www.qualiMaster.eu

QualiMaster

A configurable real-time Data Processing Infrastructure mastering autonomous Quality Adaptation

Grant Agreement No. 619525

Deliverable D5.4

Work-package	WP5: The QualiMaster Adaptive Real-Time Stream Processing Infrastructure
Deliverable	D5.4: QualiMaster Infrastructure V2
Deliverable Leader	TSI
Quality Assessors	Stefan Burkard
Estimation of PM spent	24
Dissemination level	PU
Delivery date in Annex I	30.9.2016
Actual delivery date	12.10.2016
Revisions	5
Status	Final
Keywords:	Infrastructure, Stream Processing, Adaptation, Pipelines

Disclaimer

List of Authors

Partner Acronym	Authors
TSI	Ekaterini Ioannou Minos Garofalakis Apostolos Nydriotis Nick Pavlakis Gregory Chrysos
LUH	Christoph Hube Tuan Tran Andrea Ceroni
SUH	Holger Eichelberger Cui Qin Christopher Voges Sascha El-Sharkawy
MAX	Tobias Becker
SPRING	Holger Arndt

Table of Contents

Executive summary

This deliverable provides an overall description of the final version of the integrated QualiMaster infrastructure. It provides an overview of the final architecture and of the components, layers and processing elements from all other technical work packages, i.e., WP2, WP3, and WP4. More details are provided for the parts of the infrastructure parts that have been improved based on the results of the first evaluation of WP6 as well as the previous experimental evaluation.

Furthermore, the deliverable describes the settings and methodology for our new, extended experimental evaluation over the complete QualiMaster infrastructure, the incorporated processing elements and the most representative processing pipelines.

Note that the content is complementary to the content of other deliverables. More specifically, detailed presentation of the processing elements and components was included in the recent deliverables D2.3, D3.3, and D4.3. In addition, the final experimental evaluation of the processing elements as well as the infrastructure and its components will be described and discussed in deliverables D2.4, D3.4, and D4.4, which are due at the end of the project.

1 Introduction

Deliverable D5.4 provides an overview of the final version of the QualiMaster infrastructure. It includes the incorporated components and layers as well as the processing elements and pipelines currently used for populating the infrastructure. The outcomes incorporated in the particular deliverable are results from all QualiMaster work packages and especially from the technical work packages, i.e., WP2, WP3, WP4, WP5 and WP6.

We first elaborate on the major functionalities that have been improved or incorporated during the last months. More specifically, we describe and discuss the explicit deployment of stable pipelines, the dynamic parallelization of storm components, our shedding framework, the replay mechanism that is also current incorporated in one of the new processing pipelines, the switch toolbox, our new extensions relayed to the infrastructure monitoring, the loose pipeline integration and the instantiation of the infrastructure.

In addition, we have also performed an extended experimental evaluation. With respect to this, we provide the used evaluation methodologies, list the data sets we used during this evaluation, and report the evaluation results. Our experimental evaluation touches three crucial aspects: (i) the components and layers composing the infrastructure; (ii) the processing elements incorporated in the infrastructure populations; and (iii) the processing pipelines that represent useful financial analysis processes.

In the following paragraphs, we first discuss the relation of the specific deliverable to other deliverables (Section 1.1). Then, we list the main publications that provide overview to the QualiMaster infrastructure (Section 1.2), joint work from the partners developing the infrastructure. Finally, we provide the structure of the remaining deliverable (Section 0).

1.1 Relation of other deliverables

The content of deliverable D5.4 is closely related to several other deliverables that were either recently submitted or will be submitted in the following months. More specifically, D5.4 contains a brief overview of the processing elements and infrastructure components/layers since their detailed descriptions were included in the following deliverables (submitted on month 31):

- D2.3 “Scalable, Quality-aware Data Processing Algorithms V2”,
- D3.3 “Hardware-based Data Processing Algorithms V2”, and
- D4.3 “Quality-aware Processing Pipeline Adaptation V2”.

Furthermore, we have already planned a new round of experimental evaluation for the developed processing elements. This evaluation will take place during the last months of the projects, and as such, will be described and discussed in the following upcoming deliverables (due on month 36):

- D2.4 “Final report on Scalable, Quality-aware Data Processing Methods”,
- D3.4 “Optimized Translation of Data Processing Algorithms to Hardware”,
- D4.4 “Quality-aware Processing Pipeline Modelling and Adaptation”, and
- D6.4 “Final Evaluation Report”.

1.2 Infrastructure-related publications

Part of the work performed by the QualiMaster partners for the specific work package was also submitted and accepted for publication at journals and workshops. The following table lists the publications those contributions are strongly related to the development performed for the QualiMaster Infrastructure. As shown, these are actually joint publications and this illustrates the collaboration and interconnections between the partners during the development.

1	Book Chapter	Holger Eichelberger, Claudia Niederee, Apostolos Dollas, Ekaterini Ioannou, Cui Qin, Grigorios Chrysos, Christoph Hube, Tuan Tran, Apostolos Nydriotis, Pavlos Malakonakis, Stefan Burkard, Tobias Becker, and Minos Garofalakis *"Configure, Generate, Run: Model-based Development for Big Data Processing"* European Project Space on Intelligent Technologies, Software engineering, Computer Vision, Graphics, Optics and Photonics, SCITEPRESS, 2016.
2	Workshop	First International Workshop on Big Data Processing - Reloaded (BDPR) EDBT/ICDT 2016 Joint Conference, 2016 **URL**: http://bdpr2016.l3s.uni-hannover.de/ **Programme Chairs**: Minos Garofalakis, Grraham Cormode **Organizers**: Claudia Niederée, Holger Eichelberger, Ekaterini Ioannou
3	Research Paper	Cui Qin, Holger Eichelberger *"Impact-minimizing Runtime Switching of Distributed Stream Processing Algorithms"* BDPR workshop, co-located at EDBT/ICDT, 2016
4	Research Paper	Apostolos Nydriotis, Pavlos Malakonakis, Nikos Pavlakis, Grigorios Chrysos, Ekaterini Ioannou, Euripides Sotiriades, Minos N. Garofalakis, Apostolos Dollas *"Leveraging Reconfigurable Computing in Distributed Real-time Computation Systems"* BDPR workshop, co-located at EDBT/ICDT, 2016

In addition to the above accepted publications, we are also currently working on other joint submissions. More specifically, this includes a visionary paper that discusses the novel research directions raises by the QualiMaster infrastructure, and a demonstration paper that will illustrate the functionality of the infrastructure as well as the capabilities of a few selected processing elements and pipelines.

1.3 Structure

The remaining sections of this deliverable are as follows: Section 2 provides a recap of architecture, data flow, and functionalities as well as detailed description of the recently included or improved infrastructure concepts. Section 3 presents the most representative financial processes and pipelines. Then, Section 4 describes the data sets and methodologies we used in our experimental evaluation. The following sections report the performed experimental evaluation with Section 5 reporting the results for the infrastructure components, Section 6 the results of evaluating the processing elements and of the processing pipelines. Finally, Section 7 provides conclusions, including our plans for next three months.

2 The QualiMaster Infrastructure

This section provides an overview of the QualiMaster infrastructure. For completion reasons, we start with a brief overview of architecture (Section 2.1). We then describe the main functionalities and components developed and incorporated after the submission of the D5.3 deliverable (Section 2.2-2.9).

2.1 Recap of Architecture, Data Flow, and Functionalities

As stated above, this part is included for completion reasons. A more detailed presentation of the QualiMaster architecture, data flow and functionalities is available in the previous deliverables of the particular work package, i.e., deliverables D5.1, D5.2 and D5.3.

Figure 1 gives a graphical illustration of the QualiMaster infrastructure and Figure 2 shows the overall package structure and dependencies. The infrastructure is comprised of two environments, the configuration that allows customization and instantiation towards a domain-specific application-oriented infrastructure and the runtime environment that performs the data processing following the definitions included in the configuration. The infrastructure uses two repositories that provide the actual configuration model and the software artifacts implementing the pipelines.

The runtime environment includes a number of logical layers: (i) the execution layer, responsible for the actual execution of processing tasks; (ii) the monitoring layer that collects statistics about the current load, resources and execution of the pipelines; (iii) the coordination layer that observes and modifies the actual execution in terms of monitoring and coordination; (iv) the adaptation layer that is responsible for runtime-decision making in order to optimize the execution; and (v) the data management layer, responsible to access and persist information, such as processing results, quality profiles, and adaptation history.

The execution layer is where the data analysis algorithms are executed. It includes processing systems such as Apache Storm and Apache Hadoop as well as specialized hardware systems as Maxeler data flow engines. All the necessary computational components for the pipeline execution are the responsibility of this layer. Since the computational components are provided by independent Algorithm Providers, they must follow explicit interfaces so that they can be used by the Qualimaster platform. Algorithms with similar functionality are included in the same family and implement the corresponding family interface. This enables Qualimaster to interchange them transparently and take advantage of their different quality tradeoffs.

The computational components implementation is independent of Apache Storm and different implementation forms are supported. The four different types of algorithm implementations are plain java algorithms, distributed algorithms (in terms of Storm topology), single hardware-based algorithms running on multiple Data Flow Engines and multiple hardware-based algorithms loaded together into a set of Dataflow Engines (DFEs). Each of these types is described in detail in the D5.2 and D4.1 deliverables.

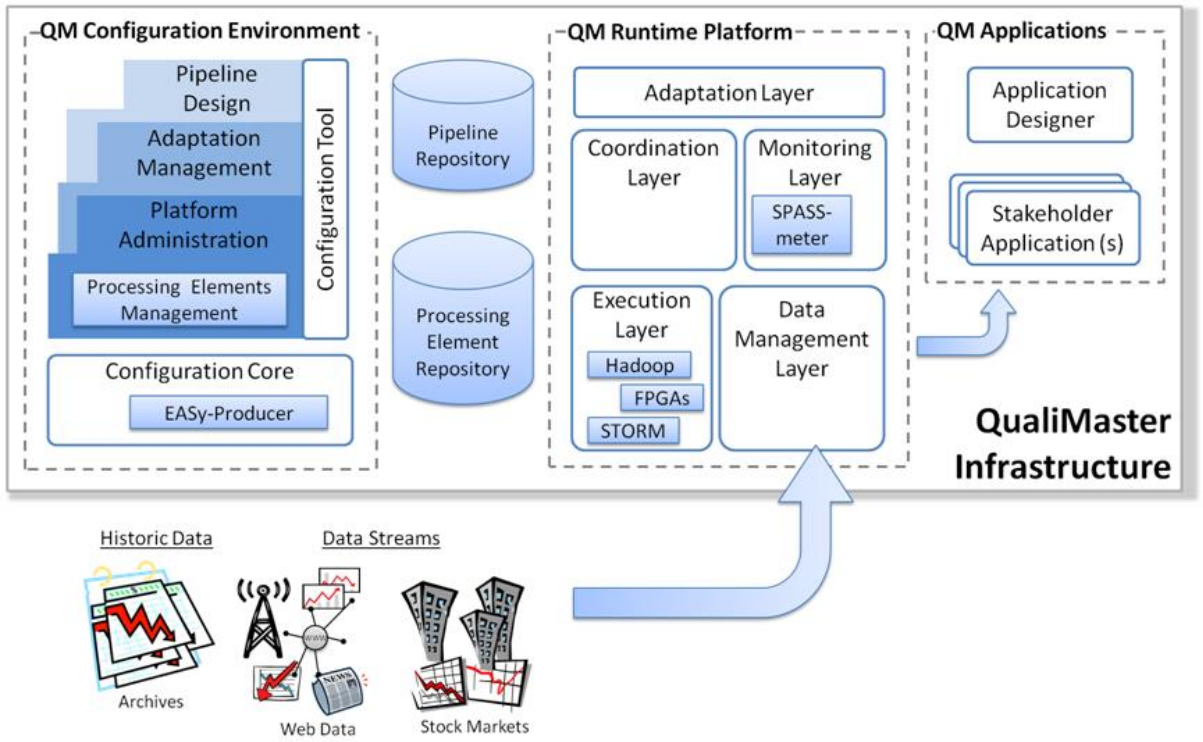

Figure 1: An illustration of the QualiMaster infrastructure.

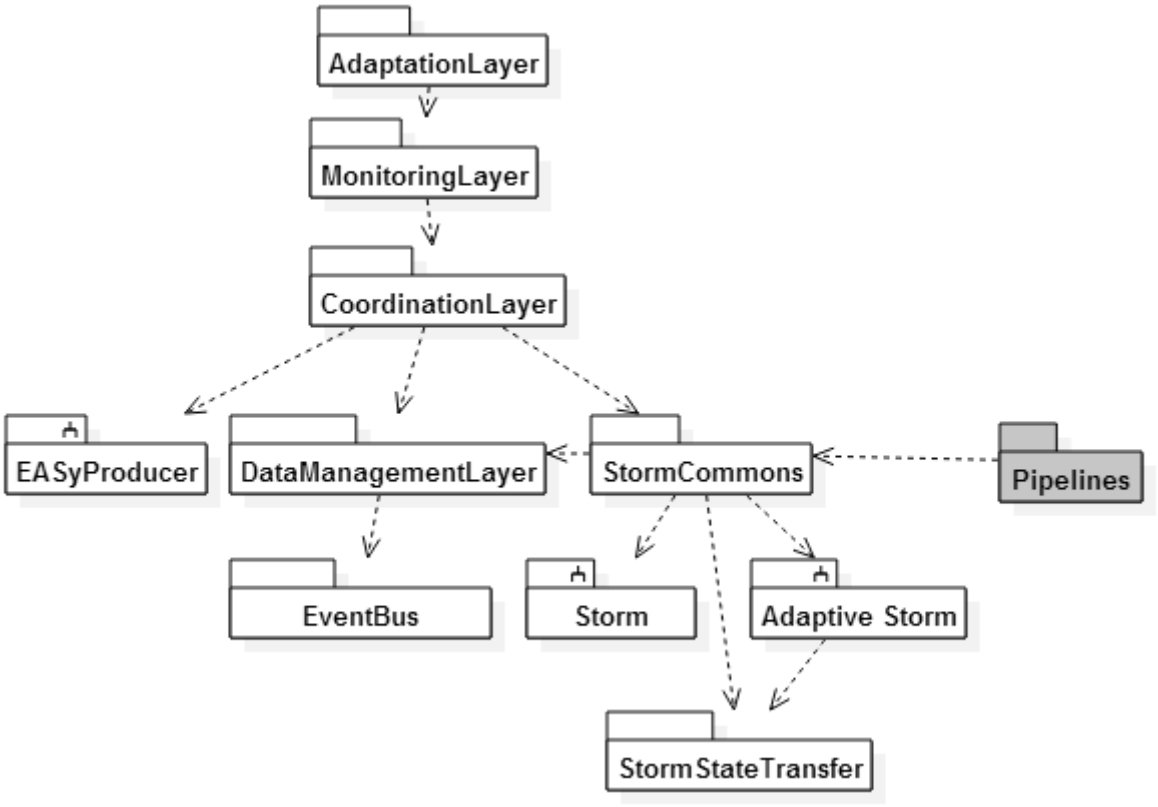

Figure 2: Overview over infrastructure components and their dependencies.

The pipeline lifecycle is handled entirely from the Coordination Layer. Therefore, its responsibility ranges from starting a pipeline to enacting adaptation decisions at runtime. Part of the Coordination Layer API is consisted of a set of executable commands that are used by the infrastructure user (e.g., to start or stop a pipeline) and the Adaptation Layer for enacting adaptation decisions.

The Monitoring Layer monitors the Execution Layer in order to obtain information related to the quality of the execution of the data processing. It is also responsible of controlling the actual monitoring process in terms of what to monitor, on which part and when, based on requests from the Adaptation Layer. Since, monitoring requires additional runtime operations, it implies a performance penalty which however can be controlled (e.g., by switching off monitoring probes). To enable quality constraints analysis, the Monitoring Layer must have knowledge of the Configuration Model of the running pipelines. This knowledge comes from the Coordination Layer during the startup process of a pipeline.

At the higher level of the QualiMaster infrastructure lies the Adaptation Layer. This layer is responsible for the decision making based on monitoring and analysis of the execution of pipelines. At runtime, the Adaptation Layer is activated only upon a trigger, e.g., an Service Level Agreements (SLA) constraint violation (issued by the Monitoring Layer), user triggers, adaptation schedule triggers and pipeline startup triggers (more triggers are described in the D4.1 deliverable). The reception of an AdaptationEvent causes the execution of the Adaptation Behavior Specification in terms of rt-VIL (see the D4.1 deliverable).

2.2 Explicit deployment of stable pipelines

In D5.3, we introduced the QualiMaster infrastructure instantiation process as part of the continuous integration. This process deploys automatically development releases of pipelines artifacts into the *Processing Element Repository*, actually realized as a Maven repository. This allows developing against and experiments with the most recent versions of the pipelines, but also implies that a running infrastructure always tries to obtain and run the most recent versions. While this form of continuous deployment is beneficial for the development of algorithms and pipelines, it can be an obstacle for running the QualiMaster infrastructure in a stable production environment.

To support a production environment setup, WP5 in collaboration with WP4 introduced a separation of development and production (stable) artifacts. The idea is that the development artifacts are still created and deployed automatically by the continuous integration (Development Processing Element Repository), while the stable artifacts are deployed into a separate Stable Processing Element Repository. For technically realizing the separation, there are two options:

- **Through the continuous integration:** A simple option would be to use the infrastructure instantiation process executed by the continuous integration not only for the deployment of snapshot releases for development, but also for the deployment of stable versions. For this, the infrastructure instantiation process needs to know what to build (development or stable, e.g., through temporary changes of the Configuration Model) and the user (here Algorithm Developer, Infrastructure Administrator, Pipeline Designer) must wait for the completion of

the whole build chain until the stable version of a pipeline is available in the Maven repository. While automatic updates are desired for snapshot releases, this is not the case for stable releases. Some repository managers such as Nexus even prevent a re-deployment of stable release. Thus, this procedure requires frequent changes of the Configuration Model to avoid that the stable releases will be overwritten with the next build.

- **Explicit deployment:** The user (here Infrastructure Administrator, Pipeline Designer) makes an explicit decision about the stability of a pipeline and whether it is ready for production. After running the infrastructure instantiation process with the QualiMaster Infrastructure Configuration tool (QM-IConf), the user can trigger an explicit deployment. There are three alternatives offering different usability:
  - The user can directly **execute the Maven command line tool**[1] within the local folder containing the pipeline created and packaged by QM-IConf. However, this requires deeper understanding about the handling of Maven, also as remote deployment requires further Maven plugins. Further, without an explicit mechanism on repository side, it is possible to accidentally overwrite already deployed pipelines.
  - The user can **upload the pipeline artifact** created by QM-IConf using a repository manager. To enable such deployments (also of algorithm artifacts and related dependencies that are not available through the official Maven repositories), WP5 set up Sonatype Nexus (as mentioned in the D5.2 deliverable) as a repository manager, which is available via a Web Browser. After authentication, the user can upload artifacts to both, the development and the stable repository. However, uploading an artifact (or multiple artifacts) is a rather technical process, which can also lead to accidental deployments.
  - The user (Infrastructure Administrator, Pipeline Designer) **triggers the stable deployment of a pipeline through QM-IConf**. This allows shielding the user from all technical obstacles, still leaving the explicit upload as a fallback. In particular, this works without deeper knowledge about Maven or the QualiMaster infrastructure.

[1] http://www.apache.org/dev/publishing-maven-artifacts.html

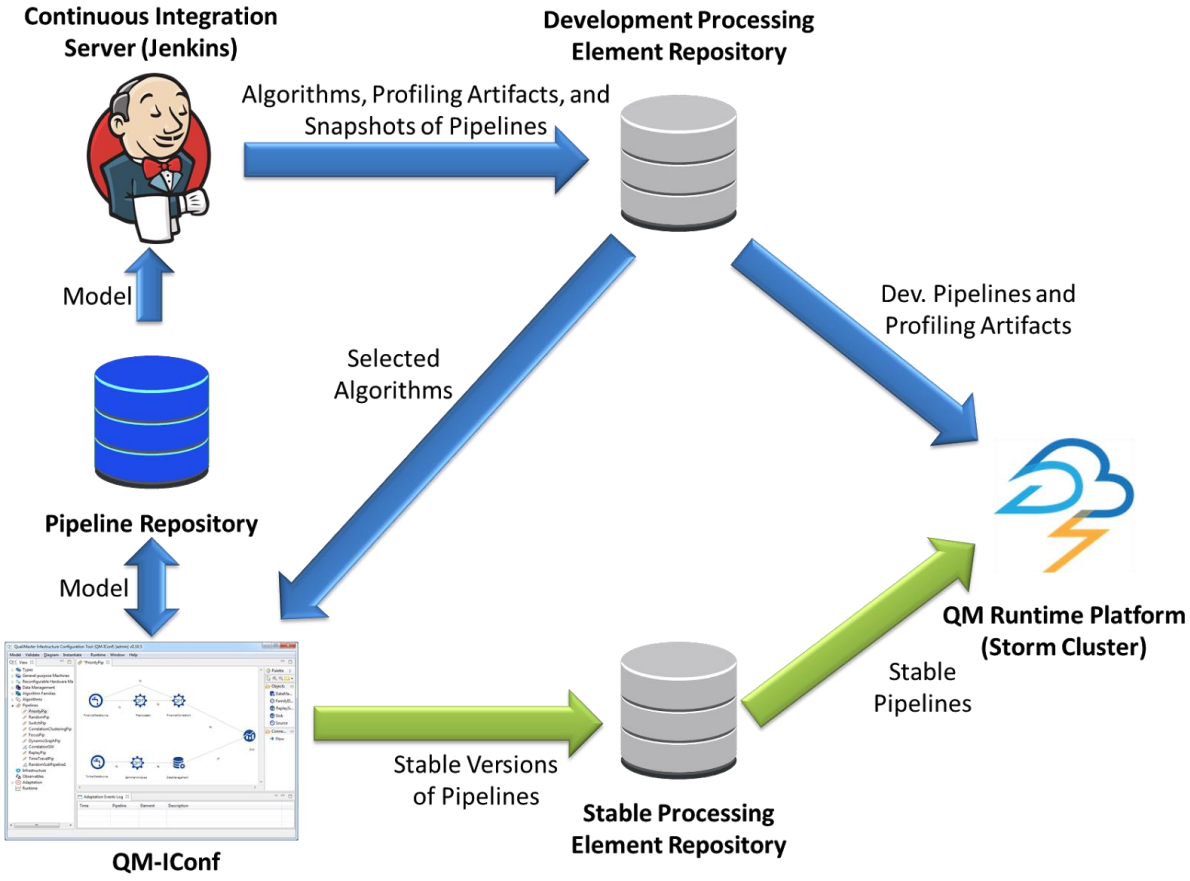

Figure 3: Infrastructure setup for deploying stable and development pipelines.

WP5 opted for the deployment of stable pipelines through QM-IConf. The introduction of the deployment functionality for the stable pipeline element repository extends the previous described infrastructure as illustrated in Figure 3. The green arrows indicate the new parts of the infrastructure: A Pipeline Designer uses QM-IConf to load the current version of the Configuration (Meta) Model from the *Pipeline Repository*, change it, and store the modified version back in the *Pipeline Repository*. The continuous integration server loads the latest version of the model for building and testing current implementation of pipeline elements. After a successful build and testing, the continuous integration server deploys the packaged pipeline artifacts to the *Development Processing Element Repository*. Moreover, the Pipeline Designer (or the Infrastructure Administrator) can use QM-IConf to deploy a pipeline as a stable version to the *Stable Processing Element Repository*. Finally, the QualiMaster Runtime Platform uses these two repositories for initialization and execution of pipelines. Depending on the setup, there may be different clusters with different settings (or just one cluster running both, development and stable pipelines). A cluster may use:

- Only the *Development Processing Element Repository* for retrieving required artifacts, in particular the configuration model, the pipelines and the initial profiling artifacts (see D4.3).
- Only the *Stable Processing Element Repository.* In this case, the (initial) profiling artifacts are not available to this cluster.

- Both repositories, *Stable Processing Element Repository* with precedence for obtaining (stable) pipelines and the *Development Processing Element Repository* as fallback, in particular for (initial) profiling artifacts or pipelines which are still in development.

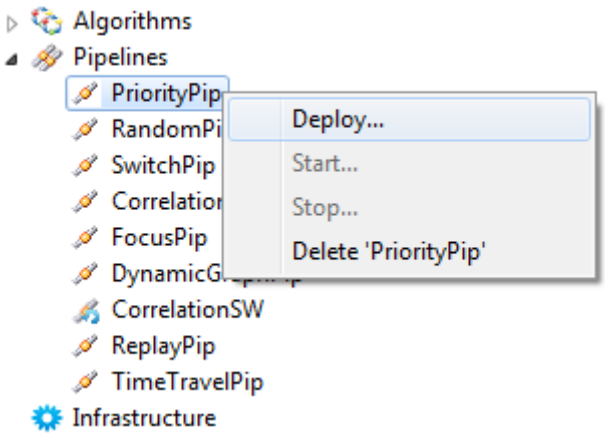

Figure 4: Explicit deployment of a stable pipeline through QM-IConf.

The realization of the concepts required some changes at the existing infrastructure and the QM-IConf application:

- **Changes to the Nexus / Maven repository server**. First, a new repository has been set up on the server, which is used as *Stable Processing Element Repository*. For this repository, we disabled the option of manual deployment via the Maven tool to avoid accidental changes as mentioned above. Initially, the plan was to use existing libraries for the implementation and to deploy directly against Sonatype Nexus. However, the libraries available to us did not serve for this purpose. As solution, we installed an FTP server to facilitate deployment through QM-IConf. This FTP connection is secured through SSL and accessible for registered users of QM-IConf.
- **Changes to the QualiMaster infrastructure.** We extended the infrastructure so that it can work with both, the primary and the optional fallback repository as explained above. The base URLs for the repositories can be defined along with various further options in the setup file of the infrastructure.
- **Changes to QM-IConf.** The application offers the deployment only for already instantiated pipelines. The tool analyzes the generated `pom.xml` file of an instantiated pipeline and locates files relevant for the upload. The Configuration Model provides all necessary information to deploy the selected pipeline at the correct position within the repository: The `deploymentURL` specifies the repository and `artifact` specifies repository path, file name, and version of the pipeline to be deployed. From the user's perspective, deploying a pipeline becomes a command of QM-IConf as illustrated in (cf. Figure 4).

2.3 Dynamic Parallelization of Storm Components

For dynamically adapting the resource usage of pipelines, the D4.1 deliverable introduced the enactment patterns for parallelizing (EP-5) and migrating (EP-6) processing elements. In

deliverables D4.2 and D5.3, we discussed an initial implementation, which focuses on dynamic migration of Storm tasks as well as transferral of queued items. The ability of migrating Storm tasks at runtime allows to split / join executors, i.e., to turn a sequential task into an own thread as well as to migrate a task to a new worker node (including creation or destruction of the containing executor). While task migration as described in D4.2 serves the need of EP-5 and EP-6 for stateless Storm tasks, migration of stateful tasks, e.g., the financial correlation computation can lead to imprecise results stream as the state must be re-created from the incoming data after the migration. For the final consolidated version of the QualiMaster infrastructure we introduce a state transfer mechanism for Storm tasks (following the Direct State Transfer strategy discussed in the D4.2 deliverable) and integrate it with the task migration.

In general, state transfer of software components is a difficult problem, in particular if the actual memory state shall be transferred at runtime among different types of components. In our case, state transfer shall be part of task migration, i.e., the state of a running task *t* of class *A* shall be transferred to a new instance *t'* (on the same machine, another executor or worker) of class *A*. In more details, task migration with state transfer happens through the following steps:

1. **Creation of task *t'* and its hosting executor if needed.**
2. **Capture memory state of *t*:** Before transferring the queued unprocessed items of *t* to *t'*, we capture the memory state of *t* (through Java object or kryo serialization) and send the state as a Storm task message to *t'* (i.e., the state is either stored locally if *t'* will run in the same worker or sent to a remote worker that will host the executor of *t')*.
3. **Integrate memory state into t':** As soon as the state arrives in the executor of t', all serializable (non-transient) fields that are part of the captured state are integrated into the memory state of t'. We will detail this below.
4. **Transfer queued unprocessed items to task *t'*** and store them in the input queue of *t'* before further items for *t'* arrive. Please note that this step runs in parallel with step 3.
5. **Terminate task *t*.** As we clear the input queue of *t*, the state captured in step 2 may miss the processing results of a few data items. As an alternative, we could transfer the state of *t* during its termination, but this means that the state of *t'* created through processing data items must be integrated with state *t*. As the implementation of the integration of the memory state of a task supports both alternatives, we consider an in-depth evaluation of both alternatives as part of the second evaluation phase of the project.

In comparison to the initial approach discussed in the D4.2 deliverable, we just added steps 2 and 3, i.e., capturing and sending the memory state as well as receiving and integrating the state into the target task. For capturing and integrating the memory state, we consider two strategies:

1. **Rely on the default mechanisms.** As mentioned above, for capturing the memory state we rely on the serialization mechanism of Storm, i.e., either explicit serialization via kryo if implemented for specific objects or default java serialization. In either case, the Algorithm Provider can influence the information that becomes part of the state as both ways allow to suppress selected attributes from the serialization, either by explicitly not serializing them (kryo) or by marking them as transient in the source code. For integrating such a state, we currently follow the rule that an attribute *a'* of *t'* is only modified if the state of *t* provides data for *a* and *a'* is not already set. This allows preserving an already created state in *t'*. In case that the serialized state is available before processing the first item in *t'*, all attributes which

are not initialized by default are filled with the corresponding information from the state of *t* (if not excluded from serialization).

2. **Use annotations to take control over capturing and integrating states.** If the default mechanism is not adequate, the Algorithm Provider can use annotations and even own mechanisms to control the state transfer. One annotation on class level marks a class for annotation-controlled state transfer and indicates whether all serializable (non-transient) attributes or just annotated attributes shall be part of the state. A second annotation on attribute level allows explicitly marking the attributes that shall be part of the state. The annotation on attribute level allows detailing (for collections) on how to integrate the state, e.g., clear the attribute in *t'* before adding the values from *t*, just add all values from *t* to *t'* or add all values from *t* that are not already in *t'*.

In case that a specific attribute type (even for a specific task type) shall be handled differently, the Algorithm Provide can register state handlers and provide an explicit implementation. For realizing and integrating the state transfer for Storm tasks, we extended our modifications of Storm (discussed in D5.3) based on a small, extensible framework for state transfer.

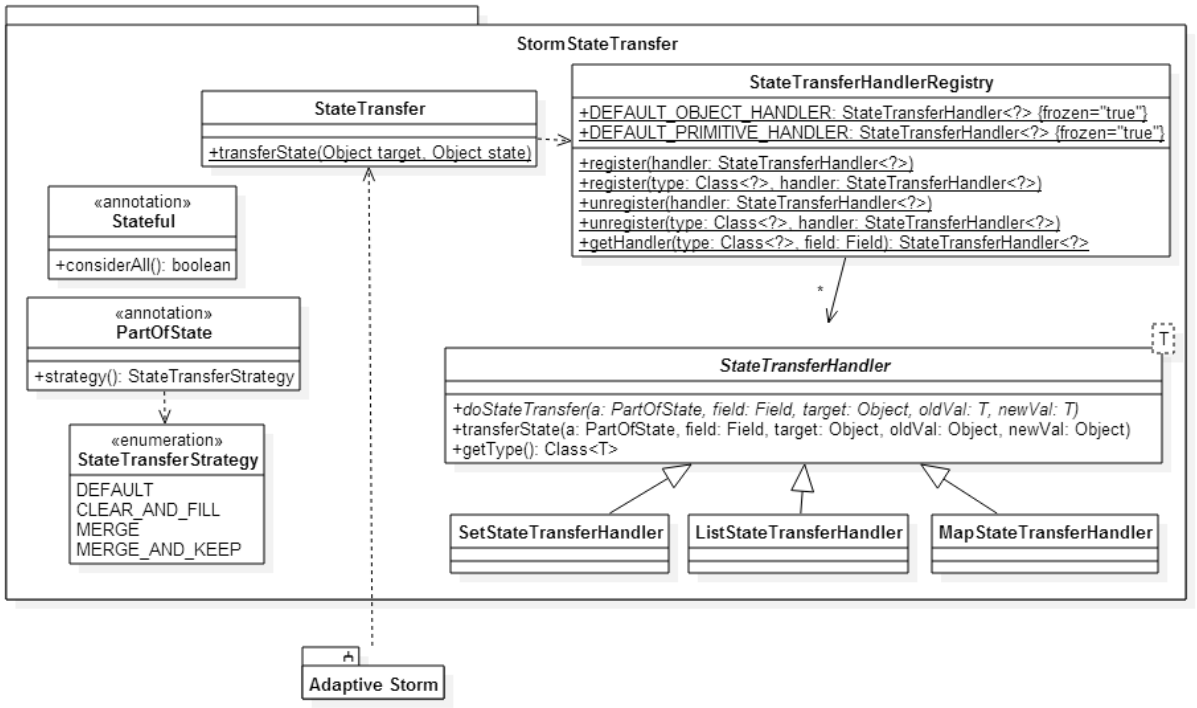

Figure 5: Overview on the state transfer framework for Storm tasks.

Figure 5 illustrates the design of the framework in terms of a class diagram. The core class is `StateTransfer`, which allows transferring the state of one (already deserialized) object into another object of the same type. During this reflection-based process, `StateTransfer` relies on the `StateTransferHandlerRegistry`, i.e., for each attribute to be transferred, `StateTransfer` asks the registry for the responsible `StateTransferHandler`. A `StateTransferHandler` is parameterized by the type it is handling, so that the specific implementation can directly work on that type. In the default case, one of the default handlers

defined in `StateTransferHandlerRegistry` are returned, in more specific cases, either a global handler (as the three specific handlers for sets, lists and maps) or a handler registered by the running pipelines is returned. As discussed above, the two annotations `Stateful` and `PartOfState` also allow the Algorithm Provider to influence the state integration.

As mentioned in the introduction to Section 2, the overall package `StormStateTransfer` is used only by the adaptive version of Storm developed by the QualiMaster project. At a glance, we might have integrated the state transfer framework directly into the code base of Adaptive Storm. However, in certain situations, e.g., for evaluation, it can be required that the original Storm version is running. In this case, pipeline code which uses the annotations would be invalid as the annotations would not be available. To support both cases, the state transfer framework became an own component, which is always present in the QualiMaster infrastructure, but which is used only if the adaptive version of Storm is running.

2.4 Load shedding framework

Load shedding [BTÖ13, CJ09], i.e., throwing away data, is the last resort to protect the QualiMaster infrastructure from overload (enactment pattern EP-3 from D4.1). In D4.3, we already discussed the integration of load shedding into configuration and adaptation. There, load shedding was represented as runtime configuration variables, which capture the name of the shedder and its parameters (similar to algorithm parameters). In this section, we detail the technical side, i.e., the underlying load shedding framework, which is controlled by the configuration and the enactment of adaptive decisions.

As discussed in D4.1, load shedding can be static, i.e., defined once based on a static analysis of the data or dynamic, i.e., controlled by a parameter. As mentioned above, we basically opt for dynamic load shedding, but also aim at fixed load shedding mechanisms without parameters that realize static load shedding. Dropping items is the simplest approach to load shedding, e.g., to discard items in a random fashion or by counting and discarding the *n*-th item. Filtering can be used to realize semantic load shedding, but of course, semantic load shedding is more resource-consuming due to additional effort of analyzing data items.

While the use cases of QualiMaster aim at financial data analysis, the QualiMaster infrastructure shall be domain independent. Thus, we cannot just determine some fixed mechanisms for load shedding, e.g., just discarding the *n*-th item, while in some specific application settings domain-specific filtering might be more appropriate. For this reason, we developed a simple, extensible load shedding framework on top of the data processor used in QualiMaster, namely Apache Storm. As already indicated in D4.3, the idea is that multiple load shedders may exist and can be identified via a unique name. A load shedder may have parameters (dynamic) that can be set from outside, e.g., via the adaptation layer or not. In contrast to typical load shedding approaches such as [CJ09], we do not focus on the data sources as the only location where load shedding can happen. This is due to the fact that in the financial use cases of QualiMaster, an extreme amount of intermediary or result data is created, e.g., through computation of correlation matrices, and load shedding may be required on these or subsequent data processing components as well.

Figure 6 provides an overview of the load shedding framework. The abstract class `LoadShedder` represents the basic implementation of a load shedder. A load shedder is described by an `ILoadShedderDescriptor` and can accept parameters denoted by the type `ILoadSheddingParameter`. Two specific shedders are pre-defined, namely `NoShedder`, the default instance which does not perform any load shedding and the `NthItemShedder`, which throws away the n-th tuple (based on the respective parameters). Instances of a load shedder are created through the `LoadShedderFactory` through the unique name of the shedder or its descriptor.

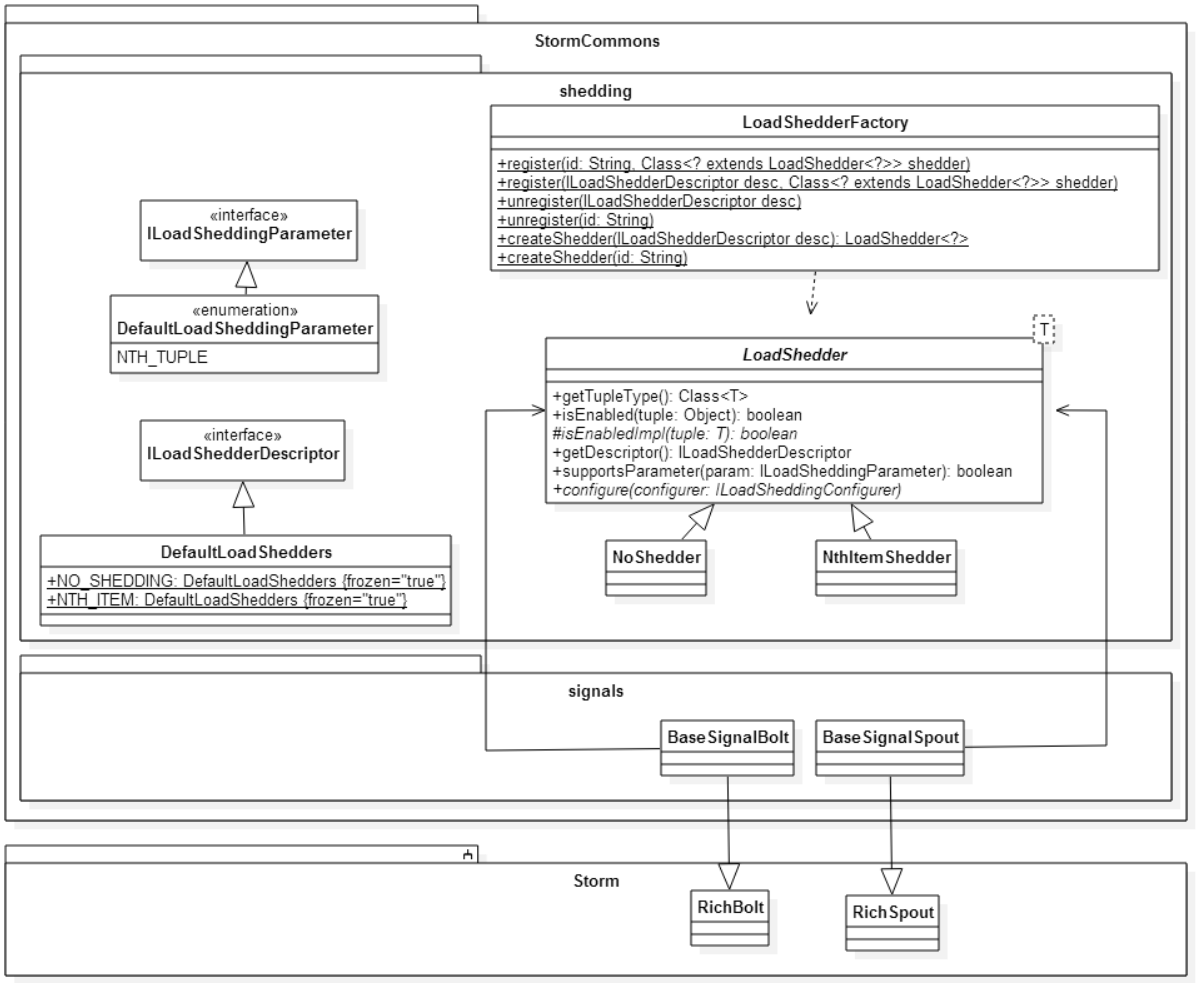

Figure 6: Design of the load shedding framework in StormCommons.

The load shedding framework is extensible, i.e., domain-specific components or even pipelines can define specific shedders and register them with the load shedder factory. A `LoadShedder` is parameterized by the tuple type to simplify the implementation of the central isEnabled method, i.e., for domain-specific shedding, the shedder implementation can be directly defined on the tuple type (created by the infrastructure instantiation). To apply these shedders, the adaptation script of the infrastructure must be adjusted so that the respective name of the shedder and adequate parameters can be passed.

The load shedding framework is integrated into the `StormCommons` component of the QualiMaster infrastructure, i.e., the component which realizes generic basic classes and mechanisms for the pipelines. For receiving and processing signals representing adaptive decisions, the `StormCommons` component contains since its very first version two refined base classes for Storm components, namely `BaseSignalBolt` and `BaseSignalSpout`. As these two classes are used as basis for all QualiMaster pipelines, we integrated the load shedding into these classes, i.e., provided basic implementations for the methods injecting a data item (`BaseSignalSpout`) or processing a data item (`BaseSignalBolt`) and provided new internal methods for the actual injection / processing of data. In this way, each data item can be discarded before it is injected / processed.

2.5 Replay mechanism

The Replay Mechanism allows the QualiMaster infrastructure to reconstruct part of historical data and "replay" them by feeding the results back into the stream, as if they come from the normal pipeline. This proves to be particularly helpful for stakeholder applications with retrospective analysis requirements. This section describes the architecture of the Replay Mechanism component and how it is structured and used to reconstruct and redirect the data in the pipeline. In the high level, it achieves this by regularly flushing the data into a materialized data store (`ReplayRecorder`), which supports timestamped data querying. The data will get fetched on demand later (`ReplayStreamer`) based on stakeholder application. Bellow, we present in detail these two work flows.

Figure 7 outlines the architecture of the Replay Mechanism component in the form of class diagram. The Replay Mechanism performs within the Data Management layer, acts as interceptor between the conventional data output (sinks) in different pipelines and the underlying data store. Specifically, the Replay Mechanism consists of the two major classes:

- **ReplayRecorder:** This class is responsible for transferring the data from the sink into the historical data store. Basically, the historical data store archives the data stream together with the timestamp, allowing for retrospective analysis from the stakeholder application. In QualiMaster, we decided to use HBase to store the historical data (see below for more detail). An instance of the `ReplayRecorder` writes out a data item via the trigger of the method `store(T)`.

  There are three things that make `ReplayRecorder` adapt to the QualiMaster infrastructure. First, each instance of `ReplayRecorder` is registered with a schema of the result (Tuple class). This Tuple schema specifies the attributes of each data item T, with at least one attribute for the key, and exactly one attribute for the timestamp. These key and timestamp attributes will be used to provide the scanner of results in the data store (`HBase`), which are needed in query time. Second, the `ReplayRecorder` is written into the Data Store using the QualiMaster serialization framework, via the `ISerializer` instance. This instance, which is generated automatically, supports the efficient serialization into the data store. Third, the data access interceptor `ReplayDataOutput` implements an underlying buffer into which the `store()` method writes the result into. This is particularly

helpful, because the `ReplayRecorder` essentially performs similarly as the "log" that materializes the real-time results and has considerable I/O cost.

- **ReplayStreamer:** This class is responsible as a data fetcher: Upon real-time requests from the sink, it constructs or updates the query and fetches data from the HBase, delivers them back to the sink as if they come from a normal stream source. It accepts two information: The query string and the time range (`start,end`). The syntax of the query string is specified based on the design contract between each pipeline and the replay recorder, which is inferred through the Tuple schema. The time range is used as a filter in HBase historical data store to get the desired data for replay.
  The `ReplayStreamer` has a major method - getData(), which is triggered from an interator in the sink, so data coming from the historical data store acts as normal items (i.e., the same as ones from a pipeline stream). Also, at runtime, the ReplayStreamer can update its query and time range via setter methods (`setData, setEnd, setQuery`). Upon the invocation of any of these methods, the `ReplayStreamer` will update its query process, and returns the new data items. The `setSpeed`() method is called to control the emission speed of the data items (slower or faster) by adding some delay within the `getData`() method. This helps the stakeholder application to explore the historical data at different granularity at will.

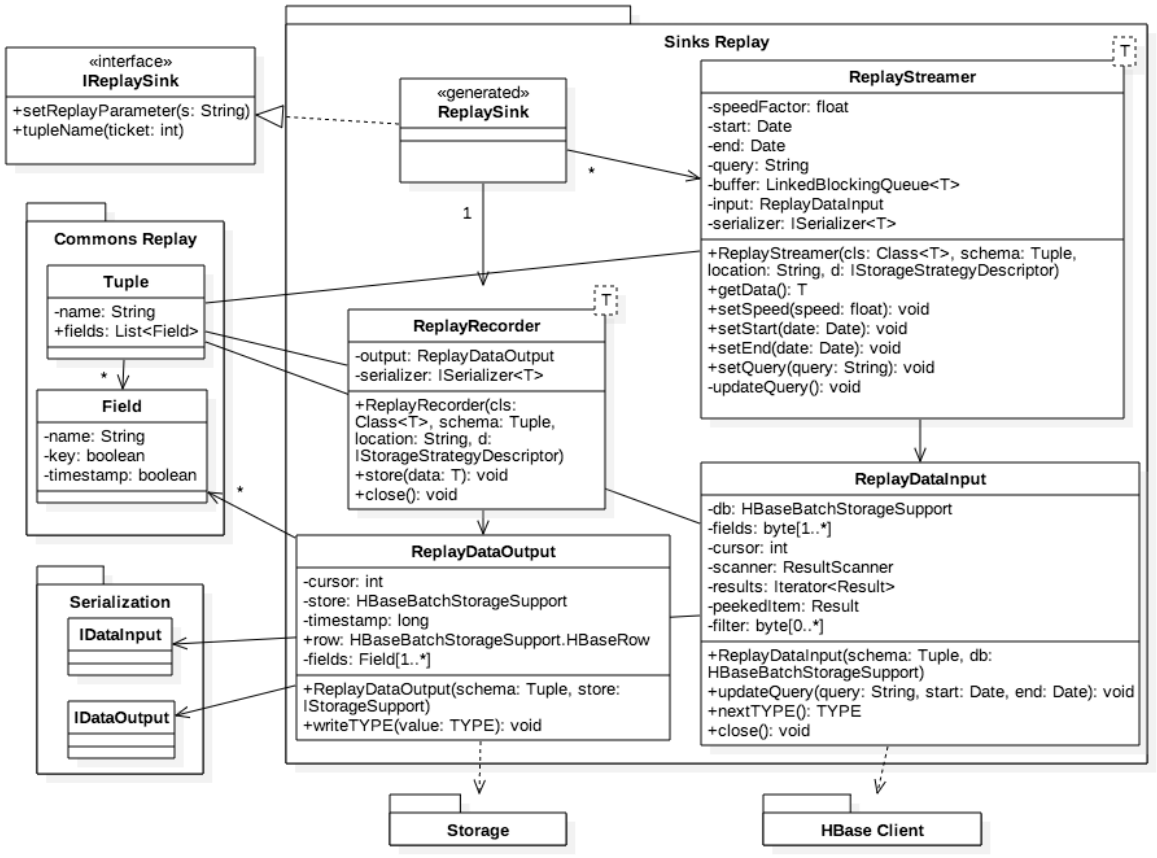

Figure 7: Class diagram for the Replay Mechanism component.

Data flow: The data flow of the two scenarios in the Replay Mechanism is illustrated below (Figure 8). Each "replayable" sink will be registered to one `ReplayRecorder` instance, and to several `ReplayStreamer` instances, each one serving an individual request from a specific end-user application. While the items of all `ReplayStreamer` instances are fed into the same pipeline (so as to simulate the stream behaviour), the replay sink distinguishes which item corresponds to each client request via a *ticket Id*. It attaches to each item the ticket id associated with its replay streamer, and forwards it back to the client through a standard sink (e.g., "TSI Sink").

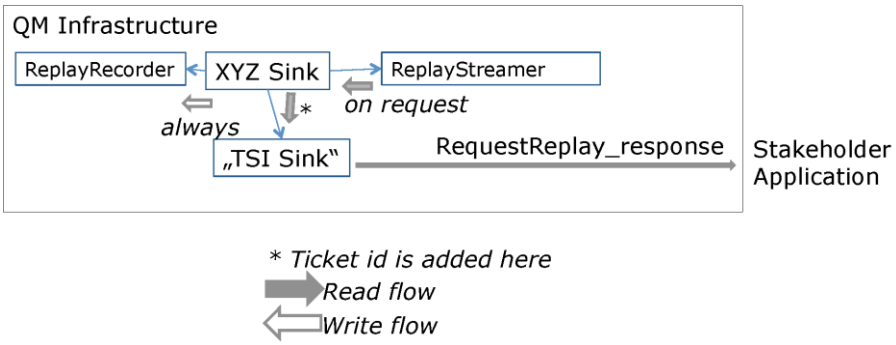

Figure 8: Data flow in the dynamic architecture.

To handle the high load of data fetching while allowing for real-time update of the query and speed, the `ReplayStreamer` implements an asynchronous buffer with its data accessor - `ReplayDataInput`. Essentially, the `ReplayDataInput` is responsible for connecting to the historical data store and writing out results into a buffer. The `getData`() method reads in asynchronous manner elements from this buffer, so that `updateQuery`() and `getData`() methods can be called in separate processes.

2.6 Switch toolbox

In QualiMaster, switching among alternative algorithms is one of key adaptation enactments. Basically, the algorithm switch is involved in the enactment pattern of selecting an algorithm from a processing family (EP-2) and switching between software- and hardware-based processing (EP-4) detailed in the D4.1 deliverable. As indicated in the D4.3 deliverable, we aim at performing a "safe" algorithm switching which takes into account the characteristics of alternative algorithms to achieve our goal of the transparency for the switch-related enactments. To support further experiments, we decided to design a switch toolbox framework with the aim of easing plain generation of the switch code and increasing the flexibility for integrating further switch techniques. In this section, we give an overview on our initial design of the switch toolbox framework.

Figure 9 illustrates the initial design of the switch toolbox framework. A `BaseSwitchSpout` extending the `BaseSignalSpout` is introduced to represent the abstract implementation for the intermediary Spout used in the switch mechanism. The purpose of the `BaseSwitchSpout` is to ease the Storm-related integration of the algorithm switch from the generation of the sub-pipeline algorithms (see loosely integrated sub-pipelines in Section 2.8). This allows us to adjust the implementation of the switch directly in `StormCommons` avoiding the heavy modification on the

pipeline generation side. We also support the switch toolbox to apply different switch mechanisms with combination of different switch strategies. The abstract class `AbstractSwitchMechanism` represents the basic implementation of the switch mechanism. A specific switch mechanism, e.g., the parallel track mechanism `ParallelTrackSwitchMechanism` used in our current experiment, is pointed to the `BaseSwitchSpout`. This can be also easily extended to use another type of the switch mechanism. To support the flexibility of using different switch strategies for a specific switch mechanism, we introduce the `AbstractSwitchStrategy` as the base implementation of the switch strategy.

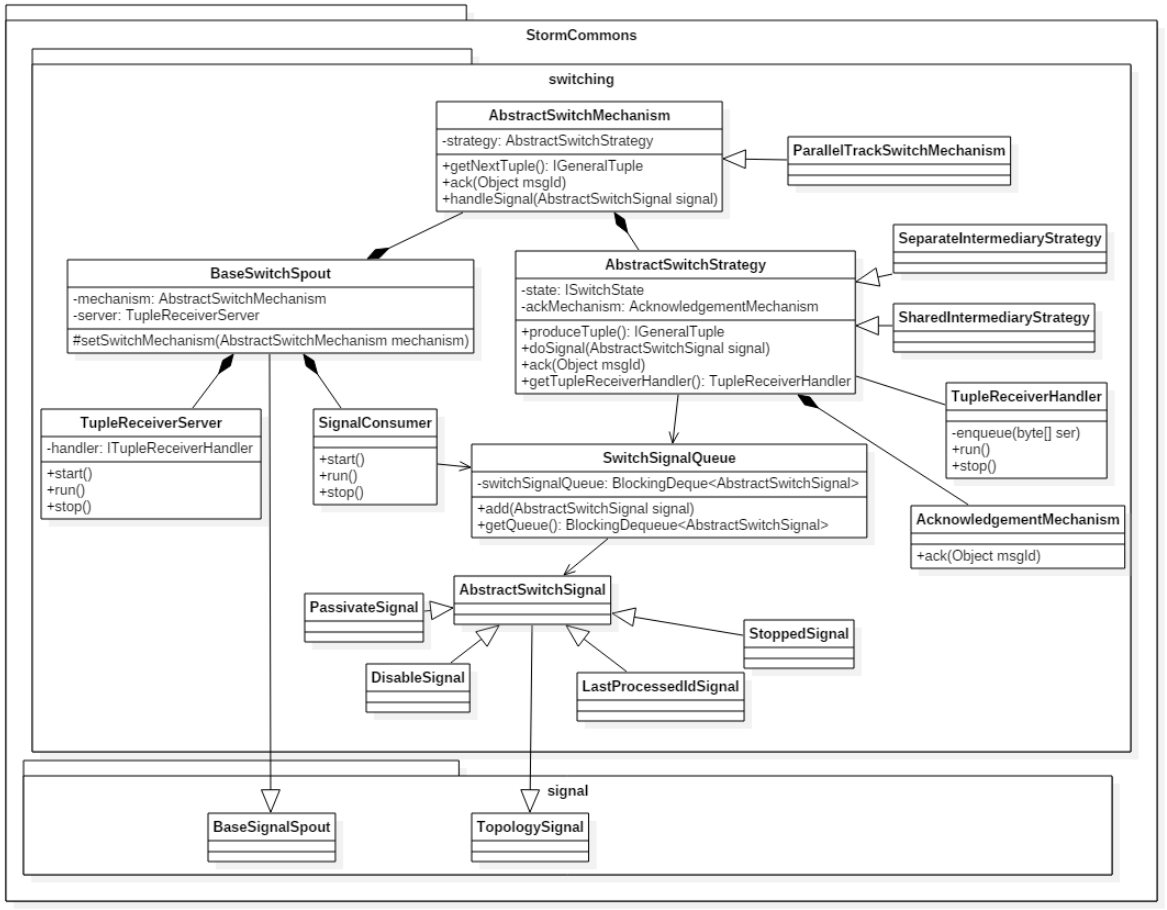

Figure 9: Initial design of the switch toolbox framework.

The examples of different strategies are the `SeparateIntermediaryStrategy` using separate intermediary Spout for each algorithm (used in our current experiment) and the `SharedIntermediaryStrategy` using a shared intermediary Spout for both alternative algorithms involved in the switch. We also support different data acknowledgement mechanism by the `AcknowledgementMechanism` to specify the reaction on the arrival of the acknowledgement of fully processed data items. The signal communication (see D5.3 Section 2.4) is one of the most important parts for realizing a switch mechanism. In this design, we support the pre-defined communication signals by implementing the `AbstractSwitchSignal`, e.g., `PassivateSignal` used to represent the “passivate” signal in the switch. Basically, the received signals are handled

along with the switch strategies. According to the received signal, we change the switch state defined by the `ISwitchState` to invoke the specific reaction during the switch. Sometimes, we even need to conduct signals for further communication. For this, we apply a `SwitchSignalQueue` to store the received signals. Meanwhile, in the `BaseSwitchSpout` we have a `SignalConsumer`, running in the background to iteratively send the signals queued in the `SwitchSignalQueue`.

As mentioned in D5.3, our advanced switch mechanism relies on the network connection for transferring data items to the intermediary Spout of the algorithm. Such network connection is also introduced in the loose pipeline integration discussed in Section 2.8. Previously, we generated the network-based integration in each intermediary Spout. To ease this network-based integration from the generation of the algorithm, we equip the `BaseSwitchSpout` with a `TupleReceiverServer` responsible for deserializing the received data and pushing them to the right queue. The `TupleReceiverServer` is basically specified with a specific `TupleReceiverHandler` defined along with the switch strategy.

2.7 Infrastructure Monitoring

The Monitoring Layer of the QualiMaster infrastructure is responsible for collecting and aggregating runtime information from various sources into a joint monitoring model. The sources represent monitoring probes corresponding to the QualiMaster quality taxonomy defined in deliverable D4.1 and refined in deliverable D4.2. Monitoring probes can transmit the raw monitored information via the Event Bus to the Monitoring Layer or the Monitoring Layer can actively poll information from the probes. Both forms are needed, as monitoring probes / sources are implemented by different parties, e.g., statistical information provided by Storm can be requested from a single interface (akin to monitoring information provided by the reconfigurable hardware), while generated pipelines actively transmit their observed information from various nodes in the cluster. As discussed in deliverable D4.3, the aggregated monitoring model is the basis for constraint analysis and adaptation (alarm) events used to indicate a potential situation where adaptation is needed to the Adaptation Layer.

In this section, we detail several parts of the Infrastructure Monitoring developed and integrated since the submission of the D5.3 deliverable. These parts are:

- Section 2.7.1: The extensible pipeline probe framework realizing UC-PA1 from D1.2.
- Section 2.7.2: The integration and improvement of the resource monitoring framework SPASS-meter.
- Section 2.7.3: The flexible online data integration and aggregation performed by the Monitoring Layer.
- Section 2.7.4: Algorithm profiling and prediction (introduced in the D4.3 deliverable).
- Section 2.7.5: Source volume prediction (introduced in the D4.3 deliverable).

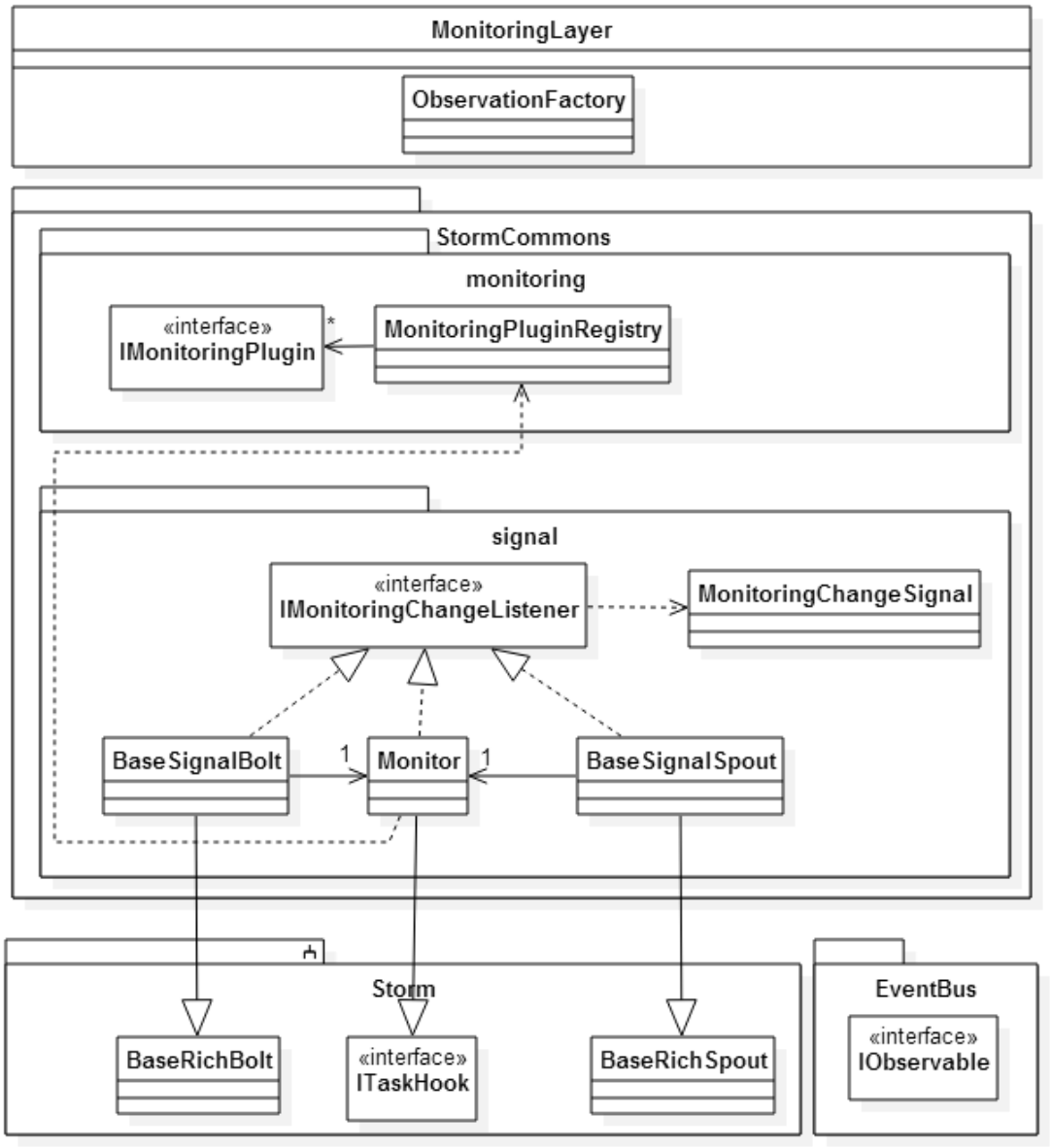

Figure 10: Design of the extensible pipeline probe framework.

2.7.1 Extensible pipeline probe framework

The QualiMaster infrastructure implements monitoring probes for the QualiMaster quality taxonomy from D4.1 / D4.2. However, application-specific monitoring may be needed in some situations. To cover these cases, UC-PA1 in the D2.2 deliverable requests that (pipeline) monitoring probes can be configured and integrated into the QualiMaster infrastructure. Although we might have also realized the default monitoring for the quality taxonomy in terms of such extensible monitoring probes, for performance reasons we decided to integrate the default probes directly into the (generated) code.

Figure 10 visualizes the design of the extensible pipeline probe framework. Pipeline monitoring happens through the task-hook mechanism of Storm, i.e., Storm defines the interface `ITaskHook` and calls registered instances of this interface upon processing and emitting data items. For pipeline monitoring we rely on this mechanism, i.e., we realized a specific task hook (called `Monitor`), which also takes care of the regular communication with the QualiMaster infrastructure, in particular the Monitoring Layer, and integrated the Monitor with the base spout and bolt classes for the QualiMaster pipelines in the `StormCommons` component. Raw monitoring results are collected in the `Monitor`, aggregated and reported as a mapping of observables (Figure 10 shows only the interface `IObservable`, but specific ones for the QualiMaster quality taxonomy are also defined) to observed values. As a result, each Storm component of a QualiMaster pipeline is equipped with default monitoring capabilities.

Moreover, `Monitor` delegates its calls to the `MonitoringPluginRegistry` to inform monitoring plugins (`IMonitoringPlugin`) shipped with pipeline code or other Maven components about relevant information, such as emitting a data item. In regular fashion, the `Monitor` requests the monitored state from the plugins and passes the information along with the observable-value mapping to the Monitoring Layer.

The following steps are needed to implement and use a monitoring plugin:

- Implement a specific observable by inheriting from `IObservable`.
- Register the observable, its aggregation and analysis procedures (for details see Section 2.7.3) with the Monitoring Layer (`ObservationFactory` in Figure 10). The analysis procedures can send adaptation events to the Adaptation Layer.
- Implement the monitoring plugin, i.e., `IMonitoringPlugin`, using the specific observable.
- Package the monitoring plugin into a Maven artifact and deploy it to an accessible repository, e.g., the QualiMaster pipeline element repositories (see Section 2.2 for stable and development repositories).
- Configure the monitoring plugin using QM-IConf. The infrastructure analyses this configuration part at runtime and loads deployed artifacts / registers the monitoring plugins with the `MonitoringPluginRegistry` automatically upon startup.

2.7.2 SPASS-meter performance analysis / improvement

As introduced in D5.1, SPASS-meter [ES14] is a flexible online resource monitoring framework for Java programs. It monitors the execution time, memory consumption, system load, file as well as network transfer of a system under monitoring (SUM), and provides access to the aggregated information at runtime. To obtain raw information on the resource consumption, it instruments the SUM with specific monitoring probes through byte code manipulation. Regarding resources, SPASS-meter can collect raw resource consumption information for:

- CPU time, i.e., the amount of time consumed by the CPU for executing a set of SUM statements.
- Response time, i.e., the amount of (wall) time that elapsed while executing a specific task, such as processing a specific data item by an algorithm.

- Memory allocation, i.e., the amount of memory requested by a SUM part.
- Memory use, the actual memory consumption as the difference between memory allocation and unallocated memory.
- File transfer in terms of the number of payload bytes read from or written to files (excluding administrative overhead).
- Network transfer as the number of payload bytes read from or written to network interfaces (excluding administrative and protocol overhead).

SPASS-meter performs online aggregation of the raw data on several levels (starting with the most fine grained one):

- Monitoring groups: A monitoring group is a user-defined composition of classes / methods and resources to be monitored. As the composition is user-defined, it allows to monitor individual methods, classes, algorithms, components or services.
- SUM: The SUM level characterizes the resources consumed by the entire SUM excluding the administrative overhead of the runtime environment, i.e., the SUM without the JVM that runs the SUM.
- Process: The process level represents the external view on the operating system resources consumed by the SUM including administrative overhead of the runtime environment.
- System: The system level represents the resource view of the entire system including all running operating system processes.

In QualiMaster, SPASS-meter serves for three purposes:

- Monitoring the resource consumption of individual algorithms. The monitoring configuration of SPASS-meter allows configuring the monitoring groups and what shall be measured for the higher aggregation levels. For QualiMaster, we configured SPASS-meter in a way that it measures the resource consumption of individual algorithms, i.e., the Storm Bolts and Spouts executing the algorithms. The consumption of the individual pipeline nodes is sent as events to the Monitoring Layer.
- Monitoring compute nodes on system level, in particular in absence of running pipelines. An optional service has been set up which sends the actual system level resource consumption to the Monitoring Layer in a regular fashion.
- Monitoring the data volume estimated in terms of data objects. We use libraries utilized by SPASS-meter to derive the memory size of data objects sent through pipelines.

While monitoring the resource consumption of individual algorithms in terms of monitoring groups, monitoring multiple pipelines on the QualiMaster infrastructure is challenging as the same component can be executed as part of several pipelines on the same node, even in the same JVM (Storm tasks). Moreover, SPASS-meter is a generic framework, which shall remain independent of the QualiMaster infrastructure and Apache Storm. For monitoring distributed execution environments such as Storm, we extended the aggregation hierarchy of SPASS-meter for cross-pipeline settings by an aggregation on instance level below monitoring groups. This requires that the instances can be identified in a generic way (SPASS-meter perspective) and in an application-specific way so that the resource consumption can be assigned to individual Storm tasks in the Monitoring Layer. For this purpose, we use the identity hashcode of Java. For linking the QualiMaster infrastructure with SPASS-meter, we allow the QualiMaster infrastructure to register

the distribution information such as host, port, task number and thread with the identity hashcode. The SPASS-meter integration for QualiMaster uses this registration to inform the QualiMaster monitoring layer about resource consumptions on instance level.

Since D5.3, we also worked on a performance optimization of SPASS-meter [Sass16], partly to evaluate and reproduce external results [Waller14], but also to analyze a memory problem with SPASS-meter on the Storm clusters of the project. Therefore, we applied the micro-benchmarking framework MooBench [Waller14], which measures the response time of an instrumenting monitoring framework for a simple recursive method of recursion depth 10 for 2.000.000 calls. This benchmark indicated several spikes in response time, which initially appeared to be statistical outliers (Figure 11). We analyzed several hypotheses and identified (by measuring memory consumption along with the response time in the micro-benchmarking) that garbage collection causes the spikes (and even prevented longer running pipelines on a Storm cluster). Actually, the memory consumption leading to the garbage collections was accidentally caused by a method of Java Management Extensions (JMX) as well as by cleaning up objects of the internal event storage of SPASS-meter. For solving this issue, we bypassed the JMX method by directly calling its native implementation and by applying object pooling to the internal event storage.

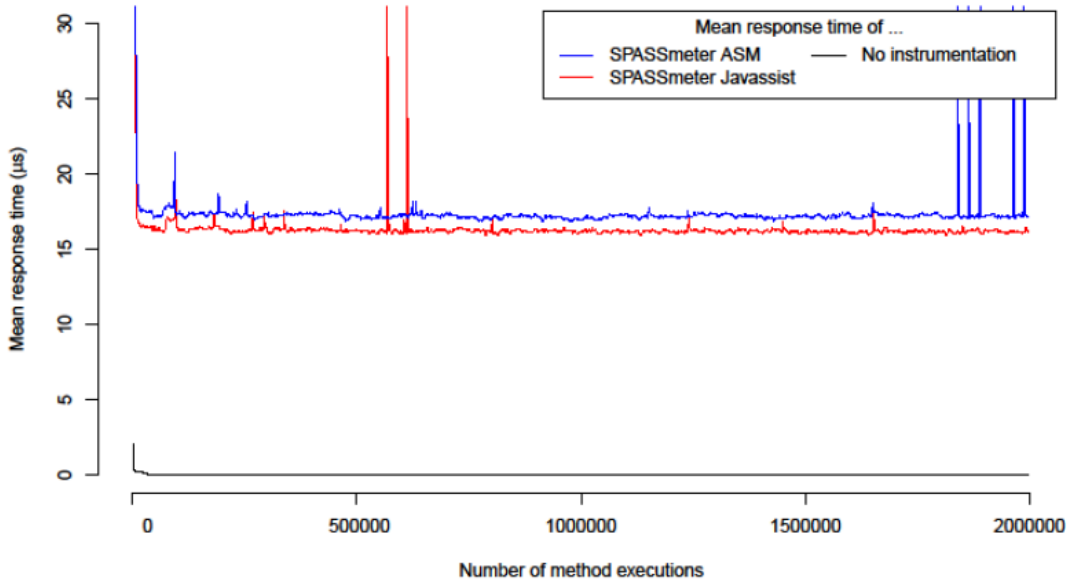

Figure 11: Initial results of benchmarking SPASS-meter with MooBench [Sass16].

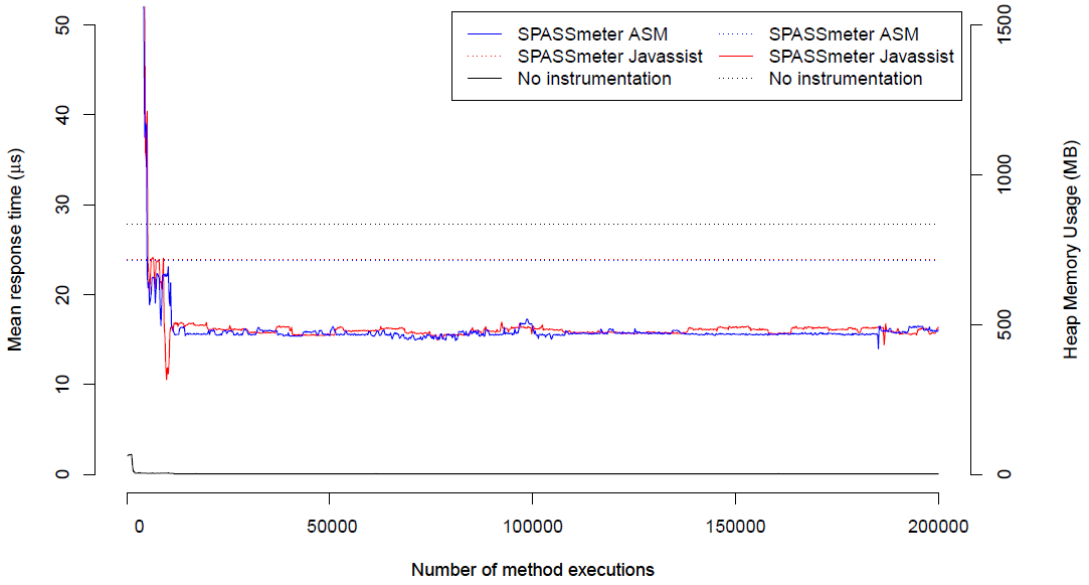

Figure 12: Benchmarking results after optimization. Dotted lines indicate memory consumption.

As a result, the memory consumption in the benchmarking is now constant (Figure 12) and the overall overhead of response time monitoring with SPASS-meter decreased by factor 2 (further factor 2 by using a more modern JVM than in the initial experiments in [ES14]). However, the initialization phase is longer and requires more resources, e.g., for setting up the internal resource pools.

In Section 5.4, we will discuss results of performance experiments with SPASS-meter, in particular regarding the extension for instance-based monitoring discussed above as well as results from running SPASS-meter on a Storm cluster.

2.7.3 Online data integration and aggregation

The data aggregation is at the heart of the Monitoring Layer. In deliverable D5.2, we introduced the basic structure of the Monitoring Layer, in particular the system state, which consists of system parts and observations with associated values. During the project, we evolved the Monitoring Layer to cope with the complexity of the infrastructure, to support:

- More monitoring data sources, e.g., the reconfigurable hardware, pipeline monitoring probes, the monitoring service for unused nodes based on SPASS-meter.
- Detailed monitoring probes as the pre-aggregated statistical information provided by Storm is not sufficient for adaptive control of task resources.
- Construction of aggregation topologies for pipeline measures such as latency or throughput.
- Extensible pipeline monitoring probes (Section 2.7.1).

In this section, we detail two novel aspects of the online data integration, namely the flexible design for data aggregation and the construction of the internal topology knowledge. Starting with D5.2, the design of the Monitoring Layer targeted flexibility rather than a fixed number / semantics

of observables. This was rather helpful when extending the quality taxonomy in D4.2, but also for realizing the extensible pipeline monitoring probes. Figure 13 provides an overview of the design for the data aggregation.

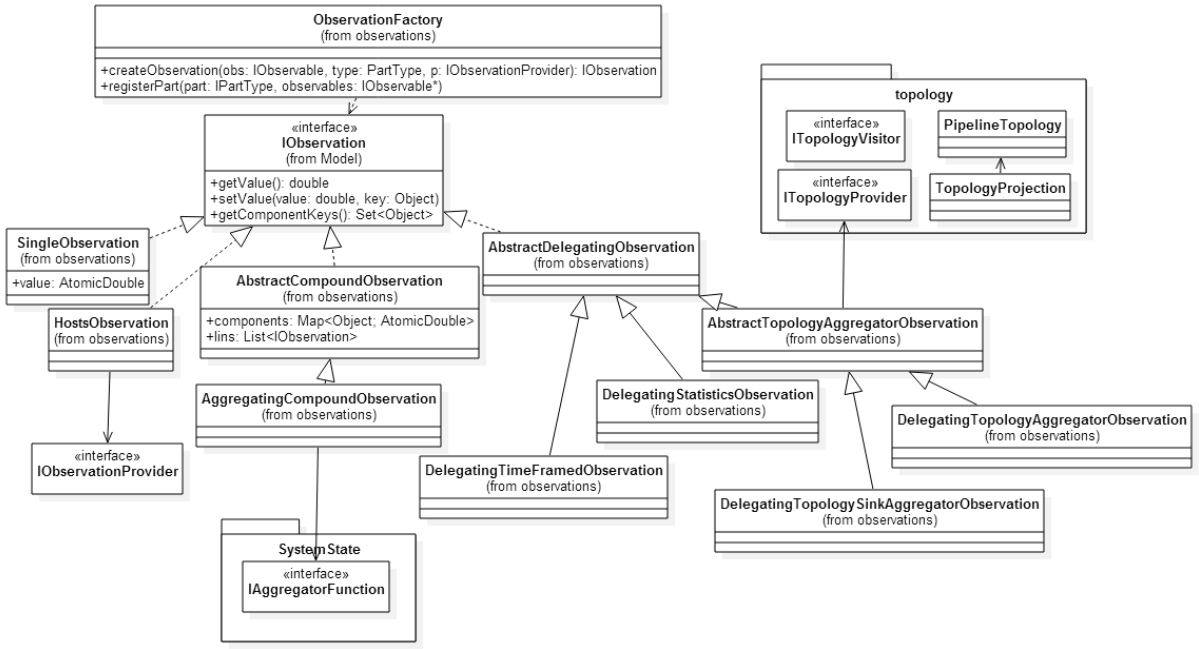

Figure 13: Design of the flexible aggregation in the Monitoring Layer.

The design builds on the initial design discussed in D5.2. There, we introduced the concept of the `SystemState`, a thread-safe structure for storing and aggregating raw monitoring data. The `SystemState` consists of several parts, e.g., pipelines, pipeline nodes, algorithms, the infrastructure (only pipeline and pipeline node are shown in Figure 13). Individual parts may consist of sub-parts, such as pipelines consist of contained nodes. All parts of the system state represent a mapping of observables (`IObservable`) to actual observations (`IObservation`). In the end, an `IObservation` return a single value for the `IObservable` they represent. Observations are of different kind:

- A single value, e.g., the number of Data Flow Engines provided by a reconfigurable hardware node (`SingleObservation`).
- A compound observation (`AbstractCompoundObservation`) representing a set of values to be aggregated. Typically, we use this kind of observation to maintain the individual values provided by different distributed tasks in a cluster, e.g., to represent the overall execution time of all tasks implementing a certain Storm component. This eases updating aggregations. The `AggregatingCompoundObservation` is a specialized compound observation, which aggregates the final value based on an aggregator function (`IAggregatorFunction`).
- A delegating observation use a single or compound observation as based and perform further aggregations on top to achieve a certain effect. Basically, the `DelegatingTimeFramedObservation` returns the average value for a certain time period, e.g., items per second and the `DelegatingStatisticsObservation` also

offers the minimum, maximum and average of the aggregated values. Both can be combined, e.g., statistics over values per time period. The topology observations are currently the most complex aggregators provided by the Monitoring Layer. They calculate their value by iterating over topologies representing (parts of) the running pipelines (we will detail the derivation of these internal topologies below). Examples are the latency of a pipeline, or of a sub-pipeline, calculated as the maximum latency of all alternative paths of the topology or the throughput as the sum of the emitted items (per second) of all sinks of a topology. The topology observations are generic and (indirectly) rely on the aggregator functions.

Observations are not created directly rather than via the `ObservationFactory`. Therefore, all pipeline parts, their supported observables, the related aggregator functions as well as the required observations are registered in this factory. More combinations can be registered if needed, e.g., as part of implementing extensible pipeline probes (Section 2.7.1). When creating a `SystemState` instance, the `ObservationFactory` provides access to the supported / configured observables and, finally, to the respective observations which transparently perform the data aggregation.

Upon receiving a monitoring event or polling raw monitoring data, the Monitoring Layer identifies the respective part *p* of the system state and passes the new value along with the respective observable to *p*. *p* identifies the observation based on the observable and passes the value to the observation, which takes the value into account when calculating its aggregated value. In case of compound observations, the Monitoring Layer passes a compound identifier (consisting of host name, port number, task identifier and, if available, thread number) along with the value so that individual values of distributed components can easily be updated.

As mentioned above, some of the observations perform aggregations over internal topology knowledge. Basically, the topology knowledge is taken from the actual processing topology as executed by Storm. However, some pipelines have “gaps”, at least from the perspective of Storm, in particular where the reconfigurable hardware or a sub-pipeline operate. As discussed in D3.2 and D3.3, hardware-based algorithms are integrated via network and do not occur as “Storm nodes”. Sub-pipelines represent alternative distributed algorithms. If alternative algorithms are integrated into the main topology, unused alternatives allocated resources without needing them. Sub-pipelines are also integrated by network connections and can be loaded and unloaded if needed, e.g., during warm-up while switching among algorithms. However, for performing aggregations over such “gaps”, the Monitoring Layer must compose the topology knowledge from both, the configuration, which knows the kinds of algorithms, and the Storm topology, which represents the implementation including additional components for switching, data storage etc.

We illustrate the creation of the topology knowledge by two examples, the integration of hardware algorithms and the network-based integration of sub-topologies (Section 2.8). Of course, both cases can be combined, e.g., in terms of the correlation family, which contains the distributed software-based and the hardware-based implementation of the Hayashi-Yoshida correlation estimator.

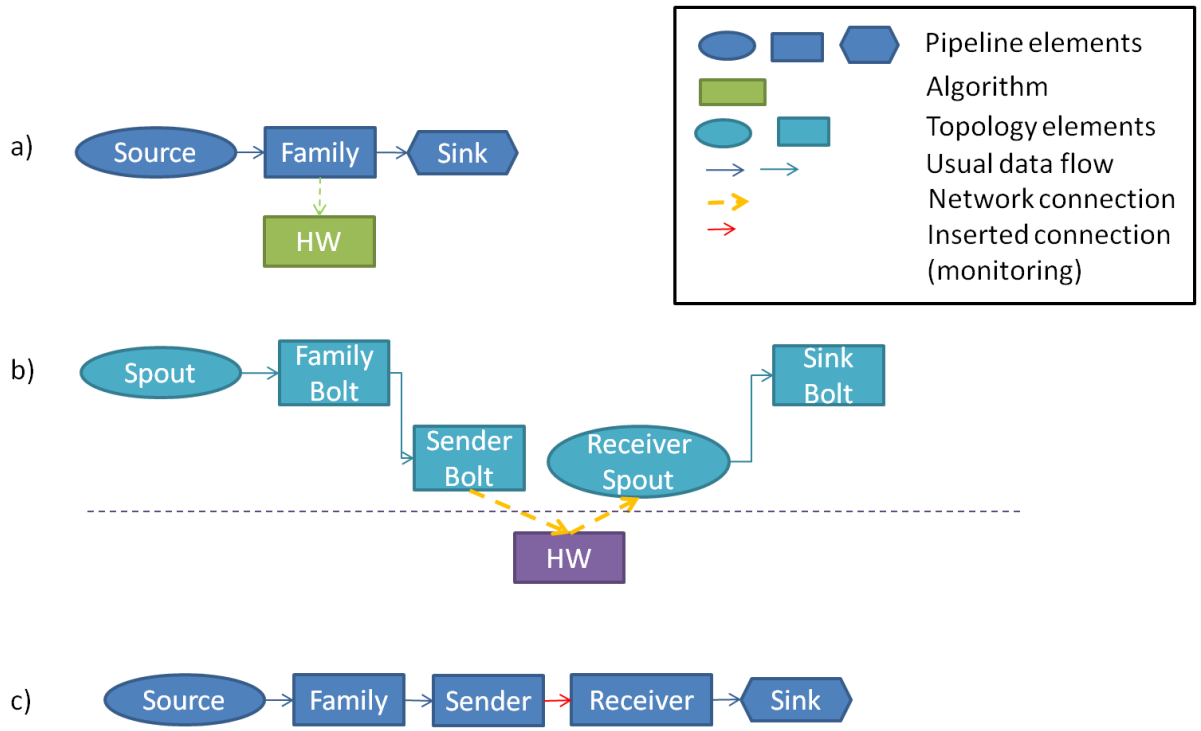

Figure 14: Constructing a Monitoring topology c) for the integration of reconfigurable hardware from the configuration a) and the Storm topology b).

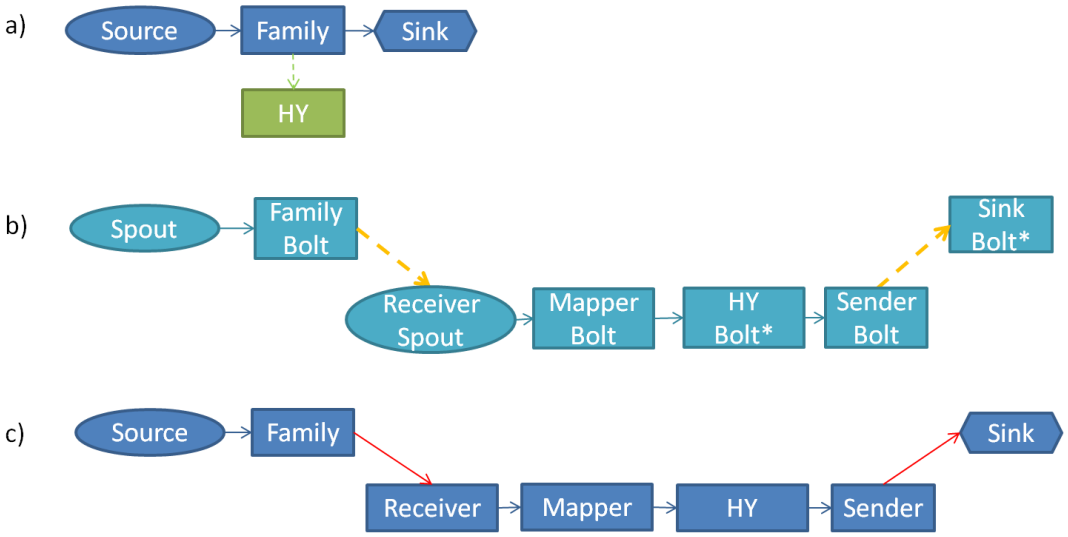

Figure 15: Constructing a Monitoring topology c) for the integration of sub-pipelines hardware from the configuration a) and the Storm topology b).

Figure 14 illustrates the creation of the monitoring topology involving a hardware-based as well as the “gap”. Figure 14 a) depicts the configured pipeline in terms of a data source, an algorithm family and a data sink. Due to technical reasons, the generated implementation in Figure 14 b) needs (conceptually) some more Storm components, in particular the sender and the receiver, which communicate with the reconfigurable hardware (see also D3.3). For Storm, there is no connection between sender and receiver component, as they communicate via network with the reconfigurable hardware. The Monitoring Layer constructs a complete topology from the configuration and the Storm view, by filling the “gap” with a pseudo connection, i.e., topology-based aggregations see the complete structure and can calculate results for the full pipeline (if requested).

Analogously, a distributed algorithm realized as a network-based sub-topology requires two pseudo connections in the monitoring topology. This is illustrated in Figure 15. While the configured topology in Figure 15a) is as simple as in Figure 14, the Storm topology would miss the implementation of the network-based sub-topology, i.e., the Receiver Spout, the Mapper Bolt, the HY bolt and the Sender Bolt. Here, the Monitoring Layer inserts two pseudo connections into its internal topology depicted in Figure 15c) and enables correct aggregations.

In the Priority Pipeline, both cases occur in the same family, i.e., the hardware-based and software-based correlation computation. Moreover, if both algorithms shall be integrated in a network-based fashion, the final topology requires 5 pseudo connections as Figure 16 illustrates.

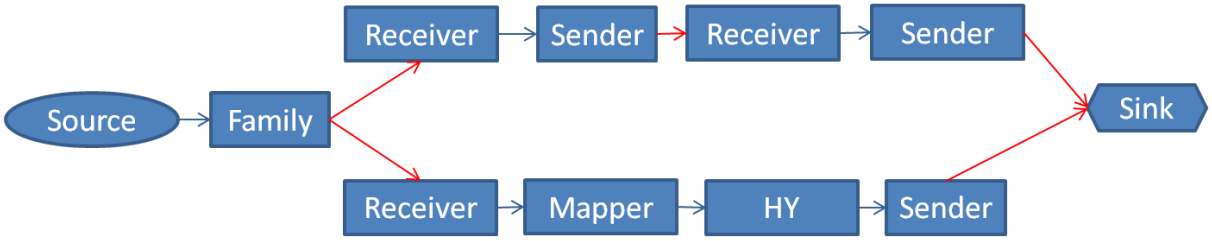

Figure 16: The integrated monitoring topology for the correlation computation family of the priority pipeline.

As discussed in D4.3, the Monitoring Layer uses the aggregated information to regularly create a frozen copy of the system state and to perform a constraint analysis in order to detect problematic situations and to notify the adaptation layer. Adaptive decision making is further supported by algorithm profiles (Section 2.7.4) as well as further alarm events sent by the source volume prediction (Section 2.7.5).

2.7.4 Algorithm Profile Prediction

Recording and learning the quality characteristics of individual algorithms as introduced in D4.3 is a core functionality of proactive adaptation in QualiMaster. As the algorithm profiles require a constant flow of observed values and just deliver predictions on request (of the Adaptation Layer), we decided to integrate the algorithm profiles and the prediction of quality characteristics into the Monitoring Layer. In D4.2, we discussed the approach as well as the initial performance for creating, updating / learning and predicting algorithm profile values. In particular, the re-implementation based on Apache Math significantly improved the performance of an individual

predictor. In this section, we discuss the integration of the profile prediction into the QualiMaster infrastructure, in particular into the Monitoring Layer. We first introduce the rationales behind the design and discuss then the design and the integration of the algorithm profiles into the QualiMaster infrastructure.

As introduced in D4.3, we rely on (distinct points in) the parameter and distribution space to characterize an individual data processing algorithm. In fact, the point *p* in the parameter and distribution space summarizes the relevant characteristics of the observations made by the Monitoring Layer for the algorithm operating with parameters / distribution settings according to *p*. Data processing algorithms are parts of families, i.e., pipeline elements, which, in turn, constitute pipelines. This forms the basic structure required for managing the profiles and the related predictors. However, not all observations made all parameters and, in particular not all potential values of the domains of observations and parameters are relevant. Also for performance reasons, we consider the parameter and distribution space as discrete and quantize the obtained values for parameters and distributes rather than allowing all individual values.

As discussed in D5.2, the QualiMaster infrastructure is layered and (mainly) event-based, in particular, communications between lower and higher layers (or better in general among layers) shall be realized in terms of events. This allows us to even distribute the layers among different machines. Among the variety of events provided by the QualiMaster infrastructure, only a small subset is relevant for maintaining the algorithm profiles, namely:

- **PipelineLifecycleEvent:** This event is sent out and processed upon changes in the lifecycle of a pipeline, e.g., when a pipeline is starting or stopping (see also D5.3 and D4.3 for the lifecycle phases). This allows loading or discarding algorithm profiles (as well as the shadow structure of pipelines used for managing the profiles).
- **ProfilingEvent:** If this kind of event occurs, a pipeline is not running in normal mode rather than in profiling mode (see also D4.3), i.e., the characteristics of a certain algorithm are currently being measured. This event allows the algorithm profiles to determine when an (initial) profile set is completed and shall be persisted for later packaging the profile along with the pipelines incorporating the algorithm.
- **AlgorithmChangedEvent:** An algorithm used for a certain pipeline element (family) was changed, either during pipeline startup (defining the initial algorithms) or during runtime. In both cases, this allows us to select the active profiles for prediction and update.
- **ParameterChangedEvent:** The parameter (of the running algorithm) of a certain pipeline element was changed. Akin to algorithm changes, this kind of event occurs either during startup (defining the initial parameters) or during runtime. A (changed) parameter value allows us determining the actual point in the parameter space (in combination with the actual distribution of the algorithm over the cluster).

In addition, the Monitoring Layer informs the algorithm profiles regularly about the actual observed values for all running pipelines. This happens through the mechanism for recording the monitored system state for profiling and for reflective adaptation. This information completes the information about the point in the parameter space where the algorithm is actually operating, as the observed information contains the distribution information (number of tasks, number of executors) as well as

the characteristics of the preceding algorithm (in particular the output rate of the preceding, i.e., the input rate of the actual algorithm in terms of tuples per second).

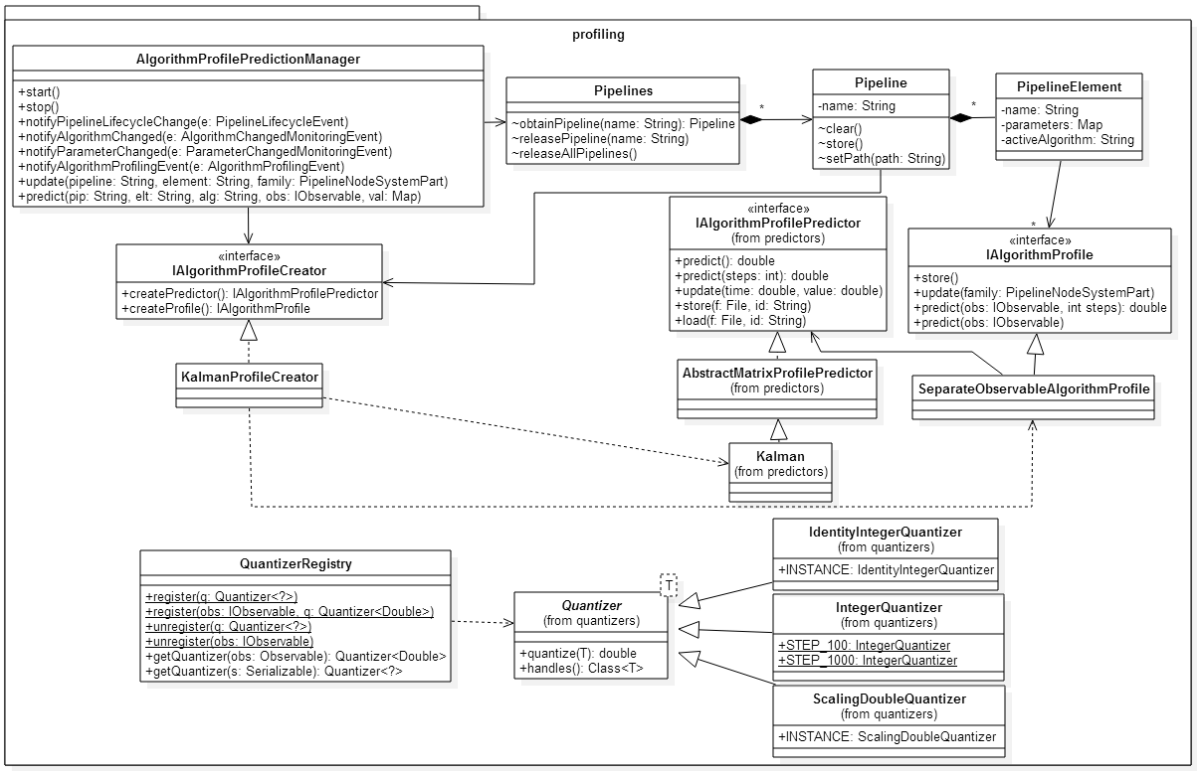

Figure 17: Design of the algorithm profiles as part of the Monitoring Layer.

The Adaptation Layer explicitly requests the predictions for individual algorithms or whole families. Following the event-based design of the QualiMaster infrastructure, the adaptation script issues a prediction request, the infrastructure turns this into a synchronous event handled by the algorithm parameters, which finally leads to a result in the Adaptation Layer.

The design for the algorithm profiles and their integration is driven by the management structure in terms of shadow pipelines, the quantization of values as well as the events and information provided by the QualiMaster infrastructure. Moreover, we also aim at a flexible design of the algorithm profile part. In particular, for experiments, we do not integrate the Kalman-based approach introduced in deliverable D4.3 in a fixed manner rather than shielding the Kalman filter behind an internal interface so that it can be exchanged by implementations of other prediction approaches. Further, the quantizers shall be extensible, so that new pipeline probes (see Section 2.7.1) can define and contribute the most appropriate quantizers. An overview of the package for algorithm profiles as part of the Monitoring Layer is depicted in Figure 17.

The `AlgorithmProfilePredictionManager` represents the external interface of this package. It consists of static methods called by the Monitoring Layer upon the events discussed above, the regular availability of new monitoring data as well as while starting and stopping the infrastructure. The `AlgorithmProfilePredictionManager` holds a reference to the active (shadow) pipelines as well as a reference to the profile creator, which is responsible for creating profile

predictors as well as compatible profile instances. In our case, the `KalmanProfileCreator` instantiates combinations of `Kalman` filter predictors with profiles that maintain one `Kalman` instance per relevant observable, i.e., instances of `SeparateObservableAlgorithmProfile`. In contrast to the system state of the Monitoring Layer, the `Pipeline` structure of the algorithm profiles is rather simple as it just represents the actual relevant information. A `Pipeline` consists of pipeline elements (`PipelineElement`), which are created and composed on demand based on the incoming events. A `PipelineElement` refers to the algorithm profiles (`IAlgorithmProfile`) of the algorithms in the respective family[2]. Finally, the algorithm profile contains one or more `IAlgorithmProfilePredictor` instances. The `QuantizerRegistry` allows registering all available quantizers, which are responsible for mapping a large / continuous domain to a quantized integer domain (just some examples are shown in Figure 17). Implicitly, the `QuantizerRegistry` decides about the relevant parameters and observables, i.e., if no `Quantizer` is registered, the respective value is considered to be irrelevant for algorithm profiles. Moreover, the `QuantizerRegistry` also holds the number of prediction steps per observable to be performed by the predictors.

At runtime, pipeline lifecycle, algorithm change, parameter change and profiling events are directly mapped to the shadow pipeline structure and cause respective changes, e.g., storing the actual algorithm or the current parameter settings in the related `PipelineElement`. An update or prediction request is passed through the structure until it finally reaches the respective predictor, i.e., the `AlgorithmProfilePredictionManager` selects the correct `Pipeline`, the `Pipeline` the correct `PipelineElement`, and the pipeline element the responsible profile. During the latter step, a persisted predictor (either the initial one from profiling or the updated one from previous runs of the same algorithm in the same pipeline) may be loaded on demand from a persistent storage. Finally, the update / predict request is passed to the matching predictor, which either updates its own structures or returns a prediction result. In turn, the result is passed back to the original caller. At latest upon terminating a pipeline or the infrastructure, the relevant information from the profiles is persisted. While update requests are issued in a regular form by the Monitoring Layer, prediction requests are created (implicitly) by the adaptation script and, as indicated above, passed as events to the Monitoring Layer, which, in turn, calls the `AlgorithmProfilePredictionManager`.

2.7.5 Source Volume Prediction Integration

The functional behavior of Source Volume Prediction has been already described and evaluated in D4.3. The main idea is to have a model that, given observations of the current volume of a source, can predict the volume of the source in the short future and raise alarms in case the volume is expected to reach critical values. In this section, we describe how the volume prediction has been integrated in the QualiMaster infrastructure. The natural location of the volume prediction component is the Monitoring Layer. Here, it monitors the available source(s) and communicates to

[2] Actually, in this simplified structure we handle pipeline sources and sinks akin in the same way as algorithm families and algorithms.

the Adaptation Layer. Figure 18 shows an overview of the integrated component for the Volume Prediction.

First, every time a source becomes available and is required to be monitored, a `IHistoricalDataProvider` object is sent to the Volume Prediction component (hereafter called *component* for sake of brevity). This is used to retrieve historical data (volumes) for the given source and train prediction models for each term. We derived the `IHistoricalDataProvider` to handle two types of sources, namely Spring and Twitter, in a dedicated way. The default terms for which historical data has to be retrieved are contained within the `IHistoricalDataProvider` and they can be changed by the Adaptation Layer at runtime via add/remove requests. The terms are organized into two sets. One contains the terms that should be monitored continuously, i.e. predictions are done based on a window of recent volumes. This terms are exactly what has been considered in deliverable D4.3. The other set contains terms that are not observed, but could be considered for monitoring by the Adaptation Layer in future. Before starting to monitor a new term, the Adaptation Layer can ask for an estimation of the volume that such new term would cause and, based on it, accept or reject the monitoring. Therefore, it is important to be able to provide an on-demand estimation only based on historical data (and not on recent volumes, since the term is not monitored yet). We refer to this case as *on-demand prediction*, since prediction requests are done asynchronously by the Adaptation Layer when observations for the given term are not available.

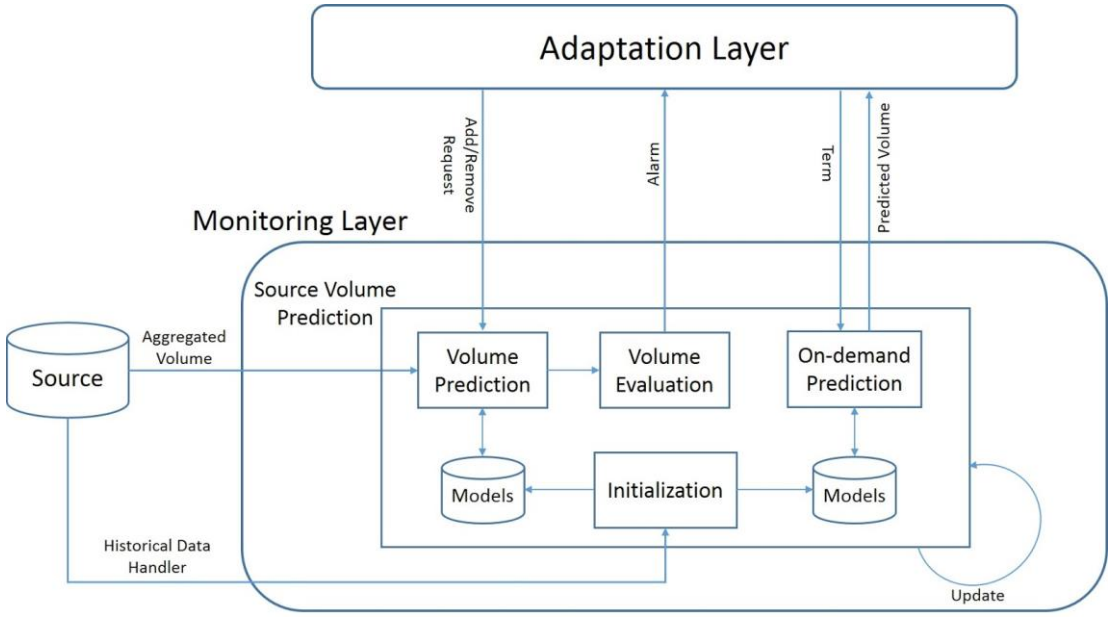

Figure 18: Overview of the integrated component for Source Volume Prediction.

Once sources have been initialized and prediction models have been trained based on the retrieved historical data, the component is triggered every time new volume observations are available. A `SourceVolumeMonitoringEvent` carrying aggregated volumes of each monitored term is raised by the source periodically, depending on the granularity assumed by the volume prediction, and handled by the component. We decided to work with a 1-minute granularity, as it offers short reaction time while not overwhelming the load of the infrastructure (see D4.3). For

each observation (term-volume pair), the component uses the model corresponding to that term to predict its volume at the next time step. If the predicted volume is declared as *critical*, then an alarm (a `SourceVolumeAdaptationEvent` object) is sent to the Adaptation Layer, which will decide how to react. The way the component determines whether predictions are critical is unsupervised and based on the recent history of volumes. More precisely, given an observed term, its average volume and standard deviation are computed within a sliding window of 10 past observations (10 minutes). The predicted volume is declared as *critical* if it exceeds such moving average by more than 3 times its standard deviation. This criterion has been frequently used in statistical quality control [Montgomery97].

Besides periodically monitoring the volumes of terms that are handled in a pipeline, the component also supports the so called *on-demand prediction.* As a results of adaptive strategies (e.g. Adaptive Crawling, WP4), the Adaptation Layer might change the set of terms handled within a pipeline. Before adding a term, it is useful to have an estimation of its volume to know in advance whether the pipeline can bear it, given the current load. The Adaptation Layer can get this estimation by sending a request to the Volume Prediction component, which answers by using the model corresponding to the requested term. Models for on-demand prediction are not based on time series forecasting, like the models for monitored terms are, since in this case there are no recent volumes available to make predictions. Differently, on-demand predictions for a given term and point in time are computed by averaging the volumes that the term exhibited at the given time of the day in all the past days contained in the historical data. Although simplified, we believe that such prediction can help the Adaptation Layer decide about accepting or rejecting the inclusion of a new term in a pipeline.

Finally, in order to keep the component accurate, the prediction models are updated every day by including the most recent volumes (last day) in the training data. Such update is scheduled nightly so that it does not overload the infrastructure during periods of high workload (i.e. during the day).

2.8 Loose pipeline integration

In QualiMaster, sub-pipelines representing distributed algorithms are introduced to implement complex stream processing algorithms, e.g., the Hayashi-Yoshida correlation algorithm. The sub-pipeline-based algorithms consist of distributed processing nodes allowing the processing to use distributed resources to run in parallel on a cluster. In the Storm topology, such distributed processing nodes are corresponding to individual Storm nodes, i.e., Spout or Bolt. However, when distributed algorithms are used as alternative algorithms in the processing pipeline, resources are requested to be allocated no matter whether the distributed algorithm is active for the current processing. This leads to inefficient utilization of resources in particular on those currently unused algorithms. To optimize the resource usage for this situation, we introduce a loose integration solution for the pipeline to allocate resources only for the algorithms which are selected to process analysis tasks. The main idea of the loose pipeline integration is to release the hard bound connection of the sub-pipeline algorithms from the main pipeline and dynamically connect/disconnect them at runtime.

As discussed in Deliverable D4.3, we have already supported the generation of sub-pipelines. For the loose pipeline integration, this provides the possibility to equip the sub-pipelines with specific nodes to control the input and output stream so that the sub-pipeline algorithm can be connected/disconnected to the main pipeline at runtime. Basically, we run the sub-pipeline individually and rely on the network connection to transfer data items between the main pipeline and the sub-pipeline. To be able to run the sub-pipeline individually as a general processing pipeline in the cluster, we equip the sub-pipeline with an intermediary Spout as the source receiving data from the main pipeline to feed the processing nodes of the sub-pipeline algorithm and an ending Bolt as the Sink forwarding the processing results back to the main pipeline.

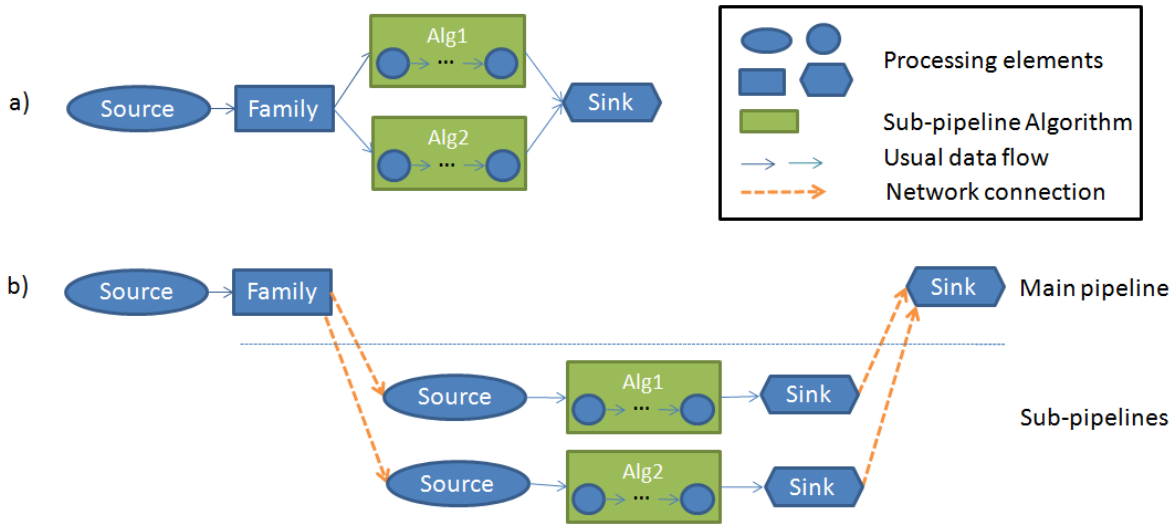

Figure 19: The design of the loose pipeline integration illustrated with an example pipeline in terms of a) original case b) loose case.

Figure 19 illustrate the design of the loose pipeline integration by showing the respective view of both original and loose cases on an example pipeline. The example pipeline is conducted with a data source, an algorithmic family and a data sink. The algorithmic family is equipped with two alternative sub-pipeline algorithms which consist of several processing nodes. Figure 19 a) depicts the original case that alternative algorithms are hard bound to the processing pipeline. When we deploy this pipeline into the cluster, resources will be allocated on all processing nodes of both alternative algorithms although there might be only one selected to run the current processing. The loose pipeline integration is illustrated in Figure 19 b). As can be seen, we replace the hard bound connections of both alternative algorithms from the main pipeline with the network connections. Furthermore, we equip the sub-pipelines with the source (intermediary Spout) to receive input streams from the main pipeline and the sink (ending Bolt) to forward processing results back to the main pipeline as discussed above. With these additional source and sink nodes, we can turn sub-pipeline algorithms into individual processing pipelines which can be running in parallel in the cluster. This loose design gains the flexibility to control the input and output streams between the main pipeline and the sub-pipelines, which allows the main pipeline dynamically connects the selected sub-pipeline algorithm at runtime. Moreover, this design is also in accord with the infrastructure layers, in particular the Coordination Layer for automatically taking control when such loosely integrated sub-pipelines have to be started or stopped as well as our design of the advanced switching in terms of the adopted idea of the intermediary Spout and the ending Bolt.

For our switch mechanism, the direct benefit from this loose integration is the optimized resource utilization.

For the evaluation of the loose pipeline integration, we will first focus on the performance of loosely integrated sub-pipelines (Section 5.2). The performance of switching along with this loose integration will be included in one of following deliverables.

2.9 Infrastructure Instantiation

As indicated in deliverable D4.3, we extended some configuration concepts and discussed briefly about respective changes of extended infrastructure instantiation. In this section, we provide the detailed discussion on the changes to the instantiation and how the changes are integrated into the QualiMaster infrastructure instantiation process.

The goal of the instantiation is to turn a valid configuration into deployable artifacts containing the data processing pipeline implementations, which fulfill the overall QualiMaster infrastructure requirements such as (parameter) signal processing, adaptation decision enactment or monitoring probes. Basically, we rely on a model-based approach using Software Product Line techniques, which aims at deriving pipeline implementations, integrating algorithms, creating supporting artifacts such as setup descriptions for the stakeholder application design environment and, finally, deployable artifacts. In QualiMaster, the infrastructure instantiation process mainly consists of three phases, i.e., the:

- **Interface phase** deriving interfaces for enabling domain-specific algorithm implementation. As described in deliverable D5.2, we basically derive interfaces for data sources, algorithmic families and data sinks to enforce type-safe implementation of pluggable components, including software- and hardware-based components.
- **Algorithm phase** in which the Algorithm Providers develop data analysis algorithms based on the above derived interfaces. In a strict sense, this phase does not seem to be involved in the automated instantiation. However, we utilize the results of the work of the data scientists to instantiate integrated pipelines. Moreover, as indicated in D5.3, we also generate specific algorithm implementations, e.g., the advanced hardware integration in terms of a software algorithm. Here, we generate the hardware integrating Bolt and Spout as well as the adapter framework for the reconfigurable hardware (FPGAs) to enable the communication between Maxeler and Storm infrastructure (see detailed design in deliverable D3.1 Section 3.2).
- **Pipeline phase** in which data processing pipelines are generated from a valid configuration. During the derivation of data processing pipelines, we support the automated integration (D5.3) based on Maven artifact specification in the Configuration, by creating and executing Maven build specifications for the pipelines and finally deploying pipelines (as well as the configuration itself) into the QualiMaster pipeline repository.

Due to the extended Configuration Meta Model described in D4.3 Section 2.1 as well as the current insights into adaptation enactments, we realize extensions for the algorithm and the pipeline phase.

For the algorithm phase, we also **generate the implementation of sub-pipelines**: As discussed in D4.3, Section 2.1.1, a recent addition is to support the configuration of sub-pipelines representing distributed algorithms. The aim is to enable adaptation on the inner processing nodes of such a distributed algorithm as well as to avoid tedious and error-prone implementation (as we experienced in the last integration phases) for alleviating the Algorithm Developers' work. Basically, for generating the sub-pipelines we reuse the previous pipeline model including the existing instantiation templates, recursively traversing the configured processing nodes through the data flow graph of the sub-pipeline. For supporting the adaptation on distributed algorithms, we generate monitoring probes along with each inner processing node to obtain the data processing information, e.g., latency and throughput of each processing node. Such monitoring information summarizing the runtime characteristics of the algorithm can significantly support for making adaptations, e.g., adjusting the resource allocation according to the needs or exchanging the algorithm for more efficient processing. The result is a set of Java classes implementing the pipeline nodes referring to the implementing algorithms. Moreover, we generate the integrating sub-topology structure, which caused some of the implementation problems in the past integration phases. We also generate the pipeline mapping specification (see deliverable D5.2 for details) for the sub-pipeline. It is important to note that we extended the specification to enable parameter redirection, i.e., to tell the Coordination Layer at runtime the actual receiver of infrastructure signals. This simplifies the implementation (a wrong or missing signal passing from the sub-topology to its implementing nodes was also identified as a source for implementation problems that were tricky to solve) and speeds up adaptive enactment, as a direct signal (typically 10 ms execution time) is faster than an explicitly redirected signal (at least 20 ms execution time). Sub-pipelines are deployed to the QualiMaster processing elements repository.

Further, we extend the pipeline phase with:

- **Support for algorithm switch.** As indicated in deliverable D4.2, Section 3.3, switching among alternative algorithms of an algorithm family is one of the key enactment mechanisms in the QualiMaster project. As experimented previously and detailed in [QE16], switching among distributed algorithms requires care to avoid gaps in data processing, e.g., transferring unprocessed tuples from the original algorithm to the target one for avoiding data loss. Initially, we experimented and evaluated several switching approaches with manually-implemented code in a test pipeline. Relying on manual written code for the beginning is reasonable, as the final code implies a rather complex communication pattern, which cannot be easily generated without having a working version. To enable advanced switching approaches as discussed in D4.2/D5.3 and [QE16] for all involved QualiMaster pipelines and to relieve the Algorithm Providers, we decided to also generate the switch-related code as part of the pipeline phase. Basically, we equip the distributed algorithm with an intermediary Spout receiving input tuples from the preceding node to feed the processing algorithm as well as an ending Bolt forwarding output results to the next consuming nodes. In the intermediary Spout, we generate explicit queues to control the incoming as well as outgoing tuples ensuring tuple sequences and guaranteeing that there is no data loss during the switch and that the actual algorithm can be terminated properly upon request. For control, signals are generated, in particular, to identify data transfer states of the algorithm, and therefore, trigger the actual switch at a certain point.

- **Support for loose pipeline integration.** As discussed in Section 2.8, we introduced a loose pipeline integration to support dynamic deployment of sub-pipeline algorithms at runtime for optimizing resource usage. Basically, we adopt the idea of above algorithm switch integration to equip the sub-pipelines with an intermediary Spout as the source node and an ending Bolt as the sink node to control the input and output stream between the main pipeline and the sub-pipelines. As the data transmission between the main pipeline and the sub-pipelines relies on the network connection, we support the dynamic port assignments for the receiving nodes to create the data-receiving server, i.e., for the intermediary Spout and the next consuming Bolt of the sub-pipeline algorithm. The assigned ports are registered globally in the cluster via ZooKeeper[3] services and can be traced when other nodes try to connect with the specific data receiving server to forward data items, e.g., when the ending node of the sub-pipeline sending results back to the main pipeline. For the startup of the pipeline with loosely integrated sub-pipelines, we first need to ensure that the selected sub-pipelines are already up and then start the main pipeline. This is supported by the Adaptation Layer in the pipeline startup phase via the infrastructure. Moreover, we support additional care of connecting the data source at right time, i.e., when the sub-pipelines are started up, to avoid data loss caused by the unavailability of processing algorithms.
- **Support for the Data Replay.** To enable insights into historical risk events on stakeholder side, we introduced the replay sink as explained in D4.3 Section 2.1.3. Based on the configuration of the replay sink, we store processed results through a replay recorder and replay the historical results on the application's requests through the replay streamer, both provided by the QualiMaster infrastructure (Data Management Layer). To ease the generation, replay recorder and streamer are generically combined in an abstract replay sink of the QualiMaster Storm library. To enable storing generic data, we represent the configured tuples types involved in a replay sink into code representations of the data scheme specifying the field types, keys for data retrieval and the timestamps of the tuples. We pass the derived schemas along with the configured location and storage strategy (cf. D4.1 for Data Management Node) to the replay recorder/streamers. Finally, we link the generic implementation provided by the QualiMaster infrastructure to the domain-specific sink implementation developed by an Algorithm Provider, also passing an identifier (called ticket in D4.3 Figure 5) for the actual data stream so that the stakeholder application can easily identify the received data.
- **Creation of setup files for stakeholder applications.** To support the design of new stakeholder applications in the stakeholder design environment (i.e., avoiding manually entering pipeline settings as it was the case so far), we turn all related configuration information into a generated setup file per pipeline. Such a setup file details the connection between pipeline and application, the data schema provided by the pipeline sinks as well as the supported general and pipeline-specific commands. Pipeline-specific commands

[3] ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

represent the user-triggers for requesting changes to the data processing at runtime based on the permissible parameters described in D4.3 Section 2.1.2.

The infrastructure instantiation also generates the respective code for **generic load shedding**. Following the design of the generic load shedding framework (Section 2.4), we generate now just specific methods for the data processing, which are called by the basic signal spouts and sinks. If load shedding is activated, the basic signal spouts and sings decide via the pluggable load shedders whether the actual data item at hand shall be processed or not and, in case that the item shall be processed, call the respective method in the generated code.

Besides improvements to the existing instantiation, all described changes in the QualiMaster infrastructure instantiation process are related to new configuration concepts. They are integrated into the overall instantiation process and used to derive pipelines. The instantiation process is also integrated into continuous integration, in order to test the instantiation, the reasoning and to automatically deploy the generated pipelines to the QualiMaster processing elements / pipeline repository.

3 Newly Incorporated Processing Pipelines

This section introduces and discusses in details the newly incorporated processing pipelines. The partners are currently in the process of finalizing, testing and improving these pipelines. Our goal is to have them fully working by the end of the project.

3.1 Transfer Pipeline

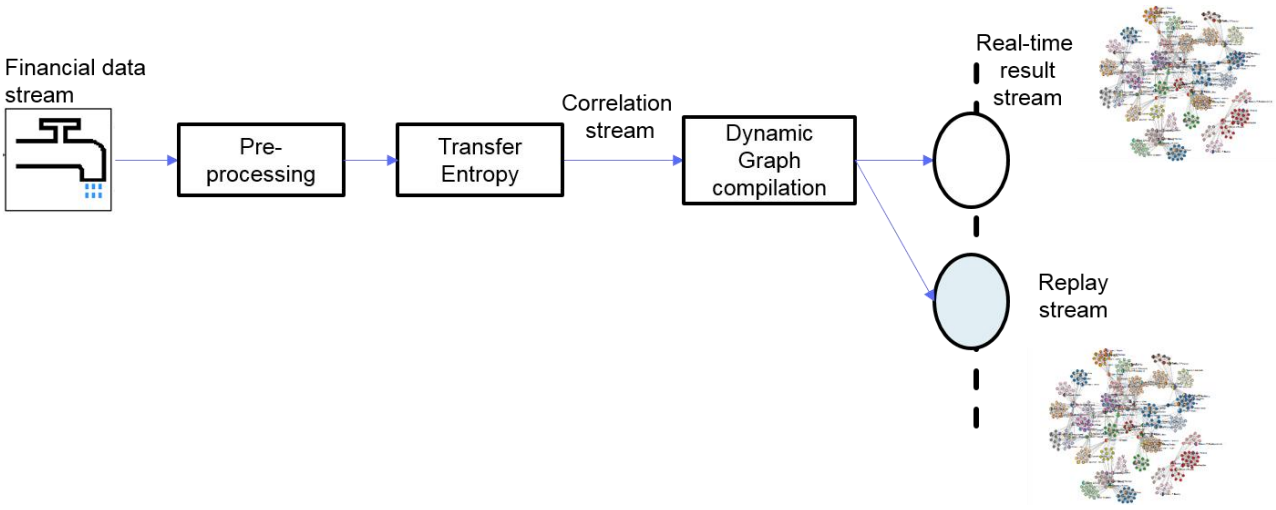

Figure 20: Graphical illustration of the Transfer Pipeline containing transfer entropy computation and the replay mechanism.

In the second phase of the project, we introduced a new pipeline, the **Transfer Pipeline** - for showcasing two core innovations of this phase of the project, the Transfer Entropy computation on Financial Stream and the Replay Mechanism. The **Transfer Pipeline**, shown in Figure 20**,**

includes a novel component for computing correlations based on the concept of Transfer Entropy. This enables the financial stakeholder to see the direction of the dependency instead of the correlation alone. This gives a better understanding on the dependency situation and enables improved financial decisions. Transfer Entropy has been implemented in Hardware and in Software. Details are described in D2.3 and D3.3.

Furthermore, the Transfer Pipeline includes functionality for replaying processing results, e.g., the correlations computed by the pipeline. This functionality reacts to the need of financial stakeholders to revisit results from the past for better understanding current developments (e.g., how did a similar situation in the past evolve, what was its impact). Therefore, in addition to real-time risk analysis, financial stakeholders are supported in digging into past analysis results on different time scales and aggregation.

This capability is provided by the Data Replay mechanism (see D2.3 and the description of its implementation in Section 2.5 of this deliverable), which enables the replay of past pipeline processing results at dynamically selected levels of granularity and speed. The Replay mechanism is a generic component that can be configured into a QualiMaster pipeline in terms of a Replay Sink (see D4.3, illustrated in Figure 2). From the technical point of view, this mechanism stores real-time data in batch modes to the data store that can handle timestamped data such as HBase, and unifies query results from the data store with the normal stream. The granularity and speed can be dynamically selected; it is efficiently supported through the use of an asynchronous buffer. More details are provided in Section 2.5.

The Transfer Pipeline aims at demonstrating and experimenting with the Data Replay mechanism, in particular regarding the aggregation capabilities, the impact on the overall performance and the utility provided to the user.

3.2 Focus Pipeline Version 2

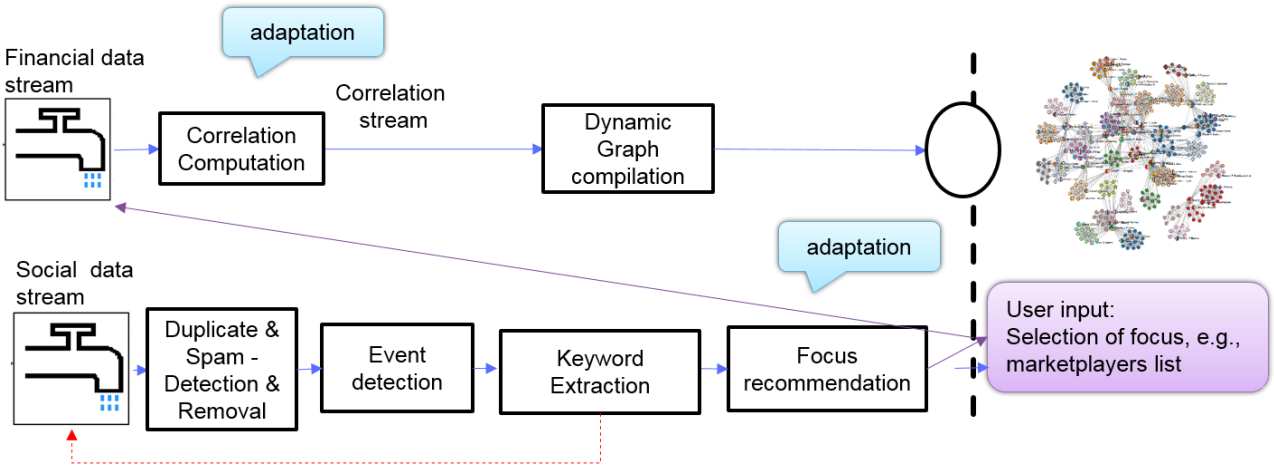

Figure 21: Graphical illustration of the focus pipeline version 2

The first version of the Focus Pipeline has been introduced in D5.3. The current version of the pipeline is depicted in Figure 21. The main idea of the pipeline is to use a semi-automated approach to focus on specific parts of the correlation graph. The user is therefore informed in real time if an event for a certain market player is taking place at the moment. She can then decide if she wants to include this market player into her personal visualization of the correlation graph which depicts current correlations between market players.

The pipeline consists of two parts, where each part contains its own source and offers an output to the user. The lower part of the pipeline uses social media data (e.g., tweets) and social media specific components to forward suggestions in the form of keyword-enriched events to the user. Therefore the social media stream source crawls current tweets and feeds them into the pipeline. The tweets are preprocessed, i.e., detected duplicates and spam are removed since these Tweets might easily lead to false conclusions. In the next step the Event Detection component uses one of the algorithms from the Event Detection Family to detect events in the stream. The component is able to monitor many different market players at the same time. If an event for a market player is detected it is further enriched with co-occurring keywords and additional information (e.g., time information and impact value of the event). Optionally, important keywords could also be used to modify the source (as indicated by the red feedback loop in Figure 21). Afterwards, the Focus Recommendation component will aggregate events and forward them to the user pointing the user to market players involved in the respective events as candidates to include into the list of her observed market players.

The upper part of the pipeline contains components for computing and visualizing the market player correlation graph in real time, including the user input for the selection of certain market players. The graph is then computed in the Dynamic Graph Compilation component and forwarded to the user interface where it will be visualized.

All components of the Focus Pipeline are explained in more detail in D2.3. Further description on the Focus Pipeline can be found in deliverable D6.3.

3.3 Dynamic Correlation Graph Pipeline

QualiMaster's Dynamic Correlation Graph Pipeline, shown in Figure 22, aims at identifying the "most important" market players that are currently monitored. The goal is to provide the stakeholder applications a list of the "most important" players, where the list will be visualized and presented to the user in a comprehending and intuitive way.

The input of the pipeline is a stream of financial data, where each tuple represents a stock transaction. To identify important players, we first create the correlation table of all market players using our correlation computation family of algorithms.

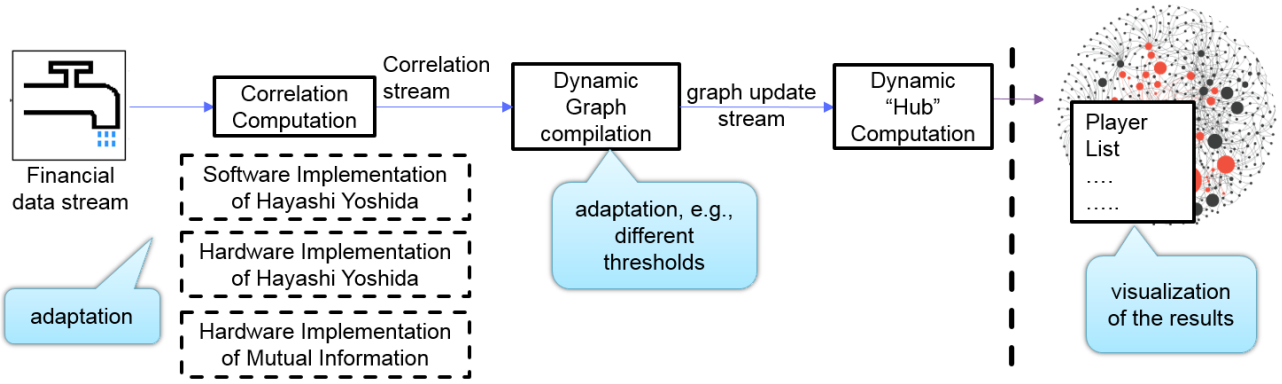

Figure 22: Graphical illustration of the dynamic Correlation Graph pipeline.

We then dynamically compile a Stock-Correlation Graph (SC-Graph) using the "Dynamic Graph Compilation" processing element. This element receives streaming updates of the correlation matrix from the "Correlation Computation" element and compiles the SC-Graph where the vertices represent market players and the edges correlations between them. To compile the graph at real-time, we use a user-defined correlation threshold. When the correlation of two market players is above the threshold, an edge between the corresponding vertices is added to the graph. In the same fashion, edges are removed from the graph when correlation values drop below the threshold.

The graph updates are streamed to the "Dynamic Hub Computation" processing element that will identify the most important market players. To calculate the most important market players, "Dynamic Hub Computation" element basically runs a Page-Rank algorithm to rank the players according to their significance. The description of this processing element has been already included in the D2.3 deliverable. Finally, the resulting list of significant players is streamed to the stakeholder applications where they will be visually presented to the user.

3.4 Time-Travel Correlation Graph Pipeline

Figure 23 presents the Time-Travel Correlation Graph Pipeline. As also discussed in the D5.3 deliverable, this pipeline provides the stakeholder applications data about the history of the Stock Correlation-Graph (SC-Graph). The main idea is that the pipeline constantly monitors the SC-Graph and stores information about the structure and data evolution. Then, upon a user request the pipeline executes the queries and sends the results back to the user. As described in the D2.3 deliverable, the queries could be either snapshot related (e.g., "give me a snapshot of the graph at time X") or path related (e.g., "retrieve the shortest path between node na and node nb from time t1 to time t2").

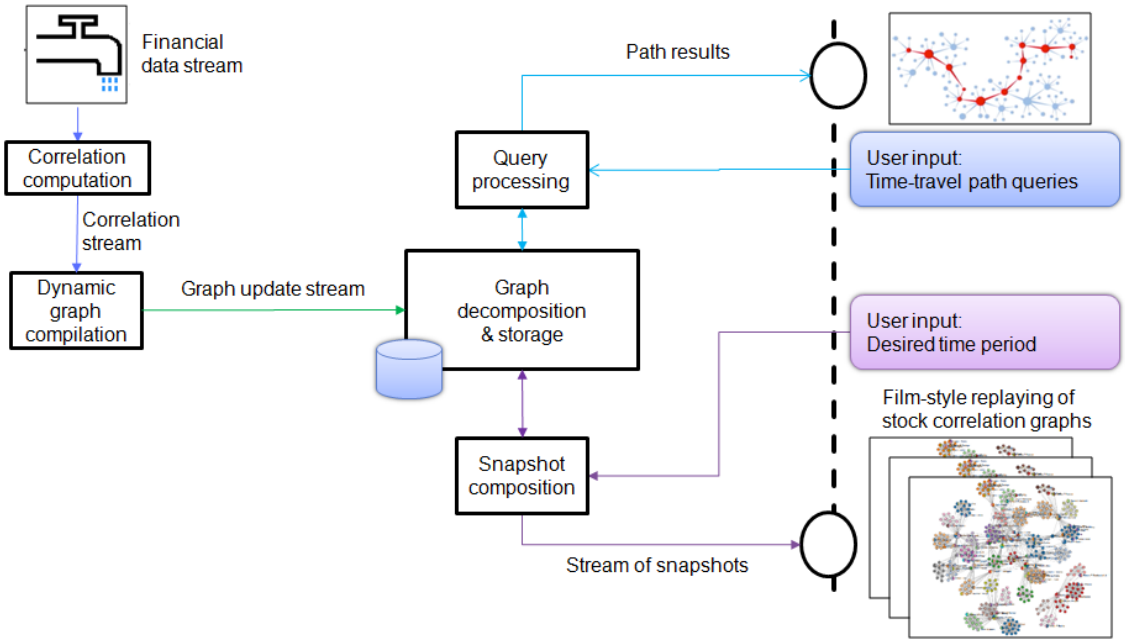

Figure 23: Graphical illustration of the time-travel Correlation pipeline.

To support such functionality, we exchange the “Dynamic Hub Computation” processing element of the “Dynamic Correlation Graph Pipeline” (described above) to the Time-Travel elements. For this pipeline, the graph update stream is fed to the “Graph decomposition & storage” processing element instead of the “Dynamic Hub Computation” one.

The “Graph decomposition & storage” processing element decomposes the graph to its basic operations in order to efficiently store them. We consider as basic graph operations the vertex and edge additions as well as the edge expirations (e.g., when an edge is removed from the graph). Having this decomposed information, the “Query processing” and “Snapshot composition” processing elements can answer the user queries. More detailed description of this processing element has been included in the D2.3 deliverable.

Currently, the pipeline supports only snapshot queries for a specific time point. Our plan is to support the full functionality, as described in the D2.3 deliverable, by the end of the project. This includes retrieval of snapshots for periods of time as well as processing path queries.

4 Data sets and Methodology

4.1 Evaluation Data Sets

Deliverable D5.3 presented a small set of data sets, which we used for performing our previous experimental evaluations (reported in D5.3). These data sets were also partially used for the evaluations reported in this deliverable. The following table provides a summary of the data sets characteristics.

Name	Market Players	Total ticks	Total seconds	Avg ticks per second
SRD-A	125	2526319	86400	29.2
SRD-B	2830	36335876	85960	422.7
I-ALL-A	100	defined according to the needs of the particular experiment		100
I-ALL-B	250			250
I-ALL-C	500			500
I-ALL-D	750			750
I-ALL-E	1000			1000
I-MP-A	250			250
I-MP-B	500			250
I-MP-C	750			250
I-MP-D	1000			250
I-TS-A	750			250
I-TS-B	750			500
I-TS-C	750			750

In order to show the performance advantages of our new developed solutions, new synthetic data were generated. The data sets, **referred to as SD**, were based on the worst-case scenarios as far as the number of needed computations and the needs for transfer rate of the computed results. Specifically, the data sets consisted of increasing number of market players and transactions per second concurrently. The characteristics of the generated dataset are shown in the following table.

Name	Market Players	Total transactions	Avg ticks per second
SD-250	250	90000	250
SD-500	500	180000	500
SD-750	750	270000	750
SD-1000	1000	360000	1000
SD-1250	1250	450000	1250
SD-1500	1500	540000	1500
SD-1750	1750	630000	1750
SD-2000	2000	720000	2000

In addition to the above, we also used real data from the UK referendum, known as Brexit. This corresponds to one of the busiest hours of one of busiest days (both in terms of active market players and in terms of total ticks), namely the 30th of June 2016 at 16:00:00-17:00:00. The particular data was divided into data sets, **referred to as RD**, where only a subset of the original market players was kept, in order to evaluate scalability incrementally. The table below summarizes the characteristics.

Name	Market Players	Pairs (thousands)	Total ticks	Avg ticks per second
RD-100	100	10	60960	16.9
RD-200	200	40	166364	46.2
RD-300	300	90	236157	65.6
RD-400	400	160	295261	82
RD-500	500	250	345032	95.8
RD-600	600	360	399328	110.9
RD-700	700	490	439524	122.1
RD-800	800	640	468077	130
RD-900	900	810	493256	137
RD-1000	1000	1000	512078	142.2
RD-1100	1100	1210	520395	144.6
RD-1200	1200	1440	521763	144.9
RD-1265	1265	1600.225	521864	145

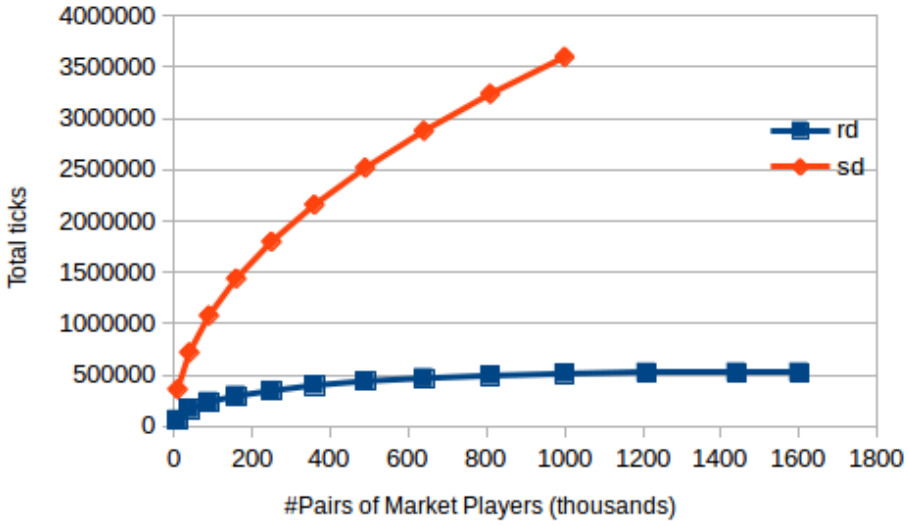

Figure 24: Overview of the RD (i.e., Brexit) and the SD (i.e., synthetic) data sets.

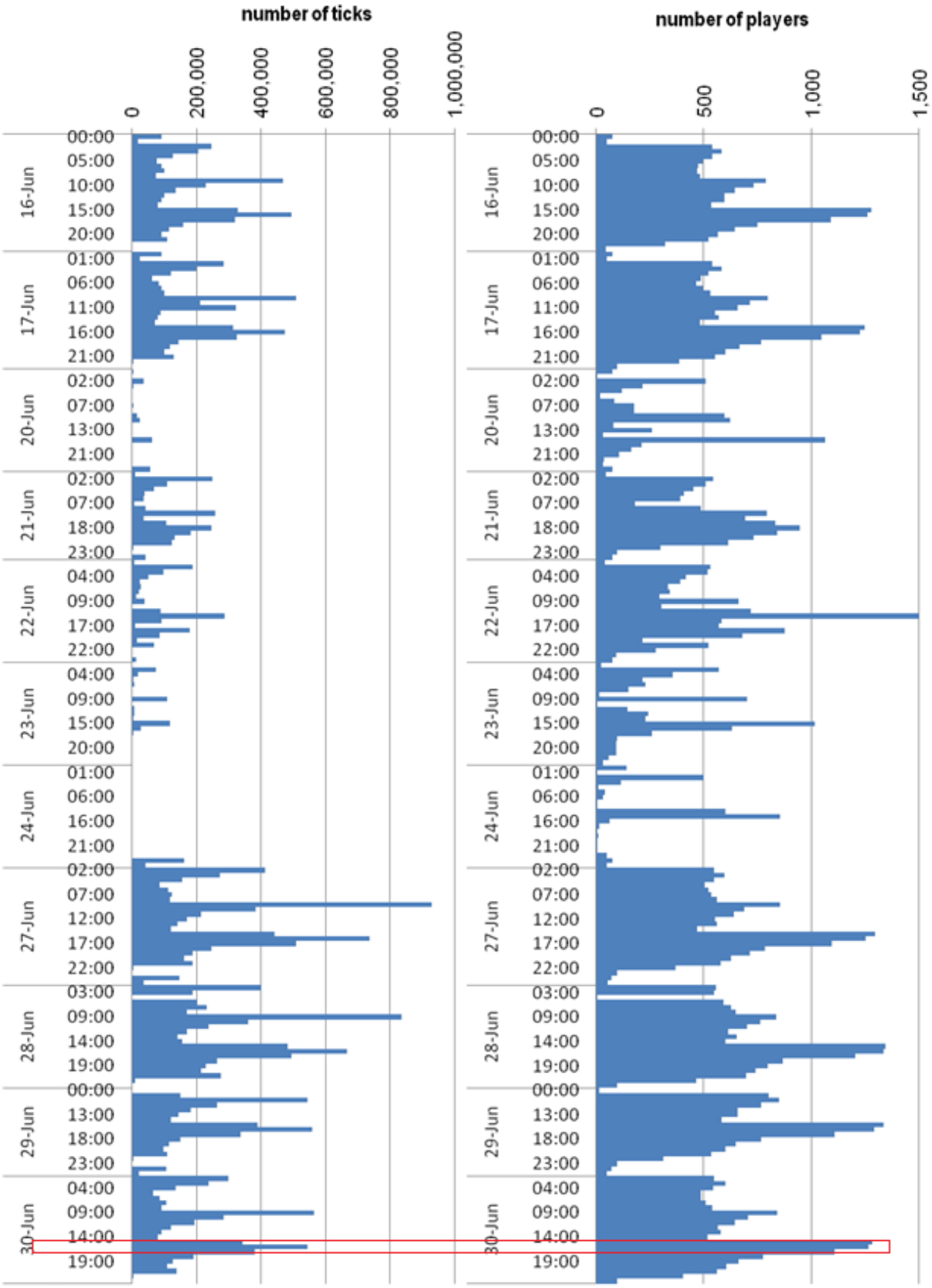

Figure 25: Number of ticks and players grouped by hour and day for the Brexit data.

Figure 24 shows how both data sets scale (in terms of total ticks) based on the number of market player pairs selected. Note that the RD data set was chosen as it represents the worst-case scenario in terms of computation volume regarding real world transactions and not synthetic data.

Among the 2-week period surrounding the BrExit referendum (i.e., 16th to 30th of June 2016), we decided to isolate an 1-hour data set to act as our real data set (RD). We analyzed the full 2-week data set, in an 1-hour granularity, both in terms of volume (i.e., total ticks for the specific hour) and in terms of the number of active data players. These are shown in Figure 25. Since all the implemented QualiMaster algorithms are calculating pair-wise correlations, the metric that mostly determines the worst-case scenario for them is the number of active market players, rather than purely the volume, since the complexity of those algorithms grows in the order of N^2, when dealing with N market players. Hence, we picked the 30th of the month at 16:00 because it shows a good combination of total ticks (i.e., 544216) and total active market players (i.e., 1266). This is illustrated with a red rectangle in the two plots of Figure 25.

4.2 Methodologies for Validation

The validation of the results that are produced by such hybrid and heterogeneous systems is of high importance. All the integrated components were validated according to generic methodologies that are followed in such cases and they are briefly described below (details provided in the previous WP4 deliverables).

Comparison against verified algorithm (VA): All the algorithms that were used in the context of QualiMaster project are verified by a traditional software implementation. Thus, we verified the final output results of each one of our software- and hardware-mapped processing elements using the results of the official algorithm distributions. Next, we moved on the verification of the final and the intermediate results of the implemented pipelines using various combinations of the processing elements vs. the results that are produced by the combinations of the corresponding official software implementations. Last, it is important to mention that a large variety of input data sets were used for the validation tests.

Expected Behavior (EB): The second validation methodology is investigating the output results for specific input data sets. More specifically, we used real and synthetic data sets, whose behavior was a-priori known. Thus, when we executed tests on the implemented processing elements and the pipelines, we were trying to verify that the output results followed the expected behavior. In addition, this test step included the exhaustive test of the implemented modules using specialized tools, such as the Google tests.

Expert Verification (EV): Last, as the consortium of the QualiMaster project consists of experts on financial domain, the final validation of the implemented systems was performed by those experts. Specifically, they validated the output results of the QualiMaster pipelines using either simple methods, i.e., manual verification, or other financial expertise tools.

4.3 Methodologies for Performance Measurements

The QualiMaster infrastructure already provides a wide set of monitoring probes to measure the non-functional quality dimensions of the QualiMaster quality taxonomy (D4.1 / D4.2). Differently, in this section we focus on measuring the functional quality of algorithms. This allows to estimate the quality hierarchies of the algorithms within a family, keeping them into account along with other non-functional qualities, and eventually to pick the best possible algorithm subject to some infrastructure constraints (e.g. based on load, throughput, available resources, etc.). Different functional quality dimensions exist (see the QualiMaster quality taxonomy in D4.1 and D4.2), each of them being applicable to a different range of problems. We decided to focus on *accuracy* because (i) it can be defined for every problem and (ii) it is perhaps the most significant indicator in most cases.

While the accuracy of an algorithm can be relatively easily assessed statically, i.e. based on a benchmark/ground truth, it can get hard to be estimated dynamically based on unseen (and unlabeled) input data. To do this, one should manually check the output of the algorithm, which is not a desirable option and it even becomes unfeasible in streaming scenarios. Therefore, to measure accuracy dynamically, we had to make some simplifications depending on the family under consideration. Given a family and a set of algorithms within it, we start by defining a “static” value of accuracy (between 0 and 1) for each algorithm based on an evaluation done on an external benchmark. This allows knowing how accurate the different algorithms are in general (based on the available ground truth data) and, consequently, to estimate how they are likely to perform for new and potentially different data. Of course this is an assumption, since the accuracy can depend on the type of input data algorithms that are inaccurate in some cases (data) can be more accurate in other cases. However, in many cases this is the only reliable assessment that can be done. For families like Event Detection and Sentiment Detection developed within WP2, we already performed a comparison of the accuracy of different algorithms based on a common ground truth (see D2.2 and D2.3). We exploit this experience to derive static values of accuracy of the algorithms.

Starting from such static assessment of accuracy, the idea is to modulate it with the values of parameters that depends on the characteristics of the streaming input data and that might affect accuracy. These parameters depend on the family: in case of Event Detection and Sentiment Detection, for instance, the input volume and noise (estimated as the amount of pruning done by the Spam Detection) might affect the accuracy of an algorithm. We plan to investigate possible dependencies of accuracy on these and possibly other parameters, to modulate the initial and static value of accuracy accordingly.

The methodology described above to estimate accuracy can be augmented in cases where more information is available and exploitable online to estimate accuracy. For instance the Sentiment Detection and Spam Detection families include algorithms based on Support Vector Machines for binary classification. Besides the binary output, the model can also provide the probability distribution of an input pattern over the two classes. This information can be interpreted as confidence of the model in making binary classifications (the closer to 0.5 is the probability, the

more the model is unsure) and, consequently, can be used to modulate the initial static value of accuracy.

Another functional quality measure that we consider is *diversity*. It is often considered along with accuracy as it offers a complementary and concurrent notion of quality (e.g. the relevance-diversity trade-off in search engines [CKC08]). Differently from accuracy, diversity can be computed in an unsupervised way, making it easier to assess in online and streaming scenarios. The meaning of diversity and the way it is computed depend on the problem being considered. In Event Detection, we measure diversity of an algorithm in terms of the number of different market player for which the algorithm detected events within a window or recent time points. The notion of diversity does not fit to other problems like Sentiment Detection and Spam Detection, therefore we do not measure it in these cases.

5 Evaluation of Infrastructure Components

5.1 Integration of hardware as part of a pipeline

This section describes the integration of the hardware platform into the QualiMaster infrastructure and the running pipelines. First, we describe briefly the recent state based on the Deliverables D3.2, D.3.3 and D5.3. Next, we identify the main drawbacks of the previous solution and we present our new revised advanced integration framework.

The previous state of the integration of hardware platform into the QualiMaster pipelines is presented in Figure 26. As described in deliverable D3.3, we implemented a software framework that is mapped on the Maxeler hardware server and it is responsible for the communication between the hardware platform and the pipeline adaptation layer. The adaptation layer sends via a dedicated TCP link all the needed information, including the needed parameters, about loading/unloading each one of the hardware-based algorithms on the Maxeler server. The software from the hardware-side responds with all the needed information, e.g., the number and the ids of the opened TCP ports, which the pipeline infrastructure uses in order to create the “connection” between the HW algorithm topology and the Maxeler platform. In addition, this software-based module is responsible for loading/un-loading the hardware-based modules, i.e., Dataflow Engines (DFEs), with the corresponding hardware-based configuration file.

The hardware algorithms are implemented as hardware libraries, which can be loaded and unloaded dynamically on a Maxeler server. Algorithms are mapped both on the software-based part of the server, i.e., CPU, and the hardware-based part, i.e. Data Flow computing Engine DFE.
The hardware part maps the corresponding hardware configuration file for each one of the implemented algorithms, i.e., Hayashi Yoshida correlation, Mutual information and Transfer Entropy. These configuration files are hardware libraries, which can be loaded and unloaded through a simple function call procedure. The main workload of the algorithms takes place in the DFE modules using the data, which are passed through specific interfaces by the software part of the algorithm.

The software part consists of three parallel running threads, i.e. the receive thread, the process thread and the transmit thread. The process thread executes the main algorithmic steps of the mapped algorithm. The received and the transmit threads are responsible for the communication between hardware platform and HW-algorithm sub-topology, which is mapped on the QualiMaster infrastructure. In more details, the Storm Bolt from the sub-topology, i.e. data source, receives streaming tuples from the Pipeline infrastructure and packs them using a Google Protobuf serialization scheme. Next, it sends the serialized data via a network link to the Maxeler server. Concurrently, the Spout, i.e. data sink, receives the results from the Maxeler transmit thread and deserializes them into tuples. At this point, it is important to mention that the previous hardware-based streaming framework used a single TCP port for transferring the input data from the pipeline to hardware and another TCP port for transmitting results from hardware to the pipeline.

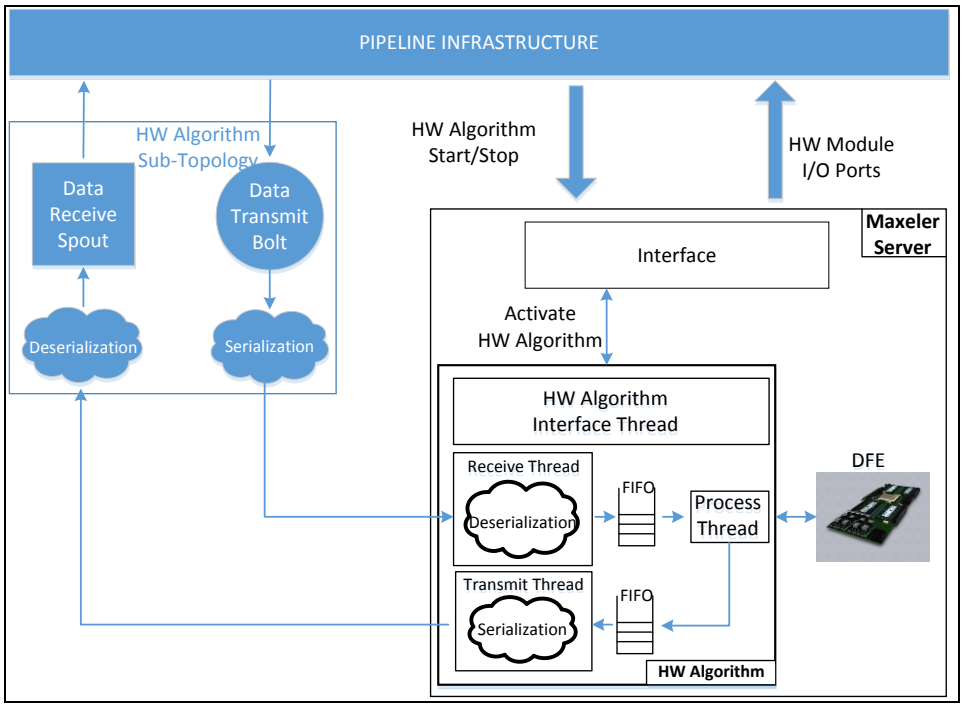

Figure 26: Previous state of hardware integration into QualiMaster infrastructure.

The main issue of the previous proposed hardware-based streaming framework was the low transmission rate achieved from hardware to Storm-based sub-topology due to internal restrictions of Storm framework. We moved towards a solution that would increase the bandwidth between the hardware platform and the hardware components integrated in Storm pipelines. Thus, we increased the parallelism level of data transmission between the hardware platform and the respective components in pipelines. Specifically, we increased the number of the TCP ports, which are used for the communication between the hardware algorithm sub-topology and the mapped hardware algorithm on the Maxeler server. In addition, we increased the parallelism of the transmission threads on hardware server by mapping parallel threads, where each one of them sends data over a different TCP port. Last, we increased the thread parallelism into the Spout module, thus, each thread receives data from a different TCP port. This solution seems to offer much better performance results for the complete pipeline infrastructure, as indicated by the results shown in Table 3 and Table 4: The results of the priority pipeline when computing the correlations over the data sets of the increasing market players and transactions per sec collection using SW and HW implementations.Table 4 (see Section 6.6.2). More technical details about the parallelization of transmission process will be described in the upcoming D3.4 deliverable.

5.2 Performance of loosely integrated sub-pipelines

As indicated in Section 2.8, we introduced a loose pipeline integration to dynamically deploy the sub-pipeline-based algorithms at runtime for optimizing the resource usage. In this section, we will report the performance of loosely integrated sub-pipelines when they are used as alternative algorithms in a processing pipeline. The aim of this evaluation is to verify that there is no negative impact caused by the loosely integrated sub-pipelines while optimizing the resource usage.

To evaluate the loose pipeline integration, we conduct a simple test pipeline consisting of a data source, an algorithm family and a data sink as illustrated in Figure 19. The sub-pipeline algorithm belonging to the family is composed of one processing node which only pass through the received data items. We evaluate the throughput at the sink for both original and loose cases at the input stream rate of 1000 and 5000 items per second respectively. For evaluating the impact of the loose integration, we take the performance of the original case as the baseline.

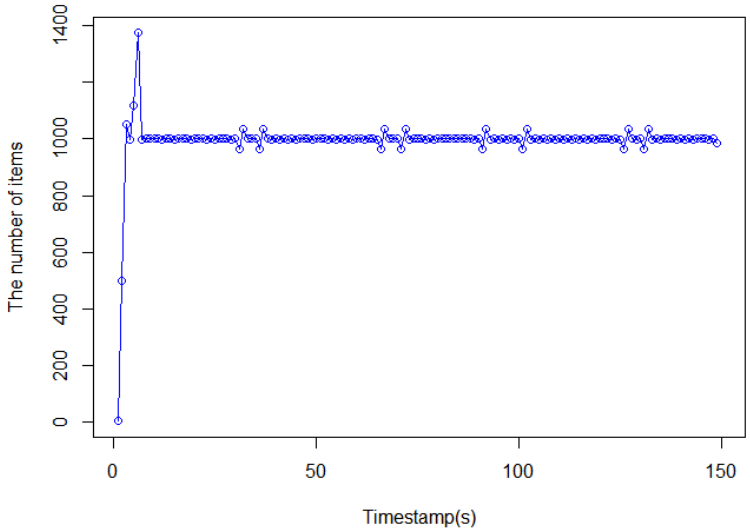

Figure 27: Sink throughput with the original pipeline running at input stream rate of 1000 items per second.

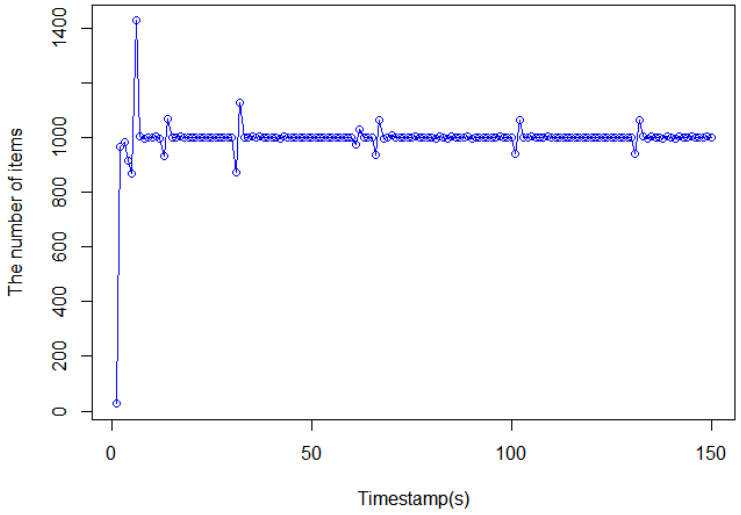

Figure 28: Sink throughput with the loose pipeline integration running at the input stream rate of 1000 items per second.

Figure 27 illustrates the sink throughput with the original pipeline running at the input stream rate of 1000 items per second while the one with the loosely integrated pipeline is depicted in Figure 28. As evaluated in deliverable D4.2, a pipeline needs some time after its startup to stabilize the processing. This also reflects on the unstable results at the beginning in this evaluation. After it reaches the stable throughput, the throughput mostly remains at constant 1000 items per second as we can see from both diagrams. There are some small fluctuations occurring over time. As we already verified in D4.2, this is caused by the internal affects of Storm framework.

To be more precise for the conclusion, we calculate the average throughput as well as the derivation for both cases. As shown in Table 1, the average throughput in both cases is very close to the expected value of 1000 items per second and the derivations are rather small.

	Average throughput(items/s)	Derivation
Original case	993	1.48
Loose case	992	1.48

Table 1: Average throughput and derivation for both original and loose cases at the input rate of 1000 items per second.

Analogously, we also evaluate the loose pipeline integration at the input stream rate of 5000 items per second. Table 2 depicts the average throughput and derivation while running the test pipeline at the input rate of 5000 items per second. The average throughputs in both cases are 4995 and 4983, respectively, which are close to the expected value of 5000 items per second. We can also see from this table, the derivation at the input rate of 5000 items per second is much larger than the one at 1000 items per second. This is due to the high workload causing more fluctuations at the sink throughput. The derivation difference between the original and the loose case is identical but still acceptable in particular with such high workload.

	Average throughput(items/s)	Derivation
Original case	4995	536.70
Loose case	4983	704.97

Table 2: Average throughput and derivation for both original and loose cases at the input rate of 5000 items per second.

Thus, we can conclude that the loose pipeline integration has no negative impact on the pipeline and we can take the advantage of the dynamic deployment of sub-pipeline algorithms to optimize the resource usage for the QualiMaster pipelines.

5.3 Performance of integrated source volume prediction

An evaluation of the Source Volume Prediction on real-world financial data, consisting of the trade volumes of 30 stocks from the Nasdaq and NYSE stock exchanges over 1 year, has been already performed and reported in D4.3. In this section we consider the behavior of the component for

volume prediction integrated in the QualiMaster infrastructure (Section 2.7.5), since this component's evaluation goes beyond the accuracy of the volume prediction itself. We consider financial data for this evaluation, but the same can be done for social data with few modifications in fetching historical data. Being an integrated component, the automatic assessment of critical volumes and the raise of alarms to the Adaptation Layer play an important role as well.

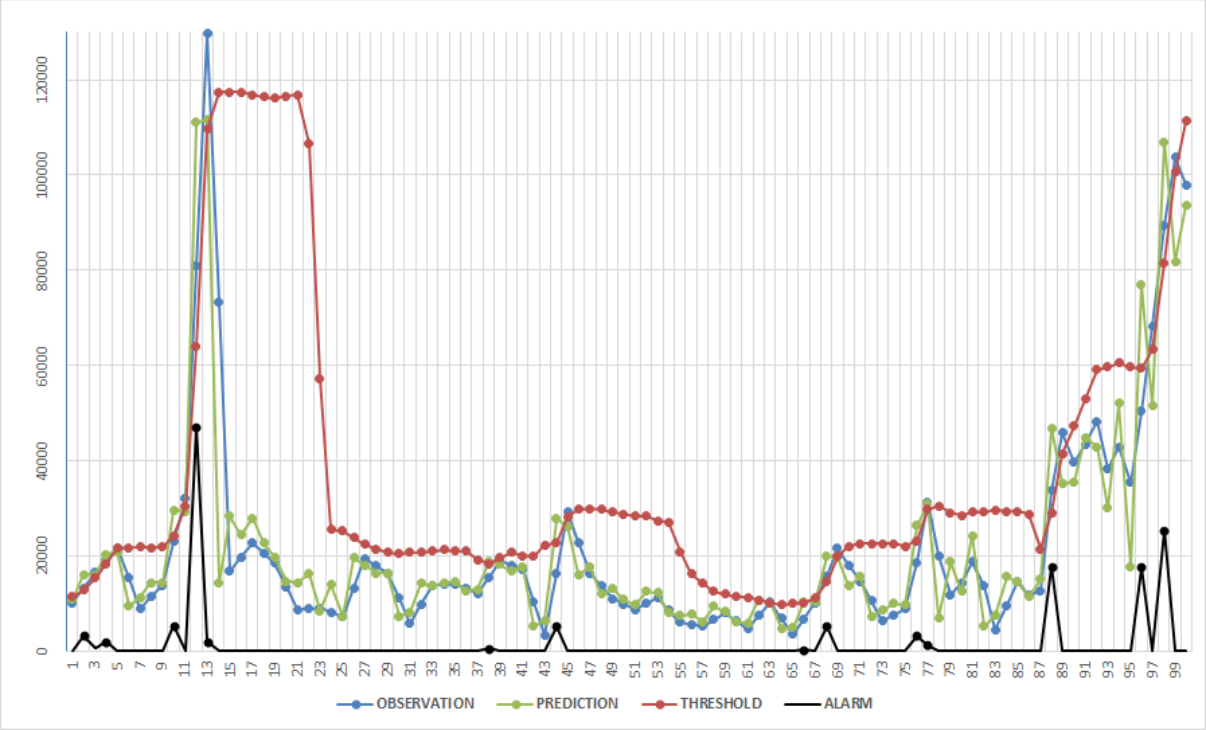

Figure 29: Integrated Source Volume Prediction applied to $NFLX.

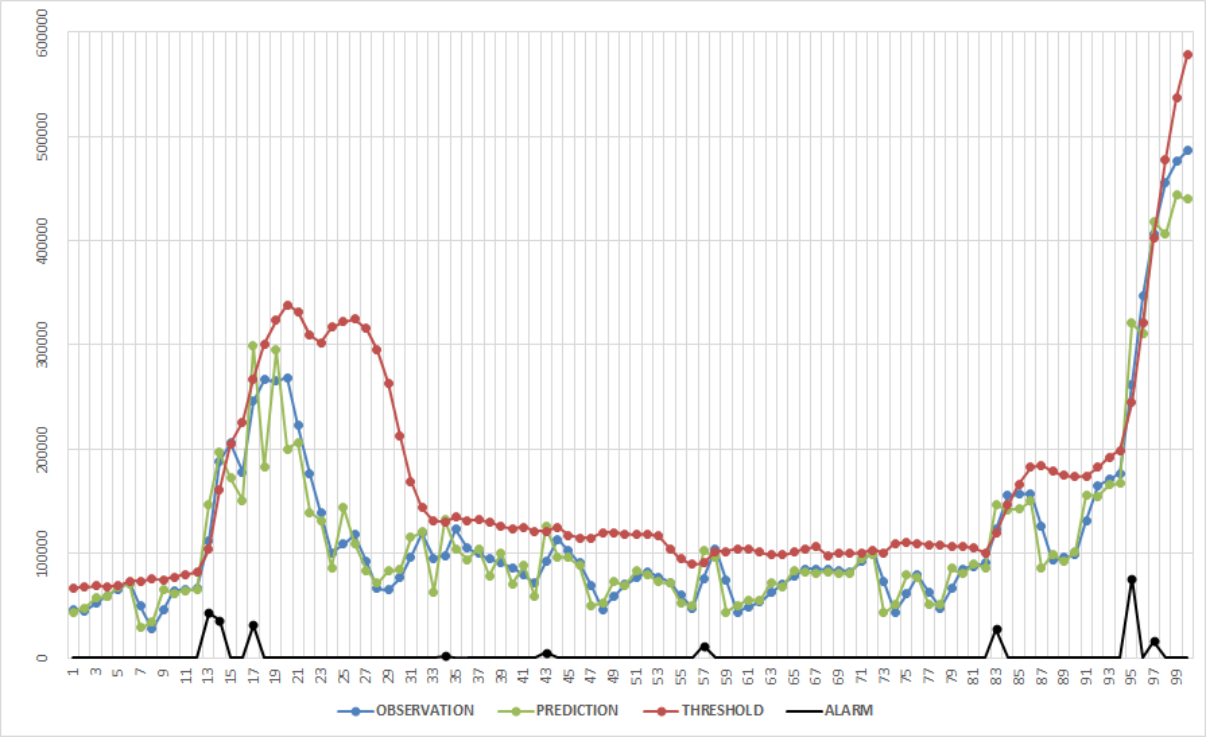

Figure 30: Integrated Source Volume Prediction applied to $AAPL.

To test these aspects, we simulated the streaming of volumes for two market players, namely $NFLX and $AAPL, and observed how the component for volume prediction behaves over time. Note that the volume data used for the test is real, only its streaming has been simulated. The time granularity used for the test is 1 minute, which means that the source volume for each market player is aggregated every 1 minute and sent to the volume prediction component, which makes a prediction of the volume of each market player within the next minute and possibly raises alarms to the Adaptation Layer. The streamed test data consists of 100 aggregated volumes for each market player. The initialization phase (see Section 2.7.5) has been executed before streaming the data: the historical data for both $NFLX and $AAPL has been fetched, the corresponding prediction models have been trained, and recent volumes required both to make predictions and to compute critical volume thresholds have been fed into the component.

The behavior of the integrated volume prediction component for the market players $NFLX and $AAPL are reported in Figure 29 and Figure 30 respectively, where 4 lines are plotted. The blue line is the aggregated volume observed within one minute. The green line is the volume predicted for the next minute. The red line is the threshold to identify critical volume values and it is computed as described in Section 2.7.5 (average of the past 10 volumes plus 3 times their standard deviation). The black line is the alarm value, measured as the difference between the volume observed at each time point and the one predicted for the next time point. The four values at each time point summarize the behavior of the component: the aggregated volume is observed (blue point); the volume within the next time unit is predicted (green point); the threshold for critical values is computed based on the recent volumes (red point); the predicted value is compared to the threshold and, if the former exceeds the latter, an alarm proportional to the difference is raised (black point).

The green, red, and black lines deserve few remarks. At a given time point t, the value of the green line represents the predicted volume for the *next* time point t+1. Therefore, to assess the accuracy of the prediction, one should compare the green point at time t with the blue point at time t+1. We preferred not to align the two time series (as it was done during the evaluation of the volume prediction in the D4.3 deliverable) because we wanted to make clear what is available and produced by the component at each time point. Again, in this test we are more interested in how the volume prediction behaves within the infrastructure than in its predictive accuracy, which have been already evaluated in the D4.3 deliverable.

Regarding the threshold values (red line), we used a window of 10 past volume values, but this size can be decreased or increased to make the average and standard deviation more or less sensitive to the most recent data. For instance, looking at the peak around t=13 in Figure 29, we can see that the increase in the average and, in particular, in the standard deviation due to the peak makes the threshold high and it stays so as long as the peak remains in the window of recent values. The factor (3 in our case) multiplying the standard deviation when computing the threshold affects the number of raised alarms: lower values bring more alarms, since the prediction exceeds the threshold more often, and vice versa.

Regarding the alarms (black line), every time the alarm value is greater than zero a `SourceVolumeAdaptationEvent` containing the name of the involved market player and the difference between the predicted volume and the threshold is sent to the Adaptation Layer. In case of more than one market players raising alarms, they are put together within the same `SourceVolumeAdaptationEvent` object.

From Figure 29 and Figure 30, we can see that the volume prediction component is able to predict most of the clear increases of volume and to promptly raise alarms to the Adaptation Layer, which decides whether and how to adapt to the imminent situation. Alarms are raised for the two biggest and clearest peaks in each figure, but also for some of the smaller ones, where the volume remains low in absolute but still significantly higher than its recent history. Besides raising alarms, another important aspect is raising them promptly, so that the Adaptation Layer has time to plan and enact any adaptation strategy to cope with the alarm situation. The worst case would be when alarms are raised in the last time point of the peak. This depends on how the observed volume evolves. When it increases relatively smoothly, e.g. during the last 5 time points of Figure 29 and Figure 30, alarms are raised when the volume is still relatively not critical (t=96 for Figure 29 and t=95 Figure 30). However, in case of spiky behaviors like the one in Figure 29 at t=13, there are only few time points (t=11,12) before the volume reaches the peak: the component raises an alarm at t=12 when the volume is already high (but still 1 minute before the peak).

5.4 Performance of SPASS-meter

As indicated in Section 2.7.2, we optimized the performance of SPASS-meter by identifying and solving issues observed in micro-benchmarking results. In this section, we report on performance experiments of the extended version of SPASS-meter for the QualiMaster project.

We extended SPASS-meter (Section 2.7.2) towards instance-level aggregation in order to capture the resource consumption of individual Storm tasks in multi-pipeline settings. Such an extension may easily impact the performance, in particular if frequently additional functionality is called such as for obtaining the identity hashcode of an instance.

The initial performance situation for the extension is characterized by the end of the optimization activities summarized in Section 2.7.2, i.e., by Figure 12. Figure 31 illustrates the results for the same benchmark on the extended version without monitoring the instance level. Except for minor fluctuations, the response time is roughly on the same level as in Figure 12.

Existing as well as specific test cases for the SPASS-meter extension show that the new version is working correctly. Figure 32 shows the performance results with the instance level monitoring enabled. Here SPASS-meter induces at a slightly higher average response time with more fluctuations due to obtaining the identity hashcode for each call of the benchmarking method and due to the additional online aggregation.

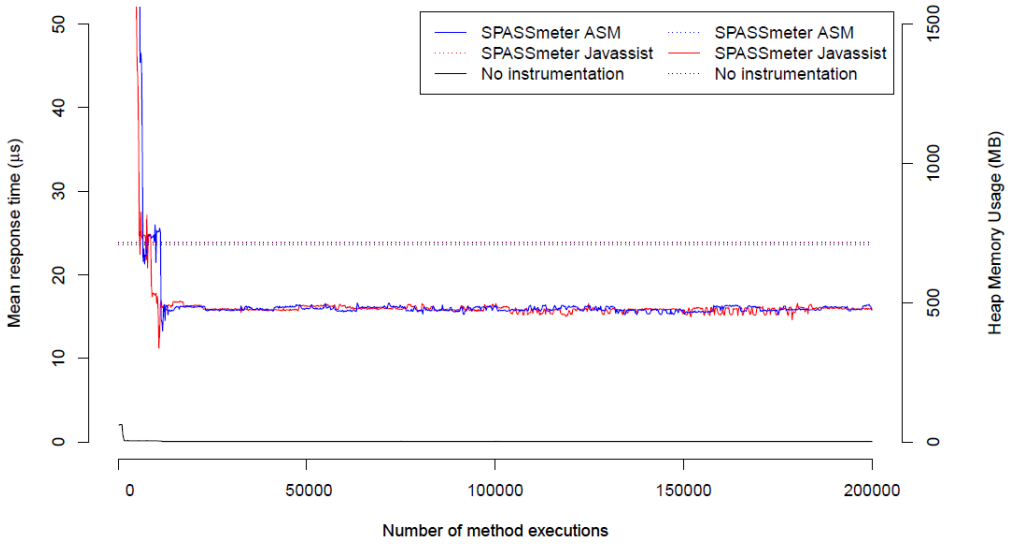

Figure 31: Benchmarking results for the extended SPASS-meter version without instance level monitoring. Dotted lines indicate memory consumption.

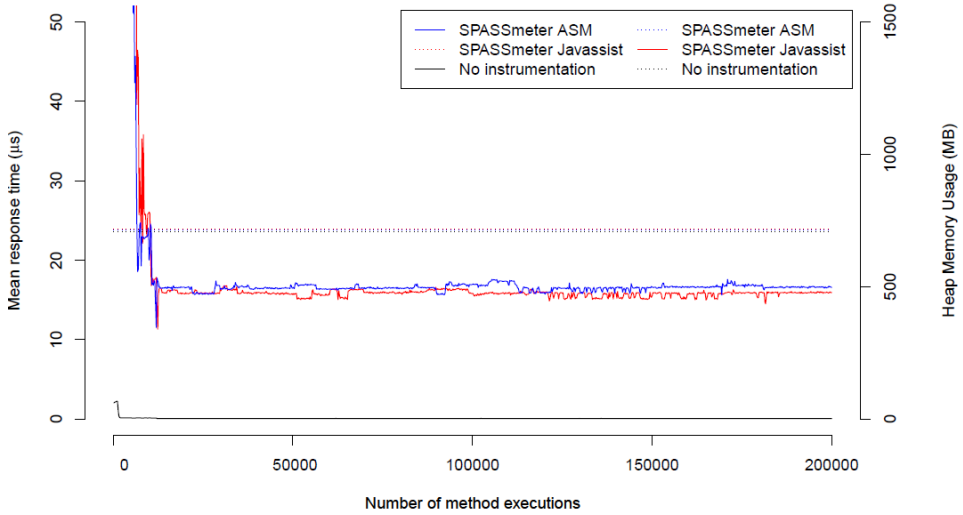

Figure 32: Performance results with instance level monitoring enabled.

Due to the memory problems on Storm clusters reported by our partners, we also experimented with SPASS-meter on a longer running pipeline. As pipeline, we used one of the fast-as-lightning pipelines described in D4.2/D5.3 and executed this pipeline for more than 24 hours without memory problems. For one hour of execution time, we determined the overhead for execution time monitoring for this pipeline to 0.136% (3140 tuples less are processed) and for execution time and memory monitoring to 2.7% (64420 tuples). Both values are improved over our initial application benchmarking against the SPECjvm2008 in [ES14], where execution time monitoring on a JVM 1.6 required 2.8% overhead and memory monitoring 3.9%.

6 Evaluation of Processing Elements and Pipelines

This section presents the evaluation of the new processing elements that can now be incorporated in the QualiMaster pipelines. We start with the time travel related elements. Next, we evaluate the hardware-based and the software-based implementations of two new modules in QualiMaster pipelines, i.e. transfer entropy (TE) and mutual information (MI). Finally, we report on the evaluation of the priority pipeline.

6.1 Time travel

In this section, we report the results of the evaluation we conducted for the “Graph decomposition & storage” processing elements, which will be incorporated in the Time-Travel SC-Graph Pipeline.

Dataset	Node Additions	Edge Additions	Edge Expirations	Indexed Operations
RD1	826	2917294	2671352	5589472
SD1	400	1417082	1376918	2794400
SD2	1600	5320469	4680531	10002600

The evaluation was performed using the three datasets shown in the above table. RD1 is composed of real data collected from our financial data source and it corresponds to one hour of execution. SD1 and SD2 are synthetic datasets that we created in order to evaluate the performance of the component to half (SD1) and double (SD2) workload of the real case (RD1).

The experiments were conducted on a single PC having an intel i7 processor with 8 cores and 8GB of RAM. The selection of this setting is based on the fact that the implementation of this module is still centralized (and single threaded) and therefore, running it locally is exactly the same as running it on a computer cluster.

Note that the particular component incorporates an index that allows efficient storage and retrieval of the data related to the correlations between the stock markets. We evaluated our component in terms of time per operation (vertex and edge additions, edge expirations) and time per snapshot query, as well as in terms of total memory consumption. However, since there is no accurate way of measuring memory consumption in Java, these results are just rough estimations of the actual memory usage.

The following table presents the experiment results for the three datasets.

Dataset	Total Execution time (sec)	Time Per Operation (ms)	Time / Snapshot Query (ms)	Total Memory Consumption Approx. (GB)
RD1	2604.161	0.466	1416	2.31
SD1	1574.773	0.564	475	1.54
SD2	2913.021	0.291	3005	3.24

As expected, the total execution time raises with the amount of indexed operations. However, as shown in Figure 33, we notice that as the total count of indexed operation raises, the average time per operation execution drops. We believe that this is related to the characteristics of each dataset but it certainly requires further research.

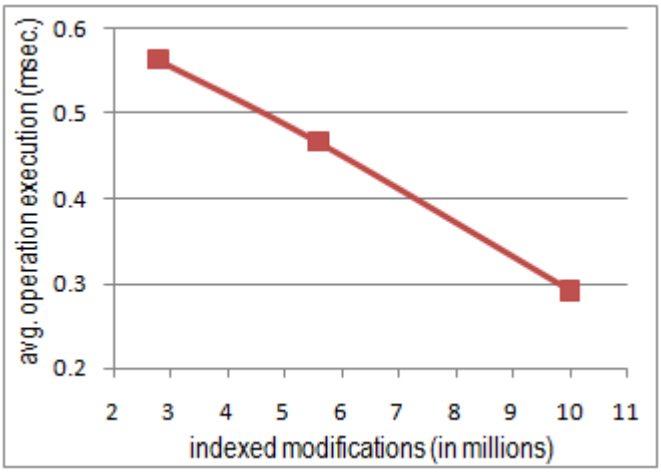

Figure 33: Average operation execution.

To measure the necessary time per snapshot query, we executed 100 queries at random timestamps for each dataset. The left plot in Figure 34 shows the average time for a snapshot query execution. When the amount of operations increases, the time for a snapshot query execution also increases since there are more operations to take into account.

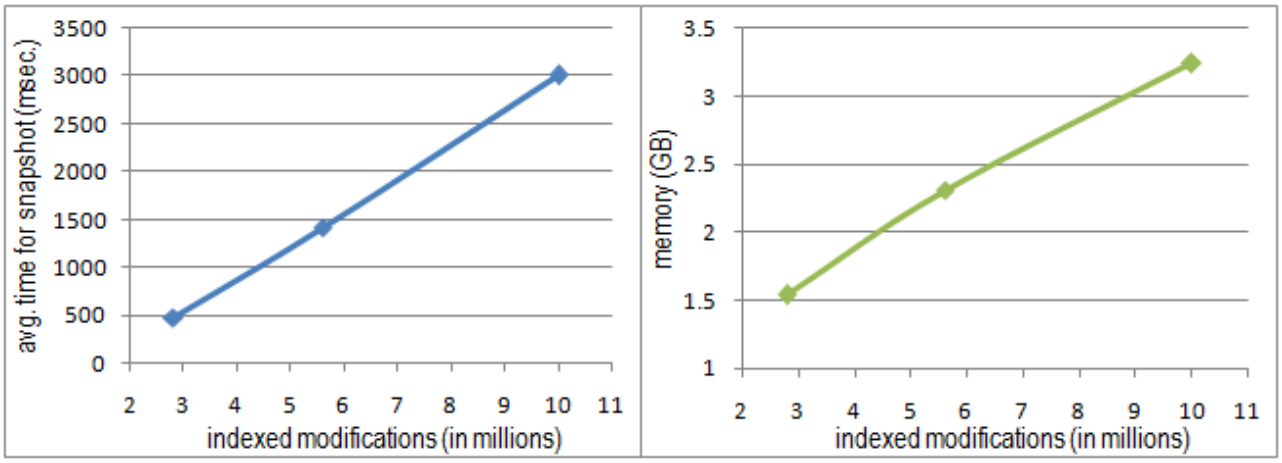

Figure 34: Average time for snapshot and memory.

Finally, the right plot in Figure 34 presents the approximate memory consumption for the execution of the three datasets. Since Java lacks mechanisms to measure the exact size of the objects, we approximated the memory consumption by monitoring the execution of the operations through the visualVM java profiler. As expected, the memory consumption increases as the amount of operations increases. This is expected because we index all operations (including deletions) in order to be able to reconstruct the graph snapshot at each time.

6.2 SW Transfer Entropy

We now report and discuss the evaluation results for the Transfer Entropy (TE) component, which will be incorporated in the Time-Travel SC-Graph Pipeline. Note that we refer to this as a component and not a processing element, since it has not yet been fully incorporated in the QualiMaster infrastructure. This is planned to be completed the following months and will be reported in the upcoming D2.4 deliverable.

As stated above, this component has not been incorporated in the QualiMaster infrastructure. We are currently working on improving the total required memory consumption since our experiments illustrated that it increases **linearly to** the number of market-player **pairs** to be monitored and not the CPU resources.

The current implementation is executed **single-threaded** and the experiments were conducted on a single machine with substantial memory (32Gb total memory, 28Gb JVM max heap size) with an Intel Xeon CPU @ 2.10GHz. When this component is incorporated in the QualiMaster infrastructure, the number of pairs to be monitored will be distributed among various cluster nodes. Thus, each node will hold only a subset of pairs, making it feasible to monitor them in real-time without requiring so much memory as the “experiments machine”.

The evaluation experiments conducted on this component use the datasets described in Section 0, namely the real dataset corresponding to one of the busiest hours of one of the busiest days during the 2-week period around the Brexit referendum (RD) and one synthetic dataset (SD), also 1-hour duration, with frequent updates for every market player. For the particular evaluation we considered a sliding window of 1 hour and a basic window (the sliding window is shifted every basic window and the results are calculated) of 1 minute. Moreover, the number of buckets used by the algorithm for PDF approximation was set to 100. The following paragraphs explain the measures and corresponding results.

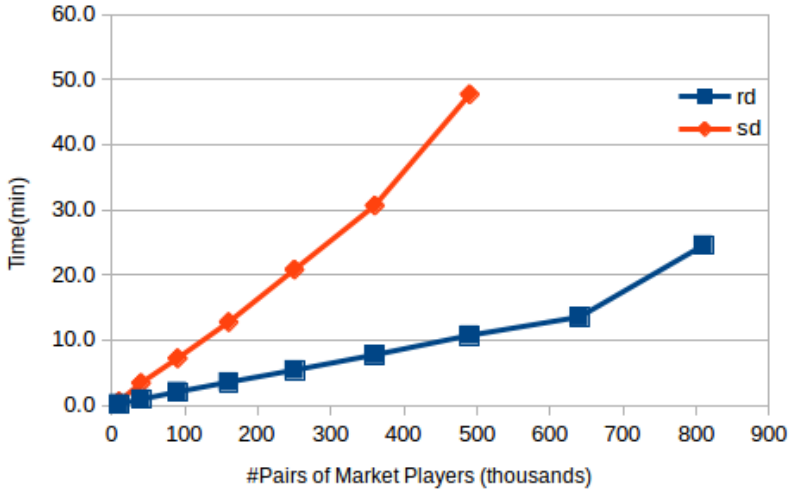

Figure 35: Time given pairs of market players.

A. Total execution time: This corresponds to the total time until the dataset is completely processed and our results are shown in Figure 35. For example, the RD dataset for 500k pairs of market players is fully processed in ~10 minutes. Since the RD and SD datasets have a duration of 1 hour, if the total execution time is less than 1 hour, then the implementation is able to process all incoming tuples in real-time. By increasing the number of marker player pairs, we can see how many pairs our implementation can handle until it is no longer considered real-time processing.

B. Throughput: This is the average number of ticks per second that the implementation can handle for increasing number of market player pairs. The throughput can also be seen as the average time it takes the system to process a single tick. Both plots are shown in Figure 36.

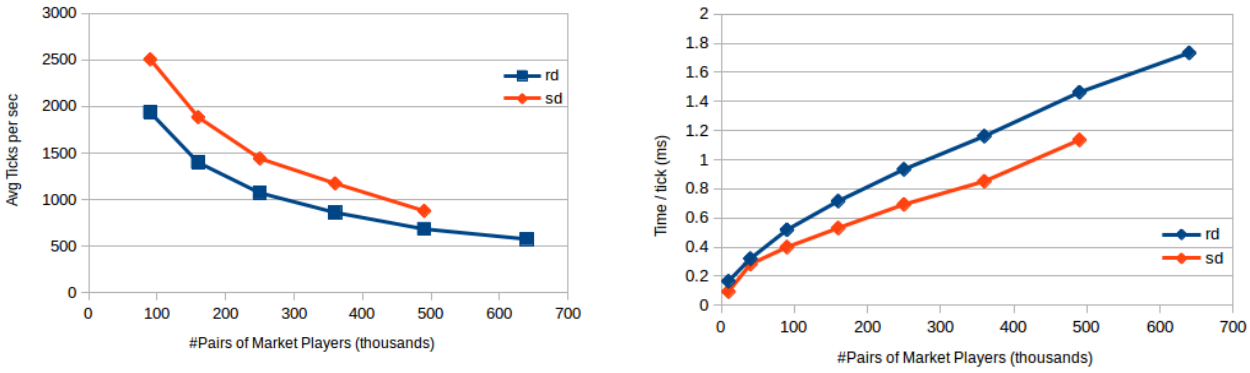

Figure 36: Average number of ticks per second and throughput.

These plots illustrate, as expected, that the total execution time scales with respect to the number of pairs to be monitored. The fact that the synthetic dataset, i.e., SD, takes more time for the same number of pairs as opposed to the real dataset, i.e., RD, lies in the fact that the synthetic dataset is much more dense and has new ticks for all the market players every second, whereas in the real dataset, there are many “slow” market players that do not have frequent updates (new ticks), so even if the number of pairs is the same, the number of ticks (not shown in the chart) differs.

As for the throughput, again as expected, we can see that it drops as the number of pairs grows. The algorithm needs to calculate the TE value for each pair of market players, so the more pairs, the more TE calculations, whereas if the number of pairs is fixed, but the arrival rate is higher, the number TE calculations stays the same, but each calculation may take more time to complete.

In terms of overall performance, it is safe to assume that this component can reliably handle ~250k pairs of market players in real-time when being executed on a conventional cluster node. Hence, the worst (real) case (1700k pairs during the Brexit dataset) could be handled by ~7-8 cluster nodes.

6.3 HW Transfer Entropy

In this section we provide the performance results for the Transfer Entropy (TE) implementation on hardware. The number of histogram bins was set to 100 for these experiments. The results presented here are from the algorithm running on hardware servers from the Maxeler partner. Note

that we created two implementations, one for the VECTIS dataflow engine and another implementation for the MAIA dataflow engine. We evaluated each implementation with 1 and with 4 dataflow engines (DFEs). The second and final architectures for Transfer Entropy were also presented in the D3.3 deliverable.

The following figures provide the results for 1 hour dataset during the two weeks near the Brexit referendum. We also tested a synthetic dataset, also 1 hour duration, with frequent updates for every market player, actually one update every second. The performance results remain the same between the synthetic and real dataset. Actually in our case we simulate the synthetic dataset by filling the missing values from the real dataset while executing. The performance of the hardware implementation does not depend on the dataset. The sliding-window size was set to 30 minutes while the slide offset was set to 1 minute just like in the MI case. This means that all the TE values between the market players are calculated every 60 seconds. The first figure shows the execution time needed in order to run the whole dataset. The execution time per experiment, especially for large number of market players is prohibitive. After we completely executed the 100 market players case and several minutes for the rest experiments, we calculated the required execution time per TE calculation and concluded that it remains consistent regardless the number of market players or their values overtime, to overcome the efficiencyissue. For each case the TE calls required are N(N-1), where N is the number of market players. The highest number of calls required by the 1265 market player case is about 1600 thousand calls. Unfortunately only the 100 market player case can be supported in the required time but only by the 4DFE final architecture on the MAIA platform. The 4DFE MAIA results are projections as we did not have a MAIA platform with 4 DFEs in our disposal.

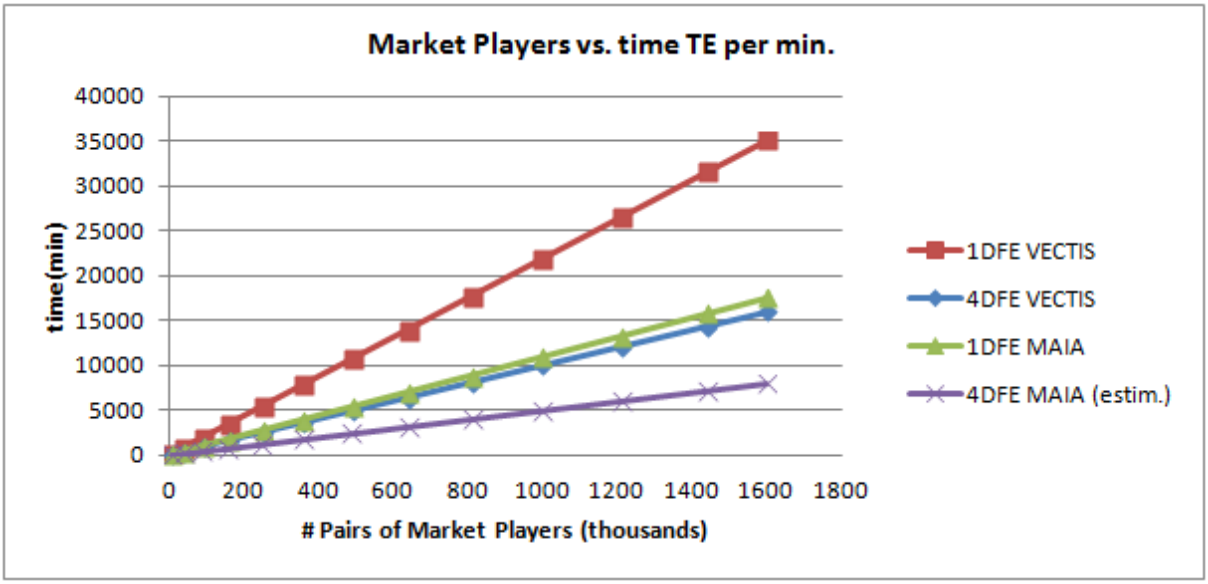

Figure 37: Transfer Entropy calculation every minute for 1 hour dataset.

The second figure presents the execution time for the sliding-window size being 1 hour, as well as the slide offset. This means that 1 TE calculation per market player pair is executed every hour, instead of 60 (Figure 37). In this case the hardware implementation can support up to 400 market players with 1DFE on the VECTIS, while the 4DFE implementation can support 600 market

players. The final architecture that is executed on MAIA can support up to 500 market players and the estimation for the 4DFE MAIA implementation supports 800 of the market players that included on the dataset.

As far as the memory required for our 4DFE implementation, it consists of about 4MB per thread/DFE for the pdfs (16 MB for 4DFE implementation) as well as the dataset (15KB per market player for 1 hour). We store each value per timestep for each market player in a buffer. The worst case is that all market players are updated every second. The memory requirement for this case is about 18MB for the 1265 market players case and 1 hour dataset. The actual processing capability of the implementation of TE on hardware is about 6000 TE calculations per minute on the 4DFE VECTIS for 100 histogram bins. We do not report the MAIA results as we do not have actual runtimes yet from that platform, but it is expected to be 2x faster.

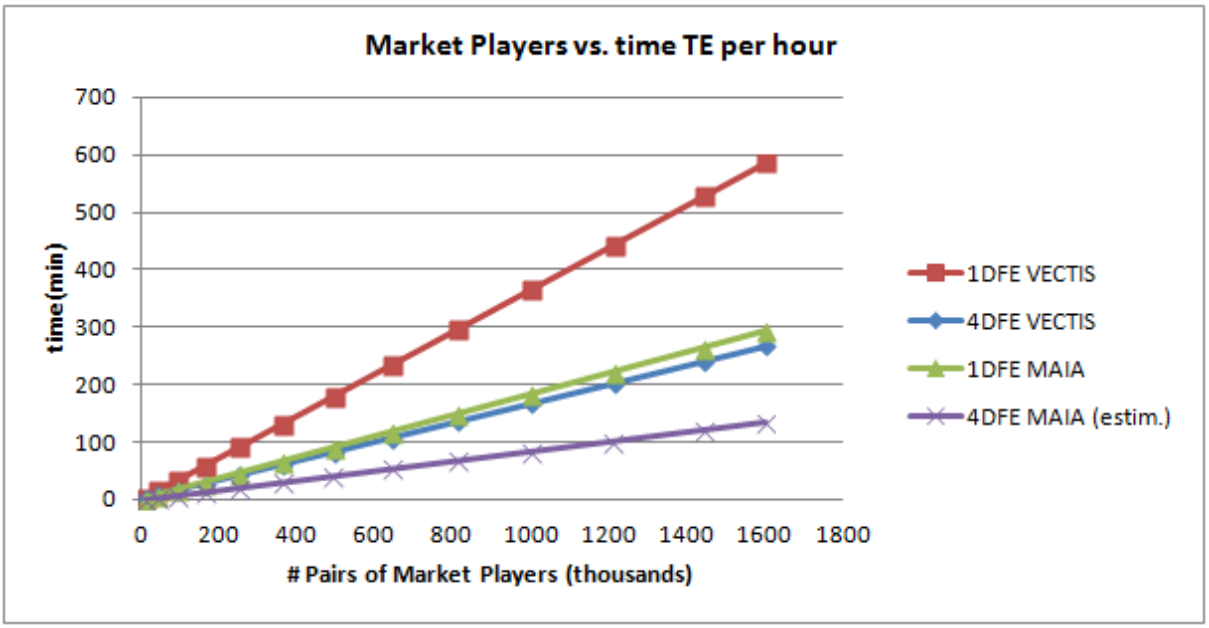

Figure 38: Transfer Entropy calculation for 1 hour dataset.

The advantages of this implementation are that it can support a very large number of market players, due to the low memory requirements of the implementation, and that the execution time is constant and does not depend on the properties of the dataset. As such, we can claim that given the calculation time presented on Figure 38 for the respective number of market players, the hardware implementation can support any dataset that uses slide window size smaller than the respective execution time.

6.4 SW Mutual Information

We now report and discuss the evaluation results for the Mutual Information (MI) component, which is also going to be incorporated in the QualiMaster infrastructure. The experimental setup follows the one of the Transfer Entropy component, so it is not reported again. Note that this implementation of Mutual Information is the streaming version reported in D2.3 and is inherently different than the corresponding Hardware implementation, thus not directly comparable.

The key difference of this implementation, is that it does not use the sliding window/basic window approach used in TE_SW / MI_HW / TE_HW which computes the results every basic window (e.g., every minute). This approach computes MI results for every new tick of data for all the pairs of market players that involve the newly updated market player. For example, when there is a new update regarding the price of Google's stock, all the pairs of market players involving Google are updated and the result is **immediately** reported to the stakeholder applications, without waiting for a basic window expiration. This approach creates a trade-off between fast update times and quality of results versus TE (TE provides a better quality regarding the results as it also provides directivity, but its computation is much "heavier").

Below are presented corresponding plot as the ones in Section 5.2. Please note that the number of market player pairs that is required to be computed by this algorithm is actually half (N*(N-1)/2 to be exact) than the number of market players that needs to be computed by the Transfer Entropy component, due to directionality (MI is bi-directional, TE is directional). Yet, we have kept the number of pairs un-altered in the horizontal axis so that they correspond with the description of the respective data set. The number of buckets used by the algorithm for PDF approximation was set to 1000 for this component.

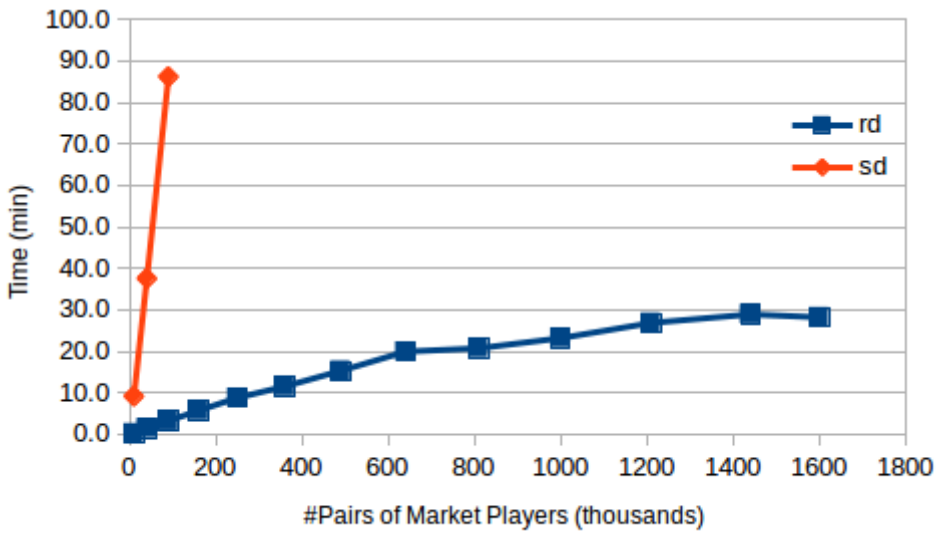

Figure 39: Total execution time.

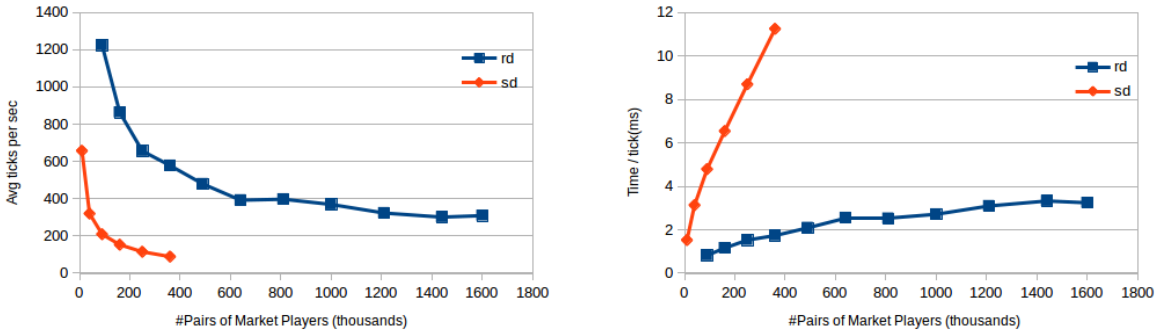

Figure 40: Throughput.

The plots of Figure 39 and Figure 40 illustrate that this component is **very efficient** when it comes to **real** datasets (notice that is processes the full 1-hour Brexit dataset of ~1600k pairs in less than 30 minutes in single-threaded execution), while being **very inefficient** when it comes to **synthetic** datasets with frequent updates for all the market players (it needed 86 minutes to process 90k pairs of 1-hour data).

6.5 HW Mutual Information

In this section we provide the performance results for the Mutual Information (MI) implementation on hardware. The number of histogram bins was set to 1000 for these experiments. The results presented here are from the algorithm running on hardware servers from the Maxeler partner. Note that we created two implementations, one for the VECTIS dataflow engine and another implementation for the MAIA dataflow engine. We evaluated each implementation with 1 and with 4 dataflow engines (DFEs). The second and final architectures for Mutual Information were presented in the D3.3 deliverable.

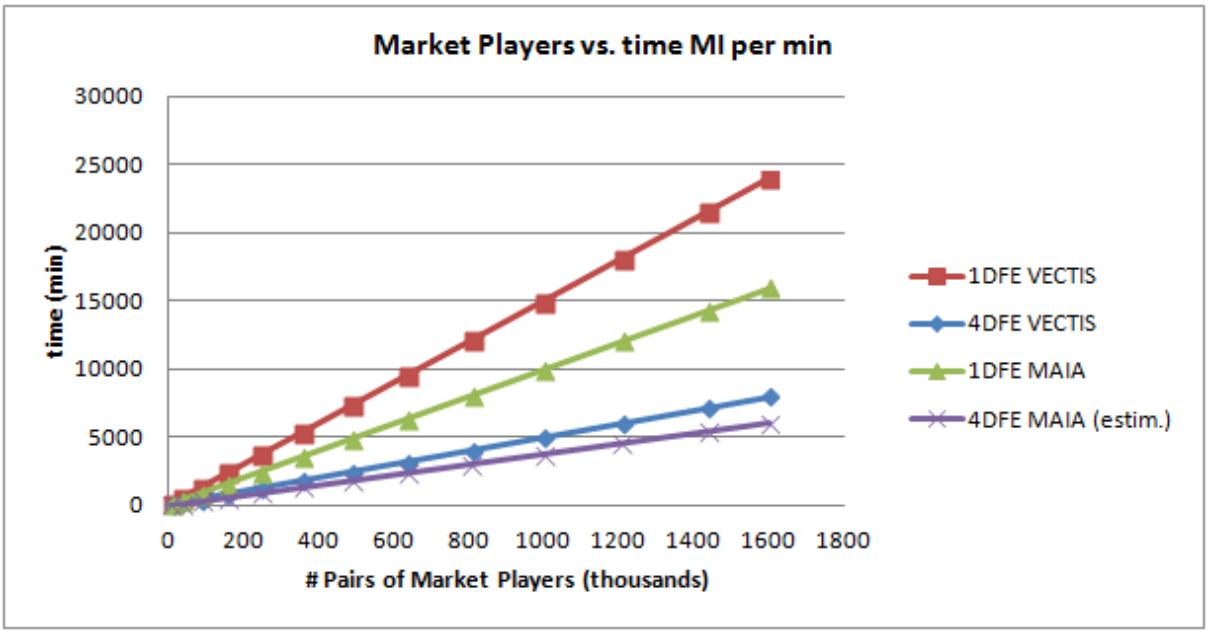

Figure 41: Mutual Information calculation every minute for 1 hour dataset.

The results are for a dataset of 1 hour during the two weeks near the Brexit referendum. We also tested a synthetic dataset, also 1 hour duration, with frequent updates for every market player. The performance results remain the same between the synthetic and real dataset. Actually in the hardware server we simulate the synthetic dataset by filling the missing values, every second for each market player, from the real dataset while executing. The sliding-window size was set to 30 minutes while the slide offset was set to 1 minute. This means that all the MI between the market players are calculated every 60 seconds. Figure 41 shows the execution time needed in order to run the whole dataset. As the execution time per experiment, especially for large number of market players was very large, we completely executed the 100 market players case and several minutes of the rest experiments. The calculation time per MI call was consistent in every case and thus we projected the expected total execution times. For each case the MI calls required are N(N-1)/2, where N is the number of market players. The highest number of calls is on 1265 market players which requires about 800 thousand MI calls. For completion and symmetry with the dataset, as well as Transfer Entropy, the following figures include the processing time for N(N-1) pairs. Unfortunately only the 100 market player case (4500 calls) can be supported in the required time both by the 4DFE VECTIS and the MAIA implementations. The rest market players' cases cannot be supported by our hardware implementation. The 4DFE MAIA results are projections as we did not have a MAIA platform with 4DFEs in our disposal.

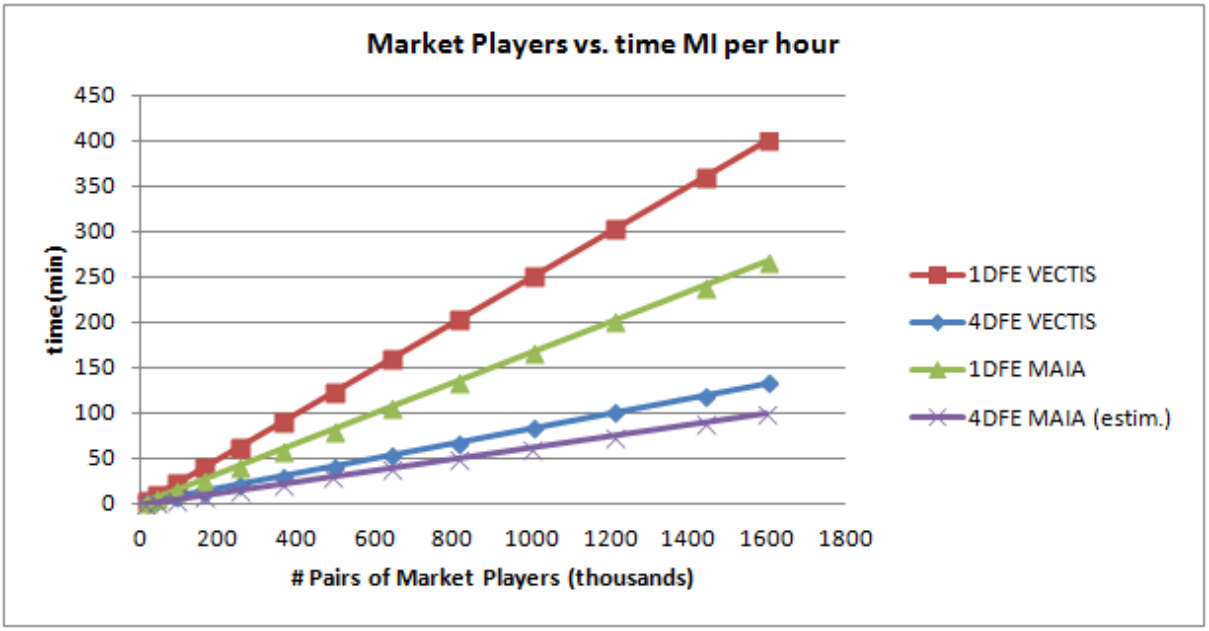

Figure 42: Mutual Information calculation for 1 hour dataset.

Figure 42 presents the execution time for the sliding-window size being 1 hour, as well as the slide size. This means that 1 MI calculation per market player pair is executed every hour, instead of 60 in the previous case. In this case the hardware implementation can support up to 700 market players with 1DFE on the VECTIS, while the 4DFE implementation can support 1200 market players. The final architecture that is executed on MAIA can support up to 800 market players and the estimation for the 4DFE MAIA implementation supports all the 1265 market players that are included on the dataset.

As far as the memory required for our 4 DFE implementation it consists of 4MB per thread/DFE for the pdfs (16 MB for 4DFE implementation) as well as the dataset (15KB per market player for 1 hour). We store each value per time step for each market player in a buffer. The worst case is that all market players are updated every second. The memory requirement for this case is about 18MB for the 1265 market players case and 1 hour dataset.

The advantages of this implementation are that it can support a very large number of market players and that the execution time is constant and does not depend on the properties of the dataset. As such we can claim that given the calculation time presented on Figure 42 for the respective number of market players, the hardware implementation can support any dataset that uses slide window size smaller than the respective execution time.

The actual processing capability of the implementation of MI on hardware is 12000 MI calculations per minute on the 4DFE VECTIS for 1000 histogram bins. We expect about 2x increase on this performance from the final architecture on the 4DFE MAIA platform.

6.6 Priority Pipeline

This section analyzes the performance of the priority pipeline, where the correlation of streaming financial data is calculated at each second. The correlations are calculated based on the transactions of stock markets, which are streamed at each tick. The following paragraphs present the performance of the priority pipeline using the previous version of the QualiMaster infrastructure and the performance advantages of the new proposed solution for various data collections. The experiments were carried out on the Mikio Cluster, the smallest cluster used in the project, described in the D5.3 deliverable.

Market players	Ticks per sec	Window size (sec)	Cluster Nodes	Number of correlations	Processing with SW	Processing with HW
100	100	30	9	4950	X	X
250	250	30	9	31125	X	X
500	500	30	9	124750	X	X
750	750	30	9	280875	X	X
1000	1000	30	9	499500		X

Table 3: The results of the priority pipeline when computing the correlations over the data sets with increasing market players and transactions per second using SW and HW implementations.

6.6.1 Performance of previous priority pipeline

As presented in the D5.3 deliverable, the QualiMaster infrastructure can process data sets with different characteristics, i.e. number of stock markets, number of transactions per tick etc. Specifically, the QualiMaster pipeline infrastructure managed to process within the defined time frame (i.e., within one second) datasets with up to 1000 market players. Thus, the QualiMaster platform can process up to 1000 transactions per second and transfer 499500 results to the sink within the time frame of 1 sec. Table 3 presents the evaluation datasets and the performance

results when datasets with increasing number of stock markets and ticks per second were tested on the previous version of the QualiMaster pipeline. Given the increase in the number of the market players, table shows the increase in the number of correlations, too.

6.6.2 Performance evaluation of priority pipeline

This section presents the performance evaluation of the priority pipeline using the new version of the correlation calculation modules. One of the factors that restricted the number of the market players that the priority pipeline could process in the time frame of 1 second was the transmission rate of the produced correlation values. Specifically, the number of correlations follows square increase vs. the increase of the number of stock market players. Thus, we moved to a more generic and multithreaded solution. Specifically, we created for hardware implementation multiple transmission links between the hardware platform and the Storm-based components that receive the correlation results. This solution contributed to increase the transmission rate between the QualiMaster platform and the end-user.

Table 4 shows the various datasets that were used for testing the priority pipeline. As the results indicate, the QualiMaster platform, as a combination of hardware and software implementations, can process up to almost 1750 market player with 1750 transactions per sec. In more details, the QualiMaster platform can return up to 1530375 correlations when multiple links between hardware and Storm-based modules are used. However, we see that the implemented platform does not reach the goal of 2000 market players with 2000 ticks per second. Specifically, 85% of the required 1999000 correlations are not delivered within the first second but in the next second. This delay will propagate to the consequent seconds (in case the same ticks per second rate continues), contributing to a non-acceptable delay for the end-user.

Market players	Ticks per sec	Window size (sec)	Cluster Nodes	Number of correlations	Processing with SW	Processing with HW
250	250	30	9	31125	X	X
500	500	30	9	124750	X	X
750	750	30	9	280875	X	X
1000	1000	30	9	499500		X
1250	1250	30	9	780625		X
1500	1500	30	9	1124250		X
1750	1750	30	9	1530375		X
2000	2000	30	9	1999000		(X)

Table 4: The results of the priority pipeline when computing the correlations over the data sets of the increasing market players and transactions per sec collection using SW and HW implementations.

Figure 43 shows a graphical illustration of the SW and HW performance for the particular collection. Given the QualiMaster infrastructure, we can easily see a possible switching from the SW to the HW implementation, when the number of players is somewhere between 800 and 1000.

In addition, we can see the processing capability of the QualiMaster infrastructure as an overall system for the worst case dataset scenarios and when a small cluster infrastructure was used. It is important to mention that the architecture of the proposed system is scalable and it can be expanded/mapped to bigger cluster infrastructures with more nodes and multiple hardware platforms offering even better performance results. The actual performance is subject to final evaluations.

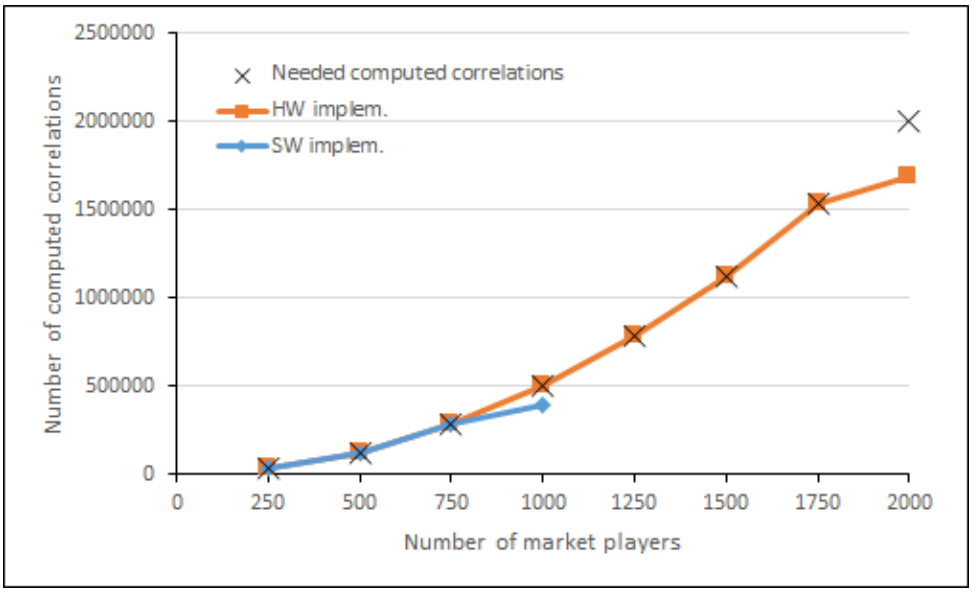

Figure 43: The number of correlations that need to be computed and they are transferred using software and hardware implementations for increasing market players and transactions per sec. A possible switching from the SW to the HW implementation is when the number of players is somewhere between 800 and 1000.

7 Conclusions

We have reported the results of the work performed by all project partners related the QualiMaster infrastructure. The preceding WP5 deliverables have described our infrastructure in terms of its architecture, design, layers, and execution environment (i.e., D5.1, D5.2 and D5.3). We thus now provided only a recap of the final architecture, components and layers, followed by a detailed description of the infrastructure parts that have been recently improved or developed and were not reported in a previous deliverable.

The deliverable also reported our new, extended experimental evaluation. With respect to this, we first provided the settings and evaluation methodology as well as the collection of used data sets, which included synthetic and real financial data. We then described the performed evaluations and reported the results. The conducted evaluations covered crucial components of the infrastructure, the newly developed processing elements, and the improved version of the priority pipeline. Note that we plan to perform additional evaluations, as also stated in the description of work, and report on the results in the final deliverables of WP2-WP4, i.e., D2.4, D3.4, and D4.4.

During the last three months of the project, we plan to finalize the integration and development of the QualiMaster infrastructure and its population (i.e., processing elements and pipelines). More specifically, we aim at performing the last integration of the components and processing elements that were recently developed by the technical work-packages, i.e., WP2, WP3, and WP4. Afterwards, we plan to further test and verify the overall functionalities and mechanisms. Our final task is evaluating the processing elements as well as processing elements. The results of those evaluations will be the D2.4, D3.4, and D4.4 deliverables.

8 References

[BTÖ13] C. Balkesen, N. Tatbul, M. Özsu. Adaptive Input Admission and Management for Parallel Stream Processing. In Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS), pages 15-26, 2013.

[CJ09] S. Chakravarthy and Q. Jiang. Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing. Springer, 1st edition, 2009.

[CKC08] Charles L.A. Clarke, Maheedhar Kolla, Gordon V. Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In Proceedings of the 31st annual international Conference on Research and development in information retrieval (SIGIR), pages 659-666, 2008.

[ES14] H. Eichelberger K. Schmid. Flexible resource monitoring of Java programs. Journal of Systems and Software, pages 93:163–186, 2014.

[Montgomery97] D. Montgomery. Introduction to statistical quality control. Wiley, 3rd edition, 1997.

[QE16] C. Qin, H. Eichelberger. Impact-minimizing runtime switching of distributed stream processing algorithms. In EDBT/ICDS Workshop on Big Data Processing - Reloaded, 2016. http://ceur-ws.org/Vol-1558/

[Sass16] A. Sass. Performanz-Optimierung des Resourcen-Messwerkzeuges SPASS-meter (Performance optimization of the resource monitoring tool SPASS-meter). MSc thesis, University of Hildesheim, 2016 (in German).

[Waller14] J. Waller. Performance Benchmarking of Application Monitoring Frameworks. PhD thesis, University of Kiel, 2014.