# SIP Server Implementation and Performance on a Bare PC

A. Alexander, R. Yasinovskyy, A. L. Wijesinha, and R. Karne
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
USA

***Abstract*—We describe the implementation and performance of a bare PC SIP server that runs without the support of an operating system (OS) or kernel. A bare PC SIP server provides immunity against OS vulnerabilities and yields performance gains due to the elimination of OS overhead. We discuss server design focusing on its novel architectural features and illustrate key implementation aspects by examining relevant task and method invocations for SIP request processing. We also study bare PC SIP server performance by comparing its latency and throughput against two conventional OS-based SIP servers running on equivalent hardware: OpenSER on Linux and Brekeke on Windows. Furthermore, we measure internal bare PC SIP server performance by providing internal timings for the most significant operations associated with registration and proxy services. Additionally, we study performance under increasing server load by obtaining the execution time spent in the bare PC SIP handler method and the total processing time including network protocol processing overhead when processing SIP requests and responses. The results show that the bare PC server performs better than the OS-based servers in most cases and that its internal processing times are small as would be expected due to the elimination of OS overhead. The design and implementation details of the bare PC SIP server presented here give insight into understanding SIP server performance on a bare machine.**

***Keywords*-SIP server, implementation, performance, internal timings, bare machine computing, operating systems.**

## I. INTRODUCTION

Bare PC or bare machine systems run on the hardware without the need for an operating system (OS) or kernel. In [1], the performance of a bare PC SIP server running on an ordinary desktop was studied. It was shown that in most cases the server performs better than two conventional (OS-based) SIP servers running on identical non-server machines. This paper is an extended version of [1] that gives details underlying the implementation of the bare PC SIP server and new internal timings for key server operations. Material relevant to server design and implementation is taken from [2]. However, it is supplemented with details concerning tasking and method invocations that were omitted in [2]. Additionally, the new internal timings under increasing server load given here are more accurate than the approximate timings initially reported in [1]. The implementation specifics and new timing results provide further insight into bare PC SIP server operation and performance.

Previous studies on bare PC email and Web servers show that they significantly outperform their counterparts running on conventional OS-based systems [3] [4]. Bare PC applications and servers also have inherent immunity to attacks that target vulnerabilities of a given OS. Many studies have dealt with the design and performance of network and security protocols in a bare PC. For example, the performance of SRTP for bare PC VoIP is evaluated in [5], and peer-to-peer communication among bare PC VoIP clients is discussed in [6]. However, there have been no studies of SIP (Session Initiation Protocol) on a bare PC. SIP is the most frequently used protocol today for initiating VoIP calls and for media session support with a variety of other applications including video streaming, instant messaging, gaming, and IPTV. For example, most SIP servers can provide voice, video, instant messaging, and presence services.

In general, SIP servers locate and register clients, provide proxy services for forwarding SIP messages, or redirect SIP requests to other servers. An optimized SIP server can thus help improve the overall performance of audio or video applications by supporting audio or multimedia sessions (although it is typically not directly involved in the actual transmission of audio or video). The throughput and latency of the SIP server when responding to requests from SIP user agent clients and other SIP servers are often used as measures in evaluating its performance.

We use a popular open source SIP workload generator to evaluate the performance of the bare PC SIP server by measuring its throughput and latency for registration, proxying, and redirection, with and without authentication, for increasing workloads. We compare performance of the bare PC server with popular OS-based (Linux and Windows) servers for the same workloads when running on compatible hardware. Our results show that the bare PC SIP server has higher or equal throughput to the Linux server and higher throughput than the Windows server, except in case of redirection, when its throughput is less than that of the Linux server. The latency performance of the bare PC server is also shown in general to be better than or equal to that of Linux server and better than that of the Windows server, except for invite with authentication and invite-not-found without authentication. We also provide internal timings measured on the bare PC SIP server.

The performance results of the server are better understood by examining its design and implementation details. To this end, we describe the bare PC SIP server components within the self-supporting application object (AO) that runs directly on the PC hardware. In particular, we examine SIP packet processing for requests and their responses. In addition, tasking is discussed and examples of method invocations are given to highlight protocol intertwining and other novel implementation characteristics in the bare PC SIP server.

Our contributions in this paper include: 1) results characterizing the performance of a bare PC SIP server running on an ordinary desktop; 2) internal timings for SIP-related operations on a bare PC SIP server; 3) comparisons of the throughput and latency for a bare PC SIP server, Linux and Windows servers running on identical machines; and 4) design and implementation details of the bare PC SIP server.

The rest of this paper is organized as follows. In Section II, we summarize related work. In Section III, we describe the design of the bare PC SIP server and relevant optimizations. In Section IV, we provide implementation details of the server. In Section V, we give the experimental setup and discuss the results of the performance study. In Section V, we present the conclusion.

## II. Related Work

There are many commercial and open source servers implementing SIP and its companion protocol SDP. While a SIP server usually runs over UDP and in some cases over TCP, the use of SCTP as a transport protocol for SIP has also been studied [7]. An early study on SIP server performance [8] found that the overhead on a Java SIP server due to security mechanisms such as authentication and TLS was negligible. However, the study in [9], which measured throughput and latency in a dedicated gigabit Ethernet for stateless and stateful proxies over UDP and TCP, showed that authentication, TCP, or the operation/server configuration can significantly change SIP server performance. Their experiments were conducted using a 3.06 GHz server class machine, and only the performance of a single SIP server (OpenSER on Linux) was evaluated. In [10], SIP server performance for several stateful SIP proxies over UDP was evaluated. The authors concluded that the overhead due to string processing operations and memory management could consume significant processing time and that performance varied considerably depending on the proxy. Recent work on SIP servers has dealt with performance under overload conditions [11], scalability issues [12] [13], load balancing [14], and the impact of transport protocols on performance [15].

The main difference between previous performance studies and the performance studies in this paper is that we study the performance of a bare PC SIP server and compare it with the performance of two OS-based SIP servers using ordinary desktop (non-server) machines. Also, in addition to evaluating performance for the usual register, invite, and redirect operations, we also evaluate SIP server performance for the register update, register logout, and invite-not-found operations likely to be encountered in practice. Internal timings for key operations measured on the bare PC SIP server are also reported. We only consider SIP over UDP with stateless proxying, which is the most common configuration when setting up VoIP calls.

Previous work describing the design and implementation of SIP servers that require an OS. For example, in [16], a SIP server is implemented on top of an existing SIP stack, and in [17], SIP servers are implemented on the Solaris 8 OS. These studies focus primarily on the high-level SIP implementation on a conventional system, whereas the design and implementation of the bare PC SIP server is based on the underlying bare computing paradigm and architecture.

## III. Design

This section describes key design details of the bare PC SIP server. We begin by briefly outlining bare application characteristics in general and then give an overview of the bare PC server.

### *A. Bare PC Applications*

Any bare PC or bare machine computing application, including the bare PC SIP server, is encapsulated in an application object (AO) [18]. Since there is no OS, minimal code for the application to directly run on the PC hardware is contained in the AO. This means that the AO contains the code for the bootable self-executing application itself, any required network interface drivers, handlers for protocols used by the application, and memory and task management mechanisms to facilitate concurrency and scheduling [19]. Real memory is used since there is no hard disk, and application code is intertwined with protocol code to eliminate redundancy and improve efficiency as in the case of bare PC Web and email servers [4] [20].

### *B. SIP Server Overview*

The bare PC SIP server AO implements a lean version of SIP that provides essential functionality only. Additional features such as those needed to support load balancing and media stream security are not included. Although a bare PC SIP server that can operate over TCP or UDP has been implemented, this paper only considers SIP over UDP since the majority of SIP servers employed in practice use UDP.

The SIP server AO consists of several objects. In addition to the Ethernet, IP, UDP, and SIP objects, the DHCP and trivial FTP (TFTP) objects provide lean implementations of these protocols and are used as needed (for example, at server start-up) as described in the next section. An MD5 object is used to provide support for user authentication via standard SIP authentication (i.e., HTTP-Authentication) when authentication is enabled for registration and proxying.

An incoming UDP packet containing a SIP message is placed in the Ethernet buffer, where the bare PC SIP application can directly access it i.e., real mode is used and there is no notion of user space or kernel space since there is no OS. The Ethernet handler processes the packet, determines that the packet is for IP, and the IP handler in turn processes the packet and invokes the UDP handler, which verifies the UDP checksum (if this feature is enabled) and the port number. In case of the SIP server, the port number is 5060 and the packet is finally processed by the SIP handler. To send a response, the SIP, the UDP, IP and Ethernet handlers add the respective headers before the packet is transmitted by the network interface hardware. Data copying is minimized in the bare PC SIP server since there is a single copy of the message and headers are added and removed as in a conventional system by manipulating pointers.

In addition to the usual Main and Receive (Rcv) CPU tasks, which are used in all bare PC systems, the bare PC SIP server has a SIP task to handle each SIP request. This task design strategy simplifies task management, minimizes context (task) switching, and increases efficiency. The Main task runs upon start-up and whenever the Rcv task or a SIP task terminates. It activates the Rcv task whenever a packet arrives in the Ethernet buffer. It also activates a SIP task for processing after it is determined that the packet is for SIP. This SIP task runs until SIP processing is complete and the response is sent. Once the SIP task terminates, the Main task runs again. Thus, when the SIP Server AO's Rcv task is activated by the Main task upon the arrival of a SIP request in the Ethernet buffer, a single thread of execution handles the request all the way from the Ethernet level through the IP and UDP handlers. Then the SIP task runs as described above. This simple task design approach reduces the processing overhead.

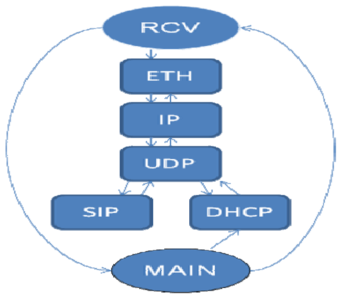


Figure 1. SIP server protocol/task relationships.

As described in [2] and shown in Figure 1, it is possible to use only two CPU tasks in the SIP server AO: a receive (Rcv) task that processes a received packet all the way from its arrival in the Ethernet buffer until a response is sent, and a Main task that runs whenever a Rcv task completes (and also when the system is booted or the system is idle). For example, for a register message, the Rcv task itself could manage the lookup and update operations and send the response to the client. However, it is more convenient and efficient (as in the present version of the SIP server) to use a separate SIP task for each request as discussed above. In case of the invite message for example, a new SIP task is activated to handle the request. Since there may be a delay in contacting the peer (callee), the SIP task could be suspended and resumed when the response arrives. In general, since a typical workload involves a mix of requests for different services, bare PC SIP server performance is improved by the concurrent handling of requests. This strategy of allowing a CPU task to run to completion unless it has to wait for an event such as a response enables the CPU to be kept busy doing useful work. Simple task management and the disabling of timer interrupts on bare PC servers also reduce context switching (compared to conventional OS-based servers) and improve performance.

## IV. IMPLEMENTATION

The section examines the key aspects of bare PC SIP server implementation. Details of processing steps and method invocations are included to illustrate novel characteristics of the implementation. The current implementation supports registrar, redirector, and proxy modes with or without authentication. Since the bare PC SIP server implementation is lean, only specific content from an incoming SIP packet is parsed. Although the server code consists of a single monolithic executable, the implementation itself is modular allowing for updates and implementation of new features. The bare PC SIP server AO contains about 2000 lines of code.

### A. *Boot Sequence*

The bare PC SIP server is booted by directly loading its AO from a USB flash drive. The bare PC SIP Server boot sequence begins when the Main task invokes the DHCP handler to send a DHCP request for an IP address (unless the server has been preconfigured to use a specific IP address). When a response arrives, the Rcv task is activated to process it. Next, a file containing username and password combinations of authorized users is transferred from another host on the network using an adaptation of trivial FTP. As discussed later, multiple data structures to facilitate server operations such as user lookup, username and password lookup, and state lookup are then created in memory. The last step in the boot process is to display the user interface for administering the server.

### B. *User Database Lookup*

After the usernames and passwords from the file are read into memory, the bare PC SIP server runs the sipservergetdb() function to store them in the USER_DATABASE structure:

```
Struct USER_DATABASE {
char username [20];
int username_size;
int username_hash;
char Password [20];
int Password_size;
};
```

The data structures HASH_TABLE and SORTED_TABLE shown below are also used.

```
Struct HASH_TABLE {
int hash_hit;
int hash_reg_db_loc[HASH_REG_DB_SIZE];
int hash_hit_size
};
Struct SORTED_TABLE {
int hash;
int hash_link;
};
```

In essence, the hash of each username serves as an index into HASH_TABLE, which is used together with SORTED_TABLE to facilitate looking up the user in the USER_DATABASE structure, and to retrieve information when making or receiving calls, or registering a user. The HASH_TABLE structure links back to the SORTED_TABLE and USER_DATABASE structures. The details are as follows. First, the hash values are stored in a SORTED_TABLE array (which allows for efficient searching for a given hash value), and each position in the sorted array is linked to the specific HASH_TABLE array corresponding to that hash value. In turn, each position in the HASH_TABLE array corresponds to a user that hashed to that value and contains a link back to the USER_DATABASE entry for that user. The HASH_TABLE structure links the index in the USER_DATABASE structure to the hash value of the SORTED_TABLE as shown in Figure 2.

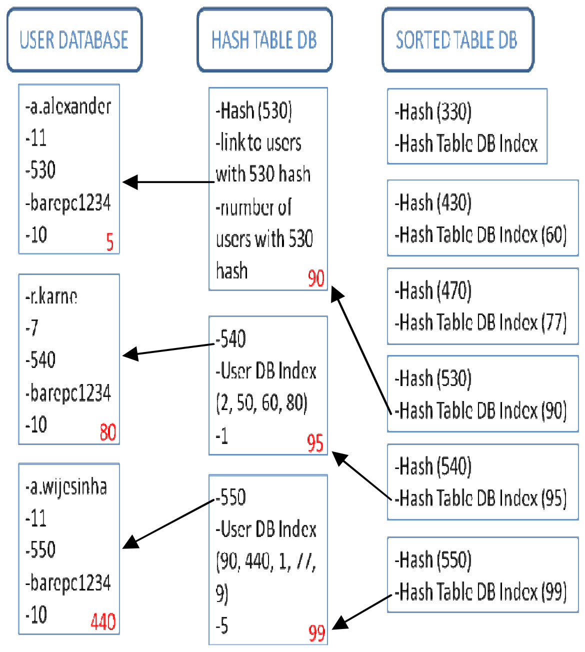

USER DATABASE
HASH TABLE DB
SORTED TABLE DB
-a.alexander -11 -530 -barepc1234 -10 5
-r.karne -7 -540 -barepc1234 -10 80
-a.wijesinha -11 -550 -barepc1234 -10 440
-Hash (530) -link to users with 530 hash -number of users with 530 hash 90
-540 -User DB Index (2, 50, 60, 80) -1 95
-550 -User DB Index (90, 440, 1, 77, 9) -5 99
-Hash (330) -Hash Table DB Index
-Hash (430) -Hash Table DB Index (60)
-Hash (470) -Hash Table DB Index (77)
-Hash (530) -Hash Table DB Index (90)
-Hash (540) -Hash Table DB Index (95)
-Hash (550) -Hash Table DB Index (99)

Figure 2. Database and hash table relationships.

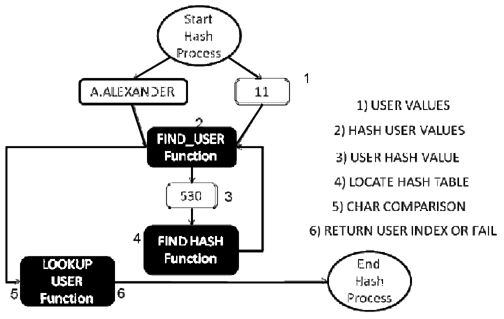


Figure 3. User lookup process.

The user lookup process in Figure 3 is done by using two functions: the find_hash_hit() function, which is based on a particular hash value, and the find_user() function that is based on the username and size. In performance tests, this search operation was found to be a likely bottleneck because of the username comparisons triggered by collisions on a single hash value. The find_user() function takes a username and username size as input. It then hashes the username and passes the value to the find_hash_hit() function, which finds the corresponding hash table containing all the users with that same hash value. The hash table is passed back to the find_user() function, which calls the lookup_user() function. The latter goes through each user in that specific hash table and first compares the sizes of the usernames; if they match, it looks for a second match on the full username. If the user is found, the location containing the user's information in the database, including the IP Address and port, is returned. To improve performance, future bare PC SIP server implementations will use adaptations of data structures and search techniques used by popular Linux SIP servers.

### C. *SIP Message Processing*

The siphandler() function manages the processing of received SIP messages. This function, which is called directly by the udp_handler() function after verifying the SIP port in the UDP header, is the key element in the bare PC SIP server. The siphandler() function calls the parse_headers() function. The latter goes through the SIP packet and parses out specific identifiers to identify the type of message (for example, REGISTER, INVITE, ACK, BYE, 180 Ringing, 200 OK and 100 Trying). Within the parse_headers() function are specific functions built to handle the following SIP tags: Header, Via, From, To, Expires, Authorization, Proxy Authorization, CallId, CSeq, Contact, and Content Length. In keeping with the lean SIP implementation, only the indicated tags are parsed to expedite the processing of SIP packets (other tags are bypassed). Once the tags are parsed and the relevant data from the packet is stored, control returns to the siphandler() function. Further processing is determined according to the request_type returned. Only the following SIP messages are processed by the Bare PC SIP Server: Register Invite, 100 Trying, 180 Ringing, 200 OK, Ack, Bye, and Unsupported. When the siphandler function has decided what to do with the SIP request, processing is carried out to forward the SIP message, or a reply is sent to the SIP User Agent (UA) by utilizing the generate_sip_response() function. This function generates the SIP reply (or 100 Trying response) based on the values retrieved earlier by parsing the SIP request. It then calls the sipsenddata() function, which calls the relevant protocol handlers to format the headers in the SIP reply.

Register Message: To process a Register message, the bare PC SIP server parses the Via (IP address:port), From and To (usernames@domain/IP), and Contact tags. It then calls the function check_registered_users(). A process similar to that described earlier is used to determine if the user is already registered (i.e., is found in the Registered_Users_Database). If so, only the relevant information is updated; otherwise, the system stores all necessary information parsed from the SIP request including the username, IP address and port number. This information is used to generate replies back to the UA on future requests until the UA re-registers or one of the parameters is updated. After the information is stored or updated, the server generates a 200 OK message and sends the reply back to the SIP UA.

Invite Message: For an Invite message, the bare PC SIP server parses almost all of the same fields as for the Register message. The server then sends messages to the caller and callee. A 100 Trying message is sent back to the caller letting the UA know that the SIP Server is processing the request. To send this message, the server looks up the IP address of the caller using the process described earlier. It also looks up the registration information for the callee and forwards the Invite message to its UA.

SIP Authentication: The Message format for an Invite request with authentication is shown in Figure 4. SIP authentication is done by challenging the initial request (Invite or Register) sent by the SIP UA. SIP uses HTTP authentication techniques. The bare PC SIP Server is designed so that each request is not authorized unless it receives the proper response for a given challenge. The server can be configured at start-up to operate with or without authentication. An authorization flag indicates if a particular request is approved or denied based on authentication. The bare PC SIP server processes the initial request, and then sends a challenge response back to the requesting SIP UA. The SIP server generates a challenge response that depends on the values of realm and nonce. The realm is typically set to the domain of the SIP server (for example, barepc.towson.edu or the IP address). The nonce is a string that is randomly generated by the server. Once the server receives the reply to the challenge, the fields in the authorization request are parsed from the SIP packet. Then the response value is computed using the MD5 algorithm and matched against the response value sent by the SIP UA. The response value is a hash that depends on the concatenation of all values in the authorization request. If the computed response matches the response sent by the SIP UA, the request is approved (authorized) and normal SIP call flow processing is allowed.

```
INVITE sip:67890111@barepc.towson.edu:5060 SIP/2.0
Via:SIP/2.0/UDP192.168.1.56:5060;brach=0320
From:<sip:0123456@ barepc.towson.edu>;tag=0
To: <sip: 67890111@ barepc.towson.edu>
Max-Forwards: 70
Call-ID: 0010-0003-DA76506F-0@AAE2A42DF82D1D0AA
 CSeq: 297386 INVITE
Contact: <sip:123456@192.168.1.56:5060>

Content-Type: application/sdp
Proxy-Authorization:Digest
username="8000",realm="BAREPC",nonce="3bd76584",
uri="sip:123456@192.168.2.81",response="6e91de67ad976997
ff"
User-Agent: BarePC SIP UA v1.0
Content-Length: 276
v=0
 o=Vega400 4 1 IN IP4 192.168.1.56
 s=Bare PC Sip Call
 t=0 0
 m=audio 10006 RTP/AVP 4 18 8 0 96
 c=IN IP4 192.168.1.56
 a=rtpmap:8 PCMA/8000
 a=rtpmap:0 PCMU/8000
 a=rtpmap:96 telephone-event/8000
 a=fmtp:96 0-15,16
 a=sendrecv
```

Figure 4. SIP invite with authentication.

## D. *User Interface*

The bare PC SIP Server has a simple user interface that displays its basic configuration and state information when the interface function sipserverstate() is called. The displayed information includes the number of users added to the username and password database, and the server's configuration mode (proxy, redirector, authentication, stateless, or stateful). The server can also show the username, ip address, and port for each user logged into the system. An administrator can toggle through the list of users, or configure the server so that the display is triggered every time a user is added or removed from the Registered_User_Database by calling sipserverstate() from the Main task.

## E. *SIP Server Internals*

The objects needed by the SIP server application such as apptask (for task implementation), SIPS (for SIP processing), DHCP, TFTP, UDP, IP, and Ethernet (for network protocol processing) or MD5 (for authentication) are implemented as C++ classes with associated .cpp and .h files as usual. Each object contains the data structures and methods for the object. Some assembly code may be used at lower levels. We do not discuss the code common to all bare PC applications such as USB boot code, Ethernet driver code, interfaces to hardware, and code to support other functionality needed by applications. The IP object is used in all bare PC applications and servers requiring network communication. The MainTask (Main task), RcvTask (Rcv or Receive task), and SipsTask (SIP task) are implemented as methods within apptask, while SIP server functionality is provided by sipsobj.

The methods in SIPS include processSIPSRequest(), sipserverinit(), sipserverget_db(), parse_authorization(), authenticate_user(), generate_sip_response(), sipsenddata(), format_sip_response(), siphandler(), register_user(), and parse_headers() as well as many others needed to implement lean SIP server functionality. We have omitted method parameters and do not discuss the specific functionality of all these methods, as we have seen the use of some of these above, and since method names suggest their functionality.

When a UDP packet containing a SIP request arrives, apptask calls insertSIPSTask() to insert a SIP task into the task queue and calls sipsobj.processSIPSRequest(), which serves as an entry point to the task and links to an entry in a table (known as the TCB table) that points to the entire packet and headers. This method in turn invokes siphandler(), which passes the packet to parseheaders() to parse the SIP packet as discussed previously. After the packet is parsed, the request is processed according to the request type. For example, in case of a register request, methods to check and register the user are called by the SIP handler, followed by a call to generate_sip_response() to form the appropriate response packet as seen earlier.

## V. PERFORMANCE

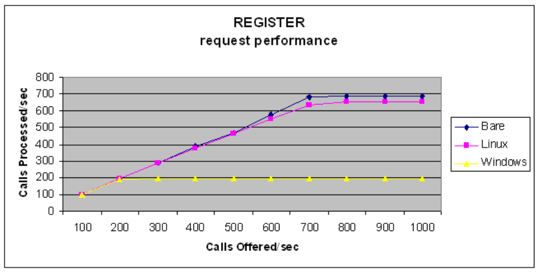
In this section, we present the results obtained from our performance studies. We compare throughput and latency for the bare PC and OS-based SIP servers using register, register update, register logout, invite, invite-not-found, and redirect operations. We also report internal timings for the bare PC SIP server for the register operation under maximum load.

### A. *Experimental Set Up*

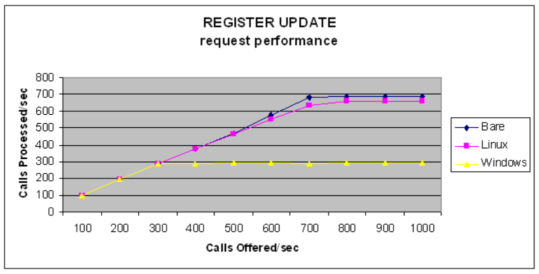
The test network consists of a 100 Mbps Ethernet to which each SIP server and the client machines running SIPp are connected. In addition to the bare PC SIP server, the details of the systems and software used are as follows: OS-based SIP servers: OpenSer SIP Server ver 1.3.2–notls (Linux) OpenSer (KAMILIO/OpenSIPS) [21] and Brekeke SIP Server ver 2.1.6.6 (Windows) utilizing the Jakarta Web Server and Java platform [22]; machines: Dell GX260's with Intel Pentium 4 (2.4 GHz), 1.0 GB of RAM and 3COM Ethernet 10/100 PCI network cards; OSs: Microsoft Windows XP Professional ver. 2002 Service Pack 2 and Linux Ubuntu 8.04 Kernel 2.6.24-16; workload generator: SIPp [23].

For register updates, the SIP Server searches its user database for a match and then updates the corresponding user's location data and registration expiration time; and in the register logout operation, it removes the user from the database. The invite operation requires the server to lookup the callee's contact details in its database, forward the request to the callee, and send the response back to the caller. The invite-not-found operation is similar to invite except that the callee is not found in the database. For redirect, the server receives an invite message, but instead of forwarding the response to the callee, it forwards a temporarily moved message back to the caller.

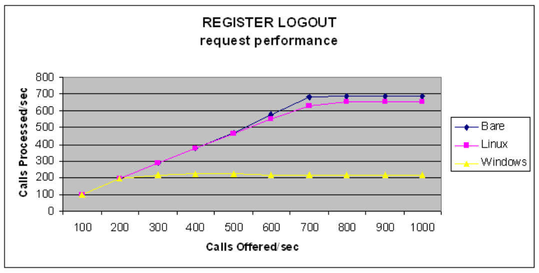
For the register, register update, and register logout operations, latency measures the delay at the user agent between sending the register message and receiving the "200 OK" message. Latency for the invite operation measures the sum of two delays: the time between the invite message and "200 OK" messages; and the time between the "bye" and "200 OK" messages. Each of these operations was also tested with authentication enabled, which adds processing overhead due to verifying the MD5 hash, and extra message overhead due to the "unauthorized" message for registration and "407 proxy authentication" message for invite (and their responses). Latency for registration with authentication measures the sum of two delays: the time between the register request and the "unauthorized message"; and the time between the new register message with authentication credentials and the "200 OK" message. Latency for invite with authentication measures the sum of three delays: the time between the invite and "407 proxy authentication" messages; the time between the "invite with authentication" message and the "200 OK" messages; and the time between the "bye" and "200 OK" messages. For invite-not-found and redirect operations, the latency is similarly measured using the "404 not found" and "302 moved temporarily" messages.



(a) Register



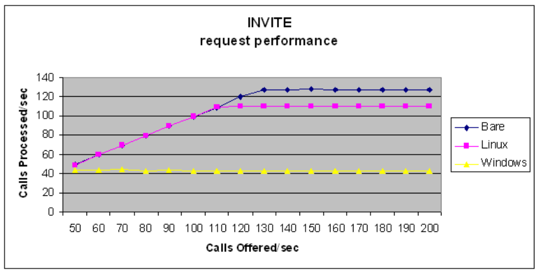
(b) Register update



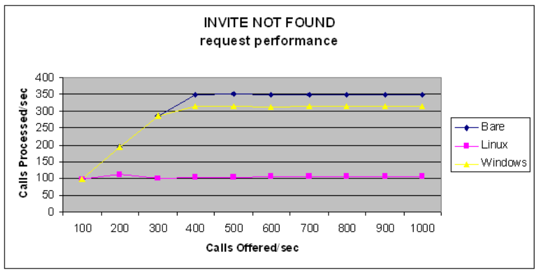
(c) Register logout

Figure 5. Throughput for register without authentication

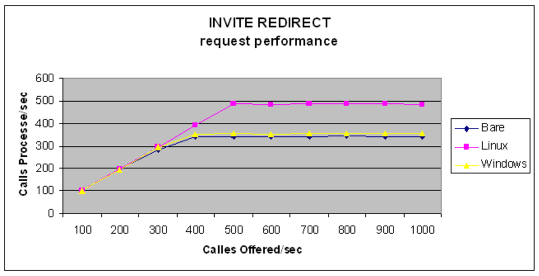
We measured the throughput and latency of a server associated with each SIP call flow. The latency for a given operation is computed by adding the respective delays between sending the relevant messages to the server and receiving their responses as described above. The throughput is the number of calls per second successfully handled with respect to the offered load, which is the number of calls per second that are generated and sent to the server. The peak throughput is the highest throughput achieved under overload while the server remains stable (and produces consistent results). To conduct the experiments, the servers were configured to operate in three configuration modes with and without authentication: registrar, proxy, and redirector. In addition, internal timings were measured by inserting timing points within the bare SIP server. Each SIP server was pre-loaded with 10,000 unique SIP username and password pairs. The call flows for register, invite-not-found, and redirect were run for a maximum of 10000 unique users, measuring the performance of each call flow with rates varying from 10 to 1000 calls/sec. The invite test call flows were run for a maximum of 5000 users with rates varying from 50 to 100 calls/sec. Each experiment was repeated a minimum of three times to ensure that the results were consistent.



(a) Invite

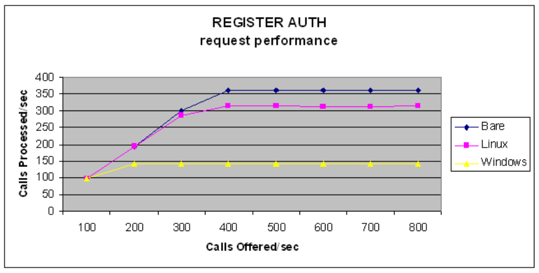


(b) Invite-not-found

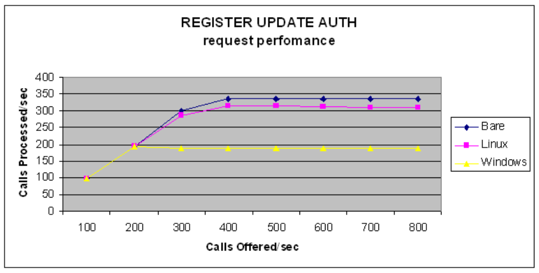


(c) Invite redirect

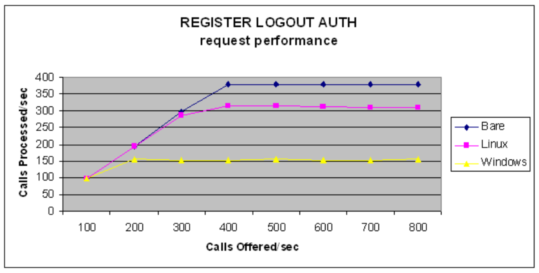
Figure 6. Throughput for invite without authentication



(a) Register

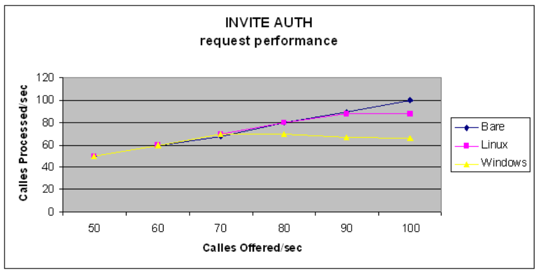


(b) Register update

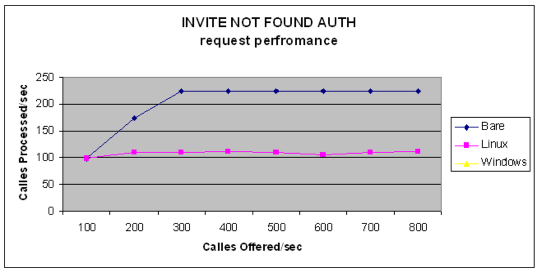


(c) Register logout

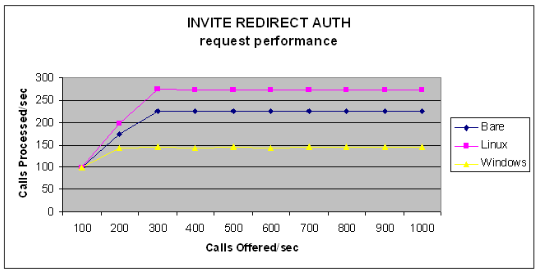
Figure 7. Throughput for register with authentication



(a) Invite



(b) Invite-not-found



(c) Invite redirect

Figure 8. Throughput for invite with authentication

## B. *Throughput*

The throughput for the register and invite operations respectively, without authentication, is shown in Figures 5 and 6. It can be seen that the peak throughput of the bare PC SIP server is always higher than that of the OS-based servers except in the case of invite redirect. The peak throughput of the bare PC server typically exceeds that of the Linux server by 50-125 calls/sec depending on the operation, although it is only 10 calls/sec more for invite and 150 calls/sec less than that of the Linux server for invite redirect. For example, the bare PC SIP server has a peak throughput of 700 calls/sec for register operations (without authentication), which is better than the peak throughput of Linux (650 calls/sec); the Windows server has a much lower peak throughput (around 200 calls/sec).

The peak throughput performance of the bare PC SIP server should be better than that of the OS-based servers, due to its simple design and the elimination of OS overhead. However, this performance advantage may be reduced or lost in certain cases due to inefficient algorithms or the lack of concurrency. The latter situation arises with the invite operation. The peak throughput of the bare PC server is only marginally higher than Linux in this case, but introducing a separate SIP task to handle an invite operation may improve performance. The apparent drop in performance of the bare PC server for invite redirect is due to a significant improvement in the performance of the Linux server in this case. Implementing Linux's search algorithm on the bare PC SIP server should improve its performance. A more efficient search algorithm should also improve the performance for the invite-not-found operation. The peak throughput of a given server does not vary much across the three register operations since the work performed in each case is essentially the same. The increase in the peak throughput of the Windows server for register update compared to that for the other two register operations is possibly due to caching.

The results in Figures 7 and 8 show that peak throughput of all servers is reduced as expected for both register and invite operations when authentication is added. This reduction in performance is due to the extra message overhead noted previously, and the overhead of computing and verifying the additional information needed for authentication with a message digest [8]. The negative impact of authentication on performance was also noted in [9]. There are no throughput values for the Windows server for invite-not-found with authentication since its message flow in this case could not be compared with that of the other two servers. It is evident that the peak throughput of the bare PC server with authentication shows a greater reduction versus its peak throughput without authentication compared to the OS-based servers. Adapting the approach used for authentication by Linux for the bare PC server could improve its performance.

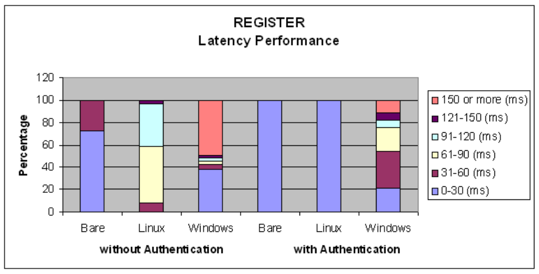
## *C. Latency*

Figures 9 and 10 compare the latencies for bare PC and OS-based SIP servers for the register and invite operations respectively, with and without authentication. In most cases, the bare PC server performs better than the OS-based servers. As seen in the figures, the highest latency percentages for the bare PC server are usually in the 0-30 ms range, and it rarely has latencies that exceed 150 ms.

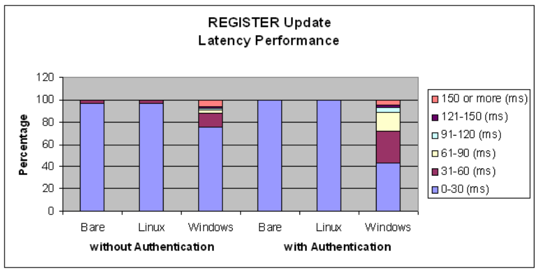
For register and register logout without authentication in Figure 9, bare PC server latency performance is better than that of the Linux server, but for register update without authentication it is the same. For example, in case of register logout without authentication, the latency performance of the bare PC server is much better than that of the Linux server: bare PC server latencies are less than 60 ms and most are less than 30 ms, whereas some Linux server latencies are in the 121-150 ms range and only a few are in the 31-60 ms range (none are less than 30 ms). In contrast, the performance of the Windows server is far worse than both of them with a large percentage of latencies exceeding 150 ms. For all register operations with authentication, the latency performance of the bare PC and Linux servers is the same.

It can be seen in Figure 10 that the latency performance of the bare PC server is better than that of the Linux server for invite and redirect without authentication, but worse for invite-not-found without authentication. Latency performance for both servers in case of redirect with authentication is the same. For invite with authentication, the latency of the bare PC server sometimes exceeds 150 ms.

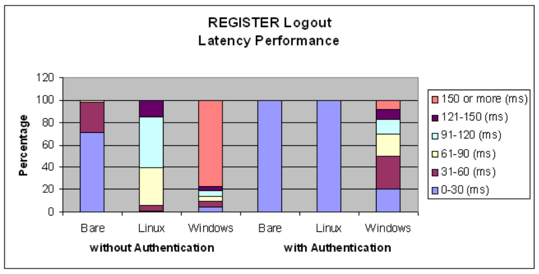
As noted above, improving concurrency and use of a more efficient search algorithm may help to improve bare PC server latency performance without authentication. Further studies are needed to determine if the techniques used to implement authentication in the Linux server will improve latency performance of the bare PC server with authentication.



(a) Register

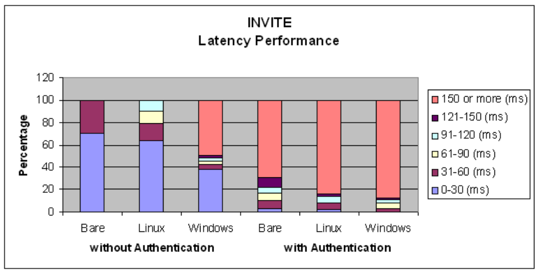


(b) Register update

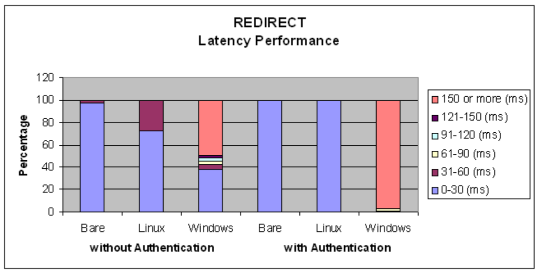


(c) Register logout

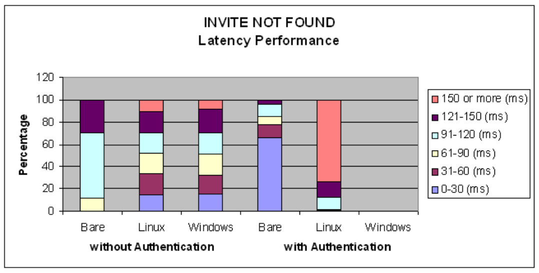
Figure 9. Latency for register with and without authentication



(a) Invite



(b) Invite redirect



(c) Invite-not-found

Figure 10. Latency for invite with and without authentication

### D. *Internal Timings*

Figure 11 compares average values of internal timings for the bare PC SIP server collected during the register operation under maximum load conditions. It is seen that FindUser, which searches for a given user, and ParseSIPHeaders, which processes the SIP header are the most expensive operations, although the former is twice as expensive as the latter. The least expensive operation is AddUser, which simply adds the information for a new user, and thus takes an insignificant amount of time as would be expected. The AuthenticateUser and FormatSIPResponse operations have approximately the same cost, which is about half that of ParseSIPHeaders. We conducted tests on the OpenSER server using OProfile 0.9.5 [24], which showed that the timings for the AddUser and ParseSIPHeaders operations exceed the corresponding timings on the bare PC by factors of 4 and 7 respectively.

We also used SIPp to increase the load on the server and obtain better estimates of internal timings when processing requests. Specifically, we varied the registration request rate from 100-800 requests/sec in increments of 100 requests/sec. We then measured the execution time spent in the siphandler() method that invokes all the other methods needed to process each request and generate the response as discussed previously. We also obtained the total internal processing time to register a user with authentication, which involved processing 2 packets sent to the server and processing two responses to be sent to the SIP UA. Thus, the total processing time includes the network delay and delay due to addition and removal of the various protocol headers.

The results are shown in Figure 12. It can be seen that the execution time spent in the siphandler() method is very small (approximately 180 microsecs) regardless of the registration request rate as would be expected due to its low overhead in processing SIP requests and responses. Likewise, while the total processing time spent per SIP request is larger due to network overheads, it drops in accordance with the increased request rate until the server reaches its capacity and then shows a slower rate of decrease. This is because the server has less ability to meet the offered load when its peak capacity is reached.

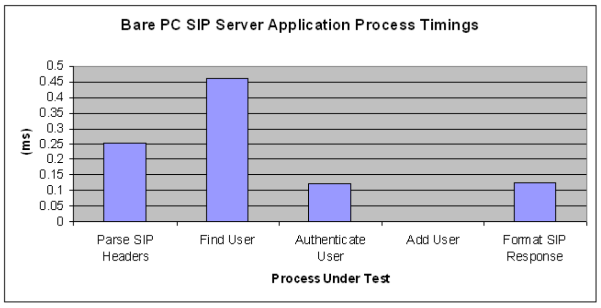
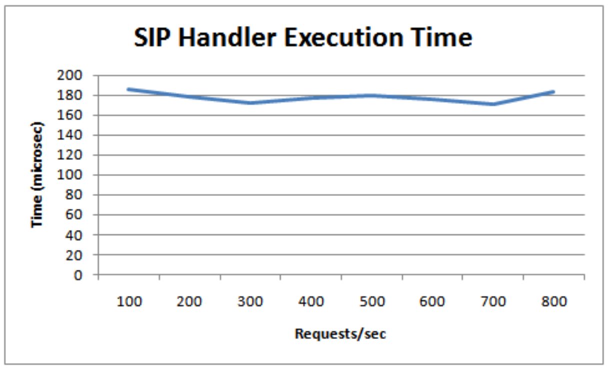
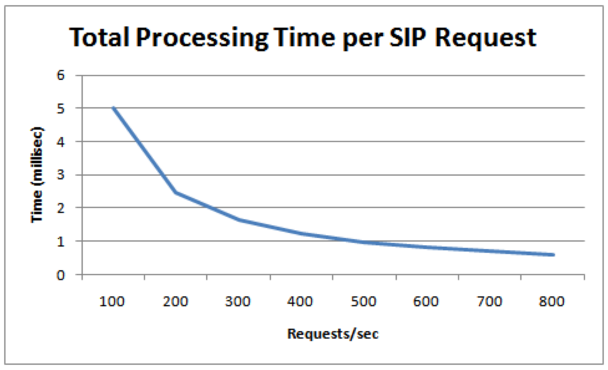


Figure 11. Internal timings for server operations



(a) Execution time spent in the SIP handler() method



(b) Total processing time per SIP request

Figure 12. Internal timings under increasing load

### E. *Throughput Analysis*

Further insight into the results on throughput may be obtained by considering sustainable throughput, which is defined as the maximum rate of calls for which the processed call rate matches the offered call rate. Sustainable throughput reflects the extent to which a server can cope with the offered load, and it can be determined from the preceding Figures 1-4.

For example, the sustainable throughput of the bare PC server for the register, register update, and register logout operations without authentication is respectively 400, 600, and 700 calls/sec (the peak throughput for all three register operations without authentication is 700 calls/sec). It can be seen that the sustainable throughput of the bare PC server exceeds that of the Linux server for all operations without authentication except for invite-not-found when it is the same. In contrast, the sustainable throughput for the two servers for all operations with authentication is the same (or differs by a small amount). As noted earlier, in the case of peak throughput with and without authentication, the bare PC server's values are higher than those for the Linux server except for invite redirect. Thus, both sustainable and peak throughput values should be used to estimate server capacity with and without authentication.

## VI. CONCLUSION

We described the design, implementation, and performance of a bare PC SIP server. Design details provided included an overview of bare PC SIP server tasking, server operation, and protocol intertwining. We also gave internal implementation details to illustrate bare PC SIP server functionality. In particular, we described the boot sequence, user lookup and database tables, and SIP message processing. In addition, we examined the relationship between tasks and method invocation in the server when processing SIP requests and responses.

Performance of the server was studied by measuring its throughput and latency for registration, proxying and redirection, with and without authentication. We also compared bare PC SIP server performance with that of the OpenSER server running on Linux and the Brekeke server running on Windows. The results show that the bare PC server has better performance than the Windows server and better or equal performance to the Linux server in most cases. The exceptions are throughput performance for invite redirect with or without authentication, and latency performance for invite-not-found without authentication for which the Linux server is better. Latency performance for the invite operation with authentication was poor for all servers.

We also provided internal timings measured on the bare PC SIP server when processing registration requests with authentication under increasing server load. It was found that Find User is the most expensive operation, Parse SIP Headers is moderately expensive, whereas Format SIP Response and Authenticate User are less expensive.

The observed performance results reflect the simple server design, efficient tasking strategy, and low implementation overhead due to absence of an OS. It is expected that the performance of the bare PC server can be improved by improving concurrency and using more efficient algorithms. The bare PC SIP server implementation could also be modified based on internal timings to reduce the cost of the most expensive operations. Our results serve as a baseline to assess the minimal overhead associated with basic SIP server operations for both OS-based and bare PC servers, and to help improve the performance of bare PC SIP servers.

## REFERENCES

[1] A. Alexander, A. L. Wijesinha, and R. Karne, "A Study of Bare PC SIP Server Performance," 5th International Conference on Systems and Network Communications (ICSNC), pp. 392-397, 2010.

[2] A. Alexander, A. L. Wijesinha, and R. Karne, "Implementing a VoIP SIP Server and User Agent on a Bare PC", 2nd International Conference on Future Computational Technologies and Applications (Future Computing), 2010.

[3] G. Ford, R. Karne, A. L. Wijesinha, and P. Appiah-Kubi, The Performance of a Bare Machine Email Server, 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 143-150, 2009.

[4] L. He, R. Karne, and A. Wijesinha, "The Design and Performance of a Bare PC Web Server", International Journal of Computers and Their Applications, vol. 15, pp. 100-112, June 2008.

[5] A. L. Alexander, A. L. Wijesinha, and R. Karne, "An Evaluation of Secure Real-Time Transport Protocol (SRTP) Performance for VoIP," Third International Conference on Network and System Security (NSS), pp. 95-101, 2009.

[6] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer bare PC VoIP Application," Proceedings of the IEEE Consumer and Communications and Networking Conference (CCNC), pp. 803-807, IEEE Press, Las Vegas, NV, 2007.

[7] K. Ono and H. Schulzrinne, The Impact of SCTP on SIP Server Scalability and Performance, GLOBECOM, pp. 1421-1425, 2008.

[8] S. Salsano, L. Veltri, and D. Papalilo, SIP security issues: The SIP authentication procedure and its processing load, *IEEE Network*, pp. 38-44, 2002.

[9] E. M. Nahum, J. M. Tracey, and C. P. Wright, Evaluating SIP server performance, in: 17th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), Urbana-Champaign, Illinois, June 2007.

[10] M. Cortes, J. R. Ensor, and J. O. Esteban, On SIP Performance. *Bell Labs Technical Journal*, 9(3), pp. 155-173, 2004.

[11] C. Shen, H. Schulzrinne, and E. M. Nahum, Session Initiation Protocol (SIP) Server Overload Control: Design and Evaluation, IPTComm, pp. 149-173, 2008.

[12] V. A. Balasubramaniyan, A. Acharya, M. Ahamad, M. Srivatsa, I. Dacosta, and C. P. Wright, SERvartuka: Dynamic Distribution of State to Improve SIP Server Scalability, ICDCS, pp. 562-572, IEEE Computer Society, 2008.

[13] K. Ono and H. Schulzrinne, One Server Per City: Using TCP for Very Large SIP Servers, IPTComm, pp. 133-148, 2008.

[14] H. Jiang, A. Iyengar, E. M. Nahum, W. Segmuller, A. Tantawi, and C. P. Wright, Load Balancing for SIP Server Clusters, INFOCOM 2009.

[15] K. K. Ram, I. C. Fedeli, A. L. Cox, and S. Rixner, Explaining the Impact of Network Transport Protocols on SIP Proxy Performance, ISPASS, pp. 75-84, 2008.

[16] L. Chen, and C. Li, "Design and Implementation of the Network Server Based on SIP Communication Protocol," World Academy of Science, Engineering and Technology 31, pp. 138-141, 2007.

[17] S. Zeadally and F. Siddiqui, "Design and Implementation of a SIP-based VoIP Architecture," AINA 2004.

[18] R. K. Karne, K. V. Jaganathan, N. Rosa, and T. Ahmed, "DOSC: Dispersed Operating System Computing", OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, pp. 55-62, 2005.

[19] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to Run C++ Applications on a Bare PC?" 6th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 50-55, 2005.

[20] G. Ford, R. Karne, A. L. Wijesinha, and P. Appiah-Kubi, The Design and Implementation of a Bare PC Email Server, 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 480-485, 2009.

[21] Kamailio (OpenSER) SIP server, [Online]. Available: http://sourceforge.net/projects/openser Accessed: December 10, 2010.

[22] Brekeke SIP Server, [Online]. Available: http://www.brekeke.com/sip/ Accessed: May 28, 2010.

[23] SIPp, [Online]. Available: http://sipp.sourceforge.net/doc/reference.html Accessed: December 10, 2010.

[24] Oprofile-A System Profiler for Linux, [Online]. Available: http://oprofile.sourceforge.net/news/. Accessed: May 28, 2010.