A Framework for Prefetching Mechanisms in Hypermedia Presentations

Rogério Ferreira Rodrigues and Luiz Fernando Gomes Soares
rogerio@telemidia.puc-rio.br, lfgs@inf.puc-rio.br
Laboratório TeleMídia – Departamento de Informática – PUC-Rio
R. Marquês de São Vicente, 225 – Gávea – 22453-900, Rio de Janeiro, RJ, Brazil

Abstract

In several hypermedia applications it is usual to face certain user impatience with long waiting times for presenting a media content. In other circumstances, even a short delay may impair the user interactivity or the synchronization among media objects, decreasing the presentation quality or generating an unintelligible presentation. In order to lessen these problems, the use of prefetching techniques is being taken into account in multimedia/hypermedia document presentation systems. This paper investigates the requirements for designing and implementing prefetching techniques in hypermedia formatters. Based on these requirements, the paper proposes a generic architecture to be adapted and reused in specific formatter developments. Besides the proposed architecture, we specify interfaces in order to make easier the integration of the prefetching mechanism with other hypermedia presentation system components, allowing the incorporation of prefetching strategies in formatter implementations currently in use.

1. Introduction

In order to have a hypermedia document high-quality presentation, several parameters need to be kept under control. Among the main temporal parameters, we can highlight: the media-object presentation lag, which can be defined as the difference between the time instant the object presentation really begins and the time instant the presentation was previously programmed to start; and the object presentation rate, often called intra-media synchronization and specially important in the case of continuous media (audio, video, etc.).

Among the undesirable effects that may occur due to uncontrolled object presentation lags are: loss of interactivity, loss of inter-media synchronization (temporal relationships among different media objects) and, evidently, the user impatience with long waiting times. Object presentation rate fluctuations, called jitter, may also cause loss of inter-media synchronization. Furthermore, jitter may result in unpleasant play-out (e.g. gaps or short jumps in the media stream) or even total loss of information intelligibility.

The maximum tolerable value for an object presentation lag depends on the object content type (text, image, audio, video, etc.) and the kind of application in use. For example, five seconds as an upper bound delay for presenting an HTML page is considered acceptable to maintain the navigation service at a good level [3]. However, for interactive voice applications this limit must be reduced to 150 ms [8].

Media presentation jitter usually only affects continuous media objects and should be avoided whenever possible. It is worth pointing up that inter-media synchronization relationships also impose constraints for the maximum acceptable lag and jitter.

Caching techniques have been widely used in several computer areas aiming to reduce system response time. The efficiency of a caching mechanism is directly related to the achieved hit rate, that is, the frequency that data are found in cache when required by the system. Cache memories can also work as buffers, being useful in jitter compensation strategies. Although the use of cache memories makes possible to decrease system response times, some analysis conclude that caching techniques in hypermedia systems, mainly the Web, presents a lower efficiency (30-60%) compared with the one generally obtained in hardware caching techniques (90%) [5, 10, 11]. Besides the issues related to the dynamic content generation, there is also a large amount of data that are requested only once by the presentation system that contributes to the poorer caching-mechanism efficiency.

Indeed, caching techniques are useful only when data are used more than once and from the second time on. Therefore, in order to minimize presentation lags and the inherent delay introduced by jitter compensation buffers, it is valuable to incorporate prefetching techniques into hypermedia presentation-control systems, from here on called *hypermedia formatters* (or simply *formatters*). Hypermedia prefetching techniques try to preview the next document data to be requested, so that they can be previously stored in a cache memory. The prefetching techniques should also anticipate system actions and schedule them in advance. For example, formatters can instantiate a media player in advance to avoid the loading-time delay. If well done, prefetching techniques may also improve caching mechanisms efficiency.

The main goal of this paper is to discuss the design and implementation requirements for hypermedia formatter prefetching mechanisms and, based on these requirements, to propose a generic architecture that can be adapted and reused in the development of specific formatters. Additionally, with the specification of well-defined interfaces, we intend to make easier the integration of prefetching mechanism with other hypermedia presentation system components, and to allow the incorporation of prefetching strategies into legacy formatter implementations, such as QuickTime, GRiNS Player for SMIL 2 and RealOne Player. Following our framework proposal, we developed a prefetching orchestrator for the HyperProp system formatter [17]. This module operates cooperatively with the system JMF-based players [15, 18].

The paper is organized as follows. Section 2 discusses common and desirable formatter structuring, highlighting the components (and their roles) a prefetching orchestrator should interact with. Section 3 presents our framework proposal, establishing a generic architecture to be used in the design and implementation of prefetching mechanisms. The section also comments some aspects associated with the framework instantiation example. Section 4 discusses related work, while Section 5 presents our conclusions and future work.

2. Hypermedia presentation environment

As aforementioned, the formatter is the hypermedia system entity responsible for adapting and controlling document presentations. The formatter should consider not only the document specification but also the presentation context (network, operating system, client I/O devices, user preferences, etc.) in order to accomplish its tasks. Although some formatting functionalities can be implemented either on the system client side (presentation subsystem) or on the system server side (storage subsystem), this paper deals only with the first option, without loss of generality when considering prefetching functions. Figure 1 illustrates the main subsystems that compose a hypermedia environment, offering a greater detail to the presentation subsystem internal organization, where we place the prefetching mechanism.

The hyperdocument presentation process starts with a request to the presentation subsystem passing the document specification (media objects identifiers and their relationships) and the context description as parameters. The document specification can be directly provided by the authoring subsystem or downloaded from the storage subsystem. The context description can be obtained from a system *context manager*. The context description can be statically maintained, for instance using a configuration file previously set up, or on a dynamic basis (ideal case), through the use of agents designated to collect the information. Network (communication service provider), client operating system and even other user agents are examples of sensor agents that could be queried about available bandwidth, local processing capabilities, user skill and preferences, etc. It is out of the scope of this paper to handle the issues related to context management, being sufficient to assume the existence of a component offering this service. It is also worth mentioning that the context manager can be part of all subsystems, working within a kind of signaling plan, and not be a separated component, regardless Figure 1 illustration.

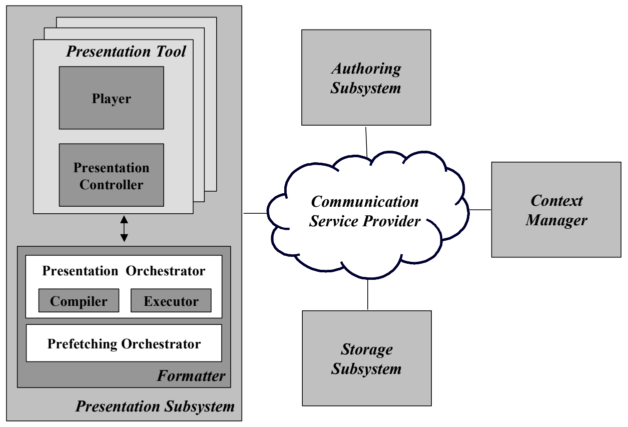

Figure 1 – Hypermedia environment subsystems

Using the presentation and context specification input, the formatter will usually build an *execution plan* (*scheduling plan*) to help itself in its tasks. This plan will essentially contain information about the object expected-durations and the expected times to perform several document presentation actions (e.g. start, stop, pause or resume object presentations). It is important that the execution plan data structure doesn't be a simple timeline, but preserves the object dependencies defined by the document relationship specifications. This will be helpful to implement adjustment mechanisms.

If any programmed action does not occur at the programmed time, for instance because of network congestion, the idea behind adjustment mechanisms is to find the presentation actions affected by the violation and try to punctually correct the execution plan in order to keep a good-quality presentation. The phase when the execution plan is at least partially built, before starting the object presentations, is known as the document *compile time* phase [4, 17]. Building the execution plan is a task delegated to the *compiler* element contained in the *presentation orchestrator* of Figure 1.

Since in hypermedia documents it is common to have object presentations that depend on unpredictable conditions, it may be impossible to establish their exact time of occurrence at compile time. Examples of unpredictable object presentations are those that depend on user interactions, like the presentation of the majority of HTML pages, or those that depend on the end of other object presentation whose duration cannot be previewed, like runtime programs or live videos. Examples of predictable object presentations are those specified as a consequence of temporal causal relationships triggered by other media object presentations with predictable durations, such as those defined by the SMIL parallel and sequential compositions [19] and NCM synchronization links [17].

An interesting strategy that can be used in the execution plan creation is to establish document time chains following the Firefly system proposal [4]. One of the time chains is designated the main time chain and corresponds to the sequence of predictable actions beginning with the presentation start action of the first document media object. Besides the main time chain, zero or more auxiliary time chains can exist, each one being a sequence of predictable actions triggered by an unpredictable object presentation.

Once the document presentation starts, the formatter enters in the *document runtime* phase. The *executor* module of the presentation orchestrator, showed in Figure 1, coordinates this phase. The executor schedules the execution-plan actions programmed by the compiler and instantiates the *presentation tools*, which are components responsible for actually playing object contents [15]. In order to manage the document presentation, the executor interacts with the presentation tools and may have to modify the execution plan to correct differences between the expected and the actual presentation times and durations. Furthermore, the executor must merge an auxiliary time chain with the main time chain, when the unpredictable action that triggers the auxiliary time chain occurs.

Both at compile time and runtime phases, document consistency checking should be made. In addition, adaptation strategies can be invoked to adjust object durations, to choose alternative objects for presentation, etc. However, these tasks are outside the scope of this paper.

The incorporation of a prefetching mechanism in hypermedia formatters is important not only to minimize object presentation lags but also to reduce the probability of adjustments needed at document runtime, thus decreasing the processing overhead of the formatter. The main idea is to build a prefetching plan, anticipating the actions specified in the document. The next section discusses in details the issues related to the design of a prefetching orchestrator, proposing a framework to be used in its implementation.

3. Framework proposal

The main purpose of incorporating a prefetching mechanism into a hypermedia formatter is to increase the probability of media object content (not necessary the whole content) being available to the presentation tools at the moment they should be played. In order to reach its goal, a hypermedia prefetching orchestrator needs to compute the fetching expected duration for each media object contained in a specific execution plan. Then, the orchestrator should join these estimated results with the media object presentation expected time, specified in the execution plan, in order to establish the expected instant for starting its preparation. All this process, performed for all media objects, will result in what is called a prefetching plan. After this computation, the orchestrator should run a scheduler that informs to prefetcher system components when it is time to start a preparation.

Analogous to the presentation orchestration presented in Section 2, the prefetching orchestration can be divided into two phases: compilation and execution. At compile time, the prefetching plan is generated using the execution plan and platform information (network throughput, hard disk seek time, data transfer rate, etc.). At execution phase, the prefetching plan actions are scheduled and real presentation durations are monitored, triggering adjustments when possible mismatches occur. Figure 2 shows the framework main elements. We chose to follow an object-oriented approach, using UML notation [16] in component and relationship specification.

The *PrefetchOrchestrator* class is the prefetching orchestrator facade, offering a single interface to the formatter presentation orchestrator, or any other application interested in making use of the prefetching services. The *PrefetchOrchestrator* is composed by a prefetching compiler (*PrefetchCompiler*), a prefetching excecutor (*PrefetchExecutor*), and the prefetching plan itself (*PrefecthPlan*). The prefetching plan will be a collection of prefetching objects (*PrefetchObject*), where each object has attributes to store the object expected prefetching time and duration. Moreover, prefetching objects have a reference to their corresponding execution objects (*ExecutionObject*) in the execution plan (*ExecutionPlan*).

The *PrefetchOrchestrator* allows a plan compilation to be asynchronously or synchronously performed offering two compiling methods: *startCompilation* (asynchronous) and *compilePlan* (synchronous). The asynchronous method is useful when the execution plan can be built concurrently with the document execution. In this case, the computation of the prefetching plan stays running in parallel with the presentation compilation, the presentation execution and also the prefetching execution. On the other hand, the synchronous method can be used when the execution plan is entirely built before the beginning of the document runtime phase. This approach may generate a better schedule for the prefetching plan, since the prefetcher can have a complete knowledge of the document presentation specification. With synchronous compilation it is also possible to estimate the delay for starting the document presentation in order to lessen object presentation jitters and lags. However, the introduced delay may prejudice interactive applications.

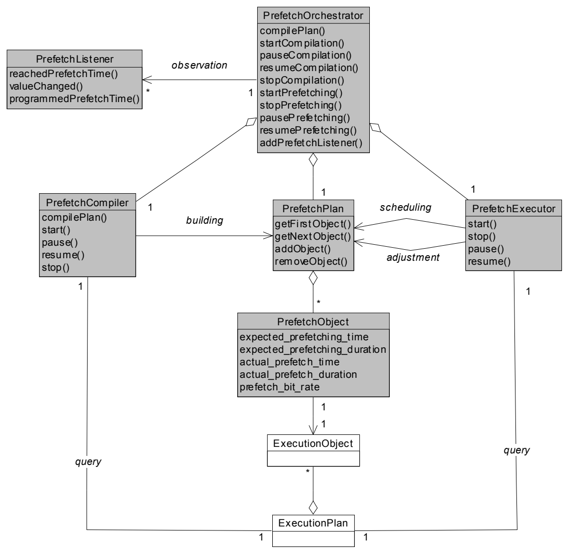

Figure 2 – Main classes and relationships for a generic prefetching mechanism

In order to act as a *prefetching-user*, an object[1] must register itself as a prefecthing listener through the *addPrefetchListener* method of the *OrchestratorPrefetcher*. A prefetching-user object can monitor one, more than one or all prefetching objects (*PrefetchObject*), and will be notified when prefetching events over these prefetching objects occur, through the invocation of one of its three following methods. The *reachedPrefetchTime* method is called when the prefetching executor verifies that a media object has reached its programmed prefetching starting time. The method receives the prefetching object (*PrefetchObject*) as parameter. Usually, software components that actually download and store media object contents (*prefetcher element*) will be the main objects interested in handle this method. The *valueChanged* method is called when an attribute value of a prefetching object is modified. It receives as parameter the prefetching object, the name of the changed attribute and its old value. This method is useful for components that need to monitor the prefetching execution and react when modifications occur. Finally, the *programmedPrefetchTime* notifies when a prefetching object is inserted in the prefetching plan, informing the expected time for starting its preparation. This method is especially useful to quality of service (QoS) negotiators present in environments with support for resource reservation in advance [2, 6, 20]. Note that these methods are called both during prefetching compile an execution time.

The only way of invoking the start of prefetching execution is calling the *startPrefetching* asynchronous method of the *PrefetchOrchestrator*. This method call will put the prefetching executor in running state, will initiate its scheduler and, if implemented, will fire its adjustment strategy, as will be explained afterward.

As previously mentioned, the prefetching plan is computed based on the execution plan information, modeled by the *ExecutionPlan* class, which is composed by a collection of execution objects (*ExecutionObject*). Actually, these classes are not part of the prefetching mechanism structure[2]. They specify interfaces that will vary according to the formatter implementation and its document model. However, we can identify some attributes that all elements of this type should have, despite of particular implementations. It would be a great contribution for interoperability if formatters were in agreement with a common standard interface for these elements.

Execution plans should offer methods supporting navigation in its schedule structure through queries about the execution-object presentation order. Each execution object should contain all information needed for its exhibition. At least, an execution object should have attributes defining its expected presentation start time, a reference to its content, its duration (or size), and its presentation rate (for continuous media). Additionally, it is desirable, mainly for prefetching mechanisms, that an execution object knows its presentation probability, and the maximum lag that is acceptable for its presentation start. All these information should be initialized outside the prefetcher, being responsibility of other formatter components (e.g. presentation orchestrator compiler). It is interesting that execution plans and execution objects also have means for notifying their users (the prefetching compiler and the prefetching executor, for example) whenever a new object is inserted in the plan or when any object attribute has its value changed.

Analyzing the prefetching compiler in more details (Figure 3), we can identify two main functions: one to

[1] We would like to highlight that the word *object* in this context means the object OO (object-oriented) concept, and not the document media object, although nothing forbids that a media object performs the prefetching listener role.

[2] They are not filled in Figure 2.

estimate object prefetching durations and another to sort the prefetching actions. We separate these functions into two classes: *PrefetchDurationEstimator* and *PrefetchCompilerStrategy.* The compilation strategy is the component that actually builds the prefetching plan, using the prefetching estimation services to accomplish its goal. Basically, the estimator offers a method (*getPrefetchDuration*) that returns the expected duration for a specific media object preparation. The method receives as arguments the object content reference and, optionally, the content percentage to be prefetched and the transmission rate to be used for the preparation action. The compilation strategy should choose the values to these later two optional parameters if the estimator will use them. In order to be aided in this choice, a good solution may be the use of context manager services to discover platform resource availabilities, increasing the probability of having the media object content ready at the programmed expected time.

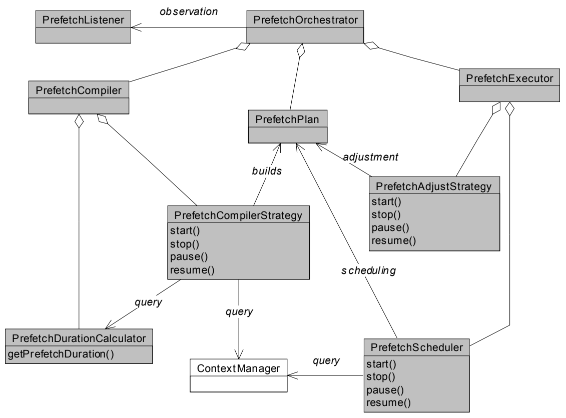

Figure 3 – Prefetching compiler and executor details

Some important issues should be taken into account by the compilation strategy when building the prefetching plan. Two undesirable situations may happen if the prefetching compiler makes a bad choice for a media object prefetching time [9, 12]. If a programmed time is later than the optimal value, the media object may have a presentation lag longer than it is acceptable. As a consequence, if there is at least one inter-media constraint defining a parallel relationship including this delayed object, the presentation synchronization may be lost. In this case, the formatter will have to perform a runtime adjustment to satisfy the document specification constraint. On the other hand, if the programmed prefetching time occurs too earlier it may generate memory and network pollution, either obliging the discard of another content that would still be used or obliging the discard of the precipitate content itself. The delayed prefetching can be minimized introducing a delay time before starting the document presentation. This solution will also increase the buffer size necessary to pre-store object contents. The memory pollution can be avoided also increasing buffer sizes and/or performing a more refined control over prefetching time choices.

In this paper proposal, both the *PrefetchDurationEstimator* class and the *PrefetchCompilerStrategy* class are left as framework hot spots. They are flexible points in the architecture that must be extended and adapted to each specific implementation.

As aforementioned, the prefetching executor controls the actions that should be performed after the prefetching plan computation. It is important to mention that the prefetching executor does not need to wait for the entire computation of the prefetching plan. For some presentations (e.g. documents with long time chains or lots of unpredictable actions), it can be interesting to maintain prefetching compilation and execution concurrently.

Similar to the prefetching compiler, we defined the prefetching executor (*PrefetchExecutor*) as an aggregation of two classes (Figure 3): *PrefetchScheduler* and *PrefetchAdjustStrategy*. The prefetching scheduler queries the prefetching plan and tells the orchestrator to notify the prefetching-users when the start time for a preparation is reached.

The scheduling strategy may also call *reachedPrefetchTime* on prefetching listeners passing an object with unpredictable duration when identifying available resources on the local machine. The scheduling strategy does this based on the object presentation probability information (if available). For this reason, like the compiler strategy, it is interesting that the scheduler be able to query the system context manager to find the better time to schedule this kind of prefetching.

The adjustment strategy should monitor the execution plan, reacting when modifications in the presentation schedule are signalized. The common notifications that will affect the adjustment strategy are changes in object expected times due to formatter runtime adaptation, and changes in object presentation probabilities, for example due to the merge of auxiliary time chains with the main time chain caused by user interaction.

Similar to the compiler design, the *PrefetchScheduler* class and the *PrefetchAdjustStrategy* class are framework hot spots, offering flexibility for the mechanism implementation.

When hypermedia formatter runs on an environment with resource reservation support, it is very important that the QoS components interacts with the prefetching mechanism in order to minimize temporal mismatches and, as a consequence, the need for runtime adjustments.

This discussion, however, is outside the scope of this paper and is left as a future work.

3.1 Implementation

A framework instantiation was developed to incorporate prefetching mechanisms into the HyperProp system formatter [17]. This system is implemented in Java and allows the exhibition of documents specified either in the NCM model (*Nested Context Model*) [17] or in SMIL (version 1.0) [19] and NCL (*Nested Context Language*) [14] languages. The HyperProp formatter has presentation tools both for static media (mainly, HTML content) as for continuous media. For the later case, the system makes use of Sun Java Media Framework [15, 18].

The current framework instantiation comprises the implementation of a prefetching plan compilation strategy, a prefetching scheduling strategy and a prefetching duration estimator.

The compilation strategy that we developed is a merge of the ideas proposed in two related work [9, 10], discussed in the next section. The estimator uses heuristics to return not only the expected prefetching duration, but also expected time spent for creating and preparing the presentation tool itself. We implemented the HyperProp formatter executor as a *PrefetchListener*, since this is the element responsible for instantiating and interacting with presentation tools. The executor delegates the content prefetching action to each presentation tool, using the JMF primitives for this purpose [18]. Actually, the executor interacts with a presentation controller (Figure 1), which adapts the JMF API to a generic formatter API [15].

As a future work, we intend to develop runtime adjustment algorithms for prefetching plans. We also intend to integrate our implementation with reservation mechanisms (of operating systems and networks [7]) and to research the possibility of using reservation-in-advance schemes.

4. Related work

Jeong et al. [9] developed a mechanism for pre-scheduling multimedia presentations. The prefetcher component, named EPS (*Event Pre-Scheduler*), estimates the maximum time for recovering the entire content of each document object and builds a prefetching plan in order to have all object contents ready at their playing time. The building algorithm postpones the start of the document presentation as a whole, in order to avoid gaps and loss of synchronization during the presentation. The execution plan data structure, used as input for the prefetching plan calculation, is based on the partition of media objects in minimum segments that satisfy the Allen equality condition (*equals*) [1]. At document runtime phase, there is a monitor that compares the object actual prefetching duration with the expected duration previewed in the plan. If the actual duration overcomes the predicted one, the system runs an instant scheduling algorithm for recalculating the object presentation durations in order to maintain the temporal segment synchronization. When necessary, the algorithm sacrifices static media objects, shrinking their durations. The interesting characteristic of this algorithm is that it allows correcting delays introduced not only by the operating system kernel interruptions, but also by the algorithm itself.

EPS is a useful reference because it presents a practical approach for identifying several requirements that must be considered when elaborating prefetching mechanisms. Other contribution of this paper is the implementation of a strategy for building the prefetching plan. However, the proposal only considers documents exclusively based on predictable relations and predictable media object durations. Furthermore, all content must be locally and contiguously stored in a hard disk, when network transmission does not need to be treated. The developed strategy always prefetches the object entire content on memory before start playing it. It is not possible that a presentation tool downloads data actually being presented in parallel with data being prefetched. Finally, another drawback is that only the document execution plan is adjusted at runtime; no adaptation is done in the prefetching plan.

Khan and Tao [10] propose a model that permits to find the best prefetching sequence for Web composite pages. They define composed page as the ones that include, besides the traditional HTML content, links to other objects as images, applets, animations, video, audio, etc. The composed pages are named hypermedia compositions and the composition components (objects) are classified into three profiles, considering network transmission issues: constant rate objects (CBR media: audio, video, etc.), objects of a single burst (impulse media: HTML pages, static images, Java applets, etc.) and objects of multiple bursts (impulse series media: multiple page PDF documents, animations, flash and slide shows, etc.). A mathematical model allows calculating, for each object, the minimum amount of data that should be fetched before its exhibition. Taking into account the traversal probability of each link, the paper demonstrates which is the best prefetching plan arrangement, i.e., which is the best order to fetch the composed pages. A proposed algorithm splits the network bandwidth to allow prefetch requests in parallel with requests for contents being effectively presented, without degrading their performance. However, the work does not consider inter and intra–media temporal synchronization relationships among document objects.

It is important to mention that both previous work could be implemented as instantiations of our framework proposal. They could be present mainly in the strategies for building the prefetching plan and for estimating object-prefetching durations.

Revel et al. [13] present a prefetcher implementation, which offers the service of anticipating the copy of application multimedia data from disk to memory, especially data from MPEG videos. The prefetcher is implemented as an operating system integrated module. It provides an interface for multimedia applications describing their data access pattern (frame size; frame rate; I, P and B pattern etc.) and programming the expected time to use this data. After scheduling the presentation, the implementation permits that multimedia applications use their traditional I/O calls to access the prefetched data. Internally, the prefetcher makes asynchronous calls to the operating system and the OS itself performs the data prefetch. However, the specified API does not provide calls for applications using a more varied set of media types with unpredictable presentation behavior. Moreover, the architecture extension to be used in distributed environments is left as a future work.

5. Conclusions and future work

This paper presented a framework to be used in prefetching mechanism design and implementation, focused but not limited to hypermedia presentation systems. We developed a framework instantiation of our proposal within the implementation of the HyperProp system formatter [15, 17]. This implementation allowed integrating prefetching mechanisms with JMF-based presentation tools.

Prefetching mechanisms are very important, mainly when the client presentation platform does not provide means for maintaining the application required QoS. Even in scenarios where it is possible to negotiate and obtain a service level agreement, the prefetcher module can be very useful identifying the better moment to start QoS parameter negotiation for a media-object transmission. If the communication and processing service providers allow quality of service negotiation in advance [2, 6, 20], we believe that better will be the profit achieved by our proposal. Following this open issue, as a future work we intend to investigate the integration of the HyperProp prefetcher implementation with traditional and anticipated resource reservation APIs. We also intend to study the integration of our framework with the set of frameworks specified in [7] for QoS provisioning in communication and processing generic environments.

Another research that we are currently working on is to instantiate our framework for interactive TV set-top box formatters. We also plan to adapt our HyperProp solution for other document formatters, like SMIL version 2.0. The work presented in this paper is part of a wider proposal, aiming to develop a framework for the design and implementation of hypermedia formatters with adaptive presentation support.

Acknowledgement

Authors would like to thank Heron Vilela for his important contribution on the framework instantiation, and Brazilian Council on Technologic and Scientific Development (CNPq) for its financial support.

References

[1] Allen J.F. “Maintaining Knowledge about Temporal Interval”, *Communications of the ACM*, 26(11), 1983, pp. 823-843.

[2] Berson S., Lindell R., Braden R. “An Architecture for Advance Reservations in the Internet”, *Technical report*, USC Information Sciences Institute, July 1998.

[3] Bhatti N., Bouch A., Kuchinsky A. “Integrating User-Perceived Quality into Web Server Design”, *WWW Conference*, 2000.

[4] Buchanan M.C., Zellweger P.T. “Automatic Temporal Layout Mechanisms”, *ACM International Conference on Multimedia*, California, USA, 1993, pp. 341-350.

[5] Cohen E., Kaplan H. “Prefetching the Means for Document Transfer: A new Approach for Reducing Web Latency”. *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[6] Degermark M., Köhler T., Pink S., Schelén O. “Advance Reservation for Predictive Service”, *5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, April 1995.

[7] Gomes A.T.A., Colcher S., Soares L.F.G. “Modeling QoS provision on Adaptable Communication Environments”, *International Conference on Communications - ICC2001*, Helsinki, Finland, June 2001.

[8] International Telecommunications Union “One-way transmission time”, *Recommendation G.114*, February 1996.

[9] Jeong T., Ham J., Kim S. “A Pre-scheduling Mechanism for Multimedia Presentation Synchronization”, *IEEE International Conference on Multimedia Computing and Systems, Ottawa, Canada*, 1997, pp. 379-386.

[10] Khan J., Tao Q. “Prefetch Scheduling for Composite Hypermedia”, *IEEE International Conference on Communications - ICC 2001*, Helsinki, Finland, June 2001.

[11] Kroeger T., Long D.D.E., Mogul J. “Exploring the Bounds of Web Latency Reduction from Caching and Prefetching”. *USENIX Symposium on Internet Technology and Systems*, Monterey, December 1997, pp. 319-328.

[12] Oren N. “A Survey of prefetching techniques”, *Technical report*, July 2000. *Available at http://citeseer.nj.nec.com/334350.html.*

[13] Revel D., Cowan C., McNamee D., Pu C., Walpole J. “An Architecture for Flexible Multimedia Prefetching”, *Workshop on Resource Allocation Problems in Multimedia Systems*, Washington DC, November 1996.

[14] Rodrigues L.M., Antonacci M.J., Rodrigues R.F., Muchaluat-Saade D.C. Soares L.F.G. “Improving SMIL with NCM Facilities”. *Journal of Multimedia Tools and Applications*, Kluwer Academics, 16(1), January 2002, pp. 29-54.

[15] Rodrigues R.F., Rodrigues L.M., Soares L.F.G. “A Framework for Event-Driven Hypermedia Presentation Systems”, *Multimedia Modeling Conference - MMM’2001*, Amsterdam, Netherlands, November 2001, pp. 169-185.

[16] Rumbaugh J., Jacobson I., Booch G. “The Unified Modeling Language: Reference Manual”, *Addison-Wesley*, 1999.

[17] Soares L.F.G., Rodrigues R.F., Muchaluat-Saade D.C. “Modeling, Authoring and Formatting Hypermedia Documents in the HyperProp System”, *ACM Multimedia Systems Journal*, Springer-Verlag, 8(2), March 2000, pp. 118-134.

[18] Sun Microsystems. “Java Media Framework, v2.0 API Specification”, 1999. *Available at http://java.sun.com/ products/java-media/jmf/2.1/specdownload.html*.

[19] W3C. “Synchronized Multimedia Integration Language (SMIL 2.0) Specification”, *W3C Recommendation*, August 2001. *Available at http://www.w3.org/TR/smil20*.

[20] Wolf L., Delgrossi L., Steinmetz R., Schaller S., Wittig H. “Issues of Reserving Resources in Advance”, *Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.