Loading Time Optimization in Broadcast TV Applications

Jean-Claude Dufourd
Telecom ParisTech ; CNRS LTCI
37-39 rue Dareau
75014 Paris, France
+33145817733
dufourd@telecom-paristech.fr

Jean Le Feuvre
Telecom ParisTech ; CNRS LTCI
37-39 rue Dareau
75014 Paris, France
+33145817169
lefeuvre@telecom-paristech.fr

Jean-Claude Moissinac
Telecom ParisTech ; CNRS LTCI
37-39 rue Dareau
75014 Paris, France
+33145818088
moissina@telecom-paristech.fr

ABSTRACT

HbbTV, an emerging standard for interactive television, is being deployed in various European countries, creating bandwidth demand for interactive services in existing saturated broadcast networks. The problem of managing often-reused resources, such as script libraries, in broadcast interactive applications, becomes acute when trying to find a good compromise between bandwidth and waiting time. This paper explores the issues that TV channels have with on-air interactive applications and their optimized delivery. More specifically, this paper addresses the issue of data caching between broadcast carousel sessions as a way to optimize the bandwidth consumption of interactive applications. Proposed extensions to the current HbbTV specification to help solve the problem are given, and typical results include significant reduced waiting time, up to a factor of 3. The implementation complexity and other potential applications of the proposal are also discussed.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Commercial services, Web-based services;* H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems; H.5.4 [**Information Interfaces and Presentation**]: Hypertext / Hypermedia – *User issues*

General Terms

Algorithms, Experimentation, Standardization.

Keywords

HbbTV, interactive TV applications, resource management in broadcast.

1. INTRODUCTION

HbbTV (Hybrid Broadband Broadcast TV), a standard targeting interactive TV, is getting more and more traction since its inception in 2009 and standardization by ETSI in June 2010 [1]. HbbTV is actually a conformance, certification and promotional effort as well as an industry standard adding interactive functionalities to TV by aggregating and profiling existing standards such as CEA for CE-HTML, MPEG for video, audio and transport, DVB for signaling, OIPF (Open IPTV Forum) for APIs to manage the link between the interactive HTML and the various subsystems, etc. It builds on well-known and proven technologies such as HTML, CSS and JavaScript to create interactive applications.

This standard differs from past standardization efforts in that its scope includes both broadcast (i.e. unidirectional link) and broadband (i.e. bi-directional link) delivery mechanisms. Applications can be delivered either in broadcast within a DSM-CC carousel [2], or in broadband by HTTP over the Internet. A carousel is a one-to-many distribution technique for the repeated transmission of a set of resources, usually a hierarchy of files, so that a client can start receiving data at any time and get access to any resource after a maximum waiting time; this time is usually referred to as the carousel period. Note that when a single carousel carries several resources, a different carousel period can be applied to each resource. Well-known examples of data carousel protocols are DSM-CC and FLUTE [3].

In the process of helping with the French roll out of HbbTV for TNT 2.0, the name of the future version of DVB-T in France, (promoted by the HD Forum), we have designed TV applications, tested applications designed by third parties, developed application validators, generators and other tools in the application production tool chain for HbbTV. During these developments, we faced a recurring problem with HbbTV applications in the management of resources. In a broadband environment, the browser, as mandated by the HTTP specification naturally caches reused resources; in other words, resource retrieval optimization happens automatically in broadband. In broadcast however, there is no mechanism available to enable caching of carousel objects between sessions. As a result, when broadcast of interactive applications is an important use case, like in France for TNT2.0, all resources need to be resent every time they are used, and no optimization is currently possible.

In this paper, we intend to optimize the waiting time of viewers when tuning into an HbbTV program with an interactive application by proposing a carousel organization and a caching scheme for script libraries. The rest of this paper is organized as follows: Section 2 provides an analysis of carousel mechanisms and explains the problem of data scheduling through the analysis of a real-world HbbTV application. A proposed solution to this problem is given in Section 3. Section 4 gives some result achieved with the given proposal, along with further considerations. Section 5 concludes this paper.

2. OF CAROUSELS AND CACHES

2.1 Carousel Basics

When carouseling data over a broadcast link, the main problem to solve is how to optimize the time taken for the data to be received at a given bandwidth, hereafter called *waiting time*. As shown in [7], usage of a carousel with multiple speeds often helps with the compromise between waiting times and the amount of bandwidth allocated for interactive applications. For example, the set of resources to be transmitted in the carousel is split into three subsets: resources that need to be present before the application can start, resources that are important for the application, and resources that are nice to have (see [6] and [7] for a more detailed discussion of classes of resources to take into account for interactive services). The first set would be repeated at a certain period, the second set would be interleaved with the first set but with a slower repetition period, and the third set would be interleaved with the other two when bandwidth is available, with an even slower repetition period.

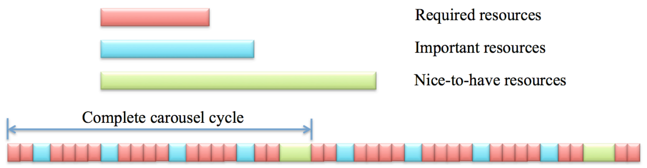

Figure 1: Multi-period carousel

The above figure describes a simplified situation where resources are sent in one piece, required resources are sent 4 times more often than important resources, which are sent 4 times more often than nice-to-have resources. In practice, resources of all types are cut into smaller sections and some interleaving reduces the impact of the variance on resource length.

For an HTML/CSS/JS-based application, as is the case in HbbTV or MHP (DVB-HTML), JS libraries required to construct the page would have to be allocated to the faster cycle to reduce the waiting time, so their impact on the required bandwidth is maximal.

2.2 Overheads in HbbTV Applications

In our experience, among the applications used within a certain TV channel, most of them come from a small number of application providers, in general specialized design studios. The applications coming from the same studios are usually designed the same way and share many JS libraries: jQuery, mootools, yui, dojo are such often used libraries, on the Internet as well as on mobiles and now on TVs. Using those libraries helps with blurring the difference of browsing quality of experience between TVs and PCs, which is an important need for TV channels. However, these libraries are sent with each application, thereby wasting precious bandwidth and potentially increasing the waiting time of the carousel.

Another source of duplication among the applications of a particular channel is the graphics charter of the TV channel: logo, background, buttons, styles, etc. However, since the application look and feel is often tuned to that of each broadcasted show, this may in practice be a smaller source of optimizations.

There are easy strategies to alleviate the impact of waiting for images; for example, using two images in the HTML document, a reduced quality version sent at a higher frequency in the carousel, which can therefore be displayed quite fast, and a full quality version sent at a lower frequency, whose availability is monitored through *onLoad* events in HTML <*img*> element.

There is no simple equivalent strategy for the JS libraries or most programmatic elements (e.g., Java Classes). Indeed, application authors tend to rely on existing libraries, well optimized for Web infrastructure but usually conceived as a monolithic piece of code: these libraries cannot be easily divided into a small initialization part for document preload, then a more advanced sub-library for document structuring and finally the rest of the library for visual effects, and are therefore not suitable for multi-period carousels.

Providing a caching mechanism for those resources, allowing them to be sent once in a while rather than within each carousel at high frequency, will greatly help in reducing the average waiting time and the management of bandwidth allocated to interactive applications.

2.3 A Real-Life Use Case

We base our analysis on a real-life HbbTV application from France Télévisions, available at the time of writing on all their channels. It includes text and images for general news, sports news, weather forecast and ads for various future broadcasts. This application intends to replace the older teletext system, and as such targets all receivers, whether connected (broadband) or not; this implies that all the application data is carried inside the broadcast link.

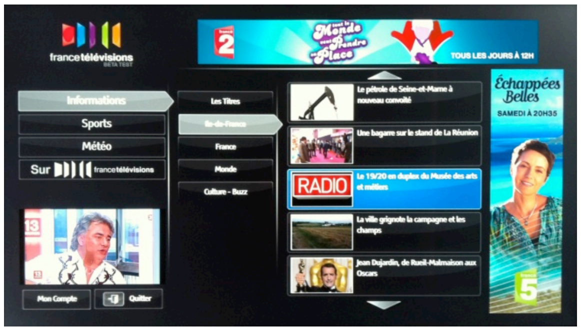

Figure 2 - Live France2 HbbTV application

The total size of this application is 550kb, of which 430kb are images and 113kb are scripts. Transmitted in a uniform carousel with 100kbit/s, where all resources are sent once per cycle, and counting 10 bits sent per byte to send to account for overhead, the maximum waiting time for any resource is 57s. In a carousel with images transmitted at a 10^{-1} priority, i.e. all other resources are repeated 10 times more often than images, the maximum waiting time would be 17s for other resources and 3mn for images. With scripts transmitted at a 10^{-1} priority than required resources, the maximum waiting time for required resources goes down to 5s.

This means that the very first time a TV uses a particular resource, it has to wait a maximum of 50s. For following accesses to the resources (i.e. after a tune out from the channel), the maximum waiting time would drop from 17s to 5s if scripts were cached. In terms of bandwidth, to achieve a similar gain of 3.4, we need to dedicate 340kbits/s instead of 100kbits/s to the carousel. Note that this simple caching would not change the waiting time of the first tune-in of the clients, as they still have to fetch the required resources with long carousel periods.

2.4 Caching in Broadcast

Caching is often mentioned in relation with DSM-CC carousels, e.g. in [5] as well as in [1] for the minimum DSM-CC cache size, but with a different focus. The caching starts when the carousel is accessed, and stops when the carousel is closed. Such caching improves the waiting time by anticipating requests for a particular resource: as soon as a carousel is open, received resources are stored in cache even if the application did not request them yet. When the request comes, if the resource is already in cache, the response is immediate. We refer to this mechanism as *intra-carousel caching*. This looks very good in theory, but does not solve our problem in practice. As shown previously, we need a persistent cache, such that JS libraries could be stored locally after closing the carousel and fetched back from the cache when re-opening the application carousel. We refer to this persistent cache as *inter-carousel caching*.

The DSM-CC object carousel mechanism, much like many tools used for file delivery over unidirectional links, does not provide for caching across sessions: when a session is no longer running, the transmitted files are no longer required and may be deleted. More importantly, identifiers used during the session are usually not meaningful outside of this session. Flute, used for unidirectional delivery of files over the Internet, provides similar carousel tools, as shown in [4]. It has however the interesting property of using URIs as file identifiers, and a useful possibility is to actually use a located URI as identifier, usually an HTTP URL. This design implies that:

- resource identifiers are independent from the carousel session, and can be interpreted once the session is killed,
- a connected device has always the choice of waiting for the resource to arrive over Flute, or to bypass waiting and get it from the Internet.

The first feature will be at the heart of our proposal; the second feature is a direct consequence of using, as an URI, an URL (e.g. HTTP URL) that resolves to a valid resource on the Internet network.

3. Inter-Carousel Caching

3.1 Design Principle

Inter-carousel caching is our answer to the aforementioned limitations: it allows sending data once for several applications, thereby avoiding duplication, and allows for sending application data before the application is loaded, thereby reducing loading times. Such a carousel can be sent whenever available bandwidth allows for it, preferably before the start of the application carousel if reducing the load time is the desired effect.

We want our solution to be as transparent as possible for the content creator (minimal or no changes in content), while providing an efficient cache for applications relying on broadcast and broadband networks. As said previously, we need a way to identify carousel resources even when the carousel is no longer received; our proposal uses URLs as resource identifiers, which gives us the possibility to identify network resources as well.

Finally, we would like our solution to provide a generic HTTP cache push system for any application, HbbTV or not. We have in mind usage of Push-VOD services relying on HTTP as transport mechanisms, such as HTTP Streaming solutions (e.g. MPEG-DASH).

3.2 Simple HbbTV Cache Proposal

Our first design is a very simple extension to the HbbTV framework, achieved by using a new attribute on elements using external resources, such as *<script>*, *<link>*, *<img>*, *<video>*. Let us take an example of a resource:

```
<script src="js/jquery.js" type="text/javascript" …/>
```

In our proposal, the resource would be marked as cachable as follows:

```
<script src="js/jquery.js" h:cacheName="jQuery142" .../>
```

The implementation would process the extended *h:cacheName* first, look for a matching resource in the cache, and if there is no such resource in the cache, request *js/jquery.js* from the carousel, then store the returned resource in the cache before handing it back to the browser. The above example assumes the definition of the attribute *h:cacheName* in another namespace, following strict XML extension rules. The value of *h:cacheName* is the global identifier for the resource, valid across sessions, channels and multiplexes. It may correspond to the official download URL of the resource if appropriate.

This proposal is quite simple and efficient, but it has some limitations:

- It requires some changes in the content.
- It cannot be used for sharing resources across applications from different providers for obvious security reasons: libraries could be overwritten with non-compatible or malicious versions, without any possibility to prevent it apart from keeping the caches separated between applications, or at least service providers.
- It cannot be used to cache data not referenced in the HTML page.

3.3 Generic HTTP Cache Proposal

To overcome these limitations, we designed a second proposal, which consists in sending the cache instructions together with the objects to be cached, in a regular object carousel. The carousel root directory contains a file labeled *cache.xml*, which provides an XML description of each item that needs to be cached in this carousel:

- name of the resource in the current carousel e.g. *jQuery142* in our previous example,
- for each carousel referring to this resource:
    - o the name of the referred resource e.g. *js/jQuery.js* in our previous example,
    - o optionally the DVB triplet identifying the referring service; this solves naming conflicts, when different resources, cached or not, are used with the same name in different channels.
- optionally, some HTTP header fields for cache directives, as a server would have replied if the resource were queried with an HTTP GET method.

The *cache.xml* file can be directly included in the application carousel, in which case caching is restricted to the current service provider. If caching needs to be shared across applications from different providers, a dedicated DSM-CC carousel is declared in one PMT, with a specific AIT type (currently a reserved one) and the AIT *autostart* flag set. Similar to DVB System Software Update, we force the device to process such a carousel as soon as it has the possibility. The proposal is naturally backward compatible with existing implementations, since the *cache.xml* file can safely be ignored; moreover, the proposal is very light, the cache file being quite small (potentially gzipped by DSM-CC) compared to other files in the carousel.

This second proposal allows the description of caching instructions of files in a carousel, possibly not tied to any particular TV channel. While more complex, this approach has the benefit of not touching the HTML part of the application, only requiring modification of the HTTP stack in a standard way, using the HTTP cache header fields given in the XML file. It can also be used to cache files not referenced by the current HTML application and with non-HTML applications.

4. Results and Analysis

4.1 Gains

Table 1 describes some more results of applying our proposals to 4 applications. The results are valid for any of our proposals, since we made sure not to break existing implementations by always including a long-period carousel for the required resources in each application. The first application is the one described in Section 2. The others are applications designed by different agencies for different customers. All are already on air in France at the time of writing. Although we studied more than 20 sequences using external JS libraries, we only show the most relevant applications.

Name	Nb res	Total Size	Scripts	Images	**Max wait A**	**Max wait B**	**Max wait C**
FTv	**147**	**576k**	**113k**	**451k**	**57s**	**17s**	**5s**
MS	**30**	**391k**	**46k**	**334k**	**39s**	**9s**	**6.8s**
CE	**14**	**169k**	**40k**	**126k**	**17s**	**5.6s**	**2s**
Go	**30**	**395k**	**43k**	**346k**	**40s**	**8s**	**4.4s**

Table 1: Waiting times for a bandwidth of 100kbits/s

The measurements were made using our HbbTV tools for bandwidth management developed in the openHbb project (www.openhbb.eu). All resources except images have been gzipped, which is a feature of DSM-CC modules. The **Max wait** columns describe the maximum time for the application to appear, even if it is incomplete. **Max wait A** is the waiting time when all resources are repeated once per cycle, regardless of their type: for the first application, the maximum waiting time is around one minute, but then the application is complete after 1mn. **Max wait B** is the waiting time when images are repeated ten times less often than other resources: it implies that the first application appears on the screen with no images in a maximum of 17s, and images appear gradually for the next 3mn. **Max wait C** is the waiting time when images and scripts are repeated ten times less often than other resources: if the scripts are in cache, the first application arrives after a maximum wait of 5s, then the images appear within the next 50s; if the scripts are not in cache (first time) then the maximum wait is for 50s. The improvement from using our proposal ranges from 25% to 70%, while maintaining reasonable maximum waiting times for cache misses. Our experiment showed that using a longer period (priority lower than 10^{-1}) for scripts does not help the overall waiting time much: an improvement of the waiting time of less than 10% is observed when scripts are in the cache, with the drawback of a first time wait increasing from 50s to 9mn for application FTv.

4.2 Discussion

The implementation requirement of our first simple HbbTV proposal is very small, affecting the resource loader of the HTML browser to hand over the value of the new attribute *h:cacheName* to the carousel manager of the device, which handles the caching. This caching can be done independently of the HTTP cache of the device, as the resource will always be accessed through the carousel transporting the application.

Our second proposal is just slightly more complex to implement, but it requires automatic mounting of the cache carousel and some processing for filling up the device cache. However, in order to be able to share the resources among applications, the cache has to be global to all carousels and therefore fits best with the HTTP cache of the device/browser. It should be noticed that better results in terms of bandwidth occupancy will be achieved by gathering small resources (e.g. JS files) into a single one when possible (e.g. a single library): although the cache description is gzipped, it still grows linearly with the number of files to cache.

One major advantage of our second proposal is its ability to push HTTP resources over a broadcast link, while keeping the possibility to access the HTTP resources online if any. We see several use cases benefiting from this feature.

Leveraging server usage: our solution can reduce the risk of overwhelming web servers distributing JS libraries or other resources at the beginning of a popular broadcast with an interactive application which uses for the first time a new version of the resource: such a situation could generate millions of quasi synchronous requests for the same resource. We avoid this by pushing the requested resource in the device HTTP cache.

Enabling Content Push: our solution allows pushing complete movies or trailers to the device, without change to the HTML content. It can also be used for more advanced HTTP streaming, by pushing the first part of the session to the device, enabling the playback to start without buffering, and then be switched to a regular HTTP streaming session if the device is connected.

5. CONCLUSION

In this article, we have reviewed the basic functionalities of data carousel, including multi-period ones; we have shown how HbbTV, the upcoming interactive TV standard, is used together with this mechanism and have identified some overhead in bandwidth management, leading to longer load time and bad user experience. We have proposed a new caching paradigm for DSM-CC, the inter-carousel cache, based on HTTP caching rules, which allows a better usage of the available bandwidth and a better user experience. We have shown how this mechanism could potentially be used for data caching at the service level (data shared among different providers) and explained the potential gains. Our future works include standardization related to this topic, on-air experiments with national broadcaster and upgrade of our HbbTV toolchain.

6. ACKNOWLEDGMENTS

This work was partly done within the French-funded openHbb project, www.openhbb.eu.

7. REFERENCES

[1] HBBTV, ETSI Technical Specification 102 796, “Hybrid Broadcast Broadband TV”

[2] R. J. Crinon, “The DSM-CC object carousel for broadcast data services,” 1997, pp. 246-247.

[3] Flute - File Delivery over Unidirectional Transport, IETF RFC 3629, http://www.rfc-archive.org/getrfc.php?rfc=3926

[4] C. Neumann, V. Roca, R. Walsh, Large scale content distribution protocols, ACM SIGCOMM Computer Communication Review 2005 35 (5) (2005) 85–92.

[5] Steven Morris & Anthony Smith-Chaigneau, Interactive TV Standards: A Guide to MHP, OCAP, and JavaTV, Focal Press, 2005

[6] J.-C. Moissinac and C. Concolato, “A model for the delivery of interactive applications over broadcast channels,” in *Proceedings of the 3rd workshop on Mobile video delivery - MoViD ’10*, 2010, p. 15.

[7] C. Herrero and P. Vuorimaa, “Optimisation Techniques for Digital Television Applications Broadcasting,” Proceedings of the 7th IASTED International Conference on Internet and Multimedia Systems and Applications, IMSA 2003, Honolulu, USA, pp. 689-694, August 2003.