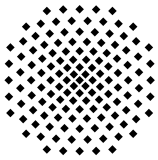
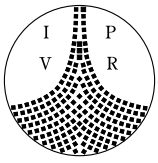


Universität Stuttgart
Fakultät Informatik



Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams

Kurt Rothermel, Tobias Helbig

CR-Klassifikation: C.2.4, D.2.2, H.5.1

Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams

Kurt Rothermel, Tobias Helbig

Fakultätsbericht 2/1994
Technical Report
April 1994

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20 - 22
D-70565 Stuttgart

Abstract

Synchronization plays an important role in multimedia systems at various levels of abstraction. In this paper, we propose a set of powerful abstractions for controlling and synchronizing continuous media streams in distributed environments. The proposed abstractions are based on a very general computation model, which allows media streams to be processed (i.e. produced, consumed or transformed) by arbitrarily structured networks of linked components. Further, compound components can be composed from existing ones to provide higher levels of abstractions.

The clock abstraction is provided to control individual media streams, i.e. streams can be started, paused or scaled by issuing the appropriate clock operations. Clock hierarchies are used to hierarchically group related streams, where each clock in the hierarchy identifies and controls a certain (sub)group of streams. Control and synchronization requirements can be expressed in a uniform manner by associating group members with control or sync attributes. An important property of the concept of clock hierarchies is that it can be combined in a natural way with component nesting.

1 INTRODUCTION

Powerful programming abstractions are a prerequisite for an effective and efficient application development. Application-specific abstractions are typically provided by development platforms, often referred to as middle ware. In the context of multimedia, those platforms close the gap between the operating system and the communication system on the one hand and the specific needs of distributed multimedia applications on the other hand. The *CINEMA* (Configurable INtEgrated Multimedia Architecture) system [RBH94], which is under development at the University of Stuttgart, is a platform providing system services for the configuration of distributed multimedia applications and the communication and synchronization of multimedia information in distributed environments.

Multimedia synchronization can be considered at different levels of abstraction [MeES93]. In this paper, we will focus on the control and synchronization of groups of continuous media streams, such as digital video and audio streams. Media streams themselves may be regarded at different abstraction levels. At the transport level, a stream usually originates at a single source and ends at one or more sinks. Further, sinks and sources are adjacent in the sense that each sink of the stream consumes the data produced by the stream's source. Therefore, at the transport level end-to-end relationships are defined between adjacent entities connected by a transport connection. If streams are considered at the application level instead, sources and sinks need not be adjacent at all. In general, a stream may be processed by a network of linked components, it may have multiple sources as well as multiple sinks, and each path leading from a source to a sink may involve several intermediate components. Consequently, at the application level an end-to-end relationship may cover any number of intermediate components as well as several transport connections at a lower level of abstraction.

This paper proposes programming abstractions for grouping, controlling and synchronizing application-level streams in distributed environments. Media clocks provide the basic abstraction for controlling the flow of media streams, i.e. by issuing clock operations the controlled streams can be started, paused or scaled as required. Related streams can be hierarchically grouped by building up so-called clock hierarchies, where each clock controls either an individual stream or a group of streams. Within clock hierarchies, two types of relationships can be defined for the members of a stream group, a control or sync relationship. If the control relationship is specified, the members of the groups are controlled collectively without synchronization of the streams. If the sync relationship is defined instead, the members of this group are processed (e.g. played out) synchronously.

A great advantage of the concept of clock hierarchies is that it can be combined with component nesting in a natural way. In order to provide higher levels of abstractions, more complex components, so-called compound components, can be composed from existing ones. The internal processing of a compound component is controlled and synchronized by means of included clock hierarchies, which are an integral part of the compound components. It is important to stress that this paper describes programming abstraction rather than the protocols and mechanisms implementing them. For a description of the underlying protocols we refer to a companion paper [RoHe94].

The remainder of the paper is structured as follows. The next section gives a brief overview of related work, and then the computation model the proposed abstractions are based upon is described in Sec. 3. We introduce the concept of a media clock in Sec. 4 and show how it can be used to control individual streams. The concept of a clock hierarchy, which provides the means for controlling and synchronizing groups of media streams, is presented in Sec. 5. While this section mainly considers clocks attached to sink components, Sec. 6 motivates and treats clocks attached to sources. In Sec. 7, we discuss how the proposed abstractions can be applied in the context of component nesting. Finally, we conclude with a brief summary.

2 RELATED WORK

As stated above, streams can be considered at different levels of abstraction. Abstractions for grouping and controlling transport-level streams are provided by the orchestration service [CCGH92]. This service allows for grouping streams and coordinating the flow of (flat) groups of streams. In particular, the streams of a group can be started and stopped collectively, while the flow rate of the streams is regulated individually. The orchestration service itself does not guarantee stream synchronization but offers a general regulation mechanism that can be used at higher layers to implement different synchronization policies.

Various abstractions for controlling groups of application-level streams have been proposed in the literature. Some of these proposals apply to non-distributed environments only (e.g. QuickTime [Appl91] or IBM's Multimedia presentation manager [IBM92]), and others are tailored to specific configurations (e.g. ACME [AnHo91] and Tactus [DNNR92]). ACME, for example, is an extension of a network window system supporting streams of digital audio and video data. The clients of the ACME server use the abstraction of a logical time system to control and synchronize the output of a (flat) group of ropes.

The Multimedia System Services proposed by the Interactive Multimedia Association (IMA) [IMA93] are based on a very general computation model and provide a rich set of abstractions for grouping and controlling media streams. The purpose of these services is to provide an environment in which a heterogeneous set of multimedia computing platforms cooperates to support distributed, interactive multimedia applications dealing with synchronized, time-based media. In this environment, the abstraction of a group is used to group related media streams. Group objects, which may include other group objects, provide an interface for controlling the streams belonging to this group. This means that an entire group of streams can be started, paused or scaled by issuing single operation at the group interface. However, the streams of a group are not synchronized. In the current proposal, stream synchronization is not yet integrated in the group mechanism. Moreover, component nesting is not supported.

3 COMPUTATION MODEL

In this section, we briefly sketch the computation model of *CINEMA* (for more details see [RBH94]). The major concepts of this model are media streams, components, ports, links, sessions and clocks.

A continuous **media stream** is defined to be a sequence of data units, each of which is associated with a media time stamp (e.g. see [Herr91]). **Components** are active entities that process continuous media streams in various ways. We distinguish between source components, which produce media streams, sink components, which consume media streams, and intermediate components, which act as both producers and consumers. Components are associated with typed **ports**. While a producer writes stream data to its output ports, a consumer reads stream items from its input ports.

Applications are configured by defining **links** between input and output ports of components. An example configuration consisting of one intermediate component and three source and sink components is shown in Fig. 1. This configuration mechanism has proven to be powerful and hence can be found in various other architectures as well (e.g. Conic project [MKS89], IMA [IMA93], Quicktime [Appl91], SUMO project [CBRS93]).

While link objects are applied to define the topology of applications, **sessions** are the abstraction for resource allocation. Media streams can be processed and communicated only after the corresponding sessions have been established. A session may comprise multiple sink and source components and any number of intermediate components. QoS of a session is specified

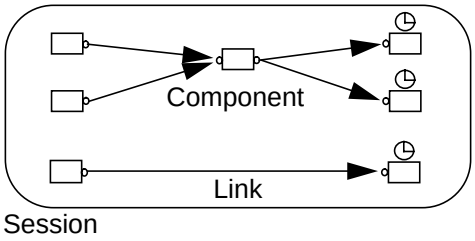


Figure 1 : An Example Application

at the session end points, i.e. at the sink components. For controlling the flow of media streams an extra abstraction, so-called **media clocks**, is provided. Media clocks are used to start, pause, or scale media streams.

CINEMA supports the nesting of components. In other words, basic components can be composed to build more complex components, called compound components. Compound components may again be constituents of other components, i.e. arbitrary levels of nesting are possible. Compound components provide the means for building higher levels of abstraction on the basis of existing components.

The computation model described above is rather general and has various similarities with other architectures (e.g. IMA [IMA93] or SUMO [CBRS93]). Therefore, the concepts presented in the remainder of the paper are not only relevant in the *CINEMA* context but are applicable in a rather broad scope.

4 MEDIA CLOCKS

The temporal dimension of continuous media streams is defined by so-called media time systems. The media time system associated with a stream is the temporal framework to determine the media time of the stream's data units. In *CINEMA*, media time systems are provided by media clocks (or clocks for short). A clock *C* is defined as follows:

$$C ::= (R, M, T, S)$$

The clock attributes have the following meaning:

- *R* determines the ratio between real time and media time: *R* time units in media time correspond to 1 second in real time.

- *M* is the start value of the clock in media time, i.e. the value of the clock at the first clock tick.

- *T* is the start time of the clock in real time, i.e. the real time of the first clock tick.

- *S* determines the speed of the clock: *S***R* time units in media time correspond to one second in real time. Consequently, media time progresses in normal speed if *S* equals 1. A speed greater than 1 causes the clock to move faster, a speed less than 1 causes it to progress slower, and a negative speed causes it to move backwards.

It should be noted that the temporal dimension of stored media is inherently bound, i.e. there exists a lower and upper bound given by the media time of the first and last stream data unit. In other words, media time for a given stream is only defined in a certain interval. The mechanisms required to ensure that clock values stay within the defined time range are beyond the scope of this paper.

Media time systems are a general concept to dimension media time in arbitrary ways. For the following example, assume a (stored) video stream with a rate of 25 data units per second. If ratio $R = 25$, media time corresponds to a frame sequence number, e.g. if $M = 5$, then stream processing is started with the 5th frame in the stream provided the lower bound of its temporal dimension is 1. If media time is counted in milliseconds instead, *R* is set to 1000. In each case, ratio *R* defines the “normal” speed of media time, while attribute *S* can be used to speed up or slow down the progress of media time.

A clock relates media time to real time as shown in Fig. 2. Therefore, after a clock has been started, media time (*m*) can be derived from real time (*t*):

$$m = M + S \cdot R\,(t - T)$$

Clocks are the basic abstraction for controlling the flow of media streams. As will be seen below, clock objects provide methods for starting, pausing, or scaling streams. In *Cinema*, clocks may be attached to source and sink components, but never to intermediate ones. As in *Cinema* stream processing can take place in arbitrary networks of interconnected components, a stream may consist of a number of substreams and may originate form multiple sources and

end at multiple sinks. Those complex streams are controlled by manipulating the clocks at the sinks and sources.

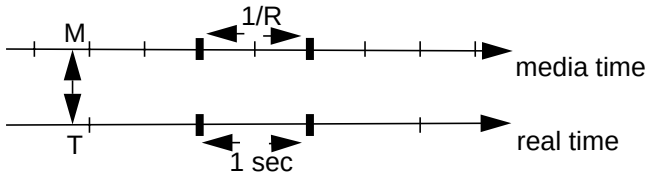


Figure 2 : Mapping Media Time To Real Time

While clocks at source components are optional, they are mandatory at sink components. A clock attached to a sink component controls the temporal progress of all data streams processed (e.g. played out) by this component. This is expressed more precisely by the so-called **clock condition**: a data unit having media time *m* is processed at real time *t* only if the controlling clock is ticking and its value equals *m* at time *t*. Conceptually, this means that the presentation of a stream is started, paused or scaled when the controlling clock is started, halted or the clock speed is changed, respectively. The semantics of clocks at source components will be introduced later.

As pointed out above, media time progresses relative to real time. In *CINEMA*, real time is taken either from a local system clock, a global clock (e.g. see NTP [Mill89]) or is derived from the temporal behavior of a given output device. Clearly, a media clock based on the timing of an output device advances in conformance with the device's natural rate. Those clocks are called master clocks.

Below, the most important clock operations for controlling streams are listed. A clock may enter two states, *ticking* (and thus advancing) or *silent* (and thus not advancing). The only clock operations that cause state transitions are `Start` and `Halt`. The former moves the clock from *silent* to *ticking*, while the latter causes the reverse state transition.

`Start(M)` This operation starts the clock at media time `M`. By starting the clock the controlled stream(s) are started. (Clock attribute *T* is set to the real time at which the clock is actually started).

`Halt(M)` This operation halts the clock when it reaches clock value `M`, i.e. the

stream(s) controlled by this clock are paused. A halted clock can be started again by operation `Start`.

`Prepare(M)` This operation prepares the event of starting the clock at media time `M`. After `Prepare` has been performed, the clock can be started immediately when `Start` is issued. To achieve this, `Prepare` preloads the buffers along the communication paths of the controlled stream(s). If this operation is not invoked, preloading is done implicitly as part of `Start`.

`Clear()` This operation clears the internal buffers associated with the controlled stream(s).

`Scale(M,S)` The default value of the clock speed equals `1`. This operation changes the speed of the clock to `S` when media time `M` is reached, i.e. it scales the stream(s) controlled by the clock.

`Lock(O)` This operation locks the clock for propagated operations of type `O`. This operation is only applied in the context of clock hierarchies.

`Unlock(O)` This operation unlocks the clock for propagated operations of type `O`.

In the simple scenario shown in Fig. 3, clock *C* controls the presentation of a 25 frames/sec video stream.

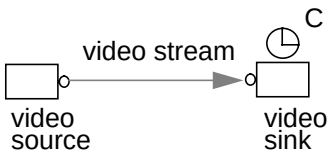


Figure 3 : Controlling a Video Stream

```
1   C.Start(15)
2   C.Scale(3000,2)
3   C.Halt(5000)
```

If we assume that clock attribute *R* equals 25, then play out is started with frame 15, the play out rate is doubled when the presentation reaches frame 3000, and the presentation is halted after frame 5000 has been played out.

5 CLOCK HIERARCHIES

In this section, we will introduce the notion of a clock hierarchy, which is the basic abstraction for grouping media streams, controlling groups of streams, and stream synchronization. The principle idea of this concept has been introduced in [RoDe92].

Related media streams may be grouped by linking clocks in a hierarchical fashion. Remember that a clock attached to a component controls all streams processed by this component. A number of streams can be grouped by linking their controlling clocks to a common clock, which then controls the entire group. Stream groups can be grouped again to groups at a higher level simply by linking their controlling clocks to the same clock. In the example given in Fig. 4, clock C_6 controls streams S_1 and S_2, while C_7 controls S_4 and S_5. C_8 controls the subgroups represented by C_6 and C_7 as well as stream S_3, and thus all streams in the given scenario can be started, halted or scaled collectively by means of this clock. Since *CINEMA* supports arbitrarily structured clock hierarchies, any type of hierarchical grouping of media streams is possible.

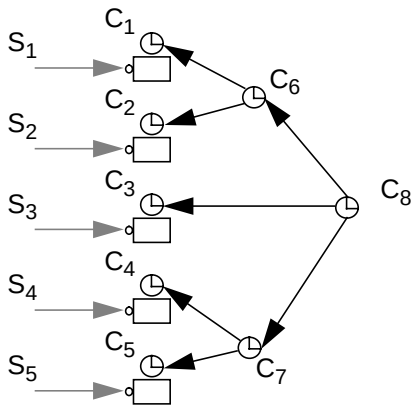


Figure 4 : Grouping Streams

A clock operation issued at a clock not only affects this clock but the entire (sub)hierarchy of this clock. Conceptually, an operation called at a clock is **propagated** in a root-to-leaf direction through the clock's (sub)hierarchy, where it is performed at every clock in this hierarchy. That is, an operation invoked at a clock is not only performed at this clock but also at every descendant clock in the hierarchy. In general, clock operations can be issued at every level of the clock hierarchy. If operation `Start` is issued at C_6 in the example depicted in Fig. 4, this operation is propagated to C_1 and C_2, which causes streams S_1 and S_2 to be started. All streams

in the depicted scenario are started if `Start` is invoked at C_8 instead. As will be seen in Sec. 7, propagation is a prerequisite for component nesting. Compound components may contain clock subhierarchies which are invisible for the component's outside world.

In some scenarios, it is desirable to lock clock subhierarchies in order to prevent propagation. For that purpose, clocks may be locked and unlocked. If a clock is locked, propagation of operations issued at ancestor clocks does not take place in the clock's (sub)hierarchy. Note that only operations propagated from a locked clock's ancestors are locked out, while all operations issued at the locked clock itself or one of its descendant clocks are performed and affect the hierarchy in the usual way. Propagation is enabled again only when the clock is unlocked.

Locking is done in an operation-specific manner. Each lock is associated with a certain type of clock operation and only locks out operations of this type. In other words, a lock defines an operation-specific filter.

Locking is especially useful in those scenarios, where multiple users are involved in the same application and hence clock hierarchies typically cover several user domains. Here, locking provides a means to shield clock subhierarchies located in a given user domain from propagated clock operations originated in some other user domain. Consequently, by locking clocks a user can dynamically control which types of propagated operations may influence the data streams in his or her domain.

Clocks may be linked in two different ways: a link may establish either a **control** or a **synchronization** relationship between two clocks. A control relationship between two clocks enables the propagation of clock operations without synchronizing the two clocks. Typically, control relationships are defined in settings, where groups of streams are to be controlled collectively and a rather loose temporal coupling of the grouped streams is sufficient. A synchronization relationship goes a step further. In addition to propagation, it ensures that the involved clocks progress in a synchronized manner.

In *Cinema*, stream synchronization is specified by means of sync relationships between clocks. From the clock condition introduced in the previous section directly follows that two streams are synchronized if their controlling clocks are synchronized. In the example shown in Fig. 4, streams S_1 and S_2 are played out synchronously if C_1 and C_2 are synchronized. This synchronization requirement can be specified by a sync relationship between C_1 and C_6 as well as one between C_2 and C_6. An alternative way to express the same is to define a sync relationship directly between C_1 and C_2.

Clocks provide individual media time systems, which may relate to each other in various ways. Clock synchronization and propagation of clock operations (as will be seen below) is done on the basis of so-called **reference points**. A reference point defines the temporal relationship of two media time systems. More precisely, reference point [$C_1 : P_1$, $C_2 : P_2$] defines that media time P_1 in C_1's time system corresponds to media time P_2 in C_2's time system, which means that P_1 and P_2 relate to the same point in real time (see Fig. 5). Given this reference point, media time can be transformed from one to the other time system as follows:

$$m_2 = (m_1 - P_1) \cdot \frac{R_2}{R_1} + P_2$$

After having introduced the basic principles, we can now take a closer look at clock hierarchies. A clock hierarchy is a directed tree structure, where the nodes are clocks and the edges represent control or sync relationships between clocks. The same hierarchy may contain control as well sync edges. Each **edge** is associated with the following attributes:

- Type of the edge, which is either control or sync.
- A reference point which defines the temporal relationship between the clocks linked by this edge.
- A delay attribute that specifies how long an operation propagated along this edge is to be delayed. In the example shown in Fig. 4 streams S_4 and S_5 are started 3 seconds later than the other streams if the `Start` operation is delayed by 3 seconds while propagated from C_8 to C_7. Obviously, the provision of this delay attribute enhances the flexibility of our scheme substantially. For the sake of simplicity, we will assume a zero delay in the following examples.

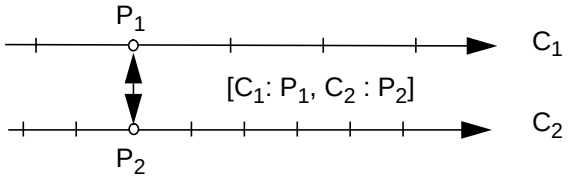


Figure 5 : Transforming Media Time

Before describing the semantics of control and sync edges more precisely, we have to introduce function *Trans*(C_i, C_j, m_i). In a given clock hierarchy, this function transforms media time m_i from C_i's to C_j's time system according to the equation above.

5.1 Control Relationship

The semantics of a control edge that is directed from a clock, say C_1, to another clock, say C_2, and is associated with a reference point *RP* is defined by the following rules:

1. Each clock operation issued at C_1 is propagated to C_2's subhierarchy provided C_2 is unlocked[1].

2. Whenever a clock operation is propagated, its media time arguments are automatically transformed from C_1's to C_2's media time according to reference point *RP*. That is, argument *m* in a propagated operation is transformed to *Trans*(C_1, C_2, *m*)

3. Each clock operation can be issued at any clock in the control hierarchy.

4. A `Start` operation issued at clock C_2 may be performed immediately independent of C_1's value or state (*ticking* or *silent*). That is, C_2.`Start`(*m*) starts C_2 immediately with initial clock value *m*.

It is important to point out that control hierarchies only allow for a very loose coupling of streams. Although a control hierarchy includes reference points defining the temporal relationship between streams, this information is not used to keep the controlled streams synchronized. Reference point information is considered only when clock operations are propagated. In particular, it is used to automatically transform operation arguments from one to another media time system. For example, when a group of streams is started, the stream's media start times conform to the reference points defined in the corresponding control hierarchy. However, after a hierarchy has been started, its clocks may drift out of synchronization in an uncontrolled manner. Clocks in a hierarchy may drift, for example, if they are based on different physical time systems (e.g. system clocks or device-internal clocks). Moreover, in control hierarchies, each clock may be manipulated without considering the state and value of the parent clock. For

[1] No guarantee is given that a clock operation and its propagated ones are performed at the same point in real time. However, they are performed at "approximately" the same time; what this means in practice mainly depends on the underlying implementation of the control mechanism.

example, two different subhierarchies of the same hierarchy may be scaled in different ways, or clocks in the hierarchy may be halted and continued at any later time with arbitrary start values.

Due to the fact of potentially drifting clocks, for operations `Halt` and `Scale` different semantics are conceivable. If, for example, *C*.`Halt`(*Now*) is performed, then all streams controlled by the clocks in C's subhierarchy are halted immediately because *Now* - per definition - corresponds to the current time in each media time system. If *C*.`Halt`(30) is specified instead, the different clocks may reach the equivalent of 30 in their media time systems at different points in real time. One reasonable semantic of the operation is to pause all streams when the first clock reaches the given halting time. Due to space limitations, a detailed discussion of this subject is out of the scope of this paper.

5.2 Sync Relationship

The semantics of a sync edge that is directed from a clock, say C_1, to another clock, say C_2, and that is associated with a reference point *RP* is defined by the following rules:

1. Each clock operation issued at C_1 is propagated to C_2's subhierarchy provided C_2 is unlocked.

2. If C_2 is *ticking*, both clocks C_1 and C_2 are progressing in a synchronized manner, where the sync relationship is defined by reference point *RP*. More precisely: Assume that *ISet* denotes the set of real time intervals during which C_2 is in the *ticking* state. Then C_1 and C_2 are defined to be synchronized[1] if

   $$\forall\ \mathrm{I} \in \mathit{ISet}\ \forall\ t \in I : C_2(t) = m_2 \Rightarrow C_1(t) = \mathit{Trans}(C_2, C_1, m_2),$$
   where $C(t)$ denotes the value of C at time t.

3. Except `Scale` each clock operation can be issued at every clock in the sync hierarchy. `Scale` can be issued at the root clock only, i.e. only the entire hierarchy can be scaled.

4. Operation `Start` can be issued at C_2 only if C_1 is in the ticking state. In order to ensure clock synchronization, the start of C_2 has to be synchronized with the progress of C_1's

[1] Of course, in practice, clocks can and need not be synchronized exactly. Thus, in *CINEMA* an upperbound for a tolerable skew can be specified at each sync edge.

media time: C_2.`Start`(m) is delayed until C_1's clock value equals *Trans*(C_2 ,C_1, *m*). An alternative way of starting C_2 is to specify start time *Now* in operation `Start`. In this case C_2 is started immediately, say at real time *t*, with clock value *Trans*(C_1, C_2, m_1), where m_1 is C_1's clock value at time *t*.

5. A sync hierarchy may contain at most one master clock, which must be the root of the hierarchy.

Sync hierarchies are a general and very powerful concept to specify arbitrary synchronization requirements between media streams. The structure of the sync hierarchy specifies which streams have to be synchronized, while the reference points in the hierarchy define how streams have to synchronized, i.e. how the temporal dimension of the streams relate to each other.

The system guarantees that all streams controlled by the clocks in the sync hierarchy are processed (e.g. played out) in a synchronous manner. Processing is started by issuing operation `Start` at the root clock of the sync hierarchy. A locked subhierarchy can be started later by issuing `Start` at the subhierarchy's root clock. The start of this subhierarchy is performed in conformance with the temporal constraints specified by the entire sync hierarchy. The same holds if a subhierarchy is halted and started once again at a later point in time. As will be seen later, sync (and control) hierarchies may dynamically grow and shrink even if clocks are *ticking*. This feature together with the capability of locking, halting and starting individual subhierarchies is very important in interactive applications, especially in those, where multiple users with their individual needs participate in the same (CSCW) application.

5.3 Example

Fig. 6 shows a simple tele cooperation scenario with two users. Subject to the cooperation is an experiment shown on video V_2. We assume that there exist extra speech channels that allow the users to talk to each other. The two users commonly view V_2 and discuss the experiment while they follow the presentation. To ensure that both users see the same information at the same time, V_2 must be played out synchronously at both user sites. Besides V_2, user 1 views video V_1, which shows the same experiment from a different perspective. Consequently, V_1 and V_2 are to be synchronized. User 2 additionally views video V_3, which shows a similar experiment. Since the two experiments roughly correspond to each other in their temporal dimension, V_1

and V_3 are grouped by a control relationship.We assume that media time 500 in V_3 corresponds to media time 5 in V_2.

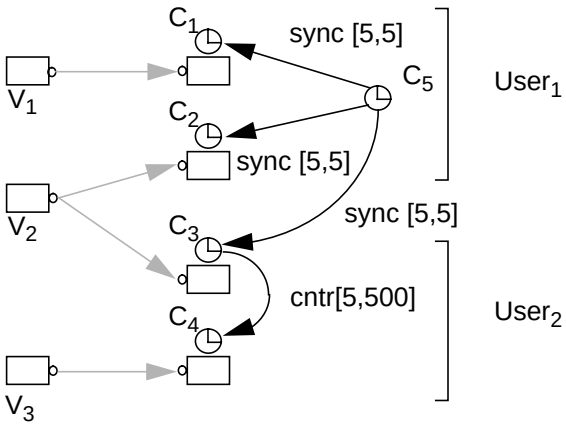


Figure 6 : A Simple Telecooperation Scenario

The presentation of all video streams can be started by issuing `Start` at clock C_5. Moreover, this clock can be used to collectively scale, pause and restart the entire configuration. User 1 may pause V_1 or V_2 by halting C_1 or C_2, respectively. Halted clocks may be continued in a synchronized fashion, i.e. after restart of C_2, for example, the presentation of V_2 is not only synchronized with V_1 but also with V_2's presentation at the site of user 2.

Since C_3 and C_4 are linked with a control edge, V_3 can be scaled, paused and restarted at any position independent of V_1's and V_2's state of the presentation. So, the presentation V_3 can be adjusted manually as needed. At user site 2, halting C_3 implies pausing V_3's presentation. If this is to be avoided, C_4 has to be locked for `Halt` operations. Note that `Scale` operations issued at C_5, for instance, are then still propagated to C_4.

If another user desires to join the scenario while cooperation already takes place between user 1 and user 2, the clock hierarchy has to be extended dynamically. Assume that the new user needs to view V_2 only. After the corresponding session has been established, the clock, say C_6, controlling V_2's presentation at the site of the new user is linked by means of a sync edge to clock C_5. When the new user is ready to participate in the cooperation, it issues C_6`.Start(`*Now*`)` to start V_2's presentation synchronous to the ongoing presentations at the other sites.

When a user desires to leave the scenario, the clocks controlling his or her streams have to be removed from the clock hierarchy.

6 CLOCKS AT SOURCE COMPONENTS

So far, we only considered clocks attached to sink components. The mixer scenario illustrated in Fig. 7 gives the motivation for having clocks attached to source components also. In general, substreams S_1, S_2 and S_3 may have individual start values. For example, if three different subsequences of a stored video clip are to be mixed together, the start values differ from sequence to sequence. However, with a clock at the sink component only, it is impossible to specify individual start values for multiple sources. The solution to this problem is obvious, a clock is attached to each source component, which then can be started with an individual start value.

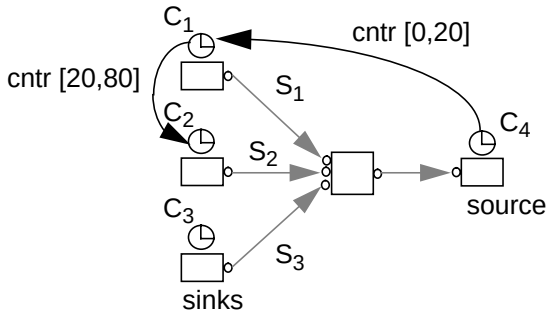


Figure 7 : Clocks at Source Components

As mentioned earlier, clocks at source components are optional. In a configuration without source clocks, start of processing of source components is implicitly triggered by starting the corresponding sink clock, where the start value is determined at the sink clock. However, as soon as a clock is attached to a source component, processing must be enabled explicitly by starting the attached clock. It is important to point out, that `Start` issued at a source clock only enables start of processing rather than starting the clock immediately. The time when the clock is actually started is mainly determined by the underlying control and communication protocols.

Like sink clocks, source clocks may be nodes in clock hierarchies. In contrast to sink clocks, however, source clocks may never be involved in a sync relationship. This is due to the fact that synchronizing a source clock with some other clock makes no sense with regard to stream synchronization or even is impossible in various cases.

In the scenario of Fig. 7, clock C_4 is the root of the control hierarchy. When `Start` is issued at C_4, this operation is propagated to clocks C_1 and C_2. During propagation, the specified start value is transformed according to the reference points associated with the control edges. If the start value specified at C_4 is 0, clocks C_1 and C_2 are started with values 20 and 80, respectively. Since clock C_3 is not part of the control hierarchy, it has to be started explicitly. For example, it may be started later when the application decides to add S_3.

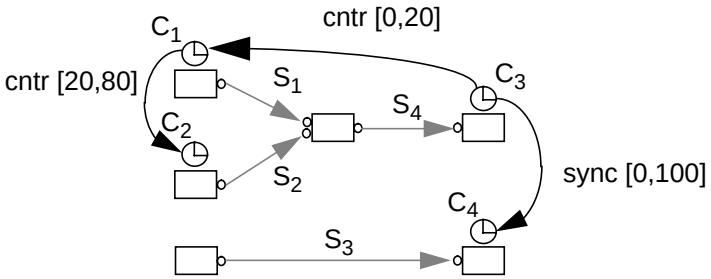


Figure 8 : A Scenario with Control and Sync Edges

The scenario in Fig. 8 combines sync and control edges. Assume that streams S_1, S_2 and S_3 are (stored) video streams with a rate of 25 frames/sec and that clock attribute R = 25 for each clock. Further assume that media times 20, 80 and 100 of S_1, S_2 and S_3, respectively, correspond to the same point in real time. The depicted configuration mixes S_1 and S_2 and synchronizes the output of the mixer with S_3. The entire configuration is controlled by clock C_3, i.e. the whole processing can be started, paused, scaled by issuing the corresponding operations at clock C_3.

```
1   C3.Start(0)
2   C3.Scale(2000,-2)
3   C3.Halt(0)
```

The presentation is started at media time 0, which corresponds to start values 20, 80 and 100 at C_1, C_2 and C_4, respectively. After 2000 frames have been played out, the presentation is continued in reverse order and double speed.

Looking at the above scenario we can indicate two points in the configuration, where stream synchronization is required. Not only S_3 and S_4 have to be synchronized but also S_1 and S_2 as mixing must be done on the basis of matching media times. That is, the media time systems of S_1 and S_2 must be related to the time system of S_4, and the mixer must preserve this relationship in order to enable synchronization of S_3 and S_4. Depending on the type of intermediate component, the time systems of its input and output streams may or may not be related. This subject is discussed in more detail in a longer version of this paper.

7 NESTING OF COMPONENTS

In many areas, nesting has turned out to be a very powerful concept for building higher levels of abstractions. As mentioned earlier, in *Cinema* more complex components can be composed from other components just by linking input and output ports. Compound components again may be used to build other compound components on even higher levels of abstraction, i.e. arbitrary nesting levels are supported.

In the context of synchronization, nesting means that clock hierarchies may be defined within compound components and thus remain invisible for the components' outside world. A clock hierarchy of a compound component is defined at the time the component is composed and specifies synchronization and control relationships between the streams processed by this component. In particular, the internal clock hierarchy of a component specifies sync and control edges between the clocks defined within this component. In addition to the clocks attached to its internal components, a compound component may also contain unattached clocks.

A compound component may contain one or more clock hierarchies. (Note that a hierarchy can consist of a single clock only.) The roots of the internal clock hierarchies are exported and thus become visible to the component's outside world. The exported clocks are attached to the component and are used to control the component's stream processing, i.e. they are used to start, pause or scale the streams processed by the component. Of course, exported clocks may again be involved in clock hierarchies at higher levels of abstraction.

The compound component shown in Fig. 9 provides the abstraction of a television set, capable of playing out a video stream and two audio streams in a synchronized fashion. The shown component contains two basic components, a video decompression component (*D*) and sink component implementing a video output window (*W*). In addition, it includes another compound component, which consists of two filter components (*F*) and two speaker components (*S*). The nested compound component provides the abstraction of an audio output device, whose operation is controlled by clock C_2. The TV component exports clock C_1, which is used to start, pause or scale the audio-visual output.

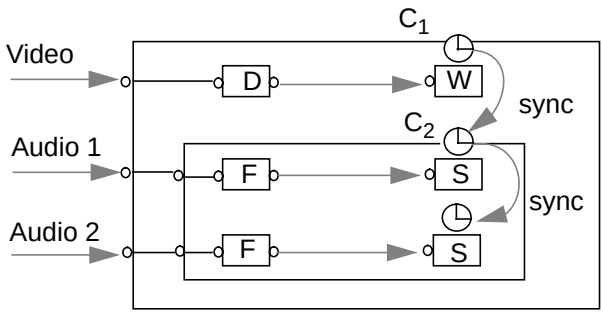


Figure 9 : Nested Components

In summary, compound components may contain arbitrary complex clock hierarchies, which are invisible from the user's point of view. The operations issued at an exported clock are propagated through the clock hierarchy and thereby control the internal processing of the exporting component.

8 SUMMARY

The abstractions proposed in this paper provide for controlling and synchronizing groups of continuous media streams. Clock hierarchies can be used to specify nested groups of streams, where each clock in the hierarchy identifies and controls a certain (sub)group of streams. By means of control and sync edges in clock hierarchies, an application can specify its individual control and synchronization needs in an uniform way. The capability of locking subhierarchies as well as the possibility of dynamically growing and shrinking clock hierarchies are important features in the context of interactive applications, especially in those supporting collaborative

work. Clock hierarchies in conjunction with component nesting provide a powerful means for the simple composition of complex components at higher levels of abstraction. As the computation model underlying the proposed abstractions is very general and has various similarities with others, the results reported in this paper are applicable in a rather broad scope.

The reported work has been conducted in the context of the *CINEMA* project. The implementation of *CINEMA* is in progress, and an early version of a synchronization manager, supporting a limited set of configurations, is already operational. Future work will be to complete this implementation.

Finally, we need to gain more practical experience with the proposed abstractions. Although the abstractions have been applied to model a great variety of application scenarios, we need to conduct extensive experimentation with applications in the field to verify the practical value of the work reported here.

9 REFERENCES

[Appl91] Apple Computer Inc., Cupertino, CA, USA. *Quick Time Developer's Guide*, 1991.

[AnHo91] Anderson, D.P.; Homsy, G.: A continuous media i/o server and its synchronization mechanism. In: IEEE Computer, 24 (10), pp. 51 -57, October 1991.

[CBRS93] Coulson, G.; Blair, G.S.; Robin, Ph.; Shepherd, D.: *Extending the Chorus Micro-Kernel To Support Continuous Media Applications*. In: Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video, November 1993.

[CCGH92] Campbell, A.; Coulson, Garciá, F.; Hutchinson, D.: *A Continuous Media Transport and Orchestration Service*. In: Proc. SIGCOMM '92, August 1992.

[DNNR92] Dannenberg, R.B.; Neuendorffer, T.; Newcomer, J.M.; Rubine, D.: *Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia*. In: 3rd International Workshop on Network and Operating System Support for Digital Auido and Video, November 1992.

[Herr91] Herrtwich, R.G.: *Time Capsules: An Abstraction for Access to Continuous Media Data*. In: The Journal of Real-Time Systems, Kluwer Acadamic Publishers, 1991, pp. 355 - 376.

[IBM92] *IBM Corporation: Multimedia Presentation Manager Programming Reference and Programming Guide 1.0,* IBM Form: S41G-2919-00 and S41G-2920-00,

March 1992.

[IMA93] *IMA Multimedia System Services, Version 1.0* (contributors: Hewlett-Packard Comanpy, International Business Machines Corporation and SunSoft Inc.), available via ftp from ibminet.awdpa.ibm.com, July 1992.

[MeES93] Meyer, Th.; Effelsberg, W.; Steinmetz, R.: *A Taxonomy on Multimedia Synchronization*. In: 4th International Workshop on Future Trends of Distributed Computing Systems, September 1993.

[Mill89] Mills, D.L.; *Internet Time Synchronization: The Network Time Protocol*. Internet Requests for Comments No. 1129 PiFC/1129, 1989.

[MKS89] Magee, J.; Kramer, J.; Sloman, M.: *Constructing Distributed Systems in Conic*. In: IEEE Transactions on Software Engineering, Vol. 15, No. 6, June 1989.

[RBH94] Rothermel, K.; Barth, I.; Helbig, T.: *CINEMA: An Architecture for Configurable Distributed Multimedia Applications*. Technical Report 3/94, University of Stuttgart, April 1994.

[RoDe92] Rothermel, K.; Dermler, G.: *Synchronization in Joint-Viewing Environments*. In: Proc. 3rd International Workshop on Network and Operating System Support for Digital Audio and Video, November 1992.

[RoHe94] Rothermel, K.; Helbig, T.: *Protocols for Synchronizing Application-Level Streams*. Technical Report, University of Stuttgart, 1994 (in preparation).