

Data Futures in DISCWorld

H.A. James and K.A. Hawick

*Distributed & High Performance Computing Research Group
Department of Computer Science, University of Adelaide
SA 5005, Australia*

Email: {heath,khawick}@cs.adelaide.edu.au

Tel +61 8 8303 4519, Fax: +61 8 8303 4366

3 March 2000

Abstract

Data futures in a metacomputing system refer to data products that have not yet been created but which can be uniquely named and manipulated. We employ data flow mechanisms expressed as high level task graphs in our DISCWorld metacomputing system. Nodes in these task graphs can themselves be expanded into further task graphs or can be represented as futures in the processing schedule. We find this a generally useful approach to the dynamic and optimal execution of task graphs. In this paper we describe our DISCWorld Remote Access Mechanism for Futures (DRAMFs) and its implementation using Java technology. DRAMFs embody these ideas and allow data products to be lazily de-referenced and to be manipulated as first class objects in the object space of the metacomputing system. Our system aids in the provision against partial failure or network disruptions in a distributed system such as DISCWorld.

Keywords: data futures; task graphs; DISCWorld; Java; distributed systems; metacomputing.

1 Introduction

It is important to provide a flexible yet powerful mechanism to express task graphs in a metacomputing or problem solving environment. We have developed a rich data pointer mechanism in our DISCWorld [6,7] metacomputing system, and in this paper we present an extension to this mechanism for controlling data

futures. We show how this can be used to enable lazy and eager scheduling placement of sub components in a task graph in a metacomputing environment.

The remote data access problem is not readily implemented by existing mechanisms in network-oriented programming systems such as Java's networking package, Java RMI or RPC. In such systems, all results are returned to the user upon completion of their request. What is needed, and is not supported by either Java or RPC, is a method whereby the user is returned a pointer to a result that can be accessed directly if they wish to inspect the data that the pointer represents, or can be passed to another server for further processing. DISCWorld Remote Access Mechanisms (DRAMs) provide for this need. The problem of global addresses for pointers is tackled by the construction and use of canonical names for data. This is possible under the general DISCWorld framework. There is no concept of the pointer referring to physical address space on any particular machine in the distributed infrastructure; this enables DRAMs to be scalable, realisable and failure resistant. The differences between DRAMs and other systems' mechanisms for representing high-level data pointers are discussed in [5].

When a DRAM is used to construct a complex processing request, the object to which the DRAM refers is not actually used until it is demanded by the DISCWorld daemon. Most other technologies do not allow for on-demand or "lazy" processing; the value that the object references is evaluated when the reference is returned. To aid construction of processing requests, DRAMs contain *metadata* describing the objects to which they refer. The Java definition of a DISCWorld DRAM is shown in figure 1. The DRAM definition mandates sufficient metadata or "recipe" information that will allow reconstruction of the pointed-to object in case of a restart. The DRAM mechanism allows a remote node the option of transferring the object, to which a DRAM refers, to any other DISCWorld node.

Using a global naming mechanism to refer to DRAMs, we allow equivalence between DRAMs to be established. This naming scheme is independent of the DRAM mechanism and we assume its existence as does CORBA [12]. We can use a simple implementation of the unique naming scheme provided by either Java/RMI or by CORBA. These are usually based on "mangled" strings derived from host names or URLs. DRAMs from different nodes can have the *same* name, implying that they are copies of the same object, even if they were created on different DISCWorld servers.

There are two direct subtypes of DRAM: a pointer to data, DRAMD; and a pointer to metacomputing services, DRAMS. Since a DRAM is a Java object, it can have extra functionality programmed in for use at the client end. This functionality could be graphical semantics or data filter/compression services [8]. For example, data to be transferred could be explicitly compressed at the server and uncompressed at the client end by the DRAM framework. This would not be possible without the bytecode portability provided by Java.

```

public abstract class DRAM implements Serializable
{
    private String publicName;    // descriptive name for GUI use
    private String globalName;    // internal ID
    private Icon icon;           // associated icon eg thumbnail image
    private String description;    // long free textual description
    private String className;     // query-able search-able text
                                // representation of class
    private String remoteServer;  // location of the Data to which
                                // the DRAM points
    private int objectMobility;   // whether the object being pointed to
                                // can be transferred across
                                // the network
    private int objectSize;       // size of the object being pointed
                                // to, in bytes
}

```

Figure 1: DRAM Java base class. The DRAM provides a high-level pointer to an object. Metadata describing the object to which this DRAM refers is recorded to enable efficient scheduling and placement decisions in a wide-area, high-latency environment.

2 DRAM Futures

In a high-level distributed system, user requests are represented as process networks of metacomputing services linked together by the sharing of data. Such sharing of data may be due to the services' reliance on the same input data, or the sharing may be more explicit, as in a producer-consumer relationship. Such process networks may be specified and their services statically assigned to hosts in the distributed system. This approach can lead to optimal solutions in the case where the characteristics of the distributed system are known in advance, but if all characteristics are *not* known in advance, the dynamic allocation of services to hosts and the optimisation of execution schedules can produce the best suboptimal solutions [8].

To implement the optimisation of process networks, we have extended the DRAM concept to include the notion of pointers to data that have not yet been created. The extended DRAMs can be passed between clients and servers. We term these pointers to not-yet-created data DRAM Futures, or DRAMFs. DRAMFs are created with the name of the data that they *will* represent, and as such, are equivalent to DRAMs in the operations that can be performed on them [5]. DRAMFs are assigned an approximate size for the data they point to, based on the mean historical size of the previous data products of the creating service on that node. However, unlike DRAMs, DRAMFs are only able to point to data, not metacomputing services. This restriction is common sense, as it is

```

public class DRAMF extends DRAM implements Serializable {

    private int estimatedTimeAvailable; // estimation of when data will
                                        // be available from the time that
                                        // this DRAMF is dereferenced,
                                        // without any execution
                                        // optimisations
}

```

Figure 2: Future DRAM (DRAMF) Java class, which extends DRAM (see figure 1). The data product to which this refers will not be created until the DRAMF's contents are requested.

not sensible to have a pointer to a service which does not yet exist. When a DRAM is *inspected*, the data to which the object points is retrieved; in the case of a DRAMF, a request is sent to the server that is to create the data. When the data has been created, a DRAMD of the same name as the DRAMF is returned containing the data. Thus, the DRAMF is replaced with a DRAMD, which contains the exact size of the data item, instead of the DRAMF's original estimate.

Instance Variable	Value
<code>publicName</code>	Future for GMS5 1998122500:IR1 (0,0), (2291,2291) 100% zoom
<code>globalName</code>	ERIC:SingleImage:GMSImage:IR1(Integer:00,Integer:25,Integer:12,Integer:1998,Integer:0,Integer:2291,Integer:0,Integer:2291,Integer:100)
<code>icon</code>	<i>thumbnailed representation of data</i>
<code>description</code>	Future GMS5 image 00Hrs 25DEC1998 IR1 channel, Original Dimensions, 100% zoom
<code>className</code>	image.satellite.GMS5
<code>remoteServer</code>	cairngorm.cs.adelaide.edu.au:1965
<code>objectMobility</code>	2
<code>objectSize</code>	5248681
<code>estimatedTimeAvailable</code>	102

Figure 3: Example contents of a DRAMF expressed as name/value pairs. The `globalName` specifies the “recipe” by which the data can be reconstructed in the event that the `remoteServer` is temporarily unavailable and then later restored.

The implementation of DRAMFs extends the DRAM Java base class. This is shown in figure 2, and an example of a DRAMF's contents is shown in figure 3. For simplicity, the `estimatedTimeAvailable` field is used as an offset from the time that the holder of the DRAMF requests the data be created. This

value consists of the current waiting time for the host node start the service, as well as the mean run time of the service that will create the data. If the data represented by the DRAMF uses any other DRAMFs as inputs to its service, the `estimatedTimeAvailable` field also includes the estimated time of the DRAMF upon which this service needs to wait, and the estimated time to transfer the resulting data between DISCWorld daemons. We consider the time taken to estimate the data's arrival time to be small when compared with the time taken to transfer necessary data and perform the services that the DRAMF represents.

In principle, DRAM Futures are similar to programming-language level Futures [14], Wait-by-necessity [1] and Promises [10] mechanisms. Whereas Futures and Promises were designed to hide the latency in RPC-based systems, DRAMFs are intended to be used as a high-level pointer to data, in exactly the same way as DRAMDs and DRAMs. The main difference between DRAMFs and the other mechanisms is their granularity, and the operations which can be performed on them. The Futures and Promises mechanisms are fine-grained. Because the Futures actually represent a memory location in the requesting program, they are tied to the instance of that program. They are not able to be sent between programs. DRAMFs are higher granularity than the other systems, abstracting away from any instance of a creating program. Their ability to be referenced outside the scope a particular instance of the DISCWorld server allows them to be treated as first-class objects.

DRAMFs can be sent between servers, and may also be used by clients in the composition of new processing requests. What this means is that a DRAMF may be copied and moved to many other servers. Subsequently, it may be used as an input to a service by a scheduler, and only when that schedule is *executed*, will the data to which that DRAMF points be copied to a remote machine. Thus, by using the DRAM Futures mechanism, and more generally, the DRAM mechanism, unnecessary bulk data transfers are prevented.

User process networks are represented as networks of DRAMs. This may be done either explicitly, with the user constructing a graph of nodes (as discussed in section 3), or implicitly by the user selecting a pre-designed process network from an available library. When the user has created their process network, it is sent to a DISCWorld daemon for processing.

The normal behaviour of the placement mechanism in DISCWorld is to assume that the server will be willing to execute a service or network of services sent to it. If the server is *not* willing to execute the service, either because it is too heavily loaded or the user who owns the job is not allowed to execute services on this machine, or for some other reason, the onus of selecting a new daemon to host the service(s) is on the local daemon. Although beyond the scope of this paper, relocation of the service requires re-placement of the affected services to other daemons and transmitting the modified process network to those daemons which have been designated to produce data that the services were to consume. Nothing needs to be sent to those daemons that will consume the data that is produced by the services on this node, as they will receive a DRAMF from the

daemon that has accepted the execution request. The case where no daemon can be found to execute the affected service(s) has not been considered, nor has the case in which a number of daemons continually attempt to assign the services to each other in an infinite loop.

When a daemon creates DRAMFs to some future data, the service does not automatically begin executing. The agreement by a daemon to execute a service with given parameters is, however, binding. The daemon has acknowledged that if requested, it will perform the computation and return the result to any requesters. If the input data required by a service is to be created by a previous service on perhaps a remote machine, then the DRAMFs for the current service cannot be made until the input DRAMFs are received. They are needed in order to estimate the time at which the data, to which the DRAMFs point, should become available. It is in this way that a complex processing request may be set up. This allows, essentially, resource reservations to be made for daemons that will participate in the processing of a request. Thus, while a processing request may be expressed as a network of services connected by data transfer, culminating in the production of some desired data, its execution is constructed in reverse order. Unlike other models of resource reservation, multiple services can be reserved on a single DISCWorld node simultaneously.

In the DISCWorld model, the partial results of processing requests are deemed to be as significant as the final results. Therefore, in addition to sending the DRAMFs to the nodes that may use them, they are also sent to the daemon or client which submitted the processing request. In the case of the user client, DRAMFs corresponding to all the partial products are returned – if the user wishes to view the contents of the DRAMF, they can request the DRAM's contents. As a DRAMF represents the result of a remote computation that has not begun execution, when the results are requested the service is started, and the result is returned when available. When the DRAMF is inspected by the user client or any other daemon, a request is sent to the producing daemon, which initiates the computation.

Optimisations are allowed on the services used to satisfy a processing request. The server may be aware of one or more servers that already possess the data which is to be created, or have supplied DRAMFs to the data sought. If the data is available, and the time spent transferring it will not exceed the time spent waiting until the DRAMF can be satisfied and the data returned, then the decision may be made to transfer the data from an alternative source. By the same reasoning, if there exists another DRAMF pointing to a different data source, and the estimated time is lower than that returned, the alternative DRAMF may be dereferenced in the expectation that either the computation has already been requested by another holder of the DRAMF, or the data should be available sooner due to the lower estimated arrival time. The method by which daemons are made aware of DRAMs on different servers is beyond the scope of this paper [13].

Using partial or final data products of previous processing requests allows the

possibility of pruning the execution schedule for the current processing request. Thus, if there is no need to re-compute a data product, then the system will avoid it if possible. Of course, the situation may arise where a user has inspected a DRAMF corresponding to a final data product, causing the computation to be started, and at the same time inspecting a DRAMF to a partial data product. If the DRAMF that is an input to the service which is to produce the final data product has an estimated time greater than a cheaper source of the same output data, then the data may be retrieved from the alternate source, even though by inspecting the partial product DRAMF, the user has dramatically reduced the time it would take to retrieve the data. This case is not addressed in the current implementation – we simply make a best-estimate of the execution times.

In the prototype version of the daemon, there is no maximum time limit between when a DRAMF is returned from a server and the execution must be started. In future implementations, depending on the cost that the user is willing to incur, multiple requests may be sent out to competing data sources, in order to achieve the fastest possible turnaround time for the processing request.

Using DRAMFs to represent the data allows the system to judge whether it is more economically feasible to wait until a result that is to be computed at one node is available than retrieve the data from another node, where it already exists but may be expensive to access. The retrieval of data from another node may have one of two effects: either the data the node was to produce now becomes redundant and a waste of processing cycles, or the data is scheduled for use by a subsequent service (after the service has found the data from another source). If the result is no longer needed, depending on the node’s management policy, and usually based on the load of the node and its storage capacity, it may choose to proceed with the computation on the chance the data will be needed within the time frame that the node is willing to store results for, or may decide to cancel the production of the result. In turn, in the event of cancelling the production of the result can be propagated up through the list of servers scheduled to create the data that is no longer needed. Of course, the creation of data is only possible if there are no other servers that use the data to be created. Thus, the execution tree is pruned so that only the services that *create data* and those that create the data that is *required*, are actually executed.

The steps by which a user processing request is executed are shown in figure 4. Figure 4i) shows DRAM representations of the data and services being sent to the daemon local to that from which the client submits the request. For example, the DRAM corresponding to data D_1 on remote daemon R is named d_{d_1} and the DRAMS corresponding to service S on remote daemon R_1 is named d_{d_s} .

When a client C connects to the local DISCWorld daemon L a copy of each of the DRAMS is sent to C . Submission of a processing request Q from C to L causes a DRAMF, d_f , representing the output of Q to be returned to C . This is shown in figure 4ii). It can be seen from this figure that the submission of the

processing request has also caused a number of DRAMs to be copied around the system. Until the data (or program) represented by each of the DRAMs is needed, it is not transferred between daemons.

When the DRAMF is inspected, the DISCWorld framework sends a message from the current holder of the DRAMF (C) to the DRAMF's originating server (R), causing creation of the DRAMF's data to begin. Shown in figure 4 iii), DRAMF d_f represents the result of processing request Q , which is executed to completion. It is at this point in the execution that optimisation can be performed. The daemon attempts to achieve service execution in the quickest amount of time. If the data that is to be used as input is available from another node, or is already present on the local node, or the current node has a DRAMF for the same data, with a sooner estimate of availability, the daemon may decide to use the alternate data source. One of the assumptions of the DISCWorld model is that the size of a data item is the same no matter where it is created. Thus, if a daemon has a DRAMD *and* a DRAMF to the same data, the estimated size of the DRAMF can be updated to the (exact) size of the DRAMD.

When the data to which the DRAMF points is actually created, F , the DISCWorld system converts the local DRAMF d_f to a DRAMD. This is done because the data is actually available now and the estimation time information contained within the DRAMF is no longer relevant. Finally, the outstanding request for the DRAMF's contents is satisfied and the bulk data item F is transferred to the requester C . At C the DRAMF is converted to a DRAMD and the processing request is satisfied. This is shown in figure 4 iv).

3 Discussion

It is also convenient to use DRAMs as a mechanism for deferred delivery of computations. A DRAM may be delivered to a client program with an indication that the remote data to which it refers will only be available within certain times. This allows the token to be redeemed at some time in the future when a long computation has completed.

The concept of a deferred delivery mechanism is also useful where a user or client program receives a *token* to some data that may not yet be created or directly available. This allows the user or client to submit a request, and upon a subsequent login to the DISCWorld environment, receive the results. The token is used both as a placeholder for the request result, but also as proof of authorisation to request the status of the pending request. DRAMs are used to implement the deferred delivery mechanism of DISCWorld, and are the equivalent of data tokens. Common mechanisms for notifying users of deferred delivery products in other systems include the creation of output files in pre-defined directories, and the receipt of electronic mail bearing an authentication key to be used to gain access to the results at a server. We believe that while these methods are indeed efficient and perform well for their intended applications,

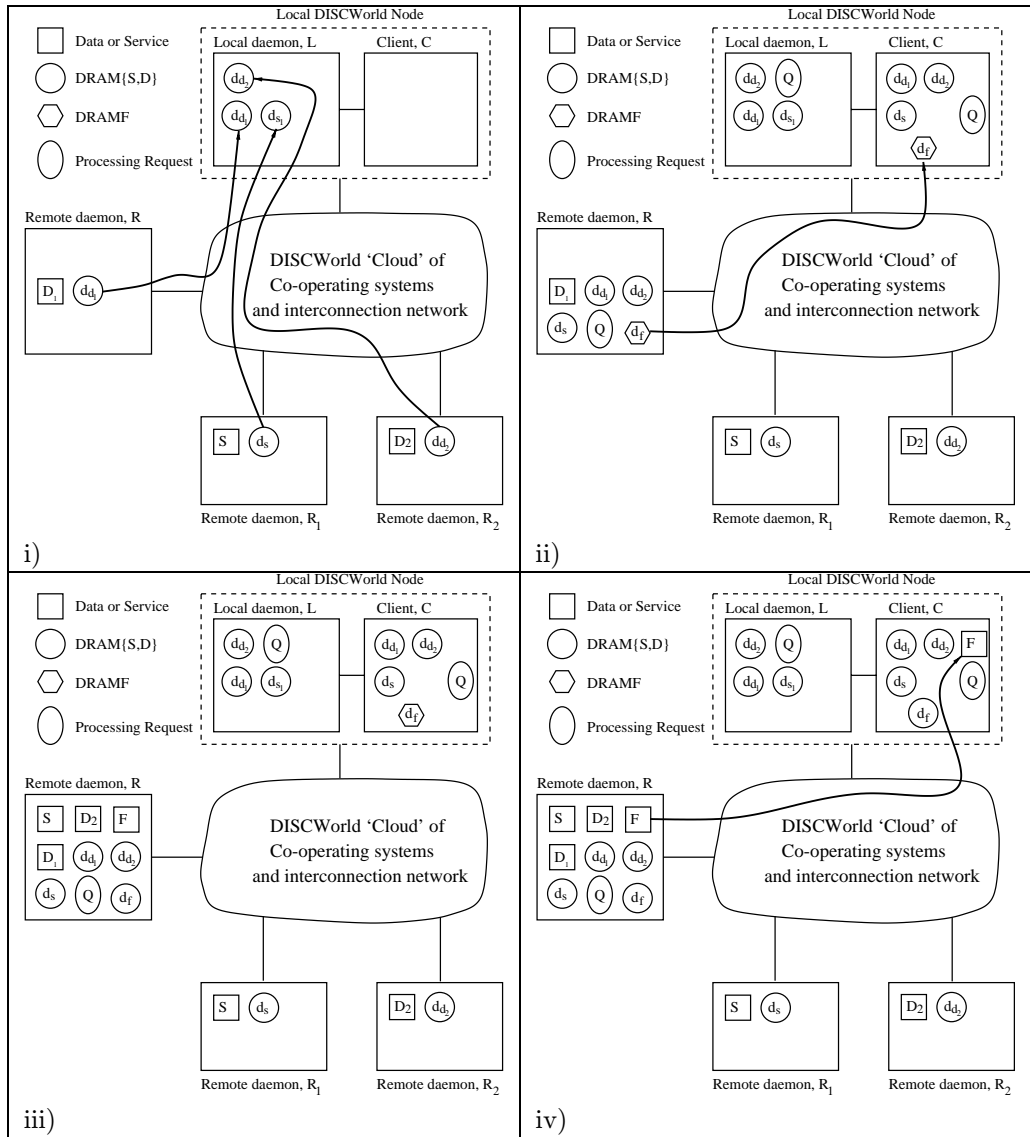


Figure 4: Execution sequence of process networks within DISCWorld. These show: i) DRAM representations of available data and services are distributed around the system; ii) user logs on to client and submits a processing request, causing a future to be returned which represents the resultant data; iii) the future is inspected, causing execution of the processing request to begin; iv) the client's request for the bulk data item F is satisfied.

the DISCWorld philosophy is to abstract away from the user’s file system and to use a more scalable, portable mechanisms such as embodied by DRAMs.

Although our original concept for a DRAM was that of a remote data pointer in an object-oriented framework. It was convenient to extend this idea to allow DRAMs to have a special graphical or iconic representation in a DISCWorld client environment. Our present implementation allows for a DRAM to be attached to a graphical icon in the client screen area, and to be manipulated using the usual mouse semantics. These include being able to drag and drop iconic DRAMs onto other applications and other sensitive screen areas. We are presently exploring collaborating graphical clients that allow two interactive users to share a graphical workspace and to exchange data using DRAMs. Graphical User Interface builder environments such as AVS and JavaStudio make use of similar graphical semantics to move around iconic representations of code and data items and allow these to be interconnected. Our DISCWorld service infrastructure lends itself particularly well to such semantics and we are also building this capability into the DISCWorld graphical client environment.

DRAMs have a direct analogy to the way in which user queries are submitted to the DISCWorld metacomputing environment. Service DRAMs and Data DRAMs are able to be combined together into a process network which depicts the manner, or “recipe” in which a result is constructed. The process network is then either serialised, and sent to the DISCWorld server for processing, or is converted into a textual form called Distributed Job Placement Language (DJPL) [8] before being sent to the DISCWorld server.

4 Summary and Conclusions

We have introduced the DISCWorld Remote Access Mechanism Future (DRAMF), which builds on our base DRAM entity. In order to facilitate the implementation of large-scale distributed service execution, we have extended the DRAM concept to allow data that has not yet been created to be manipulated. Future data is encapsulated in the DRAM Future (DRAMF). DRAMFs are used extensively in DISCWorld to optimise service execution time and cost, especially as there are no guarantees on how accurate a node’s global system state information is. Optimisation is achieved through data and service re-use on a per-node basis.

We have shown how DRAMFs’ ability to be independently named and manipulated makes them first-class objects in the DISCWorld system. Their status as DRAMs allows their contents to be referenced even in the presence of temporary network and server disruptions.

We have demonstrated how data futures can be implemented in a Java environment making use of services provided by our DISCWorld system. We believe a futures mechanism like DRAMFs is a useful feature for any metacomputing

environment to enable proper control and optimal placement of directed acyclic task graphs.

References

- [1] Denis Caromel. Toward a Method of Object-Oriented Concurrent Programming. In *Communications of the ACM*, 36(9):90–102, September 1993.
- [2] P. D. Coddington, K. A. Hawick and H. A. James. *Web-Based Access to Distributed High-Performance Geographic Information Systems for Decision Support*, Proc. of Hawai'i International Conference on System Sciences (HICSS-32), Maui, January 1999.
- [3] K. A. Hawick and P. D. Coddington. *Interfacing to Distributed Active Data Archives*, International Journal on Future Generation Computer Systems, Special Issue on Interfacing to Scientific Data Archives, 16(1999) 73–89. Editor, Roy Williams.
- [4] K. A. Hawick and H. A. James. Distributed High-Performance Computation for Remote Sensing, Proc. of Supercomputing '97, San Jose, November 1997
- [5] K. A. Hawick, H. A. James and J. A. Mathew. Remote Data Access in Distributed Object-Oriented Middleware, *To appear in Parallel and Distributed Computing Practices*, 1999.
- [6] K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus and F. A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. Invited article *Future Generation Computing Systems (FGCS)*, (15)623–635, 1999.
- [7] K. A. Hawick, H. A. James, C. J. Patten and F. A. Vaughan. *DISCWorld: A Distributed High Performance Computing Environment*, Proc. of HPCN Europe '98, Amsterdam, April 1998.
- [8] Heath A. James. *Scheduling in Metacomputing Systems*, PhD Thesis, The University of Adelaide, July 1999.
- [9] H. A. James and K. A. Hawick. *Resource Descriptions for Job Scheduling in DISCWorld*, Proc 5th IDEA Workshop, Fremantle, Feb 1998.
- [10] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation, pp260–267, June 1988.

- [11] J. A. Mathew, A. J. Silis and K. A. Hawick. *Inter Server Transport of Java Byte Code in a Metacomputing Environment*, Proc. TOOLS Pacific (Tools 28) - Technology of Object-Oriented Languages and Systems, Melbourne, 1998.
- [12] Object Management Group. *CORBA/IIOP 2.2 Specification*, July 1998, Available from <http://www.omg.org/corba/cichpter.html>.
- [13] Andrew Silis and K. A. Hawick. The DISCWorld Peer-To-Peer Architecture. In Proc. Fifth IDEA Workshop, February 1998.
- [14] Edward F. Walker, Richard Floyd and Paul Neves. Asynchronous Remote Operation Execution in Distributed Systems. In Proc. Tenth Int. Conf. Distributed Computing Systems, pages 253–259, May 1990.