Performance Modelling of a Self-adaptive and Self-optimising Resource Monitoring System for Dynamic Grid Environments

Hélène N. Lim Choi Keung, Justin R.D. Dyson, **Stephen A. Jarvis**, Graham R. Nudd
Department of Computer Science, University of Warwick, Coventry, UK
Email: hlck@dcs.warwick.ac.uk

Abstract

As the number of resources on the Grid tends to be very dynamic and potentially large, it becomes increasingly important to efficiently discover and monitor them. This paper investigates ways in which the scalability and performance of Grid discovery and monitoring services can be enhanced. Benchmarks are collated and the performance analysed at the client of the MDS3 Index Service when different notification (push-based) rates are utilised. Moreover, using autonomous concepts, two self-adaptive algorithms are implemented, based on findings from the performance benchmarks. Different workload models are also used as part of several application scenarios, where the notification rate of data is dynamically modified, based on the overhead costs at the MDS3 Index Service. Experiment results are subsequently shown when varying notification update mechanisms and decision parameters are used.

1. Introduction

Grid computing [9] has acquired immense popularity as a platform which enables collaborations amongst scientists in the form of data sharing and remote processing [11]. The overall end-to-end quality-of-service of an application is dependent on reliable performance [7] obtained from each component of Grid middleware. The Grid Information Service is a core component since it enables the discovery and selection of resource entities on the Grid. Moreover, Grid environments create the implicit need for applications to obtain real-time information about the structure and state of the meta-system, to utilise this information to make configuration decisions, and to be notified when information changes. Furthermore, the heterogeneity, dynamism and complexity of the Grid infrastructure can be tackled by the emergence of a relatively new programming paradigm and management technique called autonomic computing [12].

An autonomic computing system is one which can self-manage, self-define, self-configure and self-optimise. This paper shows how the MDS3 can be leveraged to deliver timely, dynamic and up-to-date notification messages to clients autonomously, in the context of Grid performance services [13] whilst drawing on the popular OGSA-based (Open Grid Services Architecture) [10] Grid Information Service. A new approach is proposed, using the dynamic adjustment of the notification rate for preventing the GIS from overloading and improving its service performance. This is done by characterising and evaluating the performance achievable by the Globus [5] Monitoring and Discovery System (MDS3) [2] which is a widely deployed reference implementation of a Grid information service. The self-adaptive algorithms proposed make use of the collected performance benchmarks to self-optimise and self-manage. The context of this paper is therefore, the autonomic delivery of dynamic, up-to-date events to clients using the MDS3 push-based mechanism.

Related work includes that from Schopf [14] who examined the difference in scalability obtained from three contrasting distributed information and monitoring systems. While research has been previously carried out to test the performance of GIS and queries, no work has been carried out to automatically regulate their operation.

2. MDS3 Performance Benchmarking

2.1. The Monitoring and Discovery System (MDS3)

The Monitoring and Discovery System (MDS3) is the information services component of the OGSA-based Globus Toolkit GT3. It provides services for the collection, aggregation, subscription and notification of information concerning the status and availability of a large number of remote, dynamic distributed resources. The functionality of the MDS3 is encompassed into several subcomponents.

One of these is the Index Service which provides an extensible framework for accessing, aggregating, generating and querying Grid data. The Index Service allows external programs to be plugged into the framework by dynamically generating and managing service data. Data objects returned by the Index Service have different levels of consistency guarantees, depending on their level of dynamism and user requirements. Typically, the stronger the desired guarantees for a data object, the higher the overheads of consistency maintenance for the Index Service.

2.2. Experimental Setup

The experiments were carried out on a Grid testbed at the University of Warwick, which had the Globus Toolkit 3 [5] installation. The version of the GIS utilised was MDS3 since GT3 is the current latest stable version of the Grid middleware software. For example, GT3 is being adopted in the majority of the UK e-Science projects[6] and US testbeds including NASA's Information Power Grid[4].

Across the various experiments that have been carried out, an agent [8] would represent a Grid application and act as a notification sink. The agent was written using the OGSA 3.0.2 APIs [3] and it subscribed to the Index Service for changes in one of its service data elements (SDEs). These notification sinks were set up on a maximum of ten machines ($\mathcal{M}_1$ to $\mathcal{M}_{10}$). With a maximum of 500 agents simultaneously receiving notifications from an Index Service over a period of five minutes, the objective was to load-balance the notification sinks and to stress-test the notification source. Additionally, the client *M*'s machines each had the Linux operating system installed with kernel 2.4.18-27.7.x, a 2.4 GHz processor and 512 MB RAM. These machines were also on an Ethernet LAN and were connected to the Index Service host by a 100 Mb link. Furthermore, the Index Service was configured on a Linux kernel 2.4.18-14 machine with a 1.9 GHz processor , 512 MB RAM, and was running in a 4.1.27 version Tomcat container.

Performance data was collected from a set of ten experiments, and the average results are shown in the following section. This performance study will therefore allow the informed implementation of the proposed self-adaptive notification algorithms.

2.3. Performance Metrics

The following performance metrics are used to assess the performance of the Index Service:

- 1- and 5-minute CPU load averages ($\mathcal{L}_1$, $\mathcal{L}_5$);
- Percentage of CPU idleness ($\mathcal{CPU}_{idle}$);
- Percentage of free memory ($\mathcal{M}_{free}$);
- Notification throughput ($\mathcal{T}$);
- Percentage of agents receiving $x\%$ of notifications ($\mathcal{N}_x$).

2.4. Experimental Results and Evaluation

The experiments examine the scalability of the Index Service with an increase in the number of notification sinks registered to it.

Figure 1 shows the change in the 1-minute load average as the number of sinks increases; the load average is measured as the load on the Index Service during the last minute. The load average is therefore a measure of the number of jobs waiting in the run queue. It is observed that the highest load average occurs when the notification rate is 1 s. The load average peaks to just under 10.00, corresponding to 100 notification sinks, but it gradually decreases slightly for more sinks. Such observations are seen because the smallest value for the notification rate places the most load on the Index Service, and that at most 100 notification sinks can simultaneously be sent messages at the rate of once every second. On average, the load is proportional to the notification rate. However, it is observed that when there is only one notification sink, a notification rate of 30 s gives the lowest 1-minute load average of 0.026; the remaining notification rates range from about 0.06 to 0.1. Moreover, the 10 s notification rate results in the most variable load average increase over time, while the 30 s rate gives the most consistent rise in load average.

The 5-minute load average graph which is not shown here, is consistent with the 1-minute load average graph. The 1 s notification rate gives the highest average load, and the 5 min notification rate the lowest. Furthermore, for all

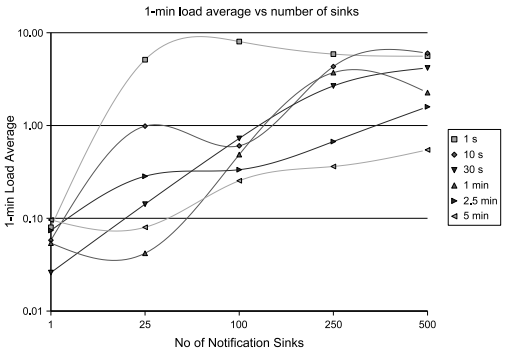

Figure 1. 1-min load average.

the notification rates except for 1 s and 10 s, $\mathcal{L}_5$ decreases as the number of sinks increases from 1 to 25. This can be explained by the sinks being serviced out of the cache, but increasing the number of sinks, further increases the load average.

The change in the percentage of free memory is shown in Figure 2. As expected, the 1 s notification rate utilises the most memory, but as the number of notification sinks increases, $\mathcal{M}_{free}$ follows an upward trend. There is approximately 20% of free memory with 500 simultaneous sinks. This indicates that memory usage is released with more notification sinks. Nevertheless, this is not the case for the other notification rates where $\mathcal{M}_{free}$ has a general downward trend. An exception to this is the 5 min notification rate where $\mathcal{M}_{free}$ is 55% at 250 concurrent notification sinks. In general, the greater the notification rate, the less memory being used.

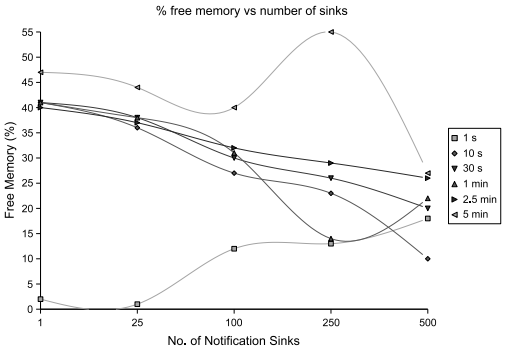

Figure 2. Percentage of free memory.

Figure 3 shows how the percentage of CPU idleness varies with an increase in the number of concurrent sinks. On average, the 1 s notification rate uses the most CPU processing power. As the number of notification sinks increases beyond 1, the CPU idleness decreases, but for more than about 25 concurrent sinks, the CPU idleness starts to increase reaching around 60% for 500 sinks. All the other notification rates produce a value of around 97% for 1 notification sink, but $\mathcal{CPU}_{idle}$ has a downward trend with more notification sinks. This trend is explained by the overhead placed in the CPU with more sinks. It is also observed that for all the notification rates, apart from 1 s and 10 s, $\mathcal{CPU}_{idle}$ stays relatively stable for under 25 concurrent sinks. However, for more than 25 sinks, the difference in CPU idleness starts to be apparent.

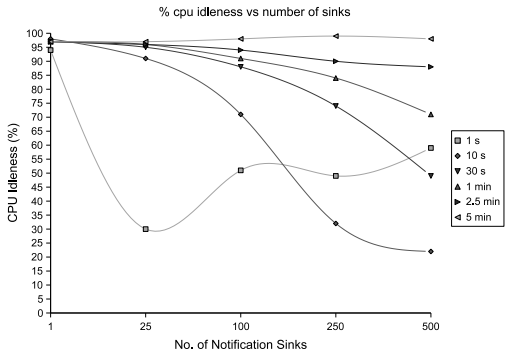

Figure 3. Percentage of CPU idleness.

The throughput performance metric is a measure of the number of notifications which the Index Service sends out on average every second. The experiment results showing $\mathcal{T}$, is given in Figure 4. All the curves, except for 1 s and 10 s, show a slight quadratic increase in their throughput. The difference in $\mathcal{T}$ with these notification rates are rather distinctive. Nevertheless, the 1 s and 10 s notification rates results both indicate a peak in $\mathcal{T}$. This occurs at just over 25 concurrent sinks with the 1 s notification rate, and at just over 250 sinks for the 10 s rate. These results suggest that there is a maximum value for the throughput and hence, the number of concurrent notifications which the Index Service can deliver per unit time. For the experiments performed, this value is just under 25 notifications per second.

The number of notifications each agent should receive throughout the length of the 5 minute experiment varies, depending on the number of concurrent notification sinks and the notification rate. On occasions, each agent can receive all scheduled notifications; on others, it receives a smaller number of messages. Figures 5 and 6 show the percentage of agents receiving 100%,

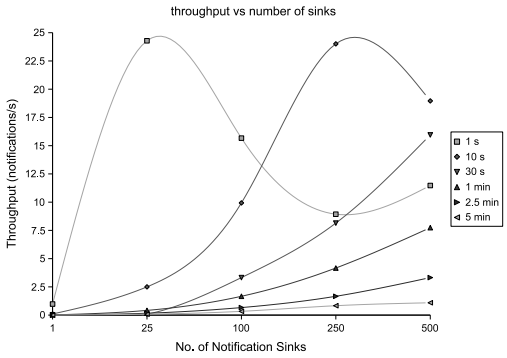

Figure 4. Throughput.

90%, 70% and below 70% of scheduled notifications when the notification rates are 10 s and 30 s respectively. These results are chosen amongst the other notification rates because they show the most variability. As can be seen in Figure 5, the expected number of notifications was received by every agent, when the number of concurrent sinks is 25 or less. Moreover, when the number of concurrent sinks is 100, 80% of the agents on average receive all the notifications, and 20% of them receive 90% of the expected number of notifications. Similarly, when the number of notification sinks is 250, only 23.6% of the agents receive all notifications, while the rest of them receive 90% of the notifications. Additionally, the QoS experienced by the agents drop significantly with 500 concurrent notification sinks where all the agents receive less than 70% of the maximum number of notifications.

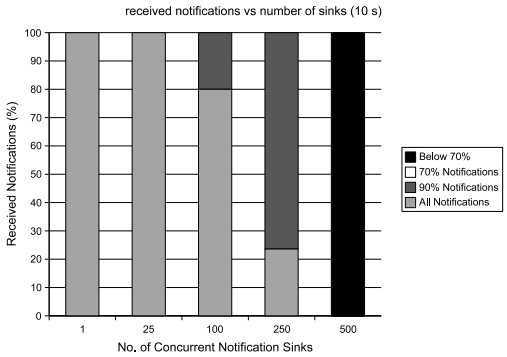

Figure 5. Percentage of received notifications (10 s).

Similarly, Figure 6 shows that with a 30 s rate of notification, all the agents receive all the notifications expected when the number of concurrent sinks is less than 25. Moreover, 21.6% of the agents receive 90% of the notifications, while the remaining of the agents receive all the notifications. This behaviour occurs at 100 concurrent notification sinks. The behaviour at 500 concurrent sinks is more variable; 61.6% of the agents receive 100% of the notifications, 34% receive 90% of the notifications and 4.4% receive 70% of the notifications.

The percentage of notifications received for the other notification rates are more stable. For example, all the agents receive all the notifications with a 1 s notification rate with less than or equal to 25 concurrent sinks. For more than 25 concurrent sinks, the agents receive below 70% of the total number of notifications. Furthermore, when the notification rate is 1 min or more, experiment results show that all the agents receive 100% of the notifications, unless the number of concurrent sinks is 500. Consequently, it can be observed that data consistency can suffer if the number of registered, active sinks increase beyond a certain value.

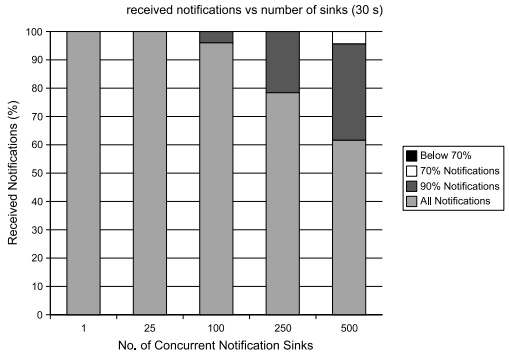

Figure 6. Percentage of received notifications (30 s).

3. Autonomous Notifications for Resource Monitoring

3.1. Self-adaptive Notification Algorithms

The above experimental results show that there is a definite trade-off between providing high data accuracy for the clients and maintaining minimal overhead at the Index Service. To prevent the Index Service from overloading, we propose policies where the notification rate is dynamically computed, based on the availability of the Index Service and the data accuracy requirements of the clients, thereby creating an autonomous system. These policies are verified using various workloads in the scenario environment. Moreover, the MDS3 performance bench-

marks showed that the notifications which the Index Service sent, quickly reached a saturation point for over 25 concurrent agents, corresponding to a notification rate of 1 s. This saturation value depends on the notification rate. The load overhead experienced by the Index Service is also directly proportional to the number of notification sinks. As this number increases, the self-adaptive notification algorithm dynamically calculates the average upper bound for the notification rate and adjusts the notification rate accordingly. Therefore, two different self-adaptive algorithms are proposed: a sink-based algorithm and a utilisation-based algorithm. The following notation is used in the description of the self-adaptive algorithms:

- *Notification rate* (NR): Represents the rate at which the service data provider is being refreshed. It is also the rate at which notification sinks are being notified.
- The performance metric which characterises the Index Service overhead most appropriately is the *CPU utilisation*.

  The utilisation-based self-adaptive algorithm will make use of the fraction of the Index Service's resources which is occupied to enforce the right decision. The occupancy of the Index Service is thus defined in terms of the CPU utilisation which has a maximum value of 100%. This algorithm also considers the number of currently registered notification sinks.

  $CU = Util_{curr}(t)/Util_{max}$
- *Number of notification sinks* ($Sinks$): The number of notification sinks currently registered with the Index Service.

The sink-based self-adaptive notification algorithm uses the current number of notification sinks to make a decision on the notification rate. It also uses the previously compiled offline performance benchmarks to deduce the optimum notification rate, given the current number of notification sinks. For each condition, the notification rate is calculated based on the optimal rate which will prevent CPU overload.

The CPU utilisation-based self-adaptive notification algorithm bases its decision to change the current notification rate on both the current number of notification sinks and an average value of CU. This average value is calculated dynamically using a moving window of the last ten CU readings throughout an application scenario. Moreover, this algorithm also uses the previously compiled offline performance benchmarks to deduce the optimum notification rate change given the current number of notification sinks. These benchmarks are used to profile the Index Service for the platform under consideration. Furthermore, decisions take into account the comparison of the current CU to its previous value. When CU is low, the notification rate is set to the optimal value which is obtained from the previously collected performance benchmarks because the load on the Index Service is low. On the other hand, when CU is high, the notification rate is decreased by an optimal value for reducing the load on the Index Service. Furthermore, when CU is in the *average range*, the notification rate increase depends on the number of sinks. The greater the number of sinks, the smaller the notification rate increase, to reduce the overall load on the Index Service.

3.2. Application Workload Scenarios

A workload is an agent application which executes as part of the application scenario which has been developed to experimentally verify the benefits and cost-effectiveness of the two proposed self-adaptive notification algorithms. Two types of workload models have been set up for the experiments. Firstly, an *increasing* workload helps to gauge the limits of the Index Service capacity. At the beginning of the application scenario, only one notification sink registers with the Index Service. Subsequently, after every 30 s, eighteen additional agents are registered for notification. A second *dynamic* workload was applied, where the number of notification sinks registered to the Index Service, is modified during the course of the application scenario. Therefore, every 30 s, the number of sinks was randomly increased or decreased by a number between 1 and 75. The starting value of the notification rate for both workloads is 1 s. Furthermore, the cycle of the self-adaptive algorithms is 10 s.

4. Self-adaptive Experiments and Results

Experiments were performed to verify the benefits of using the self-adaptive notification algorithms. During the experiments which each last ten minutes, the performance of the Index Service was monitored as the application scenario ran. Performance data was collected from a series of ten experiments, and the average results are shown in the following section. Conclusions are then drawn from the quantitative advantages of the proposed self-adaptive notification algorithms.

4.1. Experimental Results

The performance metrics: 1-minute CPU average load and the percentage of CPU utilisation, are used to assess the experimental results. Experiments have been conducted to evaluate the two proposed self-adaptive notification algorithms when subjected to the two types of workload models. The first set of experiments runs the application workload without any self-adaptive algorithm and both the second and third sets use the algorithms. Experimental results are shown, detailing the impact on the performance of the Index Service. The performance of the self-adaptive algorithms in terms of the CPU load, is analysed.

• **CPU Load Across the Experiments** Figure 7 shows how the CPU load changed with different self-adaptive algorithms and workload models.

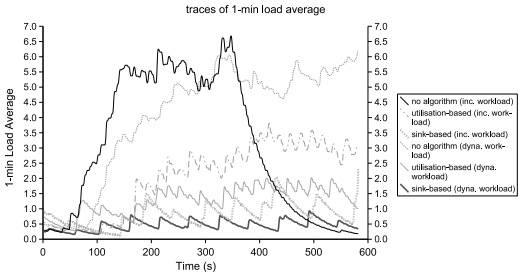

Figure 7. CPU load for all experiments.

The CPU load is measured as the average number of jobs in the run queue. For the increasing workload with no self-adaptive algorithm, it can be seen that as more notification sinks are being added at the rate of eighteen every 30 s, the CPU load generally increases to reach an average of 5.5.

When the sink-based self-adaptive algorithm is used with the increasing workload, the general trend for the CPU load is a consistent decrease followed by an increase. The periods of increase correspond to the addition of more notification sinks; the most significant increases happen when both increasing numbers of sinks are being registered to the Index Service and when the notification rate is being changed. To counter such load increases, the sink-based self-adaptive algorithm maintains a relatively small load of 0.7. In contrast, for the utilisation-based algorithm with increasing workload, the CPU load is maintained on average at about 2.8.

Furthermore, the CPU load was on average 1.3 and indicated very slight changes for the utilisation-based self-adaptive algorithm with dynamic workload. Overall, we can observe that the highest loads are produced when no self-adaptive algorithm is used. Additionally, the lowest loads are obtained when the sink-based algorithm is utilised with either workload model. This can be explained by the choice of optimal values for the notification rate which will minimise the CPU load for a given number of notification sinks, as shown by previously collected benchmarks. However, the drawback of the sink-based algorithm is that a relatively higher notification rate is needed, which is on average 65 s.

• **Application Workloads Without Any Self-adaptive Algorithm** Figure 8 shows the trace of CPU utilisation without any self-adaptive algorithm, when both application workloads are executed. The notification rate for the agents is 1 s. For the increasing workload, it can be observed that near the beginning of the experiment when there are relatively few notification sinks, the CPU utilisation is low on average. However, the CPU utilisation is on average around 50% from the start of the experiment until 66 s have elapsed. Moreover, on average, the dynamic workload shows less CPU utilisation than the increasing one which causes the CPU to saturate earlier. Furthermore, for the dynamic workload, the CPU utilisation was on average less than 10% for 13% of the length of the experiment. In contrast, the CPU utilisation for the increasing workload was on average less than 10% for 7% of the experiment duration. This can be explained by the overall greater number of concurrent sinks with the increasing workload.

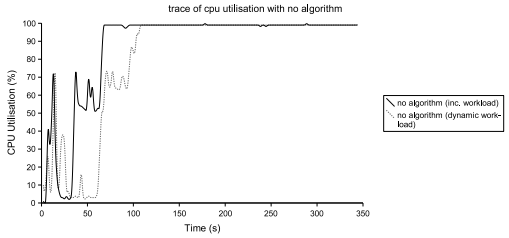

Figure 8. CPU utilisation without any self-adaptive algorithm.

• **Sink-based Self-adaptive Notification Algorithm with Increasing Workload** This algorithm has been applied to the same application increasing workload and experiment results are

shown below. Figure 9 shows how the CPU utilisation varied throughout the experiment. Figure 9 shows that the utilisation remained on average at 0% but with several spikes to almost 100%. This is due to the increased load on the Index Service which is being rectified immediately in the next cycle of the sink-based self-adaptive algorithm. On average, the CPU idleness was above 90% for 76% of the experiment duration.

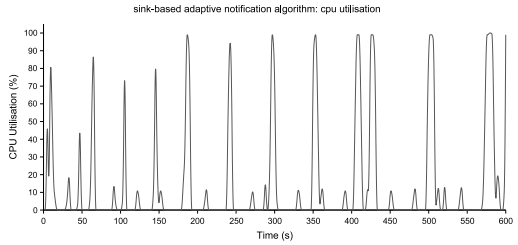

Figure 9. Sink-based self-adaptive algorithm with *increasing* workload.

• **Utilisation-based Self-adaptive Notification Algorithm with Increasing Workload** Figure 10 shows how the CPU utilisation varied throughout the experiment. On average, the CPU utilisation was below 10% for 53% of the experiment duration. The sudden increases in CPU utilisation can be attributed to the periodic registration of more notification sinks.

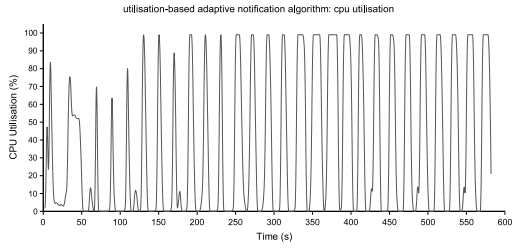

Figure 10. Utilisation-based self-adaptive algorithm with *increasing* workload.

• **Sink-based Self-adaptive Notification Algorithm with Dynamic Workload** The average CPU utilisation stayed relatively stable at around 10% throughout the whole experiment, with occasional spikes to about 90%. Figure 11 suggests the confirmation that the sink-based algorithm is an efficient method for ensuring that the CPU does not overload.

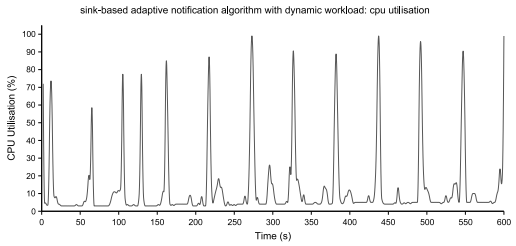

Figure 11. Sink-based self-adaptive algorithm with *dynamic* workload.

• **Utilisation-based Self-adaptive Notification Algorithm with Dynamic Workload** Figure 12 shows the CPU utilisation graph obtained with the dynamic workload. It is observed that the CPU utilisation remained on average, below 30% with frequent spikes at which the utilisation reached less than 100%.

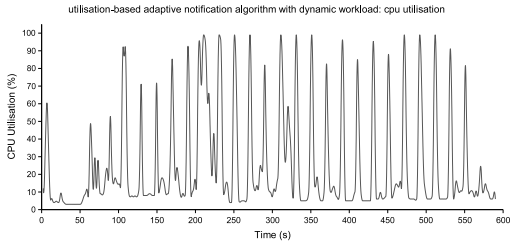

Figure 12. Utilisation-based self-adaptive algorithm with *dynamic* workload.

4.2. Experimental Evaluation

The two workload models serve to verify the benefits of the proposed self-adaptive algorithms. While the increasing workload tests the limits of the Index Service when a large number of sinks (more than 250) are receiving notifications, the dynamic workload models a more realistic behaviour where clients will register to and deregister from the Index Service. We have also found that with the increasing workload, the average notification rate throughout the experiment was 36.2 s and the average CPU utilisation was 37.16%. These results are promising because the accuracy of the service data is relatively high (updated every 30 s) and the CPU utilisation could be increased to near its maximum capacity. On the other hand, the dynamic workload showed an average notification rate of 23.91 s and the

CPU utilisation averaged 25.84%. Moreover, the average number of concurrent sinks was around 132, indicating that the Index Service can be adapted to prevent system overload, with the clients having a minimal notification rate. Such experimental results show that it is possible to increase the Index Service utilisation to a predefined value with no drastic decrease in the notification rate. Subsequently, more sinks can be registered when self-adaptation is in place.

The sink-based algorithm maintained a much lower load than the utilisation-based one, indicating that it is a reliable method for controlling the overall utilisation of the Index Service. However, this is not a scalable solution as the Index Service would need to be modelled to discover the optimum values for the algorithm. Nevertheless, the utilisation-based method results in a higher notification rate, which is useful for making sure clients receive the best possible service level. We can be confident that a combination of both algorithm mechanisms works well to ensure that the Index Service provides a reliable level of performance, whilst being self-optimising.

5. Conclusions and Future Work

This paper addressed the implementation of a self-adaptive mechanism to autonomously maximise the performance of the Index Service from a Grid application's point of view, based on previously collected performance data benchmarks. We also exhaustively verified the efficiency of these algorithms using two different workload types: an increasing and dynamic workloads. We have shown that both workload models proved to benefit from the proposed self-adaptive algorithms; the consistency margin of the data being monitored is relatively low and a fairly large number of concurrent sinks can be supported by the Index Service.

Future work will include using the experimental results obtained from the application scenarios to implement a further application workload where the notification sinks will have specific data accuracy needs. Additionally, the effect of external sources, such as the network latency, will be taken into consideration. The objective of the self-adaptive and self-tuning feedback algorithm will be to meet clients' specific quality-of-service (QoS) requirements while minimising costs at the GIS. The results obtained in this paper can also be extended to emerging implementations including the Globus Toolkit 4 (GT4). This is possible because the Web Services Resource Framework (WSRF) [1] specification, on which GT4 is based, is a re-factoring of the Open Grid Services Infrastructure (OGSI).

6. Acknowledgements

This work is sponsored in part by funding from the EPSRC e-Science Core Programme (contract no. GR/S03058/01).

References

[1] From OGSI to WS-Resource Framework: Refactoring and Evolution. http://www-106.ibm.com/developerworks/library/ws-resource/ogsi_to_wsrf_1.0.pdf.

[2] GT3 Monitoring and Discovery System (MDS3). http://www.globus.org/mds.

[3] GT3 OGSA APIs. http://www-unix.globus.org/toolkit/3.0/ogsa/impl/java/build/javadocs/.

[4] NASA's Information Power Grid (IPG). http://www.ipg.nasa.gov/.

[5] The Globus Alliance. http://www.globus.org.

[6] UK eScience Programme. http://www.research-councils.ac.uk/escience/.

[7] F. Berman, G. C. Fox, and A. J. G. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003. page 557.

[8] J. R. D. Dyson, N. E. Griffiths, H. N. Lim Choi Keung, S. A. Jarvis, and G. R. Nudd. Trusting agents for grid computing. In *Special Track, held as part of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2004)*, 10-13th October 2004.

[9] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, 22 June 2002.

[11] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl.*, 15(3):200–222, 2001.

[12] P. Horn. Autonomic Computing: IBM's perspective on the state of information technology. *IBM Corporation*, 15 October 2001. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[13] G. R. Nudd and S. A. Jarvis. Performance-based middleware for grid computing. *To appear in Concurrency and Computation: Practice and Experience*, 2004.

[14] X. Zhang, J. Freschl, and J. M. Schopf. A performance study of monitoring and information services for distributed systems. *In Proc. 12th IEEE International Symposium on High-Performance Distributed Computing (HPDC-12)*, pages 270–281, 22-24 June 2003. IEEE Press.