Timed Data Streams in Continuous-Media Systems

Ralf Guido Herrtwich

TR-90-017

May 7, 1990

Abstract

Data in continuous-media systems, such as digital audio and video, has time parameters associated with it that determine its processing and display. We present the "time capsule" abstraction to describe how timed data shall be stored, exchanged, and accessed in a real-time system. When data is written into a time capsule, a time stamp and a duration are associated with the data item. When it is read, a time stamp is used to select the data item. The time capsule abstraction includes the notion of "clocks" that ensure periodic data access that is typical for continuous-media applications. By modifying the parameters of a clock, effects such as time lapses or slow motion can be achieved.

1. INTRODUCTION

Computers are becoming increasingly capable of handling *continuous media* (CM) such as audio and video [1]. The immediate benefit of this development is the possible replacement of current single-purpose equipment (VCR's, TV sets, stereo receivers, etc.) with a single machine, resulting in the integration of the varied functions, the improvement of user interfaces, and the formation of multimedia systems [2]. An even greater potential for innovation comes from the computer's abilities to transform CM data according to any given program and to provide flexible interactive access to this data [3]. This offers a whole new spectrum of computer applications – provided appropriate programming support is available.

In this paper we investigate how the handling (storing, retrieval, and transmission) of CM data can be supported by appropriate programming abstractions. We explain why existing abstractions do not suffice and how they can be extended so that programmers can access CM data in the same way as any other data in the computing environment. This programming support is an important step towards our goal of "*integrated digital continuous media*" (IDCM) [4]. In an IDCM system, CM data

- has a digital representation that allows it to pass through standard system components such as CPU, main memory, disk, or network,
- can be processed concurrently with the execution of other kinds of applications, with no adverse effects from contention for hardware resources, and
- is handled in the same software framework (operating system, network protocols, window system, programming language) as other data types.

Only such an integration avoids the need for separate analog devices, networks, dedicated disks, *etc.* to handle CM data and applies the generality of the computer to CM data, using existing generic operating system or user program functions.

Today's proposals for handling CM data in programs often focus on digital signal processing (DSP), *i.e.*, on modifying CM information. They allow programs to be written that compress and decompress data, filter out noise, or achieve special effects. While we think that DSP facilities are essential for the success and operation of future CM systems, we doubt that average programmers will be concerned with data manipulation at this level. It will be sufficient to provide them with a reasonable set of library functions to achieve certain DSP effects.

Our approach in this paper aims at a higher level of programming that is likely to be used more often: the storage, retrieval, transport, synchronization, and presentation of CM data. We pay particular attention to the following two aspects of CM systems:

- Many CM systems are *distributed* [5]. CM data is typically transmitted from remote systems (TV stations, media data-bases, *etc.*) to the user's workstation. Examples are distributed music rehearsal or video conferencing systems.
- *Real-time constraints* are common to all CM systems [6]. CM information has to be available within a certain time to be useful for an application, it has to be handled in a regular fashion to avoid gaps or jitter, and it has to be presented to or obtained from I/O devices at certain rates to fulfill their operational requirements.

Section 2 of this paper describes the particularities of CM data. Section 3 proposes a data abstraction that takes these characteristics into account. Section 4 describes the implications this data abstraction has on the processing of CM data.

2. TIME-CRITICALITY OF CONTINUOUS-MEDIA DATA

While to the listeners or viewers audio and video signals appear to be continuously changing over time, their internal representation in a digital system is discrete. It consists of single audio *samples* or video *frames*. Just as characters or floating-point numbers, these single pieces of CM information constitute *values* of basic data types. These values, however, differ from traditional data in that they reflect a portion of the original signal over a certain period of time. Not just the sequence in which CM values are combined, but also the times at which they shall be processed or displayed (when and for how long) are important to interpret CM data according to its original semantics. Before proposing our own solution to specify time parameters of CM data, we review existing work on time parameters which was the basis for our model.

2.1. Previous Work on Time Information

Programming languages in the process control domain like Ada [7] or PEARL [8] provide mechanisms to schedule or delay processes according to time parameters. Typically, two data types are provided in such languages to express timing constraints: `time` represents values as they can be read from a clock, `duration` represents differences between time values. We also distinguish between these two kinds of time parameters. Their resolution determines how precisely temporal constraints can be expressed. In CM systems, where many data items arrive within a second, values of both types need to have a high resolution. It is usually assumed that this resolution should be in the microsecond range.

While it is common to associate time parameters with process executions [9], few systems offer the possibility of incorporating time information into data and of controlling the data processing by these parameters. MARS [10] was the first process control system we know of which used time parameters to describe how long a message would be valid. If a process received a message after its designated life span, the process would ignore it. We borrow the notion of data life spans from MARS.

In the area of database systems, most work related to time parameters concentrates on how to model data that changes over time. While a traditional database system only records the current state of some information, a *historical database* keeps old information and adds new data including the transaction time at which it is written [11]. This permits queries such as "What was the value of X 10 minutes ago?" This, however, does not allow the inclusion of time parameters in an update, *e.g.*, "Reset the value of X to 100 for the last 10 minutes!" A distinction between *valid time* and *transaction time* solves this problem [12]. In our model, we use this distinction to access time-critical information.

Investigating management techniques for musical data, Rubenstein points out that time parameters should be defined independently for each data item [13]. By this method, different pieces of music can be defined separately and combined later. This consideration reflects that a time value always refers to some *time reference system*; *e.g.*, real time values refer to a time zone, video time values refer to the beginning of the video, *etc.* FORMULA [14] uses a similar model and allows arbitrary mappings between time reference systems. We also make use of different time reference systems.

Finally, several variants of temporal logic can be used to reason about the behaviour of programs. They are useful to express temporal relations, but most of them cannot be used to derive real-time properties and have to be extended to allow a specification or analysis of real-time systems (*e.g.*, [15] extends [16]). For the model we introduce in the following section, an extension of Allen's

interval logic [17] as presented in [18] would be most appropriate as a formal basis.

2.2. Timed Data Streams

We propose the following model to express the time parameters of CM data: Associating a data value with its time parameters, we obtain a *timed data item* m that can be represented as a triple

$$m = (V, T, U)$$

where V is a *data value* of some *base type*, T specifies a *time value* (m's "time stamp"), and U gives a *duration*. The time parameters define a *life span* L: T indicates the beginning of the life span, U determines the end of the life span relative to T.

$$L = [T, T + U)$$

Here ")" indicates that the second bound $T + U$ does not belong to the life span. For notational convenience, let us write $m.V$ in the following to denote the value of item m, $m.T$ to denote its time stamp, *etc.*

A sequence of timed data items with data values of the same base type and time parameters in a common time reference system constitutes a *timed data stream* s. Items are ordered by increasing time stamps; their life spans must not overlap.

$$s = \{m_i\}: \quad \begin{array}{ll} m_i.T < m_j.T, & \forall\, i < j \\ m_i.L \cap m_j.L = \emptyset, & \forall\, i \neq j \end{array}$$

A timed data stream may contain no items for certain life spans. We define a partial *evaluation function* V for a timed data stream at a time t as follows:

$$V(t) = \begin{cases} m_i.V & \text{if } \exists\, i : t \in m_i.L \\ \text{undef} & \text{otherwise} \end{cases}$$

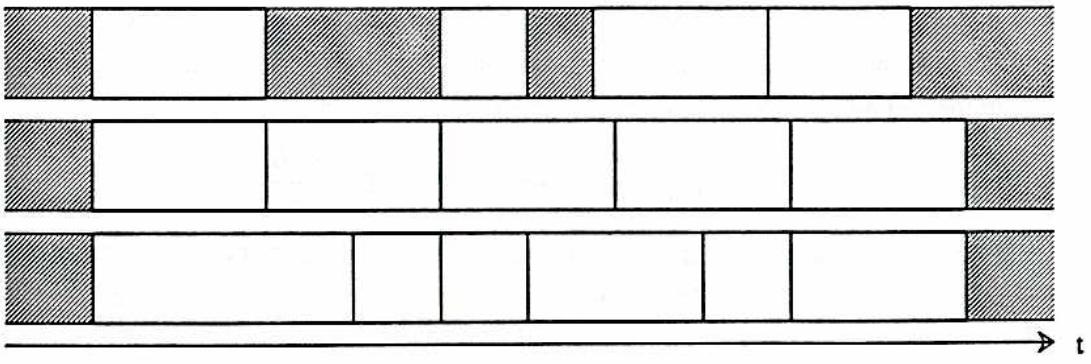

Figure 1: Different kinds of timed data streams.
Above: Non-continuous. Middle: Periodic continuous. Below: Aperiodic continuous.

Depending on the result of the evaluation function, we can distinguish two classes of timed data streams (illustrated in Figure 1):

- In a *non-continuous* timed data stream, life spans where the stream has a value are separated by life spans where its value is undefined.

$$\exists t: \quad s.V(t) = \text{undef}, \\ \exists p: (p < t, s.V(p) \neq \text{undef}), \\ \exists q: (q > t, s.V(q) \neq \text{undef}))$$

- In a *continuous* timed data stream, all data items have adjacent life spans.

$$\forall i: m_{i+1}.T = m_i.T + m_i.U$$

In this paper, we are concerned with continuous streams only. They can be classified further as shown in Figure 1:

- In *periodic* continuous streams, all life spans have the same length. This length constitutes a parameter of the stream. For each periodic stream, U defines the *value duration*.

$$\forall i: m_i.U = s.U$$

  Examples of periodic continuous streams are typical audio and video signals.
- In *aperiodic* continuous streams, life spans can have different lengths. For tractability, we assume a certain granularity of value durations in such streams, given by the *minimum value duration* U_{min}.

$$\forall i: (\exists n \in \mathbb{N}: m_i.U = n\, s.U_{min})$$

  Examples of aperiodic continuous streams are sequences of still video images (such as a slide show).

For some applications, aperiodic streams offer the advantage of more efficient encoding. However, since typical audio and video signals are our major concern, in the following we will concentrate on periodic continuous streams. This implies no loss of generality because aperiodic streams can be transformed into periodic ones as follows: According to the common understanding of time intervals, every value that is valid in a certain time interval is also valid in any subinterval of this interval. Therefore, every timed data item m_i can be substituted by n items m_{i_0}, ..., $m_{i_{n-1}}$ in the following way for any $j = 0, ..., n-1$:

$$m_{i_j} = (m_i.V, m_i.T + \frac{j}{n} m_i.U, \frac{m_i.U}{n})$$

Although the number of items in the stream is increased, the semantics of the stream remain unchanged because the life span of each item is decreased accordingly (see Figure 2). By choosing

$$n = \frac{m_i.U}{s.U_{min}}$$

for each m_i in the aperiodic stream, all items obtain the same duration, making the stream periodic.

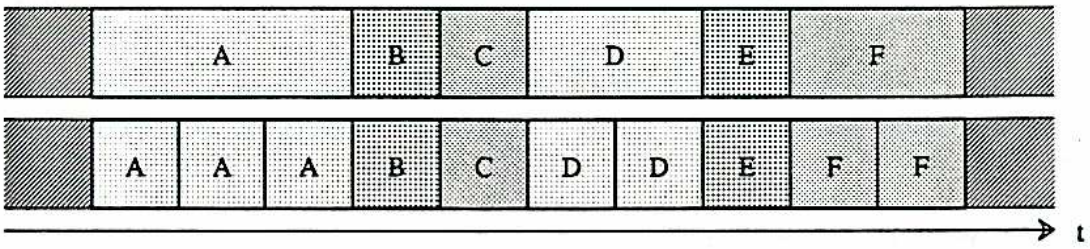

Figure 2: "Acceleration" of continuous timed data streams.
Above: Original aperiodic stream. Below: Corresponding periodic stream.

3. TIME CAPSULES

To store timed data streams, containers are needed. Traditional sequential files could be used for this purpose, but they do not reflect that time values rather than byte positions are the ordering criterion for such a stream and that time parameters govern all access operations. In this section we introduce *time capsules* as containers for periodic continuous timed data streams. A time capsule is declared by specifying the base type and the value duration of the data stream it stores.

3.1. Clocks

To facilitate the specification of time parameters in time capsule access operations, we introduce the abstraction of *clocks*. Assuming an underlying notion of physical ("real") time at its location, a clock c is defined as a quadruple

$$c = (R, S, V_0, T_0)$$

where the parameters have the following meaning:

- The *rate* R determines the number of clock ticks per second. Its reciprocal defines the real time interval between clock ticks and, thus, gives the granularity of clock values.
- The *speed* S determines the difference in clock values per second. It defines the progress of clock values. A speed larger than 1 causes the clock to move faster than real time, a speed less than 1 causes it to move slower. A negative speed causes the clock to move backwards.
- The *initial value* V_0 defines the value of the clock at its first tick.
- The *start time* T_0 defines the real time of the first clock tick.

A clock c can be understood as a periodic continuous timed data stream for which real time at the clock's location provides the time reference system. The i'th item m_i in the stream is defined as

$$m_i = (c.V_0 + i\,\frac{c.S}{c.R}, c.T_0 + \frac{i}{c.R}, \frac{1}{c.R})$$

In the following, we assume that clock parameters remain constant. How clocks behave when parameters are changed dynamically is an important subject for further study.

3.2. Time Capsule Access

Time capsules provide access operations similar to traditional files. As with a file, a process has to open a time capsule before using it. Time capsules can be opened for reading or writing. A process obtains a *stream handle* from the open call. In this call, a clock is associated with the stream handle that determines the time parameters for future access operations.

Every access operation has a time stamp t. (Section 4.2 will describe how t is obtained.) t determines a clock value that gives the time stamp of the data item to be read or written. Let c be the clock of the stream handle and s the timed data stream stored in the time capsule. In a read operation, we obtain the value

$$v = s.V(c.V(t))$$

In a write operation, we deliver the timed data item

$$m = (v, c.V(t), s.U)$$

Although every value in a timed data stream has its own life span, it is inefficient to handle values one-at-a-time or to provide time information for every single value. For example, a time stamp would require more bytes than a typical audio sample. Therefore, we propose access functions to read or write a *block* of N values at the same time (again, this is similar to files). We define these operations to have the same effect as N periodic operations accessing only a single value. The values obtained when reading a block are (for $i = 0, ..., N-1$):

$$\{v_i\} = \{s.V(c.V(t + \frac{i}{N\ c.R}))\}$$

The items delivered when writing a block are (for $i = 0, ..., N-1$):

$$\{v_i\} = \{(v_i, c.V(t + \frac{i}{N\ c.R}), s.U)\}$$

To write a periodic timed data stream or to access all information contained in a time capsule, block size, value duration, clock speed, and clock rate have to fulfill the following relation:

$$N\ s.U = \frac{|\ c.S\ |}{c.R}$$

While this relation has to be enforced when writing a stream, processes can read streams with other parameter settings and achieve the effects described in the following section.

3.2.1. Varying Access Parameters

By using other clock and block size parameters for read operations than had been used for writing we can cause special effects, many of which are known from today's analog CM equipment – in particular from advanced VCR's – and will be expected to be available by users of future CM systems.

Reading from a time capsule at a different rate than data was written to it allows adjustment for differences in device characteristics between the input and output of CM data. Unless the read rate is the reciprocal of the value duration (assuming a speed and a block size of 1), values are eventually skipped or duplicated because the times at which they are read do not coincide with the beginnings of life spans. This is shown in Figure 3. For some types of data, the loss or duplication of values reduces the signal quality. In these cases, rate changes have to be accompanied by value interpolation.

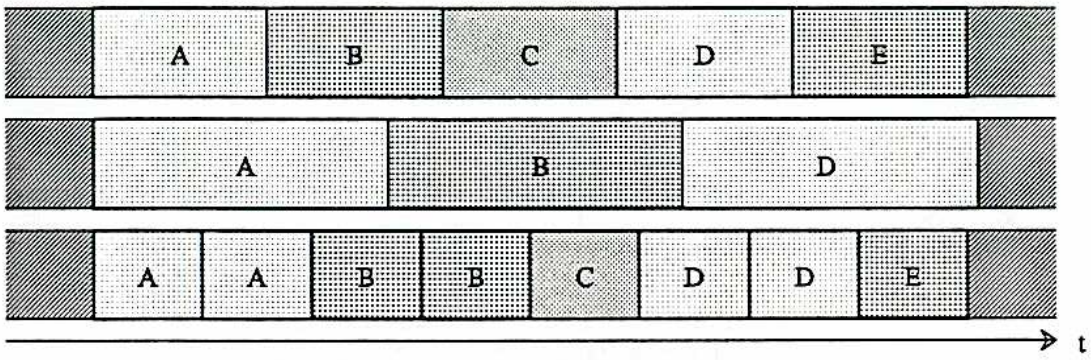

Figure 3: Reading timed data streams with different rates.
Above: Original data. Middle: Reading with smaller rate. Below: Reading with higher rate.

If the speed of the read clock is lower than the speed of the write clock, data will be retrieved in slow motion. Assuming an unchanged rate, the same value will be accessed more often. If the read clock runs at a larger speed than the write clock, the result is a time lapse. It is achieved by skipping values when displaying them. If the speed direction of the read clock is different from that of the write clock, data will be retrieved backwards. Figure 4 shows these effects.

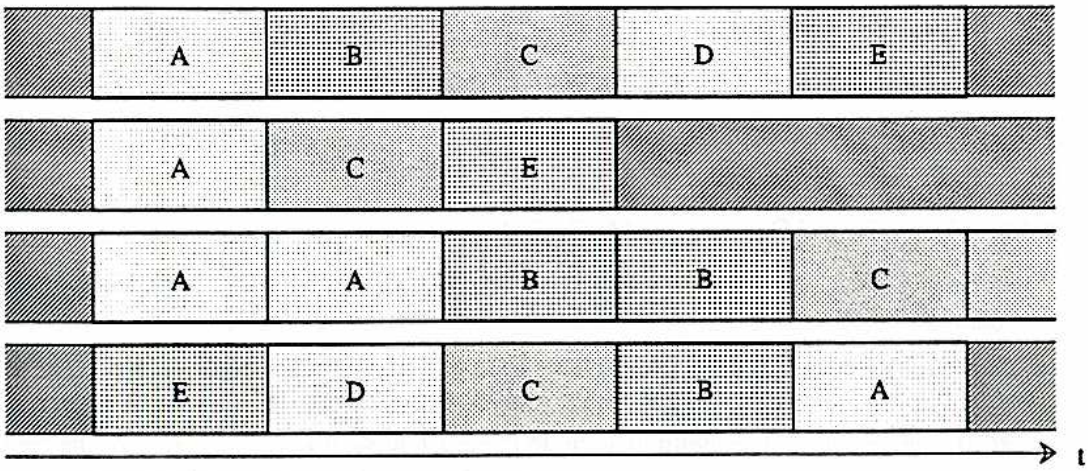

Figure 4: Reading timed data streams with different speeds.
Above: Original data. Upper middle: Double speed. Lower middle: Half speed. Below: Backwards.

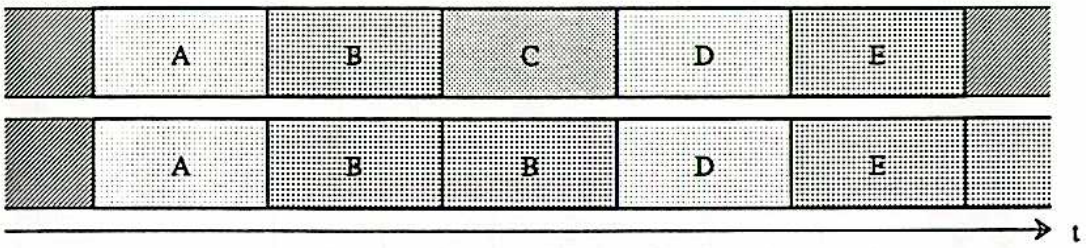

Figure 5: Reading timed data streams skipping values.
Above: Original data. Below: Reading 2 values out of 3.

By assigning a different initial value to a read clock than to a write clock, the initial data retrieved can be different from the first data written. Assuming positive speeds, initial values that are smaller than the smallest time stamp in the data stream delay the retrieval of the data for some time, values larger than this time stamp cause the data to be retrieved at some arbitrary position in the stream. One could, *e.g.*, start the display of a video at its 15th minute, or one could start it at the end and – in combination with a change in speed direction – display it backwards.

If not all values within the life span designated by the read clock are retrieved because the user has not specified a large enough block size, a stroboscopic effect results that "freezes" the value occasionally (snapshot). Again, some frames in the data stream are skipped, as illustrated in Figure 5.

3.2.2. Value Representation in Time Capsules

Changes in time capsule access parameters as described in the previous section may cause that timed data items are not accessed in their original order. In some cases, the way in which a stream of CM data is encoded can make this operation difficult.

3.2.2.1. Encoding of Continuous-Media Data

With many standardization efforts still pending, different representation formats are common for CM data today [19]:

- Audio data can be encoded in 16-bit *linear PCM* samples, the format used with the compact disc. Several derivatives of this encoding scheme (*differential PCM*, *adaptive differential PCM*, *etc.*) exist. Other methods include the *mu-law* 8-bit encoding used for the NeXT station audio input hardware or the *u-law* or *A-law* 8-bit samples in the Sun Sparcstation.
- Among the digital video encoding schemes used today are the *CD-I* format marketed by Sony and Philips, CCITT's *MPEG* proposal, and the format used with Intel's *DVI* hardware. (Well-established color video formats like *NTSC*, *PAL*, or *SECAM* are analog. Since only digital CM data offers the advantage of integration mentioned in the introduction, we do not concern ourselves here with the handling of these data formats in a computer system. Note that some HDTV proposals currently being discussed also still use analog video representations.)

Single values of CM data rarely occur by themselves, but are rather combined into a sequence, *e.g.*, representing a piece of music or a motion picture. Compared with traditional data, these sequences require a significantly large amount of memory. One second of high-quality audio requires approx. 82 KBytes per channel; one second of 512×480-pixel video frames with 24-bit RGB color encoding requires approx. 22 MBytes of storage. Therefore, video – and sometimes audio – is stored and forwarded in *compressed* form.

Compression techniques can be classified as *intra-* and *inter-value* methods: intra-value methods compress each value separately; inter-value techniques take advantage of similarities between adjacent values in a sequence. An important inter-value technique is *differential* compression: for each value, only the "difference" to its preceding value is given. Since differences between values tend to be not as large as the values themselves, fewer bits are needed to describe the data. While for audio signals a differential compression factor of 4 is common, a typical factor for video is 15 because usually the variation between adjacent video frames is much smaller.

If differential compression is used, *self-contained* and *differential* values of CM data have to be distinguished. Self-contained values not only occur at the very beginning of a sequence, but may be inserted into the sequence from time to time to start the differential decompression process anew to recover from occasional bit errors. In a differentially compressed sequence of CM data, values cannot be treated independently: they are context-sensitive.

3.2.2.2. The Differential Value Problem

If values are skipped or duplicated when reading from a time capsule, differential values cannot be interpreted correctly. We call this the "*differential value problem.*" To solve it, the time capsule implementation has to ensure that every read operation delivers values in a representation that can be processed correctly by the client. When a time capsule delivers a value it should reflect all changes in respect to the last value delivered previously.

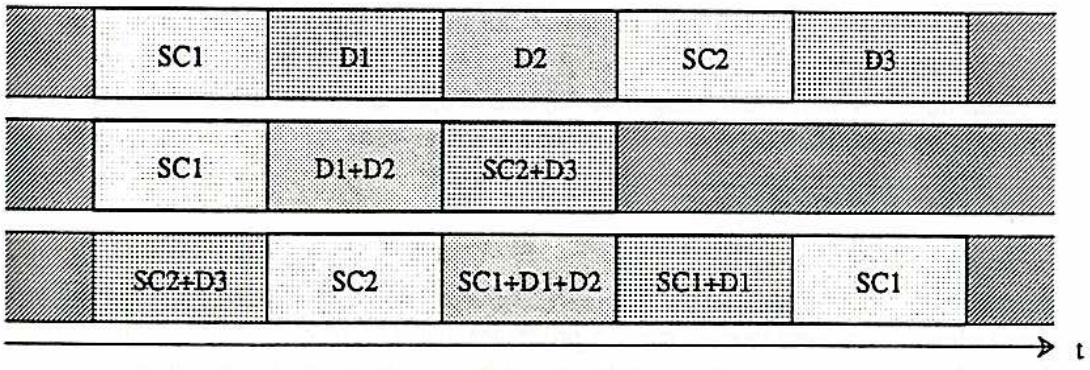

Figure 6: Resolving the Differential Value Problem.
Above: Original data. Middle: Double speed, forwards. Below: Backwards.

If the last value is accessed repeatedly, a "zero difference" value can be delivered. If some values were skipped and the next value is not self-contained, all differences since the last value have to be combined into the next value to be delivered. How this combination can be achieved depends on the compression scheme used and has to become part of the implementation of time capsule access functions.

The same effect can be achieved if the time capsule delivers all values in a self-contained encoding. In general, this solution is worse because it increases the number of operations for each time capsule access and the amount of data delivered. However, it is unavoidable if a stream is played backwards. Here, the time capsule has to determine the last previous self-contained value in the regular ordering and apply to it all differential values in between to obtain the value to be delivered. Figure 6 gives some examples for value combinations in a differentially encoded stream.

3.3. Time Capsule Variants

The UNIX file system has shown that the file abstraction is not only useful to store data, but also for communication between processes (through named pipes) and for I/O operations (through special files). The same variants are possible for time capsules, *i.e.*, time capsules serve as a general abstraction with different implementations.

3.3.1. Recording Time Capsules

Recording time capsules store timed data streams permanently. They can be implemented on top of an ordinary file system if the access times provided by this system are acceptable. The underlying file system should support large block sizes to read and write information continuously, and it should avoid extensive seek operations. Although not aimed at CM applications, these requirements were among the design criteria for the UNIX Fast File System [20] that was chosen as the basis for the Sun Multimedia File System [21]. (In fact, we could use the Sun Multimedia File System to implement time capsules if we schedule disk access operations according to real-time criteria as described in Section 4.2.1.)

Recording time capsules can be organized as follows: Data and time information is kept in separate files. All time information is collected in one file, containing pointers to the corresponding data items. When a recording time capsule is opened for reading, these pointers – ordered by time stamps – can be cached for faster access. Rubenstein has used a similar database organization with A-trees for storing musical information [13].

3.3.2. Timed Pipes

To exchange timed data without storing it, *timed pipes* can be used. Any data item written into a timed pipe is conveyed to one or more processes that read from the pipe. Values are discarded at the reader's side if their life span is not within the range of future clock values. All clocks accessing a timed capsule need to have the same speed. If a reader clock could have a higher speed than a writer clock, the reader would eventually get no more values. If a reader clock could run slower, data would accumulate without bound in the pipe.

Timed pipes can also be used to exchange timed data items in a distributed system. The non-negligible delay in transferring values from one system to another may cause a situation where a value has been written, but is still in the network and not available to the receiver. Let D_{max} be the largest delay until a message can be delivered to a reader. Assuming a global time reference

system, this abnormality will not occur if the clock value difference between a writer clock c_w and a reader clock c_r is at least D_{max}.

$$\forall t : c_w.V(t) \geq c_r.V(t) + D_{max}$$

We will continue investigating the problems of distributed time capsules in Section 4.

3.3.3. Special Time Capsules

Just as I/O devices are represented by special files in UNIX, we represent devices that produce or consume CM data by *special time capsules*. This makes devices like cameras, microphones, monitors, or speakers accessible in the same way as recording equipment that is hidden behind recording time capsules.

The operational characteristics of a device determine the possible clock rates for accessing a special time capsule and whether it can be accessed in read or write mode. Input and output occurs in the "real" world, therefore, the clocks used to access a special time capsule progress in real time.

4. PROCESSING ASPECTS

In the description of processes that handle CM data, two issues can be distinguished [22]:

- The description of *data flow* defines from which time capsules values are read, through which they are communicated, and to which they are finally written.
- The description of *control* defines which operations have to be applied to the data.

To describe, *e.g.*, a video-conferencing system as it is illustrated in Figure 7, we would define how audio and video signals from each sender are combined and how then these combined signals are mixed so that they can be displayed. The flow of data is defined using time capsule names:

```
camera@cory ⊞ microphone@cory -> signal@cory
camera@evans ⊞ microphone@evans -> signal@evans
camera@icsi ⊞ microphone@icsi -> signal@icsi

signal@cory ⊕ signal@evans -> tv@icsi
signal@cory ⊕ signal@icsi -> tv@evans
signal@evans ⊕ signal@icsi -> tv@cory
```

In the description of a multimedia show, we would define when to start each slide, film, or narration sequence. Such a description can be seen as a "multimedia document": a play list defines when each element of the document is shown. Describing a document by defining the algorithm for its presentation is not uncommon – Postscript [23], *e.g.*, is based on this approach and could be extended to accommodate CM.

```
00:00:00 title_slide -> middle_screen
00:00:30 hawaii_map -> left_screen
         volcano_slide_sequence1 -> middle_screen
         volcano_slide_sequence2 -> right_screen
         volcano_narration -> audio_out
00:05:00 hawaii_movie -> middle_screen
```

We will not concern ourselves here with syntactical issues. Instead, we propose an abstract structure for CM applications and show how programs implemented according to this structure are executed.

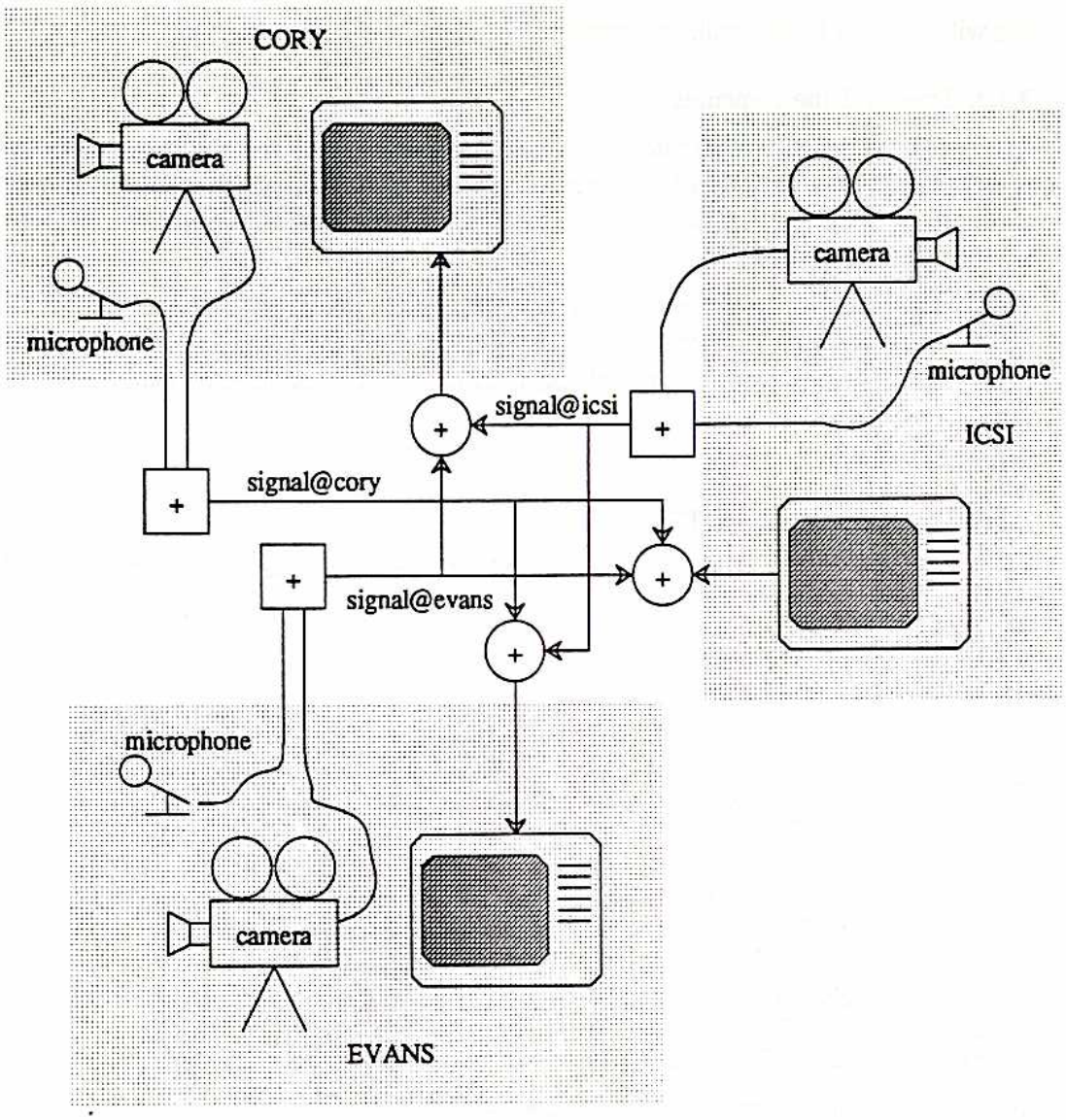

Figure 7: A video-conferencing application.

4.1. Structure of Continuous-Media Applications

Like any distributed application, a CM system is a collection of cooperating processes. Each process has a number of ports for input and output. Each port is defined by the *base type* of values that can pass through it. In our model, ports are connected to time capsules. Obviously, ports can only be connected to a time capsule of the same base type. We can represent the connections in an *application graph* G in which time capsules T, input ports I, processes P, and output ports O alternate. G is a tuple of nodes and edges defines as

$$G = ((T \cup I \cup P \cup O), (T \times I \cap I \times P \cap P \times O \cap O \times T))$$

An example of an application graph is given in Figure 8. For reasons discussed below, such a graph has to be acyclic. In this paper, we deal with static application graphs only, *i.e.*, we do not modify the graph once the application is started. Obviously, this excludes some applications from being modeled. Dynamic control in a CM system is a subject of our ongoing investigation.

In the application graph, time capsules without writers are called *sources* and time capsules without readers are called *sinks*. Sources and sinks are either recording or special time capsules. All time capsules that are neither sources nor sinks serve for communication between processes and are timed pipes. To avoid conflicts between two processes writing into the same time capsule, time capsules can only be connected to one output port.

For each process port, the programmer can define the speed, initial value, and start time of the clock controlling the time capsule access. As seen in the second example of the previous section, start times are defined relative to the start of the application. To facilitate the specification of clock start times, the programmer can assume that the flow of data through the graph does not take any time. We describe later how the system can determine delays and take them into account to achieve the proper synchronization.

Each process implements a function. Each function can read and write values in blocks. The block size is specified in the function interface.

$$p : i_1 [N_{i_1}] \times i_2 [N_{i_2}] \times \cdots \rightarrow o_1 [N_{o_1}] \times o_2 [N_{o_2}] \times \cdots$$

We do not regulate how such a function is implemented. It may, *e.g.*, combine two different streams of audio and video to a single stream. (In the terminology of [4] this is called "to weave a rope from two strands.") The process can also apply some DSP algorithm to the data. For reasons discussed below, we assume that functions have no side-effects and that an upper bound for the function servicing time, *i.e.*, the time for which the function needs the processor to be

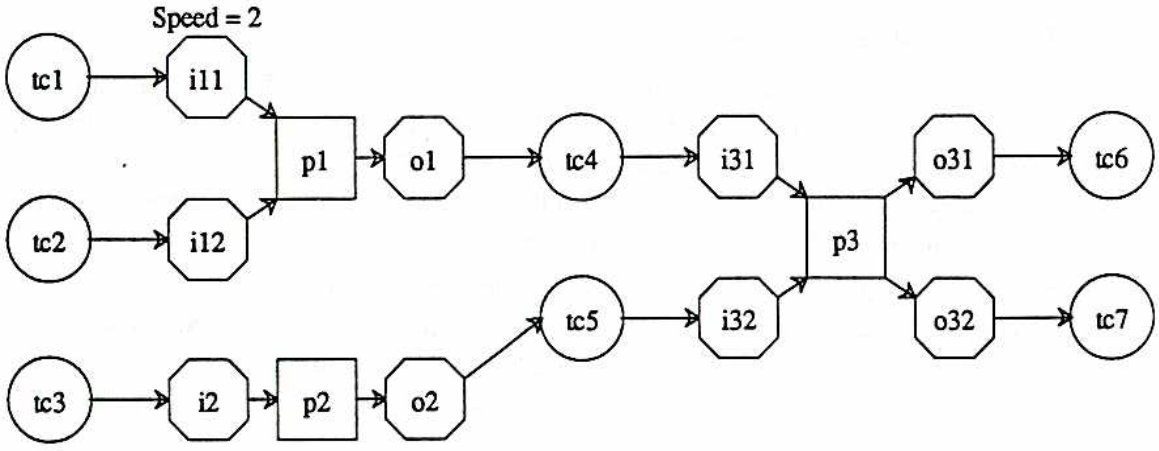

Figure 8: Graph with time capsules (circles), ports (octagons), and processes (squares).

executed, is known. It can either be calculated algorithmically or determined by monitoring tools such as [24].

The semantics of process functions can depend on the time at which they are performed. For example, in a department store monitoring system the programmer could specify that every 15 seconds the camera selected for input should be changed. To specify these changes, for each process the programmer can specify a *time table* that selects the operation to be performed by its function. To interpret this time table, each process is assigned a *process clock* that is used in the same way as time capsule clocks. If time-dependent changes result in different function servicing times the largest of these times needs to be known. For reasons given in the following section, it is recommended to implement two processes with different start times instead of one process with two time-dependent functions if the difference between servicing times for these functions is large.

4.2. Execution of Continuous-Media Applications

Traditionally, processes are *input-driven*: They are blocked until a complete set of input data arrives. By way of contrast, CM systems are *output-driven*: The need for a new output value determines when a process has to be executed. If output occurs less frequently than input, this principle is known as "lazy evaluation". In CM systems, however, there may be more output than input values, so this term would be misleading.

The last equation from Section 3.2 defines the rate of a clock c at which a writer has to service an output time capsule to meet the specification of the output stream s. This rate is

$$c.R = \frac{|\ c.S\ |}{N\ s.U}$$

If a process has more than one output port, all ports need to be serviced with the same rate. If different output rates result from the programmer's specification, a specification error occurred and the application has to be restructured so that each output capsule is serviced by its own process. This is shown in Figure 9.

4.2.1. Scheduling in the DASH Resource Model

To schedule process executions, we use the *DASH Resource Model* (DRM) [25], [26], [27]. This model allows us to determine delay bounds for each process execution and to ensure through corresponding scheduling that they are observed. In the DRM, each process establishes a *session* with the resources (CPU, disk, network, *etc.*) it uses. It provides a specification of the maximum workload of this session and, in return, obtains guarantees on how this workload is handled.

Workload specifications are given as *linear bounded arrival processes* (LBAP's) that specify a *maximum rate* R_{max} and a *maximum burst* B_{max} of work items. In any time interval of length t, the number of work items arriving at the interface may not exceed

$$B_{max} + t\ R_{max}$$

While R_{max} gives the long-term rate of the LBAP, the burst parameter B_{max} allows short-term violations of this rate constraint. Using B_{max} we can model programs and devices that generate "bursts" of messages that would otherwise exceed the rate constraint.

We define a function $b(m)$ representing the *logical backlog* of the arrival process. This is the number of work items by which the arrival process is "ahead of schedule" (relative to its long-

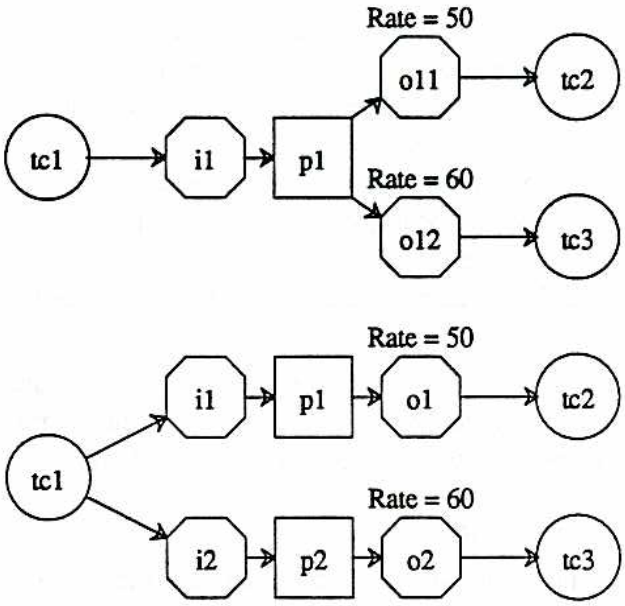

Figure 9: Modification of process structure.
Above: Conflicting output rates. Below: Conflict resolved.

term rate) when work item m arrives. The logical backlog is not necessarily the number of queued work items since a resource may process messages upon their actual arrival if it is fast enough. If t_i is the arrival time of work item m_i, $b(m)$ is defined by

$$b(m_0) = 0$$
$$b(m_i) = \max(0,\ b(m_{i-1}) - (t_i - t_{i-1}) R_{\max} + 1)$$

Using $b(m)$, we define the *logical arrival time* $l(m)$ as

$$l(m_i) = t_i + \frac{b(m_i)}{R_{\max}}$$

Intuitively, $l(m)$ is the time m would have arrived if the LBAP strictly obeyed its maximum rate.

For each session, the system reserves some fraction of the overall resource capacity to guarantee a *maximum logical delay*. This delay gives the maximum time between the logical arrival of a work item and its latest completion. It results from the servicing time of each work item and the competition among sessions for the resource. If work items arrive "ahead of schedule" they can be subject to an actual delay that is the sum of the time by which they arrive too early and the maximum logical delay. In the DRM, the system always returns the smallest possible maximum logical delay that can be achieved at the moment. If this delay is smaller than needed, the client can relax the reservation.

Future CM applications will have to share the CPU of a system with traditional non-real-time applications. This results in a scheduling conflict: time-critical work items should not be blocked by time-sharing work items, but time-sharing processes should not starve because of time-critical

ones, too. Using the DRM notion of logical arrival times, this conflict is resolved by a three-level preemptive scheduler:

- The first scheduling queue keeps work items for which the logical arrival time has already arrived. These items are deadline-scheduled. The deadline is calculated as the sum of the logical arrival time of the work item and its maximum logical delay.
- The second scheduling queue keeps processes that are not time-critical. These items can be scheduled according to any time-sharing scheduling algorithm, *e.g.*, in a Round-Robin fashion applying utilization feedback as in today's UNIX systems.
- The third scheduling queue keeps time-critical work items for which the logical arrival time has not yet arrived. These items, again, are deadline-scheduled. The deadlines are calculated as above.

If no work items are in the first queue, work items from the second queue are executed. If the second queue is also empty, work items from the third queue are executed. Once the logical arrival time of a work item in the third queue arrives, this item is moved into the first queue.

4.2.2. Process Execution Rates

Each process in a CM system structured according to our model reads data, executes its function and writes data. We partition each process into at least three *threads* [28] which are scheduled independently:

- one *read thread* for each input port, controlled by the corresponding input clock,
- a *function thread* to execute the process function, controlled by the process clock, and
- one *write thread* for each output port, controlled by the corresponding output clock.

All these threads pass data through queues: the function thread reads one value each from the queues of the read threads and writes one value to the queue of each write thread. All thread clocks run at the same rate. Each clock tick schedules the execution of the corresponding thread as a work item, *i.e.*, the clock rate determines the maximum rate according to the DRM. The time of the clock tick is the time stamp of this work item.

Input data for a work item may be available before the corresponding clock tick. If the output of a process is not directed to a special time capsule, where on-time execution is crucial, we find it desirable to make use of the "work-ahead" allowed by the DRM. If a thread can work ahead it can make use of otherwise idle resources, freeing its reserved future time slot so that this slot is available for other purposes (in particular, for time-sharing processes). In addition, fewer context switches may occur if work-ahead is allowed. The programmer can specify a *maximum work-ahead time* A_{max} for each process. This results in a maximum burst of work items of

$$B_{max} = A_{max} R_{max}$$

If work-ahead occurs, the time stamp of each work item is its logical arrival time. Each work item blocks until input is available – either in the time capsule or the queue it accesses.

4.2.3. Process Start Times

To make sure that each work item gets its input on time, *i.e.*, finds a value in the time capsule or queue when the logical time for reading has arrived, we have to take the delay of the previous work items into account. As before, the output determines when to schedule an item: If the clock of a write thread w ticks at a certain time, we have to ensure that the function thread f is

scheduled far enough ahead in time so that it has terminated when w is started. We can achieve this by letting the clock of f tick earlier than that of w. For each clock, we determine a *start time delay* ΔT_0 that is added to the user-defined start time of each thread clock. ΔT_0 ensures synchronization throughout the application. The following two properties are desired for ΔT_0 regardless of potential work-ahead:

(1) No thread should be delayed longer than it is needed for all of its input to arrive.

(2) No data item should be sent if the receiving thread would not be able to service it even if it arrives with maximum delay.

The first property ensures that delay is minimized, the second minimizes buffer space.

4.2.3.1. Synchronizing Process Starts

A two-phase protocol is used to determine the start time delays of each clock. The first phase traverses the nodes of the application graph from the sources to the sinks, the second phase proceeds in the opposite direction. Termination of the protocol is ensured because the application graph is acyclic.

For the purpose of the protocol, each node n is assigned two parameters: a *time value* V and a *delay* D. The time value will be calculated during the two-phase protocol and will later be used to define start time delays. The delay parameter contains the delay between the logical arrival of a value at the node and its latest departure. For nodes representing a port or a process, D is the maximum logical delay of the corresponding thread. For time capsule nodes, D is 0.

An additional delay may have to be added to each input port node connected to a timed pipe. While the speed for accessing a timed pipe is always 1 (*cf.* Section 3.3.2), processes may read and

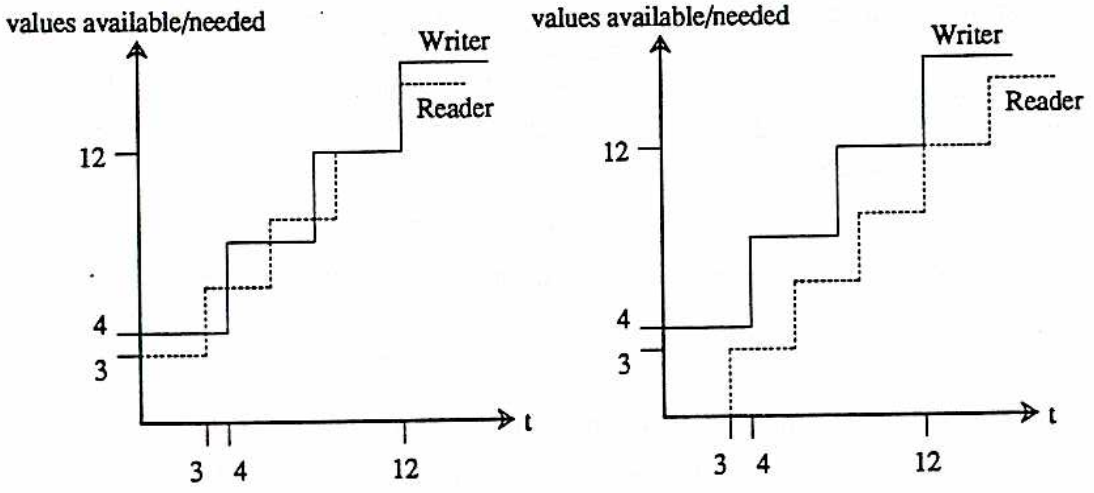

Figure 10: Delay in blocked reading and writing.
Left: Conflicting operations. Right: Delayed reading.

write data in different block sizes. If the write rate is not divisible by the read rate, the reader will eventually try to access a value that has not been written should it be started immediately after the first write block is available. This is avoided if the reader is delayed for one clock tick. Figure 10 illustrates this situation (and gives a geometrical proof): The x-axis shows the times at which read and write operations take place, the y-axis identifies the block of values accessed at this time. In this example, one process writes with $N = 4, R = 1/4$, another process reads with $N = 3, R = 1/3$. The reader has to be delayed by 3.

The operation of the protocol is illustrated in Figure 11. It proceeds as follows:

(1) Phase 1 determines how long it takes a timed data item obtained at a source to generate a corresponding data item at a sink. It uses the classical critical-path algorithm (see, *e.g.*, [29]) to calculate the time at which a data item leaves the node at the latest. If n is a source node, its value is

$$n.V := 0$$

If i is a node from which there is an edge to n, n gets the value

$$n.V := \max_i (i.V) + n.D$$

(2) Phase 2 determines when a data item has to be delivered by a source to arrive on time at the sink without being buffered needlessly. It uses an inverse critical-path procedure to derive the time at which a data item has to enter a node at the latest. If n is a sink node, its value remains unchanged. If o is a node to which there is an edge from n, n gets the value

$$n.V := \min_o (o.V) - n.D$$

To set the start times of clocks, we now choose some future real time τ_0 at which we start to run the entire application. τ_0 has to be far enough in the future to allow us to set all clock start times. (Again, we assume the time it takes to set clocks is bounded. We could establish a system session as described in Section 4.2.1 for this purpose.) For clock c of a thread represented by the node n we obtain a start time delay of

$$c.\Delta T_0 := n.V + \tau_0$$

This start time delay fulfills the properties stated above. Finally, the actual start time of the clock is set to

$$c.T_0 := c.T_0 + c.\Delta T_0$$

using the relative start time previously defined by the programmer.

For some applications, the *end-to-end delay*, *i.e.*, the difference in start time delays of source and sink clocks, is crucial. The end-to-end delay determines when a value sent by a source causes a corresponding value to be delivered to a source. When a video is played back from disk, it does not matter whether the end-to-end delay is in the order of a few milliseconds or a few seconds: any difference only affects the time it takes for the very first image to appear on the screen. People are used to their TV sets not displaying a picture immediately after being switched on, so they will not expect immediate responses from their computers. In applications like distributed music rehearsal, where some feedback between the CM data a user gets and the data he sends out oc-

curs, end-to-end delays are much more stringent. For every application, the programmer should be able to specify a desired end-to-end delay bound.

4.2.3.2. Start Time Delays in a Distributed System

Clocks are implemented on top of physical clocks (which, for the purpose of distinction, we will call *chronometers* in the following). In a non-distributed application, all clocks are based on the same chronometer and share a common time reference system, but in a distributed system each host has its own chronometer. The clock start times will, therefore, be measured with respect to these different chronometers. To make the algorithm presented in the previous section work, we either have to ensure that all chronometers show the same value at any time (*i.e.*, let them all provide the same time reference system) or take the difference in chronometer values into account.

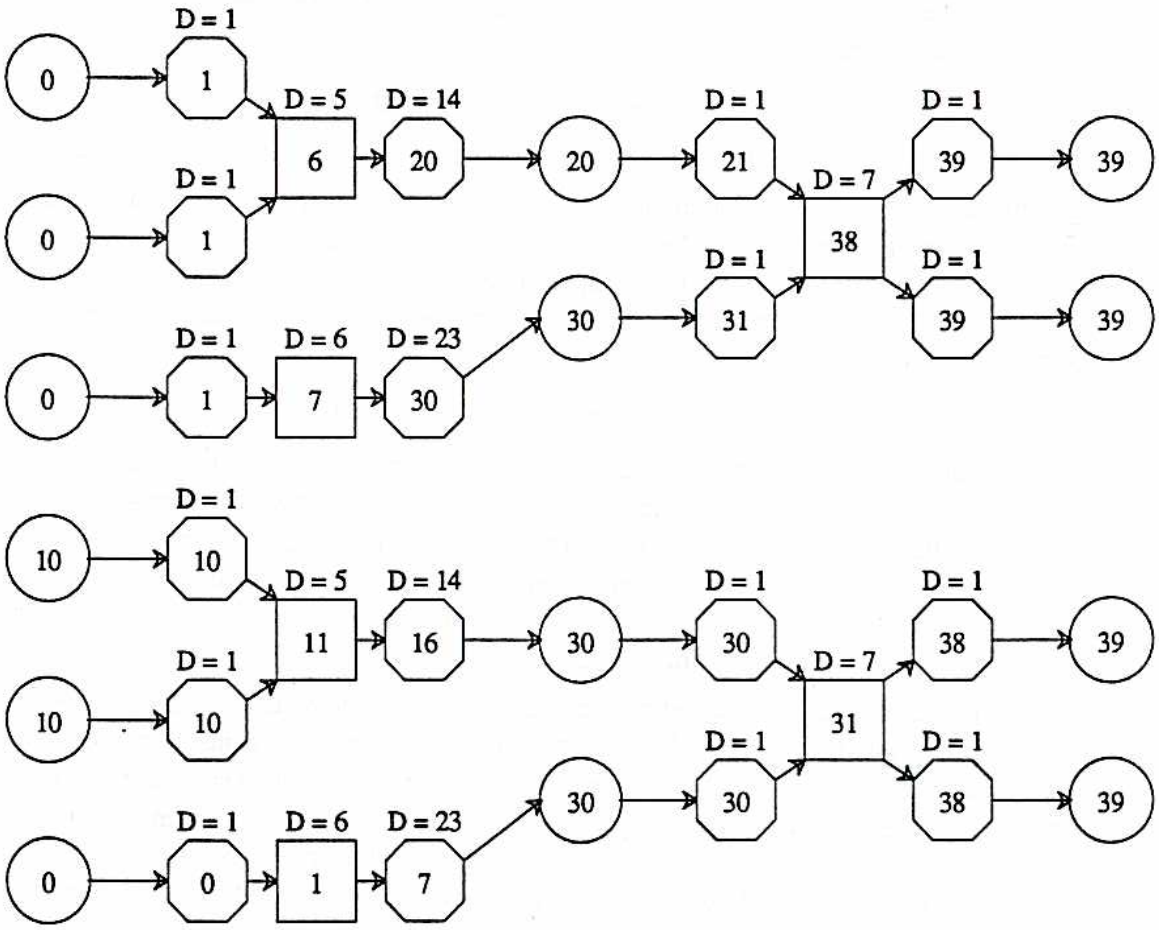

Figure 11: Determining clock start times.
Above: Phase 1 – Latest Termination. Below: Phase 2 – Latest Start.

Numerous papers have shown how to synchronize chronometers in a distributed system (*e.g.*, [30] or [31]). They have also shown that the accuracy of synchronization is limited because of synchronization message delay and chronometer drift. A sufficient synchronization accuracy is only achievable in local distributed systems. In large systems containing autonomous hosts as they will be used for CM applications, the feasibility of global chronometer synchronization is questionable because of both technical and administrative difficulties. Our model (due to its acyclic data flow) enables us to take unsynchronized chronometers into account, albeit by imposing larger start time delays.

As described in [30], a difference in chronometer values can be detected in the following way: Let s be a sending host and r be a receiving host underlying the application. A message containing the chronometer value of the sending host is transmitted to the receiver where it is compared with the recipient's chronometer value. All chronometer values (denoted as C) are determined as close to the message transmission as possible. Let D_{min} be the *minimum transmission delay* of a message between a sender and a receiver. The *lead* ΔC of the chronometer of r against the one of s is then calculated as

$$(s,r).\Delta C = r.C - (s.C + (s,r).D_{min})$$

If ΔC is positive, r's chronometer is ahead; if ΔC is negative, it runs behind.

Assuming that the message containing a sender's chronometer value arrived with minimum delay is conservative: the message may have arrived with a larger delay than D_{min}, so that ΔC is larger than the actual difference. ΔC gives an upper bound on the lead of r's chronometer. Since the message may have arrived with the *maximum transmission delay* D_{max}, ΔC contains a *delay error* ΦC of up to

$$(s,r).\Phi C = (s,r).D_{max} - (s,r).D_{min}$$

The result of the calculation of chronometer leads can be represented as a directed graph of hosts where the edges contain the ΔC values. If an edge contains a negative value, this edge is inverted by changing its direction and reversing its sign. Since the host graph may contain cycles (even if the application graph does not), we can end up with two edges connecting the same two nodes. Only the edge with the smaller value is needed because if all ΔC values give upper bounds, the smallest of them gives the best approximation.

We can now choose an arbitrary node a representing a chronometer that serves as our ultimate time reference system. We leave the start times on a unchanged and adjust start times on all other hosts relative to a. Beginning at a, we construct the shortest-path spanning tree of the host graph (in regard to the ΔC values of its edges). The direction of edges is unimportant in this processing step so that any algorithm for non-directed, weighted graphs can be used (for a survey see, *e.g.*, [29]). For each node n, we obtain the *start time correction* ΨT_0 summing up the values of edges in the path between n and a: the value of an edge e pointing away from a is added, the value of an edge f pointing towards a is subtracted. ΨT_0 is added to all start time delays of threads on n.

$$n.\Psi T_0 = \sum_e e.\Delta C - \sum_f f.\Delta C$$

To be on the safe side, we had to assume that all messages containing chronometer values arrived with minimum delay. Therefore, ΨT_0 tends to set start times too late, delaying the execution of processes. This delay can be calculated by adding up the delay errors ΦC that belong to the

edges that we used to calculate ΨT_0. If e is an edge between a and n, the *delay correction* ΨD is calculated as follows:

$$n.\Psi D = \sum_{e} e.\Phi C$$

In the application graph, ΨD is added to the delay of the first read threads on that node, *i.e.*, to read threads fetching data from sources or from timed pipes to which remote write threads send. Therefore, in a distributed system the detection of chronometer differences has to precede the calculation of start time delays.

The considerations above do not take into account that the difference in chronometer values will not remain constant, but that chronometers may drift apart. Current technology provides highly accurate chronometers. The drift of quartz chronometers lies in the range of 10^{-6}, *i.e.*, a clock will drift less than 100 ms/day [31]. Considering that typical CM application programs only last for a few hours, we can take the largest possible chronometer drift within this time into account and add it to ΨT_0 and ΨD.

5. CONCLUSION

We have introduced a model to take time parameters into account when processing and displaying CM data. Unlike proposals such as [32] or [33] currently discussed by standardization committees, we have not concentrated on finding a particular representation for CM applications or documents. Instead, we have presented a useful set of general programming abstractions and shown how they can be implemented. Moulded into an appropriate notation, our model provides the programmer with a high-level, easy-to-use interface that hides the more complicated aspects of dealing with timed data.

We have not considered in this paper how dynamic changes to an application can be handled. In CM systems, such changes are very likely, *e.g.*, it will be common for a user to suddenly decide to skip a portion of the video he is watching. Modifications like this can easily be accommodated by our model: they do not affect the rates or delays of process executions. Also, the modification itself is not time-critical: at one point in time, a clock parameter will change and from then on the video will be displayed at a higher speed. Changes in the structure of the application graph, however, will require some re-evaluation, *e.g.*, of start time delays. Our future work will concentrate on how these structural changes can be handled.

Acknowledgements

George Homsy was instrumental in refining the formal model. Ralf Steinmetz of the IBM European Networking Center in Heidelberg provided inspiring comments on a very early version of this paper. Many thanks also to David Anderson and Domenico Ferrari, not only for comments on this paper, but also for creating the project environment from which these ideas could emerge.

References

1. E. A. Fox, "The Coming Revolution in Interactive Digital Video", *Comm. of the ACM 32*, 7 (July 1989), 794-801.
2. T. V. Russotto, "The Integration of Voice and Data Communication", *IEEE Network 1*, 4 (Oct. 1987), 21-29.
3. K. A. Frenkel, "The Next Generation of Interactive Technologies", *Comm. of the ACM 32*, 7 (July 1989), 872-881.
4. D. P. Anderson, R. Govindan, G. Homsy and R. Wahbe, "Integrated Digital Continuous Media: a Framework Based on Mach, X11, and TCP/IP", Technical Report No. UCB/CSD 90/566, Mar. 1990.
5. J. H. Irven, M. E. Nilson, T. H. Judd, J. F. Patterson and Y. Shibata, "Multi-Media Information Services: A Laboratory Study", *IEEE Communications Magazine 26*, 6 (June 1988), 27-44.
6. S. Sarin and I. Greif, "Computer-Based Real-Time Conferencing Systems", *IEEE Computer 18*, 10 (Oct. 1985), 33-45.
7. "Ada Programming Language", Military Standard American National Standards Institute/MIL-STD-1815A, United States of America, Department of Defense, Washington, 1983.
8. "Programmiersprache PEARL", DIN 66253, Deutsches Institut fuer Normung, Beuth-Verlag, Berlin, Koeln, 1980.
9. I. Lee, S. Davidson and V. Wolfe, "Motivating Time as a First Class Entity", MS-CIS-87-54, University of Pennsylvania, Philadelphia, Aug. 1987.
10. H. Kopetz and W. Merker, "The Architecture of MARS", *Fault Tolerant Computing Systems 15*, June 1985, 274-279.
11. B. Schueler, "Update Reconsidered", *Architecture and Models in Data Base Management Systems*, Amsterdam, 1977.
12. R. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases", *ACM-SIGMOD International Conference on the Management of Data 14*, 4 (Dec. 1985), 236-246.
13. W. B. Rubenstein, "Data Management of Musical Information", Ph.D. Thesis, UCB, June 1987.
14. D. P. Anderson and R. Kuivila, "A System for Computer Music Performance", *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 56-82.
15. J. S. Ostroff, *Temporal Logic for Real-Time Systems*, Research Studies Press (John Wiley and Sons), 1989.
16. A. Pnueli, "The Temporal Logic of Programs", *18th Annual Symposium on Foundations of Computer Science*, Providence, RI, 1977, 46-57.
17. J. F. Allen, "Maintaining Knowledge about Temporal Intervals", *Comm. of the ACM 26*, 11 (Nov. 1983), 823-843.
18. J. Dorn, *Wissensbasierte Echtzeitplanung*, Vieweg, Braunschweig, Wiesbaden, 1989.
19. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.

20. M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems* 2, 3 (Aug. 1984), 181-197.

21. *Multi-Media File System Overview*, Sun Microsystems, Aug. 1989.

22. L. Ludwig, "A Threaded/Flow Approach to Reconfigurable Distributed Systems and Service Primitives Architectures", *Proc. of ACM SIGCOMM 87*, Stowe, Vermont, Aug. 1987, 306-316.

23. *Postscript Language Reference Manual*, Adobe Systems, Addison-Wesley, Reading, Massachusetts, 1985.

24. D. Haban and K. Shin, "Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times", *IEEE Real-Time Systems Symposium*, Dec. 1989, 172-181.

25. D. P. Anderson, R. G. Herrtwich and C. Schaefer, "SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet", Tech. Rep.-90-006, International Computer Science Institute, Feb. 1990.

26. D. P. Anderson and R. G. Herrtwich, "Resource Management for Digital Audio and Video", *IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, May 1990. (to appear).

27. D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan and M. Andrews, "Support for Continuous Media in the DASH System", *ICDCS10*, Paris, May 1990.

28. A. Tevanian, R. Rashid, D. Golub, D. L. Black, E. Cooper and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 185-197.

29. R. Sedgewick, "Algorithms", *Second Edition*, Reading, Massachusetts, 1988.

30. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Comm. of the ACM 21*, 6 (July 1978), 558-565.

31. H. Kopetz and W. Ochsenreither, "Clock Synchronization in Distributed Real-Time Systems,", *IEEE Trans. on Computers 36*, 8 (Aug. 1987), 933-940.

32. "Time Synchronization", IEC JTC1/SC 18/WG 3 N1443 (Working Paper), ISO, Oct. 1989.

33. "Representation and Protocols for the Exchange of Audiovisual Interactive Applications", IEC JTC1/SC 18/WG 3 N1428 (Working Paper), ISO, Sep. 1989.