Modelling Dependency in Multimedia Streams

Alexander Eichhorn
Distributed Systems and Operating Systems Department
Ilmenau Technical University
P.O. Box 100 565, 98684 Ilmenau, Germany
alexander.eichhorn@tu-ilmenau.de

ABSTRACT

Expressing and analysing data dependency in multimedia streams is promising, since content-aware policies at a transport level would benefit from such services. In this paper we present a format-independent dependency model aimed at specifying, validating and reasoning about structural dependency in multimedia streams. Based on this model, we developed a universal dependency description language and a dependency validation service to serve as an infrastructure for content-aware transport layers. Driven by application knowledge, this special form of a cross-layer design enables lower layers to reason about the impact of data loss and drops during transmission while being unaware of the real data format.

We outline, how this infrastructure can be used to build content-aware error protection policies and explain how applications need to specify dependency and prepare media streams in order to gain benefits from those policies. While costs and benefits of a dependency model are only quantifiable in conjunction with special policies, we report on the general worst-case costs of our model here.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols–Applications; D.2.11 [**Software Engineering**]: Software Architectures–Domain-specific Architectures

General Terms

Algorithms, Design, Performance, Languages

Keywords

Content-aware media streaming, dependency, error control

1. INTRODUCTION

Protocols for packetised media streaming over best-effort networks need to deal with transmission errors and bandwidth variations introduced by the unpredictable behaviour of existing transport protocols. Neither bandwidth nor loss or delay bounds are guaranteed here. In order to preserve realtime characteristics of media streams, streaming protocols need to include policies for flow control and error protection when layered on top of unreliable transport channels. However, building streaming protocols without considering data contents yields sub-optimal results. Hence, content-aware transport services gain increased interest. While information such as rate-distortion, data dependency and timing are suitable concepts to leverage content-awareness, this paper primarily focuses on dependency concepts.

The efficient compression methods found in multimedia coding standards introduce dependencies between sections of a media stream and thus also between network packets. Data dependency makes streams highly vulnerable to packet loss since resulting decoding errors will propagate spatially and temporally across the reconstructed signal. While most delivery channels lack appropriate support, in reaction several error protection mechanisms have been introduced at the application level, such as resynchronisation markers, error concealment [20], packetisation rules [21, 12], unequal and forward error correction (FEC) [2, 1] and retransmission schemes (ARQ) [16, 17]. However, these mechanisms rely on intimate knowledge of the encoding format and achieve acceptable results only if supported by additional information about the delivery channel. Such information is sometimes unavailable at encoding time (when delivering pre-encoded streams), highly variable (for wireless channels) and mostly transparently hidden by protocol layering. Current communication system layers handle stream data at most equal during a complete session, either protecting all data or none, while usually ignoring real-time constraints.

In consequence, design and architecture of streaming error control need to be carefully reconsidered. A central question is how to integrate application-level error control with transport-level mechanisms. Cross-layer design is a promising approach, but current cross-layer research is driven by issues of wireless delivery channels. In contrast, our work focuses on application-driven approaches, where an application is able to express data properties and transport layers are enabled to reason about them.

It is well known, that content-aware error control policies are able to protect a stream's integrity and thus information much better than best-effort policies. Selective drop policies [10, 13], unequal error protection [14, 2, 1], selective retransmissions [17, 5], rate-distortion optimal packet scheduling [15, 5] and layered quality adaptation [4] are examples. If stream data needs to be dropped in overload situations,

MM'06, October 23–27, 2006, Santa Barbara, California, USA.

policies may favour unimportant data and thus restrict error propagation [13, 10]. In particular, in error-prone environments such as wireless networks important data may be stronger protected using selective FEC and ARQ policies in order to reduce extra bandwidth demands while preserving signal fidelity.

In order to enable content-aware policies in cross-layer optimised transport layers, a universal dependency model is necessary. While several dependency models and stream description languages do exist, they are targeted at special bitstream formats only [13, 11, 14] or were designed for different purposes such as content scaling [7] and automated parser generation [8].

In this paper we propose a framework for dependency specification and validation in media streams, based on a model to reason about structural dependency, a simple description language and a robust dependency validation service. This infrastructure allows applications to specify dependency and enables lower system layers to reason about the impact of data loss and drops during transmission. Our dependency model is able to (a) validate the structural integrity of a stream and (b) estimate the importance of stream objects and (c) estimate the severity of damage caused by packet loss and explicit drops. The dependency model has been designed with the following requirements in mind:

Expressiveness In order to serve as a universal foundation, the dependency model must be able to express the complex dependency patterns found in audio-visual streams regardless of the stream format and the packetisation level (e.g. video objects, frames, slices, macroblocks or even network packets).

Purpose Independence The model must be generally applicable in different policies and mechanisms for multimedia streaming such as scheduling, error protection, fragmentation, content scaling and security.

Efficiency Since media streaming systems usually operate at the resource limits the costs for providing general dependency validation must be low in terms of extra amount of metadata appended to streams, computational overhead and storage complexity. Because multimedia streams are delay sensitive, the model and the validation service must avoid unpredictable and extra delay.

Robustness To validate a stream's integrity, the model must deliberately deal with incomplete knowledge, introduced by packet loss in unreliable networks and by the general nature of streams, where future data may be unavailable and unpredictable. In such cases, the model may give weaker information, but should recover quickly as more data becomes accessible.

In the remainder of this paper, we first examine related work focused on media stream description and dependency modelling. Next, we discuss media stream properties and relate them to dependency modelling. In section 4 we present our dependency model, show how the model supports different kinds of dependency found in recent media formats and explain how applications need to preprocess a stream in order to make use of the model. Section 5 presents the description language, the validation service, and demonstrates the integration of the dependency model into cross-layer error control policies. We then give some analytical estimates on worst-case costs of the proposed model and finally conclude the paper with an outlook on future work.

2. RELATED WORK

Several models and languages for stream syntax description exist, but they are either incomplete [5, 15], format-specific [14, 10] or lack expressiveness for dependency modelling [8, 7].

In [5], Chou and Miao describe a framework for rate-distortion optimal packet scheduling and ARQ-based error-recovery for packet-based networks. Several authors extend this framework to wireless environments [3], joint ARQ/FEC error control [18] and provide more efficient algorithms for special dependency classes [15]. While the focus of this work is rather on optimising packet schedules and error control mechanisms, the algorithms require rate-distortion measures and dependency information. However, the work lacks appropriate mechanisms to retrieve those information automatically. At this point, our dependency model is able to complement this work.

Modelling dependency in media streams is usually focused at MPEG-like bitstreams with a quite fixed frame pattern only. In [14] Mayer-Patel et. al. present an analytical model, aimed at predicting the loss-probability of different MPEG frame types in order to stronger protect vulnerable types by FEC. While abstracting from real dependency, this model is not able to serve validation purposes and correct importance estimation. In Röder et. al. [15] a simple graph-model is used to express dependency relations in media streams. Our dependency model extends this idea towards flexibility, format-independence and robustness against packet loss. Another dependency model for media streams is proposed by Hoffmann and Kühnhauser [10]. Like the model presented here, it is used for importance estimation and stream validation, but it fixed to MPEG-2 streams. Our dependency model extends the general ideas towards an universal description and validation framework. Krasic et. al. [13] propose a priority mapping framework which is used in selective drop policies for media streams. This framework is based on a dependency model and a special class of scalable MPEG codecs. However, the described dependency model is fixed and focused at simple MPEG-like dependency structures.

Several languages for stream format description and automated bitstream scaling do exist in the literature (mostly originating from MPEG-4 and MPEG-21). Flavor [8] is a formal language for stream format description and parser generation. Although it is able to express syntactical structures in media streams it lacks tools for dependency modelling. A more general framework for bitstream description, based on XML, is presented in [7]. The proposed language is explicitly targeted at stream adaptation environments in the context of MPEG-21, but it provides no tools to express dependency and does not define how to utilise the stream description. While the language is format-independent, it is intended to generate a separate description for every bitstream instead of generally modelling bitstream types. However, due to the flexibility of XML, static and dynamic dependency information is easy to express.

3. PROPERTIES OF MEDIA STREAMS

While the aim of the dependency model is to support dependency validation and importance estimation, its foundations lie in general properties of media streams. In consequence, we will first concentrate on those properties and their impact on dependency.

3.1 Dependency Relations in Media Streams

Media streams are data streams containing application-level objects with special properties such as dependency relations. Current multimedia standards define objects as instances of types, based on a format-specific type hierarchy. H.264/AVC [22], for example, knows seven types at the slice level (I, P, B, EI, EP, SI and SP-SLICEs)) while MPEG-4 Visual defines video objects located in object planes (I, P, B-VOPs). Slices are combined into frames which in turn are combined into picture groups (GOP), while they are internally structured into macroblocks. Such structural patterns are continuously repeated within a stream.

For efficient encoding, stream objects feature increasingly complex dependency relations (see figure 1 for examples), such as the combination of bi-directional, multi-picture, and weighted relations in H.264/AVC predictive video encoding [19], spatial and SNR refinement layers in scalable video coding or dyadic trees for temporal scalability in Joint Scalable Video Coding. Because of their relations, stream objects become differently important for signal reconstruction.

Dependency could be intuitively regarded as a concept to describe that the existence of one stream object is essential for processing another, related, object. We call this form *essential dependency*. However, there is an alternative form of dependency, found, for example, in Multiple Description Coding (MDC) [4]. MDC splits the encoded signal of a frame or a slice into multiple independently decodable description objects. The relation between those objects is a mutual refinement relation rather than a unidirectional existence requirement. In such schemes, every object contributes to the increase in fidelity of the reconstructed signal, if present at decoding time. If refinements get lost or corrupted the importance of the remaining objects increases proportionally since it is crucial to receive at least one description to reconstruct a signal at all. We call this second form *fragmental dependency*. For brevity we concentrate on modelling essential dependency in this paper (called dependency in the following), while integrating fragmental dependency remains an open issue.

Essential dependency relations in media streams are transitive in nature and they usually form a *dependency graph* [15]. Obviously, dependency is a property of object types, however, in recent formats object-based relations become more important. While type-based dependency was sufficient to describe early encoding formats (MPEG-1/2), recent formats specify *explicit dependency* at the object level such as the reference picture lists used for H.264/AVC B-SLICEs. So actual dependency is the property of an object while references are required to be of the correct types.

Dependency relations between objects can span considerably large sections of a stream such as a complete GOP. We call the maximal distance of all dependency relations for an object type the *dependency radius*. Even if media encoders use future frames for prediction, they reorder stream objects prior to transmission to avoid forward dependency and extra buffer requirements at the decoder. Hence, encoders guarantee that every dependency is satisfied by prior stream objects. Reordering, drop or loss at the transport level can violate this assumption.

The dependency radius of types is usually limited to small partitions of a stream. There is, for example, no dependency across distant GOPs and only limited dependency between objects in adjacent GOPs (the last B-FRAMEs of an open GOP in MPEG-2 video depend on the first I-FRAME of the following GOP). In layered streams, enhancement layer objects depend on lower layers only, even if they are of the same type. Even within a single GOP there is limited dependency between objects (consider MPEG-2 B-FRAMEs which may depend on the last two P-FRAMEs, but not all P-FRAMEs in a GOP). These restrictions will later become useful for constraining relations in our model.

Due to realtime constraints and limited buffers, the dependency radius can even encompass stream objects on distinct processing nodes in a network. While one object is decoded and displayed at a receiver node, dependent objects may still be generated by a sender node. Consequently, it is desirable to decouple actual stream processing from reasoning about dependency. Therefore we clearly distinguish between stream objects and metadata about them, and we require stream objects to contain metadata such as for example their position in the stream, their type, their relative importance, and eventually their explicit dependencies. For practical reasons we consider metadata to be attached to stream objects as object labels. Our dependency model solely keeps track of this metadata while stream objects are processed and stored in buffers. This enables the model to keep information about objects without requiring the actual objects to be stored and without introducing extra delays to stream processing.

3.2 Visibility and Predictability of Stream Structures

Media streams are continuous sequences of typed data objects. For each instance of a processing chain, new objects become visible as time advances. Information about the stream's future is usually unavailable and predictions are in general difficult. However, sometimes prediction is possible as we will discuss later. Due to resource constraints stream objects are not indefinitely stored, so old objects may no longer be available. We call this property of stream objects their *visibility*. Visibility is additionally affected by loss, corruption or reordering of objects during transmission which introduces gaps into the sequence and uncertainty about the real properties of an object, including its actual dependency. Only visible objects are subject to processing, error control and stream scheduling, while metadata of visible as well as processed, dropped and even lost objects is required for dependency reasoning.

In order to reflect these constraints we define the region of interest for our dependency model as the model's *horizon*. The horizon covers information on all objects starting at the most recent visible object and ending at the oldest object within the dependency radius of any visible and expected forthcoming object. So for any visible object and for any future object it is ensured that its dependency is covered by the horizon. Of course, different processing stages in a streaming system will have different horizons, and horizons change over time as new objects arrive and others are beeing processed.

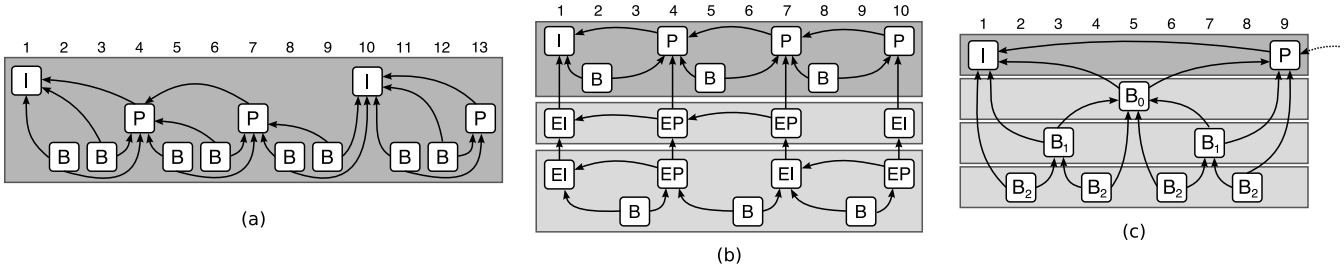

Figure 1: Bitstream dependency examples: (a) Typical MPEG-1/2/4 structure. (b) Layered H.264/AVC structure. (c) Dyadic-tree structure for temporal scalability.

Unfortunately, loss and corruption of stream objects destroy their metadata too. So metadata is unavailable if (1) the stream object itself is still invisible, (2) the object was lost or corrupted, and (3) the metadata has been discarded. While the last case is controlled by the dependency model, the other cases need special attention because metadata needs to be recovered in order to reconstruct or estimate actual dependency relations.

Streaming formats can be categorised into one of three predictability classes, reflecting the chance of correct reconstruction. If for an object type the number, types and even positions of depending objects in the stream can be predicted regardless of the horizon, the type is called *strictly predictable*. Hence, a stream containing strictly predictable object types only is strictly predictable too. Strictly predictable streams are restricted to a fixed structure. Since dependency is a property of the object type here, the benefits of such streams are a simple dependency validation and a perfect importance estimation. Examples of strictly predictable media stream formats are Digital Video (DV) and certain MPEG profiles with fixed GOP structures (e.g. MPEG-2 for Digital Video Broadcast, MPEG-2 I-frames only formats).

A second class of object types, called *limitedly predictable*, supports variable and dynamic dependency, but requires that they are known in advance. This can be accomplished by embedding dependency information into the stream prior to the involved objects or by specifying predictable properties for a subset of types (e.g. every 15th VOP is an I-VOP). This additional knowledge enables the partial recovery of metadata. While limited predictability can reveal additional benefits, we consider it a subject to future work.

All other object types are regarded as *unpredictable*, either because they alternatively depend on different object types or the number and position of referenced objects is unknown in advance. A prominent example is the already mentioned B-SLICE in H.264/AVC [22] which's actual dependency relies on encoder decisions based on media content. In unpredictable streams, the object type merely defines the set of *potential* dependency relations, while *actual* dependency is object specific. As a consequence, it becomes apparent that object types are not always sufficient to express actual dependency. Hence, we need to model both, static type-based dependency and dynamic object-based dependency. One can regard type-based dependency as a necessary and object-based dependency as a sufficient property here.

Unpredictability limits the correctness of dependency validation and importance estimation in case of lost and unrecoverable metadata. Consider, for example, a lost MPEG-4 P-VOP in an unpredictable stream. Since the type of the lost VOP is not recoverable, subsequent P-VOPs and B-VOPs would mistakenly be regarded to depend on the wrong (a preceding) P-VOP instead of detecting the broken dependency chain. Hence, it is essential to include explicit reference lists into object labels, as it is already happening in today's unpredictable formats like H.264/AVC.

4. DEPENDENCY MODELLING

Since every stream format uses separate syntactical elements and defines special relations between them, we regard media streams as continuous sequences of typed data objects and leave the type definition as well as the relation specification to application programmers and stream format designers. They may choose an arbitrary granularity level such as frames, slices, and even macroblocks or network packets as the basic unit. While the selection does not influence the expressiveness or robustness of our model, it may influence efficiency and scalability.

Since dependency relations form a *dependency graph* our model uses graph-based representations. This section introduces tools for specifying dependency and explains the process of tracking dependency relations in streams as well as the processes of dependency validation and importance estimation.

4.1 Type-based and Object-based Dependency

In our model we define stream objects as instances of types and use the function $otype : O \mapsto T$ to obtain the type of a stream object, with O being the set of all stream objects and T being the set of object types defined for a stream. While type-based dependency is known prior to stream creation, object-based dependency is unknown until the stream objects are actually created. Both kinds of dependency are modelled as graphs: a static type-based dependency graph, the *type graph* G_T, and a dynamically generated graph for object-based dependency, the *object graph* G_O. While the type graph is generated and exchanged once prior to stream transmission, the object graph is decorated at runtime as new stream objects become visible. It always contains the complete horizon. Information to decorate the object graph is stored in the type graph and in object labels.

The type graph is a directed graph $G_T = (V_T, E_T)$ which may contain parallel edges, loops and cycles in order to express arbitrary type relations found in current and future encoding formats. The vertices V_T represent the set of specified stream object types $t \in T$ and the edges $E_T \subseteq V_T \times V_T$ represent static dependency relations between those types. Thus, a relation is unidirectional and exists between exactly

two types. Types contain additional attributes, which are modelled as vertex labels, while relations are further confined by constraints, which are modelled as edge labels.

Type attributes express (limitedly) predictable features of types such as a minimal required number of references and an average importance assigned to objects of this type. Attributes are used during validation and for initial importance estimation of unpredictable types, while constraints ensure the uniqueness of relations in the type graph. Relation constraints are rules to confine actual dependency, so that only a subset of actual objects of the given type satisfies the relation. Attributes and constraints are discussed in the next section.

The type graph is used when decorating the object graph, either to provide implicit dependency for predictable types or to serve as an integrity constraint when explicit dependency is specified on the object level. The type graph can be statically validated to detect inconsistencies and to verify uniqueness. Its structure reveals a first approximation of the real importance of objects since types with more dependency relations are likely to produce more important objects.

The object dependency graph $G_O = (V_O, E_O)$ is a directed acyclic graph (DAG). The set of vertices V_O represents the currently visible subset of interdependent stream objects $o \in O$ within the horizon and the set of edges $E_O \subseteq V_O \times V_O$ represents dependency relations between those objects, such that $(v_o, v_{o'}) \in E$ *iff* object o' must be decoded in order to be able to decode object o (in other words: object o depends on o' and object o' is a reference for o). Each vertex contains a state attribute which reflects if the object is currently visible or invisible for some reason (e.g. already processed, lost or explicitly dropped).

Each stream object must be labelled by a common set of metadata (see table 1) which can be derived from encoders. Sequence, type and explicit reference information is used to build the object graph and to verify dependency. Epochs model independent stream partitions with either no or strictly limited external dependency, such as MPEG GOPs. If not specified otherwise, no dependency between objects in different epochs is allowed. This property limits error propagation in case of object loss and is used for garbage collecting unimportant information in the horizon. The optional importance correction value, when set by an application, is used to raise the importance level of an object in addition to its dependency-based importance. This may be necessary to reflect the higher weight of special headers or coding tables which are otherwise not explicitly referenced. Layer attributes define the actual layer of an object and the layer it refines. The explicit reference list, if present, contains sequence numbers of explicitly referenced objects, while if absent, type-based relations from the type graph are used instead. Without explicit references in object labels all potentially matching dependency relations as specified in the type graph are considered to be actual dependencies.

4.2 Type Attributes and Dependency Constraints

In combination, the flexibility of type-based and object-based dependency allows a format designer to express arbitrary relations. However, in order to properly express the increasingly complex dependency structures of current and future stream formats, different types of relations and several dependency constraints are required.

Label	Description
`seq`	sequence number ($\in \mathbb{N}$)
`type`	an object's type identifier ($\in T$)
`epoch`	an object's structural epoch ($\in \mathbb{N}$)
`imp_corr`	an object's additional importance ($\in \mathbb{N}_0$)
`enclayer`	encoding layer for layered streams ($\in \mathbb{N}_0$)
`reflayer`	reference layer for layered streams ($\in \mathbb{N}_0$)
`reflist`	sequence numbers of all referenced objects

Table 1: Attributes contained in object labels.

When defining type-based dependency relations, each relation points to a potential reference type. Since actual dependency exists within a maximal dependency radius only and not all objects of one type will always depend on all objects of another type, we need a way to constrain the relations.

In MPEG, for example, a P-VOP depends either on a previous I-VOP or a previous P-VOP in the same GOP. Another example are H.264/AVC B-SLICEs with multiple references to P-SLICEs and I-SLICEs. The actual count of references may differ between objects, however, there is a maximal limit for practical reasons and usually a minimal acceptable limit too. Sometimes, dependency is even *optional*, for example when special headers or codec parameter sets are intermittently exchanged. Object types usually depending on such optional headers must not essentially depend on them, or else they wrongly get rejected by validation when optional headers are missing. Essential dependency is, however, required to model relations for strictly predictable types, since their dependency must be satisfied by all object instances.

For those reasons, we define two kinds of dependency, *strong dependency* which is used to express essential relations and *weak dependency* to express optional ones. Both kinds of dependency are modelled as attributes of a dependency relation between object types. It is allowed to mix strong and weak dependency relations, but if relations are otherwise identical, the strong relation overrides the weak.

Sometimes, a minimal number of (strong or weak) dependency relations is required for all objects of a type. Hence, we define the optional type attribute $min\_deps \in \mathbb{N}_0$ which defaults to the number of strong relations defined for this type or 1 if only weak relations exist. An object of this type is considered valid (its necessary dependencies are satisfied) if at least min_deps relations to other objects actually exist, otherwise the object exhibits a broken dependency.

Without strict predictability it is unclear how many objects will later depend on an object when it initially becomes visible. In order to have a good approximation on object importance, an average importance can be obtained from the object type. This type property can be customised by a format developer using the avg_imp attribute.

Please recall from section 3.1, that dependency in media streams is unidirectional and backward only, has a limited radius, exists between types in certain layers and inside epochs, and in rare cases even across epochs. In order to model these properties we identified the following necessary and optional constraints. Note, that these constraints are intended to restrict the set of actual objects, a given object depends on. They work in backward order over the object sequence within the horizon.

Set Selector The set selector specifies how to select reference objects in the horizon. Two selectors do exist: LAST_OF selects a single object of the specified type as reference (e.g. it selects the last P-VOP prior to another VOP). The second selector, ALL_OF, likewise selects all objects of that type.

Distance The distance parameter $dist \in \mathbb{N}$ defines the size of the dependency radius for objects of type t_1 in terms of occurrences of objects of type t_2 rather than in sequence numbers. Combined with the set selectors, a distance of 2 for LAST_OF(P-VOP) selects the object of type P-VOP before the last P-VOP and a distance of 2 for ALL_OF(P-VOP) selects the last two P-VOPs.

Epoch Selector The epoch selector $epoch \in \mathbb{Z}$ specifies dependency across epochs. It further constricts the radius of potential dependency relative to the epoch of the given object to the same epoch ($epoch = 0$), a preceding epoch ($epoch < 0$) or a subsequent epoch ($epoch > 0$). It is optional and defaults to the same epoch as the object in question. Hence, the largest epoch selector for all relations in the type graph controls the maximal size of the horizon. One usage example is the relation between adjacent GOPs in MPEG video.

Layer Selector For layered streams, the optional layer selector $layer \in \{0, 1\}$ constraints the dependency relation to objects within the same layer ($layer = 0$) or to objects in lower layers ($layer = 1$). A layer selector becomes necessary when the same object types appear in more than one layer as is the case for EI, EP and B-SLICEs in H.263++ and H.264/AVC layered encoding.

Figure 2 shows an example type graph for the MPEG bitstream layout displayed in figure 1a. A B-VOP in this example depends on at least two other VOPs, a preceding I-VOP in the current or the subsequent epoch, or one or two preceding P-VOPs, if they are within the B-VOP's epoch.

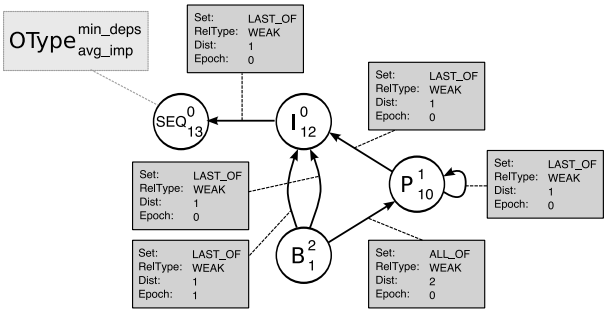

Figure 2: Type dependency graph for a simple MPEG-like stream.

4.3 Object Graph Decoration

While the type graph is statically created, the object graph is generated as new stream objects become visible. This process, called *graph decoration*, uses the dependency relations contained in the type graph and metadata contained in object labels if available. During decoration the function $decorate(G_O, G_T, o)$ adds the vertex v_o and eventually new adjacent edges to the object graph G_O. An edge $(v_o, v_{o'})$ is added *iff* one of the two following conditions is true: Either o' satisfies a dependency relation, defined in the type graph G_T for $otype(o)$ when the object label's reference list is empty, or o' is contained in the object's reference list and a dependency relation between $otype(o)$ and $otype(o')$ in G_T is satisfied.

If a lost object, an explicitly dropped object or any transitive reference of such objects is selected as reference, the new object is immediately marked to have a *broken dependency*. This information is later valuable for dependency validation. If a lost object is successfully retransmitted, the object graph is updated accordingly.

If objects are lost or corrupted, their labels are lost too. Decoration in this case requires reconstruction or prediction of metadata (at least of the object type) from missing sequence numbers. While type prediction is feasible and sufficient for strictly predictable streams, metadata of other streams is hard to reconstruct. Here, the decoration algorithm must ensure, that subsequent valid objects are decorated correctly, so that they won't get linked to wrong objects, prior to a gap of lost objects with unknown type, only because type and distance seem correct. This is only feasible having explicit reference lists. Thus, they are essential for loss robustness.

Since explicit reference lists are unrecoverable, decoration will be incomplete for a lost object, resulting in less reliable estimations of the model when such objects are involved. However, the model is able to recover quickly because even lost objects are able to collect dependants, which in turn reveals their actual importance.

4.4 Dependency Validation

The purpose of dependency validation is to ensure, that all necessary (object-based) and sufficient (type-based) dependency relations for a given object and its transitive ancestors are satisfied so that the object may be successfully decoded.

Dependency validation is implemented by the function $valid : O \mapsto \{true, false\}$, using the object graph and the type graph. Since the object graph is already prepared for validation by the decoration function, *valid* is able to efficiently check the following four conditions: A stream object $o \in O$ is valid *iff* it is (a) neither dropped nor lost itself, (b) has no broken dependency, (c) all its strong dependency relations are satisfied and (d) its minimum dependency is satisfied too.

When using the model for strictly predictable streams, dependency validation is even possible in the case of lost or corrupted objects. This is not because the dependency of the lost object itself can be reconstructed, which is unimportant anyway, but rather because the object type is predictable and decoration as well as validation of subsequent objects can rely on this information. When streams are unpredictable, the type of lost objects is not recoverable and type-based dependency alone becomes insufficient. Then, explicit object reference lists are required for compensation.

4.5 Importance Estimation

The importance of an object depends on the information encoded within the object itself as well as on the number and importance of objects which depend on this information directly or transitively. As with validation, the importance function $imp : O \mapsto \mathbb{N}$ uses information from both graphs to estimate the importance value. The importance of an

object is the maximum of its type importance and its meta-importance, additionally increased by the correction value stored in the object's label.

$$imp(o) = \max\Big(type\_imp(otype(o)), meta\_imp(o)\Big) + imp\_corr(o)$$

The *meta-importance* for the object o is defined to be the number of vertices in the transitive closure (TC) of o in G_O. The *type importance* for type t is defined to be the maximum of (a) the type-specific avg_imp value, (b) the number of vertices in the transitive closure of t in G_T increased by 1 or (c) the maximum of avg_imp over all vertices in the transitive closure increased by 1.

$$type\_imp(t) = \max \begin{cases} avg\_imp(t) \\ \Big|\, TC(G_T, t) \,\Big| + 1 \\ \max\Big(avg\_imp(t_i)\Big) + 1 \;\Big|\; t_i \in TC(G_T, t) \end{cases}$$

Unlike validation which uses backward dependency, importance estimation relies on forward dependency. Thus, exact importance values for every object can only be given in strictly predictable streams. In unpredictable streams exact importance is only available when all transitively depending objects are within the horizon. In contrast, it is impossible to give exact importance values if some depending objects are still invisible or lost. Consequently, a first estimation of object importance is initially based on the object type. As the horizon proceeds and loss is repaired, importance values for each object are refined to approach the exact values.

4.6 Model Conclusion

While the model enables format designers to express flexible constraints on static typed-based dependency as well as explicit object-based dependency, a developer needs to carefully consider the impacts of a stream's structural properties on general predictability, validation and importance estimation. When adopting the model in a real scenario, the effects subsume as follows:

- Type-based dependency is useful for partial dependency validation (in particular for strong dependency) and for initial importance estimation. It is sufficient if a stream is strictly predictable while it cannot avoid imprecise results if objects of unpredictable streams are lost. Since the type graph is static it may be statically validated and it must be exchanged only once prior to stream data.
- Object-based dependency is necessary for correct validation and precise importance estimation if streams are unpredictable or uncontrolled packet loss occurs. In combination with type-based dependency it compensates for incomplete knowledge in those cases.
- The size of the horizon solely depends on the radius of potential dependency within a stream. There is no interrelation between the model's horizon and the size of buffers required to store stream objects. Even if there is no stream buffer at all, the model is able to reason about the stream's recent past within the horizon limits.
- The model does not introduce extra delay besides its own execution time, since it simply caches information on stream objects, without requiring the objects to be actually stored.

5. IMPLEMENTATION

In order to separate the tasks of dependency specification and validation, we propose a *Dependency Description Language* (DDL) and a *Dependency Validation Service* (DVS), both based on the discussed dependency model. The DDL is used to specify type-based dependency for a given stream format at the design time of this format. This specification is used to initialise the DVS, which is intended to monitor and validate the state of a visible stream section as stream data arrives and departs at runtime.

5.1 Dependency Description Language

In order to specify object types, type-based dependency relations and constraints, we developed a simple description language and a parser for transforming the description into a type graph. The keyword TYPES specifies the set of object types while the keyword DEPENDENCY is used to define dependency rules as well as type attributes. Each dependency rule corresponds to a dependency relation in the type graph. It is of the form:

```
<SET SELECTOR> ( <TYPENAME>,
                 <DISTANCE>,
                 <EPOCH SELECTOR>,
                 <LAYER SELECTOR> ) <RELATION>;
```

The typename is one of the object types specified by the TYPES keyword. Set selector, distance, epoch selector, layer selector and relation are used as specified in section 4.2.

```
types := {seq_head, i_frame, p_frame, b_frame};
dependency(i_frame) := { last_of(seq_head) weak;
                         avg_imp(12); min_deps(0); };
dependency(p_frame) := { last_of(i_frame) weak;
                         last_of(p_frame) weak;
                         avg_imp(10); min_deps(1); };
dependency(b_frame) := { last_of(i_frame) weak;
                         # first I-frame in next epoch
                         last_of(i_frame, 0, 1) weak;
                         # one or two previous P-frames
                         # in the same epoch
                         all_of(p_frame, 2) weak;
                         min_deps(2); };
```

Figure 3: Dependency description for a simple MPEG-like stream.

Figure 3 shows a dependency description example for the typical MPEG stream displayed in figure 2 with an open GOP of average size 12 in transport order, having the form IPBBPBBPBB... (note that MPEG transport order slightly differs from encoding order to avoid forward dependency). The average GOP size is reflected in the average importance of the specified types.

5.2 Dependency Validation Service

The dependency validation service implements an instance of the dependency model and offers interfaces for manipulation, validation and estimation. It enables streaming protocol services such as scheduling, error protection and content scaling to reason about dependency and importance.

We assume, that every stream is partitioned into stream objects and that each object is properly labelled by the application. A service instance is initialised using a DDL description while the actual exchange of this description is beyond the scope of the service. As the stream flows, the DVS decorates the internal object graph and stores object states. Objects can be updated later when they are processed, dropped or retransmitted.

5.2.1 Interfaces

The interfaces of the DVS are organised into the three sections decoration, validation and estimation (see table 2). The methods implement the corresponding model functions *decorate*, *valid* and *imp*. They use the sequence number of stream objects as an index value. As new stream objects become visible or gaps in sequence numbers are detected, the user must call `insertObject` or `insertLostObject` to decorate the object graph. If a reordered or retransmitted object arrives later, the method `updateObject(seq, label)` must be used. It takes special care of properly updating the object graph. If an object is processed or dropped, a second form of `updateObject(seq, newstate)` is provided since it can be implemented more efficiently. Note that there is no special garbage collection method, since the dependency model implicitly manages its horizon based on dependency radius and object state.

Graph Decoration
`void insertObject(seq, label)`
`void insertLostObject(seq)`
`void updateObject(seq, label)`
`void updateObject(seq, newstate)`
Object Validation
`bool isValid(seq)`
`state getState(seq)`
`list<seq> listDependants(seq)`
Importance Estimation
`int getImportance(seq)`
`list<seq, imp> listByImportance(state)`

Table 2: Dependency validation service API.

For validation and inspection, the service offers the methods `isValid`, `getState` and `listDependants`. Importance estimation can be performed in two ways. Either the importance of a single known object is obtained by `getImportance` or a list of all currently known objects, sorted by importance, is obtained by `listByImportance`. The latter method confines the listing to objects in a given state. Thus a user can, for example, select the most important lost objects to consider them for retransmission.

5.2.2 Architecture and Cross-Layer Issues

The intended purpose of the dependency validation service is to make lower transport layer policies aware of the internal properties of media streams. This cross-layer approach differs from other cross-layer architectures, which mainly focus on optimising channel issues. Our approach is application-driven and focuses on format-independent content description. We believe that combining both approaches into a single cross-layer architecture is essential.

Figure 4 shows the dependency infrastructure, embedded into the application-driven part of such an architecture. While the format-independent dependency validation service is embedded into transport layers and used by error- and flow-control mechanisms, format-specific type description, stream fragmentation and object labelling are performed at the application level. In addition to traditional communication interfaces, a developer needs to provide the type description and the object labels.

Application and transport layers share the notion of labelled transport entities, the stream objects. This systematically extends the Application Level Framing concept [6] by a common set of metadata. While the DVS requires the transport layer to forward the labels, they are otherwise transparent. This can be achieved by RTP extension headers for example. However, transport layer strategies need to be changed in order to integrate with the DVS.

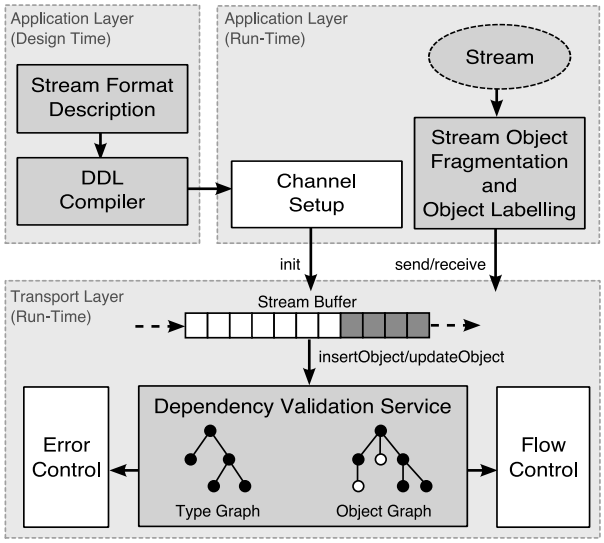

Figure 4: Cross-Layer design using the dependency validation service.

5.3 Embedding content-awareness into error control

In this section we outline on the example of error control policies, how transport-level services can become content-aware using the DVS. While the number of existing error control policies, protocols and algorithms for distributed multimedia streaming systems is immense, we do not consider any special mechanism or protocol. Instead we discuss three classes of basic error control mechanisms and outline how to make each class content-aware. We consider:

- **Protection Policies**, which preventively protect data against loss or corruption by adding redundancy or spreading the risk across multiple packets or paths.
- **Drop Policies**, which discard late or unimportant stream data in a controlled way in order to adapt a stream to resource constraints and relax overload situations.
- **Recovery Policies**, which systematically recover from loss or corruption, either by retransmissions or concealment. We focus on retransmission-based approaches here.

As a general example we consider a streaming system with a single stream sender and a single stream receiver connected by a lossy channel. Both parties use a private instance of the DVS for content-aware error control. Additionally both use a private stream buffer to store stream objects. As new stream objects arrive in the buffer and others get processed, the DVS is used to validate dependency and provide importance estimations. A sender can incorporate this into content-aware drop, importance-aware packet scheduling and unequal error protection, while a receiver can use selective retransmissions, based on object importance.

5.3.1 Content-aware Error Protection

Error protection policies are proactive and sender-based. Content-awareness enables them to protect important stream objects stronger than less important ones. Since importance is type- and context-specific, objects of the same type may have different importance values in different contexts. Consequently an error protection policy needs to interpret importance by own means.

Content-aware scheduling [15] and FEC mechanisms [9] for example would use the `getImportance` to obtain the importance value of an object in question or `listByImportance` to get a list of all objects, ordered by importance.

5.3.2 Content-aware Drop Policies

Selective drop policies such as [13] handle three kinds of problems by systematically discarding stream objects: buffer overflows, missed deadlines and broken dependency. If buffers run full, a content-aware drop policy needs to select one or more objects to discard. Drop policies usually prefer to drop objects with the least severe impact on signal fidelity, in other words the least important objects within the horizon. For such policies the DVS offers the `listByImportance` operation. Since importance values are compared here, their absolute value is not relevant. An alternative drop policy can choose objects which missed their deadlines. However, if any depending object is still in time the reference object should not be dropped until all references are eventually late. Using the `listDependants` operation the DVS creates a dependency list for a given object. This list contains only objects which are already visible and does not include lost or dropped ones.

Explicitly dropping an object can create broken dependencies. Since our dependency model assigns importance according to actual dependency, each policy which drops the least important objects first implicitly avoids broken dependencies within the visible horizon. However, future objects still outside the horizon are likely to experience broken dependencies after drop. Hence, a drop policy must inform the DVS when discarding objects via the `updateObject` operation. To detect a broken dependency for arriving objects, `isValid` can be used directly after inserting the new object.

A receiver suffering from channel loss will experience broken dependencies too. If a previous reference object was lost, it may be necessary to drop other objects too. However in combination with retransmission policies, this decision can be postponed until the objects need to be processed at the receiver side.

5.3.3 Content-aware Retransmission Policies

In selective retransmission policies [16, 17, 3, 18] either the receiver, the sender or both decide on the handling of lost data. We do not consider how a retransmission protocol is structured or if it is ACK/NACK-based. Instead we outline a content-aware decision making process.

If a receiver detects a lost object he may use the importance value provided by the `getImportance` operation to reason about a selective retransmission. For predictable streams, the estimation will always be correct, while for unpredictable streams the initial estimations are less reliable as discussed in section 4.4. Therefore the retransmission decision may be postponed until more objects become visible. Another way to obtain importance information regarding lost objects is the `listByImportance` operation, which optionally selects objects in a given state.

A sender-based approach for retransmission benefits from a more complete knowledge about dependency since the sender's horizon is usually larger than the horizon of the receiver and it contains no lost objects. Thus the importance estimation provided by `getImportance` is fairly exact for senders. If a sender is informed by the receiver about loss, he might selectively retransmit the most important objects instead of all missing ones.

6. ANALYTICAL EVALUATION

While costs and benefits of a dependency model are only quantifiable in conjunction with real streams and policies, we report on the analytical worst-case costs of our model here. These costs include the runtime and storage complexity of a model instance as used by the DVS, namely the costs for decorating the object graph, validating dependency and estimating importance. In this respect, the following properties of the type description and the stream itself are significant (see table 3).

Type-based Properties	
T	number of types in the type graph
R	number of dependency relations in the type graph
Stream-based Properties	
S	size of the stream buffer
L	maximal number of entries in object dependency list

Table 3: Significant cost factors.

Since the dependency model caches information regarding stream object labels only, the payload size of the objects is irrelevant. So, for every object there is a small and constant space complexity, subsuming to $O(S)$ for the object labels and the object state. However, the graph representations need additional but fixed storage too. Worst-case space complexity for the type graph is $O(T + R)$ since it requires list representations due to parallel edges and $O(S^2)$ for the object graph which may be implemented using an adjacency matrix.

The runtime costs differ between the case where implicit type-based dependency is available only and the case where explicit object-based dependency is expressed in object labels. Therefore table 4 shows worst-case costs for naive straight-forward algorithms in both cases. Since the model is intended to provide validation and estimation for any new inserted object, graph decoration is performed during `insertObject` and `updateObject`. If this is unnecessary or loss and retransmissions are frequent, the costs may be postponed until either validation or estimation operation are called.

Operation	Implicit Dependency	Explicit Dependency
insertObject	$O(R \times S)$	$O(R \times L)$
insertLostObject	$O(1)$	$O(1)$
updateObject (retransmit)	$O(R \times S^2)$	$O(S^2 + S \times R)$
updateObject (drop)	$O(S^2)$	$O(S^2)$
updateObject (processed)	$O(1)$	$O(1)$
isValid	$O(1)$	$O(1)$
getState	$O(1)$	$O(1)$
listDependants	$O(S^2)$	$O(S^2)$
getImportance	$O(S^2)$	$O(S^2)$
listByImportance	$O(S^3)$	$O(S^3)$

Table 4: Worst-case costs for DVS operations.

These worst-case costs are considerably larger than average case costs for real streams, since (a) only a fraction of all dependency relations must be checked for a single object type, (b) the horizon is usually smaller than its maximal size and (c) the intended algorithms are not optimised to the special case of the dependency graphs. In general, the operations for update and importance estimation suffer the algorithmic costs of finding the transitive closure in the object graph (in the class of $O(S^2)$). Algorithmic optimisation and specialisation may yield improvements here.

7. CONCLUSION

In this paper we presented a universal format-independent dependency framework for multimedia streams and discussed its benefits for cross-layer optimised error control policies. Based on a formal dependency model, we developed a universal dependency description language and a robust dependency validation service. This framework enables application layers and lower transport layers to express, validate and reason about structural dependency in media streams and the impact of data loss and drops during transmission. Since our framework is based on simple graphs and relations, it may be implemented efficiently.

The expressiveness of our model allows format designers to specify dependency relations for virtually any media format, including most layered and scalable formats. The model is robust against packet loss and deals with incomplete knowledge introduced by the unpredictability of streams. While designing the model we learned, that unpredictability of current streaming formats constricts correct validation and importance estimation if based on static type relations only. In consequence, stream data needs to be explicitly labelled to correctly reason about dependency.

While in this paper we only considered essential data dependency for enabling content-awareness, other forms of dependency as well as information on timing and rate-distortion are equally important. The extension of this model towards those concepts remains an issue for future work. Based on the presented infrastructure we are currently implementing a multimedia middleware platform and a novel content-aware IPC mechanism for operating system kernels, including content-aware error policies. Both platforms are supposed to demonstrate the real costs and the utility of our dependency framework.

8. REFERENCES

[1] J.-C. Bolot and T. Turletti. Adaptive Error Control for Packet Video in the Internet. In *Proceedings of CIP '96*, Lausanne, 1996.

[2] H. Cai, B. Zeng, G. Shen, and S. Li. Error-Resilient Unequal Protection of Fine Granularity Scalable Video Bitstreams. In *Proceedings of the ICC*, Paris, France, 2004.

[3] J. Chakareski and P. A. Chou. Application Layer Error Correction Coding for Rate-Distortion Optimized Streaming to Wireless Clients. In *IEEE ICASSP*, pages 2513–2516, 2002.

[4] J. Chakareski, S. Han, and B. Girod. Layered Coding vs. Multiple Descriptions for Video Streaming over Multiple Paths. In *ACM Multimedia '03*, pages 422–431, 2003.

[5] P. Chou and Z. Miao. Rate-Distortion Optimized Streaming of Packetized Media. Technical Report MSR-TR-2001-35, Microsoft Research, February 2001.

[6] D. Clark and D. L. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proceedings of the ACM Symposium on Communication Architectures and Protocols*, Philadelphia, 1990.

[7] S. Devillers, C. Timmerer, J. Heuer, and H. Hellwagner. Bitstream Syntax Description-Based Adaptation in Streaming and Constrained Environments. *IEEE Transactions on Multimedia*, 7:463–470, 2005.

[8] A. Eleftheriadis and D. Hong. Flavor: A Formal Language for Audio-Visual Object Representation. In *ACM Multimedia*, New York, NY, USA, 2004.

[9] M. Etoh and T. Yoshimura. Advances in Wireless Video Delivery. *Proceedings of the IEEE, Special Issue on Advances in Video Coding and Delivery*, 93(1):111–122, January 2005.

[10] M. Hoffmann and W. E. Kühnhauser. Towards a Structure-Aware Failure Semantics for Streaming Media Communication Models. *Journal of Parallel and Distributed Computing*, 65(9):1047–1056, 2005.

[11] D. Isovic and G. Fohler. Quality Aware MPEG-2 Stream Adaptation in Resource-Constrained Systems. In *Proceedings of ECRTS'04*, Catania, Italy, 2004.

[12] J. Korhonen and R. Jorvinen. Packetization Scheme for Streaming High-Quality Audio over Wireless Links. *Lecture Notes in Computer Science*, 2899:42–53, 2003.

[13] C. Krasic and J. Walpole. Quality-Adaptive Media Streaming by Priority Drop. In *Proc. of NOSSDAV'03*, Monterey, CA, USA, 2003.

[14] K. Mayer-Patel, L. Le, and G. Carle. An MPEG Performance Model and its Application to Adaptive Forward Error Correction. In *ACM Multimedia '02*, pages 1–10, 2002.

[15] M.Röder, J.Cardinal, and R.Hamzaoui. Efficient Rate-Distortion Optimized Media Streaming for Tree-Reducible Packet Dependencies. In *Proceedings of MMCN*, San Jose, California, 2006.

[16] C. Papadopoulos and G. M. Parulkar. Retransmission-Based Error Control for Continuous Media Applications. In *Proc. of NOSSDAV'96*, 1996.

[17] M. G. Podolsky, S. McCanne, and M. Vetterli. Soft ARQ for Layered Streaming Media. *Journal on VLSI Signal Processing Systems*, 27(1-2):81–97, 2001.

[18] H. Seferoglu, Y. Altunbasak, O. Gurbuz, and O. Ercetin. Rate-Distortion Optimized Joint ARQ-FEC Scheme for Real-time Wireless Multimedia. In *Proceedings of IEEE Int. Conf. on Communications (ICC)*, 2005.

[19] G. Sullivan and T. Wiegand. Video Compression - From Concepts to the H.264/AVC Standard. *Proceedings of the IEEE, Special Issue on Advances in Video Coding and Delivery*, 93(1):18–31, January 2005.

[20] B. Wah, X. Su, and D. Lin. A Survey of Error-Concealment Schemes for Real-Time Audio and Video Transmissions over the Internet. In *Proceedings of the International Symposium on Multimedia Software Engineering*, pages 17–24, Taipei, Taiwan, 2000.

[21] Y. Wang, W. Huang, and J. Korhonen. A Framework for Robust and Scalable Audio Streaming. In *Proceedings of ACM Multimedia 2004*, New York, NY, USA, 2004.

[22] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.