TOWARDS AUTOMATIC SETUP OF DISTRIBUTED MULTIMEDIA APPLICATIONS

Marco Lohse and Philipp Slusallek
Computer Graphics Lab, Department of Computer Science
Saarland University, Germany
email: {mlohse, slusallek}@cs.uni-sb.de

ABSTRACT
Recent developments in the area of multimedia middleware have proven to provide suitable abstractions for supporting various ubiquitous computing scenarios. As a typical example, home entertainment applications aim at providing a "follow-me" service, where audio/video output is performed using devices embedded in the current surrounding of the user.

However, manually setting up such distributed multimedia applications is a complex task. Therefore, middleware needs to provide further guidance. In this paper, we provide an in-depth analysis of the challenges for such a service. Based on these observations, an algorithm is derived that allows for the automatic creation of distributed multimedia data flow graphs from a given high-level description. The applicability and performance of our approach is evaluated by demonstrating relevant application scenarios.

KEY WORDS
Multimedia Information Systems, Distributed Multimedia Systems, Multimedia Tools and Architectures

1 Introduction

Due to the constantly growing number of networked devices in our environment, there is a strong trend towards ubiquitous computing. Of particular interest are context-aware applications, such as "follow-me" services for multimedia home entertainment. In such scenarios, audio and video output of media content is performed at the current location of the user by employing nearby devices embedded in the environment, for example a networked hi-fi audio system and a large video screen connected to another system.

Realizing such applications is a demanding task. Besides the sensing of the current location of the user – a topic not covered in the scope of this paper –, following tasks need to be handled. First, remote media sources need to be accessed, e.g. media servers containing a variety of different file types, but also different "live" sources, such as TV receivers. Then, nearby devices need to be determined and media streams need to be routed to chosen output devices. For being able to decode the large variety of different multimedia formats available today, the appropriate components need to be determined and integrated into media processing. Eventually, media streams also need to be adapted to output devices on-the-fly.

Software architectures for developing multimedia applications usually adopt a *flow graph* based design approach, where a specific task (e.g. media playback) is modeled by specifying a directed graph of independent fine-grained processing elements (also called "plug-ins"). Each of these elements represents a certain multimedia device (e.g. a sound board) or operation (e.g. media decoding).

Common multimedia architectures that adopt such a flow graph based approach, such as DirectShow [5] or the Java Media Framework [6], are restricted to operate on a single system. The network is only used as source of data, e.g. when receiving predefined content from a streaming server – cooperation and control of distributed processing elements is not provided.

In contrast, middleware solutions for developing *distributed* multimedia applications have recently emerged. The Network-Integrated Multimedia Middleware (NMM) [3] used as underlying framework for the work presented in this paper allows for realizing *transparently* distributed flow graphs, i.e. graphs consisting of elements located on different hosts.

Previous approaches that aim at setting up distributed applications are restricted to identify possible paths for serial configurations [7]; incomplete and general graph based flows are not supported. Another approach aims at mapping arbitrary predefined flow graphs to devices within the network; the completion of high-level descriptions is not available [1]. An algorithm for connecting a single source of media to a single sink is described in [4]. However, arbitrary flow graphs are not supported. Algorithms employed by DirectShow [5] and the Java Media Framework [6] are inherently restricted to create locally operating flow graphs.

In contrast, we use the NMM framework as enabling technology for developing a middleware service that automatically creates a distributed flow graph from a given high-level description. This description only needs to specify the intended task plus additional constraints such as the position of the user. The integration of required processing elements and their optimal distribution within the network is performed automatically. Section 2 will first describe the NMM framework in more detail. In Section 3, the challenges for the automatic setup of distributed multimedia

applications are discussed. Then, the derived algorithm is described in detail in Section 4. Finally, results and applications are presented in Section 5.

2 Underlying Architectural Model

The Network-Integrated Multimedia Middleware (NMM) [3] is used as underlying framework for our approach. It allows for realizing *transparently* distributed flow graphs. Such a graph consists of processing elements, called *nodes*, that can be located on different hosts. Connection setup of distributed nodes and their control is identical to local operation. In particular, applications control nodes by invoking methods on *interfaces*, e.g. general interfaces, such "INode", or interfaces specific for a certain type of multimedia device, such as "IAudioSink". Interfaces are described using an interface definition language (IDL) that is very similar to CORBA IDL.

Conceptually, a processing node within NMM provides a number of input and output ports, called *jacks*. Associated with each jack is a set of supported formats. A format precisely describes a certain data stream. For example, uncompressed PCM audio is specified by its general type (e.g. "audio/raw") but also by technical parameters (e.g. the number of channels, the sampling rate, or the number of bits per sample). A parameter is specified by a key (e.g. "sampling rate") together with a set of possible values, i.e. single values ("48 kHz"), ranges of values ("11 to 48 kHz") or without restrictions.

A connection between an output jack and an input jack is only allowed if a "matching" format exits. Such a *connection format* is available if the types of the input and output format are equal, all parameter keys in one format exist in the other format and vice versa, and all intersections of values of corresponding parameter keys are non-empty.

The data flow within a flow graph is generated by forwarding media samples encapsulated in data buffers from connected outputs to inputs. Depending on its specific type, the innermost loop of a node produces data, performs a certain operation on data received, or consumes data. In general, six different types of nodes can be distinguished: A source produces data and has a single output. A sink consumes data received from its input. A filter has one input and one output port. It modifies the data but does not change the format or format specific parameters of the stream (e.g. an audio filters for controlling the volume). In contrast, a converter changes the type of format (e.g. from MPEG audio to uncompressed PCM audio) or format specific parameters (e.g. the audio sampling rate). For merging different streams, a multiplexer provides several inputs and a single output; for splitting interleaved streams a demultiplexer offers one input and several outputs.

As an example, Figure 1 shows a distributed flow graph for playing back MP3 files. A source node for reading data from the file system is connected to a converter node for decoding MPEG audio streams. This node is in turn connected to a sink node for playing out decoded au-

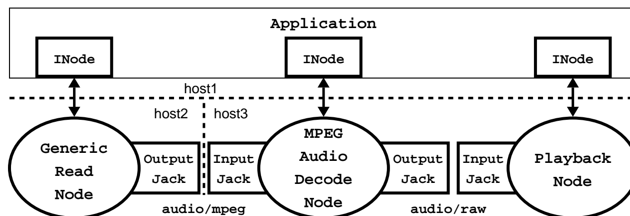

Figure 1. A distributed flow graph for MP3 playback.

dio samples using a sound board. All nodes are controlled via their exported interfaces. In this example, the application is running on host1, the source node on host2, and the decoder and sink node on host3. Therefore, data is read from the file system of host2 and decoded and played out on host3. A network connection is automatically set up for streaming media samples along the connection between the source and the decoder. Due to this distribution of nodes, compressed media data is transmitted over the network and the required bandwidth is minimized.

The registry service of NMM allows for the 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 name of a node (e.g. "PlaybackNode"), its type (e.g. "sink"), the provided control interfaces (e.g. "IAudioSink"), and the supported input and output formats (e.g. "audio/raw" plus parameters). 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. Remote nodes are requested from remote registry servers. If more than a single node satisfies the criteria given by the request, the complete node descriptions of all matching nodes are returned for further inspection.

3 Challenges and Requirements

The problem definition of *automatic application setup (AAS)* can be stated as follows. *Given a high-level description of an intended task plus additional constraints, try to find a valid and fully configured distributed flow graph that meets the given constraints and is distributed in an optimal way.* This definition and the imposed challenges will be further explained in the following.

We propose the concept of a *user graph* as high-level abstraction for specifying tasks. A user graph defines an abstract flow graph where only the key components, their connections, and additional constraints are defined. For example, the task "media playback" is represented by the user graph shown in Figure 2, which requires a uniform resource locator (URL) to be given. Using our approach, URLs can specify files ("file"), but also multimedia devices, such as

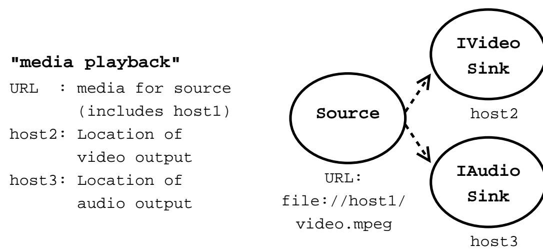

Figure 2. A user graph for distributed media playback.

CD or DVD drives ("audiocd" or "dvd"), and TV boards ("tv"). A URL already includes a location, e.g. the host from which the file should be read. Optionally, the locations of devices for media output can be defined, which are then mapped to the sink nodes within the user graph. Likewise, user graphs for tasks like capturing live media or transcoding of files (i.e. converting the format of media, such as the used codec) can be defined. An application can select the intended user graph from a repository.

A user graph is independent of a possible physical realization: Sink nodes are only specified using abstract node descriptions, e.g. the interface "IAudioSink" is used instead of the precise node name "PlaybackNode". This is important for distributed environments where a plug-in with a certain name cannot be assumed to be available on all systems. Instead, a specific interface is used as description of its functionality. Furthermore, connecting edges can be chosen independently of possible connections between nodes in terms of matching formats. To this end, the structure of the graph itself does not need to include converters; demultiplexers and multiplexers can also be left out.

Therefore, AAS can be seen as search for a flow graph of intermediate nodes that try to connect the nodes specified by the given user graph. Notice that not necessarily all nodes of the user graph can be connected, e.g. when specifying a URL of an MP3 file, the video sink given in the user graph cannot be connected. Together, two questions need to be addressed: First, the nodes and their connecting edges and formats of the resulting flow graph have to be determined. Second, since nodes in the user graph – and therefore also nodes in the resulting flow graph – can be located on different hosts, the physical location of each node needs to be determined. The distribution of nodes should be "optimal" in the sense that the required networking bandwidth is minimized.

A specific property of flow graphs further determines how AAS can be handled: Typically, a node needs to process multimedia data in order to determine its supported output formats. However, for providing multimedia data for a processing element, all preceding nodes already have to be set up and connected properly. Therefore, the creation of flow graphs should be regarded as an iterative search process that is restricted to operate step-by-step.

On the one hand, the restrictions imposed by outgoing formats greatly reduce the search space when finding intermediate successor. On the other hand, each chosen node or connection format influences the possible choices for following nodes and connection formats. This is due to the fact that a chosen input format directly determines the outgoing formats, which in turn restricts the possible successors. Therefore, each choice made can result in unresolvable conflicts later on. Therefore, AAS potentially needs to perform backtracking, i.e. created sub-graphs need to be discarded and alternative configurations have to be examined. However, since the minimization of setup times is also an important goal for AAS, we prose an alternative approach in Section 4.1.

Notice that different optimization criteria are reasonable for AAS. Especially the needed computational power of nodes and networking bandwidth of data streams influence the creation of "optimal" distributed flow graphs. However, today's environments mainly operate at best-effort, i.e. without strict Quality of Service (QoS) guarantees. Therefore, the proposed algorithm uses a greedy strategy that aims at minimizing the required bandwidth; advance QoS management is left for future work.

4 Graph Building Algorithm

Taking a user graph as input, the algorithm for automatically creating a distributed flow graph for media playback works as follows. As a first step, the given URL of the source node is mapped to a node description. This description is then used as request for the registry service. Therefore, as much information as possible is added to the node description, i.e. the node type is set to "source" and specific interfaces or formats are added. For example, the interface "IFileSource" is added in case a "file" URL is given. For URLs such as "audiocd" the fixed outgoing format of an audio CD is specified. This allows for decoupling concrete implementations of nodes from their abstract representation. In addition, the node description is specified to be directed to the host given by the location of the URL (compare Figure 2). If more than a single node matches the given criteria, the node that "best" matches is finally requested, e.g. if only a single outgoing format is available that is identical to the requested.

As a second step, the complete node descriptions of possible sink nodes for rendering audio and video are queried from the registry service by using the node descriptions as given by the user graph. Sink nodes are not yet instantiated to avoid the creation of nodes that might not be needed to complete the construction of the flow graph. Notice that the node descriptions specified by the user graph may only be subsets of complete specifications. In our example, these descriptions only include interfaces for an audio and a video sink and the different locations of these two nodes. In contrast, complete node descriptions returned by the registry include all supported formats. This information is needed for the following steps of the algorithm.

In Figure 3(a), the result of these two steps is shown

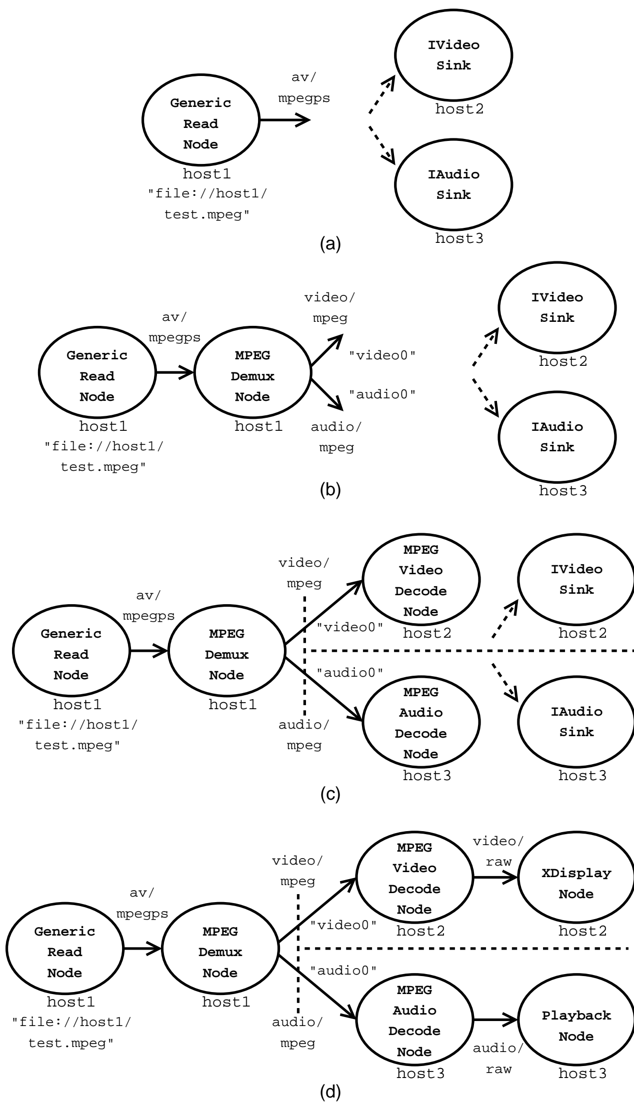

Figure 3. Automatic creation of a distributed flow graph for media playback for the user graph shown in Figure 2.

for the user graph given in Figure 2. In this case, a source node that allows to read files from the local file system was created. Since this node allows to read *any type* of file, it first needs to process some data in order to further determine the format of its outgoing data stream. This is done by testing for well-known patterns in the byte stream. In our example, an MPEG2 file is read and therefore the output format is set to "av/mpegps". Notice that no further information, such as the number of MPEG audio or video streams, is available at this stage. The used NMM framework guarantees that all data read in this step is cached within the node and kept until the node is connected. Then, the previously emitted buffers are forwarded first.

The main loop of the graph building algorithm works as follows. First, the algorithm tries to match the output format of the current upstream node (the "left" node) with the possible input formats of the sink nodes (the "right" nodes) as given by the node descriptions queried from the registry. For the first iteration of the loop, the upstream node is the source node. If the test is successful, the corresponding sink node is instantiated and both nodes are connected. If no more upstream nodes are available, the algorithm terminates. For example, when a graph URL that refers to an audio file in WAV format is given, the algorithm will find a *WavReadNode* as source that can be directly connected to an audio sink, e.g. the one called *PlaybackNode*.

However, if no connection is possible, the algorithm will try to find a matching downstream node. In this step, the output format of the current upstream node is used to create a new request in form of a node description that holds this format as input format. First, the registry is queried for converter nodes. Out of the list of all matching plug-ins returned, a converter that supports the exact format as given by the specified output format of the current upstream node is preferred over a converter that only includes this format within its input properties.

This decision can also be extended by using a look-ahead strategy that examines possible output formats of all matching converters *before* creating the actual connection. However, since nodes are not connected, only possible output formats can be tested – the actual outgoing formats are first available upon processing the first few buffers. Still, this look-ahead strategy allows to quickly identify the converter that potentially can be connected directly to an available sink node. According to this ranking, the "best" matching downstream converter node is then connected to the current upstream node.

For example, when automatically creating a flow graph for MP3 playback, an *MPEGAudioDecodeNode* will be inserted because its input format matches the output format of the source node *and* its possible output format can be intersected with one of the available input formats of the audio sink – only the actual parameters, such as the sampling rate, are not yet determined.

Since the user graph for media playback can contain three different location – for the source, the audio sink, and the video sink – the "optimal" distribution of intermediate nodes is determined by the algorithm. The heuristics used aims at minimizing the required networking bandwidth: Whenever the upstream part of the currently built graph is located on another host as the possible downstream part *and* the total estimated outgoing bandwidth of the next node to be inserted is much bigger than its total incoming bandwidth, the newly created node is tried to be instantiated at the location of the downstream part; otherwise at the location of the upstream part.

The intuition behind this idea is that decoding elements typically decompress data, i.e. the total incoming bandwidth is much smaller than the total outgoing bandwidth. Therefore, a decoding element should be located on the same host as further processing elements, such as sink nodes. The heuristics helps to reduce the needed networking bandwidth by collocating nodes that are connected with a link that requires higher bandwidth, and distributing sub-

graphs where connecting links require lower bandwidth.

The relationship between incoming and outgoing bandwidth of a node can be determined by previously performed measurements that are stored within the registry or simply by examining the type of input and output formats of a node: If the output format is of type “raw” (e.g. “audio/raw” or “video/raw”) and the input format is of a different type, the outgoing bandwidth is higher.

If no matching converter node was found, the algorithm tries to insert a demultiplexer using the same approach. This is the case for our example as shown in Figure 3(b), where the node *MPEGDemuxNode* is connected. Since its outgoing bandwidth is not bigger than its incoming, the node is collocated with the source node. The algorithm then continues the main loop by first testing all upstream nodes to be connected to possible sink nodes. Again, if this step cannot be completed successfully, additional converters or demultiplexers are tried to be inserted.

As can be seen from the demultiplexer in Figure 3(b), the algorithm further needs to process several possible outputs per node. However, since an multiplexed MPEG2 stream might contain several streams for one output media (e.g. different audio streams in MPEG or AC3 format) and additional streams (e.g. an SPU stream for subtitles), the algorithm needs further guidance. Therefore, the user graph also needs to include stream names for connecting the source to the sink nodes. By specifying name “audio0”, the first audio track will be chosen, which might be either the first MPEG audio stream named “mpeg_audio0” or the first AC3 audio stream named “ac_audio0”. Correspondingly, a stream name “video0” is given in the user graph for connecting to the video sink.

Figures 3(c) and (d) show the further steps of the algorithm. In (c), two decoding plug-ins are connected to the demultiplexer. In this case, the algorithm operates in a breadth-first manner, i.e. all outputs of demultiplexers are connected first. In (d), the sink nodes are instantiated and connected because the outgoing formats of the decoding nodes are found to be matching. Notice that for all these steps, multimedia data needs to be streamed and examined by newly connected processing elements in order to determine the precise output format, e.g. before the precise output format of the *MPEGAudioDecodNode* including parameters, such as the sampling rate, can be determined, several data buffers need to be decoded first.

Since different locations for the source node and the two sink nodes were specified within the user graph (see Figure 2), the resulting flow graph is split into three parts by applying the above described heuristics: The demultiplexer is collocated with the source node, decoding elements are collocated with corresponding sink nodes. Using this distribution of nodes, only the required MPEG streams are transmitted from the demultiplexer to the converters. In particular, additionally available MPEG streams, such as further audio tracks, are discarded within the demultiplexer.

Upon successfully creating the distributed flow graph, all sink nodes are connected to a component that handles the inter-stream synchronization of the media playback. Using NMM, this also works across the network with sufficient accuracy [2].

4.1 Parallel Branched Search

A discussed above, a node typically needs to process some incoming buffers before it can determine its precise output format. Since this information is needed to proceed with the algorithm, graph building can only work step-by-step; only a look-ahead of a single step can be performed. As a consequence, a decision in early stages of the graph building process might lead to unresolvable conflicts in terms of non-matching formats later on. While this was not the case in our example, such situations occur for more complex user graphs or when more than a single converter needs to be included per stream, e.g. a node for adapting the colorspace or resolution of a decoded video stream.

Therefore, *backtracking* can be performed. More precisely, the algorithm needs to discard already established paths within a flow graph, which do not provide a solution, and construct new paths for cases where more than one option exists. While different strategies for performing such a complete search exist, the integration of backtracking requires the ability to access data again that was already streamed within parts of the constructed flow graph. For live media, this requires to cache data within nodes during the construction of the flow graph – an option already supported by NMM.

However, as a further approach, *parallel branched search* can be used, i.e. all possible sub-graphs are constructed and data is streamed through all these sub-graphs in parallel. In the end, one solution is chosen, and all other solutions are discarded. This idea is also directly supported by NMM: Outgoing ports can be “cloned”; the same data is then forwarded to all ports. While the additional resources required for the parallel instantiation of sub-graphs are only needed during the setup phase, the approach helps to reduce the setup time of flow graphs – a requirement identified to be important for automatic application setup.

For both approaches – backtracking and parallel branched search – the corresponding algorithms need to identify possible loops during the graph building process. When creating chains with more than one processing element, elements that revert a previously performed operation are not allowed to be included. By applying these extensions, more complex user graphs can be handled.

5 Results and Applications

The described middleware service is already extensively used within several applications. The networked multimedia home entertainment system presented in [3] employs this service to set up flow graphs for playing back all supported media, e.g. CDs, specific DVD chapters, TV record-

Setup					Setup Time	
App	Src	A	V	Net	Manual	Auto
PC1	PC1	PC1	PC1	-	**0.53**	**0.57**
L	L	L	L	-	**1.40**	**1.55**
L	L	PC1	PC2	LAN	**1.26**	**1.39**
L	PC3	PC1	PC2	LAN	**1.50**	**1.52**
L	L	PC1	PC2	WLAN	**2.82**	**4.73**
L	PC3	PC1	PC2	WLAN	**3.10**	**3.54**

Table 1. Measured runtimes in seconds for the manual or automatic creation of distributed flow graphs for playing back MPEG2 audio/video files. Different PCs (PC1-3), a laptop computer (L), and different networking technologies are used (no network, LAN, or WLAN).

ings, and stored audio and video files in various formats.

In addition, a "follow-me" scenario is realized that allows a user to distribute media playback to nearby devices. In particular, a mobile web-tablet like laptop computer can be used to access media files that are stored locally or within the network. Furthermore, remote media sources, such as TV receiver can be transparently accessed. In all cases, audio and video output can either be performed locally on the laptop or by devices in the current environment, e.g. a high-fidelity stereo system and large screen connected to systems embedded in the current room. When the user moves to another location, the graph building algorithm is started again using an updated user graph.

Notice that using NMM as underlying framework, the application – and therefore also the user – has full control of distributed devices, e.g. for changing the volume of a remote audio device.

For evaluating the performance of the described middleware service, the setup times for the common task of MPEG2 audio/video playback are measured, either for the manual or for the automatic creation of flow graphs. Since the used MPEG2 file is taken from a DVD and provides high bitrates for the audio and video streams, it represents the "worst case" scenario. Table 1 summarizes the results using different setups. PC1, PC2, and PC3 are commodity computers connected via 100 MBit LAN. The laptop computer (denoted as L) provides a 700 MHz Pentium III and uses a 11 MBit WLAN connection; LAN is only used for comparing results. Depending on the used setup, the application (App), the media source (Src), the audio output (A), or the video output (V) are located on different hosts and connected using different networks (Net).

The setup times needed for the automatic creation are equal or only little higher than for the manual creation. Only in cases when WLAN is used, measured times differ significantly. This is because the graph building algorithm requires more interaction with distributed entities, such as the registry service, or more network traffic between connected nodes. Still, the results are quite promising.

6 Conclusions and Future Work

In this paper, we presented an algorithm for automatically setting up distributed multimedia applications. Starting from a given high-level description that only includes media sources and sinks and their locations, our approach determines required intermediate processing nodes and their optimal distribution. If more than a single option is available during the creation of the flow graph, the proposed concept of parallel branched search additionally helps to reduce setup times. We demonstrated the applicability and the performance of the implemented service with a "follow-me" application that automatically performs media output using devices embedded into the current environment of the user.

Future work will concentrate on including Quality of Service requirements into the algorithm. This will allow for automatically adapting media streams, e.g. when streaming content to mobile devices with scarce resources.

References

[1] Heribert Baldus, Markus Baumeister, Huib Eggenhuissen, Andras Montvay, and Wim Stut. WWICE – An Architecture for In-Home Digital Networks. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2000.

[2] Marco Lohse, Michael Repplinger, and Philipp Slusallek. Session Sharing as Middleware Service for Distributed Multimedia Applications. In *Interactive Multimedia on Next Generation Networks, Proceedings of First International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2003*, 2003.

[3] Marco Lohse and Philipp Slusallek. Middleware Support for Seamless Multimedia Home Entertainment for Mobile Users and Heterogeneous Environments. In *Proceedings of International Conference on Internet and Multimedia Systems and Applications (IMSA)*, 2003.

[4] Zhuoqing Morley Mao, Hoi-sheung Wilson So, and Byunghoon Kang. Network Support for Mobile Multimedia using a Self-adaptive Distributed Proxy. In *Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2001.

[5] Mark D. Pesce. *Programming Microsoft DirectShow for Digital Video and Television*. Microsoft Press, 2003.

[6] Sun Microsystems. *Java Media Framework API Guide*, JMF 2.0 FCS edition, 1999.

[7] Dongyan Xu and Klara Nahrstedt. Finding Service Paths in a Media Service Proxy Network. In *Proceedings of Multimedia Computing and Networking (MMCN)*, 2002.