SOPA - A Self Organizing Processing and Streaming Architecture

Gerald Friedland and Karl Pauls
Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik
Takustr. 9, D-14195 Berlin, Germany
{fland, pauls}@inf.fu-berlin.de

Abstract

This paper describes SOPA, a component framework that is an essential part of the lecture recording system E-Chalk. It envisiones a general processing and streaming architecture featuring autonomous assembly of stream processing components. The goal is to provide an easy to use framework where dynamically organized processing graphs are build out of components from various distributed sources. Based on state-of-the-art solutions for component based software development the system simplifies the implementation and the configuration of multimedia streaming applications and associated tools. It supports stream synchronization transparently while extending components are installed on the fly according to the existing requirements that may change at any time.

Contents

1. Introduction

This paper envisiones a general processing and streaming architecture and describes an implementing project, called SOPA, featuring autonomous assembly of stream processing components. The goal is to provide an easy to use framework where dynamically organized processing graphs are build out of components from various distributed sources. Components are installed on the fly according to the requirements of the processing graphs. Requirements are a result of the purpose of a specific graph assembled according to the needs of an application which may change at any time.

SOPA currently focuses on multimedia processing and Internet streaming applications on either server and/or client side. On the server side (e.g., on a video streaming server) SOPA integrates and manages codecs according to the capabilities of the connecting clients at runtime. Furthermore, reconfigurations or updates are supported seamlessly and transparent to the client. The idea is to dynamically adapt stream processing to user demands. For each specific demand a special processing graph is assembled using components discovered in the local framework and/or in remote depositories, respectively. In the latter case, required components are retrieved from their remote source and locally deployed. Examples of applications in need of a Self Organizing Processing and streaming Architecture range from extensible multimedia servers such as video and/or audio streaming systems over more specialized systems like lecture recording applications and streaming clients to rapid prototyping of, for example, filter chains.

The scientific goal for this research is to ease the development pains of applications in need for an extensible streaming and processing layer while decreasing administrative maintainance workload. More specific, to provide a round-up solution that serves as an extensible framework for managing of multimedia components (i.e., plugins). Another motivation is to enable synchronization of different independent (multimedia) streams such as slides and video streams and to propose a program independent notion to describe the handling of a concrete content, for example, to convert from one multimedia format into another.

The next Section introduces and discusses the underlying concepts and technologies of this work before a more detailed description of SOPA is given in Section 3. Section 4 presents related work and Section 5 reports on some concrete usage. Finally, Section 6 presents future work followed by the conclusion.

1.1. Example Usage Scenario

Consider university lecture recording: It is common to combine software defined radio with video and/or slides for on-the-fly streaming and recording of university lectures. However, combinability between standard Internet streaming systems is missing, for example, if one wants to combine a video conferencing system with a whiteboard presentation system. Unfortunetaly several streams are often combined by just starting different applications [9]. Of course, no control of synchronization then exists between the streams. Another way of synchronizing several streams is to have a postprocessing step using metadata, like SMIL [49], for manual synchronization (see for example [28]). The problem of not being able to automatically synchronize different independent media streams at application level has been proposed an open research question [34]. Even though most commercial multimedia streaming systems provide an API (Section 4), they are not only complicated to use but also too specialized and propietary. A common subset, like a compatibility layer is missing. Updates of codecs often force the costumers to re-install certain components (sometimes they do not even know about). Introducing a new medium results in administration work and also in an update of clients and associated tools [12]. The reminder of this paper presents a system that eases the development of compatible codecs and filters in a community, makes it easy for system administrators to use and to query for given codes, and aims towards changing the server configuration when a client connects instead of requiring to download a plugin.

2. Underlying Concepts and Technologies

Component orientation is a current trend for creating modern applications. The concept of a component is broad and includes plugins and other units of modularization. In general, component models and systems employing component-oriented approaches all define a concept similar to a component, e.g., an independently deployable binary unit of composition. Figure 1 depicts the application architecture envisioned in this paper. In the general model, a possible application is build on top of a component framework and SOPA, which in turn uses the component framework as a plugin-mechanism and a search-engine as a means of deployment-mechanism.

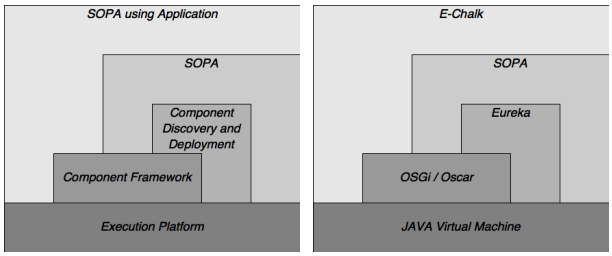

Figure 1. SOPA Architecture and Framework

In SOPA, any streaming application is a self organizing flow graph. The graph consists of six basic types of nodes: generic, sources, targets, forks, mixers, and pipes. Technically, any component that wants to live inside the graph becomes a node by inheriting one of these six superclasses. The graph that glues these components is described by an XML file. Nodes are described via general properties, for example, to tell the system to select some audio codec that compresses down to a certain bandwidth. The system then looks for an appropriate codec, first locally and then at remote locations. The structure of the graph can be changed and the nodes can be updated while the system is running.

This Section introduces a specific Java based model (see Figure 1) that currently supports underlying the multimedia platform E-Chalk. The implementation targets Oscar, a free implementation of the OSGi framework, as the SOPA underlying component model. Furthermore, Eureka, a Rendezvous based component discovery and deployment engine, that deploys a means of service discovery in order to provide component discovery, is used. The reminder of this sections introduces this aforementioned technologies.

2.1. OSGi

The Open Services Gateway Initiative (OSGi) [22] framework and service specification, was defined by the OSGi Alliance to dynamically deploy, activate, and manage service-oriented applications. The original intention was to create a specification for services that can be remotely deployed onto home network gateways, like set-top boxes and DSL-Modems, and into other networked environments, like

cars and telephones. The second release introduced methods for improving security, remotely managing service platforms, and making it easier to implement complex applications on top of it. This included a whole set of defined standard services, such as a log and an HTTP service. Therefore, other services may build on to that foundation of provided functionality. Release 3 [48] includes support for mobile service platforms and applications where data access is handled by a variety of secure interfaces. The number of services that are defined by the standard has increased by a significant number and several functions have been added in order to enhance the basic specification to simplify the deployment of OSGi based applications.

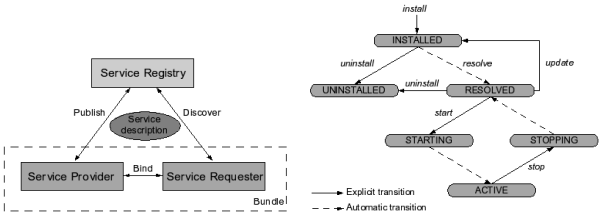

Figure 2. OSGi Service Registry (left) and Bundle Life Cycle (right)

2.2. The OSGi Framework

The OSGi framework, which sits on top of a Java virtual machine, is an execution environment for services. The OSGi framework was originally conceived to be used inside restricted environments. OSGi can however be used in other domains, as for example, an infrastructure to support underlying release 3.0 of the eclipse IDE [47].

The OSGi framework defines a unit of modularization, called a bundle, that is both a deployment unit and an activation unit. Physically, a bundle is a Java JAR file that contains a single component. The framework provides dynamic deployment mechanisms for bundles, including installation, removal, update, and activation (see Figure 2).

After a bundle is installed, it can be activated if all of its Java package dependencies are satisfied; package dependency meta-data is contained in the manifest of the bundle JAR file. Bundles can export/import Java package to/from each other; these are deployment-level dependencies. The OSGi framework automatically manages package dependencies of locally installed bundles, but it is not able to remotely discover bundles. After a bundle is activated it is able to provide service implementations or use the service implementations of other bundles within the framework. A service is a Java interface with externally specified semantics; this separation between interface and implementation allows for the creation of any number of implementations for a given service interface. When a bundle component implements a service interface, the service object is placed in the service registry provided by the OSGi framework so that other bundle components can discover it. All bundle interaction occurs via service interfaces (see Figure 2). When a bundle uses a services, this creates an instance-level dependency on a provider of that service. The framework defines no mechanisms for managing service dependencies.

2.3. Oscar

The work introduced in this paper is implemented and tested on top of the Open Service Container Architecture (Oscar) [20], an Open-Source implementation of the OSGi specification [38]. Therefore, it is not limited to this particular OSGi framework implementation but should be deployable to any standard conform OSGi framework.

The goal of Oscar is to provide a compliant and completely open OSGi framework implementation. The work on the Oscar project started in December 2000 by the founder of Oscar, Dr. Richard S. Hall. Technically, the OSGi service framework can be boiled down to a custom, dynamic Java class loader and a service registry that is globally accessible within a single Java virtual machine. The custom class loader maintains a set of dynamically changing bundles that share classes and resources with each other and interact via services published in the global service registry. In the last release 1.0 in Mai 2004 Oscar is almost fully compliant with the OSGi specification Release 1 and 2 although there are some minor issues [21]. Furthermore Oscar 1.0 is largely compliant with the OSGi framework specification Release 3. Most noticeably the ability to dynamically import packages and an initial implementation of the URL Stream Handlers service is already included.

2.4. Eureka

Discussions of network server communication pattern, especially discussions of peer-to-peer approaches, implicitly make an analogy between component discovery and file sharing. Ultimately, a component is a file or a collection of files, which leads to this analogy, but perhaps a different analogy would lead to a different solution. Conceptually, components can be viewed as service providers, which makes it interesting to investigate technological solutions that target service discovery in networks, rather than file sharing mechanisms.

Apple Computer, Inc. has recently focused attention on network service discovery. Apple realized that the Domain Name System (DNS)[35] address resolution protocol and its associated server infrastructure provided a convenient, well-tested, and ubiquitous infrastructure for advertising and querying for network services. However, to use

the DNS infrastructure for discovering network services, an approach was needed to generalize DNS from publishing domain names and querying for IP addresses to publishing and querying for network services, such as printers. To facilitate this, Apple defined DNS-based Service Discovery (DNS-SD)[39] that describes a convention for naming and structuring DNS resource records for discovering a list of named instances of services using standard DNS queries. The DNS infrastructure and DNS-SD create an effective approach for service discovery in wide-area networks, but there is also a need to discover services in local, ad-hoc networks where DNS servers are not present. For this, Apple defined Multicast DNS (mDNS)[40], which defines a way to perform DNS queries over IP Multicast in a local-link network without a DNS server. The combination of DNS-SD and mDNS forms the basis of Apple's Rendezvous technology.

Rendezvous [3], also known as Zero Configuration networking, is an open protocol that enables automatic discovery of computers, devices, and services in ad-hoc, IP-based networks. Due to mDNS, Rendezvous does not require a DNS server in the local-link network, nor does it require devices to have statically configured IP addresses. Participants in the Rendezvous protocol must be able to dynamically allocate an IP address without the aid of a Dynamic Host Configuration Protocol (DHCP) server, translate between device names and IP addresses without a DNS server, and advertise or locate services without a directory server. To do this, a participant first chooses an arbitrary link-local address (i.e., IPv4 address range 169.254.0.0/16) using a protocol to avoid address clashes. After a participant has chosen a link-local address, it is free to choose an arbitrary name in the .local domain, which is the domain that signifies the local link. Participants can then publish services under their local-link domain name using DNS resource records in the fashion specified by DNS-SD. Finally, participants may query for available services in the local-link domain using standard DNS queries via mDNS. In response to an mDNS query and depending on the type of query, participants advertising a given service will respond with its local-link name or address. When Rendezvous is combined with the standard DNS infrastructure, it provides an effective mechanism for service discovery in both local-link and wide-area networks

The DNS/Rendezvous infrastructure also has features that fit well with the requirements of a resource discovery service. For example, clients of a DNS/Rendezvous-based resource discovery services only produce network traffic in those situations where they actually make a query. Since clients of the service are not servers, like in a peer-to-peer network, they are simple and do not pay a network or computational cost when not using the service. Also, the DNS infrastructure is federated and allows cooperation among multiple servers.

Eureka [27] is a network-based resource discovery service to support deployment and run-time integration of components into extensible systems using Rendezvous' DNS-based approach. In Eureka's resource discovery service resource discovery servers are federated, where additional servers can be set up and connected to an existing resource discovery network or used as the root of another one. Publishing and discovery of components can be performed in both wide-area and local-link (i.e., ad-hoc) networks. Component providers can submit components to existing resource discovery servers, so it is not necessary for them to maintain their own server. Clients do not need to know the specific server that hosts a given component to discover it. Domain names under which components are registered provide an implicit scoping effect, as suggested by Rendezvous (e.g., a query for printers under the scope inf.fu-berlin.de produces a list of printers in the computer science department of the Free University Berlin).

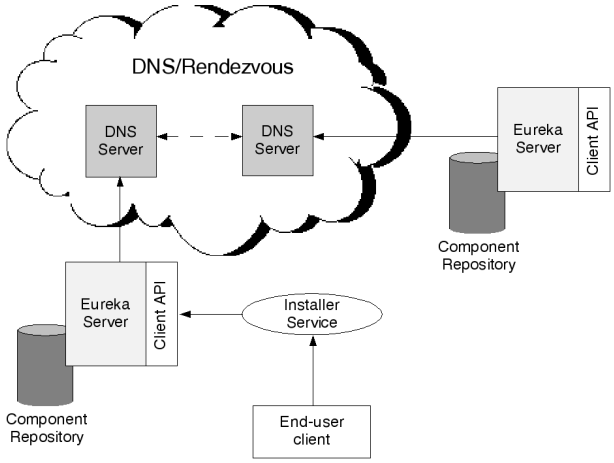

Figure 3. Eureka Architecture

Figure 3 is a conceptual view of the Eureka architecture. Each Eureka server has an associated DNS server, whose resource records the Eureka server can manipulate. A Eureka server has a client application programming interface (API) that provides access to its functionality. The client API allows clients to publish components, discover available components, and discover other Eureka servers. There are two options for publishing components. The provider can use the client API to submit the component meta-data and an URL from which the component archive file is accessible or the provider can submit the component meta-data and the component archive file itself, which the Eureka server will store in its component repository. In this latter case, Eureka uses its own HTTP server (not pictured in the Figure) to make the submitted component archive file URL accessible. Component discovery occurs in the DNS/Rendezvous

cloud of the Figure, which represents the unified local-link and wide-area networks accessible through mDNS and standard DNS, respectively.

Given the promise offered by the DNS/Rendezvous infrastructure discussed, Eureka adopted this technical approach. However, not all of Eureka's requirements for component discovery were addressed by this approach; in particular, some issues still needed to be resolved. Namely, how to map component meta-data to DNS resource records, how to allow either existing DNS servers (i.e., those involved with Internet name-address resolution) or arbitrary DNS servers (i.e., those not involved with name-address resolution) to participate in the same Eureka network, how to allow component providers to publish components into a Eureka network without having to maintain their own DNS server, and how to create a federated network out of the individual servers offering components, so that once a client has an entry point they can then discover other servers participating in the Eureka network.

As mentioned above, DNS-SD describes a convention for naming and structuring DNS resource records for discovering a list of named instances of services using standard DNS queries. A similar mechanism is needed for DNS-based component discovery, referred to in this paper as DNS-CD, for purposes of analogy. DNS-CD defines how to map component meta-data onto the three different types of DNS resource records: PTR, SRV, and TXT. As a model, the DNS-SD approach for service discovery uses SRV records to describe the location of a service, DNS PTR records to list all available instances of a particular service type, and DNS TXT records to convey additional service meta-data; for example, available queues on a printer. Similarly for component discovery, DNS-CD uses PTR records for describing export/import of libraries/services, SRV records for publishing available discovery scopes (defined in the next subsection), and TXT records to contain additional component meta-data.

As discussed previously, a Eureka server has an external DNS server associated with it; standard DNS server implementations are freely available and by leveraging them Eureka avoids a duplication of effort. This use of standard DNS server implementations is a first step toward using existing (i.e., involved in Internet name-address resolution) or arbitrary (i.e., not involved in Internet name-address resolution) DNS servers. As part of this first step, Eureka needs a way to externally configure standard DNS server implementations. Fortunately, most DNS server implementations use the DNS Dynamic Update Protocol[36], which allows clients to dynamically manipulate information stored on the server. This protocol is used by Eureka to add, edit, and remove DNS resource records that represent component meta-data.

It is important for Eureka to allow providers to publish components without maintaining there own server, since maintaining network servers is a potential security risk for less knowledgeable users. For this reason, publishing components to existing Eureka servers will probably be the preferred way to provide components for most providers. To achieve this, a scope hosted by a server can be either open or closed. An open scope allows arbitrary providers to publish their components into that scope, i.e., onto the hosting server. A closed scope requires a user name and password to publish components into that scope. For an open scope, a provider can use the client API to submit the component meta-data and an URL from which the component archive file is accessible or the provider can submit the component meta-data and the component archive file itself, which the Eureka server will store in its component repository. In this latter case, Eureka uses its own HTTP server to make the submitted component archive file URL accessible. Since components are published by arbitrary providers, like web pages, it is likely that component references will eventually get stale and will no longer be accessible. To deal with this situation, Eureka provides a garbage collection mechanism for component meta-data. A Eureka server periodically checks whether all components referenced by the meta-data in its associated DNS server are accessible via their given URL. If a component can not be accessed, its meta-data is removed from the server.

To ensure that a query for a given resource is found, no matter which entry point into the Eureka network a client is using, Eureka must define a way to interconnect Eureka servers by propagating new scopes to existing servers. As it turns out, DNS provides a mechanism for distributing resource records to other DNS servers. A DNS server that is responsible for a specific domain can be configured as the master of that domain, while other servers may act as slaves that maintain copies of the resource records. A slave server periodically checks for changes on the master server and, when changes are found, applies those changes to its copy of the resource records. These mechanisms are well understood and have been in use for a long time in DNS servers. Using this mechanism it is possible to make every DNS server that takes part in an Eureka network aware of all other scopes by creating a special scope at the root server of a given Eureka network; the root server acts as the master of that scope. When other servers join the Eureka network, they are configured as a slave for that special scope. This then propagates all scopes to the other servers. Clients of the DNS servers are then free to query this special scope at their DNS server to discover what scopes are available. New scopes are added to the master server by using the DNS controller of the associated Eureka server.

3. SOPA

This Section describes SOPA's main concepts. It introduces the notion of the media graph and the media node, explains how the graph is resolved, and how stream synchronization is achieved.

3.1. The Media Graph

The underlying structure of SOPA is a directed, cycle-free flowgraph. The graph consists of both, so called media nodes and virtual nodes. The nodes can be of six different types: sources, targets, pipes, mixers, forks, and generic. The data flow is organized as a push stream from the sources to the target. The reason for this is, that multimedia PC hardware mostly enforces a push paradigm. Source nodes read from an external source, like a special device, a file, or from a network and push the data to another node. Pipe nodes get data from a single source, usually process them, and push them further to other media nodes. Mixers get their data from several nodes, mix them, and push them to single further nodes. Forks split a stream using a certain strategy to pass it to several nodes. Targets receive a stream and push it to external targets. Generic nodes can be of any type, their application field is either wildcard usage or for building unconnected nodes. A virtual node is an LDAP query that may point to a media node that has not yet been found. Besides the multimedia stream that flows between the media nodes events are also passed through.

A media graph is in one of three states: defined, resolved, or active. In the defined state the graph only consists of virtual nodes, in other words a set of LDAP queries. In the resolved state the graph is resolved as described in Section 3.4. When a graph is resolved, at least one path exists from a source to a target, where all LDAP queries have been evaluated to match certain media nodes and the input and output formats have been set to each media node in a way, that they build a processing chain. If a media graph description led to a resolved graph the active state is reached by first initializing all non-sources of valid pathes. Then the sources are activated in order to start delivering data. A path of the media graph is deactivated by stopping its source. An event is then propagated that no further data is available, which makes the remaining nodes of the path shut down, too. After all activated path have shut down, the media graph gets back to resolved state.

3.2. A Media Node

Each media node is a service as defined by the OSGi standard. It has a name and a version that identifies it uniquely. A set of properties as well as a preference ordered list of processable formats. All properties of a media node are defined inside the class. There is no need to create an extra file containing these metadata. Media nodes are also configured via properties. Inter node communication is done via property change events. All media nodes are notified whenever a property changes. There are several predefined properties for certain events. For example, when new media nodes are initialized or pathes are started.

A node is in one of three states: constructed, initialized, or running. In the constructed state a node is constructed by first instantiating the class and then calling the `start()` method as required by OSGi. Either the node is a `BundleActivator` and Oscar calls this method or a so called NodePropagator is used to do this for several nodes at once. In the initialized state a node is initialized after the media graph has been resolved and is to be startet. During initialization, a node has to prepare everything for the immidiate receive of data. This state is introduced for synchronization. Finally, in the running state, after initialization pipes, targets, mixers, and forks immidiately receive data from their predecessing nodes. Sources are started using the `startWork()` method. If `stop()` is called on a media node, the node goes back to initialized state.

3.3. Identifying Media Formats

Media Formats are distinguished by so called format descriptors. The mechanism has been taken from the open source project Exymen, [15] discusses it in detail. SOPA provides some default implementations of format and content descriptors that supply several standard methods typically used to describe media content, such as the average framerate, the duration of an individual frame, the name, and space coordinates of a frame, and, most important, the FormatID. Exymen uses FormatIDs as a mechanism to uniquely identify media formats, because file extensions are ambiguous and unreliable. Magic bytes in headers are not always used and sometimes it is unclear how to read them, since they tend to be machine dependent (for example, Little and Big Endian representations). MIME types [29] are not sufficiently different because their primary intention is to define a mapping between format and application, not the classification of format types.

Exymen's FormatIDs work differently: Compatible formats get the same FormatID even though they may be read-in differently. For example: There may be several ways to store PCM (pulse code modulation) audio data, however, a wav-file handler plug-in will not have any problem handling PCM audio data even if it comes from an aif-file. Formats that are basically different but are handled by some API are put in the same class. For example, a node that uses Microsoft's Windows Media SDK [57] can handle all audio CODECs supported by the Audio Codec Manager (ACM). A lists of the FormatIDs defined so far is available at [52].

3.4. Resolving the Media Graph

A graph is resolved in two main steps. In the first step, SOPA tries to match the LDAP queries, in other words it tries to find concrete media nodes that match the virtual nodes. The query is matched upon the properties that each media node propagates. Media nodes are searched locally and in the Internet using Eureka. If no node is found, the regarding path cannot be resolved. If several nodes are found they are stored as a list of alternatives considered in the next step. In the second step, a list of media nodes belongs to each virtual node. SOPA now tries to create a processing chain by substituting each virtual node by a media node that input and output format matches best. Since the format list is preference ordered, best fit is defined as the minimum index in the list. If there are several choices, the source's output format and the target's input format is considered more important than the format preferences of other nodes. If there is still an ambiguity, newer versions of media nodes are preferred.

3.5. Synchronization

SOPA uses the notion of progress constrained threads. These are threads that dynamically depend on other thread's progress. The progress is measured in steps. Threads have to request allowance for progress by requesting to proceed to the next step at a central clearance manager. To avoid deadlocks, steps must increase monotonically. If a thread depends on another one, it has to wait until that other thread requests the same step. A progress constrained thread object signals the clearance manager that it has progressed to a certain step. If it depends on other progress constrained threads that have not come to this step yet, the clearance manager blocks until these other threads request progress for the same or a higher step. Each progress constrained thread implements a method that is called by the clearance manager to determine if this progress constrained thread depends on some other progress constrained thread in a certain step. The dependencies have to be constant for a certain step, but may change in further steps dynamically. The idea of progress constrained threads is derived from the well known barrier synchronisation, described in detail in [46]. The difference being, that there are multiple barriers and the barrier exists only for those threads that have to wait on other threads.

A set of arbitrary media nodes can be grouped into a synchronisation group in the media graph description. These nodes are then synchronised using a given time granularity (the default is 100ms). This is done by blocking the path to its ancestor node while a given media node has processed more data then the others, considering their frame rates. Nodes can be added or removed from a synchronization group at runtime.

3.6. SOPA in Action

```
<SOPA:script xmnls:SOPA='http://sopa.inf.fu-berlin.de'>

<service label="source"
         match="(&(&(author=Jantz)(version>=1))(outputs=*RGB*))"
         type="&source;"
         target="display">
</service>

<service label="display"
         match="(&(&(author=Jantz)(version>=1))(name=TVPipe))"
         type="&pipe;"
         target="blackhole">
</service>

<service label="blackhole"
         match="(name=BlackHoleTarget)"
         type="&target;">
</service>

</SOPA:script>
```

Figure 4. A simple Graph Description

The initial media graph is specified in an XML file. The file contains a set of node descriptions. Each node is described by a temporary label (that can be choosen freely), its type, and an LDAP query. Figure 4 shows an example. For developers there is also a shell and a simple graphical node editor (Figure 5). The shell gives access to Oscar and Eureka functionalities such as installing and publishing bundles as well as to a few Java debugging features. A given, dynamically evolved, media graph can also be serialized into a new XML graph description. Once an initial configuration has been found and debugged, devolpers published their bundles of media nodes using the Eureka system. The only thing that has to be deployed is the XML graph description and SOPA itself (optionally the bundle cache can also be deployed, so that an end user does not need to have an Internet connection).

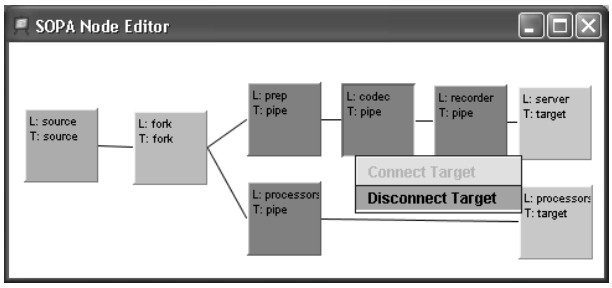

Figure 5. The SOPA Graphical Editor

At any time, a media graph can be restructured by a media node that uses the SOPA API or by loading another XML graph description. Both graph descriptions or media nodes restructure a given graph by adding virtual nodes in the form of LDAP queries. Media nodes that are not running can be removed or replaced dynamically. Nodes in the running state cannot be replaced or removed. However, an

active path can be connected to by connecting a media node to a fork or a mixer. Forks and mixers can handle new connections even when in the runnning state.

Every Node or the media graph manager inherits from *org.osgi.BundleActivator*. In this way, Oscar takes care of the component administration but is hidden to media node developers that do not want to fiddle with the OSGi system. The OSGi aware developers can just overwrite the *start()* and *stop()* methods in media node and can access the whole functionality that Oscar provides. This is especially interesting if you want to access some of the standard OSGi services.

4. Related Work

Media frameworks include Indiva, HaVi, Audio Toolbox in MacOS X, and the Java Media Framework. Indiva [50] stands for INfrastructre for DIstributed Video and Audio and is based on the Open Mash project. It is a middleware layer for a unified set of abstractions and operations for hardware devices, software processes, and media data in a distributed audio and video environment. These abstractions use a file system metaphor to access resources and high-level commands to simplify the development of Internet webcast and distributed collaboration control applications. It uses soft state protocols for communication between the individual processes. Indiva focuses very much on distributed programming. The smallest elements indiva handles are processes, not classes. The configuration is pretty easy but static. There are no automatic stream synchronisation mechanisms. HAVi [53] stands for Home Audio Video Interoperability and is a standard for networking home entertainment devices defined by several major electronics companies. It specifically focuses on the transfer of digital audio/video (AV) content between HAVi devices, as well as the processing (rendering, record, play back) of this content by these devices. HAVi provides Java API for stream management and device controls. It shares some goals with Indiva (see above). Havi has similarities to Jini [54], both aim to create a network of devices which users can easily connect and operate. There is also a HaVi-Jini bridge. However, HAVi is targeted at consumers home audio/video network and dictates the use of Firewire as a transport mechanism. It aims towards connecting hardware devices and is a protocol layer on top of IEEE1394. Apple's MacOS X operating system's audio core contains a package called Audio Toolbox [2]. Inside this, one can find the AUGraph API. An AUGraph is a high-level representation of a set of so called AudioUnits, along with the connections between them. AudioUnits are used to generate, process, receive, or otherwise manipulate streams of audio. They are building blocks that may be used singly or connected together to form the audio signal graph. Information to and from AudioUnits is passed via properties. AudioUnits are identified by a string based, Apple propietary, hierarchic identification mechanism. One can use the API to construct arbitrary signal paths through which audio may be processed, i.e., a modular routing system. The API deals with large numbers of AudioUnits and their relationships. AudioGraphs allow realtime routing changes, that means connections can be created and broken while audio is being processed. Apples API comes very near to what is described in this paper. However, the API is restricted to audio and - although available in Java - can only be used in Mac OS X. There is also no concept for self configuration or distributed programming. Sun Microsystems delivers a quite general and platform independent framework that hides the implementation details of several media formats: the Java Media Framework (JMF) [55]. JMF is a Java API that supports capture, playback, streaming and transcoding of audio, video, and other time-based media. It also provides a plug-in architecture that enables developers to support custom data sources and sinks, effect plug-ins, and CODECs. JMF supports a wide variety of different specifications, including various sound formats, MPEG-1 (ISO/IEC 11172), H.261 (ITU-T Recommendation H.261, 1990), H.263 (ITU-T Recommendation H.263, 1998), Macromedia Flash, and others. Although their plug-in loading mechanism can load classes at runtime, they do not offer package dependency checking or automatic updating, as one would expect from a well defined component management. It is therefore not suitable as a base for a dynamically configurable system. The main disadvantage, however, is its complexity. JMF consists of 11 packages of 212 different classes and interfaces. Recording audio from the default input without any volume control, requires about 20 lines of code and the knowledge of these classes: `AudioFormat`, `TargetDataLine`, `DataLine`, `DataLine.Info`, `AudioSystem`. Converting a file from one format into another, involves about 700 lines of code [56] and at least the knowledge of the following classes: `ContentDescriptor`, `MediaLocator`, `DataSink`, `SinkListener`, `TrackControl`, `Format`, `QualityControl`, `Processor`. It has to be remarked, that these classes implement propietary concepts that may sound familiar to the reader, but they are deduced from the properties and technological restrictions of the supported hardware and the implemented formats. One example that illustrates this hypothesis is, that JMF does not support synchronizing two content streams of different formats. A top down approach would make life much easier for the application programmer.

File sharing systems oriented around the peer-to-peer approach include Napster, Gnutella, JXTA, and JXTA Search. The primary goal of Napster [30] was to enable file sharing among clients with a special focus on music files. In

truth, Napster was something in-between a client-server and a peer-to-peer architecture due to its use of a centralized registry. The concept used resembles that of a service-oriented framework. The main advantages of the Gnutella protocol [13] are based on its simplicity. Due to the fact that only a few message types exist through which peers communicate with each other and that the payload of queries is undefined, the protocol is easy to implement and makes no assumptions about the resources shared. Due to its simplicity, there are numerous clients available for the Gnutella protocol, such as LimeWire[10]. The Gnutella protocol does have drawbacks, as discussed in [25]. Mainly, Gnutella has high network traffic overhead due to its lack of effective traffic avoidance mechanism. Sun Microsystems has founded an open community project called JXTA (short for Juxtapose)[45]. The goal of the project is to allow a wide range of applications to make use of distributed computing and targets some of the limitations found in many peer-to-peer systems. On the whole, JXTA introduces some new concepts to peer-to-peer, most noticeably the ”peer group” concept and the possibility of context-based querying for services. One of the available peer-to-peer applications built on top of JXTA is JXTA Search. The idea of this project is to use server peers with which other peers can register as capable of answering queries for a certain query-space. A query-space is a topic under which a query belongs and all queries made to a server node must commit themselves to a query-space. JXTA Search is not limited to queries from inside the JXTA framework, but a server node can be queried via SOAP[51] as well. The real downside of this approach is that server nodes must be interconnected by hand.

Over the last decade service sharing systems, including the Service Location Protocol (SLP)[24, 23], Jini [26], the CORBA Trader[31], and more recently UDDI [4], have gained attention. SLP is an IETF standard for resource discovery of devices that offer network-based services to other devices. SLP defines a protocol that enables clients to find each other in a network and provide services to each other. One of the concepts applied in order to structure the services offered in SLP is the notion of scopes. The purpose of the scope is to provide scalability, limiting the network coverage of a request and the number of replies. SLP focuses on the local-link, not wide-area networks. Developed by the Object Management Group (OMG) the Common Object Request Broker Architecture (CORBA)[32] is a reference implementation to aid the development of distributed object-oriented applications. The CORBA Trader provides the possibility of discovering instances of services of particular types. The concept used resembles that of a service-oriented framework. A special registry object, called a trader, supports the trading of objects in a distributed environment. Traders form a federated system that may span many domains. CORBA, unfortunately, never gained a critical mass of acceptance, so CORBA traders are not ubiquitous like the DNS infrastructure. Universal Description Discovery and Integration (UDDI) is a specification for distributed registries that allow publishing and discovery of web services. A core concept in UDDI is business publication, where a business is published into the UDDI registry based on a description of the business and the web services it provides. UDDI is a closed federation of registries, where new registries can only be added by existing members. In this federation, all registries replicates all published information, which means that information that is published at one registry is propagated to all other federation members. The downside of this closed federation approach is that it is not simple for arbitrary third-party providers to participate in the federation. Jini is a distributed Java infrastructure that provides mechanisms for service registration, lookup, and usage. Jini does not provide sophisticated deployment facilities and relies on the class loading capabilities of Java Remote Method Invocation (RMI)[44]. An interesting aspect is that services are leased to clients as a means of garbage collection. The downsides of Jini are that it is dependent on Java, and the Jini infrastructure is not ubiquitous on the Internet.

Component models [11] include COM[5], JavaBeans[43], EJB[42], and CCM[33]. EJB and CCM support non-functional aspects such as persistence, transactions, and distribution. Service-oriented platforms include OSGi, Jini, web services[14], and Avalon [1].

Gravity [37] is a research project investigating the dynamic assembly of applications and the impact of building applications from components that exhibit dynamic availability, i.e., they may appear or disappear at any time. Gravity is built as a standard OSGi bundle and provides a graphical design environment for building application using drag-and-drop techniques. Using Gravity, an application is assembled dynamically and the end user is able to switch between design and execution modes at any time. In regard to the dynamic configuration of an OSGi based system, SOPA and Gravity do have some similarities. Both focus on the dynamic, demand-driven aspect of applications. Subsequently, a promising approach could be to port this specific part of SOPA to Gravity’s service-binder. Unfortunately, Gravity does deploy a push based service binding, meaning that additional meta-data is used in order to specify dependencies of a service and the servicebinder takes care that dependencies are satisfied and binds those to the service at runtime. SOPA in comparison, relays on a pull based model, where services are requested when needed. Therefore, especially since Gravity does not have a means of specifying dependencies dynamically, a port would call for a major redesign of the very core of SOPA.

5. Validations and Project Status

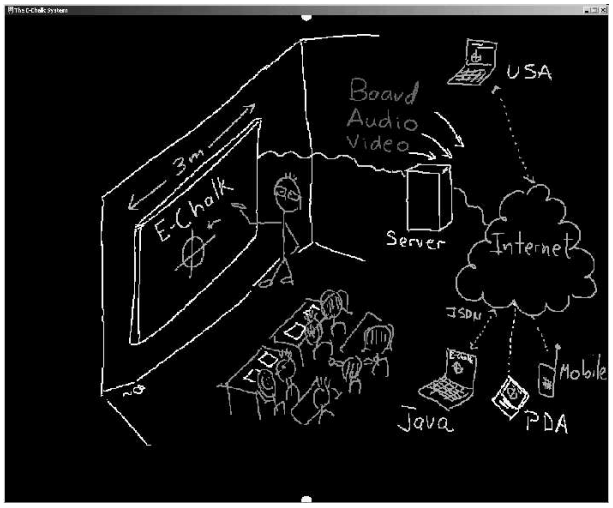

Figure 6. The E-Chalk System [19]

SOPA is an open API and is currently part of the E-Chalk system. E-Chalk is a software system for classroom teaching and distance learning. E-Chalk's philosophy is to provide a tool to enhance classroom lectures while distance teaching arises as a side effect, i. e. at no extra cost. The lecturer writes using special hardware that operates as an electronic chalkboard. The lecturer can seamlessly integrate pictures and interactive programs from the local hard-drive or directly from the Internet. The lecture can be followed in the lecture room and remotely through the Internet. The system transmits audio, video, the build-up of the board image, and optionally slides. All content is recorded and archived in such a way that only a Java enabled web browser is needed for later viewing. Figure 6 shows a sketch of the system in use. For a detailed description of E-Chalk see [18, 19].

Early versions of E-Chalk used the WWR system, see [17], to stream and archive the voice of the lecturer. This was similar to the common approach for on-the-fly streaming and recording of university lectures: software defined radio systems combined with video and slides discussed in Section 1.1. Using SOPA, E-Chalk streams can also be received with a Quicktime player or Windows Media Player. A receptor, a generic media node, waits for an incoming connection and checks its type. According to the type of the connecting client it restructures the given media graph. Downloaded media nodes are also used in Exymen [16], which is used for editing archived lectures because it shares the bundle cache with SOPA. A dedicated Exymen plugin is able to read in media graph descriptions to find out conversion pathes. This way, Exymen can edit any lecture, even if it was recorded in a new format. Before recording, the E-Chalk system uses an audio configuration wizard, measures the quality of the sound hardware used, monitors possible hardware malfunctions, prevents common user mistakes, and provides gain control and filter mechanisms. The result is a modified initial graph description, that contains a set of media nodes needed for pre- and postprocessing a given lecture recording.

6. Future Work

An important problem to solve in the future is to guarantee the realtime performance of the media graph. Up to now the system installs components on demand without any knowledge of how much CPU time is consumed. It is well possible, that the combination of certain components goes beyond the limits of the underlying computer system. The result is a denial of service. One solution is to let each developer implement a benchmark for his or her component which returns a value relative to components that come along with the framework. On the very first run of SOPA, a large benchmark using build-in components would be run to Figure out what the underlying system is able to handle. When running, SOPA would refuse to integrate further components into the graph if the sum of the benchmark results of the already integrated components exceeds the maximum.

Security is a concern in any environment that supports the execution of arbitrary dynamically downloaded code. From the point of view of safety, the environment must be protected from dynamically integrated components causing harm to the underlying resources, such as deleting files. From the point of view of privacy, the environment must be protected from components snooping or spying, such as inventorying all services being used. The OSGi framework uses a technique for dealing with security, namely executing dynamically integrated components within a security sandbox. The sandbox is used to stop unauthorized access to the underlying resources and to control the visibility of other installed services and resources. External declaration enables the creation of security policies that can assign defaults access rights to dynamically integrated components, as well as to assign different levels of rights to components from known sources using a public-key cryptography approach. The most difficult aspect of security is finding mechanisms that are very simple or support automated decision making, since the typical end user is not very knowledgeable about security-related issues. End-user involvement in security-related decisions should be kept to a minimum to avoid confusion and mistakes. Future work in this area targets the issue that especially in the SOPA vision, where the system is autonomously deployed, access-control related issues become almost unbearable to the End-user. Therefore, an access-control management system like Raccoon [8] should

be deployed in order to ease those pains. Raccoon provides access control in distributed object systems. The language in which access policies are specified is determined by the underlying access control model. An access control model specifically designed to support the design and management of access control policies in object-oriented systems is *view-based access control* (VBAC) [6, 7]. VBAC relies on roles as abstractions of callers and can thus be regarded as an extension of *role-based access control* (RBAC) [41] to distributed object systems (e.g. CORBA). The principal new feature of VBAC is that of a *view* as a static, typed language construct for the description of fine-grained access rights, which are permissions or denials for operations of distributed objects. Views are defined in the *View Policy Language* (VPL) as part of a policy design document, which is a product of the design stage in the development process. View-based access policies are type-checked with a language compiler. Access policies are delivered in descriptor files and deployed together with applications in the target environments, similar to approaches like EJB [42] or the CORBA Component Model [33]. Due to the fact that a lightweight OSGi-based version of Raccoon is currently under development future work will investigate the integration of Raccoon into SOPA.

Eureka uses a network of interconnected DNS servers that advertise component meta-data following the approach of Rendezvous from Apple. During the time following the first release Eureka's architecture has been reworked in order to support various networks. Networks introduce themselves to Eureka in form of plugins. Future work in this area includes the investigation and integration of different sources for SOPA. Until today, SOPA's architecture assumes that it can rely on a pre-configured trader, in this case Eureka, but with the new open architecture any peer-to-peer system for example could be used in order to allow anybody to provide extensions to existing SOPA-using applications. This would be a blessing as well as a curse since by the amount the number of available extensions increases the reliability of the system is decreasing, especially in the case where extensions from arbitrary providers are used. Additionally, available extensions are filtered by the aforementioned notion of a scope. Regarding Eureka, scopes correspond directly to domains in the DNS. It is unclear how to map those scopes to other distributed system architectures, which do not support scope like structures. Furthermore, even in the case of the DNS-based Eureka, the difference between a location-scope and a semantical-scope needs some investigation. A scope is a mean of denoting a location under which component meta-data can be found. From a semantical perspective it might be desirable to allow the denotation of different branches inside a location as well, like for instance in todays concurrent version systems (e.g., CVS). A possible, already feasible way to achieve this goal is to use sub-domains that represent branches inside a location but sub-domains can not be created dynamically in DNS. Subsequently, a different approach is needed that enables the usage of branches inside a scope (i.e., the actual zone of the scope).

7. Conclusion

SOPA is a framework that manages multimedia processing and streaming components organized in a flow graph. Components are discovered from remote repositories and integrated autonomously during runtime. Stream synchronization is handled transparently to the implementor. Based on state-of-the-art solutions for component based software development the system simplifies the implementation of multimedia streaming applications and associated tools.

The SOPA architecure derives from the natural and common way of designing multimedia streaming applications, namely, building a flow graph. This is enhanced by a framework that consist of a component managment system combined with a discovery and deployment engine. Therefore, plugin management, discovery, and deployment is already taken care of. The assembly of the graph is managed by SOPA. Developers concentrate on the development of multimedia components. Administrators only have to specify their wishes in a simple XML file containing LDAP queries.

SOPA is currently an essential part of the current version of a commercial application, the E-Chalk system, which is in use in about a hundred institutions. Research projects are also known to work with it.

References

[1] Apache Organization. *The Avalon Framework*. http://jakarta.apache.org/avalon, 2004.

[2] Apple Computer, Inc. *Audio and MIDI on Mac OS X*, May 2001.

[3] Apple Computer, Inc. *Rendezvous*. Official Web Site, http://developer.apple.com/macosx/rendezvous/, May 2004.

[4] Ariba Corp., IBM Corp., and Microsoft Corp. *UDDI Technical White Paper*. Technical Report, http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, September 2000.

[5] Box, D. *Essential COM*. Addison Wesley, 1998.

[6] G. Brose. A typed access control model for CORBA. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Weidner, editors, *Proc. European Symposium on Research in Computer Security (ESORICS)*, LNCS 1895, pages 88–105. Springer, 2000.

[7] G. Brose. *Access Control Management in Distributed Object Systems*. PhD thesis, Freie Universität Berlin, 2001.

[8] G. Brose. Raccoon — An infrastructure for managing access control in CORBA. In *Proc. Int. Conference on Distributed*

Applications and Interoperable Systems (DAIS). Kluwer, 2001.

[9] J. A. Brotherton. *Enriching Everyday Activities through the Automated Capture and Access of Live Experiences*. PhD thesis, Georgia Intitute of Technology, July 2001.

[10] C. Rohrs. *LimeWire Design*. http://www.limewire.org/project/www/design.html, August 2001.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[12] T. O. Ch. Bacher, R. Mueller and M. Will. Authoring on the fly. a new way of integrating telepresentation and courseware production. In *Proceedings of the International Conference on Computer in Education*, Malaysia, December 1997.

[13] Clip2. *The Gnutella Protocol Specification*. Version 0.41, Document Revision 1.2, 2003.

[14] Curbera, F., Nagy, A., and Weerawarana, S. *Web-Services: Why and How.* in Workshop on Object-Oriented Web Services (OOPSLA, August 2001.

[15] G. Friedland. Towards a generic cross platform media editor: An editing tool for e-chalk. Master's thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, May 2002.

[16] G. Friedland. Towards a generic cross platform media editor: An editing tool for e-chalk (abstract). In *Proceedings of the fourth Informatiktage 2002, Bad Schussenried*. Gesellschaft für Informatik e.V., November 2002.

[17] G. Friedland, L. Knipping, and R. Rojas. E-chalk technical description. Technical Report B-02-11, Fachbereich Mathematik und Informatik, Freie Universität Berlin, May 2002.

[18] G. Friedland, L. Knipping, J. Schulte, and E. T. a. E-chalk: A lecture recording system using the chalkboard metaphor. *International Journal of Interactive Technology and Smart Education*, 1(1), February 2004.

[19] G. Friedland, L. Knipping, and E. Tapia. Web based lectures produced by AI supported classroom teaching. *International Journal of Artificial Intelligence Tools*, 13(2), 2004.

[20] R. S. Hall. *Oscar Community*. Official Web Site, http://oscar-osgi.sourceforge.net, 2004.

[21] R. S. Hall. *Oscar Issues*. Official Web Site, http://oscar-osgi.sourceforge.net/issues.html, 2004.

[22] R. S. Hall. *OSGi Alliance*. Official Web Site, http://www.osgi.org, 2004.

[23] Internet Engineering Task Force. *Service Location Protocol*. RFC2608, 1999.

[24] J. Govea and M. Barbeau. *Comparison of Bandwidth Usage: Service Location Protocol and Jini*. Technical Report TR-00-06, School of Computer Science Carleton University, October 2000.

[25] J. Ritter. *Why Gnutella Can't Scale. No, Really*. http://www.tch.org/gnutella.html, 2003.

[26] K. Arnold et al. *The Jini Specification*. Addison-Wesley, 1999.

[27] Karl Pauls and Richard S. Hall. Eureka - A Resource Discovery Service for Component Deployment. In *Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004)*, Mai 2004.

[28] T. C. Mingchao Ma, Volker Schillings and C. Meinel. T-cube: A multimedia authoring system for elearning. In *Proceedings of E-Learn*, pages 2289–2296, Phoenix, Arizona, USA, November 2003.

[29] N. B. N. Freed. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt, November 1996.

[30] Napster, LLC. *Official Web Site*. http://www.napster.com, January 2004.

[31] Object Management Group, Inc. *Trading Object Service Specification, Version 1.0*. http://www.omg.org, 2000.

[32] Object Management Group, Inc. *Common Object Request Broker Architecture: Core Specification*. Version 3.0.2, December 2002.

[33] OMG. *CORBA 3.0 New Components Chapters, TC Document ptc/99-10-04*. OMG, Oct. 1999.

[34] T. Ottmann. Presentation recording. In W. S. C. Haythornthwaite and G. Vossen, editors, *Conceptual and Technical Aspects of Electronic Learning*, Dagstuhl, Germany, May 2003. Schloss Dagstuhl, International Conference and Research Center for Computer Science, Seminar No. 03191.

[35] P. Mockapetris. *Domain Names - Concepts and Facilities*. RFC 1034, November 1987.

[36] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. *Dynamic Updates in the Domain Name System (DNS UPDATE)*. RFC 2136, April 1997.

[37] R.S. Hall and H. Cervantes. Gravity: Supporting Dynamically Available Services in Client-Side Applications. In *Poster paper in Proceedings of ESEC/FSE 2003*, September 2003.

[38] R.S. Hall and H. Cervantes. An OSGi Implementation and Experience Report. In *Proceedings of IEEEConsumer Communications and Networking Conference*, January 2004.

[39] S. Cheshire and M. Krochmal. *DNS-Based Service Discovery*. Internet Draft, http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt, February 2004.

[40] S. Cheshire and M. Krochmal. *Multicast DNS*. Internet Draft, http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt, February 2004.

[41] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role–based access control models. *IEEE Computer*, 29(2):38–47, Feb. 1996.

[42] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0, Final Draft*, Oct. 2000.

[43] Sun Microsystems, INC. *JavaBeans Specification*. Version 1.0.1, 1997.

[44] Sun Microsystems, INC. *Java Remote Method Invocation*. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html, 2003.

[45] Sun Microsystems, INC. *JXTA v2.0 Protocols Specification*. Revision 2.1.1, October 2003.

[46] A. S. Tennenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, February 2001.

[47] The Eclipse Foundation. Eclipse Platform - Technical Overview. Technical report, Object Technology International Inc., February 2003.

[48] The Open Services Gateway Initiative. *OSGi Service Platform*. IOS Press, Amsterdam, The Netherlands, March 2003. Release 3.

[49] W3C. Synchronized Multimedia Integration Language (SMIL 2.0). http://www.w3.org/TR/smil20/, August 2001.

[50] P. P. Wei Tsang Ooi and L. A. Rowe. Indiva: Middleware for managing a distributed media environment. Technical Report 166, Berkeley Media research Center, 2000.

[51] World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework*. W3C Recommendation, June 2003.

[52] Exymen Project Homepage. http://www.exymen.org.

[53] Home Audio Video Interoperabilty. http://www.havi.org.

[54] The Community Resource for Jini Technology. http://www.jini.org.

[55] Java Media Framework. http://java.sun.com/products/java-media/jmf/index.jsp.

[56] Java Media Framework Sample Code. http://java.sun.com/products/java-media/jmf/2.1.1/samples/samplecode.html#Export.

[57] Microsoft Windows Media - 9 Series SDK. http://www.microsoft.com/windows/windowsmedia/9series/sdk.aspx.