Supporting Context Events in an Interactive TV Platform[*]

Izon Thomaz Mielke[1], Patrícia Dockhorn Costa[1], Igor Magri Vale[1]

[1]Departamento de Informática – Universidade Federal do Espírito Santo (UFES)
Av. Fernando Ferrari 514 – Vitória – ES – Brasil

{izomtm, igormagrivale}@gmail.com, pdcosta@inf.ufes.br

Abstract. *This paper aims at investigating functionality of the middleware Ginga that can be leveraged to build context-aware interactive TV applications. We identify the elements of the middleware that can effectively be used to realize each of the particular aspects of context-awareness, namely capturing, processing and reacting upon context changes. We present a prototype application in order to illustrate how these elements have been identified and how they can be used. We also define a programming framework that facilitates capturing and distributing context events in this interactive TV platform.*

1. Introduction

Context-aware applications use and manipulate context information to detect the situations of their users and adapt application behavior accordingly. Since context-awareness allows applications and services to become tailored to the user's current situation and needs, several areas of research consider context-awareness as an important tool to enhance user experience [Dey 2001].

In the interactive television domain, in particular, context information has been used in various types of applications [da Silva et al. 2009][Thawani et al. 2004], ranging from TV content adaptation (based on user's preferences) to home banking applications (selectively displaying certain private information based on presence). In a digital television application, for example, context information can be used to detect presence of children in the living room, which can be used to change the behavior of the TV to only display adequate ("child safe") content.

However, the development of context-aware applications is a challenging task. Context-aware applications, in general, should be capable of: (i) sensing context from the environment, (ii) observing, collecting and composing context information from various sources (often sensors), (iii) autonomously detecting relevant changes in the context, and (iv) reacting to these changes, by either adapting their behavior or by invoking (composition of) services.

Since context-awareness is a relatively novel technology in the scope of the Brazilian Interactive TV Platform (SBTVD), it is still not clear whether the support offered by the platform is adequate for the development of context-aware applications. In this sense, this work investigates the functionality of the platform that can be leveraged

[*] This work is partially sponsored by the project HR-TVD # 225/2008 funded by CAPES. Igor M. Vale and Izon T. Mielke are supported by CAPES and FAPES, respectively.

to build context-aware interactive TV applications. We identify in this paper the elements of the platform that can effectively be used to realize each of the particular aspects of context-awareness, namely *capturing*, *processing* and *reacting* upon context changes. In particular, in order to support *capturing context events*, we have leveraged the general event framework provided by the SBTVD with a programming framework that facilitates capturing and distributing *context events*. We further present a prototype application in order to illustrate the development of a context-aware application using the framework we have proposed.

The paper is further structured as follows: Section 2 introduces a context-aware application scenario; Section 3 presents the Brazilian Digital TV System (SBTVD); Section 4 discusses how context can be captured and handled in a SBTVD application; Section 5 proposes a programming framework that facilitates context-aware development in the platform; Section 6 presents the context-aware application prototype implementation; Section 7 discusses this work on the light of related work and, finally, Section 8 presents some concluding remarks.

2. Context-Aware Application Scenario

We have used throughout the paper a running example in order to illustrate how context-aware requirements can be fulfilled by elements of the Ginga platform. We have chosen a representative application scenario that covers the most important requirements of a context-aware application, namely capturing, processing and reacting upon context information changes. The scenario storyline is as follows:

In exhibition events, it is common to find TV sets showing demonstrative videos of some product. These types of demonstrations can present a problem: the lack of audience. This occurs mainly because the TV only shows a video in loop, which rarely sparks curiosity to get people's attention to stop and watch the video about the product. In a tentative to solve this problem, we propose a context-aware application that utilizes information about audience (obtained by means of a sensor that detects presence) in order to control the video contents exhibited on the TV screen. While there is no sufficient public watching the video, an initial short clip is shown in order to spark public's attention and to attract other participants:

"Certain things cannot be done alone. Watching this video is one of them! Invite your friends and find out why."

As soon as the required audience is achieved, the product's presentation is initiated with the guarantee that a minimum number of persons is watching the video.

3. The Brazilian Digital TV System (SBTVD)

The Brazilian Digital TV System (SBTVD) implements a middleware platform, coined Ginga [Telemídia 2007], which supports the development and execution of interactive applications in the SBTVD environment. Interactive applications can be developed upon the Ginga middleware using a declarative language, called NCL, or using Java. Supporting Java application development in Ginga is optional for portable devices. Given the mobile nature of context-aware applications, we have chosen to concentrate our efforts on investigating the particularities of the declarative environment of the middleware (NCL in combination with NCLua scripts).

NCL (Nested Context Language) is a hypermedia authoring language that was originally developed to describe multimedia applications with space-time synchronization between media objects (e.g., video, audio, images, etc.). As such, it does not have built-in concepts for context-handling and reactivity. In order to implement these concepts in NCL, we have identified a number of NCL document elements, which together with certain NCLua scripts provide the required support. Next sections discuss each of these elements.

4. Handling Context Events in the Ginga Platform

In order to obtain context information from a sensor, an NCL application should be able to access a remote device through a network. For that, the Ginga middleware offers a mechanism called Interactivity Channel. The exchange of data can be performed on the Interactivity Channel by means of NCLua scripts, which are part of the NCL application. These scripts are responsible for managing connections, exchanging and processing data and keeping the NCL document up-to-date with respect to context information received from remote devices.

The communication among NCLua scripts and other components is realized by means of events. The NCLua mechanism for event dissemination can be compared to the publish/subscribe pattern [Eugster et al. 2003] in which there is a mediator managing the exchange of events between producers and consumers. This type of event architecture allows space-time decoupling among system parts.

In order to realize exchange of events, NCLua scripts use a module (mediator) called *event module* [ABNT 2007]. Main functions of this module are generating an event, by means of a *post* method, and receiving an event, by means of registering event handlers using the *register* method. NCLua events are categorized by classes [Sant'Anna et al. 2008], each responsible for specific functions, such as: class "ncl", responsible for communicating with the NCL document, and the class "tcp", responsible for communicating with the Interactivity Channel.

Since we would like to gather context from a remote device and we also would like to keep the NCL document up-to-date with respect to context information, we are particularly interested in event classes "tcp" and "ncl", respectively. Following sections elaborate on how to use the standard event API to implement a simple NCL context-aware application.

4.1.Communicating with External Devices

As mentioned in previous sections, the communication between NCLua scripts and the Interactivity Channel is realized by means of events (class "tcp"). As in TCP Sockets, the exchange of data occurs after a connection has been established between a client and a server.

In order to request for a connection, an NCLua script should send out an event of type "connect", informing the server (host) and the port with which to connect to. As an answer to this event, the event API generates an event of type "connect" containing the identification of the connection, or an error event, indicating the reason why this connection could not be established.

After the connection has been established, data can be exchanged by means of events of type "data". In order to send out data events, the client should inform the identification of the connection. Similarly, events received contain the identification of the connection. An event of type "disconnect" indicates that the connection has been closed.

Figure 1 depicts an NCLua code snippet that shows how an NCL application should communicate with an external sensor (e.g., the Audience Sensor) through the Interactivity Channel, using the standard API. In this code snippet, a "connection" object is used to manage the connection with the Audience Sensor. An event handler object is created and registered with the event module. This handler object is responsible for answering to connection events, as well as data events.

As can be seen in this code snippet, using the standard API for handling context events (from external sources) can get quite cumbersome, since it is necessary to identify the class of the event, the type of the event, to which connection that event belongs to, etc. This is reflected by the long and confusing conditional ("if") statements. In this particular simple example, we are manipulating a single type of context event. When various context events should be supported (more realistic scenario), the number of conditional statements increases proportionally to the number of context types.

```
local connection = nil
function handler (evt)
 if evt.class == 'tcp' and evt.type == 'connect' and
  evt.host = host and evt.port  = port and connection == nil then
    connection = evt.connection
    if connection then
       event.post  {class  =  'tcp',  type  =  'data',  connection  =
connection, value = 'request_audience'}
    end
 end
 if evt.class == 'tcp' and evt.type == 'data' and
  evt.connection == connection then
    verify_audience(event.value)
 end
end
event.register(handler)
event.post {class = 'tcp', type  = 'connect', host = host, port  =
port}
```

Figure 1. NCLua code for communicating with an external device using the standard API.

4.2. Context Representation in NCL

Besides capturing context from remote sensors, NCLua scripts must be able to inform the NCL document that new data has arrived. The communication between NCLua scripts and the NCL documents, in which these scripts are embedded, is realized by means of Property Anchors. Property Anchors can be defined as shared attributes between the NCL and NCLua environments and can be used to represent context information in an NCL document.

In order to change Property Anchor values, an NCLua script should send out an event of class “ncl” (of type “attribution”) informing which property to be changed and the new value to be assigned. The assignment of Property Anchor values is a two phase activity: first, a start event should be generated in order to initiate the assignment; and then, a stop event should be generated, which ends the assignment.

Figure 2 depicts a code snippet showing how to assign a Property Anchor value, using the standard API. The function verify-audience is invoked every time audience events arrive from the Audience Sensor. If audience is sufficient (defined as “value>10”), the respective Property Anchor in the NCL document should be informed. Again, the developer must manipulate events and their attributes explicitly and can get cumbersome when various types of context events are supported.

```
function verify_audience(value)
 if tonumber(value) > 10 then
  evt = {class = 'ncl', type= 'attribution', property = 'audience', 
action = 'start', value = 'sufficient'}
  event.post(evt)
  evt.action = 'stop'
  event.post(evt)
 end
end
```

Figure 2. NCLua code for controlling context variables.

5. Programming Framework

Handling events for both, controlling connections and managing the mechanisms to keep Property Anchors up-to-date, are basic requirements for most NCL context-aware applications. We have defined and implemented a programming framework that can be used to ease the development process by solving many of the issues related to event handling in the Ginga middleware.

As part of the framework, we have developed the following NCLua libraries: TCPEventHandler and Properties, both available online at [http://code.google.com/p/itveventframework/]. The TCPEventHandler library provides reusable code for facilitating the communication between NCL applications and external devices, while the Properties library provides code to help managing Property Anchor values in NCLua scripts. Given the asynchronous nature of context-aware applications, we have based the development of the framework using classical design patterns for handling asynchronous events, as discussed in the next sections.

5.1. TCPEventHandler: Communicating with External Devices

The TCPEventHandler library adopts concepts of object-oriented design in order to manage connections and their data, since the standard API does not offer this facility. The TCPEventHandler library encapsulates functions to handle connections, errors and data exchange so that the developer does not need to worry about details of the standard mechanism.

In order to open a connection with a remote device through the Interactivity Channel, an NCL application developer using the TCPEventHandler library should make use of a Connection object, which is offered by the library. As opposed to explicitly sending/receiving events, this object abstracts from the concepts of NCLua events, by defining methods to open/close a connection and exchange data. The Connection object uses a specific event handler object (Handler) to react upon notifications of connection success, disconnection, data exchange and communication errors.

The design of this library was based on the Reactor Pattern (or Dispatcher) [Schmidt 1994]. Internally, the library manages all current connections by means of an event handler method registered with the NCLua event channel, which handles all events disseminated in the Interactivity Channel (of class "tcp"). Upon receiving an event, the library handler method identifies the Connection object to which this event belongs to and invokes (on the event handler specific of that connection), the method responsible for handling that event.

In order to create a connection to a remote device using the TCPEventHandler library, the NCL application should instantiate an object of type Connection in an NCLua script informing, as arguments, the host address and port of that remote device, and an event handler specific for managing the events of that connection.

5.2. Properties: Communicating with the NCL Document

There is no direct correspondence between Property Anchors of an NCL document and variables defined in NCLua scripts. We have implemented the Properties library, which is capable of managing context variables by means of methods *set*, *get*, *register* and *unregister*. For example, when an NCLua script needs to inform the NCL document that a particular context value needs to be refreshed, the script invokes the set method passing as arguments the name of the Property Anchor and new value. Similarly, to get a property value from the NCL document to the script, the Properties library offers the get method.

6. Prototype Implementation

We have implemented a prototype for the scenario proposed in Section 2 using the programming framework presented in Section 5. Prototype code and execution examples are available at [http://code.google.com/p/itveventframework/].

In the implementation, context information about audience is offered by a remote sensor (camera), which should be able to detect presence of persons watching TV. Audience information is constantly monitored to control the exhibition of the video contents: if the audience is sufficient, the TV should display a Product Video; otherwise, the TV should display a Home Video. To implement face recognition, we have used an OpenCV library, which provides an algorithm for object detection [Viola and Jones 2001].

Figure 3 depicts a high-level architecture of the prototype. The NCL Application communicates with the Audience Sensor by means of the TCPEventHandler library. Internally, the NCL Application is composed by an NCL document (declarative part) and NCLua Scripts. The communication between those two is realized by means of the Properties library.

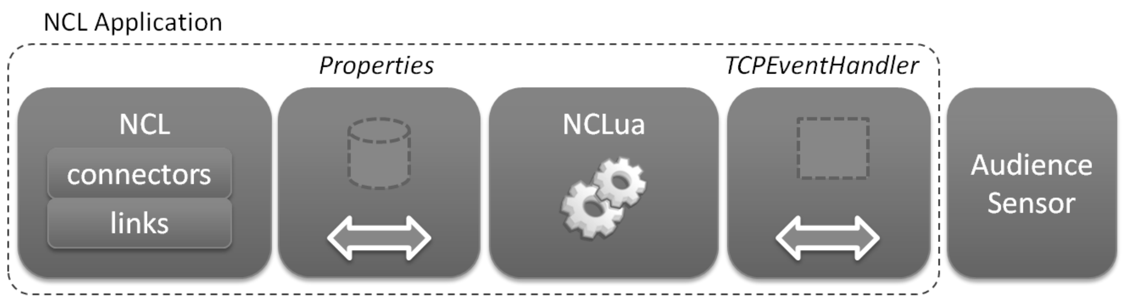

Figure 3. High-level architecture of the prototype.

Figure 4 depicts the content node (media element) that represents the NCLua script (“monitor.lua”) in the NCL document, which defines a single Property Anchor (called “audience”).

```
<media id=”monitor” src=”monitor.lua”/>
   <property name=”audience”/>
</media>
```

Figure 4. Representation of the NCLua script in the NCL document.

Figure 5 shows the “monitor.lua” script, which loads the “TCPEventHandler” and “Properties” libraries, builds an event handler object (handler) and associates this object to a connection object (Connection), which represents the connection with the Audience Sensor.

The event handler object handles events by means of methods of this object. For example, when the TCPEventHandler library receives a data event from the event channel, it first verifies to each connection this event belongs to and then dispatches the event to proper consumers by invoking the *handle_receive* method of that particular handler. In our example, the *handle_receive* method invokes the *verify_audience* method, which verifies whether the audience has achieved a certain value. The method *verify_audience* should be invoked every time audience information arrives from the Audience Sensor.

Comparing Figures 1 and 5 (part 2), we can see that the code produced using the library (Figure 5) does not need to be overloaded with conditional statements, since most of these statements are shifted to the library. Therefore, the code is cleaner, easier for the object-oriented developer to write and understand, and it is also easier to extend with other types of context events.

When the audience information is sufficient, the *verify_audience* method invokes the set method (of Properties library) in order to indicate to the NCL document that the “audience” Property Anchor value should be now “sufficient’.

Again, comparing Figures 2 and 5 (part 1), we can see that great part of the code is shifted to the library, which makes the code cleaner and easier to understand.

```
require 'TCPEventHandler'
require 'Properties'

function verify_audience(value)                          1
    if tonumber(value) > 10 then
        Properties.set('audience', 'sufficient')
    end
end

handler = {}                                              2
function handler.handle_connect (self)
    self.connection.send('request_audience')
end
function handler.handle_receive (self, data)
    verify_audience(tonumber(data))
end
audience_connection =
  TCPEventHandler.Connection(host, port, handler)
audience_connection.connect()
```

Figure 5. NCLua code of prototype using the libraries.

Our context-aware application exhibits the following behavior: if audience is sufficient, the TV should start showing a Product Video, otherwise, a Home Video should be displayed in a loop until audience is sufficient. This reactive behavior has been implemented using “connectors” and “links”, both elements of the NCL language. Figure 6 depicts the connector element, which defines that at the end of the Property Anchor assignment (onEndAttribution), if the value assigned to this anchor is “sufficient”, a content video should be started (start). A link element links the behavior defined in the connector to the Property Anchor "audience" and the Product Video.

```
<causalConnector id="onAudienceTest">
 <compoundCondition operator="and">
  <simpleCondition role="onEndAttribution"/>
   <assessmentStatement comparator="eq">
    <attributeAssessment     role="compare"     eventType="attribution"
attributeType="nodeProperty"/>
    <valueAssessment value="sufficient"/>
   </assessmentStatement>
 </compoundCondition>
 <simpleAction role="start" max="unbounded"/>
</causalConnector>
```

Figure 6. Connector implementing the application’s reactive behavior.

6.1. Considerations

The development of NCL context-aware application prototypes has allowed us to identify particularities of the middleware Ginga that can be used to implement the most important requirements of context-aware applications (capturing, processing and reacting upon context changes). For example, we have identified the following:

- Capturing context from remote sensors can be realized by means of the Interactivity Channel, which allows an NCL application to communicate with remote devices over a network. In order to facilitate context event handling and distribution, we have proposed the TCPEventHandler Properties libraries;

- Processing context may be partially performed by the NCL document (by means of Connectors) and partially by NCLua scripts. We have also shown that context information can be represented by means of Property Anchors in NCL;
- Reacting upon context changes can be realized by means of the connector element, which allows the specification of causality rules that related NCLua scripts, Property Anchors, videos and invocations to external services.

7. Related Work

Research on context-awareness in the scope of interactive digital TV is usually associated with specific types of applications (and context information), such as in [da Silva et al. 2009], [Leite et al. 2007] e [Thawani et al. 2004]. The PersonalTVware [da Silva et al. 2009], for instance, proposes a context-aware personalized recommendation architecture for the interactive TV environment. This architecture supports specific types of context, such as user profile, and history of watched TV programs. In [Leite et al. 2007] it is proposed an architecture to determine the level of adequacy of TV contents according to the user's context (mostly computational context only). For example, it is possible to assess whether a certain content is adequate to the format of the device's screen. The approach presented in [Thawani et al. 2004] proposes an architecture, which selects and inserts, at real time, context-aware commercials to the broadcast flow transmitted to the TV. For that, user's profile and history are considered as context information.

As can be seen, most related work use context information in the scope of a single domain (e.g., personalization of content using user's profile and history). None of the approaches present proposals that facilitate capturing context from remote sensors and reacting upon context changes. Our work focuses on proposing mechanisms to support capturing and handling distributed context information, independent of the type of context being supported.

8. Concluding Remarks

Context-awareness is a relatively novel technology in the scope of the Brazilian Interactive TV Platform. Therefore, there is still no adequate methodology for developing context-aware applications, which incorporates the most important requirements for manipulating context information (capturing, processing and reacting upon context changes).

We have presented in this paper the design and implementation of a context-aware application prototype, which allowed us to identify existing functions of Ginga-NCL that can be used to facilitate the development of context-aware applications. In contrast to other approaches, our work does not intend to extend the Ginga middleware, which is already defined as part of the Brazilian DTV standard.

During the development process, we have also identified that the mechanisms for context event handling in the Ginga middleware is cumbersome. In order to facilitate handling context events, we have implemented a programming framework, which hides event handling complexity easing, therefore, context-aware development process.

As part of future work, we intend to incorporate programming facilities into a Context Management Infrastructure to support acquisition and manipulation of context information from various sources of heterogeneous nature. We plan to use the programming framework proposed in this paper as bases for the development of this infrastructure, since they are generic to support various types of context and, therefore, facilitate the development of distributed context-aware applications in various application domains (e.g., telemedicine, personalization, gaming, etc.).

References

ABNT. (2007). ABNT NBR 15606-2:2007: Digital Terrestrial Television Standard 06: Data Codification and Transmission Specifications for Digital Broadcasting, Part 2 – GINGA-NCL: XML Application Language for Application Coding.

da Silva, F. S., Alves, L. G. P. and Bressan G. (2009). PersonalTVware: A Proposal of Architecture to Support the Context-aware Personalized Recommendation of TV Programs. *European Interactive TV Conference (EuroITV 2009)*, Leuven.

Dey, A. K. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing*, pages 4-7.

Eugster, P. T., Felber, P. A., Guerraoui, R. and Kermarrec, A. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35, 2, pages 114-131.

Leite, L. E. C., Lima, O., Filho, G. L. S., Meira, S. R. L. and Tedesco, P. C. A. R. (2007). Uma Arquitetura de Serviço para Avaliação de Contextos em Redes de TV Digital. *In Anais do 25º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos.* Belém, PA, pages 1015-1028.

Sant'Anna, F., Cerqueira, R. and Soares, L. F. G. (2008). NCLua – Objetos Imperativos Lua na Linguagem Declarativa NCL. Simpósio Brasileiro de Sistemas Multimídia e Hipermídia (Webmedia 2008), Vila Velha. In *Anais do XIV Simpósio Brasileiro de Sistemas Multimídia e Hipermídia*, v.1, pages 67–74.

Schmidt, D. C. (1994). Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching. In *Proceedings of the First Pattern Languages of Programs conference in Monticello*, Illinois.

Souza Filho G. L., Leite, L. E. C. and Batista, C. E. C. F. (2007). Ginga-J: the procedural middleware for the Brazilian digital TV system. *Journal of the Brazilian Computer Society*. vol.12(4), pages 47-56.

Telemídia, L. (2007). *Ginga Digital TV Middleware Specification* Source: http://www.ginga.org.br.

Thawani, A., Gopalan, S. and Sridhar, V. (2004). Context Aware Personalized Ad Insertion in an Interactive TV Environment. In *Proceedings of Workshop on Personalization in Future TV.*

Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. *In Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition*, pages 511–518.