In Architecture and Protocols for High-Speed Networks, O. Spaniol, A. Danthine, W. Effelsberg (Ed.) pp. 253-271, Kluwer Academic Publishers, 1994

1

CINEMA - An Architecture for Distributed Multimedia Applications

K. Rothermel, I. Barth, T. Helbig
*University of Stuttgart,
Institute of Parallel and Distributed High-Performance Systems,
Breitwiesenstraße 20-22, D-70565 Stuttgart, Germany*

Abstract

Distributed multimedia applications combine the advantage of distributed computing with the capability of processing discrete and continuous media in an integrated fashion. The development of multimedia applications in distributed environments requires specific abstractions and services, which are usually not provided by generic operating systems. These services are typically realized by software components, often referred to as middleware.

The *CINEMA* (Configurable INtEgrated Multimedia Architecture) project aims at the development of powerful abstractions for multimedia processing in distributed environments. This paper presents a flexible mechanism for the dynamic configuration of applications. The proposed mechanism allows for the definition of arbitrary complex flow graphs connecting various types of multimedia processing elements. Further, processing elements can simply be composed from other ones to provide higher levels of abstraction. We also propose the abstraction of a clock hierarchy to permit grouping, controlling, and synchronization of media streams.

1 INTRODUCTION

Advances in the computer and communication technology have stimulated the integration of digital audio and video with computing, leading to the development of distributed multimedia systems. This class of systems combines the advantages of distributed computing with the capability of processing discrete media, such as text or images, and continuous media, such as audio or video, in an integrated fashion. The capability of integrated multimedia processing not only enhances conventional application environments, but also opens the door for new and innovative applications. A major advantage of multimedia computing in distributed environments is the possibility of sharing resources among applications and users where shared resources may be data objects such as multimedia titles, special processing elements such as compression modules, or special devices such as professional VCRs.

The processing and communication of media streams requires specific system services. In general, media streams are associated with a certain quality that has to be maintained by the underlying system. To be able to guarantee the required stream quality, system services for allocating and reserving system resources, such as CPU cycles or network bandwidth, are needed. Moreover, applications need to control the flow of streams, i.e. they should be able to start, pause, continue or scale individual streams. In many scenarios, it is desirable to group related streams and to control groups of streams rather than individual streams. Finally, powerful services to synchronize multiple streams are required. Those services should permit applications to specify which streams are to be synchronized and how these streams temporally relate to each other.

Generic operating systems usually do not provide those specific multimedia services. The gap between the functionality offered by operating systems and the specific needs of distributed multimedia applications is closed by software components often referred to as middleware. The *CINEMA* (Configurable INtE-grated Multimedia Architecture) system, which is currently under development at the University of Stuttgart, belongs to this system category. It provides abstractions for the dynamic configuration of distributed multimedia applications. Clients may define arbitrary data flow graphs, connecting various processing elements called components. Moreover, component nesting is supported to achieve higher levels of abstractions by simply composing more complex components from already existing ones. The abstraction of a session allows for atomic resource allocation and reservation for any group of connected components. *CINEMA* provides the concept of a clock hierarchy for grouping and controlling streams and groups of streams. The same abstraction permits to express arbitrary complex stream synchronization requirements.

The remainder of the paper is organized as follows. In the next section, a brief overview of related work is given. Then, in Section 3, the way how applications are configured in *CINEMA* is described in some detail. This section also introduces the concept of component nesting. The abstractions for grouping, controlling and synchronizing media streams and groups of streams are presented in Section 4. Finally, we conclude with a brief summary.

2 RELATED WORK

The multitude of problems that arise when integrating multimedia processing into conventional computer systems and attempting to develop distributed multimedia applications are addressed in several projects, which put emphasis on different issues. In the SUMO project [1], the Chorus [2] micro-kernel is extended to support continuous media. This is done by using the real-time features of Chorus and adding stream-based data transfer and quality of service control inside the operating system. The features are accessible by a low-level API. The focus of this work is on operating system issues like scheduling, but not on providing a universal platform and high-level abstractions for developing and configuring distributed multimedia applications. The problem of configuring distributed applications by using software components that are interconnected by linked ports is addressed by Conic [3] and its follow-up project REX [4]. Conic offers languages for programming components and configuring applications without supporting multimedia data handling. The configuration process is centralized in a configuration manager which accepts change specifications for altering configurations.

Specific abstractions for controlling multimedia data streams have been proposed as well. Some of them apply to non-distributed environments only (e.g. QuickTime [5] or IBM’s Multimedia Presentation Manager [6]), while others are tailored to specific configurations (e.g. ACME [7] and Tactus [8]), and mainly are extensions of network window systems supporting streams of digital audio and video data. General requirements that should be met by architectures supporting distributed multimedia applications are specified in the Request for Technology [9] of the Interactive Multimedia Association (IMA). A response to this request contributed by some companies [10] proposes abstractions to structure and control distributed multimedia environments while using multi-vendor processing equipment. The proposal assumes generic multimedia processing elements producing and consuming multimedia data via ports that are associated with formats. However, the nesting of processing elements is not supported and, although grouping is used to handle resource acquisition, stream control and specification of end-to-end quality of service, no means to specify synchronization relationships between data streams are provided.

3 CONFIGURATION OF MULTIMEDIA APPLICATIONS

In order to build large software systems, it is necessary to decompose a system into modules each of which can be separately programmed and tested. The system is then composed as a configuration of these software components. Component programming and component configuration are separate activities which have been referred to as "programming-in-the-small" and "programming-in-the-large", respectively [11].

Configuration may be static or dynamic. In the first approach to system building, all components of the system are configured at the same time. If a modification of the system is required, the complete system has to be stopped and rebuilt according to the new configuration specification. Obviously, static configuration is not a feasible approach in the context of distributed multimedia systems, in which configurations often depend on the available resources and the quality of service the user asks for at run time. Moreover, multimedia applications are often highly dynamic in the sense that users may join and leave the application during run time. Usually, each change in the user community implies a modification of the configuration. Examples for these applications can be found in the area of video conference systems or CSCW systems. Consequently, for multimedia systems the ability to extend and modify a system while it is running definitely is required. The approach of dynamic configuration provides this ability: new components can be introduced, existing ones may be replaced and the interconnection of components can be modified at run time.

In *Cinema*, an application consists of at least one client and a set of data flow graphs. In a data flow graph, the nodes represent components, while the edges are communication links interconnecting the components. A component provides the basic abstractions for the processing of continuous media streams, such as video or audio streams. A continuous media stream is defined to be a sequence of data units, each of which is associated with a media time (for a detailed definition e.g. see [12]). The nature of a component's processing depends on the type of the component. We distinguish between source components, which produce (e.g. capture) data streams, sink components that consume (e.g. play-out) streams, and intermediate components acting as both consumers and producers (e.g. filters or mixers). Media streams may originate at multiple sources, traverse a number of intermediate components and end at multiple sinks.

A client is a software entity that - by using the *CINEMA* services - defines data flow graphs and controls the flow of data within these graphs during run time. It configures (its portion of) an application just by naming the components to be used and interconnecting them according to the application logic that has to be achieved. Furthermore, it may dynamically change the initial configuration during run time as needed. A data flow graph may be arbitrarily distributed over several nodes of a distributed system. As will be seen below, components are configuration independent, which means that their internal logic is independent of the configuration they are used in. Thus, from the client's point of view, there is no conceptual difference whether two adjacent components run either on the same node interconnected by a local link or on different nodes connected by a remote link.

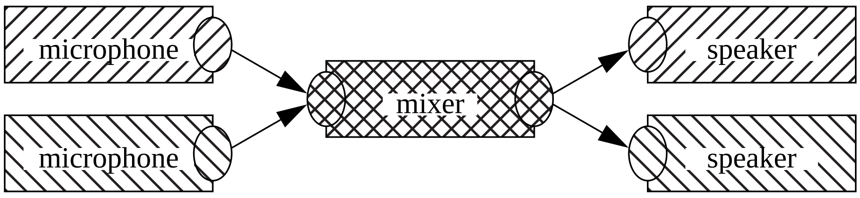


Figure 1.1 Application Domains in a Conferencing Scenario

A client may only control the flow of streams in the flow graphs defined by itself. In particular, a client may start, halt or scale data streams only in its so-called **application domain**, which is defined to be the set of data flow graphs specified by this client. Depending on the type of application, one or more clients may participate in the process of configuring the application. If multiple clients participate, the application is structured into several application domains, one for each participant. Each client only knows and controls the objects in its domain. When sharing components between clients, their domains overlap. The overlapping portions contain the shared components. In other words, shared components may be controlled by multiple clients. Refer to the simple conferencing scenario depicted in Figure 1.1 for an example. In this scenario, the application consists of several domains, each of which links two components - a virtual microphone and speaker of a given user - to a shared mixer component. Whenever a new user joins the application, a new domain linking the new user's (virtual) microphone and speaker to the shared mixer is added.

After this brief overview of the process of configuration in *Cinema*, we can now take a closer look at the concepts provided for defining flow graphs, which are components, ports and links.

3.1 Components and Ports

The processing of continuous media data streams is done by software and hardware modules, called devices. Devices may be e.g. microphones or speakers having specific hardware interfaces and software drivers. In *Cinema*, the processing functionality is abstracted by components. When creating a component, a client specifies the devices that are to be used. Components consume data units of streams reading from their input ports and produce data by writing to their output ports. To build up data flow graphs, components are interconnected by links between the components' ports.

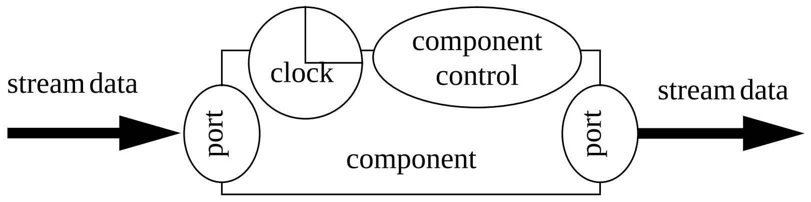


Figure 1.2 The Component's Interfaces

From the client's point of view, a component offers different interfaces to control and manipulate its behavior, the component control interface, the clock interface, and the port interface. The **component control interface** is used to access state information of a component and alter its stream handling behavior. It is specific in the sense that it depends on the processing function performed by the component. For example, the interface of a component abstracting from a speaker device may provide a method to adjust the volume of the presentation. The **clock interface** is optional for sources and mandatory for sinks and is used to control the flow of data units. A detailed description of clocks is given in Section 4. The **port interface** is used by components to send stream data to other components that are interconnected by links or to receive data from them. This decouples the multimedia processing from the transmission of data units between processing stages and allows the usage of the same component in scenarios having local as well as remote communication. To be able to check mismatching connections, each port is associated with a stream type. If a component handles multiple stream types, a new stream type containing the others may be defined. In Figure 1.3, we show an example of a stream type hierarchy. In this example, a port of type “video” can be connected to either one of type “video”, “video-grey”, or “video-color”. In a stream type hierarchy, the descendents of a node are specializations of this node.

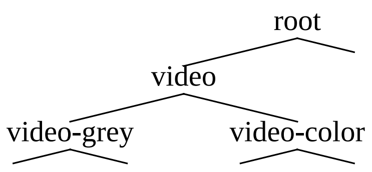


Figure 1.3 Stream Type Hierarchy

The interfaces described above are used by clients to control components and to connect them to build up data flow graphs. In the *Cinema* system, components are managed by additional interfaces. An example for such an internal interface is the resource allocation interface, that is used to negotiate the quality of service and to reserve the required resources to ensure it.

After looking at the interfaces provided by components, we now focus on the definition of components. Configuration independence [3] is a major property to build up components that can be used in a dynamically configured distributed system. This makes it possible to use a component in arbitrary configurations without having to change its processing functionality. Configuration independence is achieved by developing components using a special programming language and compiling and linking them to independent objects. In *Cinema*, we use an object-oriented programming language, the Component Programming Language (CPL) that is based on C++, to program components. It allows the creation of a class hierarchy with inheritance to build up specialized component classes out of existing ones. The following example shows the programming of a microphone component in CPL:

```
COMPONENT microphone
    :: SOURCE // class to derive from
     // define method to map devices
  MAP ( device MICRO ); // device parameter
    dev_name = MICRO;   // handle device parameter
  ENDMAP
     // define method to initialize component
  INIT (int sensitivity);           // specific client-IF
    dev = open(dev_name,"r");     // open the device
    dev_set_samplerate(dev,8000); // rate = 8000 Hz
    dev_set_sensitivity(dev,sensitivity); //set value
  ENDINIT
     // stream type definition
  TYPE 8kHz_Audio :: Audio; // derive 8KHz_Audio from Audio
     // definition of port named audio
  OUTPORT audio 8kHz_Audio;
     // define method to adjust microphone’s sensitivity
  METHOD int sensitivity_adjust(int sensitivity)
    result = dev_get_sensitivity(dev);    // get value
    dev_set_sensitivity(dev, sensitivity); // set value
    return result; // return old value
  ENDMETHOD
     // definition of stream-handling function
  ACTION
    data = dev_get_data(dev); // get audio samples
    audio->put(data); // put samples to output port
  ENDACTION
ENDCOMPONENT
```

In the *Cinema* system, the code segments of a component are executed in different threads. The stream handling segment, defined in the `ACTION` clause, is periodically executed in a real-time thread, whereas the methods of the component control interface are executed in a non-real-time thread. Resource requirements of the real-time thread are calculated when a session, which in *Cinema* is the abstraction for atomic resource reservation, is established (see Section 4.1).

3.2 Creation of Data Flow Graphs

So far, we have introduced the definition of components, the functional building blocks. In this section, we will describe how a client builds up data flow graphs by connecting the components' ports by means of links.

To build an application, a client first establishes the processing functionality by creating the appropriate components. This is done by using a library with a set of functions and classes that is provided by the *Cinema* system. No specialized configuration language is needed which offers the advantage to expand and shrink applications dynamically at run time depending on actual requirements. Moreover, it allows the integration of multimedia processing functionality into existing (non-multimedia) applications. Creating and accessing components does not differ from accessing normal C++-objects. It is done by using appropriate object methods.

As shown above, components may be shared by multiple clients if more than one client participates in the configuration of an application. In *Cinema*, shared components are associated with a globally unique identifier. All clients sharing a given component create this component in their application domain by providing the component's global identifier. Of course, only the create operation issued first establishes the component, while all succeeding ones just enable the callers to access the (already existing) component.

The following code fragment shows the creation of the component objects in the conferencing example illustrated in Figure 1.1. The mixer component is defined as a shared component using the global identifier `conference`.

```
micro   = COMPONENT("microphone",micro_dev);
mixer   = COMPONENT("audio_mixer",NULL,"conference");
speaker = COMPONENT("speaker",speaker_dev);
```

For component initialization, each component provides a method called `init`. The code example below initializes the microphone and the speaker component and specifies the sensitivity to 50 and the volume to 40. The initialization has to be done before defining a session.

```
micro  ->init("sensitivity",50);
speaker->init("volume",40);
```

After component objects have been created, they are connected by creating links among their ports. The component's port objects are accessed by using the method `port` in connection with the port identifier. In our code fragment, we link the output port of the microphone component (named `audio`) and the input port of the mixer component (named `audio_in`). A second link is established between the output port of the mixer component (`audio_out`) and the input port of the speaker component (`audio`).

```
link(micro->port("audio"), mixer->port("audio_in"));
link(mixer->port("audio_out"), speaker->port("audio"));
```

It is important to mention that building up a data flow graph only describes the topology of an application. Linking components does not imply the reservation of resources. To enable communication, sessions have to be established.

3.3 Nesting Components

In many areas, nesting has turned out to be a very powerful concept for building higher levels of abstractions. In *Cinema*, more complex components, called **compound components**, can be composed from other components. Compound components contain a part of a data flow graph. They are used like non-nested, basic components, i.e. from the client's point of view, there is no difference in using basic or compound components since the internal structure of compound components is hidden.

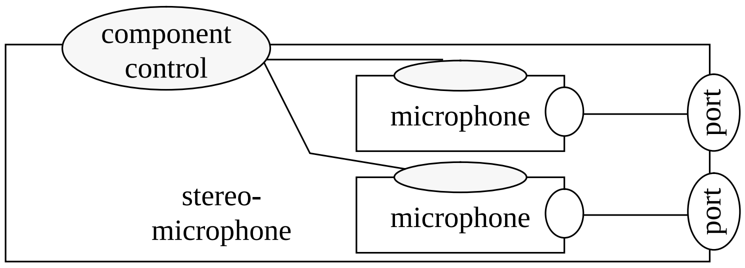


Figure 1.4 Compound Component

Constructing compound components from existing ones is straightforward. Instead of programming an `ACTION` clause, a part of a data flow graph is defined using already existing components. The components used to build the compound component are declared in the `USE` clause. The way they are interconnected by links is defined in the `LINK` clause. Component control interfaces of the nested components are accessed through a common interface provided by the compound component. The mapping of these interfaces is defined when building a compound component.

As an example for the programming of a compound component (see Figure 1.4), we show the definition of a component representing a stereo microphone component. This component uses two components of class microphone as they were declared in Section 4.

```
COMPONENT stereo_micro
     // define Method to map devices
  MAP ( device MICRO_l, device MICRO_r );
    dev_MICRO_l = MICRO_l; // handle device parameter
    dev_MICRO_r = MICRO_r; // handle device parameter
  ENDMAP
     // define mothod to initialize component
  INIT ( int sensitivity );
     // init nested components with provided parameters
    micro_l->init(sensitivity); // initialize micro_l
    micro_r->init(sensitivity); // initialize micro_r
  ENDINIT
     // define the ports of the compound component
  OUTPORT audio_l 8kHz_Audio;
  OUTPORT audio_r 8kHz_Audio;
     // create component objects
  USE
    micro_l = COMPONENT("microphone",dev_MICRO_l);
    micro_r = COMPONENT("microphone",dev_MICRO_r);
  ENDUSE
     // build up flow graph with links
  LINK   // use "this" to refer to compound component
    link(micro_l->port("audio"),this->port("audio_l"));
    link(micro_r->port("audio"),this->port("audio_r"));
  ENDLINK
     // map the specific interfaces to nested ones
  METHOD int sensitivity_adjust(int sensitivity)
    result = micro_l->sensitivity_adjust(sensitivity);
    result = micro_r->sensitivity_adjust(sensitivity);
    return result; // return value
  ENDMETHOD
ENDCOMPONENT
```

4 COMMUNICATION AND SYNCHRONIZATION

Multimedia data streams are transmitted in arbitrarily structured flow graphs of interconnected components. Ensuring a satisfying stream quality over long periods of time while using current computer and network equipment makes the reservation of resources inevitable. Furthermore, due to the temporal dimension of time dependent data streams, there is a need to specify and control temporal properties of streams. Setting initial parameters like data rate or start values has to be enabled as well as scaling (i.e. changing speed or direction) at presentation time. The appropriate control interface in *Cinema* is the media clock. However, an interface that only allows to handle individual data streams is insufficient. Due to tight relationships between different streams, they need to be grouped together and be handled as a unit. This facilitates the control over complex scenarios and is a prerequisite for specifying synchronization relationships between data streams. Especially, the latter is essential in a multimedia system where the quality of a presentation of time dependent data streams strongly depends on observing given synchronization requirements (e.g. lip synchronization of audio and video where the tolerable skew is in the range of 80 ms [13]). The grouping of data streams has to be supported by concepts that are adaptive to the dynamics of interactive and cooperative multimedia applications where at any time new users enter running applications (e.g. teleconferencing) and others leave. In *Cinema*, the means to group control interfaces, to handle them as a unit and to specify synchronization relationships is given by the concept of clock hierarchies. In the following, the concepts to meet the requirements are explained in detail.

4.1 Session

In *Cinema*, a session is the abstraction of resource reservation. It is associated with a set of quality of service parameters. By creating a session, a client causes the *Cinema* system to reserve the resources that are needed to guarantee the specified quality of service requirements. This is done in an all-or-nothing fashion. After a session has been established, the transmission and processing of multimedia data may be started.

A session encompasses parts of the flow graph which is defined by a client. Its actual extension is defined by specifying a set of source and sink components. Intermediate components and interconnecting data paths are determined from the data flow graph by the *Cinema* system. For example, a point-to-point audio session may be created by the following statement. It describes the components and their ports that are part of the session as well as the desired quality of service parameters:

```
create_session(micro  ->port("audio"),
               speaker->port("audio"),
               QoS(Rate(min = 8000, max = 44100),
                   SampleSize(min = 8, max = 16),
                   Delay(min = 50, max = 150));
```

In *Cinema*, quality of service is treated on different levels of abstraction. With sessions, application-specific quality of service specifications are associated. They represent the presentation quality a client wants to achieve at sinks. High-level quality of service parameters, such as picture size and picture rate, depend on the stream type. As resource reservation is independent from the application-level semantics of data streams, high-level parameters are mapped to low-level parameters, such as packet size and packet rate. Low-level parameters are based on parameters used in reservation protocols for multimedia transport systems (e.g. SRP [14], ST-II [15]).

The architecture of resource reservation is separated into two layers: global and local resource management. The global resource management is responsible for negotiation of the quality of service parameters at all the nodes participating in a session and mapping the high-level onto low-level parameters by using a distributed resource reservation protocol. Quality of service parameters are specified at sinks and transferred and negotiated in a sink-to-source direction. Our resource reservation protocol bases on ideas used in RSVP [16] and is designed to perform in arbitrary structured networks of components, which are distributed over any number of nodes (end-to-end reservation, for details see [17]). The local resource management reserves the resources as they are demanded by components or links. This leads to the implementation of several resource managers at each node, one for each individual resource (e.g. for memory, CPU utilization, network bandwidth). To perform resource management for CPUs, we have implemented a split-level scheduler using a modified rate-monotonic algorithm [18].

The success or failure of the establishment of a session determines whether a given application can be started and maintained according to the specified quality of service. Thus, creating a session is the prerequisite to transmit and process data units. Based on this, the following sections describe how temporal properties of streams are specified and data streams are controlled during run time.

4.2 Clocks

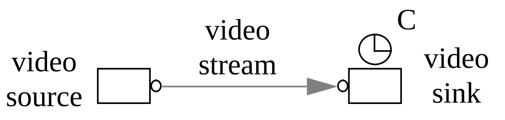
The temporal dimension of continuous media streams is defined by so-called media time systems. The media time system associated with a stream is the temporal framework to determine the media time of the stream's data units. In CINEMA, media time systems are provided by media clocks (or clocks for short). A clock C is defined as follows: $C ::= (R, M, T, S)$. The clock attributes have the following meaning: R determines the ratio between media and 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. S determines the speed of the clock. Media time progresses in normal speed if S equals 1. A speed larger than 1 causes the clock to move faster, a speed smaller than 1 causes it to progress slower, and a negative speed causes it to move backwards. A clock relates media time to real-time. It "ticks" after it has been started and media time (m) can be derived from real-time (t): $m = M + S \cdot R(t - T)$

Clocks are the basic abstraction for clients to control the flow of media streams. They may be attached to components. Clocks attached to sink components control the temporal progress of data streams processed by those components. This is expressed more precisely by the **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. Clocks attached to source components are typically required in flow graphs where multiple sources contribute data to a given sink (e.g. in a mixer scenario). In this case, source clocks are needed to individually start sources and to determine their start values. For more details on source clocks refer to [19].

The most important clock operations for controlling streams are the following. The operation `Start(M)` starts the clock at media time `M`, by doing this it starts the controlled stream as well. The clock attribute T is set to the real-time at which the clock is actually started. `Halt(M)` halts the clock when it reaches clock value `M`, i.e. the stream controlled by this clock is paused. `Pre-`

`pare(M)` prepares the starting of the clock at media time `M` by preloading the buffers along the communication paths of the controlled stream. After `Prepare` has been performed, the clock can be started immediately when `Start` is issued. `Clear()` clears the internal buffers associated with the controlled stream. `Scale(M,S)` changes the speed of the clock to `S` when media time `M` is reached, i.e. it scales the stream controlled by the clock.

In the simple scenario shown in Figure 1.5, clock *C* controls the presentation of a video stream. The play-out is started with frame 15. The play-out rate is doubled when the presentation reaches frame 3000, and the presentation is halted when reaching frame 5000.



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

Figure 1.5 Controlling a Video Stream

4.3 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.

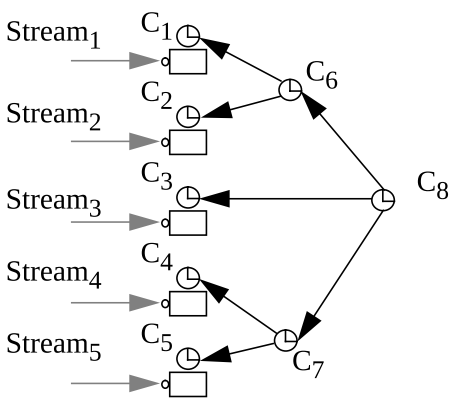


Figure 1.6 Grouping Streams

Remember that clocks attached to components control the streams processed by them. A number of streams can be grouped by linking their controlling clocks in a hierarchical fashion to a common clock, which then controls the entire group. Stream groups can be grouped again to groups at a higher level. In the example given in Figure 1.6, 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.

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. In general, clock operations can be issued at every level of the clock hierarchy. Additionally, clock hierarchies may dynamically grow and shrink even if clocks are ticking. This feature together with the capability of 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 application.

Clocks provide individual media time systems which may relate to each other in various ways. Clock synchronization and propagation of clock operations 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 Figure 1.7). Given this reference point, media time can be transformed from one to the other time system as follows:

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

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 them. 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. Although **control hierarchies** include reference points, this information is considered only when clock operations are propagated to automatically transform the operation's arguments. However, after a hierarchy has been started, its clocks may drift out of synchronization and may be manipulated arbitrarily. 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.

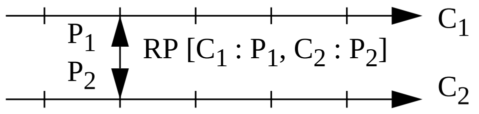


Figure 1.7 Transforming Media Time

A synchronization relationship goes a step further. In addition to propagation, it ensures that the clocks involved progress synchronously. From the clock condition introduced in the previous section it can be concluded that two streams are synchronized if their controlling clocks are synchronized. Thus, **synchronization hierarchies** are a general and very powerful concept to specify arbitrary synchronization requirements between media streams. The structure of the synchronization hierarchy specifies which streams have to be synchronized, while the reference points in the hierarchy define how the temporal dimensions of the streams relate to each other. The system guarantees that all streams controlled by the clocks of the hierarchy are processed synchronously. When a subhierarchy is halted and started once again at a later point in time, this is performed in conformance with the temporal constraints.

Example

Figure 1.8 shows a simple telecooperation scenario with two users. Subject to the cooperation is an experiment shown on video V_2. We assume that additional speech channels exist which allow the users to talk to each other. The two users commonly view V_2. To ensure that both see the same information at the same time, V_2 must be played out synchronously. 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.

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.

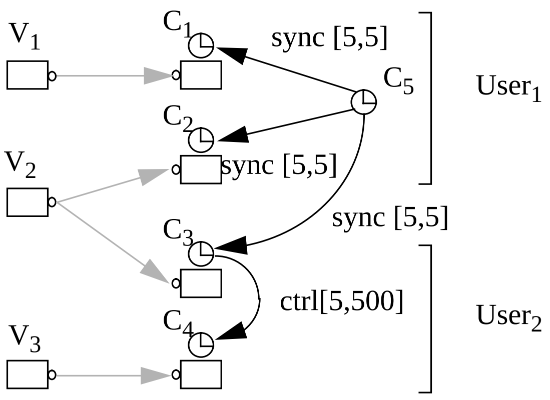


Figure 1.8 A Simple Telecooperation Scenario

If another user desires to join the scenario, the clock hierarchy has to be extended dynamically. After the corresponding session has been established, the clock controlling V_2's presentation at the new user's site is linked by a sync edge to clock C_5. By issuing the start operation, V_2's presentation is started synchronously to the ongoing presentations.

4.4 Clock Hierarchies and Nesting

In the context of synchronization, nesting means that arbitrary complex 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 internal synchronization and control relationships between the clocks defined within this component. Only the root of internal clock hierarchies is exported and thus becomes visible to the components' outside world. The operations issued at an exported clock are propagated through the clock hierarchy and thereby control the internal processing. Exported clocks may again be involved in clock hierarchies at higher levels of abstraction.

The compound component shown in Figure 1.9 provides the abstraction of a television set, capable of playing out a video stream and two audio streams in a synchronized fashion. The component shown 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.

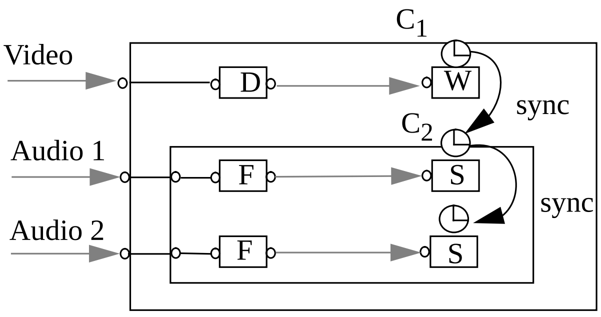


Figure 1.9 Nested Components

5 CONCLUSIONS

The efficient development of distributed multimedia applications requires abstractions and services that are provided by a specialized software layer. Such a middleware layer is based on general purpose operating systems and adds functions and protocols supporting distributed multimedia applications, including those for communication, synchronization and resource management. The *Cinema* system is such a middleware layer. Our paper focused on the description of the service interface of *Cinema*. We described components that provide multimedia processing functionality and may be nested to facilitate the reusability of software and to achieve higher levels of functional abstractions. Distributed multimedia applications are created by interconnecting the components' ports with links which allows the definition of arbitrary flow graphs. Before starting the flow of data units, the creation of sessions results in the reservation of system resources that are needed to ensure quality of service requirements. With media clocks and clock hierarchies we proposed abstractions to control individual data streams as well as groups of streams. We discussed the usage of clock hierarchies to specify synchronization relationships between data streams and showed how they may be used to handle the requirements of dynamic, interactive and cooperative multimedia applications. Finally, it was outlined how clock hierarchies are used to control the propagation of operations in compound components.

When implementing multimedia system services, certain requirements arise that have to be met by the operating systems *Cinema* is layered on. For example the ability to schedule multiple independent multimedia processing tasks by observing real-time deadlines requires the support of real-time scheduling algorithms as well as preemptive threads, which the processing functions are mapped to. Current operating systems only partially fulfil this requirement. Furthermore, multimedia data is transmitted between components on the same node as well as on different nodes. The latter involves making use of transport protocols. When designing the *Cinema* prototype that is based on IBM AIX and DCE as well as on the SUN Solaris operating system, it was decided to encapsulate multimedia transmission functionality into link objects. This offers two major advantages. It allows to use identical interfaces for data transmission between local and remote components which simplifies the configuration of applications significantly. Moreover, it decouples the *Cinema* implementation from the transmission mechanism that actually is used. Due to the lack of a real-time transport system, in our current prototype link objects are based on UDP. For the next version it is planned to replace UDP by a real-time protocol which only requires a reimplementation of the link object. However, all other parts of the system are not affected.

The implementation of the *Cinema* prototype is still in progress. The first version is working. It supports a restricted set of the functionality described in this paper. For example, it is possible to establish applications in a distributed environment and to control and to synchronize the flow of data units in limited configurations. Our future work is directed towards extending the prototype and gaining more experience in using our abstractions by experimenting with applications.

References

[1] G. Coulson, G. S. Blair, P. Robin, D. 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*, pp. 49–60, 11 1993.

[2] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Overview of the Chorus Distributed Operating System. *Chorus Systémes CS/TR-90-25*, 4 1990.

[3] J. Kramer, J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transaction on Software Engineering*, SE-11(4):424–436, 4 1985.

[4] J. Magee, J. Kramer, M. Sloman, and N. Dulay. An Overview of the REX Software Architecture. *2nd IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 10 1990.

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

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

[7] D. P. Anderson, R. Govindan, G. Homsy. Abstractions for Continuous Media in a Network Window System. *Technical Report UCB/CSD 90/596, Computer Science Division, University of California, Berkeley*, 11 1990.

[8] R. B. Dannenberg, T. Neuendorffer, J. M. Newcomer, D. Rubine. Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia. *3nd International Workshop on Network and Operating System Support for Digital Audio and Video*, 11 1992.

[9] Interactive Multimedia Association, Compatibility Project, Annapolis, MD, USA. *Request for Technology: Multimedia System Services, Version 2.0, available via ftp from ibminet.awdpa.ibm.com*, 11 1992.

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

[11] F. DeRemer, H. Kron. Programming-in-the-Large vs. Programming-in-the-Small. *Conference on Reliable Software*, pp. 114–121, 1975.

[12] R. G. Herrtwich. Time Capsules: An Abstraction for Access to Continuous-Media Data. *The Journal of Real-Time Systems, Kluwer Academic Publishers*, pp. 355–376, 3 1991.

[13] R. Steinmetz, C. Engler. Human Perception of Media Synchronization. *Technical Report 43.9310, IBM ENC, Heidelberg, Germany*, 1993.

[14] D. P. Anderson, R. G. Herrtwich, C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet. *Technical Report UCB/CSD 90/562, Computer Science Division, University of California, Berkeley*, 2 1990.

[15] C. Topolcic. Experimental Internet Stream Protocol, Version 2 (ST-II). *RFC 1190*, 10 1990.

[16] L. Zhang, S. Deering, D. Estrin, S. Shanker, D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 9 1993.

[17] M. Häuptle. Development of a Resource Reservation Protocol for Distributed Multimedia Applications (in German). *Master's Thesis, University of Stuttgart/IPVR*, 4 1994.

[18] I. Barth. Extending the Rate Monotonic Scheduling Algorithm to Get Shorter Delays. *To be published: International Workshop on Advanced Teleservices and High-Speed Communication Architectures, Heidelberg, Germany*, 1994.

[19] K. Rothermel, T. Helbig. Clock Hierarchies: An Abstraction for Grouping and Controlling Media Streams. *Technical Report 2/94, University of Stuttgart/IPVR*, 4 1994.