# Session Sharing as Middleware Service for Distributed Multimedia Applications

Marco Lohse, Michael Repplinger, and Philipp Slusallek

Computer Graphics Lab, Department of Computer Science
Saarland University, 66123 Saarbrücken, Germany
{mlohse, repplinger, slusallek}@cs.uni-sb.de

**Abstract.** The increased number of mobile and stationary devices with multimedia capabilities enables new application scenarios. Particularly interesting is collaborative multimedia access, where a number of users simultaneously enjoys the same content – possibly at different locations using different devices.
In this paper we focus on middleware support for such application scenarios. In particular our approach allows to share parts of an active multimedia session and provides synchronized and distributed media playback on various devices within different applications. As a middleware service, our solution automatically maps new application requests to already registered playback sessions. We demonstrate our approach with different application scenarios that include synchronized media playback on stationary and mobile devices.

## 1 Introduction

Mobile and networked devices with multimedia capabilities are quickly becoming ubiquitous. Devices such as tiny notebooks, portable web pads, personal digital assistants (PDAs), and even mobile phones already offer reasonable multimedia capabilities. Remaining differences are mainly due to the limited processing power and missing I/O devices (e.g. TV receivers or DVD drives).

The limitations can be overcome by augmenting these devices with the capabilities from commonly available desktop systems. The desktop systems can for instance provide network access to sources of multimedia data and can perform compute intensive tasks. This collaboration opens up many new application scenarios. In this paper we concentrate on collaborative media playback where multiple mobile devices access services of stationary systems.

Commercial approaches of combining stationary and mobile devices are mostly static with stationary systems providing a fixed set of services only: e.g. a streaming server sending media data across the network or a telephony gateway offering access to a telephone line. Clients have very limited options for configuring the service and cannot change its functionality.

In contrast, we assume a dynamic and flexible approach, where stationary systems provide fine grained access to their individual multimedia devices (e.g.

TV receiver or telephone line) and processing capabilities (e.g. MPEG compression) that can be used by other applications in the network (given suitable access rights). This approach, however, requires a *network-integrated multimedia middleware* that is capable of locating, controlling, and combining devices and processing tasks across machines in the network.

For collaborative media playback we assume the following scenarios: Users should be allowed to initiate multimedia playback sessions on any devices in the network. Other users or applications may then be allowed to share as much of this playback session as possible. This eliminates the problem of multiple access to a single multimedia device and reduces the processing requirements when many clients share common tasks in a session. Playback should be synchronized across all participating devices in these scenarios.

Possible application scenarios are a sports event that is displayed on the PC in the living room and simultaneously shown on a PDA while you prepare dinner in the kitchen. Or, different users jointly watch a DVD but use their PDA and earphones to listen to different language tracks.

In this paper we present a multimedia middleware (Section 3) that supports these requirements and scenarios while enabling a wide variety of additional applications. The major contributions discussed in this paper are:

- The realization and creation of distributed flow graphs from a high-level description (Section 4).
- Sharing parts of active sessions for joint access to devices and reduced computational load (Section 5).
- Synchronized playback across networked devices (Section 6).
- Demonstration of selected application scenarios (Section 7).

## 2 Related Work

Standards for device discovery in home networking environments like Jini [1], HAVi [2], and UPnP [3] only partly solve the requirements mentioned above. They allow the search and reservation of devices for a specific application, but these devices cannot be shared between different applications.

The idea of accessing multimedia data on nearby stationary systems is presented in [4]. However, this approach is restricted to the IEEE 1394 networking technology. An OSGi compatible solution for location dependent playback of multimedia data is described in [5]. While OSGi provides a standard way to connect devices such as home appliances it does not provide facilities especially needed for handling multimedia, e.g. synchronization [6]. Therefore, the cited approach uses third-party components for streaming and playback of multimedia data. Another approach uses standard streaming servers and clients together with a proxy architecture to create collaborative media streaming [7]; support for strict synchronization or sharing of multimedia flow graphs is not provided.

While different synchronization protocols for distributed multimedia applications have been developed, e.g. [8] or [9], we focus on providing a middleware framework that allows different protocols to be used for synchronized playback.

## 3 Underlying Middleware

The *Network-Integrated Multimedia Middleware* (NMM) allows to access and control distributed multimedia devices and software components which can be integrated into a common flow graph. The open architecture does not rely on a particular technology but allows for the flexible usage of different networking and other middleware technologies. Together, NMM offers following advantages.

- **Support for heterogeneous environments** through the usage of mediating proxy objects and parameterizable communication strategies. This allows to integrate other multimedia frameworks.
- **Explicit binding** allows to chose the communication strategy independently for the transmission of multimedia data and control of components.
- **Usage of optimized communication strategies.** Platforms with restricted resources, like PDAs, can be integrated by using light-weight transport strategies. This also allows to remove the overhead of the middleware for locally running parts of the application.
- **Reflection and event notification.** All components can be queried for their supported functionality and the application can register to be notified when a certain event occurs.

NMM is implemented in C++ and runs under Linux. The NMM framework and applications are available as Open Source (see www.networkmultimedia.org). In the remainder of this section we briefly describe the middleware. More details can be found in [10].

The general design approach of the NMM architecture is similar to other architectures. Within NMM, all hardware devices (e.g. a DVD-ROM drive) and software components (e.g. decoders) are represented by so called *nodes* (see Figure 1). A node has properties that include its input and output ports, called *jacks*, together with their supported multimedia formats. Since a node can have several inputs or outputs, its jacks are labeled with *tags*. Depending on the specific kind of a node, its innermost loop produces data, performs a certain operation on the data, or consumes data. These nodes can be connected to create a flow graph, where every two connected jacks support the same format. The structure of this graph then specifies the operation to be performed.

The NMM architecture uses a uniform message system for all communication. There are two types of messages. Multimedia data is placed into messages of type *buffer* that are streamed across connected jacks. Messages of type *event* forward control information such as a change of speaker volume. Events are identified by a name and can include arbitrary typed parameters. They can be generated within nodes and are then forwarded instream just like buffers, or they can be sent between the application and nodes (out-of-band).

To achieve network-transparent use of distributed objects, like NMM nodes and jacks, they are controlled via proxy objects. Proxy objects allow for redirecting events to and from remote objects. Furthermore, proxies can act as translators between different technologies and allow to integrate middleware like the

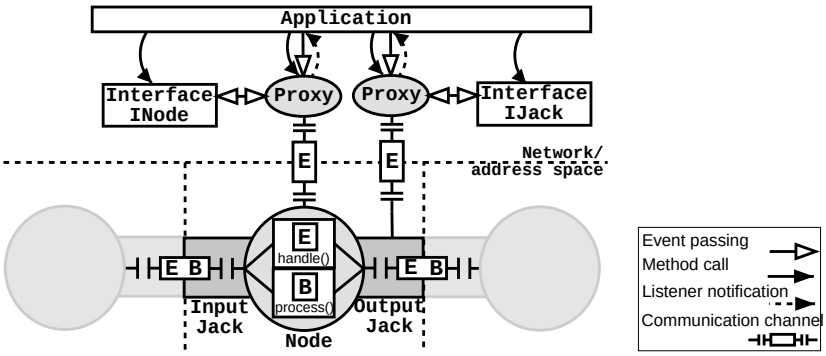


**Fig. 1.** Node connected to other nodes via input and output jack; messages ('E' event, 'B' buffer) are sent instream and out-of-band; interfaces allow to control objects via proxies. Communication channels provide an explicit binding.

Java Media Framework (JMF) into NMM [10] by means of mediating proxies that translate between the different middleware APIs and protocols.

Furthermore, object-oriented interfaces allow to control objects by simply invoking methods, which is more type-safe then sending events. These interfaces are described in an interface definition language (IDL) that is similar to CORBA IDL. For each description, an IDL compiler creates an interface and implementation class. Internally, these classes use events, and therefore provide the possibility to notify any listener. During runtime, supported interfaces can be queried by the application.

The bidirectional communication between a proxy and its controlled object as well as the data transport between two connected jacks is performed with so called *communication channels*. Messages sent across a communication channel are automatically serialized, transmitted, and then deserialized. For this, a communication channel internally uses a *transport strategy* that employs one (or more) *serialization strategies*. Communication channels are modeled as *first class data type* meaning that they can be used and manipulated like any other object. Therefore, they provide an explicit binding as opposed to the implicit binding mechanisms that can be found in traditional middleware.

Explicit binding allows for selecting and configuring the serialization and networking technology to be used for data transport. For serialization and deserialization we currently provide an XML strategy and a more efficient strategy using "magic numbers" where type information is mapped to predefined numbers during serialization. Both representations can be directly transmitted over sockets using protocols like TCP or UDP. Another option is the combination of a serialization and transport strategy that uses the CORBA any-types for transmission. We use The Ace Orb (TAO) [11] that also provides real-time extensions. Furthermore, explicit binding is especially important for selecting and configuring network protocols that are suitable for streaming multimedia data, like RTP.

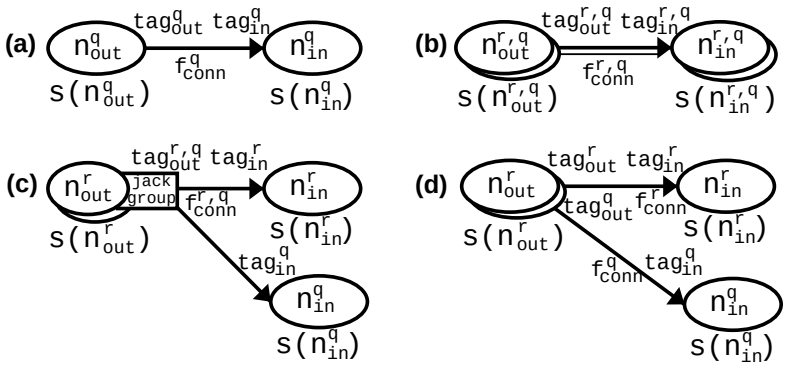


**Fig. 2.** An edge of a graph description for a query $Q$ in (a), a complete overlap of an edge of a running session $R$ in (b), a partial overlap using a duplicated output within a jack group in (c), a partial overlap with a different output connected in (d).

While establishing a communication channel, an application can also use a negotiation mechanism that automatically selects strategies for serialization and transport. If, for example, a communication channel to a locally operating object is established, a strategy that simply forwards the pointer to a message would be chosen. This allows to efficiently handle locally running parts of an application. Also, if an object migrates to a different host during runtime, only the strategy used within the communication channel has to be updated. With other approaches [12], special elements have to be inserted into the flow graph to handle network communication.

## 4 Registry Service and Setup of Flow Graphs

The registry service in NMM allows discovery, reservation, and instantiation of nodes available on local and remote hosts. On each host a unique *registry server* administrates all NMM nodes available on this particular system. For each node, the server registry stores a complete *node description* that includes the specific type of a node (e.g. "DVBReceiverNode"), the provided interfaces (e.g. "TVTuner"), and the supported input and output formats (e.g. "video/mpeg").

The application uses a *registry client* to send requests to registry servers. A request is also specified as node description but only needs to include the aspects important for an application, like a specific interface or the location of the node. A request is send to the specified host, which defaults to the local host.

After successfully processing the request the server registry reserves the requested nodes. Nodes are then created by a factory either on the local or remote host. For nodes to be instantiated on the same host, the client registry will allocate objects within the address space of the application to avoid the overhead of an interprocess communication.

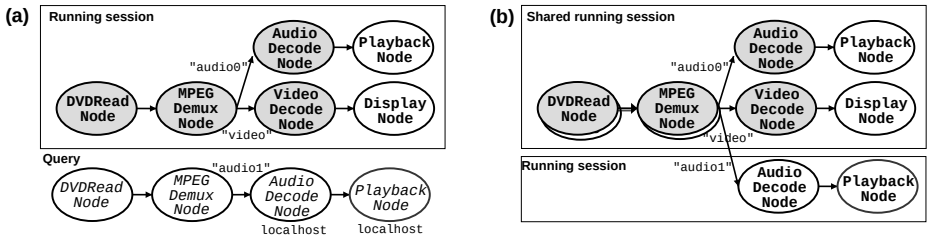


**Fig. 3.** Session sharing. The query for a different audio track of an already running session (a) is mapped by the session sharing algorithm to use a shared sub-graph (b).

To setup and create complex distributed flow graphs, an application can either request each node separately or use a *graph description* as a query $Q$. Such a description includes a set of node descriptions connected by edges: an edge $e^q$ in a graph description marks the connection from node description $n^q_{out}$ to $n^q_{in}$. Since nodes can have several inputs or outputs, the intended connection has to be further specified by $tag^q_{out}$ and $tag^q_{in}$ for the wanted output and input. In addition, a connection format $f^q_{conn}$ can be associated with an edge (see Figure 2(a)). The client registry then requests each node from the corresponding server registry and establishes the specified connections between nodes.

## 5 Automatic Session Sharing

A first application scenario motivates our core concept, namely *session sharing*. In this scenario, the user starts watching a DVD at a stationary system: the audio/video stream is read, demultiplexed, decoded, and rendered with an appropriate flow graph of multimedia processing units (see Figure 3(a)). This flow graph was created with a graph description as explained in Section 4.

Several application scenarios are possible: the user that initiated the DVD playback simultaneously wants to watch the DVD using a second device as audio/video output – for instance a mobile system. Or another user wants to join watching the DVD on a different system, maybe at a different location. Another situation might arise if two users want to watch a DVD on the same screen while listening to different audio tracks (e.g. different languages) with a mobile device.

In any case, the second application will at least need access to the node reading the DVD. Sharing of nodes is necessary not only in cases where only a single device exists to perform a certain operation, but also allows for sharing computational resources among tasks. Therefore, we introduce the concept of a *session*: a session is an abstraction for a flow graph of already reserved and connected nodes and is stored within the registry service. Nodes of this graph — and implicitly edges connecting nodes — can be marked as *sharable* to be reused by other applications within their flow graphs as *shared sessions*. Setting the sharing policy is done per node within a graph description (see Section 4). The sharing policy includes the following modes to control application demands:

– **Shared:** explicitly request a shared node.
– **Exclusive:** request a node for exclusive use; if no such node is available exclusively, the query will fail, even if the node exists as shared node.
– **Exclusive or shared:** try exclusive first, then shared.
– **Shared or exclusive:** try shared first, then exclusive.
– **Exclusive, then shared:** try exclusive, if successful share the node.

A running session with shared nodes is shown in Figure 3(a). In this example, the application chose to share all nodes (shaded dark) except the sink nodes for rendering audio and video. Shared parts of a session can be reused: another application can integrate the source node for receiving streams from DVD within its own flow graph, or, if wanted, integrate the complete flow graph for receiving and decoding the stream into its own flow graph and only provide additional sink nodes, e.g. for audio and video. Therefore, 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 $Q$ such as the one in Figure 3(a). Here, the mode "exclusive or shared" is set for all node descriptions. Furthermore, decoding and playback of audio are specified to be performed on the local host and the "audio1" output is requested from the demultiplexer (instead of "audio0").

If an application sends a graph description $Q$ to the registry service that includes node descriptions $n$ with a sharing policy that allows sharing (such as *shared*, *exclusive or shared* or *shared or exclusive*, in general denoted as $s(n) = true$), the registry searches all already registered sessions that share resources. The main idea of this search is to find overlapping sub-graphs between already running sessions $R_1, \ldots, R_n$ and the current query $Q$.

The computation of an overlapping between two graphs is divided into individual test for edges $e^q$ of graph description $Q$ and edges $e^r$ of some running session $R_i$.

The edge $e^q$ from node description $n^q_{out}$ to $n^q_{in}$ *completely* overlaps edge $e^r$ from $n^r_{out}$ to $n^r_{in}$, if all of the following criteria hold (test 1 to 4):

1. The **sharing policy** allows sharing for the node descriptions $s(n^q_{out}) = s(n^r_{out}) = true$ and $s(n^q_{in}) = s(n^r_{in}) = true$
2. The **node descriptions** of $e^q$, namely $n^q_{out}$ and $n^q_{in}$, are *subsets* of $n^r_{out}$ and $n^r_{in}$, respectively (see below).
3. Since nodes can have several outputs or inputs, the **tags** specifying output and input are equal, namely $tag^q_{out} = tag^r_{out}$ and $tag^q_{in} = tag^r_{in}$
4. If the **connection format** $f^q_{conn}$ of $e^q$ is specified, the connections formats have to be equal: $f^q_{conn} = f^r_{conn}$.

This case is termed *complete overlap*. Test 2 furthermore involves several other tests. First, the names of the nodes have to be identical. Further test are all performed as tests for possible subsets:

– The sets of supported interfaces for $n^q_{out}$ and $n^q_{in}$, have to be subsets of the supported interfaces of $n^r_{out}$ and $n^r_{in}$, respectively.
– Also, the generally supported formats required by $n^q_{out}$ and $n^q_{in}$ have to be supported by $n^r_{out}$ and $n^r_{in}$, respectively.

– If the location – the host on which the nodes are running – of $n^q_{out}$ and $n^q_{in}$ is specified, it has to be the same as the location of nodes $n^r_{out}$ and $n^r_{in}$, respectively; otherwise the nodes connected by $e^q$ can be located on any host, which includes the hosts of the nodes of $e^r$.

All these tests are only performed if the corresponding information has been specified for node descriptions $n^q_{out}$ or $n^q_{in}$; otherwise a test is assumed to be fulfilled. Figure 2(b) illustrates such completely overlapping edges $e^q$ and $e^r$.

The edge $e^q$ *partly* overlaps $e^r$ in two cases: First, if test 4 is satisfied and tests 1 to 3 are satisfied only for *outgoing* elements, namely $n^q_{out}$ and $tag^q_{out}$. This case is termed *copy overlap*. An example is a session with a shared node for reading from a DVD (`DVDReadNode`) connected to a shared node for demultiplexing (`MPEGDemuxNode`) both running on host A. This edge is only partly overlapped by another edge connecting a `DVDReadNode` on host A to a `MPEGDemuxNode` on host B because the demultiplexer node should be located on another host. As can be seen in Figure 2(c), when realizing this case, a copy of the corresponding output jack and an additional connection to $n^q_{in}$ is created. Both jacks are inserted into a *jack group*. Such a jack group then forwards the data stream to all output jacks. Although a node always uses a jack group for each of its outputs (even if only one output jack is present), we only depict them if two or more jacks are used.

Secondly, the edge $e^q$ *partly* overlaps $e^r$, if test 1 and 2 is satisfied – again, for outgoing elements, namely $n^q_{out}$ – and $tag^q_{out}$ is a valid tag but in this case with $tag^q_{out} \neq tag^r_{out}$. This case is termed *output overlap*. An example for this would be a node within a session with one or more unconnected outputs. In this case, such an unconnected output can be used without any further restrictions within another session. Figure 2(d) illustrates this case.

The session mapping procedure then works as follows. For each running session $R_i$, all nodes $r^i_1, \ldots, r^i_m$ with no incoming edges are identified. These nodes represent the sources of data for a session. Also for the query $Q$, all nodes $q_1, \ldots, q_{m'}$ with no incoming edges are identified. Intuitively, an overlapping between $R_i$ and $Q$ only makes sense, if at least one partial overlap between an edge leaving some $r^i_j$ and an edge leaving some $q_{j'}$ can be found. Otherwise, the two different sessions would try to share internal nodes for different sources of data.

For each $q_x$ and each $r^i_y$ the algorithm tries to find an overlap by comparing all outgoing edges of $q_x$ with all outgoing edges of $r^i_y$. The test for a copy overlap is performed first since it is a subset of a complete overlap. If the test was successful, the test for a complete overlap is carried out. In any case, the test for an output overlap is performed in the end. For each successful test, a recursion of a depth-first search is started with sub-graphs of the original graphs $Q$ and $R_i$. For a complete overlap, these sub-graphs no longer contain the completely overlapped edge. For a copy or output overlap, the complete sub-graph starting from $q_x$ is removed from $Q$. Intuitively, this reduction of the search space is legal because starting from this point, the graph $Q$ and $R_i$ continue to "grow" in different directions.

Even if a complete overlap exists, the tests for partial overlaps are performed and a recursion is started if they succeeded. This is due to the fact that a complete overlap might lead to a dead end in the search later on. This is the case if no overlap exists for the next edge within the subgraph of $Q$ used in the recursion. Furthermore, computing all possible overlaps allows to apply different value functions as described below. Although an exhaustive search is needed, the number of iterations performed by the algorithm is relatively small. This is the result of the limited number of edges in typical flow graphs and the even smaller number of possible overlaps between these edges due to the strict criteria. With our current implementation, the session sharing algorithm takes 0.059 seconds for the scenario shown in Figure 3 and 0.108 seconds for the one in Figure 5 (measured on a commodity Linux PC with 866 MHz).

As mentioned above, different overlaps can potentially exist for $Q$ and different $R_i$'s but also for a single $R_i$. Therefore, for all computed overlaps, all node descriptions in $Q$ that were mapped to nodes of $R_i$ are valued as reduced costs. The overlap that reduces the additional costs of $Q$ most is finally chosen. Although we are currently using a simple cost function, the costs for each node can also be evaluated with measured QoS requirements.

The final setup of a session with shared nodes is simple: nodes of a complete overlap are referenced in the new session. For a partial overlap, the output jack is duplicated within a jack group and then connected (copy overlap) or unconnected edges are connected to newly instantiated nodes (output overlap). Nodes that cannot be shared at all are created and connected.

Figure 3(b) shows this result for our example: the `DVDReadNode` and the `MPEGDemuxNode` and their connected edge are now shared for the second session, 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. With this setup, a different audio stream will be rendered on the device that runs the second session.

## 6 Distributed Synchronization of Shared Sessions

In order to synchronize playback between local and remote parts of a single application or shared sessions, our multimedia middleware provides a distributed synchronization architecture. The basis for performing synchronization is a common clock. In our case, this is a static object within each address space representing the system clock, which is assumed to be globally synchronized by the Network Time Protocol (NTP).

In distributed scenarios, it is especially important to minimize network communication needed for synchronization. Therefore, our architecture strictly distinguishes between objects realizing intra-stream (the temporal relations between several presentation units of the same media stream) and inter-stream synchronization (the temporal relations between different streams).

Figure 4 shows the architecture. First, timestamps are set within nodes (e.g. an MPEG decoding node). Synchronized sink nodes use a *controller* object to

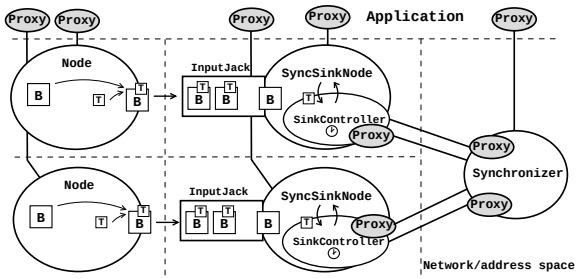


**Fig. 4.** Distributed synchronization architecture showing controller objects running within synchronized sink nodes and a single synchronizer for realizing a synchronization algorithm ('B' buffer, 'T' timestamp). Clocks within controllers are globally synchronized.

realize intra-stream synchronization: for every buffer, the controller compares the timestamp with the presentation time derived from the global clock to decide when to present the buffer. If the buffer arrived too late it is discarded. As this process is running within a node, no network traffic is involved.

If multiple data streams for different sink nodes are to be presented synchronously, the controller objects are also connected to a *synchronizer* that performs inter-stream synchronization. In general, this is done by assuring that corresponding buffers at different sink nodes are presented at the correct time. All controllers agree on the presentation time set by the synchronizer as an offset to the global NTP clock. The synchronizer derives this offset by comparing the latencies measured by its connected controllers. To minimize network traffic, these values are only updated when necessary (e.g. due to changes in networking conditions).

In order to synchronize shared sessions, the *session* object that is stored in the registry service also administrates the synchronizer of the session. If a new session is created that uses shared nodes of another session it will also try to use the shared synchronizer.

Therefore, we provide a synchronizer that allows to connect several audio and video sinks. Due to possible time drifts between different audio devices, this synchronizers uses one distinct audio sink node as master and adjusts all other audio sink nodes to the speed of the master by setting appropriate parameters at the controller objects. The controllers then adjust their playback speed, e.g. by doubling or dropping samples.

## 7 Application Scenarios

Based on the results from Sections 5 and 6, we have realized different applications. We are using commodity PCs and iPAQ H3870 PDAs with WLAN adaptors both running under Linux without any real-time extensions. The first application (that is also described in Section 5) allows a number of users to simultaneously access different audio tracks of a DVD while watching the same

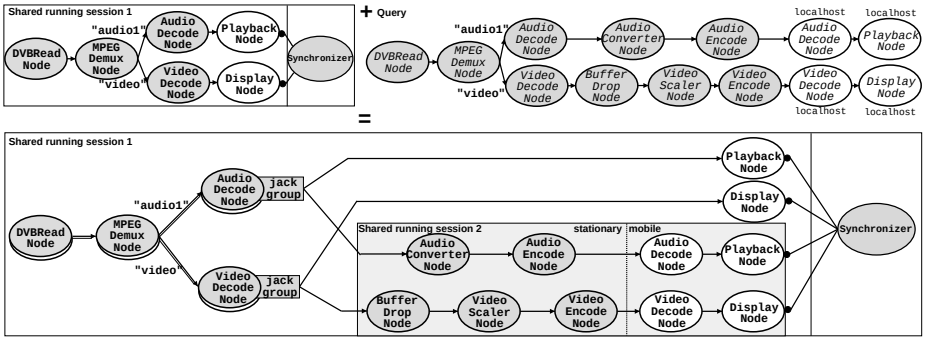


**Fig. 5.** A TV application that allows synchronized playback on a stationary and mobile system. The nodes for decoding MPEG audio and video are used shared. Multimedia data is transcoded for playback on the mobile device by additional nodes.

video stream presented at a TV that is connected to a PC. The different audio streams are decoded on the mobile device where headphones allow to listen to the specified audio stream.

Another application allows to watch TV on an iPAQ and a PC simultaneously. Figure 5 illustrates the TV application and how the common nodes are shared between the application running on the PC and the PDA. The existing session 1, which was started from the PC, allows to share all nodes except the `PlaybackNode` and the `DisplayNode`. The TV-application running on the PDA starts a query and the algorithm for session sharing uses session 1, which is the best possible match in our scenario. Right after the `AudioDecodeNode` and the `VideDecodeNode`, output jacks are dynamically duplicated within a jack group to connect the remaining nodes to a flow graph as described in the query. Additional nodes are used to subsample the audio and resize the video stream suitable for the PDA's maximum display resolution of 320x240 and its computational resources for decoding.

Before transmitting the data over the network, an video encoder for MPEG4 and an MPEG audio encoder is added. The synchronizer shared by session 1 is used to synchronize the application on the PDA as well. We measured the synchronization offset between both system by comparing their audio output and found this offset to be close to the accuracy provided by our NTP setup (e.g. less than 5 ms). Notice, that the application running on the iPAQ will both use remote and local nodes within its flow graph, where remote nodes are partly taken from another session. Furthermore, the nodes for transcoding audio and video are also to be shared (shaded dark) and can be used by another application, e.g. running on another PDA.

An additional `BufferDropNode` is used for dynamic adaptation because the iPAQ does not provide enough computing power to decode and render 25 fps. Together with the information obtained by the controller of the video sink, the synchronizer configures how many frames should be dropped by sending events

to the `BufferDropNode`. These events are send *upstream* meaning that they are forwarded from one node to its predecessor starting at the video sink. With the current implementation, the iPAQ is able to display 8-10 fps with a resolution of 320x240 and 11 kHz mono audio.

## 8 Conclusions and Future Work

In this paper we present session sharing as an additional service within a multimedia middleware. A registry service sets up distributed flow graphs due to application request. Such a graph is stored as a session within a registry service. An algorithm for mapping new requests to already running sessions allows for the collaborative multimedia access to shared devices for a number of users and reduces the computational load by reusing shared processing resources.

Synchronized audio and video rendering for such distributed applications is realized as another middleware service. We demonstrate our approach with different applications that perform dynamic adaptation to cope with the restricted resources of the mobile devices used.

Future work will focus on the migration of parts of sessions during runtime due to user mobility. Quality of Service measurement techniques will help to further improve the value functions for choosing sub-graphs to be shared.

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

## References

1. Jini. http://www.jini.org/ (2003)
2. The HAVi Specification. http://www.havi.org (2003)
3. Universal Plug and Play. http://www.upnp.org/ (2003)
4. Baldus, H., Baumeister, M., Eggenhuissen, H., Montvay, A., Stut, W.: WWICE - An Architecture for In-Home Digital Networks. In: Multimedia Computing and Networking (MMCN). (2000)
5. Eikerling, H.J., Berger, F.: Design of OSGi Compatible Middleware Components for Mobile Multimedia Applications. In: Protocols and Systems for Interactive Distributed Multimedia Systems (IDMS/PROMS). (2002)
6. Open Services Gateway Initiative (OSGi). http://www.osgi.org (2003)
7. Kahmann, V., Wolf, L.: A proxy architecture for collaborative media streaming. Multimedia Systems **8** (2002)
8. Rothermel, K., Helbig, T.: An Adaptive Protocol for Synchronizing Media Streams. ACM/Springer Multimedia Systems **5** (1997)
9. Shivakumar, N., Sreenan, C.J., Narendran, B., Agrawal, P.: The Concord Algorithm for Synchronization of Networked Multimedia Streams. In: International Conference on Multimedia Computing and Systems. (1995)
10. Lohse, M., Repplinger, M., Slusallek, P.: An Open Middleware Architecture for Network-Integrated Multimedia. In: Protocols and Systems for Interactive Distributed Multimedia Systems (IDMS/PROMS). (2002)
11. The Ace Orb (TAO). http://www.cs.wustl.edu/∼schmidt/TAO.html (2003)
12. Black, A.P., Huang, J., Koster, R., Walpole, J., Pu, C.: Infopipes: An abstraction for multimedia streaming. Multimedia Systems **8** (2002)