APP – ARCHITECTURE FOR PIPELINED PROCESSING

Rui Lopes, Luís Carriço
LaSIGE & Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, Edifício C6, 1749-016 Lisboa, Portugal
{rlopes,lmc}@di.fc.ul.pt

ABSTRACT

This paper presents an architecture for XML processing applications, defined by a set of transformations preformed over complex XML contents. The architecture is composed by stages (logical separation of concerns), each stage is defined by a set of pipelines (independent processing of contents), and each pipeline is a sequence of processing components (small-size units that perform a single transformation over a subset of the content). These components are registered with metadata about which subset of the content they will process. This way, the application pipeline configuration does not have to define which subsets are being processed, hiding details from the application configuration steps, thus simplifying the creation of complex XML-based applications. A case study is presented.

KEYWORDS

XML, XML Processing, Pipeline Processing, Digital Publishing.

1. INTRODUCTION

With the proliferation of XML (Yergeau *et al*, 2004) technologies and digital publishing, complex XML-based publishing applications have emerged. These are usually described by a set of input sources, transformed through several steps defined by a configuration, resulting in a set of final outputs. While with simple transformations it's easy to manage such tasks with simple scripts, e.g. a shell script executing an XSLT (Kay, 2004) transformation over an XML document, when the complexity grows it becomes unfeasible to manage the transformations with these scripts. The need for a higher level tool to support the definition of what, how and when to apply transformations emerges. Such tools have to support multiple types of transformation languages, different types of input sources, and also the ability to handle multiple inputs and outputs, as well as keeping the configuration of such transformations as clean as possible, hiding the details from it.

There is, though, a lack of support from current pipelining architectures for the specification of multiple outputs, and also the separation between inputs and outputs from the pipeline specifications. The need for a cleaner approach for this problem emerges, allowing the creation and maintenance of complex publishing applications to become more feasible.

This paper describes the Architecture for Pipelined Processing (hereinafter referred as **APP**), an XML framework designed to support the creation of complex transformations over sets of inputs, through the sequential concatenation of transformations (pipelines). The following section describes the requirements for complex XML digital publishing applications. Section 3 describes with more depth the logic behind APP, as well how to create different configurations. Next, in section 4, it's described a case study as an example of how to create applications with APP. Section 5 presents related work on pipelining architectures, as well as its limitations regarding the requirements. Finally, future work is drawn and conclusions are made.

2. REQUIREMENTS

XML digital publishing architectures are, by nature, complex. They have to manage complex input sets, handle several transformation steps and different output sets, be able to support several configuration possibilities, support batch publishing control, etc. Some of these issues are handled by different users, so a clear separation of concerns has to be reflected on the architecture definition. In typical digital publishing architectures, top level users need to specify which processing steps have to be applied on a given content, specify some configuration, or even just specify which content is going to be processed. On the other hand, a developer creates components and makes them available for use. To further explore the rich features this type of architecture has, without sacrificing scalability and maintenance, an *n-tier* approach has to be defined towards content processing.

This kind of flexibility leverages the following requirements towards the definition of a robust and extensible digital publishing architecture:

- The ability to handle complex inputs;
- Deliver complex processing outputs;
- Ability to define logical stages of processing;
- Support different processing configurations;
- Focus on content processing reuse and maintenance;
- Ease batch processing of sets of contents;
- Clear definition of user tasks.

3. ARCHITECTURE

APP defines an application as a forward processing chain of stages, where each stage has a well defined set of inputs and another set of outputs. The first stage's input is the application input and the last stage's output is the final output resulting from the processing chain (as seen on figure 1). All other stages use its previous one's set of outputs as the input for processing. This way, applications can create logic separation of concerns between the processing of inputs. The need to split the processing in stages arises from the nature of digital publishing applications (and, more generally, any kind of hypertext application) where several layers of content are well-defined, as stated in Halasz and Schwartz, 1994. As such, the processing of each layer has to be split in the same conceptual layers.

Each stage has a set of independent processing pipelines that process independent subsets of the stage's input, resulting also in independent subsets of the stage's output. Subsequently, each pipeline is defined by a sequence of processing components that will handle all transformations and/or generations of the current pipeline input. All APP processing components defined by an application are registered in a central registry, and then referred throughout all pipeline definitions.

In dependency-based processing chains, most of the time it's difficult (or even impossible) to handle all issues raised in requirements, especially regarding multiple outputs from a processing component. These issues arise with the notion of dependency, based on an output as the starting point for processing, and then using its processing configuration to achieve the expected output result. When one of the inputs needed to create this output does not exist yet, it's generated in the same way, and then fed to the current processing step. Having a processing component that generates multiple outputs breaks the specification of this type of processing chains.

As such, APP was developed to ease the development of complex digital publishing applications, by giving the ability to compose different processing components seamlessly, especially regarding to separation of concerns issues for creation, management and maintenance of applications and its respective contents, as well as a way to solve dependency-based processing issues.

A forward processing chain approach adequate better to the creation of digital publishing applications, mainly by it's intrinsic nature to handle a set of inputs and apply a sequence of transformations (specified in a rule-based fashion – APP's pipelines), resulting on a set of outputs. This way, the modeling of which transformation steps are to be executed, and in which order, is clearer to the user who configures the pipelines (usually it's not a developer who performs these tasks).

Another benefit from using this approach is the way complex inputs and outputs are handled. Often, on a complex digital publishing application, processing components handle a subset of the input, having as output result the modification of its input and/or the creation of new content. As a consequence of this architecture definition, managing all this complexity is easier with the use of APP.

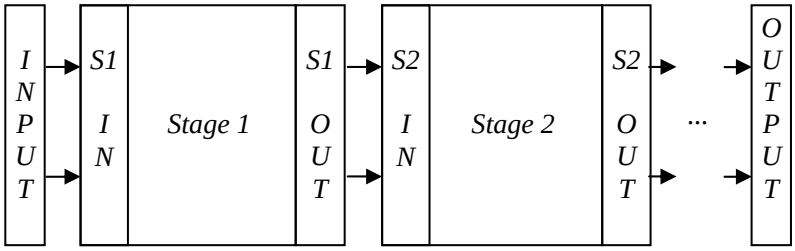

Figure 1. APP stages sequence

Currently, the APP prototype implementation is defined by an Ant (Apache Foundation, 2000) script and an XSLT stylesheet, used to dynamically transform the application definition document into a temporary Ant script, which describes the sequence of steps that are run between each processing stage.

An APP application is specified in an XML document based on RDF (Becket, 2004) syntax, describing metadata associated with the application, in Dublin Core format (Becket, 2002 *and* DCMI Usage Board, 2004), and also the sequence of processing stages. These two descriptions are identified within the document with well-known URIs (Berners-Lee, 1998) in *rdf:Description* blocks: *urn:app:project:metadata* for application metadata-related information, and *urn:app:project:stages* for the stages sequence definition. With the definition of stages in separate files, a better control of the processing stage sequence is handled in the application definition.

As an example use, for better illustration on how to use APP, a simple application for production of cooking recipes has been defined (and will be used through the article). The goal of this particular digital publishing application is to repurpose a given cooking recipe (or a set of these), following several defined criteria, as well as the ability to use different types of output formats (printing, web-based, rich multimedia, etc.). As input several contents can be defined: the recipe (split in several files describing all steps), list of all ingredients and where to buy them, similarities between ingredients, etc. The repurposing of a recipe is defined through several steps in the first stage (*repurpose.xml*), while the output format steps will be defined in the second stage (*output.xml*). Listing 1 presents the application definition for this example.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">

  <rdf:Description rdf:about="urn:app:project:metadata">
    <dc:title>Cooking recipes</dc:title>
    <dc:author>Rui Lopes</dc:author>
  </rdf:Description>

  <rdf:Description rdf:about="urn:app:project:stages">
    <rdf:Seq>
      <rdf:li rdf:resource="stages/repurpose.xml" />
      <rdf:li rdf:resource="stages/output.xml" />
    </rdf:Seq>
  </rdf:Description>

</rdf:RDF>
```

Listing 1. APP application definition

Next, in the following subsections, APP will be further discussed through a thinner decomposition of the architecture's structure (stages, pipelines and components), and an analysis is made on the relationship between all APP concepts.

3.1 Stages

A stage is defined as a conceptual subdivision of a digital publishing application defined in APP. It's a step of all processing, with well-defined set of inputs and outputs. This way it can be inserted in the global processing pipeline, as long as the previous stage output contains all the input of the current stage. This rule is applied to the next stage of the application, and so on.

Each stage can be decomposed in a set of processing pipelines (as seen on figure 2). The stage feeds a different subset of its input to each pipeline, and executes the transformations specified in each one. The resulting outputs from the pipelines are aggregated, creating the stage final output. Each pipeline is independent from other pipelines of the same stage, so that every output from the pipelines does not collide when the stage aggregates all outputs. As long as this rule is not broke, there is no restriction on the number of pipelines a stage can manage.

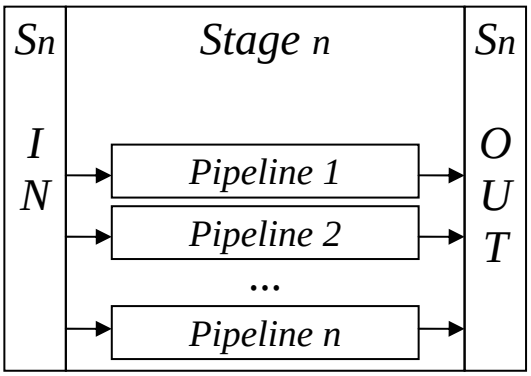

Figure 2. APP stage definition

The specification of a stage is made on an XML document, with a special purposed dialect created for this task, under the XML namespaces (Layman, 2004) *urn:app:stage:config* and *urn:app:component:registry*. The document root tag *config* encapsulates all pipeline definitions for the stage (each one defined with pipeline tags, enclosing all processing tasks to be executed sequentially on that pipeline).

Regarding prototype implementation issues, the stage execution (thus the pipelines execution) is done through the runtime creation of temporary Ant scripts reflecting each one of the processing steps defined on the stage specification. These scripts are created similarly on the application definition execution step, by applying a XSLT stylesheet to the specification, in this case the stage specification document.

3.2 Pipelines

A pipeline is a sequence of logical processing steps (defined with components) applied to a set of inputs, resulting on a set of outputs. On these steps, not only the set of inputs can be modified accordingly to the processing rules defined in each processing component, but also new content can be generated by them. The aggregation of all results that comes from these steps will be the pipeline's final set of outputs.

The composition of components in APP is defined in a way that becomes feasible to handle the processing of multiple inputs and outputs in a transparent way, when specifying a pipeline definition. This way, the inter-component conjugation becomes more flexible, as seen on figure 3. A simple example to further illustrate this flexibility, based on this figure, can be seen in the execution step of component C6. This component has C4 as the only component with outputs that will be used as inputs in C6. In the same way, C4

uses as its input, the outputs from C1 and C3, and so on. While this approach is similar to dependency-based ones, C6 does not have to use all of C4's outputs; it can use a subset of this.

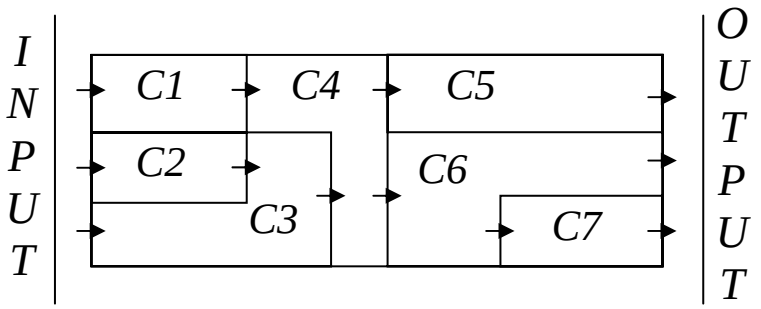

Figure 3. APP pipeline decomposition

Although the complexity level for this kind of inter-component links is higher (inherited from the complex nature of digital publishing applications), the specification of the sequence of processing steps for each pipeline is much simpler, as seen on listing 2. All inter-component links are automatically established by APP's pipeline execution, through the linkage of each component's inputs and outputs definitions, as specified on the registry.

With this approach, APP allows to explicitly define each processing pipeline, starting from the input, finishing on the output, instead of defining the pipelines through dependency chains, as discussed previously.

Listing 2 shows two pipelines defined in the repurposing stage of our cooking recipes application. The first has components to modify the recipe sauce (first, removing potentially expensive ingredients, and then adding some salt). The second pipeline describes the modifications done to a salad (adding toppings). Both pipelines handle different subsets of their respective stage input.

```
<config xmlns="urn:app:stage:config" xmlns:reg="urn:app:component:registry">
  <pipeline>
    <component reg:idref="cook:sauce:cheaper">
      <param name=”max-price” value=”5.00” />
    </component>
    <component reg:idref="cook:sauce:saltier" />
  </pipeline>

  <pipeline>
    <component reg:idref="cook:salad:topping">
      <param name=”type” value=”olive oil” />
    </component>
    <component reg:idref="cook:salad:topping">
      <param name=”type” value=”vinegar” />
    </component>
  </pipeline>
</config>
```

Listing 2. APP pipeline definition

3.3 Components

A processing component is defined by APP as a single unit of processing of a set of inputs, resulting in a set of outputs after its execution. To increase the configuration possibilities for each component, two types of descriptors are associated: links and parameters. Links define which set of input sources are fed to the component, and also which set of outputs are produced by it. This creates a high-coupling interface for a

component. However, through the identification of each link with an URI, it's possible to define a low-coupling interface to the component, so it can be reused with different sets of input sources, on any pipeline and any stage. On the other hand, parameters are defined to allow a higher configurability of the component.

Each component is registered in the central registry in a RDF resource list with a unique identifier, through the attribute *reg:id* (so it can be referenced in a pipeline definition), and the component location. Implicitly, all processing components are XSLT stylesheets (for prototype purposes).

Associated with the component register, a small set of metadata is also defined. Identified with the URI *urn:app:component:metadata*, it is presented information about the component (defined with Dublin Core specifications).With the URI *urn:app:component:plugs*, APP knows which links are defined to the current component, to allow inter-component linkage in the pipeline execution steps, either with high-coupling or low-coupling interfaces. The creation of a URI in the *plug:urn* attribute of a link, enables a low-coupling interface for the component. At last, with the URI *urn:app:component:params*, APP allows parameters for the component to be specified.

On the other hand, on the pipeline definition, components are referred as stated before, but parameters can be added, as well as low-coupling interface links. Parameters are defined with *param* tags and the attributes *name* and *value*. Links are defined with *plug* tags and the attributes *urn* and *href*.

In our cooking recipes example, several components could be registered, creating the links between the registry identification and the stylesheets to repurpose the repices description and output creation (for example HTML or some other format).

4. CASE STUDY

As a case study, a real digital publishing application was created on top of APP. This application is a framework for the production of Digital Talking Books (DTB), based on media indexing and multimodal interaction elements. The requirements for this framework balance between current standards for DTBs (ANSI/NISO, 2002, Bingham, 2002, *and* Daisy Consortium, 2001), the flexibility needed for generation of exploratory and adjustable user interfaces and the ability to integrate new multimedia units in the production process (Carriço *et al*, 2004), as seen on figure 4.

This framework was then implemented based on APP, by defining four stages of processing DTB inputs: book structuring and repurposing, output formatting, interaction configuring and, finally, user interfaces configuring. The possibilities of combination for all configuration at each stage becomes quite complex using other approaches. But with the componentized approach of APP, building, customizing, and configuring rich DTBs becomes easy.

On the book structuring stage, components were created to enhance contents and to create a standardized format for book content description. Regarding the output formatting, the content is enhanced in a way that can be read in different software, with different capabilities: XHTML+SMIL (Newman, 2002) players (such as Internet Explorer 6), SMIL (Ayars *et al*, 2001) players (such as Real Player) and DTB players (such as the Daisy player). The interaction stage was designed to allow different types of interaction with a rich DTB: speech, mouse and keyboard, either alone or combining some or all of them. At last, the user interface configuring was designed to allow the creation of UIs targeted to different kinds of displays and also to create standard UIs for sets of DTBs.

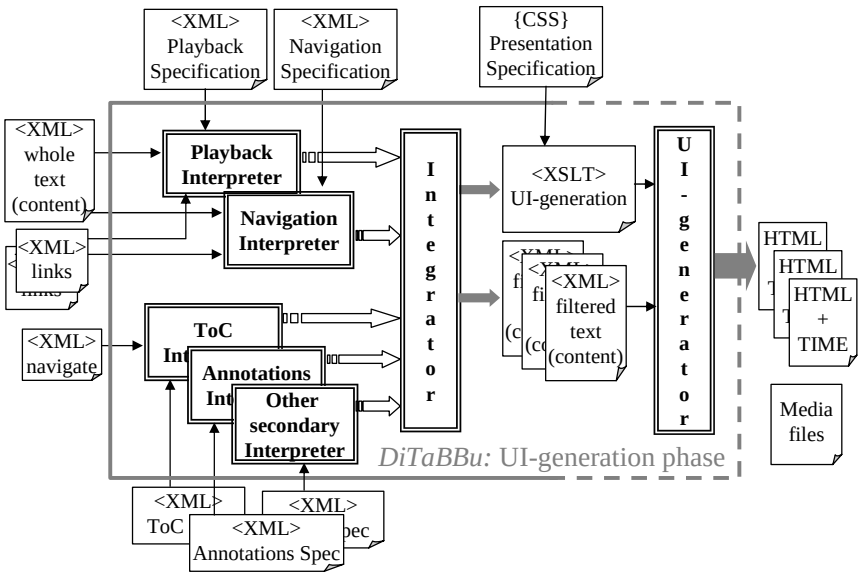

Figure 4. DiTaBBu framework specification

All these components were implemented with XSLT stylesheets, as the DTB framework inputs are XML documents.

5. RELATED WORK

Similar approaches towards XML processing pipelines have been made before. The most successful is Cocoon (Apache Foundation, 1999), which started as a digital publishing framework, but has evolved into a web development framework. The main concept behind Cocoon is the dynamic generation of an output, by dependency (like build tools, such as make or Ant). You specify which output has to be generated, and automatically is determined which inputs have to be generated and included, in a recursive fashion. Most of the times, this output is a web page, an image or a PDF file. This web-based approach is not well suited for offline digital publishing applications, because usually these generate a set of outputs, not just one output. Also, Cocoon defines pipelines as a dependency chain of processing units (where each unit itself is a sequence of transformations over a single input), but it's more useful to have a forward processing chain, regarding separation of concerns issues and pipeline construction logic (similar to a *n-tier* architecture).

There is also a specification (Walsh, 2002) that defines an XML dialect to support the construction of XML processing pipelines. Again, like Cocoon, the approach taken is a dependency-based processing chain, not a forward processing chain. Also, the specification leaves out issues regarding multiple output generation by the processing components, not well suited to develop complex applications.

Both approaches mix the pipeline definitions and their target sets of input for processing, being more difficult to split concerns regarding user tasks, as explained in the requirements.

6. FUTURE WORK

Regarding future work related to APP, a port of the current implementation to Java is being designed. This way more processing power can be achieved, through keeping documents in memory, instead of writing each processing component output to disk (Ant script implementation limitations). To speed up the processing,

there will be changes in the implementation to support specification of certain stages as starting points and/or ending points, leveraging the processing steps (useful for debugging single stages). Other enhancement to be made is enabling the support for any type of processing components (as opposed from now), possibly by creating a standard API for processing components. Other improvement to APP is the creation of schemas (Fallside, 2004) to validate all architecture resources. Additionally, enhancements to all languages for the specification of APP applications will be made, to simplify them and also to improve potentialities regarding new features. Finally, some considerations about validation steps of the content are also being thought to be added to the architecture, as a side-verification of the pipeline configuration. These validation steps will be based on namespace verification, schema validation, and related technologies.

7. CONCLUSION

This paper presented APP, an XML framework to support XML digital publishing applications. Requirements were gathered, defining the goals for APP to solve. We've discussed the framework composition, and how every processing component is used in a pipeline configuration, to ease the development, configuration and use of digital publishing applications. Also, a practical example has been shown, to further explain how to use the framework. We've also described related work on pipeline-based frameworks and its limitations regarding complex content processing and how APP solves these issues with a forward-processing chain approach. At last, future work was drawn and conclusions were made.

REFERENCES

ANSI/NISO, 2002. *Specifications for the Digital Talking Book.*

Apache Foundation, 1999. *Apache Cocoon.* Available from: http://cocoon.apache.org

Apache Foundation, 2000. *Apache Ant.* Available from: http://ant.apache.org

Bingham, H., 2002. *Daisy 2.02 Specification.* Available from: http://www.loc.gov/nls/z3986/v100/dtbook110doc.htm

Berners-Lee, T. et al, 1998. *RFC2396: Uniform Resource Identifiers (URI): Generic Syntax*. Available from: http://www.ietf.org/rfc/rfc2396.txt

Beckett, D. and McBride, B., 2004. *RDF/XML Syntax Specification (Revised). W3C Recommendation*. Available from: http://www.w3.org/TR/rdf-syntax-grammar

Beckett, D. et al, 2002. *Expressing Simple Dublin Core in RDF/XML. DCMI Recommendation*. Available from: http://dublincore.org/documents/dcmes-xml

Carriço, L. et al, 2004. *Building Rich User Interfaces for Digital Talking Books*. Computer-Aided Design of User Interfaces IV, Kluwer Academic Publishers.

Daisy Constortium, 2001. *Daisy 2.02 Specification.* Available from:
http://www.daisy.org/publications/specifications/daisy 202.html

DCMI Usage Board, 2004. *DCMI Metadata Terms. DCMI Recommendation.* Available from: http://dublincore.org/documents/dcmiterms

Ayars, J. et al, 2001. *Synchronized Multimedia Integration Language (SMIL 2.0)*. Available from: http://www.w3.org/TR/SMIL2

Fallside, D. and Walmsley, P., 2004. *XML Schema Part 0: Primer (Second Edition)*. Available from: http://www.w3.org/TR/xmlschema-0

Halasz, F. and Schwartz, M., 1994. *The Dexter hypertext reference model*. Communications of the ACM, vol. 37, number 2, pp. 30-39.

Kay, M., 2004. *XSL Transformations (XSLT) Version 2.0*. Available from: http://www.w3.org/TR/xslt20

Layman, A. et al, 2004. *Namespaces in XML 1.1*. Available from: http://www.w3.org/TR/xml-names11

Newman, D. et al, 2002. *XHTML+SMIL Profile.* Available from: http://www.w3.org/TR/XHTMLplusSMIL

Walsh, N. and Maler, E., 2002. *XML Pipeline Definition Language Version 1.0*. Available from: http://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228

Yergeau, F. et al, 2004. *Extensible Markup Language (XML) 1.0 (Third Edition)*. Available from: http://www.w3.org/TR/2004/REC-xml-20040204