Parallel processing of multimedia streams

Henryk Krawczyk, Karol Bańczyk, Jerzy Proficz
Gdańsk University of Technology
80-233 Gdańsk, ul. Narutowicza 11/12, e-mail: hkrawk@pg.gda.pl,
kbanczyk@task.gda.pl, jerp@task.gda.pl

The paper presents a new multimedia processing platform: KASKADA. The design of the platform is described: the diagram of main classes, the sequence diagram illustrating their cooperation during the processing of multimedia streams, and the details of the inter-task communication mechanism. We also present the framework supporting algorithm development, a service scenario definition, and provide evaluation of the platform usability.

1. Introduction

Multimedia systems play an important role in industrial and global development. There is a need for high performance computers to process very complex algorithms, e.g., face recognition or registration (number) plate localization. The paper presents a new computational framework which is a part of Context Analysis of the Camera Data Streams for Alert Defining Applications platform (Polish abbreviation: KASKADA, i.e., cascade,l).

The platform is going to be deployed on a high-performance computational cluster 'Galera' within the Academic Computer Centre in Gdańsk – TASK [6], consisting of 672 nodes, 1344 processors and 5376 cores. The nodes are connected by 20Gbps Infiniband [9] using the fat tree technology, supporting a fast, 5000TB LUSTRE [13] file system.

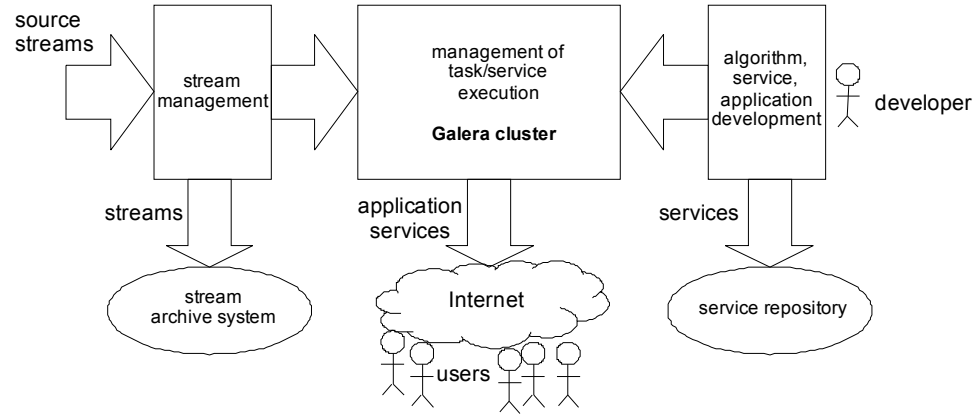

Fig. 1. The multimedia stream analysis schema for cluster computing environment

The Fig. 1 presents the general schema of processing the multimedia streams in a computation cluster environment. The processed streams need to be received and initially pre-processed, archived in a dedicated subsystem, repacked and distributed to the cluster, where the actual analysis is performed.

The analysis algorithms are designed, programmed by the developers and should be exposed to the external users as application services accessed remotely via the user interface or an additional application using remote calls from the service repository. The above characteristics require the following core features to be provided by the system and hardware components:

- Multimedia stream distribution – the incoming data including video, audio and associated metadata should be provided to each task performing the analysis, assuming they can be scattered through the cluster nodes, high-speed networks and proper communication protocols supporting streaming need to be used. Moreover, the archiving mechanisms must be deployed for the off-line processing.
- Distributed computing – the complexity of the analysis and scalability requirement enforces its decomposition into cooperating tasks, which need to be assigned and launched on the proper cluster nodes, the efficiency of this procedure is especially important for the real-time analysis of the on-line streams.
- Quality of service – the on-line analysis requires the quality constraints to be applied at the beginning and maintained for the duration of the computation, which implies introduction of continuous cluster monitoring mechanisms controlling processor, memory, and network load, of executed tasks.
- Remote access – the general user interface needs to be provided, including access to the archive, streams, and running algorithms. Similar functionality should be provided for the client application. For both types of access the proper security policies must be applied.
- Development environment – the multimedia processing programs require proper sets of the core functions supporting their execution including stream decoding, encoding, forwarding, manipulation, meta-data gathering and the input device control, e.g. camera movement and zoom adjustment.

Multimedia stream distribution can be realized using normal TCP/IP connections; however, for the unpacked data, especially video, even fast Infiniband TCP/IP setups can be easily over-saturated. A typical solution is usage of low level RDMA [4] libraries, which are often utilized by the MPI [16] implementations. Similarly, for the stream archiving, the fast remote file systems over Infiniband are used, in this case of the 'Galera' cluster it's LUSTRE [13].

Distributed computing is a key feature provided by any cluster. The usual realization is based on a queue system, enabling processor scheduling for a large number of tasks. The 'Galera' uses PBS [18] with a specific plugged scheduler: Maui [15]. The process of resource assignment is performed every minute, and

uses declared processor/memory load as an indicator of the used resources, which can be inefficient for real-time online analysis requirements.

The typical computing cluster does not usually provide any out-of-the-box quality-of-service mechanisms, some of them can be realized using special priority settings over the queue system and directly in the operating system. In our case we deployed a PBS QoS module [18] supported by a Linux quota and *nice* command policies.

The typical Linux cluster is accessible through a remote shell, usually SSH [20]. There is one special node – the access point, which is used for starting the tasks over the whole cluster. It manages the whole queue system including the scheduler process, and further uses SSH connections for task distribution over the computation nodes.

The typical configuration of the cluster contains the set of development tools including compilers (C++, Forthran, Java etc.), message passing libraries (usually one or more MPI implementations) and some debugging and monitoring. The multimedia processing requires, at least, installation of a proper decoder/encoder and stream transportation libraries.

In the KASKADA platform we provide a unified collection of the tools and components extending and partially replacing the (above) typical cluster solutions. The platform contains the common web interface for its management including algorithm and service repositories, on-line and archived stream control and external user/client administration.

The multimedia stream distribution is realized using fast RDMA functions, which utilise (directly) fast Infiniband networks. All multimedia processing algorithms are implemented using the same set of functions for decoding/encoding and forwarding the streams, with transparent conversion between online and archived data.

The platform provides its own way to distribute the tasks over the cluster. The proposed solution uses its own assignment algorithms and enables fast start-up of scheduled scenarios without additional overheads caused by SSH [20] protocol. The proper monitoring processes are deployed on each cluster node, they are also responsible for quality measurements and resource controlling.

Remote access is provided by the platform through the web user interface. The applications can execute and control the multimedia algorithms using typical SOA [10] approaches with the utilization of the SOAP [24] webservices. The index of the provided services is also accessible via user interface or using the UDDI [22] registry.

The platform provides sets of libraries enclosed within the framework, enabling easy development of multimedia processing algorithms in C++ language. There is provided also a unified procedure of the stream processing including decoding/encoding and metadata handling. Moreover, additional features like test streams, event monitor and service composer are provided.

The KASKADA platform provides a complete solution for multimedia processing application development: from algorithm construction, through its

implementation as a computational task to the exposed set of dedicated services, which can be used to solve more complex problems (e.g. person tracking), including service execution and test [3].

Each service consists of a set of cooperating tasks, described by a directed acyclic graph, where the vertices represent the tasks, and the edges indicate the data flows between them. The platform can manage multiple task graphs and return their computational results as outputs of the corresponding services. Additionally, extra tasks can be assigned for quality evaluation of the algorithms, including such factors as performance, effectiveness and scalability.

To implement a task, we need a C++ developed algorithm based on the framework library and headers. The library supports two core functionalities: audio-video stream decoding/encoding and mechanisms for inter-task communication. The latter are: object serialization, incoming data synchronization, signal handling and queue-based messaging.

The next section describes the framework design, including class and sequence diagrams. Section three contains information about task cooperation mechanisms and introduces a task graph example. In the final section, we provide conclusions, current state of development, and indicate further work to be done.

2. Framework for stream analysis

Multimedia stream processing algorithms may perform quite different types of analysis. The class of algorithms is very broad. To name just a few examples, the algorithms may perform human face recognition, voice recognition, object tracking, car registration plate detection, etc. Even though the specifics of each such algorithm are different, they all are typically built according to the same template, and they have to perform a lot of common tasks. Moreover, algorithms launched in a common environment, like KASKADA platform, need to behave according to certain rules in order to improve the whole ecosystem's manageability, stability, and so on. Also it's convenient for an algorithm developer to have the most typical and mundane tasks of a general nature solved for him, to let him concentrate on the parts that are original to his work.

KASKADA framework is a C++ library addressing these issues. It provides algorithm classes for typical algorithm types, audio/video stream decoding and encoding, C++ object serialization and inter-algorithm delivery, rtsp/file mutimedia streams handling, dynamic tasks launching and basic life-cycle management support. The framework is provided as a static Linux library with necessary headers. Additionally, the implemented algorithms can use libraries used by KASKADA which include e.g. a rich set of Boost [1] libraries and standard POSIX system features (parallelism mechanisms, including thread creation and cooperation). The user may, of course, use other algorithm needed libraries.

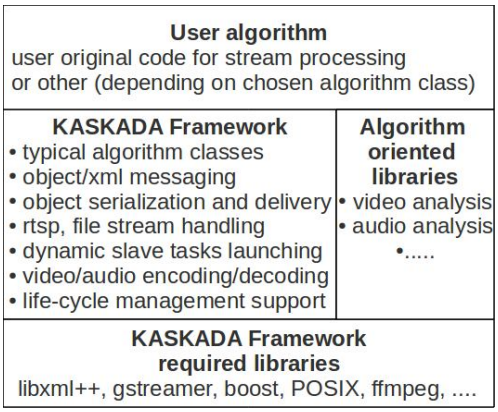

Fig. 2. The layer diagram of KASKADA framework based algorithm

The Fig. 2 shows a layer diagram of a kaskada-based algorithm. The user algorithm is always implemented as a subclass of a KASKADA framework provided abstract algorithm class. The type of user code depends on the used algorithm class. The available algorithm abstract classes are depicted in Fig. 3.

The *KaskadaAlgorithm* class is the most general algorithm available. It provides all basic features required from KASKADA supervised algorithms, which are related to algorithm life-cycle control. The class provides features related to input parameters parsing and XML based communication.

The *KaskadaMasterAlgorithm* and *KaskadaSlaveAlgorithm* are abstract classes for algorithms working in master-slave computational model. The master algorithm class provides methods for slave tasks starting, input/output data exchange and synchronization.

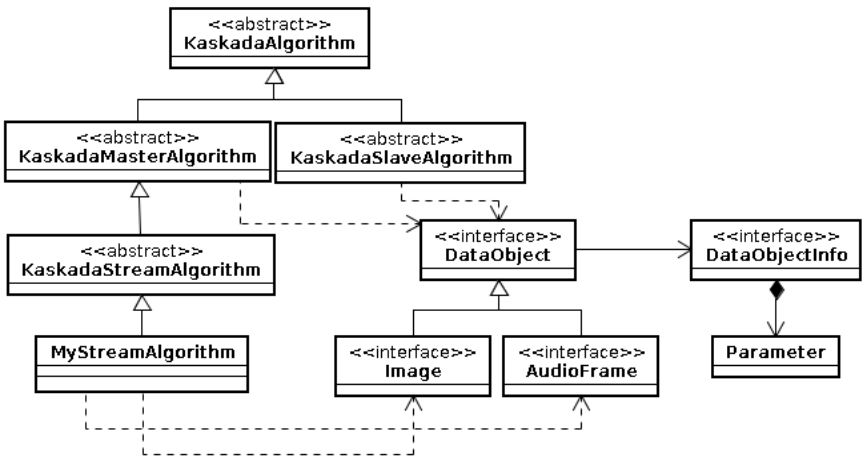

Fig. 3. The class diagram of the KASKADA framework

The *KaskadaStreamAlgorithm* class is designed for on-line streams processing, where a stream is defined as a sequence of data packets available in a packet-by-packet manner. A stream algorithm can process more than one input stream and produce more than one output stream. The output streams' limitation is that all the streams must have the same content (not necessary the same protocol). The class provides an abstract callback function *processDataObjects(),* used as a template method pattern [7] and provides the streams' objects to the concrete algorithm implementation, e.g. MyStreamAlgorithm in Fig. 1. The method's input is a vector of objects received from input streams. Each vector position maps to an input stream number and contains the last object received from the stream. The vector is passed to the algorithm for every new object in the first stream, being the synchronization stream.

Fig. 4. Stream types handled by stream algorithm

The basic KASKADA Framework's communication protocol is *kbin* (standing for KASKADA Binary protocol) and is simply a network stream of serialized objects. The serialization is implemented using Boost Serialization library [2]. However, the stream algorithm class provides special functionality related to multimedia streams. It accepts and produces *rtsp* [19] and *file* multimedia streams, taking care of all the necessary decoding and encoding. The decision on used protocol types is a matter of launch configuration. Streaming capabilities of an algorithm are depicted in Fig. 4.

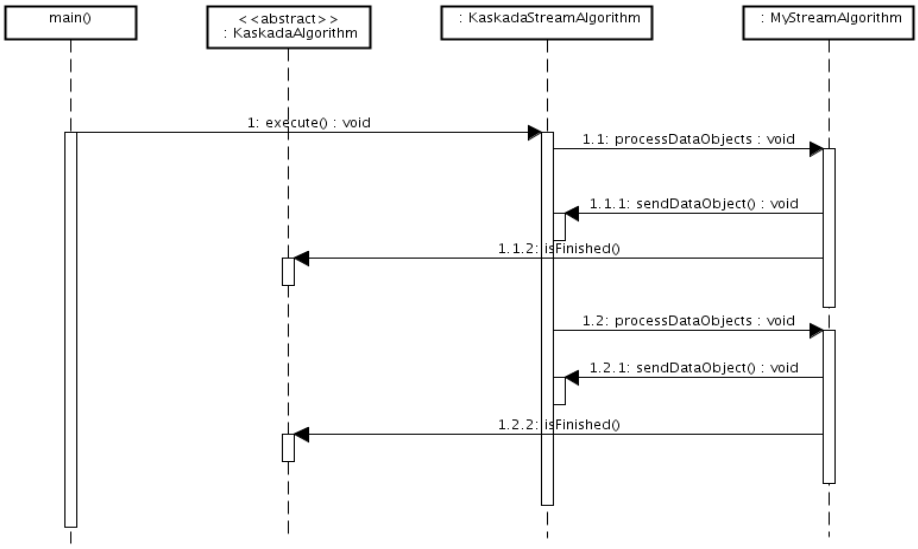

Fig. 5. The sequence diagram of the KASKADA framework stream algorithm

The framework provides classes related to the processed data. The DataObject class represents an interface of an abstract data entity. It is associated with the DataObjectInfo class containing metadata as a set of parameters. The DataObject is extended by the Image and AudioFrame classes, responsible for holding video frame and sound frame respectively.

The Fig. 5 presents typical usage of the framework. The main() function of the task creates an algorithm instance and calls the execute() method, where the main loop is invoked. During the execution, the loop provides the processed data by chunks, for video there are image frames and collected samples for audio.

In the implemented template method *processDataObject()*, the algorithm developer can use the *sendDataObject()* method for sending the processed data to the output streams, check the input parameters using the *getArgs()* method, or check if the task received a signal from the platform management module to finish its work using *isFinished()*.

Each stream algorithm is also a master algorithm. Fig. 6 shows a scenario with an exemplary master task starting new slave tasks (*startTasks()*) and waiting for their termination. The process of starting the slaves involves communication with the management platform which launches new tasks according to available resources. The master task can suspend its activity until an unspecified slave (*waitForAnyTask()*) or all of slaves (*waitForAllTasks()*) of a set have finished. In the diagram the master algorithm is represented by the myMasterAlg class and the slave algorithm by the mySlaveAlgorithm class.

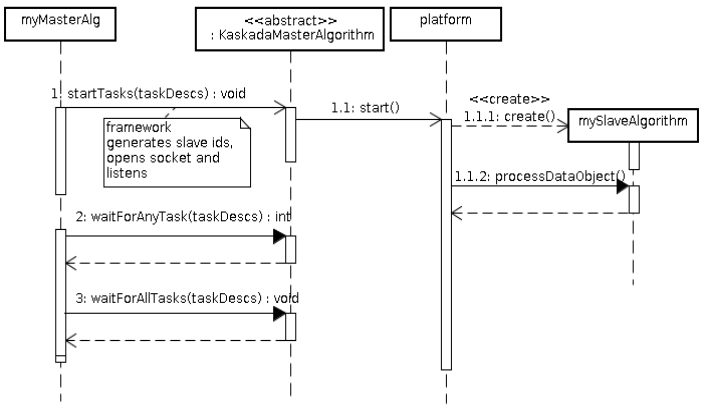

Fig. 6. The sequence diagram of the KASKADA framework master and slave algorithms

3. Execution scenarios for multimedia applications

In the KASKADA platform the term *scenario* is defined as an acyclic graph of interconnected stream processing tasks extended with master-slave task cycles. A task is a running instance of a KASKADA framework based algorithm. Each

graph's directed edge represents data flow from a source task to a target task. A master-slave cycle is formed by a master task sending a single object to a slave task and a slave task sending back a result object. Fig. 7 shows a logical and physical view of a scenario. The logical view focuses on the stream/object logical content; str – stands for stream, obj – object, alg – stream processing algorithm (implemented, e.g. by the MyStreamAlgorithm class). Identifier #n represents n^{th} type of stream/object/algorithm respectively. Slave algorithms are represented separately by the "slave alg #1" identifier. In the physical view, the algorithms are encapsulated symbolically by the striped frames which symbolize framework. The content types are replaced by protocol types to indicate that this type of concern is fully handled by the algorithm. Each launched algorithm instance is defined as a task.

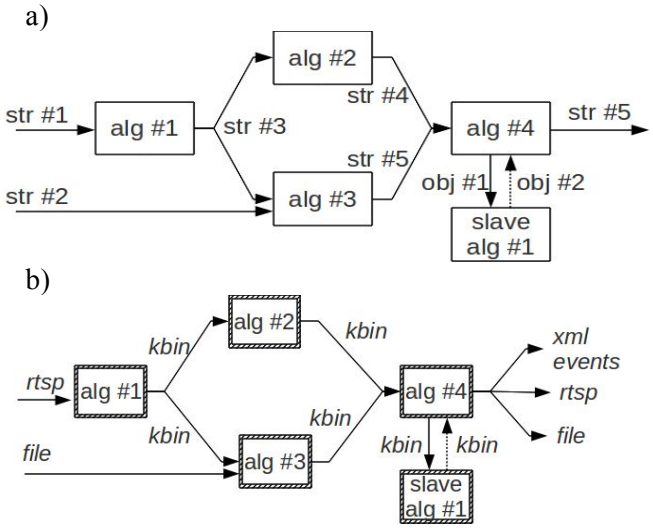

Fig. 7. Scenario logical (a) and physical view (b). The striped frames around algorithms in the physical view symbolize the KASKADA Framework wrap-up

Each task is capable of streaming and receiving data using the same three protocols: *rtsp*, *file* and *kbin*. The algorithm "alg #1" processes a single *rtsp* stream and produces a single *kbin* stream; "alg #2" processes a single *kbin* stream and produces a *kbin* stream; "alg #3" processes a *file* stream and a *kbin* stream and produces a *kbin* stream. The fourth algorithm is the most complicated one; it processes two *kbin* streams and produces *rtsp* and *file* streams. Moreover, it acts as a master algorithm and, in some cases, it launches slave algorithms. Each slave returns the result to the master after finishing its work (the dotted edge). The graph exposes a limitation of the framework. Input *rtsp* streams may be handled only exclusively, i.e. every time an *rtsp* stream is handled, no other stream may be processed. This is, in fact, no big issue because *rtsp* streams are targeted only for outside world integration purposes (camera, audio-video player communication).

The file protocol is, on the other hand, provided mainly for archiving and debug purposes. The real world task graphs always process *kbin* data streams, which benefit from the fact that they utilize a much faster Infiniband RDMA [4] interface. Fig. 8 shows an effective KASKADA platform realization of the scenario taking into account the afore-mentioned issues. The connection to resources and outside world is separated from the actual algorithm by extra front and back blocks (also implemented using KASKADA Framework). The blocks are added by the platform for every scenario execution to handle all the *kbin* conversions (all arrows without named protocol indicate *kbin*) and resource communication. The resources comprise data sources (cameras, storage) and data targets (message queue, video players, storage).

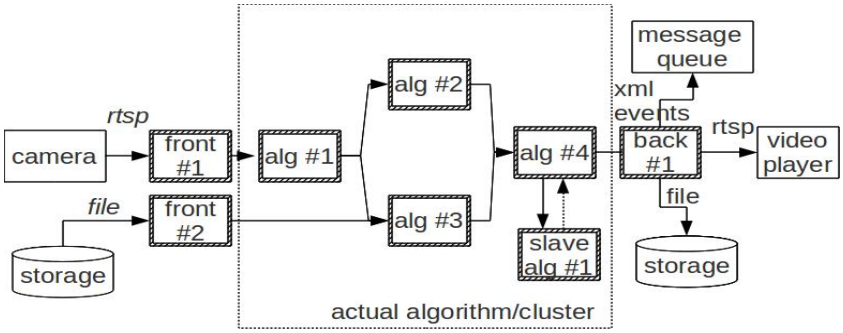

Fig. 8. Scenario of an effective KASKADA platform realization

An example of a scenario is depicted in Fig. 9. The composite task performs a simplified algorithm for identification and tracking of moving objects in a camera-observed scene. Exemplary algorithms of this vast category can be found, e.g., in [21][23].

The video is first processed by the algorithm: *detect blob outliers* – detecting shapes surrounding the blobs, i.e. clusters of pixels representing moving objects in a picture. The frames, enriched with extra information regarding the outliers, are sent to the *cut blobs* algorithm. The outliers get cut out using the detected outliers and sent to two parallel *feature* tasks, extracting visual features of two distinct categories. They could extract, e.g. chromatic and luminance properties.

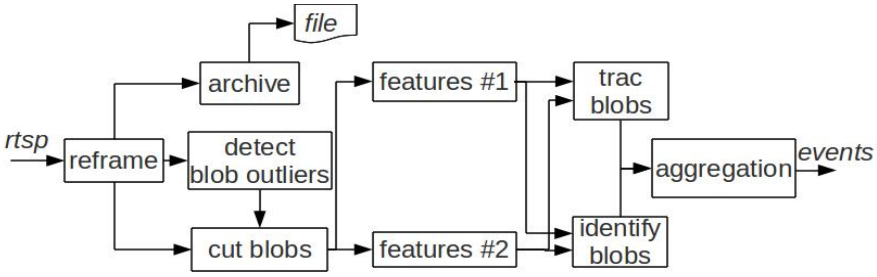

Fig. 9. An example of a scenario for object tracking and identification

Two further parallel tasks process the features. The *identify blobs* task compares the features with known identities and, as a result, produces identity events, which contain significant similarity levels between objects and known identities. The *track blobs* task compares blob locations and features in any following frames, and generates location events describing deduced routes of objects. The identity and location events are then processed by the *aggregation* block, which creates events containing both object identity information and their locations.

Fig. 10 shows a more sophisticated scenario, which is built up of many identity & tracking scenarios, and three more aggregation blocks based on [11]. Video streams from *n* cameras are processed by the incorporated scenarios and provide object location events from multiple sources. All the events are aggregated by yet another aggregation block which converts the events to a unified form. Location values are mapped to a common space, and events describing the same objects seen by other cameras are combined. The block tries to join object trajectories crossing different view scopes in time, increasing the identification reliability based on historical results and generates improved location and identification of events from the whole area of interest.

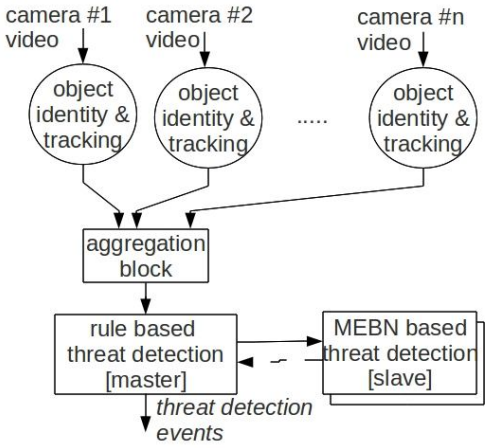

Fig. 10. An example of a scenario for threat detection

Next, the events enter the rule-based threat detection block, which uses a knowledge base, defined as a set of rules, to identify dangerous situations. The rules could find threats, e.g. when a person belonging to a group marked “dangerous” is found or , in a corporation, when a person from a certain group spends too much time in areas not assigned to them. The rules define a, so as to say, simplistic way of thinking. Uncertainty is rounded to ones and zeros, e.g. a person who has a 0.81 probability of being X is assumed to be is X. The rules prefer false positives to false negatives (i.e. prefer to falsely call a situation a threat than to miss one). The rules provide threat events of average quality, but they manage to perform their job in real-

time. Every time a threat event could be sent, the block launches a slave task to refine the assessment by calculating possibly exact threat probability using Multi-Entities Bayesian Networks Logic [12], this being a first-order Bayesian logic.

The said logic is defined as a set of small Bayesian Network template frames. Based on the frames, and a given query, the logic's deduction algorithm creates Situation Specific Bayesian Networks apt to evaluate the probabilities of the query-related variables. The deduction algorithm, integrated into slave tasks, has the potential of creating very precise probability evaluations, but at the cost of unpredictable computation time. To be more exact, the algorithm refines the quality iteratively; and the more time it spends the better the quality which can be achieved. In order the find the proper time-quality trade off, the rule-based block defines the deadline in which a slave block has to provide a solution. The deadline may vary depending on the type of threat (some types of threats may have greater real-time requirements than others). The slave finishes its computation so as to meet the time requirement. On the other hand, the platform may have no resources to launch more slaves, in which case no solution will ever be provided. That's why the master block sends its own threat assessment if no slave result arrives. Each assessment has quality information. The rule-based threat assessments have always lower quality than the MEBN Logic created.

4. Scenario execution management

In KASKADA platform, scenario execution is managed on four layers:

- complex service,
- simple service,
- task,
- processes/threads.

A **complex service** represents the whole scenario described as a directed acyclic graph. At this level the decomposition of the graph is performed, and the proper simple services are selected. In general cluster environments, usually these operations are performed manually by the programmer/designer during the software development.

At the **simple service** level, the concrete services are inspected and, according to the given quality policies, the proper tasks are selected. The result is the next graph, with the data connections as edges and the nodes represented by the algorithms to be executed as tasks. This graph is going to be assigned to the appropriate cluster nodes. In the general cluster environment, these operations are usually performer by the queue system with the support of the used scheduler.

In platform KASKADA, the **task level** management is performed directly by the dedicated software module: the monitor. It is implemented as a special purpose process, whose instances are running on every computation node of the cluster. Its core functions are as follow:

- Task start – spawning the system process and providing its proper initial environment.
- Task stop – informing the process of its finalization and stopping its execution.
- Task monitoring – constant checking of the task state during its execution, especially the usage of the resources including processor cores, memory and network load.

We can distinguish three characteristic modes of the task execution: *Constant pace* – where the task receives the data stream and processes its elements one by one. The source of data has low, limited, transmission rates, which is typical for multimedia streaming servers, e.g. video camera stream. *Speeded up stream* – where the task receives stream data as well, but it is provided as fast as the task can process it, which is typical for processing off-line data read from the archived files. *One-shot data* – where the whole data is provided at the beginning of the execution, which is typical for regular computing problems, e.g. text comparison.
Having the above modes, we can review the typical tasks configurations in the KASKADA platform. The first category is the *real-time tasks* analysing the constant pace multimedia streams. The results of such processing need to be delivered immediately; and, due to the limited buffer size, every delay in the computations can cause data lost. The second category is semantically the same tasks as above, but processing is performed in the speeded-up streams mode. In this case, the time constraints are not so important, the data is already stored, so, for resource balancing purposes, its processing can be suspended. The next category is tasks analysing *one-shot data*; they don't use long data streams, so their time constraints are not related to the data transmission. The above categories can be mixed as hybrid tasks, e.g. a stream analyser processing constant-pace streams and spawning one-shot slave tasks (see example in chapter V).

For the typical cluster computer: the task level management is partially performed by the queue system and other specialized software components controlling the computation nodes where the tasks are distributed, including their processor, network load, temperature etc. e.g. [8][25][5].

The lowest level of management: **processes/threads** is performed by the operating system, in case of the 'Galera' cluster: Debian Linux. At this level, tasks are represented by the system processes with the corresponding threads. The typical Unix mechanisms: priorities, quotas and tools: *nice, top, ps* are usually utilized for management purposes.

5. Design methodology for multimedia applications

The Fig. 11 shows a scheme of multimedia applications, scenarios and algorithms iterrelated development in time. The arrow combined applications-scenario and scenario-algorithm pairs indicate dependencies, i.e. each scenario depends on a set of algorithms and each application depends on a set of scenarios.

The development may evolve along application, scenario, or algorithm path, i.e. the designers and developers may develop the applications in a top-down approach or in a bottom-up approach. The first approach assumes that the application design creates the need for a set of scenarios which in turn require a certain set of algorithms. In the second case, the development process takes the opposite direction. The applications are built based on available scenarios which are created according to available algorithms. The algorithms are then created, according rather to technical possibilities than requests from application designers. Of course many mixed approaches are possible (a scenario requires a new algorithm which, after being invented, inspires another scenario and, finally, application).

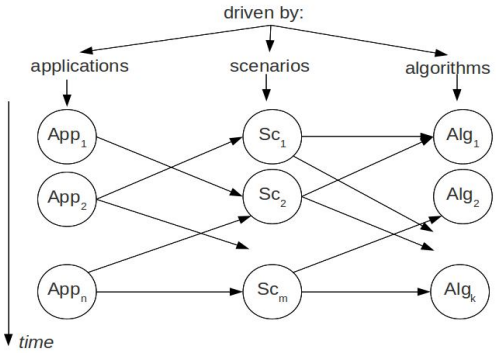

Fig. 11. The development scheme of KASKADA-based applications, scenarios and algorithms

The algorithm development process consists of seven phases as depicted in Fig. 12. The process starts with the *design* phase in which the algorithm is developed on the conceptual level. The author considers the problems of algorithm internal data flow and parallelization possibilities. A proper granularity level should be targeted, bearing in mind that the algorithm will (typically) be launched as computation tasks cooperating with other tasks on common problems. Too coarse-grained algorithms may be difficult to assign efficiently to computational nodes. Moreover, if algorithms have complicated specifications, it may be difficult to integrate them with other algorithms. Too-simple and fine-grained tasks may lead to too many nodes in cooperating task graphs, and to too much performance-degrading communication.

The second phase is the *development* of a C++ algorithm based on the KASKADA Framework library. The developer has to decide which of the provided algorithm types fits best their needs, and write the algorithm with respective header files included. The header files encompass both KASKADA Framework-provided files and headers of third party libraries used by the framework. The program has to be compiled and linked. Both of the steps have to be preformed in the cluster environment, where each of the developers has their own shell account. All the necessary files required to prepare the executable are available.

The next phase is called *simple service definition*. This time the developer has to log into the *User Console (UC)* which is the KASKADA Platform's thin client interface. Each of the developers has their own UC account, where s/he can add and configure the prepared algorithms. The developer describes the algorithm to the platform. In this step the path to the algorithm and necessary extra parameters have to be defined. In the case of stream processing algorithms, input and output data formats have to be defined. The algorithms also get associated with the event types which they send. Independently, a simple service has to be defined grouping algorithms providing the same functionality. The service has to be associated with a set of parameters, which consist of launch and quality parameters. The launch parameters have to include all the parameters expected by the associated algorithms. The quality parameters define quality properties of the algorithms realizing the service.

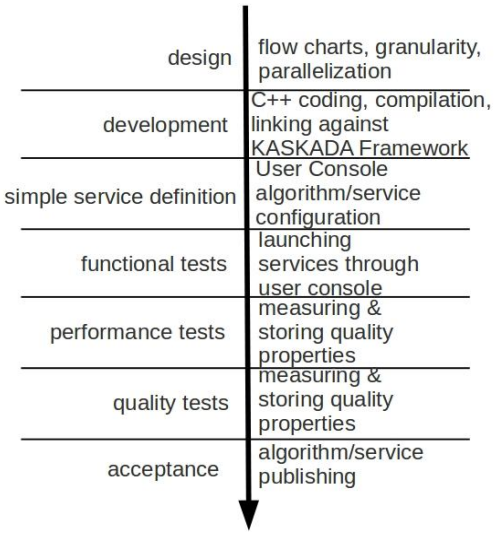

Fig. 12. Seven phases of algorithm development cycle

The algorithms can, in that stage, be executed from the UC, which lets the developer perform all the test phases. First, arrives the *functional tests* phase. Only now, master algorithms can launch slave algorithms, and stream algorithms can be tested against the production streams (or their test copy).

If the algorithms behave according to their functional specification, quality related test phases may start. The *performance tests* constitute the first phase, where properties related to resources consumed by the algorithm running within a service are measured. The properties include: used memory, processors (up to 8 on one node in the Galera cluster) and processing time required to handle a data set. The second phase is related to algorithm result quality, and is called simply *quality tests*. This phase could provide, e.g., values of false positives and false negatives of

detecting a certain property. All of the properties should be set in the service quality properties in UC.

If the algorithm meets the developer's expectations, they may contact a UC administrator who will accept the algorithm (*acceptance phase*). The algorithm's executable aspects and all its necessary configuration files, will be copied to a new area. The algorithm and the service will become public in UC and will become useful by the end users.

The scenario development process is similar. In the design phase a new scenario concept is inspired by the available simple services (algorithms); or new algorithm development processes are started if the targeted functionality can not be covered by the available ones. The development and service definition phases are combined into the *complex service definition phase* in which the graph of simple services is written into an XML document. The test phases and the acceptance phase are basically the same.

The application development process takes place outside of the KASKADA platform. Even though its progress is in no way controlled by the platform, its main phases are probably quite similar to the scenario development phases. No matter which kind of development strategy is chosen, there is surely a design, a development, and a testing phase, included. The design phase is once again focused either on seeking inspiration in the available building blocks (here scenarios) or on deciding which scenarios to develop and starting respective lower-level development phases.

6. Conclusions

The KASKADA platform supports the multimedia processing algorithms and scenarios in three dimensions:

- execution supported by the scheduler, service and task components (chapter IV),
- design and implementation supported by the user console, framework libraries and set of standard tools: C++ compiler, debugger and IDE (chapter II),
- development methodology supported by the iterative and incremental process recommendations (chapter V).

The cluster computer-centric environment provides an excellent execution environment for multimedia processing algorithms: including stream distribution, scenario decomposition, task assignment and monitoring. All these features are directly supported by both general and dedicated software and hardware components provided by the computer centre.

The developer activities, like scenario designing, algorithm implementation and testing are widely supported by the user console functionality, including test services, streams and task monitoring. Additionally, the unified framework for a programmer is provided, including a set of functionality related to the video/audio processing, encapsulated in the C++ library.

The described iterative and incremental methodology enables reliable multimedia application development; including algorithm design, service declaration and scenario definition and SOA [10] application implementation. The phases enable introduction of quality assurance elements and their evolution according to the CCMI or other maturity models.

We are concerned with the future work to be focused on the above three aspects of KASKADA platform development. We plan to introduce the quality measurements and evaluation [14]; including performance, reliability, security, safety and dependability factors [17].

References

[1] Boost C++ Libraries, http://www.boost.org/
[2] Boost Serialization Library http://www.boost.org/doc/libs/1_41_0/libs/serialization/doc/index.html
[3] Canfora G., Di Penta M.: Testing Services and Service Centric Systems: Challenges and Opportunities, IT Professional, IEEE Computer Society, March/April 2006.
[4] Cohen A.: RDMA offers low overhead, high speed, Network World, March 2003, http://www.networkworld.com/news/tech/2003/0324tech.html
[5] collectd – The system statistics collection daemon, http://collectd.org/
[6] Computer Academic Center – TASK, http://www.task.gda.pl/
[7] Gamma E., Helm R., Johnson R., Vlissides J. M.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional (1994).
[8] The Industry Standard In Open Source Monitoring, http://www.nagios.org/
[9] InifniBand Trade Association homepage, http://www.infinibandta.org/
[10] Krafzig D., Banke K., Slama D.: Enterprise SOA: Service-Oriented Architecture Best Practice, Prentice Hall PTR, October 2004.
[11] Krawczyk, H., Bańczyk, K.: Ontology Oriented Threat Detection System, conference Brunów, Poland, June 2009.
[12] Laskey, Kathrin B.: MEBN: A Logic for Open-World Probabilistic Reasoning, Working Paper, C4I Papers, George Mason University, February 2006.
[13] Lustre homepage, http://wiki.lustre.org/
[14] Maglio P., Srinivasan S., Kreulen J. T., Spohrer J.: Service Systems, Service Scientists, SSME, and Innovation, Communication of the ACM, July 2006.
[15] Maui scheduler homepage, http://www.clusterresources.com/products/maui/
[16] Message Passing Interface, http://www.mcs.anl.gov/research/projects/mpi/standard.html
[17] Oppenheimer D., Patterson D. A.: Architecture and Dependability of Large-Scale Internet Services, IEEE Internet Computing, September/October 2002
[18] Portable Batch System (PBS), http://openpbs.org/
[19] Real Time Streaming Protocol (RTSP), http://www.ietf.org/rfc/rfc2326.txt, April 1998.
[20] Secure Shell, http://en.wikipedia.org/wiki/Secure_Shell
[21] Town, C., Ontology-Driven Bayesian Networks for Dynamic Scene Understanding, University of Cambridge Computer Laboratory, UK, Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04.

[22] Universal Description Discovery and Integration, http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration
[23] Wang, Y., Van Dyck, R. E., Doherty, J. F., Tracking Moving Objects In Video Sequences, 2002.
[24] World Wide Web Consortium, Simple Object Access Protocol Specification, http://www.w3.org/TR/soap/
[25] xCAT Extreme Cloud Administration Toolkit, http://xcat.sourceforge.net/

The work was realized as a part of MAYDAY EURO 2012 project,
Operational Program Innovative Economy 2007-2013,
Priority 2 „Infrastructure area B+R".