Design and Implementation of a Distributed Content Management System

C. D. Cranor, R. Ethington, A. Sehgal[†], D. Shur, C. Sreenan[‡] and J.E. van der Merwe

AT&T Labs - Research
Florham Park, NJ, USA

[†]University of Kentucky
Lexington, KY, USA

[‡] University College Cork
Cork, Ireland

ABSTRACT

The convergence of advances in storage, encoding, and networking technologies has brought us to an environment where huge amounts of continuous media content is routinely stored and exchanged between network enabled devices. Keeping track of (or managing) such content remains challenging due to the sheer volume of data. Storing "live" continuous media (such as TV or radio content) adds to the complexity in that this content has no well defined start or end and is therefore cumbersome to deal with. Networked storage allows content that is logically viewed as part of the same collection to in fact be distributed across a network, making the task of content management all but impossible to deal with without a content management system. In this paper we present the design and implementation of the Spectrum content management system, which deals with rich media content effectively in this environment.

Spectrum has a modular architecture that allows its application to both stand-alone and various networked scenarios. A unique aspect of Spectrum is that it requires one (or more) retention policies to apply to every piece of content that is stored in the system. This means that there are no eviction policies. Content that no longer has a retention policy applied to it is simply removed from the system. Different retention policies can easily be applied to the same content thus naturally facilitating sharing without duplication. This approach also allows Spectrum to easily apply time based policies which are basic building blocks required to deal with the storage of live continuous media, to content. We not only describe the details of the Spectrum architecture but also give typical use cases.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-communication Networks—*distributed systems*; H.3.4 [**Information Systems**]: Information Storage and Retrieval—*systems and software*

General Terms

Design, Management

NOSSDAV'03, June 1–3, 2003, Monterey, California, USA.

Keywords

distributed content management, continuous media storage

1. INTRODUCTION

Manipulating and managing content is and has always been one of the primary functions of a computer. Initial computing applications include text formatters and program compilers. Content was initially managed by explicit user interaction through the use of files and filesystems. As technology has advanced, both the types of content and the way people wish to use it have greatly changed. New content types such as continuous multimedia streams have become commonplace due to the convergence of advances in storage, encoding, and networking technologies. For example, by combining improvements in storage and encoding, it is now possible to store many hours of TV-quality encoded video on a single disk drive. This has led to the introduction of stand alone digital video recording or personal video recording (PVR) systems such as TiVO [8] and ReplayTV [7]. Another example is the combination of encoding and broadband networking technology. This combination has allowed users to access and share multimedia content in both local and remote area networks with the network itself acting as a huge data repository.

The proliferation of high quality content enabled by these advances in storage, encoding, and networking technology creates the need for new ways to manipulate and manage the data. The focus of our work is on the storage of media rich content and in particular the storage of continuous media content in either pre-packaged or "live" forms. The need for content management in this area is apparent when one consider the following:

- Increases in the capacity and decreases in the cost of storage means that even modest desktop systems today have the ability to store massive amounts of content. Managing such content manually (or more correctly manual "non-management" of such content) lead to great inefficiencies where "unwanted" and forgotten content waste storage and where "wanted" content cannot be found.

- While true for all types of content the storage of continuous media content is especially problematic. First continuous media content is still very demanding in terms of storage resources which means that a policy-less approach to storing it will not work for all but the smallest systems. Second, the storing of "live" content such as TV or radio is inherently problematic as these signals are continuous streams with no endpoints. This means that before one can even think about managing such content there is a need to abstract it into something that could be manipulated and managed.

- When dealing with stored continuous media there is a need to manage such content at both a fine-grained as well as an aggregate level. For example, an individual PVR user wanting to keep only the highlights of a particular sporting event should not be required to have to store the content pertaining to the complete event. At the same time the user might want to think of content in the aggregate, e.g. remove all of the content that I have not watched for the last month except that content which was explicitly marked for archival.
- As indicated above, trying to keep track of content on a standalone system without a content management system is very difficult. However, when the actual storage devices are distributed across a network the task of keeping track of content is almost impossible. This scenario is increasingly common in network based content distribution systems and is likely to also become important in home-networking scenarios.

It would seem clear then that a content management system that can efficiently handle media rich content while also exploiting the networked capability of storage devices is needed. This system should allow efficient storage of and access to content across heterogeneous network storage devices according to user preferences. The content management system should translate user preferences into appropriate low-level storage policies and should allow those preferences to be expressed at a fine level of granularity (while not requiring it in general). The content management system should allow the user to manipulate and reason about (i.e. change the storage policy associated with) the storage of (parts of) continuous media content.

Addressing this distributed content management problem is difficult due to the number of requirements placed on the system. For example:

- The content management system must operate on a large number of heterogeneous systems. In some cases the system may be managing content stored on a local filesystem, while in others the content may be stored on a separate network storage appliance. The content manager may be responsible for implementing the policies it uses to reference content or that role may be delegated to a separate computer. A application program interface (API) and associated network protocols are needed in order for the content management system to provide a uniform interface.
- The content management system should be flexible and be able to handle differing requirements for content management policies. These policies reflect what content should be obtained, when it should be fetched, how long it should be retained, and under what circumstances it should be discarded. This means that the content management system should allow multiple applications to reference content with a rich set of policies and that it should all work together seamlessly.
- The content management system needs to be able to monitor references for content and use that information to place content in the right location in the network for efficient application access.
- The content management system must handle the interaction between implicit and explicit population of content at the network edge.
- The content system must be able to efficiently manage large sets of content, including continuous streams. It needs to be able to package this content in such a way that it is convenient for users to access.

To address these issues we have designed and implemented the Spectrum content management system architecture. Our layered architecture is flexible — its API allows the layers to reside either on a single computer or on multiple networked heterogeneous computers. It allows multiple applications to reference content using differing policies. Note that the Spectrum architecture assumes the existence of a content distribution network (CDN) that can facilitate the efficient distribution of content (for example, the PRISM CDN architecture [2]).

The rest of this paper is organized as follows. Section 2 describes the architecture of our content management system. In Section 3 we describe both our implementation of the Spectrum architecture and examples of its use. Related work is described in Section 4, and Section 5 contains our conclusion and suggestions for future work.

2. THE SPECTRUM DISTRIBUTED CONTENT MANAGEMENT SYSTEM ARCHITECTURE

The Spectrum architecture consists of three distinct management layers that may or may not be distributed across multiple machines, as shown in Figure 1. The three layers are:

content manager: contains application specific information that is used to manage all of an application's content according to user preferences. For example, in a personal video recorder (PVR) application the content manager receives requests for content from a user interface and interacts with the lower layers of the Spectrum architecture to store and manage content on the device.

policy manager: implements and enforces various storage polices that the content manager uses to refer to content. The policy manager exports an interface to the content manager that allows the content manager to request that a piece content be treated according to a specific policy. Spectrum allows for arbitrary policies to be realized by providing a fixed set of base-policy templates that can easily be parameterized. It is our belief that for most implementations this will be adequate (if not, Spectrum can easily be extended to dynamically load new base-policy template code at run time). A key aspect of the policy manager is that it allows different policies to be simultaneously applied to the same content (or parts of the same content). Furthermore content can only exist in the system so long as it is referenced by at least one existing policy. Policy conflicts are eliminated by having the policy manager deal exclusively with retention policies rather than with a mix of retention and eviction policies. This means that content with no policy associated with it is immediately and automatically removed from the system. This approach allows us to naturally support sharing of content across different policies which is critical to the efficient storage of large objects.

Note that a key difference between the content manager and the policy manager is that the content manager manages references to multiple pieces of content, i.e. it has an "application-view" of content. On the other hand, the policy manager is only concerned with the policy used to manage "standalone" pieces of content. For example, in a PVR application, the content manager layer would know about the different groups of managed content such as "keep-indefinitely," "keep for one day," and "keep if available diskspace." However, at the policy manager level, each piece of content has

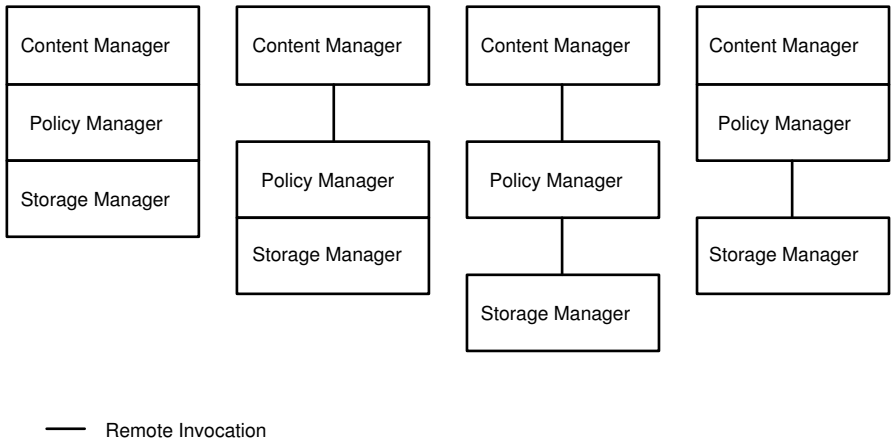

Figure 1: The components of the Spectrum architecture and the four ways they can be configured

its own policy (or policies) applied to it and is independent from other content.

storage manager: stores content in an efficient manner while facilitating the objectives of the higher layers. Specifically the storage manager stores content in sub-object "chunks." This approach has advantages for the efficient retrieval of content but more importantly allows policies to be applied at a sub-object level which is critically important when dealing with very large objects such as parts of continuous media, e.g. selected pieces of TV content being stored on a PVR. Note that the storage manager has no knowledge of the policies being used by the content and policy managers.

Another unique part of our approach is that the interfaces between the layers can either be local or distributed. Figure 1 shows the four possible cases. The case on the far left of the Figure shows the simplest (non-distributed) case where all the layers are implemented on a single box. This configuration would be used in self-contained applications such as PVRs.

The next case over corresponds to the case where there is a centralized content manager that controls distributed storage devices each of which is responsible for implementing policy based storage. In this case although the remote devices are controlled by the central manager they operate much more independently. For example, once they receive "instructions" from the central manager they typically operate in autonomous fashion. An example of this type of configuration is a content distribution network (CDN) that distributes and stores content based on a schedule determined by some centralized controller. For example, the CDN could pre-populate edge devices with content that is expected to be very popular or distribute large files to branch offices during off-peak hours in a bandwidth constrained enterprise environment.

Allowing a single policy manager to control several storage managers leads to the next combination of functions and the most distributed case. The need for this sort of separation might occur for scalability reasons or when different specialized storage devices or appliances are required to be controlled by a single policy manager.

The final case shows a content manager combined with a policy manager controlling a remote storage manager. This separation would be possible if the storage manager is somewhat autonomous and does not require continuous fine grained control by the policy manager.

We now examine the function of the three layers in detail.

2.1 Content Manager

The content manager layer is the primary interface through which specific applications use the Spectrum architecture. As such the content manager layer provides an API for the application to manipulate all aspects of the Spectrum architecture at different levels of granularity. The content manager API has functions that handle:

Physical devices: This set of functions allows physical storage devices to be added to Spectrum thereby putting them under control of the content manager and making the storage available to the system. Physical devices can be local or remote — this is the only place in the architecture where the application is required to be aware of this distinction. Once a device is mapped into the application through this interface, the system tracks its type and location. Users simply refer to the content through an application-provided label.

Stores: Stores are subsets of physical storage devices. Through these functions an application can create a store on a physical device and assign resources (e.g. disk space) to it. Stores can only be created in physical devices that are mapped into the system.

Policy Groups: Policy groups are the means whereby an application specifies, instantiates, and modifies the policies that are applied to Spectrum content. Typical usage of this set of functions is to select one of a small set of base policies and to parameterize this specific instance of the policy. Policy groups are created within existing stores in the system. The Spectrum architecture has policies that are normally associated with storage that aim to optimize disk usage. In addition a set of policies that take a sophisticated time specification enable storage that is cognizant of time. For example, a simple time-based policy could evict content from the system at a certain absolute or relative time. A slightly more involved time-based policy enabled by the Spectrum architecture could allow content to be stored in "rolling window" of a number of hours (for example, the most recent N-number of hours is kept in the system). Time-based polices are of particular use when dealing with continuous content like a live broadcast.

Content: At the finest level of granularity content can be added to or removed from the system. Content is specified to the system by means of a uniform resource locator (URL) which concisely indicates the location of the content as well as the protocol to be used to retrieve it. Optionally a time specification can be associated with content. This allows content to be fetched into the system at some future time, or at future time intervals. Again, this is particularly useful for dealing with the storage and management of live content.

2.2 Policy Manager

The policy manager layer of the Spectrum architecture has two main types of API functions. First, there are functions that operate on managed storage areas and policy-based references (*prefs*) to content stored there. Second, there are sets of functions used to implement each management policy. The first class of functions is used by the content manager layer to access storage. Operations include:

create, open, and close: These operations are used by the content manager to control its access to storage. The policy manager's create operation is used to establish contact with a store for the first time. Once this is done, the store can be open and closed using the appropriate routines. Note that the parameters used to create a store contain information on how to reach it. For example, local stores have a path associated with them, while remote stores have a remote host and remote path associated with them. The information only needs to be passed to the policy manager once at create time. For open operations, the policy manager will use cached information to contact the store.

lookup: The lookup operation provides a way for the content manager to query the policy manager about what content is currently present for a given URL. For continuous media time ranges of present media will be returned.

resource: The resource routines are used to query the policy manager about its current resource usage. There are two resource routines: one that applies to the store as a whole and another that applies to a particular policy reference. The resource API is extensible, we currently support queries on disk usage and I/O load.

pref establish/update: The *pref* establish operation is used by the content manager to reference content on the store. If the content is not present, this call will result in the content being fetched (or being scheduled to be fetched if the content is not currently available). Parameters of this function include the URL to store it under, the URL to fetch data from if it is not present, the policy to store the content under, and the arguments used to parameterize the policy. The result of a successful *pref* establish operation is a policy reference ID string. This ID can be used with the update operation to either change the storage policy parameters or delete the reference entirely.

The second group of policy manager functions are used to implement all the polices supported by Spectrum. We envision a small set of base-level policy functions that can be parameterized to produce a wide range of storage polices. For example, a policy that implements recording a repeating time window can be parameterized to function daily, weekly, or monthly. Note that the policy manager is only concerned with executing a specific policy. The higher-level reasons for choosing a given policy are handled by the content and application manager.

A base policy is implemented using six functions:

establish: called when a *pref* is established with the required URLs and base policy's parameters. The establish routine references any content already present in the store and then determines the next time it needs to take action (e.g. start a download) and schedules a callback for that time. It can also register to receive callbacks if new content is received for a given URL.

update: called to change the parameters of a *pref*, or to discard the policy reference.

newclip: called when a chunk of new content is received for a URL of interest. The base policy typically arranges for newclip to be called for a given URL when the *pref* is established. When newclip is called, the base policy checks its parameters to determine if it wishes to add a reference to the clip just received.

callback: called when the *pref* schedules a timer-based callback. This is a useful wakeup mechanism for *prefs* that need to be idle for a long period of time (e.g. between programs).

boot/shutdown: called when the content management system is booting or shutting down. The boot operation is typically used to schedule initial callbacks or start I/O operations. The shutdown operation is used to gracefully shutdown I/O streams and save state.

2.3 Storage Manager

The role of Spectrum's storage manager is to control all I/O operations associated with a given store. Spectrum's storage manager supports storing content both on a local filesystem and on a remote fileserver (e.g. a storage appliance). For continuous media, at the storage manager level content is stored as a collection of time-based chunks. Depending on the underlying filesystem, a chunk could correspond to a single file or a data node in a storage database.

The two main storage manager operations are input and output. The input routine is used to store content in a store under a given name. The output routine is used to send data from the store to a client. For streaming media both the input and output routines take time ranges that schedule when the I/O operation should happen, and both routines return an I/O handle that can be used to modify or cancel the I/O request in the future.

Much like the policy manager, the storage manager also provides API functions to create, open, and close stores. It also supports operations to query the resource usages and options supported by the store. Finally, the storage manager also has a discard routine that may be used by the policy manager to inform the store to remove content from the store.

3. IMPLEMENTATION AND USE CASES

In this section we describe our implementation of Spectrum and describe how it can be used.

3.1 Implementation

We have implemented Spectrum's three layers in C as part of a library that can be linked with Spectrum-based applications. Each layer keeps track of its state through a set of local data files that persist across reboots, thus allowing Spectrum to smoothly handle power cycles. For layers that reside on remote systems (e.g. a remote store) only the meta-information needed to contact the remote

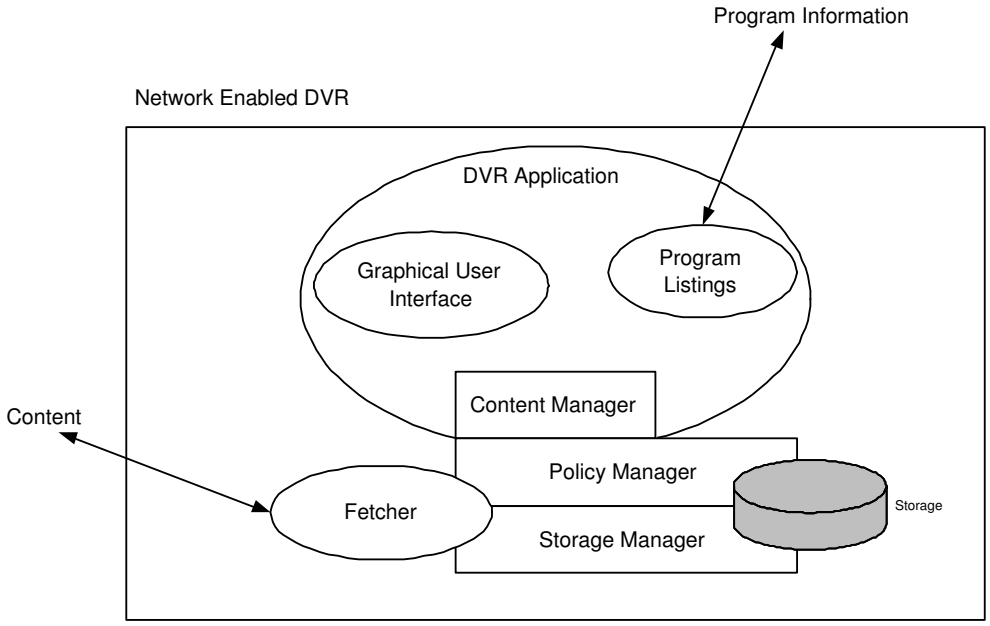

Figure 2: Spectrum in a Network Enabled DVR

node is stored locally. Our test application uses a local policy and storage manager to fetch content and store it in a normal Unix-based filesystem.

To efficiently handle communications with layers running on remote systems, all Spectrum's API calls support both synchronous and asynchronous modes through a uniform interface defined by the `reqinfo` structure. Each API call takes a pointer to a `reqinfo` structure as one of its arguments. This structure is used to hold the call state and return status. For async calls, the `reqinfo` also contains a pointer to a callback function. To use a Spectrum API function, the caller first chooses either the sync or async mode and allocates a `reqinfo` structure. For sync calls, the `reqinfo` can be allocated on the stack, otherwise it is allocated with malloc. For async calls, a callback function must be provided when the `reqinfo` is allocated. Next the caller invokes the desired Spectrum API function passing the `reqinfo` structure as an argument. For sync calls, the result of the calls is returned immediately in the `reqinfo` structure. For successful async calls, a "call in progress" value is returned. Later, when the async call completes or a timeout occurs, the async callback function is called with the appropriate information needed to complete processing.

The modular/layered design of the Spectrum architecture simplifies the objective of distribution of functionality. Furthermore, communication between functions is typically of a "master-slave(s)" nature. This means that several approaches to distributed operation are possible that would satisfy the architectural requirements. In our implementation we have opted to realize this functionality with a simple modular design. We provide a set of asynchronous remote access stub routines that allow users to select the transport protocol to use and to select the encoding method that should be used with the data to be transferred. Transport protocols can range simple protocols such as UDP up to more complex protocols such as HTTP. We currently are using plain TCP for most of our transport.

Function calls across the different Spectrum APIs can be encoded using a variety of formats include plain text, XDR, and XML. We are currently using the eXpat XML library [4] to encode our calls. While we are current transferring our XML encoded messages using a simple TCP connection, in a real world setting this can easily be replaced with an implementation based on secure sockets layer (SSL) to improve security by adding SSL as a transport protocol.

An important aspect of Spectrum is that it can manage content based on a given policy across heterogenous platforms. As we explained previously in Section 2.2, envision a small set of base-level policy functions that can be parameterized to produce a wide range of storage polices. In order for this to work properly, all Spectrum-based applications must understand the base-level policies and how they can be parameterized. To address this issue, we treat each base-level policy as if it was a separate program. Each base-level policy should have a well known name and command "line" options for parameterization. In fact, in our implementation we pass parameters to base-level policies as a string that can be parsed using a `getopt`-like function. This format is easily understood and provides portability since byte order is not an issue in a string. Since this part of Spectrum is not on the critical data path, this type of formatting is not a performance issue.

3.2 Using the Spectrum Content Management System

In this section we show two examples of the use of the Spectrum Content Management System in our environment. The focus of our previous work has been content distribution for streaming media content [2] and network enabled digital video recording [3]. The Spectrum system is applicable to both scenarios as follows.

Figure 2 shows the Network Enabled DVR (NED) architecture. In this case all layers of the Spectrum architecture reside on the same physical device in a local configuration. The DVR application obtains program listings from some network source, deals with user presentation through a graphical user interface (GUI), and interface with the Spectrum system through the content management layer APIs. This combination of higher level functions allows the user to select both content to be stored and what storage policies to

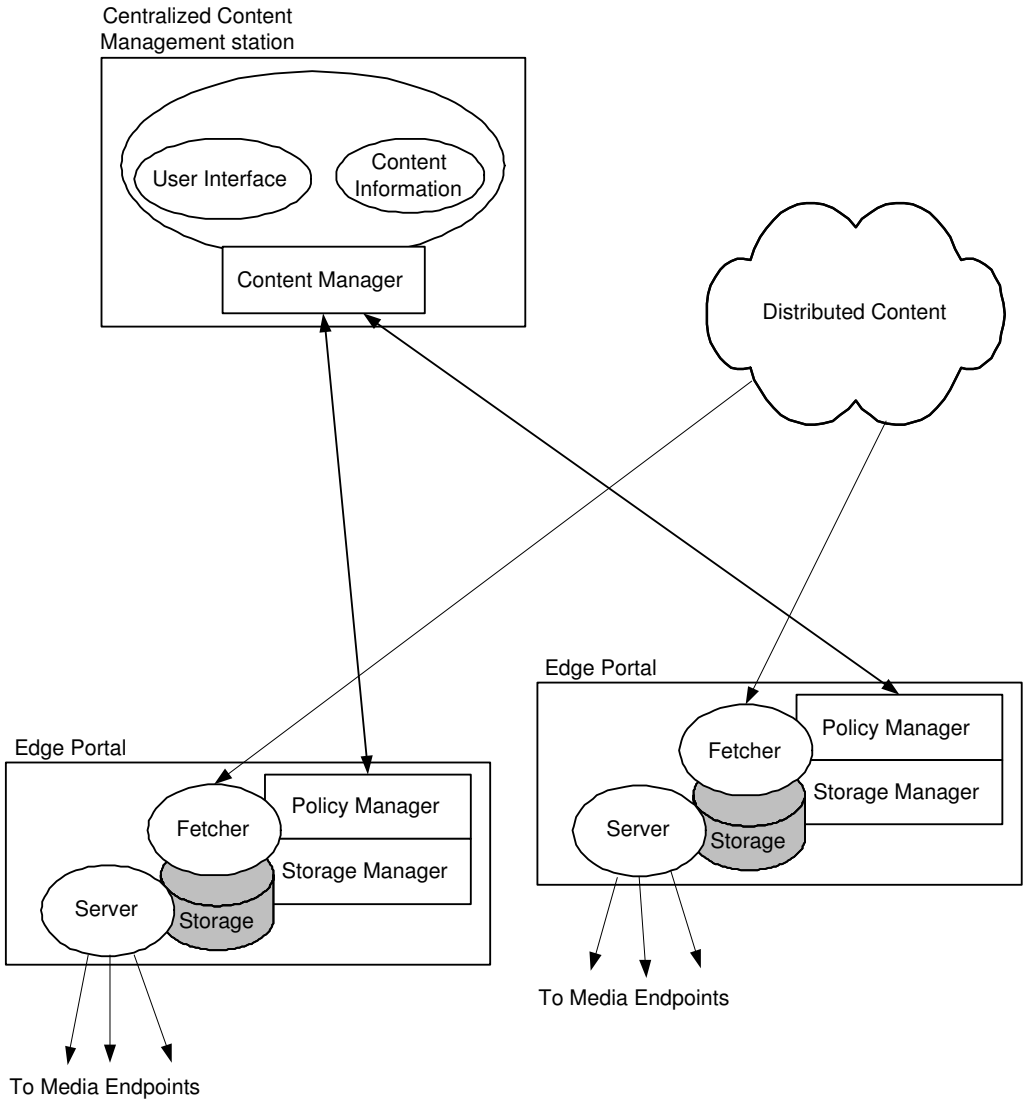

Figure 3: Spectrum in a Content Distribution Architecture

apply to such content. Obtaining the content (through the network or locally) and the subsequent storage on the local system is then handled by the policy and storage managers.

The use of Spectrum in a streaming content distribution architecture (e.g. PRISM [2]) is depicted in Figure 3. In this environment streaming media content (both live, canned-live and on-demand) is being distributed to edge portals from where streaming endpoints are being served. In our environment content distribution and storage is done from a centralized content management station which controls several of the edge portals. The centralized station allows administrators to manage the distribution and storage of content without requiring continuous communication between the content manager and the edge devices, i.e. once "instructions" have been given to edge devices they can operate independently until changes are to be made.

3.3 Spectrum Operational Example

To illustrate how Spectrum handles references to content, consider a Spectrum-based PVR application programmed to store one days worth of streaming content in a rolling window. To set up the rolling window, the application would use the content manager API to create a policy group and policy reference to the desired content. The establishment of the one-day rolling window policy reference would cause the policy manger to ask the storage manager to start receiving the stream. As each chunk of streaming data arrives, the policy manager executes the policy reference's "newclip" function. The "newclip" function adds a reference to each arriving chunk, and schedules a callback a day later. At that time, the policy will drop its now day-old reference to the content and the content will be discarded unless it is referenced by some other policy.

Now, consider the case where the user decides to save part of the content (e.g. a specific program) in the rolling window for an extra week. To do this, the application requests that the content manager add an additional new policy reference to the part of the content to preserved. Thus, the preserved content has two references to it: one from the rolling window and one from the request to preserve the content for an additional week. After one day the reference from the rolling window will be discarded, but the content will be

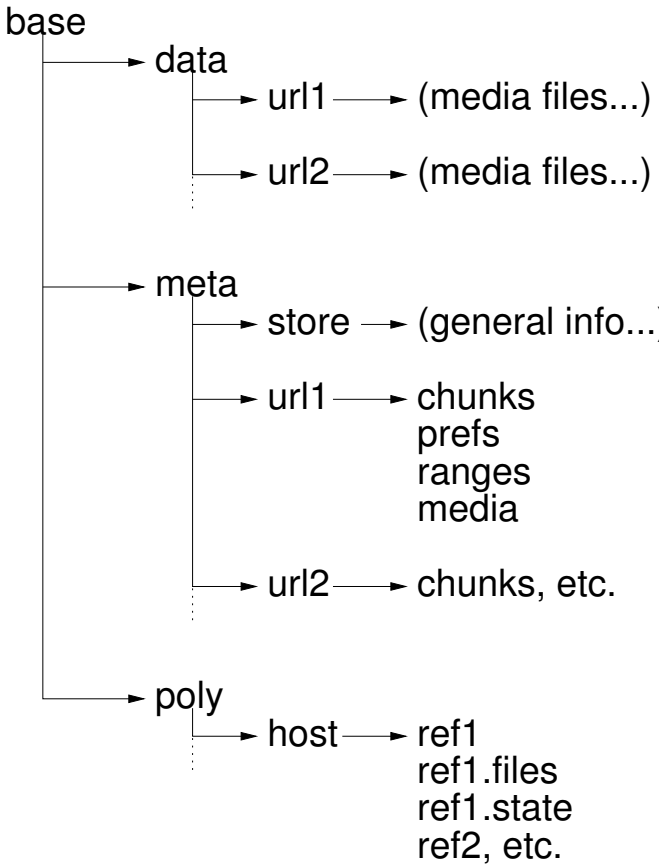

Figure 4: Data layout of Spectrum policy store

preserved by the second reference. After the additional week has past, the callback function for the second reference will be called. This function will discard the remaining reference to the content and as there are no remaining references the content will be freed.

In order to function in scenarios like the ones described above, Spectrum's policy manager must manage and maintain all the references to various chunks of media. These references are persistent and thus must be able to survive even if the machine maintaining them is rebooted. Our Spectrum policy manager implementation accomplishes this using the file and directory structure shown in Figure 4. There are three classes of data stored, and each class has its own top level directory. The directories are:

data: this directory is used by the storage manager to store each active URL's chunks of media. The media files can be encoded in any format, for example MPEG, Windows Media, or QuickTime. Note that this directory is used only if the storage manager is local. If the policy manager is using an external storage manager (e.g. a storage appliance), then the media files are stored remotely and are only remotely referenced by the policy manager.

meta: this directory contains general meta information about the storage manager being used and the data it is storing. General information is stored in the `store` subdirectory and includes the location of the store (local or remote) and information about the types of chunks of data the store can handle. The `meta` directory also contains a subdirectory per-URL that contains information about the chunks of data stored. The `chunks` file contains a list of chunks currently stored and their reference counts. The `prefs` file contains a list of active policy references that point to this URL. The `ranges` file contains a list of time ranges of data currently stored. Finally, the `media` file describes the format of the media being stored under the current URL.

poly: this directory contains a set of host subdirectories. Each host subdirectory contains the set of policy references created by that host. Information on each policy reference is broken up into three files. For example, a policy reference named `ref1` would be stored in `ref1`, `ref1.files`, and `ref1.state`. The `ref1` file contains information about the policy reference that does not change frequently. This information includes the base-policy and the parameters used to create the reference. The `ref1.files` file contains the list of references to chunks that *pref* `ref1` owns. Finally, the `ref1.state` file contains optional policy-specific state information that can change over time.

Together, these files and directories are used to track references in our implementation of Spectrum. Note that other implementations are possible. For example, a carrier-grade Spectrum manager might store all its policy and reference information in a high-performance database system.

4. RELATED WORK

Several authors have addressed the problem of the management of content in distributed networks. Much of the work focuses on the policy management aspect. For example in [5], the problem of serving multimedia content via distributed servers is considered. Content is distributed among server resources in proportion to user demand using a Demand Dissemination Protocol. The performance of the scheme is benchmarked via simulation. In [1] content is distributed among sub-caches. The authors construct a system employing various components, such as a Central Router, Cache Knowledge base, Subcaches, and a Subcache eviction judge. The Cache Knowledge base allows sophisticated policies to be employed. Simulation is used to compare the proposed scheme with well-known replacement algorithms. Our work differs in that we are considering more than the policy management aspects of the problem. After carefully considering the required functionality to implement content management in the networked environment, we have partitioned the system into three simple functions, namely Content manager, Policy manager and Storage manager. This has allowed us to easily implement and experiment with a prototype system.

Other related work involves so called TV recommendation systems which are used in PVRs to automatically select content for users, e.g. [6]. In the case where Spectrum is used in a PVR configuration this type of system would perform a higher level function and could clearly benefit from the functionalities of the Spectrum architecture.

Finally, in the commercial CDN environment vendors (e.g. Cisco and Netapp) have developed and implemented content management products and tools. Unlike the Spectrum architecture which allows edge devices to operate in a largely autonomous fashion, the vendor solutions typically are more tightly coupled to a centralized controller and do not have the sophisticated time-based operations offered by Spectrum.

5. CONCLUSION AND FUTURE WORK

In this paper we presented the design and implementation of the Spectrum content management architecture. Spectrum allows storage policies to be applied to large volumes of content to facilitate efficient storage. Specifically, the system allows different policies to be applied to the same content without replication. Spectrum can also apply policies that are "time-aware" which effectively deals with the storage of continuous media content. Finally, the modular design of the Spectrum architecture allows both stand-alone and distributed realizations so that the system can be deployed in a variety of applications.

There are a number of open issues that will require future work. Some of these issues include:

- We envision Spectrum being able to manage content on systems ranging from large CDNs down to smaller appliances such as TiVO [8]. In order for these smaller systems to support Spectrum they will require networking and an external API. When that API becomes available, we will have to work out how it can be fit into the Spectrum architecture.

- Spectrum names content by URL, but we have intentionally not defined the format of Spectrum URLs, how they map back to the content's actual name, or how the names and URLs should be presented to the user. While we previously touched on these issues elsewhere [2], we believe there is more work to be done and that consensus-based standards on naming need to be written.

- In this paper we've focused on content management for continuous media objects. We also believe the Spectrum architecture can be applied to any type of document including plain files, but we have yet to work out the details necessary to support this in our prototype environment.

- Any project that helps allow multimedia content to be easily shared over the Internet will have legal hurdles to overcome before it can achieve widespread acceptance. Adapting Spectrum to meet legal requirements will likely require more technical work.

6. REFERENCES

[1] K. . Cheng and Y. Kambayashi. Multicache-based Content Management for Web Caching. Proceedings of the First International Conference on Web Information Systems Engineering, Jume 2000.

[2] C. Cranor, M. Green, C. Kalmanek, D. Shur, S. Sibal, C. Sreenan, and J. van der Merwe. PRISM Architecture: Supporting Enhanced Streaming Services in a Content Distribution Network. IEEE Internet Computing, July/August 2001.

[3] C. Cranor, C. Kalmanek, D. Shur, S. Sibal, C. Sreenan, and J. van der Merwe. NED: a Network-Enabled Digital Video Recorder. 11th IEEE Workshop on Local and Metropolitan Area Networks, March 2001.

[4] eXpat. expat.sourceforge.net.

[5] Z. Ge, P. Ji, and P. Shenoy. A Demand Adaptive and Locality Aware (DALA) Streaming Media Server Cluster Architecture. NOSSDAV, May 2002.

[6] K. Kurapati and S. Gutta and D. Schaffer and J. Martino and J. Zimmerman. A multi-agent TV recommender. Proceedings of the UM 2001 workshop, July 2001.

[7] ReplayTV. www.sonicblue.com.

[8] TiVo. www.tivo.com.