Centralized vs. Decentralized: Allocation in Distributed Systems

Pedro Artigas and Michael Ferdman

Fri Dec 8, 2000

Abstract

This paper presents a performance comparison of centralized vs. decentralized work allocation schemes in distributed systems. The centralized approach to resource allocation has been widely accepted as an easy to implement and adequate approach to allocation of computationally intensive work in distributed systems. Decentralized allocation schemes have for the most part been considered not as efficient in making allocation decisions due to a lack of global awareness, however the lack of a central point of failure of this scheme is appealing for some applications.

In this paper we describe our implementation and experiences with the compact Java distributed system framework created for this experiment. We describe an implementation of a universal distributed allocator, capable of being used without modification with both allocation schemes. Using our framework as the test bench, we analyze the performance and scalability of the centralized and decentralized schemes on the same cluster of workstations, and draw conclusions regarding the benefits of implementing a decentralized system.

1 Introduction

An important aspect that may limit the scalability of a distributed system is the entity that controls the placement and distribution of executed tasks on the available servers. An approach where a well known centralized unit is responsible for delegating the work has the advantage of full knowledge of the current state of the system and may therefore make optimal decisions at any given time. On the other hand, a decentralized approach where multiple controlling units exist and are split amongst the clients, may be more scalable because there is no single contention point. A major concern of the decentralized approach is that sub-optimal placement decisions may be made due to the individual allocation units not having as much information regarding the state of the system as in the centralized approach. A centralized approach, therefore, has the advantage of full knowledge while a distributed approach may be more scalable if the local placement decisions are sufficiently close to optimal global ones.

1.1 System overview

To observe the difference between using decentralized and centralized schemes for resource allocation, we created a Java distributed system framework capable of being run in both centralized and decentralized modes. Our framework is centered around the concept of an allocator object. Clients in the system request remote execution of work though this object.

Using our framework, we are able to set up a decentralized distributed system, with each client machine using its own allocator; this allocator bases its decisions on the current conditions of the available servers and on the workload requested only by the local system. We are also able to observe the operation of centralized resource allocation by having a dedicated machine responsible for running the allocator. In the latter scenario, all client machines request work allocation from a single central allocator through the same interface as in the case of distributed allocators; the decisions of the centralized allocator are based on the global overview of the system, since the central allocator is aware of all of the requests made by all of the clients.

Setting up both of these systems and observing their performance gives us the ability to monitor the difference that centralized and decentralized allocation has on an application without changing any of the application or system implementation. We are able to run a controlled experiment and keep everything including the server and client hardware the same, allowing us to observe changes caused solely by the difference in the distributed system work allocation scheme.

1.2 Approach

To avoid bias in our evaluation, instead of creating our own remote execution software, we built our system on top of a standard and widely accepted remote execution package. A unit of work is denoted in our system by a single Java class. We used the Java RMI [4] system for remote class creation and method invocation. In our system, a client application requests the creation of a remote object, our Java API will contact an allocator and be instructed which server it should use to create the needed object. With the use of RMI, the Java API will create a remote object and return a reference to it to the client application. In this system, the application simply requests allocation of remote objects, unaware of the distributed nature of the allocation, allowing us to monitor the differences in allocation policies presented with the use of the centralized and decentralized approaches.

1.3 The rest of this paper

The rest of this paper describes the motivation, design and implementation, and the performance and scalability analysis of our system. In the next section we describe the motivation behind our work. In section 3 we describe the design considerations and give a high level description of our system. Section 4 gives the details of the Java framework and Client API of our system. Section 5 presents the scalability and load balancing results we obtained. Section 6 lists some past projects related to ours. In Section 7 we draw conclusions of our investigation, and finally section 8 presents some options for future work in this project.

2 Motivation

In the recent years a large number of distributed systems have been created and researched. Scalability beyond the realm of a single machine has become an actively sought and highly beneficial concept. With distributed systems being developed by various groups with only a few solid examples to build upon throughout history, the resulting systems are very diverse and use highly contrasting approaches to achieving their results.

The main difficulty in building a distributed system lies not in the design but in the implementation. Application programmers are simply not willing to put in the amounts of time necessary to design a distributed system framework, create, debug, test, and tune all of its components, and then proceed to writing their applications for this framework. The cost of making an application run under a distributed computing system is just too great for the average programmer, and therefore applications remain mostly intended to run on a single machine.

Currently, the immense benefits of distributed computing can be seen in the few active implementations. *distributed.net* [15] uses a gigantic distributed system to compute answers to crypto-system challenges and mathematically complex research. The SETI at Home[14] project uses a vast network of distributed computers[1] to search for extraterrestrial intelligence. Looking at the results of these projects and at the amounts of work they are able to compute through the use of distributed systems clearly points out that distributing highly computationally intensive applications using distributed systems is the wave of the future in computing.

Some systems, such as Abacus [12] and Emerald [11], may also employ object relocation, for moving workload from server to server in order to more properly balance load. Our framework does not support workload relocation. After allocation, an object instance remains on the machine originally designated for its execution for the complete lifetime of that particular instance. We believe that introducing migration into our system would have resulted in a deviation from our goals of comparing allocation policies. We also rely on a previous study of Java workloads [1] that identifies the lifetime of a Java object to be usually very small, consisting of no more than 3% of the lifetime of the programs that uses them. We believe that this property may hold for most applications written in Java, with object lifetimes being short enough that it is not clear that object relocation is necessary in order to achieve load balance amongst distributed servers.

Because of the number of projects dedicated to distributed systems which currently exist, it is obvious that work on distributed systems is necessary.

[1] The only requirement for being part of the pool of computers used in the SETI at Home project is willingness to give away unused CPU cycles and an Internet connection. This allows for a very high number of computers to participate in computation. The actual number of operations performed per second in the SETI at Home "System" is in the order of tens of TeraFLOPS per second. Recent SETI computation rates have been measured at 29 TeraFLOPS per second, more than twice the speed of the current fastest supercomputer, the ASCI White, capable of performing at 12 TeraFLOPS per second.

An important concept very often overlooked in the design of a distributed system is the work allocation scheme. While some distributed systems created today are designed to support multiple work allocator units, a large number of systems are made with a centralized decision maker. The reasoning for the centralized model is the claimed simplification of the design and implementation of the system. It is also argued that because a centralized allocator has global awareness of the system it is able to more properly divide up the total work amongst the worker servers.

Our intention for this project was to create a prototype system which shows that while a centralized allocator is able to make more optimal decisions regarding work placement in a distributed system, the scalability limitation of having a central bottleneck in a centralized allocator design warrants the creation of systems with decentralized allocators. We intended to show that while decentralized allocators may make sub-optimal placement decisions, even for minimal scalability (3 or 4 working servers) a decentralized allocator is likely to be a better performance solution than a centralized allocator. We also intended to show that the creation of a decentralized allocator need not be any more complex than a centralized one, and that the load balance on the server nodes that can be achieved using a decentralized allocator approach is comparable to that of the centralized model; with the major improvement in availability achieved by the removal of the single point of failure from which the centralized model suffers.

3 System design

Our distributed system framework consists of a specification for three types of system nodes. Provisions for any number of client and server nodes can be made in a final system. The system is broken up into client and server sides; client nodes are responsible for running the applications, server nodes are responsible for performing calculations requested by the clients, and allocator nodes are placed in the system to direct clients in selecting which server is to be used for performing remote calculation based on server node availability and system load.

3.1 Client nodes

Client nodes are responsible for any user interface and minor client side computation or data collection that an application requires. The bulk of the computationally intensive work is meant to be performed on the server nodes, in parallel, however the clients are responsible for dealing with the parallelism and synchronization of data, so client nodes are generally responsible for running the multi-threaded client applications. The model used for the remote execution in our system is basic RPC, with the methods of remote objects being invoked and returning values using RPC semantics. This allows for remote execution of procedures to be identical in semantics as local method invocation, and therefore contributes heavily to the ease of porting multi-threaded applications to distributed ones.

3.2 Allocator nodes

The allocator nodes are responsible for delegating which server nodes are selected for remote work. Upon startup, server nodes notify the allocator nodes in the system of their availability through a central node registrar interface. After an allocator is informed of the existence of a server node, the allocator begins to poll the server for its load, its ability to do useful work, on a regular basis.

3.2.1 Work allocation

Whenever a client needs to have a remote computation performed, it contacts an allocator assigned to that particular client, and requests for an allocation of a remote object. The allocator uses the information it has collected through its regular status polls to determine the server node most suited to perform the requested computation and informs the client about which server it should be using for that particular task. The client then instantiates the remote object on the server selected by the allocator, and uses it to perform work. Communication between the client and server nodes is direct and does not go through the allocator node for performance reasons.

3.2.2 Allocation policies

Allocator nodes are currently implemented to use one of two allocation policies to choose which server node should be selected to perform work. The two available models are probabilistic and deterministic allocation. In the deterministic allocation approach the allocator simply selects the server with the lowest load as of the last status poll[2]. The probabilistic

[2] The server with the least load is identified as the server that is able to perform the most computation in a predefined amount of time.

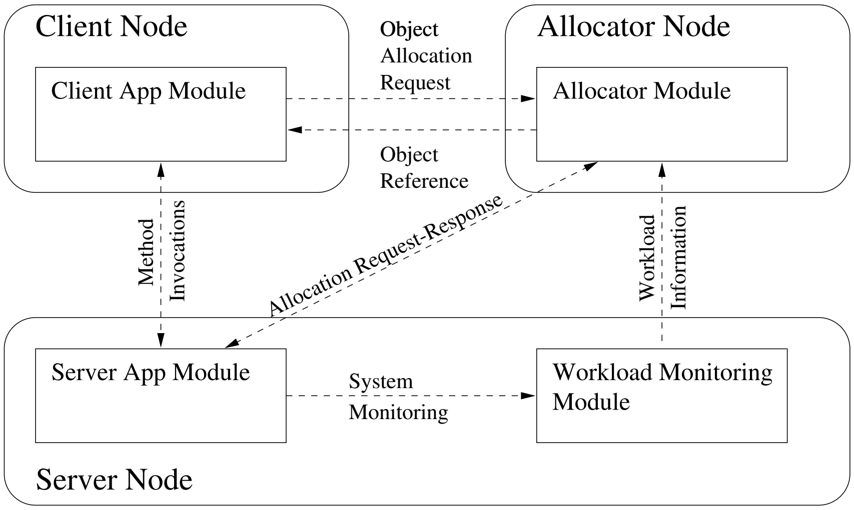

Figure 1: High level System Organization

approach picks a random machine from the server nodes, however the selection is not completely random, as the probability with which any individual server is selected is inversely proportional to the load of that server. The probabilistic model therefore has a much higher possibility of selecting a server with a lower load, however if multiple server nodes are present with roughly equal load, the selection amongst them will be random.

3.2.3 Reasoning behind allocation policies

The main motivation for the use of two distinct policies is the comparison of the allocation decisions they make when a different number of allocators are being utilized. If just one centralized allocator is being used, it is likely that this allocator will succeed in making globally optimal decisions. On the other hand if several allocators are being utilized it is very unlikely that, given the same amount of information, they will be able to make globally optimal decisions if they are invoked simultaneously, as they do not interact among themselves; therefore, a policy that tries to make sub-optimal but reasonable local choices is more likely to result in globally adequate, if not optimal, decisions[3].

[3]A more detailed description and comparison of the allocation policies is presented in section 5.1

3.3 Server nodes

The server nodes are responsible for two tasks. The nodes are first responsible for handling object allocation and method invocation requests from the clients. The server nodes are also required to measure and report to all of the available allocators the current load of the server node. The server nodes are implemented with minimum overhead, with remote object work handled by a thin Java RMI layer, and the load monitoring done via a low priority thread.

4 Implementation

The main software modules required in the system are the allocator, workload monitor, server-side application objects and client-side application objects. The first two modules are part of the framework. The other two components are written by an application developer and interface with the system using a well-defined API. This API is the interface between the code that implements the allocator and workload monitor modules and each client or server-side application objects.

The project implementation consists of four independent software components. The final system is made up of a resource allocator, a workload monitoring utility, a server node registration utility, and of various computationally intensive benchmark applications partitioned into client and server components.

4.1 Allocator

The allocator is implemented as a Java RMI [4] remote object, extending the UnicastRemoteObject[4] class. The allocator has two main tasks:

- Keep track of load on registered server nodes
- Answer client allocation requests

When started, an allocator waits for server nodes to inform it of their availability. Once the allocator is informed of a server's presence, it begins polling that server's load on a periodic basis. If multiple servers have been registered, polls of all of the servers are done on a periodic basis. The allocator continues this polling until client requests for object allocation are received. Once a request is received, the allocator uses the latest gathered statistics and using either the probabilistic or the deterministic scheme[5] picks which server will execute the request. The allocator informs the client of its decision, and proceeds to continue monitoring the server loads, expecting the client to allocate the needed object on the server that the allocator selected.

4.2 Node registrar

The node registrar is a remote object extending the UnicastRemoteObject class. The RMI name of this object will be known to all server nodes. As a server node will become available, it will call the `register` method on the registrar, notifying it of the server's availability. The node registrar will then distribute the notification of the particular node's availability to all of the allocator objects within the distributed system[6].

This architecture allows for all server nodes to be configured with a single central node registrar address. Upon invocation, all servers simply notify the node registrar of their existence, eliminating the need for updating a list of the system's allocators on every server used. This approach does not incur overhead on the actual performance of the system, since the registration process happens only once when a server becomes available to the system.

[4]UnicastRemoteObjects are non-replicated RMI server objects that are valid for the duration of the entire server process lifetime.

[5]The allocator policy is specified at allocator's invocation.

[6]The list of allocators present in the system are specified to the node registrar via a configuration file read at node registrar startup.

4.3 Workload Monitoring

The workload monitor is an object extending the UnicastRemoteObject class with a twofold function. The workload monitor's first task upon invocation will be to notify the node registrar of the availability of the server node, thus allowing for remote client applications to instantiate objects on the server node. The monitor's secondary task will be to service the allocators' requests for monitoring the availability of resources on its node. Whenever the `getLoad` method is remotely invoked by an allocator, the monitor object will return to the caller the current load count[7].

A major obstacle which we were able to overcome is the load monitoring on server nodes. We had originally intended to use the data from `/proc/cpuinfo` on Linux machines and use the idle cycle count change since the last monitoring probe to determine the processor load on the machines. What we realized in our early trials of the system is that the idle cycle count is a very poor estimate of the load, since when even a low priority process exists on a machine and is ready to run, there are no idle cycles present, so reading the idle cycle count can only report a boolean value, indicating whether idle cycles are available or not. As a fix for this problem, instead of reading an idle cycle count from `/proc/cpuinfo` we implemented a low priority thread which simply increments an integer every time it gets scheduled. The allocator then reads the change in this running counter and uses it as a measurement of machine load. This allows for a much more accurate measurement of resources since the number and priority of processes running on the server node are taken into consideration by this metric.

4.4 Client API

The client API was designed and written in such a way as to allow any multi-threaded java program to be easily converted into a distributed Java application. It was necessary to keep the API simple and straight forward, while at the same time giving enough functionality to provide the application programmer with enough resources to build a usable system.

There are two things an application programmer must do in order to convert a standard Java application into a distributed application in our framework.

[7]A running count is always returned, the caller is responsible for keeping track of the last two values and figuring out the actual load of the system.

First, the `RemoteAPI.startUp()` method must be called in the beginning of `main()`. Then the instantiation of objects that should be allocated on a remote server must be changed. To allow for an easy transition, the format of a remote allocation involves only changing the `new` line of code.

If the original instantiation of a local object read `SomeObject = new SomeObject`, in order to make that particular object be allocated remotely, the line should be changed to

```
Some Object = (SomeObject) \
   RemoteAPI.allocate("SomeObject").
```

The client source can at this point be compiled and will be operational. In order to be able to execute the application, local and remote stubs have to be generated for `SomeObject`. To do this, one must create the `Activatable` object implementation from `SomeObject` and create a Java class interface file. One must then run `rmic SomeObject.class` on the compiled byte-code of the remote object, and the object class file along with the generated stubs must be placed within an accessible path on all server nodes and registered with `rmid` running on those servers.

5 Results

Several results were collected using a simple distributed matrix multiplication application. The client application requests the allocation of several objects responsible for multiplying sub-matrices. The work performed is equivalent to the work that would be required in order to multiply large matrices whose size equals the square root of the number of unity tasks allocated times the size of the sub-matrices that are actually multiplied by each allocated object.

The application spawns multiple threads in each client, and each thread requests the allocation of an object that will perform a sub-matrix multiply. This design was used in order to ensure high level of concurrency that is expected in a distributed environment.

Each thread in each client allocates a single object, requesting that the object perform a sub-matrix multiply, and then discards the reference to that object. This process is repeated iteratively several times to simulate the multiplication of a large matrix.

The specific configurations used in each experiment are reported below. Parameters that were varied include the number of client nodes, the number of threads executing in each client, the architecture of the system (centralized or decentralized), the allocation policy used (probabilistic or deterministic), and the number of server nodes available in each run.

5.1 Load Balancing

In order to compare the ability of different allocator configurations to balance the load of the servers, the matrix multiplication application was executed using combinations of the possible architectures and the two available allocation policies. These experiments were performed using two clients and three servers; two clients were used to simulate a worst case scenario for a distributed allocator in search of local optimal decisions, more allocators would minimize the impact of infrequent poor allocation decision. The selection of the number of server nodes was arbitrary for this experiment.

The results were collected in separate runs on the same hardware. The four possible allocator configurations were tested (centralized deterministic, decentralized probabilistic, decentralized deterministic, and decentralized probabilistic). All other parameters remained constant, with 32 computation threads running on each client, each calculating the product of two 150x150 sub-matrices.

The following graphs show the load imbalance as a function of time; we note that data collection starts before the application actually started running so the imbalance presented in the beginning of each time series should be ignored. It is worth mention that the imbalance that occurs at the end of each run is due to the fact that some servers complete the final requests sooner than other servers. Since no new allocations are requested by clients, load balancing is not possible while some servers are still completing the final work requests. The interval of interest is that which occurs between these two events; this interval represents the imbalance that occurred during normal system operation.

The imbalance among servers is measured as the standard deviation among loads on each server. The thin lines represent instantaneous imbalance, the thicker lines show the moving average of the imbalance, considering 100 samples at a time.

It is clear that the probabilistic allocation policy makes sub-optimal allocation decisions in the centralized case; if the single centralized allocator has access to all server loads in the system and is aware of all client allocations in the entire system, adding a random factor into the placement decision only results in unnecessary random perturbations in the system. A centralized deterministic approach results in

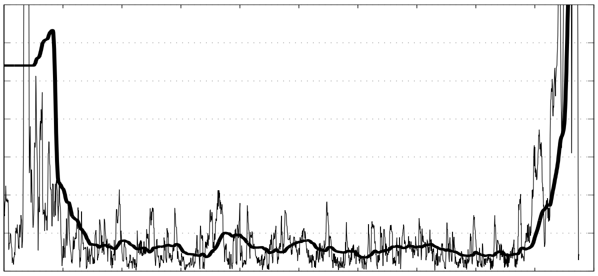

Figure 2: Centralized Allocator, Deterministic Allocation Policy. The deterministic policy offers very few peaks in load imbalance among servers in the centralized case. The overall imbalance running average is also noticeably low and stays almost constant throughout the entire run.

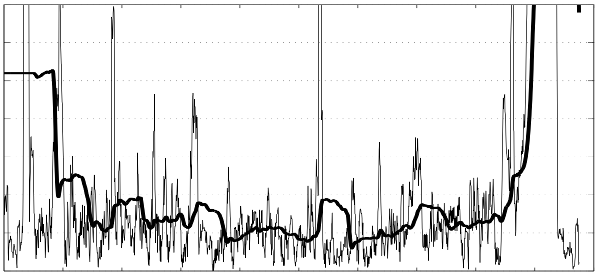

Figure 3: Centralized Allocator, Probabilistic Allocation Policy. The probabilistic policy in a centralized environment results in a large number of very tall peaks in the load distribution. These peaks are a result of the influence of random selection by a centralized allocator, with incorrect allocations often resulting in machines with very different loads.

placement decisions which take into account all of the system and use this data directly, resulting in a better placement strategy.

The graphs for the decentralized allocator traces show that while the deterministic policy resulted in the least amount of load imbalance in the centralized system, this is not the case with decentralized allocation. Allocations from different clients that are spaced closely to each other in time result in multiple clients simultaneously allocating work on the same server. This results in significantly higher load on the selected server, and causes imbalance in the distribution of load.

The probabilistic policy, while being clearly suboptimal, is more adequate. If two allocators make simultaneous decisions using the same load information, it is possible that the allocated objects will be placed in different servers. This contributes to reducing the average load imbalance, even though globally optimal server selections are not made. The centralized deterministic allocator is definitely the one that performs the best placement decisions, but in the decentralized system a probabilistic policy has better results.

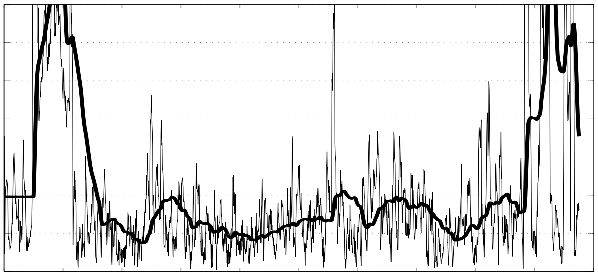

Figure 4: Decentralized Allocator, Deterministic Allocation Policy. The overall imbalance plot for a deterministic policy with the a set of decentralized allocators is filled with very tall and random peaks. These peaks can be attributed to multiple simultaneous allocations on the same server machine by different allocators, and are the cause of heavy and rough average imbalance of load.

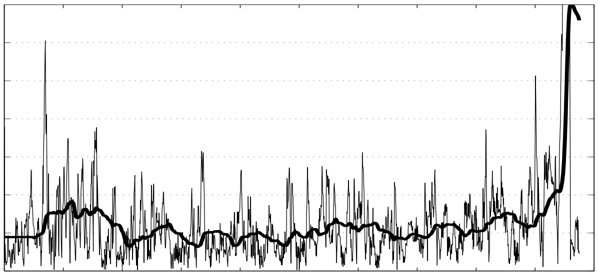

Figure 5: Decentralized Allocator, Probabilistic Allocation Policy. Although clearly not as good at balancing load as a centralized allocator running a deterministic policy, this approach results in a consistent average load balance with no major imbalance peaks; it is clear that the probabilistic policy is the better choice for a decentralized allocator system.

5.2 Scalability

The basic architecture of a distributed system has a significant impact on the system scalability. The following graphs represent the speedup of total time that the client applications needed to complete a set of multiplications. There experiments were performed using four client nodes and a varying number of server nodes. The allocation policies used in these experiments were chosen in accordance to the previous best results. Earlier experiments point out that a deterministic allocation policy is the best option in distributed systems with a centralized allocator. On the other hand, in the case of decentralized allocators, a probabilistic policy results in a more even average load on the servers.

We were interested in measuring the performance increase of client applications when the number of available servers is increased. Experiments were run using four client nodes and a varying number of server nodes. The decentralized system employs four allocators, one for each client. The centralized system also has four clients, but only one centralized allocator. The longest client running time for each run was used as the result of the experiments. Speedup results presented compare the performance increase of the two system architectures with N servers as compared to the same setups with only a single server[8].

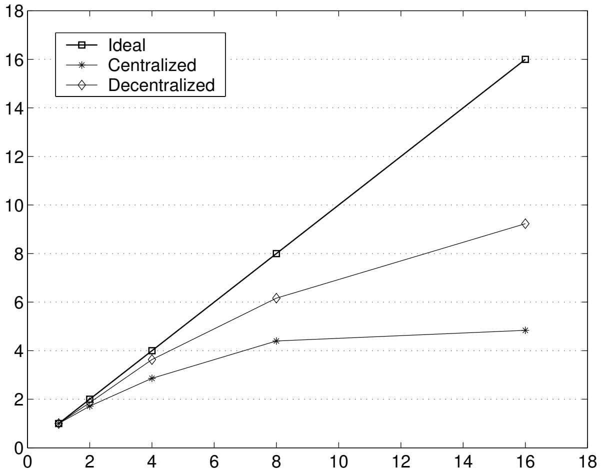

Figure 6: Speed Up. The performance scaling achieved by increasing the number of available servers compared to baseline performance of the system with a single server.

The decentralized organization is clearly much more scalable than the centralized organization, the efficiency[9] of the distributed organization when sixteen processors are used is above 50% while the efficiency of the centralized organization with the same number of servers is about 30%. It is clear that the decentralized organization is much more scalable. The difference is so pronounced that we are confident that the scalability difference among the two organizations is fundamental, not specific to the application used to exercise the system. Even the speed-up results when one to five servers are employed point the distributed approach as the more scalable.

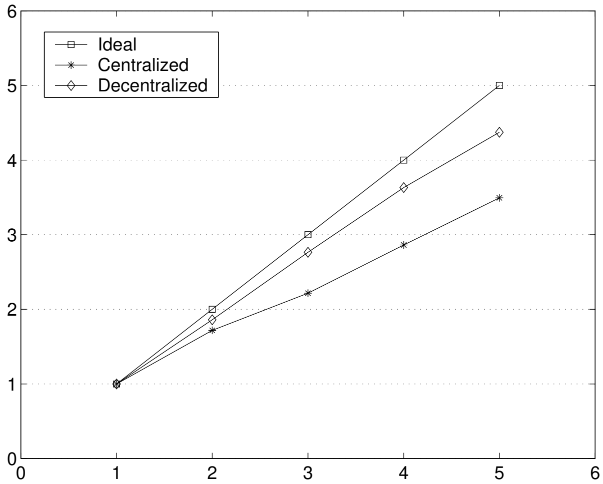

Figure 7: Speed Up, 1 to 5 servers

On the other hand the speed-up results ignore the fact that the baseline performance of the centralized approach is higher due to the more adequate load balancing of the centralized deterministic allocation policy. In order to compare the two organizations taking that fact into account we now present a graph that plots the rate of application completions per hour using each organization. The idealized results assume perfect scaling of the performance with a single server.

When single server performance is taken into account it is clear that the scalability of the decentralized approach leads to better results. The centralized organization shows better performance only when a single or up to two servers are employed, after this point there is no clear performance advantage. The performance advantage of the distributed approach with a large number of servers is pronounced. One factor that must be pointed out is that matrix multiplication has a very low communication to computation ratio, therefore the bandwidth of the network between the machines and the latency of the underlying communication infrastructure were not stressed. Other researchers [9] already pointed out that RMI

[8]Because speedup is shown here based on the base setup with only one server, actual running times are not reflected by these graphs.

[9]Efficiency is defined as the rate between the obtained speed up and the speed up in the ideal case.

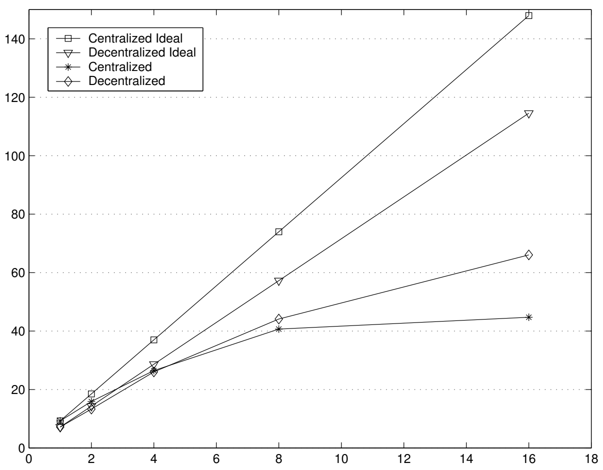

Figure 8: Rate of execution. Number of application runs that can complete in a single hour.

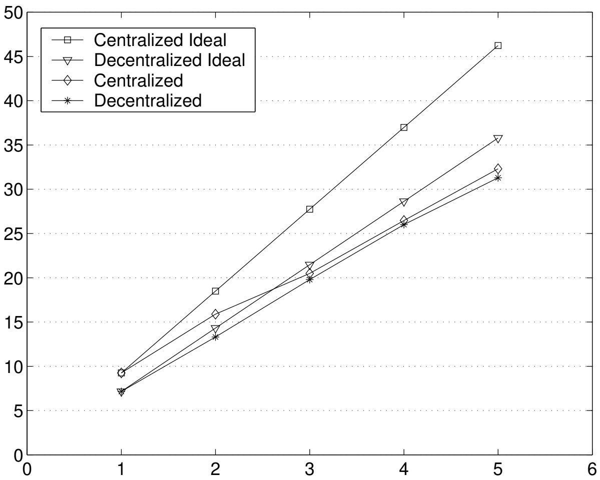

Figure 9: Rate of execution, 1 to 5 servers. Shown here to emphasize the rate of execution graph in the low server count range.

is only an adequate communications infrastructure for applications whose communication to computation ratio is low.

6 Related Work

Some distributed systems are built around the idea that the implementor is to have complete control of the work to be done on every node. Amongst such systems is the package for transparent communication for distributed objects done at the University of Pennsylvania [5]. The package is designed to allow a programmer to completely ignore the underlying communication between nodes running in a system and simply instantiate remote Java objects and use them as any other Java object that exists in the local system.

To achieve the creation of remote objects, the programmer must simply create a *base* object, giving it the IP and listening port of an available node. The programmer may then create objects on this base through a simple interface, with the returned objects executing on remote nodes appearing exactly the same as local ones to the rest of the application. This may be an adequate solution if the programmer has sufficient knowledge regarding the behavior of the system and this behavior is predictable at development time.

Other example of a system that relies entirely on the programmer for remote object placement decisions can be found in [9].

On the other hand, some implementations of distributed systems delegate the task of distributing objects amongst nodes completely to the underlying compiler and system core. For example, the Coign system [6] does not require any action on the part of the programmer to distribute objects between nodes. In fact, the system works on the basis that a completed final binary version of the program is actually distributed according to the internal COM interfaces used. The distributed core implementation is responsible for examining the history of running the particular application, noting the amount of communication and work done by individual COM components, and then deciding the correct COM object placement based on the given history. This approach completely eliminates all interaction by the programmer, taking any standard COM application and allowing the distributed execution of this application on a cluster of machines. Another example of a system that aims to distribute unmodified object oriented applications in a transparent fashion is the cJVM [10] Java Virtual Machine.

Another approach would be to require programmer assistance to partition the tasks, using for example new language keywords, and making placement decisions automatically. An example of such a hybrid system is presented in [7].

For our work, we selected a system which although helps the programmer by making placement decisions of objects, still allows the programmer to select which objects will be considered for placement. We feel this is a reasonable approach as it is the median of a large number of experimented distributed frameworks.

7 Conclusion

The proposed system was able to demonstrate several interesting aspects of distributed system design. Two object placement policies were presented, a deterministic policy that aims at optimal global placement decisions and a probabilistic policy that makes sub-optimal placement decisions but is able to perform well when simultaneous decisions have to be made from multiple sources. Each object placement policy showed it's strengths in different configurations, the deterministic policy was the better in a centralized environment while the probabilistic placement policy was able to prove it's value when multiple distributed allocators were employed.

By using the better policy for each environment, the system was able to point out the most scalable organization. The use of decentralized allocators clearly resulted in a more scalable system to the extent that the more efficient load balancing characteristics of the centralized approach were irrelevant when a large number of application servers were employed.

The system, as presented, is usable and efficient for applications with modest communication to computation ratio; the above 50% efficiency obtained when sixteen application servers were employed in conjunction with distributed allocators show that distributed system resources are being used efficiently, even when a single application is executed. If several applications were used concurrently, the efficiency should be even higher as objects from two different applications have no communication demand among themselves, lowering the system-wide average communication requirements.

8 Future Work

One aspect of the system that needs improvement is the underlying communications infrastructure; a more efficient communication infrastructure would allow the system to execute applications that have higher bandwidth and latency demands. Several researchers [9, 5] already proposed alternate communication protocols for the Java language, the development of a new protocol or the adoption of a more efficient protocol could allow the system to support a wider range of user applications.

Another system aspect whose cost has not been minimized in this experiment is the monitoring of load on application servers. Currently a unicast pull strategy is being used, that is, each allocator requests load information from each server when it sees fit. This results in a considerable amount of unicast redundant communications being performed. A push strategy coupled with an efficient multicast protocol could reduce the cost of load monitoring even further, other researchers already developed efficient low-cost multicast protocols [13] therefore a more efficient protocol for load monitoring could be adopted or developed.

References

[1] Yarsun Hsu, "Java Workload Characterization," *in 1st Annual Java Server Performance Workshop,* Hawthorne, NY, May 1999.

[2] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler, "Scalable, Distributed Data Structures for Internet Service Construction," *in the Fourth Symposium on Operating Systems Design and Implementation,* San Diego, CA, October 2000, http://www.cs.berkeley.edu/~gribble/papers/dds.pdf.

[3] Mor Harchol-Balter, Allen B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *in Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* Philadelphia, PA, May 1996, pages 13-24.

[4] Sun Microsystems, "Java Remote Method Invocation," *http://java.sun.com/products/jdk/rmi/*

[5] Michael Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore and Cristian Ungureanu, "Transparent Communication for Distributed Objecs in Java," *in ACM 1999 Java Grande Conference,* Palo Alto, CA, June 1999, http://www.cs.ucsb.edu/conferences/java99/papers/72-hicks.pdf.

[6] Galen C. Hunt and Michael L. Scott, "The Coign Automatic Distributed Partitioning System," *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999, pages 187-200.

[7] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann and Ronald Veldema, "Wide-area Parallel Computing in Java," *in ACM 1999 Java Grande Conference,* Palo Alto, CA, June 1999, http://www.cs.ucsb.edu/conferences/java99/papers/12-nieuwpoort.pdf.

[8] Hiromitsu Takagi, Satoshi Matsuoka, Hidemoto Nakada, Satoshi Sekiguchi, Mitsuhisa Satoh and Umpei Nagashima, "Ninflet: a Migratable Parallel Objects Framework using Java," *in ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, 1998, http://www.cs.ucsb.edu/conferences/java98/papers/ninflet.pdf.

[9] Denis Caromel and Julien Vayssiere, "A Java Framework for Seamless Sequential, Multithreaded, and Distributed Programming," *in ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, 1998, pages 141-150.

[10] Y. Aridor, M. Factor, A. Teperman, T. Eilam and A. Schuster, "A high performance cluster jvm presenting a pure single system image," *in Proceedings of the ACM 2000 JavaGrande Conference,* San Francisco, CA, June 2000.

[11] E. Jul, H. Levy, N. Hutchinson and A. Black, "Fine-Grained Mobility in the Emerald System," *in ACM Operating Systems Review,* SIGOPS, 1987, volume 21, pages 105-106.

[12] K. Amiri, D. Petrou, G. Ganger and G. Gibson, "Dynamic function placement for data-intensive cluster computing," *In Proceedings of the USENIX 2000 Annual Technical Conference,* San Diego, CA, June 2000, http://www.cs.cmu.edu/~dpetrou/research.html.

[13] David Cheriton, "The V Distributed System," *in Communications of the ACM,* March 1988, volume 31, no. 3, pages 314-333.

[14] The SETI @ Home Project, http://setiathome.ssl.berkeley.edu/

[15] Distributed Computing Technologies Inc. http://www.distributed.net/