Horizon 2020
FET HPC RIA Project #671553

ExaNeSt – European Exascale
System Interconnect and Storage
http://www.exanest.eu/

Deliverable **D4.3**

Implementation Notes for the Storage and Data Access Infrastructure

version 1.0 – 30 May 2017

EDITOR, CONTRIBUTORS

Partner	*Authors*
FHG	Bernd Lietzow
FORTH	Anastasios Papagiannis, Manolis Ploumidis, Manolis Marazakis
VOSYS	Angelos Mouzakitis, Alvise Rigo
INFN	Lucia Morganti
MDBS	Ying Zhang, Panagiotis Koutsourakis, Joeri van Ruth, Martin Kersten, Niels Nes, Sjoerd Mullender

REVISION HISTORY

Version	*Date*	*Description*
0.1	6 Mar 2017	First skeleton of the deliverable
0.2	15 May 2017	First draft of the deliverable
0.3	29 May 2017	Revised according to internal reviews
1.0	30 May 2017	Final version, to be delivered to the European Commission.

TABLE OF CONTENTS

LIST OF ABREVIATIONS

ACID	*Atomicity, Consistency, Isolation, Durability*
ACL	*Access Control List*
API	*Application Programming Interface*
DBMS	*Database Management System*
DMA	*Direct Memory Access*
GDK	*Goblin Database Kernel*
GUI	*Graphical User Interface*
JSON	*JavaScript Object Notation*
LRU	*Least recently used*
MAL	*MonetDB Assembly Language*
MCA	*Modular Component Architecture*
MPI	*Message Passing Interface*
NVM	*Non-Volatile Memory*
NUMA	*Non-Uniform Memory Access*
OSC	*One-sided communication (MPI)*
RAID	*Redundant Array of Independent Disks*
RDMA	*Remote direct memory access*
RMA	*Remote Memory Access*
SQL	*Structured Query Language*
SSD	*Solid-State Drive*
TUI	*Text-based User Interface*
VM	*Virtual Machine*
WLC	*WorkLoad Capture*

SUMMARY

In this deliverable of WP4, D4.3 “Implementation Notes for the Storage and Data Access Infrastructure”, we describe the implementation of each software component and tool that was specified in deliverable D4.2:

- Extensions and enhancements to the common I/O path in Linux with focus on two key areas: *(i)* supporting protected access to storage devices from user space, i.e. direct access to storage with minimal kernel related overheads; and *(ii)* enhancements to the access path for memory-mapped file access by, amongst others, making use of NVM devices. [FORTH]
- Extensions to the *BeeGFS* parallel file system: Metadata replication mechanisms to handle consistency management and resilience to failures, as well as incorporating those mechanisms in a caching extension. [FHG]
- Acceleration mechanisms for Host-to-VM and VM-to-VM interactions that take into account properties of the hardware platform and the unified interconnect. Two main technologies for hardware-assisted virtualisation are considered: *(i)* use of RDMA capabilities for accessing remote memory from within a virtual machine, and *(ii)* HPC API remoting to improve the performance of HPC APIs (specifically MPI) for applications that execute as an ensemble of virtual machines. [VOSYS]
- The design of a new replication mechanism for analytical databases, and its implementation in MonetDB. This replication scheme has been designed such a way that it will not only help improving the availability, reliability, performance and scalability of a database server in general, but also help exploiting the ExaNeSt storage infrastructure and the ExaNeSt platform in particular. [MDBS]
- Several monitoring and testing tools: the ExaNeSt storage administration and monitoring [INFN]; experiment automation, stress-load and fault injection tools [FORTH]; tests simulating HPC application I/O behaviours [INFN]; and a database profiler [MDBS].

In the storage infrastructure, resilience is mainly addressed by replication mechanisms in the file system. The system’s resilience will be stressed by the stress-load and fault injection tools, while its health is monitored by the monitoring tools. In addition, the system’s resilience level will be evaluated by the HPC and DBMS applications.

To achieve the objectives of WP4, the storage and data access infrastructure proposed by ExaNeSt is built upon the following main components: *(i)* the distributed file system BeeGFS, *(ii)* Linux data access technologies, *(iii)* KVM based virtualisation, and *(iv)* storage system monitoring and administration tools. Moreover, we will use the state-of-the-art HPC applications and analytical databases to guide the design of the envisioned infrastructure and to evaluate and showcase the result.

1. ExaNeSt Storage Infrastructure

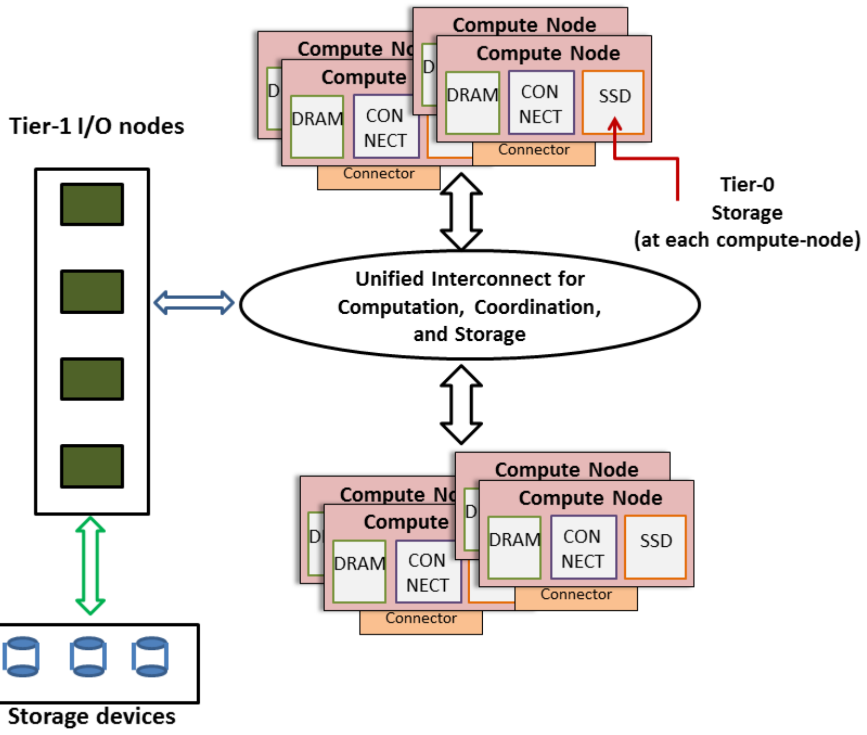

Tier-1 I/O nodes
Compute Node
Compute Node
DRAM
CON NECT
SSD
Connector
Tier-0 Storage (at each compute-node)
Unified Interconnect for Computation, Coordination, and Storage
Storage devices

***Figure 1.1.** The ExaNeSt system architecture with distributed in-node NVMs (e.g. SSDs), introduced as a new caching layer, and unified interconnect*

Figure 1.1 shows the target architecture of the ExaNeSt project. Concerning the storage subsystems, its distinguishing feature is the extra Tier-0 storage devices added to the compute nodes. The traditional Tier-1 storage devices are still used to persistently store large data sets. However, the file system will be extended to consider Tier-0 devices as large persistent caches. In this way, we can improve I/O performance by shortening the common I/O path, and reducing data transfers to and from Tier-1 servers. In addition, this architecture will also be able to reduce the energy consumption associated with frequent data movements.

In this deliverable, we describe the implementation of the storage components which together form a highly efficient and scalable storage infrastructure towards exascale. The components include:

- Section 2: extensions and enhancements to the common I/O path in Linux focusing on two key areas: i) supporting protected access to storage devices from the user space, i.e. direct access to the storage with minimal kernel-related overheads; and ii) enhancements to the access path for memory-mapped files by, amongst others, making use of NVM devices in Tier-0. [FORTH]
- Section 3: a caching extension to the BeeGFS parallel file system, together with replication mechanisms, to handle consistency management and resilience to failures, respectively. [FHG]
- Section 4: acceleration mechanisms for Host-to-VM and VM-to-VM interactions that

take into account properties of the hardware platform and the unified interconnect. Two main technologies for hardware-assisted virtualisation are considered: i) use of RDMA capabilities for accessing remote memory from within a virtual machine, and ii) HPC API remoting to improve the performance of HPC APIs (especially MPI) for applications that execute as an ensemble of virtual machines. [VOSYS]
- Section 5: A new replication mechanism, called *lazy logical replication*. It is an asynchronous logical replication management scheme using change set forwarding. [MDBS]
- Section 6: the ExaNeSt monitoring and testing tools, including the ExaNeSt storage administration and monitoring tools [INFN]; experiment automation, stress-load and fault injection tools [FORTH]; tests simulating HPC application I/O behaviours [INFN]; and a database profiler [MDBS].
- Section 7: A HPC application simulator, which launches a number of MPI processes simulating the activity of checkpoint and restart, which puts a ceratin type of stress on the file system. [INFN-CNAF]

Figure 1.2 shows how the storage components will be integrated into one system:
- Everything is run on enhanced virtual machines [VOSYS] (the inner orange coloured part):
  - The extended BeeGFS is the file system on the virtual machine.
  - Together with the enhanced I/O path it provides fast storage access.
  - The file system monitoring tools and experiment automation tools run inside the VM to monitor the file system and I/O access.
  - The DBMS and HPC applications run inside the VM to evaluate the overall performance.
- On an ExaNeSt host machine (the black coloured outer part):
  - Again, BeeGFS is the file system, as well as the improved I/O access path provide fast data access.
  - The file system monitoring tools and experiment automation tools can be used to monitor the file system and I/O access.
  - A virtual machine is just a single file managed and served by BeeGFS.

In the design of the storage infrastructure, resilience is addressed by the whole software stack, from the storage system to the applications:
- Replication mechanisms are added to the file system to ensure data availability under system failures.
- Applications, such as MonetDB, have been extended with features to improve the application's level of resilience, while also make use of resilience mechanisms provided by the underlying layers, such as the file system.
- The system's health is continuously monitored by the monitoring tools. Any significant change will immediately be captured and reported.
- With the stress-load and fault injection tools, one can extensively put the storage system under pressure to study the systems resilience, and detect possible points for improvements.
- Finally, the HPC and DBMS applications run extensive tests to evaluate the system's resilience.

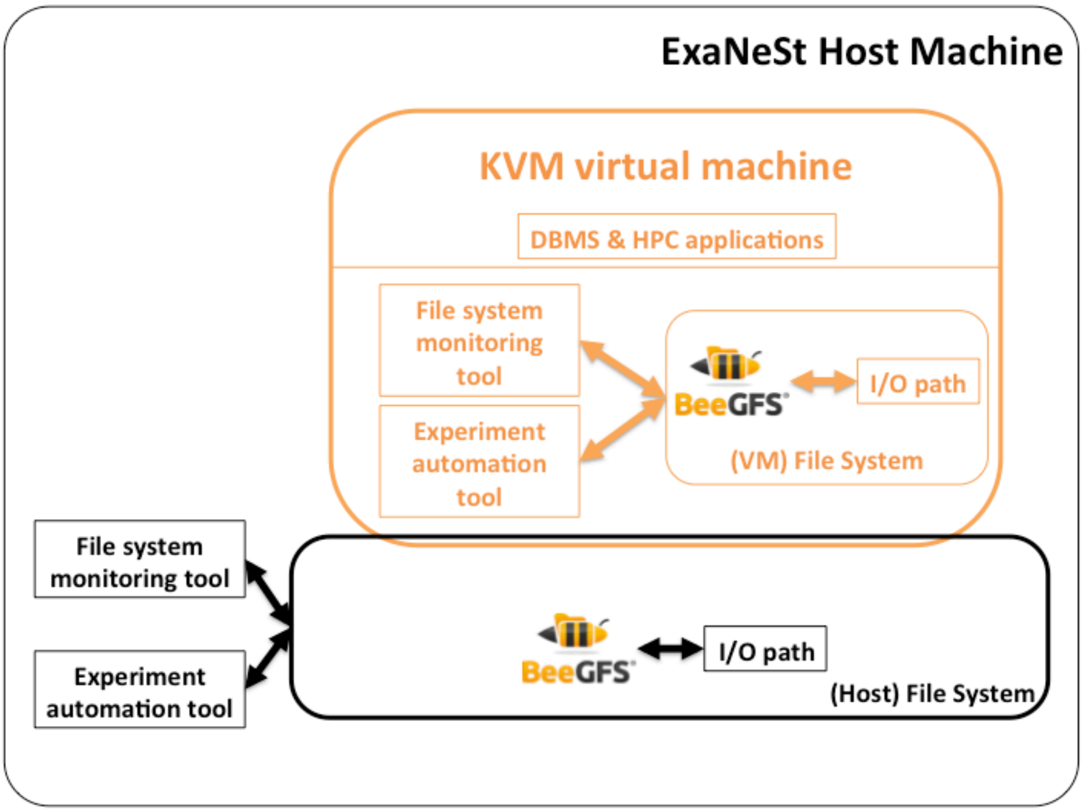


***Figure 1.2.** The ExaNeSt storage components integration*

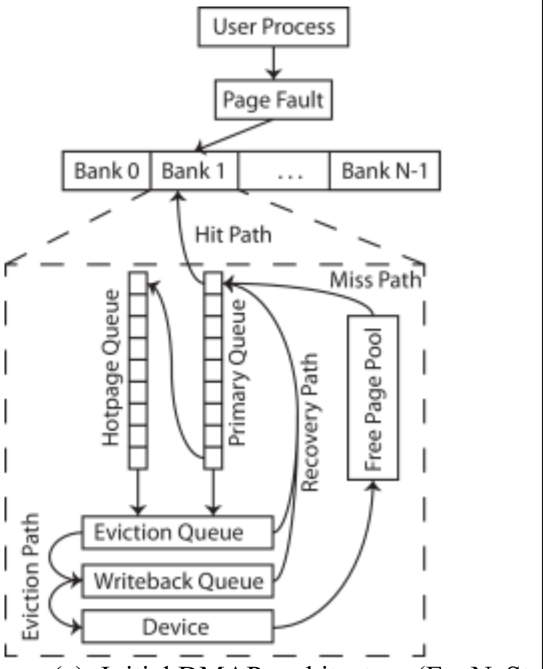
2. Extended Linux I/O Path

[Contributed by FORTH]

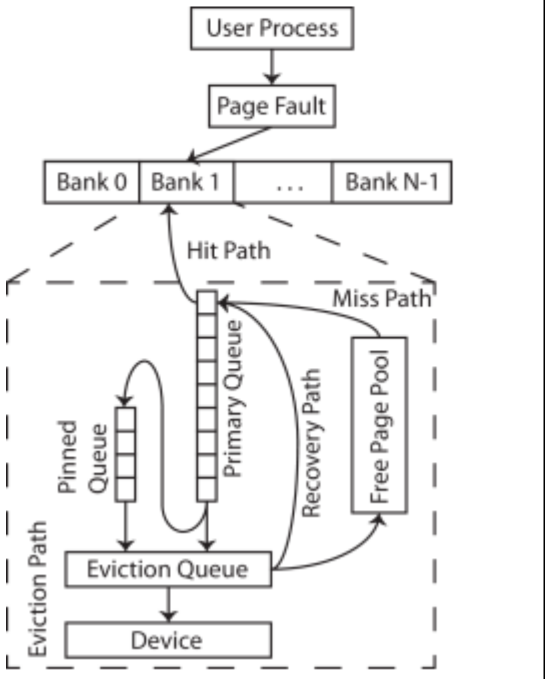
Applications that interact with storage, have two common ways to do that, one is explicit I/O using read()/write() system calls. Another way is to use memory-mapped I/O. This can be achieved using the mmap() API that Linux kernel provides. A specific example of this scenario in the context of the ExaNeSt project is the MonetDB in-memory database.

With memory-mapped I/O, common path lookups are not required, since data that resides in memory have mapped virtual addresses, whereas misses cause page faults. Additionally, there is no copy when moving data between memory and storage.

However, using memory-mapped I/O has a number of important disadvantages. First, there is no explicit control over data eviction, as with a user-space cache, mmap uses an LRU-based policy, which is not appropriate for all use cases. Applications cannot specify themselves which pages need to remain in memory and which ones can be evicted. One possible path to achieve this with vanilla mmap is to lock important pages through the use of mlock. However, the Linux kernel does not allow a large number of pages to be locked by a single process. Another alternative is to use madvise system call which, however, provides only hints. Second, each write operation in an empty page is effectively translated to a read-modify-write, since mmap does not have any information about the status (allocated or free) of the underlying disk page and the intended use. This can result in excessive device I/O. The user applications should have the ability to inform mmap if a page contains garbage in order to map it without reading it from the device. Third, mmap employs relatively eager evictions in order to free memory to avoid starving other system operations. When large portions of the application virtual address space are mapped, these aspects of mmap result in unpredictable use of memory and bursty I/O. Empirically, we have often observed large intervals (of several 10s of seconds) where the system freezes while it performs I/O and applications do not make progress. This unpredictability and more importantly, large periods of inactivity are an important problem for applications that serve data to online and user-facing applications.



(a) Initial DMAP architecture (ExaNeSt deliverable D4.2.)



(b) Current DMAP architecture (ExaNeSt deliverable D4.3)

***Figure 2.1.** Evolution in the design of DMAP.*

To address these limitations, while realizing the performance potential of memory-mapped I/O, we have proposed DMAP, A custom Linux kernel module that bypasses the tight interaction of mmap() with the Linux kernel buffer cache. In ExaNeSt deliverable D4.2, we described the design of DMAP. Through a detailed performance evaluation and profiling we opted to change some details of its architecture. Figure 2.1(a) shows the design of the previously described architecture. In Figure 2.1(b) we show the new design. The main differences are that we have removed two queues, the hotpage and writeback queues. As they do not affect the eviction policy and they only incur additional CPU overheads we removed them from the design. We also added the pinned queue in order to keep some pages with high priorities that cannot be evicted. Next, we describe the new architecture of DMAP in more detail.

2.1 *Overview of DMAP Implementation*

We have implemented a custom mmap() runtime system, as a Linux kernel module, named DMAP. We provide an API that allows applications to explicitly specify (via ioctl's) priorities between 0 (highest) and 255 (lowest) at page granularity. Then, user applications memory-mapped I/O implements its own caching mechanism that bypasses the kernel page-cache using the following structures:

- One primary priority queue that keeps all pages mapped to the application. Pages are kept in page priority order.
- One eviction queue for pages that have been unmapped and can be evicted, if required. Pages are sorted based on their device offset to assist with I/O merges.
- One pinned queue for maintaining pages that may be flashed to the device but should not be unmapped from the key-value store address space. This queue is for short-lived pages with high priority when the buffer is full of low priority pages.
- One in-memory red-black tree in order to keep dirty pages. This enables more effective merges as the pages are kept sorted and separated from clean pages.
- One free page pool that is implemented as a simple in-memory list. For each page-fault we use this pool to get an empty page.

DMAP uses replacement policy based on two levels of priority queues. These are the primary and eviction queues. When a new page-fault occurs we get a new page from the free page pool, we read the data from the device (if required) and then we enqueue this in the primary queue. If primary queue is full we have to dequeue an item. Dequeue operation removes a page that has the lower priority compared to the others. After a dequeue operation from the primary queue we unmap this page from application address space and we enqueue it to the eviction queue. In order to enable larger I/Os we dequeue pages from primary queue in batches of 256 pages. In the case where we cannot find a free page upon a page-fault we dequeue a batch of pages from eviction queue. This means that we have to write their data into the device and then add them to the free page pool. Priorities are kept only in-memory and the user application should set them.

DMAP significantly reduces unpredictability with respect to memory management during runtime. It uses as configuration input the maximum amount of memory it should use during operation. Using this information, it calculates the size of L0 index that fits in memory and based on this it calculates the sizes of its memory-mapped I/O structures and preallocates all structures. Typically, the size of the primary queue is 10x larger than the eviction queue and the pinned queue is small.

DMAP allows higher parallelism than mmap. This is necessary both in the page mapping path (at page faults) and in the eviction path, to allow SSDs to operate under high concurrency. We achieve this by using an organization of memory in banks, similar to DI-MMAP [EHA12, EHA15]. This allows us to serve page-faults that refer to different banks in parallel. Unlike DI-MMAP, we use fine grain locks within banks, which results in even more parallelism during read and write I/Os.

The replacement policy of DMAP is less "eager" as compared to mmap. This means that we try to keep as more pages as we can in memory, as we do not evict any pages if there is no pressing need to do so. This allows DMAP to generate larger I/Os during spill operations, by merging more I/O requests.

Finally, DMAP provides a non-blocking msync call that allows the system to continue operation after invoking a commit, while pages are written asynchronously to the devices. In order to accelerate msync we keep dirty pages sorted based on device address in an in-memory red-black tree. To ensure that it operates correctly we have to lock this tree in order not to add new dirty pages in it. To overcome this limitation we also keep a timestamp for each page that shows when a page becomes dirty. This allows us to do a fine grain locking as we can add new dirty pages in the red-black tree during msync. To write dirty pages, we iterate the red-black tree and we write only pages with timestamp older than the timestamp of msync call. This approach ensures that pages that become dirty after msync are not written in the device. However there will be pages that are already dirty and changed after msync which can be written which does not break msync semantics.

2.2 *DMAP Test Suite*

For correctness and evaluation purposes we develop a test suite for DMAP. We use both our custom tests and several tests based on FIO benchmark [FIO]. More specifically these include:

1. For performance evaluation we built
   (a) several scripts for the FIO I/O tester application, including both random and sequential I/O patterns, for read, write and mixed read-write I/O workloads. For all cases we use the mmap() ioengine. This is used for files.
   (b) custom multithreaded microbenchmarks for all combinations of rand and sequential patterns and read, write and mixed workloads. It reports IOPS for each thread. This is used for block devices.

3. For correctness validation we implemented:

(a) a test (not-automated yet) that checks if msync call writes only dirty pages.

(b) some FIO scripts that write a random pattern to the device and then during a second stage verify the content of files. We use it, and we have verified that it produces a correct output for both in-memory and out-of-memory datasets.

(c) also ia set of custom programs, operating as follows:
- Write sequential pages with random data and computes a crc checksum for each page. These checksums are kept outside of mmap (in memory).
- Execute msync and munmap.
- Proceed to mmap() again the device, read the pages and compute the checksums.
- Compare the checksums with the original checksums.

This test sequence covers both files and block devices, and DMAP passes all the tests.

2.3 *Preliminary Performance Evaluation of DMAP*

In this section we provide a preliminary evaluation using the FIO benchmark. Fio supports already a mmap() test backend ("ioengine") and thus no changes required. We evaluate random reads and writes with 4KB block size. In all cases the fio runs are time based and the duration of them are 10 minutes. Furthermore, we use two test datasets, one that fits in main memory and another that is out-of-core. In all experiments, we use files (not block devices). In order to limit the amount of memory in DMAP case we specify in our runtime this amount of memory. In the case of vanilla mmap we boot the machine with the desired amount of memory (using mem=XXG Linux kernel boot option). For the small dataset, we use 16GB of main memory and 8 files of 1GB each. For the large dataset, we use 4GB of main memory and 32 files of 1GB each

Our experimental platform is a server system with two quad-core Intel(R) Xeon(R) E5520 CPUs running at 2.27 GHz. The system is equipped with 24 GB DDR-III DRAM. As a storage device, we use a hardware RAID-0 with eight Intel X25-E SSDs (32 GB) – 256GB in total. The storage controller we use is LSI MegaRAID SAS 9260-8i.

In order to find out the maximum I/O performance of our testbed we use FIO benchmark [FIO], with libaio engine and direct I/O. For random experiments we use 128 outstanding I/Os and 4KB block size. For reads it achieves 45.6K IOPS (182.52 MB/s) and for writes it achieves 19K IOPS (76.2 MB/s). For sequential performance we use 1 outstanding I/O and 16MB block size. For reads it achieves 1.28 GB/s and for writes 624 MB/s.

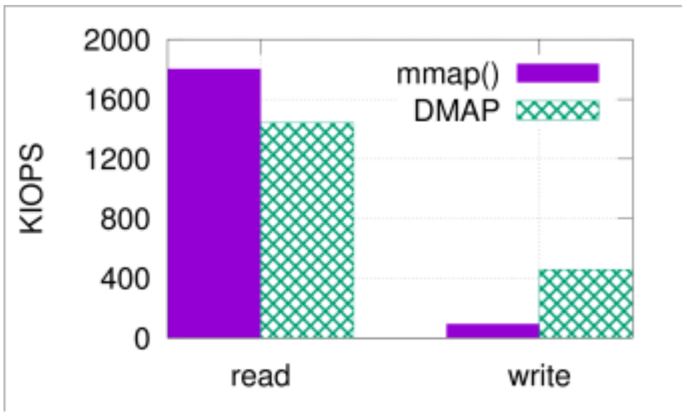


***Figure 2.2.** IOPS for DMAP and mmap() using the in-memory dataset (higher is better).*

The results of experiments with the in-memory dataset are summarized in Figure 2.2. For random reads, vanilla mmap() provides 25% higher throughput. In this case the dataset fits in memory and no I/O is required. To further investigate the reason that DMAP results in lower performance compared to vanilla mmap() we measure the number of page-faults from our internal counters. It seems that in the case when the dataset fits in memory and there should not be any page faults, in DMAP exists. On the other hand no I/O is done as the data already exists in memory but only the page faults. This is the reason that the performance drop is small. From our profiling it seems that there are some invalidations for some pages that originate from Linux kernel, and we should further investigate this issue. In the case of random writes, DMAP provides 4.9x higher throughput. This is because DMAP takes a less "eager" approach for data evictions. This number shows that for an application where the write dataset fits in memory, the performance improvement can be significant.

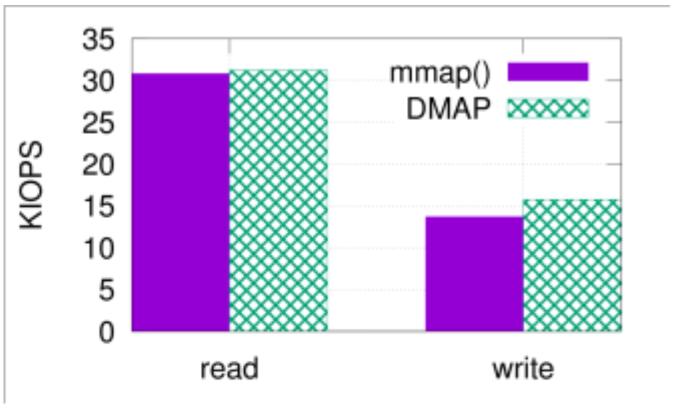


***Figure 2.3.** IOPS for DMAP and mmap() using the out-of-memory dataset (higher is better).*

Figure 2.3 summarizes the results of experiments with the dataset that does not fit in memory. In this case the bottleneck is the device throughput and the differences are not so large. More specifically in the case of random reads DMAP provides about 2% higher throughput and in the case of random writes DMAP provides 15% higher throughput.

In all cases DMAP provides better performance in the case of writes because of the more lazy eviction policy that enables larger I/Os due to merges – because there is a larger time window to do it. In the case of reads when the dataset does not fit in main memory – which is the common case – we provide similar performance. For the dataset that fits in main memory there is a performance degradation as described earlier and we have to further investigate it.

In the experiments reported so far, there is no differentiation among the data blocks being accessed. We expect performance gains with DMAP, as our design explicitly handles priorities, assuming of course that applications can distinguish important and long-lived pages compared to short-lived pages that should be evicted shortly. To evaluate this scenario we run FIO using the large dataset for both reads and writes. As the dataset consists of 32 files, 1GB each of them, in DMAP we set the priority of a single file to 0 and for the other 31 files we set the priority to 1. Figures 2.4 summarizes the results of this experiment, for reads. Using priorities in DMAP, we can keep a single "preferred" file in memory and this enables it to achieve much higher performance compared to the others. Figure 2.5 shows the same experiment for random writes where we observe the same behaviour. We plan to provide a more detailed analysis using different priorities for different files but the idea is that the subset of files that fits in memory will achieve memory access times while the others will achieve storage device access times. We do not provide a mechanism that allows the user to throttle the access time of files that have low priority and fits in main memory.

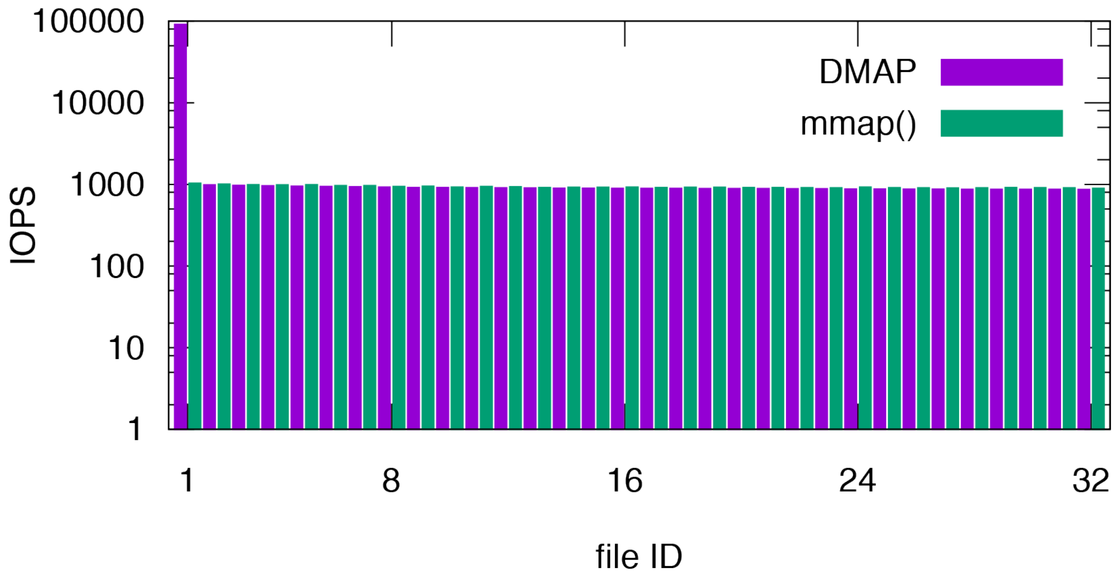


***Figure 2.4.** FIO random read experiment with priorities (y-axis is log-scale), using the out-of-memory dataset (higher is better). File 0 has been assigned higher priority.*

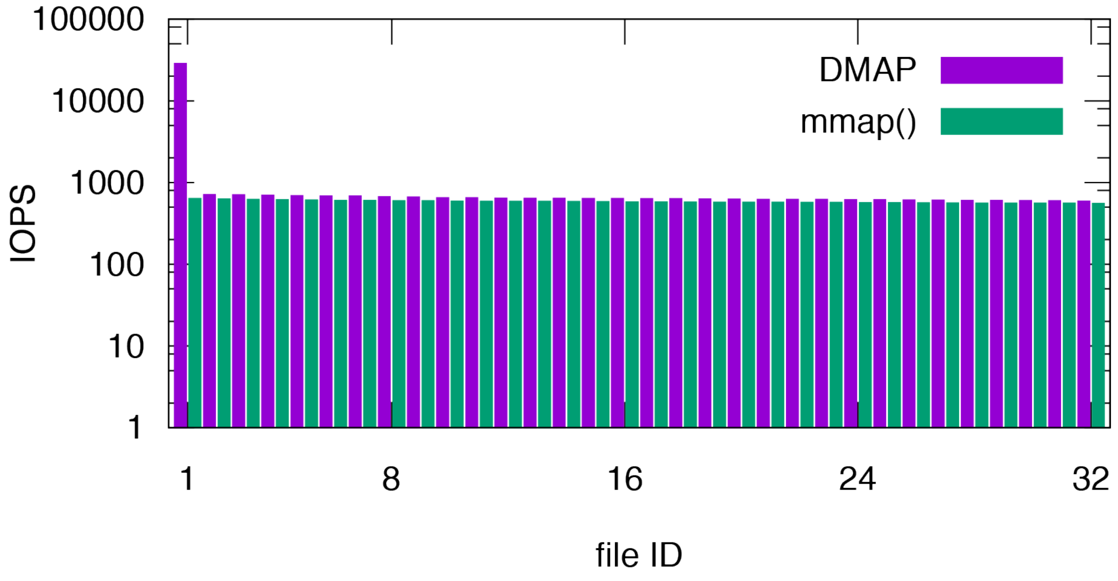


Figure 2.5. *FIO random write experiment with priorities (y-axis is log-scale), using the out-of-memory dataset (higher is better). File 0 has been assigned higher priority.*

Currently, FORTH is working with MonetDB for an evaluation using the TPC-H database analytics benchmark [TPCH]. Furthermore we have ported DMAP for ARM architecture and we plan to provide our preliminary evaluation on ExaNeSt platform.

3. Extended Distributed File System

[Contributed by FHG]

In this section, we describe the development of the extension to the BeeGFS parallel distributed file system. It consists of an intransparent caching layer based on the BeeOND technology, as well as replication mechanisms providing resilience to failures.

3.1 *BeeGFS as ExaNeSt Cache Layer*

Using the BeeOND (BeeGFS-on-Demand), it is possible to set up a parallel file system on the fly. This file system acts as an independent file system. Unlike the typical use case of parallel file systems, it does not use dedicated server hardware for metadata and bulk storage. Instead, the local storage devices, e.g. flash drives, in the compute nodes are used.

As described in previous documents D4.1 and D4.2, this system can be used to set up a separate file system for each compute job. When the scheduling system starts up a compute job on a number of compute nodes, an instance of BeeOND will also be also started, using the flash storage devices in or close to the compute nodes running the computation themselves. Because BeeOND uses the BeeGFS parallel file system technology, it can aggregate the space and bandwidth of the flash storage devices in all nodes, providing a dedicated storage for this job to be used as a data cache or an application scratch area. Data which are local to the job, for instance temporary data or data which have to be accessed numerous times during the computation, can now be stored inside this cache, keeping the load off the global cluster file system and network, as well as improving performance by using storage devices much closer to the actual computation.

In an exascale context, single node failures during job execution are expected, and we aspire to minimize their impact on performance and data consistency. For a parallel file system this means that no data may be lost or in an inconsistent state even after a node has failed.

The BeeOND package has been extended to support the storage and metadata mirroring and failover features present in BeeGFS. This way it is now possible to activate metadata and storage mirroring automatically when starting up a new BeeOND instance, as well as replacing failed nodes. On startup of a new job, storage and metadata nodes are grouped in pairs (mirror buddy groups) which mirror each other's data, using the mirroring mechanism described in the following section. If any single node fails during a job, there will be no data loss, and access to the data continues after a short interruption. Even though the application on the failed node itself might have to be restarted from a previous checkpoint, the data in the BeeOND based cache will still be available (e.g. to all the other nodes which are part of that job), eliminating the need to re-calculate or re-copy from a global storage system.

3.2 *Replication and Resilience*

The server side of BeeGFS is divided into three parts: the storage servers, the metadata servers, and the management service.

The storage servers manage the actual file contents, which are distributed across multiple storage targets. Each storage server hosts one or more storage targets. Files are split up into chunks and distributed across a number of storage targets. The metadata servers are responsible for managing the directory tree, as well as information about which storage targets hold the chunks of a file. The management service keeps track of the other nodes in the system: clients, storage servers, and metadata servers. It has information about how to contact the servers, and in a failover-enabled setup it is also responsible for deciding when a node is assumed unreachable and a failover should happen.

Previous versions of BeeGFS already supported some level of replication and failover. However, this was only available on the storage servers, so while the file contents were replicated, the metadata (directory structure, file names etc.) was not. In case of a metadata server failure, parts of the file system would be permanently lost.

The available mirroring of data on the storage servers done using two storage targets located on two different storage servers. One target plays the role of the "primary" while the other is the "secondary". While both targets are online and available, a client will always send its write requests to the primary storage target, from where they will be forwarded to the second storage target. Once data has been written to both targets, the write request is acknowledged to the client. Read requests are served from both targets in order to increase read performance.

Each target of this pair can fail at any time. Should the secondary target fail, the forwarding of requests will be impossible. However, all read and write requests are still being executed by the primary target, so from the clients' point of view, there is no interruption in service. As soon as the primary target detects the forwarding of a write request has failed, it will mark the secondary as "needs-resync", because from this point on, the data on the primary and secondary target is no longer identical. Should the secondary target come back online, its contents will need to be resynchronized with the contents of a primary target before it can start serving read or write requests again. The resync will copy over all chunks that have been modified since the target went offline.

In case the primary target fails, each request made by the client side will fail initially. But this failure is not passed on to the user application, instead the client will retry contacting the primary target. After a short delay the management service will detect that the primary target has gone offline and will initiate a failover. Once the client is aware of this failover, it will redirect its requests to the former secondary target, which has now become the primary target and will be able to complete the requests normally. From the perspective of the user application there will not be any difference, except for the slight delay before the failover is executed.

In the frame of the ExaNeSt project, a similar mirroring and failover mechanism has been implemented for the metadata servers, as well. This proved to be more complex, because on a storage target, each request will only modify a single chunk of data, while on a metadata server a single request can touch multiple objects at once – for instance, moving a file from one directory to another. Therefore, the forwarding mechanism based on requests and retries was extended using a sequence number mechanism which ensures every request is executed exactly once even if it had to be sent multiple times in case a server was unavailable because of a transient (e.g. short network outage) or a permanent (e.g. hardware damage) failure.

The resync mechanism also had to be extended. For a storage target, it is valid to copy over all chunks that have been modified, while at the same time accepting new requests and forwarding them. This will end in a complete copy, because even if a chunk is updated while the resync is running, either the update will happen before it is copied over, and the copy will already include the new version, or the update will happen after the copy, in which case forwarding the update will also result in two identical copies. In the case of a metadata server, this resync mechanism becomes more complex because modifications are not idempotent, and therefore have to be executed exactly once and in the correct order. Therefore the resync process had to be split into two phases. The first phase makes an exact copy of the contents of the metadata server. During that time, all incoming requests executing updates on already copied objects will be stored in a *modification resync queue*. The requests in this queue will be sent to the secondary server by the second resync phase. Only after the contents of the queue have been completely processed on the secondary server, the normal forwarding of requests is resumed.

The management service is a central instance of the system. It plays a very important role because it is constantly monitoring the state of all the other services (storage targets and metadata servers), as well as deciding when a server is considered offline and if a failover has to take place when a server becomes unavailable. Despite its importance, it handles a comparatively small set of data, mainly consisting of the state of each server or target (reachability and data consistency), and a set of information about it, e.g. network addresses and free capacity. Replicating the management service simply using a mirror pair, as was done in the storage and metadata services, would not be sufficient, because the mirror pairs mechanism BeeGFS uses relies on a single central instance deciding when a failover occurs. It was therefore decided to implement a distributed management service, with a leader election to determine where the central point of contact of the management service is, and which part of the distributed system is in charge of making decisions about the state of the other system components. Several instances of the management service run on separate machines, and use a distributed key-value store to ensure the data set is consistent across all management service instances. Since the management logic (i.e. the component that monitors the state of the other nodes and decides when a resync is needed, etc.) must be running exactly once, a leader election scheme has been implemented. One management service instance called the leader provides contact with the rest of the system and executes the management logic. The other management service instances are called the followers. They are only monitoring the state of the leader, but not executing any logic themselves. Should the leader become unavailable, this is detected by one or more of the followers, which will initiate a leader election. The election ensures that exactly one of the followers becomes the new leader, and executes the management logic from then on, as well as providing the new point of contact for all other components of the file system.

In order to connect to the management service, a node only needs to know the address of one of the management instances. If this instance is the leader, then the connection is initiated. If the instance is only a follower, it will reject the connection and inform the connecting node about the network address of the current leader. However, for redundancy it is necessary that each node knows more than one management service address, in case one of the instances has already failed. Therefore in the default node configuration, a list of management locations can be entered.

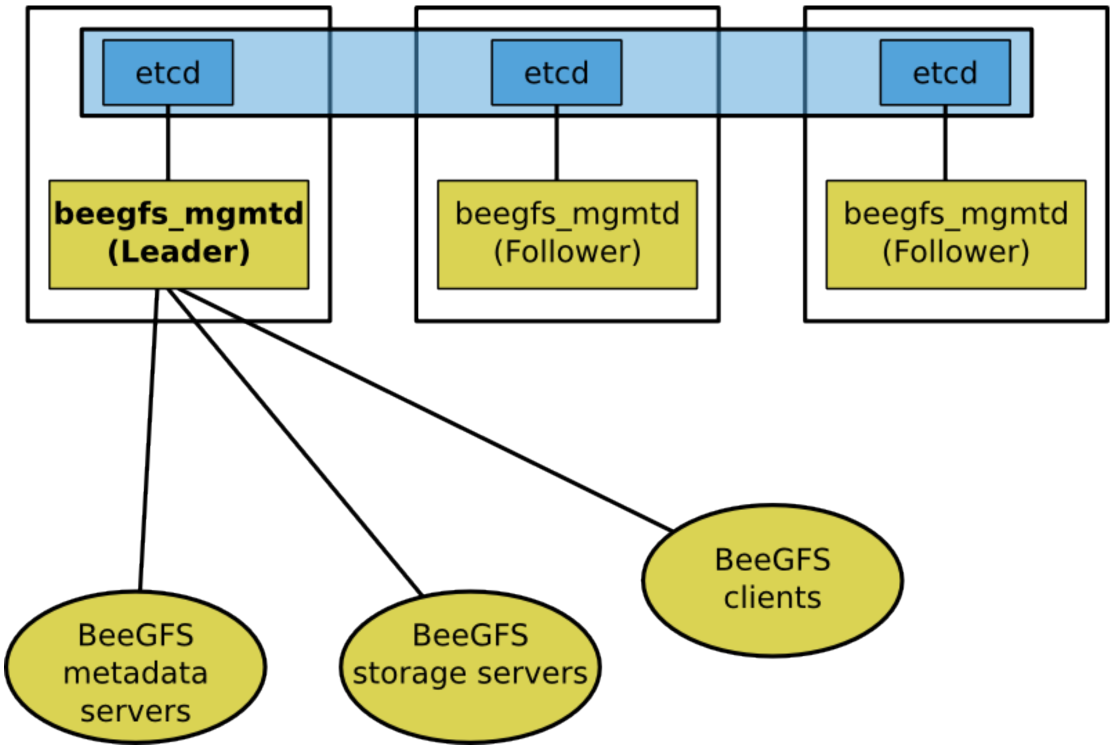


Figure 3.2. *The BeeGFS management service replication scheme showing multiple management service instances communicating over etcd*

Etcd [ETCD] is used as a distributed storage. It provides a mechanism that keeps data consistent across a number of nodes in the form of a key-value store, based on the Raft consensus algorithm [OO14]. Raft itself is designed based on leader election and a distributed state machine. It ensures that an update is persistently committed to the replicated storage, before acknowledging that update to the requester. Etcd provides a mechanism for leader election itself, which is be used to determine the leader instance of the distributed management service.

The internal representation of the state of each storage target and metadata node had to be changed to accommodate for this new model. In previous BeeGFS versions, each node or target managed its own state, and the management service was merely responsible for distributing the information, and deciding when nodes are offline and a failover has to be initiated. For the system working reliably on a distributed state machine, the flow of information has to be restricted to one direction. The management service is now solely responsible for managing the state of a node, i.e. whether or not the node is considered reachable and whether the data on that node is up-to-date. The nodes can still push updates to the management service, but ultimately have to observe the acknowledgement of the management service before assuming a new state. The management service itself (i.e. the management logic) can also change the state of a node, for example in case it has detected that a node needs a resync. This is also implemented by sending an update from the state decision component of the management service to the distributed state machine.

3.3 *Joint test with BeeGFS and MonetDB*

In order to test the interoperability of several components envisaged to run in the ExaNeSt project, an integration test was performed as a collaborative effort between Fraunhofer ITWM and MonetDB Solutions. To simulate the ARM environment, an ARM-based virtual machine was used. The physical host running this VM had four SSDs. An instance of BeeGFS was set up to allow the virtual machine parallel access to the SSDs, which is meant to simulate a BeeOND instance running on ExaNeSt compute nodes. To simulate direct access to a flash storage device inside an ExaNeSt compute node at the same time, a partition on one of the SSDs was set aside, and the virtual machine was given direct access to that partition. BeeGFS was using one metadata server (running on one of the disks not shared with the VM), and four storage targets (one on each SSD). The host machine used has 40 CPU cores and 64GB of RAM, of which only 4 cores and 8GB were allocated to the VM. This ensured that enough resources were available to run the VM as well as BeeGFS concurrently on the same hardware. Several database benchmarks were then performed on the MonetDB instance running inside the VM. The description of these benchmarks, as well as a discussion of the results, can be found in deliverable D2.3.

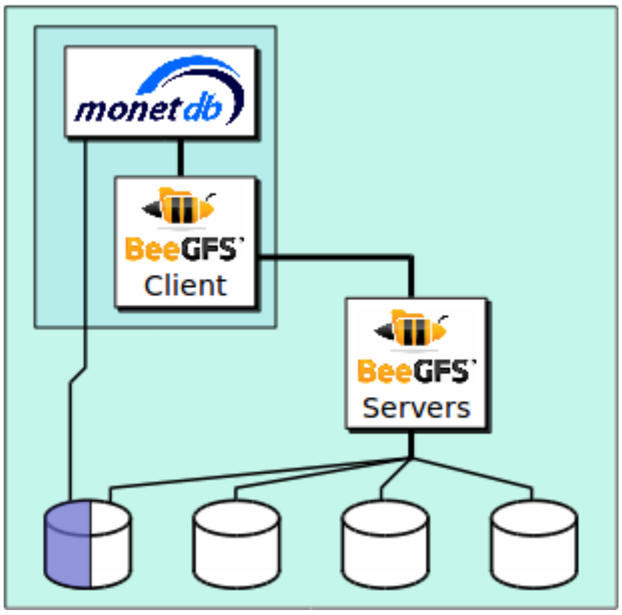


Figure 3.2. *Schematic of the BeeGFS and MonetDB test setup. Top Left: MonetDB and BeeGFS client running in an ARM VM, Right: BeeGFS server-side software running on the host machine, accessing four SSDs*

This setup is flexible. If during the optimization phase of the project, more extensive tests should be done, it can be easily extended to running multiple VMs. While now, only one SSD is accessed directly by a VM, in a multi-VM setup, each VM can have access to a partition on one of the other three SSDs, while the rest of the storage space is shared between the BeeGFS metadata and storage services. It is also imaginable to access a second BeeGFS instance, which is hosted outside the machine on a number of dedicated storage servers with spinning disks, to simulate a two-tier approach with a global BeeGFS and a BeeOND-based cache layer on flash storage devices.

4. VM-to-VM Communication based on MPI and RDMA Virtualization

[Contributed by VOSYS]

This section describes the implementation of the generic API Remoting framework. The framework is used to support the RDMA device para-virtualization and the intra-node OpenMPI acceleration. As described in a previous deliverable, D4.2, API Remoting is a software para-virtualization solution which depends on split co-operative processes design, as illustrated in Figure 4.1.

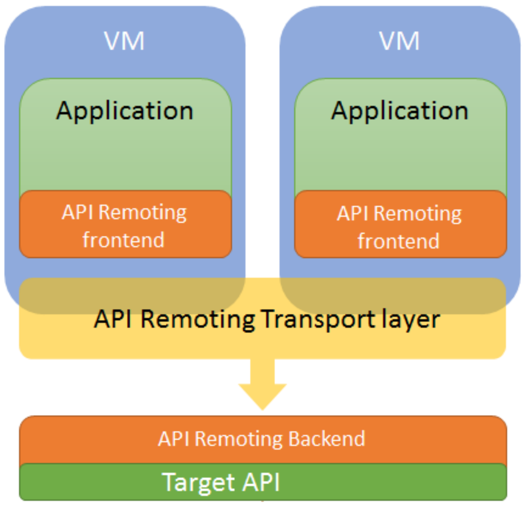


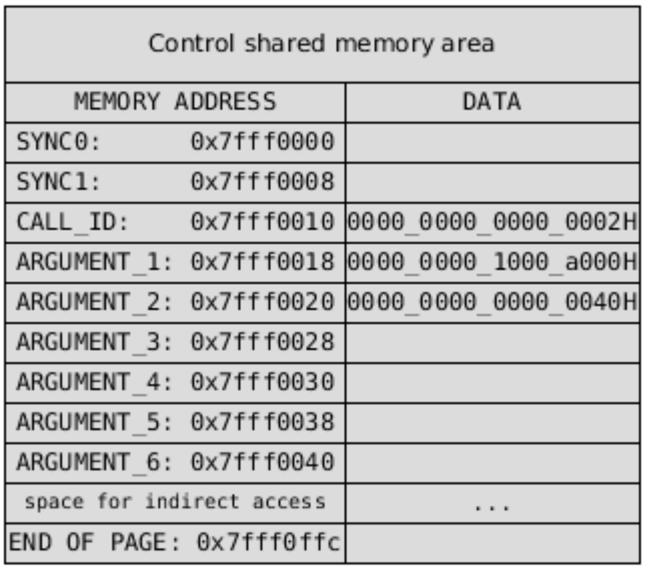
***Figure 4.1.** API Remoting abstract model.*

The design of the framework provides a 1-to-1 association between a guest process and a corresponding backend thread. A backend listener thread keeps track of a set of *eventfd* file descriptors that are handled and updated by a kernel module using the poll(2) system call. In case a new guest application needs to access the API Remoting library, a new backend thread is spawned and assigned to handle the specific guest application's requests.

During the exploration, we have identified two groups of application operations, related to the RDMA user-space library. *Control path* operations are usually function calls that can transfer all their information from the guest to the host through their arguments. The calls do not depend on the application's state, i.e., accessing process' memory through pointers. Forwarding a *control path* call introduces a small overhead of a 8-word memory copy. On the other hand, *data path* operations require the guest application to access host's resources (e.g., memory mapped devices, allocated buffers, without hypervisor's involvement). Host and guest kernels are involved in this procedure in order to share the resource between the two processes.

The key point of the framework's design is the *transport* layer. The framework uses user-space shared memory as the transport layer in order to minimize the overhead due to memory copies. Two allocation-aware shared memory procedures have been implemented in order to make feasible memory sharing between a guest process and a host process. A guest-to-host procedure shares a buffer from the guest process address space with a host process and a host-to-guest procedure shares a buffer from a host process address space with a guest process. The allocation procedures serve different call's characteristics for *control* and *data* paths.

On a guest function call, the process exchanges data with the backend thread via a shared-memory control segment which is initialized between the two processes at the first API call. Figure 4.2 shows the structure of this memory segment, whose length is 1 page. This segment contains a synchronization primitive, and the description of the forwarded API call. An unsigned integer known at compile time from both frontend and backend differentiates each supported call of the target API. API calls with data structures can be forwarded with appropriate frontend and backend serialization. After the serialization procedure the data may be copied to the 'space for indirect access'.



Control shared memory area	
MEMORY ADDRESS	DATA
SYNC0: 0x7fff0000	
SYNC1: 0x7fff0008	
CALL_ID: 0x7fff0010	0000_0000_0000_0002H
ARGUMENT_1: 0x7fff0018	0000_0000_1000_a000H
ARGUMENT_2: 0x7fff0020	0000_0000_0000_0040H
ARGUMENT_3: 0x7fff0028	
ARGUMENT_4: 0x7fff0030	
ARGUMENT_5: 0x7fff0038	
ARGUMENT_6: 0x7fff0040	
space for indirect access	...
END OF PAGE: 0x7fff0ffc	

Figure 4.2. *Control shared memory area.*

The control area is shared among multiple processes, so they should use synchronization mechanisms to preserve the consistent state of the resource. Since the design of the transport layer is based on shared memory, a *sense reversal* memory barrier can be placed into the control shared memory area. The downside of this spinlock-based approach is that it may starve the system spinning on the lock. For this reason, a new primitive for processes synchronization was designed and implemented. The new synchronization solution relies on communication between the two kernels, based on Virtio messages. This allows the calling process to be put to sleep until the remote process calls the barrier primitive. Doing so, the system can schedule a different process while the remote process is performing the barrier call, without wasting CPU cycles spinning on a spinlock.

The last feature of the API Remoting framework is the ability to support asynchronous function execution, on a guest application, by issuing a POSIX signal. This is important when an API uses user-defined callback functions on completion events. Figure 4.3 shows the execution steps, starting from the generated signal on the host process (1), until the actual execution of the user defined callback function at the guest application (6). The signal forwarding is assisted by the host and the guest kernels.

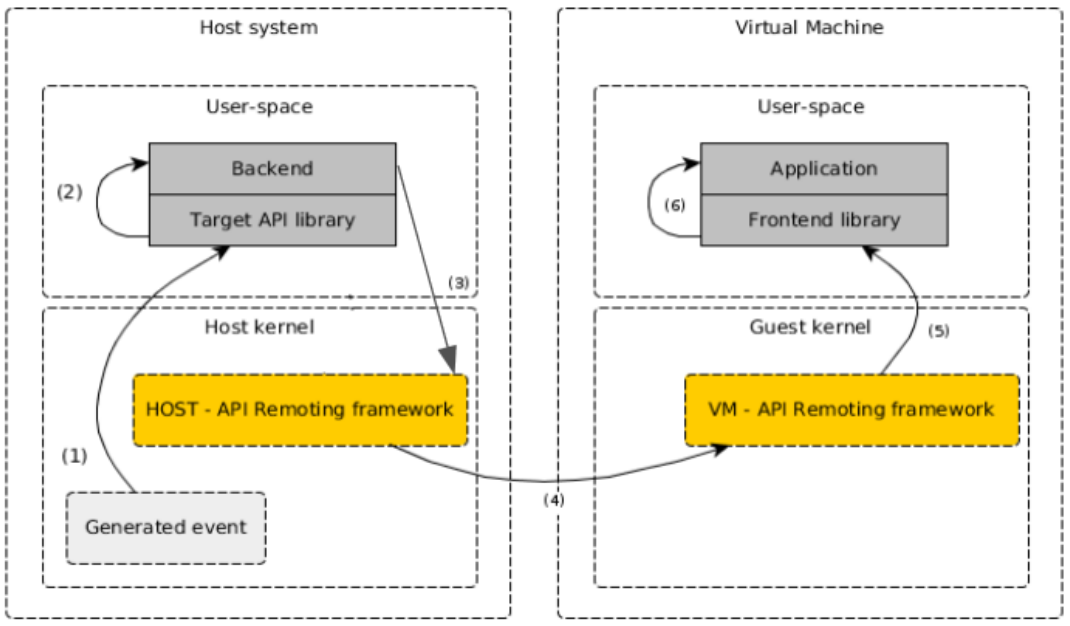


Figure 4.3. *Signal forwarding*

4.1 *RDMA virtualization*

This section describes the virtualization work and the adaptation of the API Remoting framework the RDMA device.

4.1.1 *Target Application Programming Interface (API)*

Applications can utilize the RDMA device through the 'libeusrv' library. The library takes care of the communication, which in turn programs the hardware device through memory mapped registers. The following list contains the library's API calls which are available in guest applications by means of API Remoting.

1. eusrv_dma_init - Initializes the userspace RDMA library
2. eusrv_dma_cleanup - Cleans up the userspace RDMA library
3. eusrv_alloc_buf_l - Allocates a DMA buffer on the local node
4. eusrv_alloc_buf_r - Allocates a DMA buffer on a remote node
5. eusrv_free_buf - Frees a DMA buffer
6. eusrv_get_buf - Queries for an allocated DMA buffer with an ID
7. eusrv_get_buf_id - Queries for an ID of a specific DMA buffer
8. eusrv_get_buf_memory - Gets a pointer to a DMA buffer in the process address space
9. eusrv_dma_submit_transfer - Starts a transfer between 2 DMA buffers
10. eusrv_dma_submit_transfer_from_mem - Starts a transfer from the local memory to a DMA buffer
11. eusrv_dma_submit_transfer_to_mem - Starts a transfer from a DMA buffer to local memory
12. eusrv_dma_status - Gets the status of a transfer

13. eusrv_dma_get_node - Queries information based on a node id

Calls to the virtualized RDMA API library are treated as *control path* operations. However, some of these calls require assistance from the host kernel to fully implement the functionality.

A DMA buffer allocation should give access from the guest application to the physical DMA buffer, in order to avoid overheads due to memory copies. Direct memory accesses from the guest process to the DMA buffer are achieved by inspecting the memory pages that construct the DMA buffer on the host, and remapping them to the guest address space. After the remapping procedure, the guest application can use standard load-store instructions to read and write the data from the DMA buffer. Accessing a host resource without involvement of the hypervisor is treated as *data-path* operations. Transfers between DMA buffers can be started from the guest application without any extra copy of the buffer payload. Transfers might also register a user-defined callback function in case of a successful transfer with a signal that will be forwarded from the backend application to the guest application.

4.2 *Intra-node VM-to-VM MPI One-Sided Communication acceleration*

In this section, we describe the virtualization of MPI's Remote Memory Access communication mechanisms for intra-node communication. The concept of guest and host co-operative processes of API Remoting will apply to this work with extensions to the MPI library which will be used by the guest processes. The extended MPI library will cooperate with a backend, on the host, which will act over shared memory.

4.2.1 *MPI One Sided Communication*

Remote Memory Access communication allows the initiator process to specify the send and the receive buffers (MPI memory windows) on a data transfer call. The process allocates and shares the valid memory window in a communicator with the collective call MPI_Win_allocate, or a variation when the communicator synchronization procedure differs.

4.2.2 *OpenMPI Modular Component Architecture (MCA)*

OpenMPI uses the term Modular Component Architecture (MCA) to classify the different MPI features in components; this increases flexibility by allowing to decide which one of them will be picked at runtime. An MPI application might load more than one MCA for the same type of functionality. For instance, an MPI_Send can make use of different MCAs according to the destination process' node. In fact, it might use at runtime an MCA based on sockets if the destination process is not running in the same node, while using a different MCA if the process resides in the same node. This flexibility of the MCA allows fine grained solutions able to dynamically adapt to different use cases.

4.2.3 *Virtualized OSC MCA*

This section describes the implementation of the virtualized MCA in order to support intra-vm RMA accelerated over shared memory.

The virtualized MCA is implemented and coded as a standard OpenMPI MCA in its codebase. OpenMPI existing facilities have been used and the communication primitives are based on the API Remoting framework. In that way, the internal state of the allocated MPI memory window is handled by the OpenMPI library, on the guest, but also the host is notified when a memory transfer or allocation should take place.

The backend process keeps track of the MPI window allocations. Whenever an MPI window allocation takes place, the virtualized MCA informs the host process in order to share the window buffer. Therefore, it is aware of the allocated window flavor and the synchronization policy that should apply on data transfer calls.

For data transfer calls (e.g., MPI_Put), the virtualized MCA will request to the backend a 'Put' command on specific memory windows. The request will be handled by the backend, which will make the copy over shared memory. Furthermore, the backend implements all the reduce operators which will be used to compute a result based on a guest memory window. Figure 4.4 illustrates an example state of the system after a collective MPI_Win_allocate() call. Each MPI process has a local memory window and three more windows that can perform RMA operations. During the window allocation, the backend process was informed with the addresses of the new windows. From that point the backend process is able to perform RMA operations over the registered MPI memory windows in the communicator.

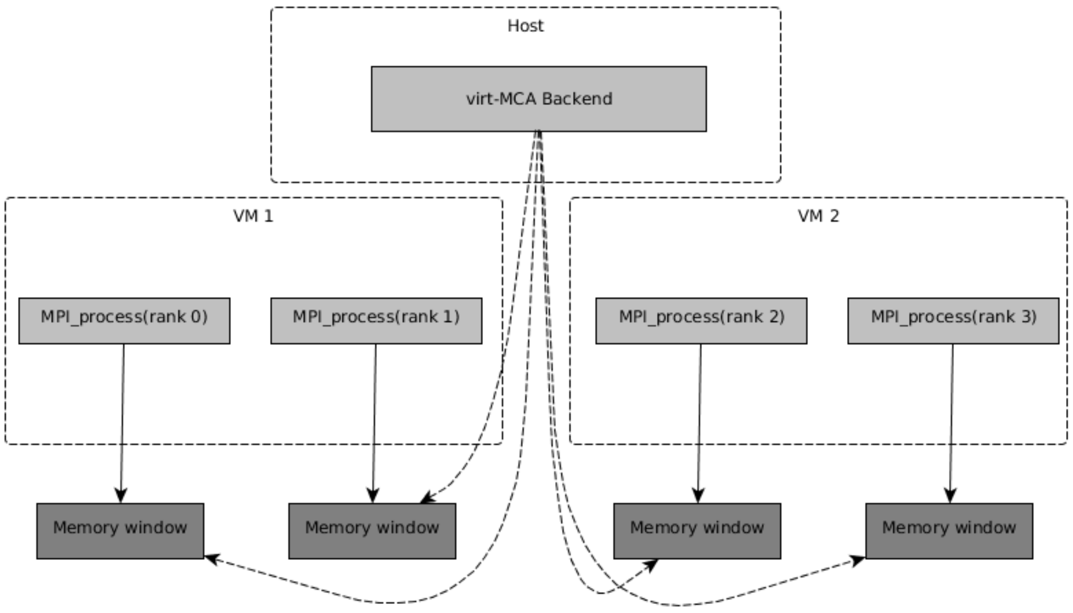


Figure 4.4. *Backend access to guest MPI memory windows*

5. Analytical Database Replication

[Contributed by MDBS]

In this section, we describe MonetDB's new replication mechanism, called *lazy logical replication*, that has been designed and implemented under the context of ExaNeSt to i) push MonetDB towards exascale (in terms of performance and scalability), ii) improvement the resilience of a MonetDB database server, and iii) facilitate the exploration of the ExaNeSt hardware and software resources for typical analytical database applications in cloud environments.

5.1 *Design*

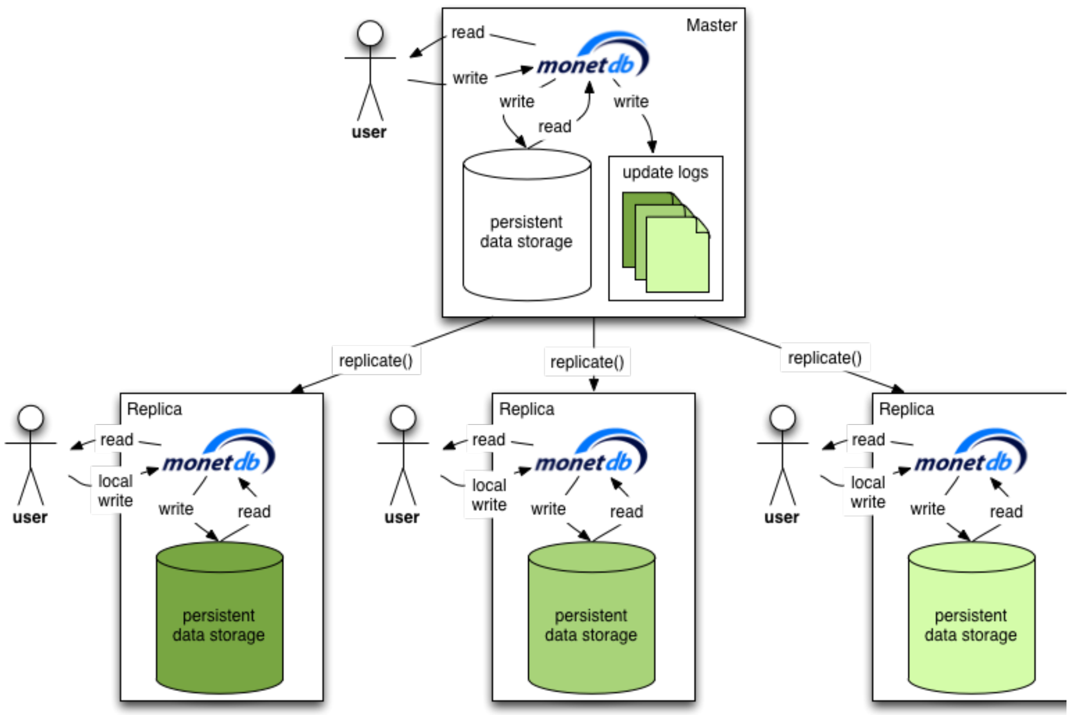


Figure 5.1. *architecture of lazy logical replication in MonetDB*

Lazy logical replication is an asynchronous logical replication management scheme using change set forwarding. Simplicity and ease of end-user control have been the driving arguments in its design and development. Figure 5.1 shows the architecture of lazy logical replication, which has been designed to achieve the following goals:

- Workload (re-)distribution
- Database scalability
- Database backup
- Availability and resilience
- Database multi-versioning
- (Partial) data (re-)partitioning

What is *lazy logical replication*?

In a data analytics environment, the workload on a database is mostly read-only. Applications grind the data for business insights and summarisations through visual dashboards. Updates are often collected as (micro-) batches and injected into the data warehouse at regular intervals.

If the ingestion rate increases, the updates become more complex, or the number of concurrent data analysis applications rises, it becomes mandatory to create a master/replica infrastructure, as shown in Figure 5.1:

- The master instance is responsible for handling all updates to the primary database, while replica instances are created to satisfy the responsiveness required by the applications.
- Updates on the primary database are propagated to the replicas in a lazy fashion, i.e. only when pulled by a replica.
- Next to replicating the primary database, a replica functions as a normal database server, so that its users can both run analytical queries as well as modify local data.
- Note that replications only go in one direction, i.e. from the master to the replicas. Although users of replicas can be allowed to modify their local data, but the modifications will stay on that replica (hence they are denoted as "local write" in Figure 5.1).

A key observation for this common business scenario is that the replicas may lag a little behind. Because data analysts often look at long-term patterns using statistical summarizations, and therefore the outcome is less dependent on what happened during the last minute. Furthermore, the replicated data warehouse is likely to run in a Cloud setting, or a cluster with a shared global filesystem. This creates room to simplify the synchronisation between instances, relying on the services provided by the filesystem. In particular, master and replicas share the same `<dbfarm>` directory.

Lazy logical replication relies on detecting change sets in the persistent tables at the master instance, which are collected in a transactional safe way, and replayed at the replica(s). Replays can be interrupted (i.e. instead of replying all change sets, replay the change sets only up to certain moment in time) to obtain time-warped copies of the master database.

When to consider lazy logical replication?

The goal of this extension module is to ease backup and replication of a complete master database with a time-bounded delay. This means that both the master and the replicas run at a certain heartbeat (e.g. in seconds), managed independently by each instance, by which information is made available by the master or read by the replicas. Such an instance can be freely used for query workload sharing, database versioning, and (re-)partitioning. For example, a replica can be used to support a web application which also keeps application specific data in the same instance, e.g. session information.

For a replication (also called a *clone*), we need either all update logs for the entire lifetime of a database, or a binary database snapshot with a collection of logs that have recorded all changes since the snapshot was created. Then, the logged updates are replayed against an empty database or the snapshot until a specific point in time or transaction id is reached, as identified by the clone itself. A backup is regarded as a simpler form of replication, because the solely purpose of a backup is to keep a copy of the original database, therefore it should not be used to process other queries.

Tables taken from a master can be protected against updates and inspections in a replica instance using the schema access policies defined by the master. Depending on the policy, a user of a replica might be permitted to update the local database of the replica (locally created tables or replicated tables), however, updates against replicated tables will not be automatically forwarded to the master.

Any transaction change set replay that fails stops the cloning process. By default, only persistent tables are considered for replication, and all constraints maintained by the master are carried over to the replicas. Updates under the 'tmp' schema, i.e. temporary tables, are ignored.

The underlying assumption of the techniques deployed is that the database resides on a proper (global/distributed) file system to guarantee recovery from most storage system related failures, e.g. using RAID disks or Log-Structured-File systems.

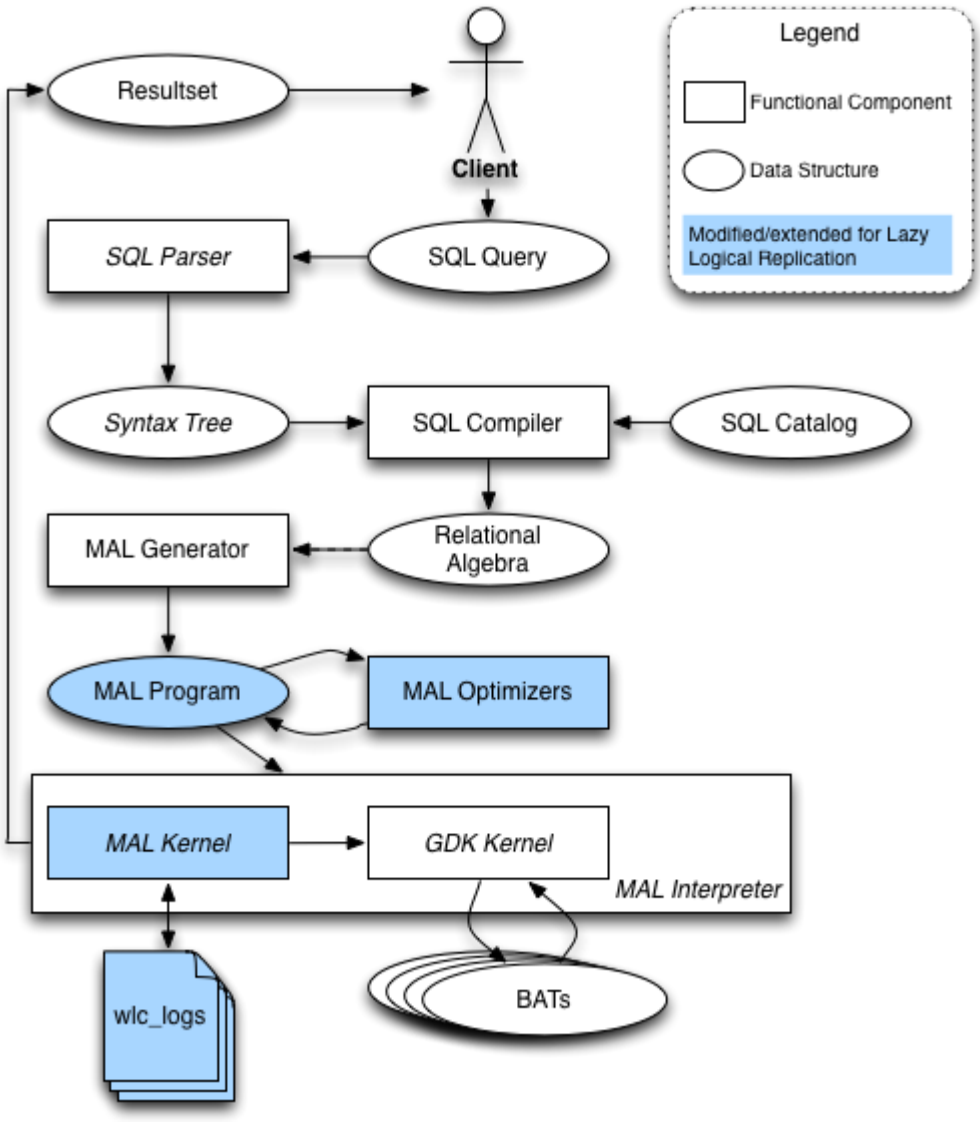


Figure 5.2*. implementation of lazy logical replication in MonetDB, highlighting the components modified and extended.*

5.2 *Implementation*

Figure 5.2 shows the MonetDB software stack with the components which have been modified or extended to implement the lazy logical replication highlighted. Because we have chosen to implement all user interactions for managing the master and replicas through SQL functions, nothing at the SQL language level (e.g. the parser, compiler and execution plan generator) needs to be changed. All changes were made in the lower level components that are involved in processing the physical query execution plans:

- "MAL[1] program" contains physical query execution plans, one per transaction.
- "MAL optimizers" is a collection of optimisers, e.g. the common expression detector and the dead code eliminator, that each optimises one aspect of a MAL program. A new optimiser has been added to detect the change set in a MAL program, i.e. MAL statements that modify the SQL catalogue or update the persistent data storage.
- "MAL kernel" is the component responsible for executing the MAL programmes. It interacts with the "GDK[2] kernel" component, which manages the columnar physical storage of MonetDB, to eventually carry out query executions.
- On a master, the MAL kernel has been extended to write the change set detected by the new MAL optimiser to the log files, called *wlc_logs* (WLC stands for WorkLoad Capture).
- On a replica, the MAL kernel has been extended to read the log files from the wlc_logs and replay the change sets on its local database. Note that, with the current implementation, a replica is only capable of reading WLC log files from a master instance under the same `<dbfarm>`.

How to set up a master instance?

The safest way to create a master instance is to put an empty database[3] into the "master" mode. Alternatively, one can stop any running server of an existing database, take a snapshot of this database (i.e. make a copy of its `<dbfarm>/<dbname>` directory), and restart this database to put it into the "master" mode. The snapshot is later used to initialise the replicas of this master instance. A database instance can be put into the "master" mode only once using the SQL command:

```
CALL master();
```

An optional path to the log record directory can be given to reduce the I/O latency, e.g. using a nearby SSD, or where there is plenty of space to keep a long history, such as an HDD or a cold storage location. By default, the command creates a directory `/<path-to>/<dbfarm>/<dbname>/wlc_logs` to hold all logs, and a configuration file `/<path-to>/<dbfarm>/<dbname>/wlc.config` to hold the state of the transaction logs, which contains the following `<key>=<value>` pairs:

```
snapshot=<path to a snapshot directory>
logs=<path to the WLC log directory>
```

[1] MAL (**M**onetDB **A**ssembly **L**anguage) is the MonetDB internal language used to denote physical query execution plans. www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

[2] GDK (**G**oblin **D**atabase **K**ernel) is the current columnar storage kernel engine of the MonetDB 5 database. www.monetdb.org/Documentation/mserver5-man-page

[3] An empty database is defined as the following: i) the database has just been created, e.g. with `monetdb create <dbname>`; ii) only its SQL catalogue has been initiated; but iii) no user schema or table has been created, and no data has been loaded to any user tables.

```
id=<next transaction id, starts with 0>
write=<timestamp of the last transaction recorded>
state=<1: started, 2: stopped>
batches=<next available batch file to be applied>
beat=<maximal delay between log files in seconds, default=10>
```

A missing "snapshot" path denotes that the master has been started from an empty database. The log files are stored as `<dbname>_<batchnumber>` in the "wlc_logs" directory. They belong to a snapshot. Each WLC log file contains a serial log of committed compound transactions. The log records are represented as ordinary MAL statement blocks, which are executed in serial mode. Each transaction is identified by a unique id, its starting time, and the responsible database user. The log records must end with a COMMIT to be allowed for re-execution. Log records with a ROLLBACK tag are merely for off-line analysis by the DBA.

A new transaction log file is created by the master at each heartbeat (in seconds) if there is any new transaction to log. The new log file is published (i.e. made accessible to the replicas) after the master has been collecting transaction records for the duration of the heartbeat. The default beat is 10 seconds, and it can be modified using the SQL command:

```
CALL masterbeat(<duration>);
```

Setting the master heartbeat to zero leads to one log file per transaction, and this may lead to a log directory with a potentially immense number of files. For production systems, a heartbeat of 5 minutes should balance the polling overhead in most practical situations. The log file is shared after "duration" seconds after the first transaction record was written into it.

The final step in the lifetime of a master instance is to stop transaction logging with the SQL command:

```
CALL stopmaster();
```

This marks the end-of-life time for a snapshot. For example, this can be particularly useful, when planning to do a large bulk load of the database. Stopping logging avoids a double write into the database. The database can only be brought back into the master mode using a fresh snapshot.

One of the key challenges for a DBA is to keep the log directory manageable, because it grows with the speed in which updates are applied to the database. This calls for regularly checking for their disk footprint, and taking a new snapshot as a new starting point of the master so that the old log files can be removed. A master instance has no knowledge about the number of clones and their whereabouts.

To ensure transaction ACID (Atomicity, Consistency, Isolation, Durability) properties[4], the WLC log records must be securely stored on a persistent disk as an integral part of a transaction processing. This incurs not only extra computation, but more importantly, a potentially significant amount of additional I/O. The I/O pressure can be notably alleviated by storing the database and logs files on an SSD or a Non-Volatile-Memory (NVM) device. The NVM devices available on the ExaNeSt compute nodes are particularly suitable for this purpose.

How to make a replica instance?

[4] https://en.wikipedia.org/wiki/ACID

Every replica starts off with a copy of the binary snapshot identified by 'snapshot'. A fresh database can be turned into a replica using the call:

```
CALL replicate('<mastername>');
```

It will grab the latest snapshot of the master and apply all available log files before releasing the database. Progress of the replication can be monitored using the `-fraw` option in `mclient`[5].

The clone process will iterate in the background through the log files, applying all updating transactions. An optional timestamp or transaction id can be passed to the `replicate()` command to apply the logs until a specific moment in time or a specific transaction. This is particularly useful when an unexpected disastrous user action, e.g. dropping a persistent table, has to be recovered from. In total, the following functions are available to manage the syncing behaviour of a replica:

```
-- make this instance a replica of <mastername> and
--   sync continuously when even a new log file is published.
CALL replicate('<mastername>');

-- sync all transactions until <TID> (but not included),
--   then stop automatic sync.
-- <TID> can be in the future,
--   then syncing will continue until <TID> has reached.
CALL replicate('<mastername>', <TID>);

-- sync all transactions executed before <timestamp>,
--   then stop automatic sync.
-- if <timestamp> is in the future, continue syncing
--   until <timestamp> has been reached.
-- eg use NOW() as <timestamp> brings the replica up to date
--   with the master, then pauze
CALL replicate('<mastername>', <timestamp>);

-- shortcut to replicate('<mastername>', <timestamp>),
--   if <mastername> is already know.
CALL replicate(<timestamp>);

-- shortcut to replicate('<mastername>', <TID>),
--   if <mastername> is already know.
CALL replicate(<TID>);

-- shortcut to replicate('<mastername>'),
--   if <mastername> is already know.
-- continue undisturbed synchronisation
CALL replicate();
```

Any failure encountered during a change set replay terminates the replication process, leaving a message in the *merovingian log*[6].

[5] https://www.monetdb.org/Documentation/mclient-man-page

[6] The log file of the MonetDB database server daemon tool, called `monetdbd`, which is use to manage multiple MonetDB servers. https://www.monetdb.org/Documentation/monetdbd-man-page

Auxiliary information, such as the state of the replication is stored in the configuration file `/<path-to>/<dbfarm>/<dbname>/wlr.config`, which contains the following `<key>=<value>` pairs:

```
master=<mastername>
batches=<next available batch file to be applied>
tag=<next transaction id to be processed>
limit=<stop replay transactions when limit is reached, -1: no stop>
beat=<maximal delay between log files in seconds, default=10>
```

Several additional SQL functions have been added to inspect the state of the master or a replica:

```
-- returns the timestamp of the last replicated transaction.
SELECT replicaClock();

-- returns the transaction id of the last replicated transaction.
SELECT replicaTick();

-- return the timestamp of the last committed transaction
--   in the master.
SELECT masterClock();

-- return the transaction id of the last committed transaction
--   in the master.
SELECT masterTick();
```

A running example

In the following table, we show a short replication session with a master and a replica. First, we create a `<dbfarm>` to contain the two MonetDB server instances, using the command line tools, `monetdbd` and `monetdb`[7], of the MonetDB database server daemon:

```
# create and start a dbfarm in "/tmp/lazy_rep"
$ monetdbd create /tmp/lazy_rep
$ monetdbd set port=60001  /tmp/lazy_rep
$ monetdbd start /tmp/lazy_rep

# create and start an instance which will be the "Master instance"
$ monetdb create mst; monetdb release mst; monetdb start mst
created database in maintenance mode: mst
taken database out of maintenance mode: mst
starting database 'mst'... done

# create and start a second instance which will be the "Replica instance 1"
$ monetdb create rep1; monetdb release rep1; monetdb start rep1
created database in maintenance mode: rep1
taken database out of maintenance mode: rep1
starting database 'rep1'... done
```

Now we can connect to these three instances to replicate transactions executed on "mst" to "rep1", as shown in the table below. Here we use MonetDB's command line client tool `mclient`. For the replica instance, we use the `-fraw` option to monitor the replication

[7] Detailed information about `monetdbd` and `monetdb` can be found in https://www.monetdb.org/Documentation/monetdbd-man-page and https://www.monetdb.org/Documentation/monetdb-man-page.

progress. Empty lines are added to indicate the order in which SQL queries were executed on the two instances.

Master instance

```
$ mclient -d mst
sql>-- put this instance in master mode
sql>call master();
sql>call masterbeat(0);

sql>-- execute two transactions:
sql>create table tmp(i int, s string);
operation successful (3.688ms)
sql>insert into tmp values(1,'hello'),
(2,'world');
2 affected rows (2.641ms)
sql>select * from tmp;
+------+-------+
| i    | s     |
+======+=======+
|    1 | hello |
|    2 | world |
+------+-------+
2 tuples (1.676ms)
```

Replica instance

```
$ mclient -d rep1 -fraw
```

```
sql>-- replicate 'mst' until TID 1:
sql>call replicate('mst', 1);
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    0 # length
sql>-- replicate the first INSERT:
sql>call replicate('mst', 2);
#Waiting for replay scheduler to stop
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
```

Master instance

```
sql>-- add more data in mst:
sql>insert into tmp val-
ues(3,'blah'),(4,'bloh');
2 affected rows (1.778ms)
sql>insert into tmp val-
ues(5,'red'),(6,'fox');
2 affected rows (3.156ms)
sql>select * from tmp;
+------+-------+
| i    | s     |
+======+=======+
|    1 | hello |
|    2 | world |
|    3 | blah  |
|    4 | bloh  |
|    5 | red   |
|    6 | fox   |
+------+-------+
6 tuples (2.563ms)
```

Replica instance

```
sql>-- replicate more transactions:
sql>call replicate('mst', 4);
#Waiting for replay scheduler to stop
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
```

```




sql>--do some updates on mst:
sql>update tmp set i = 3 where i = 1;
1 affected row (1.807ms)
sql>update tmp set s = 'blah';
6 affected rows (1.449ms)
sql>select * from tmp;
+------+------+
| i    | s    |
+======+======+
|    3 | blah |
|    2 | blah |
|    3 | blah |
|    4 | blah |
|    5 | blah |
|    6 | blah |
+------+------+
6 tuples (1.814ms)






























sql>-- delete some tuples:
sql>select * from tmp;
+------+------+
| i    | s    |
+======+======+
|    3 | blah |
|    2 | blah |
|    3 | blah |
|    4 | blah |
|    5 | blah |
|    6 | blah |
+------+------+
6 tuples (3.799ms)
sql>delete from tmp where i < 4;
3 affected rows (4.084ms)
sql>select * from tmp;
+------+------+
| i    | s    |
+======+======+
|    4 | blah |
```

```
[ 3,    "blah"  ]
[ 4,    "bloh"  ]
[ 5,    "red"   ]
[ 6,    "fox"   ]

















sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
[ 3,    "blah"  ]
[ 4,    "bloh"  ]
[ 5,    "red"   ]
[ 6,    "fox"   ]
sql>-- turn on automatic replication to
sql>--   replicate all updates:
sql>call replicate('mst');
#Waiting for replay scheduler to stop
#wlr.process:'/tmp/lazy_rep/mst/wlc_logs/
/%s_000000000004' can not be accessed
#wlr.process:'/tmp/lazy_rep/mst/wlc_logs/
/%s_000000000005' can not be accessed
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
[ 3,    "blah"  ]
[ 4,    "bloh"  ]
[ 5,    "red"   ]
[ 6,    "fox"   ]
```

```
|    5 | blah |
|    6 | blah |
+------+------+
3 tuples (2.615ms)
```

```
sql>-- deletions auto. replicated
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    4 # length
[ 4,    "blah"  ]
[ 5,    "blah"  ]
[ 6,    "blah"  ]
```

```
sql>--clear the complete table:
sql>delete from tmp;
2 affected rows (2.016ms)
sql>select * from tmp;
+---+---+
| i | s |
+===+===+
+---+---+
0 tuples (2.832ms)
```

```
sql> auto. replication continues:
sql>select * from tmp;
% sys.tmp,     sys.tmp # table_name
% i,    s # name
% int,  clob # type
% 1,    0 # length
```

```
sql>-- test stopping master:
sql>create table tmp70(i int, s string);
operation successful (1.710ms)
sql>insert into tmp70 values(1,'hello'),
(2,'world');
2 affected rows (1.507ms)
sql>select * from tmp70;
+------+-------+
| i    | s     |
+======+=======+
|    1 | hello |
|    2 | world |
+------+-------+
2 tuples (3.983ms)
```

```
sql>-- before master is stopped:
sql>select * from tmp70;
% sys.tmp70,   sys.tmp70 # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
```

```
sql>call stopmaster();
sql>insert into tmp values(40,'after be-
ing stopped');
1 affected row (0.825ms)
sql>select * from tmp;
+------+---------------------+
| i    | s                   |
+======+=====================+
|   40 | after being stopped |
+------+---------------------+
1 tuple (1.861ms)
```

```
sql>-- after master is stopped:
sql>select * from tmp70;
% sys.tmp70,   sys.tmp70 # table_name
% i,    s # name
% int,  clob # type
% 1,    5 # length
[ 1,    "hello" ]
[ 2,    "world" ]
```

```
sql>-- this instance cannot be put into
sql>--   master mode again
sql>call master();
WARNING: logging has been stopped. Use
new snapshot
```

5.3 ***Next steps***

As of May 2017 the experimental code of lazy logical replication has been made open-source in the development branch of the MonetDB source code.

One of the next steps is performance benchmark. However, it is not a simple task, since there is no standard benchmark for this. Designing a benchmark to produce meaningful results will be a big challenge itself.

Features to be considered beyond the alpha release are for instance:

- *Selective replication*: a full replication may not always be necessary. To minimise the replication overhead, one would want to partition the log files based on some predicates, so that replication can be done per partition.
- *Fine-grained access control*: on the master instance, one should be able to define which replica is allowed to replicate which parts of the database (this is one form of selective replication), and with what type of permission, i.e. if the replicated data are read-only or read-write.
- *Log shipping*: the current implementation relies on shared file systems for the replicas to read the log files produced by the master. In a next step, we want to ship the log and snapshot files between different file systems to improve local access speed and improve scalability.
- *Automated replication management*: the MonetDB database server daemon tool suite needs to be extended to facilitate automated management of master/replicas. For instance, an easier way to create and share the snapshots, which currently requires much manual work.

6. Administration and Testing Tools

6.1 *Marvin 1.0 - a database profiler for MonetDB*

[Contributed by MDBS]

A key issue in the road towards a high performance application is to understand where and when resources are spent. This information can be obtained using different tools and at different levels of abstraction. Fine-grained, platform specific information can be obtained using existing profilers, such as valgrind[8], or hardware performance counters. However, for database management systems, it is equally important to have the profiling information at the coarser-grained level of relational algebraic operators, which are the basic building blocks of an SQL query.

Profiling SQL queries and algebraic operators is a highly specific and completely different task than profiling HPC applications, therefore, MonetDB comes with a set of its own profiling tools[9], geared at understanding the internal working, scheduling and potential resource bottlenecks raised by running queries. They are all based on a profiler event stream produced by a MonetDB database server upon request.

Marvin is a new tool in the MonetDB profiler family, developed under the context of ExaNeSt. Its back end is based on a renewed version of the profiler event stream produced by a MonetDB database server. Its front end is an interactive graphical web interface, designed and implemented using modern web front end tools, such as D3 and AngularJS. The architecture of Marvin is designed in such way that it can be easily extended/adapted to accommodate changes in the event stream produced by the database server, e.g. if the database server adds more information to the event stream.

In the previous deliverable D4.2 (Section 5.4), we have described the initial design and plan for implementation of Marvin. By now, we have finished developing Marvin 1.0, i.e. all planned features for profiling a stand-alone MonetDB server. In this section, we i) give a quick recap of the architecture of Marvin; ii) present the main features of Marvin 1.0 with screenshots; and iii) wrap up with plan for Marvin 2.0.

6.1.1 *Architecture*

The architecture of Marvin is shown in Figure 6.1.

First, at the bottom layer of Figure 6.1, the MonetDB database server has been extended to produce a stream of profiling events. The events can come either as ASCII text or in a JSON structure, and write it on a TCP socket. The raw JSON stream contains the same information as the single-line format[10], which includes wall-clock time, thread ID, MonetDB internal algebra statement (i.e. written in the MonetDB internal language MAL[11]) being executed and its status ("start" or "done"), estimated or actual execution time, number of disk blocks read/written, cumulative memory consumption, etc.

[8] www.valgrind.org
[9] www.monetdb.org/Documentation/Manuals/MonetDB/Profiler
[10] www.monetdb.org/Documentation/Manuals/MonetDB/Profiler/TraceFormat
[11] www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

Then, the Marvin back end engine (written in Python) will pick up the raw JSON stream and transform it, by performing various filtering and aggregation operations and injecting timing information, into the JSON format understood by the Marvin frontend. One of the main computations done here is the lift time of each variable. The Marvin backend engine will find for each variable (representing memory regions) when it is created, when it is used, and when it is destroyed. In addition, the Marvin backend engine will filter out the excess information in the raw JSON stream that is not consumed by the Marvin frontend.

Finally, the Marvin front end visualises the information in the Marvin JSON stream. The frontend GUI is built using the AngularJS[12] framework, and the D3.js graphic library[13]. Next to the dynamic graphic interface, an important feature of the Marvin GUI is being *interactive*. The JSON profiling events are streamed. Therefore, Marvin can continuously monitor the JSON stream to update all graphs in the GUI during the execution of a query.

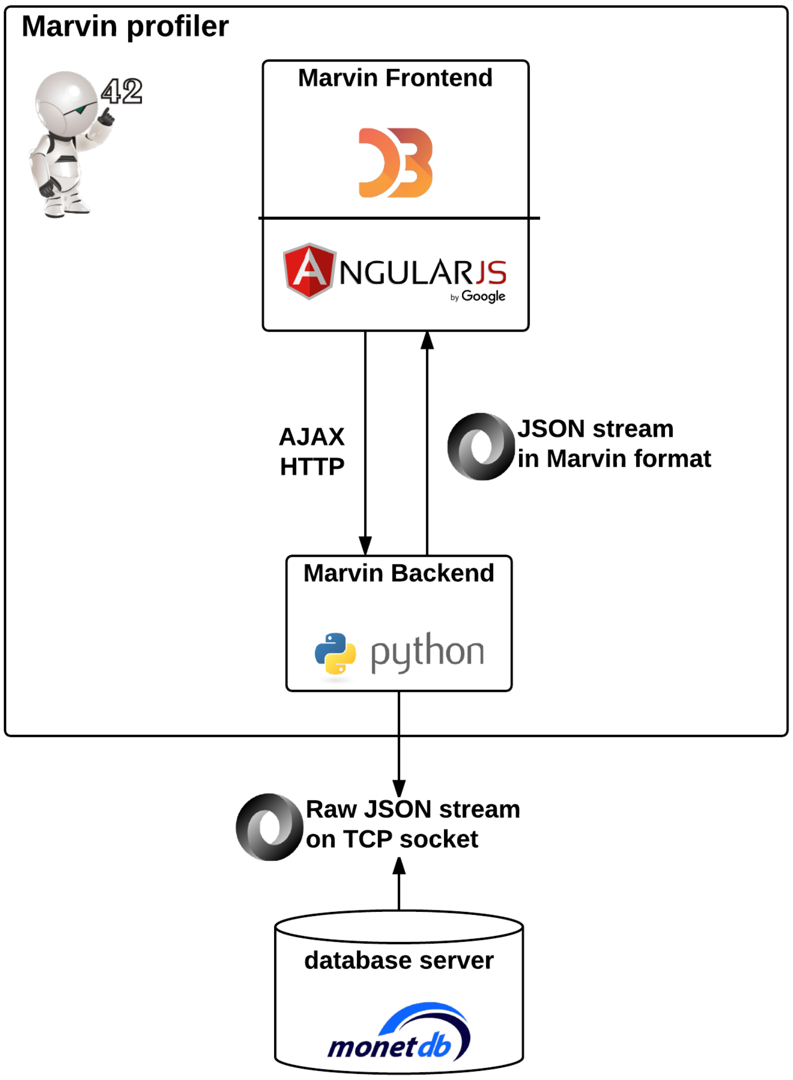


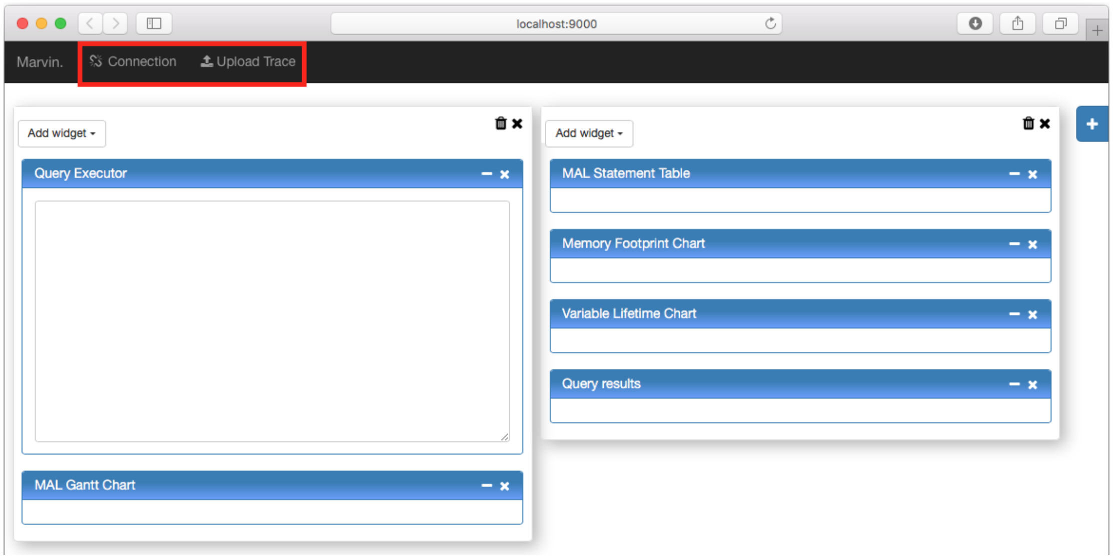
***Figure 6.1.** architecture of Marvin 1.0*

[12] https://angularjs.org
[13] https://d3js.org

6.1.2 *Main features*

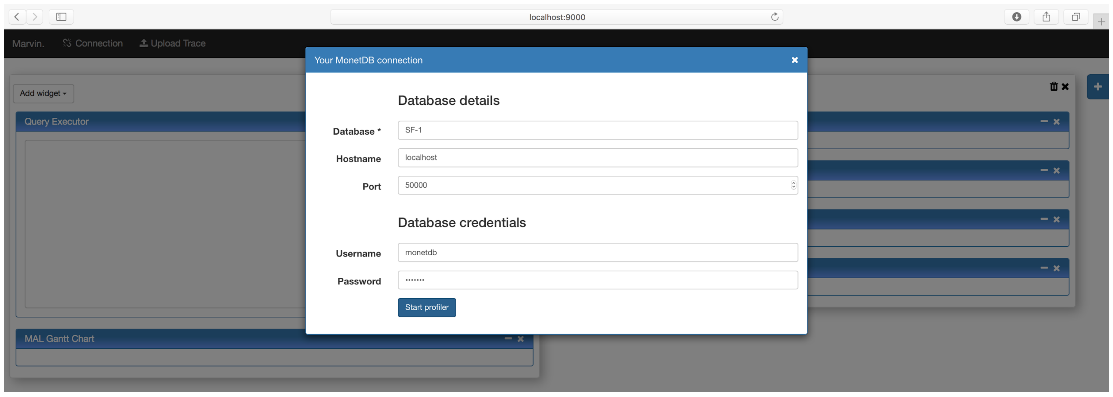
The screenshots below show the main features of Marvin. For each feature, we give a short description after the screenshot.

1. **Get profiling events**



This is the initial GUI of Marvin. In the menu bar at the top, there are two ways to get profiling events:

- "Connection": connect to a running MonetDB database server to execute queries and acquire their profiling data; or
- "Upload Trace": load query profiling data that have been saved earlier.



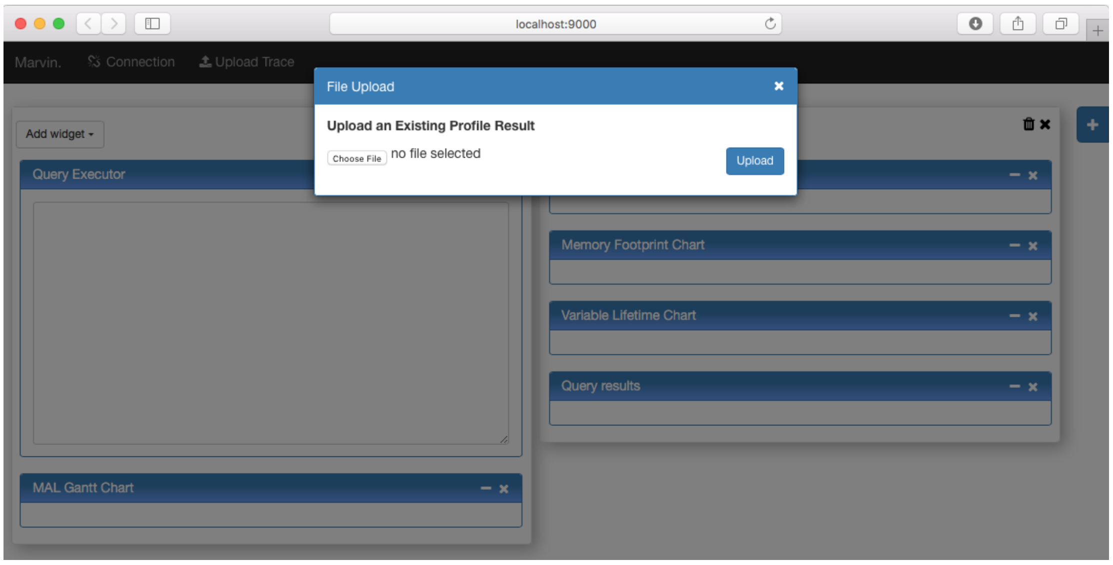
Database details

Database *	SF-1
Hostname	localhost
Port	50000

Database credentials

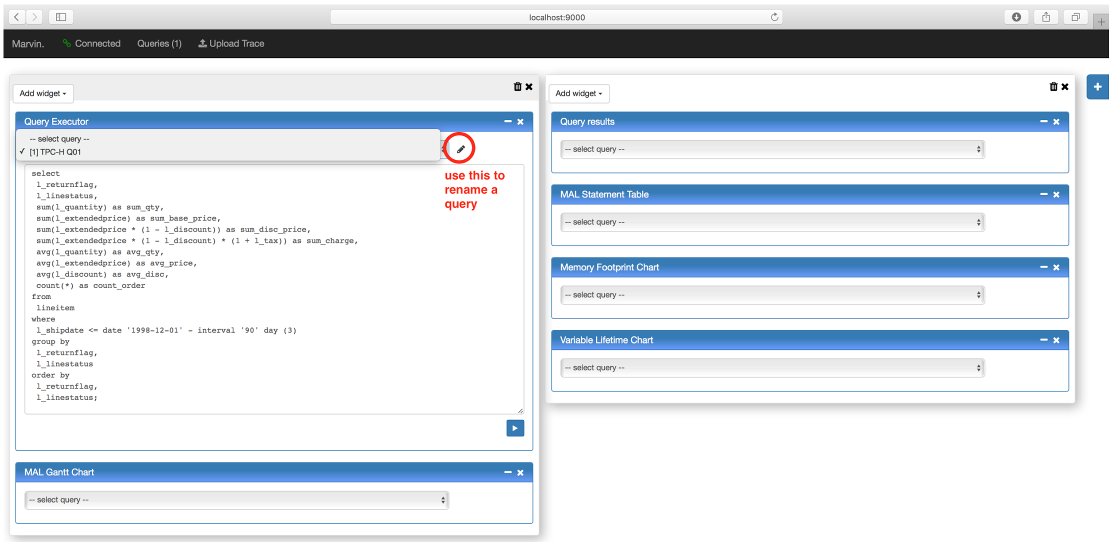
Username	monetdb
Password	·······

This is the interface to connect to a running database.



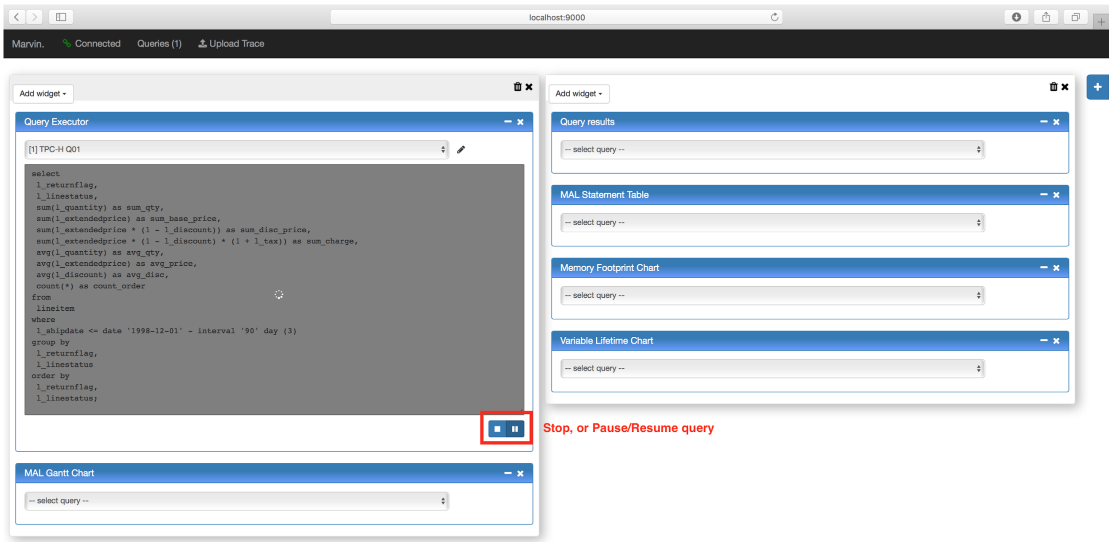
This is the interface to upload existing profiling events. The raw execution trace of each query is stored in a separate JSON file. One can upload multiple JSON profiling files here. Note that, the JSON files only contain profiling data, no query results.

2. Run/pause/resume query



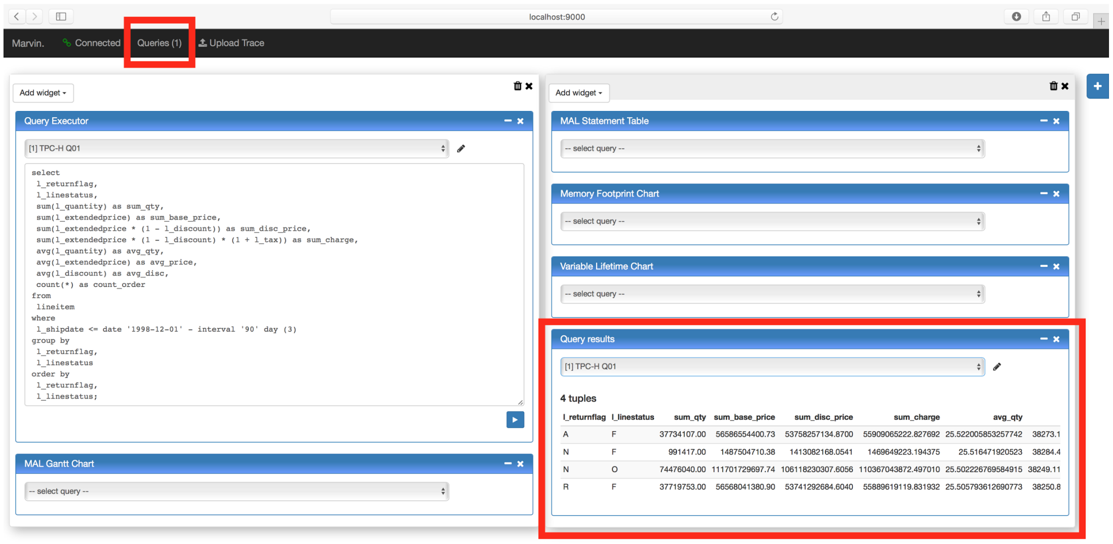
```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

If Marvin is connected to a running MonetDB database server, one can execute queries using the “Query Executor” widget. Each executed query will be saved in the query list. One can also give each query a name. Note that, because Marvin retrieves all query execution information from the database server, *all* queries that have been executed on the server will be listed here, including those executed by other client connections.



```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

One can stop, or pause/resume a long running query.



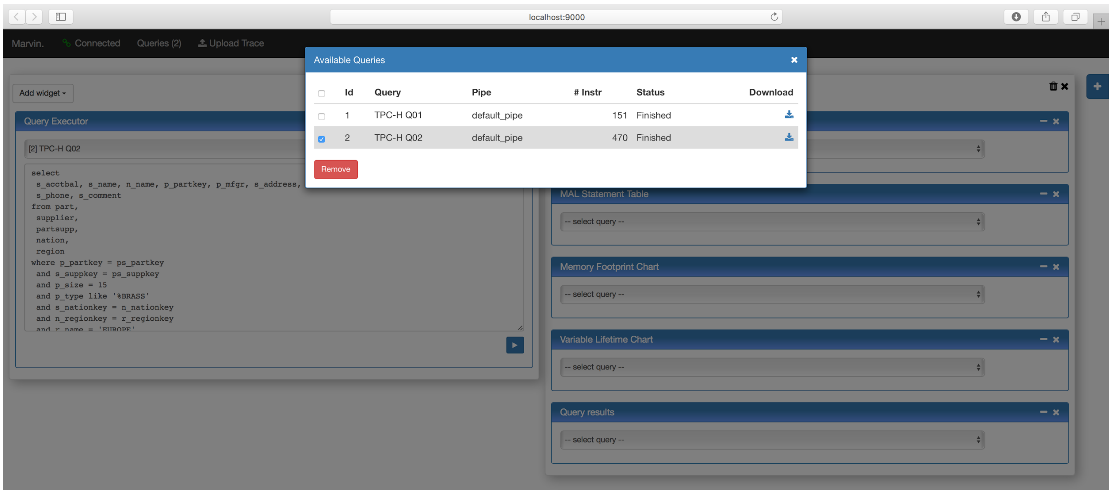
```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

[1] TPC-H Q01

4 tuples

l_returnflag	l_linestatus	sum_qty	sum_base_price	sum_disc_price	sum_charge	avg_qty	
A	F	37734107.00	56586554400.73	53758257134.8700	55909065222.827692	25.522005853257742	38273.1
N	F	991417.00	1487504710.38	1413082168.0541	1469649223.194375	25.516471920523	38284.4
N	O	74476040.00	111701729697.74	106118230307.6056	110367043872.497010	25.502226769584915	38249.11
R	F	37719753.00	56568041380.90	53741292684.6040	55889619119.831932	25.505793612690773	38250.8

In the "Query results" widget, one can view the query results in tabular format. In the menu bar, a new item "Queries" is shown with the number of all queries that have been executed on the database server (i.e. including those executed by other client connections). However, one can only view the results of queries that have been executed in the current Marvin session.



Available Queries

	Id	Query	Pipe	# Instr	Status	Download
☐	1	TPC-H Q01	default_pipe	151	Finished	⤓
☑	2	TPC-H Q02	default_pipe	470	Finished	⤓

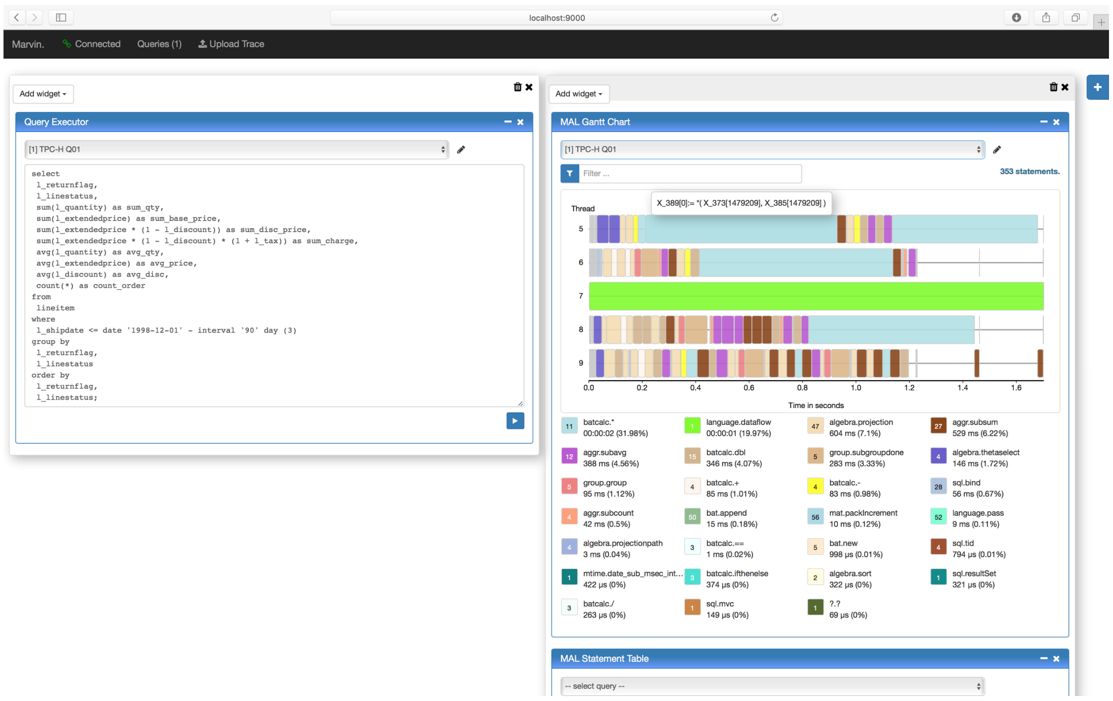
Remove

```
select
 s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
 s_phone, s_comment
from part,
 supplier,
 partsupp,
 nation,
 region
where p_partkey = ps_partkey
 and s_suppkey = ps_suppkey
 and p_size = 15
 and p_type like '%BRASS'
 and s_nationkey = n_nationkey
 and n_regionkey = r_regionkey
 and r_name = 'EUROPE'
```

By clicking on the menu item "Queries", one can view the list of "Available Queries":

- "Pipe" shows with which set of optimisers the query was executed. In MonetDB terms, each such set of optimisers is called an "optimiser pipeline", because the optimisers are applied in a pipelined fashion.
- "#Instr" shows how many MAL statements were executed for this query
- "Status" is the status of the query
- "Download" allows one to save the profiling data of this query in a JSON file on the local disk.
- Finally, with the checkboxes at the left side, one can select queries to "Remove".

3. MAL Gantt Chart



```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

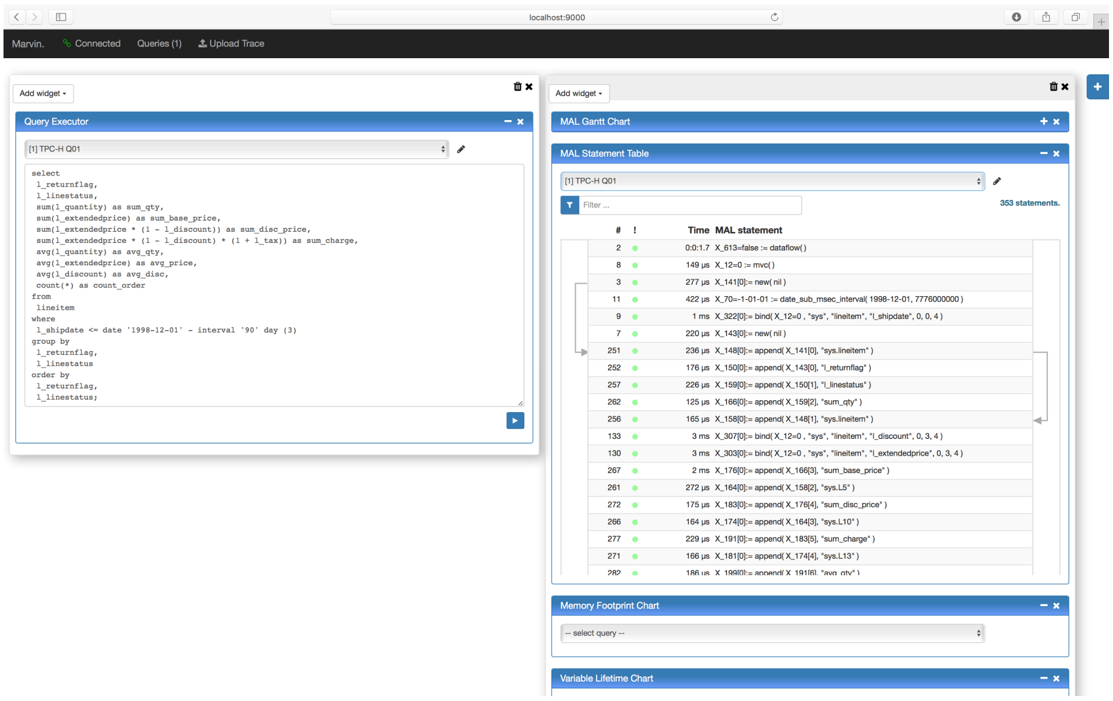
11	batcalc.* 00:00:02 (31.98%)	1	language.dataflow 00:00:01 (19.97%)	47	algebra.projection 604 ms (7.1%)	27	aggr.subsum 529 ms (6.22%)
12	aggr.subavg 388 ms (4.56%)	15	batcalc.dbl 346 ms (4.07%)	5	group.subgroupdone 283 ms (3.33%)	4	algebra.thetaselect 146 ms (1.72%)
5	group.group 95 ms (1.12%)	4	batcalc.+ 85 ms (1.01%)	4	batcalc.- 83 ms (0.98%)	28	sql.bind 56 ms (0.67%)
4	aggr.subcount 42 ms (0.5%)	50	bat.append 15 ms (0.18%)	56	mat.packIncrement 10 ms (0.12%)	52	language.pass 9 ms (0.11%)
4	algebra.projectionpath 3 ms (0.04%)	3	batcalc.== 1 ms (0.02%)	5	bat.new 998 µs (0.01%)	4	sql.tid 794 µs (0.01%)
1	mtime.date_sub_msec_int... 422 µs (0%)	3	batcalc.ifthenelse 374 µs (0%)	2	algebra.sort 322 µs (0%)	1	sql.resultSet 321 µs (0%)
3	batcalc./ 263 µs (0%)	1	sql.mvc 149 µs (0%)	1	?.? 69 µs (0%)		

The "MAL Gantt Chart" gives an overview of the execution of a query:

- The x-axis shows the elapsed query execution time in seconds.
- The y-axis shows the activities of each thread over time: which MAL statement did it execute, and from when to when.

- One can zoom in/out in the gantt chart to have a better view of the executed MAL statements at each time stamp.
- When hovering the mouse of a coloured block, the corresponding will be shown
- Above the gantt chart,
    - the “Filter…” field at the left-side allows one to filter the displayed MAL statement, e.g. show only the grouping operators.
    - the total number of executed MAL statement is displayed at the right-side.
- Below the gantt chart, the colours legenda is shown
    - Each colour box represent one distinct MAL statement, identified by *<module name>.<function name>*.
    - The number inside a colour box is the total number of this MAL statement in the query execution.
    - Under each MAL statement is the total execution time of all occurrences of this statement, and its percentage of the total query execution time.

4. MAL Statement Table

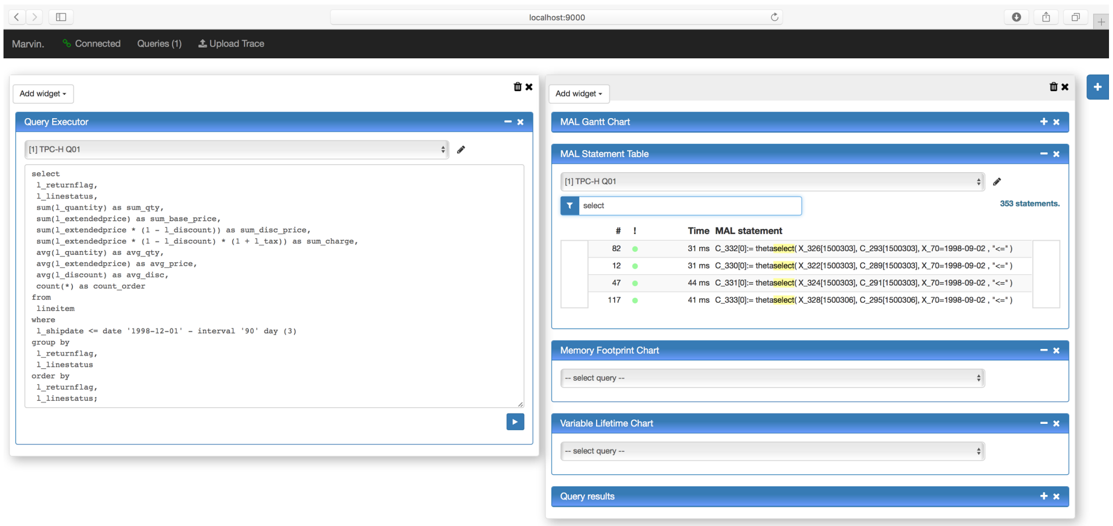


```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

#	!	Time	MAL statement
2	●	0:0:1.7	X_613=false := dataflow( )
8	●	149 µs	X_12=0 := mvc( )
3	●	277 µs	X_141[0]:= new(nil)
11	●	422 µs	X_70=-1-01-01 := date_sub_msec_interval(1998-12-01, 7776000000)
9	●	1 ms	X_322[0]:= bind(X_12=0 , "sys", "lineitem", "l_shipdate", 0, 0, 4)
7	●	220 µs	X_143[0]:= new(nil)
251	●	236 µs	X_148[0]:= append(X_141[0], "sys.lineitem")
252	●	176 µs	X_150[0]:= append(X_143[0], "l_returnflag")
257	●	226 µs	X_159[0]:= append(X_150[1], "l_linestatus")
262	●	125 µs	X_166[0]:= append(X_159[2], "sum_qty")
256	●	165 µs	X_158[0]:= append(X_148[1], "sys.lineitem")
133	●	3 ms	X_307[0]:= bind(X_12=0 , "sys", "lineitem", "l_discount", 0, 3, 4)
130	●	3 ms	X_303[0]:= bind(X_12=0 , "sys", "lineitem", "l_extendedprice", 0, 3, 4)
267	●	2 ms	X_176[0]:= append(X_166[3], "sum_base_price")
261	●	272 µs	X_164[0]:= append(X_158[2], "sys.L5")
272	●	175 µs	X_183[0]:= append(X_176[4], "sum_disc_price")
266	●	164 µs	X_174[0]:= append(X_164[3], "sys.L10")
277	●	229 µs	X_191[0]:= append(X_183[5], "sum_charge")
271	●	166 µs	X_181[0]:= append(X_174[4], "sys.L13")
282	●	186 µs	X_199[0]:= append(X_191[6], "avg_qty")

The “MAL Statement Table” shows all MAL statements that have been executed for the selected query, ordered by the time when a statement is finished. MAL is an extremely simple language. It consists of single line statements, and the result of each statement is always assigned to a variable.

When hovering the mouse over a MAL statement, the incoming arrows show where the inputs of this statement come from, while the outgoing arrows show where the results of this statement will be used.



[1] TPC-H Q01

```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

[1] TPC-H Q01

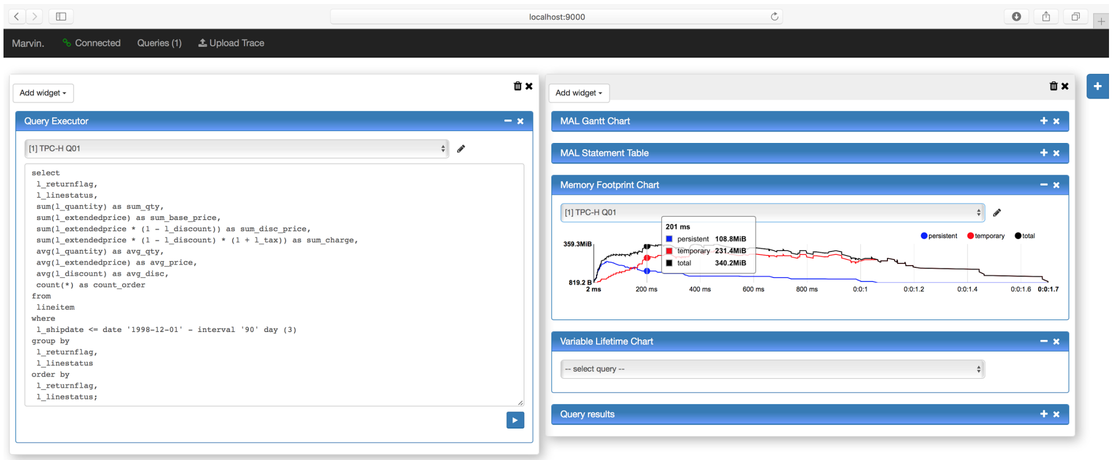
select

353 statements.

#	!	Time	MAL statement
82	●	31 ms	C_332[0]:= thetaselect(X_326[1500303], C_293[1500303], X_70=1998-09-02 , "<=")
12	●	31 ms	C_330[0]:= thetaselect(X_322[1500303], C_289[1500303], X_70=1998-09-02 , "<=")
47	●	44 ms	C_331[0]:= thetaselect(X_324[1500303], C_291[1500303], X_70=1998-09-02 , "<=")
117	●	41 ms	C_333[0]:= thetaselect(X_328[1500306], C_295[1500306], X_70=1998-09-02 , "<=")

The MAL Statement Table can easily contain hundreds or even more lines. One can scroll through the long list, or filter out the desired statements.

5. Memory Footprint Chart



[1] TPC-H Q01

```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

The "Memory Footprint Chart" shows MonetDB's memory consumption during the execution of a query:

- The x-axis shows the elapsed query execution time in milliseconds.
- The y-axis shows the consumed memory in MBs
- The blue line shows the memory consumed by the "persistent" data in the database.
- The red line shows the memory consumed by the "temporary" data, which are the intermediate data generated during the query execution.
- The black line shows the total memory consumption ("persistent" + "temporary")
- When hovering the mouse over the chart, a text box is displayed with the amount of "persistent", "temporary" and "total" memory at a particular point in time.

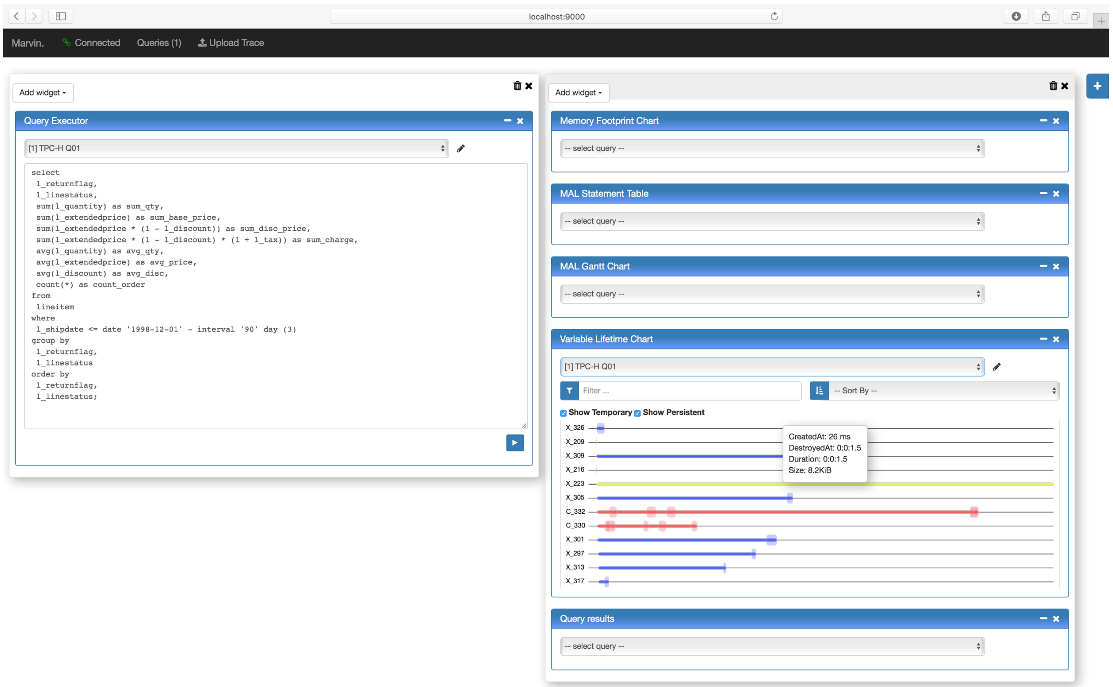
The chart displayed here shows the memory consumption of MonetDB during the execution of the TPC-H query 1. This is a typical behaviour of MonetDB:

- At the beginning of the query execution, the blue line grows fast, because all necessary persistent data are memory mapped to be processed.
- During the query execution, the blue line gradually drops, because once the persistent have been processed and no longer needed, they are memory unmapped.
- The red line gradually grows during query execution when more MAL statements

have been computed and their results are still needed.

- The red line gradually drops in a later stage of the query execution, when more and more intermediates have been consumed and no longer needed.

6. Variable Lifetime Chart



```
select
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
from
 lineitem
where
 l_shipdate <= date '1998-12-01' - interval '90' day (3)
group by
 l_returnflag,
 l_linestatus
order by
 l_returnflag,
 l_linestatus;
```

CreatedAt: 26 ms
DestroyedAt: 0:0:1.5
Duration: 0:0:1.5
Size: 8.2KiB

The "Variable Lifetime Chart" shows the lifetime of each MAL variable:

- Again, the blue colour is for variables holding persistent data, while the red colour indicates variables holder temporary data.
- For each variable, the thin line in full red/blue colour shows its full lifetime, from its creation to when it is destroyed. The exact times and duration are shown in the text box, when hovering the mouse over a variable, together with the size of this variable.
- The red/blue colour boxes in a lighter shade show when a variable is actually used. Overlapping boxes indicate that the variable is being used by multiple MAL state-ments at that time.

The chart displayed here only shows a small number of variables. Similar to the "MAL Statement Table", the list of variables can be long, and one can scroll through the list to ex-amine all variables.

For this particular query, when one scroll through the list from top to bottom, one would be able to observe that most persistent variables live and are used at the left side of the chart (i.e. at the beginning of the query execution time), while increasingly more temporary variables are at the midden and right side of the chart. This matches the information displayed in the "Memory Footprint Chart" above.

Another observation that can be made from this chart (and the variables not visible in this screenshot) is that variables, both persistent and temporary, generally live much longer life in an idle state in the memory than the time they are actually used. Thus the "Variable Lifetime Chart" shows us the opportunities to reduce the memory footprint of MonetDB for a given query. This is on our todo list for future MonetDB improvements.

6.1.3 *Next steps*

The majority of the development work for Marvin 1.0 has been finished. We will continue maintaining and improving the software. The license for Marvin is not decided yet, but for ExaNeSt partners, it is being provided for free.

Marvin 1.0 focuses on profiling a single MonetDB instance, while in ExaNeSt we will run clusters of MonetDB instances. So Marvin 2.0 will be a database profiler for distributed MonetDB. Therefore, extensions and/or redesigning in all layers of Marvin will be required:

- At the back end layers (i.e. the MonetDB database server layer and the Marvin back end layer), we will examine two options: extend the database servers to exchange the raw JSON stream with each other, or extend the Marvin back end with the ability to communicate with multiple MonetDB servers.
- At the front end layer, the GUI needs to be redesigned to incorporate information of multiple MonetDB instances. The main challenge here is how to visualise the data in a concise yet informative way.

6.2 *Monitoring System*

[Contributed by INFN]

The requirements that the monitoring system of the ExaNeSt prototype should satisfy have been identified in D.4.1, and the architecture of the monitoring system, together with the proposed tools, have been described in D4.2. Here, after a brief summary of such framework, we report on the progress we made on the configuration and the development of the monitoring tools..

6.2.1 *Development of the monitoring architecture*

The main requirements for the monitoring system, detailed in the previous deliverables, are:

- Inform the administrator about the status of the nodes composing the parallel file system and keep track of the internal communications among the nodes, i.e. identify nodes, or file system processes within a node, that got stuck for some reasons during the communications with other nodes.
- For both metadata servers and storage servers, an overview page should show the general status of the nodes. More detailed pages should provide information about single nodes, with reports on disk space usage, data throughput, disk performance on read and write operations, number of work requests and time needed to accomplish them.
- Both aggregated and detailed statistics should be available for metadata operations (i.e. create, stat, set/get attributes, set/get ACL) and for storage operations (read, write, etc.).
- Uers should be able to select time ranges and to visualize averaged quantities.

Keeping these demands in mind, we propose a monitoring infrastructure which builds on the reliable experience acquired in our datacenter [MIC01] and on a series of widely used open source tools which harmonize with each other nicely, namely:

- Telegraf (https://docs.influxdata.com/telegraf/), a plugin-driven server agent for collecting metrics and sending them to a variety of other datastores (in our case, InfluxDB). Telegraf is open-source and licensed under the MIT License, and it is easy to install and maintain. A wide choice of input and output plugins is already available and makes it easily extendable.

- InfluxDB (https://influxdata.com/), an open-source time series database, to be used as a persistency layer for the metrics collected with Telegraf. InfluxDB is easy to install and maintain, and does not require external dependencies. Moreover, it can scale horizontally (in the enterprise version) and High Availability can be configured. InfluxDB is also released with the MIT License.
- Grafana (http://grafana.org/), an open-source application to plot via web time series. It is very flexible in creating dashboards and it has native integration with InfluxDB (among others). It is a multi tenant application with LDAP integration for authorisation. The open-source version is provided with an Apache2.0 License.

The monitoring architecture is sketched in the following Figure 6.3, and it is currently in place in our testbed at INFN-CNAF, where we are testing its scalability and working on the development of the dedicated sensors.

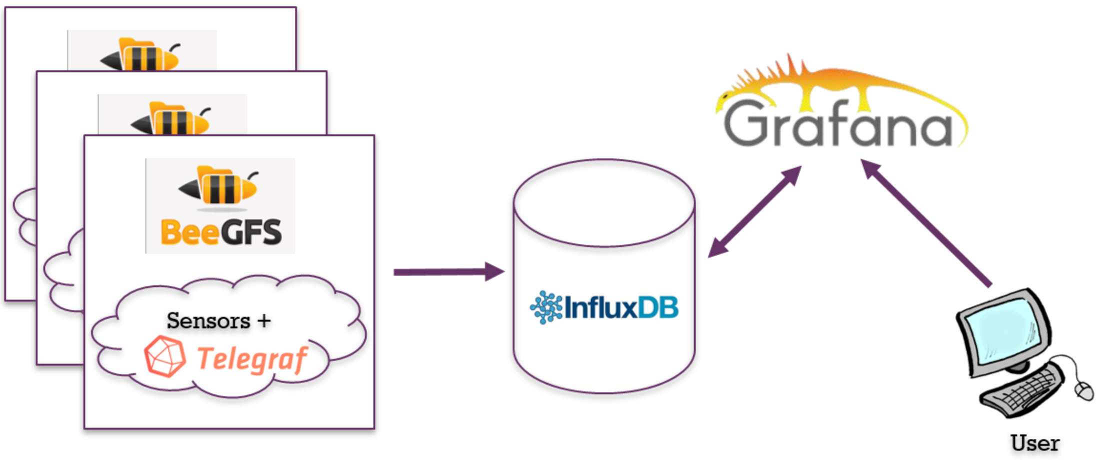
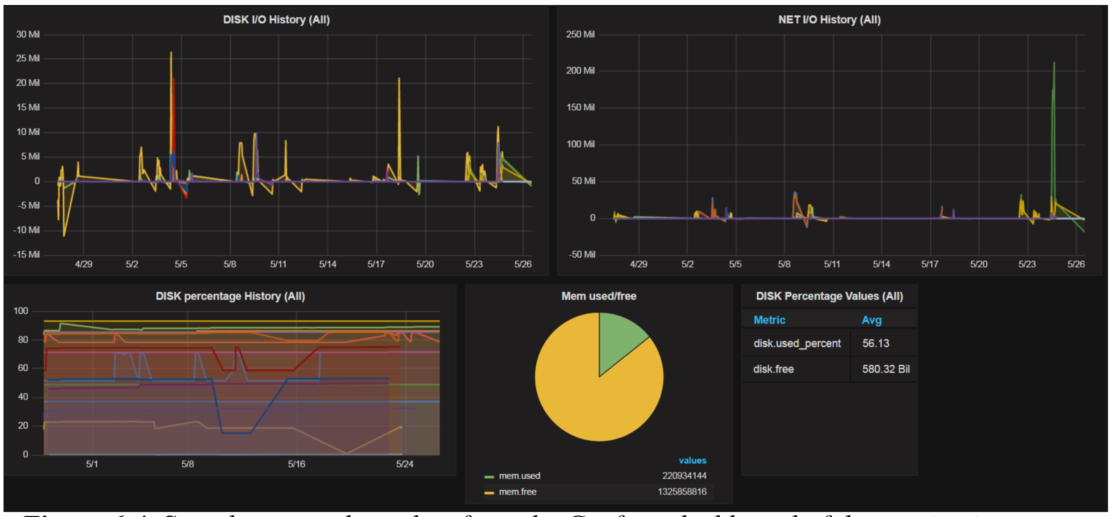


Figure 6.3. *Monitoring System Architecture.*

On each node of the ExaNeSt platform, dedicated sensors (both via plugins already available in the literature and via scripts developed around the native BeeGFS TUI) extract relevant metrics, which are collected and injected into InfluxDB by Telegraf, and plotted with a series of pre-defined dashboards using Grafana, where different end-users have access to different pages.

An example screenshot of the Grafana dashboard for the data throughput (for read and write storage operations) is shown in Figure 6.4.



	values
mem.used	220934144
mem.free	1325858816

DISK Percentage Values (All)

Metric	Avg
disk.used_percent	56.13
disk.free	580.32 Bil

Figure 6.4. *Sample screenshot taken from the Grafana dashboard of the monitoring system of the testbed installed at INFN-CNAF - this page is monitoring disk and network usage*

.

6.3 *Experiment automation, stress-load and fault injection tools*

[Contributed by INFN]

In this section, we describe a tool framework for performance stress-testing (storage traffic generator), and a fault injection framework to assist in the validation of performance and resilience targets. An important goal of the workload generator is to be able to initiate instances of different workload types from within virtual machines (VMs). Secondly, we support a set of different workload types that target different aspects of the system, for example, storage, networking infrastructure, memory, and processing resources. Additionally, there is the need to automate the initiation of realistic workloads, such as queries to databases. Among the key goals of the fault generator tool is the ability to replay *system faults* which are the effects of high load and congestion. Such system faults may be packet drops, packet re-ordering, packet duplication, packet corruption, and increased packet delay. Both tools offer the ability to specify the time when a workload instance or fault will be initiated. For the case of faults there is also provision to control their duration. It is important to note that the supported workload types are not fixed and new ones can be added in order to extend its functionality. Moreover, workload initiation is dynamically controlled through a simple configuration file that allows for different evaluation scenarios with different workloads.

6.3.1 *VM Load Injection Tool (VLITO)*

The goal is to support a wide range of workload types that may also require communication among virtual machines (e.g. MPI). A workload scenario can be described through three configuration files:

a) host machine (HM) configuration,
b) virtual machine configuration, and
c) workload configuration.

The HM configuration file is used to describe the host machines that will be involved in the workload scenario for hosting VMs. Adding or removing an entry from that file, adds or removes a new HM respectively. The VM configuration file describes the VMs that will be involved in the workload scenario and finally, the workload configuration describes instances of workloads, along with workload specific parameters. Instances of the aforementioned scenarios have been described in detail in ExaNeSt Deliverable 4.2.

The tool has been released to the partners through the ExaNeST git repository, and has been tested on the Juno-based prototype. We now describe the prerequisites for installing and using this tool, and also the configuration/setup actions that it performs on each HM involved.

<u>Linux software packages required</u>:

bridge-utils, qemu-system-aarch64, ssh-keygen.

<u>Kernel configuration flags, to be enabled on each of the host machines</u>:

CONFIG_BRIDG, CONFIG_VIRTIO, CONFIG_VIRTIO_BLK, CONFIG_VIRTIO_PCI, CONFIG_VIRTIO_MMIO, CONFIG_KVM, CONFIG_TU.

One of the HMs, the one where VLITO is run, is the *root host machine*. Different from all other HMs, on the root HM, three files must be available that will be used for creating guest OS for each VM instance installed:

a) template root file system image: cloned for each new VM installed
b) template kernel image: used for booting the guest OS of each VM
c) template initial ramdisk image.

The names of these images are specified through PATH/configs/input.conf where PATH is the directory where VLITO scripts reside.

On each HM that is involved in the workload scenario, configuration/setup actions need to be executed. This setup is cleaned upon HM reboot, and in that sense it constitutes soft state. The setup consists of:

a) adding a bridge
b) attaching each VM's back-end of the virtual interface to the bridge
c) enabling IPv4 forwarding (/proc/sys/net/ipv4/ip_forward) for forwarding packets received on a HM's interface to the bridge (packets that target a specific VM)
d) a routing table rule to guide packets that target a VM to pass through the bridge
e) routing rules for each VM IP that resides on a remote HM to use that HM as a gateway. If VMs on a HM are organized in a single subnet, the routing rules required are significantly reduced.

Overall, the current release of the VM load injection tool implements the functionality described in ExaNeSt deliverable D4.2. For the follow-up period in the ExaNeST project, we are considering improvements to address limitations that we have experienced while using the tool. First, with the current version of the tool to support a new workload type, all VM root file system images need to be updated manually. Another limitation is that currently, the workload scenario specification does not have the notion of time, that is, there is no control over the starting time and the duration of a specific workload instance. The tool needs also be tested with the Linux kernel configuration available on the more recent Trenz-based prototype. Specifically, we need to verify that the kernel configuration required by VLITO (mostly for providing virtualization support) is fully compatible with the kernel configuration required for loading the modules that support the Unimem architecture and the libraries made available to user space applications.

6.3.2 *ExaNeSt Network Fault Injection Tool (EN-FITO)*

As described in ExaNeSt deliverable D4.2, our goal is to provide functionality similar to that of NetEm in the Linux kernel. Netem is quite flexible in terms of fault emulation and allows emulating the following faults with respect to packets: drops, corruption, reordering, duplication, and increased delay. Netem is based on Linux TC filtering functionality which assumes the presence of an IP header for either filtering based on specific TCP/IP header fields or for determining offsets for moving within different parts of the packet. In the target architecture however, there will be no IP layer so another filtering mechanism is needed. Moreover, packets are passed to Linux's traffic control after they are processed by TCP/IP stack and before they are passed to the NIC's driver. ExaNeSt packets however, do not follow this path and thus modifying Netem is not possible.

Although the required functionality may seem straightforward, emulating the aforementioned network faults is not trivial due to the tight coupling of this tool with several key components of the target ExaNeSt's system. The system components with which there is strong dependence are:

a) Hardware blocks that handle communication at a higher level, such as, ExaNeSt packetizer, DMA engine, and mailbox.
b) Hardware blocks that handle communication at a lower level, such as, the block that performs the actual transmission over the physical serial links.
c) Components that realize the interconnect network including switching and/or routing logic.

Another complication with respect to the target network fault emulator is that at this stage of the ExaNeSt project, some details regarding the aforementioned components are not yet finalized. For this reason, some implementation details of the target tool might change in future versions. It should be noted though that these modifications are expected to concern only the implementation details, without limiting the tool's functionality.

The following figure illustrates the network path as perceived by the EN-FITO tool.

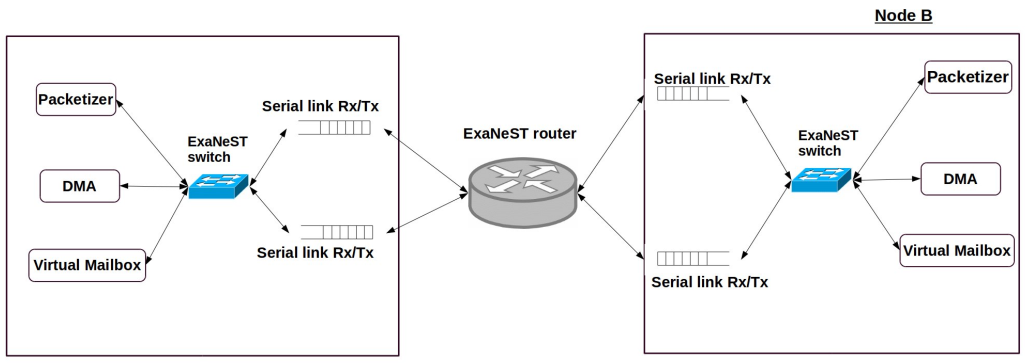


Figure 6.5. *Network path as perceived by the EN-FITO fault injection tool.*

As shown in Figure 6.5, two compute-nodes are connected together through logic that routes packets among boards, denoted as *ExaNeSt router*. Within each node (board), there are several hardware blocks that may generate ExaNeSt packets (with ExaNeSt referring to the type of network that interconnects all nodes and networking logic together). Currently only the ExaNeSt packetizer is able to generate ExaNeSt packets (EPs for the rest of the document) that can be delivered to a node's virtual mailbox. The packets generated by the aforementioned components pass through a switching logic (denoted as ExaNeSt switch) and are finally handled by a block that performs transmissions over a serial link. Then, the packets will be handled by a network device that performs routing (denoted as ExaNeSt router) that will deliver the EP to the node specified within EP's header. The final part of a packet's path is the component where the packet is written.

The current version of EN-FITO is configured through a simple input file describing the fault scenario to be actuated. The following table provides an example.

Fault Type	Ratio/ Delay Value (usec)	Source Node	Destination Node	Start TS (msec)	End TS (msec)
Drop	15%	2	3	100	200
Delay	120	1	3	0	200
Corrupt	12%	1	3	0	300

Note that the current EN-FITO implementation is aimed at interfering with the traffic of applications that run natively on host machines, that is, traffic that is not generated from within virtual machines. In the next versions of EN-FITO, we plan to enrich the configurations that are provided to it so specific faults can target traffic generated from within specific virtual machines.

The tool operated under the following operating environment assumptions:

a) Every node involved in the simulated scenario should have the same copy of the configuration file

b) To allow for different network faults to start at the same time across different nodes, all nodes involved in the simulated scenario should have Network Time Protocol (NTP) available. As also noted in [http://www.ntp.org/ntpfaq/NTP-s-algo.htm], the typical accuracy on the Internet ranges from about 5ms to 100ms so, assuming a 100 millisecond accuracy is a reasonable starting point. We plan to explore the option of using a local NTP server for allowing more fine-grain control of timestamps.

c) The current version of EN-FITO comprises a plugin within the user-space library that exposes the ExaNeSt packetizer hardware block to user space applications. This means that at the application level, there is no control over the ExaNeSt network header and possibly footer that is added to the packet and thus, there is no way to corrupt header or checksum's content (checksum for error detection). To enable corrupting packets in this way, the hardware block that computes checksums should offer EN-FITO the capability to selectively corrupt packets. Additionally, if this hardware block is virtualized by offering a different virtual channel/interface per process, it should support the ability to corrupt the next-packet-to-transmit on a specific channel/interface.

Currently only the ExaNeSt packetizer allows the transmission of ExaNeSt packets and its proper functionality is currently being tested. However in order to allow the development of EN-FITO to take place in parallel, two hardware blocks that are currently available on the Trenz-based prototype are exploited:

a) Virtualized mailbox, which implements up to 64 Hardware FIFOs. Each FIFO is accessible through the corresponding mailbox interface (MIF).

b) Axi-Packetizer The basic function of this block is to initiate an atomic packet to a user-defined destination, carrying user-defined data.

The user-space library through which the AXI packetizer is exposed generates an ExaNeSt Packet (EP) by adding a header and a footer (consistent with the current EP packet format) to the data that are to be transmitted-written to a specific node's mailbox. As an example, assume that node 2 needs to send N consecutive messages to node 3. A process running on node 2 will pass the data for each of the N packets to the AXI packetizer, and will denote as destination the virtual mailbox of a process running on node 3. Then the library, having parsed the EN-FITO configuration file, will randomly pick 15% of the packets transmitted within time window [100, 120] msec to be dropped.

6.3.3 *Checkpoint-restart simulation suite*

[Contribution of INFN]

Given that in typical HPC applications the most demanding operation in terms of stressing the storage system is represented by checkpoint and restart, we developed a synthetic MPI application that simulate precisely this activity.

This checkpoint-restart simulator was released in the Gitlab of the ExaNeSt project. The application, which is available in both a C++ and a Python version, launches a number of MPI processes, each one generating an array of a specified size. When all the processes are done (MPI_Barrier), they all start writing the array on a binary file to disk, and then perform fsync after flushing file. This step is intended to simulate a "checkpoint" step, in which a considerable amount of data in RAM (one array per process) is dumped on the file system. Then, the arrays are read from the previously written file, so to simulate the "restart from latest state" step.
The code allows to repeat writing and reading operations for a desired number of iterations in order to compute a meaningful average time.
The suite is parametrized and the number of MPI processes, the size of array handled by each process, the frequency of the store/load operations, the path where files reside can be configured..

The simulator is a test bench to simulate real case scenarios is HPC applications before that real use cases can be run on the ExaNeSt prototype.

7. Conclusion

[Section contributed by FHG]

In this deliverable, we presented the implementation of the ExaNeSt storage architecture. The ExaNeSt hardware design places the storage closer to the compute elements than traditional storage cluster based systems. This requires a number of software changes when handling the designed hardware architecture.

We described an extended Linux I/O path, with an optimized implementation for memory-mapped I/O called dmap. Then we presented the extensions to the distributed parallel file system BeeGFS, which include metadata mirroring, management service high-availability and an extended BeeGFS-on-demand cache layer. Afterwards, we showed the implementation of a generic API remoting framework, and its adaption for an RDMA device. In the next section, the implementation of an asynchronous replication mechanism for the column-store database MonetDB was described. The lase section introduces a database profiler, useful for maintaining performance of SQL queries, and also describes the implementation of a monitoring system for the ExaNeSt prototype as well as a way to automate stress-test experiments and network tests.

The software developed in Task 4.3 which is presented in this document is an integral part of the ExaNeSt project, where it is needed as a data access infrastructure. It provides ways for software applications to make use of the newly developed hardware and especially its storage components. It will be useful even outside the ExaNeSt project, for example in the HPC community, as well as for data analysts and the big data marked.

The components implemented as illustrated in this deliverable will be subject to further optimization, which will be done by the WP4 partners in Task 4.4.

8. References

[EHA12] Brian Van Essen, Henry Hsieh, Sasha Ames, and Maya Gokhale. 2012. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12). IEEE Computer Society, Washington, DC, USA, 731-735.

[EHA15] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. 2015. DI-MMAP--a scalable memory-map runtime for out-of-core data-intensive applications. Cluster Computing 18, 1 (March 2015), 15-28.

[FIO] FIO: Flexible I/O Tester. https://github.com/axboe/fio

[MIC01] D. Michelotto, S. Bovina *"The evolution of monitoring system: the INFN-CNAF case study"*, Proceeding of the The 22nd International Conference on Computing in High Energy and Nuclear Physics, CHEP 2016

[TPCH] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch/

[ETCD] Etcd – A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/

[OO14] D. Ongaro and John Ousterhout *"In Search of an Understandable Consensus Algorithm",* In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ACT 14). USENIX Association, Philadelphia, PA, USA, 305-319