Enabling Full Service Surrogates Using the Portable Channel Representation

Micah Beck*, Terry Moore*
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee
Knoxville, TN 37996-3450, USA
+1 865 974 3548
{mbeck, tmoore}@cs.utk.edu

Leif Abrahamsson
Chistophe Achouiantz, Patrik Johansson
Lokomo Systems AB
Svärdvägen 27
aSE-182 33 Danderyd, Sweden
+46 8 5490 4380
{leif, chris, patrik}@lokomo.com

ABSTRACT

The simplicity of the basic client/server model of Web services led quickly to its widespread adoption, but also to scalability and performance problems. The technological response to these problems has been the development of technology for the creation of *surrogates* for Web servers, starting with simple proxy caches and reverse proxies, and leading more recently to the development of Content Distribution Networks. Surrogate technologies based on caches have proved quite successful in reducing the load due to delivery of cacheable content (HTML files and images), but in general they cannot replicate services that are implemented through the execution of programs at the server. A *full service surrogate* is a technology that is designed to address this issue directly because it is a *copy* or *mirror* of the server that is created, managed and updated automatically. One of the central issues in the creation of full service surrogates is *portability* of interpreted content, and the representation of metadata necessary to support execution. In this paper we describe the Portable Channel Representation, which is an XML/RDF encoded data model developed to enable full service surrogates, and we discuss the implications of the increasing importance of executable Web services.

Categories and Subject Descriptors

D.3.3 [**Computer-Communication Networks**]: Distributed Systems – *client/server, distributed applications.*

General Terms

Management, Standardization, Languages

Keywords

Web server, dynamic content, portability, replication, content distribution, surrogate, mirroring.

1. INTRODUCTION — TWO VIEWS OF SURROGATES

For those who share the goal of creating wide area information systems that are ubiquitous, universally accessible, and media rich, the simplicity of the original Web model represents both a profound strength and a profound liability. It is a strength because it makes it relatively easy for people who have publishable content to set up a Web site and make that material widely available; this ease of use for content publishers is one of the primary reasons that the Web spread with such incredible speed when it was introduced. But this same simplicity also makes the basic approach liable to scalability problems that have likewise been apparent from early on [5].

In the basic Web model a client generates a Hypertext Transfer Protocol (HTTP) request that can be fulfilled at a unique server, and the server's response takes the form of a set of objects delivered in an HTTP response [4]. For simple cases the response to a given request is stable over at least short periods of time, and when it changes, it changes in a predictable manner [7]. Because each request is fulfilled at a single unique server, only one server must be configured to respond to any particular request. Since Web clients are distributed across the globe, however, the more numerous and media hungry they become, the more bandwidth the responses to their requests consume across an increasingly congested Internet backbone. Poor performance for users, in the form of high interaction latencies and slow transfer times, tends to be the result.

There are several recent and ongoing commercial efforts to solve this problem by reengineering the Web to create high performance *Content Distribution Networks* (*CDNs*). CDNs are based on the use of *surrogates*, i.e., on the deployment of *multiple nodes within the network that can, under the control of the content provider, fulfill the service requests of users in the appropriate manner.* According to the working definition used in the discussions of IETF's Web Replication and Caching Working Group, a surrogate is

> *A gateway co-located with an origin server, or at a different point in the network, delegated the authority to operate on behalf of, and typically working in close co-operation with, one or more origin servers. Responses are typically delivered from an internal cache* [12].

Although this wording weights the idea of a surrogate towards cache-based implementations, surrogates of other forms have been well known and widely used to achieve the same purpose for some time. The *full service surrogate* model that we propose below draws

* The work of Micah Beck and Terry Moore was supported by the National Science Foundation Internet Technologies Program under grant # ANI-9980203, Internet2, and a grant from the IBM Corporation.

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

on this alternative tradition in order to create an approach to CDNs that we believe has novel capabilities and strengths.

This "full-service" approach derives from a characteristic analysis of how a Web service, and therefore a service surrogate, is constituted. On this view a Web service node generally consists of a server process running in a conventional operating system environment. The state of this server is defined by two kinds of files: configuration files and stored source objects. When the server process receives a typical request, such as an HTTP GET, it uses the information in the HTTP header to interpret the control information in the configuration files to determine which source objects must be retrieved in order to fulfill the request, what their type is, and how they must be interpreted. In some cases, the request generates a call-out to some other Web service node, and the response generated by that node is relayed back to the client.

Note that this description of a Web service node is quite general, encompassing both Web caches and other Web servers for various protocols. Indeed, in our view a Web cache can be characterized as a Web service node whose stored objects are previous responses to Web requests that have been generated by origin servers and captured by the cache. Capturing HTTP responses according to a cache management policy is the most convenient way to implement a surrogate, since it does not require the operation of the origin server to be duplicated. As the widespread use of Web caching and cache-based CDNs suggest, this approach is highly effective where requests are predictable and sufficiently stable over time.

Unfortunately, many services implemented in the Web today do not fit the simple form of the caching model. Some services, for example, are implemented dynamically through the execution of a program by the Web server [14]. Important types of dynamic, non-cacheable content services include (1) content generated from a database query, (2) quickly changing content (e.g. live content), and (3) highly interactive interfaces (e.g. those needing an applet). The most common mechanisms for implementing such dynamic services are programs invoked through the Common Gateway Interface (CGI) and Java servelets executed as part of the server process [10, 15]. With either mechanism, however, the service request leads the server to a stored, executable object, and this object is subsequently executed using an interpreter determined by its type. To replicate such non-cacheable services, you have to replicate the server itself, creating an identical copy that can act as a *full service surrogate*, invoking executable replicas of the appropriate source objects on every request it fulfills.

Now the desire to create such a full service approach to content distribution was one of the primary motivations for the Internet2 Distributed Storage Infrastructure (I2-DSI) project. This project attempted to draw on ideas from traditional Internet mirroring in order to implement a general, scalable network of servers for the replication of both static content and dynamic content services across heterogeneous operating environments [2]. In I2-DSI the basic unit of replication is characterized as a *channel*, i.e. as "*… a collection of content which can be transparently delivered to end user communities at a chosen cost/performance point through a flexible, policy-based application of resources*." From the beginning this concept of a channel explicitly included the kinds of dynamic content that cache-based approaches have problems addressing.

But the idea of mirroring channels with dynamic content faces challenging problems of its own. This fact is evident to anyone who has tried to use a standard mirroring approach to replicate Web servers with executable source objects. It proves too hard to do because, as our analysis above shows, the behavior of such a Web server depends on two critical factors and both of them are problematic when you try to replicate them:

- **Server configuration files are non-standard** — The essential configuration files that determine the response of a Web server to client requests are not standardized; they depend on the particular server software and can even differ between server versions.
- **References to source objects are file system dependent** — The Web server configuration files refer to directory names that are dependent on the installation of stored source objects in the file system, and file systems tend to vary across platforms and system management styles.

Taken together these two factors mean that trying to create a surrogate by simply copying the configuration files and source objects to the target node only works where the servers are identical. Where they are not identical, the mirroring operation must take into account any heterogeneity in the architecture, operating system, or server software on the target node, and the resulting copy must also be compatible with the other operations the target node is configured to perform. For this reason, porting a sophisticated Web site to a non-identical server node can be a frustrating, time-consuming task, where even substantial amounts of effort cannot always guarantee that the result will be an identical copy of the site.

The concept of a *Portable Channel Representation* (*PCR*) was developed in the context of the I2-DSI project to attack precisely this problem, and thereby make possible a full service alternative to content distribution mechanisms based on caching technology alone [1]. Because PCR focuses on the problems associated with mirroring of source objects, it draws from the substantial experience in cooperative Internet mirroring that predates the Web [9, 11, 13]. It is also informed by more recent work on active networks from the past decade, the same work that has been incorporated into an Extensible Proxy Framework in order to add dynamic services to cache-based content distribution networks, which are still based only on capture and replay of Web responses [18].

2. CONTENT PORTABILITY AND CHANNEL REPLICATION

A useful place to begin the discussion of content portability is with the common distinction between "active" and "static" content. Our view is that in the area of content distribution, the distinction between active (dynamically interpreted) content and static (or passive) data is blurred to the point of being meaningless. One reason for this confusion is that the most common forms of Web programming are declarative, and for that reason are not considered to be forms of programming at all by the author/programmer. But examining a few cases at close range suggests otherwise.

The most legitimately "static" content on the Web is a file delivered by FTP; admittedly in this case, the bits stored on the disk are not interpreted at all by the FTP server, but are simply passed over a network link. But at only a slightly higher level of complexity, simple HTML files are in fact interpreted by the HTTP server to generate an HTTP response, even though they are often thought of as static content. The most universal form of this interpretation occurs when the server rewrites the URLs in hyperlinks, or even more ambitiously, when the server processes directives in the HTML and generates text to replace them in the response. Moreover, HTML files are augmented by metadata that determines how they are processed and what the nature of the response generated should be: <META> tags can cause redirection to another URL entirely, among other altered behaviors, and password protection alters the behavior of the server, although not the contents of the delivered file.

It is convenient to think of an HTTP response as if it were a simple copy of the HTML file which generates it, as this allows us to conceive of the "static" portion of the World Wide Web as a file delivery mechanism, i.e., a form of communication network. Caching technology can then be thought of as an extension of that network. But the conclusion that we draw from a review of the facts is that almost every form of Web content is in some measure interpreted, and therefore liable to encounter portability issues. For that reason a sounder operating assumption to make is that *source files are not passive data but programmed objects that must be interpreted in order to generate the server's response.* We believe that if solutions to the Web's scalability problems take the distinction between active and static content for granted and focus on caching technology alone, they will be ill equipped to deal with the Web as it really is, i.e., as a distributed system with chronic programming and portability issues.

A few examples from ordinary Web authoring and content management make obvious the relevance of this point of view to questions of portability. While standard HTML processing is usually portable, any *Server Side Includes* (SSIs) that pages contain are server-dependent and they may make reference to auxiliary files by file name (rather than URL). This will tend to make them non-portable. URLs for non-HTML file types, such as streaming media objects, use metadata files that invoke local auxiliary programs and can make explicit reference to file names. These local metadata files are not, as a rule, portable. Again, many HTTP server features, such as multi-lingual processing and security, are controlled through server configuration files that are not standardized and that typically make reference to local directory names. Finally, CGI programs and servelets commonly make use of local interpreters, files, and other resources through interfaces that are not portable across servers.

Once we begin to think of the Web as a distributed system with standard programming and portability issues, then it becomes clear that, as with other such systems, the key to portability is the use of standard languages and application programming interfaces (APIs) which can be interpreted uniformly on a broad class of execution platforms. As things currently stand, the Web falls far short of satisfying this condition.

HTML may be a standard language, but it comes with a distressingly broad choice of APIs. These range from standard HTML with no server-side directives, to HTML with a small class of directives supported by many available HTTP servers, and finally to HTML with powerful database access extensions supported only by a small number of HTTP servers. Similarly, in typical HTTP implementations, although servelets have a well-defined language (Java) and API, CGI programs are arbitrary executable files that are invoked by the HTTP server with no notion of their language or API. CGI programs are typically implemented in an interpreted language such as Perl, but a given CGI program may be compiled binary. Interpreted languages, such as Java byte code or Perl, have the benefit of placing an intermediary layer of software between the code and the full power of the operating system and the machine architecture. However, incompatibility between different versions of the same language and the use of powerful, non-portable APIs can eliminate such benefits. As a result, CGI programs may be highly non-portable, and there is no metadata available to determine their portability characteristics.

Now there are basically two strategies for achieving portability in the face of this diversity of APIs. One approach says that we must all agree on a single API and use only the features of that API in order to achieve "write once, run anywhere" status. This is what Java promises. The other approach requires only that code port *safely*, not *universally*. According to this point of view, it is not necessary for everyone to use a universally implemented language and API, just that the choice be known and that the interpreter be safe even if the API is violated. We term this freer and more open approach *descriptive,* in contrast to *prescriptive,* one-language-only approaches to portability.

It is worth noting that the one-language-only strategy for content portability was attempted unsuccessfully in the early Java–only Content Distribution products from Marimba in the context of "desktop push." More generally, we believe that the more expansive goal of "write once, run anywhere" cannot be practically achieved in a world where languages, APIs and execution platforms change constantly and the behavior of the developer community is not under centralized control. The approach to portability we have developed, which is based on the *Portable Channel Representation* (*PCR*), is an instance of a descriptive, metadata-based strategy that offers more freedom to developers to choose their language and API, but requires them, in return, to provide the CDN with critical information characterizing the portability of the resulting content. While PCR does not promise to make every Web site portable to every platform, once the information characterizing portability is encoded as PCR metadata, management software can check to see if that content can be safely ported to a given target server.

3. THE PORTABLE CHANNEL REPRESENTATION

3.1 The PCR Data Model

The Portable Channel Representation (PCR) was originally defined in order to facilitate the creation of mirrors on the heterogeneous servers of the Internet2 Distributed Storage Infrastructure [1-3]. It addresses each of the problematic areas of mirror creation highlighted above:

- *Server-Independent Specification of Behavior* — A PCR description is an encoding of metadata (the eXtensible Markup Language (XML) [6] and the Resource Definition Framework (RDF)[17]) that specifies the behavior of the server in response to a set of requests, collectively known as a "channel." In order to avoid dependence on configuration files specific to particular server software, PCR specifies in a platform-independent manner the source object and the method of interpretation that should be invoked on any particular request.
- *File-system-Independent Specification of Objects* —References to objects from within a PCR channel description do not name them through their installed location in a file system directory structure, but instead use local names defined by a "file store".

The PCR description and the file store together define a complete channel description that can be correctly implemented on any platform that correctly interprets both elements.

PCR's *descriptive* approach associates with each user request both a source file and a type. The type specifies the language and API of that file and is used to determine the interpreter that will be used to generate a response. Since these types are not reflected in the contents of the source file, but are specified by the PCR metadata external to the file, a single source file may be treated as having different types when accessed through different requests (e.g. using different protocols and servers). Possible types include standard HTML, GIF, 3Mb/s MPEG-1 video, Perl with a standard minimal API (no file or network access), Java bytecode with an SQL database access API.

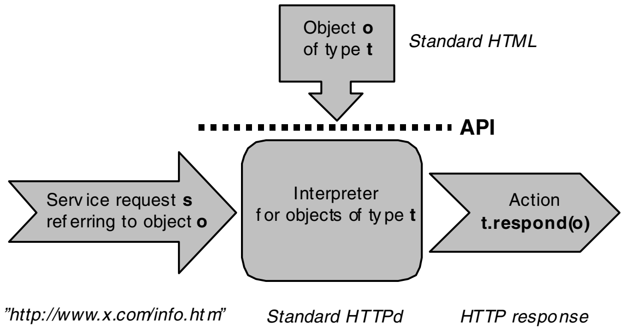

Figure 1: The PCR Behavior Model

The PCR data model is a language intended for the specification of the behavior of a server. This language does not support arbitrary behaviors (i.e., it is not a general model such as Java), but instead works within a highly structured *server behavior model*, shown in Figure 1. The central notion in PCRs server behavior model is a *request fulfillment rule*, which can be thought of as a *pair* consisting of a *pattern* and an *action*. When a request is presented to a server and matches a *pattern*, the server responds by performing the associated *action*. In concrete terms,

- a pattern consists of several fields which correspond to a standard URL: domain, port, and object name, and
- an action is specified by a source file and a type.

Every type is associated with a *method* or *interpreter*, and the action specified by the rule is to invoke that method on the specified data file. For example, a request associated with a stored file with type "Standard HTML" would be interpreted by a standard HTTP server allowing no non-standard extensions.

Every source object type defines syntactic rules for data objects of this type. We refer to these syntactic rules as the *language* in which the object is expressed. If there is a single language with multiple variants, we refer to these variants as *APIs*. Thus, HTML is a *source object* language, but a specific set of allowable Server Side Includes defines an *API*. Perl and Java byte codes are also source object *languages*, and each of them may have multiple *APIs*. The combination of *language* and *API* together define determine the object *type*. The server must perform a combination of install- and run-time checks to ensure that a particular object conforms to its declared type.

The sum of all APIs constituting the channel is referred to as the *Channel API*. Extending the functionality of PCR requires introducing a new type, defining the interpreter and extending the Channel API to include the new object type. New service types can easily be added as they emerge, thus the PCR approach is highly extensible. The current implementation of PCR (Section 5) supports HTML, streaming media and server-side HTML extensions, but it will be extended to support CGI execution (Perl, Java), database access (read-only) and other APIs.

In section 4 below we present the RDF schema for PCR in detail. But to understand the different aspects of that schema, as well as the File Store API that complements it, it is helpful to be familiar with the way in which it used to create CDNs based on full service surrogates. The next few sections describe the different facets of the PCR approach to creating CDNs.

3.2 PCR-based Content Distribution

PCR-based content distribution has the same goal as cache-based content distribution, i.e., to serve content from distributed servers preferably situated in proximity to end-users. In the example (Figure 2), an Internet Service Provider (ISP) provides a PCR-based distribution system for Web sites. To distribute a channel the content provider *publishes* a channel to a point of distribution located within the network of an ISP. From this point the ISP *distributes* the channel to a number of PCR servers at the edge of his network. Users of the ISP then access the channel locally through PCR servers instead of the origin server, generally improving the quality of their application experience in the process. Thus PCR servers are surrogates, and the server at the distribution point plays the same role as the "point of origin" does in cache-based CDNs.

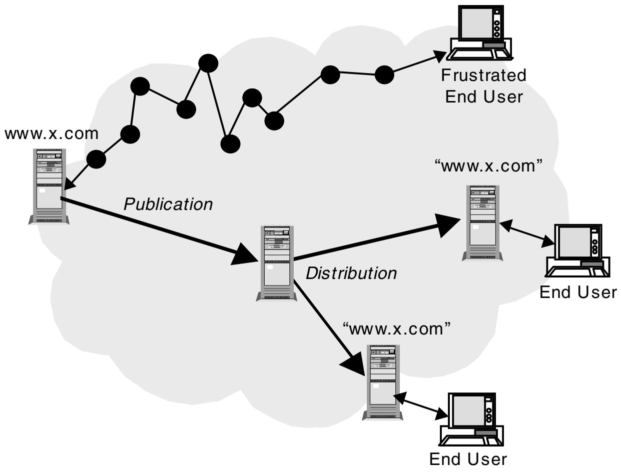

Figure 2: Distribution of a channel using PCR

The distribution process can be divided into four subprocesses, sequenced as pictured below (Figure 3). They include

- Creation and publication of a PCR channel, also known as *content authoring* (see sec. 5)
- Distribution of the channel to a number of PCR servers
- Installation of the channel on the PCR servers
- Service of the channel to end-users

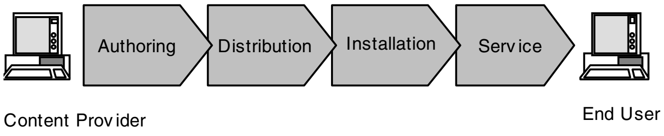

Figure 3: The PCR Distribution Process - Logical View

In the distribution of a channel the metadata describing the different aspects of the channel are separated from the source objects (application files) that constitute its substance (Figure 4). This separation is handled by the PCR tools and servers and is transparent to both the content providers and end users.

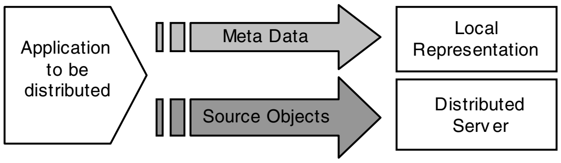

Figure 4: The PCR Distribution Process - Meta data and source objects

The different software components of this system are shown in Figure 5 below. A PCR-encoding of an Internet application is generated using a *Channel Creator* or *Channel Authoring* tool. Channel authoring tools, which can be extensions of current Web authoring tools, create a PCR representation of this application that, together with the application files (i.e., the channel source objects), is published to the *Distribution Server*. The Distribution Server distributes the channel to a number of *Channel Servers*. The Channel Server (or actually the *Installer* of the Channel Server) generates a local representation of the channel, using the PCR-representation. To complete the distribution process, the local representation of the channel at the Channel Server is activated, giving the end-user to local access to the channel with enhanced quality of service.

3.3 Content Authoring for Highly Portable Surrogates

In discussions of Content Delivery Systems (CDSs), the expression “content authoring” refers not only to the creation of the actual information or digital material that will get delivered, but also to the process of modifying that material so that it is well adapted for the mechanisms that the CDS will use to deliver it. In our approach to creating highly portable surrogates, this process primarily focuses on discovering and encoding the PCR metadata necessary to create PCR-enabled surrogates from Web sites or other Internet applications. Our experience shows that without the appropriate tools, this tends to be a labor intensive and potentially error prone task. To address it, tools need to be provided to create PCR metadata for two different cases:

- **PCR for legacy sites** — Some of the metadata required to PCR-enable an existing service entity, say a Web site, is present in the embedding of the selected service entity in the server directories of its Web host and the configuration files of its servers. If a Web site uses only standard servers, then it is possible to derive the PCR and create the file store automatically from the file system. For instance, a Web site which uses the Apache file server and a Real Audio streaming server can be easily mined for PCR metadata.

- **Native PCR channels** — The other approach is to use a PCR-based tool to author the channel from the start, entering the necessary metadata directly and never embedding it in the file system at all. While this approach is more ambitious from a tool-development point of view, it guarantees that the required behavior of the site is correctly expressed, and is not limited in expressiveness by the capabilities of standard servers.

Tools that implement a mixed approach are also possible. Structured authoring tools, which maintain their own internal metadata structures, could also generate PCR as a publication option. Thus a tool like Microsoft FrontPage might become a PCR Channel-authoring tool, much as Microsoft Word has become an HTML authoring tool. The key point in all these cases, however, is that the discovery and encoding of the essential PCR metadata should be as automated as possible, so as to maximize the completeness and accuracy of the encoding and minimize effort for the user.

3.4 PCR Publication

PCR also carries information concerning the publication of the channel. Information is provided specifying date and time when the channel should be made accessible, for how long it should be active as well as when to delete it.

PCR introduces a level of indirection between the identity of a source file and the service request that accesses it. A file may exist in the file store, but unless the current PCR view of the channel accesses it, it has no impact on the behavior of the node in serving the channel. The ability to switch instantaneously between PCR views allows us to atomically update a channel.

Instantaneous switching between PCR views is possible because the PCR view is a much smaller data structure than the source file, and a node can easily hold more than one. Thus, a PCR file can be delivered and interpreted but held in an inactive state until a synchronization event makes it the active view of the channel. This feature allows simple, secure and seamless updates of

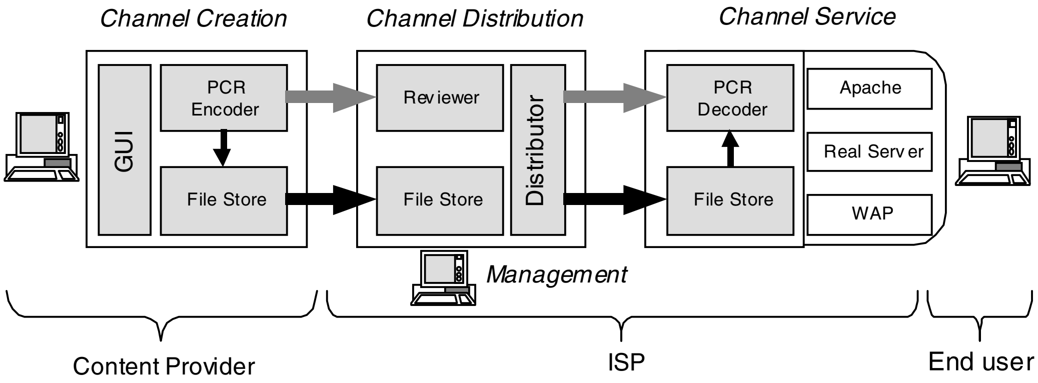

Figure 5: PCR Distribution Process – Software Components for Content Management

channels, be it a complete update of the entire channel or a partial update of near-real-time information. It is even possible to maintain different views of the channel for different sessions, if, for instance, a session ID is encoded in the service request.

4. THE RDF SCHEMA FOR PCR

We have chosen the Resource Description Framework (RDF) as the standard language for encoding PCR. RDF is a foundation for processing metadata, providing interoperability between applications that exchange machine-understandable information. The RDF data model [16] defines a simple approach for describing interrelationships among resources in terms of named properties and values. RDF properties are used to represent attribute-value pairs associated with resources, and they also represent relationships between resources.

The data model, however, provides no mechanisms for *declaring* properties, nor does it provide any mechanisms for *defining* the relationships between properties and other resources. That is the role of *RDF Schema*. The RDF Schema defined in [8] is a collection of RDF resources that can be used to describe properties of other RDF resources (including properties) defined by *application-specific* RDF vocabularies. The RDF schema for PCR is shown in Figure 6.

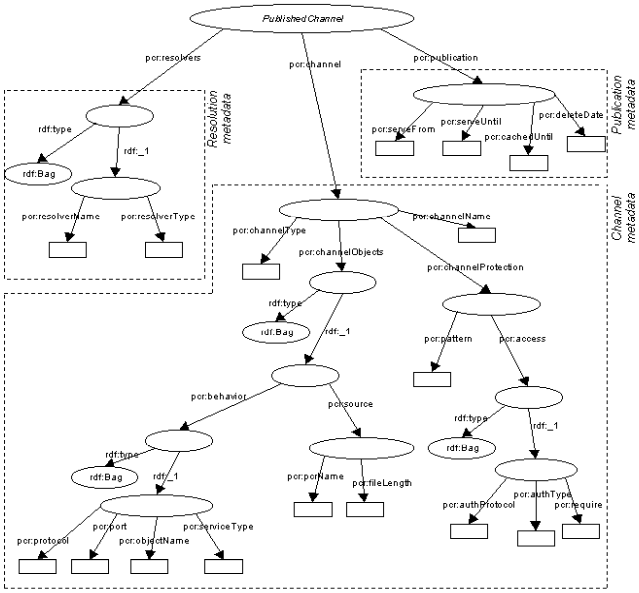

Figure 6: The PCR RDF Schema

4.1 The Channel Schema

A **Channel** describes the behavior of the server in response to a class of requests.

- ***channelType***
  The channeltype declares that the channel conforms to the invariant which is associated with that type.
- ***channelObjects(****Bag)*
  Each source object has associated with it a set of a set of a server behaviors. Each behavior assigns a type to the underlying data file in order to invoke the interpreter of the appropriate type. The type is a function of the protocol, port number and objectname used in the request.
  - ***behavior (Bag)***
    - ***protocol***
      For instance, FTP or HTTP.
    - ***port***
      The port number on which the request arrives. The server must listen on all such ports.
    - ***objectname***
      The objectname referenced in the request.
    - ***serviceType***
      The servicetype specifies the interpreter to be invoked, such as HTTP, FTP, streaming MPEG, Perl.
  - ***source***
    - *pcrName*
      The URI of the source object data file.
    - *fileLength*
      The length of the source object data file.
- ***channelProtection***
  One of the basic services offered by Web servers in addition to interpretation of source objects is protection of requests in a variety of protocols. In this simple version of PCR, a single pattern specifies a set of objectnames to protect, and then a set of access rules can be specified for objects matching that pattern.
  - ***pattern***
    A pattern which matches objectnames in requests.
  - ***access(Bag)***
    An authorization indicates the protocol which is to be protected, the type of protection to be used, and the required credential.
    - ***authProtocol***
      For example, HTTP or FTP**.**
    - ***authType***
      For example, password/id or certificate.
    - ***require***
      The required credential
    - ***channelName***
      A unique name used for the channel by content management utilities.

4.2 Publication Metadata

Because of the changing nature of Web services, it is necessary to determine when the requests should be fulfilled.

- ***serveFrom***
  The server must not fulfill requests based on this channel description before this starting date and time (on its local clock).
- ***serveUntil***
  The server must not fulfill requests based on this channel description after this starting date and time (on its local clock).
- ***cachedUntil***
  After this time, falling serveFrom and serveUntil, the channel content must not be allowed to reside in caches

(using protocol-specific cache control mechanisms to ensure this). This allows atomic updates to be made at the server after this time.

- ***deleteDate***
  A time, falling no sooner than serveUntil, when the channel files should be deleted from the server.

4.3 Resolution Metadata

In order for client requests to reach a server, the channel must be registered with one or more resolvers such as the DNS.

- ***resolverName***
  The name under which the channel should be resgistered
- ***resolverType***
  The resolver type associated with that name (e.g. DNS).

4.4 Published Channel

The information necessary to publish a channel and make it accessible by clients consists of a channel specification, publication metadata, and a set of resolver specifications.

- ***channel***
- ***publication***
- ***resolvers (Bag)***

4.5 The PCR FileStore

A standard tool for the implementation of Web services is the use of the native file system of the server which provides those services to provide a mapping between client requests and stored data. Thus, the filename field of an HTTP GET request is usually interpreted as a file name relative to some root directory, determined by the configuration of the HTTP server. While this is a convenient implementation mechanism, reliance on this mapping can lead to non-portability; for example:

- Early Microsoft file systems did not support long file names or file extensions of more than three characters.
- Some embedded file systems do not support hierarchical directory structures.
- File system permissions are sometimes used to implement important security constraints *but they are not portable across operating systems.*
- One direct approach to this problem would be to include all data files along with their associated metadata in the PCR description. Then no names external to the PCR description would be needed, and portability of names and other attributes would be ensured. However, this leads to implementation issues because of the fact that very efficient means exist for the movement of files across processors, and encoding them in the PCR description makes use of those mechanisms difficult.

We have chosen instead to factor the storage of files and their binding to names into a separate facility we call the PCR FileStore. A FileStore is simply a mechanism for storing files and associating them with names that are local to the FileStore. When a FileStore is moved between servers, the binding of names to files does not change. This means that FileStore names can be used in the PCR description file without any loss of portability. The metadata associated with requests that access the files is still implemented in the PCR description.

The problem with using the native file system for name-to-file binding becomes more acute when files are accessed during the interpretation of content, be it from a HTTP Server Side Include or as input to a program written in Perl. In every case, current implementations result in a local file name being used by the interpreted code, and any use of naming which reflects global knowledge of the file system directory structure will be non-portable. The FileStore also solves this problem, as PCR allows for the naming of local files that reside in the FileStore. The combination of PCR and the FileStore can thus implement some services normally provided by the local file system, eliminating dependences on non-portable mechanisms.

A FileStore can be as simple as a tar or zip file archive which is copied between servers as a single file transfer, or it can be a directory which is distributed using an efficient differential update mechanism such as rsync or even a proprietary block-level replication mechanism implemented by a mass storage archive [19]. The intent is to allow data movement to be implemented in the most efficient and cost-effective manner independent of the distribution of the PCR description.

4.5.1 The PCR FileStore API

A File Store has an API, which allows the sender to pass it a source file and return a File Store Name, which is meaningful only to that File Store. The File Store Name is encoded in the PCR, and delivered to the recipient along with the File Store, usually through a separate mechanism. The receiver unpacks the PCR and uses the File Store API to retrieve the source file using the File Store Name found in the PCR.

Examples of File Store delivery mechanisms are transfer of an archival image, rsync between file systems or block level mirroring of disks. PCR is usually delivered as a stream to a connected socket, although file transfer protocols such as FTP may also be used.

4.5.1.1 Transaction Management

- ***abort()***
  Abort transaction and rollback.
- ***beginTransaction()***
  beginTransaction locks the whole channel identified by channelname.
- ***endTransaction()***
  endTransaction must end a transaction that was started by call beginTransaction
- ***newTransaction(String channelname)***
  newTransaction returns a transaction object which is used to request service from FileStore.

4.5.1.2 FileStore Management

Current FileStore API implements transaction based management. Typically, to do object management on a specific channel you must request a transaction object from FileStore for that channel. The reason for a transaction based API is that we have experienced a need to synchronize object management on channels to keep a PCR channel view consistent with the FileStore.

- ***deleteChannel()***
  deleteChannel removes channel and all its bindings from FileStore.

- ***transfer(String host, int port, String userid, String password)***
  transfer FileStore, bindings and source objects, to remote host.

4.5.1.3 Object Management

As described in previous section, to perform object level management on a channel you request a transaction object from the FileStore. You use that object to perform following services:

- ***deleteObject(String pcrname)***
  deleteObject removes a binding from the FileStore (removing source object if local copy in FileStore).
- ***getObjectPcrname(URL sourceLocator)***
  getObjectPcrname puts a source object in the FileStore and gets the pcrname (filestore name) for the resource that is local to the FileStore.
- ***getObjectSource(String pcrname)***
  getObjectSource returns the data of a source object identified by a pcrname.
- ***getObjectSourceLocator(String pcrname)***
  getObjectSourceLocator returns the local locator to a source object identified by a pcrname.
- ***setValidObjects(String[] pcrnames)***
  setValidObjects is used to make sure that application view is consistent with FileStore state. That is, only those bindings that are listed by pcrnames are valid, the rest should be removed from.PCR Applications

5. SOFTWARE IMPLEMENTATION OF PCR-BASED FULL SERVICE SURROGATES

Swedish based ***Lokomo Systems*** has developed a solution to the problem of the end-to-end management Content Distribution Networks in heterogeneous environments using a full service surrogate approach based on PCR. Lokomo's software suite supports the creation and management of Web sites and other content-based services that can be automatically replicated to CDN edge servers acting as full service surrogates. More detailed examples showing how PCR technology makes this approach possible are given below.

The Lokomo's CDN software suite (Figure 7) is divided in four components:

- The *Channel Creator*, which is a tool for creating PCR channels,
- The *Distribution Server,* which is a server for storing and forwarding channels,
- The *Channel Server*, which is the full service surrogate, managing and executing channels at the edge of the CDN, and
- The *Management Server*, which allows the CDN operator to manage the system as a whole.

Since a key goal of this approach is to allow content providers to focus on a single version of their content, and yet support the local distribution of that content on CDNs managed by different Internet Service Providers, this suite supports a variety of operating platforms, network protocols, and web server types.

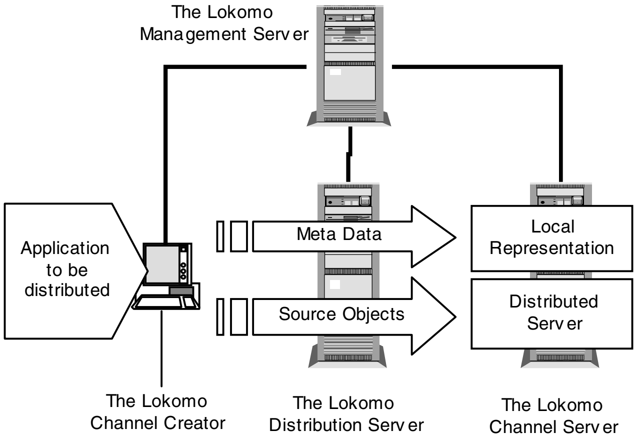

Figure 7: Lokomo Software Suite

The Lokomo software suite has been designed to embody all the important characteristics that CDN builders will expect from an adequate CDN software suite, regardless of the approach it takes: the *scalability* and *redundancy* to handle a large number of complex nodes under extremely heavy traffic; the *flexibility* to support various special system configurations and network topologies; requisite *portability* to manage heterogeneity in the edge server environment; robust *extensibility* to third party application servers that can provide new types of content and services to end users; and *ease of use* that supports agile control of the CDN from a single point in the network.

Experiences we have already had from two widely different applications illustrate some of the qualities that set the PCR approach apart. In particular Lokomo software has been used to create PCR-based full service surrogates (i.e., *channel servers*) that support active content in the form of Apache Server Side Includes (SSIs) and Java Servlets. In addition, the use of PCR extends beyond web sites to interactive services. We successfully use the representation for distributing services like streaming media and games.

Both these application examples have provided useful experience in supporting active web content with PCR technology. The examples described here illustrate very different requirements on the interpretation of content in the surrogate in two examples:

- An ISP has very high requirements that code port safely even when the API is violated. Many different content providers share the ISP's surrogates and the services must not interfere with each other.
- An ASP, on the other hand, is in full control of a number of dedicated surrogates. The ASP require a much more powerful API in the surrogate and care less about competition of shared resources as his service is the only one executing on the server. Consequently, if the API is violated only his service will suffer.

5.1 Example 1: Distributed Web Hosting with Server Side Includes

The Lokomo software suite is used by a large ISP network to manage distribution of Web sites from their Web hotel to their distributed Web servers. A content provider can gain access to distributed hosting services with minimum effort simply by building their Web site on the hotel. Content distribution is initiated and supervised by the ISP from an easy-to-use web interface. By moving the rich, interactive and executable content to the edge of the access network the end-users experience of the service is significantly increased.

Web sites located at the ISP Web hotel are translated into PCR by an authoring tool (the *Channel Creator*) automatically extracts metadata from the installed source files. The language and API used by the content provider in constructing their Web site is standard HTML extended with Lokomo's dialect of Apache Server Side Includes (Lokomo Apache SSIs). SSIs are directives embedded in HTML pages that the server parses and interprets before the page is served to the Web client. Interpretation of a directive generates an HTML fragment that replaces the SSI directive in the page. While there are several different dialects of HTML augmented with different directives implemented by specific Web servers, Apache was chosen due to its popularity.

Every Apache SSI directive takes an argument that specifies some value or file from the server's execution environment, and the safety of the directive depends on which directives are allowed and the values allowed for their parameters.

- The `include` and `exec` directives name files in the server's environment, in the former case for textual inclusion in the HTML page and in the later case for execution and inclusion of their output in the page. The standard implementation finds these files in the file system directory in which the HTML page resides. In the Lokomo dialect, `include` names are bound in the PCR to files in the File Store, and `exec` names are restricted to a fixed set of programs that are installed in a distinguished local directory associated with the server.
- Other directives, such as `echo` and `set`, name variables which, in the standard implementation, may reflect more of the server's execution environment than is safe or portable. The Lokomo dialect restricts these variables to a small, portable set.

By restricting the scope of Apache SSIs to files which are part of the channel, to executing a standard and restricted set of external programs, and to accessing a standard and restricted set of environment variables, Lokomo's Apache SSI provides a safe and highly portable API for applications requiring a moderate level of execution on the Channel Server.

5.2 Example 2: Distributed Media Management using Java Servelets

A commercial service provider with a very large and processor-intensive Internet service deploys the Lokomo software suite in order to increase end-users quality of service for CPU-intensive applications. The channels are comprised of hundreds of thousands of large objects accessed by end users all over the world. Access possibilities and access rights differ from user to user and follow different services schemes. Highly CPU-intensive interpretation is performed on the source objects to generate responses that require fast and secure download to the end-user. The Lokomo implementation replaces an earlier one based on a centralized ASP model that suffered from unpredictable delays and slow delivery speeds. Using Lokomo software, the source objects are encoded in PCR and distributed to a number of data centers. The whole distribution chain is managed by the Lokomo system, including selected parts of the massive central database and service application.

The executable portion of this service is implemented using Java servlets (JSP), which are invoked from directives embedded in standard HTML pages just as they are for SSI. And as in the case of SSI, files names are mapped to the FileStore using PCR bindings. However, the standard Java servelet API includes very powerful features. For example, a servlet can create a socket and connect to another server or access a local read-only SQL database. Such features can cause conflicts on a surrogate shared by many different services and content providers. In this specific example, few restrictions have been applied to the standard API, and the ASP has designed his service carefully, making sure that the servlets will port to his distributed environment. Misuse of this API, could cause the service to fail.

6. CONCLUSION

The notion of Web surrogates is usually associated with caches and reverse proxies that do not replicate interpreted content. Nonetheless, mirroring is an obvious solution for the problems of scalability in the Internet, and has long been used in the context of distributing files by FTP. Mirroring has sometimes been applied to the case of more general Web services, but at a cost in human effort that is proportional to the complexity of interpreted content to be replicated and the heterogeneity of operating environment. In this paper we have presented the Portable Channel Representation (PCR), which is a mechanism for the automated management of replicated or mirrored content that addresses the problems of portability introduced by interpreted or "active" content. It is based on this idea: if creating an abstraction of the operating environment in which the interpretation of source objects occurs can automate the management of mirrored content in heterogeneous environments, then surrogates and CDNs based on mirroring are feasible. The PCR-based system now being developed promises to give us back a natural and intuitive solution to the problem of scalability in Internet content, which must otherwise be addressed by more intricate and more limited solutions based on HTTP caching.

The World Wide Web was created around a simple and highly structured notion of content (HTML) and a standard protocol for delivering it (HTTP). Of the many benefits that could accrue from the adoption of the Web as the a universal fabric for information interchange, some derive from the use of HTML as a uniform content language, including the ability to use a single encoding of content for many diverse purposes and the ability to use a single encoding of content across many different computing platforms. Other benefits accrue from using HTTP as uniform delivery mechanism, including the ability for a single server platform to fulfill requests from diverse clients and the ability to develop networking infrastructure which is adapted to the characteristics of that protocol. As the Web has developed, the growing domination of the latter has meant the progressive diminution of the former.

The source of the problem is that, as a language, HTML succumbed to a universal tendency in the development of computing systems: programmers will modify any tool until it becomes a fully general computing environment, with little or no respect for the strong properties intended by the original designer. That is how functional programming languages get augmented with imperative constructs, graphics formats become multimedia scripts, and declarative text markup languages become page layout tools. The

Web has been augmented by sources of content, such as CGI scripts, which bear no resemblance to HTML, but which do conform to HTTP and magnify the power of the Web as a delivery mechanism. As a consequence, the Web becomes something amazing: a medium for commerce and entertainment, a competitor for the television and the telephone, a fabric for human interactions of all sorts. In the process, however, many of the strong properties that might have made Web content more manageable have been lost.

The move towards the use of XML in the Web is providing a framework for many communities to define highly structured notions of content that are intended to provide manageability, and their intent is to defend those tools against extensions which would violate their fundamental design principles. The Portable Channel Representation is an attempt to define a language that factors out, from the myriad mechanisms (languages and APIs) for generating HTTP responses, enough commonality and structure to allow for automated management of content. If it succeeds it will restore to the management of Web content a property that some people are not even fully aware has been lost — the independence of content from the execution environment of the server, i.e. portability.

Standards activity in Web content has focused on the format and interpretation of source objects: HTML, XML, GIF, JPEG, MPEG, etc. These activities have enabled a generation of content authorship and management tools that can accurately preview the behavior of Web browsers, publish entire Web sites into heterogeneous operating environments, and modify and combine Web content that has been developed independently. It has not been possible, however to achieve the same degree of platform independence for more highly interpreted content due to a lack of standards, and this has limited the degree to which content management can be automated in an interoperable manner. Acceptance of a representation standard for interpreted content generally, such as PCR, would overcome this limitation and enable a much greater degree of automation in content management across heterogeneous platforms.

REFERENCES

[1] Beck, M., Chawla, R., Dempsey, B. and Moore, T., Portable Representation for Internet Content Channels in I2-DSI. in *4th International World Wide Web Caching Workshop*, (San Diego, CA, 1999).

[2] Beck, M. and Moore, T. The Internet2 Distributed Storage Infrastructure Project: An Architecture for Internet Content Channels. *Computer Networking and ISDN Systems*, *30* (22-23). 2141-2148.

[3] Beck, M., Moore, T. and Dempsey, B., Internet2 Distributed Storage Infrastructure, Innovative Computing Laboratory, University of Tennessee, 2001. http://dsi.internet2.edu

[4] Berners-Lee, T., Fielding, R. and Frystyk, H., Hypertext Transfer Protocol -- HTTP/1.0, Internet Engineering Task Force, 1996. http://www.w3.org/Protocols/HTTP/1.0/spec.html

[5] Bowman, C.M., Danzig, P.B., Hardy, D.R., Manber, U. and Schwartz, M.F., The Harvest Information Discovery and Access System. in *The Second International WWW Conference*, (Chicago, IL, 1994), 763-771.

[6] Bray, T., Paoli, J. and Sperberg-McQueen, C.M., Extensible Markup Language (XML) 1.0, World Wide Web Consortium, 2000. http://www.w3.org/TR/REC-xml

[7] Brewington, B. and Cybenko, G. How dynamic is the Web? *Computer Networks*, *33* (1-6). 257-276.

[8] Brickley, D. and Guha, R.V., Resource Description Framework (RDF) Schema Specification 1.0, World Wide Web Consortium, 2000. http://www.w3.org/TR/rdf-schema/

[9] Browne, S., Dongarra, J., Grosse, E. and Rowan, T. The Netlib Mathematical Software Repository (Accessible at http://www.dlib.org/) *D-Lib Magazine*, 1995.

[10] Coar, K. and Robinson, D., The WWW Common Gateway Interface Version 1.1, 1999. http://CGI-Spec.Golux.Com/draft-coar-cgi-v11-03-clean.html

[11] Cooper, G., Hill, J., Kelly, B., Rogerson, R., Rusbridge, C., Tedd, M. and Wiseman, N., CEI/ACN Working Group on Mirror Services: Final Report, Joint Information Systems Committee, 1998. http://www.jisc.ac.uk/pub98/mirrors.html

[12] Cooper, I., Melve, I. and Tomlinson, G., Internet Web Replication and Caching Taxonomy, IETF Internet Engineering Working Group, 2001. http://www.wrec.org/archive/200011/att-0093/01-draft-ietf-wrec-taxonomy-06.txt

[13] Grosse, E. Repository Mirroring. *Journal on Mathematical Software*, *21* (1).

[14] Houh, H., Lindblad, C. and Wetherall, D., Active Pages: Intelligent Nodes on the World Wide Web. in *First International Conference on the World-Wide Web*, (Geneva, Switzerland, 1994).

[15] Javasoft, Java Servlet Specification v2.3, Sun Microsystems, 2000. http://java.sun.com/aboutJava/communityprocess/review/jsr053/servlet23_PublicDraft1.pdf

[16] Lassila, O. and Swick, R., Resource Description Framework (RDF) Model and Syntax Specification, World Wide Web Consortium, 1999. http://www.w3.org/TR/REC-rdf-syntax/

[17] Swick, R., Miller, E., Schloss, B., Singer, D. and Brickley, D., Resource Description Framework (RDF), World Wide Web Consortium, 2001. http://www.w3.org/RDF/

[18] Tomlinson, G., Orman, H., Condry, M., Kempf, J. and Farber, D., Extensible Proxy Services Framework, Internet Engineering Task Force, 2000. http://www.ietf.org/internet-drafts/draft-tomlinson-epsfw-00.txt

[19] Tridgell, A. and Mackerras, P. The rsync algorithm, Australian National University, Canberra, Australia, 1996.