1-1-1999

# THE NETWORK DESKTOP of THE PURDUE UNIVERSITY NETWORK COMPUTING HUBS

Nirav H. Kapadia
*Purdue University School of Electrical and Computer Engineering*

José A. B. Fortes
*Purdue University School of Electrical and Computer Engineering*

# THE NETWORK DESKTOP OF THE PURDUE UNIVERSITY NETWORK COMPUTING HUBS

NIRAV H. KAPADIA
JOSÉ A. B. FORTES

TR-ECE 99-1
JANUARY 1999


SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

# THE NETWORK DESKTOP

# OF

# THE PURDUE UNIVERSITY NETWORK COMPUTING HUBS

Nirav H. Kapadia

José A. B. Fortes

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

This work was partially funded by the National Science Foundation under grants MIPS-9500673, CDA-9617372, EEC-9700762, ECS-9809520, and EIA-9872516, and by a Purdue University academic reinvestment grant.

Table of Contents

List of Tables

List of Figures

## Abstract

A computing system that is universally accessible and is able to harness networked resources as and when necessary can be said to provide computing on demand. This report describes and evaluates the design of a network desktop infrastructure that provides access to distributed operating system services via standard world-wide web lsrowsers. Unmodified browsers can be used to access and use the desktop because it appears as a normal web server to them; the semantics associated with computing services are supported by treating URLs as locations in a dynamic, virtual, and side-effect based address space. The desktop interface is not hard-wired to the characteristics of any specific tool; the use of a programmable state machine in conjunction with a mechanism that embeds variables and objects within standard HTML allow the desktop to dynamically generate interfaces for tools and to emulate interactivity. Finally, users do not need physical accounts on the resources utilized by the desktop infrastructure; logical user accounts are created and managed with the help of filesystem and cache proxies, and the use of shadow accounts and software fault isolation techniques. The described infrastructure serves as the front end for the Purdue University Network Computing Hubs (PUNCH), a widely used demand-based network-computing system that allows users to access and run unmodified software tools via world-wide web browsers.

# 1. Introduction

## 1.1 Introduction

This report describes and evaluates the software architecture of the network desktop infrastructure for PUNCH. The desktop infrastructure serves two primary functions: 1) it provides users with distributed operating system services (e.g., file and process management) that can be accessed and used via standard world-wide web browsers, and 2) it user-transparently interfaces to a distributed computing system (SCION) capable of scheduling and executing software applications across wide-area networks on demand. The network desktop can be viewed as a user's window to web-based wide-area computing.

The network desktop infrastructure is unique in several ways, as described below. The desktop appears as a normal web server to browsers – the semantics associated with computing services are supported by treating URLs as locations in a dynamic, virtual? and side-effect based address space. Most tools can be made usable via the infrastructure with very little effort. The tools do not need to be modified, and access to source and/or object code is not required. For tools with text-based user-interfaces, the desktop utilizes administrat'or-supplied specification files to dynamically generate tool interfaces and to emulate interactivity. This is achieved by way of a programmable state machine that works in conjunction with a mechanism that embeds variables and objects within standard HTML. Tools with graphical user-interfaces are supported by leveraging display management technologies such as Broadway [8, 9, 10] and VNC [26]. Finally, the desktop creates and manages its own (logical) user accounts. As a result, users do not need physical account's on the resources utilized by the desktop infrastructure. The logical user accounts are managed with the help of filesystem and cache proxies, and the use of shadow accounts and software fault

isolation techniques, as discussed in Section 5.6.

The work on the desktop infrastructure was motivated by the fact that many of the systems and technologies that currently allow computing on the web target a single or a relatively small set of tools and/or work within controlled environments in which many of the issues that arise in production environments can be ignored. Solutions that target individual tools tend to be non-reusable in spite of the fact that they involve a significant amount of duplicated effort.[1] For example, a large number of systems (e.g., the Exploratorium [1], JSPICE [29], and others) are based on scripts that need to be modified in order to add any new application to the system. Other des'igns are more flexible. The MOL prototype [25], for example, employs static web interfaces that can be adapted for individual tools. The NetSolve [11], Ninf [27, 28], and RCS [3, 4] systems are based on structured designs that target numerical software libraries. However, static interfaces are not adequate for all tools, and structured approaches cannot be easily applied to general-purpose applications. Another problem is that the majority of these designs assume the availability of the source and/or object code for the applications – which effectively precludes the installation of most commercial tools. Solutions that address individual issues are generally reusable, but the tasks of adapting them for a production environment and integrating them into a complete computing infrastructure are non-trivial. For example, VNC [26] and WinFrame [12, 13] provide very flexible mechanisms for exporting graphical displays to remote consoles in a platform-independent manner – but, by themselves, the technologies do not help address other issues (e.g., access control and management) that arise in a wide-area distributed computing environment'.

The report is organized as follows. Chapter 2 provides a brief overview of PUNCH. Chapter **3** introduces the different components that make up the network desktop infrastructure. Chapters 4 and 5 describe and evaluate the individual components. Finally, Chapter 6 presents the conclusions of this work.

[1] Reusability, in this context, implies an ability to reuse the computing system with other applications without making any modifications to the system itself.

# 2. The Purdue University Network Computing Hubs

## 2.1 Introduction

PUNCH, the Purdue University Network Computing Hubs, is a demand-based network-computing system that allows users to access and run *unmodified* software tools via standard world-wide web browsers. Tools do not have to be written in any particular language, and access to source-code is not required. The PUNCH infrastructure is geographically distributed, hut this is transparent to users, who can run tools wherever they reside.

PUNCH can be logically divided into multiple discipline-specific "hubs". Currently, there are four hubs that contain tools for semiconductor technology, VLSI design, computer architecture, and parallel processing. These hubs contain over thirty tools from eight universities and four vendors, and serve more than 500 users from Purdue, across the US, and in Europe. PUNCH serves as the underlying distributed computing infrastructure for several collaborative efforts funded by the National Science Foundation: two projects involving five universities on the integration of design tools into new undergraduate and graduate curricula, and the Distributed Center for Advanced Electronics Simulations (DESCARTES). PUNCH is also the enabling infrastructure for a state-wide Purdue University network-computing system currently being deployed. Student's throughout Indiana will use this system to run tools on machines located at all Purdue campuses. The current system schedules most runs among approximately ten shared compute-servers distributed across Purdue University, the University of Illinois at Urbana-Champaign, the University of Maryland, and the University of Texas at Austin. During the past three years, PUNCH users have logged more than one million hits and have performed over seventy- thousand simulations. PUNCH can be accessed at "`http://www.ecn.purdue.edu/labs/punch/`";

courtesy accounts are available.

## 2.2 The PUNCH Infrastructure

PUNCH [21, 22] is made up of two parts: the front-end (network desktop) and SCION (see Figure 2.1). The front-end serves two primary functions: 1) it provides users with operating system services that can be accessed and used via standard world-wide web browsers, and 2) it user-transparently interfaces to one or more distributed computing systems that are capable of scheduling and executing software applications across wide-area networks on demand. SCION serves as PUNCH's user-transparent middleware. It consists of a collection of hierarchically distributed servers that cooperate to provide the following services: 1) user-transparent managernent of the run-time environment. 2) independent control over resource access and visibility pol cies at each management unit node, and 3) cost ancl performance driven scheduling of available resources. As implied earlier, hardware resources managed by SCION can include different types of platforms, and software resources can consist of arbitrary tools.[1] Resources can be located at any network-accessible site. and can be dyr.amically added or removed from the infrastructure.

From a user's perspective, PUNCH is a WWW-accessible collection of simulation tools and related information. It allows geographically dispersed tools to be indexed and cross-referenced, and makes them available to users world-wide. The infrastructure hides all details associated with the remote invocation of tools from its users. Functionally. PUNCH allows users to: 1) upload and manipulate input files, 2) run programs, and 3) view and download output – all via standard WWW browsers. It provides a context-sensitive help facility that assists users in the use of the tools and the infrastructure itself. Access to information and resources via PUNCH can be personalized and/or restricted according to user-specific needs and access-rights. Finally, the use of machine learning techniques allows PUNCH to *predict* run-times for tools. This information is primarily used for on-demand resource management (e.g., longer

[1] Tools with graphical user interfaces are supported with the help of display management technologies such as VNC [26].

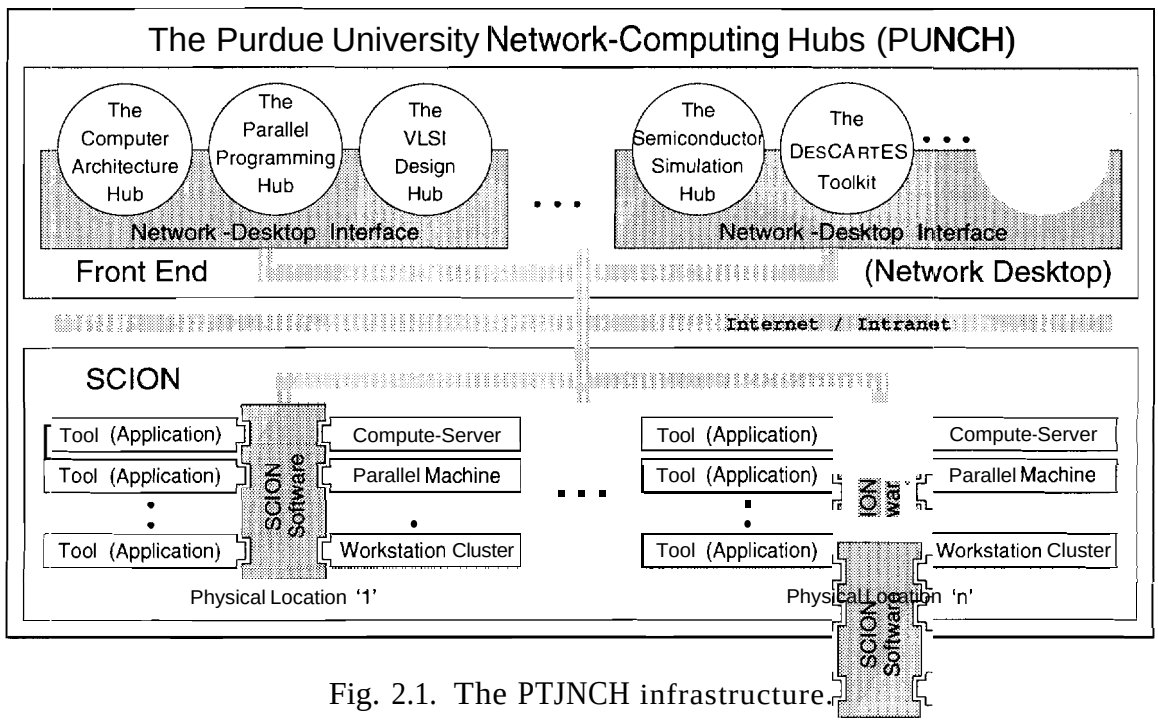


Fig. 2.1. The PTJNCH infrastructure.

runs are routed to faster machines). A detailed description of PUNCH is available in [20, 23, 21, 22].

Running a typical simulation on PUNCH is a three-step process. The first step involves the creation of the input file(s) required for the relevant simulation. In the second step, users define the input parameters (e.g., command-line arguments, etc.) for the program and start the simulation. Finally, after the simulation is complete, users can see, postprocess, and download the results via the web-based front-end. PUNCH runs programs in a "background" mode by default. This means that the user's browser window is freed up as soon as the run has been successfully initiated.

### 2.3 Summary

PTJNCH is the result of a concerted effort to harness the existing networking and computing infrastructures and the rapidly-advancing world-wide web technologies with the goal of building a *.functional* demand-based network-computing infrastructure. Over the years. we have found the system to be an extremely useful resource for students and collaborators, and a highly flexible testbed for network-computing research. The ideas and solutions presented in this report are based on (and validated by) our experiences in scaling PUNCH from a research project to a "live" system that is regularly used by several hundred students each semester. Results from user-surveys indicate that the system performs well under the highly peaked usage patterns (very high usage in the hours before homeworks and projects are due) characteristic of an academic environment.

# 3. The Network Desktop Architecture

## 3.1 Run-Time System

The network desktop infrastructure (see Figure 3.1) consists of: 1) a run-time system, 2) databases and templates (specification files) containing dynamic information that controls the behavior of the run-time system, 3) meta-information compilers that are used to generate and/or update the databases and templates, and 4) support hardware such as file servers. The run-time system is made up of an access manager, a set of run-time services, and a programmable state machine. The access manager handles network protocols, enforces access control, and routes transactions to appropriate service modules. The run-time services include commands for file manipulation, process management, directory services, and the like. The programmable state machine dynamically customizes the behavior of the virtual desktop interface for users and resources, according to criteria specified in the behavior template. The behavior template consists of a server template and one or more hub databases. As indicated in the figure, the server template specifies the server configuration and the run-time mappings for URLs and filesystems, among other things. Hub databases contain pointers to meta-information for tools on the corresponding "hubs" (tools can be logically grouped/separated across hubs according to desired criteria – by discipline, for example), along with associated configuration information (see Figure 3.1).

## 3.2 Tool Specification

A tool specification provides PUNCH with the meta-information required to make a given tool usable via the distributed computing framework. As shown in Figure 3.2, the specification describes different aspects of the tool such as general .information, virtual interface generation, and application *management.*[1] This will generally be writ-

[1]The specification file for a simple, batch-oriented tool can be written in about twenty minutes.

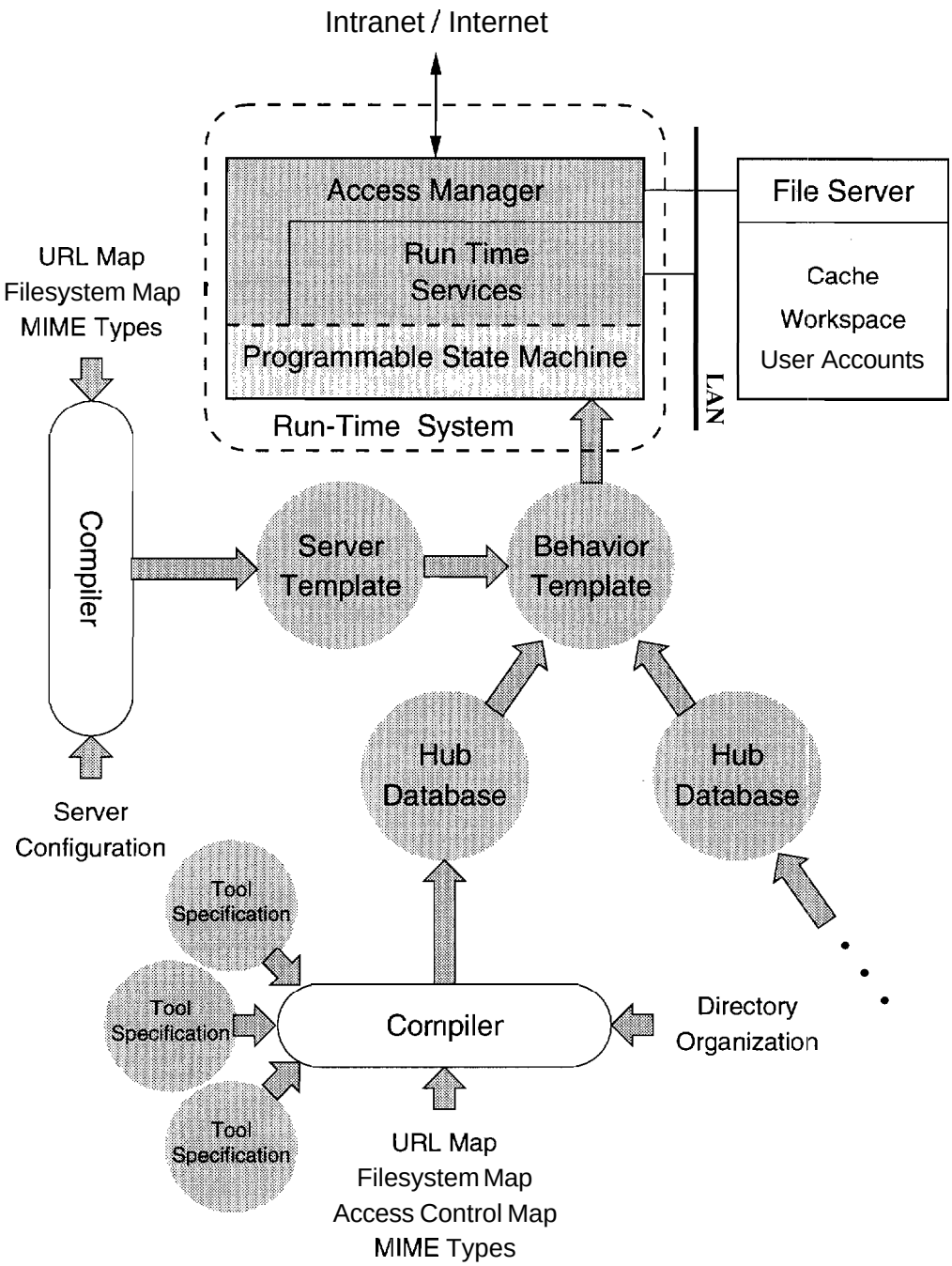


Fig. 3.1. A simplified functional view of the network desktop infrastructure.

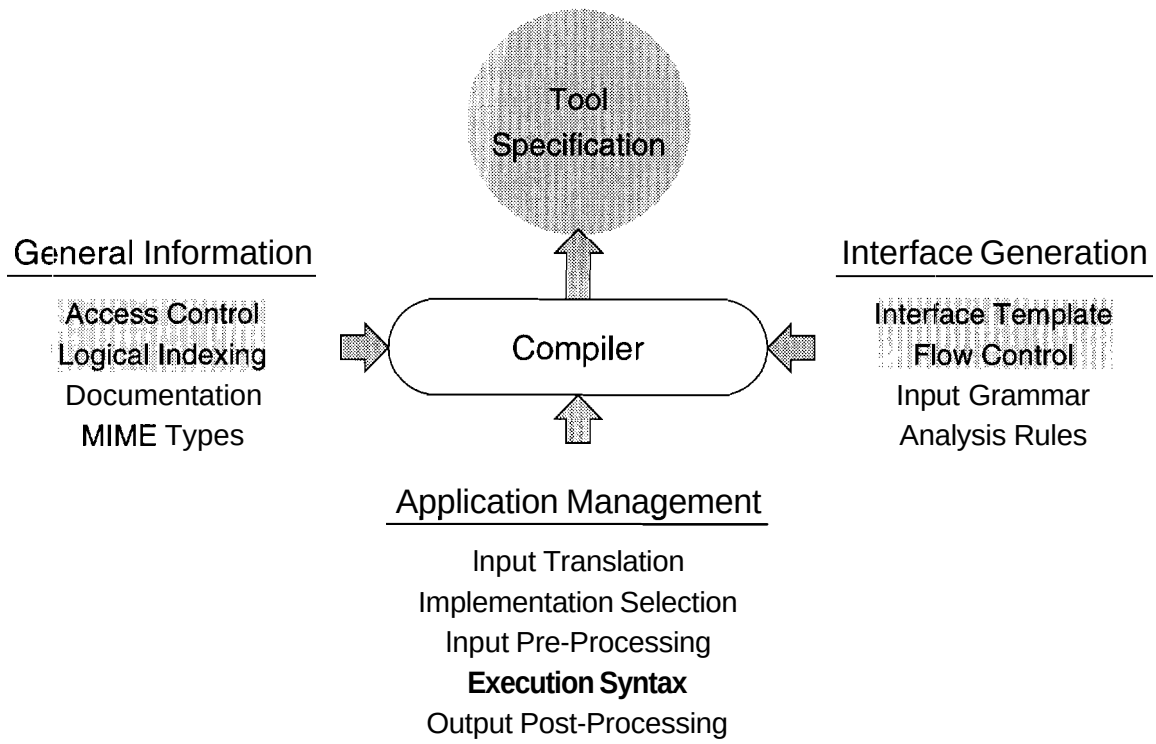


Fig. 3.2. Tool specification. The shaded items are required by all tools.

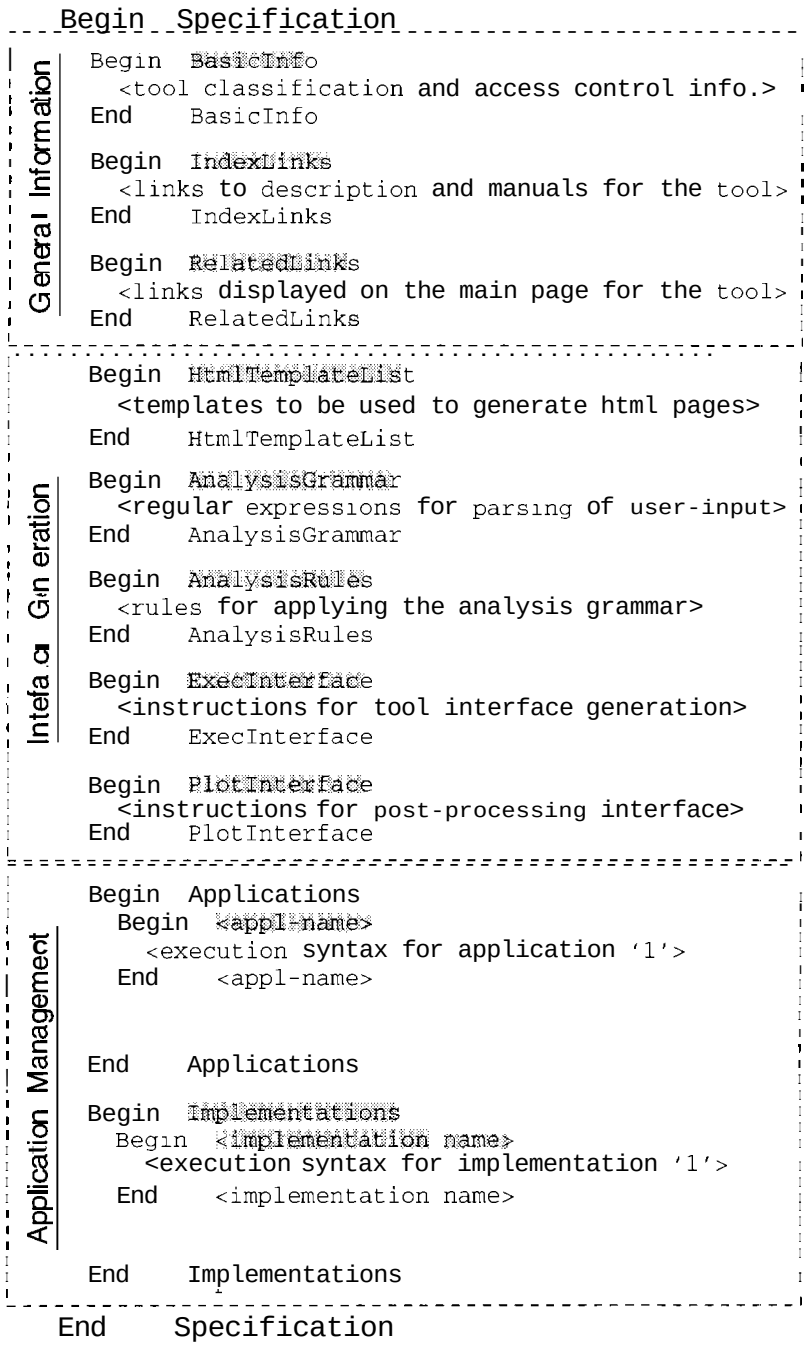


Fig. **3.3.** Skeletal structure of the tool specification file.

ten by an administrator or a tool installer when a tool is initially added to PUNCH. It can subsequently be reused at different sites with minor modifications. The structure of the tool specification file is shown in Figure **3.3.** With refererice to the figure, the modules for general information specify: 1) the logical classification of a tool for crossreferencing purposes: 2) access control policies, and 3) links to tool-specific documentation and related information. The interface generation modules describe the characteristics (structure and workflow) of the tool-specific virtual interfaces that are accessed by users via standard web browsers. Finally, the application management modules specify: 1) the manner in which different implementation:; or components of a, given application are to be selected, and 2) the execution syntax and protocol (e.g., command-line arguments, standard input, platform requirements, scheduling constraints, etc.) for each implementation or component. These modules implicitly specify a mapping between the virtual interface(s) and the tool's native interface(s).

### 3.3 Summary

'The next two chapters describe and evaluate the run-time system shown in Figure 3.1; aspects of the compiler that relate to the run-time system are explained when necessary. The software for the network desktop interface is divided into three components (see Figure 3.4), each of which is consists of several modules. The access management component authenticates requests and enforces access control policies. The virtual desktop component processes transactions that involve document serving?directory information, system and process status queries, file manipulation, and tool interface generation. Finally, the message-passing component provides the protocols that allows the desktop interface to communicate with SCION (the PUNCH back-end; see Figure 2.1) and other resource management systems (e.g., Condor).[2] The software is implemented in Perl [31]. Except when stated otherwise, the results presented in this report were obtained on a 300MHz Sun Ultra-4 running version 5.005-01 of the Perl interpreter.

[2]The message-passing component primarily implements established communication protocols, and is not discussed further.

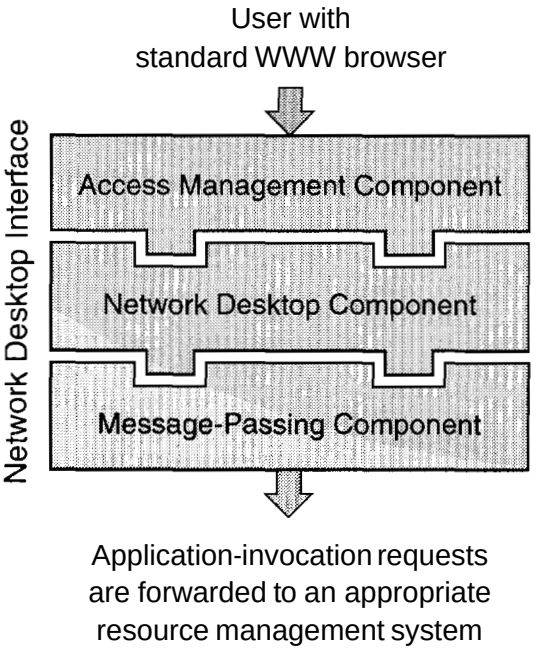


Fig. 3.4. Software architecture of the run-time system in the netvvork desktop.

# 4. Access Management

## 4.1 Introduction

'The access management component is responsible for: 1) translating external protocols into a standard format for internal use, 2) authenticating requests, **3)** enforcing access control policies, and 4) managing the information stored in PUNCH datisbases (see Figure 4.1). The component is made up of several modules, as shown in Figure 4.2. The following sections describe and evaluate each of these modules.

## 4.2 System-Interface Management Module

'The *system-interface management* module is responsible for translating external requests from different sources into a standard internal format. The internal format uses the syntax specified by the Multipurpose Internet Mail Extensions (MIME) [14, 17]. After translation, the data contained within a request are organized into key value pairs and stored in associative arrays. Multiple occurrences of identical identifiers (keys) are supported, and the value fields can contain arbitrary text and/or binary data.

Currently, three external protocols are supported. The HT*TP interface* manages browser-based requests. It accepts requests that conform to the HTTI' protocol [6, 15] and translates them into the internal MIME-based format for subsequent processing. In addition, the interface generates authentication challenges when necessary and encapsulates outgoing messages within HTML [5, 24].

'The *shell znterface* is designed to interact with very simple shell clients that can be ntegrated into traditional operating system command interpreters (e.g., UNIX shells). Functionally, the operating system parses user-supplied commands, decomposes them into their components (e.g., tool name, arguments, redirection, etc.), and

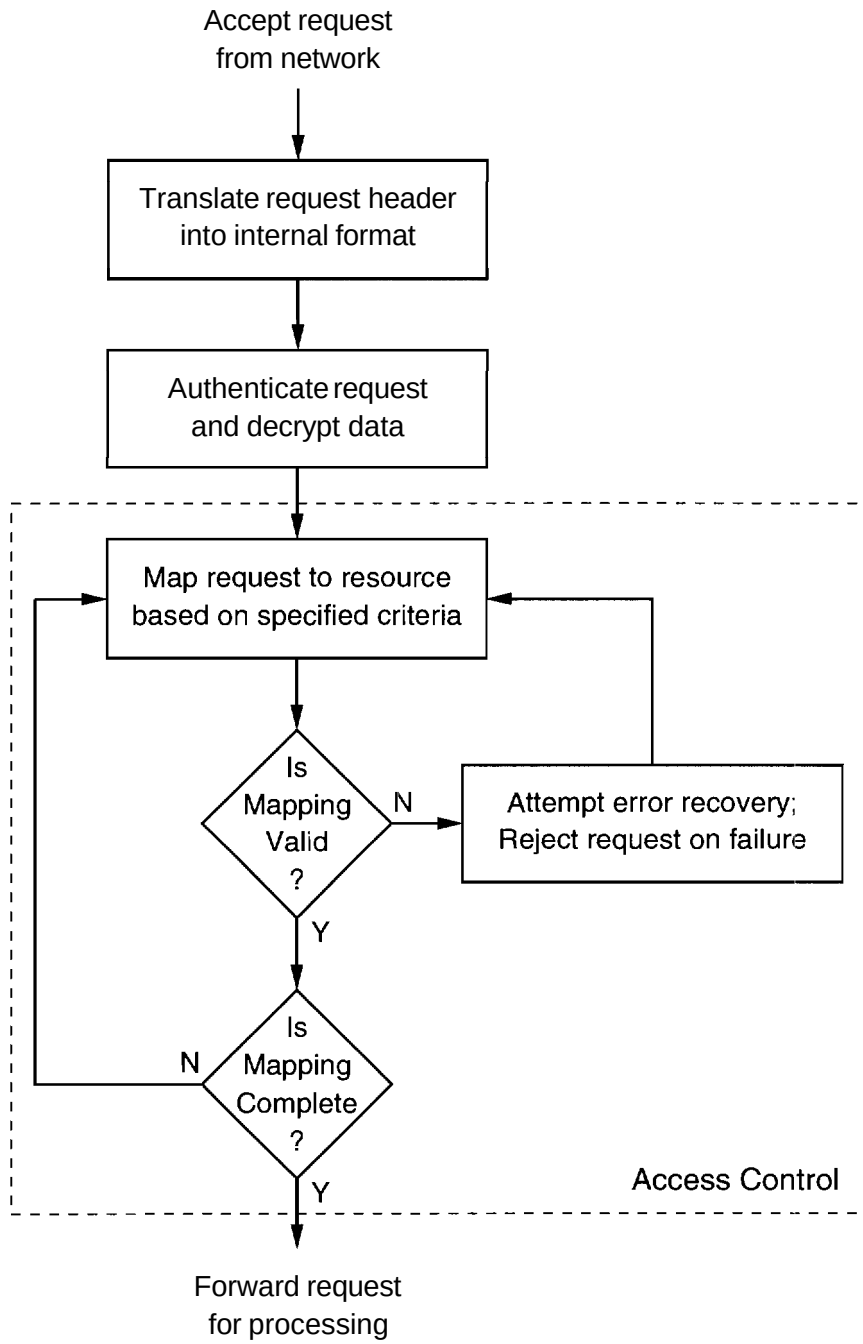


Fig. 4.1. Functional overview of the access management component.

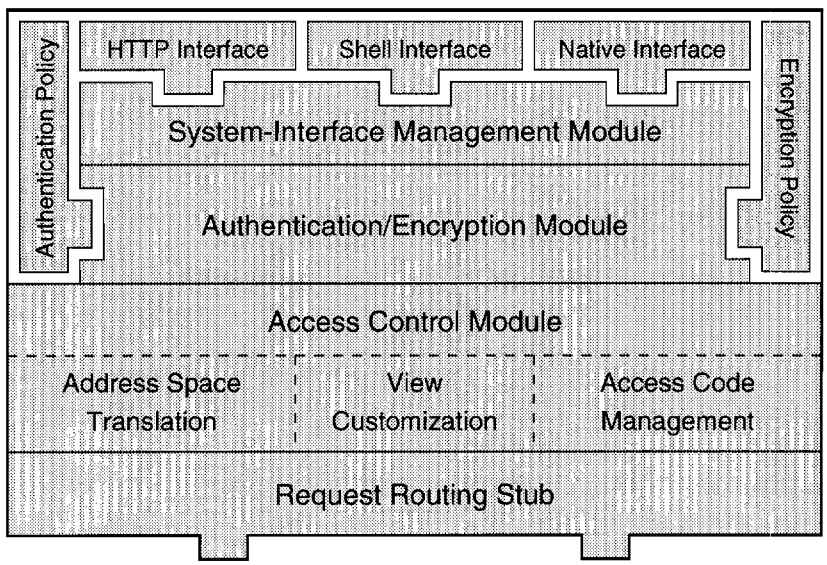


Fig. 4.2. Organization of the access management component.

and uses a shell client to forward selected commands to one of a set of nodes running the network desktop, along with appropriate authentication information. The role of the shell client can be illustrated by way of a simple example in which a user enters the following command on a UNIX platform: `fishld < demo.inp > demo.out`. On parsing the command, the operating system shell determines that: 1) `fishld` is the name of the tool to be executed, 2) standard input (`stdin`) will be read from the file `demo.in`, and 3) the standard output (`stdout`) will be redirected to the file `demo.out`. An unmodified shell would then look for an executable file that matches the name `fishld` within the directories listed in the user's path environment. The modified shell enhances the lookup procedure by allowing users to specify logical handles to network-computing systems in their path environment, in addition to directories. When the shell encounters a logical handle, it uses the client to query one of the network desktops within the network-computing system. If the network-computing system is aware of `fishld`, the request is forwarded to it for processing. The input for `fishld` will be read from `demo.inp` and its output will be redirected to `demo.out` in a user-transparent manner. If the network-computing system is not aware of `fishld`, the shell continues its search with the next directory or logical handle in the path environment.

Finally, the *native interface* accepts requests that conform to the internal MIME-based format. This interface is primarily used by PUNCH nodes to communicate with each other. The translation processes associated with the currently-supported pro ocols are straightforward, and do not contribute significantly to the overhead of processing a request.

### 4.3 Authentication/Encryption Module

The *authentication/encryption* module is responsible for authenticating transactions and decrypting/encrypting the data stream. The specific authentication and encryption policies to be used can be configured by a system administrator; the current system uses the basic challenge-response authentication mechanism described

in [6] and does not encrypt data. lclentification and authentication information is encapsulated within a Base64 encoding [7, 17] before transmitting it over the network.

Each transaction is authenticated by way of the user's identity, a password, and a session-specific dynamic identifier generated by the access control module described in Section 4.4. For a guest user, the identity field is associated with the IP address of his/her machine, and the password field is ignored. For a member user, the identity and password fields consist of his/her PUNCH login and password, respectively. The dynamic identifier consists of a string followed by a sequence of numbers that represent *access codes.* The string identifies the type of user, and takes on one of two values: 'Guest' or '`Member`'. The access codes allow efficient and scalable lookup of PUNCH resources, as described in Section 4.4.1. The dynamic identifier is automatically generated by the access control module at the PUNCH node that serves the request if it is incorrect or absent (see Section 4.4.1 for details). It is then cached by the client and presented to the PUNCH node with subsequent requests. The actual mechanism by which caching is achieved depends on the type of interface. For example, with the HTTP interface, the identifier is cached implicitly by making it a part of the URL. On the other hand, with the shell and native interfaces, it is cached explicitly on the client side.

Authenticating requests from guest users is straightforward. Each active guest user is represented by a persistent object that is stored in a database on the local (to the server) filesystem. The object is created when a user registers as a guest, and contains the IP address of the user's machine, identifying information such as the user's email address, and session-specific state information. When PUNCH receives a request from a registered guest user, it utilizes the dynamic identifier associated with this request to locate the corresponding object in the database. Once the object, is located, the request is authenticated by comparing the IP address of the machine from which the request was received to the IP address stored in the object.

Authenticating a request from a member user involves looking up the appropriate password-file entry and verifying the password supplied with the request. A

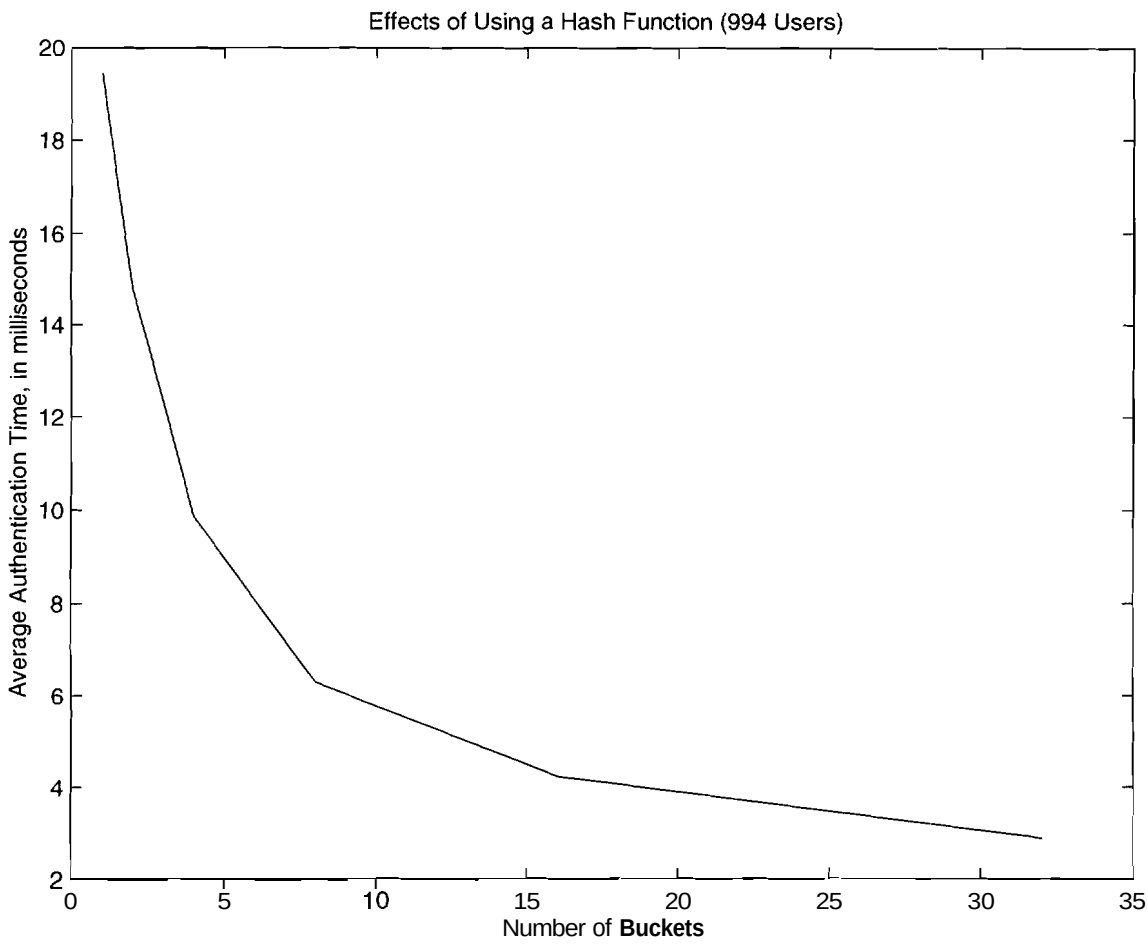


Fig. 4.3. Effects of using a hash function on authentication time.

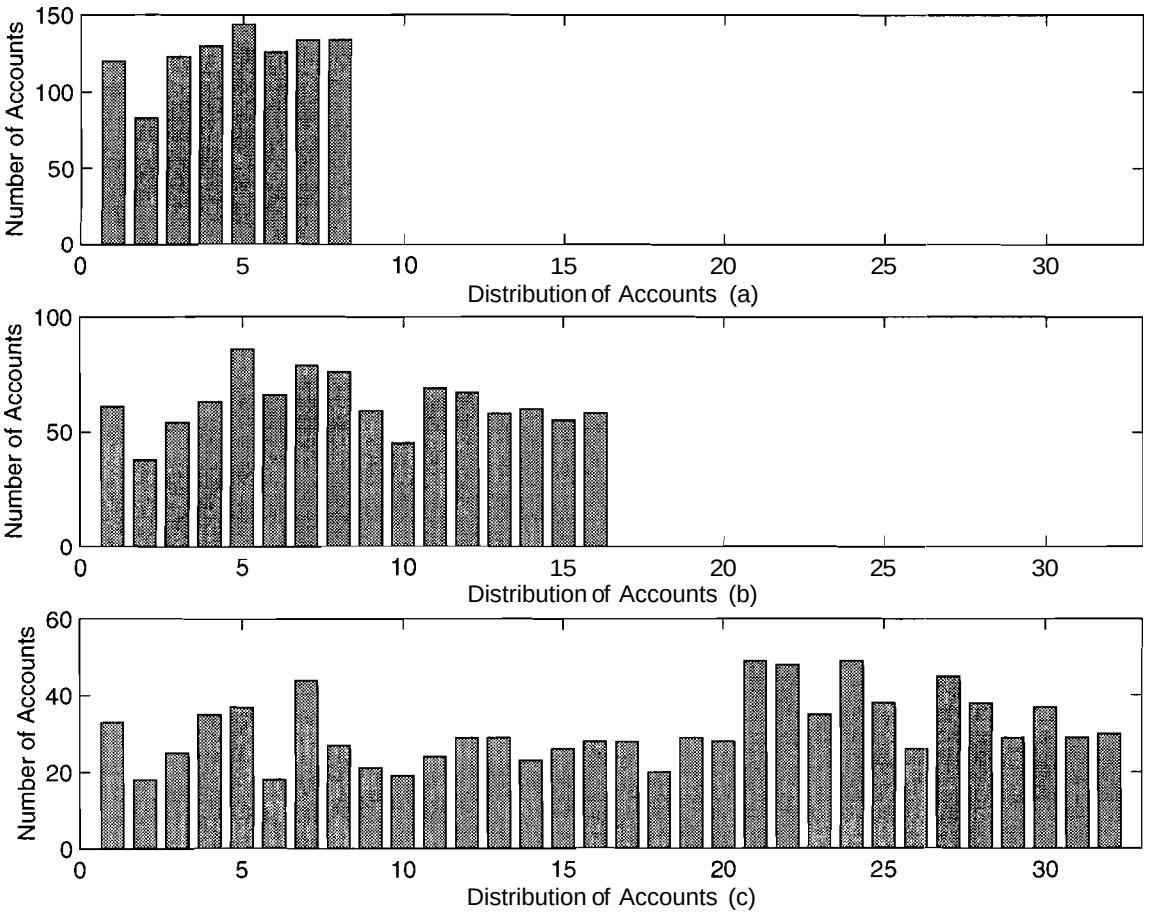


Fig. 4.4. Distribution of user accounts achieved by the hash function for: a,) eight, b) sixteen, and c) thirty-two buckets.

highly efficient authentication process is crucial to the scalability and performance of the overall system because the stateless nature of the PUNCH and HTTP protocols makes it necessary to individually authenticate each transaction. The efficiency issues are addressed as follows. For requests that contain correct access codes, authentication can be accomplished in O(1) time (see Section 4.4.1). When access codes are not available or are incorrect, the authentication procedure involves a search. The latency associated with the search is minimized by distributing the user-account entries across multiple password files. The distribution is accomplished by way of a very simple hash function that selects a file on the basis of the ASCII values of the initial characters of the user's login and the encoded or encrypted password. In particular, the password file identifier is obtained by evaluating the following expression: $(ord(login) + ord(password))\%32 + 1$. The function ord is a standard Perl5 function that returns the numeric ASCII value of the first character of its argument.

Figure 4.3 illustrates the effects of the hash function on the average time required to authenticate a request. The results are based on a real dataset consisting of 994 PUNCH user accounts. The plots in Figure 4.4 show that the (simple) hash function described above achieves an acceptably even distribution of accounts across available buckets for a real dataset.

At the end of the authentication process, each request is tagged with one or more logical *access* groups and forwarded to the access control module. The access groups for member users are retrieved from their password-file entries, assuming that the request was successfully authenticated. The access group for successfully authenticated requests from guest users is set to **'GUEST'.** Requests that could not; be successfully authenticated have an empty access group.

## 4.4 Access Control Module

The access *control* module dynamically maps user requests to the underlying resources based on the access group(s) associated with the request, the state information associated with the user, and the source of the request.

### 4.4.1 Access-Code Management

In the course of processing any given transaction, PUNCH modules access one or more databases in order to retrieve information associated with users and resources. Examples of such information include authentication and access control information for users, user-interface templates?learned resource usage characteristics, and portability information for tools, and availability and status information for hardware resources. It is extremely important to be able to access this information in a quick, efficient, and scalable manner. Information stored in database systems is retrieved by way of a query mechanism that typically involves a search. For large databases, the high latency associated with this search often translates to poor performance.

The PUNCH database system utilizes dynamic *access codes* to allow O(1) access to the stored information [21]. An access code represents the exact byte offset of a given record in a given database, and is dynami ally encoded into all resource identifiers. For any query that includes an access code, the database system bypasses its normal search mechanism and performs a seek operation to a byte offset that is one less than the numeric value of the access code. It then reads the word at that position. If the supplied access code is correct, this word will be a record separator. Assuming that a record separator is found, the subsequent read operation will fetch the first word of a record. PUNCH databases are organized so that the first word in every record consists of a string that uniquely identifies the record within that database. Typically, this string is the numeric value of the access code for that record: in a few cases, the name of the resource is used for the string when it (the name) is guaranteed to be unique (e.g., as with user logins). As a result of this organization, if the word read by the system matches the supplied access code (or the resource name when appropriate), the record being read is guaranteed to be the one that was requested. If the supplied access code is determined to be incorrect,, the requested information is retrieved via, a normal database search, and an access code is generated for use in subsequent queries.

Once obtained, access codes are cached at the source of the request, allowing them

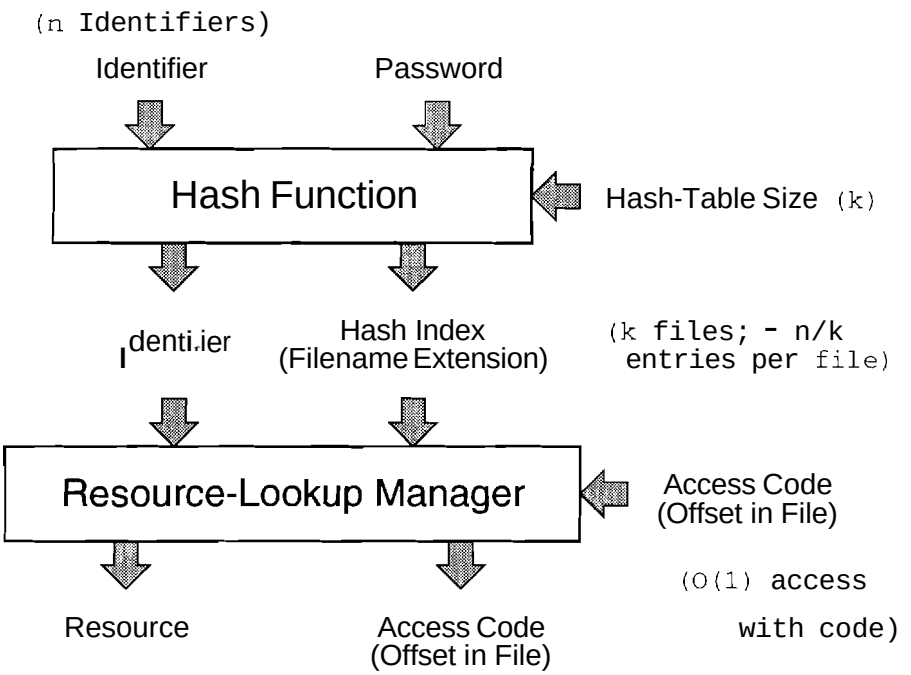


Fig. 4.5. Schematic of the lookup mechanism used by the PUNCH database system.

to be reused. However, access codes can change over time – although this happens relatively infrequently in practice. (The change is generally a result of a resource being added to or removed from the database.) In order to minimize the search penalty when access codes are not available or are incorrect, resources are hashed across multiple databases on the basis of the initial characters of their identifiers and encrypted or encoded passwords. A schematic of the lookup mechanism used by the PUNCH database system is shown in Figure 4.5.

Figure 4.6 shows the average time required to authenticate a reyuest when correct access codes are available. The results represent the same dataset (994 PUNCH user accounts) used to generate the plot in Figure 4.3; compare the range of Y-axis values in the two plots. The utility of access codes can be illustrated more effectively by way of a synthetic clataset. Consider the information associated with tools (e.g., templates employed for user-interface generation, learned resource usage characteristics, portability information, etc.). In order to process any tool-related transaction, the run-time system must first locate the database for the tool. The scalability characteristics of the lookup time, when the access code is not known or incorrect, are shown in the left hand plot in Figure 4.7; the right hand plot shows the (constant) lookup time when a correct access code is supplied to the database system. Both plots were generated for a synthetic dataset using a database system with a linear search.

A typical reyuest will contain several access codes – one for each database record that is accessed in the course of processing the corresponding transaction. For database records that consist of multiple sub-records, access codes are defined as a hierarchy of byte offsets.

Table 4.1 shows the observed proliferation of access codes in the PUNCH environment. About 96% of the transactions initiated by member users contained access codes that could be used for authentication purposes. Access code:; associated with tool-related information were present in virtually all transactions that involved such references, as indicated by the second row of the table.

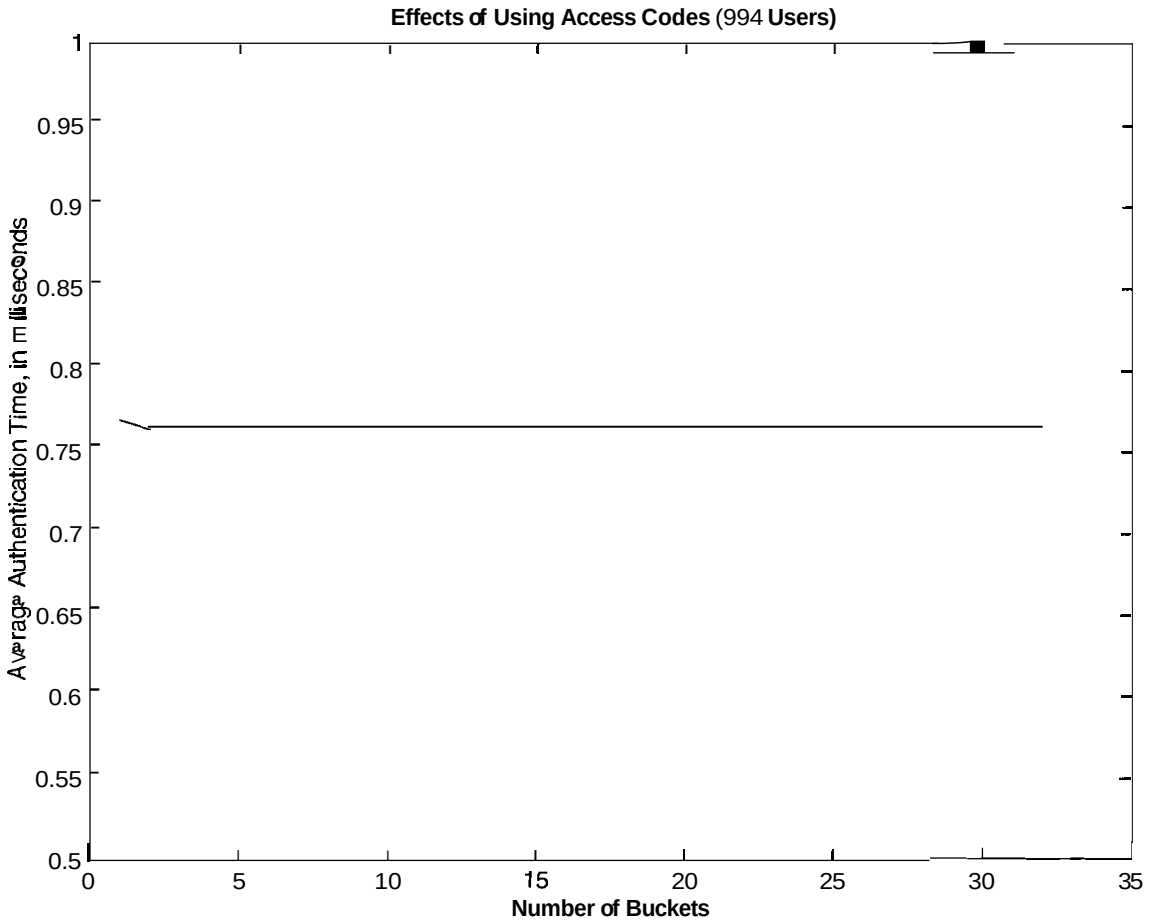


Fig. 4.6. Effects of using access codes on authentication time.

Table 4.1
Observed access code proliferation.

| Access Code Proliferation | | |
|---|---|---|
| Type of Use | Access Code | |
| | Present | Absent |
| Member Authentication | 1,086,732 | 45,896 |
| Internal References | 946,652 | 81 |

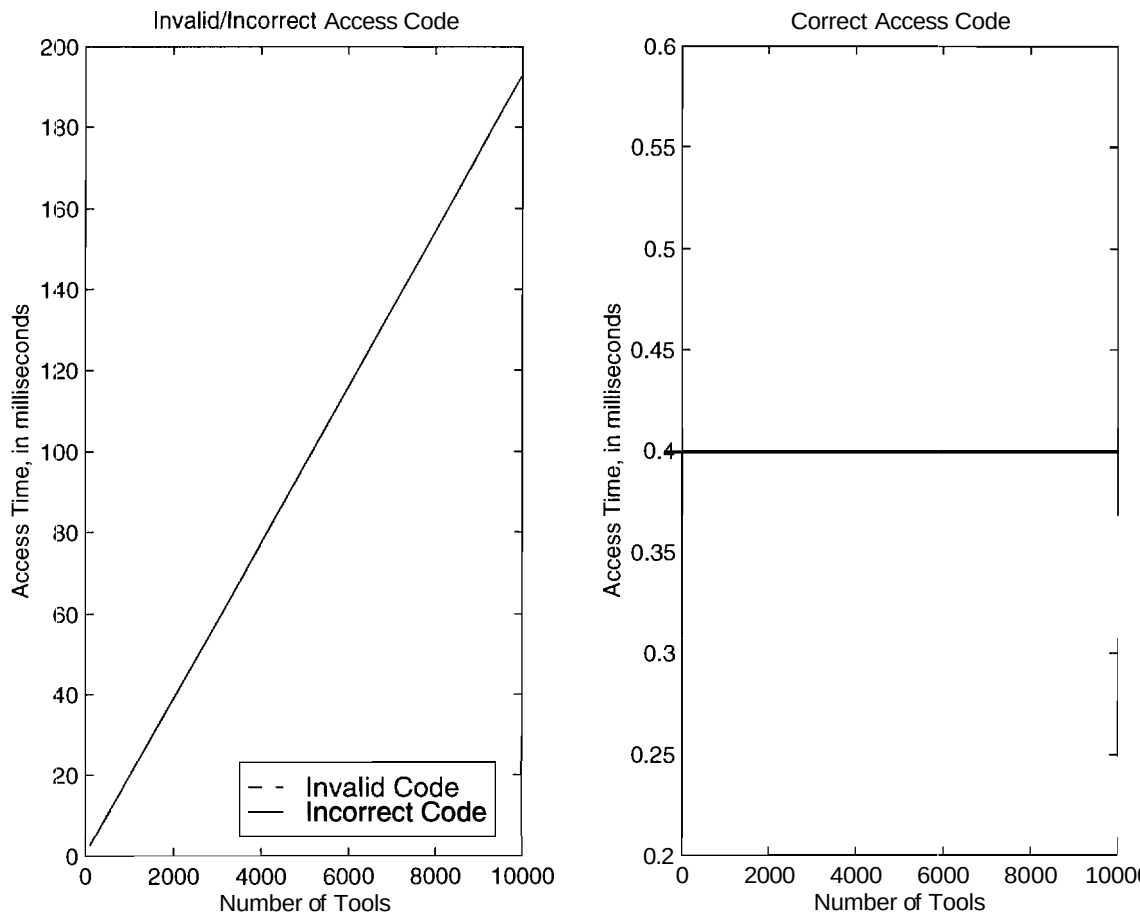


Fig. 4.7. Effects of access codes on information retrieval latency.

### 4.4.2 Address-Space Translation

PUNCH draws on the basic premise of virtual memory management to control access to its resources. Requests are assumed to contain virtual addresses (identifiers) that can be dynamically translated to physical ones. This model is used within the framework of the world-wide web by treating URLs as locations in a clynamic, virtual, and *active* (side-effect based) address space. The client initiating a given transaction is equated to a distinct user-level process in a multi-user system?and the request is treated as an attempt to access a memory location with a lexicographical address within the private address space of the associated process.

Using this perspective, analogous to virtual memory management, each request can be verified for validity in terms of: 1) being within the address space bounds for the associated process (user), and 2) the type of access (read or write). *All* accesses to PUNCH undergo this validation process, allowing the system to control access to its :resources on a per-user *and* a. per-resource basis.

### 4.4.3 View Customization

Once the validity of a request has been verified, it is *mapped* to a physical resource. This mapping process is implicitly dependent upon the source (user-id) of the request, and can be customized by system administrators. Thus, different users attempting to access the same logical resource identifier (URL, in the case of the world-wide web) could potentially trigger completely different actions.

PUNCH leverages this fact to dynamically generate a *logical* (virtual) view of available tools and resources for each class of users. For example, users who have member access to a given tool on PUNCH can view the corresponding on-line documentation by following an appropriate set of web links. Other users who follow these links are shown a different set of documents (typically a message stating that the user documentation for the tool is only available to authorized users). It is also possible to configure PUNCH so that such links are not generated at all for users who do not have member privileges for the corresponding tool.

# 5. The Network Desktop

## 5.1 Introduction

The network desktop component processes transactions that involve document serving, directory information, system and process status queries, file manipulation, and tool interface management. Data. obtained by profiling the user activity on PUNCH over the past twenty-eight months (see Table 5.1) indicates that transactions associated with tool interfaces, static and dynamic information, and process status queries account for more than 95% of the total. Consequently, the evaluation focuses on the modules that process these transactions. A schematic.for the network desktop component is shown in Figure 5.1.

## 5.2 Document Server

The *document server* module handles transactions involving access to static information that is stored in files that reside on locally mounted filesystems. The information is static in the sense that, except for minor substitutions (e.g., server-side includes)?it does not need to be processed or personalized before it is transmitted over the network. These transactions are the PUNCH equivalent of traditional document serving on the world-wide web. Of the 247,152 static information transactions (see Table 5.1), 92,524 (37%) resulted in accesses to text (e.g., HTML, PDF, postscript) files; the other 63% resulted in accesses to image (e.g., GIF, JPG) files. URLs associated with static information transactions are dynamically mapped to files by way of an administrator-specified *URL* map that allows the document server module to translate URLs that. match specified templates to corresponding paths to files. A sample URL map is shown in Figure 5.2; default values are enclosed within square brackets, and an asterisk indicates a wildcard.

Table 5.1
Summary of PUNCH user activity over twenty-eight months.

| Summary of PUNCH Access Statistics | | |
|---|---|---|
| **Type of Transaction** | **Number of Transactions** | **Distribution in Percent** |
| **Tool Interface** | **777,795** | 59.52 |
| **Static Information** | **247,152** | 18.912 |
| Process Status | **168,548** | 12.910 |
| Dynamic **Information** | **66,252** | 5.07 |
| **Directory Service** | **36,186** | 2.77 |
| **Account Creation** | **9,006** | 0.69 |
| **Invalid** | **1,753** | 0.13 |
| **Total** | **1,306,692** | **100.00** |

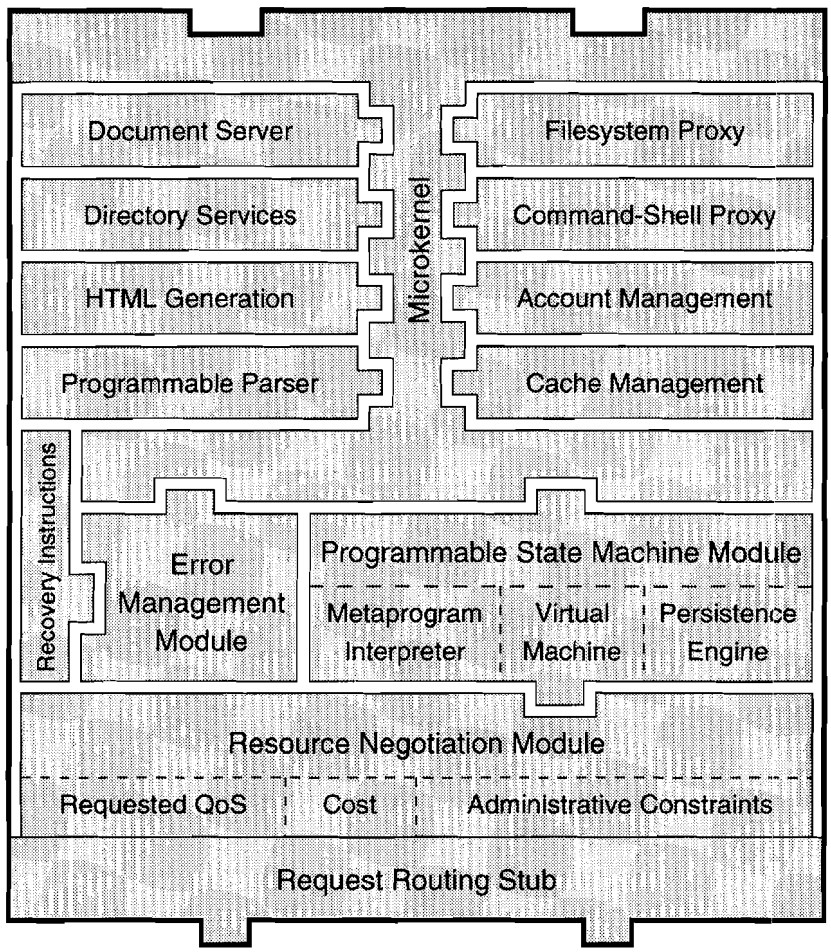


Fig. 5.1. Organization of the network-desktop component.

```
Begin  UrlMap
  /              /home/users/punch/www/[punch.html]
  /Directory/    /home/users/punch/www/directory.html
  /Help/*        /home/users/punch/help/[manual.html]
  /Images/       /home/users/punch/images/
  /People/*      /home/users/punch/info/people/[people.html]
End    UrlMap
```

Fig;. 5.2. A sample map for translating static information URLs. Default values are enclosed within square brackets, and an asterisk indicates a wildcard.

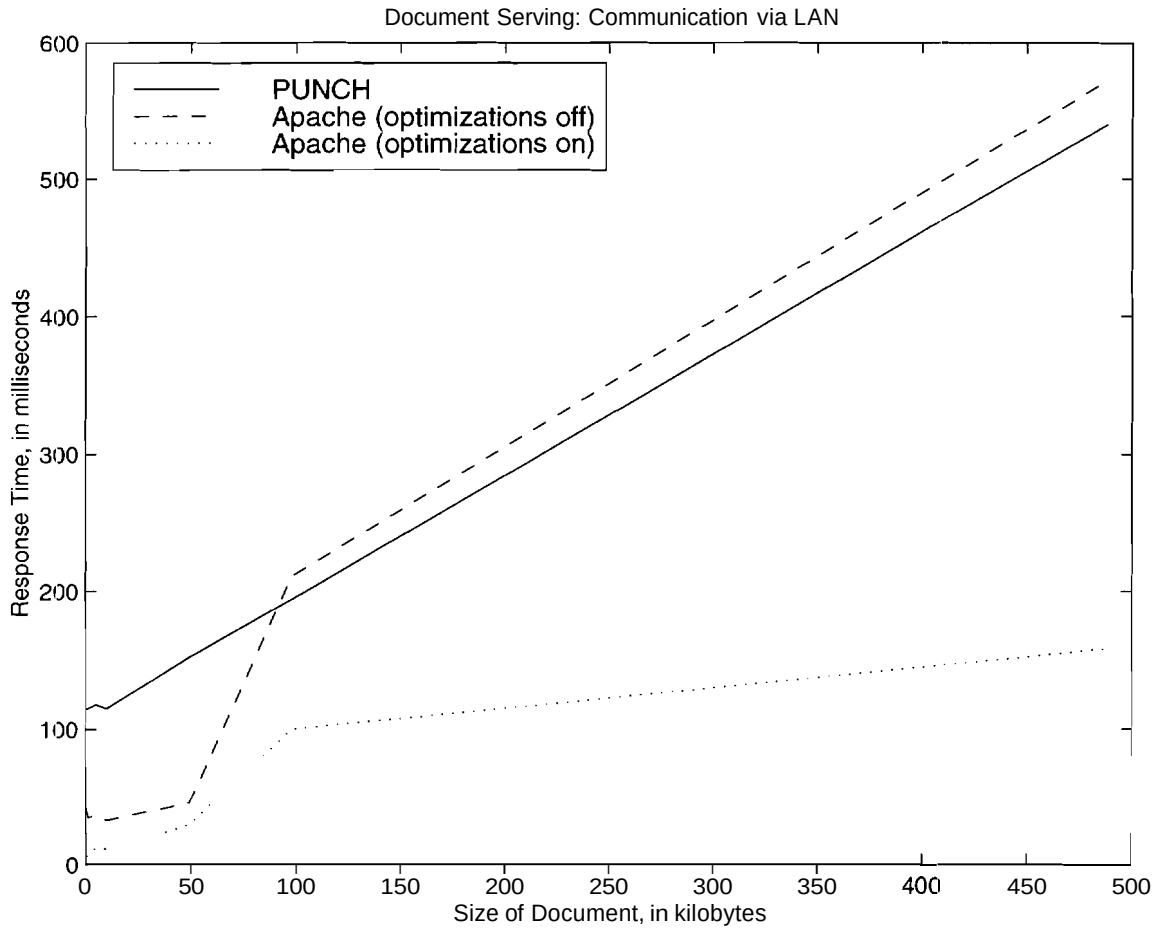


Fig. 5.3. Apache and PUNCH server response times as a function of document size.

The following results characterize the response time of PUNCH for static information transactions. Figure 5.3 shows the average response times associated with PUNCH and Apache [2, 16] for clocument serving, as a function of document size. The results were obtained with the client(s) running on a 167MHz Sun Ultra-2 and the servers running on a 300MHz Sun Ultra-4. The machines were connected via a 100Mb switched ethernet. Observe that, for similar configurations[1] (plots labelled PUNCH and Apache (optimizations off) in the figure), both servers scale equally well, although Apache exhibits a better response time for smaller clocuments. The PUNCH server has a higher overhead because it runs in an interpreted environment. The plot labelled Apache (optimizations on) in the figure shows the performance of Apache when it is allowed to pre-fork multiple processes and run spare servers in anticipation of demand – note the improved response time characteristics.

Once implemented, these optimizations can be expected to result in similar performance improvements in PUNCH. Figure 5.4 demonstrates the effects of preforking server processes in PUNCH – observe that the response time improves by a factor of two for small document sizes. In order to filter out the effects of the network, the response time characteristics shown in the figure were obtained with the client and the server running on the same machine (a 300MHz Ultra-4) and communicating via filehandles.

Figure 5.5 shows the average response times as a function of the number of simultaneous (but independent) requests. The results were obtained with the client(s) running on a 167MHz Sun Ultra,-2 and the servers running on a 300MHz Sun Ultra-4. The experiment's were repeated using different document sizes (two of which are shown in the figure) to account for artifacts due to network traffic. For the range shown in the plots, the PUNCH server scales somewhat better than a similarly configured Apache server, but the results (plot labelled Apache (optimizations on) in the figure) clearly show the performance benefits of the optimizations implemented

[1]The following changes were made to the default parameters provided in the Apache distribution (Version 1.3.3): MinSpareServers=1, MaxSpareServers=1, StartServers=1, and MaxRequestsPerChild=2. Apache's performance optimizations are largely disabled with this configuration.

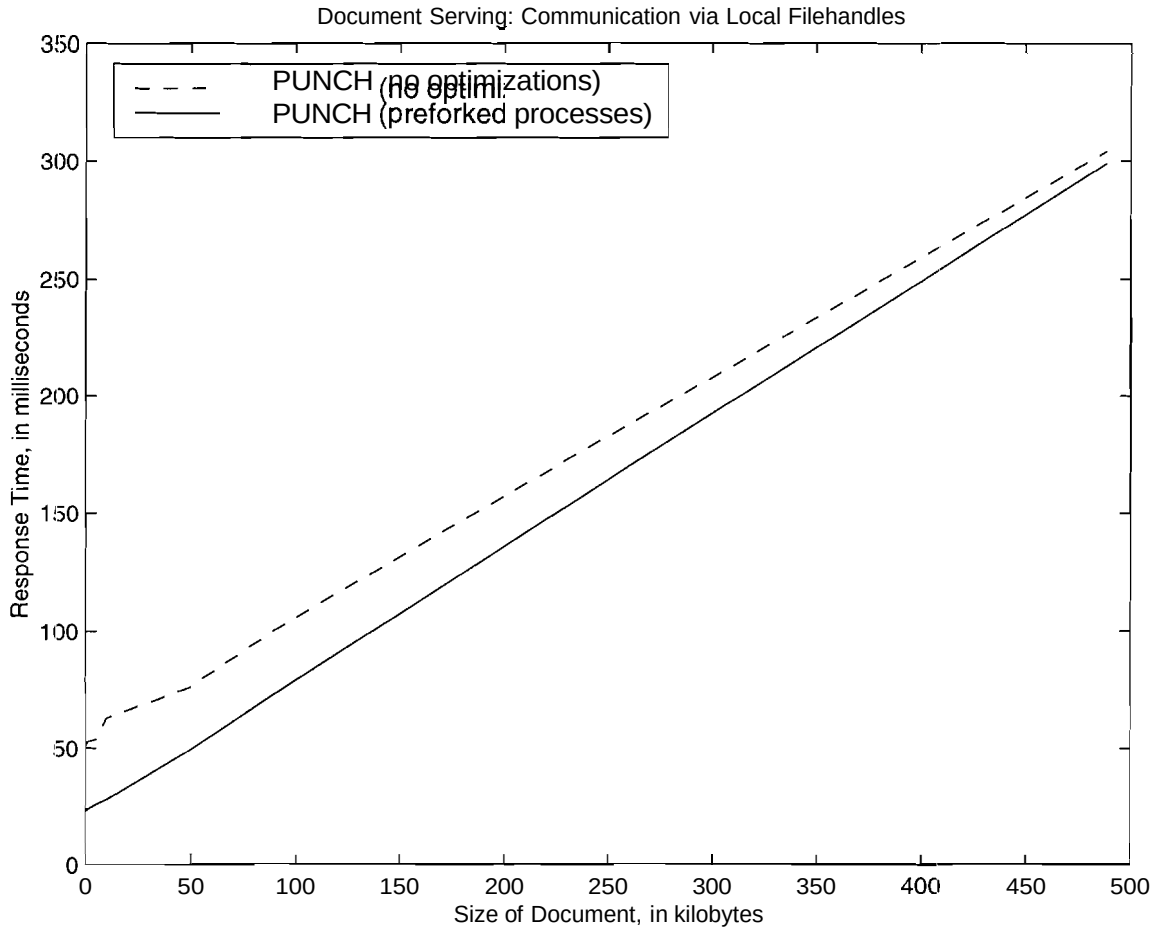


Fig. 5.4. PUNCH server response time as a function of document size.

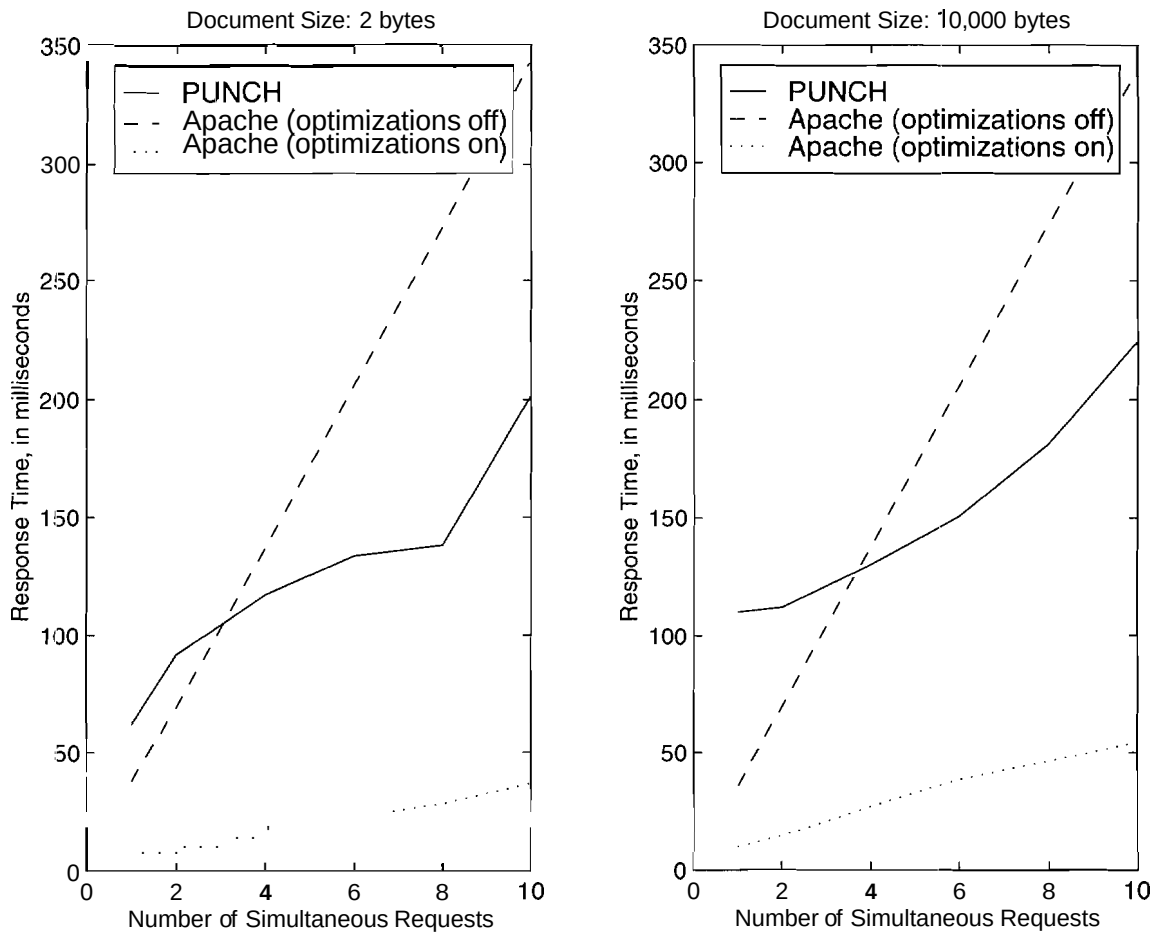


Fig. 5.5. Server response time as a function of the number of simultaneous requests.

in Apache.

### 5.3 Directory Services

The directory *services* module handles requests for complete or partial (cross-referenced) listings of tools available to users. These transactions typically access information that is distributed across multiple PUNCH databases. In order to minimize the latency visible to users, a full listing is generated once for each class of users and then cached. Crossreferenced listings are dynamically generated from the cached (full) listings. In practice, requests for full listings dominate requests for crossreferenced listings – of the 36,186 directory service transactions in Table 5.1, about 10,487 (29%) involved requests for crossreferenced listings. When a cached listing is available, the response time characteristics for directory service transactions are similar to those for document serving.

### 5.4 HTML Generation

'The network desktop interface provides support for embedding *variables* and objects within standard HTML; documents (HTML pages) that contain these variables and/or objects are called *HTML* templates. The *HTML* generation module behaves as a filter for HTML templates – it dynamically replaces the tokens representing these variables and objects with their current values as the template is being sent to a client (browser).

Variables include standard types provided by high-level languages – integers, floating point numbers, and strings. Objects include HTML-specific constructs such as check-boxes: menus, and radio-boxes. HTML templates are made up of two parts: 1) variable and object declarations, and 2) HTML code. The variables and objects serve the same purpose as corresponding constructs in high-level languages – they allow the compiler and the run-time system to track, interpret, and manipulate information (flow control is handled by the programmable state machine; see Section 5.9).

The functionality provided by the HTML generation module is best explained by way of an example. Consider a situation in which users need to "walk" through the

```
Begin  PageTemplate
  Begin  Declarations
    menu  myFiles = 1:<UserFiles>;
    bind  myFiles = <WorkingFolder>;
  End    Declarations

  Begin  HTML
    <FORM>
      Select a file (or folder to open): <myFiles> <P>
      <CENTER><INPUT TYPE="submit" VALUE="Proceed"></CENTER>
    <FORM>
  End    HTML
End    PageTemplate
```

Fig. 5.6. An example HTML template.

files and directories in their (PUNCH) accounts via a browser. Figure 5.6 shows a simple HTML template that can be used to accomplish this task. The left hand side of the first declaration statement specifies an object of type menu called `myFiles`. The right hand side specifies the size of the menu (one) and initializes its contents via a directory handle (`UserFiles`) exported by the desktop infrastructure. The second declaration statement binds the contents of the menu object to the current directory of the user for whom the transaction is being processed. Once a variable or an object has been declared, it can be referenced by name in the HTML part of the template, as shown in the figure. The name will he dynamically replaced with appropriate "static" HTML code and data when the template is accessed via an appropriate URL. For example, the reference to `myFiles` in Figure 5.6 will be replaced with the HTML specification for a menu whose contents are the list of files in the current directory of the user who initiated the transaction that references the template.

The syntax associated with the template shown in Figure 5.6 was specifically designed to supplement standard HTML without requiring any changes to the language. Unfortunately, this syntax does not lend itself to efficient run-time parsing of the template. This problem is addressed by using a compiler to translate the template into a format that is more conducive to efficient run-time substitution of variables and objects. The compiled version of the template is shown in Figure 5.7. The first line contains the size of the compiled template file in bytes; this is used to eliminate race conditions that arise when the file is accessed while it is in the process of being (asynchronously) loaded into the disk cache for the desktop. The second line contains information that is used for version control. The database "record" starts at the third line. As noted in Section 4.4.1, this line contains the access code for the record. The next four lines list the names and access codes of the four sub-records that make up this record (byte offsets for sub-records begin after the end of the record header). The next two lines contain the (sub)record separator and the access code for the first sub-record (`VarSpec`), respectively (the "5" represents the number of lines in the sub-record). The `VarSpec` record tells the HTML generation module how to locate and

```
543
v2.0
70
VarSpec:1
PageTemplate:117
BindSpec:290
VariableNames:397

1 5
2 5 8
B B M
_formTarget $formTarget
_hiddenTags @hiddenTags
_myFiles 1 0 0 - /$dirName/Input/$cUrl
End VarSpec

117 11
<FORM
_formTarget
>

_hiddenTags

Select a file (or folder to open):
_myFiles
 <P>
<CENTER><INPUT TYPE="submit" VALUE="Proceed"></CENTER>
</FORM>
End PageTemplate
```

Fig. 5.7. The compiled version of the HTML template.

interpret the variables and objects in the PageTemplate sub-record. The location information is contained in the first line within the sub-record: the numbers (2 5 8, in the figure) refer to the line numbers in the PageTemplate record that require substitution. The rest of the record contains information that supplies the semantics for the substitution process.

Observe that each variable/object is stored by itself on a separate line in the `PageTemplate` record. This organization allows for very efficient variable/object substitution. When a template is accessed, the HTML generation module first loads the VarSpec record. Then, it jumps to the PageTemplate record and starts sending its contents over the network connection. When it reaches a line whose number matches the ones listed in the `VarSpec` record, it sends (generates) the contents of the appropriate variable (object) instead of sending the tag on the line. Experiments show that the overhead of variable and object substitution using this procedure is negligible in comparison to the document serving times reported earlier.

### 5.5 Programmable Parser

The *programmable parser.*module provides the desktop infrastructure with a means to extract information that is embedded in strings (e.g., a specific flag in the set of command-line arguments to a tool) or files. The extracted information is subsequently used by the the programmable state machine module described in Section 5.9 to modify the behavior of the infrastructure according to specified criteria. This allows the desktop infrastructure to *react* to input provided by users and/or output generated by tools.

'The current implementation parses its input using grammar specified via regular expressions that are supplied by tool installers. This approach is limited to parsing context-free grammars, but it has been found to be adequate in most cases. This is a consequence of two factors: 1) the structure of the input to most programs tends to be relatively simple. and 2) the kind of information needed by the computing infrastructure (e.g., the names of files referenced within a given input file) does not

generally require the parser to understand the complete grammar for the underlying input.

The functionality provided by the programmable parser is best explained by way of an example. Figure 5.8 shows the grammar and the parsing rules supplied to the parser for T-Suprem3, a commercial package (from Technology Modeling Associates, Inc.) that simulates the processing steps used in the manufacture of silicon integrated circuits and discrete devices. A basic feel for the type of input being parsed can be obtained from the sample T-Suprem3 input file shown in Figure 5.9. The programmable parser uses Perl's regular expression engine for string matching; the state machine associated with the regular expression engine and the syntax for the regular expressions are described in the Perl5 documentation and in [18, 31]. The following explanation assumes a basic familiarity with Perl5 regular expressions.

The first two statements in the `AnalysisGrammar` section in Figure 5.8 are special cases – they provide the programmable parser with the grammar needed to parse files that contain comments and have statements that span multiple lines. The first statement specifies that, for T-Suprem3, lines that begin with a string that matches either CUMMENT, comment, or $ should be treated as a comment line (the '^' anchors the regular expression to the beginning of a line). A statement that specifies the comment-line grammar can optionally contain a second regular expression – in which case the comments are assumed to be enclosed within strings that match the first and second regular expressions, respectively. This format would be required to parse a file containing 'C'-like comments, for example. The second statement specifies that a given line in a T-Suprem3 file is to be treated as a continuation of the preceding line if the first character in the line is a plus sign (the '\s*' matches zero or more occurrences of spaces). For input files whose statements are not delimited by a newline character, an arbitrary regular expression for a statement delimiter can be specified via the `StatementDelimiter` keyword.

The remaining statements in the `AnalysisGrammar` section are treated as normal declaration statements. The left hand side of the statement can be any valid variable

```
Begin  AnalysisGrammar
  CommentLine_Grammar = '^((COMMENT)|(comment)|(\$))';
  ContinuationLine_Grammar = '^\+\s*';
  # The following regular expressions are used to determine the names
  # of data files referenced within other input files.
  RegExp1 = '^\s*init\S*\s+((in\.file)|(structur))\s*=\s*([0-9a-zA-Z._]+)';
  RegExp2 = '^\s*loadfile\s+((in\.file)|(file))\s*=\s*([0-9a-zA-Z._]+)';
  RegExp3 = '^\s*mask\s+in\.file\s*=\s*([0-9a-zA-Z._]+)';
  RegExp4 = '^\s*profile\s+((in\.file)|(file))\s*=\s*([0-9a-zA-Z._]+)';
  RegExp5 = '^\s*plot\s+((in\.file)|(data))\s*=\s*([0-9a-zA-Z._]+)';
  RegExp6 = '^\s*extract\s+((in\.file)|(profile))\s*=\s*([0-9a-zA-Z._]+)';
  RegExp7 = '^\s*call\s+file\s*=\s*([0-9]a-zA-Z._]+)';
  RegExp8 = '^\s*assign\s+in\.file\s*=\s*([0-9a-zA-Z._]+)';
End    AnalysisGrammar


Begin  AnalysisRules
  Ref1 = file:<inputFile>    RegExp1, RegExp2, RegExp3, RegExp4,
                             RegExp5, RegExp6, RegExp7, RegExp8;
  Ref2 = file:<*:RegExp7:1> RegExpl, RegExp2, RegExp3, RegExp4,
                             RegExp5, RegExp6, RegExp7, RegExp8;
End    AnalysisRules
```

Fig. 5.8. Sample grammar and parsing rules for the programmable parser.

```
TITLE       TMA SUPREM-3 - Example 1
+           MOS gate region simulation.

COMMENT     Start with the result of an earlier simulation
INITIALIZE  IN.FILE=S3EX1S

COMMENT     Perform the process steps for the source
+             and drain regions.
CALL        FILE=s3call1.inp

COMMENT     Plot the net chemical impurity distribution.
PLOT        CHEMICAL NET RIGHT=1.5 device=postscript
-           TITLE="Example 1 - Gate" plot.out=s3ex1g.ps
LABEL       LABEL="Chemical Net" START.LE LX.F=.9 X=1.1 Y=1e20

COMMENT     Save the final gate region structure.
SAVEFILE    STRUCTURE OUT.FILE=S3EX1GS
```

Fig. 5.9. Example T-Suprem3 input file.

name (using the 'C' naming semantics). and the right hand side can contain arbitrary Per15 regular expressions. As an example, the regular expression associated with `RegExp1` will match a statement that begins with a keyword whose initial part contains the string `init` (the '`\S*`' matches zero or more occurrences of non-space characters) and is followed by either of the strings `in.file` or `structur`, an equality sign, and a string that is made up of one or more occurrences of alphanumeric characters, underscores, and/or periods. Thus, regular expression associated with `RegExpI` will match the third statement in Figure 5.9. Similarly, the regular expression associated with `RegExp7` will match the fifth statement in Figure 5.9.

The `AnalysisRules` section in Figure 5.8 tells the programmable parser how to apply the regular expressions that were defined in the `AnalysisGrammar` section. Again, the left hand side can contain any valid 'C'-language variable name. The first token in the right hand side specifies the target file or string to be parsed via a PUNCH variable that has been previously declared in a template file (e.g., the `<myFiles>` object defined in the preceding section) or a metaprogram (see Section 5.9). The remaining tokens on the right hand side are the (names of) regular expressions that are to be applied to the file or string specified in the first token.

The results of regular expression matches can be referenced in templates and metaprograms via special variables. These variables are created and initialized at run time as and when they are referenced, using dataflow semantics. Special variables are named in terms of the names of the corresponding rules and regular expressions, and the index of the required match within the regular expression. The index of a match within a regular expression is defined in terms of Perl backreference variables – thus, any match that is referenced via a special variable must be enclosed within parentheses in the regular expression. The value of the index associated with a given pair of parentheses within a regular expression is determined by counting the number of opening parentheses from the beginning of the expression up to (and including) the pair in consideration. Thus, for `RegExp1` (see Figure 5.8), the indices for the matches corresponding to the patterns `in.file`, `structure`, and `[0-9a-zA-Z._]+` are two,

three, and four, respectively. The associated variahle names are <RefI:RegExp1:2>, <RefI:RegExp1:3>, and <RefI:RegExp1:4> (for rule **RefI).** If a particular regular expression matches more than one statement in a given file or string, the results are stored in the appropriate variable(s) as a space delimited list. The number of statements in a file or string that were matched by a particular regular expression can be determined by omitting the index field in the corresponding special variable (e.g., <RefI:RegExp1>).

T-Suprem3 input files can reference other input files via a **call** keyword (see fifth statement in Figure 5.9). The referenced input files, in turn, can reference other input files. This recursive behavior is captured by the second rule in the AnalysisRules section in Figure 5.8. Observe that the first field in the special variable that specifies the target file for the rule contains a wildcard character. The wildcard specifies that the corresponding rule should be applied to all files matched via RegExp7. Thus, if the file specified by <inputFile> in **RefI** contains a **call** statement that references another file. the second rule will be applied to the referenced file. If the referenced file contains another **call** statement, the second rule will subsequently be applied to the called file. This process will continue until the second rule has been applied to all files matched via RegExp7.

### *5.6* File, Process, and Account Management

'The *filesystem* proxy, command-shell proxy, and the account *management* modules supplement the file, process, and account management mechanisms provided by the local operating systems running on individual machines. File management is left entirely up to the local operating systems, except that all requests that manipulate file:, are routed via filesystem proxies. Filesystem proxies allow the logical filenames used within the distributed computing environment to be dynamically mapped to real pathnames within physical filesystems.

Similarly, system commands are routed via shell proxies, which rnap logical commands to real ones. Process management is accomplished by maintaining extended

process tables that complement the ones provided by the local operating systems. The extended process tables contain information such as the process identifier within the distributed computing environment, the machine(s) on which the process is running and the corresponding native process identifier(s), the current status of the process (e.g., queued, running, being terminated, etc.), the specific execution protocol (e.g., PVM, VNC, X-RX, etc.) associated with the process, the expected execution time, and local directory information. In the interest of efficiency, these tables are only updated periodically. Also, a separate table is maintained for each user that has an active process. This allows process status queries to be handled very efficiently (the scalability characteristics are almost identical to the ones shown for document serving). Modifications to the process status (e.g., a request to abort a process) involve significantly higher overhead, however, because the desktop component is required to contact remote machines and wait for acknowledgement. Most of the observed process status transactions involve status queries.

Account management in a distributed computing environment is complicated by the fact that, in general, it is impractical to create physical accounts for individual users on all the resources available to the system. This problem is a, consequence of three factors: the size of the system (in terms of the number of users: and resources), the dynamic nature of the system (users and resources can be addled and removed at will), and the varying administrative policies (available resources typically span multiple institutions). The basic problem is addressed by dynamically creating *logical* user accounts within a single physical account on the underlying operating system. The filesystem proxies and shell proxies ensure that users can only access and modify their own data – the associated mapping processes are implicitly keyed to the user who initiates the request. Security concerns with respect to tools that allow users to make direct system-level calls are addressed by way of *shadow accounts.* Shadow accounts consist of a pool of physical accounts that are dynamically assigned to individual users when they attempt to execute "unsafe" commands, allowing such commands to be executed safely within separate physical accounts. The accounts are subsequently

reclaimed by the system. Shadow accounts are managed in a secure manner without requiring superuser privileges by leveraging protocols that can establish one-way trust (e.g., secure shell [30]). Access management with smaller granularity can be achieved via software fault isolation techniques (e.g., [19]).

## 5.7 Cache Management

The *cache management* module is designed to work as an independent process (or thread) that periodically clears cached templates and database entries. The cache is managed on a local disk (e.g., in `/tmp`). This module does not affect the performance of the run-time system during normal operation, except to the extent that the contents of the cache need to be reloaded after they have been flushed.

## 5.8 Error Management

The *error management* module handles error logging, management, and recovery operations. Control is passed to this module when an abnormal condition or an internal error is detected. When this happens, the module first logs critical state information and a stack trace. (A synopsis of this information is mailed to specified system administrators periodically.) Then, if necessary, the error management module undoes the side effects of the partially processed transaction. (The information required to do this is dynamically generated by PUNCH modules in the course of processing a transaction.) Finally, the module initiates an appropriate, configurable error recovery procedure, and, depending on the results, instructs the calling module to retry or abort its procedure. This module does not affect the performance of the system during normal operation.

## 5.9 Programmable State Machine

The *programmable state machine* module has three functional units: a metaprogram interpreter, a virtual machine, and a persistence engine. A metaprogram is a program whose instructions are themselves programs. In this environment, metaprograins are used to define the behavior of the programmable state machine, and are

Table 5.2
List of instructions supported by the metaprogramming language, in addition to standard flow control. (i.e., conditionals and loops) instructions.

| Instruction | | Description |
|---|---|---|
| cache | on\|off | Enable/disable the caching of templates for this metaprogram. |
| chdir | <directory> | Change directory - this instruction updates internal :state information. |
| cleardir | <directory> | Delete the contents of the specified directory. |
| dirof | <file> | Get the name of the directory in which the specified file resides. |
| display | <html template> | Generate HTML code using specified template; stop metaprograrn. |
| end | <metaprogram> | Terminate rnetaprograrn - execution cannot be resumed. |
| execute | <metaprogram>\|<tool> | Execute metaprogram/tool on possibly remote machine; non-blocking. |
| exists | <filename> | Finds the absolute path to a file, if it exists within searched directories. |
| isdef | <variable> | Evaluates to one if variable is defined, zero otherwise. |
| isdir | <variable> | Evaluates to one if variable contains a valid directory, zero otherwise. |
| length | <string> | Determines the length of the string in bytes. |
| logstatus | <status file> | Log current status; used for error notification and recovery. |
| maxof | <list> | Determines the largest value within a space delimited list. |
| minof | <list> | Determines the smallest value within a space delimited list. |
| retrieve | <name>, [<var list>] | Retrieve saved values of specified variables. |
| retrievestate | <name> | Restore the state of the metaprogram from a previously saved image. |
| save | <name>, <var list> | Save specified variables to persistent storage. |
| savestate | <name> | Save an image of the current state of the metaprogram. |
| set | <variable> = <value> | Assign a value to the specified variable. |
| stop | | Stop the execution of the rnetaprograrn - execution may be resumed. |
| sumof | <list> | Determines the numerical sum of the values in a space delimited list. |
| writeln | <var list> | Writes values of the variables in the list to the current output. |

written in a specially designed 'C'-like language. In addition to the standard flow control constructs (conditionals and loops) available in high-level languages, this language provides instructions to: 1) customize and manipulate information for different types of users, 2) manage files and directories, 3) customize and serve documents and templates in response to user requests, 4) save and retrieve state information, and 5) start and stop child metaprograms and processes on local or remote machines. These instructions are summarized in Table 5.2. As the name suggests, the metaprogram interpreter parses and executes the metaprograms. The virtual machine manages the run-time environment for the metaprogram interpreter. It execute:; the lower-level code that makes up the instructions in the metaprograms, enforces access control at the instruction level, and maintains low-level data structures (e.g., the stack) and state information (e.g., the program counter) associated with the underlying compute engine. Finally, the persistence engine maintains state information that allows metaprograms to be started and stopped as and when necessary.

The programmable state machine can be used to generated a customized HTML interface for each tool. The input to this state-machine consists of: 1) a list of available states, and 2) a description of the transitions between these states (flow control). States are specified in terms of HTML templates. Flow control information is specified via metaprograms. The state machine executes metaprograms in response to an attempt to access corresponding URLs. It keeps tracle of the values of variables and objects for interfaces that span multiple HTML pages by storing them in hidden form fields and/or encoding them within URLs. Moreover, the state-machine can react to any run-specific information (e.g., values embedded within user-supplied strings, data within files, etc.) that can be extracted by the programmable parser described earlier. This allows the desktop interface to support programs that accept input in an interactive manner.[2]

The roles of the three functional units described above are best explained by

[2]Programs with graphical user interfaces are managed via browser-compatible remote-display protocols such as VNC [26].

```
string myFiles = :/';
retrievestate 'directory';
display 'Page1';
while(isdir(<myFiles>))
{
  chdir(<myFiles>);
  display 'Page1';
}
savestate 'directory';
          •
          •
          •
```

Fig. 5.10. An example metaprogram.

using an example. Figure 5.10 shows a sample metaprogram that could be used in conjunction with the HTML template described in Section 5.4 to allow users to "walk" through their files and directories. The execution of the metaprogram is triggered when a user accesses an appropriate URL, and proceeds as follows. The first instruction shown in the figure is a declaration statement that causes the value of the variable **myFiles** to be initialized to the root directory (for this user). The second instruction shown in the figure causes the persistence engine to set the variables and objects in the metaprogram to the values saved previously (for this particular user) via a 'savestate' instruction. This instruction does not affect the values of variables and objects for which no state inforination exists. The third instruction triggers several steps. The virtual machine: 1) instructs the persistence engine to save the internal state of the metaprogram, 2) generates access code and program counter information that can be used to restart the metaprogram at the appropriate instruction (i.e., the one following the current 'display' instruction), **3)** uses the HTML generation module to send the template associated with 'Page1' (previously specified via a declaration statement) to the user who initiated the transaction, and 4) shuts the metaprogram down. At this point, the transaction that initiated the execution of the metaprogram is considered to be complete. On the browser (client) side, it simply appears as if a standard HTML document was returned in response to an attempt to access a given URL. Subsequently, when the user selects an entry from the contents of the menu and submits another request, the metaprogram is automatically restarted at the instruction that follows the previously executed 'display' instruction. This is accomplished as follows. The access code and program counter information generated by the virtual machine in response to a 'display' instruction is cached within hidden fields in the corresponding document and/or is encoded into the URLs that appear within that document. On the subsequent request, this information is extracted by the microkernel (see Figure 5.1) and presented to the metaprogram interpreter. The interpreter, in turn, restarts the metaprogram and instructs the persistence engine to restore its internal state. At this point the instruction that follows the 'display'

instruction can be executed without regard to the fact that the previous instructions were executed by a different process, possibly on a different machine. In this particular example, the first instruction in the restarted metaprogram is the test for the 'while' statement. If the menu entry selected by the user happens to be a directory, the programmable state machine will change the current directory for the user and display the HTML template again. The contents of the menu will now represent the contents of the new current directory. Eventually, when the menu entry selected by the user is not a directory?the test for the loop will fail and the metaprogram interpreter will execute subsequent instructions.

In the interest of run-time efficiency, metaprograms are compiled prior to execution. The instructions generated by the compiler for the example metaprogram in Figure 5.10 are shown in Figure 5.11. Each instruction in the compiled metaprogram consists of multiple fields that are separated by two colons, as shown in the figure. The number in the first field of any given instruction represents the byte offset of the following instruction. When a metaprogram is stopped after executing an instruction (e.g., `display`), this number represents the byte offset at which the metaprogram should be restarted. The number in the second fielcl identifies the relative position of the instruction in the metaprogram – it is used by the virtual machine to maintain a program counter. Observe that the information conveyed by the first; and the second fields is redundant – this allows the virtual machine to maintain correct operation in the presence incorrect or missing access codes. The string in the third fielcl represents the name of the instruction; the set of valid strings is the instruction set supported by the virtual machine. The semantics of the remaining fields depend on the type of instruction. For example, in the case of the `test` instruction, the fourth and the fifth fields represent the byte offset and the relative position of the instruction to be executed if the result of the test is true; the sixth and the seventh fields contain the address of the instruction to be executed if the test evaluates to either zero, false, or undefined using the evaluation semantics of Perl5. The final field in the instruction contains the expression that is to be evaluated for the test. This expression is

```
27::1::set::<_myFiles>::/
59::2::retrievestate::directory
121::3::display::Page1::/home/users/punch/templates/page1.meta
167::4::test::167::5::277::8::-dir <_myFiles>
193::5::chdir::<_myFiles>
256::6::display::Page1::/home/users/punch/templates/page1.meta
277::7::goto::121::4
306::8::savestate::directory
```

•
•
•

Fig. 5.11. The compiled version of the example metaprogram.

represented in the prefix notation, and is executed by way of a stack: machine.

The ability to start and stop metaprograms at will has several advantages. For example, it allows the virtual desktop infrastructure to work within the framework of the stateless world-wide web protocols. On the other hand, this ability makes it necessary to specifically address information consistency issues. Given the time-distributed nature of the execution process, it is possible that a metaprogram will be updated and recompiled by an administrator before it has been completely executed. In order to help detect the resulting information inconsistency, all compiled rnetaprograms contain a sequence number that represent's the time of modification of the source metaprogram file. This sequence number is encoded into the program counter used to restart a metaprogram. During the startup procedure, the encoded sequence number is compared to the one contained in the compiled metaprogram and execution is aborted if the numbers do not match.[3]

'The programmable state machine handles all transactions that need to be interpreted within a specified context (i.e., involve flow control). This includes tool interface, dynamic information, and account creation transactions (see Table 5.1). In order to optimize the response time for common types of transactions, a few metaprograms are integrated into the programmable state machine module. Such metaprograms can be executed more quickly because they do not incur any startup overhead. On the other hand, integrated metaprograms cannot be updated without modifying the programmable state machine module. Consequently, this optimization is only useful for situations in which the metaprograms are not expected to change. The integrated metaprograms handle entry page, input management, and output management transactions associated with tool interface generation (see Table ***5.3).*** They also handle dynamic information links for which customization is limited to simple variable substitution. Entry pages (see Figure 5.12) contain links to tool-specific information (e.g., manuals) and to the metaprograms associated with input

[3]In principle, one could lock and/or copy metaprograms being executed. However, this does not guarantee consistency because a. metaprogram may reference other metaprograms that could also have been modified.

Table *5.3*
Distribution of tool interface transactions.

| Distribution of Tool Interface Transactions | | |
|---|---|---|
| **Type of Transaction** | **Number of Transactions** | **Distribution in Percent** |
| Entry Page | 35,139 | 4.52 |
| Input Management | 287,861 | 37.01 |
| Run Management | 178,677 | 22.97 |
| Output Management | 276,118 | 35.50 |
| Total | 777,795 | 100.00 |

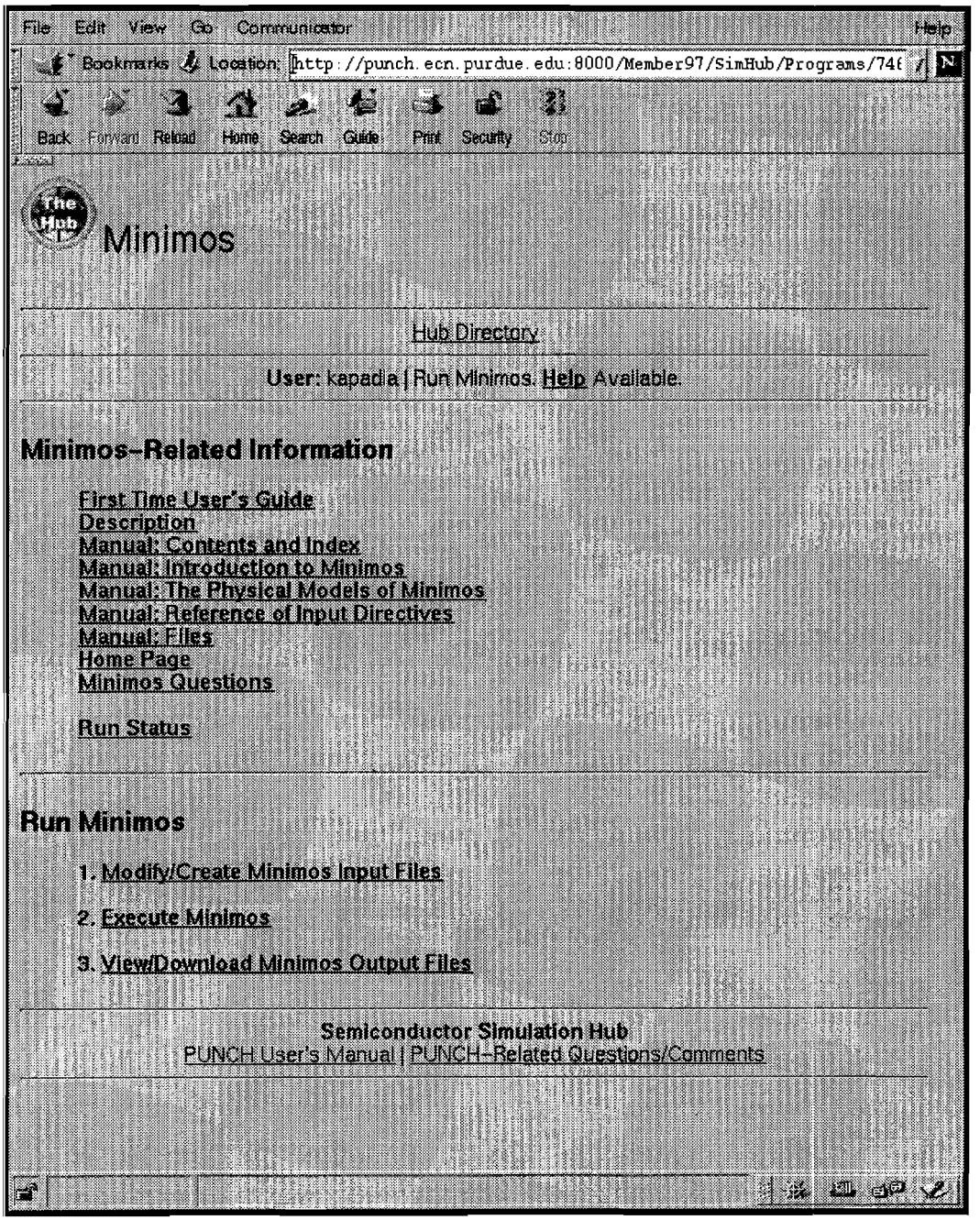


Fig. 5.12. The entry point to the PUNCH user interface for MINIMOS 6.0.

management (Step 1 in the figure), run management (Step 2 in the figure), and output management (Step 3 in the figure). Input management and output management transactions involve processing and manipulating input and output data files, respectively. Run management transactions allow users to select input and output files and directories, specify command-line arguments, and initiate runs.

The average response times of the PUNCH server for several common types of transactions are shown in Table 5.4. In order to filter out the effects of the network, the response times shown in the table were obtained with the client and the server running on the same machine (a 300MHz Ultra-4) and communicating via filehandles. The responses generated by the PUNCH server for the specific transactions used to obtain the results shown in Table 5.4 ranged in size from about one to three kilobytes. Experiments (see Figure 5.4) show that, for the described setup and the given range of data sizes, the differences in data transfer times are less than two milliseconds. (Large differences and/or variations in data transfer times would adversely affect the reliability of the results.)

Several conclusions can be drawn from the results in Table 5.4, the most obvious one being that there is a significant time penalty associated with forking a process in response to a (real-time) request. The response times associated with entry page, input management, and output management transactions are about 20% lower than the corresponding times for run management transactions. This improvement is the result of using integrated metaprograms that do not incur any startup overhead. The response times for static information transactions are included in the table because they serve as a benchmark for the other results. Static information transactions that access "public" documents involve no user-specific authentication or access control – a request of this type is simply mapped to a physical file which is then returned to the client. Static information transactions that access "private" documents are processed in a similar manner, except that they undergo the dynamic authentication and access control procedures described in Section 4.1. The results in the table show that these procedures add about five milliseconds to the time required to process a transaction.

Table 5.4
PUNCH server response times for different types of transactions.

| | Type of Transaction | Response Time (milliseconds) | |
|---|---|---|---|
| | | No Preforking | With Preforking |
| Tol In trfa | Entry Page | 60 | 27 |
| | Input Management | 61 | 27 |
| | Run Management | 75 | 35 |
| | Output Management | 61 | 27 |
| Stat'c Info | Publicly-Accessible URL | 47 | 20 |
| | Access Control Enforced | 53 | 24 |
| | Process Status | 64 | 30 |
| | Dynamic Information | 58 | 29 |

Finally, the last two rows in the table show the response times associated with process status queries (with one active process) and dynamic information transactions (using an integrated metaprogram) – these are largely included in the table for completeness.

### 5.10 Resource Negotiation

The *resource negotiatzon* module selects an appropriate managernent sub-system for transactions that require a tool to be invoked. The selection process is based on criteria that can be customized for specific sets of users and resource:,. In the current implementation, administrators specify a prioritized list of management sub-systems to contact for each class of users and resources; the resource negotiation module uses this list to forward tool-invocation requests to the first system that is willing to serve the request. Also, in order to distribute peak loads, the resource negotiation module dynamically lowers the priority associated with a given management sub-system as the number of pending requests associated with that system increase. Support for QoS negotiation is not yet available.

# 6. Conclusions

## 6.1 Conclusions

The desktop infrastructure described in this report serves as the front end for PUNCH. It currently provides access to over thirty tools developed by eight universities and four vendors. The ideas and solutions presented in this report are based on (and validated by) our experiences in scaling PUNCH from a research project to a "live" system that is regularly used by several hundred students each semester. Results from user-surveys indicate that the system performs well under the highly peaked usage patterns (very high usage in the hours before homeworks and projects are due) characteristic of an academic environment.

The PUNCH infrastructure has been successfully applied to education, research, and technology-transfer. PUNCH serves as the underlying distributed computing infrastructure for two collaborative efforts involving five universities: the integration of design tools into new undergraduate and graduate curricula, and the Distributed Center for Advanced Electronics Simulations (DESCARTES). PUNCH is also the enabling infrastructure for a statewide Purdue University network-computing system currently being deployed. Over the years, we have found PUNCH to be an extremely useful resource for students and collaborators, and a highly flexible testbed for network-computing research.

List of References

[1] Christopher Adasiewicz. Exploratorium: User friendly science and engineering. *NCSA Access*, 9(2):10–11, 1995.

[2] The Apache HTTP server project. WWW site at www.apache.org.

[3] Peter Arbenz, Walter Gander, and Michael Oettli. The Remote Computation System. In *High-Performance Computing and Networking (Lecture Notes in Computer Science. 1067)*. Springer-Verlag, Berlin, 1996.

[4] Peter Arbenz, Walter Gander, and Michael Oettli. The Remote Computation System. *Parallel Computing*, 23:1421–1428, 1997.

[5] T. Berners-Lee and D. Connolly. RFC 1866: Hypertext markup language - 2.0. Internet Engineering Task Force (IETF) request for comments, November 1995.

[6] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0. Internet Engineering Task Force (IETF) request for comments, May 1996.

[7] N. Borenstein and N. Freed. RFC 1521: Mime (multipurpose internet mail extensions) part one – mechanisms for specifying and describing the format of internet message bodies. Internet Engineering Task Force (IETF) request for comments, September 1993. Obsolete; see RFC 2045.

[8] Broadway overview. www.camb.opengroup.org/tech/desktop/x/broadway.htm, 1996. Open Group Desktop Technologies.

[9] *X Window System Version 11 Release 6.3: Release Notes*, 1996

[10] X web white papers. Web Documents at www.broadwayinfo.com/bwwhites.htm, 1997. Broadwayinfo.com.

[11] Henri Casanova and Jack Dongarra. NetSolve: A network solver for solving computational science problems. In *Proceedings of the Supercomputing Conference*, 1996. Also Technical Report #CS-95-313, University of Tennessee.

[12] ICA positioning paper. WWW document at www.citrix.com/technology/, March 1996. Citrix Systems, Inc.

[13] ICA technical paper. WWW document at www.citrix.com/technology/, March 1996. Citrix Systems, Inc.

[14] David H. Crocker. RFC 822: Standard for the format of ARPA internet text messages. Internet Engineering Task Force (IETF) request for comments, August 1982.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2068: Hypertext transfer protocol – HTTP/1.1. Internet Engineering Task Force (IETF) request for comments, January 1997.

[16] Roy T. Fielding and Gail Kaiser. The Apache HTTP server project. IEEE Internet Computing, 1(4):88–90, July/August 1997.

[17] N. Freed and N. Borenstein. RFC 2045: Multipurpose internet mail extensions (mime), part one – format of internet message bodies. Internet Engineering Task Force (IETF) request for comments, November 1996.

[18] Jeffrey E. F. Friedl. *Mastering Regular* Expressions. O'Reilly & Associates, 1997.

[19] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In Proceedings of the Sixth *USENIX Security* Symposium, San Jose, California, July 1996.

[20] Nirav H. Kapadia, Carla E. Brodley, José A. B. Fortes, and Mark S. Lundstrom. Resource-usage prediction for demand-based network-computing. In Proceedings of *the* 1998 Workshop on *Advances in* Parallel *and* Disfributed Systems *(APADS)*, West Lafayette, Indiana, October 1998.

[21] Nirav H. Kapadia and José A. B. Fortes. On the design of a demand-based network-computing system: The Purdue University Network-Computing Hubs. In *Proceedings* of the 7th IEEE International Symposium on *High Performance* Distributed *Computing (HPDC'98)*, pages 71–80, Chicago, Illinois, July 1998.

[22] Nirav H. Kapadia and Jose A. B. Fortes. PUNCH: An architecture for web-enabled wide-area network-computing. *Cluster Computing*, 1999. To appear in the special issue on High Performance Distributed Computing.

[23] Nirav H. Kapadia, José A. B. Fortes. and Mark S. Lundstrom. The Semiconductor Simulation Hub: A network-based microelectronics simulation laboratory. In *Proceedings* of the *12th* Biennial *University Government Industry Microelectronics Symposium*, pages 72–77, July 1997.

[24] HTML 4.0 specification. World Wide Web Consortium (W3C) Recommendation, April 1998. Available at http://www.w3.org/TR/1998/REC-html40-19980424/.

[25] A. Reinefeld, R. Raraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Romke, and J. Simon. The MOL project: An open, extensible metacomputer. In *Proceedings* of the 1997 IEEE *Heterogeneous Computing Workshop (HCW97)*, pages 17–31, 1997.

[26] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. IEEE Internet Conzputing, 2(1):33–38, January-February 1998.

[27] Mitsuhisa Sato, Hidemoto Nakacla, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In High-Performance *Computing* and *Networking* (Lecture *Notes* in *Computer Science. 1225)*, pages 491–502. Springer-Verlag, Berlin, 1997.

[28] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. -Ninf-: Network-based information library for globally high performance computing. In *Proceedings of Parallel Object-Oriented Methods and Applications (POOMA)*, Santa Fe, February 1996.

[29] Dan Souder, Morgan Herrington, Rajat P. Garg, and Dennis DeRyke. JSPICE: A component-based distributed Java front-end for SPICE. In *Proceedings of the 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[30] SSH 2.0 protocol specifications. Internet Engineering Task Force (IETF) drafts available at http://info.internet.isi.edu/l /in-drafts.

[31] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 2nd edition, 1996.