A Collaborative Composition System Based On A Service Oriented Architecture

Matthew Dovey[*] and Ian Gibson[†]

[*] University of Oxford
M.Dovey@oucs.ac.uk
Faculty of Information and Technology, Leeds Metropolitan University[†]
I.Gibson@leedsmet.ac.uk

Abstract

This paper describes a system for web-based collaborative music composition. Traditionally, software systems for music composition have been single user systems or multi-user systems evolved from single user systems. Although these systems allow music composition using the Internet, they may not address issues relating to shared resources. The system described here uses Services Oriented Architectures, allowing compositional tools and components to be shared over a distributed environment.

1 Introduction

Work has begun on the implementation of a distributed environment to allow the sharing of sound synthesis techniques and eventually to allow real-time collaborative composition. The current prototype system uses SOAP based Web Services to allow access to a database of audio samples, audio processors and compositions.

2 System Overview

The prototype client establishes a link with the server. Available resources are established using WebServices. These resources might be compositions (e.g. samples, MIDI files etc), sound processing routines or composition control data. The client requests the required data, processes it as necessary, and returns any resulting data to the server.

3 Service Oriented Architectures

The current methodology in developing distributed systems is Service Oriented Architecture (SOA) (Erl, 2005), building upon methodologies such as Object Oriented programming, Components and Distributed Object Request Brokers. Within a SOA, systems are composed of multiple individual services located and maintained on different heterogeneous machines, and administered by different organizations. The key in SOA is that the component services should be loosely coupled i.e. be well-defined, self-contained, and should not depend on the context or state of other services (What is Web-Service Oriented Architecture, 2003). To achieve this, a SOA should display the following properties:

- The services should implement a small set of simple, ubiquitous and well known interfaces which only encode generic semantics.
- The interfaces should deliver messages constrained by extensible schema for efficiency. This allows both services and consumers to work with well defined message structures, while allowing new versions of the services to be introduced without breaking existing systems
- The messages should be descriptive not instructive and the interfaces should not define system behaviour. This allows internals of a service to be viewed as a "black box".
- Service Oriented Architectures must have mechanisms for the discovery of services matching the consumers' requirements.

There are a number of emergent technologies which can underpin SOA, namely REST WebServices; SOAP WebServices and GRID Services (Foster & Kesselman, 1999):

- Representational State Transfer (REST) works on the basic of "resources" which can be referenced by URIs (Fielding 2000). A REST web service is limited to using HTTP interfaces (GET to obtain a representation of the resource; DELETE to remove a representation of a resource; POST to update or create a representation of a resource; PUT to create a representation of a resource). REST messages are in XML, constrained by schema definitions in the XML Schema language (http://www.w3.org/XML/Schema) or Relax NG (http://www.oasis-open.org/committees/tc_home.php?w-_abbrev=relax-ng)
- SOAP Web Services use messages encapsulated in a structure defined by the SOAP specification (http://www.w3.org/2000/xp/Group/). This adds additional information in the form of headers for message routing scenarios and mechanisms for reporting errors using faults (a style similar to exceptions in various programming

languages). SOAP Web Services use the Web Service Description Language (WSDL) to define both the structures (using schema languages such as XML Schema) along with messaging semantics (for example, this might be to test whether the message is initiated by the client or the server, and what messages can be used as a response to a particular message).

- GRIDServices are based on WebServices but provide additional semantics. In particular they add some object-oriented and REST concepts. The object-oriented concepts are demonstrated by the ability to inherit service definitions (portTypes in WSDL terminology), to add new messages (using a multiple inheritance model) and to add properties (or service data elements) to WebServices. The REST concept introduced is that of creating a new representation of resource. In the GRIDServices model this uses a factory model whereby a new instance of a GRIDService can be created by its corresponding factory GRIDService. GRIDServices also offer an extensibility model whereby part of the structure of the message can be left undefined, but the allowed structures can be determined dynamically by querying the appropriate service data elements.

The current prototype is based upon SOAP based WebServices, although future versions may need to take advantage of GRIDService based technology for some of the more advanced features (such as remote execution of compositions).

3.1 Web Service Definitions

The Web Service definitions define how the client communicates with the server using XML based messages. The prototype definitions are defined within WSDL which would allow clients and servers to be implemented on different platforms using different programming languages. The definition defines three different "PortTypes" (which is the WSDL term for a collection of functions). The division into different functional groups allows for a system in which different servers implement different groups, although our current prototype server includes implementations of all three. The defined PortTypes are described in the following sections.

Processor Service PortType. This defined two functions: **processAudio** and **getProcessorsDescriptions**. The **processAudio** function allows the client to send audio data to be processed at the server. It takes as its parameters an identifier, a collection of audio samples and a collection of parameters and returns a single audio sound file. The identifier determines what code will be used to process the audio. The collection of audio samples consists of MIME based 64 bit encoded binary of the actual wave data plus a name to identify that particular input to the code – typical names might be "input1", "input2" etc. The collection of parameters also consists of a list of name, value pairs and allows fine control over the functioning of the server side sound synthesis code. The **getProcessorsDecriptions** returns a list of all the sound synthesis techniques available on the server. For each technique, it also returns the list of names for the input audio, the list of parameters including descriptions and type (e.g. integer, real or Boolean), and also the provenance of the sound synthesis technique (using Dublin Core metadata such as 'title', 'creator' and 'description').

Audio Service PortType. This defines functions for managing a server-based database of audio samples. It implements the functions **getAudio** and **submitAudio** for adding new samples and retrieving existing samples respectively. It also has the function **getAudioDescriptions** which returns a list of available samples and their provenance using Dublin Core metadata.

Composition Service PortType. This defines functions for managing a server based database of compositions. A composition is represented as an XML document detailing the identifiers of the audio samples and processors used (i.e. the identifiers to use as a parameter for the **getAudio** and **processAudio** functions respectively) and the workflow (i.e. how the audio and synthesis are composed to form the final audio). It implements the functions **getComposition** and **submitComposition** for adding new compositions and retrieving existing compositions. It also has the function **getCompositionsDescriptions** which returns a list of available compositions and their provenance using Dublin Core metadata.

3.2 Server Implementation

A prototype server has been developed in Java implementing all three port types, i.e. performing the roles of an audio repository, composition repository and a repository of various audio synthesis processors. This has been developed using OpenSource components such as Apache Tomcat (http://jakarta.apache.org) to provide the base HTTP functionality and the Apache Axis (http://ws.apache.org) to provide SOAP functionality.

The Audio and Composition repositories have currently been implemented as simple file stores with additional metadata store in an XML index file. The processor web service dynamically loads new processors by compiling and loading java classes stored in the processor directory. A web interface has been developed to allow remote uploading of new processors.

Helper java classes have been written to hide the Web Service complexity. These make use of the new Java 1.5 annotation feature so that existing code can easily be modified to work within the framework without major changes.

Consider the following small piece of java code:

```
    public void StereoSwap
      ByteArrayOutputStream leftOut,
      ByteArrayOutputStream rightOut,
      ByteArrayInputStream leftIn,
      ByteArrayInputStream rightIn,
      int swap

      while    leftIn.available     >    0    &&
rightIn.available   > 0
        int rightByte = rightIn.read();
        int leftByte = leftIn.read();
        if  rightByte == -1    leftByte == -1)
          return;

        leftOut.write (rightByte  *  swap/100)  -
(leftByte *  100-swap) 100 );
        rightOut.write( leftByte  *  swap 100)  -
(rightByte *  100-swap  100));
```

Essentially this code provides a very simple channel swap function – taking input from byte streams leftIn and rightIn and returning an output using byte streams leftOut and rightOut. The function also has an additional parameter (swap) which affects its behaviour.

To tell the framework to expose this Java method as an available WebService we merely need to add the annotation "@AudioProcessor(...)" to the function. This marks the function as being available through the web service. Within the parenthesis we include additional metadata such as the name and description of the function. The input channels are similarly marked by using @AudioIntput(..) (including title and description within the parenthesis); output channels using @AudioOutput. and additional parameters by using @Parameter. Hence. the marked-up Java code becomes:

```
    @AudioProcessor contributor="Matthew       J.
Lovey", title = "Stereo channel swap", description
= "Swaps channels  with  crossover bleed " public
void StereoSwap
        @AudioOutput name      =      "leftOut",
description="Left         output         channel"
ByteArrayOutputStream leftOut,
        @AudioOutput name      =     "rightOut",
description="Right         output         channel"
ByteArrayOutputStream rightOut,
        @AudioInput name = "leftIn", description
=  "Left   input   channel"   ByteArrayInputStream
leftIn,
        @AudioInput name = "rightIn", description
=  "Right   input   channel"   ByteArrayInputStream
rightIn,
        @Parameter name    =    "bleedPercentage",
description = "Percentage of signal to swap" int
swap

      while    leftIn.available()   >    0    &&
rightIn.available   > 0
        int rightByte = rightIn.read  ;
        int leftByte = leftIn.read  ;
        if (rightByte == -1    leftByte == -1
          return;
        }
        leftOut.write (rightByte  *  swap 100   +
(leftByte * (100-swap) 100 );
        rightOut.write  leftByte  *  swap 100   -
(rightByte * (100-swap  100  ;
      }
    }
```

The marked-up code can still be compiled and used within its original context but can now also be used within the framework. The server allows the code to be added either in a compiled form (e.g. as a jar file) or by dropping the source code in the processor directory with the extension .jps. In the latter case. the file is automatically and dynamically detected and compiled by the server. and the processor is then available for use.

3.3 Client Implementation

A prototype client has been developed using Java Swing. The client establishes a link to the server and uses the getAudioDescriptions and getProcessorsDescriptions WebServices to enumerate available resources. These are listed in the Java JTree control (on the left hand side of the screen). Audio samples and audio synthesis processors can be dragged and dropped onto the flowchart control on the right hand side of the screen (implemented using the open source JGraph Swing control). The various components can be then linked up to form a composition. Right-clicking on a processor brings up a list of the parameters that can be changed for that processor. The file menu allows the saving and loading of compositions either to the server (via submitComposition and getComposition WebServices) or to the local machine. The output of the composition can be sent to the speakers of the local machine. a local wave file or to the server (via the submitAudio WebService).

3.4 System Testing

The current system is an early prototype and still requires testing and feedback by users. However there are a number of areas which have already been identified as requiring further development. In particular. the network traffic generated is considerable. and data relating to this is being gathered for analysis. Modifications to the simple user interface are being undertaken during these early testing stages. However. a review of user interface requirements will be used to establish a specification for the final version.

4 FUTURE RESEARCH

4.1 Searching

At present, a client must request the data for all the resources (audio, processors or composition) on the server in order to discover what is available. This is clearly not scalable as the number of resources grows. It would be better for the server to support a search interface so that audio processors, audio clips or compositions can be found matching a user's query. The use of the generic SRW WebService protocol will be investigated for this purpose.

4.2 Multiple Servers

The architecture should be expandable to support multiple servers, so that the client might use processors, audio, etc. from numerous servers. For this to work there needs to be a naming convention so that a resource (audio sample or synthesis processor) and the server on which it is located can be determined from an identifier stored in a composition. Various systems such as WS-Addressing will be investigated for this purpose. There are also various ways as to how a client discovers the various servers and their resources. For example, a client might send a get...Descriptions web service (or a search) to multiple servers simultaneously. An alternative approach might be for the servers to replicate metadata descriptions between themselves so that a client need only send a get...Descriptions or a search to a single server. This will then initiate a response with information about the resources of all servers. It is likely that a hybrid approach will be implemented for greater flexibility.

4.3 Server execution of compositions

The current prototype is inefficient in the use of network traffic. In a typical composition, the client will retrieve a number of audio samples from the server using the getAudio WebService, then send those audio samples back over the network to the server for a synthesis technique to be applied (via the processAudio WebService). The resultant audio will be transferred back to the client. As this may be an input to another audio processor the resultant audio may be passed over the network to and from the server many times. A more efficient solution would be for the composition to be sent to the server and executed on the server. With multiple servers it will be necessary to calculate the most efficient workflow. GRID technologies will be investigated to provide this functionality.

4.4 Peer to Peer technologies

The prototype architecture is a client-server approach. A more flexible approach might be to use a peer to peer approach whereby a user can publish audio samples and synthesis processors from their local machine. Combining the client with the server would enable this once a multiple server architecture is in place. However, this might have security implications.

4.5 Instant Messaging Technologies

The system described is meant to encourage collaborative composition. This can only be achieved if the collaborative composers not only have access to shared resources but also to real time communication tools such as text based chatting, whiteboards, video/audio conferencing etc. Various technologies will be investigated for integration into the client.

References

What is Service-Oriented Architecture.
http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

ERL, T (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River: Prentice Hall PTR.

FIELDING, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Ph.D. Dissertation (University of California, Irvine), Chapter 5

FOSTER, I. & KESSELMAN, C. (1999). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann.