Introduction

Interactivity is very prominently placed in current MPEG-4 activity. Even though a back channel is specified to support interactivity, the syntax and semantics for such interaction messages is not specified. Existing standards such as DSM-CC and RTSP support traditional VCR-type interactivity to reposition a media stream during playback. These existing specifications are not adequate, as MPEG-4 applications potentially require complex interactive control beyond the VCR-type control offered by these models. The concept of generic, CGI-like application-specific interactive messages was proposed in our earlier contribution (MPEG97/M2888) to the 41st MPEG meeting. In this contribution, we detail the syntax and semantics for the control messages for user interaction exchanged between clients and servers (the terms clients and servers are functional notations and can represent clients and servers depending on their functionality and connectivity).

Interactive messages are caused by certain user actions and sent to the server, which cause it to modify the object stream it is delivering by adding an additional object, removing an object etc. The user actions may include clicking an object, inputting a text string, etc.

Interactivity provided is very application specific and MPEG cannot possibly define the interactive behavior completely in terms of user events. To support complete application-dependent interactivity, CGI-like approach to interactivity should be adopted. User input such as clicks and data entries causes an application-specific command sent to the server (if this type of event is specified). The server can then respond typically by sending a BIFS update command (that changes the view, adds a new object etc.). This allows total freedom for supporting complete interactivity as required by the applications.

2 Interactivity in MPEG-4

MPEG-4 essentially uses two modes of interactivity: local, and remote. Local interactivity can be fully implemented using the native event architecture of BIFS, as well as the BIFS update mechanism for events generated by a hosting application. No additional normative support is necessary.
Remote interactivity currently consists of URLs. There are two types of URLs defined in the MPEG-4 Systems CD. The first is BIFS URLs, where a BIFS field, instead of referencing an object descriptor it actually contains a URL string. The second type is object descriptor URLs. These are actually placed in the object descriptor (ES configuration descriptor). The difference between the two is that the former refer to raw data, while the latter refer to AL-formatted data.

While a simple form of remote interactivity can be fully supported by using BIFS URLs (that invoke CGI scripts at the server), this implies that the only means of interactivity in an MPEG-4 session can occur through an IP channel, and in particular using HTTP access methods (POST/GET). For MPEG-4 sessions that want to exchange messages using FlexMux connections (non-IP connections in general), this approach cannot be used (the HTTP server will not know how to parse FlexMux PDUs).

More importantly, URLs are currently used for data retrieval and are not integrated with the event facilities. For example, the trigger of a proximity sensor cannot be sent back to the server.

It is then appropriate to define a very simple mechanism to communicate information back to the server using the existing FlexMux channels set up using DMIF. Such facility has to be integrated with the native event routing mechanism of BIFS.

The generic format for these messages can indeed be extremely simple, as the syntax and semantics of control messages are entirely left to application designers. Also, one can assume that a fully reliable connection is present. DMIF can ensure that the proper QoS attribution is associated with the upstream control channel so that this assumption holds true. Even though one could potentially associate different QoS categories to control messages, we believe that in practice such an approach would have little use and would unnecessarily complicate the syntactic structures.

In the following we detail the syntax for the command descriptor as well as the message being sent to the server (AL encapsulated). In order to use such a command descriptor, a modification is also proposed in the ROUTE definition to allow event targets to point, in addition to BIFS nodes, to command descriptors as well. Pointing to command descriptors by reference allows the reuse of commands at different ROUTEs and times in an MPEG-4 presentation. To accommodate command descriptors, descriptor tag 0x05 from the unused pool 0x05-0xFF (see Clause 7.3.2.3, Table 7-3 of MPEG-4 CD) can be used.

3 Command Descriptor

Continuing in the MPEG-4 Systems methodology of using descriptors to support key functionalities, we define a command descriptor to communicate control messages. Command descriptors define a format to communicate control messages between two entities (clients and/or servers) participating in an MPEG-4 session. A command descriptor specifies a format to communicate control messages. It does not define any specific control messages (e.g., it does not specify a command format to advance a particular object in time). Such specifics are application-dependent and are left to the application designers.
### 3.1 Syntax

```python
class CommandDescriptor: bit(8) commandDescriptorTag = 0x05 {
    bit(10) CommandDescriptorID;

    // stream count; number of ES_IDs associated to this message
    unsigned int (8) count;

    // ES_Id of the streams
    unsigned int (16) ES_ID[count];

    // application-defined parameters
    do {
        unsigned int (8) paramLength;
        char (8) commandParam [paramLength];
    }
    while (paramLength!=0);
}
```

### 3.2 Semantics

- **CommandDescriptorID**: 10-bit value uniquely identifying this command descriptor.
- **count**: number of objects that are affected by this control message.
- **ES_ID**: ES_ID of the objects affected by this message. This information can be useful, for example, for simple implementation of commands such as stop, pause etc. Since the content creator inserts these values, however, their meaning can be arbitrary.
- **paramLength**: length of the command parameter.
- **commandParam**: application-specific parameter string.

### 4 Modification Required in ROUTE

The definition of ROUTE (Clause 7.2.3.1.12.3.1) needs to be updated as follows, in order to support command descriptors.

```python
class ROUTE {
    bit(1) isUpdateable;
    if (isUpdateable)
        bit(10) ROUTEid;
    }
    bit(10) outNodeID;
    outID outFieldReference;
    bit(1) isServerRoute;
    if (isServerRoute) {
        bit(10) CDid; // command descriptor ID
    } else {
        bit(10) inNodeID;
    }
```
The modification consists of the introduction of a single-bit flag (isServerRoute), which signals
the presence of a command descriptor ID, rather than the inNodeID.

Note that nothing prohibits an event to be routed both to a local node, as well as to the server.

5 Server Command

This documents the syntax and semantics of the command, as it is transmitted back to the server. Transmission is performed using AL packetization. This allows timestamping to be used (if the AL is configured as such using DMIF facilities when setting up the back channel(s)).

5.1 Syntax

class Command {
    // which node generated this message
    unsigned int (16) nodeID;

    // the following are simply copied from the command descriptor
    // stream count; number of ES_IDs associated to this message
    unsigned int (8) count;

    // ES_Id of the streams
    unsigned int (16) ES_ID[count];

    // application-defined parameters
    do {
        unsigned int (8) paramLength;
        char (8) commandParam[paramLength];
    } while (paramLength!=0);
}

5.2 Semantics
	nodeID: The nodeID of the node associated with the command. This field must be filled by the MPEG-4 terminal based on the node that directly pointed to the command descriptor.

All other fields are simply copied from the command descriptor.

6 Comments

This particular structure does not communicate the field value for the event at hand. Such functionality can be easily added, by including the inFieldReference as well as the value for the field that generated the event.