Enabling Full Service Surrogates Using the Portable Channel Representation

Micah Beck, Terry Moore

Innovative Computing Laboratory, University of Tennessee

Leif Abrahamsson, Chistophe Achouiantz, Patrik Johansson

Lokomo Systems

Abstract

The simplicity of the basic server/client model of Web services led quickly to widespread adoption but also to scalability and performance issues. The technological response to these issues 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 cachable content (HTML files and images) but cannot in general replicated services that are implemented through the execution of programs at the server. The notion of a *full service* surrogate is a copy or *mirror* of the server which 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. This paper we describe the Portable Channel Representation, a an XML/RDF encoded data model developed to enable full service surrogates, and discuss the implications of increasing importance of executable Web services.

Keywords

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

Word Count: 5,800

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 content to publish 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 also been apparent from early on [1].

In 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 [2]. 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 [3]. 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 working definition currently 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 [4].*

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 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 its type is, and how it 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 Web service node 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 [5]. Important types of dynamic, non-cachable 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 [6, 7]. With either mechanism, however, the service request leads the server to a stored object that is executable, and which is subsequently executed using an interpreter determined by the object's 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 services across heterogeneous operating environments [8]. 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. This proves to 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 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 usually 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 only on caching technology [9]. Because PCR focuses on the problems associated with mirroring of source objects, it draws from the substantial experience in cooperative Internet mirroring which predates the Web [10-12]. It is also informed by more recent work from the past decade on active networks that has been incorporated into an Extensible Proxy Framework, adding dynamic services to cache-based content distribution networks; but this approach is still based only on capture and reply of Web responses [13].

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 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 server processes directives in the HTML generates text to replace them in the response. Moreover, HTML files are augmented by metadata that determines how they are processed and the nature of the response generated: `<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 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 just 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 programming and portability issues.

A few examples from ordinary Web authoring and content management make the relevance of this point of view to questions of portability apparent. While standard HTML processing is usually portable, any server side includes that pages contain are server-dependent, and they may make reference to auxiliary files by file name (rather than URL), which will tend to make them non-portable. URLs for non-HTML file types, such as streaming media, 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 far from satisfying this condition.

HTML may be a standard language, but it a 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 endeavors to offers more freedom to developers to choose their language and API in return for requiring them to provide critical information affecting the portability of the resulting content. While PCR does not promise to make every Web site portable to every platform, once the information critical to 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 (PCR)

3.1 The PCR Data Model

PCR was originally defined in order to facilitate the creation of mirrors on the heterogeneous servers of the Internet2 Distributed Storage Infrastructure [8, 9, 14]. 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) [15] and the Resource Definition Framework (RDF) [16]) 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.

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*, then 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.

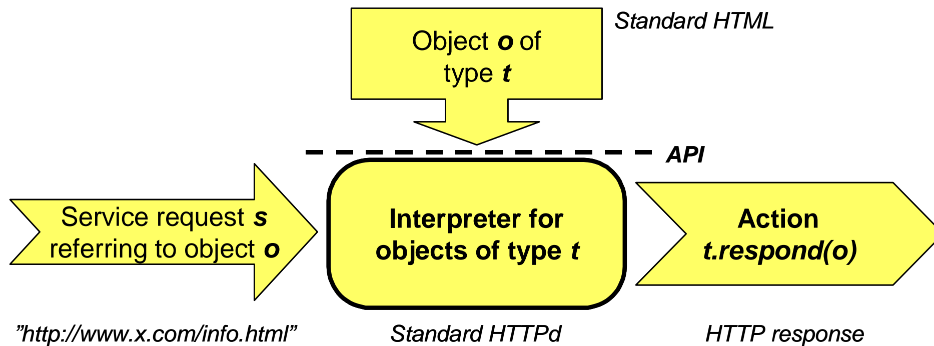

Figure 1: *The PCR Behavior Model*

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 includes defines an *API*. Perl and Java byte code are also a 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 is a question of 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 and streaming media, but will be extended to support WAP (WML), server side HTML extensions, CGI execution (Perl, Java) and database access (read-only).

In section 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 the 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 below (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.

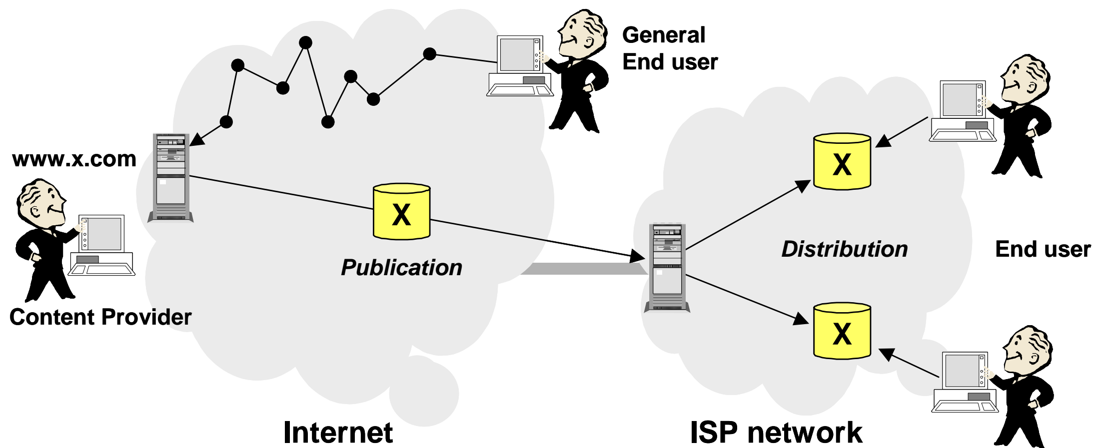

Figure 2: Distribution of a channel using PCR

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

- Creation and publication of a PCR channel, also known as content authoring (see XX)
- 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

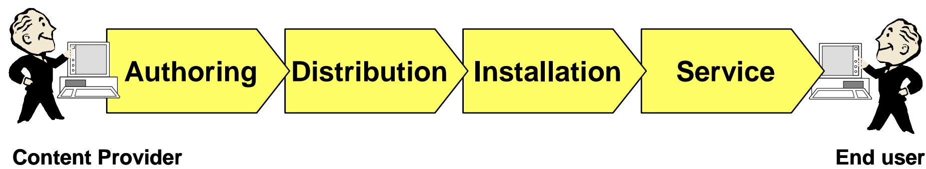

Figure 3: *The PCR Distribution Process – Logical View*

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 the end users.

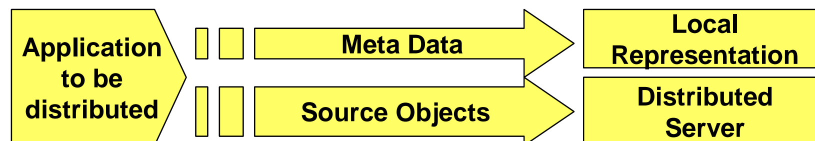

Figure 4: *The PCR Distribution Process – Meta data and source objects*

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 an extension of current Web authoring tools, creates PCR representation of this application and, together with the application files (i.e. the channel source objects) is publishes the channel 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. The installer program makes adaptations for the specific local platform, e.g. operating system, file system, and web server.

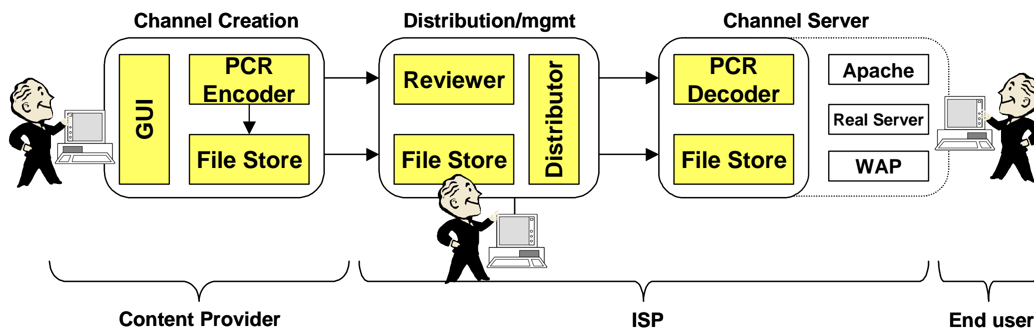

Figure 4: *The PCR Distribution Process – Software Components for Content Management*

To complete the distribution process, the local representation of the channel at the Channel Server is activated, allowing the end-user to enjoy 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, WAP sites, or other Internet service entities. 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 a 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.

Thus, a channel may be distributed to the distribution points and installed but held in an inactive state until a time event causes it to become the active view of the channel. This feature allows simple, secure and seamless updates of channels, be it a complete update of the entire channel or a partial update of near-real-time information.

PCR introduces a level of indirection between the identity of a source file and the service request, which accesses it. Files may exist in the source file, 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. This means that 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 time or synchronization event causes it to become the active view of the channel. 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 XML Resource Description Format as the standard language in which to encode PCR (RDF), but other descriptive notations can serve as well. RDF is a foundation for processing metadata; it provides interoperability between applications that exchange machine-

understandable information. The RDF data model, as specified in [17], defines a simple model for describing interrelationships among resources in terms of named properties and values. RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF properties 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 [18] 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.

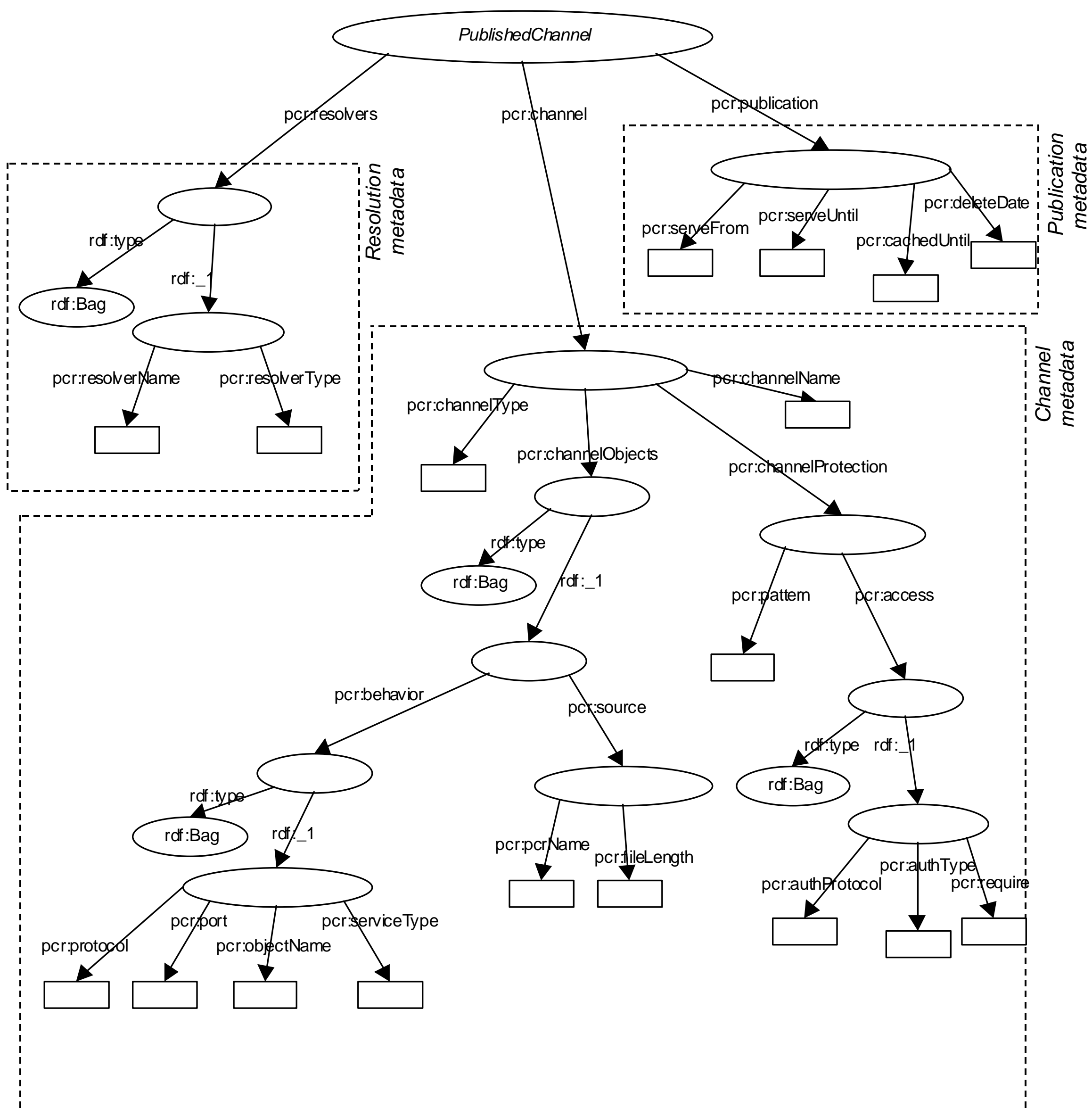

Figure 6: *The PCR RDF Schema*

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 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 of 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 definition of APIs which define the naming of local files through the FileStore. The FileStore can thus implements the 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(boolean)**

  beginTransaction locks the whole channel identified by channelname.

- **endTransaction()**

  endTransaction must end a transaction that was started by call beginTransaction

- **newTransaction(String)**

  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 it's bindings from FileStore.

- **newTransaction(String channelname)**

  newTransaction returns a transaction object which is used to request service 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.PCR Applications

5. PCR-based Tools and Systems

Swedish based Lokomo Systems has developed a comprehensive product suite for Internet content distribution based on the Portable Channel Representation. The software suite automates and supports the creation, management and delivery of Web sites including automated support for the creation of replicated edge servers (mirrors). PCR technology allows Lokomo's tools to manage distribution of complex sites and dynamic content in a heterogeneous environment corresponding to the interconnected independent networks which comprise the Internet. A variety of system hardware and operating systems, network protocols, web server types and versions can be managed. This allows a content provider to focus on developing, maintaining and testing only one single version of his site while providing local access in a number of edge networks managed by different Internet Service Providers. The system also provides the Content Provider with enhanced content security as PCR includes ways to control and limit content access.

Lokomo Systems' implementation of the Portable Channel Representation is divided in four software components.

- A tool for creating PCR channels, the Lokomo Channel Creator
- A server for storing and forwarding channels, the Lokomo Distribution Server
- An edge server for managing and executing channels, the Lokomo Channel Server
- A management and supervision system, the Lokomo Management Server

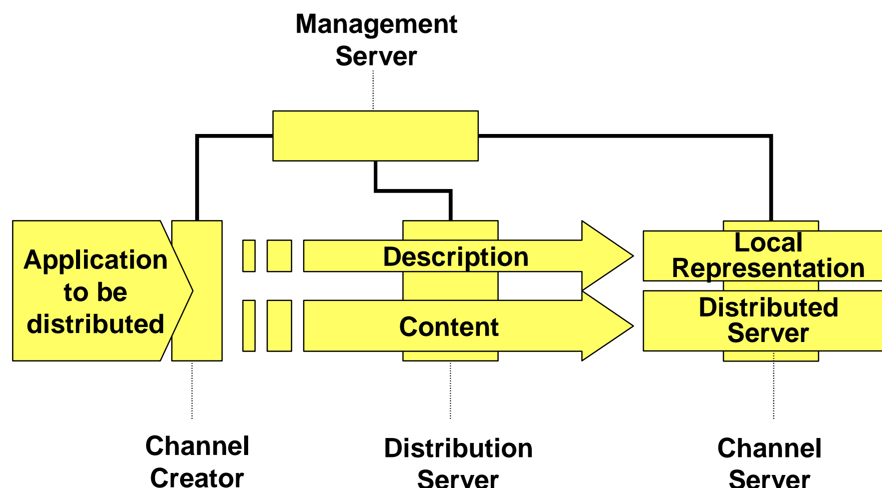

Figure 7: *The Lokomo software suite*

The Lokomo system is a complete implementation of the end-to-end management and distribution of replicated content within the PCR framework.

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 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 than must 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 and which addresses the problems of portability introduced by interpreted or "active" content. It is based on the idea that if the automated management of mirrored content in heterogenous environments can be enabled by creating an abstraction of the operating environment in which the interpretation of source objects occurs, then surrogates 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 stuctured 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 characteristsics of that protocol. As the Web has developed, the growing domination of the latter has meant the progressive degradation 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 delivery mechanism. As a consequence, the Web becomes something amazing: a medium for commerce and enterainment, 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 managable have been lost.

The move towards the use of XML in the Web is providing a framework for many communities to define high structured notions of content that are intended to provide managability, 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 (lanaguages and APIs) for generating HTTP responses, providing enough commonality and structure to allow for automated management of content. In doing so it will restore to the management of Web content a property that some people are not even fully aware that it was 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 such as PCR for interpreted content generally would enable a much greater degree of automation in content management across heterogeneous platforms.

Bibliography

1. Bowman, C.M., et al. *The Harvest Information Discovery and Access System*. in *The Second International WWW Conference*. 1994. Chicago, IL.

2. Berners-Lee, T., R. Fielding, and H. Frystyk, *Hypertext Transfer Protocol -- HTTP/1.0*. 1996.
3. Brewington, B.E. and G. Cybenko. *How dynamic is the Web?* in *Ninth International World Wide Web Conference*. 1999. Amsterdam, Netherlands.
4. Cooper, I., I. Melve, and G. Tomlinson, *Internet Web Replication and Caching Taxonomy*. 2000, Internet Engineering Working Group.
5. Houh, H., C. Lindblad, and D. Wetherall. *Active Pages: Intelligent Nodes on the World Wide Web*. 1994. First International Conference on the World-Wide Web.
6. Microsystems, S., *Java Servlet 2.3 Specification*. 2000.
7. NCSA, *The Common Gateway Interface*.
8. Beck, M. and T. Moore, *The Internet2 Distributed Storage Infrastructure Project: An Architecture for Internet Content Channels.* Computer Networking and ISDN Systems, 1998. **30**(22-23): p. 2141-2148.
9. Beck, M., et al. *Portable Representation for Internet Content Channels in I2-DSI*. in *4th International World Wide Web Caching Workshop*. 1999. San Diego, CA.
10. Grosse, E., *Repository Mirroring.* Journal on Mathematical Software, 1995. **21**(1).
11. Jones, P., *ibilio Linux Mirrors*. 2000.
12. Service, U.M., *Annual Report 1 Aug, 1999 to 31 July 2000*. 2000, Joint Information Services Committee.
13. Orman, H., *Extensible Proxy Services Framework*. 2000.
14. Beck, M., B. Dempsey, and T. Moore, *Internet2 Distributed Storage Infrastructure Project*. 1999, Innovative Computing Laboratory.
15. Bray, T., J. Paoli, and C.M. Sperberg-McQueen, *Extensible Markup Language (XML) 1.0*. 1998, W3C.
16. Miller, J., T. Berners-Lee, and R.R. Swick, *Resource Description Framework (RDF)*., W3C.
17. Lassila, *Resource Description Framework (RDF) Model and Syntax Specification*. 1999, W3C recommendation.
18. Brickley, e.a., *RDF Schema Specification*. 1999, W3C Proposed Recommendation.
19. Tridgell, A. and P. Mackerras, *The rsync algorithm*. 1998, Australian National University: Canberra, Australia.