Mason, Richard S. and Kelly, Wayne A. (2007) Enhancing Data Locality in a Fully Decentralised P2P Cycle Stealing Framework. In Dobbie, Gillian, Eds. *Proceedings Thirtieth Australasian Computer Science Conference (ACSC2007)* CRPIT, 62, pages pp. 41-47, Ballarat, Victoria, Australia.

Enhancing Data Locality in a Fully Decentralised P2P Cycle Stealing Framework

Richard Mason
Queensland University of Technology
r.mason@qut.edu.au

Wayne Kelly
Queensland University of Technology
w.kelly@qut.edu.au

ABSTRACT

Peer-to-peer (P2P) networks such as Gnutella and BitTorrent have revolutionised Internet based applications. P2P approaches provide a number of benefits, however most cycle stealing projects, such as SETI@home, have concentrated on centralised methods which still require massive amounts of concentrated network bandwidth in order to scale. More recent P2P research has developed the concept of distributed hash table (DHT) P2P overlays. These overlays provide efficient and guaranteed message delivery unlike earlier P2P networks which relied on large scale replication to probabilistically find data. Our G2:P2P framework makes use of a DHT overlay to provide a fully decentralised P2P cycle stealing system. Its distributed object programming model allows direct communication between objects and it remains reliable even as the set of peer nodes changes.

In this paper we describe extensions to G2:P2P which allow us to optimise object distribution for locality. The importance of optimising data locality is well understood and has received extensive research, however, in the context of cycle-stealing systems and more generally DHT based P2P networks it is completely unexplored. Whilst our work is motivated by parallel programming, it is generic in nature and may have applicability to other DHT applications.

1. INTRODUCTION

Peer-to-peer networks have become popular of late for a range of applications, including the well known file sharing systems such as Gnutella [9] and BitTorrent [7]. Peer-to-peer in general refers to distributed systems where resources are obtained from an ad hoc collection of client computers (as opposed to dedicated central servers). Typically the set of peer nodes participating in the network changes over time. Early P2P systems such as Napster used an underlying client-server architecture to facilitate communication between peers. Later, fully decentralised (pure) P2P networks where developed to eliminate the impact of a single point of failure and to make it cheaper to scale. These early pure P2P networks were inefficient, requiring a great deal of network bandwidth which prevented their use on lower bandwidth connections [10]. Their design also meant that data present in "distant" parts of the network may not always be found.

Pure P2P research has since centred around the idea of a distributed hash table (DHT) [19, 24]. A hash function is used to map objects to be stored in the P2P network to the peer node that should hold that object. The peer node who's randomly assigned identifier is "closest" to an object's hash value will always hold that data. As the set of peers that make up the P2P network changes over time, the ownership of data changes so as to maintain the above invariant. It is this invariant that allows messages to be efficiently routed (average O(log N) hops) to arbitrary objects without needing to update object references and without needing to resort to forwarding schemes.

While DHTs have been used for a number of applications [20, 18], we believe our G2-P2P framework [13] is the first parallel computing/cycle-stealing framework to exploit this approach. Most volunteer/cycle-stealing systems are based on client-server [12] or hybrid [15] topologies. This typically limits them to simple task based embarrassingly parallel applications. Our DHT based cycle-stealing framework supports a distributed object programming model where remote method invocations can be made between arbitrary peers in the network and remains reliable even as the set of peer nodes that make up the network changes over time.

In this paper we describe extensions to our parallel computing framework that allows us to optimise for locality. The importance of optimising for locality in parallel programs is well understood and there has been extensive work in this area. However, in the context of cycle-stealing systems and more generally DHT based peer-to-peer systems this topic has been completely unexplored. As it turns out, there is room for movement within the DHT concept that allows us to achieve our new goals related to locality while preserving all of the original benefits of DHTs that have made them so popular. Whilst out work is motivated by parallel programming, it is entirely likely that the locality ideas proposed here will have wider applicability to other DHT applications.

Section 2 provides an overview of our G2:P2P framework. It includes details on the underlying Pastry network as well

as the G2:P2P object model and programming interface. In Section 3 we describe four schemes for improving object locality. The schemes build upon one another to provide increasing performance improvements. Section 4 demonstrates the effect of these optimisations on communication and load balancing performance. In Section 5 we discuss other related work before outlining future directions and concluding in Section 6.

2. G2:P2P OVERVIEW

G2:P2P is a fully decentralised (pure) P2P system for executing parallel applications on volunteer machines. The system provides a distributed object based programming model.

2.1 Pastry

A number of DHT approaches [24, 17, 21] have been developed with very similar characteristics. Our framework is based on a DHT system called Pastry [19]. Pastry networks consist of a set of machines (nodes) which are each assigned a unique k-bit key termed a *NodeID*. The assignment of NodeID's to nodes must be done in a fully decentralised manner but in such a way that the NodeIDs are approximately evenly distributed across the address space (so as to achieve approximately even load balance). The simplest approach is to assign them in a pseudo-random fashion (either by generating random GUIDs or by hashing their IP addresses). The address space generated by these NodeIDs is circular; i.e. in an 8-bit address space address 0xFF is adjacent to 0x00.

Messages may be addressed to any NodeID in the address space, regardless of whether an actual node exists with that ID or not, and will be delivered to the node whose ID is closest to the target. Pastry's routing scheme will deliver messages in $O(\log\ N)$ hops using a scheme reminiscent of hypercube routing but with dynamically changing nodes.

Nodes have two notions of locality: physical locality and virtual locality. Physical locality represents closeness with respect to the physical network (the closer the physical locality - the shorter the message latency). When lacking any better physical locality data we generally assume that the numeric difference between two IP addresses is a rough approximation to their physical locality (i.e. two nodes on the same subnet will generally be closer than two nodes on different subnets, etc). Virtual locality is measured by the difference between NodeIDs.

Nodes maintain information to facilitate the routing of messages. Each node remembers a small set of the nodes (size $O(1)$) that are closest to it physically (termed its neighbourhood set), and closest to it virtually (termed its leaf set). The primary routing mechanism is a routing table containing $O(\log(N))$ entries that allows each node to forward any incoming message onto a node that is at least one significant bit closer to its final destination.

2.2 Object Addressing & Programming Model

G2:P2P applications create objects which are distributed on a network of volunteers organised in a Pastry network. Communication between these objects is performed through remote method calls (as in .NET Remoting [16] or Java RMI [22]. However, when using G2:P2P the application programmer does not need to specify a specific server to host a particular class of remote objects, but rather simply indicates that they should be hosted somewhere within the collection of volunteer machines. In fact the volunteer that hosts a given remote object may change over time as volunteers come and go from the network.

When an object is created, it is assigned an *ObjectID* which has the same form as a Pastry NodeID. A construction message is then sent to that ID, informing the node with the closest ID (i.e. the node that receives the message) that it should create a remote object instance. The only information required to refer to and communicate with the object is its ObjectID. If another node joins the network and is assigned a NodeID which is closer to the object then the object should be migrated to the new node. The G2-P2P framework automatically performs these migrations as well as performing logging and check pointing to maximise reliability.

3. EXTENSIONS FOR IMPROVING COMMUNICATION LOCALITY

Objects are assigned an ObjectID when they are created. It is not possible to change an object's ObjectID at a later stage as references to that object may have spread throughout the network and they would need to be updated in a globally synchronised manner. We still however, have some freedom that allows us to optimise for locality. The following four subsections describe four separate schemes that we have developed to optimise different kinds of locality.

3.1 ObjectID Assignment

Our first extension is designed to increase the likelihood that communicating objects are hosted on the same node. Previously our approach to assigning ObjectIDs was to generate them randomly (similar to the allocation of NodeIDs). This was done so as to approximately evenly spread the ObjectIDs over the address space so as to achieve approximately even load balance. However, unlike NodeIDs, ObjectIDs don't need to be generated de-centrally – a single client or node often activates a collection of related remote objects at a time. Further, these remote objects typically don't exhibit purely random communication behaviour (patterns such as "nearest neighbour" are very common).

By allocating ObjectIDs in a more intentional manner we can optimise for communication locality while preserving load balance. Our new approach is to allocate ObjectIDs to a collection of related objects so that they are uniformly rather than randomly distributed over the address space. We do this in such a way that objects that communicate regularly are closer together with respect to their ObjectIDs than objects which don't regularly communicate. For this optimisation to be beneficial we assume that the number of remote objects created will be much larger than the number of volunteers, so by assigning similar ObjectIDs to objects that communicate regularly, we ensure that in most cases they will be assigned to the same physical host. The set of volunteer nodes participating in the network may change over time and therefore so will the mapping of remote ob-

jects to volunteer nodes, but the locality properties will be preserved.

Our notion of closeness is based on ObjectIDs and is therefore 1 dimensional (more precisely a 1D-ring). Other communication patterns between objects must therefore be "folded" by the programmer onto this 1D space.

The API for allocating ObjectIDs in this way basically tells the system to start uniformly assigning ObjectID to any remote object activations that are about to be performed. The *StartLayout* method needs to know the number, N, of objects that will be created in this group, so that they will be properly spaced to collectively cover the entire address space. Following this call, the next N remote object activations will automatically be assigned ObjectID using this scheme.

```
Island[] islands = new Island[numIslands];
G2P2PChannel.Current.StartLayout(numIslands);
for (int i = 0; i < numIslands; i++)
islands[i] = new Island(i);
```

The same pattern can be repeated later if another set of remote objects need to be created that are unrelated to the original set (with respect to communication pattern). Later allocations will use a different random offset so as to not collide with the original uniformly generated IDs.

Section 4.1 presents the results of the new ObjectID assignment process on communication efficiency and load balancing.

3.2 Object Groups

The optimisation in Section 3.1 increases the chances that two objects that communicate often will be located on the same machine. It cannot however guarantee that they will always be hosted on the same machine. The optimisation described in this section is designed for situations in which a set of objects communicate so frequently with one another that they should always be collocated on the same host. The way we do this is by adding extra bits to the length of our ObjectIDs beyond the length of our NodeIDs. We can think of this as turning ObjectID into decimal numbers rather than integers. Objects which share the same integer part will always be mapped to the same node. The actual physical node may change over time as nodes come and go, but the group of objects will always be collocated. Note that this locality comes at the cost of load balance and therefore parallelism. If a group of objects are assigned the same integer part, they will always map to a single machine, even if there are a large number of other nodes in the network which are completely unused.

An alternative to this optimisation is to encapsulate these objects inside a single remote object container that forwards messages to them. The advantage of the approach we have described is that each of the objects in the group remain individually addressable by remote clients. Whether or not a set of objects should always be collocated is an performance optimisation which ideally should be kept separate from the application logic and the abstractions used.

3.3 Node Movement

The assignment of ObjectIDs and NodeIDs described so far will lead to approximately the same number of objects being allocated to each node. In some cases however, particularly with smaller networks, this balance will not be achieved.

As we discussed earlier, it is extremely problematic to change an object's ObjectID after it is initially assigned as references to that object may have spread throughout the entire network. It is, however, possible for a node's NodeID to change at a later time. A simple way to explain why this is possible is to view the process as equivalent to a volunteer node leaving the network and then immediately rejoining (with a new NodeID). So, clearly it is possible, and with a little ingenuity we can develop a process that is much more efficient than the naïve implementation hinted at above.

But why would we want to change a node's NodeID? What we generally want to do is reassign NodeIDs so that each node gets a more even number of objects. To do this perfectly we would need global knowledge and even if we had such global knowledge, what might be optimal one minute might not be the next as nodes and objects come and go over time. As with all decentralised systems we must attempt to achieve close to global optima through local acts.

At the local level a node's goal is to try to evenly space itself between its two neighbours. Since each node is trying to do this independently it may take a large number of small adjustments to get to a steady state. To minimise the total number of adjustments nodes may make use of the extra information they contain in their leaf set. Instead of simply placing itself halfway between its two immediate neighbours, it measures the distances to all of its known neighbours and attempts to balance them.

It is important that these NodeID changes are moving incrementally towards a global optima. If the changes are too dramatic the system may not converge, in particular, we don't want nodes to move so far that they move past other nodes and thereby change their relative order. By keeping nodes relative order stable, their leaf set will remain constant. This guarantees that the Pastry routing mechanism will continue to function correctly. We therefore use the following formulas to calculate incremental NodeID changes.

$$
\begin{aligned}
&N' = N + S\frac{(W_{anti} - W_{clock})}{2} \\
&W_i = \sum_{n \in LS_i} 1 - \frac{(N - n_{pos} n_{weight})}{S_{leafset}} \\
&N = Node\,position \\
&S = Size\,of\,network \\
&S_{leafset} = size\,of\,leaf\,set
\end{aligned}
$$

Where W_i indicates the weight of the respective half of the leaf set and LS_i represents the set of nodes in each half of the leaf set. Essentially W_{clock} and W_{anti} calculate the force expressed on the node by the nodes by the clockwise and anti-clockwise half leafset. Nodes that are further away express less force than nearby nodes. The new position is calculated to equalise those two forces.

We also include a weighting factor (n_{weight}) on each node. This allows us to take in to account peer nodes that have greater processing power (e.g. a cluster computer rather

than a PC). Such nodes will be responsible for a greater portion of the address space and hence will host more objects.

Since it is imperative that we maintain the relative order of the nodes we further restrict node movement so that any single move may not travel more than half the distance to the next node. Without this restriction nodes may "cross over" each other as they independently calculate their moves. This restriction may slow down the progress towards the global optima, however, once the network is well dispersed node movements are generally small anyway and this restriction is rarely encountered.

Once a node calculates a new position it informs all of the members of its leaf set. Additionally it must now monitor incoming communications to detect other nodes that have references to it (such as references in their routing table). When these are detected the node can respond with its new id. Since most movements will be relatively small, messages sent via a routing table will usually still send the message closer to its target, and even if it doesn't, routing can continue with a temporary disadvantage of some extra network hops.

The node's routing state must be updated to reflect the new NodeID. The leaf set is not affected because the nodes with adjacent NodeIDs are guaranteed not to change. Similarly, the neighbourhood set is unchanged since it is only related to physical locality, not the NodeID. The amount of the routing table affected is directly related to the distance moved by the node. Generally portions of the routing table will stay valid, while later sections will need to be repopulated.

When a node joins there is a significant opportunity to optimise its location before it even advertises itself to other nodes. As part of our node movement technique we extend the joining process to reduce the number of movements a node must make before the network stabilises. Normally to join a Pastry network a node routes a special join message to its prospective new NodeID. This is received by the node whose ID is closest to the new position who then replies with confirmation of the IDs acceptance and with some initial data to start populating the node's routing state. With minimal changes we can alter the processing of join messages so that the receiving node can reply with a different NodeID for the node to use during joining. This new ID can be selected so that it already has a balanced leafset, reducing the need for further adjustments directly after joining.

Section 4.2 presents the results of moving node's IDs on communication efficiency and load balancing.

3.4 Physically Related Nodes

The optimisation in Section 3.1 increases the chances that two objects that communicate often will be located on the same machine. There will however usually be situations where objects that communicate often are located on different machines. If they can't be on the same machine then we would prefer that they are on machines which are physically close. To achieve this we continue to assign ObjectIDs as described in Section 3.1 but change NodeIDs so that nodes with similar NodeIDs will be physically close to one another. If two objects communicate frequently then they will be assigned similar ObjectIDs. If we are lucky this will mean they will be hosted on the same node. If not, then it they will be hosted on nodes that have similar NodeIDs which implies they are physically close.

Our basic approach is to use IP addresses (or some other measure of physical locality) as our NodeIDs. This ensures that nodes with similar NodeIDs will tend to be physically close. The problem with this approach is that the resulting NodeIDs will not be evenly spread over the address space. Thankfully, the optimisation described in Section 3.3 can correct that situation.

However, using the node movement technique from Section 3.3 complicates the joining process. As nodes adjust their IDs, they may find themselves moving significantly away from the ID that was generated from their IP address. This means that we need to alter the way join messages are routed so that new nodes can still be placed adjacent to nodes with similar IP addresses. To do this we introduce two NodeIDs – the initial NodeID which was generated directly from the node's IP address and the current NodeID which is being used by the node. While normal communication messages use the current NodeID for routing, join messages will be routed to the node whose initial ID is closest.

Unfortunately the routing table maintained by each node is designed for standard routing and hence can not be used for routing to initial IDs. We can however rely on the order of current NodeIDs being the same as the initial NodeIDs. This means we can use the leaf sets to route messages, albeit with a worse case of N/2 network hops. In practice, joining is generally performed by contacting a physically close node and using it to initiate the join message. This means that regardless of how far node IDs move during execution, join messages will always be initiated reasonably close to their final target.

There is one disadvantage of applying this optimisation. In a normal Pastry network, nodes in a particular physical locality are likely to be widely spread throughout the network (with respect to their NodeIDs). So, if some fault in that physical locality occurs (such as a local power loss or a local network going down) then loss of nodes will be felt in a more dispersed fashion across the network rather than in a single large cluster of nodes. Pastry networks are designed to be able to recover from individual nodes disappearing, provided that other nodes its leaf set remain. So, by changing this aspect of the pastry network we decrease its ability to recover from local faults. This is obviously a trade-off that must be made between efficiency and reliability. Our previous paper [14] describes additional techniques that we have developed to enhance reliability.

4. RESULTS

The optimisations presented in Sections 3.1 and 3.3 have been implemented in our G2:P2P system as well as a simulator. We have used the simulator to test the optimisations as it allows for testing of large networks. The simulator generates a Pastry network and simulates a set of objects communicating periodically in a 1D nearest neighbour configuration.

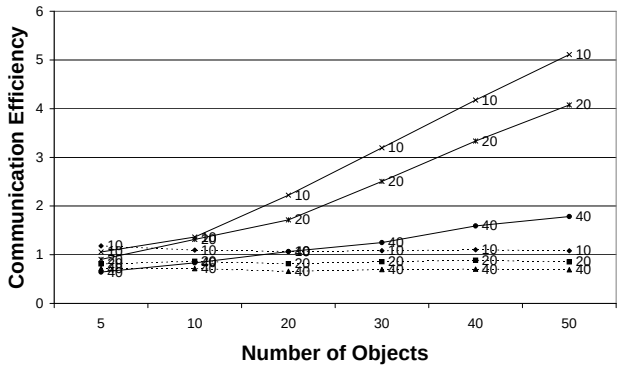

Figure 1: ObjectID Adjustment Communication Efficiency

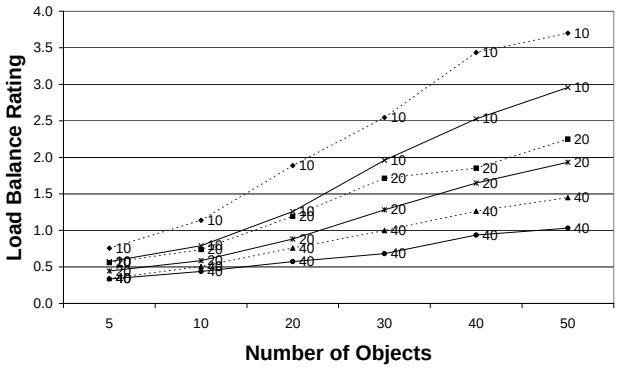

Figure 2: ObjectID Adjustment Load Balance Rating

We use two metrics to analyse the effectiveness of our enhancements. The first is the communication efficiency. This is the ratio of messages sent between objects to the number of network hops used by those messages. An efficiency of one indicates that each message required a single hop across the network while higher values indicate that some messages were delivered without requiring the network.

The simulator has not taken into account the physical communication costs between different nodes. Pastry already accounts for physical locality in its selection of which nodes to use for network hops. The work in Sections 3.1 and 3.3 is mainly concerned with minimising the number of hops required.

Our second metric measures how well balanced the objects are across the network. This is provided by calculating the standard deviation of the number of objects on each node, lower values indicating that each node was carrying approximately the same load.

4.1 Enhanced ObjectID Results

Our first test shows the effect of our enhanced ObjectID assignment. The even spacing of ObjectIDs significantly improves the communication efficiency of the network especially as the number of objects outstrips the number of nodes. Load balancing is also mildly improved, though that effect declines as more nodes are added to the network.

Figures 1 & 2 show the results of activating the ObjectID assignment optimisations. Dotted lines indicate tests without the optimisation while solid lines indicate tests with the optimisation activated. Each line indicates a separate test with the label indicating the number of volunteer nodes involved.

4.2 Node Movement Results

The second test shows the effect of adding the node movement extensions to the new ObjectID assignment method. As expected we find that node movement significantly improves the load balancing, though the effect decreases as more nodes are added to the system. Communication efficiency on the other hand decreases slightly when node movement is added. This is directly related to the load balancing, since as the objects spread amongst all of the nodes, less communication is performed internally within nodes.

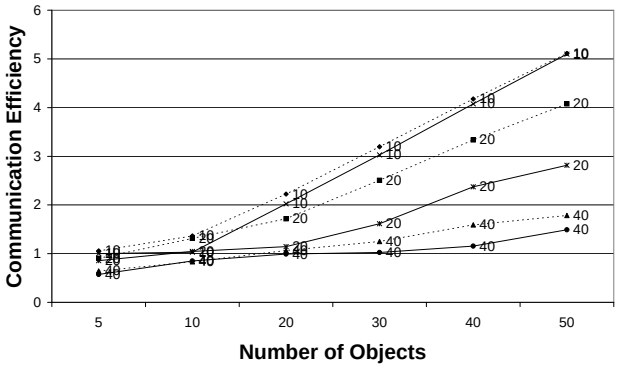

Figure 3: Node Movement Communication Efficiency

Figures 3 & 4 show the results of activating the node movement optimisations. Dotted lines indicate tests without node movement while solid lines indicate tests with the optimisations activated. As before, each line indicates a separate test with the label indicating the number of volunteer nodes involved.

5. RELATED WORK

Current cycle-stealing systems are predominantly client-server based [12, 1, 6] or use a hierarchical layout to improve scalability [4]. Applications making use of these frameworks are generally limited to embarrassingly parallel approaches, however, more recently work has been performed to allow a other application models such as branch and bound [15] and continuations [11]. G2:P2P's direct inter-object communication is unavailable, and difficult to imitate, using the client-server approach of these systems.

Condor [23] provides direct communication between cycle-stealing processes using messaging APIs like MPI and PVM, however this feature requires a highly reliable environment. Similarly, Kinitting Factory [5] uses Java RMI for communication, but is unable to cope with volunteers leaving the

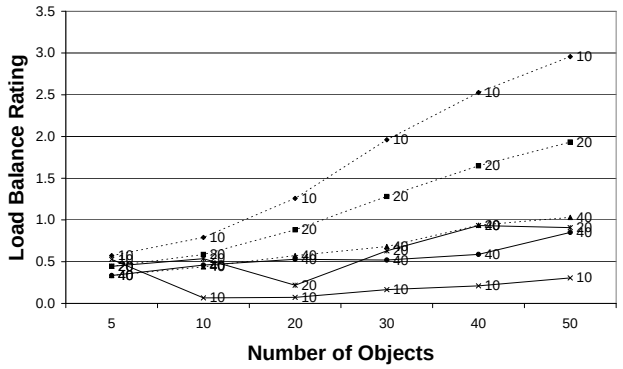

Figure 4: Node Movement Load Balance Rating

system. G2:P2P provides direct communication and is reliable across volun-teer arrivals, departures and failures.

Some other attempts have been made in creating decentralised cycle-stealing applications [8, 3], but this research concentrates on the economic aspects of cycle-stealing and doesn't address how pure P2P features such as direct communication can extend the programming model of cycle-stealing applications. The DREAM project [2] provides an innovative approach for doing pure P2P cycle-stealing but primarily concentrates on the field of evolutionary algorithms.

6. FUTURE WORK & CONCLUSIONS

G2:P2P's direct object-to-object communication ability is unique amongst cycle stealing projects. The extensions explored in this paper effectively enhance this communication feature, significantly improving its performance for common communication patterns. Additionally, the alterations made to the Pastry layer provide some generic enhancements which may be beneficial to other applications using distributed hash table P2P overlays.

There is still scope for further improvements to object locality. In particular, support for a wider range of communication patterns. Currently nearest neighbour communication is well supported. This is the obvious starting point as it maps well to the Pastry address space, however methods for supporting other topologies such as mesh or tree based communication may also be able to be supported.

A secondary benefit of this locality work has been a substantial improvement to the load balancing of G2:P2P, particularly for smaller networks. There is scope for further work in this area taking into account the variety in processing power of volunteers and the workload incurred by different objects. Like the enhancements presented here, adjusting allocation for different loads is difficult, though not unachievable, in fully decentralised networks.

6.1 Bibliography

7. REFERENCES

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA*, November 8, 2004.

[2] M. G. Arenas, P. Collet, M. J. A. E. Eiben, J. J. Merelo, B. Paechter, M. Preuβ, and M. Schoenauer. A framework for distributed evolutionary algorithms. *Lecture Notes in Computer Science*, 2439:665–675, 2002.

[3] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Unstructured peer-to-peer networks for sharing processor cycles. *Parallel Computing*, 32(2):115–135, February 2006.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proc. of the 9th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-96)*, 1996.

[5] A. Baratloo, K. M., H. Karl, and Z. M. Kedem. Knittingfactory: An infrastructure for distributed web applications. Technical Report TR1997-748, New York University, 13, 1997.

[6] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3):417–437, 2005.

[7] B. Cohen. Incentives build robustness in bittorrent. `http://www.bittorrent.com/bittorrentecon.pdf`, 2003.

[8] R. Gupta and A. Somani. Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems, Cambridge, MA*, 2004.

[9] G. Kan. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Gnutella. O'Reilly & Associates, Inc, 2001.

[10] G. Kan. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Peformance. O'Reilly & Associates, Inc, 2001.

[11] W. Kelly, P. Roe, and J. Sumitomo. An enhanced programming model for internet based cycle stealing. In H. R. Arabnia and Y. Mun, editors, *PDPTA*, volume 4, pages 1649–1655. CSREA Press, 2003.

[12] W. Kelly, P. Roe, and J. Sumitomo. G2: A grid middleware for cycle donation using .net. In *2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[13] R. Mason and W. Kelly. Peer-to-peer cycle sharing via .net remoting. In *Proceedings of the Ninth Australian World Wide Web Conference (AusWeb03)*, 2003.

[14] R. Mason and W. Kelly. G2-p2p: A fully decentralised fault-tolerant cycle-stealing framework. In *Proceedings of the Australasian Workshop on Grid Computing and e-Research (AusGrid05), Newcastle, Australia*, 2005.

[15] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: scalability issues in global computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[16] P. Obermeyer and J. Hawkins. Microsoft .net remoting: A technical overview. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp`, 2001.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[18] S. Rhea, P. Eaton, D. Geels, H. Weathersppon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, March 2003.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[20] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[22] Sun Microsystems. Java remote method invocation - distributed computing for java. `http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html`, 2004.

[23] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[24] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.