IEEE Journal on Selected Areas in Communications
Synchronization Issues in Multimedia Communications
Vol. 14, No. 1, January 1996

Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams

Kurt Rothermel, Tobias Helbig
University of Stuttgart,
Institute of Parallel and Distributed High-Performance Systems (IPVR)
Breitwiesenstraße 20-22, D-70565 Stuttgart, Germany
{rothermel,helbig}@informatik.uni-stuttgart.de

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 of 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 middleware. 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 [MES93]. 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. Application-level streams, which we will assume in this paper, may be processed by a network of linked components. They 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.

We propose 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.

Media clocks and clock hierarchies provide a flexible and powerful abstraction for controlling and synchronizing media streams in distributed applications. These abstractions are independent of the underlying synchronization protocols. They allow for the specification of the application's synchronization needs on a rather high level of abstraction without making assumptions about the applied control mechanisms. Clock hierarchies provide a uniform interface to different synchronization protocols which may be used alternatively. In *Cinema*, we have implemented, for example, a simplified version of the protocol proposed by Escobar et al. [EDP92] and the Adaptive Synchronization Protocol described in [RoHe95].

Clock hierarchies allow for expressing complex relationships among distributed media streams. Furthermore, dynamic changes of those relationships are supported. Both characteristics make them a suitable basis for CSCW applications or for viewing (distributed) interactive multimedia documents. For example, changes in the user community of a CSCW application require to add or remove streams or alter synchronization relationships during run-time. Interactive viewing of a document requires the flexibility to control groups and subgroups of streams depending on user interactions.

An additional 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 of 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.

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. Whereas 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, and then, in Sec. 8, describe how media clocks are integrated in the CINEMA system. 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. On the same level of abstraction is the concept of *channel groups* [GuMo93]. It allows to express relationships among real-time connections with regard to resource sharing and jitter tolerance. The approach offers a network-oriented interface. Synchronization is based on jitter-controlled channels instead of timestamps, and hence is restricted to streams originating at the same node.

Various abstractions for controlling groups of application-level streams have been proposed in literature. Some of these proposals apply to non-distributed environments only (e.g., IBM’s Multimedia presentation manager [IBM992]). 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. QuickTime [Appl91] in its current version provides a rich set of tools for developing non-distributed multimedia applications. Several data streams can be grouped together according to synchronization relationships to form multimedia documents, so-called *movies*. The streams are played out synchronously. However, dynamic changes to pre-defined documents are restricted to some operations (e.g., to muting a stream). In general, the document has to be edited to change its contents. QuickTime’s run-time environment, upon which the toolbox is based, provides more flexibility. It offers concepts to model temporal relationships and implements a general timing service for a single node. However, this service provides low-level support for the timed activation of functions without offering a uniform stream-oriented control interface. An implementation of the CINEMA abstractions in a local environment could be based on these services.

The Multimedia System Services proposed by the Interactive Multimedia Association [Hewl93] 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 operations 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 data units from its input ports.

A **client** is the entity that establishes and controls an application. 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 [Hewl93], Quicktime [Appl91], SUMO project [CBRS93], or the work done by Gibbs [Gibb91]). Following the notation introduced in [Hewl93], a configuration of linked components is called **flow graph**.

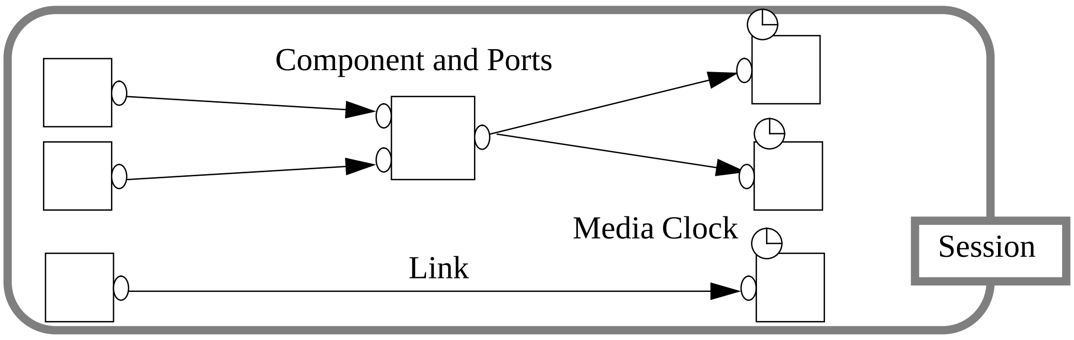


Figure 1: Example Application

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. For controlling the flow of media streams an extra abstraction, the so-called **media clock**, is provided. Media clocks are used to start, pause, or scale media streams.

A unique feature of *Cinema* is component nesting. 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 to other architectures (e.g., IMA [Hewl93] 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[1]:

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

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: $S \cdot R$ 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 higher 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 the ratio R is 25, media time corresponds to frame sequence numbers. If the start value M is 5, stream processing is started with the 5th frame in the stream provided the lower bound of its temporal dimension is 1. If, however, media time is counted in milliseconds, R is set to 1000. In each case, ratio R defines the "normal" speed of media time, whereas attribute S can be used to speed up or slow down the progress of media time.

[1] The type of clock was proposed in [Herr91], where clocks were used to time the internal processing of data streams. We will use media clocks in a totally different context. A clock is an abstraction used by *clients* to control the play-out of data streams and to express synchronization relationships between streams.

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*:

$$m = M + S \cdot R(t - T) \tag{E1}$$

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. Clocks may be attached to components. *Cinema* allows arbitrarily structured flow graphs, which may consist of any number of source, sink and intermediate components. A client of *Cinema* can control the processing of all components in the flow graph by starting, stopping and scaling the streams consumed by the graph's sinks. As a simple example consider a pipeline consisting of a video source, some filter component and a video sink. The client may begin to play out the video just by starting the media clock associated with the sink. Of course, starting the video presentation requires not only the sink but also the source and filter component to be activated at the proper time. From a client's point of view, however, it suffices to control the video output by issuing the appropriate clock operations at the sink component. Invisible for the client, these clock operations trigger internal control procedures that orchestrate and monitor the processing of the entire pipeline. Clocks at source components are primarily used to enable and disable the output of sources and to define start values for the outgoing streams. Clocks at intermediate components are not needed, which simplifies stream control for clients significantly.[2]

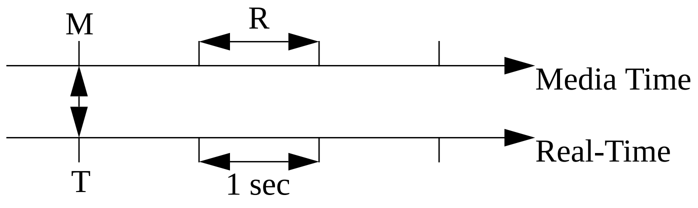


Figure 2: Mapping of Media Time to Real-Time

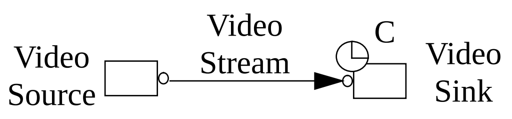
Whereas clocks at source components are optional, they are mandatory at sink components. A clock attached to a sink component controls the temporal progress of all data streams processed (e.g., played out) by this component. This is expressed more precisely by the so-called **clock condition**: a data 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.

[2] Synchronization relationships between streams consumed by the same intermediate component are defined as part of the specification of intermediate components.

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*, whereas 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 the 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`.



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

Figure 3: Controlling a Video Stream

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

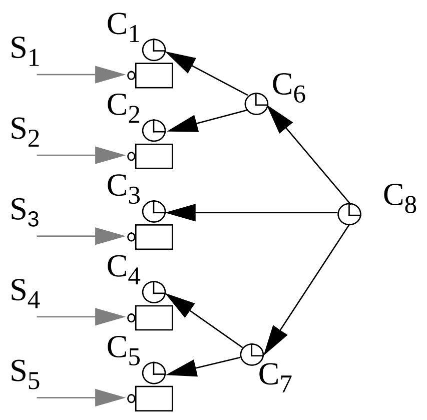


Figure 4: Grouping Streams

A clock operation issued at a clock not only influences 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. Propagation is a prerequisite for component nesting, as will be seen in Sec. 7. 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 influence 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. This is especially useful in those scenarios, where multiple clients are involved in the same application and hence clock hierarchies typically cover several client domains. Here, locking provides a means to shield clock subhierarchies located in a given client domain from propagated clock operations originated in some other client domain. Consequently, by locking clocks a client 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 time system to the other as follows:

$$m_2 = (m_1 - P_1) \cdot \frac{S_2 \cdot R_2}{S_1 \cdot R_1} + P_2 \qquad \text{(E2)}$$

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 as 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 skew attribute ([StEn93], [AnHo91]) which describes the tolerated deviation from the temporal relationship defined by the reference point. This attribute is only specified for sync edges.
- 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.

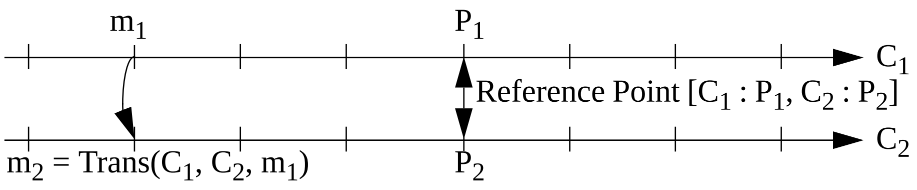

m_1
P_1
C_1
Reference Point [$C_1 : P_1$, $C_2 : P_2$]
C_2
m_2 = Trans(C_1, C_2, m_1)
P_2

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 equation E2.

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[3].
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. An `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*.

[3] 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.

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 media time system to another. 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 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, different semantics are conceivable for operations `Halt` and `Scale`. If, for example, `C.Halt`(*Now*) is performed, then all streams controlled by the clocks in C's subhierarchy are halted immediately because *Now* - by 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

A sync edge is directed from a clock, say C_1, to another clock, say C_2. The edge is associated with a reference point *RP* and a skew attribute *s*. Its semantics 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* and the skew attribute *s*. 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 if

   $$\forall \mathrm{I} \in ISet\ \forall t \in I : C_2(t) = m_2 \Rightarrow C_1(t) = Trans(C_2, C_1, m_2) \pm s,$$
   where $C(t)$ denotes the value of C at time t.[4]

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] For the sake of simplicity, we assume the skew to be symmetric. Extending the scheme to asymmetric skew specifications is straight forward.

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 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, whereas the reference points in the hierarchy define how streams have to be 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 clients with their individual needs participate in the same (CSCW) application.

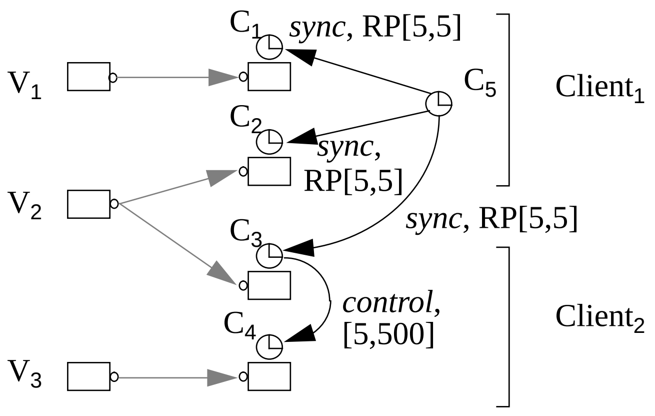


Figure 6: A Simple Tele-Cooperation Scenario

5.3 Example

Fig. 6 shows a simple tele-cooperation scenario with two clients. The subject of the cooperation is an experiment shown in video V_2. We assume that extra speech channels exist that allow the clients to talk to each other. The two clients commonly view V_2 and discuss the experiment while they follow the presentation. To ensure that both clients see the same information at the same time, V_2 must be played out synchronously at both client sites. Besides V_2, client 1 views video V_1, which shows the same experiment from a different perspective. Consequently, V_1 and V_2 are to be synchronized. Client 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. If we assume that media time is equivalent to frame numbers, frame 5 of video V_2 corresponds to frame 500 of video V_3 while frames 5 of all other videos correspond to each other.

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. Client 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 client 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 client 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 client desires to join the scenario while cooperation already takes place between client 1 and client 2, the clock hierarchy has to be extended dynamically. Assume that the new client 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 client is linked by means of a sync edge having a reference point [5,5] to clock C_5. When the new client 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 client 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 also gives the motivation for having clocks attached to source components. 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.

As mentioned earlier, clocks at source components are optional. In a configuration without source clocks, the 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 moment 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.

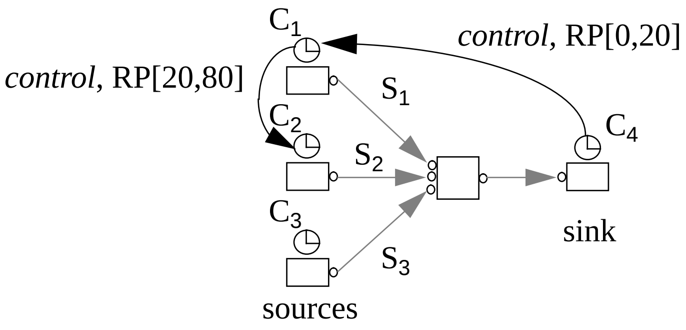


Figure 7: Clocks at Source Components

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 client decides to add S_3.

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 is 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 scenario in Fig. 8, 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 if mixing requires synchronized input streams. The temporal relationship between S_1 and S_2, which defines how these streams are to be synchronized, is specified by reference point [20,80].

Whether or not a component expects synchronized input streams can be derived from the component's attributes. These attributes together with the sync edges specified by the client are used by *Cinema* to find out where in a given configuration stream synchronization is to be performed.

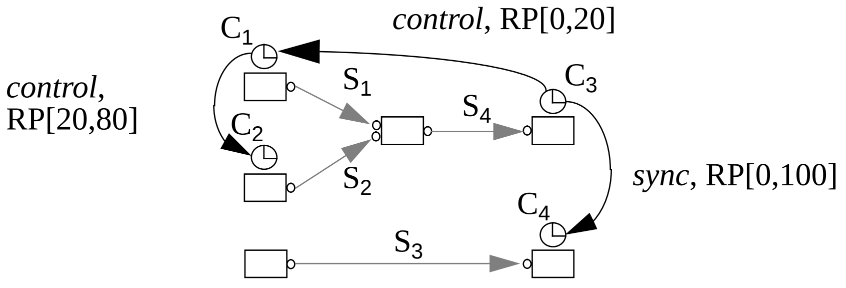


Figure 8: A Scenario with Control and Sync Edges

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 a 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.

In summary, compound components may contain arbitrary complex clock hierarchies, which are invisible from the client'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.

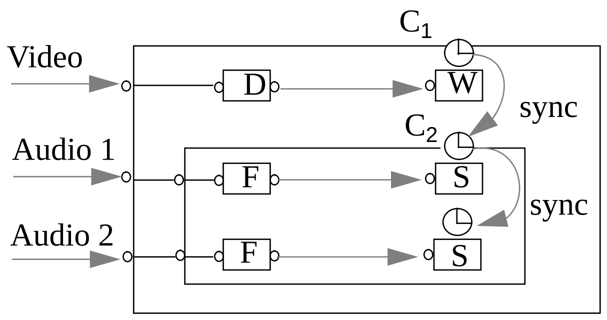


Figure 9: Nested Components

8 IMPLEMENTATION OF CLOCK HIERARCHIES

8.1 Synchronization Architecture

The abstraction of clock hierarchies has been developed in the *Cinema* system, which provides a development platform for distributed multimedia applications. In particular, *Cinema* supports the configuration of distributed applications, resource management as well as communication and synchronization of multimedia streams [RBH94].

The main concepts of the synchronization architecture of *Cinema* are depicted in Fig. 10. Its three layers provide the following functionality:

Clock Management Layer: This layer implements the abstraction of clocks and clock hierarchies and provides a programming interface which allows clients to create, manage and access clock hierarchies. This layer will be described in detail below.

Control and Synchronization Layer: This layer implements the actual synchronization and control protocols. The major concepts offered by the Control and Synchronization Layer (CS Layer) are stream groups and controllers. Streams whose play-out is to be synchronized are grouped into so-called stream groups. Each group is associated with a controller module which is responsible for synchronizing and controlling the streams of this group. In particular, it offers operations to create and delete stream groups, to add streams to and remove streams from existing groups, and to prime, start, halt, scale and lock a set of streams, where the specified set of streams may be a subset of a group. The stream groups may consist of one or more members, where the same operations can be used to control individual streams and multi-member groups.

The temporal parameters of the CS operations are based on a uniform time system which is called CS time system. For this time system it is assumed that *R=S=1*. When a stream is introduced to the controller, the latter is supplied with the information needed to map CS time to the stream's media time.

Internally, controllers interact with so-called agents according to a synchronization protocol. Whereas controllers handle stream groups, agents represent and control individual data streams. The agents' interface and functionality strongly depend on the synchronization protocol applied. Our architecture is extensible in the way that different types of synchronization mechanisms can be "plugged in" without affecting the generic controller interface.

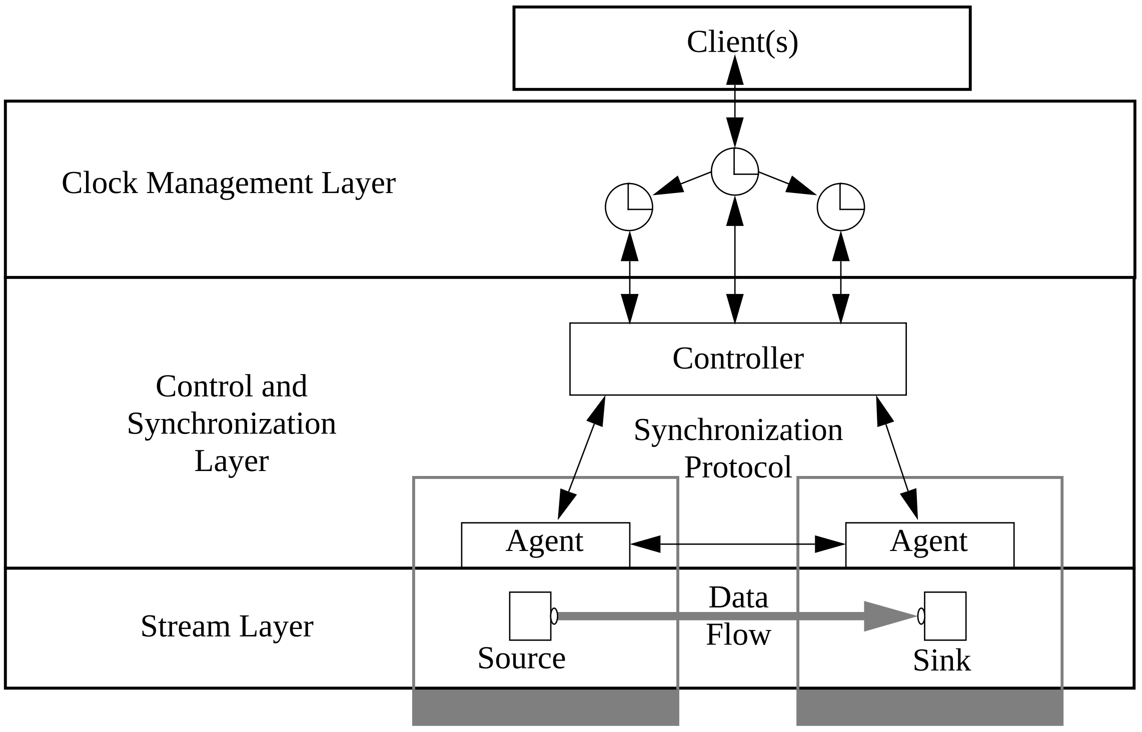


Figure 10: Distributed Synchronization Service Architecture

For synchronizing data streams we have developed the Adaptive Synchronization Protocol (ASP) [RoHe95]. ASP supports distributed sinks and sources and is highly adaptive in terms of the applied synchronization strategy (e.g., shortest delay versus best stream quality).

Stream Layer: This layer provides the basic functionality for transmitting continuous data streams. In particular, it includes mechanisms for resource reservation, real-time scheduling and buffer management. For more details refer to [HNR95], [Bart94], [Derm94].

In the next subsection, we will concentrate on aspects of the Clock Management Layer. In particular, we will describe how distributed clock hierarchies are realized and clock operations are mapped to the corresponding CS Layer functions.

8.2 Management of Clock Hierarchies

In *Cinema*, clock hierarchies may be distributed among multiple nodes of a computer network. Clocks are global objects which are identified by globally unique names. A client may access local and remote clocks by calling the clocks' operations, where proxy objects [Shap86] transparently provide access to remote clock objects. Clocks may even be shared by multiple clients in order to cooperatively control applications. However, mechanisms for synchronizing the access of concurrent clients are out of the scope of our implementation, and hence must be provided by the application. A clock attached to a component is located at the component's node, whereas all other clock objects are local to their creators.

Before describing our implementation, we have to introduce some terminology. A clock hierarchy may contain any number of so-called *sync-subhierarchies*, which are spanned by the clock hierarchy's sync edges. The root clock of a sync-subhierarchy is called *sync-root clock*, which has no ingoing but at least one outgoing sync edge. In the example depicted in Fig. 11 there is one sync-subhierarchy consisting of clocks C_2, C_3, C_4 and C_5, where C_2 is the sync-root clock.

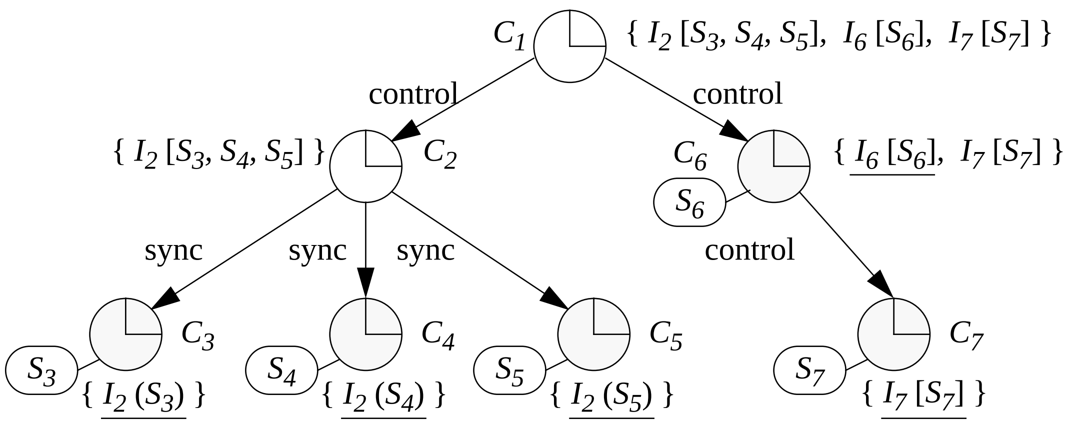


Figure 11: Example Clock Hierarchy

Clocks attached to components define the temporal characteristics of the data streams consumed by these components. These clocks, which define the media time system of individual data streams, are called *stream clocks*. Note that the leaves in the clock hierarchy are always stream clocks, but not vice versa. Stream clocks may be part of a sync-subhierarchy or may be *independent*, i.e., an *independent* stream clock has neither an ingoing nor an outgoing sync edge. In our example in Fig. 11, all shaded clocks are assumed to be stream clocks, where a stream clock C_i is attached with stream S_i. In the depicted hierarchy, only C_6 and C_7 are *independent* stream clocks.

Let us briefly sketch the mapping of clock hierarchies to the concepts provided by the CS Layer. Remember, the major abstraction of the CS Layer are stream groups, consisting of one or more streams. Since all streams belonging to the same sync-subhierarchy must be played out in a synchronized manner, they constitute a multi-member stream group at the CS Layer interface. In contrast, each stream attached to an *independent* stream clock can be played-out independently of any other stream, and thus forms a single-member stream group. For each stream group there exists one controller function which is responsible for controlling the group member's play-out.

Clearly, all operations concerning the streams of a given stream group must be directed to the group's controller. In our example in Fig. 11, we can indicate single-member groups for streams S_6 and S_7 and one multi-member group consisting of streams S_3, S_4 and S_5.

From the clients' point of view, clock operations are propagated in a root-to-leaf direction through the clock hierarchy. While this is a simple and powerful model for the client, our implementation of clock hierarchies internally works without propagation of clock operations in order to avoid unnecessary message overhead. Each clock in the hierarchy collects information about its subhierarchy during an initialization phase. After this phase, each clock in the hierarchy knows all streams attached to the stream clocks in its subhierarchy, i.e., the clock knows all streams that may be affected when one of its operations is issued. Moreover, it knows the identity of the responsible controller for each of these streams. When a clock operation is issued, the clock maps the operation's temporal parameters to CS time, and then sends the corresponding CS Layer requests directly to the responsible controllers rather than propagating the clock operation to its descendants.

Now, we will consider initialization in more detail. The initialization process proceeds in two phases. While in the first phase, controllers are assigned to sync-subhierarchies and *independent* stream clocks, the second phase derives stream groups from the structure of sync-subhierarchies and supplies each clock with the necessary information about its descendants. For the sake of simplicity, we will consider static clock hierarchies only when describing our algorithm below, although our implementation is also able to handle dynamic hierarchies.

During the **first phase of initialization**, ***Initialize*** requests are propagated in a root-to-leaf direction through the entire clock hierarchy. Controllers are assigned by *independent* stream clocks and sync-root clocks only. When a sync-root clock receives ***Initialize***, it assigns a (local) controller function to its sync-subhierarchy. The stream group will only be defined in the second phase of initialization, in which the clock acquires the knowledge about the members of this stream group. The identifier of the assigned controller is included in each ***Initialize*** messages propagated to a clock of this sync-subhierarchy, and hence each of these clocks will eventually learn about this controller.

When an *independent* stream clock receives ***Initialize***, it assigns a (local) controller and requests this controller to establish a (single-member) stream group. It does so by defining a group at the controller's interface where it delivers a descriptor of the clock's stream. Then it forwards the ***Initialize*** message to its subordinates if existing.

Additionally, in the first phase of initialization the reference points in each path of the clock hierarchy are aligned as shown in Fig. 12. Remember, a reference point defined between two clocks consists of two reference times, one for each clock, where both reference (media) times correspond to the same point in real-time. Each ***Initialize*** request includes the sender's aligned reference time, the receiving clock aligns its reference time with the one included in the request according to equation E2. Consequently, after phase 1, the reference times of all clocks in the hierarchy correspond to the same point in real-time. By convention the CS time system assumed by controllers is aligned with reference time zero. Having reference points aligned this way has the great advantage that CS time can be easily mapped to any time system in the clock hierarchy and vice versa with the knowledge of the time system's aligned reference time and its R and S parameters.

After the first phase of initialization, each stream clock knows the controller responsible for its stream, where all clocks belonging to the same hierarchy assume the same controller. Moreover, *independent* stream clocks have already established (single member) stream groups for their streams, and finally, each stream clock has set up a so-called stream descriptor containing the following information:

- the identifier of the clock's stream which is meaningful for the CS Layer, and
- temporal parameters of the clock including the clock's *R* and *S* parameters as well as its aligned reference time.

In our example depicted in Fig. 11, after phase 1 controllers I_2, I_6 and I_7 have been assigned by clocks C_2, C_6 and C_7, respectively. Furthermore, the controllers assigned by C_6 and C_7 have already established single-member groups for streams S_6 and S_7.

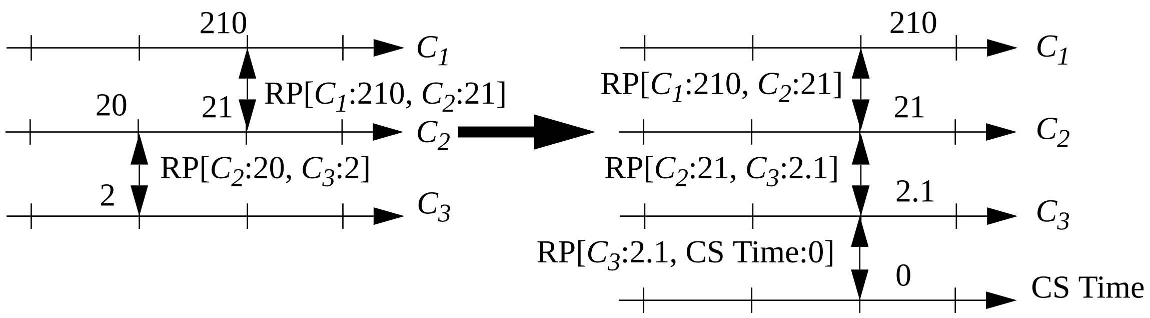


Figure 12: Alignment of Reference Points

During **phase two of the initialization protocol,** stream information is propagated in a leaf-to-root direction. For the purpose of collecting stream information, each clock in the hierarchy maintains a so-called *KnownGroups* set. A set of this type may contain any number of group descriptors, each of which consists of a set of stream descriptors. Each group descriptor is associated with the identifier of the controller controlling the specified stream group.

Initially, the *KnownGroups* set of a stream clock contains a single group descriptor. This stream group only has a single member, the descriptor of the clock's stream, and it is associated with the identifier of this stream's controller. The included group descriptor is marked *complete* if the stream clock is *independent*, and is marked *incomplete* otherwise. For all other clocks *KnownGroups* is initially empty. In our example illustrated in Fig. 11, we will use the notation $I(S_1,..,S_i)$ to denote an *incomplete* group of streams being controlled by controller *I*, whereas the notation $I[S_1,..,S_i]$ is used to denote *complete* groups. In our example, the groups with an underscore show the initial values of the stream clocks' *KnownGroups* set.

A leaf clock answers an ***Initialize*** request by sending a ***Done*** message including its *KnownGroups* set to its superior clock. Whenever a non-leaf clock receives a ***Done*** message from a subordinate, it includes the group descriptors contained in the received message into its local *KnownGroups* set. After having received a ***Done*** message from each subordinate, it merges all group descriptors marked *incomplete* in its *KnownGroups* set which results in a single *incomplete* group descriptor. The succeeding actions depend on the clock's role.

If the clock is not a sync-root clock, it just transfers a ***Done*** message encompassing the clock's *KnownGroups* set to its superior if existing. If, however, the clock is a sync-root clock, the group descriptor marked *incomplete* in its *KnownGroups*, say *G*, identifies all streams attached to the clock's sync-subhierarchy. To establish the corresponding stream group, the clock requests the (local) controller associated with *G* to define a stream group, where the controller is supplied with the stream descriptors included in *G*. The clock then marks *G complete*, includes its *KnownGroups* set in the ***Done*** message and transfers this message to its superior clock if existing.

Initialization ends when the root of the clock hierarchy has received ***Done*** messages from all subordinates and has finished its local processing. After that, all stream groups of the hierarchy have been defined and established by controllers of the CS Layer. Furthermore, each clock in the hierarchy knows from its *KnownGroups* set all stream groups (*complete* or *incomplete*) defined by its subhierarchy, and for each of these stream groups it knows the responsible controller. Consequently, when a clock operation is issued, the clock has all information available to directly contact the controllers of the affected data streams (for more detail see below).

Fig. 11 shows the *KnownGroups* set for each clock in the hierarchy. The root clock for example, can derive from its set that streams S_3, S_4 and S_5 are synchronized by controller I_2 and streams S_6 and S_7 are controlled by I_6 and I_7, respectively. Consequently, it knows all streams that are affected by a clock operation as well as the controllers responsible for these streams.

After initialization, clock operations can be issued at the clocks of the hierarchy. When an operation is issued, a clock first transforms the temporal parameters of this operation to CS time by applying equation E2. Then it requests the corresponding controller operation for each stream group specified in its *KnownGroups* set. The request sent for a stream group is destined to the controller associated with this group and includes the identifiers of the stream group's members[5].

In requests issued to controllers, all temporal parameters, such as the start values of streams, are given in CS time. Since a controller stores for each stream it controls the stream's descriptor, it has all information available to internally map CS time to the time systems of the individual streams.

Finally, we will briefly discuss the communication cost of the proposed implementation of clock hierarchies. The initialization protocol requires *2(n-1)* messages, where *n* denotes the number of clocks in the hierarchy. In the worst case, two messages are needed to map a clock operation to a controller call for a given stream group. The worst case occurs only if the client accesses a remote clock, and this clock and the controller reside on different sites. However, we expect that in most cases clients access local clocks. Remember that stream clocks are local to the end-point of their streams, whereas all other clocks are local to their creators. Moreover, we expect that the majority of clock operations is issued at sync-root clocks and at *independent* stream clocks, whose associated stream groups are controlled by local controller functions. The message overhead for controlling and synchronizing the stream groups in the CS Layer depends on the applied synchronization and control protocols (see [RoHe95]).

5 Note that the stream groups described in *KnownGroups* may be *incomplete*.That is why stream identifiers have to be included in controller requests.

9 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 to 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 the *Cinema* prototype is in progress. The first version of the proposed architecture of the synchronization management is operational. We use the prototype 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.

10 REFERENCES

[AnHo91] David P. Anderson and George Homsy. Synchronization Policies and Mechanisms in a Continuous Media I/O Server. *Report No. UCB/CSD 91/617, Computer Science Division (EECS), University of California, Berkeley, CA*, 2 1991.

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

[Bart94] Ingo Barth. Extending the Rate Monotonic Scheduling Algorithm to Get Shorter Delays. *International Workshop on Advanced Teleservices and High-Speed Communication Architectures, Heidelberg, Germany*, 9 1994.

[CBRS93] Geoff Coulson, Gordon S. Blair, Philippe Robin, and Doug Shepherd. Extending the Chorus Micro-Kernel to Support Continuous Media Applications. *4th International Workshop on Network and Operating Systems Support for Digital Audio and Video,* pages 49–60, 11 1993.

[CCGH92] Andrew Campell, Geoff Coulson, Francisco Garcia, and David Hutchison. A Continuous Media Transport and Orchestration Service. *SIGCOMM'92 Communications Architectures and Protocols*, pages 99–110, 8 1992.

[Derm94] Gabriel Dermler. Ein Konzept zur Integration von Netzwerk-Filtern in verteilte Multimedia-Systeme. *Innovationen bei Rechen- und Kommunikationssystemen,* pages 196–203, 9 1994.

[DNNR92] Roger B. Dannenberg, Tom Neuendorffer, Joseph M. Newcomer, and Dean Rubine. Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia.

3rd International Workshop on Network and Operating System Support for Digital Audio and Video, 11 1992.

[EDP92] Julio Escobar, Debra Deutsch, and Craig Partridge. Flow Synchronization Protocol. *IEEE Global Communications Conference*, 12 1992.

[Gibb91] Simon Gibbs. Composite Multimedia and Active Objects. *OOPSLA '91, Phoenix*, pages 97–112, 10 1991.

[GuMo93] Amit Gupta and Mark Moran. Channel Groups - A Unifying Abstraction for Specifying Inter-stream Relationships. *Technical Report 93-15, UC Berkeley*, 3 1993.

[Herr91] Ralf Guido Herrtwich. Time Capsules: An Abstraction for Access to Continuous-Media Data. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, pages 355–376, 3 1991.

[Hewl93] Hewlett-Packard Company and International Business Machines Corporation and SunSoft Inc. *Multimedia System Services, Version 1.0, available via ftp from ibminet.awdpa.ibm.com*, 7 1993.

[HNR95] Tobias Helbig, Soufiane Noureddine, and Kurt Rothermel. Activation Set: An Abstraction for Accessing Periodic Data Streams. *IS&T/SPIE Multimedia Computing and Networking '95, San Jose*, pages 368-378. 2 1995.

[IBM992] IBM Corporation. *Multimedia Presentation Manager Programming Reference and Programming Guide 1.0, IBM Form: S41G-2919-00 and S41G-2920-00*, 3 1992.

[MES93] Thomas Meyer, Wolfgang Effelsberg, and Ralf Steinmetz. A Taxonomy on Multimedia Synchronization. *4th International Workshop on Future Trends of Distributed Computing Systems*, 9 1993.

[Mill89] David L. Mills. Network Time Protocol (Version 2) Specification and Implementation. *RFC 1119*, 9 1989.

[MKS89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering, Vol. 15, No. 6*, 6 1989.

[RBH94] Kurt Rothermel, Ingo Barth, and Tobias Helbig. CINEMA - An Architecture for Distributed Multimedia Applications. In *Architecture and Protocols for High-Speed Networks*, pages 253–271. Kluwer Academic Publishers, 1994.

[RoDe92] Kurt Rothermel and Gabriel Dermler. Synchronization in Joint-Viewing Environments. *3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, 11 1992.

[RoHe95] Kurt Rothermel and Tobias Helbig. An Adaptive Stream Synchronization Protocol. *5th International Workshop on Network and Operating System Support for Digital Audio and Video*, 4 1995.

[Shap86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. *6th International Conference on Distributed Computer Systems*, 5 1986.

[StEn93] Ralf Steinmetz and Clemens Engler. Human Perception of Media Synchronization. *Technical Report 43.9310, IBM ENC, Heidelberg, Germany*, 1993.