Chapter 4

Server Architecture

A software system that is to automatically integrate different types of content streams also needs an architecture that fundamentally supports this. This chapter provides a conceptual overview of the architectural approach of the E-Chalk Server system. Further details can be found in [Friedland and Pauls, 2004], [Friedland and Pauls, 2005a], [Friedland and Pauls, 2005b], and Appendix B.

4.1 Preliminary Considerations

As could be seen in the discussions of the previous chapter, schools and universities are a heterogeneous playground. A software system that wants to achieve sustained success in more than a few institutions has to be able to survive in an environment consisting of different software and hardware configurations. It should be able to adapt to different software ideologies (e. g., it should not interfere with political discussions about operating systems). The software has to fit into the existing hardware infrastructure and should readily combine with other multimedia applications. Structural modifications, such as the addition of new media, changes in technical formats, upgrades to new hardware, or functional enhancements should not cause tremendous administrative overhead. A teacher must be able to step into the classroom and start lecturing as usual. Reliability should also be considered. It is important to be able to continue working when individual parts of the system fail, at least on the level of switch-over and/or backup facilities.

In the beginning, E-Chalk basically consisted of three monolithic servers. The chalkboard simulation and server, the audio server, and the video server. The three servers were started simultaneously by the *E-Chalk Startup Wizard* (see [Friedland et al., 2002]), a GUI wizard that is shown on startup to handle the configuration of the E-Chalk server before the beginning of a lecture. Figure 4.1 illustrates the setup.

As E-Chalk was being used in more and more universities, the heterogenity of different university hardware and software prerequisites required more and more special solutions. A common case was the following: After the initial introduction of some lecture recording system, professors wanted to do chalkboard-based lectures while the students and the infrastructure of the university had been

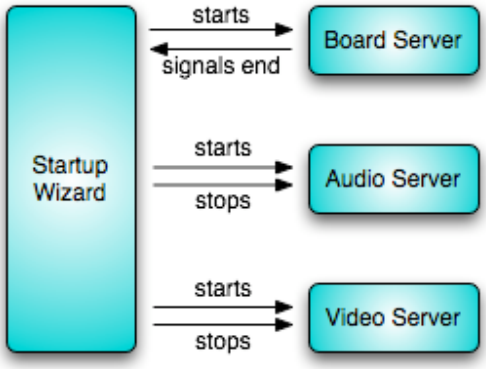

Figure 4.1: An overview of the old architecture of the E-Chalk server: The E-Chalk Startup Wizard starts the monolithic audio, video, and board servers.

optimized to a lecture recording system that only supported slide-show presentations. E-Chalk had to fit into already established software configurations and work flows in different departments and subject areas. For example, E-Chalk had to be combined with other universities' lecture recording systems or with commercial Internet broadcasting systems. Users wanted to use different codecs than those that were built into the system. The early monolithic architecture did not allow proper integration into many different systems. Any update of the software system did not only require a manual patch of the source code, it also required a complete re-installation by the administrators of the "client university". Lectures, archived in different formats than the built-in E-Chalk formats, could not be edited using Exymen. Last but not least, E-Chalk is a research project and thus constantly underlies changes. This forced us to create an architecture that would be able to provide us with both system stability and the possibility of a rapid integration of new ideas.

Even though most commercial multimedia streaming systems provide extensibility through an SDK (see Section 2.3), these are not only complicated to use but also too specialized and proprietary. A common subset, like a compatibility layer, is missing. Updates of codecs often force the customers 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, see for example [Bacher et al., 1997].

This chapter presents a system called *SOPA: Self-Organizing Processing and Streaming Architecture* that was build as a reaction to these different demands [Friedland and Pauls, 2005b]. The system eases the development pains of applications in need for an extensible streaming and processing layer while decreasing administrative maintainance workload. It tries to provide a round-up solution that serves as an extensible framework for managing software components (sometimes also called plug-ins). The system allows the synchronization of different independent streams, such as slides and video streams and proposes a format-independent notion to describe the handling of a concrete content, for example, to convert from one multimedia format into another. SOPA encourages the development of compatible codecs and filters in a community, makes

it easy for system administrators to use and to search for available codes, and supports changing the server configuration when a client connects instead of requiring the client to download a plug-in. Exymen also builds on SOPA, which allows it to edit content that was created with newly introduced codecs.

4.2 Existing Multimedia Architectures

Although none of the following architectural approaches could be directly used as a bottom layer architecture for E-Chalk, many concepts of E-Chalk's server architecture had already been introduced in a similar way before. This section therefore provides a short survey on the existing solutions for multimedia systems and explains their differences to SOPA.

Indiva [Ooi et al., 2000] 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 individual processes. Indiva focuses very much on distributed programming. The smallest elements Indiva handles are processes, not classes. The configuration is simply static and there are no automatic stream synchronization mechanisms.

MacOS X's audio core by Apple contains a package called *Audio Toolbox* [Apple Inc, 2001]. Inside the toolbox, one can find the *AUGraph* SDK. 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 isolated 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, proprietary, 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. The API is restricted to audio and – although available in Java – can only be used on Mac OS X. The AuGraph SDK has no concept for self configuration or distributed programming.

Microsoft Direct Show [31] is a component architecture for multimedia streaming and processing which is part of Direct X which is a support package for the Windows operating systems. Direct Show features dynamic assembly of stream processing graphs. However, Direct Show contains no layer on the administrative end, i. e., assembly has to be done in source code. DirectShow is not platform independent and does not feature a remote discovery mechanism for components.

Sun Microsystems delivers a quite general and platform independent framework that hides the implementation details of several media formats: the *Java Media Framework (JMF)* [84]. 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. The architecture is deduced from the properties and technological restrictions of the supported hardware and the implemented formats. 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 demands discussed above are very similar to the problems that have to be solved by the manufacturers of digital TV set-top boxes. The difference being that the software inside a set-top box can often rely on predefined hardware. *HAVi* [80] 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 a Java API for stream management and device control. However, HAVi is targeted at consumers home audio/video network and dictates the use of *Firewire* (IEEE-STD-1394-1995) as a transport mechanism. It aims towards connecting hardware devices and is a protocol layer on top of IEEE 1394. Standards similar to HAVi also exist from ISO (International Organization for Standardization) and ETSI (European Telecomunication Standards Institute). ETSI standardized an open middleware system called *Multimedia Home Platform (MHP)* which is part of the Digital Video Broadcasting (DVB) specifications [72]. ISO's *Home Electronic System (HES)* aims to "standardize software and hardware so that manufacturers might offer one version of a product that could operate on a variety of home automation networks" [Milutinovic, 2002].

Although not specifically a multimedia architecture, a related component-assembly system is *Gravity* [R.S. Hall and H. Cervantes, 2003]. Gravity is a research project investigating the dynamic assembly of applications and the impact of building applications from components that exhibit dynamic availability, i. e., components may appear or disappear at any time. Gravity provides a graphical design environment for building applications 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. The architecture presented here is driven by the same idea, but specializes on stream processing. Gravity, however, implements automatic service binding, meaning that additional meta-data is used in order to specify dependencies of a service. The service binder makes sure that component dependencies are satisfied and binds the services provided by the components automatically to the application. The architecture presented here relies on a model, where services are bound upon user request. For this reason, Gravity cannot directly be used because it does not have a means of specifying dependencies dynamically by the user.

4.3 Architecture Overview

In the proposed architecture, the end-user application is built on top of a component-assembly mechanism, which in turn uses a component framework as a plug-in mechanism and a component search-engine as a means of deployment-mechanism. Figure 4.2 shows both, the general model of the architectural

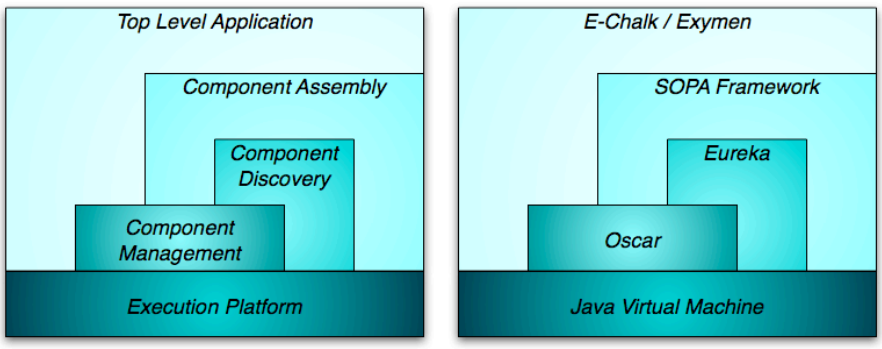

Figure 4.2: Conceptual diagram of the proposed service-oriented architecture for multimedia applications (left), concrete implementation in the E-Chalk server system (right).

approach described here and E-Chalk's server architecture. The architecture is based on an execution platform – the operating system or a virtual machine. The component framework provides mechanisms for installing, updating, and deleting components. It also takes care of component dependencies and manages their lifecyles. E-Chalk uses the Java Virtual Machine by Sun Microsystems as execution platform. The Java Virtual Machine runs on a variety of hardware and operating systems and therefore provides platform independence at the price of a slight memory and execution performance overhead. On top of the virtual machine, *Oscar* is used as a component-management framework. Oscar is an open-source implementation of the OSGi standard [3]. Oscar manages the installation, update, and removal of special Java archives, so-called *Bundles*. Oscar also handles the lifecycle of Bundles as well as dependencies between them. Bundles can be searched in the Internet using *Eureka*, an *Apple Bonjour*-based component-discovery and deployment engine. SOPA manages the assembly of a set of specialized Bundles, so-called *media nodes*, to form one or more *media graphs*. A media graph connects different codecs, filters, or other stream-processing units in order to perform a certain operation. On top of the processing graphs, E-Chalk's Startup Wizard as well as the board application control the life cycle of the entire system. In the following sections, E-Chalk's concrete implementation will serve as an example of how the approach can solve several of the problems discussed in Section 4.1.

4.4 Java as Execution Platform

A thorough discussion of why Java may be favored against a native implementation would go well beyond the scope of this dissertation. This section only provides a brief summary of the relevant experiences that played a role when developing the E-Chalk system. The decision to use the Java Virtual Machine as the primary execution platform for E-Chalk was made very early in the project. The primary reason, back in the beginning of the project, was platform independence. Java allows to run the system on any available machine in the university. E-Chalk's capability to run on Windows, Linux, Mac OS, Mac OS X, or Sun OS

was actually one of the main reasons why other educational institutions found it easy to try out the system. A second important reason for using Java was the possibility to have one rendering engine for both the board server and the board client. The board client could be written as a Java Applet and the board server as a Java application. The same code is used for displaying content in the classroom and in the Web Browser [Raffel, 2000]. This approach did not only save development work, it also helped to minimize differences between the classroom view and the remote view. Slight differences due to varying hardware properties (such as screen resolution) and different virtual machine versions might still be perceived, though. Some computer scientist might object to using the Java Virtual Machine as an execution platform because of its poor performance – especially when it comes to audio and video processing. The Java developer community has been discussing and comparing the execution speed of Java programs versus native programs for several years now. The actual numbers vary, but even older articles (see for example [Prechelt, 2000, Shirazi, 2003, Mathew et al., 1999], [46, 52]) agree, that the speed penalty and memory overhead are tolerable when weighed against programming productivity. A disadvantage of using a platform-independent virtual machine is that special features of a certain system cannot be utilized. A platform-independent virtual machine can only implement a common subset of the functionality of the platforms it supports. Special features, like using the pressure sensitivity of digitizer tablets, are for example not part of the Java Virtual Machine. To get around this problem, native code had to be used sometimes and encapsulated using the so-called *Java Native Interface (JNI)*. Of course, native code has to be implemented individually for all targeted platforms. Java is a pure object-oriented language and supports dynamic class loading. This feature allowed to implement many of the board's dynamic functionalities, like integrating third-party Applets or Chalklets. It has also spawned many implementations of component-management frameworks. Last but not least, Java binaries are rather small. None of the components described in this chapter is larger than 1 MB. The next section will introduce Oscar, a component-management framework that was used as the underlying layer in E-Chalk.

4.5 The Component Framework

Component orientation is becoming increasingly popular in modern applications and is being more and more discussed for multimedia architectures [Nahrstedt and Balke, 2004, Nahrstedt and Balke, 2005]. The concept of a *component* is broad and includes plug-ins or other units of modularization. In this text, a software component is defined as "a binary unit of composition with contractually specified interfaces and explicit context dependencies only" [C. Szyperski, 1998]. The notion of component orientation is strongly connected to the idea of object orientation. Compositions of components are usually created by an *actor* (either the user or another software program) that instantiates some components through a managing framework. The instances are then appropriately connected to each other by this actor. Component models and frameworks include the *Component Object Model (COM)* [Box, 1998], *Java-Beans* [Sun Microsystems Inc, 1997], *Enterprise Java Beans (EJB)* [Sun Microsystems Inc, 2000], the *Corba Component Model (CCM)* [Object Management Group (OMG), 1999],

the OSGi standard, Jini [83], and *Avalon* [4]. EJB and CCM support non-functional aspects such as persistence, transactions, and distribution. OSGi, Jini, and Avalon are so-called *service-oriented* platforms. Service orientation shares the component-orientation idea in that applications are assembled from independent building blocks. However, the essential building blocks are not components but the services they are providing. In other words, a component can provide more than one service. A service is a functionality that is contractually defined in a *service description*, for example as a Java Interface. The idea of service orientation is that application assembly is based only on these service descriptions and the actual components are located and integrated into the application later, either prior to or during execution of the program. A more detailed discussion of component- and service-oriented programming can be found in [Cervantes and Hall, 2004].

The OSGi specification defines a framework that is an execution environment for services. Compared to related specifications the core framework is very small. The specification does not refer to too many concepts and the OSGi initiative is also trying to keep it compact since it is still being targeted to restricted environments, such as embedded devices. Implementations are therefore kept small and efficient. OSGi can, however, be used in other domains, for example, as a support-infrastructure underlying the *Eclipse IDE* [The Eclipse Foundation, 2003]. Even before Eclipse, Exymen had introduced the use of the OSGi specification on the desktop for building an extensible multimedia editing application [Friedland, 2002a]. Both Exymen and E-Chalk are implemented on top of the *Open Service Container Architecture (Oscar)* [Hall and Cervantes, 2004] [23], an Open-Source implementation of the OSGi specification [The Open Services Gateway Initiative, 2003], that has recently been renamed to *Felix*. However, the approaches are not limited to this particular OSGi framework implementation and should also be deployable to any other standard OSGi framework.

Oscar was created with the goal to provide a compliant and completely open OSGi framework implementation. Work on the Oscar project started in December 2000 by Richard S. Hall. Technically, the OSGi service framework can be seen as 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. Oscar is almost fully compliant with the OSGi specification release 1 and 2 and largely compliant with release 3. The OSGi specification is a document that contains more than 600 pages [The Open Services Gateway Initiative, 2003], therefore the next paragraph will summarize only the most important facts that are relevant to E-Chalk.

The OSGi framework defines a unit of modularization, called a *Bundle*. Physically, a Bundle is a Java JAR file that groups together all classes, together with their resources (native libraries, icons, help files), into a component. Archive attributes, among them the dependencies on other Bundles, are described in the JAR file's Manifest. Every Bundle contains a Java class that inherits from `org.osgi.BundleActivator`. A `BundleActivator` provides two methods: `start()` and `stop()`. The framework provides dynamic deployment mechanisms for Bundles, including installation, removal, update, and activation. Figure 4.3 shows that a Bundle can be in one of the following states:

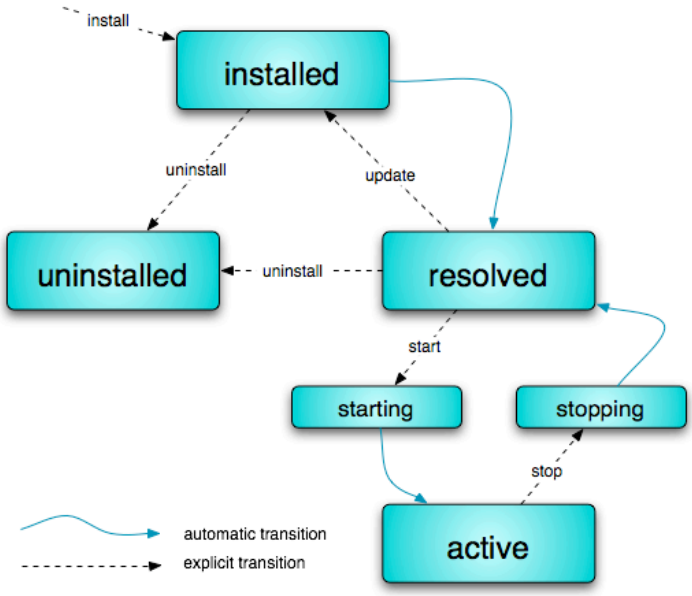

Figure 4.3: A state diagram of the life-cycle management provided by Oscar. Drawing after [The Open Services Gateway Initiative, 2003]

- *active* - the Bundle is now running,
- *installed* - the Bundle is installed but not yet resolved,
- *resolved* - the Bundle is resolved and is able to be started,
- *starting* - the Bundle is in the process of starting,
- *stopping* - the Bundle is in the process of stopping, or
- *uninstalled* - the Bundle is uninstalled and may not be used.

Oscar mainly provides the following operations to manage Bundles:

- `Install` a Bundle from a URL. The Bundle is downloaded and archived in a local repository. The Bundle enters installed state.
- `Start` a Bundle. If all dependencies of the installed Bundle can be resolved, the `start()` method of the Bundle is called and the Bundle enters the active state.
- `Stop` a Bundle. The `stop()` method of the Bundle is called and the Bundle changes to the resolved state.
- `Uninstall` a Bundle. This stops the Bundle and tags it as uninstalled. The Bundle is removed at the next refresh.
- `Update` a Bundle. This puts a new JAR file in place without refreshing it. The Bundle enters the installed state.

- `Refresh` a Bundle. Refresh causes all Bundles that depend on installed or uninstalled Bundles to stop. Oscar resolves any updated Bundles and then restarts them all, if possible, effectively creating new instances for every dependent Bundle.

After a Bundle is installed, it can be activated if all of its Java package dependencies are satisfied. Bundles can export and/or import Java packages to and/or from each other. The OSGi framework automatically manages package dependencies of locally installed 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. When a component implements a service, the service object is placed in the service registry provided by the OSGi framework so that other Bundles can discover it. When a Bundle uses a services, this creates an instance-level dependency on a provider of that service. When the top-level application (e. g. E-Chalk) is exited, the states of all Bundles are saved and restored at the next startup. This decreases start-up time by avoiding to resolve all Bundles again. Bundles can be uninstalled or updated while the application is running without ever requiring a restart of the application.

Using the diagram in Figure 4.3, one can see what happens if Bundle *A* is uninstalled while it is needed by another Bundle *B*: Bundle A stays physically where it is stored until the next refresh operation. At the refresh operation, A is deleted and B is stopped. Stopped Bundles cannot be used, but are still installed. If the user installs an update of Bundle A that B can use, B enters the resolved state and can be used again after the next refresh. A refresh command is invoked automatically after every install command by the SOPA framework.

4.6 Component Discovery

The OSGi specification allows the installation of Bundles from any URL. However, it is not able to remotely discover Bundles. In order to be able to query and locate Bundles and the services they are providing from remote locations, E-Chalk integrates the Eureka system.

Eureka [Pauls, 2003, Pauls and Hall, 2004] is a network-based resource-discovery service to support deployment and run-time integration of components into extensible systems. Eureka is based on Apple Inc's *Bonjour Networking* [5] technology, formerly named *Rendezvous*. Bonjour is an open protocol that enables automatic discovery of computers, devices, and services in ad-hoc, IP-based networks. The DNS/Bonjour infrastructure has features that fit well with the requirements of a component- or service-discovery service. For example, clients of a DNS/Bonjour-based resource-discovery services only produce network traffic when they actually make a query and they 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 (e. g., a query for components under the scope `inf.fu-berlin.de` produces a list of all components available in the computer science department of the Freie Universität Berlin). 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. A closed scope requires a user name and password. To submit a component, meta-data and an URL from which the component archive file is accessible is provided to Eureka. The developer may also submit the component archive file itself. Then, the Eureka server will store it in its component repository and use its own HTTP server to make the submitted component accessible.

Eureka also 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 cannot be accessed, its meta-data is removed from the server.

The E-Chalk server system uses the scope `sopa.inf.fu-berlin.de`. This results in each service being referenced as ⟨`eurekaid`⟩`.sopa.inf.fu-berlin.de`. The identifier ⟨`eurekaid`⟩ is generated by Eureka – with the exception of `www`, which is reserved for the website of the project. The meta-data for media nodes are automatically generated by the SOPA framework and consist mainly of a set of Java properties. The publishing and unpublishing of nodes is reduced to specifying a download URL and a node name (see Appendix B).

4.7 Component Assembly

The SOPA framework is based on Oscar and Eureka. It uses them to achieve its goals and provides several services on top of it. The next sections provide a detailed explanation of the SOPA framework as integrated into E-Chalk.

4.7.1 Processing Nodes

Building a graph that combines individual filtering units for stream processing appears in many systems and can be considered canonical (see Section 4.2). The basic units in SOPA are called *media nodes*. There are six basic types of nodes: generic, sources, targets, forks, mixers, and pipes. The conceptual difference between them is the semantics which is determined by the number of inputs and outputs.

- *Source nodes* have one outgoing edge and no incoming edges. A source node generates data or gets its data from anywhere outside the graph. This node is typically used for accessing sound devices, video cards, or for file readers.

- *Target nodes* only have one incoming edge and no outgoing edges. A target node acts as a sink: It takes the incoming data and writes it somewhere. This node is typically used for playback or for file writers.

- *Pipe nodes* inherit the properties from source nodes and target nodes. They have one incoming edge and one outgoing edge. This type of node is the most frequently used because it can be used to implement filters, converters, measuring devices, and many more.

- *Fork nodes* are pipe nodes that have two or more outgoing edges. They are helpful to branch the data flow in oder to have several processing chains for the same input.

- *Mixer nodes* unify two or more incoming processing branches into one. Consequently, they have several incoming edges and only one outgoing edge.

- *Generic nodes* have neither incoming nor outgoing edges. However, they can communicate with the rest of the nodes in the graph via events. Generic nodes are meant to be extended for different purposes. Their typical use is as a receptor that captures information from outside the graph and reacts by rebuilding the graph as necessary.

Technically, the nodes are defined as abstract classes inside the SOPA framework. The developer has to define the final semantics of each node by inheriting from one of the six superclasses. To use the framework, a developer has to learn only a limited number of concepts in order to create his or her own nodes. Appendix B shows the overhead required for implementing a `PipeNode`. Since the node acts as target and as a source, the methods of both types of nodes have to be overwritten. On other words, the overhead doubles for this type of node. Nevertheless, no more than ten methods have to be implemented by the developer, most of them being one-liners.

The framework already defines a set of standard nodes. Standard nodes include testing methods and default implementations for frequently-used functions. Examples include a pipe node that introduces bitwise noise into any stream that is passed through, a bandwidth delimiter, a traffic-measurement pipe, a compression pipe node that applies ZIP compression on any byte stream, a buffer pipe that caches any incoming data before it is passed through, a file-reading source, a source node that generates zeros as output, or a file-writing target. Several predefined nodes are useful for component assembly. The so-called `IdentityPipe` just outputs the incoming data, the `BlackHoleTarget` is a target node that discards all incoming data, the `GenericFork` copies any incoming data to all the outputs ports, and the `GenericMixer` node mixes the multiple incoming content into a single sequence on a first-in, first-out basis.

Every class that inherits from `MediaNode` is a service as defined by the OSGi standard. This way, Oscar actually takes care of the component administration but is hidden to developers that do not want to fiddle with the OSGi system. Every node has a name and a version that identifies it uniquely. It comes along with a set of properties as well as a preference-ordered list of processable formats. The formats are described using a `FormatDescriptor` as explained in Section 4.7.4. Every node class provides methods that describe the properties needed for graph assembly. There is no need to create any extra file that contains meta-data for a particular node.

Nodes are configured via Java properties. SOPA features a central property management system that builds upon Oscar's property management which in turn builds on Java's property management. Properties can be made persistent and can quickly be restored upon a restart of the system. The E-Chalk Startup Wizard also generates property files that are read in by SOPA. Any node's parameters, for example filenames or sampling rates, are set this way. Nodes communicate via property change events: Whenever a property changes, the property manager notifies all nodes. There are several predefined properties for certain events. Properties are set, for example, when new media nodes are initialized or new processing paths are started.

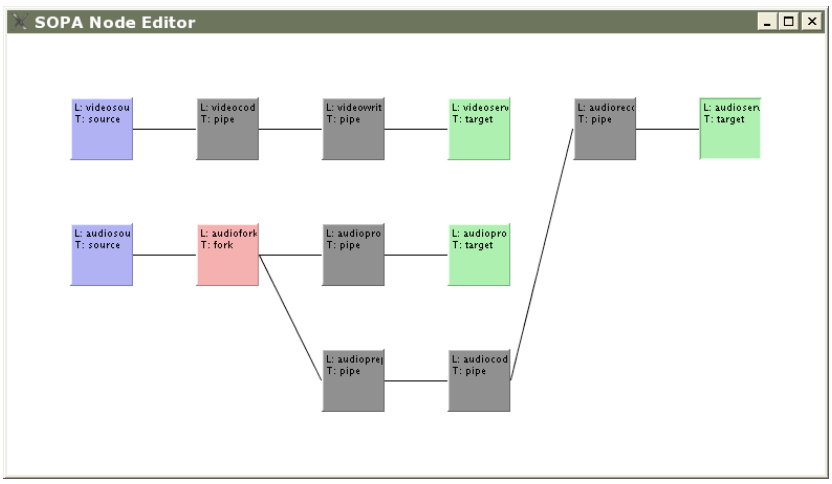

Figure 4.4: A screenshot of a visualized media graph inside SOPA's graphical node composition editor. The visualization is updated at runtime as the graph is updated. The editor can read and write XML graph serializations independent of the framework. The visualization shows a typical processing graph in the E-Chalk system containing independent audio and video paths.

At any time 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. Unlike the OSGi standard, in SOPA a service can be instantiated multiple times. This allows to use several instances of the same node in different locations of the graph. In the initialized state a node is initialized after the graph has been resolved and is to be started. During initialization, a node has to prepare everything for the immediate receive of data. This state is introduced for synchronization. Section 4.7.5 explains synchronization in detail. Finally, in the running state pipes, targets, mixers, and forks immediately receive data from their predecessing nodes. If `stop()` is called on a media node, the node goes back to the initialized state. This saves object disposal and construction (rather costly operations in Java), since a node may be reused in another graph location.

4.7.2 The Processing Graph

Component assembly is performed by arranging the media nodes into one or more directed acyclic graphs. In reality, the framework is able to handle several unconnected graphs at once, but for simplicity this text will refer to them as "the graph". The actual assembly and resolution algorithm of the graph is described in Section 4.7.3. Multimedia PC hardware, such as sound or video cards, mostly enforces a push paradigm (hard disks, however, are accessed using a pull paradigm). For this reason, data is flowing from a source node to a target node with the source nodes pushing the data.

The media graph can be created and changed in two ways. A media node can use the methods provided by the framework, or the framework itself can load a serialized version of the graph. The framework can load or serialize the structure of the graph at any time. The serialization is stored in a simple XML format

(see Appendix B). Additionally, the framework provides a graphical editor for visualizing and building graph descriptions (Figure 4.4 shows a screenshot) and a command-line console for developers. The shell gives access to Oscar and Eureka functionalities such as installing and publishing Bundles as well as to a few Java debugging features. Upon startup of the framework, the initial graph is always loaded from the XML description.

Each node is described by a temporary label (that can be chosen freely, or is assigned randomly by the framework), its type, and an LDAP query [Howes, 1996]. The framework searches for nodes matching the LDAP query, first locally in Oscar's Bundle repository and then remotely using Eureka. The specification of nodes using LDAP queries allows incomplete descriptions which enables system administrators to only specify the important properties and to include wildcards. Appendix B shows the grammar of the LDAP query language. The following listing shows some sample node descriptions.

```
<service label="source"
         match="(&(&(author=Friedland)(version>=1))(outputs=*RGB*))"
         type="&source;">
 target="display">
</service>
<service label="display"
         match="(&(&author=Friedland)(version>=1))(name=TVPipe))"
         type="&pipe;"
         target="sink">
</service>
<service label="sink"
         match="(name=BlackHoleTarget)"
         type="&target;">
</service>
```

To deploy a running media graph, it suffices to copy an XML graph description, the framework itself, and optionally a few property files. Usually a Bundle repository is also included, so that the end user is not required to have an Internet connection already at startup. Media nodes can be removed or replaced dynamically at any time if they are not in the running state. However, forks and mixers can handle new connections even when in the running state. Therefore an active path can be connected to by connecting a media node to a fork or a mixer. A media graph is in one of three states: defined, resolved, or active. In the defined state the graph only consists of node descriptions. In the resolved state the graph is resolved as described in Section 4.7.3. 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 such a way that they build a processing chain. If a media graph description led to a resolved graph, then the active state is reached by first initializing all non-sources of valid paths. Then the sources are activated in order to start delivering data. If there is an error during initialization of a node and there are still alternatives that also match the LDAP query, they replace the erroneous node. When running, sources continously push the stream of data through the pipes to the target. A path of the media graph is deactivated by stopping its source. An event is then propagated saying that no further data is available, which makes the remaining nodes of the path shut down, too. After all activated paths have shut down, the media graph gets back to the resolved state.

4.7.3 Resolving the Media Graph

The graph resolution algorithm is the core of the framework. Because of this importance, developers might want to change its behavior. SOPA's graph resolution algorithm can therefore easily be exchanged by third-party developers who want to provide their own resolution. Apart from a class that implements an interface with the new resolution method, a new resolution algorithm might also need its own serialization, which can easily be changed by creating a new XML DTD (and providing the methods for reading and writing the serialization). Several different resolution algorithms have been implemented. One of the question was whether the connecting edges have to be specified manually by the user. In the end, the following method seemed to be the most practical one. It became the default behavior in the E-Chalk system. The DTD shown in Appendix B is used by this resolution method, it has been extended for better user editability, see Section 4.9. In the beginning, the graph consists only of a set of connected node descriptions. The graph is then resolved in two main steps.

In the first step, the SOPA framework tries to match the LDAP queries. The query is matched to 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 node description. The framework now tries to create a processing chain by substituting each node description by the media node that matches best concerning its input and output format. Since the format list is preference ordered, best fit is defined as the minimum index in the list. 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 an ambiguity, newer versions of media nodes are preferred.

Technically, the following steps are preformed during resolution. The input is a serialization of the graph that already contains all edges.

1. Count the incoming and outgoing edges of all node descriptions. If a specified node type does not match the incoming or outgoing edges: throw the description out and notify the user.

2. Use union-find to separate unconnected graphs. The following steps are performed for each connected set of nodes.

3. Try to match each LDAP query using the properties defined in the system and by the nodes, both locally and in the defined Eureka search scopes.

4. Associate each set of matching nodes to the node description.

5. Find a path from a source to a target such that:

   - The sum of all preference-ordered format list indeces of each node's used format is minimal. The index numbers of the source node and the target node are each weighted twice.
   - If there is ambiguity, use the media node that has a higher version number.

- If there is still ambiguity, use the node that was found first (these are usually the local ones).

6. Test whether the path from source to target is complete. If yes, tag all nodes in the path as startable.

4.7.4 Identifying Media Formats

Data can only be exchanged between two nodes if they speak the same language, that is, if they work on data in the same format. Sometimes, it suffices to describe a format using basic data types. For example, a node that uses ZIP compression on any incoming data can work on a byte-per-byte basis. Usually, however, nodes need to exchange a more differentiated description of the structure of the data they have to deal with. The degree of provided detail and the way this structural information must itself be structured is difficult to standardize because it depends heavily on the format per se and how it is to be handled. In the end, if a totally different, new media format is to be handled, a group of node developers will have to agree on an appropriate description.

In SOPA, media formats are distinguished by so-called *format descriptors*. Format descriptors are actually Java Interfaces that provide get and set methods for certain format properties. The mechanism has already been established in Exymen, [Friedland, 2002a] discusses it in detail. The SOPA framework provides several default implementations of format and content descriptors. Examples include a generic descriptor for byte streams, a descriptor for uncompressed video formats and a descriptor for raw audio formats. Each descriptor supplies several standard methods typically used to describe media content such as the average frame rate, the duration of an individual frame, the name, time and space coordinates of a frame, and a so-called *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 by formats 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 [Freed and Borenstein, 1996b] contain too few information because their primary intention is to define a mapping between format and application, not the classification of format types. Exymen's FormatIDs group compatible formats on the SDK/API level. Formats that are basically different but are handled by the same SDK/API are given the same ID. For example, a node that uses Microsoft's Windows Media SDK [34] can handle all audio codecs supported by the *Audio Codec Manager (ACM)*. A lists of the FormatIDs defined so far is available at [76].

4.7.5 Synchronization

When different media are to be processed in parallel they need to be synchronized in most cases. If content flows continuously at a constant data rate, synchronization is trivial since at any point in the stream the time-position is clear. Given a certain stream position, any time position in the past or in the future can be easily extrapolated. When started at the same time, a video stream and an audio stream directly grabbed from a camera and a sound card can be easily kept in sync this way. Whenever there is a time difference, one stream

can either wait or skip bytes. However, if bit rates vary or a medium delivers event-based data, such as strokes from a chalkboard, one cannot easily get and interpolate the time-position of a stream at any time. It is particularly impossible to predict any future time position. Since components may be combined freely, node developers are, of course, not required to handle synchronization on their own.

The first type of synchronization has already been described in Section 4.7.2. When the media graph is activated, all non-sources are initialized first. Then the sources are activated in order to start delivering data. This is done in order to be able to handle all possible errors as soon as possible and to achieve a rather simultaneous start. Deactivation works the other way round: First all sources are stopped, then an event is propagated saying that no further data is available. Thus the remaining nodes can process the remaining data before they shut down.

In addition, the SOPA framework provides a content-independent synchronization scheme that works on a node-to-node level. Several nodes can be grouped together into a synchronization group. Nodes can be added or removed from a synchronization group at runtime. The synchronization mechanism provided by SOPA has been derived from the well-known *barrier synchronization* scheme, which is described in detail in [Tanenbaum and van Steen, 2002]: "A barrier is a synchronization mechanism that prevents any process from starting phase n+1 of a program until all processes have finished phase n. When a process arrives at a barrier, it must wait until all other processes get there as well."

SOPA deals with threads instead of processes and extends the notion of a barrier in two aspects.

1. Instead of a fixed number of barriers, the mechanism assumes an infinite number of barriers and all barriers are ordered and identified by a natural number.

2. A thread is only blocked by barriers of the same synchronization group and only if other threads define themselves dependent on it at a certain barrier.

In the framework this new mechanism is called *Progress-Constrained Threads* since it offers a notion for threads to dynamically depend on other threads' progress. Each Progress-Constrained Thread implements a special interface and registers to a central *Clearance Manager*. Each Progress-Constrained Thread implements a method which allows the Clearance Manager with get the dependencies of this thread on any other threads at a certain barrier. Threads have to request permission for progress by requesting to proceed over the next barrier at the Clearance Manager. The Clearance Manager asks all registered threads of a synchronization group for dependencies on other threads at this barrier. If a requesting thread depends on another, it is blocked until that other thread requests the same or a higher barrier. To avoid deadlocks, the Clearance Manager forces the requested barrier numbers to be monotonously increasing. The dependencies have to be constant for a certain barrier, but may change at later barriers. That means, if a certain barrier has been requested by a thread (and thus marked as reached), the synchronization group cannot be changed for this barrier or any previous barrier.

Any set of media nodes can be grouped into a synchronization group, for example in the XML graph description. When a node is put into a synchronization group, the framework automatically registers the media nodes with the Clearance Manager. By default, the system associates barriers with time stamps using a granularity of 10 ms. The path to an ancestor node is blocked while a given media node has processed more data than the others according to the time stamps. However, other behavior can easily be implemented by overriding the dependency-definition method. Nodes have to decide for themselves what to do while blocked: Either they discard the incoming data, or they accumulate it in a buffer for later flushing.

The following practical example illustrates the approach: Let us assume that we want to synchronize an audio track with a pipe that presents a GUI with a pause button. The pause pipe overrides the default behavior so that only when the button is pressed it reports itself dependent on any node in the synchronization group. The audio stream is then blocked until the pause mode is released and the pipe requests progress again.

4.7.6 Top-Level Application

The SOPA framework is integrated into the E-Chalk system and is invisible to the user. As described in [Knipping, 2005] in Section 3.10.1, any Java application that implements the `de.echalk.util.Launchable` interface can be started by the E-Chalk Startup Wizard. The framework is started and stopped by a wrapper class that implements the interface. Property files are generated by the E-Chalk Startup Wizard and an initial serialization of a video and an audio graph is deployed with the E-Chalk system. In addition to the functions discussed here, the framework also provides some minor features that are not discussed in detail here. Examples include an error and debug message management and a node update management. The update management is configured to determine when new versions of a node should be favored over ones already downloaded in the local repository during graph resolution. Usually update policies are defined inside the LDAP queries, but using the update manager they can also be enforced or forbidden. Further information on the details of the system can be found in the SOPA's developer documentation [17].

4.8 Limits of the Approach

The presented architectural approach facilitates the maintenance and configuration of streaming and processing components which are usually organized in graphs. The validity of the approach has been time-tested since its integration into the official E-Chalk distribution in the beginning of 2003. Still, several issues remain unimplemented or unsolved. This section discusses some of them.

The resolution algorithm depicted here works well with several dozen nodes. It was never intended nor tested for a system with more than a hundred nodes in one graph. Apart from efficiency problems concerning the graph resolution, the realtime performance of the sum of filters in a media graph is not guaranteed. Up to now the system installs components on demand without any knowledge of how much CPU time is consumed. Therefore, the combination of certain components may go beyond the limits of the underlying computer

system. The result would be a denial of service. One solution is to let each developer implement a benchmark for their components which returns a value relative to components that come along with the framework. On the very first run of SOPA, a large benchmark using built-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. One problem of this approach is that CPU time consumption may depend on the semantics of concrete content.

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 (such as deleted files) to the underlying resources. 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 actually provides a technique for dealing with security, namely it executes dynamically integrated components within a security sandbox. The sandbox is used to prevent unauthorized access to the underlying resources and to control the visibility of other installed services and resources. External rules enable the creation of security policies that can assign default access rights to dynamically integrated components, as well as assigning different levels of rights to components from known sources using a public-key cryptography approach. The approach, however, is rarely used because of performance problems. 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.

Due to the generic approach of components that provide their functionality by means of services, the most important thing that a node developer has to do is to define proper *service contracts* in the form of properties describing the node. An inherent problem that occurs when arbitrary parties want to share their component implementations is that all service contracts must be understood by everybody and must provide enough semantic to describe the service in all contexts and all purposes. SOPA reduces this problem by restricting the context to a specific domain, namely multimedia signal processing. Due to this restriction, the syntactic interface descriptions enforced by the architecture, combined with the meta-data encouraged by the framework plus a few properties, are sufficient in most practical cases. In theory, however, it is very easy to describe the same component with totally different service contracts. The result is that components that could fit into a particular position of the graph might not be considered because service contracts are incompatible.

4.9 Practical Usage Examples

The following section presents a few practical scenarios that illustrate the usefulness of the architectural approach discussed in the preceeding Chapter. The SOPA framework was integrated into E-Chalk in 2003. Since then, about a hundred nodes have been developed for both experimental and productive use. Both the audio system presented in this dissertation and the video system have

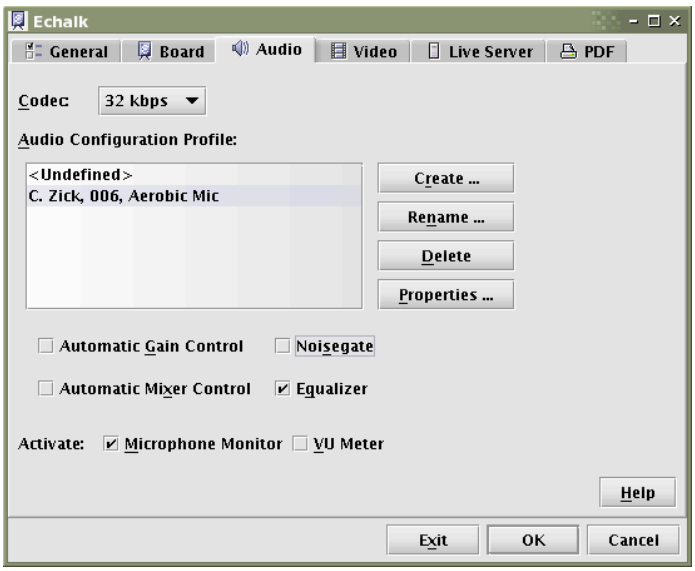

Figure 4.5: A screenshot of the audio panel of the E-Chalk Startup Wizard. The wizard outputs property files that directly affect the graph assembly.

been created using the framework and have been in use for several years now. Some of the presented advantages result from mere component orientation, others results from the functionality of remote component discovery, a third class might result from automatic graph resolution. In the end, the combination of the features discussed earlier offers even more opportunities.

As discussed in Chapter 2 of [Knipping, 2005], E-Chalk is mainly configured through the E-Chalk Startup Wizard. Figure 4.5 shows a screenshot. The output of the Startup Wizard consists of Java property files. Property files are actually hash tables that map a unique key encoded as a string to any value. Before SOPA had been integrated into the E-Chalk system, each property value resulted in a case distinction somewhere in the Java code. Using the LDAP queries, many of these case distinctions are now handled by the graph resolution. In order to promote this feature, LDAP queries can also be used for case distinctions. Nodes that contain disabled functionality are just not considered any more. The following code snippet illustrates this functionality for the VU-Meter checkbox:

```
<on match="(!(audio.tools.vumeter=true))">
<service label="audioprocessors2"
         match="(name=BlackHoleTarget)"
         type="&target;">
</service>
</on>

<on match="(audio.tools.vumeter=true)">
<service label="audioprocessors2"
         match="(name=VU-Meter)"
         type="&pipe;"
         target="audiosink">
</service>
<service label="audiosink"
         match="(name=BlackHoleTarget)"
         type="&target;">
</service>
```

As can be seen from the DTD shown in Appendix B, the standard XML serialization is extended for user editability. Several commands that are usually part of the LDAP queries have been externalized and have their own XML tags. This helps system administrators to edit the file directly. This attribute turned out to be a useful feature when supporting users: They can individualize the processing graph according to their needs. By developing new nodes, the functionality of E-Chalk can be extended using the SOPA framework and without fiddling with any E-Chalk source code. These extension capabilities go beyond traditional plug-in extensibility since even basic functionality can be exchanged.[1] But not only third-party development is made easier: Students in our department found it useful to be able to integrate their own extensions directly into the system. The integration of Eureka into the framework, for example, facilitates deployment of updates and customized nodes. Assume an educational institution has several installations of E-Chalk that have been individually customized by a dedicated developer. The institution can now define its own scope in Eureka and deploy the newly developed nodes individually to all installations in the institution. Scopes can be defined so that experimental nodes are not searched for E-Chalk installations in productive environments. Nodes provided by the E-Chalk developers can still be searched in the home scope `sopa.inf.fu-berlin.de`. The institution just extends the search domain to include their own institution.

The audio expert system presented in Chapter 7 guides the user through a systematic setup and test of the sound equipment. The result is a modified initial XML media graph description that contain a set of filtering services needed for pre- and postprocessing audio recordings. During the lecture recording, the system monitors and controls important parts of the sound hardware. This example shows, that using SOPA, machines can also easily alter the configuration of processing graphs, not only users and system administrators.

Both the configurability using XML and the possibility of reusing components facilitates fast prototyping and debugging of filter processing chains. For example, many of the experiments on the instructor extraction presented in Chapter 9 where conducted in a testing environment that allows to discover and execute media nodes. It was used to test experimental video nodes frame by frame. Figure 4.6 shows a screenshot.

In a typical live streaming scenario, a fork node connecting to different codec nodes can be used to convert a media stream into different formats. A receptor service (for example a generic node) receives the incoming connection request. It then assembles a media graph containing converter nodes that convert the format of the captured media to the format playable by the client software. Instead of forcing the user to install the right plug-in, the server adapts itself to the needs of the user.

Exymen shares the local repository with E-Chalk. Since the format description mechanisms in SOPA and Exymen are identical, a plug-in could easily be developed that acts as a wrapper between Exymen's plug-in API and SOPA's API. The plug-in enables Exymen to import and export any media format as soon as a conversion path can be built between the requested import format and a format Exymen can edit. For exporting, conversion pipes must exist for the

[1] One of the examples where this extensibility was appreciated were requests by researchers of the University of Regensburg, who wanted to combine E-Chalk lectures with RealVideo.

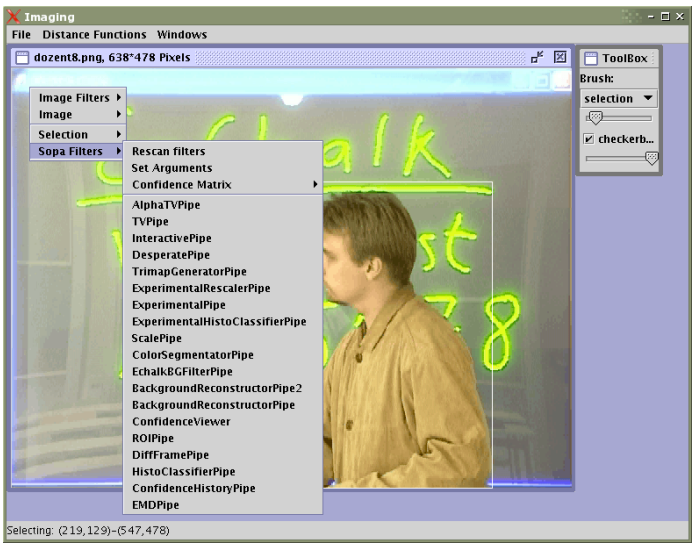

Figure 4.6: A screenshot of the testing environment for rapid prototyping and debugging of video nodes on a frame-by-frame basis.

currently edited format and the requested output format. This way, Exymen adapts itself automatically to newly introduced streaming formats in E-Chalk.

4.10 Conclusion

The SOPA framework manages multimedia processing and streaming components organized in a flow graph. Based on state-of-the-art solutions for component-based software development, the system simplifies the assembly of multimedia streaming applications and associated tools. Components are discovered from interconnected remote repositories and integrated autonomously during runtime. Stream synchronization is handled format-independently. Most parts of the E-Chalk server system, namely the audio system and the video system are based on the SOPA framework. The next chapter describes the implementation of the client system.