Implementing a Caching and Tiling Map Server: a Web 2.0 Case Study

Zao Liu, Marlon E. Pierce, and Geoffrey C. Fox
Community Grids Lab
Department of Computer Science
Indiana University
Bloomington, IN
{zaliu, mpierce,gcf}@cs.indiana.edu

Neil Devadasan
Polis Center
Indiana University Purdue University Indianapolis
Indianapolis, IN
ndevadas@iupui.edu

Abstract: *We describe our efforts to build a caching and tiling map server that greatly improves the performance and interactivity of traditional geographic map servers. We have used this system to integrate and effectively federate 15 Indiana county map servers with Google map images and state-wide ortho-photography data. Our approach is an example of the so-called Web 2.0 style of development, in which we integrate external, third party services into a higher level service. This approach also allows for lightweight client development using relatively simple JavaScript programming libraries. We demonstrate this by building a Google Map client interface to our tile server. Finally, we discuss our initial efforts to make collaborative clients using a shared event model that captures and broadcasts browser events to other, listening browsers.*

Keywords: Geographical Information Systems, Web map services, Web 2.0, Web-based collaboration

Introduction

Modern Geographical Information Systems (GIS) [1] provide a service-oriented architecture for interacting with geographical data sets and related maps. Web-based GIS systems are architected around the same principles as more general Web service systems based on SOAP [2], WSDL [3], and REST. Mirroring the World Wide Web Consortium and OASIS Web service standards-making bodies, the Open Geospatial Consortium [5] defines open standards for messages, XML data formats, and access protocols that are specific to the GIS community. In addition to OGC-based services, there are many companies (such as ESRI and AutoDesk) that provide proprietary, commercial solutions. Services from these various providers are not normally interoperable.

The methods of the traditional GIS community have been challenged in the last two years by the emergence of new, lighter-weight approaches towards building clients and integrating data. The availability of Google Maps, Google Earth, Microsoft's TeraServer, Yahoo! Maps and similar systems has enabled enthusiasts and part-time developers to make highly interactive Web interfaces to these companies' services and to integrate their maps with local data.

Google Maps in particular is an important example of the so-called Web 2.0 development approach [6]: Google has built and maintains a high performance, highly scalable map service (available for free) that has a relatively simple, JavaScript-based programming interface. This simple but powerful public interface to a very complicated service is the hallmark of Web 2.0, since it democratizes the client development process: very little programming skill is required to build custom Web applications and to combine them with data from other sources (so-called Web "mash-ups"). The ProgrammableWeb [7] is an excellent source for browsing mash-ups and discovering APIs to on-line services. Currently over 380 services make themselves available for mash-up building. Google Maps is used in about 50% of the registered mash-ups.

Google Maps also provides an important lesson in interactivity: by storing data in map tiles and by using a programming technique known as Asynchronous Javascript and XML (AJAX) [8], they are able to provide highly interactive interfaces that don't rely on direct user requests for map updates. Instead, the user drives the map updates indirectly by panning and zooming.

AJAX is a development technique rather than a new programming language. It relies heavily on the relatively recent standardization of XmlHttpRequest within JavaScript engines in most major browsers. XmlHttpRequest provides a mechanism for the page running in a browser to call back to its original Web server to obtain additional information without going through the direct, user driven HTML <form/> submission process. That is, the browser's request/response cycle does not have to be directly initiated by a user hitting a submit button. Instead, it can be initiated indirectly by user interactions. In the case of Google Maps, for example, the user's map panning can initiate a request to the server for more map tiles to be cached locally. From the user's point of view, the panning process is seamless, and minimal interruption in interactions takes place.

We present in this paper an examination of these issues through a specific case study: federating Google Map data with more detailed county data obtained from map servers run by state and local governments in Indiana. This is a particularly interesting example of federation, since there are 15 counties with public, on-line GIS services as well as a collection of state-wide services. These services collectively span four different, non-interoperable GIS products (ESRI ArcIMS, ESRI ArcMap, Autodesk MapGuide, and the OGC-based Minnesota Map Server). The county services have very detailed layer information: Marion County (Indianapolis) GIS services [9], for example, provide more than 100 map layers (such as park boundaries, parcel boundaries, voting precincts, and school districts) that are obviously not available from Google. These maps also provide higher level zooming capabilities and pin-point addressing that are superior to Google's geocoding service.

What the county GIS services lack, however, is scope: county GIS services stop at the county boundaries, but natural and man-made disasters (floods, tornadoes, chemical spills, etc) do not. There is currently no means for integrating data from multiple services into a unified view. Thus we see integrating local GIS services with Google's broader coverage into a single system as both an interesting Computer Science challenge as well as an activity with potential benefits to emergency management, disaster planning, and similar problems.

Building a Federated Cache Server

System Architecture: The basic system architecture is depicted in Figure 1. Pre-existing and externally managed map servers for various Indiana counties are shown at the top, as is Google's map service. In order to harvest map data from these servers, we build adapters that can be invoked by the Cache Server to construct appropriate requests to the county servers. Adapter construction is described below. We have also built Google Map adapters that can directly harvest data from Google. This is described in more detail in [10].

Map segments from the various county servers can be requested in any size, but to support integration with Google map tiles through Google Map client libraries, we will make bounding box requests for tiles that will be identical to the tiles used by Google Maps.

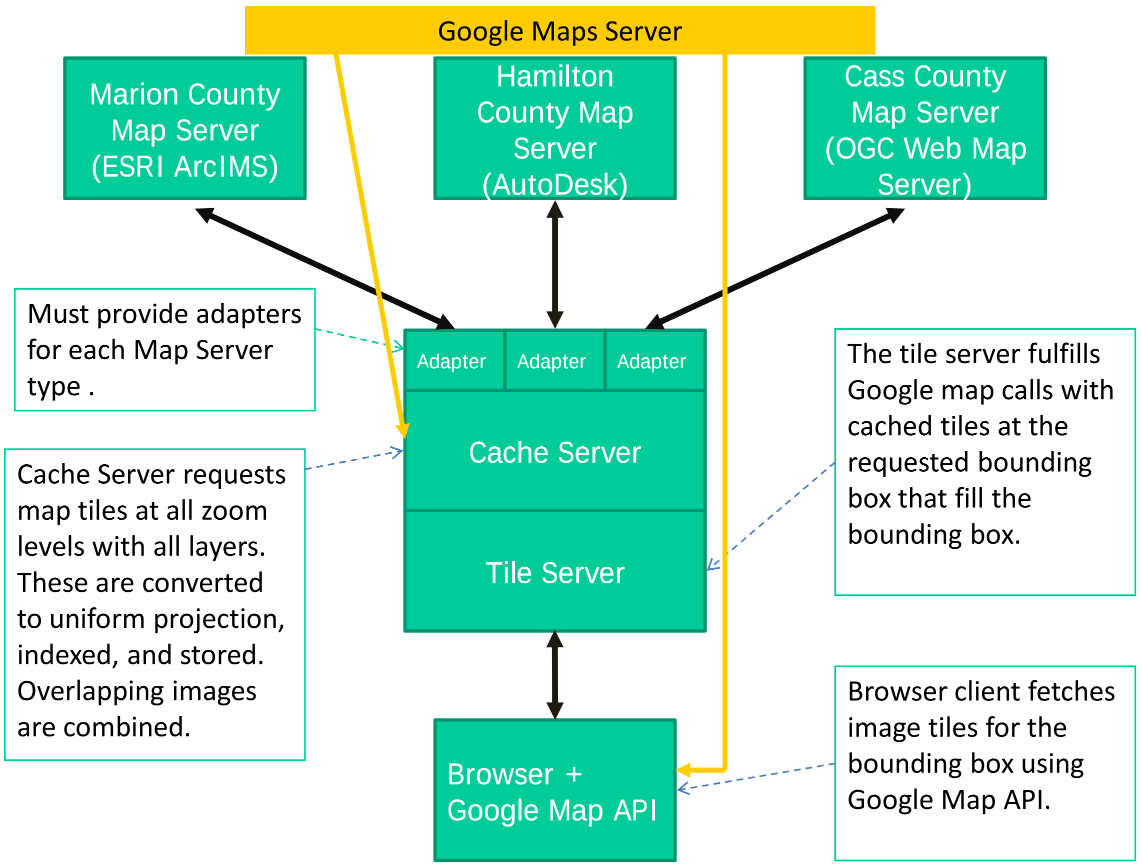

Google Maps Server
Marion County Map Server (ESRI ArcIMS)
Hamilton County Map Server (AutoDesk)
Cass County Map Server (OGC Web Map Server)
Must provide adapters for each Map Server type .
Adapter
Adapter
Adapter
Cache Server
Tile Server
Cache Server requests map tiles at all zoom levels with all layers. These are converted to uniform projection, indexed, and stored. Overlapping images are combined.
The tile server fulfills Google map calls with cached tiles at the requested bounding box that fill the bounding box.
Browser client fetches image tiles for the bounding box using Google Map API.
Browser + Google Map API

Figure 1 shows the basic system architecture. Solid arrows indicate network connections (Web Service request/response over HTTP). Filled boxes indicate network-enabled system components.

Message Patterns, Adapters, and Federation: All of the local GIS map servers that we have encountered follow a request-response style message pattern: the requesting agent (the adapter in Figure 1) constructs a request for maps, specifying the image bounding box's latitude and longitude, the zoom level, the size in pixels of the desired images, the desired map layer (i.e. parcel boundaries), styling options, and so on.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ARCXML version="1.1">
  <REQUEST>
    <GET_IMAGE>
      <PROPERTIES>
        <ENVELOPE minx="-73.985" miny="40.756" maxx="-73.972"
maxy="40.765" />
        <IMAGESIZE width="250" height="175"/>
      </PROPERTIES>
    </GET_IMAGE>
  </REQUEST>
</ARCXML>
```

Figure 2 shows a sample request to an ESRI ArcIMS or ArcMap server.

Figures 2 and 3 show a typical request and response for ESRI map services. These XML messages are transported over HTTP. Note that in Figure 2 the image is not directly returned in the response, but instead the service sends back a URL to the image. This allows the server to communicate in a non-blocking fashion with the client. This is convenient as map images are typically created on demand, and so it can take several seconds to actually generate the image.

```
<?xml version="1.0" encoding="UTF8"?>
<ARCXML version="1.1">
  <RESPONSE>
    <IMAGE>
      <ENVELOPE minx="-180" miny="-144" maxx="180" maxy="144" />
      <OUTPUT
url="http://mycomputer.domain.com/output/world_MYCOMPUTER2102209.png"
/>
    </IMAGE>
  </RESPONSE>
</ARCXML>
```

Figure 3 shows the response of the request in Figure 2.

In contrast to the XML-over-HTTP approach of ESRI, AutoDesk's MapGuide and the OGC's Web Map Server use HTTP GET style messages to construct requests. A sample AutoDesk request is shown in Figure 4. OGC Web Map Server requests are similar. The response images are included in the HTTP response. This approach has the obvious drawback that it blocks until the requested image can be constructed and returned.

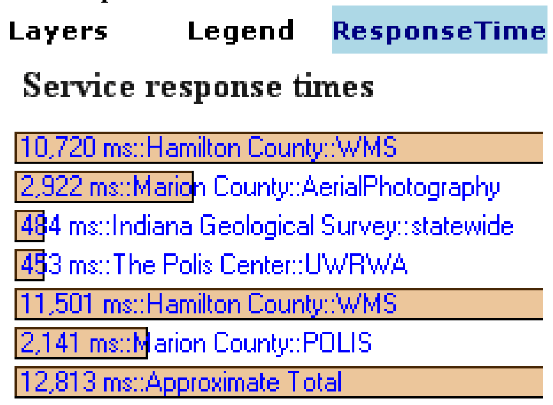
Request for AutoDesk MapGuide Server

```
http://litemap.co.hamilton.in.us:8080/liteview/servlet/MapGuideLiteView?VERSION=1.1.1&REQ
UEST=Gemap&LAYERS=COUNTY_PLAN.MWF\parcels&SRS=EPSG:4326&BBOX=86.0009765625
,40.06125658140474,85.99960327148438,40.062307630891&WIDTH=256&HEIGHT=256&FOR
MAT=image/png&BGCOLOR=0xFFFFFF&TRANSPARENT=TRUE&WMTVER=1.1.1&STYLES=
```

Figure 4 shows an example of a client request to an AutoDesk map service.

Abstracting these requests with adapters is not technically difficult. As we have reviewed, one must know the syntax needed to construct the commands, which may be serialized as XML or HTTP GET name/value pairs. It is possible to construct a federating map server solely from these adapters, which dynamically make requests to the backend map servers every time a user requests a new map. This non-caching approach, however, has very limited interactivity.

To illustrate this, we present sample response times for various Indiana county map servers are shown in Figure 5. These values are intended only to show orders of magnitude. As can be seen, the response time is measured in seconds. For a federated map server delivering maps from several different counties, the total response time will be limited by the slowest server. This problem is compounded by the great variability of server hardware, software, and network connectivity of the various county servers.



Service response times

10,720 ms::Hamilton County::WMS
2,922 ms::Marion County::AerialPhotography
484 ms::Indiana Geological Survey::statewide
453 ms::The Polis Center::UWRWA
11,501 ms::Hamilton County::WMS
2,141 ms::Marion County::POLIS
12,813 ms::Approximate Total

Figure 5 lists sample response times for various county map servers.

Obviously these times are not acceptable for AJAX-style clients. The solution is of course to exchange computing time for disk space. By pre-requesting and caching the images as tiles, we can greatly decrease the time required to deliver the map images.

Requesting, Storing, and Indexing Tiles: Although any reasonable tiling strategy can be used, we have adopted Google's approach in order to enable integration with the Google Map API on the client side, as discussed below. In constructing our tile requests, we have relied on Mapki [11], a Google Map community Wiki that provides information for developers.

In summary, our basic procedure is first to discover the size of tiles used by Google at a particular zoom level and next to harvest map data from state and county map servers using this same tile size as a bounding box in the request. These images are stored on the cache server's file system using a convenient naming convention.

The first step in this process is to pick a zoom level and bounding box. These are then used to determine the tiles used by Google to represent this map. As described at [12], Google Map tiles have a simple matrix notation. It is possible to discover the particular matrix element of any given point (say, the upper left and lower right corners of the map in Figure 6). From this, we may infer all the tile elements and the latitude/longitude values of their bounding boxes. Note this is repeated for all zoom levels.

The next step is to iterate through all the tiles at all zoom levels and request local map layers from the county servers. To do this, we must determine which county or counties are at least partially in the bounding box. This can be done by sending the bounding box coordinates to the Indiana Geological Survey's boundary service [13], which will return all the counties in this tile. We then iterate over these counties to request the desired map layer.

When a bounding box lies across county boundaries, we can still use the original bounding box. The resulting response from each county will be only a partially filled box. We then (in the cache server) combine the two partial tiles into a single tile, as shown in Figure 7. The original tiles are discarded to save space.

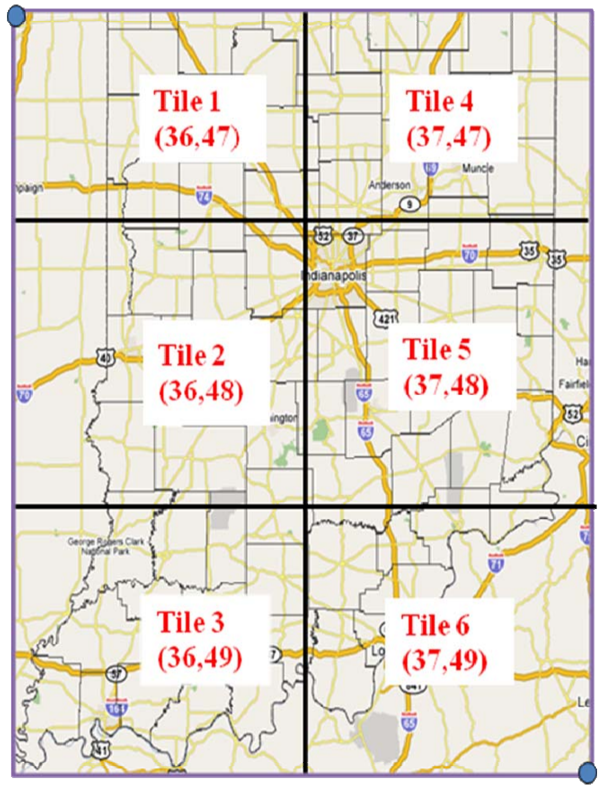


Figure 6 shows the Google tiles associated with the indicated bounding box. Note the Indiana county boundaries obtained from the Indiana Geological Survey's map survey have been overlaid on this map.

All tiles are stored in the Mercator project used by Google. This requires that we transform county imagery, since all Indiana county servers use the EPSG4326 projection by default.

Tile Storage Strategy: We cache all tiles using the same matrix notion as the corresponding Google map tiles and use this to give each tile a unique file name. These are stored in hierarchical file system directories using the zoom level and layer as the directory name. When a client sends a request to get the tile from the server, the server can directly retrieve the exact tile without the requirement of querying any database, which would decrease the response time of the server.

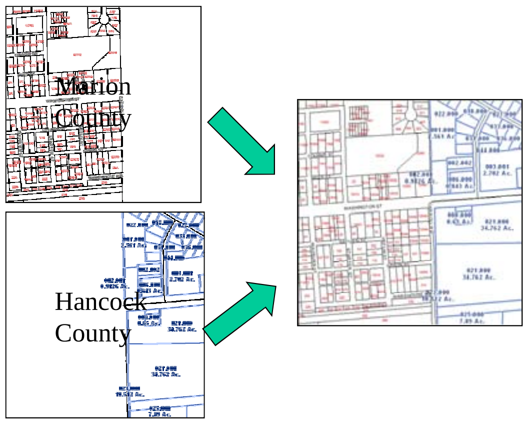


Figure 7 illustrates how partial tiles are combined. This situation occurs when the tile's bounding box includes partial information from two or more county servers. The partial images are combined and stored as a single tile, and the original tiles are discarded.

We now illustrate this process. A client may request a parcel boundary tile image from our server by using an HTTP GET request like http://.../CacheClient/servlet/WMSConnection?layer=parcel&x=36&y=40&z=0. This request directly maps to the tile location of /CacheTileDir/zoom0/parcel/tile36.40.png on the server's file system. The server can then return this image in its response. This HTTP GET service interface approach was chosen specifically to support Google Map clients, as discussed below. We can also provide other programming interfaces, such as WSDL, to support other clients.

This approach is sufficient for our current implementation, but we must address scaling issues to store more data. Since the number of tile file grows exponentially along with the increase of map magnification (zoom) level, we will eventually experience operating system limitations for numbers of files in a single directory. A more efficient mechanism for directory organization needs to be applied. We are currently investigating solutions. A related problem is storing tiles across multiple independent file systems. Modern storage area networks can store data in the hundreds of terabytes [18], but simpler service-based approaches may also be a low-cost alternative. This would imply a hierarchy of cache servers in Figure 1.

Tile Storage Requirements: Currently in our implementation, for storing 15 Indiana counties at 13 zoom levels for 13 map layers, there are more than 3,591,013 tiles for each map layer. The tiles are 4KB in size, except aerial photography tiles, which are 25KB. Our current storage requirement is thus [3,591,013* (12*4KB +25KB)], or approximately 250 GB.

Indiana has 92 counties. Assuming we can obtain access to these additional counties' data, we estimate the storage for the whole state as follows. Each tile at one zoom level represents the same area as four tiles at the next lower level. In the zoom level 0, there are 4 tiles to cover the whole state. In the zoom level 12, there are 4*(4^12) tiles. So, it requires 67108664*13 tiles to cache the whole states for 13 layers. Caching the whole state for 13 layers would require over 4.5 TB disk space. Extending this approach to the entire country would require hundreds of terabytes.

Creating Overlays: Maps are generally constructed as overlays: one or more partially transparent top layers can be placed upon a base map. We do this with standard Java Advanced Imaging libraries [14]. Note that the layer overlays are actually composed this way on the client's request. For example (in Figure 8), the parcel layers (top left) and the base map (bottom left) are combined into a single image. The imaging libraries' performance is sufficient to layer these on demand.

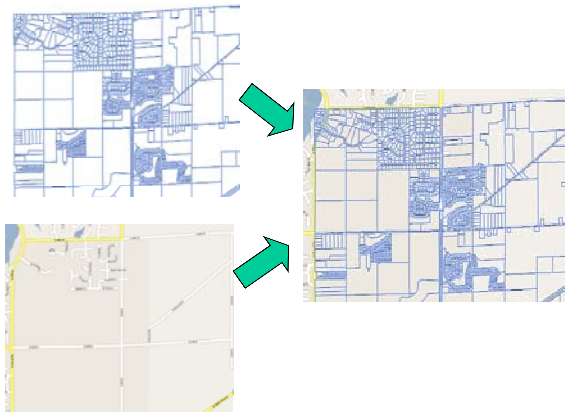


Figure 8 show how to tiles are overlaid. See text for a description.

Building Clients

In our current implementation, our caching and tiling service can be accessed via HTTP GET-style requests, allowing us to integrate with version 2 of the Google Map API. The general details are overviewed at [15]. The advantage of the Google API is, of course, that it allows us to build highly interactive browser clients.

A portion of a sample client is shown in Listing 1. Note that *GCopyrightCollection*, *GTileLayer*, *CustomGetTileUrl*, and *GTileLayer* are defined in the Google API. Different layer families in our service begin with the URL *http://.../GoogleCacheClient/servlet/WMSConnection*. The example shows how to retrieve orth-photography image tiles, parcel boundaries, and parcel IDs.

In addition to the map requests, we have also built adapters for interacting with county data servers that allow us to obtain (for example) parcel information across county boundaries. The approach is very similar to the map service adapters discussed above.

Figure 9 shows a sample client that can be constructed using client code based on Listing 1. The display shows the boundary between Marion and Hancock counties (eastern Indianapolis). The ortho-photography tiles in the image (originally harvested from Ref [17]) are obtained from our server (that is, these are not Google Maps' "satellite" images). The red and blue numbers are parcel IDs taken from local county map servers. We have not attempted to unify the styling across the two map servers. The left-hand side bar displays results from a parcel ID query to the Marion county feature data service.

Building an Event-Based Collaborative System

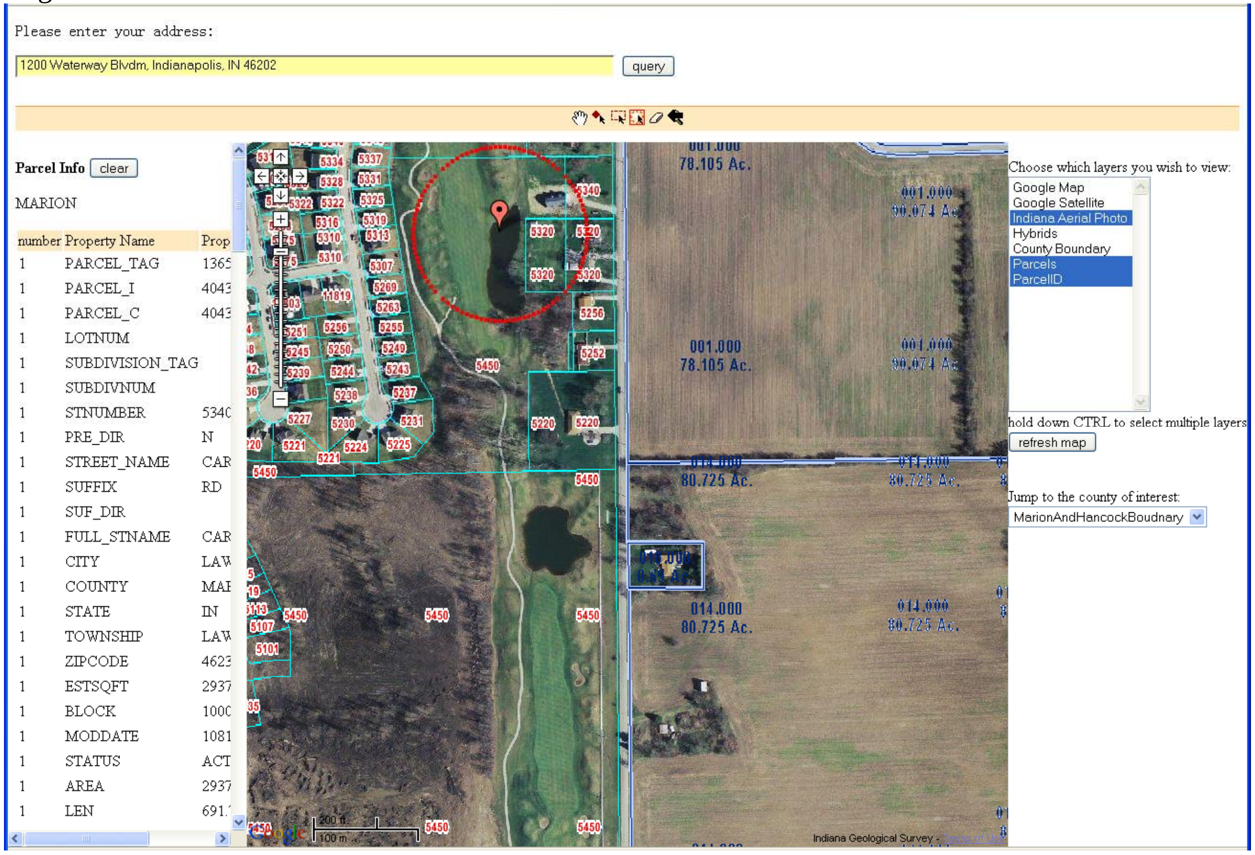
We have extended our clients and services to build a collaborative system using shared events. We have based our prototype implementation on Flex libraries from Adobe [16]. Our basic system allows two or more browsers to synchronize their displays using shared events: if the controlling user zooms or pans the display, the other participants in the session will have their browser displays automatically updated.

```
  // ===Create
GCopyrightCollection
  var copycol = new
GCopyrightCollection("");

  // ===Create tile layers
  var indiana_orthos= new
GTileLayer(copycol,6,19);

  indiana_orthos.myBaseURL
  ='http://.../GoogleCacheClient/
servlet/WMSConnection?layer=0';

  indiana_orthos.getTileUrl
       =CustomGetTileUrl;

  // ====== County Parcels
(Spring 2006) ======
  var parcels= new
GTileLayer(copycol,16,19);

  parcels.myBaseURL='http://.../G
oogleCacheClient/servlet/WMSConnec
tion?layer=9';

  parcels.getTileUrl=CustomGetTil
eUrl;

  // ====== County ParcelID
(Spring 2006) ======
  var parcelID= new
GTileLayer(copycol,16,18);

  parcelID.myBaseURL='http://.../
GoogleCacheClient/servlet/WMSConne
ction?layer=10';

  parcelID.getTileUrl=CustomGetTi
leUrl;
```

Listing 1 provides an example of how to call our map cache server through the Google Map (version 2) API using JavaScript.

To enable "co-browsing" and synchronous collaboration of multiple clients, the tile server uses a Flex data service, which defines a destination and provides a channel for clients to subscribe. The clients who are subscribes to the same channel can publish messages to all others synchronously. The communication between client and server in Flex uses Adobe's ActionScipt.

In our prototype, when a client subscribing to the collaboration channel takes an action such as pan or zoom, these actions will lead to an event call in JavaScript that can directly call back to the ActionScript function, which pushes the event message back to the channel registered in Flex data service in the server side. Then the channel will broadcast the event message to all its subscribed clients. The client browsers that receive the message will then run the events without user intervention. We have used this approach to make collaborative versions of the interface shown in Figure 9. We can also add shared tools, such as whiteboard markups and annotations by simply using libraries provided by Flex.


Please enter your address:

1200 Waterway Blvdm, Indianapolis, IN 46202

Parcel Info

MARION

number	Property Name	Prop
1	PARCEL_TAG	1365
1	PARCEL_I	4043
1	PARCEL_C	4043
1	LOTNUM	
1	SUBDIVISION_TAG	
1	SUBDIVNUM	
1	STNUMBER	5340
1	PRE_DIR	N
1	STREET_NAME	CAR
1	SUFFIX	RD
1	SUF_DIR	
1	FULL_STNAME	CAR
1	CITY	LAW
1	COUNTY	MAR
1	STATE	IN
1	TOWNSHIP	LAW
1	ZIPCODE	4623
1	ESTSQFT	2937
1	BLOCK	1000
1	MODDATE	1081
1	STATUS	ACT
1	AREA	2937
1	LEN	691.

Choose which layers you wish to view:

Google Map
Google Satellite
Indiana Aerial Photo
Hybrids
County Boundary
Parcels
ParcelID

hold down CTRL to select multiple layers

Jump to the county of interest:

MarionAndHancockBoudnary

Figure 9 displays a screen shot of a Google map client to our cache map server. This image is discussed in detail in the text.

Although the Flex model gives us an easy way to send message between clients, we must still address some classic issues in collaborative systems.

- We must be able to save and replay stored events. This will allow late-joining and rejoining participants to synchronize themselves with the system's global state.
- In our prototype implementation, anyone joining the session can control the map. It is thus relatively easy to create unsynchronized states in the participants (e.g., User A attempts to pan before receiving a "zoom" event generate by User B). There are numerous event throttling strategies that can prevent these situations. Typically, only a single user's events are published to the system at any time, and the states of the passive participants can be forced to synchronize with the controller's state.
- Shared whiteboard displays must take into account the scaling between different screen resolutions of the participants.

We are currently investigating the best approaches for solving these issues.

Conclusions an Future Work

We have described the architecture and implementation of our caching and tiling map server. We have followed basic Web 2.0 design principles by investing heavily in the server side development and providing a relatively simple client API (that is, we make our system compatible with the Google Map API, version 2). The simplicity of the client programming interface allows so-called mash-ups such as Figure 9 to be developed with relative ease. These mash-ups can be easily developed by users without detailed knowledge of the underlying service implementation.

Our caching and tiling service demonstrates how to federate data from multiple map providers into a single map service. We have implemented an HTTP GET-style interface to this service that is suitable for integration with Google Map clients, but other service interface to support additional clients can be developed.

We note generally that Web 2.0 is very similar to the "service oriented architecture" approach of both Grids [20] and enterprise systems. One of the major differences is the simple client development model: by supplying or supporting existing open and simple programming interfaces for clients, we allow end users to develop clients and integrate their own data, thus democratizing the web development process. This has been extremely successful for general Web development, and we hope to extend this approach to the Grid developer and user communities.

Several challenges are ahead. First, we must systematically investigate the performance options of our tile naming scheme to efficiently support more extensive map coverage. The map data for significant portions of the United States (for example) will be hundreds of terabytes in size. These may easily fit on modern storage area networks, which can provide seamless access to hundreds of terabytes of data. Indiana University's 535 TB Data Capacitor [18] provides an extreme example of this approach. There are other interesting approaches for more loosely coupled, federation-style architectural approaches that we would like to investigate as well.

We must also investigate how to integrate large statistical maps, especially choropleth maps for block groups or census tracts, and also how to have the ability to change the color schemes and number of breaks of clusters. For example, to provide the population density view for whole United States census and give the user the ability to change the rendering colors according to what a user selects while delivering acceptable performance is a challenging problem.

Finally, we are very interested in coupling our mapping system with scientific plotting, which will be driven by the geophysical requirements of the QuakeSim project [19]. One important example is the plotting of InSAR map images (both directly observed and synthetically produced by models) as map tile layers. This problem will require the coupling of our caching server to high performance computing resources, as the InSAR images can take several minutes to hours to create from both raw observational and simulation data.

Acknowledgments: Liu, Pierce, and Fox acknowledge support from NASA's Advanced Information Systems Technology program, part of the Earth-Sun Systems Division.

References

[1] Zhong-Ren Peng and Ming-Hsiang Tsou, Internet GIS: Distributed Geographical Information Services for the Internet and Wireless Networks. Wiley, 2003.

[2] [Gudgin, 2003] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. (2003), SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation 24 June 2003. Available from http://www.w3c.org/TR/soap12-part1/.

[3] [Christensen, 2001] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001), Web Service Description Language (WSDL) 1.1. W3C Note 15 March 2001.

[4] Roy T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", PhD thesis, UC Irvine, 2000. Available from http://roy.gbiv.com/pubs/dissertation/top.htm.

[5] The Open Geospatial Consortium Web Site: http://www.opengeospatial.org/.

[6] Paul Graham "Web 2.0" Available from http://www.paulgraham.com/web20.html.

[7] The ProgrammableWeb: Mashups and the Web as Platform, http://www.programmableweb.com/

[8] Jesse James Garrett, “Ajax: A New Approach to Web Applications.” Available from http://www.adaptivepath.com/publications/essays/archives/000385.php. See also http://en.wikipedia.org/wiki/AJAX.

[9] City of Indianapolis and Marion County Geographical Information Services: http://www.indygov.org/eGov/County/ISA/Services/GIS/home.htm

[10] Galip Aydin, Ahmet Sayar, Harshawardhan Gadgil, Mehmet S. Aktas, Geoffrey C. Fox, Sunghoon Ko, Hasan Bulut,and Marlon E. Pierce Building and Applying Geographical Information System Grids. Submitted to special issue on Geographical information Systems and Grids Concurrency and Computation: Practice and Experience.

[11] Google Mapki: www.mapki.com.

[12] Simple Analysis of Google Map and Satellite Tiles: http://www.dunck.us/collab/Simple_20Analysis_20of_20Google_20Map_20and_20Satellite_20Tiles.

[13] Indiana Geological Survey: http://igs.indiana.edu/.

[14] Java Advanced Imaging API Documentation: http://java.sun.com/products/java-media/jai/docs/index.html.

[15] “Add Your Own Custom Map”, Google Mapi: http://mapki.com/wiki/Add_Your_Own_Custom_Map.

[16] Adobe Flex2: http://www.adobe.com/products/flex/productinfo/overview/

[17] 2005 Indiana Orthophotography Project—gis.iu.edu: http://www.indiana.edu/~gisdata/05orthos.html.

[18] Stephen C. Simms, Matt Davy, Bret Hammond, Matt Link, Craig Stewart, Randall Bramley, Beth Plale, Dennis Gannon, Mu-Hyun Baik, Scott Teige, John Huffman, Rick McMullen, Doug Balog, Greg Pike: Bandwidth challenge - All in a day's work: advancing data-intensive research with the data capacitor. SC 2006: 244.

[19] Andrea Donnellan, John Rundle, Geoffrey Fox, Dennis McLeod, Lisa Grant, Terry Tullis, Marlon Pierce, Jay Parker, Greg Lyzenga, Robert Granat , Margaret Glasscoe QuakeSim and the Solid Earth Research Virtual Observatory. To be published in Special Issue of Pure and Applied Geophysics (PAGEOPH) for Beijing July 2004 ACES Meeting, Volume 163, Numbers 11-12 / December, 2006 DOI See also the QuakeSim project Web Site: www.quakesim.org.

[20] Ian T. Foster: Globus Toolkit Version 4: Software for Service-Oriented Systems. J. Comput. Sci. Technol. 21(4): 513-520 (2006)