A Component-Based Architecture for Ginga

Marcio Ferreira Moreno, Luiz Fernando Gomes Soares, Renato Cerqueira
Departamento de Informática – PUC-Rio
Rua Marquês de São Vicente, 225 – Rio de Janeiro/RJ – 22453-900 – Brasil
{mfmoreno, lfgs, rcerq}@inf.puc-rio.br

Abstract — This paper discusses how component-driven development can be used in the design of the Ginga middleware architecture, including its Ginga-NCL presentation module. Presentation engines have an important facility, since they allow for previewing when each specific media player is needed. Therefore, to maintain temporal consistency during application presentations, instantiation time of media players can be computed. The paper describes how this approach has been considered in the design and implementation of Ginga, the middleware of ISDB-T terrestrial digital TV system and ITU-T Recommendation for IPTV services. The evaluations presented in the paper illustrate the benefits component-driven architecture can bring to digital TV middleware systems, such as decreasing the amount of needed resources and improving their dynamic evolution capability.

Keywords— component-driven architecture, NCL, Ginga, Multimedia Synchronism, Digital TV, Interactive TV.

I. INTRODUCTION

In digital TV (DTV) systems, middleware is the software layer that gives support to application execution and makes them independently from receiver platforms.

The middleware design and implementation must take into account some special constraints. Usually, receivers have scarce resources offered to applications, due to cost limitations: low power CPU and limited memory. However, they usually provide specialized hardware targeting TV content presentation.

Component-based development [1] can be an interesting approach in this scarce resource scenario, in which a given set of functionalities must be present, only when they are required at presentation time. However, a critical issue must be stressed, the delay imposed in loading a component may not impair the temporal synchronization among media content during DTV application presentation. To find a strategy that enables multimedia presentations with the minimum software components in memory during application running without breaking the spatiotemporal relationships specified by application authors is thus a key issue.

Another key issue in DTV middleware design that can also take profit of component-based approach is the real time support to software updates. Dynamic middleware evolution support allows for integrating new functionalities, for replacing old ones, and for architectural redefinitions coming from unpredictable changes in the original project [1].

This paper discusses how component-driven development can be used in the design of the Ginga middleware architecture, including its Ginga-NCL presentation module. Ginga is the middleware of the ISDB-T (International Standard for Digital Broadcasting) standard [2]. Ginga-NCL supports the presentation of applications developed using the NCL (Nested Context Language) declarative language and its Lua scripting language [2] [3]. Ginga-NCL and NCL are also ITU-T H.761 Recommendation for IPTV services [3]. The component-driven approach proposed has been used in the Ginga reference implementation [4].

The paper is organized as follows. After this introduction, Section II presents some related work. Section III describes the modular architecture of Ginga in its support to declarative applications. Section IV discusses the component-driven implementation applied to Ginga architecture. Section V presents and comments performance measures coming from the Ginga monolithic and the Ginga component-driven implementations. Finally, Section VI concludes the paper.

II. RELATED WORK

Restraining our discussion to middleware systems, several component-driven solutions aiming at providing high-degree of adaptability and better control of computational resources have been discussed in the literature.

Coulsen et al. [5] present OpenCom as a software component model to design low-abstraction-level systems, e.g. middlewares, trying to provide the same reconfiguring facilities other component infra-structures provide for the final application levels. OpenCom has been used in middleware systems for different application domains, such as programmable network processors, reflexive communication middleware systems, and routing systems for mobile ad-hoc networks [6] [7].

The Fractal component model [8] has also been used to design middleware systems. Layaida et al. [9] present an archetype based on this fractal model for multimedia application design, named PLASMA. In their paper, they show how the architecture provides real time adaptations with low performance impact, even if it is used by mobile devices with scarce computational resources.

Souza Filho et al. proposed the FlexCM model [10] aiming at the automatic composition of middleware architectures by explicitly representing component connections using an XML
file to specify the middleware architecture. In the specification, component interfaces are identified by Globally Unique Identifiers (GUID) that guarantees the global individuality. Each component can declare the required and provided GUID. An execution environment is then able to compose the final architecture. FlexCM was used in the design of FlexTV [10] middleware for DTV Java applications.

Although there are reports about component-driven designs for DTV imperative middleware systems, like those previously introduced, to the best of our knowledge our proposal is the first one targeting DTV declarative middleware engine. DTV presentation engines have an important facility, since they allow for previewing when each specific media player is needed. Therefore, to maintain temporal consistency during application presentations, instantiation time of media players can be computed. If these players are not yet present in the presentation platform, a prefetching plan for loading them can be built.

The experimental results obtained for Ginga component-driven implementation reinforce the advantages raised in works presented in this Section II. However, it must be stressed that the Ginga implementation was developed based on services directly provided by the operating system, that is, without using any software infra-structure for component-driven design, as is the case of all previously mentioned approaches. This feature is significant for our purpose since it can contribute to reduce memory and CPU consumption, which are important constraints for DTV receivers, as aforementioned.

Table 1 shows some central key points regarding the work efforts mentioned in this section. Note that all proposals, apart from Ginga, do not have temporal synchronization in their requirements. This means that delay problems coming from component loading are neglected. However, this is a very important requirement for multimedia presentations by low cost receivers, as those provided for DTV systems.

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Evolution</th>
<th>Resource Management</th>
<th>Infra-structure Independency</th>
<th>Temporal Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCom</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fractal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FlexCM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ginga</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

III. GINGA COMPONENT-DRIVEN ARCHITECTURE

Figure 1 shows the Ginga modular architecture divided into two main subsystems: Ginga-NCL presentation engine and Ginga Common Core (Ginga-CC).

Ginga-CC provides basic media transmission/reception and decoding services to Ginga-NCL and Ginga’s optional extensions (not discussed in this paper). Ginga-CC is the single part of Ginga that depends on the receiver hardware and the operating system platform. It allows Ginga-NCL to be platform independent.

Ginga-NCL is the logical subsystem of Ginga that controls the entire life cycle of an NCL DTV application.

In presenting the Ginga architecture, a module is defined as a set of software components that provides a specific functionality. Software component is defined as in Szyperski [1]: a unit of composition with contractually specified interfaces and explicit context dependencies only.

There are two types of components: permanent and temporary. The first ones are those that are used uninterruptedly, while the device is in operation. Therefore, they must be kept all time in memory. Updates in these components should be done when the receiver is in the standby mode. On the other hand, temporary components are those only needed in specific moments of the presentation. They should be kept in memory only when needed.
these components are kept in memory is defined by the Player Manager, discussed in the next subsection. Ginga-NCL defines an API that shall be followed by all media players, in order to standardize the communication between the NCL Player and the media players. If third party media player components are integrated to Ginga, it can need the service of Adapters modules. Content to be exhibited by media players can come from the Data Processing module, if it comes multiplexed in the service transport flow, or can be received from the Transport module.

A special media player is the Lua engine for Lua code execution. Like other media players, Lua engine is kept in memory only during periods defined by the Player Manager, discussed in the next subsection.

The Graphic Manager controls the global spatial content rendering, including the main DTV video. The module’s components must be permanent.

All Ginga components can be independently updated. Updates can come as pushed data or received on demand using the services of the Transport module. In the first case, a permanent component of the Dynamic Evolution Manager module is in charge of monitoring if there are component updates multiplexed in the data received by the Tuner module. In the second case, a temporary component of Dynamic Evolution Manager module is in charge of querying servers for updates, by using the services of the Transport module. This temporary component should be kept in memory only during query resolutions. Update query polices can be determined by viewers. The other Dynamic Evolution Manager components are responsible for the updating process, without interrupting the middleware execution. These components are temporary and should be kept in memory only during the updating process.

The Device Manager module deals with tasks related to distributed presentation on multiple exhibition devices: device registering, device intercommunication, distributed-media synchronization control, etc. [2] [3]. The module’s components should be kept in memory only during distributed presentations.

Finally, the Context Manager module administers information about viewers and devices profiles, used in content and content presentation adaptations. Since this information is persistent, the module’s components can be kept in memory only during information updating and access.

B. Ginga-NCL Componentization

The main module of the Ginga-NCL subsystem shown in Figure 1 is the Formatter, or NCL Player. This module is in charge of receiving and running NCL applications, no matter if they are resident applications or if they are applications received from Ginga-CC. The Formatter’s components can be kept in memory only if there is an application to run.

Upon receiving an NCL application, the Formatter requests the services of the XML Parser module that translates the NCL textual specification into data structures that represent the NCL conceptual data model, called NCM [2] [3]. The XML Parser’s components are only needed during the translation process and to compute NCL live editing commands discussed ahead. Therefore, they may be all temporary components.

The resulting NCM entities are grouped in a data structure called Private Base. Ginga-NCL associates at least one private base with each TV channel (assembling a set of channel services). Other private bases can be opened (or created), but at most one to each service in a tuned channel. NCL documents in a private base may be started, paused, resumed, stopped and may refer to each other.

As soon as the XML Parser ends its translation process, the Formatter calls the Scheduler to orchestrate the NCL application presentation. The Scheduler interprets the NCL data structures, using the services of the Converter module, in a second translation step done during application presentation, step by step. The Converter’s components are needed during all the application presentation, differently from the XML Parser’s components that do their job before presentations begin. This is one of the reasons why two-step conversion has been used.

During the presentation, the Scheduler requests the Player Manager module to instantiate players according to the content type being presented. When a content presentation finishes, the media player notifies the Scheduler. If there is no other content of the same type to be exhibited, the Scheduler commands the Player Manager to destroy the media player instantiation. The same procedure can happen if the Scheduler needs to stop a media presentation. Therefore, the Player Manager is in charge of loading and freeing media player components.

Media content in exhibition must be placed in a display area in agreement with the NCL application specification. The Layout Manager module associates content rendered by media players to display regions of one or more device screens. If multiple exhibition devices are used, the Layout Manager calls the Device Manager’s services to transmit content to be presented to appropriate devices and to control their presentation. The Layout Manager’s components must be kept in memory during the whole presentation.

The Private Base Manager must control all created private bases. This module’s components shall be kept in memory while there is at least one active private base. The Private Base Manager is also responsible for processing live editing commands that allow for private base control (activation, open, close, etc.), for controlling of NCL application life cycles, and for changing applications during runtime. Changes on NCL application requested by NCL editing commands may be specified as XML parameters. Therefore, when needed, the Private Base Manager module calls the XML Parser’s services to translate these changes into NCM data structures.

Finally, the NCL Context Manager adapts NCL applications, according to information provided by Ginga-CC and NCL instructions programmed by application authors.

IV. COMPONENT-DRIVEN IMPLEMENTATION OF GINGA

From version 0.10.1 on, the Ginga reference implementation has added a new optional module called Component Manager. At compile time the instantiation or not of this module defines if middleware libraries will result in a component-driven Ginga implementation or in a monolithic version. The two versions are provided to allow for implementing Ginga in platforms that do not offer support to component-driven development. These two versions are compared in Section 5.
The Component Manager module, shown in Figure 2, has the necessary functionalities to load and to release each component of the Ginga architecture. These operations are accomplished by the ComponentManager permanent component through the libdl library, which is the single introduced dependency to allow for loading and unloading components into system memory, by using the IComponentManager interface. The code for handling components is kept centralized to make easier the middleware embedment into receiver platforms that need an alternative to libdl library.

![Figure 2. Component Manager](image)

The architecture specification is defined in an XML document created by a script on component compilation and installation. When the receiver device starts its operation, the ComponentManager initiates and calls the services of the CMParser component, through the IComponentParser interface, to interpret the XML document. As a result, a set of directives on component loading and unloading are represented by the IComponent interface for all Ginga components.

If the middleware must be updated, the Dynamic Evolution Manager module receives the updates and changes the XML document by means of the IComponent Descriptor interface. The Dynamic Evolution Module is also responsible for notifying the ComponentManager, by means of the IComponentManager interface, that the update took place.

Figure 3 shows the DTD (document type definition) of the Ginga architecture XML specification. The <middleware> element is defined as the parent element of one or more <component> elements. To assist the Dynamic Evolution Manager, the required attributes of the <middleware> element define the receiver platform (identifier, operating system - OS, OS kernel version) where Ginga will be embedded.

The <component> element defines the necessary information to load and to unload the component represented by the element. Its package, name and version attributes define the architecture module the component pertains, the name of the component and its version, respectively. The <component> element may contain one or more <symbol> elements, zero or more <dependency> elements, the <location> element, and zero or more <repository> elements.

The <location> element gives the exact site of the component, which can be remote or local, depending on the value of the type attribute. The uri attribute defines the component address. When a component is loaded from a remote location, the uri contains its new local address, the remote address is set to the uri attribute of the <repository> element.

The <dependency> element allows the Dynamic Evolution Manager to deal with more than one repository of components. In case of local component fails or loses, these repositories can be accessed, as in the case of component updates.

To allow for using the component functionalities after its loading, symbols that can be found during runtime must be defined. The <symbol> element allows for defining the creator, destroyer and object attribute. The object defines the component functionality to be found; its creator and destroyer are set to attributes of same names. The interface attribute allows the ComponentManager to know all objects that implement the same interface. For example, a middleware component can use the ComponentManager to check all platform network interfaces.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT middleware (component+)>
<!ATTLIST middleware platform CDATA #REQUIRED
system CDATA #REQUIRED
version CDATA #REQUIRED>
<!ELEMENT component ( (symbol, location) |
(dependency, location, repository) |
(dependency, symbol, location, repository))>
<!ATTLIST symbol
creatorCDATA #REQUIRED
destroyerCDATA #REQUIRED
interfaceCDATA #REQUIRED>
<!ELEMENT dependency EMPTY>
<!ATTLIST dependency
nameCDATA #REQUIRED
versionCDATA #REQUIRED>
<!ELEMENT location EMPTY>
<!ATTLIST location
uriCDATA #REQUIRED>
<!ELEMENT repository EMPTY>
<!ATTLIST repository
uriCDATA #REQUIRED>
<!ELEMENT repository EMPTY>
<!ATTLIST repository
uriCDATA #REQUIRED>
<!ELEMENT repository EMPTY>
<!ATTLIST repository
uriCDATA #REQUIRED>
```

Figure 3. DTD for XML component descriptions of the Ginga architecture.

The <dependency> element specifies the dependency that a component has of another component (its name and version). This is used by the Dynamic Evolution Manager to update not only a component but also all its dependencies.

The Component Manager and the Dynamic Evolution Manager modules are also useful when applications need components that were not predicted in the original implementation. As an example, Figure 4 shows a fragment of an NCL application in which the content type (DivX), not required in a conformant Ginga implementation [2] [3], is referred for presentation (line 4). To present this application, the Ginga-NCL Player Manager will be called by the Presentation Scheduler (see Section 3.2) to create the DivX player.

In the Ginga monolithic implementation, when there is no player able to display some content type, the Player Manager notifies the failure to the Presentation Scheduler that tries to maintain the presentation temporal consistency. However, in the component-driven implementation, the player must be downloaded from the location specified in the player attribute.
(see Figure 4). A failure is reported only if the player instantiation cannot be succeeded. In the example of Figure 4, the Player Manager calls the Dynamic Evolution Manager to analyze the “videoMM” media object and the corresponding information specified (line 4). From the `<media>` element, it infers that there is a description for a player adapter component addressing the content type with the “avi” extension in the set of component descriptions whose address is specified in the `player` attribute.

```xml
1:  <?xml version="1.0" encoding="UTF-8"?>
2:  <ncl id="unknownContent"
     xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
3:    ...
4:  <media id="videoMM" src="bolinaha.avi"
     player="http://www.gingancl.org.br/plugins/avi.xml"/>
5:    ...
6:  </ncl>
```

Figure 4. NCL application with a non-predicted media type

If the remote XML document describing the component is the one shown in Figure 5, the Dynamic Evolution Manager module then knows that the “avi.xml” file has the DivX player component targeting many platforms and operation systems.

```xml
1:  <component platform="ST7100" system="stlinux"
     version="2.4">
2:    <dependency name="divxplayer" version="1.0.1"></dependency>
3:    <dependency name="gingancladapter" version="0.13.5"></dependency>
4:    <location type="remote"
     url="http://www.gingancl.org.br/plugins/st7100/"
     interface="IPlayerAdapter"/>
5:    <symbol object="DivXPlayerAdapter"
     creator="createDivXAdapter"
     destroyer="destroyDivXAdapter"
     interface="IPlayerAdapter"/>
6:  </component>
```

Figure 5. avi.xml file: remote description of components

The Dynamic Evolution Manager is responsible for analyzing the XML file to get the receiver platform description. In this example the module will get the following descriptions for its platform “ST7100”; the operation system is “stlinux” in its version “2.4” (line 1); the “gingancldivxadapter” component, in its “1.0.1” version, pertains to the “gingancl” module (line 2); the location of this component is given in line 5; the component depends on “divxplayer” (“1.0.1” version) and “gingancladapter”(“0.13.5” version) components (lines 3 and 4), so, all of them must be updated together. The symbols for creating and destroying the “DivXPlayerAdapter” player are also defined in the figure (line 6). In line 9 begins the description of the DivX player for another platform.

As soon as a component and its set of dependencies are downloaded using the Transport or Tuner modules, they are stored in a local memory. Before any operation, the Dynamic Evolution Manager updates the XML document that describes the middleware architecture, adding or updating the new downloaded components, and then notifies the `ComponentManager`.

While new components are being downloaded, the Player Manager keeps the Presentation Scheduler informed that the download process is in progress, and assures that the application presentation continues even if the component downloads fail. It should be stressed that using the solution proposed in this paper the Ginga reference implementation may be updated on-the-fly, and not only when it is halted in background.

The Presentation Scheduler and Player Manager modules assure that download delays do not impair the temporal synchronization among content that compounds DTV applications. For this sake, the Presentation Scheduler keeps a presentation data structure (presentation plan [11]) knowing in advance each content type to be presented and thus commanding the Player Manager to do its job in time. Based on the presentation plan, prefetching of updating components can start to keep the temporal synchronization consistency. However, the current version of Ginga used in the evaluations presented in the next section does not implement an efficient prefetching algorithm yet.

Another issue that deserves a better solution in the current Ginga reference implementation is media player component unloading. The Player Manager should evaluate if a media player component will be reused near in the presentation sequence. In the current version of Ginga, the Player Manager holds a player component idle in memory for one second after it has completed its job. If after this short period of time the player component is still not needed, it will be unloaded by the Player Manager.

V. SOME EVALUATIONS BASED ON MEASURES

This section presents some measures comparing the Ginga monolithic and component-driven implementations (referred as Mono and Comp in all figures, respectively), with regards to resource usage, CPU consumption and delays, both implementations have the same base platform: the Fedora Core 15 distribution of Linux operating system is used, all decoding and rendering processes are implemented in software (there are no specific hardware codecs for any media content type); the DirectFB library [12] is used to handle graphical interfaces, to decode and to render text, images, audio and video content types; media objects with Lua code and HTML code are implemented using public libraries controlled by Ginga itself; the RAM memory was limited to 256 MB, with only 144 MB free memory (112 MB are used by Linux OS and other processes); the swap partition and disk cache policy was disabled.

Our first test document (document A) starts presenting in sequence an image, a text, an object running Lua language code, an audio, and an HTML page. The sequence is repeated twice. In a pure monolithic implementation all players are loaded from the beginning of the document presentation. However, it should be stressed that the DirectFB library implements the dynamic loading of media codecs without any Ginga control. So, the Mono version used in the evaluations indeed is not a pure monolithic case. To be more realistic, the Comp version presenting document A should be compared with the Mono version using document B, which contains the same media objects, but all of them starting at the same time. Note that when we load all media object of the document B, we are not using the dynamic loading characteristics of DirectFB library, which is more real for the Mono version evaluation. Of course the presentation duration of both A and B documents must be the same.
However, although taking care of simulating correctly the component loading of the Mono and Comp versions, our measures are still not precise due to another feature of DirectFB: when a player is loaded it is kept in memory until the end of the application presentation. Therefore, for the Comp implementation version, we are not considering the unloading of the image, text and audio codecs controlled by DirectFB. Therefore, even better results than those presented in this section could be obtained if unloading of those components could be done. As Lua and HTML players use libraries controlled by Ginga, the unloading problem does not happen. Even at a disadvantage, the measures presented have sufficient data for analysis.

It should also be mentioned that the absence of video content in the tests is due to the fact that this media type is usually implemented in hardware in DTV receiver platforms. The decoding and rendering of this content type in software usually implemented in hardware in DTV receiver platforms. Therefore, even better results than those presented in this section could be obtained if unloading of those components could be done. As Lua and HTML players use libraries controlled by Ginga, the unloading problem does not happen. Even at a disadvantage, the measures presented have sufficient data for analysis.

Figure 6. Memory use: max confidence interval equals to 174 kB, with 0.95 of confidence level and sample size equals to 100

Note in Figure 6 that near 0s, the amount of memory required by the Mono version is 3.7 times the one required by the Comp version. This is an important time instant since at this point only the Ginga implementation is in memory. During the [0,02 s, 0.78s] time interval the Mono version loads all DirectFB functionalities needed to run document B, while the Comp version loads only the components (see Section 3) needed to start the NCL application. When the first media content starts its presentation at 0.78s, the difference between the Comp version and the better case for the Mono version (Mono-A) is 14567 KB. At t=9.78s the HTML player is loaded and this is the moment at which the difference between the two versions is the least, remembering that DirectFB does not allow player unloading. This fact explains why at t=10.78s, t=13.78s, t=25.78s and t=28.78s we have a considerable reduction of memory use by the Comp version, since the HTML and Lua players are not controlled by DirectFB, but by the Ginga Component Manager, and these components are unloaded at these moments.

As it is predictable the efficiency in memory use is very high. Figure 7 confirms that this efficiency doesn’t happen at great processing costs. Indeed, the Comp version demands less CPU use than Mono-A and Mono-B. Note the difference in CPU use required by the best case of the Mono version and the Comp version due to instantaneous loading of all Ginga libraries required by the Mono version. It is very important to note that the loading and unloading process do not demand relevant augments in CPU usage in the Comp version due to the Ginga component model, developed based on services directly provided by the operating system, that is, without using any software infra-structure for component-driven design.
VI. FINAL REMARKS

One of the main requirements, if not the most important one, in the design and implementation of a DTV middleware is to take into account the scarce resources of receiver platforms. Another important requirement is the support to constant middleware evolution, allowing fast, secure and easy updating procedures. The use of component-driven implementations plays an important role in this context, since it provides a high degree of adaptability and of computing resource control.

One of the main contributions of this work is to show how component-driven techniques can be used in DTV middleware architectures and implementations, presenting the Ginga implementation as an example. The proposal presented in this paper allows for DTV middleware updating during application runtime, without putting at risk the presentation in exhibition. The component-driven implementation favors the minimum waste of memory and CPU resources.

Ginga-NCL presentation engine, as other DTV declarative environments, allows for building a presentation plan (a hypermedia temporal graph - HTG) that allows for predicting when each media content type player is needed. An intelligent prefetching algorithm should be devised to not impose any unnecessary memory burden of having a component in memory for a time longer than needed. Intelligent prefetching algorithms are in our future working plans.

We plan also to continue with performance measures using commercial receiver platforms, possibly without the limitations we have had (like the previously mentioned DirectFB characteristics) and possibly with other limitations coming from their hardware/software scarce resources. We intend to extend the evaluations to also take into account swap partition use.

REFERENCES