Load-balancing for MySQL

by **Dennis Haney & Klaus S. Madsen**

Datalogisk Institut, Københavns Universitet

Fall 2003

Abstract

The databases of today have to handle high load while providing high availability. The solution often used to cope with these requirements is a cluster.

In this paper we implement a load-balancer for MySQL; a widely used open source database. Within this load-balancer we have implemented different algorithms to distribute load. We present benchmarks for the algorithms to show their capabilities. The metric used is throughput and response time. Benchmark workload are synthetic to highlight many aspects of balancing load correctly.

From the benchmarks and by analyzing the different algorithms we will show a simple algorithm that is platform independent, easy to implement and has the best performance of any of the algorithms implemented.

Contents

Chapter 1

Introduction

Computers can figure out all kinds of problems,
except the things in the world that just don't add up.

- James Magary

The databases of today have to handle high load while providing high availability. The solution often used to cope with these requirements is a cluster. A cluster is a group of highly interconnected database servers. Historically a server in a cluster is called a node. The cluster provide high availability in allowing transparent failover in case of failure on a node. The cluster also provide a means to distribute load, since requests can be handled on any of the cluster nodes.

MySQL is a widely used open source database. When MySQL is used in a cluster, one node is selected as master. The master's job is to distribute changes to the data to the other MySQL nodes. These nodes are named slaves. Changing data on a slave will not distribute the changes to the rest of the cluster. Therefore, the slaves are only capable of handling read-only queries. Another problem with the clustering capabilities in MySQL is that it provides no means to distribute the transaction load.

Distributing load is usually handled by a load-balancer. A load-balancer selects on which node a request is to be executed. The goal for the load-balancer is to equalize the load on all nodes, to avoid bottlenecks and to achieve maximum throughput. Some load-balancers blindly follow a specific pattern to reach this goal, others evaluate information obtained from the nodes and base their decision on that.

In this paper we focus on developing such a load-balancer. Especially we will focus on the algorithm used to select a node and the performance of this selection. Performance in this context is defined as overhead of the algorithm, response time of the executed queries and throughput.

Chapter 2 discusses where in the system the load-balancer should be placed. Chapter 3 explores different algorithms. Chapter 4 is about how to estimate the load of a node, as some of the scheduling algorithms needs this information. Chapter 5 is an overview of our implementation, highlighting parts of the different sub-systems. Chapter 6 contains benchmarks results for the implemented system, with explanations of how the different algorithms performed in relation to our expectations. Chapter 7 summarizes the entire paper, and suggests further work areas.

Chapter 2

The system

It is not enough to do good;
one must do it the right way.

- John, Viscount Morley, of Blackburn

In this chapter we will explore the different options of where to place the dispatcher. The dispatcher is the is the part of the system that determine on which node a transaction should be executed. The dispatcher may need information from the cluster, and we will also explain how to propagate this information. Choosing where to place the dispatcher, is a trade-off between throughput, security, error safety and latency.

We will start by defining the terms and notation that we will use throughout the paper. We will then define what a job is, and which properties it has. The different options for sending information though an IP network will be discussed, and lastly we will evaluate different solutions of how to pass the job from the client application to a node in the cluster.

2.1 Terms and notation

The following terms and notation are used:

Cluster	A number of nodes strongly interconnected.
Node	Server in the cluster.
Frontend server	Server placed in front of a cluster.
Client	Machine which has one or more client connections.
Client connection	A single connection from some application to the cluster.
Client daemon	Load balancing service running on a client.
Load deamon	Load balancing service running on a node/frontend.
The dispatcher	Generic term for program that chooses where a job should be executed.
Phase	The period between updates of load information.

M	Number of client connections.
C	Number of clients.
N	Number of nodes.
D	Number of load daemons.
F	Number of frontends.

2.2 The job

The job is the work that should be executed on the cluster. In the context of databases it is a transaction. A job is generated on a client, sent across the network to a node which processes the job. The node then returns the result to the client, when the job has been executed. For databases communication between node and client is usually via a TCP-connection.

The client has several choices when it needs to submit a job. First is to ask the dispatcher which node it should use and then send the job there. Second, is to send the job to the dispatcher and let it send it to the correct node. The latter usually means that the answer back must also go through the load balancer. In the following we will assume that having the dispatcher on the client uses the first option. If the dispatcher is on a frontend the second option is used. And if the choice is made on a node both options are possible.

2.3 Propagating the load

Some scheduling algorithms require information about the load of individual nodes. The information must ultimately come from each node in the cluster. Also the information must be transported from the nodes to the scheduling algorithm. We assume that the cluster and the clients communicate through an IP network. Further, we assume that the nodes are on the same physical network, but that this is not necessarily true for the clients.

An IP network has several ways of communicating. The two main protocols used are TCP and UDP. TCP is a protocol where messages are guaranteed to be delivered in order and without loss. To make this happen the two ends of the communication must go through a setup phase. For this reason, a TCP connection is always between two machines (called unicast). Another detail is that TCP is not packet oriented, so a message may span several packets. UDP does not require a connection setup phase and does not guarantee that packets sent are also received, nor that those received are in the same order as they were sent.

UDP packets are not limited to unicast. The alternatives are broadcast and multicast. Broadcast works by transmitting a packet to a special address, which all machines connected to the local network picks up. Multicast is a special form of broadcast that can work across different networks (i.e. across the Internet). Instead of the packets being broadcast to all machines, only machines that have registered for them receive them. Multicast requires a special address to transmit from. These special addresses cannot be used on the Internet, unless bought. Because of the restrains on multicast, it cannot be considered a viable communication method.

This is a very short and incomplete description of basic IP networking. Further information can be acquired from any standard textbook on basic IP networking such as [1].

The optimal usage of the different protocols depend highly on the environment they are used in. We will try to describe the options available under the different scenarios.

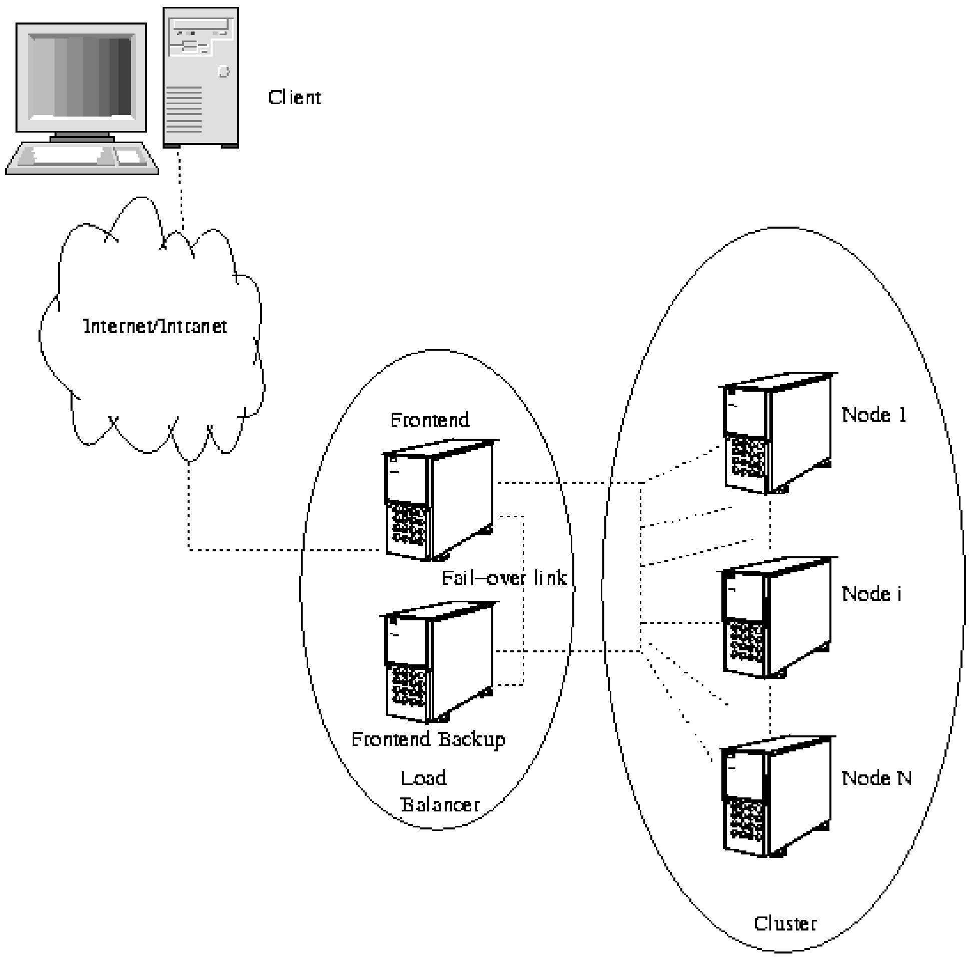

Figure 2.1 – *A setup with frontend load balancers. Only one frontend backup is depicted, but nothing prevents more from being added.*

2.4 Dispatching from a frontend

A frontend is a machine placed in front of the cluster (see figure 2.1 on the preceding page). This setup has the advantage that there is a clear single entry and exit point to and from the cluster.

In practice this setup makes it possible to use the frontend as a black box, that hides the complexity of the cluster from the clients. It is therefore possible for clients to access the cluster without the need to have the special functionality usually involved in dealing with clusters. All the complexity is then placed inside the frontend instead.

The frontend can also be used as a black box for the nodes. Then the application running on the nodes do need not implement anything special to handle the cluster situation. Again, the complexity to abstract the nodes from their reality is placed in the frontend.

What is even more interesting, is that the frontend can work as the black box for both the clients and the nodes at the same time. C-JDBC[2] has implemented such a frontend solution for databases, but only for JDBC connections (Java Data-Base Connectivity).

C-JDBC works by delegating parts of the database query to different nodes and then combining the results. Among other things, it allows many different databases (i.e. from different vendors) to work together. C-JDBC must therefore parse the query, split it up, choose the nodes to run the different parts on and recombine the results without breaking the meaning of the original query. All of these tasks are very resource demanding and the frontend quickly becomes the bottleneck. C-JDBC is so complex a solution that it is almost the same as implementing a cluster aware database from scratch. Further C-JDBC is implemented in Java, and their solution allow only clients that use the Java.

For a MySQL only system, a simpler solution would also be able to do the job. MySQL already has replicating of data implemented, so it is not necessary for the frontend to also handle the replication. With this simpler solution, the frontend would understand the MySQL protocol, and thus can act to the clients as a MySQL server. However, as the frontend needs to understand the MySQL protocol, it will still be complicated to implement.

Another way of implementing the frontend, is to make it forward the network traffic from one client connection to a node, without trying to understand the data. Usually there are three ways of doing this. First is to use NAT, a technique that rewrites network packets so that the frontend becomes the machine both the client and the node talks to. NAT is the only solution that does not require the nodes in the cluster to have access directly to the client.

Second option is to encapsulate the IP packets into other IP packets, a technique called IP tunneling. With tunneling all data from the client is sent to the frontend, while the return packets can be sent directly to the client. This way, the frontend only need to process the packets from the clients to the nodes, and not the packets sent from the nodes. Often this will decrease the load on the frontend, as the packets from the database will be larger than the packets from the client.

The last technique is called direct routing, which instead of encapsulating packets from the client, transmit them directly to a node. Direct routing however require that it can be transmitted directly, which is only possible if the nodes and the frontend are physically connected. The difference between direct routing and IP tunneling is that the tunneling step can be eliminated. IP tunneling require support from the OS, but both on all the nodes and frontend. Direct routing only requires support from the OS on the frontend. Linux has a solution for all three in the newest kernels [3].

A problem with implementing the frontend in this manner, is that absolutely nothing is known about the state of the connection. This makes it impossible to reschedule a client connection to another node. For applications where the life-time of the connection is low, i.e. web solutions without persistent connections, this is not a big issue. However for other types of applications, which keep the connections open for a long time, there is a high chance of causing load skew.

A frontend can be a serious bottleneck no matter which solution are chosen. For example, using NAT to handle the network traffic will require substantial CPU power to rewrite all packets and also the network interface on the frontend may not be fast enough to handle all the traffic. [3] calculate that for a normal web server, about 22 nodes can be handled with a 100 Mbit network interface in the frontend.

The choice of which node a client is connected to happens centrally in all the frontend solutions. Placing the choice on a central entity eliminates problems that would otherwise arise when the scheduling algorithm is expecting to know what happens. Another advantage is that the frontend usually have access to the newest load information and can thus make the best choice. The downside is that the client must send the job to the frontend. The frontend sends the job further, thus introducing a latency penalty for all communication. Also the most flexible of frontend solutions are also the ones that are the most complex to implement.

2.4.1 Communication in a frontend solution

In order for some of the scheduling algorithms to work, the frontend needs to receive information about the load of the nodes. The frontend can obtain this information in several ways.

The most obvious way is to have the frontend connect to each node in the cluster, and through this connection obtain the load of each node. This will result in N connections to the nodes. And if there is more than one frontend (e.g. to make the system fail-safe) this will become $N \cdot F$. All these connections will use system resources, and if there are more than one frontend, it will also cause the same data to be sent twice.

If the frontend is on the same network as the nodes, it is possible to improve on this. Since the local network can be considered safe, all the nodes can just send an UDP packet with their load information in to the frontends. If there are more than one frontend a UDP broadcast seems the best solution.

2.5 Dispatching from the client

When the dispatcher is placed on the client, there are several ways to implement it. Either some client daemon makes the choice for all client connections on the client or the individual client connections make the choice. Having the choice on the client lowers the latency dramatically since a potentially long round-trip to the cluster is avoided.

Having each client application make its own choice places the choice as far away from the cluster as is possible. The advantage gained from this, is that the latency only depends on the speed of the algorithm used. A problem with this approach is, since the choice is being made by the application all information about the choice is unknown to all other clients. An aspect that does not work well with some scheduling algorithms.

A solution to some of these problems would be to move the choice into a daemon on the client. This will increase the latency because a round-trip to this daemon is needed whenever a choice must be made. The round-trip is just a context-switch and thus nowhere near the round-trip time to the a node connected to the network. This solution does not remove the problem that choices made on one client machine, is visible to the others, but it makes it easier to handle. A solution to this problem is discussed in section 3.11.

2.5.1 Communication on the client

The main problem in getting the load information to the client, is that the nodes and the clients are not located on the same network. Because of this limitation, the only way of transporting the data to the client is by using TCP unicast. Also this connection must be initiated from the client, as the nodes may not be able to connect to the client, because it may be behind some firewall. In the following we assume that either a local load daemon is used, or that only a single connection is created per client.

The obvious way of collection the load information, is to have each client connect to each of the nodes; resulting in $C \cdot N$ connections. A number that could become a problem in scalability. If the client instead only connect to one of the nodes, the number of connections would fall to C instead. For this setup to work, nodes must know the load of all the other nodes in the system. But since it is assumed that the cluster is located on the same network, it is easily achieved by letting each node broadcast its load and listen to the others transmission of their load.

2.6 Dispatching from the nodes

Placing the dispatcher on a node can be equivalent to the frontend solution with the exception that the node playing frontend is also a node where the application is running. This solution gives no apparent advantages over the actual frontend solution, though.

A similar solution to the one used on the client can also be used. Here the client sends the job to any of the nodes. This node will then determine which node is the best, and redirect the client to that. The advantage is that the nodes have access to the newest load information. If the client is just sending the job to be dispatched to a node, this solution is one of the best. If the client however simply asks which node to connect to, it introduces a large latency penalty because the client cannot submit the job before getting an answer.

Yet another solution is to let every node share the same IP and thus they all receive the network traffic from the clients. Since all nodes actually received the job, it is only a matter of the load balancer to decide where to execute it and tell the chosen node to begin. If the job contained a write, it is even possible to tell all nodes to execute the job and thus keep the entire cluster up to date. This solution both hides the fact that there is a cluster for the client and keeps the latency and overhead to an absolute minimum. The biggest problem with this setup is that it requires the nodes to have the same IP. For the nodes to be able to talk to each other the cluster must also have a second network where the individual nodes can be reached. Both are configuration issues, that are easily dealt with. Emic Networks[4] implements this solution for MySQL. Information about their implementation can be retrieved from the white-paper on their homepage.

One huge advantage of placing the dispatcher on a node is that it can be integrated into the application. Thus it is only necessary to administrate the application on the nodes. Further, the situation where a node only partially fails, i.e. only the application or the load balancing daemon fails is no longer possible.

2.6.1 Communication on the nodes

The communication when the choice is being made on the nodes, is not a big issue. Since the nodes are all located on the same network, it is very easy for them to communicate with each other at low cost. Actually the same solution as suggested when the choice is being made on the clients, applies here. I.e. broadcasting each node's load on the network, and let the other nodes pick it up.

2.7 Anticipating system failure

When evaluating the different solutions against each other, one of the factors to consider is how the system handles failures. The basic premise in making a solution fail-safe, is to avoid having single points of failure. This is when the failure of a single component of the system is able to bring the entire system down.

If the dispatcher is on a node the system is just as fail-safe as a normal MySQL installation. The clients all connect to one of the nodes, which makes this node a single point of failure. If this node fails, the clients cannot connect to any node. If this solution should be made more resistant to failure, the client will need to be modified so that it will be able to connect to a range of nodes, instead of just one. Or the nodes should be configured to do IP fail-over, so that if the main node is down, one of the other nodes take over its IP-address and job. Both of these solutions lie outside the modifications otherwise needed in order to place the choice on the nodes. So from that point of view, this is not a very practical solution. However the IP fail-over solution is well known, and have been implemented many times.

If a frontend solution is used, the frontend becomes a single point of failure. As was the case with the nodes, this makes the system just as fail-safe as a one server solution. However any of the nodes can fail, and only cause disrupted services until the frontend discovers that the node has failed. In order to make the frontend solution fail-safe, the same method can be used, as when the dispatcher is placed on the nodes. So either the clients must be configured to connect to several different frontends, or IP fail-over must be implemented for the frontends.

If the dispatcher is on the client, it is easy to make the system fail-safe. In both the proposed solutions, there really are no single points of failure. One could argue that when a local scheduling daemon is running, this it is effectively a single point of failure for all the client connections running on that client.

2.8 Concluding remarks

In this chapter we have reviewed several solutions as to where the dispatcher that determines where to execute a query should be placed.

A frontend seems to be most flexible solution, as it makes few assumptions to the network topology, and is total transparent for the clients. But the solution is very complex to implement and it is not clear if this complexity makes up for the flexibility.

Placing the choice on the node seems reasonable in complexity, but could impose a huge penalty in overhead for the selection of a node. The client-side solution seems to be the least complex of the solutions, while still maintaining most of the good properties.

Chapter 3

Scheduling algorithms

It is better to know some of the questions than all of the answers.

- Aldous Huxley

In the following sections we will describe some general scheduling algorithms in order to evaluate their advantages and disadvantages. The goal is to find one or more algorithms that will work with any workload and any cluster configuration.

The information available to the algorithms are basically a list of nodes. This list contains information about the nodes that can be used as a base for scheduling algorithms decisions.

All of the algorithms are illustrated by pseudo-code. The syntax used is `C++`-like. To reduce the length, the pseudo-code does not include handling special situations nor any minor optimizations. All the scheduling algorithms described here use two functions. One, called `OnLoadUpdate` which is called whenever the information in the list of nodes is updated, and one called `GetNextNode` which is used by the load-balancer to get the next node.

3.1 What makes a good scheduler?

In order to evaluate the different scheduling algorithms against each other, we will need a measurement of what makes a good scheduler. Two metrics are used to measure the performance of a scheduling algorithm: The throughput and the response time. The throughput is how many jobs that can be executed in a given time interval. The response time is how fast each of the jobs are executed.

The problem with designing a good scheduler is that throughput and response time contradict each other. To illustrate this contradiction, consider the following jobs.

Job	Execution time
J1	24
J2	8
J3	6
J4	2

Assume that multiple jobs can be run at the same time on a node, but two jobs running at the same time will double the execution time.

There are 16 ways to schedule these jobs on two nodes if the jobs all arrive at time 0. The 16 ways compose of 2 (All on one node) + 8 (1 job on one node) + 6 (Two jobs on both nodes). If the nodes are of equal capacity half of them are identical except the nodes are reversed, which thus results in 8 ways. The numbers for two identical nodes are shown here:

Node 1	Node 2	Avg. response	Time to complete
1,4	2,3	14	26
1,3	2,4	14	30
1	2,3,4	15	24
1,2	3,4	15	32
2	1,3,4	15	32
1,2,4	3	16	34
1,2,3	4	20	38
	1,2,3,4	23	40

The above is an extremely simple scheduling task given the time required for each job to complete. No scheduler without this information would consistently have chosen the best schedule. Even if it is possible to schedule the jobs perfectly, one either has to choose if throughput or response time is more important. The combination with the lowest average response time is 8.3% slower than the fastest. And the combination with the highest throughput has an average response time that is 7% higher. Adding facts like jobs not all arriving at time 0, multiple CPUs on nodes, node of different capacity etc. only makes the scheduling even more impossible to do perfectly.

The only solution to scheduling is then to try and guess which node to use. The guess can be based on information gathered from the nodes or it can be as simple as the random choice.

3.2 Round-robin

Round-robin is by far the simplest algorithm available to distribute load among nodes. It is therefore often the first choice when implementing a simple scheduler. On of the reasons for it being so simple is that the only information needed is a list of nodes.

The algorithm traverses the list, returning the nodes one by one. When the end of the list is reached, the algorithm starts from the beginning of the list again. Thus returning the same nodes in the same order.

ROUND-ROBIN SELECTION (3.1)

```
Input: A list of nodes (nodes)
Input: Iterator into nodes (next_node)
Output: The node to use
GetNextNode()
    Node *result = next_node;
    next_node++;
    if (next_node == nodes.end()) {
        next_node = nodes.begin();
    }
    Return result;
```

As can be seen, the algorithm requires very little computation. The time-complexity of the selection is $\mathcal{O}(1)$. Combined with its low implementation complexity and its low information requirements, the round-robin scheduler is also often the most efficient scheduler algorithm. However this is only when several key assumptions are true:

1. **The nodes must be identical in capacity.** Otherwise performance will degrade to the speed of slowest node in the cluster.
2. **Two or more client connections must not start at the same time.** Should they, the node chosen will be the same, because the order of nodes retrieved from the cluster is the same every time.
3. **The jobs must be similar to achieve optimum load distribution among the nodes.** If a single node is more loaded than others it will become a bottleneck in the system.

Any or all of these assumptions might be challenged in the face of a real application and cluster setup.

3.3 Weighted round-robin

To solve one of the problems with the normal round-robin scheduler a weight is added to each node to represent computational capacity. This simple addition to the information required reduces the problem where nodes must be identical in capacity. If a node has weight 1 and another node has weight 2, the latter will be chosen twice as often as the former.

The first implementation uses the weight as a number of tokens. For each node the original number of tokens and a current number of tokens are stored. The selection happens by scanning the list of nodes until a node with a token count above zero is found. When a node is selected, a token is removed from the node. The next selection continues the scan where the prior left off, thus keeping the round-robin property. When all tokens in the list are used up, each node replenish its tokens to the original token count. The time-complexity for this is $\mathcal{O}(\mathcal{N})$, because a complete scan of the node list could be made every time.

This is however suboptimal. We realized that the algorithm can be implemented with a time-complexity of $\mathcal{O}(1)$. This faster version is implemented by splitting the node list in two. A list of nodes with tokens remaining and a list with the nodes that have used up their tokens. When no nodes are left in the first list, the two lists are exchanged.

WEIGHTED ROUND-ROBIN SELECTION USING EXPIRED LIST (3.2)

```
Input: A list of nodes with tokens (nodes)
Input: A list of nodes with no tokens (nodes_expired)
Output: The node to use
GetNextNode()
    if (nodes.size() == 0) {
        nodes.swap(nodes_expired);
    }
    Node *result = nodes.begin();
    result.tokens--;
    if (result.tokens == 0) {
        nodes.pop_front();
        result.tokens = result.original_token_count;
        nodes_expired.push_back(result);
    } else {
        nodes.pop_font();
        nodes.push_back(result);
    }
    Return result;
```

It turns out that the implementation of this faster version is actually simpler. Also the notion of using the weight as a number of tokens is also very good and we use it for the rest of the algorithms that uses weights.

The problems with the weighted round-robin are still many. The weight only helps with the non-equal capacity problem of the round-robin algorithm. Also it can be hard to choose the optimal weight. The weight will depend on the mix of queries, how they effect the different subsystems on the nodes, and how the subsystems of the different nodes behave. So in some situations, it will be impossible to choose a correct weight.

3.4 Random

The second problem mentioned for the round-robin scheduler was that during the startup phase, several client connections may choose to connect to the same node. There are two obvious ways of handling this problem. First is to randomize the list of nodes when it is received. This solution prevents client connections from choosing nodes in the same order. The second option is to choose a random node every time. This selection is more random, meaning that it will keep on selecting the nodes in a different order each time. The reasons for this added complexity is that some workloads may perform the worst possible way with any of the round-robin schedulers. Consider a cluster with four nodes; if unlucky every fourth job is a job that requires much work. A round-robin algorithm would then assign these jobs on a single node and slow everything down. Here it matters little that the list of nodes was randomized to begin with.

The random algorithm can then be combined with the idea of weights. This will allow it to work with nodes of different capacities, just like the weighted round-robin algorithm.

WEIGHTED RANDOM SELECTION (3.3)

```
Input: A list of nodes with tokens (nodes)
Input: A list of nodes with no tokens (nodes_expired)
Output: The node to use
GETNEXTNODE()
    if (nodes.size() == 0) {
        nodes.swap(nodes_expired);
    }
    int random_number = rand(nodes.size());
    Node *result = nodes[random_number];
    result.tokens--;
    if (result.tokens == 0) {
        nodes.delete_index(random_number);
        result.tokens = result.original_token_count;
        nodes_expired.push_back(result);
    }
```

```
16:        Return result;
```

The algorithm uses the same way of administrating the tokens as weighted round-robin. Therefore the time-complexity is also the same ($\mathcal{O}(1)$).

Another way of viewing the weight is as the fraction of probability a node has of being selected. However the only way we could think of to implement this algorithm was with a time-complexity of $\mathcal{O}(N)$. Therefore it was not considered a valid option.

3.5 Job informed

The problem with assuming all jobs are equivalent is twofold. The first problem is that two jobs may use different subsystems on a node, i.e. one may be disk bound while another is CPU bound. The second is much simpler, two jobs may simply differ in the time they take to execute. In this algorithm we try to solve this second part.

The problem can be solved by viewing the number of active jobs on a node as the node's load. The problem of getting load from a node is explored in the next chapter.

The first algorithms we cite here are described in [5], where two algorithms are proposed, *BasicLI* and *AggresiveLI*. They both attempt to even out the number of active jobs on the nodes by looking at the current amount of active jobs. The algorithms then try to distribute new jobs to even out the load.

The following notation is used:

N	no of nodes
L_i	Reported load for a node
L_{tot}	$\sum_{i=0}^{n-1} L_i$
$arrive_T$	Number of requests expected to arrive during a phase

BasicLI takes the simple approach to distributing the new jobs. After an update of the load information, *BasicLI* determines each node's probability (P_i) to be selected. The probability is calculated as:

$$P_i = \begin{cases} \frac{\frac{L_{tot}+arrive_T}{N} - L_i}{arrive_T} & \text{if } \forall i(\frac{L_{tot}+arrive_T}{N} > L_i) \\ \text{see below} & \text{otherwise} \end{cases}$$

If $\frac{L_{tot}+arrive_T}{N} > L_i$ for any i the phase is too short to distribute the load properly. We will ignore this special case, read the article for further information.

The *basicLI* approach is suboptimal. Instead of immediately trying to even out the load, the algorithm will only converge toward an equalized load. Further, if the estimate for $arrive_T$ is underrated the least loaded nodes will quickly become the most loaded. If the estimate is overrated however, it means that the load will converge slower. Graphs for the impact of miss-estimation of $arrive_T$ is shown in [5, page 17].

AggresiveLI attempts to first even out the load on all nodes, and then revert to a random choice. They do it by constantly calculation the percentage with which a node should chosen. However the solution in the article does not fit very well with the setup we have.

The basic idea of *AggresiveLI* can be combined with weights and the constant recalculation can be avoided. The result is a simpler algorithm:

JOB INFORMED (3.4)

```
 1:  Input: A list of nodes (orig_nodes)
 2:  OnLoadUpdate()
 3:      double token_tot = 0;
 4:      double active_jobs_tot = 0;
 5:      for (iterator it = orig_nodes.begin() ;
             it != orig_nodes.end() ; ++it) {
 6:          token_tot += it.original_token_count;
 7:          active_jobs_tot += it.active_jobs;
 8:      }
 9:      nodes_expired.clear();
10:      nodes.clear();
11:      for (iterator it = orig_nodes.begin() ;
             it != orig_nodes.end() ; ++it) {
12:          it.tokens = floor((it.original_token_count/token_tot -
                 it.active_jobs/active_jobs_tot) * active_jobs_tot);
13:          if (it.tokens <= 0) {
14:              it.tokens += it.original_token_count;
15:              if (it.tokens <= 0) {
16:                  it.tokens = 1;
17:              }
18:              nodes_expired.push_back(it);
19:          } else {
20:              nodes.push_back(it);
21:          }
22:      }
23:  GetNextNode()
24:      Return WeightedRoundRobin.GetNextNode();
```

This algorithm requires some in-depth explanation. The first loop (line 5-8) finds the total of both the number of tokens and the number of active jobs in the system. Second loop (line 11-22) calculate the load for each node. The calculation

$$\text{it.original\_token\_count/token\_tot} - \text{it.active\_jobs/active\_jobs\_tot}$$

is a node's percentage of capacity in the cluster minus the percentage of node's load. The result is the distance from the optimum load for the node. Multiplying the distance with the amount of active jobs, the distance in jobs is obtained. If the resulting number is negative it means that the node is more loaded than the average node.

To get optimum and fast balancing the nodes with a positive amount are placed in the `nodes` list, those with negative number are put on the expired list. However those placed on the expired list are not balanced correctly when the expired list is swapped with the `nodes` list. This is because the least loaded has only been balanced to become averagely loaded and the most loaded (those placed on the expired list originally) are still the most loaded. Thus instead of putting them on the expired list with their original token count, the distance from optimum load is added (it is a negative number).

This algorithm is very dependent on the amount of jobs scheduled in each phase. If the number of jobs are high, all nodes will eventually become equally loaded. If however the jobs are few only a partial balancing towards the least loaded nodes are achieved. The optimal phase length is not apparent. A long phase gives the algorithm opportunity to balance better, but could base the decisions on outdated info. Too short a phase could prevent the algorithm from achieving optimum balancing.

Since his algorithm uses the weighted round-robin's GetNextNode it runs in $\mathcal{O}(1)$. `OnLoadUpdate()` runs in $\mathcal{O}(N)$. However `OnLoadUpdate()` will always take $\mathcal{O}(N)$ time because each node must be processed when information comes from an load update anyway. Thus this algorithms `OnLoadUpdate()` only adds a minimal constant factor to the time spent.

3.6 Load informed

All of the weighted algorithms all have one thing in common. They assume that the weights are representative for the nodes individual capacity and representative for the jobs executed. Another issue is that it can be very hard to tune the weights correctly. Also, all the other algorithms have assumed that it matters little which subsystem a job utilizes.

A way to remove these limitations are to base the selection on a number that more closely resembles the actual load of the node at a given point in time. The load number can then be used by the scheduler algorithm to choose the node which has the best chance of executing the job fastest. How to actually measure load fast and effective is covered in chapter 4.

The simplest version of a Load informed algorithm is the following. It works by simply selection the node with the least load:

SIMPLE LOAD INFORMED (3.5)

```
1: Input: A priority heap of nodes sorted by load (nodes)
2: Input: A list of nodes in the system (orig_nodes)
3: OnLoadUpdate()
4:     for (iterator it = orig_nodes.begin() ; it != orig_nodes.end() ; ++it) {
5:         nodes.delete(it);
6:         nodes.insert(it.load, node);
7:     }
8: GetNextNode()
9:     Return nodes.first();
```

The `delete` and `insert` operations on a priority heap can be implemented in $\mathcal{O}(\log N)$, and the `first` operation can be implemented in $\mathcal{O}(1)$. Therefore the time-complexity for `OnLoadUpdate()` function is $\mathcal{O}(N \cdot \log N)$ and for `GetNextNode()` it is $\mathcal{O}(1)$.

The main problem with this algorithm is that it assumes that the load is constant throughout the phase. So in between two updates, the same node will be chosen each time. This is of course not desirable.

A way to solve the problem would be to choose randomly between the nodes with the lowest load. This way the jobs will be evenly distributed between the least loaded nodes. The problem is then to determine which of the nodes are the least loaded ones. A very easy way to do this is to choose the k least loaded nodes. But this is still not a very good solution, since we are only interested in using nodes that are currently loaded less than the average node. So in order to avoid the problem, the scheduler must look at the load of the individual nodes, and from that make a choice of how many should be included in the random pool. The algorithm could be implemented like this:

RANDOMIZED LOAD INFORMED (3.6)

```
Input: A list of nodes in the system (orig_nodes)
Input: A balanced tree of nodes sorted by load (nodes)
Input: The max number of nodes to consider (max-nodes)
Input: The max leap in load allowed (load-interval)
OnLoadUpdate()
    for (iterator it = orig_nodes.begin() ; it != orig_nodes.end() ; ++it) {
        nodes.delete(it);
        nodes.insert(it.load, node);
    }
GetNextNode()
    int num_nodes = 0;
    node tmp_nodes[max-nodes];
    for (iterator it = nodes.begin() ; it != nodes.end() &&
            num_nodes < max-nodes ; ++it, num_nodes++) {
        if (it.load > nodes.begin().load + load-interval) {
            break;
        } else {
            tmp_nodes[num_nodes] = it;
        }
    }
    int random_number = rand(num_nodes);
    Node *result = tmp_nodes[random_number];
    Return result;
```

For a balanced tree, the `delete` and `insert` functions can be implemented with a time-complexity of $\mathcal{O}(\log N)$. Thus the `OnLoadUpdate()`-function will have a time-complexity of $\mathcal{O}(N \cdot \log N)$. `GetNextNode` first locates the node with the lowest load, and from this it traverses the tree, until it has encountered enough nodes. Since this tree traversal is limited by the `max-nodes` number, this part of the function runs in `max-nodes` $\cdot\mathcal{O}(\log N)$. `GetNextNode` therefore also runs with a time-complexity of $\mathcal{O}(\log N)$.

However if the phase is long this approach fails for the same reason as the previous. In that case, we can expand the algorithm, so whenever it returns a new node for a client connection it adds some artificial load to the actual load. This artificial load is then kept at least until a new update of the load information is received. If a connection moves to another node, the node it is moved from receives a small decrease in load. This procedure reduces the chance of the least loaded nodes becoming the most loaded. It can be implemented using the following algorithm:

LOAD ADJUSTING LOAD INFORMED (3.7)

```
Input: A list of nodes in the system (orig_nodes)
Input: A priority heap of nodes sorted by load (nodes)
OnLoadUpdate()
    nodes.clear();
    for (iterator it = orig_nodes.begin() ; it != orig_nodes.end() ; ++it) {
        it.adjust_load();
        nodes.insert(it.load, node);
    }
```

```
GetNextNode()
    if (connection.node != NULL) {
        connection.node.decrease_load();
        nodes.delete(connection.node);
        nodes.insert(connection.node);
    }
    Node *node = nodes.first();
    node.increase_load();
    nodes.delete(node);
    nodes.insert(node.load, node);
    connection.node = node;
    Return node;
```

The `node.increase_load()` and `node.decrease_load()` increases and decreases the artificial load of the node in question. The `node.adjust_load()` function removes old artificial load. A problem is that when receiving a load update, it is not certain that the information included is the newest. It could be that a job has just been submitted, but that the load of the job is not yet visible in the load data. Therefore the `node.adjust_load()` should be able to only remove the load that the load estimation algorithms already have discovered. A very simple and easy way to implement this would be to keep timestamps from the artificial load. Later, benchmarks revealed that these timestamps were imposed to large an overhead and was therefore removed again.

The `GetNextNode()` will then be able to run in $\mathcal{O}(\log N)$, since the `insert` and `delete` functions have this complexity.

If we analyze this algorithm from a higher level. We can see that this algorithm works by returning the node with the least load until it has attained a load level that is equal to the next least loaded node. Then the algorithm will return these two nodes in a round robin fashion, until their load reaches the load of the third least loaded node; and so forth. So if there is no load updates, and if there are enough client connections, this algorithm will end up being equal to the round-robin algorithm. However each time the load information is updated, the algorithm will begin to favor the least loaded nodes once again.

3.7 Affinity based scheduling

Yet another way to schedule jobs, is to use affinity based scheduling. Affinity based algorithms try to utilize reference of locality by looking at some aspect of the request. In a database request that could be the database name, tables or some other specific part of a query to determine which nodes are the better choice for the request. On the basis of this, a subset of the nodes in the cluster is selected as having the ability to execute the request.

Because affinity based schedulers look at a request itself it is under certain circumstances no longer necessary for each node to have access to the same data. The circumstances are that there must be classes of requests, that do not need to mix data from different sources. This could for instance be a web hosting environment, where a cluster of nodes work together serving several domains. [6] shows that delegating requests as described increases both throughput and latency greatly because of much better cache utilization.

The affinity based scheduling does however not conflict with the other schedulers, which are still needed to choose the exact node from the subset that the affinity based algorithm found.

However another kind of affinity based scheduler is a scheduler which bases its load information on the response time of the queries executed. This scheduler needs to look at the query, because in a mix of queries, the scheduler will need to know which kind of query was executed. Otherwise the scheduler would have no way to determine if the query executed slower or faster than normally.

To limit the scope of this paper no algorithms for affinity based scheduling are presented.

3.8 Prioritizing the current node

Changing the connection from one node to another, results in a small performance penalty. Therefore the scheduling algorithms should take this into consideration, by keeping a connection on the same node for as long time as possible. This of course only makes sense when the scheduling algorithm is not based following a specific pattern like the round-robin.

For weighted algorithms, staying on the same node can be implemented by checking if the current node has any tokens left and use one of them. However, there are some problems with this approach. Effectively it overrules the normal scheduling policy, and can possibly cause the weight of the nodes selected initially to be exhausted before connections are moved to other nodes. Especially this approach fails in the case when the number of client connections are lower than the number of nodes.

If the scheduling is based on load information, we must use another method to obtain this effect. When a client connection wishes to change to a new node, we simply give the node it is connected to a small temporary artificial load bonus. This way increases the chance of choosing the same node again, while making certain that a overloaded node is not chosen again.

How much the temporary bonus should be is not easy to determine, however. It depends on the environment of the system and on whether the client has a cached connection to the node it would otherwise be assigned to. Also it is highly dependent on the time it takes to create a new connection to a node. Therefore the number must be a guess or determined through benchmarking. Our benchmarks show that there is no measurable overhead involved in changing nodes, when caching the connections.

3.9 Estimating weight

In all the algorithms that use a weighted scheme, it is expected that the administrator of the cluster gives each node a weight.

There is no easy solution to finding the weight automatically however. It would be nice if a simple benchmark could be run and the number used from there. Or if the Linux kernel could somehow give a number. The problem is that giving a node a weight requires knowledge about the node, the settings used and workload expected to be run.

There are at least two possible ways of estimating the weight. The first is for the administrator to run a sample workload on each node of the cluster and compare the run times. This both takes a lot of time running and preparing. And there is no guarantee that the result is a valid number nor that the benchmark was even representative for the actual workload. Another option is to guess the weight based on feedback from the database node (not the client for obvious security problems). This feedback could be the execution time of the queries, or how many running queries there currently are. It could also be the number of executed queries. Either way, it resembles what we are doing with the Job and Load informed algorithms.

Estimating the weights incorrectly can lead to problems. If the cluster consists of fast and slow machines, there are two ways to estimate the weights incorrectly. If the fast machines are overestimated, the slow machines will not receive enough requests to perform optimally. In this case the maximum throughput of the cluster is lower that what it could be. If on the other hand the fast machines are underestimated, the slow machines can end up having all the threads. This can happen if the slow machines are not fast enough to finish a job, before a new one is submitted to them, thus overloading the slow machines more and more, worsening the situation. So it is better to overestimate the fast machines, than overestimating the slow machines in a cluster.

3.10 Update algorithms

The update algorithm is the algorithm that decides when and how to update the load information in the clients and nodes. There are 3 basic ways of doing this. The two first algorithms can be viewed as having a bulletin board where information about each servers load is displayed.

The first algorithm is the simplest and is called *periodic update*, where all information on the bulletin board is updated every T seconds. This interval between these updates is called a phase. So in the beginning of a phase the information on the board is accurate, but deteriorates as we near the end of the phase. With this algorithm the average age of the information is T/2 seconds old.

The second algorithm is much simpler to implement, because instead of a global update every T seconds, each server individually updates its own information on the board every T seconds. Thus there is no need to coordinate the update between the servers and therefore simpler to implement. Because of the more continuous nature of this update algorithm, it is called *continuous update*. [7] found that the variance in distribution of the updates between servers is very important for many algorithms. More specific is was found that updates that are clustered together, almost as in the first algorithm, are often outperformed by updates that are distributed evenly throughout the phase.

The third algorithm is called *update-on-access* and was developed in [5, sec 3.2]. The idea is that whenever a new node has been chosen, the latest information about the node is sent back along with the answer to client. This kind of update is suited for applications that have bursts of requests or where the service on the node is used seldom. T in this algorithm is the time that a client updates.

3.11 Problems running schedulers in parallel

Some of the scheduling algorithms maintain a state which helps them choosing the correct nodes. This information is modified whenever the node is chosen. If two scheduling algorithms run in parallel, and they does not share this information, they will be inclined to make the same choices.

This problem can be solved in several different ways. The most obvious is that the algorithms should communicate with each other, in order to maintain the same state. However this will only work if the scheduling algorithms are running a node or a frontend. It will not work when the algorithms are run on the client, because it might not be possible for the clients to communicate directly with each other. So if the clients will need to communicate with each other, it will have to happen through the cluster. However having the clients trusting each other is not a good solution security wise.

Another way to do accomplish the goal, is to make the number of clients available to the algorithms. When one of the clients make a choice, it then estimates how many other clients could have made the same choice. This estimation can then be used to simulate the knowledge of knowing when nodes are chosen on the other clients. This approach can be implemented without changing the scheduling algorithms like this:

CLIENTS IN PARALLEL (3.8)

```
Input: The number of clients connected to the system (num_clients)

OnGetNextNode()
    int same_choices = estimate_same_choice(num_clients);
    Node *choices[same_choices];
    for (int i = 0 ; i == same_choices ; ++i) {
        choices[i] = GetNextNode();
    }
    int random_number = rand(same_choices);
    Node *result = choices[random_number];
    Return result;
```

The `estimate_same_choice` function returns an estimate of how many clients could have made the same choice as the one just made. Then the algorithm simulates asking for the next node as many times as there are other clients, and stores this result. The node to return is chosen randomly between these.

The reason the node to return is chosen randomly is that it will make deterministic schedulers work better. If the returned node was not chosen randomly, the different clients would be prone to make the same decisions.

3.12 Concluding remarks

Each of the scheduling algorithms has its pros and cons. Which of them works best depends on a lot of factors. The round-robin and the weighted round-robin are great when the queries are alike in complexity. The job- and the load informed, both tries to measure how well the different nodes handles the load given to them, and then adjust how often the nodes are returned.

Let us sum up the algorithms:

Round-robin:

+ Fast.

+ Requires only information about the names of the nodes.

+ Does not require any global decisions.
+ Easy to implement.
+ Superb balancing, when the workload is not skewed, nor are the nodes.
– Breaks down completely when the above item does not hold.

Weighted Round-robin:

+ Requires only information about the names of the nodes and their weights.
+ Does not require any global decisions, but when the number of jobs are low it greatly improves when available.
+ Superb balancing, when the workload is not skewed.
– Breaks down completely when the above item does not hold.
– Requires more work than the Round-Robin scheduler.

Weighted Random:

+ Requires only information about the names of the nodes and their weights.
+ Does not require any global decisions.
– Unpredictable balancing.

Job informed:

+ Good at preventing worst-case scenarios. I.E. overload of a single node.
+ The information required is platform independent.
– Require global decisions to work properly.
– Requires information about the names of the nodes, their weights and a constantly updated amount of current jobs.
– The optimal phase length is not clear.

Load informed:

+ Can prevent worst-case scenarios.
– The information required is not platform independent.
– Require global decisions to work properly.
– Requires information about the names of the nodes, their weights and a constantly updated specification of the load of each node.
– The optimal load adjustment when choosing a node is not clear.
– Very complex to implement and even more complex to get right.
– Requires a large machinery to calculate the load of a node as we shall see in the next chapter.

Chapter 4

Estimating load

Imagination is more important than knowledge.

- Albert Einstein

This chapter is about trying to get a number that represents the load of a node. We present how to (or not how to) interpret the information available from the different subsystems of a machine and how to combine it into a single number.

We assume that the nodes are running Linux, or a operating system capable of providing the same or equivalent metrics.

We will in turn describe the different metrics that can be used as a measure of the nodes load, and discuss the relevance of each of them. We will also in some cases provide pseudo-code for implementing some of these functions.

4.1 Estimating CPU load fast and accurately

The Linux kernel exposes three different ways of detecting the current CPU load on a node.

1. The amount of time the kernel was busy since last we asked.
2. The current number of processes waiting to be run plus currently running. It also includes processes that are waiting for I/O.
3. The moving average for last 1, 5 and 15 minutes of the above. [1]

The Linux kernel measures time in jiffies. Jiffies are the number of timer interrupts since system start. Every time a interrupt happens the kernel looks at what is running at the CPUs in the system. If a process was running, the CPU usage statistics for the process is updated. The CPU statistics for a process is split up into three numbers: The amount of time the process was running inside the kernel, the time doing normal processing at default priority and the time running at a lower priority. These three numbers are also recorded for the total system. It is these numbers that we can extract to get the amount of time the kernel was busy since last we asked. It is in the file `/proc/stat` that we find the lines:

```
cpu  186520020 27617445 30850976 12999990579
cpu0 74115796 9838026 13861517 6524674171
cpu1 112404224 17779419 16989459 6475316408
```

[1]Moving average is calculated like this: $A = \frac{L + A*(S-1)}{S}$. Where L is the current number of running processes. S is the number of samples per time unit and A is the moving average.

Here taken for a dual CPU node. The first number in a line is jiffies spent in normal priority, next is what was spent in low priority followed by jiffies spent in the kernel and last is the total number of jiffies since system start minus the prior numbers. On this particular architecture and kernel version, IA64 and 2.4, the timer interrupt happens 1024 times per second. Thus if we check the file every few seconds and look at the last number to see how busy the CPU, was we get the CPU usage.

Consider the situation where the CPU was busy more than 95% of the time. With the previous method it is impossible to detect if it was the same process or was it a hundred processes competing for the CPU. For that we need to know the number of processes that are waiting to be run.

The file `/proc/loadavg` tells this number along with the moving average of the number:

```
0.35 0.41 0.54 4/148 10011
```

The fourth number is the current number of processes waiting in the run queue. The first three numbers show the moving average of queue length over the last minute, five minutes and fifteen minutes with sampling every five seconds. The number of currently running processes can be very misleading because it includes the process that is reading the number and also any process that may be waiting for something that process just did, e.g. printing via syslog would put syslog in the queue. Experience show that under very few circumstances anything will be gained from this number. An accurate load average for the last five seconds would be much more useful.

The last two numbers are not important for our purpose, but for completeness they are the total number of processes in the system and the process id of the last process started.

Lets then look at how to get an accurate number for the CPU load out of the combination of the three. The following algorithm returns a number for the load per CPU on the node.

ESTIMATION LOAD FAST AND ACCURATE (4.1)

```
1:  Input: Number of processes in waitqueue plus running (currently_waiting)
2:  Input: Number of CPUs in the node (nr_cpu)
3:  Input: CPU usage since last check (pct_cpu_time_used_since_last)
4:  Input: Linux' load average for the last minutte (load_avg_1min)
5:  Output: Estimate of load
6:  ESTIMATE_CPU_LOAD()
7:      if (pct_cpu_time_used_since_last < 95%) {
8:          return pct_cpu_time_used_since_last/100;
9:      }

10:     bool load_is_increasing =
                ( load_avg_1min > 1.25*load_avg_1min_last_time_we_looked);

11:     if (load_avg_1min < 0.3*nr_cpu ||
                (load_avg_1min < 1.2*nr_cpu && !load_is_increasing)) {
12:         return 1.0;
13:     }

14:     if (load_avg_1min < 1.2*nr_cpu && load_is_increasing) {
15:         return 2.0;
16:     }
```

```
17:        if (currently_waiting > 3*nr_cpu &&
                  currently_waiting > 2*load_avg_1min) {
18:             return (load_avg_1min+currently_waiting)/(2*nr_cpu);
19:        }

20:        return load_avg_1min/nr_cpu;
```

- Lines 1-5 are the input variables. We assume the `currently_waiting` are exclusive ourself and any processes we may have started.
- Lines 7-9 check if `pct_cpu_time_used_since_last` is useful in its own right. If the node is not using its CPUs fully, this is the fastest way of detecting it.
- Line 10 check if the load seems to be increasing.
- Line 11-13 handles the case where a single process (per CPU) seems to be using 100% CPU or the load is so low that we can't really predict what is going on.
- Line 14-16 tries to prevent that the nodes gets overloaded with new requests. Because incoming requests may not yet have materialized into a constant load.
- Line 17-19 checks if the number of processes that are waiting in queue is much greater than the actual load. If so the average if the two is returned.
- Line 20 simply return the current load average if it is constant, because we are in a situation where it matters little if the actual load defers a lot. Because if it does we will catch it in the next iteration where our load would not be constant.

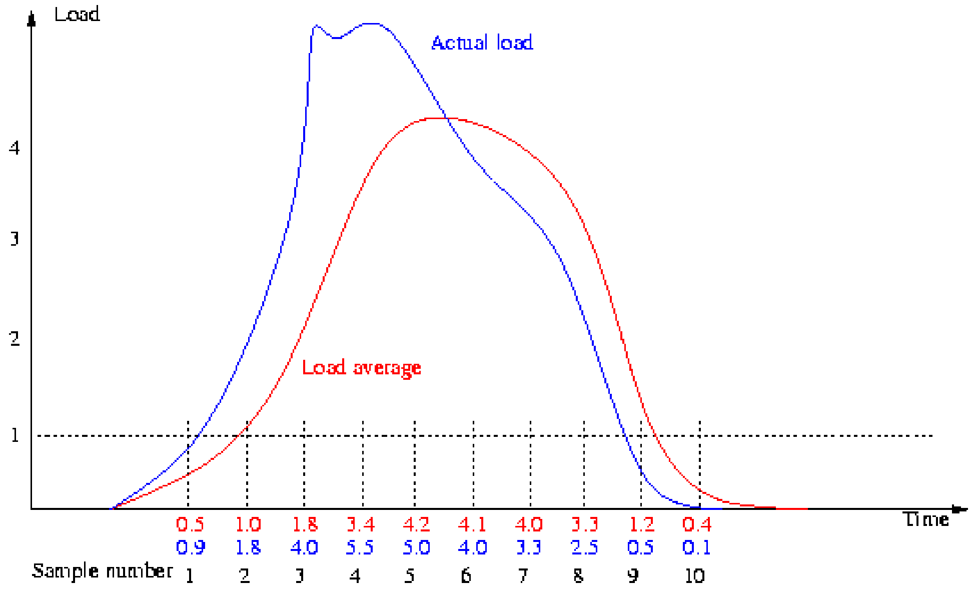

Figure 4.1 – *A synthetic example of how load varies.*

Lets look how the algorithm works (figure 4.1). The graph is a synthetic example how the actual load of a machine (the "actual load" line) might be interpreted into a load average (the "Load average" line). At sample point 1 the actual load is below 1.0 and that number returned by the algorithm as the load. Next sample is a load close to 1.0, but since it is detected that the sampling was taken before an increase in load (because `load_is_increasing` is true) 2.0 is returned. At sample three the number of queued processes is much higher than the average load, so the average of the two are returned, 2.9 in this case. Sample four only has a minor discrepancy between the average load and the actual load, so the load average is used, 3.4. Sample 5, 6, 7 and 8 follow the same pattern. Between sample 8 and sample 9 the CPU is not used fully, so that number is used as the load. Likewise in sample 10.

4.2 Estimating disk load

It is well known that the most efficient way is to read data from a disk is to read it sequentially, because the movement of physical parts inside the disk lowers transfer rate considerably. Thus a request for data that is spread throughout a disk will load the disk longer than the same request where the data is stored sequentially. The throughput of the disk however will be much greater for the latter request. Therefore we can conclude that nothing would be gained by looking at the current transfer rate itself.

So a better metric would be the current number of outstanding requests, like we can get the number of processes waiting in the run queue. Unfortunately the current Linux kernel (2.4) has no standard way of obtaining this number.

Now consider a system, where accurate information about the usage of disks can be obtained. Here we observe that on a disk system it is not necessarily true that performance decrease when several competing processes are trying to get something from/to disk. Consider a situation where a process P1 wants some data from disk and a process P2 wants the same data. If P1 is running alone, it will just read the data. With P2 running concurrently with P1, the scenario could look like this: P1 is lucky and gets permission to read the data from disk first. Then P2 gets permission to read, but at this point the data is already in memory, so P2 won't need to read it from disk. Thus in this simple situation P2 will never load the disk at all. Many variations can be found over this simple example. They all show that it is not possible to predict how much a disk would be loaded extra, simply by looking at the current usage.

A special variation to consider is that unlike the CPU, the disk system may be a combination of many physical disks. This means that one process could read some data from one disk and a competing process some data from another disk. Removing either from the system would not impact the others performance.

Finally not all disks in a system are equal. Some may be allocated to temporary tables, others to the database and yet others for backup and recovery. Thus taking the mean value of a systems disk usage would be fruitless.

We can therefore conclude that reading a disks load is both non-trivial and even if possible not very useful.

4.3 Estimating network load

To understand load of networks, some queuing theory is needed. The simplest queuing theory is called M/M/1. M/M/1 means that the distribution of arrival

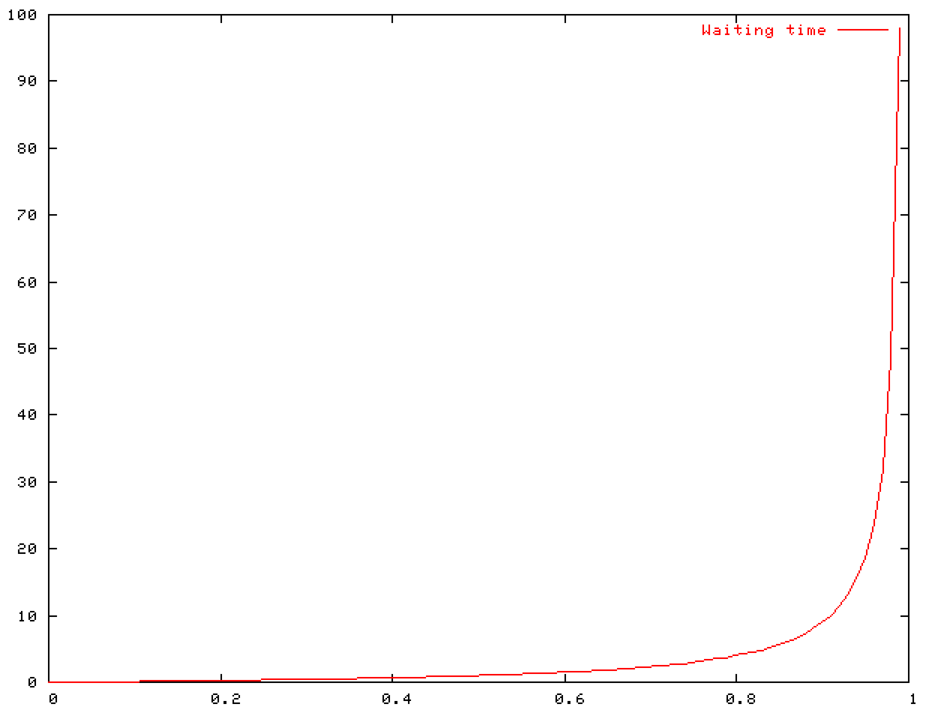

Figure 4.2 – *Plot of waiting time given percentage of network utilization.*

times of packets and distribution of service times (time it takes to process a packet) are both Markovian and that the system is working with a single queue (a single network interface). For most purposes M/M/1 holds in the application of networks. See [8] and [9] for further information on other types of queuing systems.

We now introduce λ as the expected number of packet arrivals per time unit, μ as the expected number of packets completing service per time unit and a 'traffic intensity' $\rho = \frac{\lambda}{\mu}$. The average waiting time in queue for a packet is then $\frac{\rho}{(1-\rho)}$. Which is plotted in figure 4.2. From this graph we can see that estimating load on a network interface is as simple as taking the utilization in percentage ρ and returning $\frac{\rho}{(1-\rho)}$.

To calculate the utilization, the information required is the network interface speed and bandwidth utilization. Unfortunately the Linux kernel supports the former for some types of network interfaces, but not for all. The latter however is trivially fetched from the file `/proc/net/dev`. A sample of this file is too wide to be displayed here, so it was placed in appendix B on page 67. From the output the number of bytes read, written and various other metrics from the network can be collected.

Just like the disk system, the network interfaces in a system may not be equal. On important nodes it is not unusual to have two interfaces backup each other. On a cluster it is not unusual for the nodes to be connected internally on one interface and to the rest of the world on another. Plus that on the latter both interfaces may have backups.

The simple load calculation assume that the network interface in the node is the limiting factor on the network. But the assumption does not hold if the interconnecting network is not designed to handle a full network load. If a switch, hub or something else on the network path is overloaded, it will behave in the same bad manor as an overloaded network interface. However, without knowing the complete layout of the network, these situations are almost impossible to detect. It is however possible to try and guess the situation based on congestion on the network. The problem is then what to do about it. If the entire network is overloaded the only option would be to reduce load overall, but that should be up to the client application instead.

4.4 Estimating memory load

When considering memory in a node, it is not just one item, but rather 5.

Memory for temporary buffers. When the kernel receives a network packet or something similar it needs a temporary buffer to store it in. Usually the amount of memory allocated to this usage is minimal and thus not important in our context.

Memory for the kernel itself. Besides these temporary buffers, the kernel needs lots of space to keep track of its own internal state. That is somewhat vague, but it includes items such as keeping track of processes usage of memory. This may seem like a small thing, but just this simple thing can take up a lot of memory if the amount of memory in the node is very large. A node with 4GiB of memory will use about 50MiB just for this.

Memory for disk cache. Memory that is not used for anything else is allocated to disk cache.

Memory on the swap disk. Swap space is a little different than the above, because it is used under two conditions. First, under high memory pressure the kernel will begin moving the oldest used memory onto disk. Secondly, it might move it even if there is no pressure. If the kernel detects that a process has not been using some part of its allocated memory for an extended period of time, the kernel could decide that the memory would be better used for other things.

The process of moving data to and from the swap space is called swapping. Extensive amounts of swapping often causes a machine to slows to a crawl. Monitoring how much the node is swapping will give a good indication of the problems the node is in, once swapping has started, the node is already crawling and thus it would be much better to catch the situation before it begins.

Memory for processes. Process memory itself is split up into several segment. However only one segment is important in our context, the heap segment. This segment of memory is where the process keeps all of its dynamically allocated memory. In most applications dynamically allocated memory is the majority of the memory usage.

However all of this matters little if the applications on the system being monitored does not allocate memory dynamically or does so very little. Database systems belong in this category. The only memory dynamically allocated are small buffers used in the client connections. The rest is usually allocated in huge blocks during startup.

We can therefore conclude that it is not particularly relevant for our purpose.

4.5 Job load

Instead of estimating how much any of the subsystems are working, it is also possible just to look at the number of jobs running. This assumes that all jobs executed are equivalent in the sense that they are resource-limited by the same subsystem. The difference between this domain specific number of jobs and the number of jobs in the run queue is that the run queue only contains those jobs that are ready to be run. It may be that a job is waiting for some other important resource, e.g. a lock.

The implementation of getting the job load is specific to the domain. In MySQL, it is possible to execute queries that will give back how many connections there are and how many active jobs there currently is. The query to retrieve the latter is “`SHOW STATUS LIKE 'Thread_running'`”. That returns a row with two columns. First column is the string “`Thread_running`” and the second column is the number of current threads that are executing, including ourself.

4.6 Concluding remarks on load estimation

Estimating load for the primary subsystems is a non-trivial task. Even the simplest of them all, estimating CPU load requires a complex scheme to get reasonable results. Estimating disk is close to impossible because of lacking support from the kernel and the impossibility to predict behavior precisely. Network load is trivial under assumptions that only the network interface of the local node is the bottleneck. Beyond that further kernel support is required that is simply not available and even if it were, nothing could be done about the problems it indicate. Memory load is not useful for applications that allocate most of its memory usage before clients begin to connect. Even for those were it is useful, the problem would probably require more work than simply limiting the number of jobs on the node.

Combining the load of the subsystems is trivial. It is as simple as adding them. Unfortunately the CPU load is the only one that is really usable.

Job load is trivial since it can just be read, but is domain specific.

Chapter 5

Design

A really great man is known by three signs...
generosity in the design,
humanity in the execution,
moderation in success.

- Otto von Bismarck

In the previous chapters, we have described the concepts behind the different components of the system. In this chapter we will continue with the overall design of the load-balancer. Starting by evaluating the different options with regard to its incorporation in MySQL.

5.1 Integration with MySQL

When integrating out load-balancer into MySQL, our main goal is that the client applications should not have to be recompiled. As result of this the only thing we can change on the client side is the MySQL client library, which all applications uses to connect and communicate with the database. This library is called `libmysqlclient`.

As described in chapter 2, the execution of the scheduling algorithms can be placed either on a frontend, on the clients or on the nodes. We will now look at how each of these possibilities can be combined with MySQL.

If the choice is placed on a frontend, the best solution would be to make the frontend understand the MySQL protocol. This will allow us to move the connections to another node, without the client noticing. To implement this, either the protocol must be reimplemented, or somehow the protocol code already present in MySQL must be reused. Neither of these tasks are easy, and both will take a substantial amount of execution time. For the reimplementation we would have to examine the MySQL protocol in order to determine how it works. For the reuse of the MySQL code, the protocol code would have to be extracted from MySQL. Another option is to add the frontend to the MySQL source tree. In all cases a lot of work will have to be done in order to make the frontend cooperate with the MySQL code.

The other frontend solutions does not require this intricate knowledge of how MySQL works, but they do not solve the problem well enough to be really useful to us. Especially the fact that a client connection cannot be rescheduled makes them impractical.

If the choice is to be placed on the client, we concluded that there were two options. The first is to have the scheduling algorithms included in the application that connects to the MySQL server. This would be done by implementing the scheduler logic in the `libmysqlclient` library. The other solution is to implement a daemon running on the client host, to execute the scheduling algorithms. The client application would then connect to this daemon in order to determine which node it should connect to. With this solution, we will have to make `libmysqlclient` connect to the local daemon in order do be redirected to the correct node.

The last option is to place the choice on a node. In this case, we have two options for implementation. Either we make something that resembles a frontend solution, where our program listens on the MySQL port, and then determines where to redirect the clients when they connect. The MySQL server will then have to listen on another port, but that is a simple configuration issue. This solution has the same issues as the similar frontend solution. I.e. it is necessary for our program to understand the MySQL protocol in order to move the clients after their initial assignment. The other option is to implement the scheduling logic inside the MySQL server. This will require that the scheduler is somehow inserted into the protocol layer, so that it can understand the queries issued, and only reassign clients when they have no outstanding transactions. Again this will require knowledge of how MySQL works internally.

The two best solutions are definitely the frontend and the client solutions. As already mentioned, to move the choice from the frontend to the node does not change much. Between the frontend and the client solution, the frontend solution is clearly superior in terms of flexibility and features. The frontend solution is also much more complex, and will take more time to implement. It will require reading and understanding much of the MySQL code. The client solution can be implemented almost without touching any MySQL code. The only part that need to be changed is the `libmysqlclient`, which is a relatively small C library, with a well documented interface. This makes it easy to find the places to insert node-changing code.

Because the client solution is so much easier to implement, we have chosen to implement the system this way.

5.2 System overview

As already mentioned in section 2.5.1, the best solution (measured in data transfered and in the number of connections) is to implement two daemons. One for running on the nodes, in order to gather load-information. These daemons distribute the load information between the nodes, by broadcasting it to the network they all are connected through.

The other daemon is the local load-balancing daemon for the clients. The job of this daemon is to make the scheduling decisions, and to update the load information from one of the daemons running on the nodes periodically. This update is done across a TCP connection, which is initiated by the clients.

The third part of the system is the `libmysqlclient` library. This must be modified to connect to the client daemon, and find out which of the nodes it should connect to. This connection is made through a local UNIX socket.

The rest of the chapter will give an overview of the design and structure of the three different components in the system. We will also focus on what security measures that can be made for all three parts of the system, in order to prevent misuse of the system.

The code is well commented, and the full documentation can be generated with the `doxygen`-tool. Graphs, the Doxygen documentation, a copy of the source code and a pdf version of this paper can be found at:
`http://davh.dk/projects/mysql-load-balance/`

5.3 The daemon for the nodes

In this section we will describe the problems and choices we encountered during the implementation of the daemon for the nodes in the cluster. This daemon will in the following be called `loadbalanced`.

The core of the `loadbalanced` daemon is the function that gather the load information. These are implemented separately so they can be easily be replaced, if `loadbalanced` needs to be ported to another platform than Linux 2.4. The functions to implement, if the functionality is to be ported to another platform are:

`void init_loadfunctions()`
: This function is called before any of the load functions are called. Its purpose is to initialize the different variables that are needed for the proper functioning of the load functions.

`void shutdown_loadfunctions()`
: This function is called before `loadbalanced` is stopped, in order to clean up after `init_loadfunctions`.

`void read_cpu_load(struct load_t &l)`
: Read the current cpu-load of the system. The method used for this is described in section 4.1 on page 31.

`void read_disk_load(struct load_t &l)`
: Read the current disk load. This function was implemented before the analysis in section 4.2 on page 34 was conducted. So the current version of the code still uses the disk load.

`void read_net_load(struct load_t &l)`
: Read the current net-load, and return the load for the last 15 seconds, the last 1 minute and the last 10 minutes.

`unsigned int read_client_load()`
: Retrieve the number of currently connected clients.

The `loadbalanced` daemon calls these functions every 5 seconds in order to get a sample of the load. Once all the information is retrieved, the information is broadcast to the other nodes.

Every time the `loadbalanced` daemon receives a broadcast packet, the information retrieved from it, is inserted into a `hash_map`. When a client then asks for the list of nodes, the load information is compacted into one number, which is then sent off to the client. This compacted load number is of course cached.

5.3.1 Security

`loadbalanced` has both input and output to and from broadcast and clients. So in order to evaluate which security measures must be taken, we will analyze these cases.

There is no real security problems with broadcasting the load-information to the local network. The only information that can be gained from this is the load and IP-address of the node. The load-information is not really sensitive information, and IP-address is readily available through other means.

The information received through the UDP port, on the other hand, must be handled more carefully. First of all, it is not possible to determine the origin of the packet, as it is easy to send an UDP packet with a fake sender address. This can be used to inject false load-information into the system. There is no easy solution for this, as long as the UDP protocol is used, apart from cryptographic signing of the messages. Even though the daemon cannot check the sender, it can guard itself against maligned packets, by ensuring that the contents of the messages are at least valid and reasonable.

When `loadbalanced` communicates with the client, it has no way of determining if a client is a malicious computer. This will allow any computer with access to the `loadbalanced` daemon to retrieve a list of machines that exists on the database network, and from this attack these. A way to prevent this, is by only letting the known clients connect to the socket, either by using a firewall, or by having a list of allowed clients in the `loadbalanced` daemon. Apart from this the input from the clients should be validated properly.

As the current implementation of the `loadbalanced` daemon is only a proof of concept, few of these security measures are implemented in the system. These will, however have to be implemented before the system is used in a production environment.

5.4 The daemon for the clients

This section will describe the client side daemon in detail, especially with regard to the interface available to the scheduling algorithms. The daemon will in the following be referred to as `lbclient`.

The `lbclient` is the daemon which serves as an interface between the client-applications running on a client and scheduling algorithms. Thus the client needs to do two things. First it needs to be able to communicate with the load daemons running on the nodes in order to figure out which nodes exist and are alive, and also to retrieve the load information for them. Secondly it must, from this information, be able to select a node for a given client to connect to.

The first part is done through a TCP connection as already mentioned in 2.5.1. In order to make the system fail-safe, we will have to make a list of nodes that the `lbclient` can connect to. If one of these does not work, the next one on the list will be tried instead.

5.4.1 The scheduling system

The scheduling system implemented in the client is implemented so that it is easy to add new scheduling algorithms. This makes it easier to try out different scheduling methods, as the barrier for implementing one is low.

The interface consists of two classes. The `Node` class which represents a single node in the system. The class keeps track of the last reported load and job information for one node, as well as the artificial load information. The `Scheduler` class is an abstract class defining the interface of a scheduling algorithm.

We will now explain the usage of the `Node` class, and afterwards which functions are expected to be implemented by a `Scheduler` class in order for it to work.

The `Node` class contains the following methods:

`void set_load(float load, int job)`
: Calling this function, will set the load and the number of jobs that is currently running on the node. It will also result in a readjustment of the artificial load of the node.

`void increase_load()`
: Increases the artificial load on the node. This works by increasing an internal counter, each time the function is called. `set_load` then looks at this number, in order to determine how much artificial load to add to the node.

`void decrease_load()`
: Decreases the artificial load on the node, by decreasing the internal number for the artificial load.

`float get_artificial_load()`
: This function returns the artificial load combined with the real load of the machine. This number is needed by load informed schedulers.

Apart from these functions the class also have two public variables, `weight` and `jobs`. `weight` is the weight for the node, as reported by the `loadbalanced` daemon, and the `jobs` variable is the last reported number of jobs running on the node.

The `Scheduler` class is an abstract class that describes the interface, that all scheduling algorithms must support. It contains the following methods, some of them which needs to be implemented for a new scheduling class:

`Scheduler(node_map_t &sm)`
: The constructor for the scheduler. This makes sure that a `node_map_t` called `nodes` are available to the other functions. This provides a mapping between IP-addresses and nodes.

`void new_node(u_int32_t node)`
: This function is called whenever a new node is discovered. The parameter is the IP-address of the node. This can then be used to lookup the node in the `nodes` map.

`void dead_node(u_int32_t node)`
: This function is called when a node is declared dead, i.e. when it is no longer present in the list received from the `loadbalanced` daemon. The parameter given is the IP-address of the node; just like `new_node`.

`void on_load_update()`
: This function is called whenever the load information in the `nodes` map has been updated. This allows the scheduling algorithms to adapt their internal structures to the new load and job information.

`Node const * get_next_node(int client_id)`
: Returns the next node for the client to connect to. This is the function that actually implements the scheduling algorithm. The `client_id` uniquely identifies the client that is going to make use of this node.

`void forget_client(int client_id)`
: This function is called whenever a previously returned `client_id` is no longer valid. Once this function has returned, the `client_id` is likely to be reused. So the function should remove any references to it.

So in order to implement a new scheduling algorithm, only four simple steps are needed:

1. Implement a class for the new scheduler that inherits from the `Scheduler` class.
2. Add the scheduler to the `scheduler_t` enumeration.
3. Edit `create_scheduler` so that it returns the new scheduler when it is called with the appropriate `scheduler_t` value.
4. To make it possible to select the scheduler from the configuration file, an entry will also have to be added to the `schedulers` variable declared in the function `read_configuration`.

All these changes need to be done in the `client.cc` source file.

5.4.2 Client security

The client receives data from two sources. One is the `loadbalanced` daemon running on one of the database nodes, and the other is through the local UNIX socket, which `libmysqlclient` uses to connect through.

The connection to the `loadbalanced` daemon is initiated from the client. So the client does not listen on a TCP-port, which makes it less prone to attacks. However the client still receives information from the `loadbalanced` daemon, which might not be correct. If the client receives malformed data in this way, the client should disconnect from the node, and connect to one of the others in its list of nodes. However even with this precaution, it is impossible to make the client secure, when the node it is connected to is compromised. This is because the client needs the node to tell it which nodes exist that it can issue requests to. Also for the algorithms that require load information, this information needs to be read from the node. If the node returns incorrect information it is able to direct clients to non-existing nodes, thereby causing the system to fail. It is therefore imperative that the node is not vulnerable to attack, as this will be able to affect the client.

The UNIX socket that the client applications connect through, also needs to be protected. This problem can be separated into two parts. The first is access control to the socket, i.e. who should be able to connect. The second what should be done about applications connecting to the client, but which are not keeping to the protocol.

The access control to the socket, can be controlled by basic UNIX file permissions. One option is to restrict the read access, so that only programs running as the same user as `lbclient` will be able to connect to the socket. This is a good solution, if for example all the applications are run as the web servers user. However if more than one application needs access to socket, this solution will either require that the two applications are running as the same user, or that the socket is made accessible by more than just the user.

The second security consideration for the client is the data received through the UNIX-socket. The data that need to be sent to the client, is only one byte, telling it that a new node is required now. Since the protocol is this simple, we do not need any special handling of the input data. However it would be a good idea to require the one byte to be certain value, and if it is not, close down the connection. This way we make sure that an application sending too much data, does not cause scheduling algorithm to be run many times, without a client actually using the result. Another way to do this, could be to "cache" the result returned to the application, so that if it asked more than once in, say 5 seconds, it would return the same node.

In its current implementation the client does not implement most of these security measures, as it is only meant as a proof of concept. However if this solution is ever used in a production environment, the above mentioned security measures needs to be implemented, in order to assure the systems well-functioning.

5.5 Changes to `libmysqlclient`

In this section we will highlight the issues we encountered when making the MySQL client library connect to the local `lbclient` daemon, and then connecting to the node supplied by the daemon.

The first problem is how to make it possible for `libmysqlclient` to connect to the local load daemon. The trouble is that we wish to allow the same library to be used both with and without the load-balancing. This is required, as some client programs connect to several different MySQL servers, something that the rest of our system does not support.

There are two ways to solve this. One is to provide an alternate `mysql_connect` function (i.e. `mysql_connect_lb`), that will initialize the connection to use load-balancing. The other solution is to have a special hostname that results in the load-balancing being activated. While the first solution is the cleanest, the second solution fits in better with the demand that no client applications should have the need to be recompiled in order to use the load-balancing.

The next problem is how to make the client-library connect to the different nodes. Because we have chosen to enable the load balancing through a connect string, we need to make the library aware of this connect string. The obvious place to do this is in the function called `mysql_real_connect`. This function is the lowest of the connect functions. It determines amongst other things if the connection is to a local UNIX socket, or to a TCP/IP socket. Our code is placed just before it determines to connect to a TCP/IP socket. This way we can change the IP-address that the function tries to connect to. So with this change in place, we can now connect to a machine, chosen by the `lbclient`.

The only problem left now, is how to change the connection to a different node. In this case, the function called `mysql_send_query` can be used. This function is already used by the build-in load-balancing to change node, so it is a natural place to insert our code. We choose only to change the node, when the library receives a `START TRANSACTION` or a query beginning with `BEGIN`, in order to keep the applications on the same node during a transaction.

These two simple changes make `libmysqlclient` capable of connecting to a node given by `lbclient`, and to change the connection another node.

Chapter 6

Benchmark

Just because we increase the speed of information doesn't mean we can increase the speed of decisions. Pondering, reflecting and ruminating are undervalued skills in our culture.

- Dale Dauten

Now that we have described the different parts of the load-balancing system, the missing piece is to test the different scheduling algorithms against each other, in order to determine which of these works best.

In order to measure this, we will need some benchmarks. The rest of this chapter describes the system used for testing, the benchmarks and throughly analyzes the results of the benchmarks.

6.1 The benchmark setup

There are many possible environments the system could be tested in. We test the system in two different environments that we have available.

The first is where all the nodes are equal. This setup have 4 Dell PowerEdge 1650, each with a 1.4GHz Pentium III and 1Gib memory. All data is located on a RAID 0. The RAID is provided by the perc3/di SCSI controller on two physical disk. Each disk spins with 10K RPM and is capable of a sustained transfer rate of about 40Mib/s. Each machine also have a 1Gbit network interface. The client used is a dual Pentium III 1GHz with 1Gib memory and with a 100Mbit network interface. The entire setup is connected through a 8 port 1Gbit switch.

The second setup is the same as the first, except that a fifth machine has been added to the cluster. The machine is a Pentium II 350MHz with 256Mib memory. There is only a single disk in the system, an IBM 7200RPM IDE disk. The network interface is a 100Mbit Intel Pro/100. This machine is connected to the 8 port 1Gbit switch through another 9 port switch. This 9 port switch has a 1Gbit uplink to the Gbit switch and the 8 others are 100Mbit ports.

Each benchmark, except the latency test, is with 8 worker threads per node. The latency test only uses a single thread. Each of the benchmarks are run with a ramp-up-time of 30 seconds and a runtime of 360 seconds. The ramp-up-time determines how long the system will run the benchmark before it starts recording the results. Every transaction ended during ramp-up-time is not included in the benchmark result, but a transaction started during the ramp-up-time and ended after is included. The ramp-up-time is used to stabilize the system before the benchmark is run.

6.2 The benchmarks

In order to measure the performance of the different scheduling algorithms, we will need some benchmarks.

We need to determine what we need from the benchmarks. First of all they need to be read-only, since our system is not capable of handling writes. This also includes the use of temporary tables. Secondly, the benchmark should be able to run on a MySQL database.

Not many standardized benchmarks have these properties. Therefore we have opted to make our own. This also allows us to craft benchmarks to show a particular property of one of the scheduling algorithms.

6.2.1 The database schema

We have 3 different tables.

The first table "`a`" has two columns. First column is an id and is the primary key for the table. Second column is a number. The table has 10000 entries, all have the number 1 in the second column, except for a single row that has the number 2.

The second table "`b`" has the same columns as "`a`", but only 1000 rows. Again, only a single of the rows contain the number 2.

The third table "`c`" is larger, so that most machines would have to visit the disk in order to give a result. To get the total size of the table up, a third column is added with a 200 character long string. The other columns are the same as "`a`" and "`b`". The table has 1000100 rows and 100 of the rows contain the number 2.

6.2.2 Short CPU bound transactions

Short CPU bound transactions are designed to make it impossible for any scheduler to place the transaction correct. Scheduling this query type is difficult, because the impact on a node is both little and very short. We measure the time needed to below 0.1 second.

The query itself is a point query into "a" to a random id. E.g. "`select * from a where id=9474`".

We do not run the short transactions without mixing them with other types of queries, because the client becomes the bottleneck.

6.2.3 CPU bound long transactions

To make it easier for the schedulers to detect overload on a node, a longer running query was designed. This query is a cross join between "`a`" and "`b`" and then find the one tuple where the numbers in the second column add up to 4: "`select * from a,b where a.tal+b.tal=4`". The time consumed running by itself is about 2.5 seconds; 8 concurrent threads increase the time to about 18.25 seconds.

6.2.4 Disk bound transactions

The disk bound transactions serve a special purpose, because they make it difficult to predict the impact of scheduling another transaction on the same machine. Therefore the load informed scheduler is extra punished by this kind of load. The job informed, however, do not care about the type of query and should thus be unaffected.

The query we use is a range query on “`c`”. Each query tries to find the maximum number in the second column for a range over 5000 ids: “`select max(tal) from c where id > 550547 AND c.id < 555547`”. This query can be executed as a sequential disk read.

6.2.5 Mixing long and short

Each of the type of benchmark mentioned so far is designed to prove different points about the system. In a real situation, the work load of a database would not be limited to just one type of transaction. If the nodes in the cluster are equal in capacity, running just one kind of query will favor the round-robin, as it treats all nodes alike.

To eliminate these issues, a new kind of benchmark is needed, which uses a mix of the different types of queries. Since the short transactions are created to confuse the schedulers, a good mix would be a mix of the short and long transactions, or short and disk transactions. The factor which the queries should be mixed with, is also important. If there are too many long-running transactions in the mix, the load of the short transactions will not be noticeable. Therefore three different mix factors are used: 3 short transactions to one long, 25 short to one long and 100 short to one long. So we have 6 different transaction loads.

When generating a mix of queries, the mix factors are used as a probability factor. So a mix of 3 to 1 means that there are 3 times as high a chance of selecting the first transaction, as there are of selecting the second transaction. Using a probability factor allows us to generate the mixes randomly, which minimizes the chance of the factor colliding with the number of machines. This would otherwise happen with the round-robin scheduler, when it is run on four machines with a 3 to 1 transaction mix.

With these benchmarks we expect to prove that the round-robin algorithm performs badly, when a mix of different transactions are used. Also we expect that the job and load informed algorithms will perform better. As the mix moves against more short transactions we will expect the performance of the round-robin scheduler will improve, as the chance of putting a long job on a machine that already has one becomes smaller.

6.2.6 Latency

Latency testing is needed to show how much direct overhead the scheduling introduces. One way of testing is done by connecting to this node as a cluster, i.e. using the scheduler or connecting directly to the node. This will show the overhead of the communication in the system. A second method is to use a cluster of many nodes, but only a single client connection. This will show the overhead of the scheduling algorithm. Combined these two should hopefully be able to show that the overhead of the scheduling is negligible.

We measured the latency by evaluating the query `begin transaction` with a single client connection.

6.2.7 The weight for the round-robin

When the fifth machine is included in the benchmarks, the weights for the weighted round robin algorithm needs to be adjusted to the fact that not all the machines are equal. In order to choose the weights so that they match the abilities of the machines, we have run the CPU bound long benchmark on a single Dell machine, and on the fifth machine. The same is done for the long disk-bound query. The results from these four runs can be seen in the following table:

	Dell	The slow machine	Ratio
Long CPU bound transactions	166	41	4.0
Long disk bound transactions	1850	422	4.2

A good choice for the weight would be to let the Dell machines have a weight of 4 and the slow machine a weight of 1. However our initial tests for this showed that the slow machine was five times as slow as the Dell machines. Because of the wrong measurement all the benchmarks are run with the weight for the Dell machines set to 5, and the weight of the slow machine set to 1. Also as mentioned in 3.9 it is better to underestimate the slow machine, than overestimate it.

6.3 Results

The benchmarks produce a large number of graphs; 3439 to be exact. For each node in each experiment we monitor the current number of active queries, the CPU activity, the number of context switches per second, the disk activity measured in block, the disk activity measured in megabytes, the memory usage, the network interface activity, the socket usage and finally the response times for the queries executed. The response times are plotted with a histogram; Each query is placed in a one second interval; Each interval is plotted as the percentage of the total amount of queries. As a means to a quick overview, a bezier approximation is placed on the same graph. To prevent inflation of graphs only those that show anything significant is generated. E.g. for nodes with no disk activity, no graphs are generated for the disk. The precise exclusion criteria for each type of graph can be found in the shell scripts that generate the graphs.

For each experiment there are two additional graphs; One combining the number of active queries for all the nodes and the other combining the response times. The lines plotted in response time graph are the bezier approximation mention before. The response time graph also includes a line for the combination of response times.

For each class of experiment we plot the speedup ratio of the average response times with error bars and the throughput. The error bars are the standard variance of the average. A large variance, compared to the other algorithms, means that the load was poorly balanced.

The response time speedup ratio is calculated as the speedup over the same benchmarks result on a single node. Notice, that benchmarks on 5 nodes will have higher average latencies, because the fifth node is significantly slower. The throughput ratio is calculated differently, because it makes no sense comparing throughput directly. So instead of using the single node run as baseline, the weight of the single node run is used. This means that the five node runs throughput is divided with 21

(4 nodes of weight 5 and one node of weight 1) and then compared to the single node run which has been divided by 5. If the numbers are higher than one, the benchmark has resulted in a super-linear speedup. It should be mentioned that the weights used does not correspond to the actual performance-ratio of the machines as mentioned in section 6.1. This will however not affect the results significantly as the difference between 21/20 and 17/16 is approximately 1%.

The shell scripts used to generate the graphs can be found in `sqlreplay/`. The `gen_graphs.sh` analyze the data collected for each benchmark and generate the graphs and HTML pages for each benchmark. `overviewgraphs.sh` generate the HTML pages and graphs for each class of benchmark. `overviewgraphs_alg.sh` generate the HTML pages and graphs for each algorithm. `overviewgraphs_all.sh` generate the last three overview graphs.

6.3.1 Benchmark naming

The naming of the mixes are "`sm`<type><ratio>". E.g. smdisk100to1 is the benchmark where the number of disk bound transactions are one in a hundred and the short transactions are the remaining 99. `longdisk` is of course the disk bound transactions. Each benchmark is suffixed with the number of nodes used.

We originally had a query type that ran about ten times slower than our current long CPU bound transaction, which we named "long". But benchmarks showed that nothing was gained from this extra long query. So in the graphs, the `mediumcpu` refer to the query we mentioned as the long.

The algorithms are called `ji` for "Job informed", `jis` for "Job informed" with a short phase, `li` for "load informed", `rr` for "round-robin", `wrandom` for "weighted random" and finally `wrr` for "weighted round-robin".

6.3.2 Latency testing

When connecting to a single node, the latency of the request is about 0.102 milliseconds. The number of context switches per second also shows that there are two for every query submitted. The network round-trip time between the client and the node is very close to 0.08 millisecond, thus most of the work must be done somewhere in the network. This is verified by examining the CPU utilization graph (not displayed), which tells that 3% of the CPU time is actual work in the benchmark program. The majority of the remaining 0.02 milliseconds are then spent in MySQL processing the request.

Connecting to a cluster with a single node, the latency rose to 0.148 milliseconds, or 0.046 milliseconds more. Only the Load Informed scheduler seems to introduce a larger overhead, topping out at a latency of 0.162 milliseconds. The CPU percentage of time spent in the benchmark program and the scheduler rises to about 7%. Instead of two context switches per query, there are now seven switches. We can therefore conclude that the reason for the large time difference, yet small difference in the CPU time spent is the context switch overhead. Measured with `lmbench`, a context switch takes between 0.003 and 0.030 milliseconds depending on the amount of cache pollution between switches and the number of concurrent threads.

Increasing the number of nodes in the cluster to 4 makes no difference. Thus we can again conclude that the majority of the latency penalty is the context switches.

6.3.3 The Round-Robin scheduler

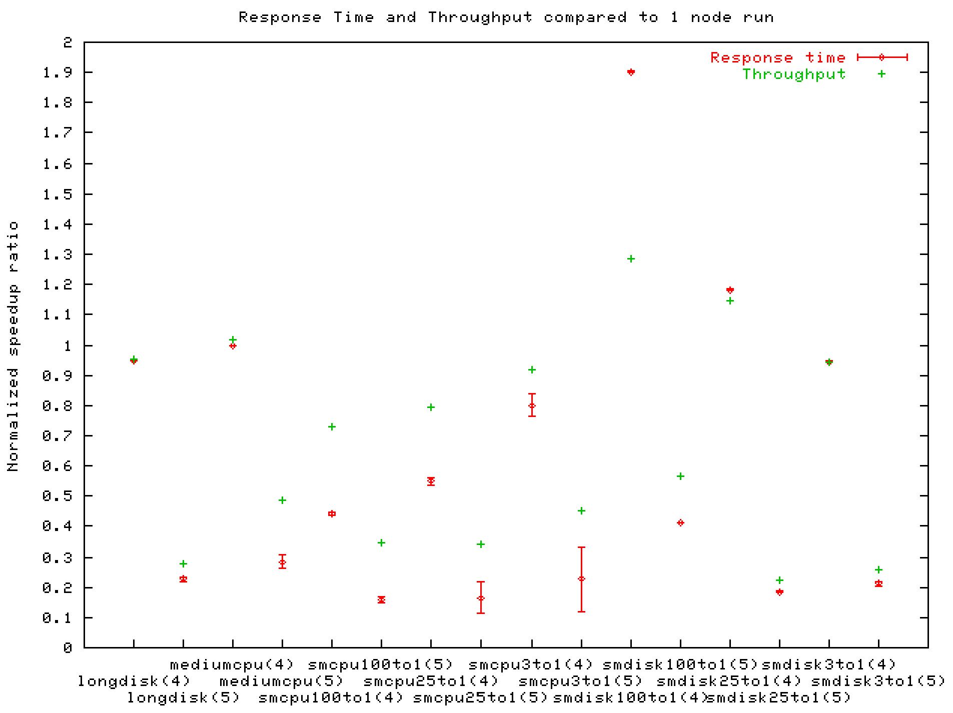

Figure 6.1 – *Response times and throughput for the round-robin algorithm.*

One of the advantages we gave in the previous chapter was that the round-robin scheduler is fast. But the latency test shows no difference to prove this appraisal.

Let us instead examine how it fairs in the benchmarks. We refer to the figure 6.1. The `longdisk(4)` and the `mediumcpu(4)` show the conditions, that we perceived as perfect for the round-robin scheduler. Both show an almost perfect speedup when adding more nodes to the cluster and the latency penalty is minimal. However, adding the fifth node completely breaks the algorithm; the speed is reduced to below a fourth, or about the speed of a single node. Thus proving our initial expectations. The `mediumcpu(5)` is not too bad; reaching about the double throughput of the single node. But the reason can be found in the specifics of the benchmark. It turns out that for this specific benchmark, the ramp up time is to low; Not until after 100 seconds (compared to the normal ramp up time of 30 seconds) the algorithms breaks completely (see figure 6.2).

Moving on to the query mixes. We already determined that the algorithm breaks when the fifth node is added, so we will ignore the rest those results, as they only pour oil on the fire. The CPU mix test shows something we did not expect, as the ratio of short transactions increase the performance degrades. The expectation was that as the ratio of short transactions increase the penalty of miss scheduling a slow query decreased. Thinking more about it, this is actually the wrong way of viewing the situation. When the number of short transaction increase, the likelihood of getting a worse result increase, because the short transactions become more and more dominant; thus the penalty of placing two or more long queries on the same

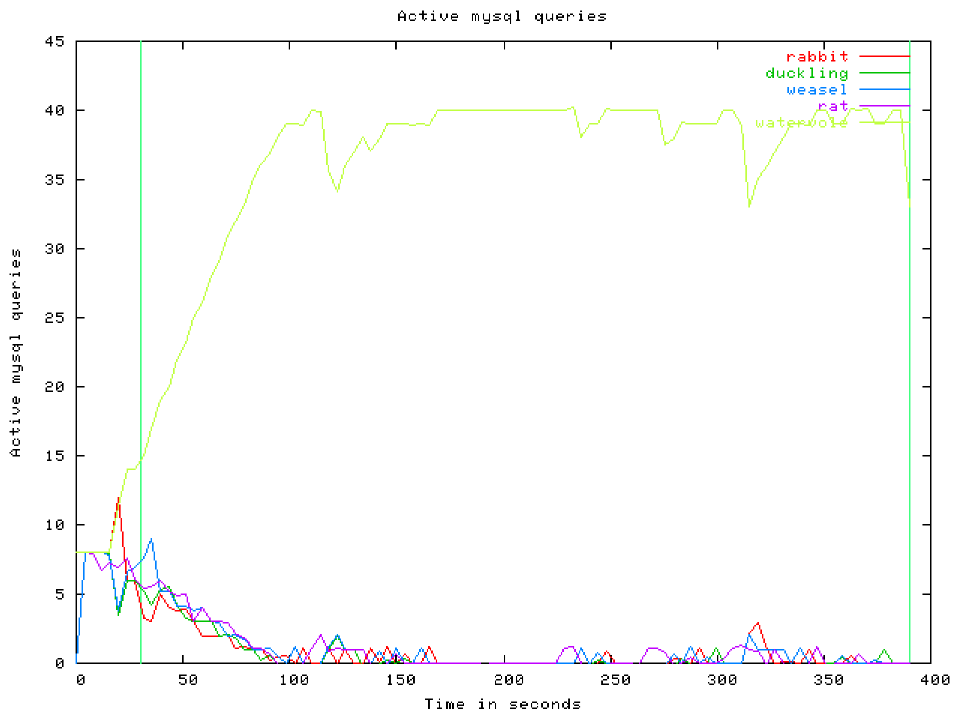

Figure 6.2 – *Graph over active queries on the nodes participating in the* `mediumcpu(5)` *benchmark. Notice that* `watervole` *ends up with all the work.* `watervole` *is the fifth, slower node.*

node becomes large. This is only the case for mixing CPU bound transactions, our expectation to the disk mix still hold.

The disk mix test show exactly what we expected, as the number of disk transfers increase the disk is more and more free to complete a single query before the next. We will return to these benchmarks in section 6.3.8 on page 60.

6.3.4 The Weighted Round-Robin scheduler

We introduced the weighted round-robin scheduler as a means to prevent the worst situations of using nodes of different capacity. From the benchmark results (figure 6.3 on the following page) we can see that this goal was largely achieved.

Let us first compare the results to the round-robin scheduling without weights on the 4 node runs. We can see that the results are identical. Therefore we can conclude that adding weights to the algorithm does not diminish the advantages of the general round-robin scheduler. It also tells us the benchmarks we have run variates little, and is thus trustworthy.

The five node run looks good; all result show that the throughput only variates little from a cluster of identical nodes. The response time is a little worse in all the benchmarks, but that number does not take the slowness of the fifth node into account. Thus the general conclusion of the weighted round-robin must be that the weights do exactly what they are supposed to do. Earlier we said that the algorithm was more work than the round-robin, but based on the latency test and the identical results we must conclude that it does not.

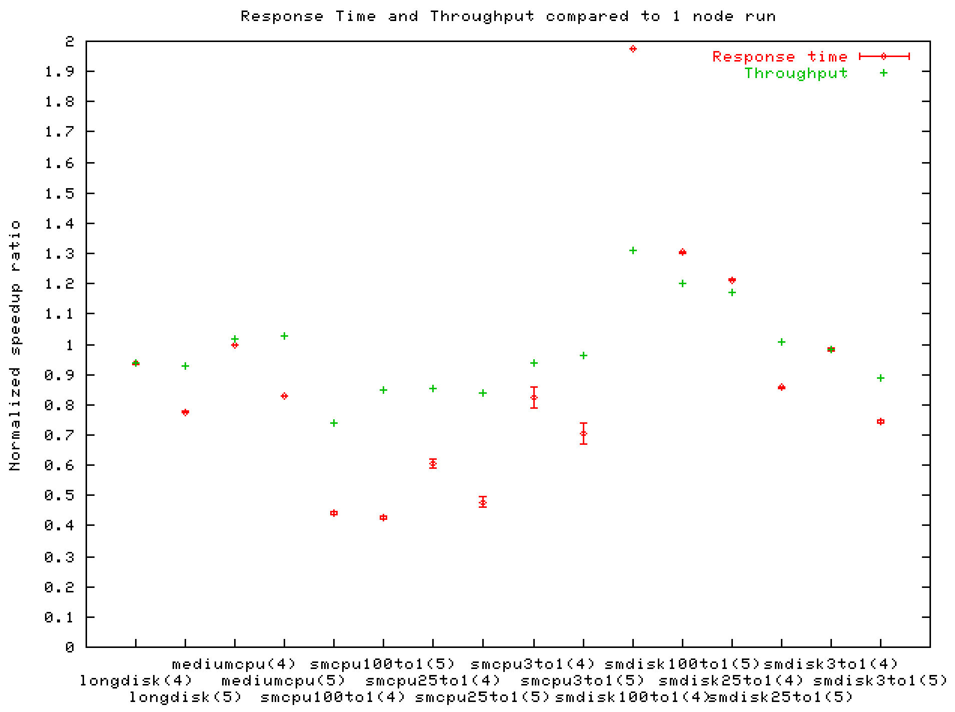

Figure 6.3 – *Response times and throughput for the weighted round-robin algorithm.*

6.3.5 The weighted random scheduler

The weighted random algorithm was introduced in its current form to prevent the round-robin based schedulers from hitting a worst-case scenario where the long transactions would otherwise be placed on the same node. We did not, however, make any such workload. Instead we can verify that the random choice is as good as the round-robin scheduler under normal circumstances. This can be seen from the results (figure 6.4 on the next page) where only the `smdisk1004to1(4)` and the `smdisk25to1(4)` differ from the weighted round-robin. Both in a negative fashion. It seems that the reason is that the randomization simply gets unlucky in its choice of node compared to the round-robin and misplaces a few queries.

6.3.6 The job informed scheduler

The job informed schedulers task is to prevent any node from getting overloaded. To begin with we had the client update the load information every 5 seconds and the nodes also polled the database server every 5 seconds. Thus the clients information could be at max 10 seconds old. The benchmark results for this configuration are shown in figure 6.5 on the facing page. The majority of the benchmarks are a few percent better than the weighted round-robin; an accomplishment that must be called excellent.

And for a while we though that this was the best the algorithm had to offer. Until we looked at the specifics of some of the benchmarks. For instance the `smcpu100to1(5)` benchmarks shows that something is wrong. Look at figure 6.6, which is the graph over the number active queries on all nodes when the phase

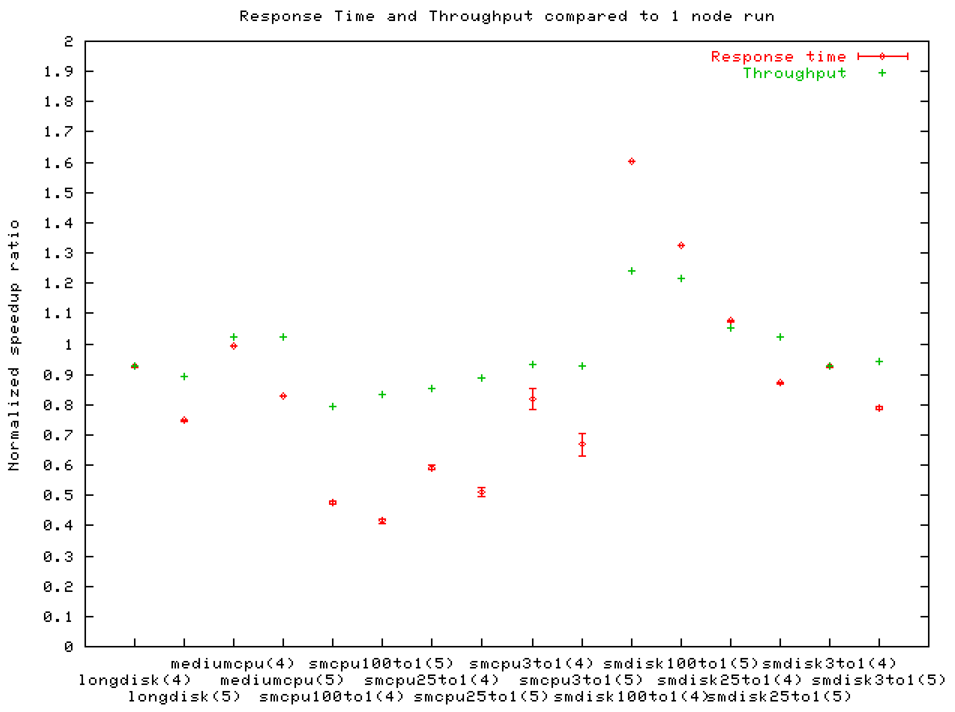

Figure 6.4 – *Response times and throughput for the weighted random algorithm.*

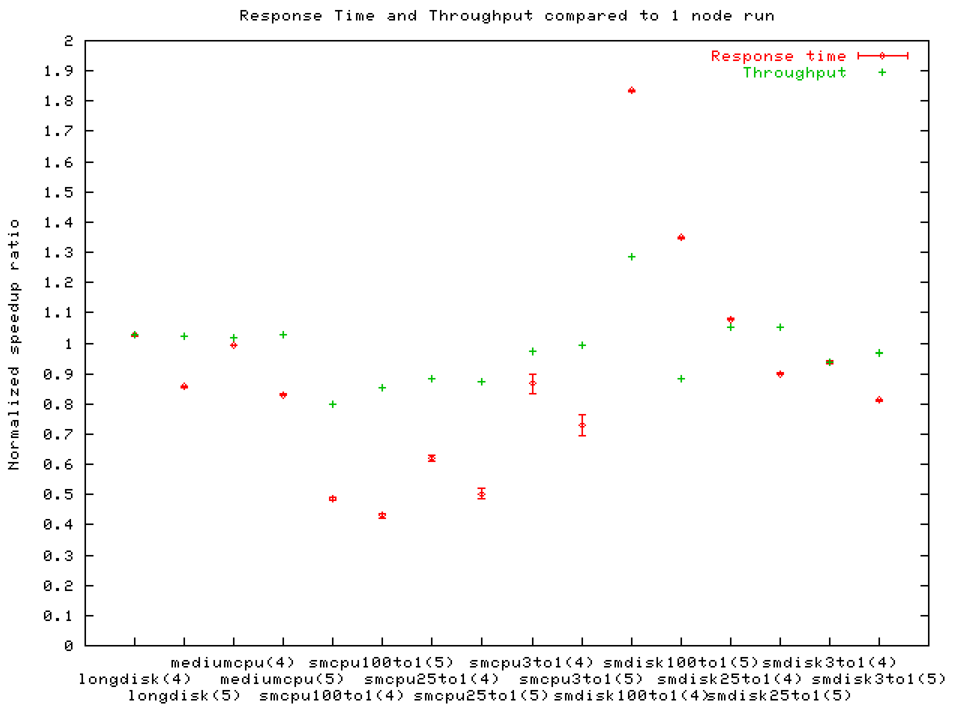

Figure 6.5 – *Response times and throughput for the job informed algorithm. Here with 5 second update and 5 second poll.*

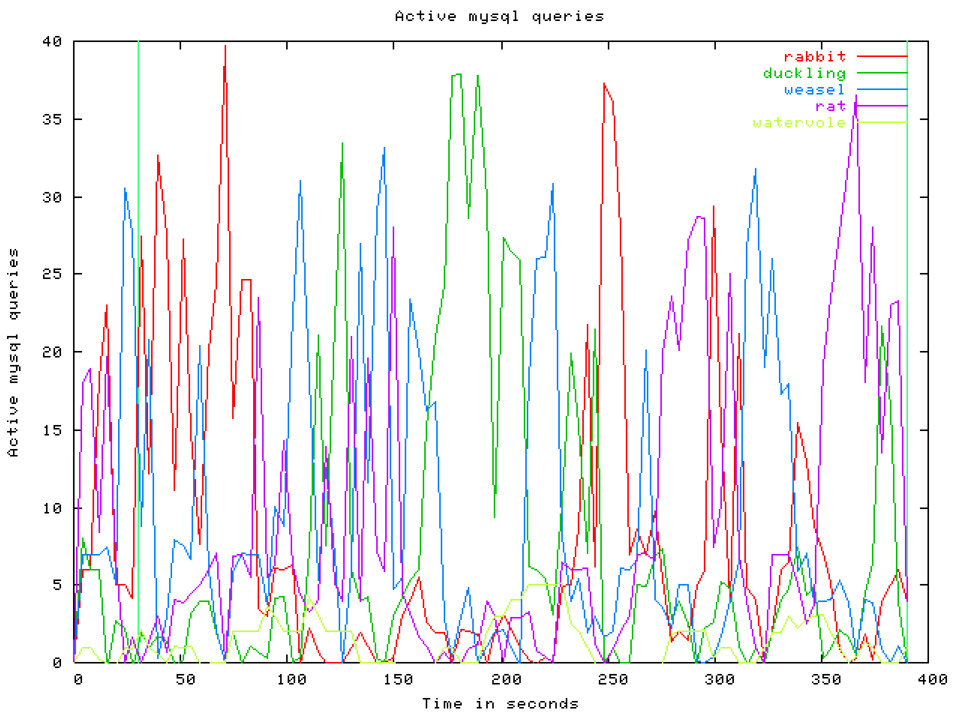

Figure 6.6 – *The smcpu100to1(5) benchmark with a maximum of 10 second old load information.*

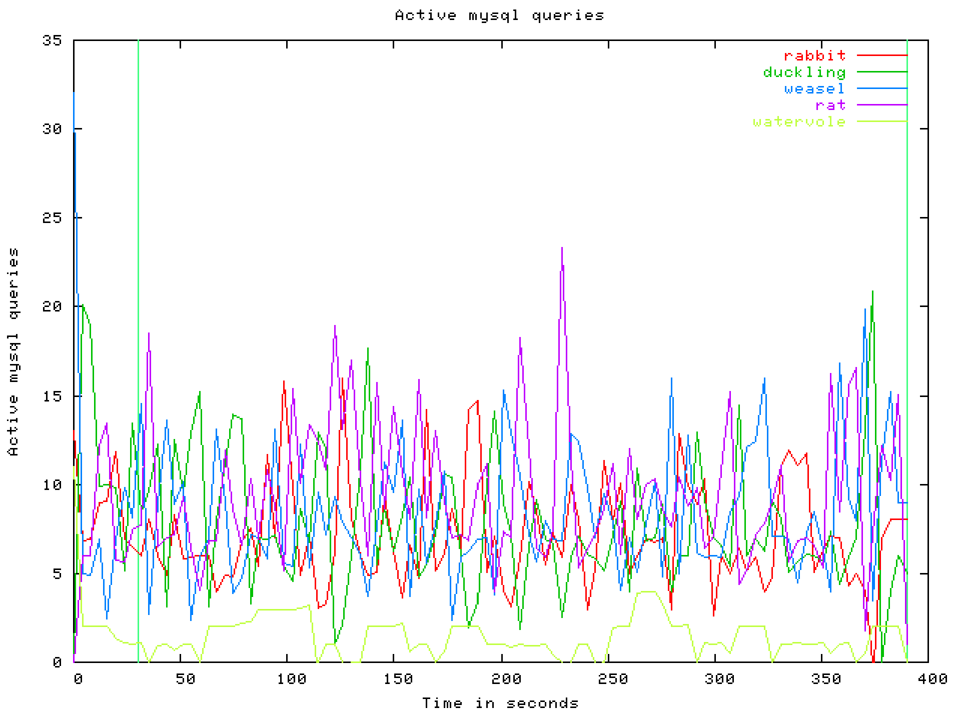

Figure 6.7 – *The smcpu100to1(5) benchmark with a maximum of 2 second old load information.*

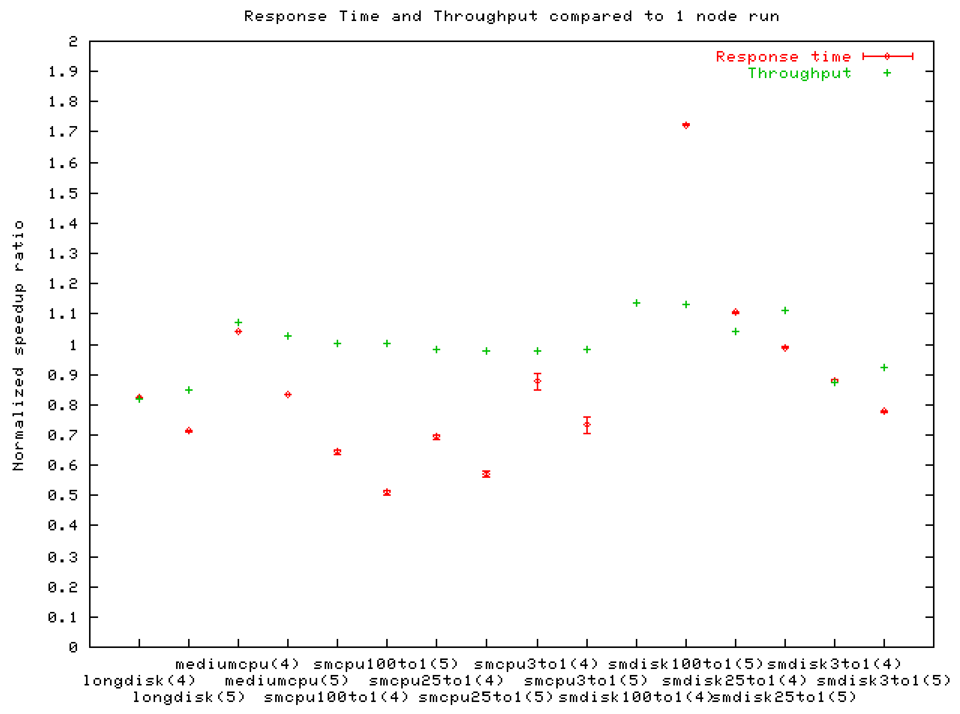

Figure 6.8 – *Response times and throughput for the job informed algorithm. Here with 1 second update and 1 second poll.*

length is 10 seconds. It is clear that the phase is too long by observing to long peeks of the nodes. We then tried to reduce the phase length to 2 seconds. Figure 6.7 is the result. No longer is there large periods where a single node is overloaded. The worst case is “rat” at 230 seconds, but that is quickly corrected. The result is that most benchmarks are improved by several percent. Only the benchmarks with disk bound transactions are worse; an abnormality that could be explained by a lower disk cache hit. What is strange is that the response times are much better; the `smcpu100to1(4)` is 2.25 times faster in response times than the single node benchmark.

6.3.7 The load informed scheduler

The load informed algorithm tries to do the same as the job informed, using different metrics. Figure 6.9 on the next page shows how good the attempt of this was. The answer is - poor. Every single benchmark on four nodes is beaten; on five nodes only the round-robin algorithm is worse. We can blame some of it on a phase length of 10 seconds like the first attempt on the job informed. The long phase does however not seem to be the biggest problem. Instead it seems that our poor attempt at adding some extra load to a node when selecting it goes very wrong. In the light of the job informed algorithms success, we decided not spend any more time on trying to fix these problems. Especially because it was not apparent what the fix should be.

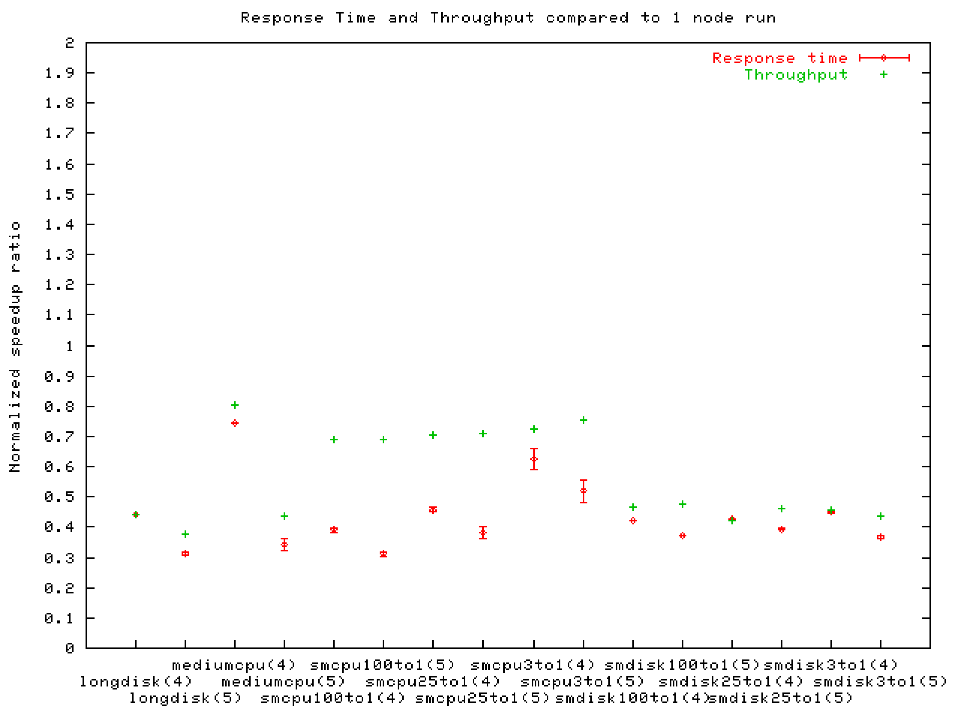

Figure 6.9 – *Response times and throughput for the load informed algorithm.*

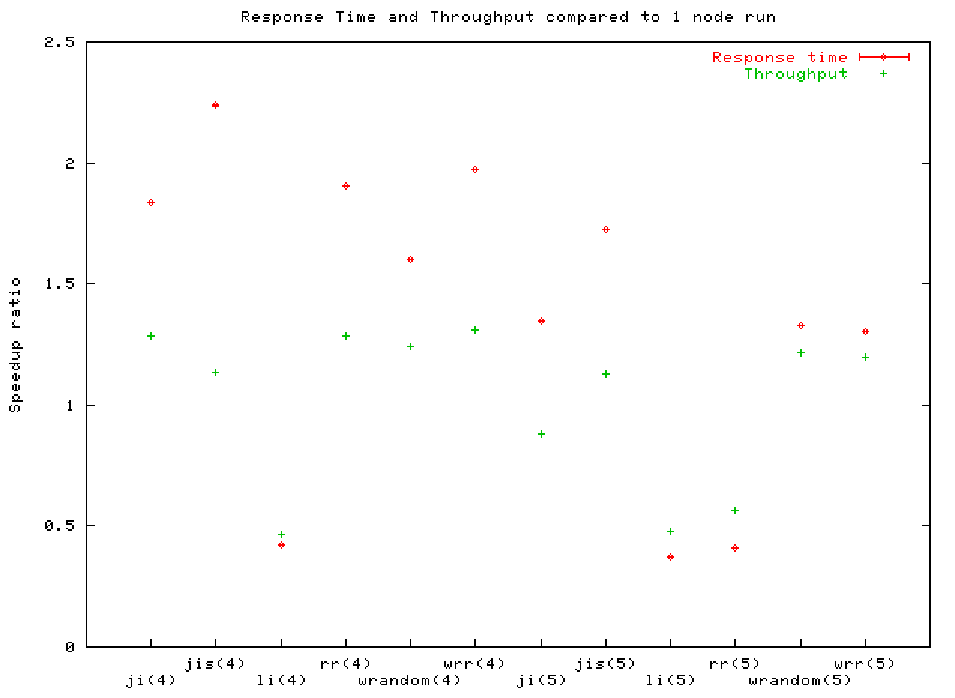

Figure 6.10 – *The mix with 1 long disk bound transactions per 100 short CPU bound.*

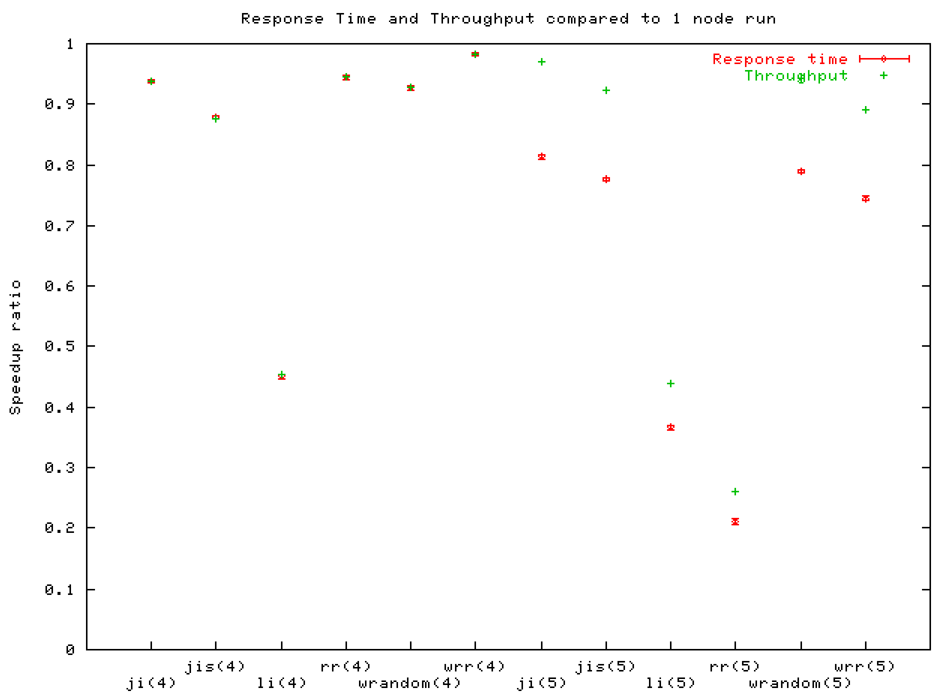

Figure 6.11 – *The mix with 1 long disk bound transactions per 3 short CPU bound.*

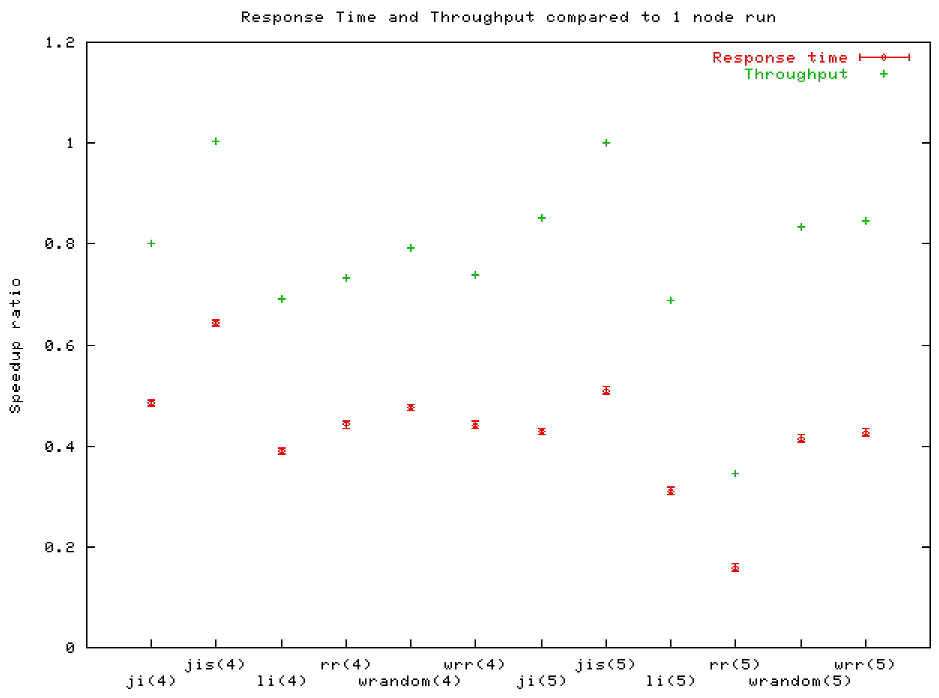

Figure 6.12 – *Worst performance for any benchmark (the `smcpu100to1`).*

6.3.8 The best and worst benchmarks

Some of the more spectacular results come from the `smdisk100to1` benchmarks (see figure 6.10 on page 58), because it achieves super linear speedup. The explanation can be found in way a disk works. When a single disk must satisfy all requests, the consequence is that disk requests are interleaved and the disk must seek several times for every request; instead of doing a single sequential read. The reason the super linear speedup is not as apparent in the throughput is that the execution speed of the disk activity is now only a minor percentage of the entire benchmark. An explanation that is confirmed by observing the results of the `smdisk3to1` benchmarks (see figure 6.11 on the page before). Instead of a super linear speedup, the result is at par with a single node. The reason being that the number of outstanding disk requests are comparative.

The worst performance of the cluster is from the `smcpu100to1` benchmark (figure 6.12). All algorithms, but `jis`, have a response time below half that of a perfect speedup and the throughput is only marginally better at about 75%. The problem arises when placing a long transaction on a node where another long transaction is already running. The result is that the long transaction accumulate on a single node. The job informed with the long phase does not react in time to prevent the situation, even if it has the tools to do it. When applying the short phase to the job informed, the power of the job informed approach is observed.

6.3.9 Summing it up

To sum up, we have compressed the response times of the different benchmarks for the different algorithms into a single graph in figure 6.13; and the same for the throughput in figure 6.14. From these we can extract the average numbers and display those in figure 6.15. However, the results from the `smdisk100to1` benchmarks variate to much from the rest of the benchmarks, so figure 6.16 has the averages without the `smdisk100to1` numbers.

The benchmarks covers a wide variety of situations and has generally fulfilled their task, which was to highlight the highs and lows of the algorithms we implemented. The round-robin algorithm showed exactly the bad performance we expected, when the situation is not ideal. The weighted version helped; when adding the fifth node, it was no longer overloaded beyond help. The performance of the random algorithm was a bit better than expected; we had anticipated that there would be more situations where a random choice was unlucky. The load informed was unable to deliver the performance it was meant to. The latency benchmark show that much more work is involved just selecting a node, the rest of the benchmarks show that even then, the algorithm is unable to perform even in the best of situations.

That leaves the job informed algorithm. We showed that a short phase made the algorithm perform much better. Even without this tweaking, the results are still better than any of the other algorithms. It does exactly what it was intended to do. Avoid the worst situations.

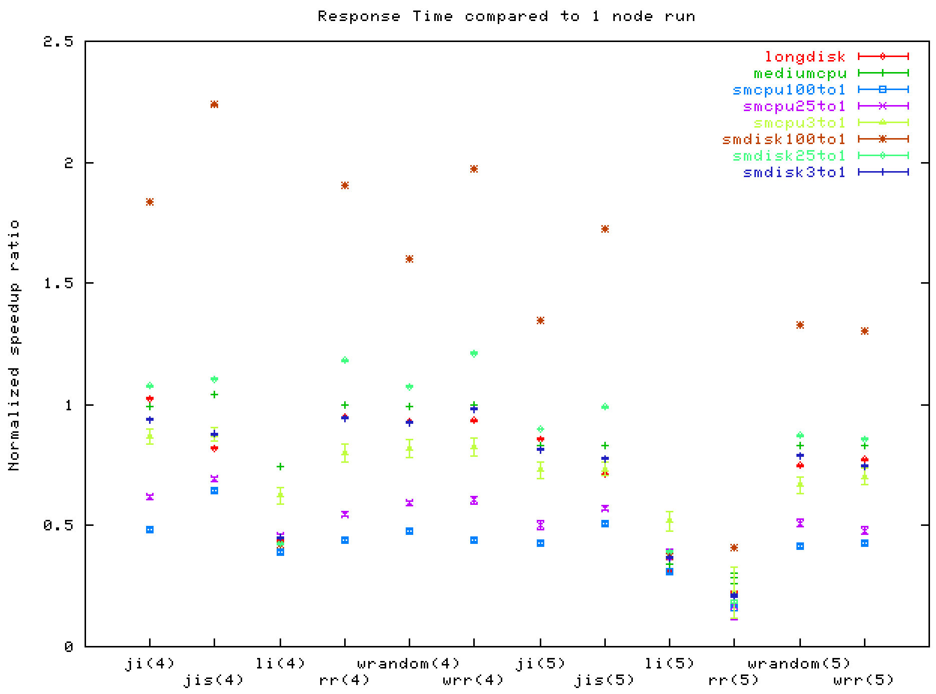

Figure 6.13 – *Graph showing the speedup of all algorithms with regards to latency, here with each algorithm separate.*

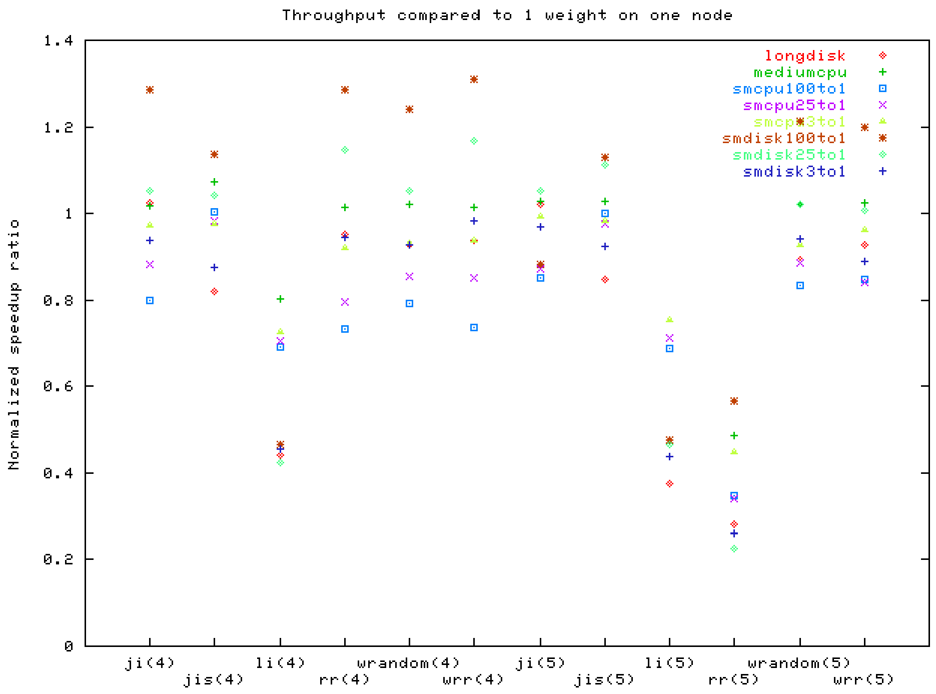

Figure 6.14 – *Graph showing the speedup of all algorithms with regards to throughput, here with each algorithm separate.*

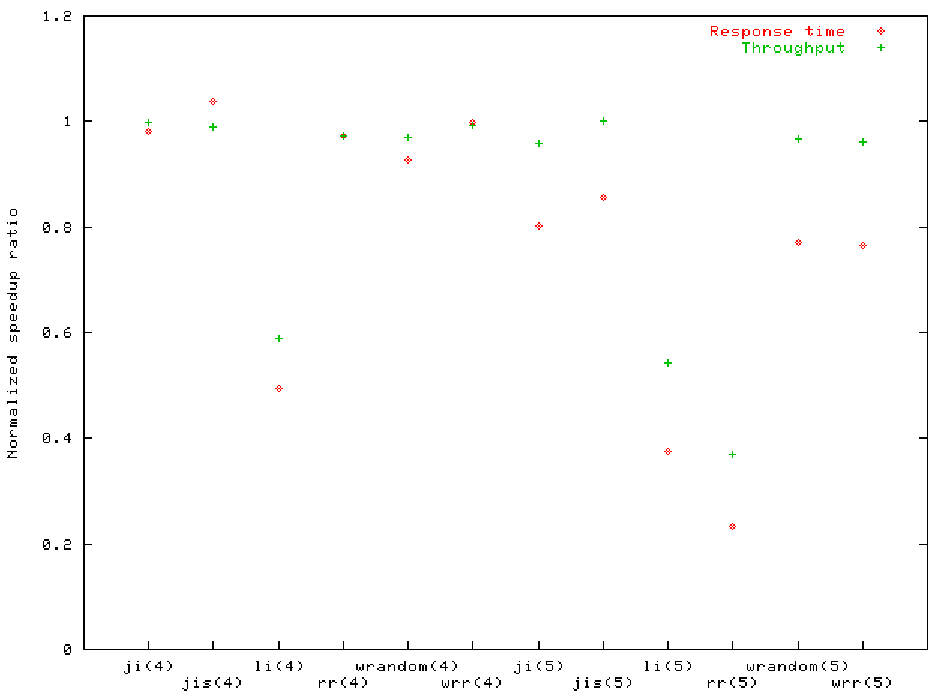

Figure 6.15 – *Graph showing the average speedup ratio of all algorithms both latency and throughput wise.*

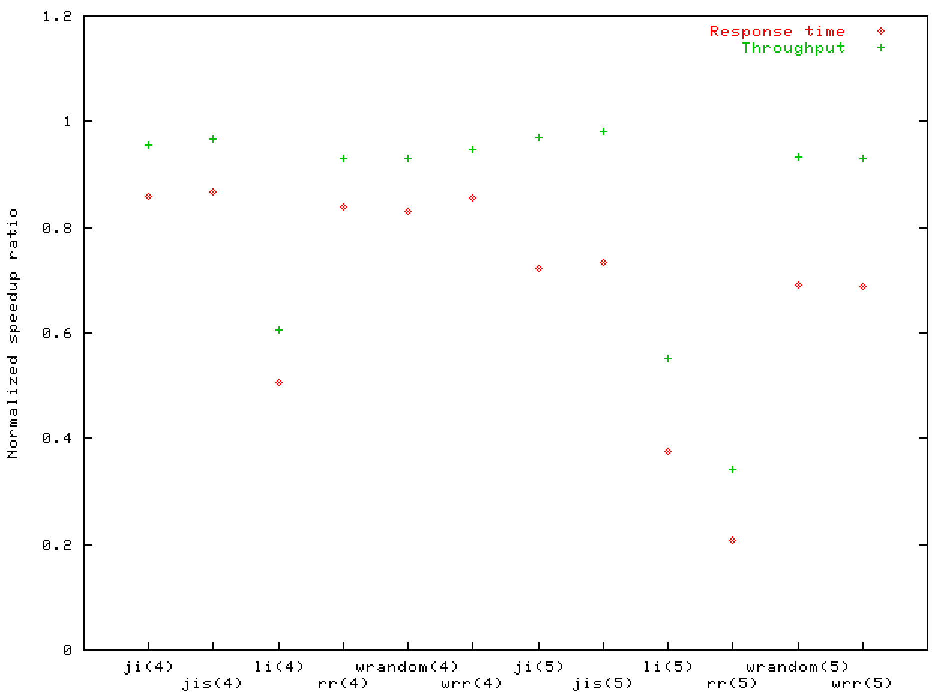

Figure 6.16 – *The average speedup ratio of all algorithms both latency and throughput wise, but without the `smdisk100to1` numbers.*

Chapter 7
Conclusion

[End of diatribe. We now return you to your regularly scheduled programming...]

- Larry Wall in Configure from the Perl distribution

In this paper we have presented several algorithms to equalize load among nodes in a cluster. Through benchmarks and through analyzing the different algorithms we have found a single algorithm that has several good properties. This algorithm is called "Job informed". It is platform independent, has the best performance of any of the algorithms we implemented and it was easy to implement.

7.1 Further work

A first priority is the integration of the load-balancing with MySQL. Using the job informed scheduler, everything that is needed in the MySQL server is that the nodes in the periodically transmit their current number of jobs to each other. Next is to integrate the client daemon into `libmysqlclient` (the C-API library). Instead of talking over a socket, we can use the observation that most often it will be the same user on the client that is using the database. Therefore, instead of a daemon handling the dispatch of jobs, a shared memory model could be used instead. All the complexity of choosing a node is then placed in the library code.

We experimented with having a short phase for the job informed algorithm. Further experiments could detect the optimal phase length. Possibly develop an algorithm to tune it dynamically.

Another scenario that was not tested was when there are multiple clients (or multiple dispatchers). We already showed a possible way of handling the situation, but no work was done to verify that it works.

None of the schedulers we have implemented work optimal when the load is very low. Instead of sending a job to the fastest node, when all nodes are unloaded, most of the algorithms will choose a random node.

Another benchmark worth running is to see how the algorithms implemented scale to larger systems. The benchmarks we used are very synthetic, it would be useful to see how the system behaves under an actual application.

Appendix A

Bibliography

[1] Richard Stevens.
The Protocols (TCP/IP Illustrated, Volume 1).
Addison-Wesley Pub Co.
ISBN 0201633469.

[2] Emmanuel Cecchet.
Raidb: Redundant array of inexpensive databases.
URL http://c-jdbc.objectweb.org/.
Not known where it is published, if it is., 2003.

[3] Several.
Linux virtual server, 2003.
URL http://linuxvirtualserver.org/.
Linux Virtual Server is a highly scalable and highly available server built on a cluster of real servers, with the load balancer running on the Linux operating system.

[4] Several.
Emic application cluster for mysql, 2003.
URL http://www.emicnetworks.com/.

[5] M. Dahlin.
Interpreting stale load information.
IEEE Transactions on Parallel and Distributed Systems, 11(10):33–??, 2000.
URL http://citeseer.nj.nec.com/article/dahlin98interpreting.html.

[6] Vivek S. Pai et al.
Locality-aware request distribution in cluster-based network servers.
In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.
URL http://citeseer.nj.nec.com/article/pai98localityaware.html.

[7] M. Mitzenmacher.
How useful is old information?
IEEE Transactions on Parallel and Distributed Systems, 11(1):6–??, 2000.
URL http://citeseer.nj.nec.com/mitzenmacher98how.html.

[8] Henry Morgan et al.
Queueing theory.
unknown, 1999.
URL http://www.new-destiny.co.uk/andrew/past_work/queueing_theory/.

[9] L. Kleinrock.
Queueing systems, volume i: Theory.
unknown, 1975.

Appendix B

/proc/net/dev

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
    lo: 5959584    8704    0    0    0     0          0         0  5959584    8704    0    0    0     0       0          0
  eth0:427097759  406500 141438    0    1 141444          0         0 134724400  400849    0    0    0     0       0          0
  sit0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
```