Adaptive Scheduling Across a Distributed Computation Platform

Andrew Page, Thomas Keane and Thomas J. Naughton
Department of Computer Science,
National University of Ireland,
Maynooth, Ireland.
Email: andrew.j.page@may.ie, tkeane@cs.may.ie, tom.naughton@may.ie

***Abstract*— A programmable Java distributed system, which adapts to available resources, has been developed to minimise the overall processing time of computationally intensive problems. The system exploits the free resources of a heterogeneous set of computers linked together by a network, communicating using SUN Microsystems' Remote Method Invocation and Java sockets. It uses a multi-tiered distributed system model, which in principal allows for a system of unbounded size. The system consists of an n-ary tree of nodes where the internal nodes perform the scheduling and the leaves do the processing. The scheduler nodes communicate in a peer-to-peer manner and the processing nodes operate in a strictly client-server manner with their respective scheduler. The independent schedulers on each tier of the tree dynamically allocate resources between problems based on the constantly changing characteristics of the underlying network. The system has been evaluated over a network of 86 PCs with a bioinformatics application and the travelling salesman optimisation problem.**

I. Introduction

The computational demands of modern scientific research have been the driving force behind distributed computing [13], providing large computational resources cost effectively. Currently there are a few notable distributed computing platforms such as SETI@home [3], United Devices [30], and distributed.net [http://www.distributed.net], which have been built to try and satisfy the worlds increasing need for computational power, without the traditional high costs associated with dedicated parallel hardware and clusters. They work on the principle of a user donating their machine's spare clock cycles to the system across an intranet or the Internet, so that its free resources can help to process computationally large problems. The widespread success of the Internet has meant that these distributed systems have been able to harness large amounts of computational resources from donors' machines, which would otherwise have not been utilised to their full potential. These systems are generally referred to as Internet computing systems, where their resources are massively distributed across the Internet, and the problems they attempt are generally trivially parallelisable.

There are however problems with existing distributed systems which can be addressed through the use of Java. Many systems [2], [3], [21], [29], [30] are based on languages that are not platform independent, such as C and Fortran, resulting in the requirement to have multiple versions of the software, thus incurring higher maintenance and development costs. Java is a platform independent language, which allows byte-code to be generated which will run on a large variety of different platforms. The performance of Java is also comparable to languages which use machine native code [8]. Many existing systems are limited to trivially parallelisable problems, and are hard coded to perform only a single task [3], [21], [26]. Java allows new classes to be loaded or updated whilst a program is running, allowing for a programmable system to be easily created.

Security must be considered in distributed systems due to the use of insecure networks, such as the Internet, because donors put their machines under the control of others. Security is essential to protect the integrity of the results obtained from distributed systems. Many existing systems use languages that make the implementation of security difficult whereas Java has multiple security orientated API's at its core, such as the Java Cryptographic Architecture, available for the programmer to implement a secure system without the need to understand the underlying workings of the cryptography. Digitally signed JAR files, security policies, and program execution in a sandbox promote confidence in any programs running, protect results from interference, and protect the donor's machine from harmful damage.

Another problem with many existing distributed systems [2], [3], [21], [30] is that they are not extensible enough, and have fundamental limits on their scalability. They use models such as the single-tier client server model [11], which fundamentally limits the size of the system. In practice this model has served SETI@home very well, with up to four million client machines as part of the system [3]. But since there is only one server (single machine or cluster) for all of the clients there is thus a finite limit on the number of clients the system can handle at any one time, with this limit depending on the network resources and computational resources at the server. A common solution is to increase the bandwidth of the server's Internet connection and to upgrade the power of the server, but this can be expensive. Another solution, and one adopted by SETI@home, is to parallelise the computation at such a coarse level that clients (relatively) infrequently return to the server for more data units. This tactic, however, is only suitable for particular problems and might not be applied successfully to the arbitrary problems of a general-purpose programmable system.

The most commonly used communication technologies for parallel computing, such as PVM and MPI, make the task of using more complex and unbounded models harder to implement due to the requirement for more low-level development. Java however allows for the development of distributed systems at a high level, through the use of RMI and Java sockets, allowing for the creation of more extensible systems without a corresponding increased in the complexity of the users task.

Many systems have been developed to attempt to address the limitations of previous distributed systems. The Berkeley Open Infrastructure Networking Computing (BOINC) [2] system is a programmable successor to SETI@home, and attempts to make a more generalisable system. Although it is programmable, only trivially parallelisable problems are considered. Also BOINC only considers problems which will be appealing enough to get large numbers of users across the Internet to donate their free resources to the project. Its extensibility is also limited because it retains a client-server architecture, and implements a one-step processing stage. If a computation requires further processing of intermediate results, separate dedicated machines must be used. United Devices [30] provide their programmable distributed system on a commercial basis, with appealing Internet computing projects primarily being used to promote their commercial distributed system software. Their system is also limited by the use of platform dependent native byte-code, and their use of the client-server model limits the extensibility of their system. The programmable distributed systems from Krieger and Vriend [21], and Silvestre *et al.* [26] have similar aims to ours, but suffer from the same scalability problems as outlined for previous client-server systems. In addition, the native code used in their system leads to platform-dependence and serious security concerns, both of which are alleviated in our system through the use of Java.

A common limitation with the generalisable systems we reviewed [1], [2], [5], [21], [22], [29], [30] is the fact that it is not possible to run any computation on any client. If the correct client (i.e. operating system and architecture) is not available for a particular computation, then the computation will never get run on these systems. Almost every operating system and hardware architecture supports a JVM, from desktop PCs to mobile phones, and it is totally platform independent.

A number of other distributed systems implemented in Java exist [1], [22], [26], [28]. Work by Ai-Jaroodi *et al.* [1], using Java to create a distributed system, is similar in many respects to the system described in this paper, although they rely on a distributed shared memory model, which causes significant overheads in terms of synchronisation, and consistency of data. Our use of RMI and Java sockets reduces the overheads incurred, and allows for a more scalable system. Silvestre *et al.* [26] have created a distributed system in Java to process a bioinformatics problem, but their system is not programmable. Surdeanu and Moldovan [28] created a distributed JVM, providing a platform for sequential multi-threaded programs to be processed by multiple processors. Their focus is on speeding up existing programs, rather than using computational resources efficiently.

Our aim is to design a programmable distributed computing platform that is unrestricted in terms of the type or structure of computations that can be performed. A scheduler, which can adapt to the ever changing resources available to the system in a heterogeneous computing environment, is required to allow multiple different problems to be processed simultaneously. We retain aspects of the client-server model, but introduce peer-to-peer communication within a tree of scheduling nodes that serves to overcome the scalability and extensibility limitations of employing a single server. Java is used to ensure platform independence for both scheduler nodes and processing nodes. We have applied our distributed computing platform to problems from the field of bioinformatics. The software we have developed is open source and available under the GNU GPL license free of charge, and is not limited to trivially parallelisible problems, due to its ability to handle message passing between different parts of computations and the implementation of a pipeline processor.

The rest of the paper is organised as follows. In Sect. II, we introduce the multi-tiered model. In Sect. III, the designs of the main components of the system are presented. Implementation and performance evaluation are discussed in Sect. IV, and we conclude in Sect. V.

II. Overview of the System

The foundations for the multi-tiered distributed computation system were laid in the Java Distributed Computation Library (JDCL) [15] and its extensions [20], which provided MIMD capabilities through emulated pipeline processors. The JDCL provided a simple client-server based development platform for developers who wished to quickly implement a distributed computation system. It arose out of the need for a platform-independent distributed system that was simple to create, adapted to system changes, and was simple to deploy. Systems such as SETI@home did not address these issues very well and were designed to be platform dependant and for a single purpose only. The JDCL does, however, suffer from similar scalability problems to those of SETI@home in that it has one server (single machine or cluster). The design of the current multi-tiered system aims to address this fundamental limitation. An adaptive scheduler has also been developed to attempt to minimise the overall computation time of problems processed by the system, whilst also matching problems to donor machines, and allowing the user to prioritise problems.

A. Multi-tiered Model

The multi-tiered distributed computing system was created with the intention of utilising the maximum computational resources of the machines under its control, while not placing any limitations on the maximum size of the system. The client-server model alone was not sufficient, so a hybrid model was created that combines the advantages of peer-to-peer and client-server architectures within the one model. The system consists of an n-ary tree of nodes where the internal nodes perform the scheduling and the leaf nodes

do the processing, as shown in Fig. 1. The scheduler nodes communicate in a peer-to-peer fashion, which permits top-down reconfigurability, extensibility, and tree balancing. The processing nodes operate in a strictly client-server fashion with their respective scheduler, which promotes donors' trust in the system (anonymity and security) and admits a simple and robust design, through the use of Java RMI and the Java Cryptography Architecture API's.

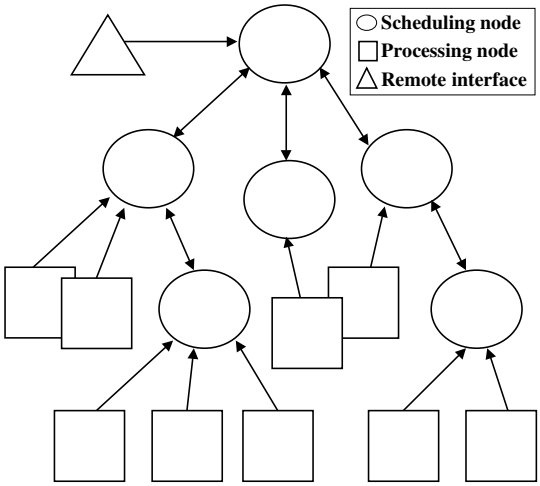

Fig. 1. Example topology of multi-tiered distributed computing system.

The potential size of the distributed system, in width and depth, is unlimited in principle due to the ability of the system to dynamically add another scheduling node, tier of scheduling nodes, or processing node. Also, since the scheduling nodes are distributed, the potential for bottlenecks is reduced, thus improving the performance of the system. The distributed nature of the scheduling nodes means that if one (other than the root) were to fail, the rest of the system could always compensate for the loss of that branch. By having multiple scheduling nodes it means that the distributed system is a MIMD architecture. Although not an increase in capability, going from a pure MISD client-server framework to a MIMD framework does increase the sophistication of the algorithms that can be distributed over the system, and therefore runs the risk of increasing the user's task in programming a distributed computation. We have attempted to structure the programmers' interface as much as possible in this regard, to find a balance between expressiveness and simplicity. For example, the programmer is required only to extend two classes to fully specify a multi-tiered distributed computation, as explained later in Sect. III.

III. Design

There are three distinct parts to the system. These are the client (processing node), the server (scheduling node), and the remote interface. The user is required to extend two classes to create a problem to run on the system, with the use of Java enabling high level abstract design compared to other more low level technologies such as C and Fortran [8]. The `DataManager` class (in the scheduler) specifies how the problem is to be partitioned into units of work and the intermediate results put together, facilitating the computation of more generalisable problems, rather than being limited to trivially parallelisable problems [2], [3], [21], [26], [30]. The `Algorithm` class (in the client) specifies the actual computation. Figure 2 depicts how a problem is treated within the system.

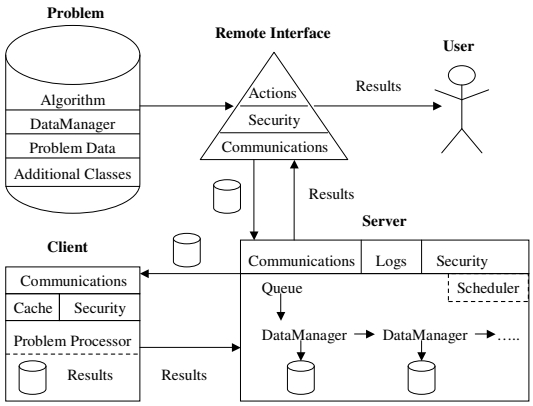

Fig. 2. How a problem is treated within the system.

The users of the system do not need any knowledge of the topology or workings of the system in order to submit problems and get their processed results back. They just provide a `DataManager`, an `Algorithm`, additional required classes, data to be processed (if required), and an integer priority weighting, to create a self contained `Problem` object, which then gets propagated to all servers in the system. A `Problem` object provides a predefined interface to the functionality that the user has provided in their `DataManager`. They can also associate minimum CPU and memory limits, to enable the scheduler to match problems to client machines with sufficient free resources (see Sect. III-E.2).

Communication within the system is based on a combination of Java RMI [27] and ordinary Java sockets. RMI is a built-in facility in Java that allows one to interact with objects that are actually running in Java Virtual Machines (JVM's) on remote hosts on a network, and avoids the need for the designer to worry about low level communication issues. By also using Java sockets for the transmission of problem data files, large amounts of data can be streamed directly from the hard disk without bringing the large files completely into memory, thus reducing the memory requirement, and improving the scalability and performance of the system.

A. Client

The main purpose of the client (processing node) software is to continually request and process data units from its server (one of the scheduling nodes). All communication is initiated by the client, providing anonymity and improved security. Once the client is started, the client attempts to connect to the server, and requests a data unit to process. Once it has received the data unit, it checks if it already has the required algorithm

in its `Algorithm` cache, and downloads it from the server if necessary. The client then breaks its connection with the server and will not contact it again until the unit is processed, the maximum processing time is exceeded for the unit and it needs to request an extension, or an exception occurs. Hence the server is always kept informed as to the status of the unit being processed by the client. If an extension request is not received by the server, it is assumed that the client has been terminated (donor has switched off his/her machine) and the server redistributes the data unit to another subordinate node. Finally, if at any time the client cannot contact the server, it will go into a 'sleep mode' and attempt to connect to the server periodically.

B. Remote Interface

The remote interface is a stand-alone application that communicates over TCP/IP, and allows the administrator to fully control all of the problems and scheduling nodes in the system. We use Java RMI communications technology for message passing between the remote interface and the server and Java Sockets for large data files, which can write data directly to disk. `Problem` objects can be added, removed, configured, and their current state viewed. The remote interface can connect and disconnect from servers, to allow them to be updated, queried, paused, and shutdown without affecting the distributed system. The remote interface also allows the owner of a particular problem to view its current state and download the results and log files, which are compressed into a single archive using Java's in built GZip API to reduce the bandwidth required.

C. Server

The server (scheduling node) is the engine of the entire distributed system, controlling subordinate servers (subordinate scheduling nodes) and clients (processing nodes) and is largely described in [20]. We have modified this system in order to allow it to be a multi-tiered distributed system. A server can be added to the distributed system while the system is running, and likewise can be removed from the system without affecting the stability of the system or causing running problems to become corrupted (the root server begin the exception). A new server contacts the server above it and synchronises itself with the rest of the system. If a server is removed, or fails, the rest of the system compensates for this loss. The lost problems are reallocated by the server, above the failing one, to others at the same level as the failing one. This resilience to failure ensures the system can perform in unpredictable operating conditions.

Units of work which are handed out by a `Scheduler` contained within the `Server`, to subordinate nodes, are cached on the hard disk using a `DiskHashTable` we developed. The built-in hash table in Java stores everything in memory, reducing scalability. The availability of cheap large hard disk storage space allows for a much more scalable server and system as a whole, by only storing the units being used in memory.

D. Scheduler

There is a scheduler in each scheduling node (server). In the multi-tiered distributed system the scheduler was created to allow a limitless number (memory limits aside) of large scale distributed computations to run in parallel on the system and to allow the system to adapt to the changing conditions of the network and computational resources available. Significant optimisations have been achieved by using adaptive scheduling in distributed systems [23], compared to static non-adaptive scheduling algorithms. The information used by the scheduler to adapt its internal scheduling mechanism is collected passively, so the scheduler can only gather information when a client presents it. This is opposed to other systems which adapt by proactively sensing system conditions [5], [7], [18], [32] or by running platform dependant third party programs [5], [32] which cause additional overheads to be incurred.

When a scheduler receives a new `Problem` object, it adds it to the end of its queue of processing problems. Subordinate nodes request units of work from the scheduler (via the server), which are generated by instances of the user-defined `DataManager` [20] encapsulated within the `Problem` objects in the queue. Subordinate clients and servers both receive the same units when they request more work, although the client processes the unit of work while the server breaks it up further into more subunits, creating a new `Problem` object for each subunit. The problem is recursively divided (see Sect. III-E.3) into `Problem` objects in this manner by instances of the `DataManager`. Processed results are sent back to the `DataManager` in the server above, and combined in a similar manner. When a `Problem` is finished processing it is removed from the queue of current `Problems`. At the root, the final results and log files are written to files and compressed for collection by the user through the remote interface.

If the processed results of a unit sent by a server are not returned within a specific period, the unit is said to expire and the server resends it to the next requesting server/client. Each unit also contains its own timer so that if the unit is close to expiring the client, or server, requests an extension from the server above.

E. Scheduling Algorithm

A scheduler has been designed which aims to adapt to the changing resources available to a distributed system and to minimise the processing time of computations. The scheduler takes into account user specified priorities for each problem, attempts to efficiently allocate and manage the available system processing power offered by the set of donor machines, and dynamically alters the granularity of data units distributed.

The efficient allocation of system processing power is loosely based on the 'matchmaking' feature in Condor [24] (see Sect. III-E.2). The dynamic adaptation of the granularity of data units that are issued by the individual problems was inspired by the dynamic window size of the TCP protocol [25] (see Sect. III-E.3). The overall scheduling strategy is brought together in the user-defined fair matchmaking scheduling

mechanism drawing from research in [4], [16], [18], [19], [24] (see Sect. III-E.1).

1) User-defined Fair Matchmaking Scheduler: Each scheduling node in the system has its own scheduler, which is independent of all other schedulers. Therefore, each sub-tree in the system has the potential for self-optimisation and has the potential to adapt to the constantly changing conditions of its own resources. The scheduler decides dynamically how the resources of the system are to be divided among the problems in its queue, employing strategies based on a user-defined fair matchmaking scheduling mechanism [4], [16], [18], [19], [24]. The user sets an integer weighting P_i for each problem $i \in \{1, 2, \ldots, N\}$ in the system. The value $\tilde{P}_x = P_x \left(\sum_{i=1}^{N} P_i \right)^{-1}$ denotes the normalized weighting for problem x over all N problems in the system.

As processed data units are returned to the scheduler for problem x, the time taken (in seconds) to process the unit τ is noted and incorporated into the typical unit processing time t_x for that problem. Rather than a simple average, t_x is calculated from $t_x = t'_x + L(\tau - t'_x)$, where t'_x is the previous value of t_x. The learning rate $L \in [0, 1]$ defines the influence of previous values, with the influence of older values tending towards zero over time. This technique is borrowed from machine learning [4], and allows scheduling nodes to continuously adapt to the constantly changing conditions encountered by the system. The value $\tilde{t}_x = t_x \left(\sum_{i=1}^{N} t_i \right)^{-1}$ is the normalized typical processing time for the data units of problem x. The scheduler tries to fairly allocate the parallel computation time resources by adopting its own internal weighting F_x proportional to P_x and inversely proportional to t_x, given by $F_x = c\tilde{P}_x / \tilde{t}_x, c \in \mathbb{R}$, for each problem x.It is this weighting F_x that ultimately defines the priority conferred on each problem x in the system in order to fairly allocate the parallel computation time.

2) Allocation and Management of Processing Power: As the title to this section suggests, there are two main aims here. Firstly the system attempts to balance the computational requirements of the problems in the system with the capacity of the individual donor machines. The second aim is to efficiently allocate and manage the total amount of available system processing power between the set of problems in the system. There are two different sets of inputs to the scheduling algorithm. The first type of input occurs when a client makes a request for a data unit (supplying its CPU speed and amount of memory available on the donor machine). The other type of input occurs when a client returns a set of results. In this case the input consists of the problem ID and how long the unit took to complete.

If a client is making a request for a data unit, then the scheduler algorithm goes through its queue of problems and queries each problem to see if the donor machine meets its minimum requirements. When a suitable problem is found that is ready to issue a data unit,such a data unit is returned to the client. If no suitable problem is found in the queue, then the client is sent a message to sleep and retry later.

If the input consists of a results set, then the scheduler records how long the unit took to be processed. After every result is received, the scheduler calculates what we call a "servicing value" (defined below) for each problem and resorts the queue of problems by their servicing value in descending order. The servicing value, $S_i \in [-1, 1]$, is a ratio of how much system processing time each problem has received compared to the priority of the problem. A servicing value $S_i < 0$ means that problem i is over-serviced, $S_i > 0$ means that problem i is under-serviced, and $S_i = 0$ means problem i is currently being serviced appropriately.

The servicing value for each problem i is given by

$$S_i = \left(\frac{P_i}{\sum_{j=1}^{N} P_j} \right) - \left(\frac{F_i \times U_i}{\sum_{j=1}^{N} (F_j \times U_j)} \right), i \in \{1, 2, \ldots, N\}, \sum_{i=1}^{N} S_i = 0,$$

where U_i is the number of units handed out so far for problem i, and where all other variables were explained in Sect. III-E.1. Problems that have received the least processing time relative to their priority will be promoted to the top of the queue and will be allocated more system time.

3) Recursively Splitting up a Problem: Any problem that is run on our distributed system must be parallelised by the user. The user specifies how the problem can be partitioned into units of work in the `DataManager` they provide. For example, this could be pairs of indices specifying subsequences of a genome to be analysed. They specify what the smallest possible unit of work can be for the problem, also called granularity, allowing the scheduler to create units of work which are multiples of this atomic grain size. This partitioning happens at each tier in the system, with the problem being broken up into smaller parts at each tier. A carefully chosen granularity can give significant performance increases [10].

Given a problem, the maximum speedup achievable is limited by the number of atomic units that the problem can be broken into, and the number of clients in the system. Techniques borrowed from machine learning [4] allow us to dynamically adjust the partitioning of data units according to the ever-changing computational resources at the disposal of the system.

Even though users specify the minimum requirements (memory, processor speed) for their problem when they enter their problem into the system, it is not possible to have a priori knowledge of the capacity of the network and the power of the donor machines. Therefore the appropriate granularity of the parallel computation will have to be ascertained dynamically. With too fine a parallelism, clients will return results after a very short processing time and might overload the network. Too coarse a parallelism might result in some processors being left idle and could cause large amounts of processing time to be wasted if a donor machine is unexpectedly switched off. After every x units are received by a problem (default value for x is 50), our system tries to modify the problem's granularity so that the average time a to process a unit will approach some

target t. If a is not sufficiently close to t, then an attempt is made to alter the granularity for subsequent units. The fraction of a (represented by d) by which the granularity needs to be altered is calculated from

$$d = \begin{cases} \dfrac{t-a}{a}, & |(\frac{t-a}{a})\times 100| > v \\ 0, & \text{otherwise} \end{cases}$$

where t is the target time, $a = \frac{1}{x}\sum_{i=1}^{N} U_i$ is the average processing time of the previous x unit times U_i, N is the number of problems in the scheduler, and v is the percentage variance threshold below which the processing times are allowed to fluctuate. The default values for v and t are 15% and one hour, respectively. A positive value for d indicates that the problem's granularity should be increased and a negative value means that the problem's granularity should be decreased. The value of d is sent to the user's `DataManager` and it is up to this code to take the appropriate actions to alter the granularity of subsequent units. Choosing lower (respectively, higher) values for x and v will allow the system to adapt more (respectively, less) quickly to changes in the network, but will cause it to be more (respectively, less) likely to overreact in the presence of transient fluctuations in network capacity and client processor power.

F. System Security

Security should be a very important aspect to any distributed system. The essence of our system is that is takes an arbitrary piece of code and sends it out to a client to be dynamically loaded. This principle alone throws up many implications for the security of the donor machine, and it is a major problem with all existing distributed systems that do not use Java [1], [2], [3], [7], [21], [29], [30]. The JVM allows us to include a comprehensive security manager in the client software. The downloaded code is restricted in the operations it can perform, by a security policy, which is a standard feature of Java. If a security exception is detected, the code is immediately stopped and the server is notified of the breach in security. In the same manner, any security exceptions on the server will result in the user's problem code being ejected from the system. One other important security mechanism in our software is that all of the system JAR files are digitally signed using the "MD5withRSA" algorithm, which is included as standard in Java. If our software is tampered with in any way, or an unauthorised client is downloaded, it will not run.

IV. Implementation

The entire system was designed using objects at a high level, thus, when it came to implementation, Java was chosen because of its object orientated capabilities and also for its platform-independence and ease of implementation. Java programs are compiled to an architecture neutral byte-code format, therefore a Java application can run on any system, as long as that system implements the JVM [14].

The computational resources available for this system's deployment were based on machines with varying operating systems, such as Microsoft Windows 98/2000/NT, and Linux distributions Redhat 7.2/Debian/Fedora Core 1/Mandrake 9.2/Gentoo 1.4, and varying hardware architectures such as those of Intel, SUN, and HP.

A. Applications

The multi-tiered distributed system has so far been used to analyse tuberculosis and E-Coli genomes searching for duplicated patterns, to break cryptographic schemes based on the discrete logarithm problem, such as ElGamal [12], using a distributed Pollard-Rho algorithm, and has been used to model light propagation in tissue using the Monte Carlo method. All of these problems are trivially parallelisable, so to show the generalisability of the system, the travelling salesman optimisation problem was also processed using the system.

B. Performance Evaluation

We carried out performance tests to evaluate the capabilities of the multi-tiered distributed system. We set out to show that adding more servers would increase the capacity of the distributed system, and we also set out to show that the scheduling algorithm would attempt to optimally balance the computational resources at its disposal. The capacity of the system is the maximum number of processing nodes in the system, where the addition more of processing nodes would reduce the performance of the system. These experiments were carried out in a computer laboratory with a dedicated network of 86 PCs. Each had a 600 MHz Pentium III processor with 128 MB of RAM and 20 GB of hard disk space and was connected to a 10 Mb/s Ethernet LAN.

Figure 3 shows how balanced the scheduler is during operation. We recorded the sum of the absolute values of all of the servicing values, $\sum_{i=1}^{N} |S_i|$, where N is the number of problems in the system, every time a processed unit was returned to the scheduler to demonstrate how balanced the set of problems was at each point in time. A value of zero indicates a balanced scheduler and is the optimal value, with higher values indicating the degree to which the scheduler is unbalanced. Initially five problems were added to the system, with two additional problems being added on three separate occasions, corresponding to the peaks in the graph. The problems used for the test were instances of the travelling salesman optimisation problem, and each problem had an equal priority of 1. The scheduler quickly moves to try and reduce the imbalance in the scheduler, thus tending towards a balanced system.

Figure 4 shows the speedup that was achieved. To show how generalisable the system is, by allowing message passing between intermediate results in the scheduling nodes, we again used the travelling salesman optimisation problem. Based on the speedup data, the average efficiency using 86 processors is 95.6%. In Fig. 4 the speedup was calculated from the equation $S(n) = \frac{P_1}{P_n}$, where P_n denotes the processing time for n processors and P_1 denotes the processing time for one

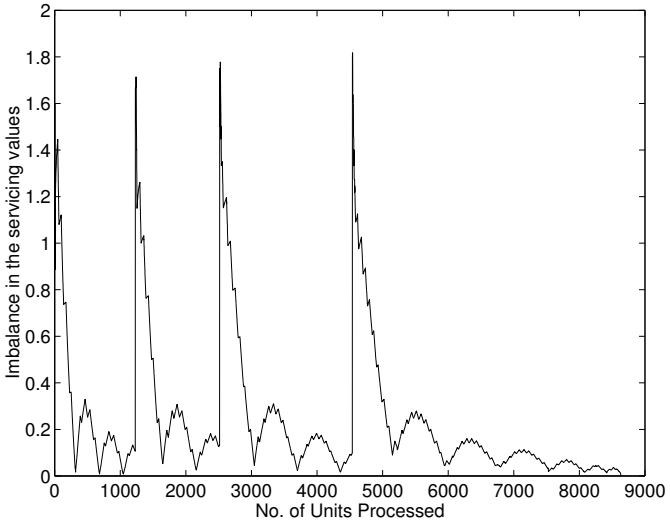

Fig. 3. Experimental results showing the scheduler dynamically balancing the system as new jobs are added.

processor, with each point on the graph being the average of a minimum of 5 runs of the experiment.

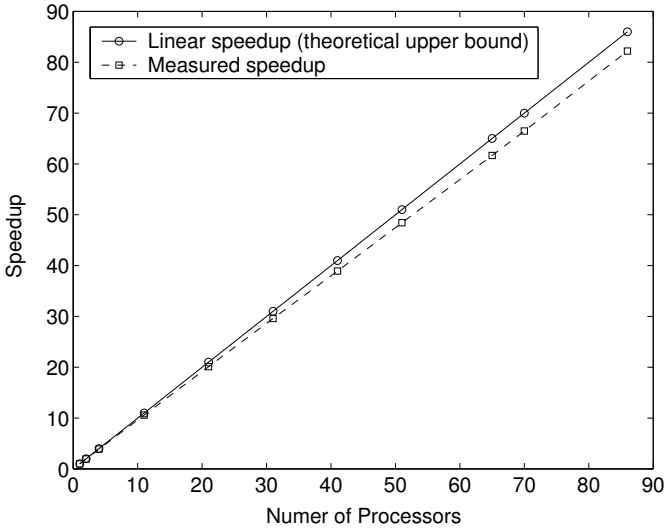

Fig. 4. Speedup achieved with a travelling salesman optimisation algorithm.

Since a server can handle in the order of thousands of clients, and such a number of machines was not available to us, we had to simulate congestion to demonstrate that the multi-tiered distributed system can increase the number of clients in a system compared to the traditional client-server topology. This was achieved by using Shunra's freeware Nimbus bandwidth throttling software [http://www.shunra.com]. The bandwidth of each server was constricted to only 14.4 kb/s, and in addition, each unit returned from a server or client was bloated with extra data to make the returned results larger. The size of this bloated data was proportional to the size of the unit. The problem posed to the distributed system was a pattern matching exercise with the tuberculosis genome (four million nucleotides in length) to find all duplicated strings within the genome.

The problem was initially run with 1 server and n clients, a one-tier traditional client-server model for several values of n. Next we ran the problem using 5 servers, arranged as 1 server in the top tier and 4 servers in the next tier, with n clients. Figure 5 shows the resulting plot for several values of n. This plot shows that after a certain number of clients, network congestion at the server causes the processing time to actually increase (approximately 20 clients, with congestion, in the case of traditional 1-server topology). As the number of servers increases, this critical number can be increased. Eventually, the multi-tiered system too reaches its capacity but it can handle many more clients than the single server system. The optimum was calculated assuming linear speedup from the timing with one client.

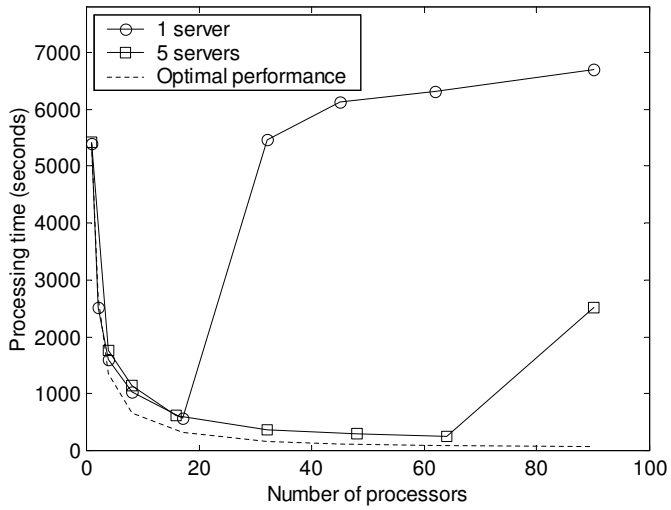

Fig. 5. Processing time comparisons with 1- and 5-server distributed computation systems, in the presence of simulated congestion.

V. Conclusion

We have examined a variety of other existing distributed systems [1], [2], [3], [5], [7], [9], [15], [20], [21], [24], [26], [28], [29], [30] and have produced a system that combines the advantages offered by each of these existing systems and overcomes many of the disadvantages of each system. Central to our success was the use of Java to implement the distributed system. Our distributed system is capable of being deployed in a typical internet/intranet environment. Some of the features of our system include a multi-problem adaptive scheduler operating independently on multiple servers organised in a hierarchical model, a remote server interface, remote updating of client software, dynamic changing of data unit sizes, and inbuilt compression of data. We also presented an adaptive scheduler that attempts to minimise the processing time of problems in the system, and balance the set of problems based on user priority and problem complexity.

Future improvements to the system will allow for the dynamic rebalancing of the topology of the system to improve

parallel efficiency, allow subordinate servers to take over superior servers in the event of failure thus providing a more robust system, and enhancement of the scheduling strategy to include a neural network which employs online learning [6]. We will also incorporate support for SQL databases, to allow more standardised and less complex message passing between intermediate results generated by problems, as well as storage of state information about problems to facilitate a more robust system such as in [7], [9], [17], [31].

The software is freely available under an open source GNU GPL licence from the system homepage located at http://www.cs.may.ie/distributed/

VI. Acknowledgement

Support is acknowledged from the Irish Research Council for Science, Engineering, and Technology, funded by the National Development Plan.

References

[1] J. AiJaroodi, N. Mohamed, H. Jiang, and D. Swanson. Middleware infrastructure for parallel and distributed programming models in heterogeneous systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1100–1111, November 2003.

[2] D. Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, pages 17–19, Madrid, Spain, November 2003.

[3] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Massively distributed computing for SETI. *Computing in Science & Engineering*, 3(1):78–83, Feb 2001.

[4] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11(1-5):11–73, 1997.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, April 2003.

[6] J. P. Bigus and J. Bigus. *Constructing Intelligent agents with Java*. Wiley Computer Publishing, New York,USA, 1998.

[7] K. Birman, R. van Renesse, and W. Vogels. Navigating in the storm: using astrolabe for distributed self-configuration, monitoring and adaptation. In *Autonomic Computing Workshop*, pages 4–13, Seattle, WA, USA, June 2003.

[8] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. In *Java Grande*, pages 97–105. ACM Press, 2001.

[9] G. Deen, T. Lehman, and J. Kaufman. The Almaden OptimalGrid project. In *Autonomic Computing Workshop*, pages 14–21, Seattle, WA, USA, June 2003.

[10] M. Drozdowski and P. Wolniewicz. Out-of-core divisible load processing. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1048–1056, October 2003.

[11] H. Edelstein. Unraveling client/server architecture. *DBMS*, 7(5):34–41, May 1994.

[12] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[13] M. J. Fischer and M. Merritt. Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2–3):239–247, September 2003.

[14] D. Flanagan. *Java in a Nutshell*. O'Reilly and Associates, UK, 4th edition, 2002.

[15] K. Fritsche, J. Power, and J. Waldron. A java distributed computation library. In *2nd International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 236–243, Taipei, Taiwan, July 2001.

[16] S. Hariri, H. Topcuoglu, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[17] M. Jelasity, M. Preuß, and B. Paechter. A scaleable and robust framework for distributed application. In *Proceedings of the Congress on Evolutionary Computation*, pages 1540–1545, Honolulu, Hawaii, USA, May 2002.

[18] C. Jin, D. Wei, S. H. Low, G. Buhrmaster, J. Bunn, D. H. Choe, R. L. A. Cottrell, J. C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, and S. Singh. FAST TCP: From theory to experiments. submitted to IEEE Communications magazine, April 2003.

[19] S. Kanhere, A. Parekh, and H. Sethu. Fair and efficient packet scheduling using elastic round robin. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):324–326, March 2002.

[20] T. Keane, R. Allen, T. J. Naughton, J. McInerney, and J. Waldron. Distributed Java platform with programmable MIMD capabilities. In N. Guelfi, E. Astesiano, and G. Reggio, editors, *Scientific Engineering for Distributed Java Applications*, volume 2604, pages 122–131. Springer Lecture Notes in Computer Science, February 2003.

[21] E. Krieger and G. Vriend. Models@Home: distributed computing in bioinformatics using a screensaver based approach. *Bioinformatics*, 18(2):315–318, February 2002.

[22] M. Migliardi, V. Sunderam, A. Geist, and J. Dongarra. Dynamic reconfiguration and virtual machine management in the Harness metacomputing system. In *Lecture Notes in Computer Science*, volume 1505, pages 127–134. Springer Verlag, 1998.

[23] F. Paganini, Z. Wang, J. C. Doyle, and S. H. Low. Congestion control for high performance, stability and fairness in general networks. submitted for publication, April 2003.

[24] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, USA, 1998.

[25] J. Semke, J. Mahdavi, and M. Mathis. Automatic tcp buffer tuning. In *Proceedings of ACM SIGCOMM '98*, pages 315–323. ACM Press, 1998.

[26] T. Silvestre, E. Nugues, G. Perrière, M. Gouy, and L. Duret. Phylojava : a generic client-server tool for phylogenetic tree reconstruction - application to grid computing. In M.-F. Sagot and H.-P. Lenhof, editors, *European Conference on Computational Biology*, Paris, France, September 2003.

[27] Sun Microsystems Inc. *Java RMI - Distributed Computing for Java*. White paper.

[28] M. Surdeanu and D. Moldovan. Design and performance analysis of a distributed java virtual machine. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):611–627, June 2002.

[29] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, A. Hey, and G. Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.

[30] United Devices. *Grid MP Platform Architecture*, 2003. White Paper.

[31] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed monitoring, management and data mining. *ACM transactions on Computer Systems*, 21(2):164–206, May 2003.

[32] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.