Distributed Tool Services Via the World Wide Web

Stephen E. Dossick, Gail E. Kaiser, and Jack Jingshuang Yang
Columbia University
Department of Computer Science
Mail Code 0401
New York, NY 10027, USA
212-939-7184, sdossick@cs.columbia.edu
CUCS-042-96

October 1, 1996

Abstract

We present an architecture for a distributed tool service which operates over HTTP, the underlying protocol of the World Wide Web. This allows unmodified Web browsers to request tool executions from the server as well as making integration with existing systems easier. We describe Rivendell, a prototype implementation of the architecture described.

Keywords: tool service, world wide web, http, resource management, url

This paper is based on work sponsored in part by Advanced Research Project Agency under [D]ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197.

1 Introduction

Software tools are to the engineer as the hammer and chisel are to the carpenter: essential components in accomplishing the daily tasks required by his or her work. Imagine telling a carpenter that some of her favorite tools could only be used on a specific style of workbench, and other tools she uses every day can only be used on a different style of workbench, and a third set of tools are only available on a particular workbench located in a neighboring city. Engineers face an analogous problem. Many of the tools we use run only on particular computing platforms (e.g., Solaris vs. Microsoft Windows), and economic constraints may restrict licensing to a particular host (because either the hardware or software is very expensive, e.g., a Cray numerical analysis package), making it difficult to integrate the use of these tools with a user's daily tasks — preferably performed sitting at that user's office workstation.

All these hurdles get in the users' way. They must remember not only where they can run certain tools, but also the specifics on starting them (including environment variables needed, command line options, etc). A distributed tool server system can solve many of these problems by handling the knowledge about how and where to run a tool. All the user need do is request a tool from the tool server. The tool server is then responsible for remembering and deciding the specifics on running the tool.

In addition, many existing systems can benefit from the addition of (or integration of) a tool services component. The problem of managing the use of software resources needed by an organization (making tools as easy to run and as ubiquitous as possible) is one which can be solved by incorporating a tool server into the use of the system.

The World Wide Web provides a unique platform for distributed tool services. The HTTP protocol [7], the underlying protocol of the WWW, includes facilities which allow a client to request something from a server (the HTTP GET method). This request format can be generalized to allow a client to request a tool from a tool server "masquerading" as a Web server. The client, in this case, can be either a standard unmodified World Wide Web browser (including Netscape, Microsoft Internet Explorer, or NCSA Mosaic) or any application capable of making an HTTP GET request.

In this paper, we describe an architecture for a generic Web-based tool server and also our prototype implementation of such a system, dubbed "Rivendell". We start off with a motivating example, which helps us illustrate some overall requirements for a tool service. Next, we discuss some approaches to fulfilling these requirements. Then, we describe the workings of our Rivendell tool server and its integration with a substantial existing workflow system. Finally, we describe our planned future work in this area as well as draw some conclusions from what we've accomplished.

2 A Motivating Example

As a motivating example for our work, imagine a situation in which a company, XYZ Industries, has a rather expensive software system which they use for mathematical simulations running on a Cray machine. CAD work is done on SGI UNIX workstations running AutoCad. They've standardized on the UNIX version of Adobe's FrameMaker for all their internal documentation, and they use a mix of the UNIX and MS Windows versions of Adobe Photoshop for creating illustrations for the documentation. Microsoft Project is used for scheduling projects, including tasks like producing and updating Gantt charts of project milestones and deadlines. All the engineers at the company require access to all these tools in order to complete their work.

Unfortunately, the MIS department at XYZ Industries has let things get a little out of hand. Every engineer sits in front of either a UNIX workstation or an Windows-based PC. Problems occur because all the users require access to all the types of tools, no matter what kind of computer they happen to be sitting in front of. Training is a nightmare, because the Windows PC users do not want to learn UNIX commands, and the UNIX users don't want to be forced to find a Windows PC whenever they need a tool from that environment. Giving each engineer both a UNIX workstation and a PC on their desks is prohibitively expensive.

In addition, complicating things even further, is the need to share tools among multiple sites on the XYZ corporate WAN. Engineers in Palo Alto and New York need access to the Cray's numerical analysis tools, and the Cray happens to be in the Chicago office. Everyone needs access to AutoCad running on the powerful SGI Onyx ins the Palo Alto office. XYZ can afford the relatively inexpensive telecommunications links between the three cities, but they can't afford to duplicate the extremely expensive Cray and associated software in every city. Figure 1 provides an illustration of the fictional WAN/LAN setup.

From this example, we can identify a number of requirements for a distributed tool server system:

- Distribute tools across multiple platforms/operating systems
  We would like the users to be able to request tools that run on a range of computing platforms and operating systems from a set of independent but cooperating tool servers. Users should not have to remember, for example, that the Cray software is in Chicago and the associated network commands to access it. They should be able to ask any one of a set of "well known" tool servers for the tool, and the tool server should be responsible for either remembering the details on how to start the tool or remembering which other peer tool server to forward the request to. Each peer tool server is responsible for running tools on the machine it is running on.

  Although it may seem that an organization might need a large number of tool servers running, each one uses few system resources compared to have many users logging into the system constantly in order to run tools. In addition, if resources are scarce and the tool server is only needed infrequently, it can be started automatically upon a user request by a lightweight system daemon. This system daemon would only run when a user tried to connect to its particular TCP request port.

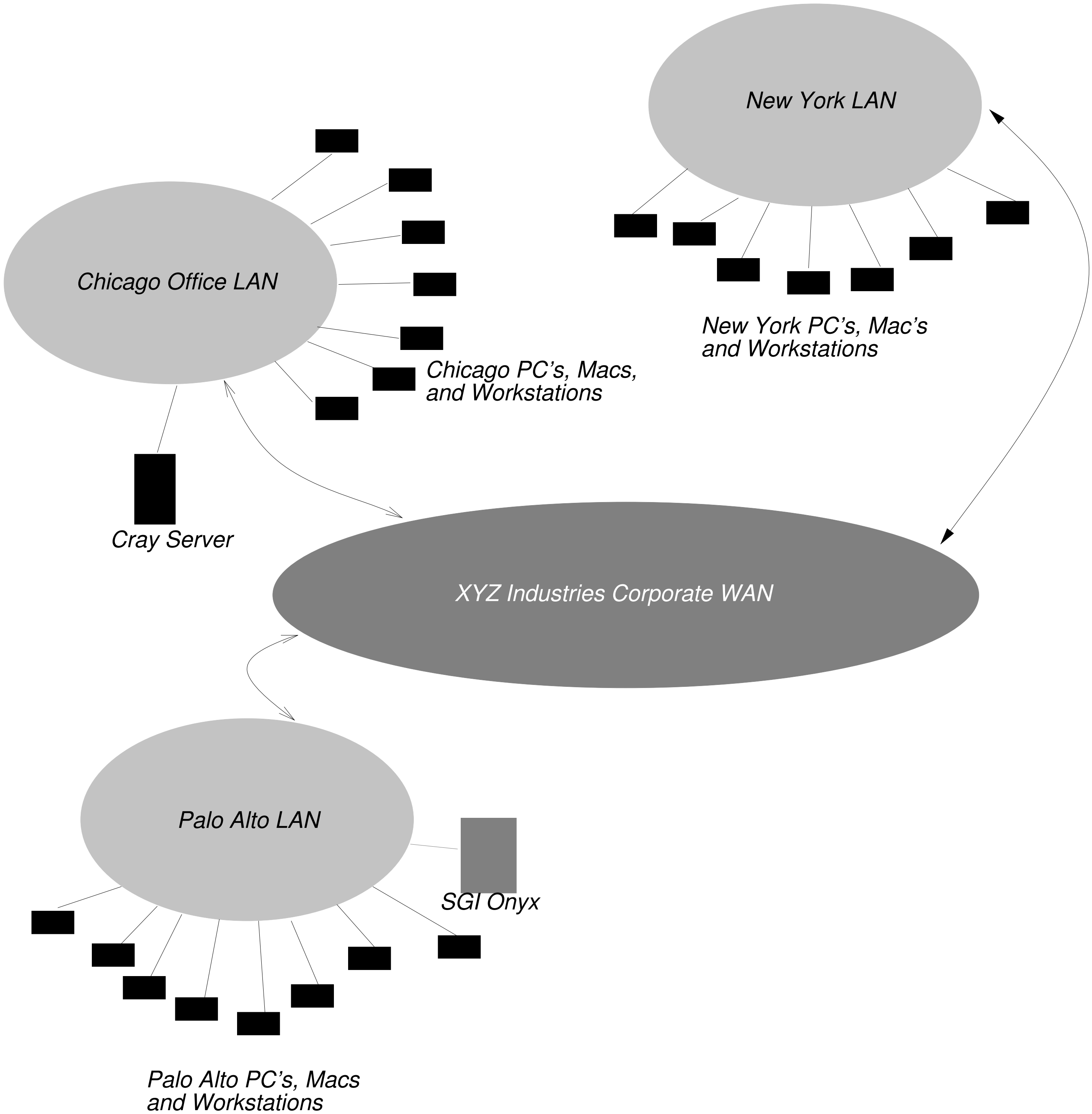

Figure 1: The XYZ corporate WAN connects the local networks of the New York, Chicago, and Palo Alto offices. Engineers at all locations require access to various tools, and everyone needs access to the corporate Cray system in Chicago, as well as the SGI Onyx in Palo Alto.

- Support for running tools on the user's own UNIX workstation, PC, or Mac

  The tool server should be able to decide (if provided with appropriate configuration information) that a given tool should be run locally on the client user's machine. This is used mainly as an optimization, as it's generally better (in terms of network bandwidth) to run a tool locally than to run a tool remotely and redirect GUI output using a remote display facility like X Windows.

- Wrap tools and sequences of tools

  It should be possible to wrap sequences of tools and have the wrapper appear to the user as a single activity that they are able to request from the tool server. This is useful in cases of tools whose input and/or output needs to be converted between file formats and those which need environment variables set before running. These wrappers can also handle startup configurations, including moving all relevant data files to a temporary directory, etc.

- No local software required on client computers

  It should be possible to request all tools from the tool server. The tool server may be able to exploit locally available software (software installed on the user's computer), but it must be able to make remote tools available via display redirection facilities like those of X Windows. This minimizes user and system administrator burden, since it is not strictly necessary for software to be installed on every computer.

- Easy to use on its own

  In order for a tool services component to be of use, the users must be able to use it. It should be possible for users to be able to connect to the tool server to receive a list of potential tools or activities they can run. This must be accomplished without the creation of a heavyweight user interface which would need to be ported to all computing platforms in use.

- Easy integration with existing systems

  We would like our tool services to be easily integrated with existing systems which might benefit from external tool integration. Such systems range from commercial CAD/CAM, workflow or medical care plan automation systems to simple menu-based systems which present a list of potential tasks to the user.

  In order to interface with existing systems, the tool server must have an API of some sort which would allow integration via program-to-program requests and responses. The existing system must be able to connect to the tool server and request tools on behalf of the system's users, without their intervention.

  An HTTP, CORBA [13], or RPC style interface would make integration easier. These are well known protocols, and all can be used as the underlying mechanism for fulfilling many of the above requirements.

3 Architecture

From these requirements, we can devise an architecture that combines remote tool servers running on a variety of computer systems along with a "personal tool services" component that runs on the user's office machine. From here on, we will use the phrase **tool server** to denote a remote tool server running on a server computer, and the phrase **personal tool server** to denote a tool server used to run tools on behalf of a user directly on his or her local UNIX workstation, PC, or Macintosh.

To satisfy the requirements that the tool server must be easy to integrate into existing systems as well as be easy to use as a stand-alone service, we model the tool server as an HTTP server, and use HTTP as the communication method not only between clients and servers, but between peer tool servers as well. HTTP is chosen because it is a simple TCP/IP based request-response protocol, making it easy to "graft" onto existing systems (possibly using toolkits like **libWWW** [6] from the World Wide Web Consortium or **ASHeS** [12], the Application Specific HTTP Services toolkit from our research group).

The HTTP specification provides methods for a client to request a document (GET) and send data (PUT) to a server. The documents are specified by URLs (Uniform Resource Locators [2]) which are really just strings that request something from the HTTP server. The object requested may be an document or an image (as when a Web browser sends a request to an HTTP website server), or it may be a request for some service. In our architecture, the URL is treated as a request for a tool.

In addition, HTTP is well suited for use both on a organizational Intranet (the organization's internal networks) as well as on the public Internet. Use of HTTP as the underlying protocol for a tool server does not require making the server available to the general Internet. Indeed, it is quite common to find many internal HTTP servers inside a given organization, each serving a specific purpose, while there may only be a few externally available servers (providing general information and services to the public).

The tool server can be implemented as a dedicated HTTP server or as a set of CGI [4] scripts (a standard method used to allow HTTP servers to run external programs when they receive requests for them). A dedicated server is probably preferable since it places a lower overall load on system resources (the main drawback to CGI is that it requires the HTTP server to start a new operating system process on *every* request, while a dedicated server could perform many tasks internally). Users connect to tool servers with standard Web browsers (Netscape, Microsoft Internet Explorer, NCSA Mosaic, etc.)

When a tool execution is desired, the user, or an existing system which needs to run a tool on behalf of a user, simply requests a URL from a tool server. Rather than returning a document, as a standard Web server would, the tool server starts the tool on behalf of the user (perhaps by employing a peer server, see below). Any return code information, or output in the case of a batch tool, can then be sent to the user's Web browser.

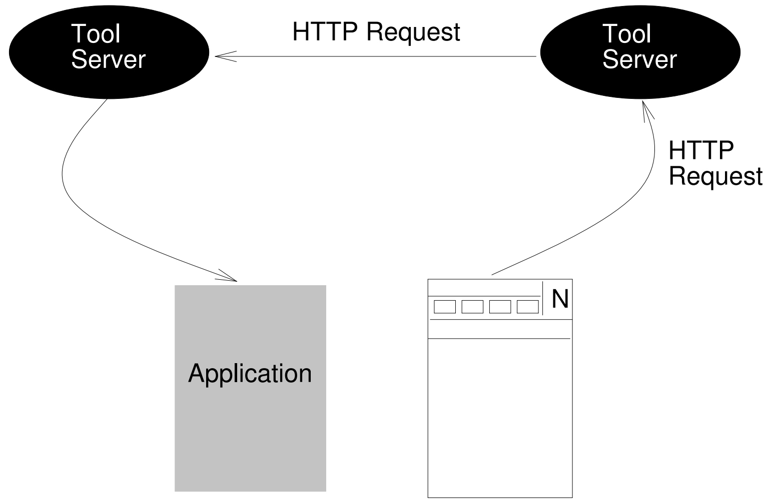

Figure 2: From the user's World Wide Web browser, the tool server is contacted. The tool server decides that the tool is best run by a peer tool server. The request is forwarded on to the peer tool server, which launches the application on behalf of the user. All pieces of this diagram may reside on different hosts connected via a LAN or WAN.

When a tool server is started (usually during bootup of a server machine), it may contact other running tool servers to exchange tool information. These other tool servers may be specified in a configuration file or may have been learned from a directory service. By storing information on what tools a remote tool server is capable of running , it is possible for one tool server to handle requests for tools it may not be able to run directly. When a request for the execution of a tool comes in from a client, the tool server can simply relay the request to a peer tool server which knows how to perform the execution. Figure 2 depicts a user requesting a tool from a given tool server. The tool server then forwards the request on to a peer tool server who is better able to run the application for the user, perhaps for licensing or performance reasons.

In addition, this makes possible a "clearinghouse" approach to tool management. In this scheme, multiple tool servers are set up to know how to run specific classes of tools on particular architectures, hosts, etc. A "clearinghouse" server is set up and is configured to contact the other servers and retrieve their tool definitions. Users can then connect to the clearinghouse server to request a tool which actually resides on any one of the other servers.

Modeling tool servers as World Wide Web servers makes it easy to run tools directly on the user's local office machine. We can employ the World Wide Web's standard mechanism for capability augmentation: a new MIME type and MIME helper application. Every response from an HTTP server is tagged with a type tag, known as a MIME [3] type. Figure 3 shows an example response and its associated MIME type field. Every World Wide Web browser can be configured to start what is known as a **Helper Application** when it receives data with a certain MIME type. If data is received which is destined for a helper application, the Web browser simply saves the data to a temporary file and starts the helper application, giving it the temporary file name as input.

```
HTTP/1.0 200 OK
Date: Fri, 04 Oct 1996 05:10:34 GMT
Server: Apache/1.1.1
Content-type: text/html
Content-length: 5149
Last-modified: Wed, 14 Aug 1996 15:13:00 GMT

<more data...>
```

Figure 3: Response headers from an HTTP transaction. The Content-type: field defines the MIME-type of the data being sent in response to the request. New MIME-types can be defined which force WWW browsers to start tools on behalf of the user.

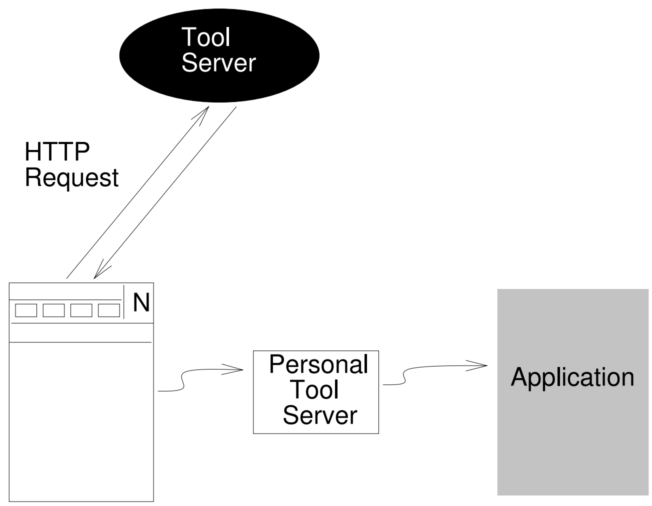

Figure 4: The tool server may decide that the tool is best run directly on the user's office machine. In this case, it outputs data with a specific MIME type, which causes the browser to start a helper application. This helper app then starts the tool for the user.

For example, suppose the tool server receives a request from a user who is using a PC. The user wishes to run Microsoft Excel, which happens to be installed locally on that user's computer. The user's system administrator has configured her World Wide Web browser so that receipt of data with the MIME type application/x-toolserv causes a MIME helper application to be started on the computer. This helper app is the Personal Tool Server, and it is responsible for actually starting the tool on behalf of the user.

The data from the HTTP server (which is sent to the MIME helper app — in this case our Personal Tool Server, or PTS) tells the PTS which local application is being requested and the pathname to the executable program. (The pathname may point either to the real application or to a wrapper script or batch file which may do other processing before or after running the tool). The tool is started, and when the user is finished with it, the PTS closes down and exits. Figure 4 illustrates this.

For some classes of tools, there may be multiple individual tools (which run on specific platforms or hosts) which can accomplish the same task. For example, imagine a user, Jane, at our fictional XYZ Industries who wishes to edit a C source code file. If Jane is using a UNIX workstation, it is appropriate for the tool server to run the Emacs text editor for her. If Jane happens to be sitting in front of a Windows-based PC, it is probably a better choice to start a Windows tool. In both cases, Jane will accomplish the same task: editing a source file. The user requesting the tool may leave it up to the tool server to determine which tool is more appropriate to use in a given situation.

To handle these types of situations, our tool server models **Activities** and **Scenarios**. In the case of Jane from above, her **Activity** might be called "Edit". The possible **Scenarios** for Edit are running Emacs on a UNIX workstation and running WinEdit on an MS Windows machine. When a user requests a tool from the tool server (see below), they can specify a particular scenario name if they need to run a specific tool from the available choices.

The request for a tool, sent by the client, is a standard HTTP GET request for a URL of the following form:

GET /Activity_Name/Scenario_Name/Arch/Host/Parameter_1/.../Parameter_N

Activity_Name is the only parameter that MUST be specified in a tool request. This parameter tells the tool server which Activity the user is requesting.

Scenario_Name allows the user to choose a particular Scenario for running the tool. If they have no particular preference, and want the tool server to decide, "*" must be specified here.

Arch allows the user to specify that this Activity must run on a particular computer architecture and operating system (for instance "Sparc/Solaris" or "Intel/Solaris"). This can be used to differentiate among Activities that use the same parameters as input but generate different side effects when run. For example, a Compile activity may have two possibilities defined, one for the HP/UX operating system and one for the Solaris operating system. One set of source code may be used to generate object code on both platforms, and the **Arch** parameter allows this to be chosen. Again, if the user wishes for the tool server to decide which architecture is best, "*" may be specified.

Host allows the user to request that a particular Activity be run on a particular host. This allows the user to try to override the tool server's choice of which host is the best place to run a tool. "*" may be specified for the default action of letting the tool server decide on a host.

After the **Host** parameter, any other portions of the URL are considered parameters to the tool execution. They are passed on the command line when starting the tool.

Other additional or different fields may be used in other implementations of this general approach. For instance, it might be desirable to add a user preferences facility whereby a user can select his preferences for tools or platforms for particular activities. The following section describes our current implementation of the above concepts.

4 The Rivendell Tool Server

The preliminary, proof-of-concept realization of our vision for a distributed tool server component is known as **Rivendell** and follows the architecture outlined above. Rivendell acts as a standard HTTP 1.0 server, receiving GET requests of the form specified above to launch tools on behalf of users.

Rivendell is typically started at system boot time, although it would be possible to employ a lightweight system daemon which would start Rivendell only when a user request is received. During startup, a Rivendell tool server attempts to read its configuration files (which specify the Activities and Scenarios it knows how to run, as well as which peer tool servers it should contact) and then contact the peer tool servers (running on other machines on a LAN, across a corporate WAN, or even across the Internet). When a peer tool server is contacted, Rivendell attempts to download the list of activities and scenarios known by the foreign server. Once this has been accomplished, Rivendell performs an HTTP PUT to upload into the foreign server the list of activities and scenarios it has been configured to know about.

Figure 5 contains an example Rivendell configuration file. The configuration file format strongly resembles the configuration information from MTP [19], a previous tool execution effort in our lab, which allowed for (among other things) specifying particular hosts where tools had to run. The MTP effort also considered shared groupware tools, one of the extensions we hope to make to Rivendell during our future work on the system.

The first portion of the configuration file, the **Peer:** lines, tell this particular Rivendell instance who its Peer Tool Servers are. These tool servers will be contacted, and configuration information will be shared with them at startup time. This Rivendell instance has two peers, running on the machines "pearl.psl.cs.columbia.edu" and "marginal.psl.cs.columbia.edu", both on TCP port 7777.

The **PTS_MIME:** line tells Rivendell what MIME type to send to the user's Web browser in order to have the browser start the Personal Tool Server (for local tool execution). The browser must have been configured beforehand to start the Personal Tool Server upon receipt of this MIME type. This configuration is often handled for users by an organization's system administrators.

Following the MIME type definition are the definitions for the Activities and Scenarios known to this particular instance of Rivendell. Two Activities are defined, "Editor," and "Compiler," each of which has two scenarios. The Editor Activity defines a Scenario called "Emacs" on the "Sparc/Solaris" system architecture, as well as one called "WinEdit" on the "Intel/Windows" platform. The actual pathname to the executable

```
Peer: "pearl.psl.cs.columbia.edu:7777";
Peer: "marginal.psl.cs.columbia.edu:7777";

PTS_MIME: "application/x-toolserv";

Editor
 [ Name  : "Emacs";
   Arch  : "Sparc/Solaris";
   Host  : "*";
   Exec  : "/usr/local/gnu/bin/emacs";
   Graphical : T;
 ]
 [ Name  : "WinEdit";
   Arch  : "Intel/Windows";
   Host  : "*";
   Exec  : "\edit\winedit.exe";
   Graphical : T;
 ]

Compiler
 [ Name  : "GCC";
   Arch  : "Sparc/Solaris";
   Host  : "*";
   Exec  : "/usr/local/gnu/bin/gcc";
   Graphical : F;
 ]
 [ Name  : "SparcWorks";
   Arch  : "Sparc/Solaris";
   Host  : "pearl.psl.cs.columbia.edu";
   Exec  : "/opt/SUNWspro/bin/cc";
   Graphical : T;
 ]
```

Figure 5: Rivendell Configuration Sample

file is given in the "Exec" parameter in each Scenario. The "Graphical" parameter tells Rivendell if the tool is a GUI or batch tool.

Once the configuration file has been read, Rivendell contacts each Peer Tool Server defined. A specific protocol is followed. Rivendell issues the following HTTP GET request:

GET /RIVENDELL_ACTIVITIES

The remote tool server is expected to respond with its Activities and Scenarios. The following example illustrates this format, which is the same format as the Activity definitions from the configuration file read at startup.

```
Project
[ Name : "MS Project";
  Arch : "Intel/Windows";
  Host : "*";
  Exec : "\msoffice\project\project.exe";
  Graphical : T;
]
```

The response indicates that the contacted tool server has one local Activity defined, called "Project". Rivendell stores this information, and if a client requests the Project Activity, the request will be forwarded to this Peer Tool Server.

Once Rivendell has downloaded the Activity and Scenario information from the remote server, it performs an HTTP PUT of the following form:

```
 PUT
/RIVENDELL\_ACTIVITIES [local activity information]
```

Rivendell sends the Activity definitions from its configuration file on to the remote peer. Rivendell contacts each peer tool server in this fashion, transferring information. Once this has been completed, Rivendell is ready to receive requests from users.

A user request for a tool (as described above) is an HTTP GET request where the URL specifies the Activity the user wishes to run. Again using an example from our fictional XYZ Industries, imagine that Bob, an engineer, needs to run the Emacs text editor. Bob's tool server defines an activity called 'Editor' which includes an 'Emacs' scenario. The closest tool server to Bob, who is in New York, is running on TCP port 7777 on a machine called 'nyserv.xyz.com'. In this case, Bob points his Netscape browser at the following URL:
http://nyserv.xyz.com:7777/Editor/Emacs/*/*/

When the tool server running on 'nyserv' receives Bob's request, it first makes sure there is an Activity named 'Editor' and a Scenario named 'Emacs' defined. The Activity can be one from nyserv's configuration file, or one which a peer tool server has informed nyserv about.

Once nyserv has determined that the request is valid, it determines how to run the tool. First, it checks to see if the UNIX workstation Bob is sitting at is able to run Emacs for him (via a Personal Tool Server). This is done as an optimization — it is usually faster and more efficient (for network bandwidth) to run a tool locally than to run it remotely and use X11 or WinDD-style display redirection.

The tool server on nyserv has no special knowledge of Bob's workstation, so it cannot assume that Bob's workstation is suitable for running Emacs locally. Since the 'Editor' Activity is a locally defined one (i.e., it was not learned from a foreign tool server), nyserv moves on to the architecture and hostname requirements, which can be set in an Activity definition. Since these are both set to "*", this Activity is free of restrictions. Thus, nyserv decides it is the best place to run Emacs for Bob, and does so. X11 display redirection is used to make Emacs display to Bob's workstation (i.e., the DISPLAY environment variable is set to Bob's workstation before the tool is started).

If the 'Editor' Activity from the above example had been one that had been learned from a remote peer tool server, nyserv would have forwarded the request for the tool on to the remote tool server. Bob would not need to know that his request had been forwarded, it would be the responsibility of the tool server on nyserv to do this transparently.

Rivendell has also been integrated with the WinDD software from Tektronix [17]. WinDD allows a user sitting at a UNIX workstation to remotely execute tools on a Windows NT Server machine, handling all display redirection on behalf of the user. WinDD employs special software, running on the UNIX workstation, to contact the Windows NT server. When Rivendell runs a tool whose "arch" configuration parameter is set to "Intel/Windows", a WinDD client is started for the user, and WinDD is then instructed to start the required tool. Figure 6 is a screen shot of WinDD running an application on a Windows NT server and displaying onto a UNIX workstation. Similar software, notably NTrigue from Insignia Solutions [8] and WinFrame from Citrix [5] accomplishes the same function as WinDD and could be integrated as well.

5 Integration

We've integrated Rivendell with the OzWeb Information Management System, described in [11]. OzWeb is an extended Web server which provides workflow, transaction management, object oriented data modeling and concurrency controls on top of distributed documents. OzWeb utilizes an architecture of both specially modified HTTP servers as well as specially modified HTTP proxies in order to provide its services to unmodified Web browsers.

The actual integration of Rivendell with OzWeb required small changes to the OzWeb server. OzWeb already provided a tool execution component that received user requests for tools and then started them directly. There was no facility for requesting tools from another server, nor was there a facility for running tools on a particular host for licensing or other reasons (unless that host happened to be the host the OzWeb server was running on).

We modified the OzWeb tool execution functions to pass on all tool requests to a running Rivendell tool server. This was done by having OzWeb perform an HTTP GET to Rivendell each time a tool was needed. Thus we were able to bypass the internal tool execution code in OzWeb, and replace it with a simple function that simply connects to a Rivendell server. The connection to the Rivendell server is transparent to the user of the OzWeb system — the tool is simply brought up on behalf of the user. In the case of a tool executed by the Personal Tool Server, the only user intervention necessary is the one-time configuration of their Web browser to run the Personal Tool Server upon receipt of the relevant MIME type. If this is done beforehand by a system administrator, no user intervention is required.

Other small modifications were needed to complete the integration. We added a configuration file that tells the OzWeb server the location of (i.e. the hostname and TCP port number) of its associated Rivendell. If this configuration file is absent, or if the Rivendell instance cannot be contacted (in the case of a system

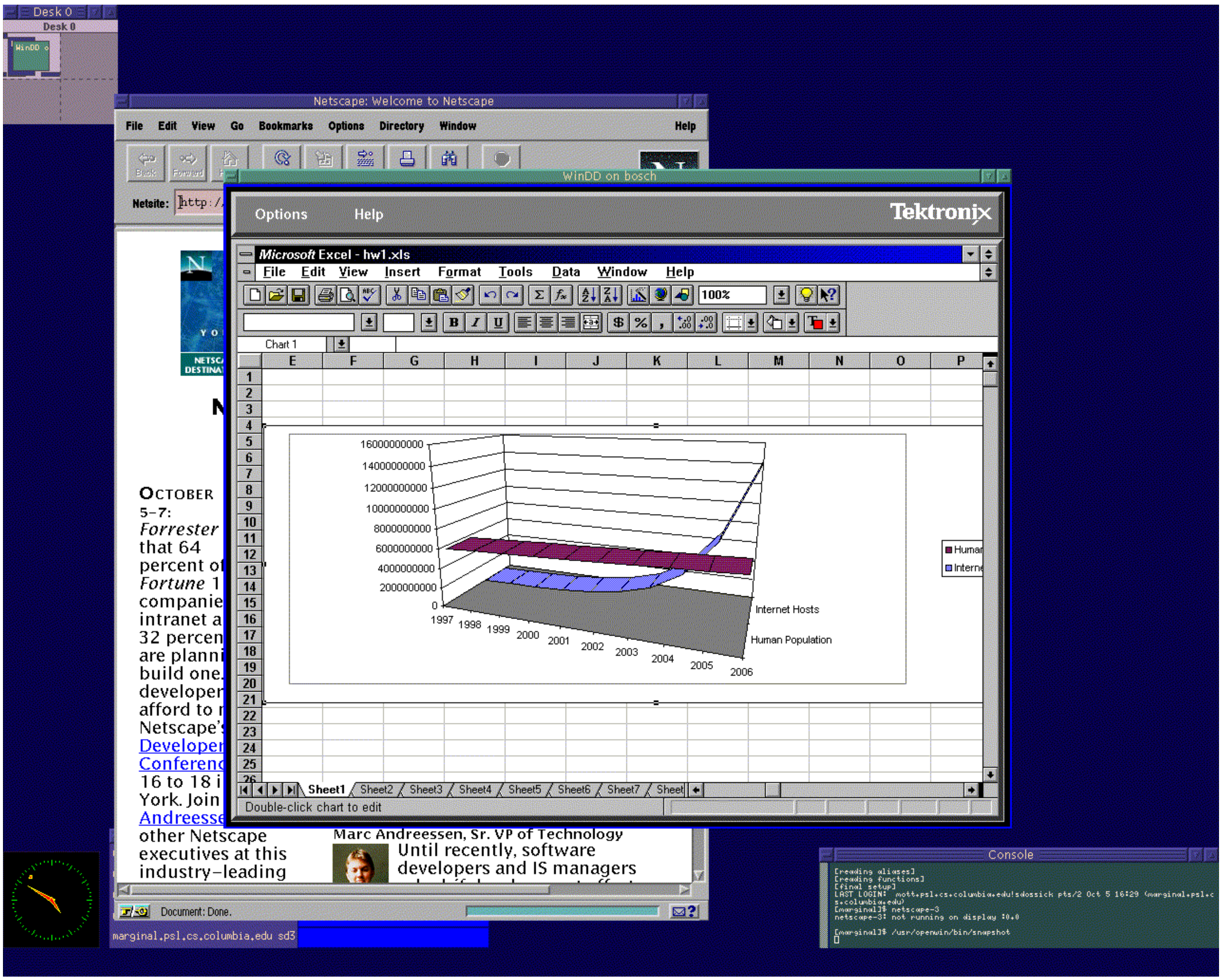

Figure 6: WinDD allows a user using a UNIX workstation to remotely execute tools on a Windows NT server. Rivendell has been integrated with WinDD in such a way that a Scenario with an "Arch" type of "Intel/Windows" causes Rivendell to start a WinDD client running the Windows tool on behalf of the user.

crash, etc), the OzWeb server defaults to its original method of running tools directly for the user.

6 Related Work

WebMake [1] uses a specialized set of Web documents, along with a modified World Wide Web browser to allow software development via HTTP. External tools can be invoked based on context transmitted from this specially modified client.

Field [15] uses a message bus to incorporate tools into an integrated software development architecture. This requires either source code availability (to modify the tool to understand Field messages) or an existing API through which the capability can be added. There is little support for integrating other types of tools.

Sun's ToolTalk [10] protocol allows applications from differing vendors to share information via a message bus type architecture. ToolTalk could conceivably be employed as another communication option via which users could request tools from a tool server. Integration with Sun development tools would be made easier, since they support the integration of third-party software via the ToolTalk protocol.

HP's SoftBench [18] framework, a tool-integration platform that provides an open, common set of communication and user interface services to all tools integrated with the SoftBench environment would also be a useful platform with which to integrate a tool services component. A tool server which connected to SoftBench would make it possible to more easily add non SoftBench-aware tools to the development system.

Matchmaker [9] is a distributed computing interface specification language which allows a programmer to define RPC interfaces between remote processes. A multi-targeted compiler then generates C, Pascal, Lisp, or Ada code which implements the interfaces defined.

Ockerbloom [14] proposes an alternative to MIME types, called Typed Object Model (TOM), that could conceivably be employed instead of a MIME extension to incorporate a Personal Tool Server into our architecture. Objects types can be registered in "type oracles", specialized servers that may communicate among themselves to uncover the definitions of types registered elsewhere. Web browsers that happen upon a type they do not understand can ask one of the type oracles how to convert it into a known supertype. In this way, the Web browser would not have to be set up to handle a new MIME type. They could simply query the type oracle, which could return information on how to run the tools.

7 Conclusions and Future Work

Rivendell is a "base prototype" of a distributed tool server system. Its chief advantage over previous tool execution efforts is that it can be used on its own (via the simple HTML interface for users) or easily integrated into an existing system.

There are a number of extensions we'd like to make to Rivendell. We'd like to integrate it with LDAP [20], the Lightweight Directory Access Protocol, an industry standard that allows for simple queries to be made to X.500 compliant directory servers. We envision using these directory servers to allow each Rivendell instance to find its Peer Tool Servers, and also to allow each Rivendell to publish the list of Activities and Scenarios it handles.

MTP, a previous tool execution effort in our lab, supports the execution of groupware and other multiuser tools as well as floor passing among multiple users of single-user tools using a utility such as XMove [16]. The latter can be used, for instance, to share an inherently single-user tool, such as the Emacs editor, among multiple users (taking turns). This functionality would be a useful addition to Rivendell.

We would also like to add the ability to send the data a tool may need along with the request for a tool. Currently, we assume that all cooperating tool servers have access to the same files, usually via NFS or some other transparent file sharing mechanism. It should be possible for the Rivendell to accept a set of files (packaged in much the same way a UNIX tar file packages multiple files together) that are needed to run a particular Activity. This is definitely needed by the Personal Tool Server component of the system, since the user's workstation or PC may not have access to the same set of files as the machine running Rivendell.

Further, we plan to enhance the process by which Rivendell chooses where (i.e., which machine) to run a given tool. Currently, Rivendell uses a simple algorithm to decide this: if the activity can be run on the user's local computer (via a Personal Tool Server), that is chosen because of possible network bandwidth constraints. Next, Rivendell will try to run a tool by itself, and if that is not possible it will use a Peer Tool Server.

It should be possible to make better decisions about the location of tool execution. Certainly optimizations can be made when choosing a Peer Tool Server to make sure we use one which is "closer" in terms of network topology and data residence. In making this decision, consideration might be paid to connection speeds between two interconnected networks, choosing the fastest Peer Tool Server when forwarding a request.

Finally, we would like to investigate using other transport mechanisms besides HTTP as the underlying protocol for our tool servers. CORBA seems particularly well suited for this task, however the lack of ubiquitous client software that speaks the CORBA protocol (a tool analogous to the Web browser that speaks HTTP) made us choose HTTP for our initial implementation. A number of Web tool vendors have expressed interest in integrating CORBA functionality into their systems, and this may make the use of

CORBA in Rivendell easier.

References

[1] Michael Baentsch, Georg Molter, and Peter Sturm. WebMake: Integrating distributed software development in a structure-enhanced web. In *3rd International World-Wide Web Conference*, Darmstadt, Germany, April 1995. Elsevier Science B.V. www.igd.fhg.de/www/www95/proceedings/papers/51/WebMake/WebMake.html.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url), December 1994. www.w3.org/hypertext/WWW/Addressing/rfc1738.txt.

[3] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of Internet message bodies, September 1993. Network Working Group Request for Comments: 1521.

[4] cgi@ncsa.uiuc.edu. The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[5] Citrix, Inc. Citrix WinFrame. http://www.citrix.com/.

[6] World-Wide-Web Consortium. W3C Reference Library. http://www.w3.org/pub/WWW/Library/.

[7] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, July 1996. HTTP Working Group INTERNET-DRAFT. http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v11-spec-06.txt.

[8] Insignia Solutions, Inc. What is NTrigue? http://www.insignia.com/marcom/DataSheets/NTrigue_DataSheet.html.

[9] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, New Orleans LA, January 1985.

[10] Astrid M. Julienne and Brian Holtz. *ToolTalk & Open Protocols: Inter-Application Communication*. Prentice Hall, Englewood Cliffs NJ, 1994.

[11] Gail E. Kaiser, Stephen E. Dossick, Wenyu Jiang, and Jingshuang Yang. An Architecture for WWW-based Hypercode Environments. Technical Report CUCS-037-96, Columbia University Department of Computer Science, August 1996.

[12] Programming Systems Laboratory. Application Specific Http Services (ASHeS). http://www.psl.cs.columbia.edu/software/ashes.html.

[13] Thomas J. Mowbray and Ron Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons, 1995.

[14] John Ockerbloom. Introducing structured data types into Internet-scale information systems, May 1994. PhD Thesis Proposal. www.cs.cmu.edu/afs/cs.cmu.edu/user/spok/www/proposal.html.

[15] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[16] E. Solomita, J. Kempf, and D. Duchamp. Xmove: A pseudoserver for X window movement. *The X Resource*, 1(11):143–170, July 1994.

[17] Tektronix, Inc. WinDD Network Display. http://www.tek.com/Network_Displays/Support/PAPERS/Web-5.doc.html.

[18] Gary Thunquest. Supporting task management & process automation in the softbench development environment. In Ian Thomas, editor, *7th International Software Process Workshop: Communication and Coordination in the Software Process*, pages 133–135, Yountville CA, October 1991. IEEE Computer Society Press.

[19] Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into process-centered environments. *Journal of Automated Software Engineering*, 3:309–345, 1996.

[20] W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol*, March 1995. Network Working Group Request For Comments: 1777, http://andrew2.andrew.cmu.edu/rfc/rfc1777.html.