

# Real-time Processing of Media Streams: A Case for Event-based Interaction

Viktor S. Wold Eide<sup>1,2</sup>, Frank Eliassen<sup>1,2</sup>, Olav Lysne<sup>1,2</sup>, and Ole-Christoffer Granmo<sup>1,2</sup>  
{viktore, olegr}@ifi.uio.no, {frank, olavly}@simula.no

<sup>1</sup>Department of Informatics, University of Oslo  
P.O. Box 1080 Blindern, N-0314 Oslo, Norway

<sup>2</sup>Simula Research Laboratory  
P.O. Box 134, N-1325 Lysaker, Norway

## Abstract

*There are many challenges in devising solutions for on-line content processing of live networked multimedia sessions. These include the computational complexity of feature extraction and high-level concept recognition, the massive amount of data to be analyzed under real-time requirements and the intricate correspondence between low-level features and high-level concepts. Our approach to these challenges is a distributed architecture consisting of interacting components encapsulating feature extraction and concept classifier algorithms. The purpose of the framework is to simplify the development of applications for the domain of on-line multimedia content processing.*

*In this paper we focus on the architecture of the framework and argue that it fits well to the publish / subscribe interaction paradigm, leading to an event-based interaction model. Furthermore, we analyze different aspects of the application domain in more depth, such as requirements for scalability, reconfiguration, migration, event notification selection, filtering, and ordering. The main contribution of this paper is, that we for each aspect show how a suitable event notification service may satisfy the corresponding requirements. We also describe parts of a framework prototype. In particular we report on how the event notification service used satisfies the identified requirements.*

## 1. Introduction

The technical ability to generate volumes of digital media data is becoming increasingly “main stream” in today’s electronic world. On the other hand, technology for automatic indexing (associating meta-data to) such media data is immature.

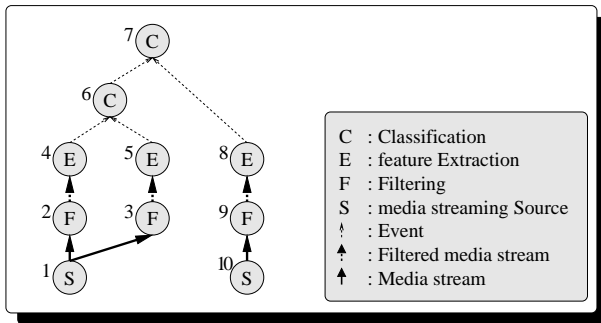
The main challenges that must be addressed include the computational complexity of feature extraction and high-level concept recognition, and the massive amount of data to be analyzed under real-time constraints. By taking ad-

vantage of a parallel processing architecture, features can be extracted in parallel. A multi agent based system for coarse-grained distribution of feature extraction is presented in [12]. Another challenge faced by real-world applications is the noise introduced into the extracted features (e.g. shadows in a video). In [8], a generic automatic video surveillance system is described, for recognizing various kinds of human activities. A statistical approach (Bayesian network) is used for noise suppression.

In our research we are developing a component based framework with the goal of simplifying the development of distributed scalable applications for on-line media content analysis. The framework is a generic modular application which is instantiated during the development of specific content analysis applications. An advantage of a carefully designed framework is that new sub-technologies can be plugged into the framework as they become available. As an example, third parties may extend the framework by providing new and/or better feature extraction algorithms.

The focus of this paper is on the requirements for communication and distribution of the content analysis framework. Based on our analysis of different aspects of the application domain, such as requirements for scalability, reconfiguration, and migration, we show that requirements for communication and distribution is better supported by a suitable distributed event notification service rather than by a synchronous communication service such as RMI. Furthermore, our experiments with a framework prototype approve this conclusion from the analysis.

The rest of the paper is organized as follows. In Section 2 we describe the architecture of typical content analysis applications. In Section 3 we introduce our computational architecture and claim that it fits well to the publish / subscribe interaction paradigm. The rest of this section is devoted to various aspects of our application domain and the requirements imposed upon the event notification service. In Section 4 we describe those parts of a prototype of the framework which are relevant from an event notification service point of view. In Section 5 we offer some conclusions and outlook to future work.



**Figure 1. An example of a content analysis hierarchy.**

## 2. Content Analysis

A common way to build content analysis applications is to combine low-level quantitative media processing into high-level concept recognition. Typically, such applications are logically organized as a hierarchy, as shown in Figure 1. At the lowest level of the hierarchy there are media streaming sources. At the level above, media streams are filtered and transformed. The transformed media streams are then fed to feature extraction algorithms. Feature extraction algorithms operate on samples/segments from the transformed media streams and, in case of video, calculate features such as texture coarseness, center of gravity, color histograms, and motion vectors. Results from feature extraction algorithms are generally reported to classification algorithms higher up in the hierarchy that are responsible for detecting higher level domain concepts such as a “person” occurring in a media stream. In other words, classification is interpretation of extracted features in some application specific context.

Typically, content analysis applications are implemented as monolithic applications making reuse, development, maintenance and extension by third parties difficult. Such applications are often executed in single processes, unable to benefit from distributed processing environments. In the following section we present a framework developed in the DMJ project[4], addressing these weaknesses.

## 3. Application domain and Event Notification Service Requirements

As a solution to the inherent problems of traditional monolithic content analysis systems, we suggest a component-based approach. Logically, the media processing hierarchy is similar, but the different algorithms are now encapsulated in components - F (Filter), E (feature Extrac-

tion), and C (Classification) components. The content analysis task is realized as a collection of interacting components, where components indirectly monitor other components and react to particular changes in their state.

As we will see in the following, the requirements for our application domain and the communication patterns fit very well with the publish / subscribe interaction paradigm, leading to an event-based interaction model. Event-based systems rely on some kind of event notification service. The responsibility of the event notification service is to propagate events from the event producers to event consumers, generally in a many-many manner.

We will now characterize and describe application requirements and see how each of these translates into requirements for the event notification service.

### 3.1. Event-based component interaction

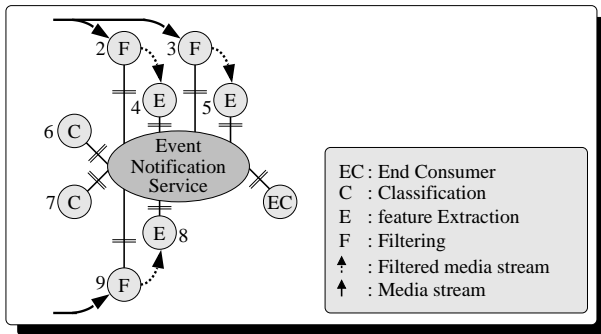
From the example in Figure 1 it should be clear that components interact in different ways, having different causalities such as - one-one, one-many, many-one, and many-many (not illustrated). In this paper we mainly focus on the interaction between E and C components, which is designed as exchange of event notifications.

One-many communication, often used for sharing results between a number of components, is important for resource consumption and hence scalability reasons. If a component has high data rate input, spends a lot of time processing, has relatively low data rate output, and the output is of interest to a number of other components, then it is a good candidate for sharing.

The causality of the interaction as well as the loose coupling between components are some of the arguments for an event-based interaction model, illustrated in Figure 2. The “End Consumer” component in the figure represents the final destination for the meta-data extracted from the media streams, such as a database or a user interface component. In our framework, components are the unit of distribution.

### 3.2. Distribution

The framework must support distributed processing to cope with the massive amount of data to be analyzed in real-time and the computational complexity of feature extraction and concept recognition. Scalability is important along several axes, including the complexity of the content to be recognized and the number of media streams concurrently analyzed. Additionally, a distributed solution may be more appropriate for problem domains having an inherent distributed nature. As an example, in a traffic surveillance application video cameras are distributed geographically. Processing of video data in a host located close to a camera reduces network bandwidth consumption. As a



**Figure 2. The event notification service introduces a level of indirection between components.**

consequence, the event notification service should be able to operate across wide area networks.

On the other hand, parts of an application may consist of a number of tightly coupled components, configured to execute inside a single host. Even more tightly coupled components may execute inside a single process, in order to exploit data locality even further and avoid copying large amounts of data between different address spaces. Other cases requiring intra host and/or intra process component interaction include situations where the runtime environments available for the media processing task is limited to a single computer. From our framework perspective, the event notification service should handle such cases too, in order to simplify application development and offer consistency with respect to both programming and execution.

A distributed event notification service is also required for scalability and in order to avoid a single point of failure.

Figure 3 illustrates how the set of components may be deployed onto a set of hosts. As can be seen, some of the computers host several components. Neither the distribution of the event notification service itself nor the process boundaries are illustrated in the figure, but as described, a number of components may share a single address space.

### 3.3. Resource management

Common processing environments do not support resource reservation of CPU, memory, network bandwidth, etc. and can not give any guarantees beyond “best effort”. As a consequence, the available resources change dynamically over time. A component executing in this kind of environment may experience overflow situations when it is not able to perform processing within the limited time frame, determined by the real time properties. Similarly, underflow situations may occur if the network is overloaded, causing

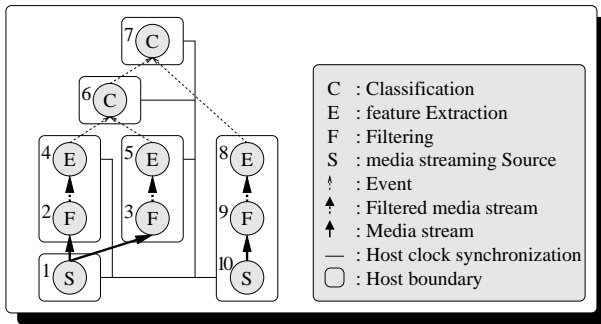
starvation at components. The infrastructure is also dynamic, although on a different time scale. It undergoes evolution where computers, cameras, sensors, etc. are added and removed. Handling such changes gracefully is important, especially for large scale, continuously running applications. Adaptation might be sufficient for handling small changes on a relatively short time scale, while reconfiguration, having more overhead, has the potential of handling larger changes. The level of indirection between components introduced by an event-based interaction model simplifies both reconfiguration and migration, also described in [5]. However, a resource management part of the framework must know the hosts and which components are currently executing at each, in order for reconfiguration and migration to be meaningful.

An approach where *all* components process and push results continuously is not always suitable, although it fits periodic processing of time-based media and one-many communication well. Some components are interested in results from other components only in certain cases and a demand driven event distribution is more appropriate from a resource consumption viewpoint. As an example, consider a C component, C3, producing events, e3, according to a specification such as *e1 before e2*, where e1 is generated by component C1 and e2 is generated by component C2. In this case C3 is not interested in events from component C2 before an event e1 from C1 has been received. In this case component C3 may reduce overall resource consumption by explicitly requesting, pulling, results from component C2. A hybrid between push and pull is also possible, such as *push for n seconds*. The most suitable style may change during runtime. The event notification service should allow components to dynamically select the appropriate operation point from the push/pull spectrum. In [7], a method for constructing C components is described which allows a C component to pull E components in a “hypothesis driven” fashion - when the information gain is expected to be significant compared to the computational cost.

### 3.4. Event notifications

Our approach for specifying event notification types is to use a sequence of type, name (and value) tuples. As an example, an E component may generate event notifications containing the following types and names:

```
string  media_type
string  media_source
string  function
string  component_type
float   motion
time    start
time    stop
```



**Figure 3. An example of distributed processing of the configuration in Figure 1.**

The purpose of the *motion* variable is to represent a quantization of the level of motion, calculated by an E component on the basis of two consecutive video frames. The variables *start* and *stop* represent the interval in which the event, reported in an event notification, took place. An instance of such an event notification may look something like:

```
string  media_type      video
string  media_source    rtp://224.0.7.1/1111
string  function        motion_estimation
string  component_type  E
float   motion          0.32
time    start           3215927128.020
time    stop            3215927128.060
```

### 3.5. Event selection and filtering

Following the example above, a C component may subscribe to event notifications generated by E components processing video from a specific source, performing motion estimation, but limited to event notifications where the motion is above some threshold, by supplying the following filter:

```
string  media_type      video
string  media_source    rtp://224.0.7.1/1111
string  function        motion_estimation
string  component_type  E
float   motion          > 0.5
```

The effect of filtering depends on where the filtering takes place. At one extreme, the code responsible for event notification filtering is supplied as a library and linked into the component code. In this case the event notification service “broadcasts” all event notifications to all components which perform filtering by executing this library code. The scalability of this approach is very limited, since all network components (routers, switches), all hosts (their network interface cards and operating systems) and all processes (the

library code) see all events. At the other extreme, the event notification service is implemented as a distributed service, either provided as a native and ubiquitous service of the network itself or alternatively added as an overlay network, executing on some hosts and/or network nodes. The filter specifications from each subscriber are propagated towards the event producers, and at each internal node new filter specifications are merged with old ones. Event notifications which do not match any subscriptions are stopped early. The event notification service should also provide an API for notifying producers when there are no interested consumers, to improve scalability even more.

Filtering also allows a number of event consumers to share a single event producer in a conform way. As an example, consider a component specifying the filter given previously, and a new consumer on the same host which subscribes with a similar filter, where  $motion > 0.3$ . The event notification service resolves the filter specification incompatibility, but a more restrictive filter is effective immediately (although delivery is not optimized) while a less restrictive filter may have to propagate through all the event notification service nodes and possibly also notify the producer component. A low and predictable delay is important for the domain of real-time media processing, because it determines on which time scale filters may be used as a means of turning on and off event producers.

### 3.6. Event notification delivery

The event notification service must balance requirements for real-time communication, low event propagation delay, and high throughput against reliability and ordering guarantees associated with event notification delivery. Performance numbers for an implementation of the CORBA Notification Service [13] is given in [19] for both “best effort” delivery and the highest event delivery guarantee possible with CORBA notification service - consumers receive all events in spite of supplier, consumer, and notification service crashes and network failures. A decrease in performance of more than 80% is reported when using both event and connection persistence compared to “best effort” delivery. In both cases the number of event/second as seen by each consumer decreases rapidly as the number of consumers increases, because the event notification service itself performs the group communication.

Our approach is to handle the unreliability of event delivery in the same way as we handle the unreliability of analysis algorithms, which may fail to detect and report an event or report false positives. In our framework, the C components are designed to handle this by the use of a statistical approach [7]. Consequently, an event notification service which also supports unreliable transport, such as UDP, is desirable in order to capitalize on the scalability of native

IP multicast[1].

Another issue related to the real-time characteristics, is the buffering policy used in event notification delivery. The event notification service should provide an API for specifying a policy to use when buffers are filled up, such as “discard oldest” or “discard newest” event notification.

### 3.7. Event notification ordering

Applications built on top of our framework need to temporarily relate event notifications, potentially originating from arbitrarily distributed components, and hence some ordering mechanism is required. Each event notification is associated with a time interval, given by a *start* and a *stop* time, illustrated in Section 3.4. This indicates that we allow events to have a duration in time, supporting a wide class of sustained actions to be interpreted as events. A language for specifying temporal relations (e.g. *before*, *after*, and *within 10 seconds*) is needed, but not addressed in this paper. The aggregation of simple sustained events into composite sustained events is performed according to such a temporal specification. The ordering mechanisms must be sufficiently strong in order to support the specification language expressiveness. The design of our framework, supporting such global ordering, is based upon a common knowledge of time in all components.

It is well known that in a distributed system, global time can only be available down to a certain level of accuracy. The most widely used global time service in the Internet is NTP (Network Time Protocol, RFC 1305) where synchronization accuracies between different hosts is reported to be in the order of tens of milliseconds on the Internet in general and one millisecond over LANs[11]. In general the required level of accuracy is application dependent, but considering video streams with 25 frames per second, NTP may provide frame level accuracy even over wide area networks. Some additional inaccuracy is introduced by the indeterministic skew (OS scheduling, etc.) on the data path from capture device to the timestamping application. Hence, timestamping should happen as close to the source as possible. As an example, a computer with a video camera connected through a video grabber card, should timestamp each video frame in the video grabber card or in the driver. Inaccuracies introduced by NTP and the timestamping process introduce intervals of uncertainties, where the system is unable to determine ordering (e.g. which event took place first).

As a result, our framework does not require any kind of ordering or synchronization support in the event notification service. Ordering is handled and implemented at the framework level. Assuming that timestamps are obtained from globally synchronized clocks, components are able to detect and handle delays introduced by the event notification service. In SIENA[2] (Scalable Internet Event Notification

Architecture) a similar approach is used to detect and account for latency effects.

As a conclusion, event notification services designed to provide ordering guarantees may have problems related to scalability, and in this case they are unsuited for our application domain.

### 3.8. Event notification size

As an example of a “medium sized” event notification, consider an E component doing motion estimation, treating each frame as a number of blocks of, say  $16 \times 16$  pixels, calculating a motion vector and difference value for each block on the basis of two consecutive video frames. A  $640 \times 480$  pixel frame size results in 1200 motion vectors and difference values. Another example of space consuming events is compressed images, an important class of data in media processing applications. A component may perform skin classification on particular video frames, where the intensity in the resulting gray scale image represents the skin color probability. An event notification service which is able to handle such potential space consuming event notifications is beneficial.

However, for filtering reasons all these data should probably not *always* be embedded inside a single event notification. There is a tradeoff between filtering granularity and performance. Event consumers may have interest in only parts of a video frame. Event notifications which fit in a single link layer frame reduces fragmentation and reassembly costs in the communication protocol stack and at the same time allows relatively fine grained filtering.

### 3.9. Event notification service and media streaming

Until now we have focused on the communication between E and C components and argued that an event-based interaction model fits well. From a media processing point of view, it is also interesting to use the event notification service itself for interaction between F and E components, and even for streaming time-based media (e.g. video) to F components. The extension of event notification services for handling stream based interaction internally has been described in [3] and [18].

An event notification service which is capable of handling such “stream events” is attractive, especially when used in combination with filtering. A producer publishing video frames as event notifications may associate a type, name, and value tuple expressing the type of a particular video frame, such as an I (intra), P (predictive), or B (bidirectional) coded frame. Different media processing components may then subscribe and register interest in I frames only, or in only every 10th I frame (temporal division), depending on the expressiveness of the filter specification

language. A producer may send different areas of a video frame as different event notifications, providing consumers with the ability to express interest in certain areas of a video frame only (spatial division). An E component doing motion estimation may subscribe only to events which contain the boarder blocks of a video frame, and a C component may use these values to detect entry or exit from a camera view.

## 4. Prototype

In this section we describe a prototype of the framework with emphasis on the parts relevant for distributed event-based systems.

### 4.1. Event notification service

Different event notification service technologies, such as CORBA Event Service[14], CORBA Notification Service, and SIENA are available. We are at the time of writing in the process of starting to use SIENA and CORBA Notification Service. The event notification service in this prototype is based on Mbus[16], which we will now describe and discuss.

Mbus is a standard[17] currently being worked out by the IETF. Mbus is designed to support *coordination and control* between different application entities, corresponding roughly to components in our terminology. The standard defines message addressing, transport, security issues and message syntax for a lightweight message oriented infrastructure for ad-hoc composition of heterogeneous components. The authors of Mbus state that Mbus is not intended for use as a wide area conference control protocol, for security (conforming Mbus implementations must support both authentication and encryption[17]), scalability, message reliability and delay reasons[16]. In the following we evaluate the suitability of Mbus as an event notification service, with respect to the requirements discussed in Section 3.

Mbus supports binding of different causalities by using a “broadcasting” and filtering technique. All components participating in a specific Mbus session subscribe to an IP multicast address and in effect Mbus messages are “broadcasted” to the set of computers hosting components participating in this Mbus session. Currently, Mbus is implemented as a library which is linked into the applications. By linking the Mbus code into the application itself, the Mbus layer in each component sees all Mbus traffic and must filter messages. Mbus could have been implemented as a part of the operating system, pushing the filtering one step closer to the event notification producer. In this respect Mbus is rather suboptimal. As an optimization, several Mbus sessions may be started, using a set of IP multicast addresses.

Each component participate in a subset of these Mbus sessions and messages are sent to different sessions based upon some predefined scheme.

Important for event notification selection and filtering is the addressing used in Mbus. The address of a component is specified when initializing the Mbus layer. In other words, selection and filtering is associated with the address given to a component, not by specifying filters. The Mbus header includes source and destination addresses, each a sequence of attribute-value pairs, of which exactly one pair is guaranteed to be unique (combination of process identifier, process demultiplexer and IP address). Each Mbus component receives messages addressed to any subset of its own address. A Mbus component is able to address a single, “(id:7-1@129.240.64.28)”, a subset, “(media\_type:video component\_type:E)”, or all, “()”, Mbus components by specifying an appropriate sequence of attribute-value pairs. As a result, bindings between components are implicit.

It should by now be evident that Mbus supports push-based interaction. Some higher level Mbus services are described in [9], such as abstractions for remote procedure call. Pull style interaction is achieved by either sending requests as event notifications or by using the remote procedure call service.

An Mbus-based event notification service acts as a layer of indirection between components giving both access and location transparency, simplifying reconfiguration and migration. Component awareness is supported by a soft state approach, where the Mbus layer listens and periodically sends self-announcements messages on behalf of its component. When migrating a component to another host, its Mbus address remains the same (except for the value of the *id* attribute, reported in succeeding self-announcements).

Regarding scalability, message propagation delay, and reliability of event delivery, an Mbus-based event notification service inherits many of its characteristics from IP multicast, which is realized as a distributed and scalable service. The state necessary for forwarding IP multicast packets is calculated and stored in both routers and in hosts acting on behalf of multicast receivers in a distributed fashion. The Mbus component awareness functionality limits scalability, but the rate of self-announcements is adapted to the number of entities participating in a session.

IP multicast also decreases latency (by sending only one instance of a packet over any link) which is very important for the domain initially targeted by IP multicast, real-time, high bandwidth multi-user applications, such as video and audio conferences.

At the transport level, Mbus messages are encapsulated in UDP packets and transported unreliably by IP multicast. In the special case where the message is targeted at exactly one receiver, reliable unicast delivery is supported by the Mbus layer, using acknowledgement, timeout, and retrans-

missions mechanisms. The Mbus/UDP/IP multicast protocol stack does not give any ordering guarantees, but assuming global time and associating a time interval to each event (see section 3.7) handles this ordering problem, except for very time-sensitive applications.

From the discussion above, we believe that Mbus is a reasonable alternative as an event notification service for small scale experiments. From a prototyping viewpoint it is easy to integrate, requiring few lines of code, and the text-based message format simplifies message snooping.

## 4.2. F, E, and C, components

In the current prototype of the framework, ways to implement F, E, and C components have been identified.

We have used applications such as JMStudio, bundled with JMF[21] (Java Media Framework), and vic[10] for streaming video from a file or from a capture card connected to a video camera.

The F and E components are realized using JMF. JMF performs low level media tasks, such as capture, transport, streaming, (de)multiplexing, (de)coding, and rendering. JMF also provides a pluggable architecture for integrating custom media processing algorithms. The F and E components developed for the prototype are implemented as classes in the Java programming language and pluggable into the JMF framework. F and E components implement a method (e.g. iterating through all pixels of a video frame performing some calculations), which is invoked by the JMF system whenever a new media sample is available.

The C component implemented for this prototype is based on dynamic object-oriented Bayesian networks (a generalization of the hidden Markov model) and particle filters [7].

## 4.3. Component interaction

In our implementation of the media processing framework, the communication between media sources and F components is done by standard techniques for streaming media data, such as MPEG/RTP[20]/UDP/IP multicast. An E component executes, together with some F components, inside a single process and communication between these F and E components is handled by shared buffers and performed by JMF. Interaction between E and C components is handled by the Mbus-based event notification service.

## 4.4. Event notification ordering

As described in Section 3.7, a common knowledge of global time in all components is assumed. When media (e.g. video) is streamed by using RTP, each RTP packet contains an RTP timestamp. This RTP timestamp is only

relative and it is used by receivers to determine the duration between two consecutive media samples. However it is possible for receivers to determine the global NTP time of the media sample in spite that such RTP packets contain no NTP timestamp themselves. RTP has a companion protocol, RTCP[20], which is used for sending reports about the session itself. Such reports are sent “out of band” on a separate port. RTCP packets include so called “sender reports” which, for each source gives a NTP timestamp and the corresponding RTP timestamp. When a receiver, a F component, has seen two such RTCP “sender reports” from a media source, it is able to derive a mapping from an arbitrary RTP time, for this source, to global NTP time. The timestamps then follow the filtered and transformed media data to E components, which inserts the timestamps into event notifications. Then C components use these timestamps in order to relate events temporally.

## 4.5. Experiences

The purpose of the prototype is to gain experience, serve as a proof of concept, and to verify the flexibility of Mbus with respect to the different requirements discussed in Section 3.

Component interaction uses push style communication. The media source applications push video streams over the network, using IP multicast. The video streams are received by processes executing the JMF runtime system and hosting F and E components. In each process, the JMF runtime system invokes the media processing methods of F and E components whenever a new video frame arrives. Each E component pushes the calculated results, features extracted from video frames, over the event notification service. A C component, hosted by a separate process, receives these results and performs classification.

The deployment of components onto hosts is performed manually in this prototype. The flexibility of the Mbus-based event notification service was confirmed by experiments - configuration and compilation was unnecessary before starting components on different hosts or when reconfiguring the media processing task. Mbus performs as expected, both as an intra host event notification service, but also when components are hosted by different computers interconnected by a local area network. Some successful migration experiments have also been conducted [15], using Voyager[6] for moving running components between hosts.

We have performed some preliminary experiments, testing different distribution strategies. On one extreme, all components executed on a single machine while on the other extreme each component was hosted by a different computer. Distributed processing of the media analysis task allows the development of more resource demanding, but also more reliable components, improving scalability.

## 5. Conclusions and future work

In this paper we have presented an architecture for a framework for developing applications supporting distributed real-time processing of time-based media streams. We have described different aspects of our target application domain and argued that this domain is a case for distributed event-based interaction. The main contribution of this paper is the analysis of different aspects relevant to the application domain and the translation to corresponding requirements for a suitable event notification service. These requirements include scalability, reconfiguration, migration, event notification selection and filtering. Ordering is handled at the framework level by assuming globally synchronized clocks and association of timestamps to events. A prototype of the framework has been implemented, serving as a proof of concept and for evaluation purposes. We find the results promising.

We are currently working on a new prototype of both framework and test application. The test application chosen is tracking of objects in video, an application which is challenging and extensible along several axes. It should be possible to increase complexity, with respect to feature extraction and classification, by going from object tracking to recognition of persons, and maybe also their identity. For scalability tests, a number of video streams can be analyzed concurrently. Additionally, some of the algorithms are relatively compute intensive, but also suited for parallel processing (e.g. motion estimation).

## 6. Acknowledgments

We would like to thank all persons involved in the Distributed Media Journaling project for contributing to ideas presented in this paper. We also would like to thank the reviewers for valuable comments.

The DMJ project is funded by the Norwegian Research Council through the DITS program, under grant no. 126103/431.

## References

- [1] K. C. Almeroth. The Evolution of Multicast: From the MBone to Interdomain Multicast to Internet2 Deployment. *IEEE Network*, 2000.
- [2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland OR, USA, July 2000.
- [3] D. Chambers, G. Lyons, and J. Duggan. Stream Enhancements for the CORBA Event Service. In *Proceedings of the ACM Multimedia (SIGMM) Conference, Ottawa*, October 2001.
- [4] DMJ, (Distributed Media Journaling) project. <http://www.ifi.uio.no/~dmj/>.
- [5] V. S. W. Eide, F. Eliassen, and O. Lysne. Supporting Distributed Processing of Time-based Media Streams. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001), Rome, Italy*, pages 281–288, Sept 2001.
- [6] G. Glass. Voyager - the universal orb. Technical report, Objectspace, January 1999.
- [7] O.-C. Granmo and F. V. Jensen. Real-time Hypothesis Driven Feature Extraction on Parallel Processing Architectures. In *Proceedings of the 2002 Special Session on Parallel and Distributed Multimedia Processing & Retrieval (PDMMPR 2002), Las Vegas, USA*, June 2002.
- [8] S. Hongeng, F. Brémont, and R. Nevatia. Bayesian Framework for Video Surveillance Application. In *Proceedings of the 15th International Conference on Pattern Recognition*, pages 164–170, Sep 2000.
- [9] D. Kutscher. The Message Bus: Guidelines for Application Profile Writers. *Internet Draft, draft-ietf-mmusic-mbus-guidelines-00.txt*, 2001.
- [10] S. McCanne and V. Jacobsen. Vic: A flexible Framework for Packet Video. In *ACM Multimedia '95*, pp. 511–522, 1995.
- [11] D. L. Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *IEEE Transactions Networks*, pages 245–254, 1995.
- [12] Y. Nakamura and M. Nagao. Parallel Feature Extraction System With Multi-Agents-PAFE. *11th IAPR International Conference on Pattern Recognition (ICPR)*, vol. 2:371–375, 1992.
- [13] Object Management Group Inc. CORBA services, Notification Service Specification, v1.0. <http://www.omg.org/>, 2000.
- [14] Object Management Group Inc. CORBA services, Event Service Specification, v1.1. <http://www.omg.org/>, 2001.
- [15] R. W. Olsen. Component Framework for Distributed Media Journaling. Master's thesis, (in Norwegian), Department of Informatics, University of Oslo, May 2001.
- [16] J. Ott, D. Kutscher, and C. Perkins. The Message Bus: A Platform for Component-based Conferencing Applications. *CSCW2000, workshop on Component-Based Groupware*, 2000.
- [17] J. Ott, C. Perkins, and D. Kutscher. A message bus for local coordination. *Internet Draft, draft-ietf-mmusic-mbus-04.txt*, 2001.
- [18] T. Qian and R. Campbell. Extending OMG Event Service for Integrating Distributed Multimedia Components. In *Proceedings of the Fourth International Conference on Intelligence in Services and Networks, Como, Italy*. Lecture Notes in Computer Science by Springer-Verlag, May 1997.
- [19] S. Ramani, B. Dasarathy, and K. S. Trivedi. Reliable Messaging Using the CORBA Notification Service. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001), Rome, Italy*, pages 229–238, Sept 2001.
- [20] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobsen. RTP: A Transport Protocol for Real-Time Applications. *RFC 1889*, 1996.
- [21] Sun Microsystems Inc. Java Media Framework, API Guide, v2.0. <http://java.sun.com/>, 1999.