Dynamic Distributed Multimedia: Seamless Sharing and Reconfiguration of Multimedia Flow Graphs

Marco Lohse Michael Repplinger Philipp Slusallek

Computer Graphics Lab, Saarland University, Germany

Abstract

Mobile devices with multimedia and networking capabilities are quickly becoming ubiquitous through the availability of small notebooks, PDA, and in particular, mobile phones. However, most multimedia systems are still centralized, concentrating all multimedia computing on a single device. At best they deploy a strict client/server architecture mostly for streaming multimedia from some server while processing it locally.

Ubiquitous computing environments on the other hand require systems that allow multimedia content and processing to be flexibly distributed across different devices that are currently available in a network while supporting dynamic migration between devices as users move or requirements change.

This paper concentrates on the two most important operations for dynamic distributed multimedia: combining two flow graphs so that common processing can be shared between them as much as possible, and – based on this service – reconfiguring a running multimedia flow graph across a network. We use a number of different application scenarios to motivate the basic middleware requirements including device control and distributed synchronization. Finally, we present two key applications and report on results of an implementation of this approach.

1 Introduction

Today, there is a strong demand for mobile and ubiquitous computing in the area of multimedia entertainment. Mobile devices like Personal Digital Assistants (PDAs), portable web pads, or even cellular phones already provide decent multimedia and from reasonable to very good networking capabilities even in mobile scenarios.

While we are increasingly surrounded by many of these devices, multimedia processing in most systems is still concentrated in a single device. For example, we cannot seamlessly play the music stored on an MP3-player, a phone, or a PDA on our car stereo when getting into the car or switch to the living room speakers when coming home. Nor can we use the mobile devices to control the volume or any other aspect of multimedia processing.

This is even more surprising given that multimedia systems generally implement a very flexible processing approach, where many small processing elements or nodes are interconnected into a multimedia flow graph such as in Microsoft's DirectShow [Microsoft 2003] or in the Java Media Framework [Sun 2003]. Each of these nodes provides fine grained access to specific multimedia devices (e.g. TV receiver, DVD drives, or audio output) or processing capabilities (e.g. MPEG compression or demultiplexing). In such a system, distributed multimedia is conceptually simple to implement by routing the data to and from processing or device nodes located on remote machines. Several such systems have been built in the research community as will be discussed below.

While the details of such distributed multimedia systems are non-trivial, the main challenge is the dynamic (re-)configuration of active flow graphs. In this paper we describe the functionality of sharing parts of active flow graphs from within different applications running on different computers (Section 5). We then extend this service to realize the dynamic reconfiguration of flow graphs. The flow graph of running applications can be reconfigured and migrated during runtime to remote systems (Section 6). During this reconfiguration, media playback stays continuous and fully synchronized.

To motivate the requirements of such services, we discuss some application scenarios and related work in Section 2. Section 3 describes the underlying middleware while Section 4 gives an overview of the synchronization architecture, which forms the basis for synchronized playback across distributed devices. Finally, we conclude this paper and describe future work in Section 7.

2 Application Scenarios and Related Work

To motivate the requirements for the presented work we introduce two application scenarios. The first scenario is a media playback application started on some computer, e.g. watching a movie from a DVD drive. Another user could then join watching the movie but might want to listen to a different language track using his PDA to receive and play the audio via earphones.

To realize this scenario, the PDA needs access to parts of the DVD application so it can connect to the running flow graph. Furthermore, in a scenario where several users share one display but listen to different audio tracks, synchronization across all connected applications must be provided to keep lip-sync. In Section 5, we introduce the concept of a *session* as an abstraction for a flow graph of already connected and active devices and refer to this scenario as *session sharing*.

Previous work, such as [Kahmann and Wolf 2002], uses a proxy architecture together with standard streaming servers and clients for collaborative media streaming. In contrast to our approach, no support is provided for synchronized playback or sharing of flow graphs. In [Roman et al. 2002], an application framework for active environments is proposed that can map running applications between different active environments depending on the users position. Furthermore, another user can connect to these running applications and receive the same data. Our approach is similar to this one but is more flexible because instead of mapping whole applications we map only the required parts of a flow graph.

Based on this service, several other applications, that require reconfiguration can be realized, such as seamless and synchronized

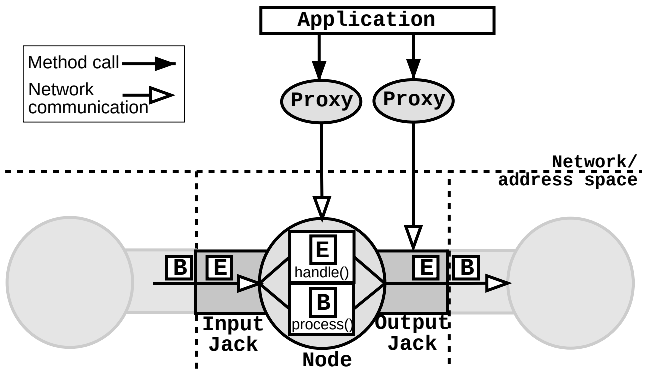

Figure 1: A node is connected to other nodes via its input and output jacks. Messages ('E' event, 'B' Buffer') are sent instream. Proxies allow distributed access to nodes.

handoff. Let us consider a playback application running on a mobile device, e.g. a PDA, playing audio data through its internal speakers or earphones. When the user moves around and enters an environment with richer I/O capabilities (e.g. a hi-fi system in a living room, a car, or an office) the output could be handed off to these more capable devices. If the user later leaves this environment the output should be handed back to the mobile device.

To realize this scenario, the application must be able to dynamically adapt an active multimedia flow graph to changing environments. In addition, the handoff should be seamless and continuous. This means that no data is lost or duplicated and media playback is fully synchronized at all times, which is especially important for the playback of music. Contrary to a common *handoff* scenario, which describes the change of an existing network connection, this scenario requires to reconfigure and migrate parts of an active flow graph. Therefore we use the term *handover* to describe it.

The main disadvantage of existing approaches is the lack of synchronized playback during handover. For example, the approach described in [Lin et al. 2002], is based on moving application sessions across different devices. They use existing media players as clients and therefore cannot handover parts of a flow graph during synchronized playback. In [Wedlund and Schulzrinne 1999] the usage of standard protocols like SIP is examined but without the aspect of synchronized playback. Another approach, similar to ours, is described in [Carlson and Schrader 2002]. They are using two simultaneous processing chains to enable a zero-loss behavior. However, they cannot provide synchronized playback during handover, either. Our approach presented in Section 6, allows the reconfiguration of an active flow graph without loss of any data. In addition, the playback remains synchronized during handover.

3 Middleware Requirements

For the following discussions we assume a multimedia middleware that is based on a distributed flow graph approach where any node can be distributed across the network while transparent access is provided by some means, e.g. using a proxy architecture (see Figure 1). Furthermore, this middleware must provide a registry service for node discovery and a distributed synchronization infrastructure (discussed in the next section).

These requirements are fulfilled by the *Network-Integrated Multimedia Middleware* (NMM) that is especially designed to access, control, and integrate distributed multimedia devices. A detailed description of the middleware can be found in [Lohse et al. 2002].

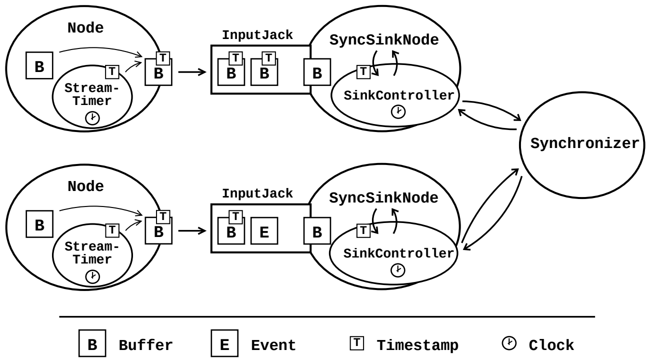

Figure 2: Time stamped buffers are handled by controllers. A synchronizer coordinates playback for different nodes.

NMM represents hardware devices or software components by *nodes* that offer input and output ports, referred to as *jacks*. Jacks specify the multimedia *formats* supported for multimedia data and are used to connect nodes into a flow graph. Since a node can have several input and output jacks they are labeled with a *jack tag*. Jacks receive or send messages, as shown in Figure 1. Furthermore output jacks can be dynamically duplicated if they are requested multiple times. They then form a *jack group*, that forwards outgoing messages to all its output jacks. The concept of a jack group is an essential requirement for the middleware services described in Section 5 and 6. The messaging system provides two different types of messages. *Buffers* are used to transport multimedia data and *events* are used for arbitrary control information.

A registry services must be provided for discovery, reservation, and instantiation of possibly remote devices. We assume that this registry service stores a complete *node description* of all available nodes, which includes the specific type of a node (e.g. "TV-CardNode") and the supported formats (e.g. "video/raw"). The registry processes application queries specified as *graph description*. A graph description includes a set of node description, connected by *edges*.

After successfully processing the request by matching available nodes to the query, the queried nodes are instantiated and returned by the registry. The registry service must also be able to setup and create distributed flow graphs by delegating instantiations to remote registries.

4 Synchronization Architecture

To realize synchronized playback of nodes distributed across the network, NMM provides a generic distributed synchronization architecture.

Since the synchronized play out of different streams (e.g. lip-synchronized audio and video) is important for every single buffer (e.g. some audio samples or a single video frame), it is especially important to minimize communication needed for synchronization in distributed environments. Therefore, NMM strictly distinguishes between *intra-stream* and *inter-stream* synchronization [Gordon Blair and Jean-Bernard Stefani 1998]. Intra-stream synchronization refers to the temporal relations between several presentation units of the same media stream (e.g. subsequent frames of a video stream), whereas inter-stream synchronization maintains the temporal relations of different streams (e.g. for lip-synchronizing of audio and video).

Figure 2 provides an overview of synchronization during playback in the NMM architecture. Synchronization at source nodes (e.g. for capturing of live data) is performed similarly. The follow-

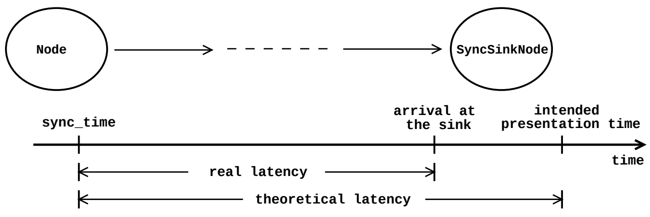

Figure 3: The real latency is computed for each stream; the desired or theoretical latency of all streams has to be identical for synchronized playback.

ing sections will then show how this architecture is used to realize different synchronization protocols, e.g. for shared session or seamless reconfiguration of flow graphs.

The basis for performing synchronization is a common source for timing information. We are using a static `Clock` object within each address space. This clock represents the system clock that is globally synchronized by the Network Time Protocol (NTP) [NTP: The Network Time Protocol 2003] and can therefore be assumed to represent the same time basis throughout the network. With our current setup, we found the time offset between different systems to be in the range of 1 to 5 ms, which is sufficient for our purposes.

Each buffer also holds a `Timestamp` that represents the time when this buffer should be presented referring to some arbitrary time base. The time base and the value of the timestamp is specific for the flow graph instantiated by an application, e.g. a timestamp might be taken from the multimedia data stream (e.g. the MPEG timestamps) or might be generated by a node. `StreamTimer` objects help setting timestamps for these different cases.

All sink nodes that allow synchronized processing of buffers are subclasses of `SyncSinkNode`. Such a node delegates intra-stream synchronization to a `SinkController` that decides if and when to present the buffer by inspecting the timestamp. As controller objects resides within the corresponding node (i.e. within the same address space), no network traffic is involved in this step.

If multiple data streams are to be presented in sync, the controller objects involved are connected to a `Synchronizer` that realizes inter-stream synchronization by implementing a specific synchronization protocol. Notice, that the different controller objects might be running on different hosts or in different address spaces as the synchronizer. In order to minimize network traffic, the controller objects locally implement the intra-stream decisions of the `Synchronizer` that are communicated only when needed.

How can a controller decide when exactly to present a buffer in comparison to another stream of buffers? The main idea is that the different controllers running within different sink nodes should present their buffers as if they had the same *latency* (called *desired* or *theoretical latency*, see Figure 3) with respect to processing in the flow graph.

During runtime, a controller computes the latency for each incoming buffer – the *real latency*. If this latency exceeds a predefined value depending on the desired latency, the buffer is too old and declared as invalid. If the real latency is smaller than the desired latency, the presentation of the buffer will be delayed.

The goal of the inter-stream synchronization is that the latencies for the different streams are equal. To achieve this, every controller sends its computed latency for the first buffers to arrive to the synchronizer. These values are taken as a first estimate of the overall latency: the synchronizer computes a new desired latency as the maximum and sets this value for all connected controllers. During runtime, all controllers also compute the average value of the latencies of incoming buffers, where the number of buffers to consider may be varied by the synchronizer or the application, a typical value would be for every 25 video buffers or 50 audio buffers. Only this value is sent to the synchronizer. The synchronizer then again computes a theoretical latency for all connected controllers and updates its controllers. Notice, that with this architecture, network traffic is reduced to a few messages per second with only a few bytes per message.

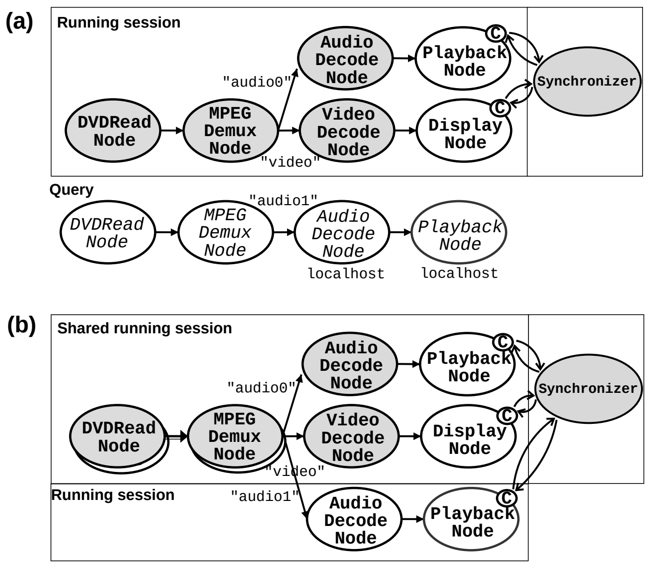

Figure 4: Session sharing. The query for a different audio track of an already running session (a) is mapped by the session sharing algorithm to use a shared sub-graph (b). The controllers (marked as 'C') of all sink nodes are connected to the synchronizer.

We use a more advanced protocol that further reduces network traffic: since interruptions or dropped buffers of audio streams are much more disturbing than for video streams, the controller of the first audio stream is chosen as master; all other controllers act as slaves that have to adjust their playback speed to the master. To realize this, the synchronizer propagates the latency of the master as theoretical latency to all slaves; only if the controller of a slave detects that its real latency has increased by a multiple of the allowed values (e.g. due to long-term changed networking conditions), it will report this to the synchronizer and the synchronizer will use this value as new desired latency – and therefore also interrupt the master stream once.

Within this architecture, we have implemented a synchronizer that can handle several audio and video sinks and allows to dynamically add and remove streams.

Notice, that this synchronization protocol also compensates for the drift between the internal clock of the sound board (that controls audio playback) and the system clock (that controls for example video playback). We have found this drift to be in the range of one second for 30 minutes of video. When using more than one audio sink, the controllers of the slave sink nodes have to adjust their playback speed, e.g. by dropping or doubling samples.

5 Automatic Session Sharing

As described in Section 2, there are a number of scenarios that need the possibility to share parts of a flow graph. Therefore, if the specific nodes for a flow graph are requested, a sharing policy can be set: nodes can be marked as *shared* for explicitly shared access, *exclusive* for explicitly exclusive access, and *exclusive then shared* to share an exclusively requested node. A combination of policies such as *exclusive or shared* is also possible. In this case a shared reservation is chosen if an exclusive request failed.

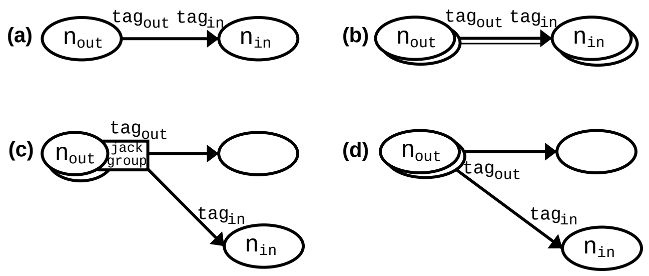

Figure 5: An edge of a query in (a), a complete overlap (b), a partial overlap using a duplicated output within a jack group in (c), a partial overlap with a different output connected in (d).

A flow graph of reserved and connected nodes is stored as a *session* within the registry service. Shared nodes of a session can then be reused within newly created flow graphs. To realize the synchronized playback for shared sessions, a session also contains a synchronizer object that can handle several audio and video tracks and that allows to dynamically add new tracks as described above.

As an example, a running session with shared nodes is shown in Figure 4(a). In this example, the application chose to share all nodes (shaded dark) required for watching a DVD, except the sink nodes for rendering audio and video. Another application that wants to access and playback a different audio stream of the DVD, being used in the running session, will use a query such as the one in Figure 4(a). Here, the mode "exclusive or shared" is set for all node descriptions, except the nodes for decoding and playback. These nodes should run on the local host and use the "audio1" output of the demultiplexer (instead of "audio0").

If a query (described as a graph description, see Section 3) is processed by the registry service, a session sharing algorithm searches the running sessions for overlapping sub-graphs that can be reused. Intuitively, an overlapping between two flow graphs only makes sense, if they share some common source of data, like the same DVD drive. Otherwise, the two different flow graphs would try to share internal nodes for different data, which does not make sense.

The algorithm tries to "grow" overlapping sub-graphs starting from such a source node: for each outgoing edge e^q from node n_{out} to n_{in} of the query graph (see Figure 5(a)) and an edge e^r of the already running graph, different types of overlaps can exist:

- A *complete overlap* exists if all constraints of e^q are fulfilled by e^r. These constraints include the node types, the sharing policies and the hosts the nodes n_{out} and n_{in} should run on. Additionally, the input and output jack tags have to match (see Section 3). In such a case, both nodes of e^r and the connecting edge can be used by e^q without the creation of any additional node.

- A *partial overlap* of e^q and e^r exists in two cases: first, if only the *outgoing* parts of the above mentioned constraints are fulfilled. An example would be an edge where n_{in} does not allow sharing. In such cases, only the node n_{out} can be reused within the query and an additional node for n_{in} has to be created. Then, the already connected output of n_{out} is copied and inserted into the jack group (see Section 3). The data flow will be forwarded to both outputs (see Figure 5(c)). Although nodes always use jack groups, we only depict them if two or more jacks are used.

  Secondly, two edges partly overlap, if the *outgoing* parts match, but e^q corresponds to a different previously unconnected output tag_{out} of node n_{out}. In this case, again, n_{in} has to be created and will get connected to the other output (see Figure 5(d)).

For a complete overlap, the algorithm recursively continues the search starting from the node n_{in}. For a partial overlap, the sub-graph connected by e^r is removed (since it can not be reached anymore) and the search is continued. Within all recursions, the search ends if no more nodes can be overlapped. Although in general, an exhaustive search is performed, the strict criteria for the different types of overlappings and the pruning of the search tree greatly reduce the number of iterations needed for typical flow graphs. More details on the algorithm can be found in [Lohse et al. 2003].

If different overlaps for a query were found, a value function is used to decide which overlap should be taken. We currently use a very simple function that prefers the overlap with the most shared nodes and edges, but more sophisticated approaches can be used as well.

5.1 Results

Figure 4(b) shows the result of the graph sharing algorithm for our example: the `DVDReadNode` and the `MPEGDemuxNode` and their connected edge are now shared for the second session (*complete overlap*), whereas an additional edge was created to connect the second audio output of the `MPEGDemuxNode` to the newly instantiated nodes `AudioDecodeNode` and `PlaybackNode` that are running on the local host (*partial overlap* to previously unconnected edge). With this setup, a different audio stream will be rendered on the device that runs the second session. On a commodity Linux-PC, the runtime of the sharing algorithm for this setup is about 59 milliseconds.

The synchronizer described in Section 4 is used to provide synchronized playback for this distributed flow graph: the audio sink of the running session is chosen as master; the video sink and the second audio sink act as slaves. Although the first few buffers for the second audio sink usually arrive too late and will be discarded, the new sub-graph catches up after a few dropped buffers.

6 Seamless and Synchronized Reconfiguration

In order to provide the highest quality output to the user or to distribute multimedia data processing, parts of an active flow graph can be seamlessly migrated to other devices during runtime. This service builds upon the session approach as described below.

Figure 6 shows one possible application scenario for such a reconfiguration: the user wants to playback media files stored on a mobile device. If no other system is nearby, the presentation of the media data is performed on the mobile system. In our case, this would be the playback of decoded MP3 files through internal speakers. The flow graph for this example consists of three nodes, all running locally in the beginning, but all accessed via proxies and interfaces as described in Section 3. The `ReadfileNode` reads the encoded file from the internal memory or a memory card, the `AudioDecodeNode` decodes the data (e.g. MPEG-audio or Ogg/Vorbis), and the `PlaybackNode` performs the audio output.

If a stationary system with richer I/O capabilities – e.g. high-quality stereo audio output – is nearby, the playback of audio data should be handed over from the mobile device to the stationary system. If the stationary system also provides the possibility to perform the decoding of the audio data, the corresponding node should also be migrated. As mentioned above, the selection of the nearby system is currently done manually. If the user moves on, the session might get handed over back from the stationary system to the mobile device or from one stationary system to another.

As there are different application scenarios where such a dynamic adaptation of an active flow graph is needed, the basic idea is always the same: We configure the new parts of the current flow

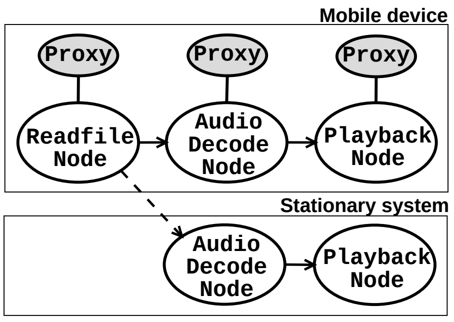

Figure 6: Application scenario showing handover from mobile to stationary system.

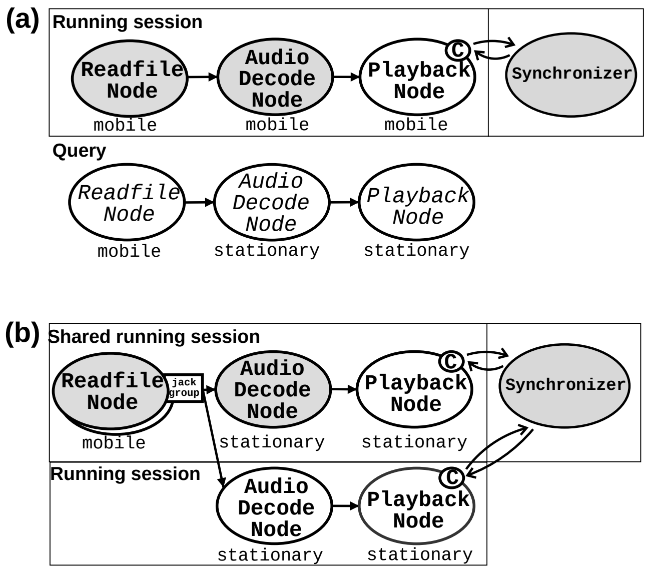

Figure 7: Automatic setup of the slave graph: the query specifies how the running session should be reconfigured (a). The slave graph as found by the session sharing service in (b): additional nodes were created on the stationary system; both controllers (marked as 'C') are connected to the synchronizer.

graph while keeping the original media processing running continuous until both are synchronous. This synchronized handover might also include the migration of nodes to remote hosts.

There are two main steps that have to be performed for a seamless handover:

- The first step is to create reconfigured second instances of the specific parts of the current flow graph (possibly on another host). The sub-graph of the current flow graph is called the *master graph*; the newly instantiated sub-graph *slave graph*.
- The second step is then to switch the data and control connections from the currently used instances to these newly created instances while keeping media processing continuous and synchronized.

Due to the fact that at some time during this procedure, the old data connection has to be torn down while the new data connection will be established, there is no guarantee that the playback will stay continuous and synchronized. Therefore, the main idea to provide this feature is to setup the slave graph as soon as possible and additionally start streaming data through the slave graph while still

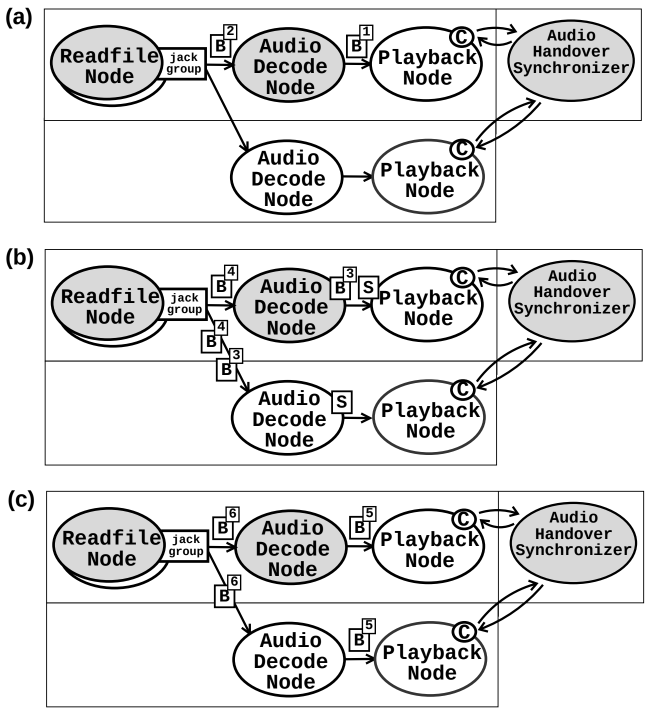

Figure 8: Synchronized continuous handover. While data buffers (marked as 'B', timestamp as number) are flowing in the master graph, remote nodes are setup (a), instream events are sent ('S' syncReset) and the slave graph is filled (b), data is flowing in both flow graphs simultaneously (c).

streaming data through the master graph. Presentation of multimedia data (e.g. playback of audio) will be done within the master graph only until the slave graph can present data synchronized with the master graph. Then, the presentation within the master graph is stopped and synchronously started within the slave graph.

The session sharing service described in Section 5 already provides all facilities needed for the first step, namely the automatic creation of the slave graph. All an application has to do, is to create a query that is first of all a copy of the currently running flow graph. Then, in this copy, all parameters can be reconfigured as wanted: in our example, the node for decoding audio and the sink node are configured to no longer run on the mobile system but on the stationary system (see Figure 7(a)). Also, additional nodes can be inserted, as desired.

The query is then forced to be mapped to the running session by additionally specifying its ID. The session sharing algorithm automatically maps the query to the running session; additional nodes are created and connected: again, for our example, the `ReadfileNode` is found to be shared (as it runs on the specified host "mobile"), and the `AudioDecodeNode` and the `PlaybackNode` are instantiated on the stationary system as specified in the query (see Figure 7(b)). The output jack of the `ReadfileNode` is duplicated within its jack group and connected to the remote AudioDecodeNode (*partial overlap* with copied output, see Section 5).

To realize the second step – namely the seamless and synchronized switching from the master to the slave graph – a special synchronizer is used that is – for our example – called `AudioHandoverSynchronizer`. This synchronizer gets connected to the controller objects of the sink nodes involved in the handover process. Furthermore, the functionality of the jack group is extended as described below.

In Figure 8(a), timestamped data buffers are only flowing in the

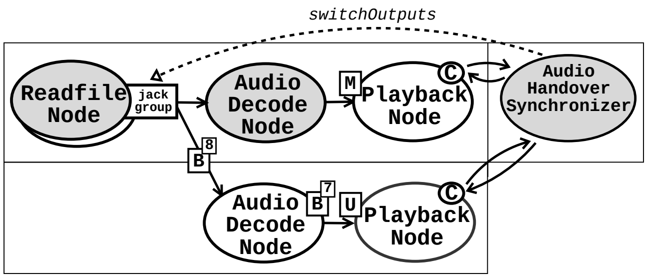

Figure 9: Complete handover using instream events for switching playback from the master to the slave graph ('M' mute, 'U' unmute).

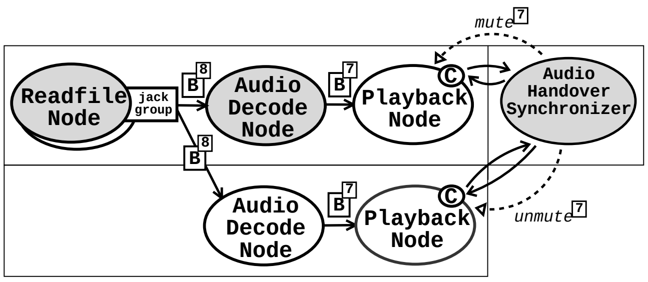

Figure 10: Complete handover using method calls for switching playback from the master to the slave graph (mute and unmute with additional timestamp).

master graph and playback is therefore only performed on the mobile system; the sink node is connected to the synchronizer. The slave graph is then set up on the stationary system as described above.

Once the connection is setup, the jack group sends an instream event called "syncReset" to both of its connected outputs *before* sending any multimedia data to its second output (see Figure 8(b)). This instream event will flow downstream and reach both sink nodes where it will force to reset the controller objects and send a new measured latency to the synchronizer object. From this moment, the same multimedia data will arrive at both sink nodes although with different latencies.

The synchronizer will then send the controller of the slave graph the latency as measured in the master graph. This is due to the fact, that the playback within the master graph should not been interrupted during handover and therefore the slave graph has to "catch up" with the master graph.

Depending on the used networking technology and protocol and due to the fact that the pipeline of the slave graph has to be filled first (including the delay of the algorithm for decoding multimedia data), the first data buffers usually arrive too late at the sink in the slave graph. Therefore, the controller of the slave graph will discard these buffers.

The next step in the handover procedure can be performed as soon as data in the slave graph arrives "in time" at its sink node. Since the slave graph will discard data that arrives too late, this will always be the case, if the time it takes for one buffer to stream through the complete slave graph is smaller than the time the same buffer will be presented (in our case: the raw audio data will be played back in the master graph). If this is *not* the case, the delay of the slave graph is *always* too large to ensure timely playback, which – in our case – would mean that the processing power provided for the slave graph is not sufficient for timely decoding and playback of the MP3 data. Of course, such a precondition should be fulfilled.

As soon as the first data buffer arrives in time, it could be played back synchronous with the sink node in the master graph (see Figure 8(c)). Depending on the behavior desired by the application there are two possibilities: The audio streams in both graphs – the master and slave graph – can be played back *simultaneously* from now on (*case 1*). This is basically the same behavior as used for session sharing in general (see Section 5).

The second possibility is to perform a *complete* handover (*case 2*): the audio playback would then stop within the master graph and start within the slave graph at the same time. To realize this strategy, the synchronizer will notify all jack groups that were created during the setup of the slave graph. Such a jack group will then insert an instream event "unmute" into the outgoing data stream to the slave graph, and an instream event "mute" into the outgoing data stream to the master graph (compare Figure 9). It will then stop sending data to its first output, namely the output that is connected to the master graph. Upon receiving the (synchronous) "unmute"-event, the sink node in the slave graph will seamlessly start synchronized playback of the audio data. The sink node in the master graph will stop playback when receiving the "mute"-event.

Another realization of this last step lets the synchronizer decide when directly to "unmute" playback in the slave graph respectively to "mute" playback in the source graph. This is done by estimating the maximum delay it takes to send a message to the two controllers. The "mute" and "unmute" method calls then additionally hold a timestamp that marks the execution time of the commands (see Figure 10). We found both approaches to be similarly efficient.

6.1 Results

We have measured the times for the described reconfiguration with two different setups: one uses two commodity Linux-PCs connected over 100 MBit Ethernet as master and slave; the other uses a Linux-PDA (namely the iPAQ H3870) with 11 MBit WLAN as master. For two PCs, it takes 0.562 for connecting the slave graph and another 0.461 until the playback is synchronized; together 0.983 seconds (see *case 1* above). A complete handover (see *case 2* above) takes additional 0.302 seconds. Using the PDA as master, these values raise to around 4 seconds (case 1) respectively 4.5 seconds (case 2) due to the lower computational power of the mobile device. Still, we think that these delays are tolerable since the media playback stays continuous and synchronized all the time.

7 Conclusions and Future Work

In this paper we presented an approach for sharing parts of an active flow graph within different applications. Using a generic synchronization architecture, this allows for joint and synchronized access of multimedia data on different devices. Based on these services is a mechanism for dynamically reconfiguring active flow graphs. During such an adaptation process, media playback stays continuous and synchronized. We demonstrated our implementation with different application scenarios and pointed out the most important results.

Future work will concentrate on the integration of Quality of Service measurements to guide the reconfiguration process. We will also study different policies for sharing devices in multi-user scenarios.

Acknowledgements This research has been supported by Motorola, Germany, and the Ministry of Saarland, Germany.

References

CARLSON, D., AND SCHRADER, A. 2002. Seamless Media Adaptation with Simultaneous Processing Chains. In *ACM International Conference on Multimedia*.

GORDON BLAIR AND JEAN-BERNARD STEFANI. 1998. *Open Distributed Processing and Multimedia*. Addison-Wesley.

KAHMANN, V., AND WOLF, L. 2002. A proxy architecture for collaborative media streaming. *Multimedia Systems 8*, 5.

LIN, J., GLAZER, G., GUY, R., AND BAGRODIA, R. 2002. Fast Asynchronous Streaming Handoff. In *Protocols and Systems for Interactive Distributed Multimedia Systems (IDMS/PROMS)*.

LOHSE, M., REPPLINGER, M., AND SLUSALLEK, P. 2002. An Open Middleware Architecture for Network-Integrated Multimedia. In *Protocols and Systems for Interactive Distributed Multimedia Systems (IDMS/PROMS)*.

LOHSE, M., REPPLINGER, M., AND SLUSALLEK, P. 2003. Session Sharing as Middleware Service for Distributed Multimedia Applications. In *Multimedia Interactive Protocols and Systems (MIPS)*.

MICROSOFT, 2003. DirectShow. http://msdn.microsoft.com/.

NTP: THE NETWORK TIME PROTOCOL, 2003. http://www.ntp.org/.

ROMAN, M., HO, H., AND CAMPBELL, R. H. 2002. Application mobility in active spaces. In *1st International Conference on Mobile and Ubiquitous Multimedia*.

SUN, 2003. Java Media Framework. http://java.sun.com/products/javamedia/jmf/.

WEDLUND, E., AND SCHULZRINNE, H. 1999. Mobility Support Using SIP. In *International Workshop on Wireless Mobile Multimedia*.