

# A Distributed Event Logging System

Ashutosh Jaiswal, Richa Kumar, Rudrava Roy  
{ashutosh, richa, roy}@cs.umn.edu  
(2401138, 2501329, 2527658)

CSci 8101: Advanced Operating Systems Class Project

## 1 Motivation

In the present environment of "pervasive" computing (not to use it in the general sense of the word, but to mean the existence of computers everywhere and for every purpose) and with the widespread installation of computer networks, the face of a modern day "application" has changed to an extremely complex system with a tremendous number of components which might have come from different vendors. And with the way these applications are deployed in the modern day distributed computing environment, it becomes very difficult to track the performance of such applications across the system.

Monitoring and troubleshooting has come to become a significant part of the Total Cost of Operation (TCO) of these systems. In light of that, we are of the view that a system wide logging facility which somehow makes the distributed nature of the underlying framework transparent and lets System Administrators (or even applications themselves) monitor the performance of the system as a whole, will prove a significant boon to the acceptability and ease of use of these systems, and would help further expansion. Such a mechanism will also help in detecting co-relations between similar events occurring at different nodes within the system, which in turn could prove to be useful in deterring/detecting attempts to break into the system.

In this project, we aspired to provide a system-wide logging facility, which is transparent to the application as well as the user. At the same time we tried to add robustness and some amount of security to the process. Although we recognize that there could be a large number of enhancements to make the system really secure - but our idea was to put the infrastructure in place and then allow people to make it secure in their own ways.

At the application level, system calls are made available to write to the distributed logging mechanism, and the standard logging mechanism (the kernel log and system log) were transparently moved to this domain. That is, the programs that used the standard system calls for logging still work the same way, but will log into the distributed event log. The logs work like a distributed database, which can also be queried by a set of APIs. Again, the querying process is transparent to the application generating the query.

There would also be a set of authentication mechanisms in place to check unauthorized users/applications from gaining strategic information about the running

of the distributed system. We are currently targeting the arena of service maintenance and security auditing on a distributed environment consisting of a set of Linux machines.

## 2 System Calls

The foremost requirement of our system was to enable the huge pool of existing applications that use conventional unix event logging mechanisms to be able to use our mechanism without much hassles. Rewriting the applications was out of the question because that just doesn't scale for the number of applications present. The next best choice was to hook the existing function calls from shared libraries and add a stub that does logging through our mechanism. For this, we had to create a new version of 'libc' which contains the instrumented implementations of the 'openlog', 'syslog' and 'closelog' function calls that are used by applications to write to the local system logs. The code added to the above functions enables them to connect to the local SQL server and dump the messages to it. The local server is able to distinguish between local and remote logging requests and thus forgoes authentication on the logging request. Another more secure way of implementing it would be to use Unix Domain sockets to which the SQL server listens, just as the message logging daemon `sysklogd` does on conventional unix systems.

The local SQL server keeps a running buffer of messages passed on to it from processes running on the machine, and uses checkpointing to decide when to forward the logs to be replicated at other sites. There are a variety of criteria to be used for checkpointing. The implemented rule is to forward to other hosts if the last message was written to a host one minute earlier or there are 10 messages in the buffer pending remote writes. It would also be feasible to implement an immediate write depending on the criticality of the message as specified in the `priority` parameter passed to the function call

```
void syslog(int priority, char *format, ...)
```

When the client sends a log request to a remote server (the client in this case is part of the local SQL server), the following transactions take place. The client encrypts the message with it's private key and writes it to a remote server. The remote server, on finding an 'Insert' request (as discussed later), looks up the public key of the client and decrypts the message. It then selects a random length string at a random location in the message, and encrypts it with it's own private key and sends it back to the client. The client, knowing which remote machine it is talking to, picks up that machine's public key (which it will have if it is an authentic client), decrypts the last received message, and sends it in plain text to the remote server. The remote server then compares the returned string with the originally selected string and if a match is made, inserts the messages to it's local log or else rejects it as a bogus request from an unknown client.

For applications that are written with the event logging framework already existing can make better use of the system by making specific function calls which extend the event logging functionality by allowing messages to write binary data to the logs. The following functions are exposed for easy use of the logging mechanism:

```

void DSopenlog(char *ident)
void DSwritelnlogtext(int priority, char *message, int len)
void DSwritelnlogbin(int priority, void *message, int len)
void DSwritelnlogmix(int priority, char *msgText, void *msgBin, int lenText, int lenBin)

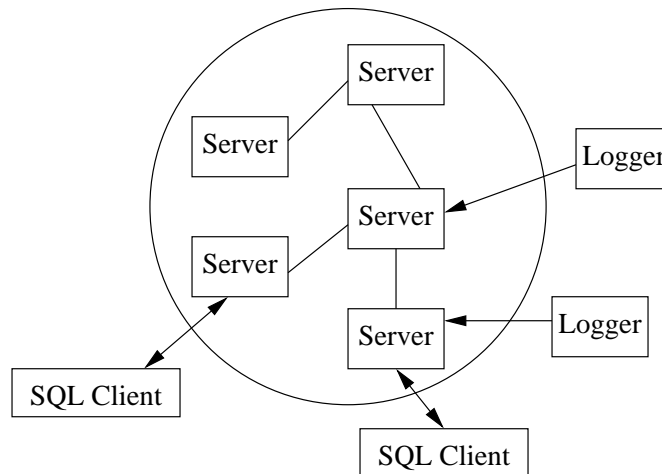
```

The first function call sets up the data structures needed for the subsequent logging requests. The second function takes as input a preformatted text message of length 'len'. The priority option is the same as specified to the `syslog` function call. The third message logs binary data of length 'len' bytes, and the fourth logs a message containing an initial text segment followed by the binary segment.

### 3 Distributed Event Logging System: Overview

In order to implement a distributed database for event logging, it is essential that an SQL server be running on each host that is part of the system being monitored. Since a full fledged SQL server would put immense load on the machines, each machine in the system runs a stripped down version of an SQL server which provides support for the bare minimum set of SQL commands.

The reason we do not need a more powerful SQL server is that the system is more of a distributed query tool and does not need any transaction semantics. We simply need some basic queries that allow a system administrator to extract meaningful information from the logs, while at the same time presenting the notion of a central repository of such logs (while the underlying implementation is distributed). In other words, here's a tool that gives you all the information you need about any events occurring on any of the machines in that network. It also makes it easier to correlate events.



The figure above shows what the system looks like. The system is designed to run on a network of Linux machines. The system calls for logging, namely `syslog(3)`, have been modified so that messages are sent not only to the standard `syslogd`, but also to our log server `logger`. The `logger` simply connects to the SQL server on the same host, and inserts one record into the database for each message that needs to be logged. How and where this record is stored will be discussed in the remaining few sections. When a system administrator needs to check the logs, he/she simply

runs the SQL client on any machine on the network and can query for the required information. The SQL client connects to the SQL server on the same host. However since all the SQL servers communicate with each other any relevant information that is present on a remote machine will be made available to the SQL client.

## 4 SQL Database Design

The SQL database that we have implemented for distributed event logging system described above is extremely primitive, yet effective enough (we hope).

At this point, think of the database as a central store of all information. In the next few sections we'll make this distributed. Assuming that its central, we have the following design for the database.

The database is file based, meaning that all information is stored in flat files. There is still a notion of a table, which is essential in a SQL database. Records of a table Table1 are stored in a file named Table1. Since this is a logging system, where data only gets added and never deleted, it's not hard to imagine that these files may get too big. Periodically, the files are backed up and a new file is generated. This means that any queries that are run, will only look through the records present in the current data file, unless the query is looking for a particular timestamp in which case the appropriate data file will be searched. This periodic backing up has not been implemented yet.

Every line in the data file represents a record in the table. In order to be able to query based on specific fields and to extract only certain fields of the record, the record is inserted into the file with fields tagged with field descriptors. There exist two meta data files: meta.tables and meta.tags. Every table in the database must have an entry in meta.tables which indicates the different fields in the table. So if we have a table logs having the fields timestamp, pid, user, host, process, message, then its entry in meta.tables would be

```
logs ~fields~:timestamp:pid:user:host:process:message
```

Similarly, the following entries should be present in the meta.tags file

```
logs.timestamp ts
logs.pid pid
logs.user u
logs.host h
logs.process p
logs.message m
```

These two files help in validating queries and searching within the data files. The meta.tags file gives information about what tags are used for each field of the table. So the data file for the table logs may have records of the following format (showing 2 records)

```
<ts>0ct 28 4:59:41</ts><pid>4567</pid><u>root</u><h>r</h>
<p>sample</p><m>listening on port -1</m>
<ts>0ct 27 14:59:17</ts><pid>123</pid><h>z</h>
<p>/etc/hotplug/usb.agent</p><m>.. no drivers for USB product 0/0/0</m>
```

By tagging information in this way, searches can be done based on specific fields values. Ultimately, its pattern matching, but done in a slightly more effective manner.

Commands supported by our primitive SQL server implementation:

Desc<tablename> Lists the fields within the table

Select\*|<listofcommaseparatedfields>from<tablename>[where<fieldvaluepairs>] Searches within the data file corresponding to the table for required values in the specified fields and display the results (the desired fields only). "and" and "or" operations are supported for the where clause. The only two relations that are currently supported between the key field and its value are "=" and "like".

Insertinto<tablename>values(<commaseparatedlistofvalues>) Inserts the record into the data file for the table

Quit Disconnects from the server

Delete has not been implemented, as records from logs would normally not be deleted. However, if required it could be easily incorporated.

Currently, table creation and deletion are manual, meaning the database administrator needs to create the necessary entries in the meta files and create the data file for the table. In the future, table creation would also be supported. Since the database implementation is specifically in the context of the distributed logging system, the focus is more on query implementation.

Simple triggers are supported. Inserts are checked against some predefined patterns (potential attack signatures). If any of the patterns are matched, it triggers an email to the system administrator.

When the SQL client connects to the local SQL server, a session is created. Queries are sent to the server, which returns the results (as if the local server had a view of the entire database) to the client. When the client quits, the session ends.

## 5 Making it Distributed

In the earlier section we described how the database is implemented assuming that all the data is present on a single machine. The following discussion explains the approach to making the database distributed.

In place of a central SQL server we now have one server running on each machine with the log information distributed (possibly replicated - see next section) among them. In the absence of replication, all the logs for a machine would reside on that machine only, and all the logs present in the local database would pertain only to that machine. However a query initiated on any of the machines should be propagated correctly so as to hide the distributed implementation of the system. Furthermore, since a query can reference information that could potentially exist anywhere on the system (e.g, a query for all logs generated by the application 'apache'), it is more often than not necessary to search through the whole system.

Each server, at startup, reads in a routing table that contains information about how to communicate with the other servers. Every time the server receives a request from a client (either select or insert, since desc and quit are handled locally), it

consults the routing table to decide which other machines it needs to forward the query to. Ultimately the query has to propagate to each machine, except for very few cases in which the references are only to logged messages on the current host. Once the server knows where to forward the query, it then acts like the client for these servers, which in turn might forward the query still further and so on, until we reach every machine and the results begin to come backwards along the same path.

The servers are implemented as asynchronous state machines to achieve parallelism. Each server receives:

- Requests from the sql client interface
- Request from the logger
- Replies from other servers (when requests are forwarded)

The server classifies each of its sockets as UNUSED, REQUEST or REPLY. Appropriate data structures are used to maintain information about all existing connections. When a server receives a request and needs to forward it to other servers, it sends asynchronous messages to the servers and updates the state information for the new connections. In the event of a reply, the server looks up the state information to determine who the original client was and sends back the results.

While implementing the above features, we were faced with some interesting design decisions. Here is a brief discussion of some of those. Firstly, while querying, is it possible to not bog down other servers with queries if it can somehow be figured out which server need not be queried. The final decision taken was to propagate all queries except for the ones in which the `host` field is exactly the local host. Second, we had to decide whether the logged messages should reside only on the local host or do we gain an advantage by replicating them? On this issue, we took up a specific scenario where a host is compromised, and a malicious user can alter the local logs to erase traces of a security breach. In such a case, it would be highly desirable to have a copy(ies) of the log on other host. The task of erasing traces becomes further difficult if there is no deterministic way of deciding where the messages reside apart from the local host. Third, how do we structure the network of the participating machines itself so that the system remains scalable and still does all we want it to do? The most heavy weight solution to the problem would be to implement a graph with each node having the knowledge of it's immediate neighbours. Under such an implementation, one would have to deal with very tricky problems of cycling of messages. The next best solution was to implement it as a tree and hence get rid of cycles - but that leads us to some other problems - what if a node with subtrees under it crashes, and what about the traffic through the root node when a query propagates through the whole system? For the first problem, an elaborate solution was carved out (but it was not possible to implement it due to access restrictions on workstations). For the second, we adopted the ostrich solution - we buried our head in the sand and pretended the problem didn't exist. On a more serious note, one could structure the network such that the machine acting as the root has the best processing power. Fourth was the issue of authentication of peers. This is a very elaborate issue and is discussed in detail in it's own section.

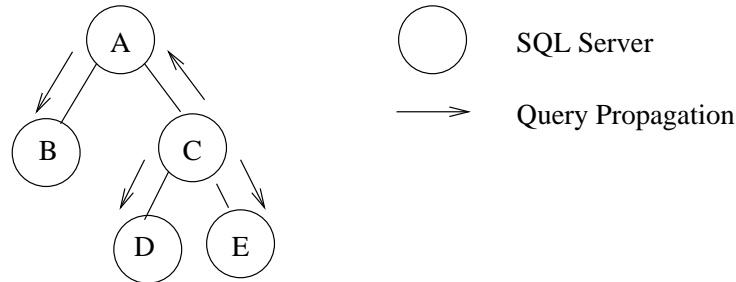
## 6 Query Propagation and Optimization

In order to collect all data pertaining to a query, the query needs to be propagated to all the machines in the system, since the data can potentially reside on any machine.

We refer to the server, which receives the initial query as the source. One possible way to achieve this is for the source to send the query to each other machine on the network and finally send the results back to the client. A more efficient way, and the way we have implemented it, is to propagate the query along a tree rooted at the source. In this way we can achieve some form of parallelism in the query propagation.

Each server, when it starts up, reads in a routing information from a file. This information contains the information about the neighbouring nodes of this machine in the propagation tree. The algorithm for forwarding a select query works as follows:

If the select is originated at a machine, it is forwarded to all neighbours, the results are collected and checked for duplicates before being returned to the client. If the select is not originated at that machine, it is propagated to all its neighbours except the one from which it received it. In this manner the query is propagated to every machine.



*eg.* referring to the above figure, a select query is originated at machine C. Machine C propagates the query to A, D and E. D and E both have only one neighbour, C, from which they received the query, so they do not forward it anymore. A forwards the query to B. The results flow back along the same path. At the originating machine, C, the results are filtered for any possible duplicates.

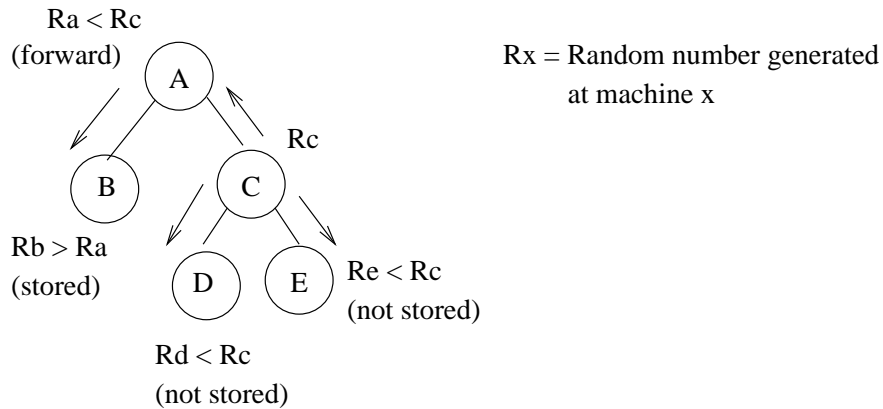
If the data is not replicated, or is done so in a deterministic manner, query propagation can be optimized, since certain machines will contain information that is also available at other machines, in which case the query does not need to visit all the machines. Also, queries that have a where clause based on the host field can be optimized. In this case, the query propagation stops once it reaches the host mentioned in the query.

## 7 Data Replication and Handling Server Crashes

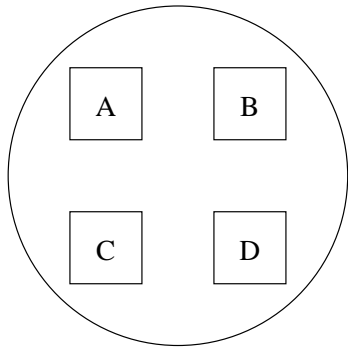
So far we have been assuming that logs for a particular machine reside locally and only that machines logs are present in the SQL database. Even if our system can handle crashed servers, in that the rest of the system still functions, in the absence of data replication query results will not be accurate at least for the time period when the server is down. This means that even if a single server crashes, the system will not function correctly, although it might appear to.

Therefore, in order to make the system more robust, we replicate the logs in

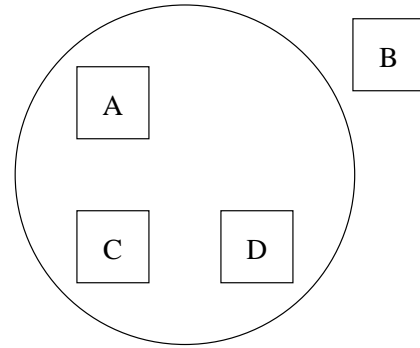
the following manner. Every time a new log is inserted, it is inserted both in the local database and atleast at one other machine in the system. There are interesting issues around the choice of these other machines. One of the goals of our system is to provide security of the logs. This is to prevent an attacker from covering up his/her tracks by modifying the logs appropriately. If we make the data replication static, where in the logs of machine A are always replicated at machine B, an attacker simply needs to get access to both these machines. In order to thwart such an attack, the logs are replicated at some randomly selected machines besides being stored locally. This choice is made on a per log basis, so logs for machine A could possibly be scattered on every machine in the system. Obviously, the attacker now has a much bigger problem at hand. However this makes the query propagation a little trickier, since optimizations are harder and the results may need filtering for duplicates.



The log replication is done in the following manner. Suppose machine B receives an insert request from the logger. It first stores the inserted record locally. It then generates a random number, appends the number to the original request and propagates the new request to one random neighbor (from amongst its parent A and children D, E). The selected neighbor (say A) generates a random number on its own. This random number is compared to the random number in the propagated request. If A's number is greater than B's number then A stores the log request locally and does not propagate the insert request any further. However if A's number is less than that of B, then it propagates the insert request further in a similar manner, updated with its own random number. Finally, if the request reaches a node with no children, it is terminated there and the message is inserted into the log there. This ensures that for every log insert request there are atleast two copies: one present on the original machine which got the log request and the other(s) on some randomly selected machine(s).

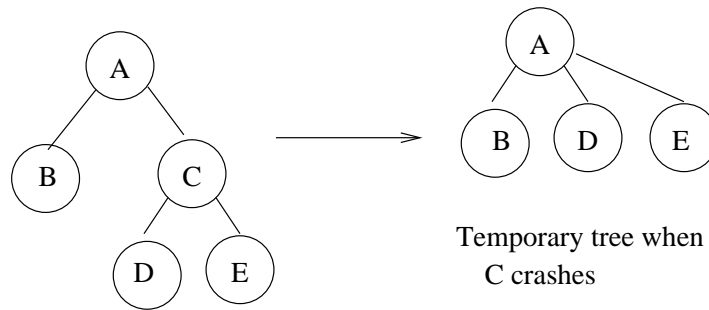


Group of Servers



Server B crashes, group updated

We refer to the set of active servers as a group. The query propagation mechanism requires that all servers currently part of the group be functioning correctly. This means that somehow, all servers should be informed of server crashes, so that they can update their notion of the group and propagate queries accordingly. Currently, this is implemented as follows: before a server forwards the query to another server, it simply pings the server to check if it is active. This mechanism could possibly be replaced by some process group membership mechanism. [Flaviu Cristian's work on Process Group Membership in a Distributed System].



Initial Propagation Tree

Temporary tree when C crashes

The way we envisage handling node crashes is as follows. If a server tries to propagate the query to one of its children, and realises that the child server is not responding, it issues a broadcast request asking for the children of the server that is down. It then temporarily adds these children to its list of children and the children update their parent information. In this manner, the server that is down is effectively removed from the propagation tree and the system is not affected. The temporary change in the tree structure takes place only after the appropriate authentication has been performed (This is described in the next section). The “broadcast” itself is not a normal TCP/IP broadcast limited to the current subnet as we wanted the system to remain scalable. Instead, each member of the DEL cluster subscribes to a specific multicast group and the “i’m-looking-for-bob’s-kids” messages are sent to that multicast group. In this way, even if a segment of the cluster resides on a remote network, one can enable communications between them using MBone.

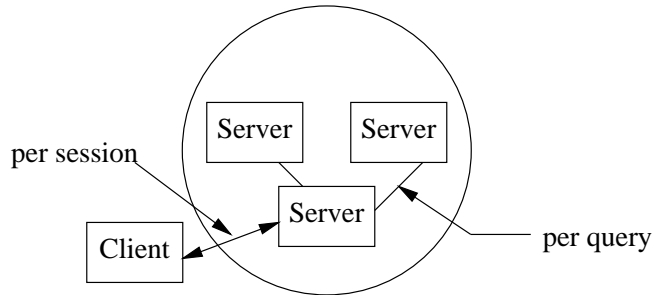
In the absence of authentication, an attacker could possibly crash one of the servers and then reply to the broadcast, masquerading as one of the children. When

the crashed server comes up again, it sends a message to its neighbors to update their routing information.

The probability of a server crash is assumed to be low. Therefore broadcasting is limited to such rare times.

## 8 Authentication

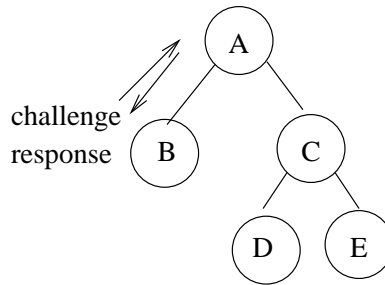
Authentication between the SQL client and the local SQL server is done on a per session basis. This means that if the client issues more than one query during that session, authentication takes place only initially and not for every query. In the case of server-server communication the authentication takes place per query.



The kind of attacks that are possible in a distributed system (in absence of authentication), include "the man in the middle" attack, where an attacker masquerades as one of the servers in an attempt to access the log information. To prevent this, we decided to have public key authentication, using challenge response to preclude replay, as the authentication mechanism.

Using machine A as an example, each machine contains its private key  $[K_A^-]_K$ , in encrypted form. The symmetric key  $K$ , used for encrypting and decrypting the private key is entered by the system administrator at the server startup and is henceforth stored in memory only. Here, we come across an elusive problem. How secure can we make the key itself? Well, not too much. We could lock the key in a box and lock the box and lock the key of the box in another box and lock that box ad infinitum. But then, it is still a problem of getting the key to the last box! So why have so many boxes at all? We again adopt the ostrich solution to the problem, and note that such security issues are 'beyond the scope of this project'. On the other hand, we would surely like to put a very interesting OS feature on our 'Wish List' - that of 'secure address spaces'. We would like to have a segment of memory all to ourselves that can never get copied anywhere and never be read by anyone other than the process that created the 'secure chamber'(even if the administrator on the machine wants it to - because the administrator himself will become the single point of failure in that case). The viability of the rest of the security of our system rests heavily on the foregoing assumption. The server A also maintains its own certificate  $[A, K_A^+]_{K_{sys}^-}$ , signed by the system administrator. Each server also maintains the public key of the system administrator,  $K_{sys}^+$  and the public key of all its neighbors in this case,  $K_B^+, K_C^+$ . Referring to the figure below, if A is propagating the query to its child B, A sends a connect request to B. B encrypts a nonce with A's public key. A then decrypts the message with its private key (which only A can

have!) and sends a signed reply to the nonce. B verifies this using A's public key against the expected reply. If this reply is verified, B accepts the connection and the query is propagated from A to B.



In the event of a server crash *say C*, when the crashed server's parent, *A*, detects the server crash, it sends out a broadcast message to update its routing information (as described in the earlier section). However before temporarily adding the crashed server's children, *D* and *E*, to its own list of children, it needs to authenticate them. This is done by authenticating *D* and *E* through their certificates, which are sent in response to *A*'s broadcast message. The whole authentication scheme follows the concept of challenge-response and the certificates are verified with the help of the system administrator's public key. It is thus essential that the system administrator's private key, be secure. Please note that we were not able to implement this feature as creating multicast groups requires the existence of multicast capability on at least one IP interface on the machine and the existence of a route for multicast messages. As we did not have access to configure such on any machines except for one, we could not implement the mechanism.

## 9 Comparison with existing systems

Event logging is an everyday phenomenon in the systems world. During our research on the subject we came across the following systems/proposals, which seemed similar to our goals:

1. The Enhanced Event Logging System ([http://docsrv.caldera.com/SM\\_eels/PREFACE.html](http://docsrv.caldera.com/SM_eels/PREFACE.html)): This is a system, which provides a mechanism to centralize the logging, reporting and management of standard UNIX logging systems. It uses a RDBMS to log the events obtained through various logging mechanisms such as syslog. The system can receive messages from remote systems and log them centrally.
2. Linux Event Logging for Enterprise-Class Systems (<http://evlog.sourceforge.net>): This system tries to provide event-logging facilities for Linux based systems on an enterprise scale. The basic aim of the project is to offer the capability currently available in commercially available, enterprise class UNIX based operating systems. The main emphasis is on using POSIX-compliant format for storing the event logs. The system also possesses ability to store the logs in a private log besides the general event log in a binary format. Logs can be queried to obtain information pertaining to a particular user or to set up filters for selective retrieval of records.

3. DECALS: Distributed Experiment Control and Logging System (Alex Hubbard, C. Murray Woodside, and Cheryl Schramm. Department of Systems and Computer Engineering Carleton University, Ottawa K1S 5B6, October 1995): This is a system developed mainly with the intent of running distributed experiments over a network remotely, rather than event logging. However part of the system provides the capability of logging the results of the experiments and reporting them back to the controlling workstation. Data is collected separately from various workstations and then merged together later, after adjusting for global time.

All these systems are primarily centralized. Although some of them can collect data remotely, but the information is stored in one place only. This introduces a single point of failure. Besides since these systems don't consider replication, the logging data is lost if the server/machine on which the logs are stored, crashes and loses the data on it. A central repository is also an Achilles' heel when it comes to security. Once the repository is compromised, the attacker has access to all the logs of all the machines! We've tried to make our system distributed by storing the generated logs at the local machine and replicating them on another server in a random way. This ensures that in case of machine going down, there is a copy of the logs somewhere on another machine. Also if an attacker gets into a machine, he/she has no way of getting all the logs, as the log storage is not deterministic and hence hard to figure out. This can also help in intrusion detection.