_message->get_mp_data()+type_size,
  data->mp_message->get_m_length()-type_size,true);

/*Some lines with data extraction procedures*/
/*.....
.....*/

//acknowledge the regions
SHCore::Buffer response(sizeof(
   SHNetwork::CallbackServer::CALLBACK_SERVER_REQUEST_TYPE));
SHNetwork::CallbackServer::setCRequestType(&response,
   SHNetwork::CallbackServer::CALLBACK_SERVER_REQUEST_TYPE_OK, type_size);
data->mp_frame->sendFrame(&response);

createPalette();

break;
```

Thus the first steps on client side are:

1. Receive a parameter frame of type *WorkCallbackServer::WORK_COM_REQUEST_TYPE_SET_REGION*.

2. Reply to the server with frame of type *SHNetwork::CallbackServer::CALLBACK_SERVER_REQUEST_TYPE_OK*.
3. Calculate the color palette.

Once the server received the clients response it will send a frame of type *WorkCallbackServer::WORK_COM_REQUEST_TYPE_START_PROCESSING*, which will be processed by the second case-block, which contains the following simplified code

```
/*unspectacular calculations*/

//calculate the patches
for(unsigned int i=0;i<fragment_count_x;++i)
{
  for(unsigned int j=0;j<fragment_count_y;++j)
  {
    calculatePatch(...);
    mp_patch_image->serialize();

    SHCore::Buffer response(type_size+
      sizeof(struct MandelFragment) +
      mp_patch_image->get_mp_serialized_data_buffer()->get_m_length());

    /*set the type*/
    SHNetwork::CallbackServer::setCRequestType(&response,
     WorkCallbackServer::WORK_COM_REQUEST_TYPE_INCOMING_RESULT_UNIT, type_size);

    /*fill the MandelFragment struct*/

    /*copy the structs contents into the response*/

    /*copy the serialized image into the response*/

    data->mp_frame->sendFrame(&response);

    /wait for OK
    SHCore::Buffer* ack = data->mp_frame->readFrame();
    int ack_ = SHNetwork::CallbackServer::getCRequestType(ack,
      sizeof(WorkCallbackServer::WORK_COM_REQUEST_TYPE));
    if(ack_ != WorkCallbackServer::WORK_COM_REQUEST_TYPE_OK)
    {
        printf("Send result was not acknowledged %u instead of %u\n",
          ack_,WorkCallbackServer::WORK_COM_REQUEST_TYPE_OK);
    }
    delete ack;
  }
}
```

The final part of the communication protocol can be summarized as

1. Wait for start frame of type
   WorkCallbackServer::WORK_COM_REQUEST_TYPE_START_PROCESSING
2. Compute an image tile t_i.
3. Send t_i to the server.
4. Wait for an acknowledgement from the server, i.e. a frame of type

   textitWorkCallbackServer::WORK_COM_REQUEST_TYPE_OK
5. If any tiles are left to compute go back to step 2).

Although very simplistic in its nature, the protocol shows several important concepts in designing a communication in the callback server/client paradigm. At first one should note the sequentially called function *readFrame*, this would not be possible on the server side because such a call might consume events which are designated for other connections (the connections share the same epoll system for incoming network events). Furthermore the client enters an semi-autonomous state in which he transmits the computed data and continues to do so after receiving a receive acknowledgment. This could be enhanced by omitting the receive acknowledgment, the reason for the implemented approach lies in the possibility for the server to control the data flow from each client (i.e. congestion control).
Let us briefly talk about some technicalities which happen in the background. Once a SHU gets fully deployed, i.e. the sources are compiled and the executable is started as a new system process, it can live at most until the connection between management client and server is closed. The daemon will ensure that no deployed SHU resides on the system once the connection ended! Although this approach ensures a consistent and clean system state it also induces some subtle problems. In case of an active SHU the corresponding process will be "brutally" terminated once the management connection breaks down, this might result in e.g. data loses or hanging server applications. Thus the developer must account for such situations in the design of his protocol. It is entirely legit to let the SHU process die before actually tearing down the management connection.
The SHU can be compiled by issuing *cmake .* followed by *make all_units* in the base directory, this will create all four types (static debug, static release, shared release and shared debug) of the SHU in the mentioned folder. Additionally it is possible to build the binary directly through a call of *make*, this won't create any SHU files yet enables one to debug the SHU process locally e.g. by running it in a GDB session. It is highly advisable to take an existing SHU project, e.g. this one, and adapt it to the specific needs, this of course requires some basic knowledge about CMake. If one desires to e.g. create a deployment for JavaScript-based SHUs (i.e. client applications written in JavaScript), we strongly suggest the study of sections XII.2.2, XII.4 and XII.2.3.

XII.2.2. SHU Compilation *

A SimpleHydra Unit file is compiled by a toolchain of CMake and bash scripts. The CMake target for the shared release SHU looks like

```
 add_custom_target(unit_shared_debug
COMMAND -mkdir "/tmp/SHUnits"
COMMAND -mkdir "/tmp/SHUnits/tempsrc"
COMMAND cp -f -r "${PROJECT_SOURCE_DIR}/*" "/tmp/SHUnits/tempsrc"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/bin"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/DeploymentScripts"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/SHUnits"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/CMakeFiles"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/CMakeCache.txt"
COMMAND rm -rf "/tmp/SHUnits/tempsrc/Makefile"
COMMAND cp -f "${PROJECT_SOURCE_DIR}/DeploymentScripts/deployment \
  _shared_debug.sh" "/tmp/SHUnits/tempsrc/deployment.sh"
COMMAND cp -f "${PROJECT_SOURCE_DIR}/DeploymentScripts/rapid \
  _deployment_shared_debug.sh" "/tmp/SHUnits/tempsrc/rapid_deployment.sh"
COMMAND sh "${PROJECT_SOURCE_DIR}/SHUnits/_deploy_helper.sh" 0
COMMAND rm -rf "/tmp/SHUnits/tempsrc"
COMMAND cp -f "/tmp/SHUnits/*.tar.gz" "${PROJECT_SOURCE_DIR}/SHUnits"
COMMAND rm -rf "/tmp/SHUnits" )
```

One can see that it relies on access to the /tmp folder where it creates a temporary directory called *SHUnits*. The following lines are self explanatory; up to line 10 the *tempsrc* folder will contain only the *src* and *data* folder and the main CMakeLists.txt. In line 11 the bash script *deployment_shared_debug.sh* will be copied to the tempsrc folder as well, yet its name will be changed to *deployment.sh*. This will be the main deployment script, all required logic is contained in it. We will discuss the internal mechanics later on, for now we continue to analyze the CMake target from above. Line 12 copies another script, *rapid_deployment*, into the tempsrc folder, this script is very similar to the previous one, yet it will be used for deploying cached SHUs. The 13th line in the target calls the script *_deploy_helper.sh* with the parameter 0, looking into the script one finds the following switch-block

```
 case $1 in
0 )
    GZIP=-9 tar -zcvf ../sh_unit_shared_debug.tar.gz ./
    echo "0"
    ;;
1 )
```

```
    GZIP=-9 tar -zcvf ../sh_unit_shared_release.tar.gz ./
    echo "1"
    ;;
2 )
    GZIP=-9 tar -zcvf ../sh_unit_static_debug.tar.gz ./
    echo "2"
    ;;
3 )
    GZIP=-9 tar -zcvf ../sh_unit_static_release.tar.gz ./
    echo "3"
    ;;
* )
    echo "Unknown option"
    ;;
esac
```

Thus the script packs and deflates the contens of *tempsrc* into corresponding archives, which will be saved in the parent directory of *tempsrc* i.e. *SHUnits*. Once this has been done the CMake target will delete the directory *tempsrc* and copy the created archive into the projects *SHUnits* folder, afterwards the temporary directory will be deleted from /tmp.

XII.2.3. SHU Deployment *

A SHU deployment is a two phase process; firstly the SHU file must be extracted on the node and the deployment script must be involved, secondly the compiled application must register itself in the nodes database and the shared memory area of the client daemon. Additionally one has to distinguish between an initial deployment and a rapid deployment, which in turn can be asynchronous or synchronized! Let us first discuss initial deployment (be advised that the following listings might contain memory leaks due to simplification for this documentation).

```
//register the shu and the shu server within the shared memory object
struct SharedMemSHUComStruct* shu_mem_object =
  (struct SharedMemSHUComStruct*)mp_shared_mem_raw;
shu_mem_object->m_shu_server_port = shu_serverport; //shu server port
shu_mem_object->m_shu_pid = 0;
shu_mem_object->m_rapid_deployment = 0;
shu_mem_object->m_synchronized_deployment = 0;
int l = std::min(512,(int)(mp_node->m_system_db_file.size()+1));
memcpy(shu_mem_object->m_system_db_filename,
  mp_node->m_system_db_file.c_str(),l); //sh db
l = std::min(512,(int)(mp_node->m_shu_deployment_path.size()+1));
```

```
memcpy(shu_mem_object->m_system_deployment_path,
  mp_node->m_shu_deployment_path.c_str(),l); //deployment path
memcpy(shu_mem_object->m_shu_id,shu_name,shu_name_length); //shu id
memcpy(shu_mem_object->mp_shu_server_ip,
  shu_server_ip,shu_server_ip_length); //shu server ip

//extract the unit
std::string command =
  std::string("tar -C")+ path + std::string(" -xzvf ")+ file ;
system(command.c_str());

//delete the archive
command = std::string("rm ") + file ;
system(command.c_str());

//execute the deployment script in the background
command = std::string("sh ") +
  path + std::string("/deployment.sh &");
system(command.c_str());
```

This listing is a simplified fragment from the file *Network/TCPClient/ClusterNodeManagementClient.cpp*, it illustrates the logic behind the deployment. Once the SHU file will be transferred with additional meta data regarding the server counterpart e.g. IP, port and SHU name, this information is saved in the shared memory segment of the client daemon and will be accessed by the SHU process later on. Afterwards the archive is extracted into the nodes deployment path, the archive will be removed and the deployment script *deployment.sh* will be executed. The listing shows that this last step actually backgrounds the script execution i.e. the *system* call returns immediately. That's the position where we distinguish synchronous and asynchronous (as shown in the listing) deployments, the corresponding synchronous call has the form

```
//execute the deployment script in the background
command = std::string("sh ") + path + std::string("/deployment.sh &");
system(command.c_str());

//wait until it started up
int res = mp_shu_deployment_semaphore->decSemaphore();
if(res == -1)
{
  printf("ERROR: could not decrement the semaphore\n");
}

m_shu_active = true;

//send a corresponding answer to the server
SHCore::Buffer response(type_size);
ClusterNodeManagementServer::setRequestType(&response,
  ClusterNodeManagementServer::CLUSTERNODE_MANAGEMENT_REQUEST_TYPE_SHU_OK);
mp_frame->sendFrame(&response);
```

The script is still sent to the background, yet a semaphore in the daemons shared memory is used by the daemon to recognize when the SHU process has started, i.e. the SHU process will increment the semaphore. This is the point at which we focus on the deployment script

```
cmake .
make
GZIP=-9 tar cvfzp rapid.tar.gz rapid_deployment.sh bin data
cd bin/shared/debug
./simpleHydraUnitTemplate_shared_debug
```

which calls CMake in order to generate the makefile, compiles the sources through a call to make and curiously packs the binary in a compressed archive. Finally it calls the compiled binary and thus starts the SHU process. As mentioned before, the first action in a SHUs main function must be the call of *SHU_INIT* which contains the following logic

```
SHCore::SharedMemoryPOSIX shared_memory(0777); \
int res____ = shared_memory.attachMemory(
        SHNetwork::ClusterNodeManagementClient::SHU_SHARED_MEM_NAME, \
        sizeof(struct SHNetwork::SharedMemSHUComStruct)); \
if(res____ == -1) \
{ \
  printf("ERROR: could not attach shared memory for shu com struct\n"); \
} \
\
struct SHNetwork::SharedMemSHUComStruct* shu_com_struct =
  (struct SHNetwork::SharedMemSHUComStruct*)
  shared_memory.getAttachedMemory(); \
\
/*get the semaphore*/ \
SHCore::SharedSemaphore shu_sem(&shared_memory); \
res____ =
  shu_sem.loadSemaphore((unsigned char*)&(shu_com_struct->m_deployment_sem)); \
if(res____ == -1) \
{ \
  printf("ERROR: could not load semaphore from shared memory for shu" \
    "com struct\n"); \
} \
\
/*register within the com struct*/ \
SHCore::Process* self_process = SHCore::Process::getSelf(); \
shu_com_struct->m_shu_pid = self_process->m_pid; \
\
/*get shu server information*/ \
unsigned int node_id = shu_com_struct->m_node_id; \
std::string shu_server_ip((const char*)shu_com_struct->mp_shu_server_ip); \
unsigned int shu_server_port = shu_com_struct->m_shu_server_port; \
/*any path longer than 512 chars will be truncated*/
int string_length = std::min((int)strlen(argv[0])+1,512); \
memcpy(shu_com_struct->m_shu_binary_filename,argv[0],string_length); \
\
if(shu_com_struct->m_rapid_deployment == 0) \
{ \
    /*save the deployed shu in the database for later rapid deployment*/ \
    SHCore::SHUDB db; \
    db.setDBPath( std::string((const char*) \
    shu_com_struct->m_system_db_filename) ); \
\
    /*delete any previous entry*/ \
    db.deleteEntry(std::string((const char*) shu_com_struct->m_shu_id )); \
\
    /*read the compiled package*/ \
    SHCore::Buffer compiled_package; \
    compiled_package.readBufferFromBinaryFile(
        std::string((const char*)shu_com_struct->m_system_deployment_path) \
        + std::string("/") + \
        std::string((const char*) shu_com_struct->m_shu_id ) \
        +std::string("/rapid.tar.gz") ); \
\
    /*add it to the db*/ \
    SHCore::SHUDBEntry entry; \
\
    entry.m_Name = std::string((const char*) shu_com_struct->m_shu_id ); \
    entry.m_BinarySize = 0; \
    entry.m_PackageSize = compiled_package.get_m_length(); \
    entry.m_PackageFilename = \
        std::string((const char*) shu_com_struct->m_shu_id ) \
        + std::string(".tar.gz"); \
    entry.m_DeploymentPath = \
        std::string((const char*)shu_com_struct->m_system_deployment_path); \
    entry.m_RequiredLibs = std::string("-"); \
    entry.mp_SHUPackage = compiled_package.get_mp_data(); \
    entry.m_packageBlobSize = entry.m_PackageSize; \
    entry.mp_Binary = 0; \
    entry.m_binaryBlobSize = 0; \
    entry.m_ConfigFileContent = std::string(""); \
    entry.m_ConfigFileName = std::string(""); \
    entry.m_VAR1 = std::string(""); \
    entry.mp_VAR2 = 0; \
    entry.m_VAR2Size = 0; \
\
    db.addEntry(&entry); \
\
    /*abandon the data pointers as they are managed by other entities,
    otherwise the destruction of 'entry' would free them*/ \
    entry.abandonData(); \
} \
\
/*indicate a finished deployment*/ \
if(shu_com_struct->m_synchronized_deployment == 1) \
{ \
  res____ = shu_sem.incSemaphore(); \
  if(res____ == -1) \
  { \
          printf("ERROR: could not increment semaphore in" \
          "shared memory for shu com struct\n"); \
  } \
}
```

First a handle to the shared memory is obtained by the system in order to access the semaphore and meta data. The SHU process will then obtain its PID through a *Process* object called *self_process*, which is also accessible after the macro call, and write it into the shared memory, thus making it available for the client daemon. The attribute *m_rapid_deployment* within the shared memory indicates if a rapid deployment should commence, in our current situation (i.e. initial deployment) this value will be set to 0 by the client daemon (see the deployment listing further above). Thus the if-clause will be entered and the SHU will be registered in the nodes SH database, this also includes the saving of the archive *rapid.tar.gz* which was created by the deployment script! Although the archive won't be deleted after the call, it will be disposed right after a reboot as it resides in the /tmp directory.
The same indicator-variable strategy is used for the synchronized deployment, the shared variable *m_synchronized_deployment* indicates if the SHU process should increment the shared semaphore. All shared data is stored in structure of type *SharedMemSHUComStruct* and accessible by the SHU process under the name *shu_com_struct* right after the macro call. As this point it should be clear that the shared memory is indeed an active communication gateway between client daemon and the active SHU process.
Let us briefly talk about the macros counterpart, the macro *SHU_CLOSE*

```
/* this step is mandatory for each SHU */ \
/*----------------------- clean up the shu information */ \
shu_com_struct->m_shu_pid = 0; \
shu_com_struct->m_shu_server_port = 0; \
memset(shu_com_struct->m_shu_binary_filename,0,512); \
shu_com_struct->m_shu_binary_filename[0] = '\0'; \
memset(shu_com_struct->mp_shu_server_ip,0,64); \
shu_com_struct->m_shu_server_ip[0] = '\0'; \
memset(shu_com_struct->m_system_db_filename,0,512); \
shu_com_struct->m_system_db_filename[0] = '\0'; \
memset(shu_com_struct->m_system_deployment_path,0,512); \
shu_com_struct->m_system_deployment_path[0] = '\0'; \
shu_com_struct->m_rapid_deployment = 0; \
shu_com_struct->m_synchronized_deployment = 0; \
res____ = shared_memory.detachMemory(); \
if(res____ == -1) \
{ \
    printf("ERROR: could not detach shared memory for shu com struct\n"); \
} \
delete self_process; \
```

Which essentially wipes all data from the shared memory. There is one subtle danger in the current approach, the client daemon detects active SHUs only via the PID in the shared memory. Should a SHU process terminate due to errors, the PID won't be updated and might be reused by some other process. If the client daemons server connection breaks, it will read the PID from shared memory and attempt to kill the corresponding process, which could obviously result in severe problems. Future versions of SH will circumvent this problem by additionally comparing the process' start time.
So far we have discussed the initial deployment, synchronized and asynchronized, let us now venture to the process of rapid deployment. The corresponding passage in the TCP cluster client (asynchronized) is

```
//get the shu from the database
SHCore::SHUDB db;
db.setDBPath( mp_node->m_system_db_file );
SHCore::SHUDBEntry* entry = db.getEntry(std::string(shu_name));

//write the file
std::string file = path + std::string("/");
file = file + std::string(shu_name) + std::string(".tar.gz");
//save the file
SHCore::Toolbox::writeArrayToFile(file,
  entry->mp_SHUPackage, entry->m_packageBlobSize);

//register the shu and the shu server within the shared memory object
struct SharedMemSHUComStruct* shu_mem_object =
  (struct SharedMemSHUComStruct*)mp_shared_mem_raw;
shu_mem_object->m_shu_server_port = shu_serverport;//shu server port
shu_mem_object->m_shu_pid = 0;
shu_mem_object->m_rapid_deployment = 1;
shu_mem_object->m_synchronized_deployment = 0;
int l = std::min(512,(int)(mp_node->m_system_db_file.size()+1));
memcpy(shu_mem_object->m_system_db_filename,
  mp_node->m_system_db_file.c_str(),l);
l = std::min(512,(int)(mp_node->m_shu_deployment_path.size()+1));
memcpy(shu_mem_object->m_system_deployment_path,
  mp_node->m_shu_deployment_path.c_str(),1);
memcpy(shu_mem_object->m_shu_id,shu_name,shu_name_length);
memcpy(shu_mem_object->mp_shu_server_ip,
  shu_server_ip,shu_server_ip_length);

//extract the unit
std::string command = std::string("tar -C")+
  path + std::string(" -xzvf ")+ file ;
system(command.c_str());

//delete the archive
command = std::string("rm ") + file ;
system(command.c_str());

//execute the deployment script in the background
command = std::string("sh ") + path +
  std::string("/rapid_deployment.sh &");
system(command.c_str());
```

The notable differences are that the SHU file is acquired from the local database, the *rapid_deployment.sh* script is called instead of *deployment.sh* and *m_rapid_deployment* is set to 1. Since all the remaining code is very similar if not identical to the initial deployment we won't further discuss it. Let us conclude this section with a brief look into the new deployment script

```
 cd bin/shared/debug
./simpleHydraUnitTemplate_shared_debug
```

which does nothing besides executing the already compiled binary.

XII.3. Creating Cluster Applications

A short recap about the concept of cluster applications might be in order at this point. A cluster application is executed on the management node and consists of a single process which is being parametrized by an XML file. This process will start the management and computation server (usually as separate threads), it will handle the cluster setup (e.g. RCS configuration) and deploy the SHUs. before discussing the Mandelbrot server application we will elaborate on the XML configuration file (which is mandatory for all server applications).

```
<!-- An example configuration file for the cluster node
management server -->
<config>

<section name="General">
```

```
<!-- An unique server identifier -->
<parameter name="CNMServerName" type="STRING">
An unnamed management server
</parameter>
<!-- The port on which the UDP RCS server will listen
for replies -->
<parameter name="RCSServerPort" type="UINT">
16000
</parameter>
<!-- The port to which the UDP RCS client will attempt to send
the beacons -->
<parameter name="RCSClientPort" type="UINT">
16001
</parameter>
<!-- The port to which the TCP RCS client will attempt
to connect -->
<parameter name="TCPRCSServerPort" type="UINT">
16005
</parameter>
<!-- The time a TCP RCS client will wait for a reply from a
TCP RCS server. -->
<parameter name="TCPRCSClientTimeout" type="INT">
250
</parameter>
<!-- Determines the type of used RCS services; 0 = UDP RCS Only,
1 = TCP RCS only. If one chooses to use TCP, the use of a TCPRCSTargetList
list becomes mandatory!
-->
<parameter name="RCSType" type="UINT">
1
</parameter>
<!-- The RCS group the server wants to connect-->
<parameter name="RCSClientGroup" type="UINT">
0
</parameter>
<!-- These settings also apply to the an IP target list; RCSBeaconCount is
the maximum for each target!-->
<!-- The amount of beacons to send, -1=continuous-->
<parameter name="RCSBeaconCount" type="INT">
10
</parameter>
<!-- The time between beacons in ms-->
<parameter name="RCSBeaconDelay" type="UINT">
50
</parameter>
<!-- The recv timeout for beacons-->
<parameter name="RCSBeaconRecvTimeout" type="UINT">
50
```

```
</parameter>
<!-- Boolean value which determines if the RCS client should register the
server once he received a beacon. This comes in handy if one decides to use
TCP RCS instead, in that case the beacons can be used a survey of reachable
machines before actually a TCP connection is being made. In other words,
this option is only relevant for UDP RCS.
-->
<parameter name="UseConnectionRequestBeacons" type="UINT">
1
</parameter>
<!-- Boolean value which determines if UDP unicast
beacons should be send instead of UDP subnet broadcasts -->
<parameter name="UseRCSTargetList" type="UINT">
0
</parameter>
<!-- A list of IPs for UDP beacon unicasts -->
<parameter name="RCSTargetList" type="STRING_ARRAY">
<string>10.2.129.184</string>
<string>10.2.129.185</string>
<string>10.2.129.186</string>
</parameter>
<!-- A list of IPs/Ports for TCP RCS, if no port is specified
the value of TCPRCSServerPort will be used -->
<parameter name="TCPRCSTargetList" type="STRING_ARRAY">
<string>10.2.129.184:16005</string>
<string>10.2.129.185</string>
<string>10.2.129.186:16005</string>
</parameter>
<parameter name="CNMServerPort" type="UINT">
10001
</parameter>
<parameter name="CNMServerWTCount" type="UINT">
1
</parameter>
<parameter name="CNMServerMaxConnections" type="UINT">
1000
</parameter>
</section>

</config>
```

The options are

- CNMServerName: A string which indentifies the server
- RCSServerPort: This is the port under which the server accepts UDP RCS replies, this number will be advertised via UDP beacons.
- RCSClientPort: The port to which UDP broadcasts will be directed.

- TCPRCSServerPort: The port to which the TCP RCS client will connect by default. This value may be overridden in the target list!
- TCPRCSClientTimeout: The time in ms which the TCP RCS client will wait for an answer from the server. This value must be greater than 0, setting a high value might result in a less responsive client.
- RCSClientGroup: The RCS group which we desire, as described before the client will only positively reply to messages that feature a group ID within their set of registered IDs.
- RCSBeaconCount: The amount of UDP beacons to send, a value of -1 indicates a continuous broadcast of beacons. The beacon count refers to a single target in case of using a target list, in which case the value must be greater than -1!
- RCSBeaconDelay: Time in ms between to consecutive beacon broadcasts.
- RCSBeaconRecvTimeout: Timeout in ms while waiting for UDP replies.
- RCSType: The RCS type to be used, valid values are 0 = UDP RCS Only, 1 = TCP RCS only. If one chooses to use TCP, the use of a *TCPRCSTargetList* list becomes mandatory!
- UseConnectionRequestBeacons: Determines if the beacons (UDP) or messages (TCP) should induce a connection attempt of the nodes management client to the corresponding network endpoint.
- UseRCSTargetList: Determines if the server should use a target list, i.e. use RCS (UDP/TCP) only for the listed IPs. Setting this option to 1 will result in unicast communication, thus UDP beacons won't be broadcasted anymore which is useful for cluster setups over different subnets or network areas. Yet one sacrifices the ability to discover unknown nodes in the network.
- RCSTargetList: The RCS list for UDP RCS.
- TCPRCSTargetList: The list for of IPs for TCP RCS, this list is mandatory for TCP RCS as no broadcast mechanics are available in TCP.
- CNMServerPort: The port on which the management server listens for incoming connections.
- CNMServerWTCount: The amount of worker threads for the management server.
- CNMServerMaxConnections: Limit for established connections.

The concept of using UDP as well as TCP RCS will become more clear in section XII.3.1 when we explain the Mandelbrot server application. We will assume that the application uses mostly default values as listed above, any exception will be mentioned explicitly.

XII.3.1. The Mandelbrot Server Application

The content of *SHMandelbrotDemoApp* looks similar to that of *SHMandelbrotDemoUnit*, there are less subdirectories yet those which remain are equally named as in the previous example. Just as before we begin with the file *Main.cpp*, which contains much more logic than before

```
/*variable declarations*/
unsigned int total_timeout = 30000;
unsigned int expected_clients = 3;
unsigned int ms_time_counter = 0;
bool sync_deploy = false;

SHCluster::ClusterServer clusterServer;

if(argc == 1)
{
  clusterServer.startClusterServerFromXML("serverConfig.xml");
}
if(argc == 2)
{
  clusterServer.startClusterServerFromXML(argv[1]);
}
```

These lines show that the application expects either a configuration file in its runtime directory or a CLI parameter pointing to the file's location. They also fully deploy the cluster management server, i.e. the RCS and management services are started according to the values in the configuration file. This is followed by a while loop which delays the processing until either the expected amount of clients (*expected_clients*) has connected to the server or the timeout *total_timeout* was exceeded.

```
//give it some time to start up and the clients to register
while(clusterServer.getMgmtServer()->getNodeCount()
  < expected_clients && ms_time_counter < total_timeout)
{
  SHCore::Toolbox::sleep_ms(500);
  ms_time_counter += 500;
}
ms_time_counter = 0;

SHCore::FastKTuple<unsigned int>* nodeIDs =
    clusterServer.getMgmtServer()->getNodeIDs();
```

```
expected_clients = nodeIDs->getm_size();
```

Afterwards the server continues with the assumption that all expected clients have connected, i.e. it updates the variable *expected_clients* to the number of connected machines.

```
int res = 0;
long long res2 = 0;
unsigned int shu_port = 20000;
std::string shu_ip = clusterServer.getMgmtServer()->getAddress();
std::string shu_file = "/root/workspace/SHMandelbrotDemoUnit/SHUnits/sh_unit_shared_debu
std::string shu_name = "TestUnit";
```

At this point we must note a technical fact from behind the scenes, the method *startClusterServerFromXML* will use the first real NIC in the system and obtain its IP. The code above shows how to query this address, additionally it shows which SHU file we are about to deploy to the nodes. The string *shu_name* defines the name under which the SHU will be registered on the clients, this identifier must be unique otherwise it might happen that an already deployed SHU will be overwritten on the client (see section XII.2.3 for details about the deployment process). We are going to start the computation server on port 20000, which is indicated by *shu_port*. With these values we can continue to the actual deployment, yet first we have to start the computation server, this is a wise decision as the deployed SHU will immediately attempt to connect to the computation server.

```
//the server logic is contained within this callback object
WorkCallbackServer server_callback;

//get the queue
SHCore::Queue<MandelFragmentQueueElement*>* queue =
  server_callback.getResultQueue();

SHNetwork::CallbackServer* computation_server =
  SHNetwork::ServerFactory::getCallbackServerInstanceIPv4(shu_port,
      1,100,&server_callback);

computation_server->startServer();
```

Here we also see the callback servers "logic object" *server_callback*, which will be discussed later. Now finally to the deployment

```
for(unsigned int i=0;i<nodeIDs->getm_size();++i)
{
  //deploy a SHU
  if(sync_deploy == false)
  {
    res = clusterServer.getMgmtServer()->sendSHUnit(nodeIDs->getElement(i),
        shu_file, shu_name,shu_ip,shu_port);
  }
  else
  {
    res = clusterServer.getMgmtServer()->sendSHUnitSync(nodeIDs->getElement(i),
            shu_file, shu_name,shu_ip,shu_port);

    //get the SHU's PID on the client (ONLY FOR SYNC DEPLOYMENTS)
    res2 = clusterServer.getMgmtServer()->getSHUPID(nodeIDs->getElement(i));
    if(res2 == -1)
    {
            printf("ERROR getting SHU PID\n");
    }
    else
    {
            printf("SHU is active on client %u with PID %lld\n",
              nodeIDs->getElement(i),res2);
    }
  }

  if(res == -1)
  {
    printf("ERROR deploying SHU\n");
  }
}
```

In this example we commence an asynchronous SHU deployment, i.e. we continue right after the dispatching the SHu file to the node, this allows us to quickly dispatch the SHU data, yet it introduces the problem of finding out if a client has deployed the unit. We address this problem with the same strategy as in the case of finding the available nodes.

```
//increase the timeout
total_timeout *= expected_clients;
 //wait until all clients connected
while( computation_server->getNodeCount() < expected_clients
  && ms_time_counter < total_timeout)
{
        SHCore::Toolbox::sleep_ms(500);
        ms_time_counter += 500;
}
ms_time_counter = 0;
printf("%u/%u computation clients registered\n",
  computation_server->getNodeCount(),expected_clients);
computation_server->printNodeIDs();
```

The code shows that we wait until either all expected clients have connected to the computation server or a timeout value has been exceeded. That was the last preparation step before starting the actual computation! Since the following steps involve a huge amount of parameter calculation we will greatly simplify the code and only discuss the important structural steps.

```
//initial deployment to all nodes
for(unsigned int j=0;j<nodeIDs->getm_size();++j)
{
  /*accumulate all required data and send it
  to the current client*/
  //......

  int res = computation_server->sendCommandSync(nodeIDs->getElement(j),
        WorkCallbackServer::WORK_COM_REQUEST_TYPE_SET_REGION,&data);

  //once the node received the data; start processing
  res = computation_server->sendCommandAsync(nodeIDs->getElement(j),
        WorkCallbackServer::WORK_COM_REQUEST_TYPE_START_PROCESSING);

  //error checks
}
```

We remind the reader of the clients callback structure, the first message the expected by the client is of type *WorkCallbackServer::WORK_COM_REQUEST_TYPE_SET_REGION*, which is indeed sent in the first place. There is a very important reason why the first message is sent in a synchronous way whereas the second message is being sent asynchronously. SimpleHydra's default frame assembler are designed for single frames and do not feature any kind of queueing mechanics for frame streams. Thus we can only send the last frame without synchronization, since it is the last message which is actively sent by the server, afterwards the client will enter a request-response loop with the server. Back to the actual messages, once the client received all computation parameters we send the "go" signal through a message of type
WorkCallbackServer::WORK_COM_REQUEST_TYPE_START_PROCESSING.

```
while(processing_counter < expected_results)
{
  //process the results obtained so far
  MandelFragmentQueueElement* client_res = queue->dequeueBlockingOnEmpty();
  /
  //unpack the data and copy it into the final image
  final_image->insertSubImageFast(client_res->mp_image,
    client_res->m_global_pos_x, client_res->m_global_pos_y);


  delete client_res->mp_image;
  delete client_res;

  if(processing_counter % update_interval == 0)
  {
    ws->updateImageInImageViewerSync(0,final_image);
  }
  ++processing_counter;
}
```

The listed while-loop runs as long as there are missing tiles, in each iteration the client will query the the queue for new results. We note that the queue is contained within the callback object and filled once data is obtained inside the objects call method, the while-loop attempts to dequeue one element in each iteration (and blocks in case of an empty queue). Each received tile will be copied into the final image. Once a total of *update_interval* tiles has been received the current partial image will be redisplayed in an image viewer.
After receiving all tiles the final image is being shown for a short time, which is followed by an "ungraceful" termination of all deployed SHUs

```
//kill all deployed SHUs
for(unsigned int j=0;j<nodeIDs->getm_size();++j)
{
  printf("Attempting to kill SHU on node %u\n",nodeIDs->getElement(j));
  //kill the SHU
  res = clusterServer.getMgmtServer()->killActiveSHU(nodeIDs->getElement(j));
  if(res == -1)
  {
    printf("ERROR terminating SHU on node %u\n");
  }
  else
  {
    printf("SHU terminated\n");
  }
}

//------------ kill the computation server
computation_server->stopServer();
delete computation_server;
```

In the end it is not surprising how much logic is contained within the main function, after all it must set up the infrastructure! We can now focus our attention onto the callback server's logic within the file *NetworkCommunication/WorkCallbackServer.cpp* and the contained member function *call*

```
/*message extraction*/
//................

switch(type)
{

  /*
   * Incoming results are always of the form
   * <serialized_MandelFragment | serialized_image>
   **/
  case WorkCallbackServer::WORK_COM_REQUEST_TYPE_INCOMING_RESULT_UNIT:
  {
  SHCore::Buffer* data_unit = new SHCore::Buffer(
      data->mp_message->get_mp_data()+type_size,
      data->mp_message->get_m_length()-type_size,true);

  //get the fragment header
  struct MandelFragment fragment;
  memcpy(&fragment, data_unit->get_mp_data(), sizeof(struct MandelFragment));

  SHCore::Buffer* serializedImage = new SHCore::Buffer(
    data_unit->get_mp_data()+sizeof(struct MandelFragment),
    data_unit->get_m_length()-sizeof(struct MandelFragment),true);

  //get the image
  SHCore::Image<unsigned char>* image = new SHCore::Image<unsigned char>();
  image->set_mp_serialized_data_buffer(serializedImage);
  image->deserialize();

  MandelFragmentQueueElement* res = new MandelFragmentQueueElement();
  res->mp_image = image;
  res->m_global_pos_x = fragment.m_global_pos_x;
  res->m_global_pos_y = fragment.m_global_pos_y;
  res->m_node_id = data->mp_serving_node->get_m_node_id();

  mp_queue->enqueue(res);

  delete data_unit;

  //send a response
  SHCore::Buffer response(sizeof(
    SHNetwork::CallbackServer::CALLBACK_SERVER_REQUEST_TYPE));
    SHNetwork::CallbackServer::setCRequestType(&response,
        WORK_COM_REQUEST_TYPE_OK, type_size);
  data->mp_frame->sendFrame(&response);

  break;
  }
}
```

Since it is a very short method we listed the complete content. The only expected message type is *WorkCallbackServer::WORK_COM_REQUEST_TYPE_INCOMING_RESULT_UNIT*, each message also contains the tile image with some meta data. Once the attached data has been extracted it is packed in a *MandelFragmentQueueElement* structure and enqueued within the callback objects result queue. The last lines show the sending of an

acknowledgment to the client.
In order to build the server application one has to issue the commands *cmake .* followed by *make*.

XII.3.2. The Neural Network Server Application

Another included demo application is the distributed training of neural networks (*SHNeuralNetworkDistApp*). We won't describe the application in too much detail since its communication structure exhibits an approach which is similar to the distributed mandelbrot application, thus we defer the reader to the (documented) source code which follows the canonical neural network implementation contained in the SHML module. We will instead briefly explain the strategy for computation distribution; the application distributes the computation by splitting the training data into equally sized fragments, one for each node. The network structure and all starting weights are deployed to all nodes, i.e. each node knows the entire network and the starting weights. The following training procedure can be summarized as

1. Send start command to all nodes.
2. All nodes compute an average gradient over their training subset.
3. All nodes send their result back to the management node, which sums up the averaged gradients and updates the network weights
4. The management node sends the updated weights to all nodes, which in turn update their local network representation.
5. The process starts back at 1) or stops if the maximal amount of iterations was reached or the maximal error was undershot.

Each node will of course split its local training set among all available computation cores (which is identical to the canonical implementation).

XII.4. Deploying Java / Python Units*

SimpleHydra features proof of concept examples for deploying Java and Python SHUs, furthermore it provides an extensible subsystem which allows external applications to access the client daemons shared memory. We strongly recommend to read sections XII.2.2 and XII.2.3 before studying this feature.
SimpleHydra is a native C/C++ framework, which makes communication with, or integration of other languages reasonably difficult. An example being data exchange, e.g. exchanging unsigned long long with a Java application (which provides no native primitive

datatype of that kind). It is of course possible but a cumbersome endeavor in general. SimpleHydras main feature is the controlled deployment of processes among a set of connected nodes, the shear size of its API makes a JNI approach unfeasible, let alone a wrapping for Python. We decided to provide a generic interface to a small subset of SH's main features, namely the controlled distribution of processes.
As described in previous sections the deployment relies heavily on access to the client daemons shared memory. Thus we developed a wrapper application *SimpleHydraSystemFacilities* which provides access to this memory segment. It can be installed through the well known calls *cmake .*, *make* and *make install*, this should result in a binary named *SHSF* located in /usr/bin. The compilation requirements are an SH installation with at least Core, Network and XML module.

XII.4.1. Java SHUs

We assume the reader has obtained the sample projects *SimpleHydraUnitJavaTemplate* amd *SimpleHydraJavaAppTemplateJava*, of which we additionally assume to be extracted in equally named folders. Let us first discuss the SHU template, the structure within *SimpleHydraUnitJavaTemplate* looks very similar to native SHUs, yet the source folder contains no C/C++ files, in the aspect of source code only a Java file called *Main.java* resides in it.

```
public class Main
{
    private interface CLibrary extends Library
    {
     CLibrary INSTANCE = (CLibrary) Native.loadLibrary("c", CLibrary.class);
     int getpid ();
    }

    public static void main (String[] args)
    {
        int pid = CLibrary.INSTANCE.getpid();
        Process child;
        String cmd;
        String line;

        System.out.println("java unit active with PID "+pid);

        try
        {
            cmd = "SHSF" + " --register-shu-pid "+pid;
            child = Runtime.getRuntime().exec(cmd);

            cmd = "SHSF" + " --print-shu-node-id";
            child = Runtime.getRuntime().exec(cmd);
            BufferedReader input = new BufferedReader(
              new InputStreamReader(child.getInputStream()));
            while((line = input.readLine()) != null)
            {
              System.out.println(line);
            }
            input.close();

            cmd = "SHSF" + " --print-shu-server-ip";
            child = Runtime.getRuntime().exec(cmd);
            input = new BufferedReader(
              new InputStreamReader(child.getInputStream()));
            while((line = input.readLine()) != null)
            {
              System.out.println(line);
            }
            input.close();

            cmd = "SHSF" + " --print-shu-server-port";
            child = Runtime.getRuntime().exec(cmd);
            input = new BufferedReader(
              new InputStreamReader(child.getInputStream()));
            while((line = input.readLine()) != null)
            {
              System.out.println(line);
            }
            input.close();

            Thread.sleep(10000);

            cmd = "SHSF" + " --unregister-shu";
            child = Runtime.getRuntime().exec(cmd);
        }
        catch(Exception e)
        {
                System.out.println(e);
        }

        System.out.println("java unit inactive");
    }
}
```

As mentioned before this is merely a proof of concept example, the Java application will first determine the PID of its corresponding Java system process (which is done via JNA and a set of jar archives included in the SHU file). The idea is to use the exact and already existing deployment routines in the *ClusterNodeManagementClient* to start and control the Java process. The client daemon exerts control by knowing the process's PID, this

is where the SH system facilities come into play. The Java code above shows a system call with the command *SHSF –register-shu-pid PID*, this will result in an update of the shared pid variable within shared memory, its value will be updated to PID. The beauty lies in the hidden logic, the tool SHSF will not only register the new PID but also indicate the deployment through the shared semaphore (in case of a desired synchronous deployment)! This strategy is applicable to any language which provides access to the systems shell and starts applications in a new system process. Keep in mind that just as with native C/C++ SHUs the first call in a Java SHU's main function must be the registration via SHSF.
Let us dive a bit into technicalities, the registration of a PID via SHSF induces a call to a parametrized macro called *SHU_INIT_EXTERNAL(PID)*. This macro is completely identical to its unparameterized counterpart *SHU_INIT*, remember that the deployment type will be indicated by the client daemon through shared variables e.g. *m_synchronized_deployment*, in this respect no communication with the SHU process is required, all required information is the SHUs PID. Even the caching of the compiled source code (in this case a jar archive) can be done transparently through the deployment scripts in combination with the mentioned macro.

```
cd ..
cd "${DIR}/src"

cp *.jar ../bin
javac -cp jna-platform-4.1.0.jar:jna-4.1.0.jar -d ../bin Main.java

GZIP=-9 tar cvfzp rapid.tar.gz rapid_deployment.sh bin data

cd ..
cd "${DIR}/bin"

java -cp jna-platform-4.1.0.jar:jna-4.1.0.jar:. Main
```

The content is similar to the native SHUs, at first the rapid deployment package will be created (after the java sources have been compiled), which will be written by the macro into the SH database. Afterwards the Java process will be started. One subtle difference exists when it comes to the compilation of Java SHUs, Java does not distinguish between e.g. static or shared builds. Thus the example only features two make targets called *unit_java* and *all_units*, if one desires to debug the Java application we strongly suggest to do it in a native Java development environment. SimpleHydra should be seen in this concept only as a way of deploying Java clients among the nodes.

XII.4.2. Java Server Application

Within *SimpleHydraJavaAppTemplateJava/src* we find two main files, *Main.java* and *Main.cpp*. The idea is to create an executable binary which deploys a controllable local Java process.

```
//........

/*the SHU params*/
unsigned int shu_port = 20000;
std::string shu_ip = clusterServer.getMgmtServer()->getAddress();
std::string shu_file =
  "/home/lowwang/workspace/SimpleHydraUnitJavaTemplate/" \
  "SHUnits/sh_unit_java.tar.gz";
std::string shu_name = "TestUnit";

//————————————- start the computation server (i.e. python script)

//at this point one could provide the script with some information
//about the available clients
SHCore::FastKTuple<std::string> params(3);

params.setElement(0, "-cp");
params.setElement(1, SHCore::Toolbox::getCurrentDirectory());
params.setElement(2, "Main");
SHCore::Process* java_proc =
  SHCore::ProcessToolbox::spawnProcess("/usr/bin/java",&params);

printf("Java process spawned with PID %d\n",java_proc->m_pid);

//wait a bit until the server started up (e.g. initialized all
//required network facilities)
SHCore::Toolbox::sleep_ms(5000);

//————————————- deploy the SHU

SHCore::FastKTuple<unsigned int>* nodeIDs =
  clusterServer.getMgmtServer()->getNodeIDs();
expected_clients = nodeIDs->getm_size();

for(unsigned int i=0;i<nodeIDs->getm_size();++i)
{
  //deploy a SHU
  res = clusterServer.getMgmtServer()->sendSHUnitSync(nodeIDs->getElement(i),
      shu_file, shu_name,shu_ip,shu_port);
  if(res == -1)
  {
    printf("ERROR deploying SHU\n");
  }

  //get the SHU's PID on the client
  res2 = clusterServer.getMgmtServer()->getSHUPID(nodeIDs->getElement(i));
  if(res2 == -1)
  {
    printf("ERROR getting SHU PID\n");
  }
  else
  {
    printf("SHU is active on client %u with PID %lld\n",
      nodeIDs->getElement(i),res2);
  }
}

//let the main process sleep until the java process finished (spinlock)
while(java_proc->doesExist() == true)
{
  SHCore::Toolbox::sleep_ms(500);
}

delete java_proc;

//------------------- kill the SHU

//kill all deployed SHUs
for(unsigned int j=0;j<nodeIDs->getm_size();++j)
{
  printf("Attempting to kill SHU on node %u\n",nodeIDs->getElement(j));
  //kill the SHU
  res = clusterServer.getMgmtServer()->killActiveSHU(nodeIDs->getElement(j));
  if(res == -1)
  {
    printf("ERROR terminating SHU on node %u\n",nodeIDs->getElement(j));
  }
  else
  {
    printf("SHU terminated\n");
  }
}
```

This code should look familiar as it follows for the most part our discussed Mandelbrot application, at least regarding the cluster setup. The difference is the type of computation server, which is started as a Java process with effectively no logic (in this example)

```
 public class Main
{
  public static void main (String[] args)
```

```
    {
        System.out.println("Hello World!");

        try{
        Thread.sleep(10000);
        }catch(Exception e)
        {
            System.out.println(e);
        }

        System.out.println("Exiting");
    }
}
```

For distributed Java based computation one might use this example and implement the required network communication, SH merely provides the rails for process delivery. Keep in mind that just as with native SH applications, once the client daemon loses the connection to the server it will kill the deployed Java SHU.

XII.4.3. Python SHUs

The deployment of python based SHUs is very similar to the Java case thus we won't discuss it in detail. The only difference comes from Pythons script nature which implies the absence of a compiled binary. This results in a rapid deployment archive (see the previous section) which is essentially the original SHU file, i.e. the rapid deployment will be identical to the initial deployment. This should become even clearer by looking at the deployment script

```
GZIP=-9 tar cvfzp rapid.tar.gz rapid_deployment.sh src data

cd ..

cd "${DIR}/src"

python Main.py
```

We provide a proof of concept in the projects *SimpleHydraUnitPythonTemplate* and *SimpleHydraClusterAppPythonTemplate*, analog to Jave case the network communication should be implemented in Python and the application itself should be developed in a Python environment.

XII.5. Docker support

It is also possible to deploy SH Docker instances, currently we do not provide any docket image which could host an SH client, yet we have successfully tested this form of deployment. A reader interested in this application should refer to e.g. [?] or one of the many tutorials regarding the setup of docker images. A common problem which we encountered was the question about port forwarding of UDP ports (e.g. for UDP RCS), the following line shows the command string in order to start the SH cluster node management client (it can not run as a daemon due to the nature of docker containers)

```
sudo docker run --name CLIENTNAME -i -p 16001:16001/udp -t base/archlinux /usr/bin/SHC
```

issuing the listed command will start an instance of the management client in a new Docker container named *CLIENTNAME* (adding '-d' and removing '-i' as a flag will execute the Docker command in the background). Once the client (and docker container) exited it is not possible to restart the management client again, yet by deleting the old (inactive) container one can bypass this restriction through

```
sudo docker rm CLIENTNAME
sudo docker run --name CLIENTNAME -i -p 16001:16001/udp -t base/archlinux /usr/bin/SHC
```

In our experiments we used a casual ArchLinux Docker distribution which was retrofitted with all required libraries in order to compile every SH module. Updating SH in an existing Docker container can be done by starting a new container with the current SH/Linux image, updating the local installation followed by e.g.

```
docker export CONTAINER-ID | docker import - base/arch
```

which will update or create the specified image.

XIII. Cluster Management Protocol

This chapter describes the network protocol for communication in the context of cluster management. As many elements in the protocol are similar to each other we will focus on explaining the corresponding technical details and avoid redundancies by only mentioning the communication strategy for similar components. A reader interested in more technical details should refer to the documented source code otherwise he might wait for the SH cluster communication whitepaper (which is planed to be released but this might take some time...).
The following section will briefly introduce the associated component and explain it from a logical and technical perspective, any special cases and design choices will be discussed along the way.

XIII.1. Sending (Asynchronous) System Commands

We distinguish between two kinds of system commands, synchronous and asynchronous ones. A synchronous command is a shell command (i.e. an arbitrary shell string) which will be executed on a cluster node, the command source (i.e. the node which issued the command) will wait until the recipient finished executing it and has returned its std output. In other words the issuing node will wait until the target returned the commands complete standard output. There is a limit to the outputs length, should it be exceeded then only the first output fraction will be returned. An asynchronous command on the other hand will not return any results, which in turn implies that the issuer will not wait for any data.

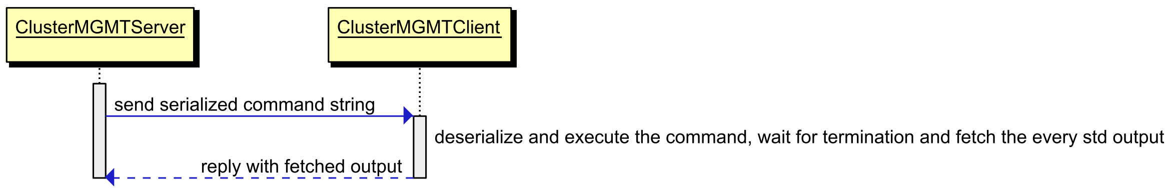

ClusterMGMTServer
ClusterMGMTClient
send serialized command string
deserialize and execute the command, wait for termination and fetch the every std output
reply with fetched output

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

Figure XIII.1 shows the logical view on the communication strategy (a simple "ping-pong" message exchange of serialized binary data) behind sending synchronous commands to the client. Yet when it comes to the technical realization one has to deal with synchronization challenges among multiple threads which are distributed among cluster nodes. Fig. XIII.2 depicts the most simple situation; one caller sends a system command to a single node, no other actors compete for communication with the same node.

We assume that a complete cluster has been set up. At first the actor (e.g. the main routine) calls the management servers *sendSystemCommand* function, which in turn will create an instance of *SingleClusterNodeSynchronizationCallback*, this helper object so_1 provides an interface which enables the server to synchronize itself (in terms of data and time) with the following communication cascade. Once the synchronization object was instantiated the server continues by calling *sendRequestToNode* from his *ClusterNode-ComFacility* member and provides so_1 as a parameter. At this point the communication facility object takes charge of handling the communication, the first step is the mutex locking of the node object (keep in mind that we are about to start a communication sequence, i.e. a temporary and virtual P2P communication with a node). Under the assumption that every communication sequence attempts the same mutex lock in the beginning we can assure that only one sequence is active at a time. Afterwards the node object will be provided with a reference to so_1 followed by obtaining the nodes frame object via *get-Frame*. The communication facility resumes its actual task by calling *sendFrame* with the corresponding serialized data (which merely contains type IDs), this function will return immediately once the data was sent over the socket. At this point the synchronization object steps into action, the communication facility synchronizes itself with the nodes response via a call to *lock* (will leads the communication facility into a spinlock wait, yet the *lock* method accepts a boolean flag which allows the use of a semaphore instead of a spinlock, for short latency communications one should omit this parameter and stick to the spinlock). The worker thread will eventually receive the nodes response and deliver it to the corresponding frame assembler (which will commence the assembling *in the worker threads context*). Once the data was assembled successfully it will be handled by the associated frame handler (still in the same thread context), which in turn delivers the data (i.e. the programs output on the node) to so_1 and stops the communication facilities (spin)lock through a call to *unlock*. Once the communication facility is allowed to continue it will return immediately with an error code which indicates if the actual transmission was successful. The server will check so_1s sequence indicator in order to determine if everything went according the protocol (i.e. if the program was executed and successfully transmitted the output). In this case the server will query the received output from so_1 (since so_1 will be destroyed when *sendSystemCommand* returns) and return it to the actor.

An asynchronous command is very similar to the synchronous call (see Fig. XIII.3), the only difference, from a technical point of view, lies in the type of response from the node, instead of receiving the commands output the server will receive a confirmation about the successfully transmitted command. Since the server will not wait for the clients successful command execution the error code returned by *sendAsyncSystemCommand* (Fig. XIII.4) can not indicate any kind of successful execution.

XIII.2. Requesting (Large) Files from a Node

The cluster node management server provides two methods to obtain files from a given node, these two methods *getFile* and *getLargeFile*. The function differ slightly regarding their signature, the first one will return the requested files content inside a buffer object, while the second function writes the returned data to a local file. The rationale behind these two different approaches is as follows; *getFile* should be used in situations where the files content is small enough in order to handle it within system memory, the second function should be used if one requires a large amount of data (e.g. ¿4Gb) from a single file (since it is to large for system memory anyway, it will be saved to the local disk). Both functions are not restricted to a certain file sizes, e.g. one can use *getFile* for large files and *getLargeFile* for small files as well.

The logical and technical description for *getFile* is very similar to the case of executing a synchronous system command, thus we will not describe it any further, the interested reader should refer to the source code in order to see all details. Requesting a large file on the other hand incorporates a slightly different form of communication since the client will deliver multiple messages. Fig. XIII.5 depicts the logical view onto the communication, after the client has received the request for a large file he will respond with a confirmation which has the first chunk and the total number of chunks attached to it. Afterwards the server will request iteratively request all remaining file chunks in the already described ping-pong manner. The technical realization is quite similar since it follows the previously illustrated paradigm, yet in this case the communication sequence consists of more than just two exchanged messages. Fig. XIII.6 shows this in more detail, as long as the sequence has not finished no other communication will be possible by using the frame object from node X.

XIII.3. Sending Files to a Node

The interface (i.e. *sendFile*) for sending files from the management server to a single cluster node expects only three parameters; the node ID, the local file path and the remote path (i.e. the local path on the target node). Thus there is only one function which inherently works with files stored on local disks. This function distinguishes internally between large and small files, it will switch to a chunked transmission should the local file size exceed the threshold value (see Fig. XIII.7), which is identical to the case of requesting large files.

XIII.4. Node (Un-)Registration

The process of registering a node is the very first action before any kind of communication can occur in a cluster environment. At this point one should not confuse the registration with the actual TCP connection establishing, a client might very well be connected to the cluster management server, yet without a successful registration procedure he will be unable to perform any sort of communication! Thus we will describe the registration under the assumption that a client has established a TCP connection to the server. Figures XIII.8 and XIII.9 depict the logical structure of the node registration and unregistration, respectively.

From Fig. XIII.8 it becomes obvious that a total of four messages will be exchanged, the corresponding technical realization is very similar to the situation of requesting large files, thus we will not explain it beyond the logical representation. The unregistration process is a simple ping-pong process.

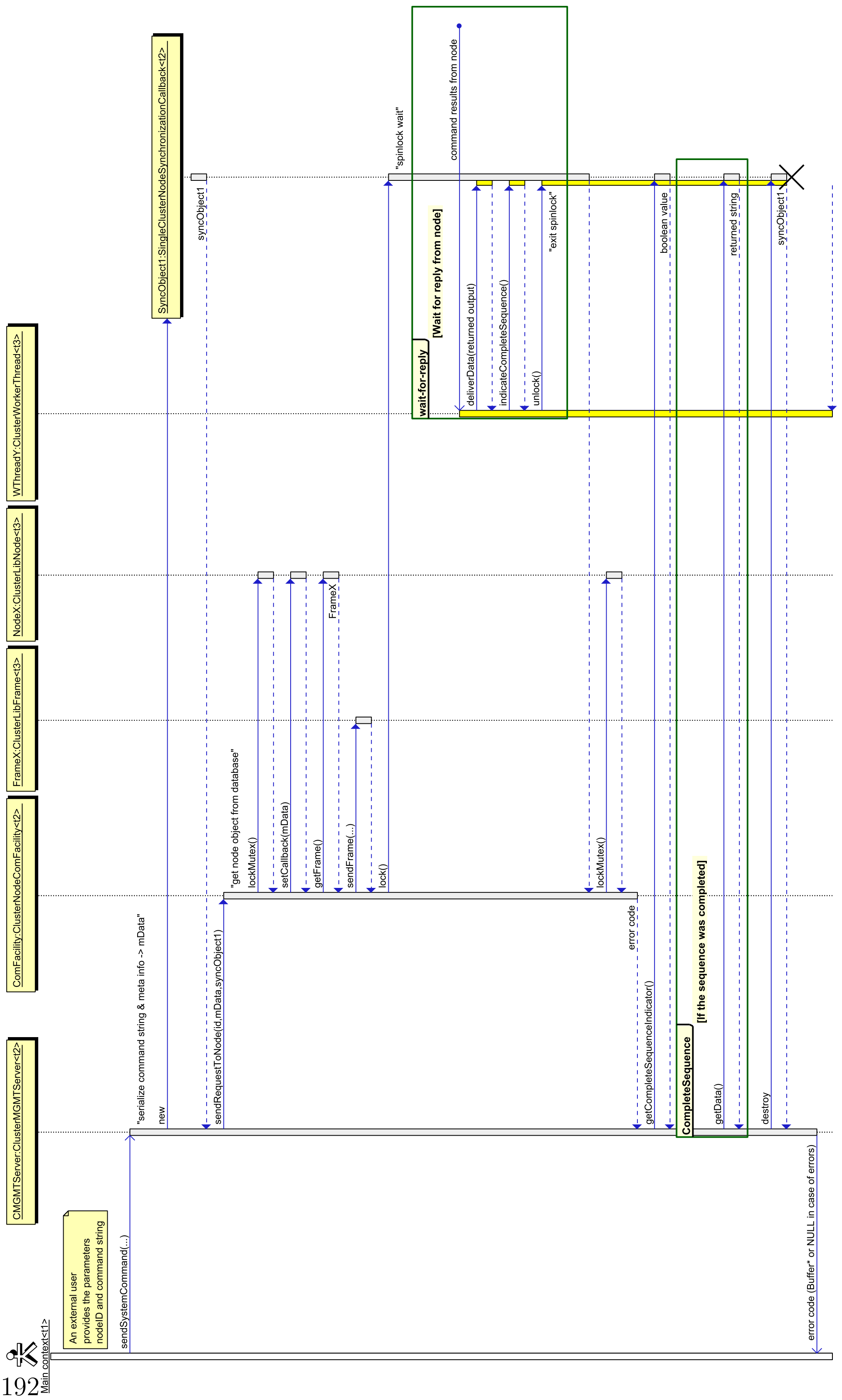
XIII.5. (Rapid) (Aynchronous) SHU Deployment

We omit the logical view on the process of (asynchronous) SHU deployment since it corresponds to the case of sending a (a)synchronous command to the client. Fig. XIII.10 shows the technical details (and similarities to sending synchronous commands), one should keep in mind that the process of SHU deployment can take a long time since the deployment confirmation will only arrive after the unit files have been compiled and the binary successfully registered itself on the node. For parallel deployment onto multiple nodes one should utilize asynchronous deployments and poll the clients for the currently active SHU PID via *getSHUPID*, if a value > 0 is returned one can be confident about a successful deployment.
The same statements hold for rapid SHU deployments, be it synchronous or asynchronous! Although a rapid deployment bypasses the compilation process it also may take quite some time to get the SHU running (e.g. because of long setup procedures due to custom CMake scripts). Thus SH provides asynchronous rapid deployments.

XIII.6. Requesting System & Short CPU Usage Reports

Requesting a complete system report via *requestSystemReport* or a short CPU report via *requestShortCPUReport* involves a simple ping-pong communication scheme as in the previously described cases. In this (short) section we merely want to point out two important aspects; firstly that both operations essentially induce the same computation overhead on the client, secondly that for highly frequent polling operations one should refer to *requestShortCPUReport* as it helps to reduce the memory and computation overhead.

The method *requestSystemReport* will return a complete system CPU report i.e. all existing CPU jiffie values for all CPUs on the client, in technical terms this amounts to a (client side) serialization and (server side) deserialization of *SystemStatus* objects. In situations of high frequent polls one is motivated to reduce every kind of overhead to reduce latencies induced by computation time. In exactly these situations one should use *requestShortCPUReport* as it merely commands the client to read all jiffie values, compute the corresponding load values for each CPU and transmit these floating point numbers to the client. Thus one reduces the amount of transmitted data and circumvents the computation overhead as no object marshalling occurs.



Main context<t1>
An external user provides the parameters nodeID and command string
sendSystemCommand(...)
CMGMTServer:ClusterMGMTServer<t2>
ComFacility:ClusterNodeComFacility<t2>
FrameX:ClusterLibFrame<t3>
NodeX:ClusterLibNode<t3>
WThreadY:ClusterWorkerThread<t3>
SyncObject1:SingleClusterNodeSynchronizationCallback<t2>
"serialize command string & meta info -> mData"
new
syncObject1
sendRequestToNode(id,mData,syncObject1)
"get node object from database"
lockMutex()
setCallback(mData)
getFrame()
FrameX
sendFrame(...)
lock()
"spinlock wait"
wait-for-reply
[Wait for reply from node]
command results from node
deliverData(returned output)
indicateCompleteSequence()
unlock()
"exit spinlock"
lockMutex()
error code
getCompleteSequenceIndicator()
boolean value
CompleteSequence
[If the sequence was completed]
getData()
returned string
destroy
syncObject1
error code (Buffer* or NULL in case of errors)

Figure XIII.2.: The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

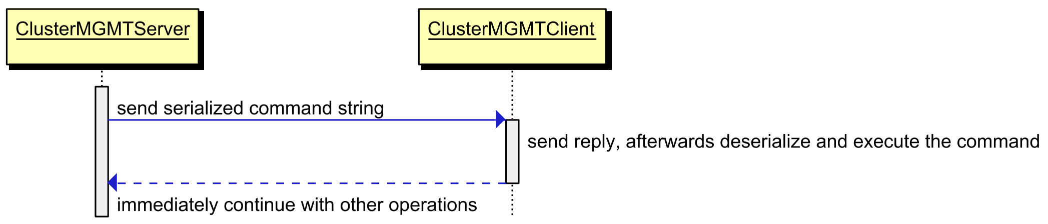

ClusterMGMTServer
ClusterMGMTClient
send serialized command string
send reply, afterwards deserialize and execute the command
immediately continue with other operations

Figure XIII.3.: The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

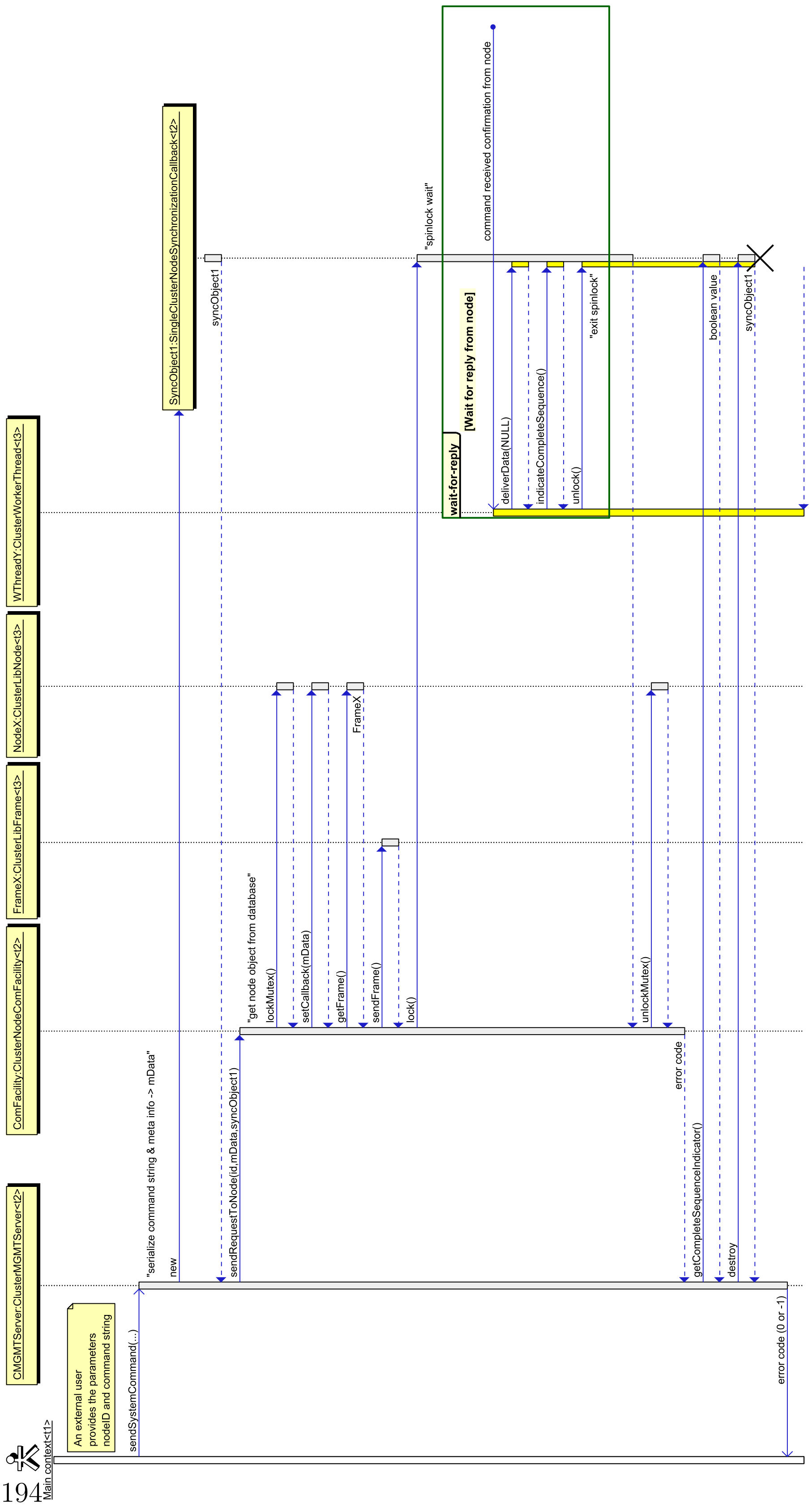

Main context<t1>
CMGMTServer:ClusterMGMTServer<t2>
ComFacility:ClusterNodeComFacility<t2>
FrameX:ClusterLibFrame<t3>
NodeX:ClusterLibNode<t3>
WThreadY:ClusterWorkerThread<t3>
SyncObject1:SingleClusterNodeSynchronizationCallback<t2>
An external user provides the parameters nodeID and command string
sendSystemCommand(...)
"serialize command string & meta info -> mData"
new
syncObject1
sendRequestToNode(id,mData,syncObject1)
"get node object from database"
lockMutex()
setCallback(mData)
getFrame()
FrameX
sendFrame()
lock()
"spinlock wait"
command received confirmation from node
wait-for-reply
[Wait for reply from node]
deliverData(NULL)
indicateCompleteSequence()
unlock()
"exit spinlock"
unlockMutex()
error code
getCompleteSequenceIndicator()
boolean value
destroy
syncObject1
error code (0 or -1)

Figure XIII.4.: The structure of SimpleHydra, each block represents a module, all modules beneath another module are required for its functionality

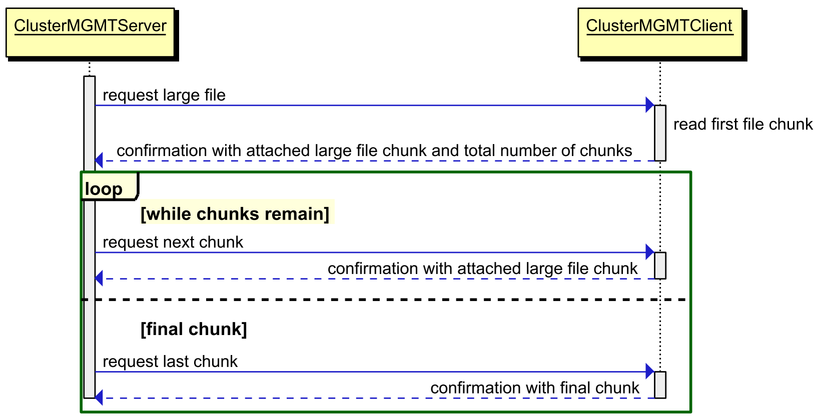
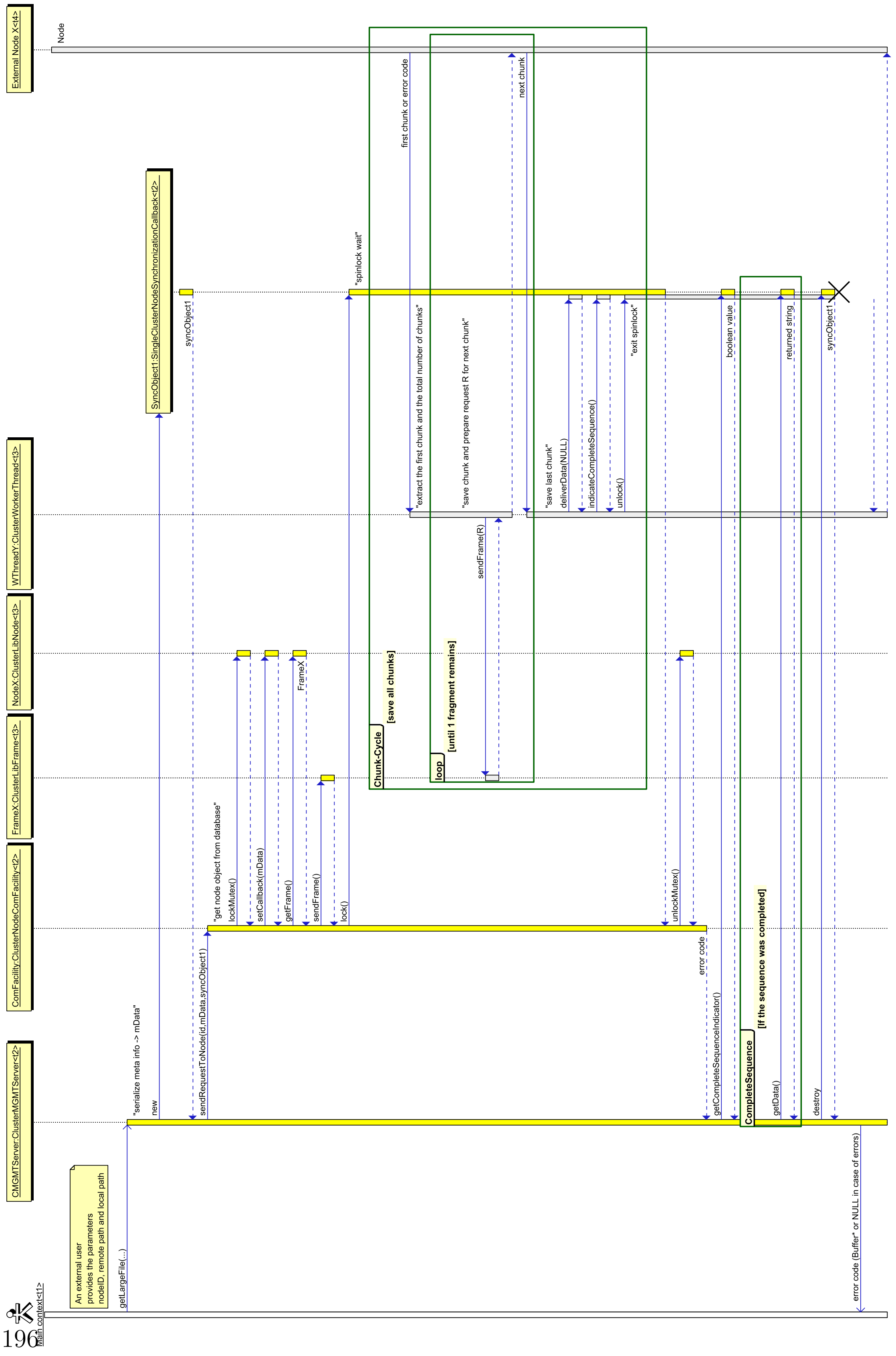


Figure XIII.5.: The request for a single large files from a given node, which will deliver the file in small chunks.



External Node X<t4>
Node
SyncObject1:SingleClusterNodeSynchronizationCallback<t2>
WThreadY:ClusterWorkerThread<t3>
NodeX:ClusterLibNode<t3>
FrameX:ClusterLibFrame<t3>
ComFacility:ClusterNodeComFacility<t2>
CMGMTServer:ClusterMGMTServer<t2>
Main context<t1>
An external user provides the parameters nodeID, remote path and local path
getLargeFile(...)
"serialize meta info -> mData"
new
syncObject1
sendRequestToNode(id,mData,syncObject1)
"get node object from database"
lockMutex()
setCallback(mData)
getFrame()
FrameX
sendFrame()
lock()
"spinlock wait"
Chunk-Cycle
[save all chunks]
first chunk or error code
"extract the first chunk and the total number of chunks"
loop
[until 1 fragment remains]
sendFrame(R)
"save chunk and prepare request R for next chunk"
next chunk
"save last chunk"
deliverData(NULL)
indicateCompleteSequence()
unlock()
"exit spinlock"
unlockMutex()
error code
getCompleteSequenceIndicator()
boolean value
CompleteSequence
[if the sequence was completed]
getData()
returned string
destroy
syncObject1
error code (Buffer* or NULL in case of errors)

Figure XIII.6.: The process of receiving a chunked file from node X in an existing cluster environment

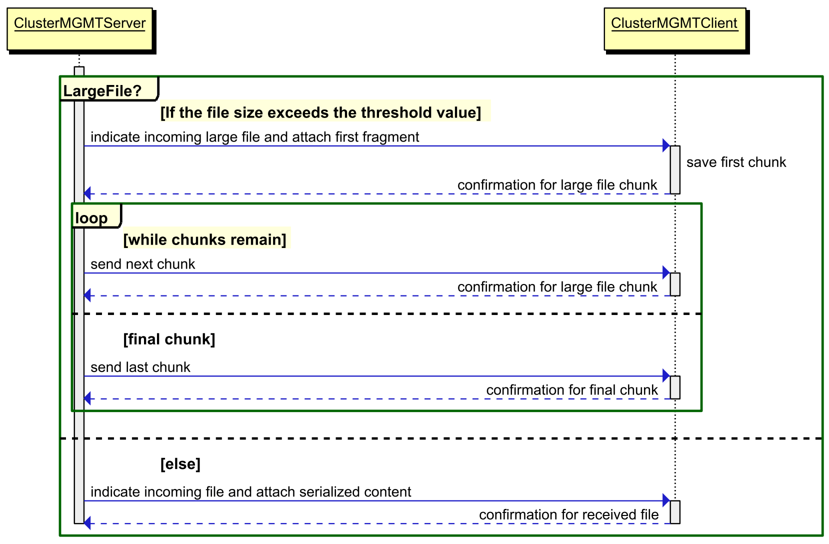

ClusterMGMTServer
ClusterMGMTClient
LargeFile?
[If the file size exceeds the threshold value]
indicate incoming large file and attach first fragment
save first chunk
confirmation for large file chunk
loop
[while chunks remain]
send next chunk
confirmation for large file chunk
[final chunk]
send last chunk
confirmation for final chunk
[else]
indicate incoming file and attach serialized content
confirmation for received file

Figure XIII.7.: Logical view onto the deployment process of a single file to a single node.

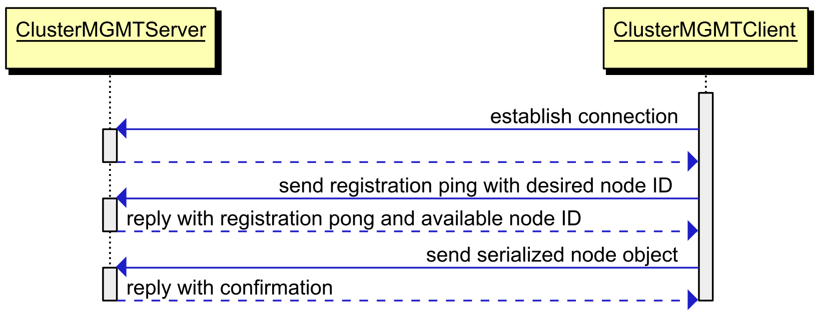

ClusterMGMTServer
ClusterMGMTClient
establish connection
send registration ping with desired node ID
reply with registration pong and available node ID
send serialized node object
reply with confirmation

Figure XIII.8.: the logical view on node registration

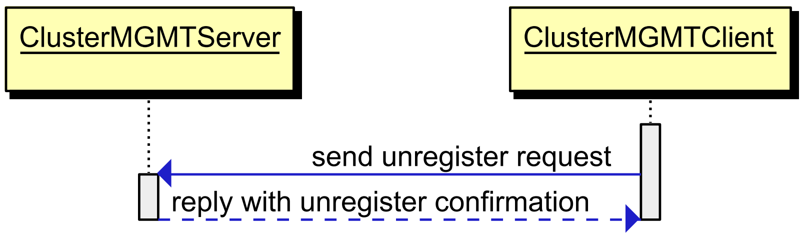


Figure XIII.9.: The logical view onto the process of node unregistration

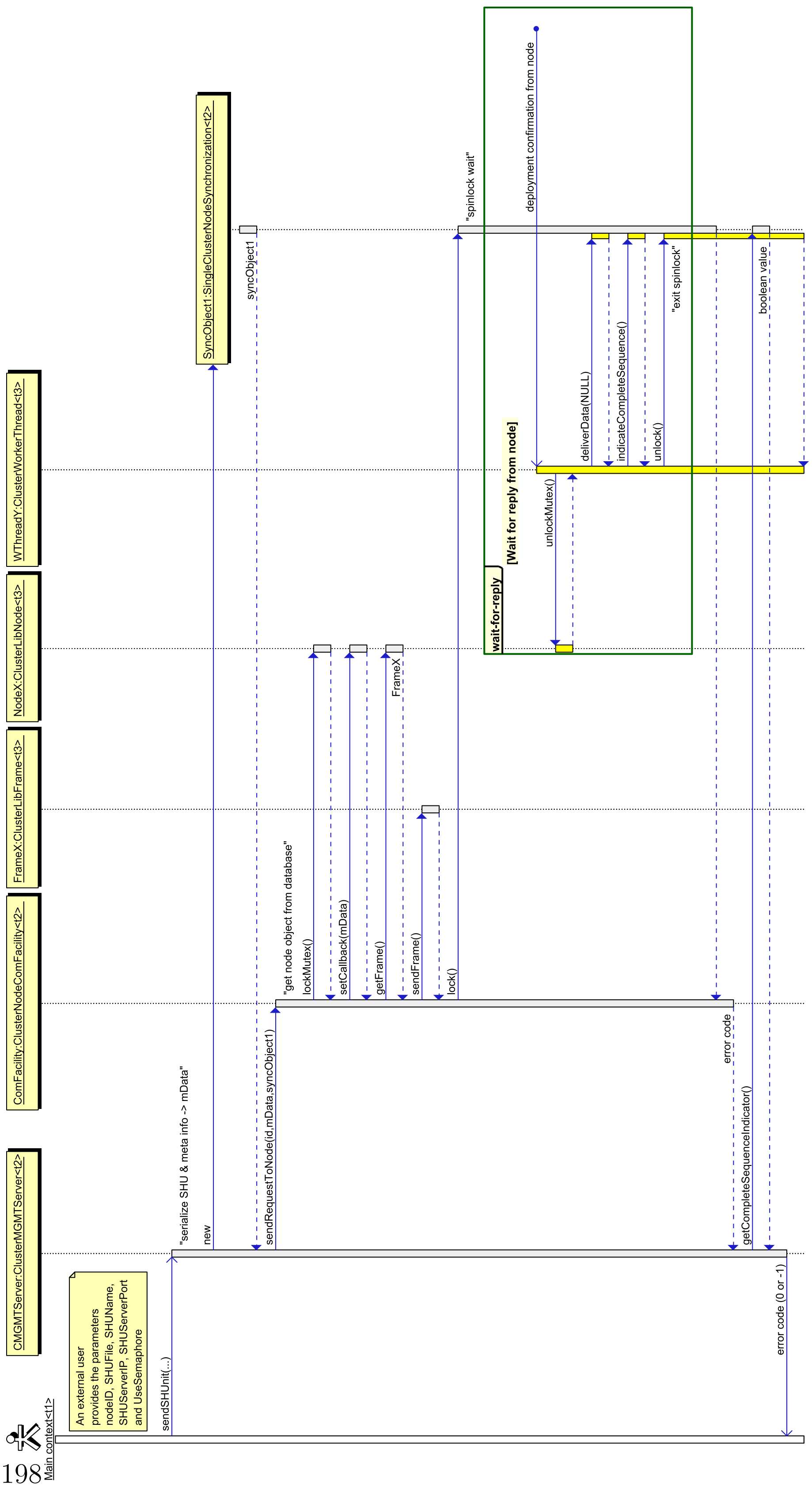

Main context<t1>
CMGMTServer:ClusterMGMTServer<t2>
ComFacility:ClusterNodeComFacility<t2>
FrameX:ClusterLibFrame<t3>
NodeX:ClusterLibNode<t3>
WThreadY:ClusterWorkerThread<t3>
SyncObject1:SingleClusterNodeSynchronization<t2>
An external user provides the parameters nodeID, SHUFile, SHUName, SHUServerIP, SHUServerPort and UseSemaphore
sendSHUnit(...)
"serialize SHU & meta info -> mData"
new
syncObject1
sendRequestToNode(id,mData,syncObject1)
"get node object from database"
lockMutex()
setCallback(mData)
getFrame()
FrameX
sendFrame()
lock()
"spinlock wait"
wait-for-reply
[Wait for reply from node]
deployment confirmation from node
unlockMutex()
deliverData(NULL)
indicateCompleteSequence()
unlock()
"exit spinlock"
error code
getCompleteSequenceIndicator()
boolean value
error code (0 or -1)

Figure XIII.10.: The sequence diagram for deploying a SHU onto node X in a set up cluster environment.

Bibliography

[12693] MPI: A message passing interface. In: *Supercomputing '93. Proceedings*, 1993. – ISSN 1063–9535, S. 878–883

[AV02] Adams, Joel ; Vos, David: Small-college Supercomputing: Building a Beowulf Cluster at a Comprehensive College. In: *SIGCSE Bull.* 34 (2002), Februar, Nr. 1, 411–415. `http://dx.doi.org/10.1145/563517.563498`. – DOI 10.1145/563517.563498. – ISSN 0097–8418

[CK01] Cherkasova, L. ; Karlsson, M.: 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, S. 212–221

[Cou09] Council, HPC A.: Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing / HPC Advisory Council. 2009. – Forschungsbericht

[DBPDP$^+$06] Di Biagio, C. ; Pennella, G. ; De Paoli, E. ; Grandi, R. ; Giammarino, F.: PVM advanced load balancing in industrial environment. In: *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, 2006. – ISSN 1066–6192, S. 5 pp.–

[DHL$^+$03] Dubinski, John ; Humble, RJ ; Loken, Chris ; Pen, Ue-Li ; Martin, PG: 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

[DNGFV00] Di Napoli, C. ; Giordano, M. ; Furnari, M.M. ; Vitobello, F.: PVM application-level tuning over ATM. In: *Parallel and Distributed Processing, 2000. Proceedings. 8th Euromicro Workshop on*, 2000, S. 391–397

[GBSP04] Gammo, Louay ; Brecht, Tim ; Shukla, Amol ; Pariag, David: Comparing and evaluating epoll, select, and poll event mechanisms. In: *In Proceedings of 6th Annual Linux Symposium*, 2004

[GDMO02] Grant, Jeffrey D. ; Dunbrack, RL ; Manion, Frank J. ; Ochs, Michael F.: BeoBLAST: distributed BLAST and PSI-BLAST on a Beowulf cluster. In: *Bioinformatics* 18 (2002), Nr. 5, S. 765–766

[HMH13] Hommel, S. ; Malysiak, D. ; Handmann, U.: Model of human clothes based on saliency maps. In: *Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on*, 2013, S. 551–556

[Jin01] Jin, Hai ; Buyya, Rajkumar (Hrsg.) ; Cortes, Toni (Hrsg.): *High Performance Mass Storage and Parallel I/O: Technologies and Applications.* 1st. New York, NY, USA : John Wiley & Sons, Inc., 2001. – ISBN 0471208094

[LLSW04] Liang, Tyng-Yeu ; Liu, Yen-Tso ; Shieh, Ce-Kuen ; Wu, Chun-Yi: A new approach to distribute program workload on software DSM clusters. In: *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, 2004. – ISSN 1087–4089, S. 201–206

[MG14] Matthias Grimm, Sebastian Hommel Uwe H. Darius Malysiak M. Darius Malysiak: Analyse von Personenbewegungen an Flughäfen mittels zeitlich rückwärts- und vorwärtsgerichteter Videodatenströme (APFel) - Teilvorhaben Videobasierte kameraübergreifende Bildsequenzanalyse. In: *BMBF Abschlussbericht* (2014)

[MH14] Malysiak, D. ; Handmann, U.: 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*, 2014

[SBS+95] Sterling, Thomas ; Becker, Donald J. ; Savarese, Daniel ; Dorband, John E. ; Ranawake, Udaya A. ; Packer, Charles V.: Beowulf: A Parallel Workstation For Scientific Computation. In: *In Proceedings of the 24th International Conference on Parallel Processing*, CRC Press, 1995, S. 11–14

[Ste97] Stevens, W. R.: *UNIX Network Programming: Networking APIs: Sockets and XTI.* 2nd. Upper Saddle River, NJ, USA : Prentice Hall PTR, 1997. – ISBN 013490012X

[Sun90] Sunderam, V. S.: PVM: A Framework for Parallel Distributed Computing. In: *Concurrency: Practice and Experience* 2 (1990), S. 315–339

[UPAM00] Uthayopas, P. ; Paisitbenchapol, S. ; Angskun, T. ; Maneesilp, J.: 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 Bd. 2, 2000, S. 935–940 vol.2

[UPAS01] Uthayopas, Putchong ; Phatanapherom, Sugree ; Angskun, Thara ; Sriprayoonsakul, Somsak: Sce: A fully integrated software tool for beowulf cluster system. In: *Proceedings of Linux Clusters: the HPC Revolution* Citeseer, 2001, S. 25–27

[WL10] Wang, Leping ; Lu, Y.: Power-efficient workload distribution for virtualized server clusters. In: *High Performance Computing (HiPC), 2010 International Conference on*, 2010, S. 1–10

List of Algorithms

List of Figures

Impressum

internal report 16-01

ISSN: 2197-6953

1. Auflage, 29.01.2016

© Institut Informatik, Hochschule Ruhr West

Anschrift

Institut Informatik

Hochschule Ruhr West

Lützowstraße 5

46236 Bottrop