Technical Report DHPC-049

Inter Server Transport of Java Byte Code in a Metacomputing Environment

J.A.Mathew, A.J.Silis and K.A.Hawick
Department of Computer Science, University of Adelaide, SA 5005, Australia
Tel +61 08 8303 4728, Fax +61 08 8303 4366,
Email {jm,din,khawick}@cs.adelaide.edu.au

30 July 1998

Abstract

In a distributed metacomputing environment, facilities for the movement of code from one server node to another are highly desirable. Often movement of code to data is preferable to the movement of data to code, particularly where the data is large. In this paper we describe our 'Code Server' or database of Java byte code. We discuss design and implementation issues and results of performance benchmarking we have undertaken as well as performance for a practical image processing example. We also discuss how our implementation addresses issues such as name spaces, client-server communication and distribution. Our results show that code serving appears viable, but more work remains to identify a suitable general database schema for describing codes.

1 Introduction

DISCWorld [6] is a metacomputing environment which allows collaborating users to share resources through the provision of well defined and high granularity services. DISCWorld is implemented as a software daemon that can be run on each participating host or group of hosts, that offers services to users and other systems. In this paper we describe our efforts to provide a mechanism for dealing with the movement of code between DISCWorld nodes.

Within DISCWorld, and other similar metacomputing environments, facilities to support the exchange of code are highly desirable. With the advent of Java [4] it is now possible to have a potable set of codes that operate on different hosts in a heterogeneous environment.

There are a number of reasons why the transport of code is desirable. One reason is that often

the data required for an operation, or group of operations, is much greater in size than the code to perform the operations. Hence, code movement can help to reduce total data transfer required for an operation. We also want to provide a framework which enables code to be moved to execute on systems with the most appropriate computational resources. Another reason is that a metacomputing framework such as DISCWorld requires some way of moving code for bootstrapping services since each node may not necessarily have code for all operations when it is initially started. Thus will also enable new services to be added more easily.

The code that we are concerned with in our work is mainly Java byte code. However it should be possible to extend our framework to support the storage of native code, which may provide substantial performance improvements in some cases. We may also support the storage of data such as images or serialised Java objects which will allow us to persistently maintain the state of each node, allowing nodes that fail to be restarted in a consistent state and hence improving the robustness and reliability of the system.

Our solution to the problem of moving code is to use what we call a 'Code Server' - a database of code. In addition to the actual byte code we also store contextual information (metadata) relating to the code. This allows us to provide facilities for searching in order to identify code that is suitable for some specific task according to some criteria.

This approach differs from that of having autonomous mobile agents [13], as in our system the code is moved in response to request from a remote client which requires the code to perform a particular operation, rather than the alternative of an object autonomously deciding to move. It also differs from the approach of having a server offer the service as a remote method, as this requires data for the operation to be transferred to the server offering the service.

The integrity of the code is also of concern, since we want to be able to authenticate the sender and ensure the code has not been tampered with. The problem of securely transmitting and storing portable code objects is an important one for scalable, wide-area distributed computing environments. We have previously described how authentication might be incorporated into the Code Server framework [5].

The Code Server supports distribution transparently in that each user need only to know about one DISCWorld (and hence Code Server) node. If the user makes a request that can't be fulfilled by the node, the node itself will forward the request to other nodes, obtain the result and return it to the user. We discuss distribution issues in more detail in section 2.4.

We have tested our Code Server with some benchmarks and some practical image manipulation operations. A number of simple image operators are implemented in portable Java byte codes and are available as portable code objects in the distributed DISCWorld database. Images processing operations are a good example of class of operations in which the code is likely to be much smaller than data, particularily for satellite imagery.

2 Architecture

The Code Server that we have implemented is based on a client-server architecture. The system stores code as well as 'metadata' or 'auxilary data' which is contextual data describing the code. We have chosen to store the metadata in a relational database, while the actual Java byte code is stored on a conventional file system.

The client and server are implemented in Java and are portable across platforms. The server communicates with a database using the Java Database Connectivity (JDBC) package, and makes use of pure Java database drivers. The client and server communicate using Java Remote Method Invocation (RMI). We have also implemented a version of the Code Server that uses sockets instead of RMI in order to compare the performance of the two approaches. The design we have adopted allows both the client and server to be installed on new platforms quite easily. We have successfully tested the Code Server on a number of platforms including Digital Unix, Linux (i386), Windows NT, Solaris (SPARC) and Solaris (i386). Databases that we have successfully tested with the Code Server include PostgreSQL [12] under Solaris (SPARC), Digital Unix and Linux as well as Informix [8] Dynamic Server under Solaris.

Applications seeking code for specific services will interact with a particular instance of the daemon, which has a single database associated with it. Our current model has a single Code Server for each database, with Code Servers having the ability to communicate with each other. This approach was chosen as we intend to have a DISCWorld daemon running on each participating node and controlling all the resources and services for that node. However, the database does not necessarily have to reside on the same host as the server, and only trivial modifications to the existing system are required to allow a single Code Server to deal with multiple databases.

If a server receives a request that it can not satisfy, it can forward the request on to other servers, obtain the relevant code fragment, and send it back to the requesting application. This

distribution is implemented transparently, so that the users or applications that want to use the code servers only need to deal with a single host, and if the host is unable to satisfy the request it will forward it to other servers, obtain the result and send it back to the requesting application or user.

2.1 Metadata

A problem that needs to be addressed in developing our Code Server is how to identify the code fragment of interest based on some sort of request from a client. We achieve this by providing search functionality for the metadata. A relational database provides a convenient means of searching through the data and locating items of interest. We have previosuly discussed querying auxillary data in the DISCWorld environment in [10]. Our initial list of metadata attributes is provided in table 1. When code is entered into the system, the metadata must be provided along with the byte code. The Class Name is mandatory, while the other fields are optional.

The attributes described in the table are a trial set of attributes we have chosen. In the future we will intend to transform our metadata to conform to standards such as Dublin Core [3]. From our experiences in actually using the system, we will use feedback from developers to further refine the metadata list. We are also intending to introduce extra metadata to store feedback on the performance of the code on various platforms and usage patterns.

Our Code Server Implementation uses a class 'ClassMetaData' to describe Java classes stored in the system. All of the fields described are represented by Java strings. The role of this class for querying is described in section 2.2.

As previously mentioned, the Code Server daemons communicate with the databases using the JDBC. This provides significant advantages over using a proprietary solution, the most important of which is portability. The particular database product and version used by each site is unimportant as long as an appropriate JDBC driver is available. Substituting one database for another can be achieved by simply creating the appropriate tables using an SQL script, and referring the server to the appropriate JDBC driver and host.

A design decision that arises is whether the database should hold Java byte code as well as the metadata, or whether it is better to store the byte code on the file system. In our initial implementation, we chose the later alternative since our preliminary tests have shown that there

Metadata Item	Description
Unique ID	A unique key value generated by Java
Group ID	A unique key value to group a set of related classes as chosen by the developer
Class name	The fully qualified class name (including package)
Code Description	A free text description of the code
Category	Each code fragment is assigned to one of a set of predefined categories (domains)
Keywords	One or more keywords or phrases describing the code functionality
Input Parameters	Input Parameters of the code and their types represented as strings
Return Type	Return type of the code and it's type
Dependencies	Other code modules required to execute this code
URL	Pointer to the developers of the code
Address	Address of developers
Citation	Any published material describing the code
Vendor, Product, Version, Development Platform	Since the various Java implementations have subtle differences, it may be important to have some knowledge of development platform and software
Platform	If the code includes native methods, this specifies what platform(s) it can execute on
JDK version	Version of the Java Development Kit used to compile the code

Table 1: Byte Code Metadata

is a significant performance overhead associated with storing large binary objects in a database, and hence no real advantage in doing so. In the future the relative advantages and disadvantages of storing the byte code within the database will be investigated more thoroughly. Each class is assigned a unique ID when entered into the system, and this ID is used for the filename to store the byte code on the file system. Java has a mechanism to generate unique IDs as part of the distributed garbage collector class. We use this VMID() method of java.rmi.dgc to generate the unique IDs. The ID generated is based on the IP address and current time and is unique assuming the IP address is fixed.

2.2 Querying

The Code Server can handle two basic forms of query. The first case is where the class name is known and the corresponding byte code is required. The second is where the class name is unknown and the appropriate code needs to be identified based on a query. In the later case, the query method is called with a ClassMetaData class as its input parameter. The class is populated with data to be used for matching. In addition, the user specifies a weight to be assigned to each attribute in order to determine how 'good' a match is. The classes are returned sorted in order of how good the match is. An example is provided below in table 2.

Attribute	Code Attribute	Query Attribute	Query Weight
Filename	CloudCover.class	Cloud*	10
Category	Image	Image	20
Keywords	GMS5	GMS5	15
Total	-	-	45

Table 2: Query Matching

One of the metadata attributes that we have defined is a group. The group is a mechanism for identifying related files and serves two purposes. Firstly, if classes are known to refer to each other, it may be preferable for the client to download all such classes at once, instead of downloading them one at a time. This is particularly useful in the case of small files, where the latency for each RMI call can be a very significant overhead, and the extra overhead of downloading code that may not be required is minimal. This is similar to the approach of using Java 'jar' [9] file archives, except that we are not using any compression. Compression may be

incorporated into the system at a later time. The performance advantages of this approach are investigated in detail in section 3.

The groups also allow for more effective management of the class name space. The class name that is stored in the system is the fully qualified class name including package name. We can deal with name clashes by storing the classes in different groups. When a request is made for a particular filename a group must also be specified if the filename is not unique. A group can contain one or more packages and a package can be spread across groups.

When code is entered into the system, a group can be assigned by the developer. If a group is not specified a new group is created, with the code being entered as the only member of the new group.

In our current implementation, each class can only belong to one group, and the groups are not hierarchical. In the future, there may be advantages in supporting this, since this could provide finer grained control over which classes are loaded and allow for performance to be optimised.

A number of querying functions have been provided. Some methods return a single class, others return the metadata for classes that match a query while others return a group of byte classes. In all the methods that accept the filename as an input parameter, the group must be specified if the filename is not unique.

2.3 Communication

Since both client and server are implemented in Java, RMI provides a convenient communications mechanism. An alternative to RMI is CORBA [11], [1], however in our experience CORBA is more difficult to work with, since it is a more 'heavy weight' interface and implementations of CORBA are not as portable, robust and inter-operable as Java RMI. CORBA has two main advantages over RMI, which are the ability to incorporate non Java objects and the extra functionality it provides such as traders [2]. These are of no benefit to us since our system is pure Java and the DISCWorld daemon itself provides much of the extra functionality that CORBA offers over RMI.

We have also implemented a sockets based version of the Code Server and comparisons between the performance of the RMI and sockets based version can be found in section 3. The sockets based version is a simplified version, and the server listens on a defined port. This was developed

only for a performance comparison with the RMI implementation, and only supports a subset of the full Code Server functionality. When a connection is made the server attempts to read the filename of the class being requested, searches the database to find the appropriate ID, loads the file and writes it to the socket.

2.4 Distribution

Our Code Server was designed so as to provide support for distribution transparent to the user. We have implemented distribution by making each Code Server a clients of the others. When a server is initialised it binds to other Code Servers that it knows about in it's database. This database needs to be bootstrapped by the system administrator of the DISCWorld node. When a request is made for code that is unavailable, a server can request the code from other code servers, and return the result to the client. Thus each client need only know about and interact with one server. Our initial implementation allows interaction across two servers, but this can be easily extended to further servers. If each request to another server is spawned off as a separate thread, the extra latency introduced will be largely independent of the number of servers. We have to be careful not to generate cycles for requests that cannot be fulfilled. This is avoided by tagging each request with a unique ID, and if a server receives the same request ID more than once, it will not generate further requests to other Servers for the code.

The approach of each Code Server knowing about all other Code Servers is obviously unscalable. In the DISCWorld framework, we introduce the concept of 'gossip' between hosts, where each Server knows about some number of servers which in turn know about other servers. This is not hierarchical so there is no single point of failure. Work into how many hosts each host needs to know about in order to maintain connectivity between hosts and what patterns of connectivity are necessary are still ongoing

Another potential mechanism for efficient distribution in a large system is to propagate the metadata between servers. Hence each server will know what code exists on other servers, and can hence can request code from a server that is known to store it, instead of requesting code from all servers that it knows about. If this approach is combined with the gossip idea, it is likely that the code can be transported efficiently and effectively.

2.5 Client and Class Loader

The client operates by making use of a network class loader. This first attempts to find the class locally by searching the class path. If this fails, it makes an RMI call to obtain a byte array with the class data and then makes use of Classloader methods defineClass() & resolveClass() to actually instantiate the class. Each class is stored in a hashtable when downloaded so that the class does not have to be downloaded again if it is reused. The client can make use of Java Reflection to determine what constructors, method names and class variables are available for a particular class. The client can either download a single class at a time, or choose to download an entire group at a time in order to minimise the number of RMI calls. We have also implemented a client that saves the byte code to the local file system and then loads it like a 'normal' class in order to determine whether there is any overhead introduced by the network class loader. By default Java will attempt to verify the integrity of any code loaded via a network class loader, but not code loaded directly from the file system. Our performance measurements attempt to measure both approaches. We have previously investigated the throughput of the Code Server with authentication enabled [5]. In the DISCWorld framework, we assume that other DISCWorld nodes are trustworthy and our security framework only needs to verify that that the code does originate from where it is claimed to. Hence, digitally signing the code is a good way of achieving this, and in this case it is quite reasonable to disable verification of code by the Java Virtual Machine (JVM).

3 Performance

We have performed extensive testing of our Code Server to investigate the performance of the system. Our tests were intended to investigate a number of major issues. These include the portability of server and client across platforms, and influence of the JVM (version and platform) on performance. We also investigated the overhead introduced by RMI both in terms of initial binding and latency for each method call. Other issues include the influence of network factors, such as latency and bandwidth and the choice of database product on performance. We were not attempting to benchmark the performance of databases for complex queries, so all test queries simply involved searching by filename.

Our initial experiments made use of different versions of the JDK, and we found that most

implementations of RMI prior to JDK 1.1.6 are defective. These early versions had memory leaks and problems with distributed garbage collection that cause them to crash after a period of time. We have not observed this behaviour when using JDK 1.1.6.

The measurements presented here are based on three series of tests. The first involves measuring the performance of a set of chained classes (classes that call each other). A set of ten classes was constructed, each of which had a method that simply instantiated the next class, and called the appropriate method on the this class. This test was designed to allow us to investigate the latency of loading each class, and the trade offs associated with loading a group at a time. A second set of tests was devised to investigate the performance as a function of code size. The final test compares the performance of an image processing example in the cases where we move the code to the data (using the Code Server) and the alternative of moving data to code.

All tests were performed on relatively unloaded networks and hosts (in the early hours of the morning) and for at least twenty iterations. Any results that we believed to be anomalous were repeated to confirm their accuracy. Measurements were performed by calling the System.currentTimeMillis() method of Java.

3.1 RMI Binding Overhead

Measured binding times for various client and server combinations are provided in table 3.1. Cairngorm and Turquoise are Digital Alpha Stations 255/300 located at Adelaide, Bremerer is an SUN Ultra-1 located at Adelaide, while dhpc01 and dhpc02 are Digital Alpha Stations 500/266 MHz located at Canberra. It is important to note that the Canberra Alphas are significantly faster than those at Adelaide. Canberra is approximately 1100 km from Adelaide, so the light speed limited latency for a round trip is about 7.5 ms. Using the Unix 'ping' command, the measured latency is about 16 ms. The measured binding times are about two orders of magnitude greater than this, and so the network latency appears to be a relatively minor component in the binding time. Issues such as the the hardware configuration of the machine and Java Virtual Machine type appear to be more important. From these figures we can conclude that the binding process, where an RMI client obtains a reference to a remote server object, is extremely inefficient. For the sockets based implementation, the initialisation time is of the order of a few hundred milliseconds: an order of magnitude less than for RMI. In the subsequent results, we do not include the binding times, and measure the time to obtain code from the Code Server

assuming a remote reference already exists.

Client	Server	Time (s)
Alpha 255/300 (Adelaide)	Same as client	2.6
Alpha 255/300 (Adelaide)	Another Alpha 255/300 (Adelaide)	3.3
Ultra-1 (Adelaide)	Same as client (Adelaide)	4.0
Alpha 255/300 (Adelaide)	Alpha 500/266 (Canberra)	2.7
Alpha 255/300 (Canberra)	Same as client	1.2
Alpha 500/266 (Canberra)	Another Alpha 500/266 (Canberra)	1.7
Alpha 500/266 (Canberra)	Alpha 255/300 (Adelaide)	1.5

Table 3: RMI binding times

3.2 Influence of Class Size

For this test a set of classes of varying size were created. Each class had a filler method, which had a variable number of 'println' statements to pad the size. The classes also had a generic 'Hello World' method, which was the method actually called by the test client. For each client/server/database combination the following tests were performed. The results with the key 'verification' had verification by the Java interpreter enabled. Those labelled 'no verification' had the JVM verifier disabled. Those marked 'group' were performed by loading an entire group at a time and also had the verifier disabled. The 'socket' results used the version of the Code Server that used sockets instead of RMI. Finally, the results labelled 'file' were those for which the code was saved to the file system and then loaded as 'normal' classes. Classes with sizes of between 1 kB and 200 kB were used in the tests. We found that the verifier could not verify classes with sizes greater than 50 kB on all platforms and JDK versions that we tested. We believe this to be a bug in the Java interpreter.

Timing measurements were made of both the time to bind to the RMI registry and get a reference to the remote object, and to obtain a class having previously obtained the reference. Measurements were made for Digital Alpha hosts located at both Adelaide and Canberra. The hosts at Adelaide were Alpha Station 255/300s while those at Canberra were Alpha Station 500/266s. All hosts were equipped with 128 MB RAM and JDK 1.1.6 revision 2. Hosts at

Adelaide were connected via a 10 Mb/s LAN while those at Canberra were connected by OC-3 ATM rated at 155 Mb/s. The link between Adelaide and Canberra was using AARNET with typical available bandwidth of about 4 Mb/s.

First we consider the case were client and server reside on the same host. These results can be seen in figure 1. In this case there is obviously no effect attributable to network issues. A number of conclusions can be drawn from these results. Firstly, it is obvious that verification of the byte code is an expensive process. We believe that use of digital signatures is a better security mechanism in our system. It is also apparent that as class size increases the overhead associated with saving the byte code to the file system and reloading it becomes quite significant. Thus there are no unexpected overheads associated with instantiating a class from byte code in memory.

The performance of the sockets based version is very similar to the RMI version. This is somewhat surprising as given the large overheads of RMI for binding we might expect that there will also be significant overheads for each method call.

We also see that the overall performance of the Canberra Alphas is superior due to their faster processors.

An interesting point to note is that although the loading of the entire group (containing all of the test classes) is obviously a bad idea in this case, the overhead is surprisingly low. For example the overhead in the case of code size of 1 kB is a factor of four, despite loading 288 kB of code when only 1 kB was required. This is due to the large invocation overhead associated with making an RMI method call.

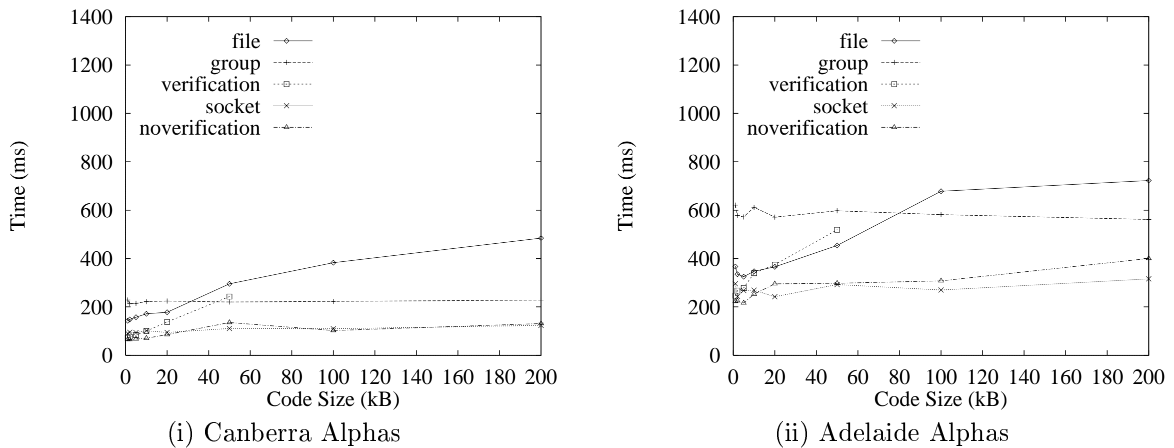
Now we consider performance across a LAN. These results are presented in figure 2. It is apparent that performance for small code sizes is quite variable. This is believed to be a TCP effect due to to Nagle's algorithm [14]. Nagle's algorithm restricts the number of outstanding small segments that have not been acknowledged to one for each TCP connection. Hence additional small segments will collected into a larger segment and sent when an acknowledgement for the oustanding segment is received. This algorithm is designed to reduce congestion in networks due to excessive numbers of small packets, but can also but can also increase the latency for sending a small message. The Digital Alphas computers used in our tests used a segment size 1500 bytes.

We believe that it is not possible to alter the TCP settings used by RMI, and hence Nagle's algorithm can not be disabled. The results for the Canberra LAN show that performance is

virtually identical to the case where client and server were located on the same host. This shows that there is virtually no overhead introduced by using the network in this case, due to the presence of the 155 Mb/s ATM. In the case of the Adelaide LAN, where the network was 10 Mb/s Ethernet, some overhead was introduced when using the network. We again see that verification introduces significant overhead. Performance for both RMI and sockets based versions is quite similar. As expected in the case where a group was loaded at a time was roughly constant. Due to the faster network in Canberra, the case where files are saved to the file system is more expensive, on a relative basis, than at Adelaide.

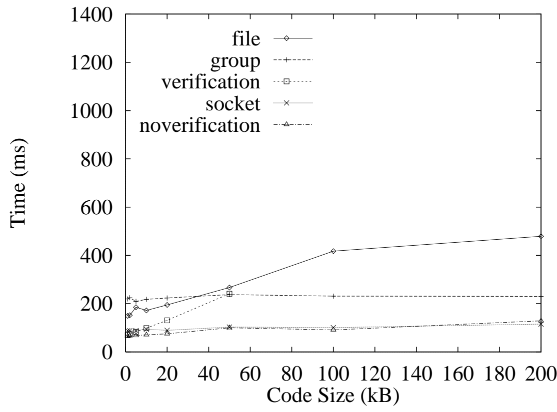
We also see that the trends are the same as for the case where client and server were on the same host, although actual measured values are higher.

Results of performance measurements between Adelaide and Canberra are presented in figure 3. These results are remarkably similar to the LAN measurements, given the much greater network latency and reduced bandwidth. The reduced bandwidth results in the performance of loading an entire group to be worse, although the worst case overhead is only a factor of six. As discussed in section 3.1, client initialisation is significantly quicker for the sockets based version. It may be be desirable to have both versions running simultaneously and allow each client to determine which alternative is better in a particular case.

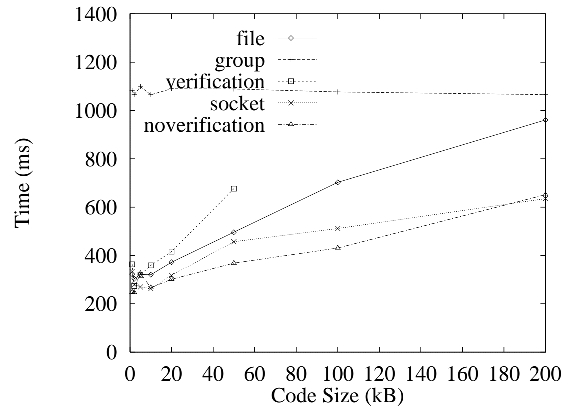


(i) Canberra Alphas (ii) Adelaide Alphas

Figure 1: Client and Server on the same host

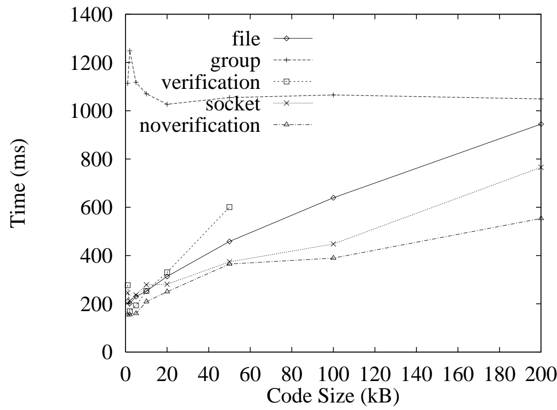


(i) Canberra Alphas

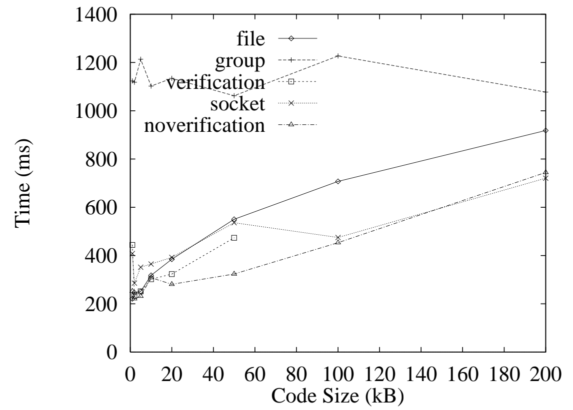


(ii) Adelaide Alphas

Figure 2: Local Area Network



(i) Adelaide to Canberra



(ii) Canberra to Adelaide

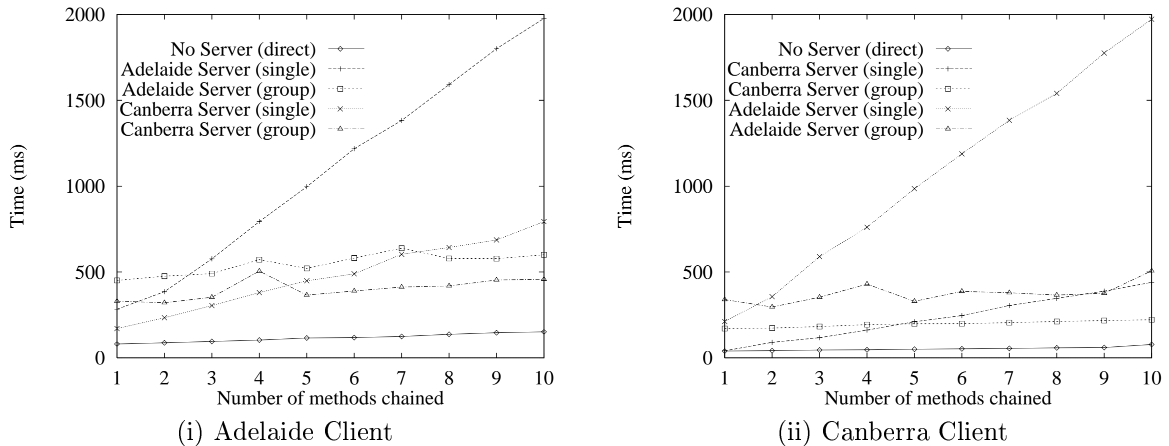
Figure 3: Wide Area Network

3.3 Class Chaining

This test measures the performance of a set of classes that reference each other. The results are presented in Figure 4. Results ending with '(single)' are for loading a single class at a time, those ending with '(group)' are for loading an entire group at a time, while those ending with '(direct)' are for tests where the client loaded the code directly from the local file system and did not make use of the Code Server. These last set of results are included to provide a baseline for the comparisons. As might be expected, in the case where classes are loaded one at a time, the time taken increases linearly with the number of classes that need to be loaded. In the case where a group is loaded at once, the execution time is roughly constant. In both cases, the point at which it becomes more efficient to load a group at a time is between 6 and 7 classes for the Adelaide LAN which is Ethernet based, while the corresponding trade off point is between 4 and 5 classes for the ATM LAN in Canberra. The difference is due to the lower latency and higher bandwidth of the ATM LAN. For the Wide Area tests between Adelaide and Canberra, the results show that the trade off point at which is is quicker to load a group at a time is between 1 and 2 classes. These figures will vary with class sizes and the number of classes in each group, but they give an indication that effective use of grouping can substantially improve performance. It is important to note that the results presented here represent a worst case scenario as each class does nothing other than call the next one. In a more realistic situation some processing will be performed so the overhead compared to the direct case will be less. An interesting point to note is that in general, better performance is obtained by an Adelaide client obtaining a group of classes from Canberra than locally. This is another example of how network issues are less important than Java performance which is determined by the CPU of the host and JVM type.

3.4 Distributed Performance

The performance of the Code Server, working in distributed mode was also investigated. In these tests, a client at Adelaide first attempted to obtain code from a Code Server at Adelaide. The code was not available locally, so the Code Server forwards the request to a Code Server located in Canberra. We present results for these test in figure 5. Results are presented for loading a single class at time without verification, and loading groups at a time, for the case where the code is available locally and the case where it is not. For small code sizes, the overhead introduced by the distribution is minimal, and it increases with code size. This shows that the



(i) Adelaide Client (ii) Canberra Client

Figure 4: Class chaining

bandwidth is the limiting factor, not the latency.

3.5 Database Influence

We also performed some tests to compare the influence of database type on performance. The results are presented for client and server on an UltraSparc-1 host with Informix and PostgreSQL databases in figure 6. It is evident that Informix offers superior performance, particularily in the case where classes are loading one at a time, resulting in many database accesses. This could be due to the PostgreSQL database driver creating a new connection for each database access while Informix creates a connection on initialisation and uses this for subsequent accesses.

3.6 Cloud Cover Calculations Example

In the initial prototype system that we are building are using imaging functions to investigate the Code Server performance for real world applications. We currently archive images from the GMS5 satellite, and we consider the particular example of calculation of percentage cloud cover for these images [7].

Every hour, we obtain 4 8 bit raster images (1 visual & 3 infrared) each of which is 2291 by 2291 pixels in size. Uncompressed, each of these images is over 5.2 MB in size, and even when compressed the images are still quite large. (The exact size varies significantly with time of day). Finding cloud involves looking for 'cold & bright' areas, and requires analysis of a visual image

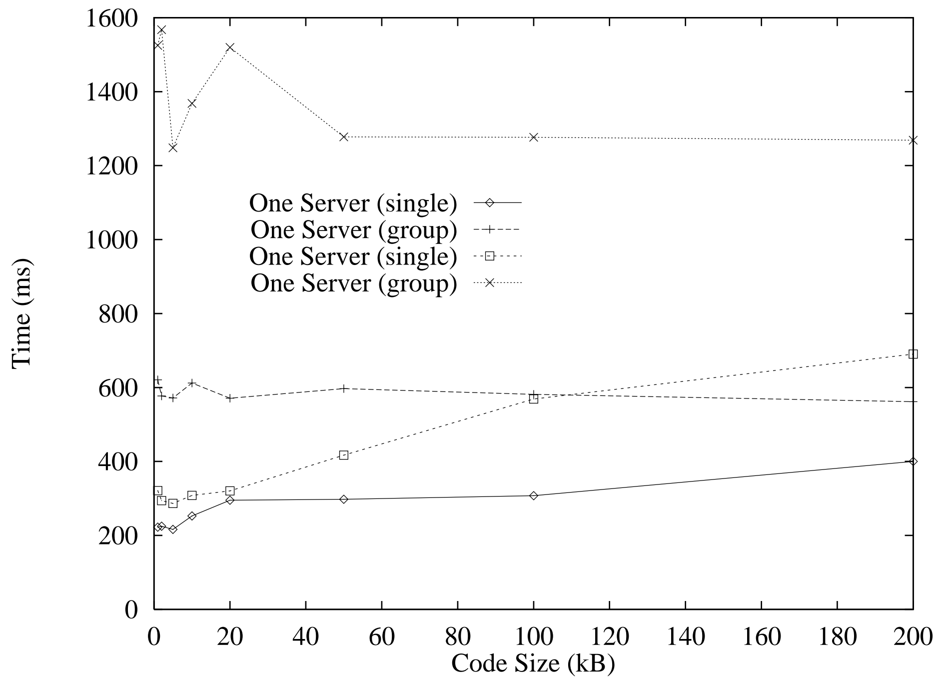


Figure 5: Distributed Code Server Performance

and one of the infra red bands. The final result required is a scalar percentage value. The Java byte code to perform the analysis is under 10 kB in size, while the data required over 10 MB uncompressed (for two images), and on average 4-5 MB when compressed using unix compress or gzip.

We performed tests with the client (the host with the image) data at Adelaide and server at Canberra and vice versa. The two cases we considered were firstly where the Code Server was used to download the code to the client site, and in the second case the file was transferred to the server using the Unix 'rcp' (remote copy) command and then the code was directly executed locally. As we expected, the results show that use of the Code Server provides a substantial improvement in performance. This improvement would be magnified in an environment with lower netowrk bandwidth availablity.

The results of the tests are provided in table 4 below. In this case the overhead introduced by the Code Server is minimal compared to the overhead of moving the data required by the operation.

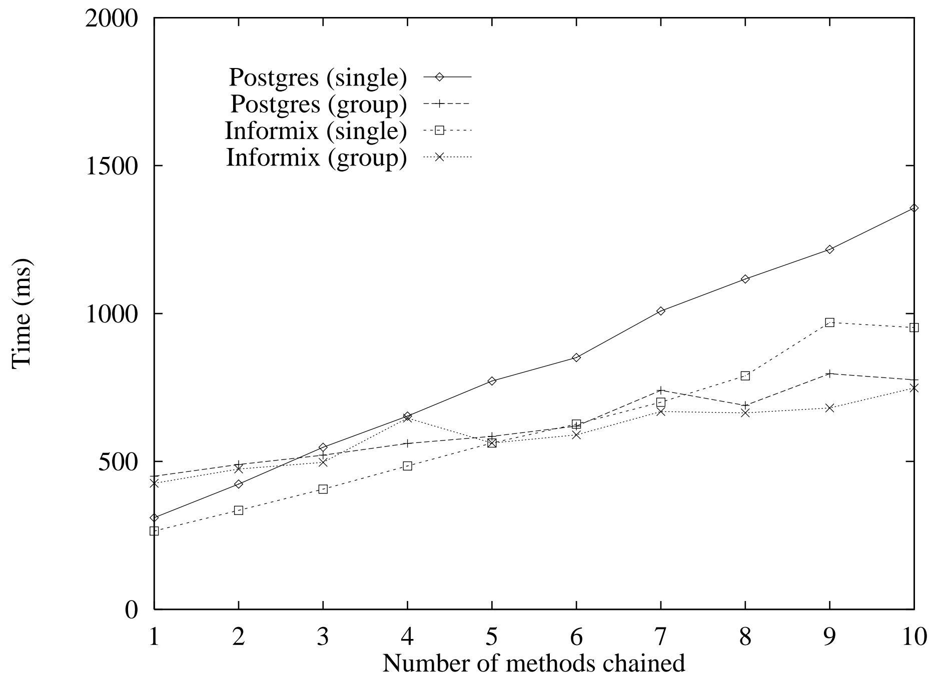


Figure 6: UltraSPARC Host:Comparing Informix and PostgreSQL

4 Conclusions and Future Work

We have described our Code Server and discussed how we addressed the various issues that arose during it's design and implementation. The performance results show the Code Server is a viable solution to the problem of movement of code. The results show that in a typical scenario where an operation requires a number of small class files, the Code Server introduces an overhead of a few hundred milliseconds for a typical system. We see that the overhead is mainly due to latency that can be attributed to the Java virtual machine and RMI and that network latency has a

Client	Server	Remote Copy time (s)	Execution time without Code Server (s)	Total time without Code Server (s)	Execution time with Code Server
Alpha 255/300 (Adelaide)	Alpha 500/266 (Canberra)	21	4.0	25.0	5.4
Alpha 500/266 (Canberra)	Alpha 255/300 (Adelaide)	21	3.6	24.6	4.0

Table 4: Percentage Cloud Cover Calculations

relatively minor effect in typical cases. This may change in the future as the efficiency of the available JVMs improves. Our results show that verification of the Java byte code by the JVM is expensive, and we suggest that digital signatures are a better alternative. They also reveal that appropriate use of groups in our system can provide a substantial boost in performance, as well as helping to deal with naming conflicts. In the Code Server system, we have found that although RMI introduces significant initialisation overheads compared to sockets, once a remote object reference is obtained the time to obtain each class or group of classes is comparable in both cases. The performance results show the Code Server is a viable solution to the problem of movement of code.

Future work on the system that we are planning include investigating issues that arise through extending the distribution framework, such as suitable protocols for finding a server with the appropriate code and for the sharing of information about code between servers and also suitable policy for determining what metadata and codes should be stored locally by each server. An enhancement that could improve the performance for large byte classes under some circustances is to compress the byte code being transmitted, and we would also like to investigate the performance trade offs associated with this approach.

References

[1] "CORBA/IIOP 2.0 Specification", http://www.omg.org/corba, July 1998.

[2] "Federating Traders: an ODP Adventure", Corba Services 2.0 Specification, Chapter 16. http://www.omg.org/corba, July 1998.

[3] "Dublin Core Metadata Element Set: Resource Page", http://purl.oclc.org/metadata/dublin_core/, July 1998

[4] "Java Language Specification", James Gosling, Bill Joy and Guy Steele", Pub. Addison Wesley, 1996.

[5] "Authenticated Transmission of Discoverable Portable Code Objects in a Distributed Computing Environment", D.A. Grove, A.J. Silis, J.A. Mathew and K.A.Hawick, To appear Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applica-

tions (PDPTA) 1998. Also available as DHPC Technical Report DHPC-027, April 1998. http://www.dhpc.adelaide.edu.au/reports/

[6] "DISCWorld: An Environment for Service-Based Metacomputing", K.A. Hawick, P.D. Coddington, D.A. Grove, J.F. Hercus, H.A. James, K.E. Kerry, J.A. Mathew, C.J. Patten, Andrew Silis, F.A. Vaughan, invited paper for a special issue of the Journal of Future Generation Systems, to be published. Also available as DHPC Technical Report DHPC-042, April 1998. http://www.dhpc.adelaide.edu.au/reports/

[7] "A Web-based Interface for On-Demand Processing of Satellite Imagery Archives", H.A. James and K.A. Hawick, Proc ACSC'98 Perth, Februray 1998. Also available as DHPC Technical Report DHPC-018, April 1998. http://www.dhpc.adelaide.edu.au/reports/

[8] "Informix WWW Site", http://www.informix.com, July 1998

[9] "JAR Guide", http://www.javasoft.com/products/jdk/1.1/docs/guide/jar/jarGuide.html, 1998

[10] "Querying and Auxiliary Data in the DISCWorld" J.A.Mathew, K.A.Hawick, Proc 5th IDEA Workshop, Freemantle, February 1998. Also available as DHPC Technical Report DHPC-030, January 1998. http://www.dhpc.adelaide.edu.au/reports/

[11] "The Essential CORBA: Systems Integration Using Distributed Object", Thomas J. Mowbray & Ron Zahavi, John Wiley & Sons, Inc., 1995.

[12] "PostgresSQL WWW Site", http://www.postgres.org, July 1998

[13] "The Artificial Life Route to Artificial Intelligence", Luc Steels, Rodney Brooks (Editors), Pub Lawrence Erlbaum Associates, 1995

[14] "TCP/IP Illustrated Volume 1", W. Richard Stevens, Addison-Wesley 1994.