Slogger: A Profiling and Analysis System based on Semantic Web Technologies

Mark Baker
School of Systems Engineering,
The University of Reading,
Whiteknights,
Reading,
Berkshire,
RG6 6AY, UK

Email: mark.baker.computer.org
URL: http://acet.rdg.ac.uk/~mab/
Tel: +44 (0) 118 378 8615
Fax: +44 (0) 118 975 1994

Richard Boakes
Distributed Systems Group,
University of Portsmouth
Mercantile House,
Hampshire Terrace,
Portsmouth PO1 2EG,
Hampshire, UK.

Abstract

Increasingly, distributed systems are being used to host all manner of applications. While these platforms provide a relatively cheap and effective means of executing applications, so far there has been little work in developing tools and utilities that can help application developers understand problems with the supporting software, or the executing applications. To fully understand why an application executing on a distributed system is not behaving as would be expected it is important that not only the application, but also the underlying middleware, and the operating system are analysed too, otherwise issues could be missed and certainly overall performance profiling and fault diagnoses would be harder to understand. We believe that one approach to profiling and the analysis of distributed systems and the associated applications is via the plethora of log files generated at runtime. In this paper we report on a system (Slogger), that utilises various emerging Semantic Web technologies to gather the heterogeneous log files generated by the various layers in a distributed system and unify them in common data store. One unified, the log data can be queried and visualised in order to highlight potential problems or issues that may be occurring in the supporting software or the application itself.

Keywords: post-mortem profiling, log analysis, Semantic Web, distributed systems and applications.

1. Introduction

Distributed systems provide platforms for all manner applications whose design is based on paradigms such as client-server, multi-tier, object-based, and loosely coupled components. It is well known that diagnosing problems with these distributed platforms and the associated applications can be time consuming and often difficult to solve. Unlike parallel computing platforms, where there is a range of mature tools for diagnosing problems, such as debuggers and profiling tools that can be used either on executing applications or in post-mortem modes. In the distributed computing arena, few tools exist to assist programmers with fault diagnoses and execution profiling, apart from networking tools based on the Internet Control Message Protocol (ICMP), e.g. `ping`, and `traceroute`, or the Simple Network Management Protocol (SNMP). These tools can aid fault or problem diagnoses, but really only help in the manual process of elimination. Unlike, parallel processing platforms, which are typically tightly coupled and homogeneous, loosely coupled heterogeneous distributed systems add further complexity into the fault or problem diagnoses process.

Another common way of trying to diagnoses problems is to examine the log files of the operating system, middleware (e.g. application servers, message passing libraries, or database transactions) and application specific logs. Typically logs are examined in isolation. For example, if there were a problem with an Apache Axis application, Apache Tomcat's logs would be examined to find events generated during operation. This is itself a tedious and time consuming process, but also takes expertise to identify the events related to problems that may exist. Moreover, some events may indicate a problem, but its root cause may actually be caused by the operating system or may be an underlying hardware problem. In reality to fully understand a problem or fault in a distributed system or application analyses of events generated by the operating system, associated middleware libraries and the application are necessary.

We believe that it is possible to generate sufficient log-based information within a distributed system and application to understand more or less any fault or problem that may occur. However, the events associated with these faults or problems are typically written to log files in different formats and in numerous locations. Therefore, to identify a fault or problem all these log files would to need be gathered, synchronised, normalised and joined. Then, one could make various queries across this combined log data store and visualise the output in order to identify potential faults or problems. The immediate issue that appears is how to gather these heterogeneous log files, synchronise their events so that they can be logically ordered, normalise their data into a form, which means that different but equal terms are fully understood, so a common form in available in unified data, and finally unify all the data into a common store that can be subsequently queried. There are obviously several ways that these issues could be addressed.

One way to unify these log files would to be use relational database technologies, an alternative way would be to use various emerging Semantic Web technologies. We believe that relational database technologies are too rigid, and unfortunately the format and syntax of the log file would have to be known posteriori, as would the database schema, and other information related to the set up and format of the database. Whereas, with Semantic Web technologies there is much more scope to use relatively free formats, and the data store can ingest irregular data.

2. Related Work

2.1 Introduction

Profiling and analysing parallel and distributed systems and their applications is not new, there are various open source and proprietary systems that attempt to undertake this task. In the following section, we outline the most popular open source contenders in this field, and then we discuss how these systems are unable to tackle the over arching problems that typically exist in the systems that we wish to explore. It should be noted that we only detail and compare systems that are capable of being used for both understanding the behaviour of parallel and distributed systems and their applications.

2.2 Related Tools and Systems

The NetLogger Toolkit [1][2] is a collection of tools for analysing the performance of distributed systems and applications. NetLogger provides a logging API and is available for several languages including Java, C, Python and Perl. The logging API is used to instrument applications that then writes trace data to a log-forwarding daemon (`netlogd`) when they execute. `netlogd` can multiplex logs from different sources, combining them into a single log file. NetLogger also provides a visualisation tool that represents events against a timeline, and can also model profile data in the same view, enabling the comparison of resource-use against application progress. The timeline can be zoomed in to focus on details.

SvPablo [3], is a graphical environment that can be used to instrument application source code and subsequently browse dynamic performance data. SvPablo supports instrumentation via its graphical user interface (GUI) and automatic use via its standalone Parser. During the execution of instrumented code, its libraries capture performance data and produce metrics, including general software statistics and hardware counter data on the execution dynamics of the instrumented code. SvPablo can capture the execution behaviour of codes that execute for hours or days on hundreds of processors, as it produces statistics of execution behaviour. After application execution, data from each processor is integrated into a single performance file with additional statistics, which can be loaded into the GUI and displayed along with the application source code. The GUI provides an intuitive interface that allows a user to examine source code that reveals further statistics and useful details. By browsing the performance data correlating it with the source code, users can identify the performance bottlenecks in their applications rapidly. In addition, it provides a means for users to conduct load balancing analysis and scalability studies. SvPablo also includes Autopilot [4], which allows performance data to be captured and rendered in a three dimensional visualisation environment at runtime, so that users can investigate their application's behaviour during execution and potentially optimise its performance by manipulating individual variables or initialising independent threads of control.

The Tuning and Analysis Utilities (TAU) toolkit [5][6] provides facilities to instrument, monitor, analyse and visualise parallel applications. TAU can collect program trace data from Fortran, C, C++, Java and Python-based applications using automatic instrumentation capabilities, provided by the Program Database Toolkit (Dyninst), or via manual instrumentation using the API directly. TAU ParaProf provides various (graph based) visualisation mechanisms to assist in analysing parallel application performance. Emphasis is placed on ParaProf's ability to identify bottlenecks. It can show aggregated information as well as focusing on single threads and nodes. TAU includes conversion tools that allow its data to be analysed and visualised in several systems, such as, Jumpshot [7], Vampir [8], and Paraver [9].

The Ganglia [10] system is a widely used monitoring suite for distributed systems, which focuses on the gathering operational data that describes hardware events and system status. It communicates event information through a hierarchy of daemons, which can subsequently be visualised via a web-based interface with information rendered on a timeline using bitmaps. Events and system details are summarised in an XML format. The information monitored is low-level and system based, and not that of higher-level middleware or applications.

2.3 Discussion

The systems listed earlier in this section are all proficient and used widely within sections of the parallel and distributed computing communities. To fully understand why an application executing on a distributed system is not behaving as would be expected it is important that not only the application, but also the underlying middleware, and the operating system are analysed too, otherwise problems could be missed and certainly overall performance diagnoses would be harder to understand.

To obtain the necessary information to undertake a full investigation of a distributed system, its middleware and applications, the following would be necessary with the tools just outlined:

- With Netlogger, calls would have to be added to all the Python, Java, C and Perl programs, the output events would be written to Log4J/Commons-based log files, these files could be subsequently viewed with the NetLogger Visualisation tool.
- SvPablo would either automatically or manually instrument the applications; as well as use hardware counters via the PAPI API [11], the output would be written to a file and the resulting traces can be visualised and analysed via its GUI.
- TAU would be used to instrument Fortran, C, C++, Java and Python-based applications, the output would be written to a file and then the output would be visualised via ParaProf.
- Ganglia gathers low-level operating system-level information, marks it up in XML, and the information gathered is visualised via a Web browser. Ganglia is an example of a suite that can provide useful system information, however, it does not integrate middleware and application data. Moreover, Ganglia does not help identify issues or problem in systems, this is the task of the system administrator who must analyse output graphs.

With three of the systems just mentioned (NetLogger/SvPablo/TAU) in order to obtain a complete view of a distributed system and its applications, it would be necessary to instrument the application, all the middleware and the operating system. This would be an almost impossible task as it would involve instrumenting hundred of programs, and then visualising and trying to understand the resulting output from thousands of events. Moreover, Ganglia, just gathers operating system-based data, it provides no tools for profiling or analysis. As it can be seen from this brief review of the most popular profiling and analysis systems; they are designed to be used with mainly homogeneous parallel systems and software, rather then the heterogeneous and loosely couple distributed systems.

2.4 The Semantic Web

The Semantic Web is a vision of what the Web could become. The World Wide Web Consortium (W3C) describes the Semantic Web as "a common framework that allows data to be shared and reused across application, enterprise, and community boundaries" [12]. In the Semantic Web, information must be recorded and communicated in such a way that a computer can interpret whether a particular item of data is relevant to its task. Semantic Web technologies allow data to be augmented with machine-readable information, so that they can do more useful things with it. The Semantic Web is an evolving framework and the technologies that it consists of are often described in terms of a stack, whose structure of allows independent improvement and innovation at each layer.

We believe that the concepts and technologies that underpin the Semantic Web can provide mechanisms for unifying and storing the heterogeneous information held in the distributed logs. These technologies have the advantage that:

- Information in any language can be encoded,
- There is a flexible, scalable, universal addressing system,
- The XML format can provide serialisation for data transfer between systems,
- The graph-based data model of the Resource Description Framework (RDF) can, in theory, represent any data structure,
- RDF Schema and the ontology language (Web Ontology Language - OWL) provide an extensible system through which vocabularies describing data types and data structures can be formally defined,
- Also, the basic inference concepts of OWL may allow conceptual unification of data.

The following section gives an overview of Semantic Web technologies.

2.4 The Semantic Web

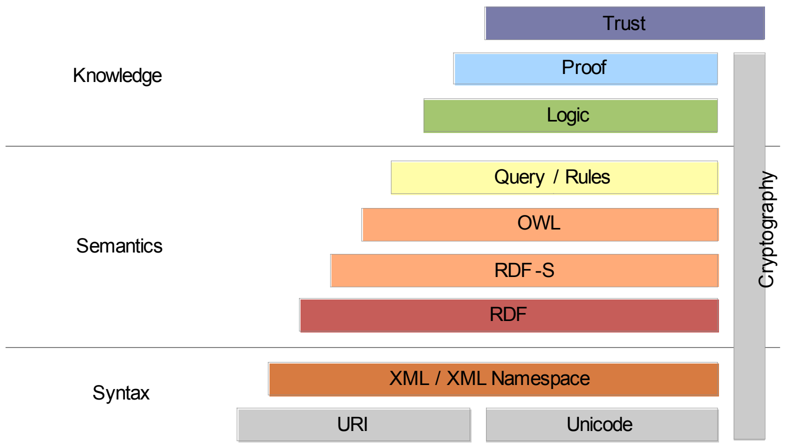

Figure 1: The Semantic Web Stack

The layers of the Semantic Web are shown in Figure 1. The layers have been categorised broadly as relating to syntax, i.e. the representation and identification of raw data; semantics, the mechanism for describing the meaning and structure of information; and knowledge, which covers mechanisms for machine reasoning based on semantically encoded information. We are particularly interested in the semantic layer of the stack in our research.

2.4.1 Representing Data in RDF

The W3C RDF Primer [13] describes RDF as a language for representing information about resources. It can represent information about things that can be identified on the Web, even when they cannot be directly retrieved from the Web. RDF is a conceptual graph-based data model. In RDF, information is constructed by defining statements [14] that consist of a node, an arc, and another node, which know as the *subject*, the *predicate* and the *object* (see Figure 2).

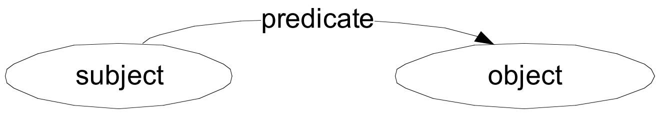

Figure 2: An RDF statement

This basic construct can be used to represent any data structure. For example, a database table has rows and columns, each row is identified using a unique key, and within each row, its column ascribes the meaning of data. The same structure can be represented in RDF. A URI defines the unique key for the row, and each column (property) is described by a URI. The combination of unique key (subject), column (predicate) and value (object) makes up an RDF statement. By creating a statement for every data value, the database table can be modelled.

2.4.2 The RDF Schema (RDFS)

Whilst RDF provides the ability to create a graph of data, it is the RDF Schema [15] that provides the building blocks for more complex vocabularies by enabling the definition of data-types and structures within an RDF graph. RDFS provides "all that is needed for interoperability of the vast amount of data on the web" [16]. RDFS defines the basic classes such as Resource, Property and Literal, as well as useful properties such as type, label and comment. RDFS also defines the concept of classes, subclasses and properties, and introduces concepts such as Bags, Sequences, Lists and Reification. An important feature of RDFS is that it is written in RDF, and (commonly) exists in the same data model as the RDF data they describe. Data structures and formats can therefore be amended or enhanced without any need to adjust database schema.

2.4.3 The Web Ontology Language (OWL)

Broadly speaking in computer science an ontology is seen as a data model that represents a set of concepts within a certain domain, as well as the relationships between those concepts. The ontology is then used to reason about the objects within that domain. For our purposes, RDFS can be used to describe structures and data that are sufficient for many uses, however, it lacks some of the capabilities that may be required for some data structures. For example, there is no concept of cardinality, so it is not possible to create a Tricycle class that requires its instances to describe exactly three wheels. The Web Ontology Language (OWL) [17] extends RDFS and provides a more descriptive schema layer that can be used where the basic definitions afforded by RDFS are not expressive enough.

2.4.4 Rules/Query Layer

The Rules/Query layer encompasses any system for querying semantic data and applying the basic rules defined in the two schema layers; it is where information and knowledge can be inferred. There are several co-existing query systems [18], most of which use an SQL like mechanism to query and filter data. A W3C working group is working towards a standardised query language called SPARQL [19].

3. System Design

3.1 Data Modelling

As a recap, the overall idea behind this work is that we can capture logs of different types from within a distributed system and use these to analyse and profile the system itself and the applications that it hosts. Therefore, it is necessary to consider four general types of log file:

- Application: Executing applications often record runtime events and debugging information into log files. It is common for these logs to be generated by logging libraries [20], which have the benefit of simplifying the creation of the more common logging formats.
- Middleware: These logs provide details about what has been happening in the middleware layer. This may include application server logs, such as those of a service container such as Apache Tomcat [21], or more specific communication logs generated by message passing libraries, such as MPICH [22], which may record details of messages received and transmitted.
- System: These are created by operating systems and their daemons. For example, in GNU/Linux and BSD the syslog daemon [23] is used to capture and record logging messages from processes.
- Bespoke monitors and probes: These may examine a system without necessarily being part of it. For example, applications that record system statistics such as CPU load or free memory, or network monitors that record the number of collisions in a subnet.

These different log files all record sequences of events. Each event may be something that has occurred in the system that is worth recording, or in the case of a system monitor, the event may be a measurement of some system properties. In a distributed system, events occur on different nodes, so the description of an event consists of a time, a location and a description. This requirement forms the basis for unification of different logs.

3.2 Transformation Requirements

The majority of log data formats offer terse and barely human readable content that requires the ability to infer meaning where it is often neither implied or explicitly stated. For example, a human reader who understands the purpose and provenance of a log-file may understand that the message “freemem 300K” is of concern. However, without an explicit definition of the semantics of that message a machine cannot interpret it on behalf of the user. The explicit discovery, extraction and semantic mark-up of data contained in logs are therefore essential to their automatic interpretation and analysis. Transforming different types of log requires domain specific knowledge, in order to extract the appropriate information from each log record. It may also require specialised vocabularies in order to classify the extracted data.

3.3 Time Requirements

Log events of different types, on systems, may differ in their timing granularity. For example, Web servers typically provide a coarse timing resolution of 1 second, whilst instrumentation logs that focus on performance may record their events with millisecond or nanosecond resolution. Where granularity is coarse-grain, the ordering of rows within the file may imply event ordering. Furthermore, records from different systems may not correlate chronologically due to poor or non-existent clock synchronisation. Analysis and visualisation rely on the correct temporal ordering of log records across different machines

3.4 Visualisation Requirements

The parallel performance profiling and debugging tools reviewed earlier in this paper collectively fulfil many of the basic requirements for the analysis of distributed systems. However, the tools lack the adaptability required for unifying heterogeneous data and can typically only help to analyse a limited range of data types using a visualisation tool that has been written for a specific purpose. This narrow, specialised focus reduces the usefulness of the tool, and limits their widespread use. Many existing tools provide duplicate functions instead of providing a generalised solution. For example, Jumpshot, NetLogger, and Vampir, provide mechanisms for plotting events against a timeline, which was identified as a useful means of validating whether heterogeneous data is correctly unified. Timeline visualisation of unified data can be achieved by either exporting data to a format that can be used by existing tools, or, by creating a visualisation interface.

3.5 The Proposed Solution

We propose that Semantic Web technologies will provide the necessary capabilities for Slogger's mark-up and unification of the heterogeneous log data, in particular:

- Unicode allows text in any language to be incorporated, a key requirement for integrating log files from a diverse global resources.
- The use of URIs to address each data item is flexible and scalable.
- XML provides a widely accepted vehicle for text-based data serialisation; this can enable unified data to be stored as flat files or communicated to other systems.
- RDF is a graph-based data model capable of representing any data structure. RDF is a conceptual data model, so it can be implemented in different ways without affecting the operation of other Semantic Web technologies. For example, the Semantic Web stack on small systems may utilise an in-memory data model, whereas in larger systems a database-backed model may be necessary. In very large systems, a distributed database could provide programmatic access to data.
- RDF Schema can be defined within RDF so data structures and types can be adapted as the application demands.
- OWL provides potential for mathematical analysis of systems, and allows basic logical inference.

4. Outline System Framework

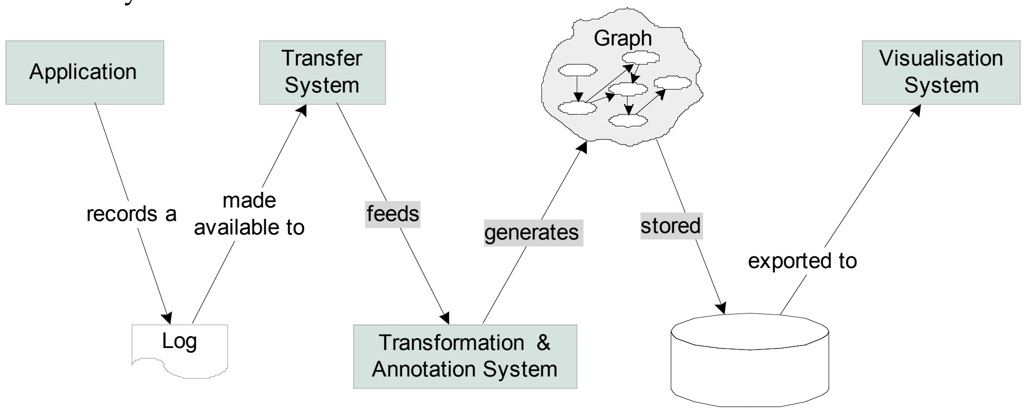

Figure 3: The flow of data from the actual logs to the visualisation phase

The Outline Framework (see Figure 3) models the flow of information from its point of creation, to visualisation. Each functional capability was considered as a separate module, allowing independent development and evolution of distinct sub-systems, these are show in Figure 4

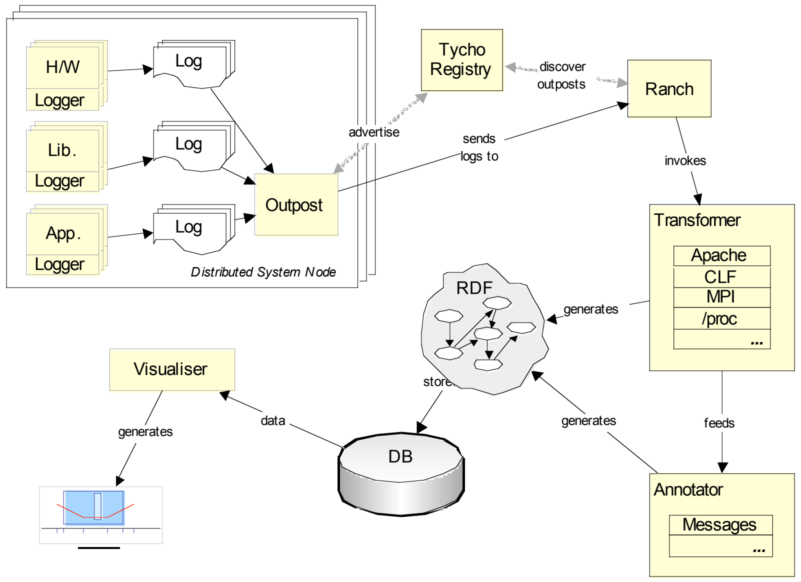

Figure 4: The Slogger Framework

4.1 System Components

4.1.1 Log Date Transferral

The Transfer Component consists of a number of *Outposts* and a *Ranch*. Outposts advertise log availability, prepare logs for transfer and send logs to the ranch. The Ranch searches for outposts, discovers available logs, arranges their delivery, and passes the retrieved log to the Transformer. Once a log has been generated, it must be moved from its initial location to where it will be transformed. There is a technical burden of discovery and transfer of data from multiple nodes that may be subject to security measures such as firewalls. Also noteworthy is the problem that log files may be large but only a small time-slice may be of interest to the investigator. For example, if an error is known to have occurred within a specific period, the ability to transfer only log events that relate to that period may be necessary. The design of the Transfer Component is such that the ordering of the transfer and transformation stages could potentially be switched, allowing the transformation process to be distributed within the observed system. Solving the administrative details of data transfer, for example, access control and establishing trust across boundaries, are not a focus of this work.

4.1.2. Transformers

Transformers are programs that convert log data into RDF, adding semantic mark-up that describes the meaning of each event contained within each log record. A transformer manager must recognise the type of each log that it receives so that the appropriate transformer can be selected. This may be achieved by parsing the log content to discover patterns common to specific file types, or by examining a log description that is transmitted by the Outpost. Once an appropriate transformer has been selected, it must process every line of the log. For any log file, two levels of transformation can occur. First, every log entry is described in terms of a Generic Unified Log Format (GULF) schema, which provides a basic vocabulary for describing the properties, and structure common to all log files. Second, the content of the log entry may be more precisely described using specialised schema.

4.1.3 The GULF Schema

The review of monitoring and profiling tools identified that the NetLogger file format provided an extensible basis for generic log description. We also considered other logging frameworks, for instance, the GGF-DAMED Top-n Events [24]. The GULF Schema defines five basic items that can be used to describe every log entry; these are outline in Table 1.

Property	Description
timestamp	The number of seconds elapsed since the epoch (00:00:00 1st Jan 1970).
nanopart	The number of nanoseconds elapsed within the second designated by the timestamp (if specified or calculated).
target	A unique identifier that describes the source of the event. This can be free text, which can be matched against other free text, or a URI in which case the protocol gulf:// should be specified, followed by the host IP address (or DNS name if non-ambiguous) then a path that identifies the resource, e.g. `gulf://Comp00/cpu.user` `gulf://Comp00/jvm.thread.0001`
Host	The IP address or hostname where the event occurred (overlaps with "target").
message	The content of the original message after time information has been removed.
date	The date in whatever format the original log-file defines it.

Table 1: Properties in the GULF schema

This format encodes no semantics other than formally identifying the content of the message and giving it a common timestamp, however, this in itself is an important inclusion in the graph because it ensures that the log can be reinterpreted after the initial transformation. This constitutes a lossless record of the original log entry so that subsequent re-interpretation is possible, if required. For brevity a `QName` of "`g:`" is used when referring to properties in the GULF schema. There is some overlap between the function of `g:host` and `g:target`, which arises from expressive nature of URIs.

Time	Message
Thu 17th Feb 2005 11:30:01	bd /tmp/eer
Thu 17th Feb 2005 11:30:01	gm /tmp/rjb
Thu 17th Feb 2005 11:30:01	wd /tmp/eer /tmp/rjb
Thu 17th Feb 2005 11:30:02	lv hea /tmp/cpl

Table 2: A simple log file that contains event records

The log shown in Table 2 contains four records, each consisting of an event time and a message describing the event. Transformation of the first line should proceed as follows: a unique resource identifier is created (see Figure 5). The time is extracted and this is added to the graph as a `g:timestamp` property. Next, the message should be extracted and added to the graph as a `g:message` property. Finally if no further processing is necessary, the source IP address should be added as a `g:target` property. This process must be repeated for each entry in the file until all entries have been processed.

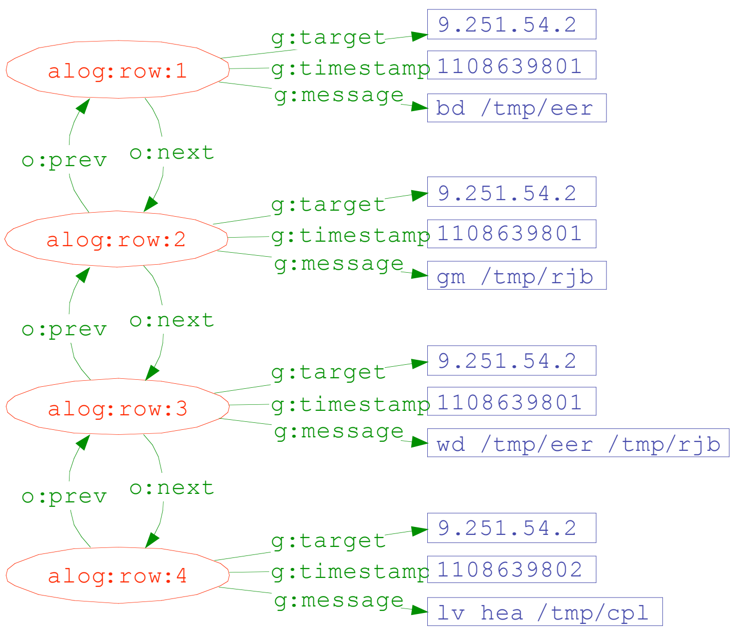

Figure 5: The log described in Table 2 after transformation into RDF.

The implicit ordering present in the file is maintained by the creation of `o:next` and `o:prev` properties provided by the GULF-Order vocabulary. In the case where times are not provided in the log file, there should be no entry made in the graph, this is true for both the nanosecond time (`g:nanopart`) and the epoch seconds time (`g:timestamp`). In Figure 5 for example, there is no nano-part because this is not defined in the timestamp of Table 2.

4.1.4 Annotation

The process of annotation is one of adding meaning to the graph as a whole as opposed to the transformation process, which imparts meaning to individual data items. Annotators are used to add data to the model in order to simplify its interpretation. Two forms of annotator may be present in Slogger:

- Manual annotations may be a feature provided by the user interface. For example, the addressability of every data item allows book marking of particular points of interest.
- Automatic annotation may occur when data is imported. After transformation, the new log data shall be passed to any annotators that are available. Annotators may parse the graph and create additional statements about the incoming data to describe an artefact that is not apparent from transforming log records in isolation. For example, when message logs are imported from several nodes the log entries represent messages that are transmitted from one node and received at another. An annotator might correlate these

individual messages, creating an entry that addresses and describes each in terms of its transmission, receipt, size and duration.

4.1.5 Storage and Indexing

Storing data in a graph provides flexibility but results in relatively large data sizes when compared to other storage systems, because the structure must be encoded within the data. We estimated that a controlled test could generate up to 500 Mbytes of graph data (see Table 3) and that this would enable the model to be manipulated in the memory of a desktop computer, and possibly avoid the need for complex data management solutions.

Assumption	Estimate
An average of 5000 lines per log-file; 5 log-files per host; up to 8 hosts	Maximum of 200,000 log records
Each log record will require on average 10 statements to describe it	Maximum of 200,000 records * 10 = 2,000,000 statements
An average statement comprises 256 characters	2m*256 = 512000000 bytes (488 Mbytes)

Table 3: Test data size - some assumptions and estimates

4.1.6 Visualisation

The Slogger visualisation of a timeline view combined some of the features found in existing systems and as such it was designed to show several types, including:

- Events, a basic record that an event has occurred, which may include common application debugging information.
- Profiles, system performance or any other numerical system state data that varies over time.
- Program Traces, using a topographical layout similar to Jumpshot.
- Descriptions of messages that pass between nodes.

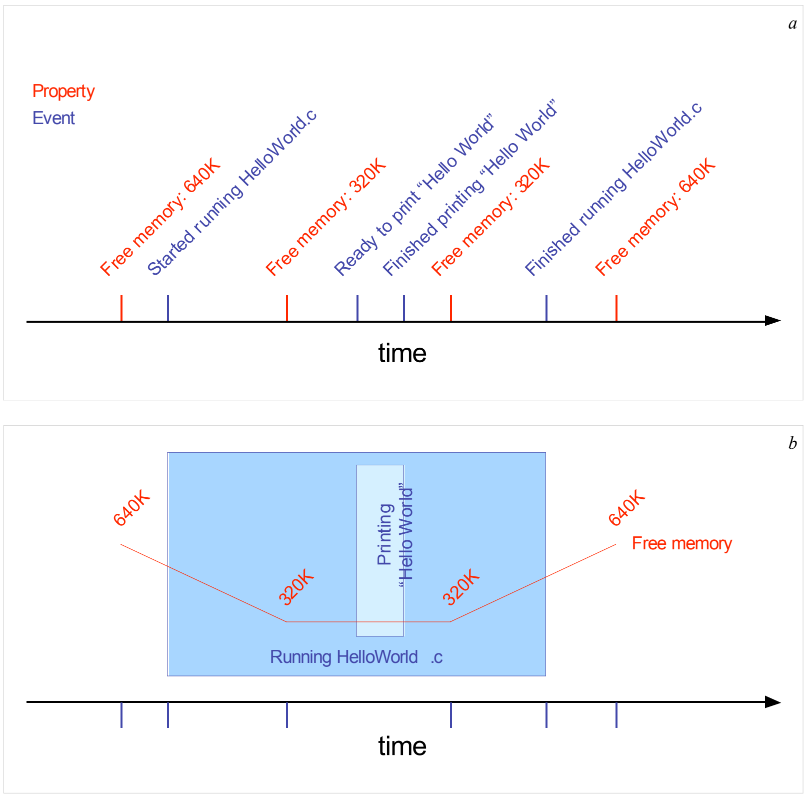

Figure 6: The draft designs illustrating (*a*) a raw data timeline and (*b*) a combined program trace and profile representing the same data as (*a*).

Figure 6 (a) shows an example timeline representing the events in a program execution. Figure 6 (b) shows how the topographical view, provided by Jumpshot, can be combined with profile data, similar to the visualisation provided by NLV, to provide a more intuitive view of the same information.

5. System Implementation

Java was chosen as the main implementation language for several reasons, the most important being the portability of Java byte-code, which increases the likelihood that Slogger can be utilised by a larger community of researchers. The availability of several free, open source RDF libraries was also influential. The Jena library [25] was selected based upon its support for several emerging query mechanisms that had the potential to assist in analysing data, as well as its successful use in other projects and the contribution of its developers to the state-of-the-art through their extensive involvement in the specification of Semantic Web standards. Jena uses the MySQL database as backing store, providing the option of directly using SQL in or embryonic RDF query languages, such as SPARQL [26].

The core functions of Slogger were developed using a test driven approach, where component development was accompanied by the creation of a test program. This helped in two ways; firstly, it kept development focused on the higher-level requirements, because when all the tests passed for a particular component, its development was complete. Secondly, and more importantly it helped trace a fundamental problem with the initial API design.

5.1 Component Implementation

5.1.1 Transformers

The transformation system was implemented as a Java interface containing a transform method that takes a stream of log data as input and returns a stream of RDF/XML that can be added to a unified model. A Combined Log Format (CLF) transformer was developed. This format was selected because of its straightforward structure and in widespread use. The Combined and Common log formats are identical, except that the combined format may include two additional fields (referrer and user-agent).

The Apache Error Log Format [20] is a generic log format used by server software of the Apache project. It complements the CLF by providing a more flexible repository for describing details of errors, debugging information and program traces. The error log format specifies only four fields: date, host, error level and an error description. The processing of the error description provides an example of extracting specific information from a log entry. If a path is discovered within the error description (discovered by the presence of slash-separated text) this is added as an extra field, complementing the other data. There may be cases where different problems have similar symptoms or properties, so enabling the identification of those relationships may be useful. To facilitate this, the path is added to the graph as a URI, rather than as a literal, which potentially provides a point for different graphs to spontaneously link when merged (see Figure 7).

MPJ-Express (MPJE) [27] is an implementation of MPI-like bindings for Java. The MPJE log records every message that is sent and received and the MPJExpress Transformer (MPJT) converts this information into RDF, using two schemas. The first of these, the GULF Message Schema that describes general message-passing properties that might be used to describe any type of message sent between any two nodes. In this case it is used as the basis for describing an MPI like message but its abstraction allows it to be used for other types of message, for instance, an HTTP request. The second schema, which is MPI specific, records the tag and context of the message. Tag and context are fields used to uniquely identify a message in MPJE. The Transformer utilises its knowledge of the log source to distinguish whether a record is describing message transmission or receipt, this allows the time field to be appropriately recorded.

Three system probes were created that record various aspects of kernel performance on Linux systems. The probes used the `/proc` file system in order to extract data at a configurable time interval, which was then written to a log file. Each of the probes had an accompanying schema and log transformer that can extract the multiple highly specific data items from each line and convert them to RDF (see Table 4).

Name	Description
MemInfo	Memory use information
VMStat	CPU, IO and swap information
LoadAvg	Long term run queue information

Table 4: Hardware monitor transformers.

A network probe was introduced to record ICMP Ping times between nodes and a Ping Transformer was created to utilise the information it records. Its role was to test the ping time between two nodes on a network.

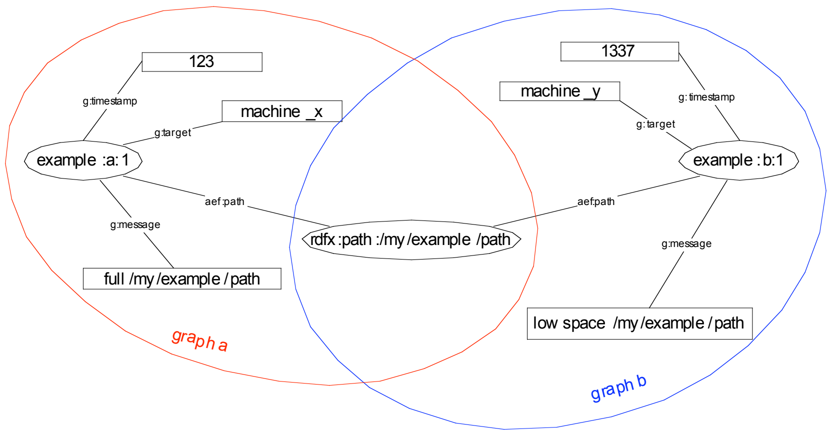

Figure 7: Identifying a common factor between different events

5.1.2 Log Transfer

In order to unify the heterogeneous log information it needed to be collected in one place. To facilitate the discovery and collection of logs, an application was developed for their transfer from node to Transformer. Whilst rudimentary, the system provides a mechanism for nodes to advertise available logs and transfer them to a host that can transform them into RDF. The system also provides a mechanism for filtering log files, so only records describing events that occur during a specified period are transferred. The transfer system was implemented in Java, which was chosen because of its availability on multiple platforms that potentially allows data collection from a diverse range of systems. We used a framework called Tycho [28] that is a pure Java system that provides a distributed registry for publishing and discovering remote endpoints and an integrated wide-area asynchronous messaging capability.

There are three participants in any log transfer process, an *Outpost*, the *Tycho Registry*, and the *Ranch* (see Figure 8):

- An Outpost is installed on each compute node and can be configured to provide details of specific logs that are available for transfer. Each Outpost in the system uses the Tycho Registry to advertise its presence,
- The Ranch queries the Tycho Registry to discover the available Outposts, and then communicates directly with each Outpost to request details of the available log files,
- When the Outpost receives a list of file requests from the Ranch it responds with details of the available log files including their name and type, which can be used to select an appropriate transformer. The Ranch then requests the files. A request may define a period of interest, which is used by the Log Filtering System.

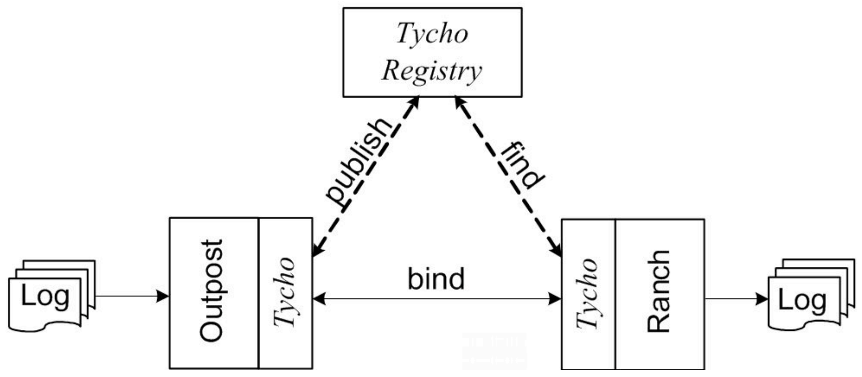

Figure 8: The log advertising and discovery process

The Log Filtering System provides two services that improve the efficiency of log transfer. Firstly, a range check efficiently discovers log files that contain no relevant events so that they can be ignored. Secondly, a binary search of the file identifies the first and last logs entries that are relevant to a specified time period. The process of opening each log file, reading each line, extracting the timestamp, and comparing it to the period of interest, is one which must be extensible, because different types of log use different means of encoding the timestamp. The performance of regular expression matching, however, was found to be a significant limitation on the overall performance of the transformation process. The range check provides an efficient means of checking for file relevance that can be applied prior to the use of the binary search to discover the exact position of relevant data. The filtering algorithm has two requirements for operation: firstly, log files must be sequentially ordered; and secondly the underlying file system must support random access files so that the binary search can be executed directly on the file without streaming the majority of content.

5.1.3 Visualisation

The literature review identified that the event-timeline concept is common to many parallel program analysis tools, and that it can provide a coherent overview of different aspects of the operation of a distributed system. The Scalable Vector Graphics language (SVG) is a W3C specification [29] for describing two-dimensional graphics. SVG was selected over other plotting libraries because it is based on XML, so languages such as PHP that can be used to generate a Web page and also generate SVG. It is suited to the creation of drawings that are based on templates, where specialised data can be added in order to finish the drawing, such as plots. SVG can be viewed using several common standards compliant Web browsers such as Firefox [30] and Opera [31], where there is no requirement to install a special client on any machine that is used to analyse data, removing a barrier to adoption. Timeline interactivity is supported through SVG's compatibility with ECMAScript [32] allowing plots to provide more information on their content. For example, by letting the mouse hover over an event marker, a full explanation can be revealed.

Figure 9 shows the four key modules of the Visualisation Component, which are:

- The Control Panel (CP) module provides an XHTML based interface that allows the user to discover and select information that is included in the time-line.
- The Timeline module, under direction from the CP, creates an SVG plot incorporating events, program trace information and hardware profiles.
- The Message Data module provides raw RDF data describing messages that can be used by the plot to render messages passing between hosts.
- Supporting these classes is the RDF Lookup Layer, a collection of several utility scripts that provides database access capabilities, transforming the retrieved RDF into objects and data structures that can be used directly by the PHP.

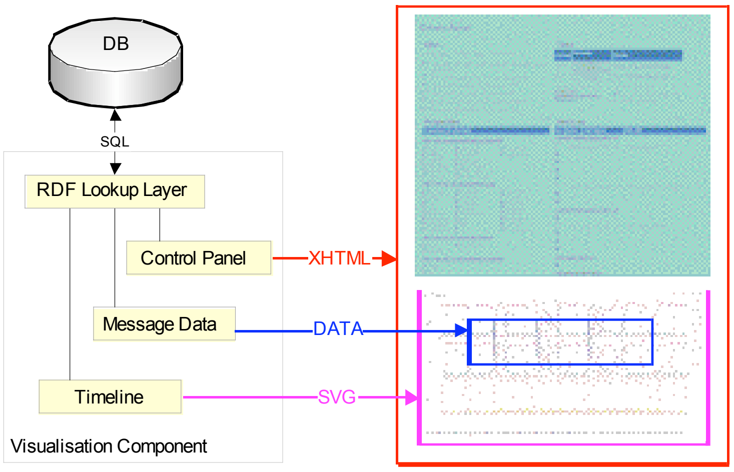

Figure 9: The main components of the SVG Visualisation Component

5.1.4 Using Slogger

The Slogger framework represents a proof of concept and is not intended as a system for deployment. It may, however, be useful to describe the necessary process that a user might expect to follow in order to utilise the framework:

- The *Outpost* is provided in a Java Archive (jar) file and must be installed on each machine that is to participate in the analysis. The log files that the Outpost will make available to the Ranch are specified in a configuration file, which details the location and file type of each log.
- The *Ranch* is also deployed as a jar file and when invoked it automatically discovers all running Outposts via the Tycho's distributed registry. Log file can then be requested and retrieved. If a date range is specified this is communicated to the Outpost so that retrieved data can be filtered before transfer from the file.
- As log files are retrieved they are put into a transformation queue. A bespoke transformer is required for each different type of log file in order to extract useful data. The semantic encoding of the extracted data will typically be based on a specialised schema that the user must create.
- When log files arrive in the transformation queue the Ranch selects the most appropriate transformer and uses it to transform the log into RDF. The resulting graph is then added to the RDF store and thereafter its data may be included in a timeline, by entering the appropriate start and finish times in the control panel.

6. Slogger Testing

The test environment shown in Figure 10, which consisted of eleven machines: *Holly*, *Nellie*, *Labs* and eight compute nodes *Comp00 - Comp07*. All the machines used were configured with dual Xeon 2.8 GHz and 2 Gbytes of RAM. Holly and the compute nodes ran GNU/Linux and Nellie ran Windows XP.

6.1 The Test Data

Within the test harness, several different types of data were captured, using different recording mechanisms. For example:

- System profile data was recorded using background scripts.
- Environmental data (for example, ping times between nodes) was also recorded using background scripts.
- Programme trace data was extracted by instrumenting the C version of the Linpack Benchmark, which was written to an Apache Error Log file.
- Middleware communications data was gathered from MPJ-Express (MPJE) logs.
- Data was transformed using the GULF schema, with several specialised additional vocabularies.

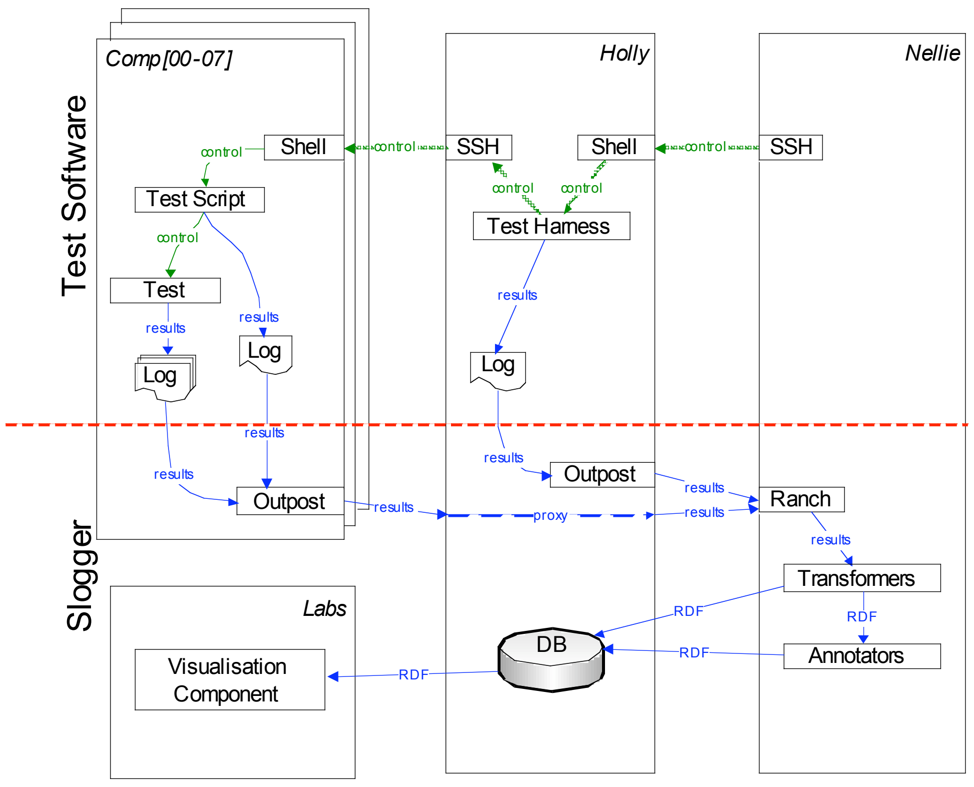

Figure 10: Test machines showing the flow of data.

6.2 Tests

Some of the tests were speculative and designed to identify a suitable basis for subsequent tests. The initial focus was exploratory and aimed at understanding the capabilities of the test platform and visualisation component, before shifting towards program behaviour analysis.

There were four main test groups:

- The Baseline tests establish how Slogger records and represents data from an unloaded, and artificially loaded, system.
- The Linpack tests establish how the operation of a sequential program appears within Slogger. Various additional loads were applied to the test nodes and the variation from the basic Linpack performance was analysed.
- The MPJ-Test introduced more nodes to test Slogger's capabilities for handling larger amounts of data, and analysing the operation of more complex situations.
- The Ping-Pong tests introduced a second node so program operation was dependent on communication between nodes. Basic operation was measured and described, then further tests were run to apply different types of load so the programs response could be analysed.

6.2.1 Baseline Test

The first test we undertook was that of establishing an absolute baseline. Here we recorded a nodes performance over a period of 300 seconds under effectively idle conditions. We expected to see CPU User and CPU System idling, memory in use not altering, and IO remaining static throughout. The results from this test were as expected. That CPU User only hits a maximum of 3%, and CPU System use hits 4%. The net result of this use is reflected in the CPU Idle time (the magenta line at the top of the plot), which appears to drop below 96% idle on only one occasion. As would be expected there was very little change in terms of memory use. This test also measure Interrupts per second and Procs RTQ. There are fluctuations near the beginning or end of the test, which were caused by the test harness itself starting up. Overall, this test showed firstly that Slogger visualisation could recognise well identified measurements, and secondly the expected performance of a "quiet" node.

6.2.2 Baseline Stress Tests

A tool was created to load nodes (stress tests) to produce specific and controllable load levels. The tool was produced because it provided a straightforward approach to generating configurable stress levels, which can be easily incorporated into the test scripts. Stress [33] (version 0.18.4) can generate four types of load:

- CPU creates n processes that each repeatedly calculate a square root;
- IO creates n processes that repeatedly write any kernel buffer data to disk (via the UNIX `sync()` system call);
- VM creates n processes that repeatedly allocate, then free, a section of memory;
- HDD creates n processes that write a configurable amount of data to the hard drive.

The stress tool was configured to create a 10 second burst of four different types of load, CPU, VM, IO then HDD. The aim was to demonstrate how each load affects a test node with no other programs running, and should result in a plot with four distinct regions of activity, interspersed with periods of inactivity at similar levels to the baseline graph. As expected, four distinct active periods can be identified in, Figure 11 marked as [a], [b], [c] and [d]:

[a] The CPU User reading rises sharply to 100% as the CPU Idle reading drops to 0%. This reflects that the stress CPU option was successful in maximising CPU use.
[b] Under VM load, CPU User peaks at around 90% with the remaining CPU time used by the system (seen in the CPU System line). There is also a sharp drop in free memory.
[c] Shows the period of IO load where CPU User remains idle and instead, CPU System takes almost 100% of the available CPU.
[d] Corresponds to HDD load, and shows that interrupts and context switches are seen to rise when the hard disk is used.

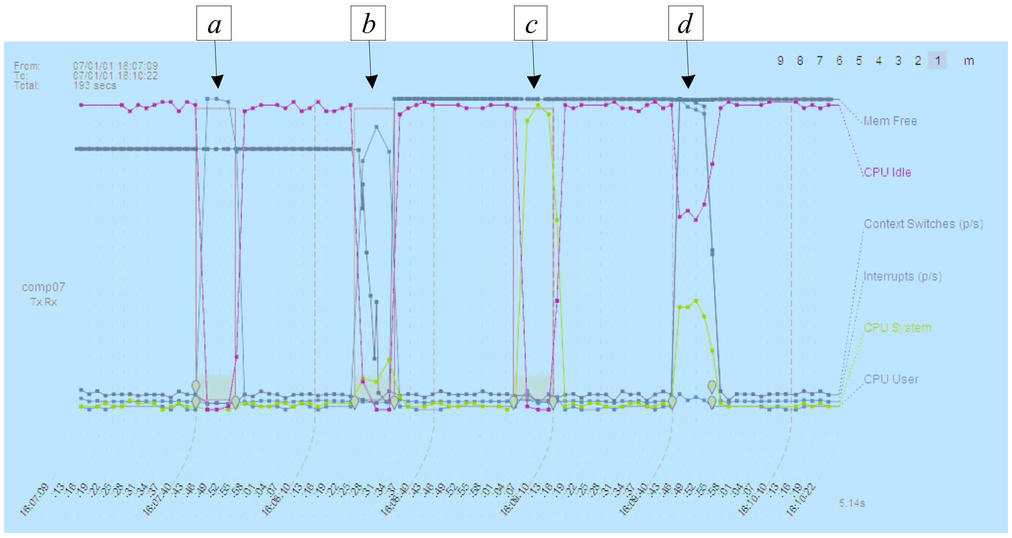

Figure 11: Stress Test Results

The area of Figure 11[b], where there is an apparent drop in free memory, is magnified in Figure 12 to show how free memory decreases from 1650.58 Mbytes to a low of 49.52 Mbytes. There is an apparent correlation between the amount of memory dropping to 64 Mbytes and a resulting drop in Memory Cache and Buffers.

The Baseline tests show that it is possible to correctly represent and visualise very specific system loads using Slogger. Firstly, an absolute baseline was established, then simple loads were applied and their affect on the system observed. Various other loads were applied, (reported elsewhere [34]) to the system and visualisations reflected the controlled variation within these loads, confirming that Slogger was creating a true representation of the operation of the system.

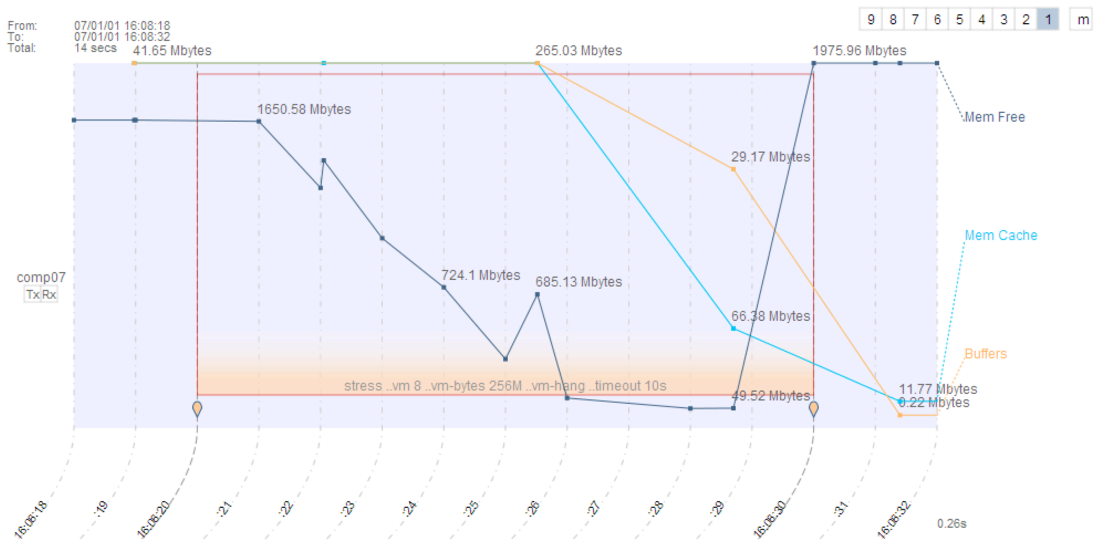

Figure 12: A simple stress test - zooming in on memory use

6.2.3 Linpack

Having established a basis for interpreting the output of Slogger through the Baseline tests, we now show the operation of a more complex sequential application. For this purpose a C version of Linpack [34] was used. Linpack performs linear algebra on a dense matrix (of configurable size) filled with pseudo-random numbers. Two modifications were made to the standard program:

- A mechanism to repeatedly load a text file *n* times was introduced. This served two purposes, it allowed disk input to be analysed and it provided a convenient mechanism for increasing the size of some of the smaller program loops, so they could be more easily seen.
- Instrumentation was added to create a program trace written to an Apache Error Log file.

The Basic Linpack tests were run in order to understand how different configurations of the same single threaded program might look in Slogger, and to establish a suitable program configuration for combining Linpack with artificially generated loads. Five tests were created with matrix sizes of 4,000, 20,000, 30,000, 40,000 and 80,000. It was expected that Slogger would show the instrumented Linpack program divided into sections, with each of the main matrix multiplication loops in a section of its own. Each loop consists of a call to the file load method, so these should be visible in the plot. The plots looked similar across all five-matrix sizes, however, their durations range from 30 seconds, to approximately 25 minutes for the fifth test. In each, the sampling frequency of the system monitor remains constant at 1 second, but appears to increase in frequency (i.e. data points are closer together) as the duration of the visible tests increases. This shows that Slogger can represent program operation over different time periods successfully.

Linpack is a computationally intensive test, so variations in CPU load provide a good indicator of changes in activity. The visualisation results, an examples is shown in Figure 13, show that the basic profile of the program does not change as matrix size increases, and that a matrix size of 20,000 lasts long enough, and contains enough detail, to be used in further load tests. The Linpack tests built on the previously established basis for interpreting output from Slogger. They have show how Slogger can be used to analyse the operation of a program, and visualise a behavioural change in the program brought about by external factors.

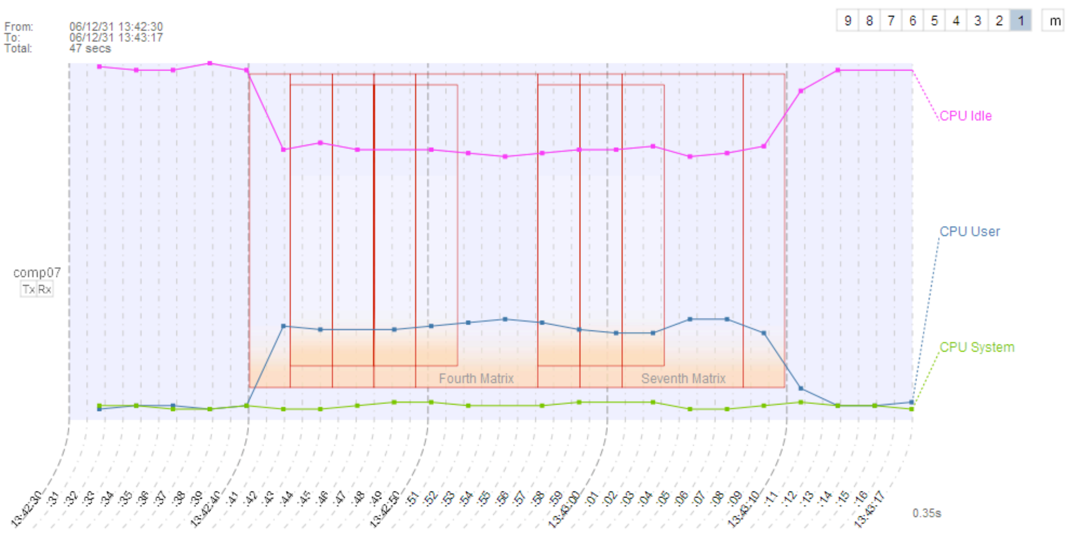

Figure 13: A basic Linpack test (matrix size 4,000)

6.2.4 MPJ-Express (MPJE) Tests

A basis has been established for analysing the operation of sequential programs through the Linpack tests. The MPJE tests identify the additional capabilities necessary to analyse a distributed program. Additional communicating processing is introduced and Slogger's capabilities for handling and presenting larger amounts of data are tested. The MPJTest program is part of the MPJE suite. It is used for checking the correct operation of each MPJE method, to see if MPJE is working correctly. With MPJTest, Slogger can be used to observe the communication on multiple nodes. In this test, MPJE was started on eight nodes and the MPJTest program configured to run all (see Figure 14 and Figure 15).

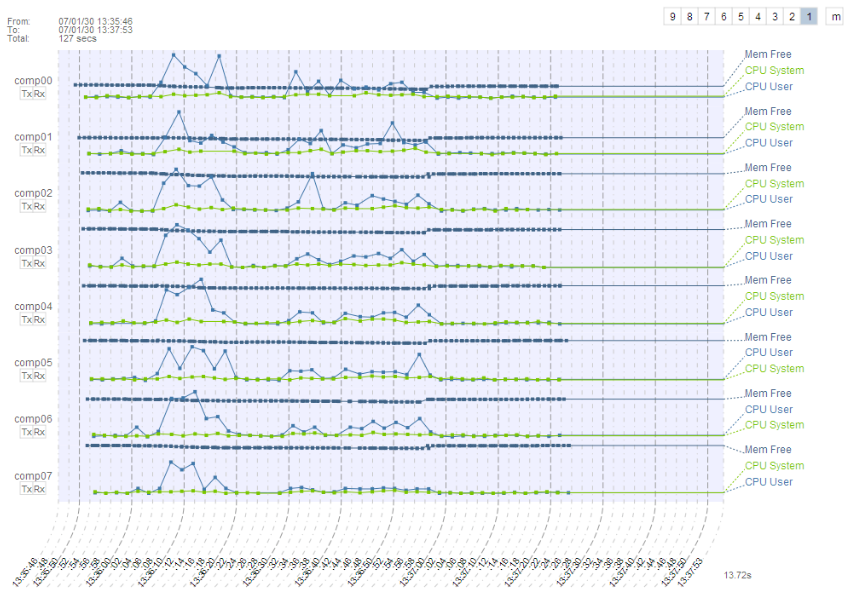

Figure 14: The MPJTest running on eight nodes

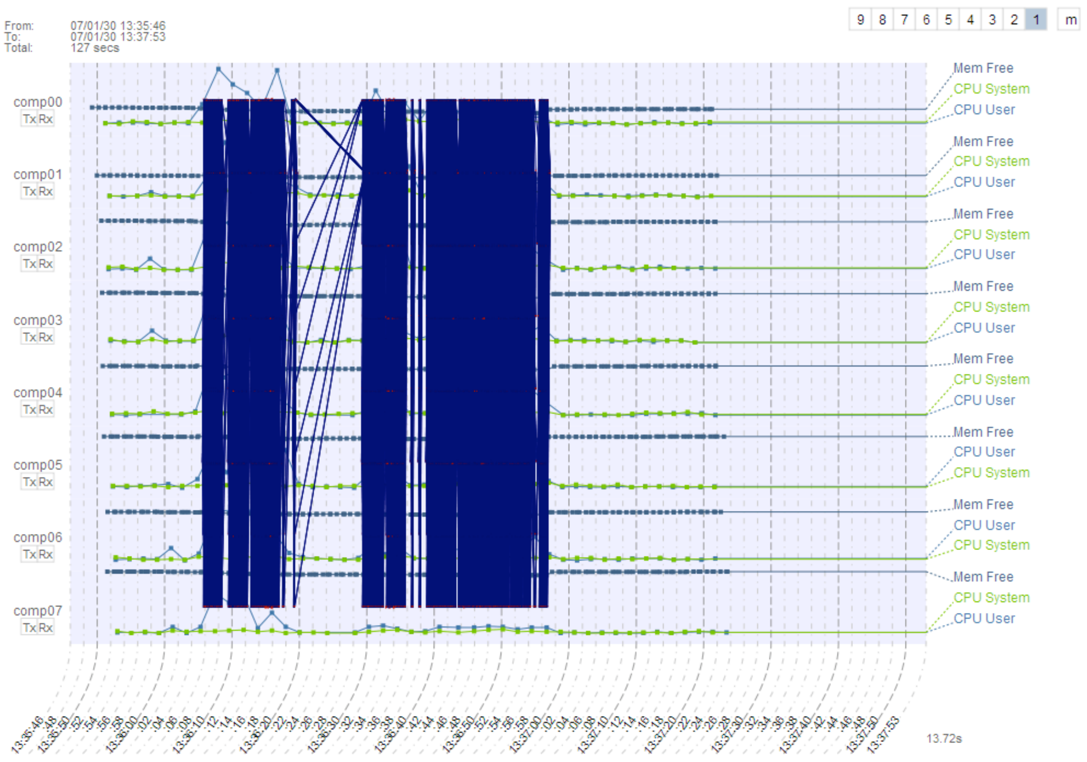

Figure 15: MPJTest running on eight nodes (showing 18,718 messages).

6.2.5 The Ping Pong Test

Previous tests have established that Slogger can be used to analyse sequential programs, and, to visualise logs from concurrently executing applications on multiple nodes. This next series of tests aimed to show that Slogger can be used to analyse the operation of a simple distributed application (a Ping Pong test), whilst different loads are applied to the system. The Ping Pong test sends messages between nodes. The size of these messages increases from 1 byte to 1 Mbyte. To measure this, an additional annotator was created that measures the average message size for all messages during the last second.

Two nodes were configured to run the Ping Pong application. This test establishes the basic operational profile for comparison in subsequent tests. The overall profile of the Ping Pong application is shown in Figure 16, and is magnified in Figure 17. The application appears to place a slightly higher CPU User load on Comp06 than on Comp07 (a peak of 96% as opposed to 74%). The change in free memory appears to be approximately the same for each node, although the amount of memory that is actually free is different. At the start and end of each Ping Pong run, "barrier" messages are sent between the participating processes. Slogger allows messages to be filtered based on the sending method or function. In Figure 16 this capability is used to show only the barrier messages. This test illustrates that Slogger can record the operation of a distributed application, and, visualise its basic operation.

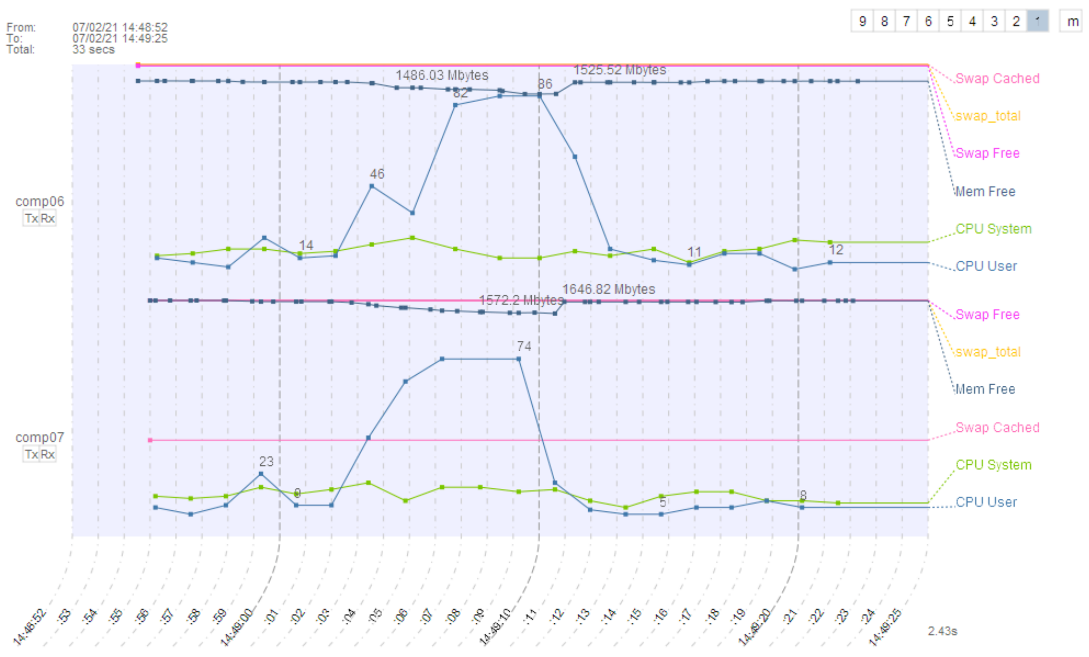

Figure 16: The Ping Pong test showing CPU and memory information

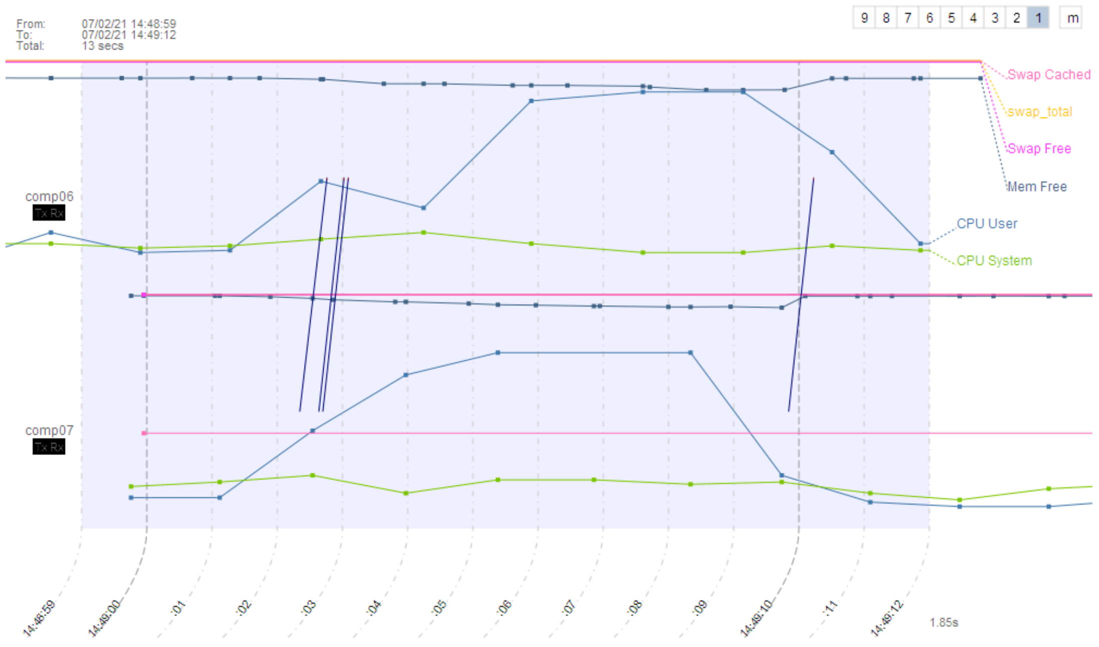

Figure 17: The Ping Pong test magnified with "barrier" messages shown

6.2.6 Testing Summary

In total over 500 end-to-end tests were executed. A small selection are presented in this paper, the rest can be found elsewhere [34]. Each test needed to be designed and written in order to generate useful and repeatable data. This data was then used to help develop the Transformers and the Visualisation Component. In this section we have presented a representative sample of those tests, that progressed incrementally from observing an unloaded node, to analysing a simple distributed application, combining data from different types of log files showing system profile data, messages detail captured by middleware logs, program topography captured from a scheduler and external data describing the state of the interconnect. The tests illustrate that, although only a proof-of-concept system, Slogger is capable of assisting post-mortem analysis of distributed systems and applications.

7. Conclusions

The aim of this research and development was to create a system that could transform and unify data from different types of log file. We wished to discover whether RDF was a suitable data model for unifying the large heterogeneous data sets that are common in parallel and distributed systems for post-mortem profiling and analysis. The capabilities and file formats of existing parallel profiling and analysis tools were reviewed, which provided a useful insight into the issues that must be tackled in order to achieve the desired unification. We chose to focus on presenting data against a timeline view, because this representation is common to many of the suites that were reviewed.

We have designed and implemented a system that can:

- Efficiently filter log files, whilst still on the originating server, in order to reduce the transfer load on source and destination systems, and reduce the amount of data that is transformed into RDF.
- Advertise, discover and transfer any type of log file to a central server for inclusion in the unified data store.
- Transform raw log data from any structured or semi-structured log format into RDF, using a combination of general and specific schema, allowing all data to be described in common terms, as well as allowing a precise definition of items described by the log event.
- Annotate the transformed data in order to provide supporting data structures that span multiple events on different nodes, for example, message descriptions.
- Plot and visualise different types of information against a timeline, with diverse types of data rendered differently depending on the most appropriate way of communicating their meaning. For example, a topographical view of program traces and lines for profiles.
- The system also utilises OWL's inference description to illustrate how data from different sources, that uses different property types, can be combined to form a single view of something measured from different systems at different frequencies.

Slogger has been developed using Java, which allows its components to be executed on any node that has a JVM. The Outpost and Ranch systems have been tested on GNU/Linux and Windows XP machines. Slogger has been, and will continue to be, developed and released under the GNU Public License. The various GULF schemas have been released under the GNU Free Documentation License.

References

[1] Netlogger Toolkit, http://dsd.lbl.gov/NetLogger/

[2] B. Tierney and D. Gunter, NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging, LBNL Tech Report LBNL-51276, 2002.

[3] SvPablo, http://www.renci.org/projects/pablo.php

[4] Autopilot, http://www.renci.org/software/autopilot/

[5] TAU, http://www.cs.uoregon.edu/research/tau/

[6] S. Shende and A. D. Malony, The TAU Parallel Performance System, International Journal of High Performance Computing Applications, SAGE Publications, 20(2):287-331, 2006

[7] Jumpshot, http://www-unix.mcs.anl.gov/perfvis/

[8] Vampir, http://www.vampir-ng.de/

[9] Paraver, http://www.cepba.upc.es/paraver/

[10] Ganglia, http://ganglia.sourceforge.net/

[11] PAPI, http://icl.cs.utk.edu/papi/

[12] The Semantic Web, W3C; http://www.w3.org/2001/sw/

[13] RDF Primer, http://www.w3.org/TR/rdf-primer/

[14] Resource Description Framework (RDF): Concepts and Abstract Syntax, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[15] RDF Vocabulary Description Language 1.0: RDF Schema, http://www.w3.org/TR/2004/REC-rdf-schema-20040210, visited 2007-03-21.

[16] Berners-Lee, T. and Miller, E., 2002, The Semantic Web, http://www.w3.org/2002/Talks/01-sweb/

[17] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, OWL web ontology language 1.0 reference, July 2002, http://www.w3.org/TR/owl-ref/

[18] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages, World Wide Web, http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf
[19] Prud'hommeaux, E., Seaborne, A., SPARQL query language for RDF, Technical report, World Wide Web Consortium (2007), http://www.w3.org/TR/2007/WD-rdf-sparql-query-20070326/
[20] Apache Logging Services Project, http://logging.apache.org
[21] Apache Tomcat, http://tomcat.apache.org
[22] MPICH, http://www-unix.mcs.anl.gov/mpi/mpich2/
[23] RFC3164: The BSD syslog Protocol, http://tools.ietf.org/html/rfc3164
[24] GGF DAMED, An analysis of ”Top N” Event Descriptions, http://www-didc.lbl.gov/damed/documents/TopN_Events_final_draft.pdf
[25] Jena, http://jena.sourceforge.net/
[26] SPARQL, http://www.w3.org/TR/rdf-sparql-query/
[27] MPJ Express, http://acet.rdg.ac.uk/projects/mpj
[28] Tycho, http://acet.rdg.ac.uk/projects/tycho/
[29] SVG, http://www.w3.org/Graphics/SVG/
[30] Firefox, http://en.www.mozilla.com/en/
[31] Opera, http://www.opera.com/
[32] ECMAScript, http://www.ecmascript-lang.org/
[33] Stress, http://weather.ou.edu/~apw/projects/stress/
[34] Richard Boakes, Semantic Web-Based Log Analysis for Distributed Systems and Applications, PhD Dissertation, May 2007, http://acet.rdg.ac.uk/~rjb/pubs/slogger.pdf
[35] Linpack, http://www.netlib.org/linpack/