

object server, it will reply in the affirmative indicating when the object server should next check in. In addition, the directory server may also schedule an update event to synchronize its database with that of the object server should the version number of the object server included in the original request for directory service not match the one inside the database of the directory server for that object server. Should an object server fail to re-request directory service by the time indicated in the reply, then the directory server will mark the object server's objects as "temporarily unavailable."

When a directory server requests an update from an object server, it stores the result in a meta-data database and, at the very minimum, stores the NUID for each object in a database keyed on the name of the object. Thus resolving a name into a list of object NUIDs becomes very simple indeed. In addition to the NUID, additional meta-data may be stored with each entry for the object. For example, the object type (e.g., image/mjpg, audio/raw, application/playlist) is currently appended to the NUID vector to speed up pruning unsuitable objects on the basis of type without having to contact the object server for the object.

Clients wishing to access the Media Bank initially do so through the object meta-data interface module. This module takes the client's request in the form of an RPC (remote procedure call) and then extracts and queries the meta-data database on the client's behalf. Among other things, the interface removes the NUIDs of objects on object servers that failed to check in with the directory server. A typical exchange between a client and the directory server is shown in Figure 8.

Notice that the directory server responds to the client's request for information regarding objects that match the name *t2-movie-browser* with an object with type set to *item multipart locator* and the data set to a vector of length two, where each entry corresponds to an object stored on another server. The two entries in the data vector are in no specific order and must be further examined before the client can determine which one is the more appropriate. The mechanisms involved in this process will be discussed in the section detailing the operation of Media Bank clients.

Object servers

Object servers do exactly what their name suggests, deliver objects. However, in keeping with the design philosophies of pull-based delivery and segregating

Figure 8 Typical client/directory server interaction

```
Client to directory server
(item-fetch "t2-movie-browser")

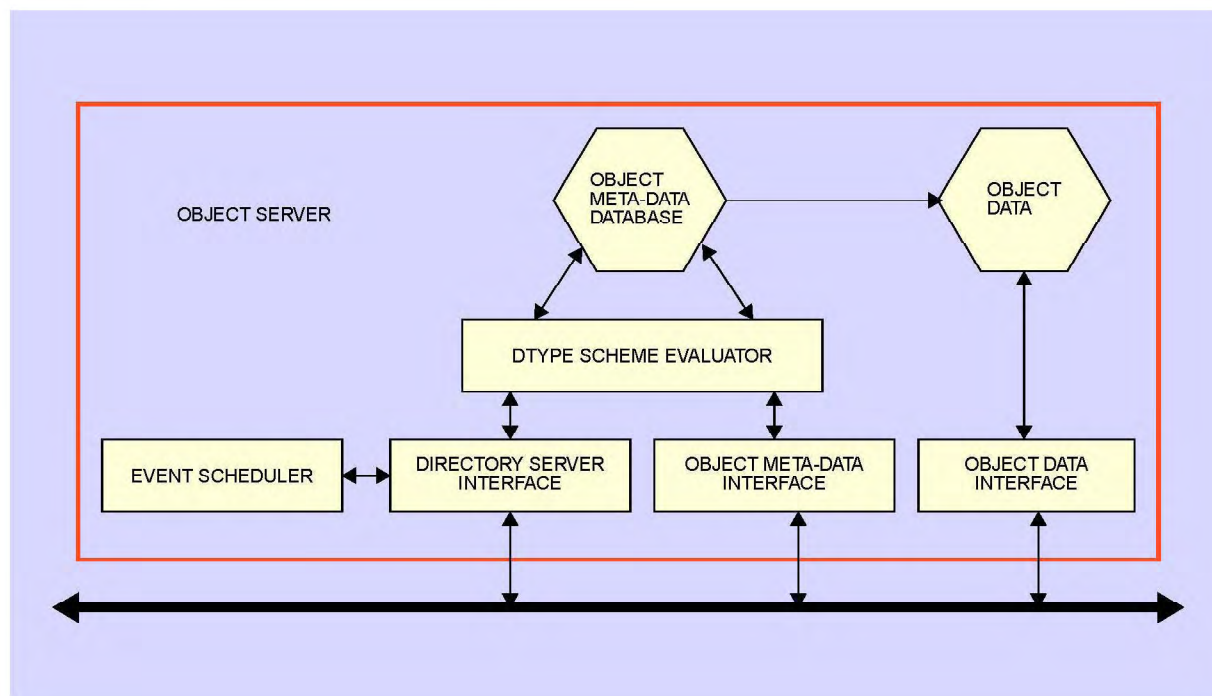
Directory server response
#((type item multipart locator)
 (data . #
  #((name . "t2-movie-browser")
   (uniqueid . "<t2-movie-browser>")
   (type item application sprog)
   (host . "alphaville")
   (port . 41501)
   (load-average . 1.36155525677))
  #((name . "t2-movie-browser")
   (uniqueid . "<t2-movie-browser>")
   (type item application sprog)
   (host . "bad-taste")
   (port . 41501)
   (load-average . 3.26552888373))))))
```

an object's meta-data from its data, the fashion in which they do so is a little different from normal. A block diagram showing the internal pathways of a Media Bank object server is shown in Figure 9.

Like the directory server, the object server's core consists of a Scheme evaluator built on top of the Dtype and Dsys libraries. There is also an event scheduler that is used to schedule requests for directory service to the various directory servers that provide directory service for the object server. The directory server interface is somewhat different from the remote server interface found in the directory server. For one, it initiates object server/directory server communications. The directory server will not contact the object servers unless the directory server initiates the first contact. This is best shown by describing the communication exchanges that result when a new object server is brought on line.

Upon startup, the object server first attempts to find a directory server from a local list of possible directory servers. As soon as the object server locates a directory server that is active, it issues a joint request for its objects to be served and also for the most recent list of directory servers. Assuming that the response is in the affirmative, the directory server returns the time for the next subsequent check-in along with a list of other

Figure 9 Object server functional block diagram



directory servers that this object server should contact. At this stage the object server schedules the check-in event and replaces the original list of directory servers with the new one. The object server then repeats the process contacting each directory server in the new list, this time requesting directory service only.

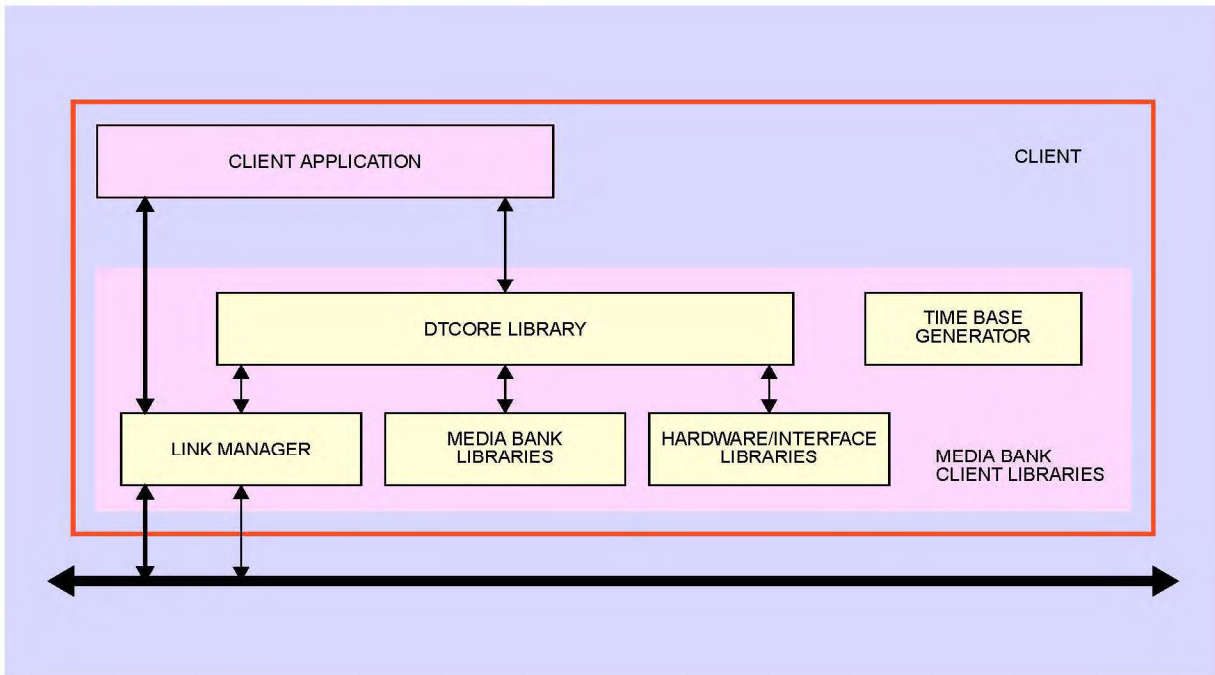
When the directory server receives and subsequently accedes to the request for directory service, it schedules an update event to occur shortly after sending the reply back to the object server. When the event scheduler determines that it is time for the update to occur, the directory server contacts the object server and requests an update based on the difference between the version number of the update request and the current version number of the object server. The response of the object server is generated using the version number difference and results in a list that selectively adds and deletes objects from the meta-data database of the directory server.

The meta-data interface of the object server is identical to that of the directory server, because the commu-

nications protocol for meta-data access is symmetric. This is important as it means that any media bank server or client can access any server and use the same suite of RPCs to extract meta-data information regarding objects known to that server.

Perhaps the biggest difference between directory servers and object servers is the ability of the object server to serve the actual object data via the object data interface. Two different protocols are currently supported at the object data interface. The first is the networked file system (NFS), but the amount of control possible over the latency and loading of the network is somewhat restricted. In fact the grade of service possible in accessing the object data is inferred from the state of the meta-data interface. The second protocol is a custom protocol developed at the Media Laboratory called the Media Bank data protocol (MBDP). MBDP transmits data from the object server by packetizing within Dtype wrappers and transmitting them using a slightly modified TCP/IP in which the timers are tuned to expire considerably quicker than normal. This design achieves two things, the first being a greater control over the use of network bandwidth by clients,

Figure 10 Media Bank client functional block diagram



and second the ability for the Media Bank to provide remote access beyond the Media Lab's local network. Ongoing work is being undertaken to create the next revision of MBDP that will shift from TCP/IP to multicast UDP (user datagram protocol) as the primary means for delivering object data. This new version of MBDP will be completely distributed and will blur the distinction between clients and object servers by allowing the clients to cache object data and subsequently apply the data to their peers as requested. This merging of functionality should also enhance scalability and performance, and may lead to a single unified architecture that could be used to deliver both real-time communications and prerecorded entertainment on a global scale.

The directory and object servers for the Media Bank may not find wide deployment around the Internet in the near term; however, we believe that their architecture is representative of what servers will look like once gigabit per second networking, connectivity, and capabilities become commonplace. In such an environment, the need to store media on large centralized servers directly connected to the primary Internet backbone diminishes significantly. Objects will be

spread and duplicated over the entire net. Furthermore, the ability to publish multimedia and have it seen will continue to diffuse further from publishing houses and into the home. However, changes to servers are only half the story as the clients will change dramatically as well.

Clients

One of the key differences between Media Bank clients and practically every other current Web client or network video client is the endowment of sufficient intelligence to perform *synchronized assembly of multiple isochronous objects* via pull-based delivery. The decision to shift to pull-based delivery stems from the realization that a client that buffers object data from multiple servers is in a better position to manage its buffers and synchronize playback from them than any of the servers would be. An example of this kind of presentation is combining several dependent video and audio streams from different locations during a sports broadcast. The ramifications of shifting these responsibilities from the server to the client are far-reaching. Basically, the client now becomes an extremely powerful entity that can operate auto-

mously from the network. No longer is the client tied to a single server, but the client is free to switch between servers whenever desired. Server dropouts and broken networks can both be detected and handled by the client as appropriate. In addition, the ability to access and pull data from multiple servers gives clients the ability to create a whole suite of heretofore unforeseen presentations that exploit both the wealth of information available on the network and the user's own preferences and tastes.

Accordingly, Media Bank clients are given the ability to execute smart objects that define procedures and applets in a similar fashion to those found in the Tcl/Tk (tool command language/toolkit) clients of the Berkeley VoD (video-on-demand) project^{2,3} and the Java*-capable clients of Sun Microsystems and Netscape Communications Corporation. The Media Bank clients execute objects that are written in Scheme, the same language used to represent the object meta-data. Correspondingly, Media Bank clients tend to be very similar internally to the object and directory servers they access. A diagram showing the internal structure of a Media Bank client is shown in Figure 10.

There are five main modules in the Media Bank client library suite. As with the Media Bank servers the Dtype and Dsys) libraries and Scheme evaluator lie at the center. However, in addition to the Dtype libraries there are several other libraries that augment the functionality of the Scheme evaluator.

The second module is the link manager, which manages the client's interactions with Media Bank servers. Like the object server, the link manager actively seeks out a directory server that will provide access to the Media Bank and cache the list containing any alternate directory servers that may be present. In addition, the link manager also keeps track of which servers are not available and provides automatic error recovery during server dropout.

The third module, the Media Bank libraries, consists of a suite of dynamically loadable shared libraries that extend the functionality of the Scheme evaluator by providing procedures to manipulate Media Bank objects. Negotiating access to objects and servers, parsing playlists, object type checking, name resolution, sorting objects on preferences, and executing smart objects are all handled by the Media Bank libraries. Written as a mixture of Scheme-only functions and Scheme hooks into optimized C++ code,

these libraries are dynamically loadable, and as such can be replaced by more recent versions or upgrades without having to recompile the client application.

The hardware libraries make up the fourth module. Basically, these libraries provide access via Scheme to any specialized hardware on the local machine. Examples of hardware- and architecture-specific code supported are access routines to MJPEG cards, access routine to the native audio server, and X library support.

The fifth and final module is the time base generator. Derived from the audio server clock, the time base generator generates periodic events signals that are then sent to any objects that have been instantiated and are in use. For example, a movie player may be fed 24 update events per second that cause frames to be displayed and audio to be played in strict synchronization.

The beauty of performing synchronization in this manner is that implementation of fast forward, fast reverse, and slow motion becomes trivial. All one needs to do is to change the rate at which the update events are generated. If one wants to play the movie at twice the normal speed, then the time base generator needs only to issue updates at 48 updates per second or at 24 updates per second, with the difference in time code between each update being doubled to one-twelfth of a second. Likewise half-speed playback will result if the time base generator issues only 12 updates per second. Because the client's display is completely decoupled from both the playback rate and the recorded rate, this does not necessarily mean that the required network bandwidth is tied to the playback rate. If a client is only capable of decoding and displaying 24 frames per second, then every second update frame event will be skipped for the double-speed playback at the 48 updates per second example just mentioned. A more powerful client may set an upper bound on the number of updates per second that is willing to process in deference to other users.

Sitting above all these modules is the client application. Usually, the client application will be written in one of two ways. The first is as a precompiled C++ program that uses Dtype hooks to execute the necessary Scheme code to access the five modules just described. The second option is to write the application as a Scheme program that can then be suitably wrapped up to become a Media Bank object. There

are advantages and disadvantages to both of these methods. The C++ option will result in faster code, particularly for clients that perform any sort of memory or compute-intensive manipulation of video or audio data, while the Scheme route offers the benefit of portability and accessibility at the expense of speed. However, there is a provision for a way around this shortcoming; one can extract the compute-intensive Scheme code, rewrite it in optimized C++, and put it in a Scheme wrapper inside a dynamically loadable library that can then be sent to the client as an object as needed.

In order to best demonstrate the operation of Media Bank clients, we have included the following four examples that detail some typical client actions, and we also show screen shots from several Media Bank clients.

The first example shows the steps taken by a client that wishes to access the movie *Terminator 2* (see Figure 11).

First the application issues a request to the Media Bank client libraries requesting the location of objects with a name that matches *Terminator 2*. Other information may be present in the form of preferences that will be used to rank any matching objects on the basis of a series of tests between any of the fields contained in the objects. Upon reception of the request, the Media Bank client libraries will contact the first available directory server and request a list of objects that match the name *Terminator 2*. The directory server will return a list of objects that match (Figure 11, top).

Upon receiving this list, the Media Bank client then sorts the objects in the reply according to the users' preferences and also a number of system preferences (e.g., the preferred video format is MJPEG, or motion JPEG, because the client has an MJPEG card). This process is performed in four stages. In the first stage, *pre-filtering*, any objects that mismatch on type are pruned from the list of possible matches (e.g., all still images will be discarded here). Assuming that there is still more than one object in the list, it is *sorted* on the basis of the information currently known about each object and the combined user and system preferences. At this stage one object may satisfy the preferences sufficiently well that no other object need be considered. In this case the meta-data for the object are fetched from the appropriate object server and then remaining objects are placed into a field tagged "backup-info" that is then appended onto the meta-

data for the object. Should insufficient information be available to make a decision, the Media Bank client libraries will enter an iterative phase of *resolving* unresolved objects: fetching the meta-data about them from the object server that serves them, re-evaluating the preferences for the now resolved object, and *resorting* by inserting that object into the list (Figure 11, middle).

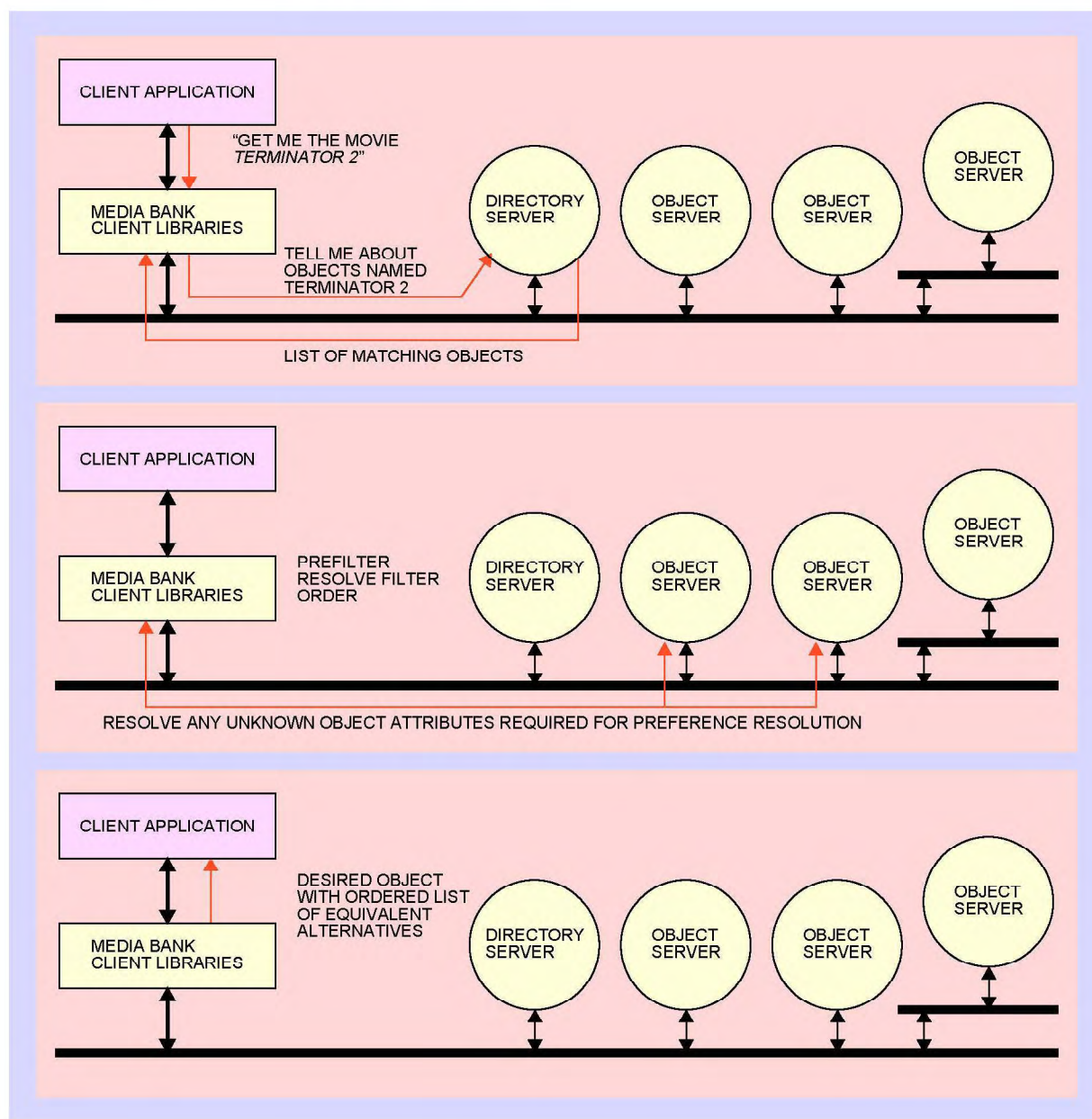
Finally, having resolved the matching objects, fetching the meta-data about them, and sorting them according to preferences, the Media Bank client libraries pass the best object (with the others in the backup-info field) to the client application (Figure 11, bottom).

The second example shows how a Media Bank client retrieves and synchronizes data from multiple isochronous sources once the objects have been located (see Figure 12). Perhaps the most trivial case that illustrates this capability is the playback of a movie. However, before delving into the exact mechanism for playback within the client it is worthwhile to examine the object format that makes synchronization possible.

Movies in the Media Bank are generally constructed from a number of objects that correspond to separately stored snippets of video and audio. Obviously the act of de-interleaving the audio and video requires that measures are taken to maintain the temporal dependencies between corresponding audio and video objects. Within the Media Bank this is achieved through the inclusion of offset, duration, number of samples or frames, and playback rate information in every audio or video object. Because some of this information is redundant, it is not required to include a value for each of these parameters with each object, as long as the missing parameters can be derived from the ones that are included. In addition, the Media Bank also allows the derivation of objects that are subranges of other larger objects. This design is particularly useful for synchronizing audio and video that have been digitized independently. An example of an audio object that is subranged from a large one is shown in Figure 13.

Having defined the storage format for individual objects, one needs to define a mechanism for synchronizing multiple streams. The Media Bank achieves the goal through the use of playlist objects that define the temporal relationships between multiple pieces of data at the object level. Two examples of a playlist are

Figure 11 Negotiating access to the movie *Terminator 2*



given in Figures 14 and 15. Figure 14 shows how three audio objects can be concatenated into a single larger one by issuing consecutive play commands, while Figure 15 shows how two separate temporal objects can be played simultaneously through the use of the play-multiple command.

As was briefly explained earlier in this section, the client coordinates the simultaneous playback of multiple objects via a local time base generator. The time base generator exploits the fact that objects "know" about their own temporal properties by sending updates that specify an offset in terms of time to a handler that is

Figure 12 Fetching the meta-data about an object from the directory server

```
Client request to directory server
(object-fetch "Terminator 2")

Director server reply to client
#((type item multipart locator)
 (data .#(
  #((name . "Terminator 2")
   (uniqueid . "<t2-movie-browser>")
   (type item application sprog)
   (host . "alphaville")
   (port . 41501))
  #((name . "Terminator 2")
   (uniqueid . "<t2-movie-browser>")
   (type item application sprog)
   (host . "bad-taste")
   (port . 41501))))))
```

Figure 13 An object derived by subranging from a larger one

```
#((uniqueid . "<t2-audio-ch1-ch28-adjusted>")
 (name . "t2-audio-ch1-ch28-adjusted")
 (type . (item application subrange))
 (object . "<t2-audio-ch1-ch28-raw>")
 (offset . 60.8)
 (duration . 81038000/23987))
```

instantiated for each object. The handler then performs the necessary actions to display the portion of the object at the specified time. Compound objects result in a hierarchy of handlers where the top or parent handler passes down the update event to the relevant children. A new update event is not generated until a handler has finished processing the previous update event. In this way, each object is informed in unison as to where it should be in the playback process. Obviously, the more objects one has, the greater load one places on the system, which can lead to the total handling time for all the handlers exceeding the time between update events. In these cases audio objects are given preference over video objects for resources, because the human ear is much more sensitive to temporal discontinuities than the human eye.¹⁸

Figure 14 Example of using the play command to concatenate multiple objects

```
#((uniqueid . "<t2-audio>")
 (name . "t2-audio")
 (type . (item application playlist))
 (version . 0)
 (script .#(
  (play #((object . "t2-audio-ch1-ch28-adjusted"))))
  (play #((object . "t2-audio-ch29-ch57-adjusted"))))
  (play #((object . "t2-audio-ch58-ch78-adjusted"))))))))
```

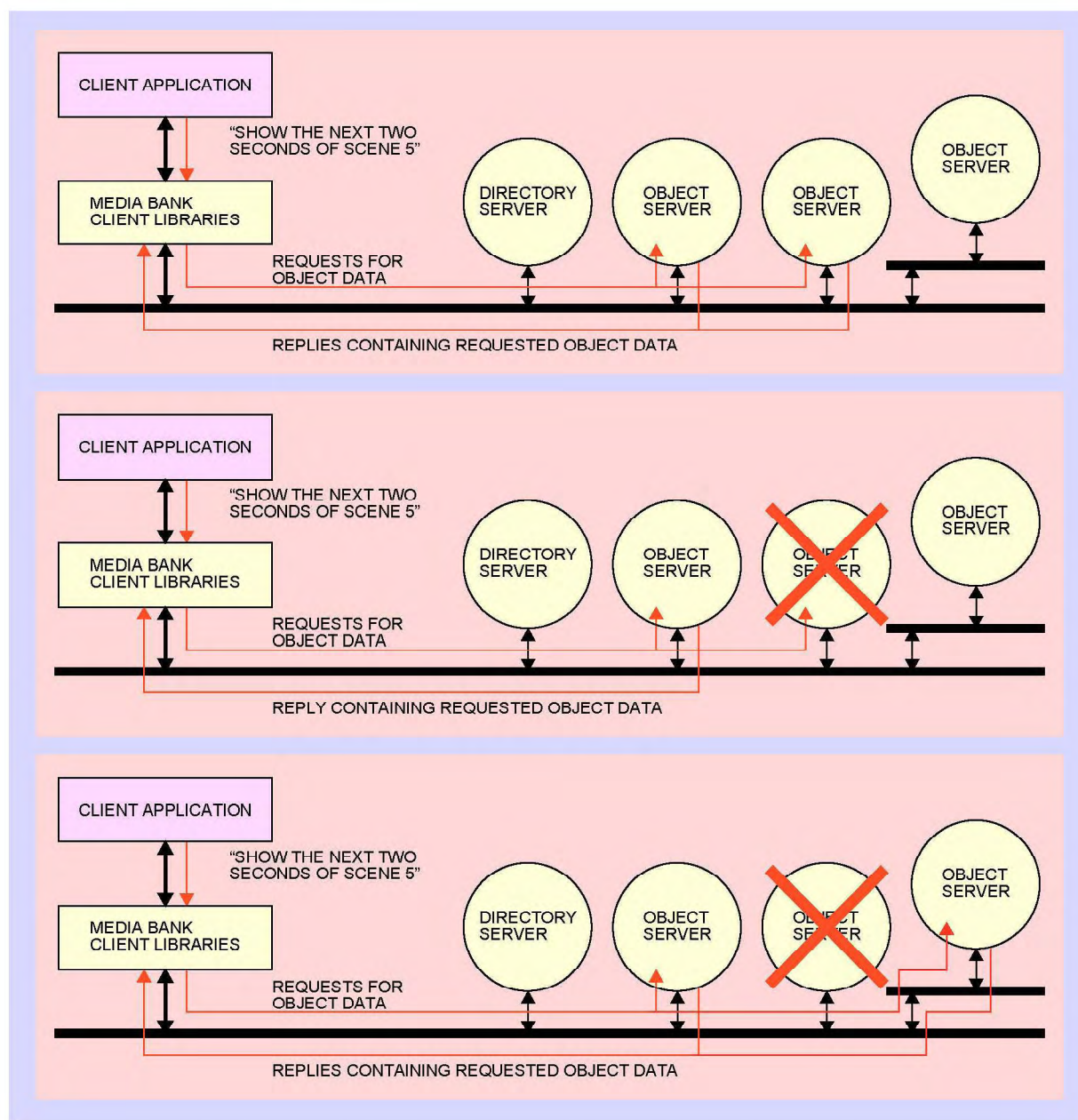
Figure 15 Example of using the play-multiple command to synchronize multiple objects

```
#((uniqueid . "<t2-movie>")
 (name . "t2-movie")
 (type . (item application playlist))
 (script .
  #((play-multiple
    #(
      #((play #((object . "t2-video"))))
      #((play #((object . "t2-audio")))))))
  )))
```

An interesting side effect that arises out of using this particular mechanism for playback is the decoupling of the playback rate from the recording rate. Consider a video object recorded at 24 frames per second and sent updates as though it was recorded at 30 frames per second. In this case the handler would choose the frame nearest in time to the desired sample for playback, thereby performing automatic (albeit less than optimal) frame rate conversion.

The third example demonstrating the client's capabilities is the ability to perform *autonomous error recovery* (see Figure 16). This is an especially useful feature, because it ensures that the quality of service given to a client is not bound to a single central server, as is the case with most solutions today. Thus the

Figure 16 Autonomous error recovery



Media Bank provides a much more robust means for delivery whereby the quality of service is much more closely tied to the client under the user's control, thereby minimizing the possibility of having a perfectly healthy client that is incapable of accessing data due to the failure of a single node in the network.

Having followed the first example that showed how a client negotiates access to the Media Bank, it should be no surprise that the means for error recovery lies in the *backup-info* field of the returned object. This field contains a list of alternate equivalent objects that may be used in lieu of the parent object, should access to