Techniques and Applications of Information Data Acquisition on the World Wide Web During Heavy Client/Server Traffic Periods

Stathes Hadjiefthymiades, Drakoulis Martakos
Department of Informatics, University of Athens

(in "Database and Data Communication Network Systems" (editor Cornelius Leondes), Academic Press, June 2002.)

1. INTRODUCTION

In the second half of the 90s, the WWW has evolved to the dominant software technology and broke several barriers. Despite the fact that it was initially intended for a wide (universal) area network, the WWW, today, enjoys a tremendous success and penetration even in corporate environments (intranets). Key players in the software arena, like Oracle, Informix and Microsoft, recognize the success of this open platform and constantly adapt their products to it. Originally conceived as a tool for the co-operation between high-energy physicists, this network service is nowadays becoming synonymous with the Internet as it is used vastly by the majority of Internet users even for tasks like e-mail communication.

The universal acceptance of the WWW stimulated the need to provide access to the vast legacy of existing heterogeneous information[1]. Such information ranged from proprietary representation formats to engineering and financial databases, which were, since the introduction of the WWW technology, accessed through specialized tools and individually developed applications. The vast majority of the various information sources were stored in Relational DBMS (RDBMS), a technology which enjoys wide acceptance in all kinds of applications and environments. The advent of the Web provided a unique opportunity for accessing such data repositories, through a common front-end interface, in an easy an inexpensive manner. The importance of the synergy of WWW and database technologies is also broadened by the constantly increasing management requirements for Web content ("database systems are often used as high-end Web servers, as webmasters with a million of pages of content invariably switch to a web site managed by database technology rather than using file system technology", extract from the Asilomar Report on Database Research [Ber98]).

Initial research on the WWW-database framework did not address performance issues but since this area is becoming more and more important, relevant concern is beginning to grow. The main topic of this chapter, namely the study of the behavior exhibited by WWW-enabled information systems under heavy workload, spans a number of very important issues, directly associated with the efficiency of service provision. Gateway specifications (e.g., CGI, ISAPI) are very important due to their use for porting existing systems and applications to the WWW environment. Apart from the gateway specification used, the architecture of the interface towards the information repository (i.e.,

[1] The significance of this issue triggered the organization of the workshop on Web Access to Legacy Data, in parallel to the Fourth International WWW Conference, held in Boston, MA in December 1995.

RDBMS) is also extremely important (even with the same specification quite different architectures may exhibit quite different behavior in terms of performance). The internal architecture of HTTP server software is also an important issue is such considerations. Older single process architectures are surely slower in dispatching clients' request than newer multi-threaded schemes. Pre-spawned server instances (processes or threads) also play an important role.

Lastly, as part of our study, we are considering how existing WWW-based systems are being measured. We provide a brief description of some of the most important performance evaluation tools.

It should be noted that the entire chapter is focused on the 3-tier scenario where access to a legacy information system is always realized through a browser-server combined architecture (i.e., HTTP is always used), as shown in Figure 1, (as opposed to an applet based architecture where access to the legacy system can be realized directly, without the intervention of HTTP; a 2-tier scheme).

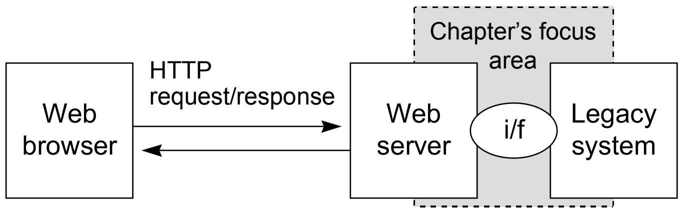

Figure 1: Basic architectural assumption: a 3-tier scheme

As illustrated in Figure 1, we do not consider network-side aspects (e.g., network latency, HTTP, etc.). The network infrastructure underneath Internet is continuously becoming more and more sophisticated with technologies like ATM, xDSL, etc. The best-effort model for Internet communications is being supplemented with the introduction of QoS frameworks like Integrated Services/Differentiated Services. Lastly, on the application layer, HTTP/1.1, Cascading Style Sheets (CSS) and Portable Network Graphics (PNG) show W3C's intention to "alleviate Web slowdown" [Kha97].

2. GATEWAY SPECIFICATIONS

In terms of gateway specifications considerable work has been done in the area of the Common Gateway Specification (CGI) and its comparison with proprietary interfaces like ISAPI and NSAPI. Other specifications like Fast CGI, the Servlet API and Netscape's CORBA-based WAI (Web Application Interface) have also emerged and are extensively used nowadays. In the following paragraphs we will try to set the scene for studying accesses to legacy systems by providing very brief introductions to major gateway specifications.

2.1 Common Gateway Interface

The Common Gateway Interface (CGI) is a relatively simple interface mechanism for running external programs (general purpose software, gateways to existing systems) in the context of a WWW information server in a platform independent way. The mechanism has been in use in the WWW since 1993. By that time, CGI appeared in the server developed by the National Center for Supercomputing Applications (NCSA http demon, httpd). The CGI specification is currently in the Internet Draft status [Coa98]. Practically, CGI specifies a protocol for information exchange between the information server and the aforementioned external programs as well as a method for their invocation by the WWW server (clause 5 of the Internet Draft). Data are supplied to the external program by the information server through environment variables or the 'standard input' file descriptor. CGI is a language independent specification. Therefore, CGI programs may be developed in a wide variety of languages like C, C++, Tcl/Tk, Perl, Bourne shell or even Java. Owing to its simplicity, support for CGI is provided in almost all WWW servers (WWW-server independence). A very important issue associated with the deployment of CGI is the execution strategy followed by the controlling WWW server. CGI-based programs (also referred to as scripts) run as short-lived processes separately to the httpd. As such, they are not into position to modify basic server functionality (i.e., logging) or share resources with each-other and the httpd. Additionally, they impose considerable resource waste and time overhead due to their one process/request scheme. A more detailed discussion on the pros and cons of the CGI design can be found in Section 2.7.

During the evolution of WWW software, key commercial players like Netscape and Microsoft recognized the deficiency of CGI and introduced their own proprietary interfaces for extending basic server functionality. Such mechanisms were named NSAPI (Netscape Server API) and ISAPI (Internet Server API) respectively. NSAPI was supported in the Communications/Commerce generation of servers and now, in the FastTrack/Enterprise series. ISAPI first appeared in the initial version of Internet Information Server (IIS) and is a standard feature of Microsoft server software ever since then. Other efforts towards the same direction are the FastCGI specification from Open Market Inc., the Java Servlet API from Sun, and the Web Application Interface (WAI) from Netscape Communications.

2.2 Netscape Server API

The NSAPI specification enables the development of server plug-ins (also called Service Application Functions or SAFs) in C or C++ that are loaded and invoked during different stages of the HTTP request processing. Since server plug-ins run within the Netscape server's process (such plug-ins are initialized and loaded when the server starts up), the plug-in functions can be invoked with little cost (no separate process needs to be spawned as in the case of CGI). Also, NSAPI exposes the server's internal procedure for processing a request. It is, therefore, feasible to develop a server plug-in that is invoked during any of the steps in this procedure. These steps are briefly presented below:

- **AuthTrans** (authorization translation): verification of authorization information included in the request.
- **NameTrans** (name translation): mapping of the request URI into a local file system path.
- **PathCheck** (path checking): verification that the local file system path is valid and that the requesting user is authorized to access it.
- **ObjectType** (object typing): determination of the MIME-type of the requested resource (e.g. text/html, image/gif).
- **Service** (service): the response is returned to the client.
- **AddLog** (adding log entries): server's log file is updated.

The server executes SAFs in response to incoming requests through a MIME mapping approach. Should the incoming request pertain to a hypothetical MIME type (also called internal server type), the associated SAF code is invoked to handle the request. Such mapping is maintained in a basic configuration file. Each SAF has its own settings and has access to the request information and any other server variables created or modified by previous SAFs. The SAF performs its operation based on all this information. It may examine, modify, or create server variables based on the current request and its purpose within its step.

2.3 Web Application Interface

WAI [Net97] is one of the programming interfaces, provided in the latest Netscape servers, that allows the extension of their functionality. WAI is a CORBA-based programming interface that defines object interfaces to the HTTP request/response data as well as server information. WAI compliant applications can be developed in C, C++, or Java (JDK 1.1). WAI applications accept HTTP requests from clients, process them, and, lastly generate the appropriate responses. Server plug-ins may also be developed using WAI. Netscape claims that WAI accomplishes considerable performance improvements when compared to CGI. Since WAI-compliant modules (or WAI services) are persistent (in contrast to CGI programs), response times are reduced thus, improving performance. Additionally, WAI modules are multi-threaded so the creation of additional processes is unnecessary.

Specifically, WAI allows: accessing the HTTP request headers, accessing server internal information, reading data transmitted from the browser (i.e., the content of a POST request triggered by the submission of an HTML form), setting the headers and the status in the HTTP response, redirecting the interacting client to another location, or composing the response that will be send to the client (e.g., an HTML page). WAI modules may either be applications that run outside the server process in the same or different machines (the CORBA ORB deals with all the details) or be in the form of server plug-ins that run within the server's process. Server plug-ins can be shared libraries or dynamic link libraries.

2.4 Internet Server API

ISAPI [Mic97a], [Tra96] is an interface to WWW servers that allows the extension of their basic functionality in an efficient way. ISAPI compliant modules run in Windows 9x/NT environments and have the form of dynamic link libraries (DLL). ISAPI DLLs can be loaded and called by a WWW server and provide similar functionality to CGI applications (such ISAPI DLLs are called extensions). The competitive advantage of ISAPI over CGI is that ISAPI code runs in the same address space as the WWW server (i.e., in-process) and has access to all the resources available to the server (thus, the danger of a server crash due to error-prone extension code is not negligible). Additionally, ISAPI extensions, due to their multi-threaded orientation, have lower overhead than CGI applications because they do not require the creation of additional processes upon reception of new requests (the creation of a new thread by the server is much faster) and do not perform time-consuming communications across process boundaries. ISAPI DLLs may be unloaded if the memory is needed by another process. Interaction between the ISAPI extension code and the WWW server is performed through a memory structure called Extension Control Block (ECB). Each request addressed to the ISAPI extension causes a new ECB structure to be allocated and filled with information such as the QUERY_STRING parameter [Coa98] encountered in conventional CGI applications. Filters are another type of ISAPI extensions. Filters are loaded once, upon server's start-up, and invoked for each incoming request. Practically, ISAPI Filters allow the customization of the flow of data within the server process. ISAPI is now used by several Web servers including Microsoft, Process Software, and Spyglass.

2.5 FastCGI

FastCGI [Bro96a] is basically an effort to provide a "new implementation of CGI" [Bro96b] that would enjoy the portability of its predecessor while overcoming its performance handicap. FastCGI processes (gateway instances according to the terminology in this chapter) are persistent in the sense that after dispatching some request remain memory resident (and do not exit as conventional CGI dictates) awaiting for another request to arrive. Instead of using environment variables, stdin and stdout, FastCGI communicates with the server process - similarly to CGI it runs as a separate process - through a full duplex Socket connection. Basing the interface on Sockets allows the execution of the gateway instance on a machine different from that of the WWW server (a distributed approach). The information communicated over this full-duplex connection is identical to the one exchanged in the CGI case. Thus, migration from classical CGI programs to FastCGI is a fairly easy process. FastCGI supports the language independence of its predecessor (languages like Tcl/Tk and Perl can be used). FastCGI applications can be programmed either in a single-threaded or in a multi-threaded way.

The FastCGI framework supports a performance enhancement technique called "session affinity". Session affinity allows the server to route incoming requests to specific memory resident copies of FastCGI processes based on information conveyed within requests (e.g., username/password of the authentication scheme). The performance benefit comes from the caching that FastCGI applications are allowed to perform on user-related data.

In a database access scenario presented in [Bro96c], the performance of API-based applications in a single-threaded server architecture is slightly better than that accomplished by FastCGI processes (with internal caching/session affinity disabled). In those tests, the connections established towards the database system are not torn-down after the execution of individual queries (database connection caching).

2.6 Servlet Java API

Servlets are protocol- and platform-independent server side components written in Java. Servlets run within Java enabled web servers (e.g., Java Web Server, Netscape, Apache, etc.) and are to the server side what applets are to the client side. Servlets can be used in many ways but generally regarded as a replacement to of CGI for the dynamic creation of HTML content. Servlets first appeared with the Java Web Server launched in '97 [Cha97]. Servlets allow the realization of three-tier schemes where access to a legacy database or another system is accomplished using some specification like Java Database Connectivity (JDBC) or Internet Inter-ORB Protocol (IIOP). Unlike CGI, the Servlet code stays resident after the dispatch of an incoming request. To handle simultaneous requests new threads are spawned instead of processes. A web server uses the Servlet API, a generic call-back interface to control the behavior of a Servlet (i.e., initialization - termination through the invocation of specific methods). Servlets may be loaded from the local file-system or invoked from a remote host much like in the applet case.

As reported in [Cha97], FastCGI and Java Servlets accomplish comparable performance which is much higher than that of conventional CGI. Specifically, Organic Online, a San Francisco based Web development company, measured a 3-4 requests/sec throughput for FastCGI or Servlets while CGI managed to handle only 0.25 requests/sec.

Some concerns about the performance of Servlets in Web servers other than the Java Web Server are raised in [Maz98]. Servlets execute on top of the Java Virtual Machine (JVM). Spawning the JVM is a time- and resource-consuming task (a time of 5 seconds and a memory requirement of 4 MB is reported for a standard PC configuration). Such a consumption could even compromise the advantage of Servlet over conventional CGI if a different instance of JVM was needed for each individual request reaching the web server. In the Java Web Server case there is no need to invoke a new instance of JVM since the Servlet can be executed by the JVM running the server module (a "two-tier" scheme). In the Apache case, the Servlet engine is a stand-alone application (called JServ), running independently to the server process. The web server passes requests to this pre-spawned engine which then undertakes their execution (a proxy like scheme). A quite similar scheme is followed in every other Web server whose implementation is not based in Java. Indicative examples are the latest versions of Microsoft's IIS and Netscape's FastTrack. Popular external Servlet engines include JRun from Live Software

Inc. (http://www.livesoftware.com/) and WAICoolRunner from Gefion Software (http://www.gefionsoftware.com/). WAICoolRunner has been used in some of the experiments documented in subsequent paragraphs.

2.7 Comparisons between CGI and Server APIs

Gateway specifications are compared in terms of characteristics or the performance they can achieve. A detailed comparative analysis of the characteristics of CGI and server APIs can be found in [Eve96] and [PJu97].

Specifically, CGI is the most widely deployed mechanism for integrating WWW servers with legacy information systems. However, its design does not match the performance requirements of contemporary applications: CGI applications do not run within the server process. In addition to the performance overhead (new process per request), this implies that CGI applications can't modify the behavior of the server's internal operations, such as logging and authorization (see Section 2.2). Finally, CGI is viewed as a security issue, due to its connection to a user-level shell.

Server APIs can be considered as an efficient alternative to CGI. This is mainly attributed to the fact that server APIs entail a considerable performance increase and load decrease as gateway applications run in or as part of the server processes (practically the invocation of the gateway module is equivalent to a regular function call[2]) instead of starting a completely new process for each new request, as the CGI specification dictates. Furthermore, through the APIs, the operation of the server process can be customized to the individual needs of each site. The disadvantages of the API solution include the limited portability of the gateway code which is attributed to the absence of standardization (completely different syntax and command sets) and strong dependence to internal server architecture.

The choice for the programming language in API configurations is extremely restricted if compared to CGI (C or C++ Vs C, C++, Perl, Tcl/Tk, Rexx, Python and a wide range of other languages). As API-based programs are allowed to modify the basic functionality offered by the web server, there is always the concern of buggy code that may lead to crashes.

The two scenarios involve quite different resource requirements (e.g., memory) as discussed in [PJu97]. In the CGI case, the resource needs are proportional to the number of clients which are simultaneously served. In the API case, resource needs are substantially lower due to the function-call like implementation of gateways and multi-threaded server architecture.

In terms of performance, many evaluation reports have been published during the past years. Such reports clearly show the advantages of server APIs over CGI or other similar

[2] Differences arise from the start-up strategies followed: some schemes involve that gateway instances/modules are pre-loaded to the server while others follow the on-demand invocation.

mechanisms but also discuss performance differences between various commercial products.

In [Hay95], three UNIX-based web servers (NCSA, Netscape and OpenMarket) have been compared using the WebStone benchmark (see Section 5.1). Under extensive load, the NSAPI configuration achieved 119% better throughput[3] than the CGI configuration running in NCSA. In terms of connections per second, the Netscape/NSAPI configuration outperformed NCSA/CGI by 73%. OpenMarket/CGI achieved slightly better performance than the NCSA/CGI combination. Under any other load class, NSAPI outperformed the other configurations under test but the gap wasn't as extensive as in the high load case.

In [Hay96], the comparison between server API implementations and CGI shows the ISAPI is around 5 times faster than CGI (in terms of throughput). NSAPI, on the other hand, is reported only 2 times faster than CGI. A quite similar ratio is achieved for the connections per second metric. In terms of Average Response Times, ISAPI is reported at the 1/3 of NSAPI's performance.

In [Min96], published by Mindcraft Inc. (the company that purchased WebStone from Silicon Graphics), quite different results are reported for the two APIs. Specifically, NSAPI is shown to outperform, in certain cases, ISAPI by 40%. In other cases, NSAPI achieves a 82% of the best performance exhibited by ISAPI. Such differences, in the measurements included in the two aforementioned reports, have been attributed to certain modifications which were incorporated in the NSAPI code provided with the WebStone benchmark. No clear results can be obtained by the two performance evaluation tests. Microsoft's IIS (and ISAPI) seem to be better harmonized with the Windows environments while NSAPI is a portable specification (it can be applied to Netscape servers irrespective of the underlying OS).

[3] Throughput measured in MBps.

3. ARCHITECTURES OF RDBMS GATEWAYS

One of the most important application of the WWW platform refers to the integration of database management systems and RDBMS in particular. Many issues are associated with this particular type of WWW applications namely the generic architecture, the stateful operation and performance, which is of prime concern in this chapter.

The first WWW interfaces for relational management systems appeared in the 93-94 period. The first products begun to come into view on '95 with WebDBC [Nom95] from Nomad (later StormCloud), Cold Fusion [All96] from Allaire and dbWeb [Lau95] from Aspect Software Engineering (later acquired by Microsoft). The importance of such middleware triggered a very rapid pace in the development of this kind of software. Such interfacing tools have become crucial and indispensable components to the Internet as well as enterprise intranets. The first generation of WWW-RDBMS (e.g., WebDBC) interfacing products was mainly intended for the Windows NT platform and capitalized on the ODBC specification to gain access to databases.

Throughout the development of the generic RDBMS interface presented in [Had96] the authors observed that, during users' queries for information, a considerable amount of time was spent for the establishment of connections towards the relational management system (Informix in our prototype which was based on the CGI specification). Irrespectively of the efficiency of the adopted gateway mechanism (i.e., the "slow" CGI Vs a "fast" server API), establishing a new connection (per request) to the relational management system is a time- and resource-consuming task which should be limited and, if possible, avoided. Of course, the CGI specification is not efficient in terms of execution speed (and, in any case aggravates the performance of the system) but if portability and compliance to standards do necessitate its adoption, a "clever" solution has to be worked out.

In [Had97] we adopted the scheme of permanent connections towards the database management system [PJu97]. Permanent connections are established by one or more demon processes which reside within the serving host and execute queries on the behalf of specific clients (and gateway instances i.e., CGI scripts). Demons, prevent the inefficient establishment of a large number of database connections and the relevant resource waste; the associated cost is incurred by the demon processes (prior to actual hit dispatch - query execution) and not by the gateway instances (e.g., CGI script, ISAPI thread) upon hit dispatch. Thus, no additional time overhead is perceived by the interacting user in his queries for information. The discussed scheme is shown in Figure 2.

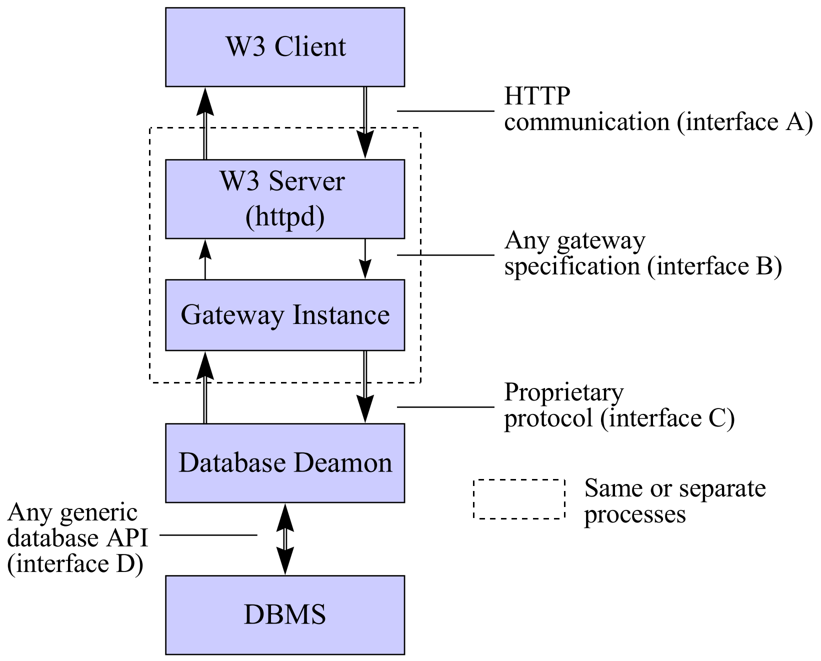
W3 Client
HTTP communication (interface A)
W3 Server (httpd)
Any gateway specification (interface B)
Gateway Instance
Proprietary protocol (interface C)
Database Deamon
Same or separate processes
Any generic database API (interface D)
DBMS

Figure 2: Optimized WWW-DBMS interface

3.1 Generic DBMS Interfaces

A very important issue in the demon-based architecture shown in Figure 2 is the interface towards the DBMS (interface D) as well as the communication mechanism between the gateway instance (interface C in Figure 2). The demon process should be able to dispatch any kind of queries - SQL statements irrespective of database, table and field structures. This requirement discourages the adoption of a static interface technique like the embedding of SQL statements in typical 3GLs (i.e., the Embedded SQL API - ISO SQL-92 standard). Instead, a generic interface towards the database should be used.

Contemporary RDBMS offer, as part of the Embedded SQL framework, the Dynamic SQL capability [Mic97b], [Inf96], which allows the execution of any type of statement without prior knowledge of the relevant database schema, table and field structures (unlike Static SQL, where such information is hard-coded in the respective programs). Dynamic SQL statements can be built at runtime and placed in a string host variable. Subsequently, they are posted to the RDBMS for processing. As the RDBMS needs to generate an access plan at runtime for dynamic SQL statements, dynamic SQL is slower than its static counterpart. This last statement deserves some more discussion: if a generic interface towards a DBMS is pursued, then the execution efficiency of the system could be undermined. Embedded SQL scripts with hard-coded statements execute faster than CGI scripts with dynamic SQL capabilities. The real benefit, in this architecture, comes from the de-coupling of the dynamic SQL part (database demon) from the actual gateway instance (Figure 2).

A second option for generic access to a DBMS is the SQL Call-Level Interface (CLI). The SQL CLI was originally defined by the SQL Access Group (SAG) to provide a unified standard for remote data access. The CLI requires the use of intelligent database drivers that accept a call and translate it into the native database server's access language. The CLI is used by front-end tools to gain access to the RDBMS; the latter should incorporate the appropriate driver. The CLI requires a driver for every database to which it connects. Each driver must be written for a specific server using the server's access methods and network transport stack. The SAG API is based on Dynamic SQL (statements still need to be Prepared[4] and then Executed). On the fall of 1994, the SAG CLI became an X/Open specification (currently it is also referred to as X/Open CLI) and later-on an ISO international standard (ISO 9075-3). Practically, the X/Open CLI is a SQL wrapper (a procedural interface to the language); a library of DBMS functions - based on SQL - which can be invoked by an application. As stated in [Mic97b], "a CLI interface is similar to Dynamic SQL, in that SQL statements are passed to the RDBMS for processing at runtime, but it differs from Embedded SQL as a whole in that there are no embedded SQL statements and no pre-compiler is needed".

Microsoft's Open DataBase Connectivity (ODBC) API is based on the X/Open CLI. The 1.0 version of the specification and the relevant Software Development Kit (SDK), launched by Microsoft in 1992, have been extensively criticized for poor performance and limited documentation. Initially, ODBC was confined to the MS-Windows platform, but later was ported to other platforms like Sun's Solaris. The ODBC 2.0, which was announced in 1994, has been considerably improved over its predecessor. 32-bit support contributed to the efficiency of the new generation of ODBC drivers. In 1996, Microsoft announced ODBC 3.0. Nowadays, most database vendors (e.g., Oracle, Informix, Sybase) support the ODBC API in addition to their native SQL APIs. But, a number of problems and hypotheses undermine the future of ODBC technology. ODBC introduces substantial overhead (especially in SQL updates and inserts) due to the extra layers of its architecture (usually, ODBC sits on top of some other vendor-specific middleware like Oracle's SQL*Net). It is a specification entirely controlled by Microsoft. The introduction of the OLE/DB framework gave grounds to the suspicion that Microsoft doesn't seem committed to progress ODBC any more.

Since the advent of the Java programming language, a new SQL CLI has emerged. It is named JDBC (Java DataBase Connectivity) and was the result of a joint development effort by Javasoft, Sybase, Informix, IBM and other vendors. JDBC is a portable, object-oriented CLI, written entirely in Java but very similar to ODBC [Orf98]. It allows the development of DBMS independent Java code which is, at the same time, independent of the executing platform. JDBC's architecture, similarly to ODBC, introduces a driver manager (JDBC driver manager) for controlling individual DBMS drivers. Applications share a common interface with the driver manager. A classification of JDBC drivers suggests that they are either direct or ODBC-bridged. Specifically, there are four types of JDBC drivers [JDB97]:

- Type 1 refers to the ODBC-bridged architecture and involves the introduction of an translation interface between the JDBC and the ODBC driver. ODBC binary code,

[4] It is requested by the RDBMS to parse, validate, and optimise the involved statement and, subsequently, generate an execution plan for it.

and the required database client code must be present in the communicating party. Thus, it is not appropriate for applets running in a browser environment.

- Type 2 drivers are based on the native protocols of individual DBMS (i.e., vendor-specific) and were developed using both Java and native code (Java methods invoke C or C++ functions provided by the database vendor).
- Type 3 drivers are exclusively Java based. They use a vendor-neutral protocol to transmit (over TCP/IP sockets) SQL statements to the DBMS thus, necessitating the presence of a conversion interface (middleware) on the side of the DBMS.
- Type 4 drivers are also exclusively based on Java (pure Java driver) but, in contrast to Type 3, use a DBMS specific protocol (native) to deliver SQL statements to the DBMS.

As discussed in [Nan97], Type 1 drivers are the slowest owing to the bridging technique. Type 2 drivers are on the extreme other end. Type 3 drivers load quickly (due to their limited size) but do not execute requests are fast as Type 2. Similarly, Type 4 execute quite efficiently but are not comparable to Type 2. Type 1 is generally two times slower than Type 4. In the same article, it is argued that the highest consumption of CPU power in JDBC drivers comes from conversions between different data types and the needed translation between interfaces. JDBC is included, as a standard set of core functions, in Java Development Kit (JDK) ver. 1.1.

3.2 Protocols and Inter-process Communication Mechanisms

Another very important issue in the architecture shown in Figure 2 is the C interface (i.e., the interface between gateway instances and the database demon). As discussed in [Had97], the definition of interface C involves the adoption of a protocol between the two co-operating entities (i.e., the gateway instance and the database demon) as well as the selection of the proper IPC (Inter-Process Communication) mechanism for its implementation.

In the same paper we proposed a simplistic, request/response, client/server protocol for the realization of the interface. The gateway instance (ISAPI/NSAPI thread, CGI process, Servlet, etc.) transmits to the database demon a ClientRequest message and the database demon responds with a ServerResponse. The ClientRequest message indicates the database to be accessed, the SQL statement to be executed, an identifier of the transmitting entity/instance as well as the layout of the anticipated results (results are returned merged with HTML tags). The Backus-Naur Form (BNF) of ClientRequest is:

```
ClientRequest = DatabaseName SQLStatement [ClientIdentifier] ResultsLayout
DatabaseName = *OCTET
SQLStatement = *OCTET
ClientIdentifier = *DIGIT ; UNIX PID
ResultsLayout = "TABLE" | "PRE" | "OPTION"
```

ClientIdentifier is the process identifier (integer, long integer) of a CGI script (or thread identifier in the case of a server API) which generated the request. This field is optional, depending on the IPC mechanism used and could be avoided in a connection-oriented communication (e.g., Sockets). *OCTET denotes a sequence of printable characters and thus, represents a text field. Results are communicated back to the client processes by means of the ServerResponse message. The BNF of ServerResponse is:

```
ServerResponse = ResponseFragment ContinueFlow
ResponseFragmet = *OCTET
ContinueFlow = "YES" | "NO"
```

The ResponseFragment (text) field contains the actual information which was retrieved by the demon process from the designated database. As mentioned, such information is returned to the client, embedded within HTML code. The type of tags used is the one specified in the ResultsLayout field of ClientRequest. The ContinueFlow field is used for optimizing, in certain IPC scenarios (e.g., Message Queues), the transmission of results back to the gateway instance.

The ClientRequest-ServerResponse protocol is very simplistic, and shifts processing from the gateway instance (client) to the database demon (a “thin” client scheme). More advanced protocols (i.e., RDA/DRDA) may be used over the C interface to allow more complicated processing at the side of the gateway instance.

Protocols pertaining to the C interface are deployed using either some IPC mechanism like Message Queues [Had97], [Had99a] or Sockets [Had99b] (low level middleware) or some kind of middleware like CORBA/IIOP. Message Queues [Ste90] are a quite efficient, message oriented, IPC mechanism that allows the simultaneous realization of more than one dialogs (i.e., various messages, addressed to different processes or threads, can be multiplexed in the same queue). BSD Sockets are ideal for implementation scenarios where the web server (and, consequently, the gateway instances) and the database demon execute on different hosts (a distributed approach). The advent of the WinSock library for the Microsoft Windows environments rendered Sockets a universal, platform independent IPC scheme. Message Queues are faster than Sockets but restrict the communication in a single host since they are maintained at the kernel of the operating system. In some recent tests, which are also presented in this chapter, for the realization of a scheme similar to the one shown in Figure 2, we have employed Named Pipes under the Windows NT environment. A Named Pipe is a one-way or two-way communication mechanism between a server process and one or more client processes executing on the same or different nodes (networked IPC). Under Windows NT, Named Pipes are implemented as a file system and share many characteristics associated with files with respect to use and manipulation.

A type of middleware, used extensively nowadays in many application domains, is the CORBA Object Request Broker [OMG97]. CORBA simplifies the development of distributed applications with components that collaborate reliably, transparently and in a scaleable way.

The efficiency of CORBA ORB for building interfaces between Java applications is discussed in [Orf98]. It is reported that CORBA performs similarly (and, in some cases better) to Socket based implementations while, only buffering entails a substantial improvement in Socket communications. Another comparison between several implementations of CORBA ORB (e.g., Orbix, ORBeline) and other types of middleware like Sockets can be found in [Gok96]. In particular, low level implementations such as Socket-based C modules and C++ wrappers for Sockets significantly outperformed their CORBA or RPC higher level competitors (tests were performed over high-speed ATM networks - traffic was generated by the TTCP protocol benchmark tool). Differences in performance ranged from 20 to 70% depending on the data types transferred through the middleware (transmission of structures with binary fields has proved considerably "heavier" than scalar data types). A very important competitor of CORBA is DCOM developed by Microsoft. DCOM ships with Microsoft operating systems thus, increasing the impact of this emerging specification.

3.3 Experiments and Measurements

In this section, we present two series of experiments which allow the quantification of the time overheads imposed by conventional gateway architectures and the benefits that can be obtained by evolved schemes (such as the one shown in Figure 2).

Firstly, we examined the behavior of a web server setup encompassing a Netscape Fast-Track server and Informix Online Dynamic Server (ver.7.2), both running on a SUN Ultra 30 workstation (processor: SUN Ultra 250 MHz, OS: Solaris 2.6) with 256 MB of RAM. In this testing platform we have evaluated the demon-based architecture of Figure 2 and the ClientRequest/ServerResponse protocol of Section 3.2, using, on the B interface, the CGI and NSAPI specifications (all tested scenarios are shown in Figure 3) [Had99a].

The IPC mechanism that we have adopted was Message Queues, member of the System V IPC family. Quite similar experiments were also performed with BSD Sockets [Had99b] but not reported in this Section. Both types of gateway instances (i.e., CGI scripts or NSAPI SAFs) as well as the server demon were programmed in the C language. The server demon, for the D interface, used the Dynamic SQL option of Embedded SQL (Informix E/SQL). Only one database demon existed in the setup. Its internal operation is shown, by means of a flowchart, in Figure 4. It is obvious, from Figure 4, that the database demon operates in a generic way, accessing any of the databases handled by the DBMS and executing any kind of SQL statement. If the, until recently, used DBMS connection (e.g., to a database or specific user account) is the same with the connection needed by the current gateway request then, that connection is being used. Lastly, we should note that incoming requests are dispatched by the demon process in an iterative way.

As shown in Figure 3, we compared the conventional (monolithic) CGI-based Informix gateway (C and Embedded SQL - Static SQL option) against the CGI ↔ Database Demon ↔ Informix scheme (Scenario 2) and the NSAPI SAF ↔ Database Demon ↔ Informix combined architecture (Scenario 3). In all three cases, the designated database access involved the execution of a complicated query over a sufficiently populated Informix table (around 50,000 rows). The table contained the access log of a web server for a period of six months. The layout of the table followed the Common Log Format found in all web servers and the row size was 196 bytes. The executed query was the following: Select the IP address and total size of transmitted bytes (grouping by the IP address) from the access log table where the HTTP status code equals 200 (Document follows). The size of the HTML page produced was 5.605 KB in all scenarios (a realistic page size, considering the published WWW statistics [Bra96], [Bar98]). The tuples extracted by the database were embedded in an HTML table (ResultsLayout = "TABLE").

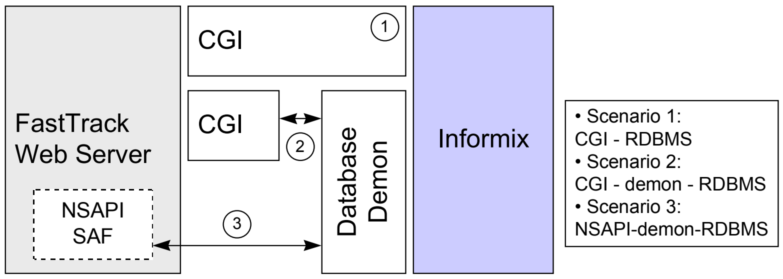
CGI
1
FastTrack Web Server
CGI
2
Database Demon
Informix
NSAPI SAF
3
• Scenario 1: CGI - RDBMS
• Scenario 2: CGI - demon - RDBMS
• Scenario 3: NSAPI-demon-RDBMS

Figure 3: Database access scenarios (1)

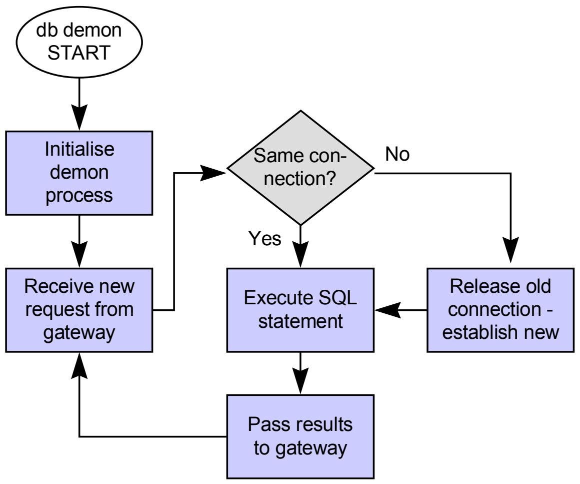

Figure 4: Flowchart for Database Demon operation

The above described experiments were realized by means of an HTTP pinger, a simple form of the benchmark software discussed in Section 5. The pinger program was configured to emulate the traffic caused by up to 16 HTTP clients. In each workload level (i.e., number of simultaneous HTTP clients), 100 repetitions of the same request were directed, by the same thread of the benchmark software, to the WWW server. The recorded statistics included:

- Connect Time (ms): the time required for establishing a connection to the server.
- Response Time (ms): the time required to complete the data transfer once the connection has been established.
- Connect rate (connections/sec): the average sustained throughput of the server.
- Total duration (sec): total duration of the experiment.

The pinger program executed on a MS-Windows NT Server (version 4) hosted by a Pentium II 300 MHz machine with 256 MB of RAM and a PCI Ethernet adapter. Both machines (i.e., the pinger workstation as well as the web/database server) were interconnected by a 10Mbps LAN and were isolated by any other computer to avoid additional traffic which could endanger the reliability of the experiments.

From the gathered statistics, we plot, in Figure 5, the Average Response Time per request. The scatter plot of Figure 5 is also enriched with polynomial fits.

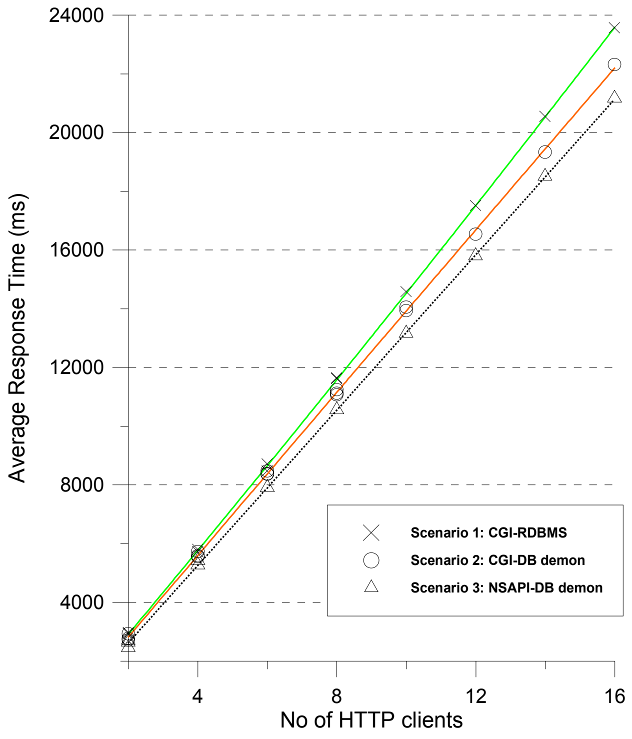

Figure 5: Response Time Vs Number of Clients

Figure 5 shows that the CGI ↔ demon architecture (Scenario 2) performs systematically better than the monolithic CGI gateway (Scenario 1) and worst than the NSAPI ↔ Demon configuration (Scenario 3), irrespective of the number of HTTP clients (i.e., threads of the pinger program). The performance gap of the three solutions increases proportionally to the number of clients. Figure 5 also suggests that for relatively small/medium workload (i.e., 16 simultaneous users) the serialization of ClientRequests in Scenarios 2 and 3 (i.e., each ClientRequest incurs a queuing delay due to the iterative nature of the single database demon) doesn't undermine the performance of the technical option.

A number of additional tests were performed in order to cover even more specifications and gateway architectures. Such tests were performed in the Oracle RDBMS (ver. 7.3.4) running on a Windows NT Server operating system (version 4). The web server setup included Microsoft's Internet Information Server (IIS) as well as Netscape's Fasttrack Server (not operating simultaneously). Both the web servers and RDBMS were hosted by a Pentium 133 HP machine with 64MB of RAM. In this setup we employed, on the B interface (Figure 2), the CGI, NSAPI, ISAPI and Servlet specifications, already discussed in previous paragraphs. On the D interface we made use of Embedded SQL (both Static and Dynamic options), ODBC and JDBC. Apart from conventional, monolithic solutions, we have also evaluated the enhanced, demon - based architecture of Figure 2. All the access scenarios that were subject to evaluation are shown in Figure 6.

The IPC mechanism that we adopted for the demon - based architecture was Named Pipes. All modules were programmed in the C language and compiled using Microsoft's Visual C. Similarly to the previous set of experiments, only one database demon (Dynamic SQL) existed in this setup. The flowchart of its internal operation is identical to the one provided in Figure 4. The database schema as well as the executed query were also identical to the previous series of experiments.

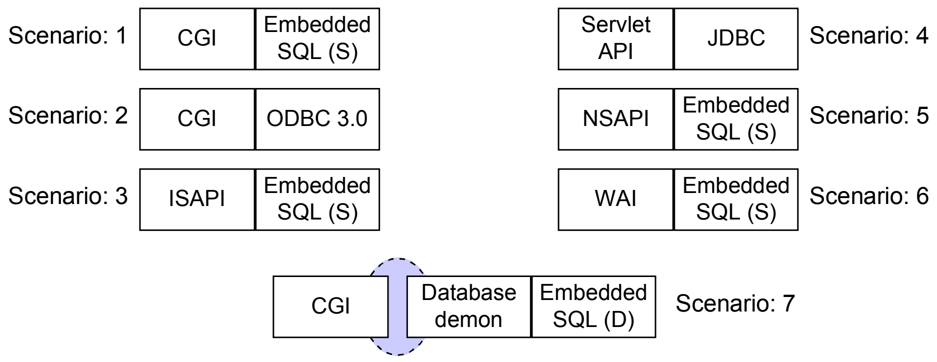

Figure 6: Database access scenarios (2)

In this second family of experiments we employed the same HTTP pinger application with the previously discussed trials. It executed on the same workstation hosting the two web servers as well as the RDBMS. The pinger was configured to emulate the traffic caused by a single HTTP user. Each experiment consisted of 10 repetitions of the same request transmitted towards the web server over the TCP loop-back interface. As in the previous case, Connect Time, Response Time, Connect Rate and Total Duration were the statistics recorded. Apart from those statistics, the breakdown of the execution time of each gateway instance was also logged. This was accomplished by enriching the code of gateway instances with invocations of the C `clock()` function which returns the CPU time consumed by the calling process. Thus, we were able to quantify the time needed for establishing a connection to the RDBMS (to be referred to as Tcon) as well as the time needed to retrieve the query results (to be referred to as Tret). Such logging of CPU times was performed in scenarios:

- 1 (CGI/Embedded SQL),
- 2 (CGI/ODBC),
- 4 (Servlet/JDBC) [5] and
- 7 (CGI ↔ Database Demon/Dynamic SQL).

We restricted the time breakdown logging in those scenarios since they involve different database access technologies.

Figure 7 plots the Average Response Time for each scenario.

[5] Gefion's WAICoolRunner was used as a Servlet engine in this scenario.

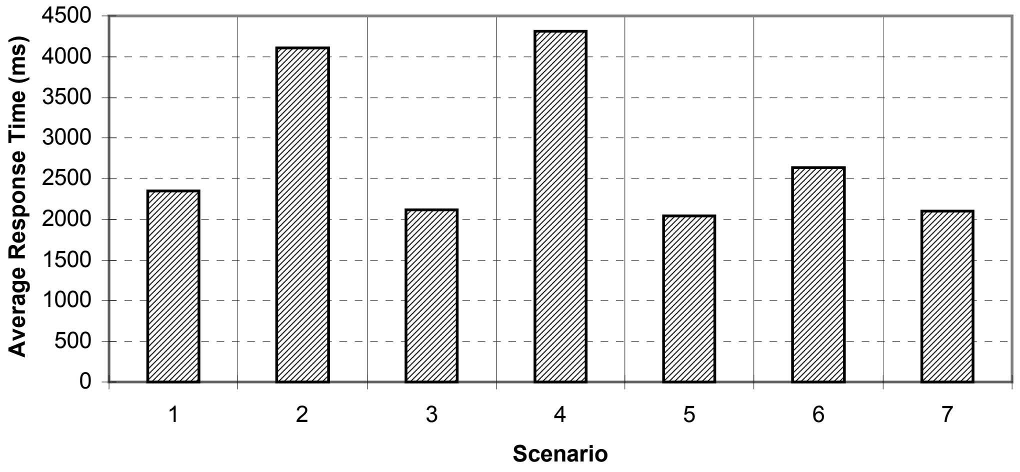

Figure 7: Response Times for Scenarios 1-7

Figure 7 shows that the CGI ↔ Demon accomplishes quite similar times to those of the NSAPI and ISAPI gateways. In the multi-threaded gateway specifications (i.e., NSAPI, ISAPI and WAI), connections towards the RDBMS are established by the executing thread (hence, the associated cost is taken into account). Connections by multi-threaded applications are only possible if mechanisms like Oracle's Runtime Contexts [Ora96] or Informix's dormant connections [Inf96] are used. Other multi-threaded configurations are also feasible: a database connection (and the associated runtime context) could be pre-established (e.g., by the DLL or the WAI application) and shared among the various threads, but such configurations necessitate the use of synchronization objects like mutexes. In such scenarios a better performance is achieved at the expense of the generality of the gateway instance (i.e., only the initially opened connection may be re-used).

In Scenario 2, ODBC Connection Pooling was performed by the ODBC Driver Manager (ODBC 3.0) thus, reducing the time overhead associated with connection establishments after the initial request. In Scenario 4, the JDBC drivers shipped with Oracle 7.3.4 were used. Specifically, we employed Type 2 drivers.

In Figure 8 we show where the factors Tcon and Tret range.

Error! Not a valid link.

Figure 8: Time breakdown for Scenarios 1, 2, 4 and 7

As shown in Figure 8, in the ODBC and JDBC scenarios (i.e., Scenarios 2 and 4) a very important percentage of the execution time of the gateway is consumed for the establishment of connections towards the DBMS. The Embedded SQL scenario (i.e., Scenario 1) achieves the lowest Tcon. The highest Tret is incurred at Scenario 7 where query execution is fully dynamic (see Section 3.1 for the Dynamic SQL option of Embedded SQL). Tcon is not plotted in Figure 8 for Scenario 7, as this cost is only incurred once, in our experiments, by the database demon.

3.4 State Management Issues

In general, stateful web application impose considerable additional overhead. Extra information is carried across the network to help identify individual sessions and logically correlate user interactions. The Cookies mechanism [Kri97] specify additional fields in the HTTP headers to accommodate state information. Complicated applications, though, may require a large volume of Cookie information to be transmitted to and from the browser. At the server's side, additional processing - handling is required in order to perform state management. In [Had98] a framework for the adaptation of stateful applications to the stateless Web has been proposed. This framework involved the placement of an extra software layer in the browser-database chain. Such additional, server-side, tier would associate incoming Cookie values with the appropriate threads of a multithreaded database application, pass execution information, relay results and assign new Cookie values to the browser in order to preserve a correct sequence of operations. An additional entity responsible for handling state information is also required in the architecture presented in [Iye97]. Considerations concerning the overheads introduced by state management techniques are also discussed in [PJu97].

3.5 Persistent or Non-persistent Connections to Databases

The database connection persistence problem is mentioned in [PJu97]. A conventional scheme, with monolithic CGIs establishing connections directly towards the RDBMS, may exhaust the available licenses due to the sporadic nature of WWW requests. Additionally, as discussed in previous paragraphs, a substantial time overhead is to be incurred in this scheme. Maintaining a database connection per active session (e.g., through a server API or FastCGI) may not be a sound strategy as the maximum licenses limit may also be easily reached while extensive periods of client inactivity cause a terrible waste in resources. In these two scenarios, uncontrolled connection establishment with the RDBMS is definitely a problem. Persistent database connections may, however, be beneficial for response times as state management is simplified [Had98].

A demon based architecture, on the other hand, drastically restricts the number of connections simultaneously established towards the RDBMS (controlled rate of connection establishment). Multiple demons may be established to serve incoming requests in a more efficient way (so as to reduce potential queuing delays caused by the availability of a single dispatching entity [Had99a]). In such scenario a regulating entity will be needed to route requests to idle demons and handle results on the reverse direction (Figure 9).

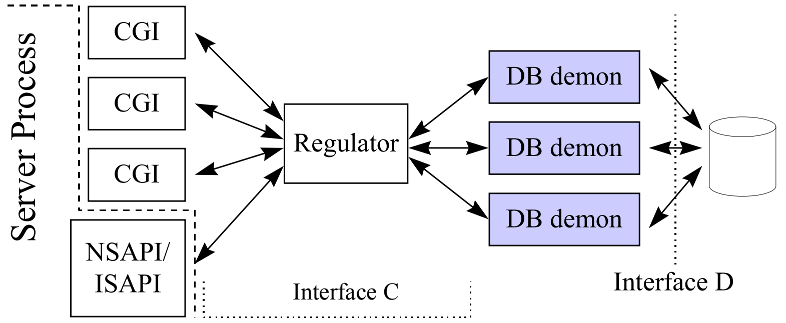

Figure 9: Generalized demon-based architecture

3.6 Template - based Middleware

Software tools intended for building database-powered WWW sites usually follow a template approach. A generic engine processes pre-programmed templates (in an interpreter-like way) and performs database interactions as needed (run-time binding of reusable code, eg. some ODBC interface, with a specializing template). Templates practically are the combination of HTML with a proprietary tag set (dealing with query specification, transactions, result set handling, flow control, etc.). An indicative example of a template-based middleware is Allaire's Cold Fusion with its Cold Fusion Markup Language (CFML)[6]. Surely, the interpreter like approach dictated by the adoption of templates causes a slow down in the dispatch of queries. On the other hand, such tools are extremely efficient programming tools which drastically reduced the time required to develop database gateways in the old-fashioned way using techniques such as Embedded SQL.

[6] Formerly known as DBML - DataBase Markup Language

4. WEB SERVER ARCHITECTURES

The architecture of HTTP demons (or WWW servers) have been a very important issue since the advent of the WWW. The concern of the members of the WWW community about the implications of WWW server architecture on performance is clear in works like [McG95] and [Yea96]. The very first servers (e.g., NCSA httpd 1.3 and CERN 3.0) were designed to fork of a new server process for each incoming request (i.e., the server clones itself upon arrival of a new request). Newer server architectures adopt the so-called pre-forking (or pool of processes) approach. The pre-forking approach involves the provisional generation, by the master server process, of a set of slave processes (the number is customizable). Such processes are awaiting idle until an HTTP request reaches the master process. Then, the master process accepts the connection and passes the file descriptor to one of the pre-spawned clients. If the number of simultaneous requests at the server exceeds the number of pre-spawned clients, then the master httpd process starts forking new instances (similarly to the older techniques). Measurements reported in [McG95] show that this technique practically doubles the performance of the WWW server. Such measurements were based on one of the very first benchmarking tools for WWW servers, the NCSA pinger.

Newer server architectures were designed in the more efficient, multi-threaded paradigm. Notable examples are Netscape's FastTrack/Enterprise servers. In those architectures, upon WWW subsystems' boot, only one server process is spawned. Within this server process the pool of available instances principle is still followed but on the thread level. In other words, a number of threads are pre-spawned and left idle, awaiting incoming requests to serve. If the threads in the thread pool are all in use, the server can create additional threads within the same process/address space to handle pending connections. It is suggested that the number of pre-spawned server processes should be greater to one only in those cases where multi-processor hardware is used. Servers are optimized on the thread model supported by the underlying OS (e.g. in the HP-UX case where not native threading is supported a user-level package is provided while in the Solaris case the many-to-many model is adopted).

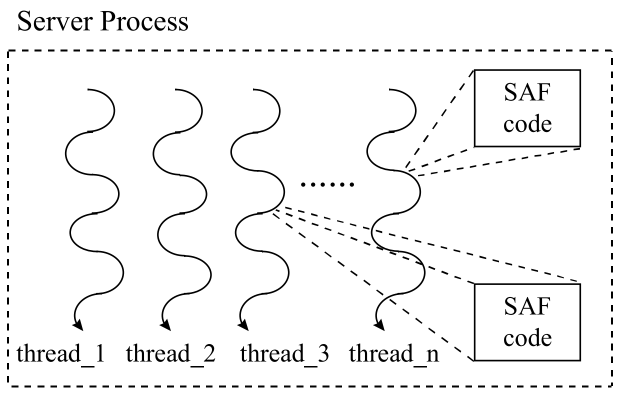

Figure 10: Multi-threaded Netscape Server architecture

As discussed in Section 2.2, gateways can be build in Netscape servers using the NSAPI specification. Taking into account the server's multi-threaded architecture, the performance benefit is two-fold: (a) SAF/plug-in code is pre-loaded in the memory space of the server process and (b) the server spawns autonomous threads instead of forking processes for each incoming request (Figure 10). Quite similar behavior is exhibited by Microsoft servers.

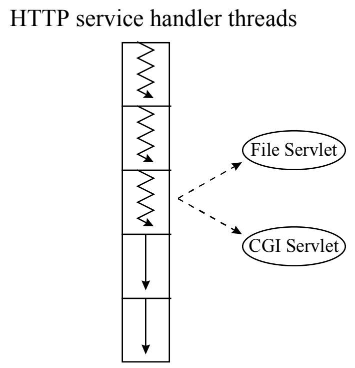

Figure 11: HTTP connection handling in JavaServer

The Java Server (http://jserv.javasoft.com/products/java-server) also adopts the multi-threaded architecture discussed above, but relies heavily on the Servlet technology (see Section 2.6). A pool of service handler threads (e.g., HTTP handler threads) is provided and bound to a specific ServerSocket. Such handler threads, when associated with an incoming request, know how to dispatch it to HTTP Servlets (after authorization and name translation). Built-in HTTP Servlets are core Servlets that the HTTP service handler threads use to provide standard functionality. Indicative examples are the FileServlet which is used to respond to simple file requests and the CGIServlet to provide basic CGI functionality (Figure 11). A meta-Servlet, called InvokerServlet, is used for loading, invoking and shutting down Servlets which were explicitly invoked by the client (i.e., using the URL scheme).

5. PERFORMANCE EVALUATION TOOLS

During the past years a number of tools have emerged for the performance evaluation of web server systems. Some of these tools take the reported Web traffic models into account (analytic workload generation). The vast majority of tools, however, follows the so-called trace-driven approach for workload generation. The trace-driven approach, practically, tries to imitate pre-recorded traffic logs either by sampling or by re-executing traces.

5.1 Analytic Workload Generation

Traffic modeling for the WWW has been studied in considerable detail in [Cro98] where a self-similar nature is suggested. Self-similarity is attributed to the multiplexing of a large number of ON/OFF sources (which in the case of the WWW are interacting users). The period spent by the sources in the ON or the OFF states is following heavy-tailed distributions. In particular, ON times (times corresponding to resource transmissions) are best modeled through the Pareto distribution. OFF times are classified either as Active (attributed to the processing time of the Web client, rendering time, etc.), or as Inactive (attributed to user think time). The former are much more extended than the latter. Different distributions seem suitable for those two time components. The Weibull distribution is best suited for Active times [Bar98] while the Pareto distribution best approximates the period of Inactivity. Such time distributions are shown in Figure 12.

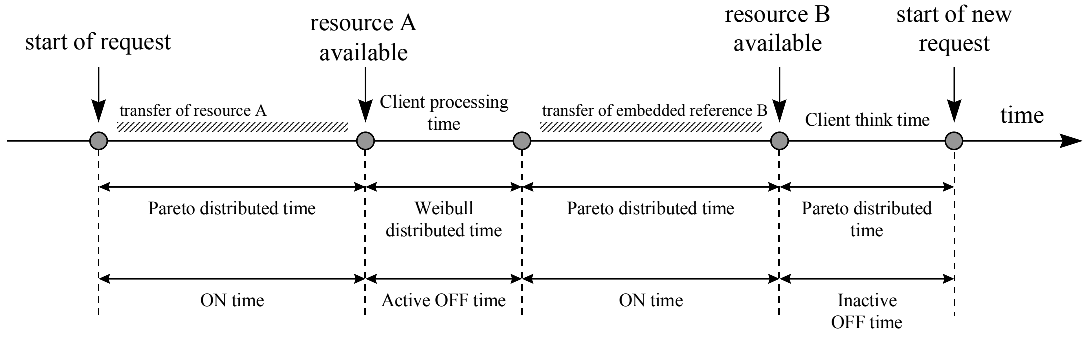

Figure 12: Time distributions for WWW traffic

The SURGE (Scaleable URL Reference Generator) tool, presented in [Bar98], is a workload generator which is compliant with the reported traffic models for the WWW. Specifically, SURGE follows the Web models for file sizes, request sizes, popularity, embedded references, temporal locality and OFF times. SURGE manages to put much more stress on the CPU of the measured system than conventional benchmarks (eg., SPECWeb96 - see Section 5.2.2) do. Additionally, under conditions of high workload, SURGE achieves the generation of self-similar network traffic which is not the case with other tools.

5.2 Trace-driven Workload Generation

An impressively high number of simpler tools for the performance evaluation of WWW server systems have appeared during the past years (http://www.charm.net/~dmg/qatest/qatweb.html). In the following paragraphs we briefly describe the most popular benchmark tools currently used by the WWW community.

5.2.1 WebStone

WebStone [Tre95] is a WWW server benchmark - performance evaluation tool originally developed by Silicon Graphics Inc. (SGI). Currently, the tool is being progressed by Mindcraft Inc. (http://www.mindcraft.com) which acquired the WebStone rights from SGI.

WebStone allows the measurement of a series of variables namely the average and maximum connect time, the average and maximum response time, the data throughput rate and, lastly, the number of pages and files retrieved. The benchmark software is executed simultaneously in one or more clients positioned in the same network with the measured server. Each client is capable of spawning a number of processes, named “WebChildren”, depending on how the system load has been configured. Each WebChild requests information from the server based on a given configuration file. WebStone treats the server as a black box (black box testing). The execution of WebChildren is coordinated by a process called WebMaster. The WebMaster distributes the WebChildren software and test configuration files to the clients. Then, it starts a benchmark run and waits for the WebChildren to report back the performance they measured. Such information is combined in a single report by the WebMaster. WebStone has been extended to allow the performance evaluation of CGI programs and server API modules (in WebStone 2.5 both ISAPI and NSAPI are supported). The performance measured in largely dependent on the configuration files used by WebChildren. Although such files can be modified, using the standard WebStone file enables the comparison between different benchmark reports. This standard file, practically, focuses on the performance of the Web server, operating system, network and CPU speed.

WebStone configuration files determine basic parameters for the testing procedure (e.g., duration, number of clients, number of WebChildren, etc.) as well as the workload mix which should be used. Each workload mix corresponds to a specific set of resources that should be requested, in a random sequence, from the measured server. Mixes may contain different sizes of graphic files or other multimedia resources (e.g., audio, movies). As discussed in the previous paragraph, WebStone mixes were recently enriched to invoke CGI scripts or ISAPI/NSAPI compliant modules.

5.2.2 SPECWeb96

SPECWeb96 [SPE96] was released from the Standard Performance Evaluation Committee (SPEC) in 1996. The benchmark tool is quite similar to WebStone; it implements black box server testing (also termed SUT - System Under Test). Workload is generated by one or more client processes (“drivers”) running on one or more workstations. Similarly to WebStone, SPECWeb96 can be configured to generate different workloads. To provide a common indication of server efficiency, SPECWeb96 introduced the SPECWebOps/sec metric calculated as the number of successfully completed requests divided by the duration of the test.

SPECWeb’s advantage over WebStone is that the former is a standardized benchmark, agreed among a number of manufacturers, while the latter represents an individual effort which has, however, attracted the interest of the WWW community. In general, the abundance of Web benchmarks renders the comparison of measurements quite difficult and unreliable. This problem is addressed by SPECWeb96, which, though, is not as sophisticated as other tools are.

Although both the WebStone and SPECWeb96 report the number of HTTP operations per second, their measurements are not comparable since their generated workload is quite different. Due to its larger file-set sizes and its pattern of access, SPECweb96 gives emphasis on a system's memory and I/O subsystem.

The workload mix in SPECWeb comprises files organized in four classes: files less than 1 KB (35% of all requests), files in the range 1 KB - 10 KB (50% of requests), files between 10 KB and 100 KB (14%) and, finally, larger files (only 1% of total requests). Within each class there are 9 discrete sizes. Accesses within a class are not uniformly distributed. Instead, a Poisson distribution is being used within the limits of the file class. Files are retrieved from a series of directories with a specific structure; this scheme tries to emulate real-world conditions. The contents of the directories are compatible with the file side distributions discussed above while the number of directories varies according to the maximum load that is to be reached.

SPECWeb is confined only to the GET HTTP method. SPECWeb cannot handle other types of requests like POST. Additional, CGI issues are not addressed by the benchmark.

According to [Ban97], both the WebStone and SPECWeb96 tools are not capable of generating client request rates that exceed the capacity of the server being tested or even emulate conditions of bursty traffic. This is mainly due to certain peculiarities of the Transport Control Protocol (TCP) which the HTTP uses as a reliable network service. To tackle the aforementioned problem, the same paper proposes a quite different Socket connection handling by the benchmark software (called S-Client for scaleable client). Measurements presented in the same paper show that S-Clients manage to really put the server software under stress.

Other tools in the Web benchmarking area include: Microsoft's InetLoad (http://www.microsoft.com/msdownload/inetload/inetload.html), ZDNet's WebBench (http://www.zdnet.com/zdbop/webbench/webbench.html).

6. EPILOGUE

Database connectivity is surely one of the most important issues in the constantly progressing area of WWW software. More and more companies and organizations are using the WWW platform for exposing their legacy data to the Internet. On the other hand, the amazing growth of WWW content forces the adoption of technologies like RDBMS for the systematic storage and retrieval of such information. Efficiency in the mechanisms for bridging the WWW and RDBMS is a very crucial topic. Its importance stems from the stateless character of the WWW computing paradigm which necessitates a high frequency of short-lived connections towards the database management systems. In this chapter, we have addressed a series of issues associated with the considered area of WWW technology. We have evaluated different schemes for database gateways (involving different gateway specifications and different types of database middleware). Although aspects like generality, compliance to standards, state management and portability are extremely important their pursuit may compromise the performance of the database gateway. The accumulated experience from the use and development of database gateways over the last 5-6 years suggests the use of architectures like the database demon scheme, which try to meet all the above mentioned requirements to a certain extend but also exhibit performance close to server APIs.

ACKNOWLEDGMENTS

We are indebted to Mr. J. Varouxis for extremely useful discussions and technical assistance on the preparation of the experiments presented in this chapter.

REFERENCES

[All96] "COLD FUSION User's Guide Ver. 1.5", Allaire Corp. 1996.

[Ban97] G. Banga, and P. Druschel, "Measuring the Capacity of a Web Server", proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, December 1997.

[Bar98] P. Barford, and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", proceedings of ACM SIGMETRICS, International Conference on Measurement and Modeling of Computer Systems, July, 1998.

[Ber98] P. Bernstein et al., "The Asilomar Report on Database Research", ACM SIGMOD Record, Vol. 27, No. 4, Dec. 1998.

[Bra96] T. Bray, "Measuring the Web", Computer Networks and ISDN Systems, Vol. 28, No. 7-11, 1996.

[Bro96a] M. Brown, "FastCGI Specification", Open Market Inc., http://fastcgi.idle.com/kit/doc/fcgi-spec.html, April 1996.

[Bro96b] M. Brown, "FastCGI: A High Performance Gateway Interface", Position paper for the workshop "Programming the Web - a search for APIs", 5th International WWW Conference, Paris, France, 1996.

[Bro96c] M. Brown, "Understanding FastCGI Application Performance", Open Market Inc., http://fastcgi.idle.com/kit/doc/fcgi-perf.html, June 1996.

[Cha97] P.I. Chang, "Inside the Java Web Server: An Overview of Java Web Server 1.0, Java Servlets, and the JavaServer Architecture", http://java.sun.com/features/1997/aug/jws1.htm, 1997.

[Coa98] K. Coar, and D. Robinson, "The WWW Common Gateway Interface - Version 1.2", Internet Draft, February, 1998.

[Cro98] M. Crovella, M. Taqqu, and A. Bestavros, "Heavy-Tailed Probability Distributions in the World Wide Web", in "A Practical Guide to Heavy Tails - Statistical Techniques and Applications", R. Adler, R. Feldman, and M. Taqqu (ed.), BIRKHAUSER, 1998.

[Eve96] P. Everitt, "The ILU Requested: Object Services in HTTP Servers", W3C Informational Draft, March, 1996.

[Gok96] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High Speed Networks", proc. ACM SIGCOMM Conference, 1996.

[Had96] S. Hadjiefthymiades, and D. Martakos, "A Generic Framework for the Deployment of Structured Databases on the World Wide Web", Computer Networks and ISDN Systems, Vol. 28, No. 7-11, 1996.

[Had97] S. Hadjiefthymiades, and D. Martakos, "Improving the Performance of CGI compliant Database Gateways", Computer Networks and ISDN Systems, Vol. 29, No. 8-13, 1997.

[Had98] S. Hadjiefthymiades, D. Martakos, and C.Petrou, "State Management in WWW Database Applications", proceedings of IEEE Compsac '98, Vienna, Aug. 1998.

[Had99a] S. Hadjiefthymiades, D. Martakos, and I. Varouxis, "Bridging the gap between CGI and server APIs in WWW data-base gateways", Technical Report TR99-0003, University of Athens, 1999.

[Had99b] S. Hadjiefthymiades, S. Papayiannis, D. Martakos, and Ch. Metaxaki-Kossionides, "Linking the WWW and Relational Databases through Server APIs: a Distributed Approach", Technical Report TR99-0002, University of Athens, 1999.

[Hay95] "Performance Benchmark Tests of Unix Web Servers using APIs and CGIs", Haynes & Company - Shiloh Consulting, http://www.tedhaynes.com/haynes1/bench.html, November 1995.

[Inf96] "Informix-ESQL/C Programmer's Manual", Informix Software Inc., 1996.

[ISO95] IS 9075-3, "International Standard for Database Language SQL - Part 3: Call Level Interface", ISO/IEC 9075-3:1995.

[Iye97] A. Iyengar, "Dynamic Argument Embedding: Preserving State on the World Wide Web", IEEE Internet Computing, March-April 1997.

[JDB97] "JDBC Guide: Getting Started", Sun Microsystems Inc., 1997.

[Kha97] R.Khare, and I.Jacobs, "W3C Recommendations Reduce 'World Wide Wait'", World Wide Web Consortium, http://www.w3.org/pub/WWW/Protocols/NL-PerfNote.html, 1997.

[Kri97] D.Kristol, and L.Montuli, "HTTP State Management Mecha-nism", RFC 2109, Network Working Group, 1997.

[Lau95] J. Laurel, "dbWeb White Paper", Aspect Software Engi-neering Inc., August, 1995.

[Maz98] S. Mazzocchi and P. Fumagalli, "Advanced Apache Jserv Techniques", proceedings of ApacheCon '98, San Francisco, CA, October 1998.

[McG95] R. McGrath, "Performance of Several HTTP Demons on an HP 735 Workstation", http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html, April, 1995.

[Mic97a] "Internet Server API (ISAPI) Extensions", MSDN Library, MS-Visual Studio '97, Microsoft Corporation, 1997.

[Mic97b] "ODBC 3.0 Programmer's Reference", Microsoft Corpora-tion, 1997.

[Nan97] B. Nance, "Examining the Network Performance of JDBC", Network Computing Online, May, 1997.

[Net97] "Writing Web Applications with WAI - Netscape Enterprise Server/FastTrack Server", Netscape Communications Co., 1997.

[Nom95] "User's Guide, WebDBC Version 1.0 for Windows NT", Nomad Development Co., 1995.

[OMG97] "CORBA: Architecture and Specification", Object Manage-ment Group, 1997.

[Ora96] "Programmer's Guide to the Oracle Pro*C/C++ Precom-piler", Oracle Co., February 1996.

[Orf98] R. Orfali and D. Harkey, "Client/Server Programming with JAVA and CORBA", Wiley, 1998.

[PJu97] P. Ju, and Pencom Web Works, "Databases on the Web - De-signing and Programming for Network Access", M&T Books, 1997.

[SPE96] "An Explanation of the SPECweb96 Benchmark", Standard Performance Evaluation Corporation, http://www.spec.org/osg/web96/webpaper.html, 1996.

[Ste90] W.R. Stevens, "UNIX Network Programming", Prentice Hall, 1990.

[Tra96] M. Tracy, "Professional Visual C++ ISAPI Programming", Wrox Press, 1996.

[Tre95] G. Trent, and M. Sake, "WebSTONE: The First Generation in HTTP Server Benchmarking", Silicon Graphics Inc., February 1995.

[XOp94] "Data Management: SQL Call-Level Interface (CLI)", X/Open CAE Specification, 1994.

[Yea96] N. Yeager, and R. McGrath, "Web Server Technology - The Advanced Guide for World Wide Web Information Provides", Morgan Kaufmann Publishers, 1996.