G2-P2P: A Fully Decentralised Fault-Tolerant Cycle-Stealing Framework

Richard Mason **Wayne Kelly**

Centre for Information Technology Innovation
Queensland University of Technology
Brisbane, Australia
Email: {r.mason, w.kelly}@qut.edu.au

Abstract

Existing cycle-stealing frameworks are generally based on simple client-server or hierarchical style architectures. G2:P2P moves cycle-stealing into the "pure" peer-to-peer (P2P), or fully decentralised arena, removing the bottleneck and single point of failure that centralised systems suffer from. Additionally, by utilising direct P2P communication, G2:P2P supports a far broader range of applications than the master-worker style that most cycle-stealing frameworks offer.

G2:P2P moves away from the task based programming model typical of cycle-stealing systems to a distributed object abstraction which simplifies communication. It uses a distributed hash table based overlay network to provide an efficient method of referencing application objects while still allowing volunteer machines to come and go from the network. Most importantly, G2:P2P provides a sophisticated fault tolerance mechanism to ensure applications execute correctly. This mechanism is entirely automated, requiring no special effort on the application developer's part. The framework is implemented as an extension to .NET's Remoting framework, providing a familiar model for application programmers and an easy upgrade path for existing .NET sequential applications.

Keywords: Pure Peer-to-Peer, P2P, kCycle Stealing, Fault Tolerance, .NET

1 Introduction

Many cycle-stealing applications and frameworks have been developed (SETI@home 1999-2004, Kelly, Roe & Sumitomo June 2002, Cappello, Christiansen, Ionescu, Neary, Schauser & Wu 1997). These systems are often loosely referred to as peer-to-peer (P2P) systems, as compute cycles are contributed by peer nodes. The peer nodes in these cycle stealing systems are, however, almost always coordinated by a dedicated central server node (or nodes) and support only master-worker style programming models, or simple variations thereof. The central server in such systems often becomes a bottleneck and represents a single point of failure.

The idea of fully decentralised P2P networks has been studied for some time, however, it has largely been applied to file sharing style applications such as the notorious music sharing networks (Kan 2001, Sharman Networks 2004). The framework that we present in this paper is the first that we know of that attempts to support generic cycle stealing over a fully decentralised P2P network.

The benefits of utilizing a fully decentralised P2P network are obvious – unlimited scalability and no single point of failure; however the challenges to achieving this goal are also great. The basic problem is that the peers that make up a cycle stealing network may vary considerably during the lifetime of a computation running on the network. In a file sharing environment, this is not a major problem; it simply means that searching for a particular song may temporarily return no hits. With a long running parallel application, however, the consequences of losing even part of the applications state in most cases will be catastrophic – requiring the entire application to be completely rerun.

Clearly, if a peer node decides to leave the network, its current application state and tasks must be taken on by some other node, i.e. they must be able to migrate. For any message passing style framework, whether it be based on communicating processes (such as MPI or PVM), or based on communicating distributed objects (such as Java RMI (Sun Microsystems 2004), .NET Remoting (Obermeyer & Hawkins 2001) or Gardens (Roe & Szyperski February 1998)), there is the need for some form of remote reference or addressing scheme. That is, we need a way of referring to the entities that we wish to communicate with. In traditional message passing systems, these entities reside permanently on a given physical node. The remote references include the address of this physical node, so routing messages to these entities is relatively trivial.

A number of schemes have been proposed for handling remote references to entities that may migrate. The original Gardens system (developed at QUT), for example, required all machines that held remote references to be informed whenever a migration took place. This required a global synchronization and so limited scalability. Other systems, such as JavaParty (Philippsen & Zenger 1997) use a forwarding scheme, that is, the remote references are not actually updated, messages are simply send to the machine where the object used to reside, which is expected to forward them to the object's new home. This requires nodes that have departed from the P2P network to continue to participate (albeit, to a lesser extent) by agreeing to forward such messages. Our approach allows entities to migrate without requiring global synchronization and without requiring departed nodes to continue to participate in any way. We do this by using a physical location independent address scheme, i.e. an addressing scheme that does not include the physical address of the entity.

The idea for our addressing scheme comes from modern P2P networks, such as Pastry (Rowstron & Druschel 2001*a*), Tapestry (Zhao, Huang, Strib-

ling, Rhea, Joseph & Kubiatowicz January 2004), Chord (Stoica, Morris, Karger, Kaashoek & Balakrishnan 2001) and CAN (Ratnasamy, Francis, Handley, Karp & Shenker 2000), that implement a fully decentralised distributed hash table (DHT). Pastry and Tapestry assign a random n-bit identifier to each physical node currently participating in the network. Data to be stored in the hash table in then hashed and stored on the physical node whose ID is numerically closest (modulo 2^n). The beauty of this approach is that as nodes come and go from the network, the hash table entries will be automatically mapped (migrated) to new physical nodes. The references to these entries (i.e. the keys) don't need to be updated. The major technical innovation is the use of a fully decentralised routing algorithm that guarantees the delivery of lookup messages to the appropriate physical node in at most $O(\log N)$ hops, where N is the number of physical nodes currently participating in the network. Our approach mirrors that of Pastry and Tapestry, but we could equally have chosen to follow other modern DHT implementations such as Chord and CAN which achieve the same effect using slightly different approaches.

For our cycle-stealing framework (G2-P2P) we have opted to provide a message passing paradigm in the form of a distributed *object model*; however, our basic approach could equally have been applied to distributed *process* models such as MPI. We assign a 128-bit GUID to each physical node when it joins the network. Our objects are also assigned and referenced via a 128-bit GUID. Since we are (randomly) generating the keys ourselves, there is no need to hash them; we simply use the object's GUID directly to map it to the physical node whose GUID is numerically closest. The random generation of GUIDs and associated mapping algorithm will tend to generate a roughly even number of objects per physical node in the same way that traditional hash table buckets tend to have the same number of elements on average. In Section 3.5 we describe how this probabilistic load balancing approach can be refined.

Section 2 explains how our G2:P2P framework presents itself to application programmers via .NETs extensible Remoting framework (Obermeyer & Hawkins 2001). Section 3 describes the infrastructure that we have developed to allows volunteers to effortlessly contribute computational resources to one of our P2P networks. The highlight of this paper is Section 4 which describes how all of this can be done while ensuring fault tolerance even in the most volatile of environments. Section 5 briefly discusses one of the sample applications which we have implemented using our current framework. Finally Sections 6, 7 and 8 discuss related work, our future directions and conclude the paper.

2 The Programming Model

The .NET Framework includes support for developing distributed applications through a framework called .NET Remoting. Like Java Remote Method Invocation (RMI), .NET Remoting allows objects to be created and accessed remotely without requiring the application programmer to be concerned with the communication details. Notably, .NET Remoting is a particularly extensible framework, allowing middleware developers such as ourselves to intercept messages and to plug in custom components at many points along the message pathways. For example, as well as directly supporting standard transport protocols such as HTTP and TCP, the .NET Remoting framework allows custom transport protocols to be implemented and used. Much of our G2-P2P framework is implemented in the form of such a custom transport channel. Application programmers can therefore implement parallel applications using our G2-P2P framework in much the same way they would any other .NET Remoting application. By exposing our framework via such a "COTS" interface, together with .NETs support for a wide range of "COTS" programming languages makes G2-P2P widely accessible and simple to use.

.NET Remoting channels are generally responsible for routing messages to remote objects based on the object's URI. These URIs are assigned when an object is created and generally consist of: a protocol, a machine address and the ID of the object. In our case, we define a custom protocol G2P2P. An exact machine address is not given as the object may be held on different physical machines throughout its life time. We instead identify only a logical network name together with a GUID to identify the object. The network name doesnt refer to the address of an actual machine, but rather a changing collection of machines that are working together to form a P2P network. Including such a network name in the URIs allows multiple P2P network instances to coexist. An example URI might be g2p2p://plas.qut.edu/fccf281d-47bd-45c7-8f2b-a48d462d171b. Only the network name is chosen by the application programmer, the GUIDs are auto generated by the G2-P2P framework.

.NET Remoting uses a configuration file to list the types of objects that should be created remotely; and for each of those types, the protocol to use and the URL of the machine that should host such objects. In our case, the URL is the name of a P2P network instance rather than a specific physical machine. After having read this standard configuration file, the .NET runtime then knows to convert any attempts to create instances of such types into remote activations. The end result of such a remote activation is a local transparent proxy object (containing the remote object's URI) which to the client looks exactly like an instance of the type that it was originally trying to create. Any method calls on this transparent proxy object will automatically cause the method call to be serialized and routed to the corresponding remote object using the specified routing protocol. If the URL specified the use of the G2P2P protocol then our Pastry-style routing algorithm will cause the message to be delivered ultimately to whichever physical node is currently responsible for hosting the remote object.

3 G2:P2P Volunteers

Normally for .NET Remoting to work, a process must be executing on each of the remote machines that are configured to host remotely activated types. These processes basically listen for activation messages and remote method invocation messages from other nodes. The application code that is run on these processes includes the implementation of the remotely activated types that they are configured to host.

In our cycle stealing scenario two things are different. Firstly, the host processes should only execute on volunteer machines when they would otherwise be idle. Secondly, the implementation of the types that will be hosted on volunteer machines cannot necessarily be expected to already reside on those machines. We want volunteers to be able to volunteer their machines for whatever use they can best be put, rather than only being able to volunteer to help a specific application or fixed set of applications.

3.1 Run When Idle

We provide a generic volunteer host application that runs as a service (or daemon) process and automatically detects when the machine is idle. The owner of each volunteered machine can configure the criteria by which their machine is judged to be idle. This includes the option of allowing cycle-stealing computations to linger-longer (Ryu & Hollingsworth 2000) provided the machine's processor load remains relatively low.

3.2 Joining and Leaving

Before a volunteer can receive any messages it must join the P2P network. To do so, it must locate and introduce itself to some other node already participating in the P2P network. We currently use a well known server machine to assist in locating such a peer node. Once a node is accepted into a P2P network, objects will automatically be migrated to it that are closer to its GUID than the GUID of the node where they previously resided. Similarly, when a node indicates that it wishes to leave the P2P network, its objects are automatically migrated to whichever nodes they will become closest to.

3.3 Dynamic Code Download

Before a volunteer can host a new type of object it needs the code that implements that type. The generic volunteer host is responsible for automatically and dynamically downloading such code and caching it for possible future use. We support a couple of different mechanisms for serving this application code to volunteers. The first option is to make it available on a dedicated server machine. The second option (in keeping with the spirit of a pure P2P system) is for the peer nodes to serve up the application code. Under this scheme, the name of the type is hashed and its code stored on the peer node those GUID is closest to that hash value. As with our objects, the physical machine that hosts a given type's implementation may vary over time. When an application first creates objects there will be considerable demand for the associated code. To help spread the load associated with such code access, a distributed caching scheme can be used such as that proposed by the PAST (Rowstron & Druschel 2001*b*) project.

3.4 Object Migration

The .NET Framework includes a serialization framework for storing the state of an object; however, this simply stores the state of the object, ignoring any threads that are currently executing methods on the object. There is research into saving thread state in .NET (Cooney & Roe 2003), however, this requires each application assembly to be run through a preprocessor before thread migration is possible. Our framework instead waits till all threads have ceased execution before migrating objects. Once migration is triggered a flag is set to prevent any further incoming remote method calls from commencing. The system pauses until any currently executing remote method calls complete. For this reason G2P2P application programmers are strongly encouraged to ensure that all remote methods execute for only relatively short periods of time. This apparent limitation can be worked around by breaking longer methods or loops into a series of method calls, each of which calls the next via a remote interface. Future work will endeavour to develop a less restrictive programming model.

3.5 Improved Load Balancing

The probabilistic load balancing scheme described earlier means objects will be fairly evenly distributed on average. Obviously in specific cases, the actual load balance may be quite poor. The uniform manner in which objects are mapped to nodes cannot be arbitrarily changed as the decentralized routing algorithm crucially depends on it. We can, however, make *local* adjustments to the mapping of objects to nodes in order to "manually" improve the load balance. All Pastry nodes maintain a *LeafSet* which consists of the set of nodes whose GUIDs are closest to it. Rather than insisting on objects always being hosted on the node that is closest to its ID, we allow objects to be hosted on one of the other nodes in the designated node's leaf set in order to better spread the load. Standard pastry routing will therefore deliver the message to the "wrong" node, the node where the object "should" have resided. However, the node where the object actually resides is not far away (just an additional hop) and all nodes within this local vicinity know about this "special arrangement".

4 Fault Tolerance

The major issue facing fully decentralised cycle-stealing systems is the possible loss of application state resulting from peer nodes either crashing or leaving the P2P network unexpectedly.

The situation for cycle-stealing frameworks using a client-server architecture is considerably simpler. Firstly, the dedicated server can be relied on to persist application state. Secondly, the programming models used by these systems are typically much simpler – most employ a functional style programming model in which individual tasks don't directly communicate. Fault tolerance in such systems can be achieved by either:

- Periodically checkpointing tasks on volunteer machines so that they can later be resumed from that point (SETI@Home), *or*
- Simply discarding any work that might have been done towards executing a task on a particular volunteer and re-execute it from scratch on some other available volunteer based on basic information retained on the central server (G2-Classic (Kelly et al. June 2002)).

Neither of these approaches is applicable for a fully decentralised architecture that allows direct and arbitrary communication between peer nodes. Allowing objects to disappear for a significant period of time (option 1) could seriously impact the rest of the application, as they will be unable to contact the missing portion. Option 2 allows objects to be always available, however, since objects maintain state which may be influenced by communication they have received, the system can not simply restart an object; effort must be made to return the object to the state it was in when the volunteer left.

G2:P2P requires a more sophisticated fault tolerance system. A major challenge for fault tolerance systems is minimising overhead. If we are not careful we will end up with a parallel system that is extremely fault tolerant but executes slower than the same application executing sequentially. The degree of fault tolerance needed depends on what assumptions we are willing to make about what might go wrong, how likely those occurrences might be and what their consequences would be. The overhead of fault tolerance is typically proportional to the degree of tolerance achieved. There are two kinds of costs involved: the cost of recovery when something goes

wrong, and the overhead associated with achieving fault tolerance (checkpointing, logging, etc) even if nothing actually goes wrong. We generally assume that failures are relatively infrequent, so we largely ignore recovery type costs – what is crucial in that case is simply that we are able to recover and don't have to rerun the entire application. We are therefore most interested in minimizing the ongoing overhead.

We consider three sets of assumptions, which lead to fault tolerance mechanisms with correspondingly different levels of overhead. We leave it to the application user to decide which of these three mechanisms they wish to adopt for a particular execution of their parallel application.

4.1 Scenario 1

The first scenario is the optimistic one in which we assume volunteer nodes never crash or leave the network unexpectedly. They are free to leave the network at any time, but they agree to migrate their existing objects to other volunteers before departing. This scenario requires no special fault tolerance mechanisms beyond basic object migration, it therefore incurs no overhead.

4.2 Scenario 2

The second scenario assumes that volunteers will occasionally crash or temporarily lose connection, but that they will return to the network within a relatively short period of time. The major problem here is trying to recover the memory state when a volunteer crashes unexpectedly. For this we use a combination of object checkpointing and message logging. Each volunteer periodically checkpoints the state of each of its objects to the local disk using the same serialization process used for object migration. Between checkpoints volunteers also maintain a log of all incoming and outgoing remote method call messages. If a fault occurs, the objects' states are recreated by first restoring them to their most recent checkpoint. Each object's incoming messages since its most recent checkpoint are then replayed in sequence. Like all message logging systems, G2:P2P assumes that the system is piecewise deterministic, that is, the only non-determinism is the order in which messages are received. This is important so that when messages are replayed, the same results are obtained. Any outgoing messages generated by this replay are discarded to avoid those messages being sent multiple times.

4.3 Scenario 3

The most pessimistic scenario assumes that volunteers will occasionally crash or leave the network without notice and never return. The fault tolerance approach used in Scenario 2 relies on the local disk of the fault machine to be able to recover the lost memory state. Clearly this is not appropriate for this scenario. Our approach is still based on periodic checkpointing and message logging; however, we must make use of other nodes to store this information rather than just the local disk.

When checkpointing objects externally to the volunteer, the checkpoint must be saved to multiple nodes to ensure that the object is available when required. Since this requires significant communication, erasure codes (Plank 1997, Luby, Mitzenmacher, Shokrollahi, Spielman & Stemann 1997) may be used to decrease the amount of data sent. Erasure codes allow the data to be split into n blocks which are then distributed amongst different nodes. To recover the initial data only m blocks are required (where $m < n$). This both lowers the amount of data that is sent during the checkpointing and increases the likelihood of being able to recover the data in the case of a failure.

When replaying messages it is important to ensure that any messages generated during the re-execution do not result in duplicate execution on their target objects. This is prevented by storing which messages have been received by an object. Additionally the results of these messages should be stored so that if a particular method call is received again, its result may be returned without requiring the method to be actually executed.

The following points outline all places where data is stored during a method call:

- The method *caller* stores the details of the method call on its *local* store
- The method *receiver* stores the caller's identity (i.e. its GUID) and a unique identifier for the message on its *local* store
- If this is the first message received by this caller then the caller's GUID and method's identitifier are stored in *permanent* storage (eg. adjacent volunteers)
- The result of the method call is stored on both the caller's and the receiver's *local* store

The details of a method call consist of the target of the call along with the parameters of the call. Each method call is assigned a unique identifier by its caller. This is generally the GUID of the calling object along with a simple counter which is incremented on each call. This identifier is used to replay messages when they are requested and to search for duplicate messages that may be received during object reconstitution.

To recover an object after a volunteer has crashed, its last checkpoint is loaded. The local store is then inspected to find any messages that were received after the checkpoint. These messages are retrieved from their respective sources and replayed. As the messages are re-executed they may recreate method calls on other objects. Before these calls are sent the local store is checked to see if the result is already available. If it is then it is simply returned. If the method call is found but its result is not yet available there is no need to resend the request.

If a catastrophic failure occurs then the local store will be unavailable. The latest checkpoint available from the network is retrieved by another node and reconstituted. The sources of all messages sent to the object are also retrieved and a message is sent to each of these objects requesting any messages be resent. The message identifier stored in the permanent storage is used to indicate to the source object which messages to resend. As checkpoints are taken the message identifier stored is updated to reflect the last message received.

Like the previous case, the re-execution of methods will result in method calls being recreated. In this case there is no local cache to retrieve results from. The method calls will be resent but on the receiver's side their local cache is checked for replicas. If the method has already been completed then results can simply be returned since results have been cached there as well.

When an object is checkpointed the local caches may be cleaned slightly. Any results of outgoing messages may be removed. Additionally messages may be sent to objects that have called the object to let them remove any logs of method calls on the object, as they will no longer be required.

5 Sample Application

This section briefly describes a sample application that one of our colleagues has implemented using our G2:P2P framework. The sample application uses a parallel genetic algorithm to solve the travelling salesman problem. The application uses remotely activated objects to represent islands containing a population of individuals, each of which is a possible solution to the travelling salesman problem in question. Each island independently reproduces, mutates and tests these individuals in an attempt to find better solutions. Periodically the best individuals are exchanged with neighbouring islands.

G2:P2P uses a standard .NET Remoting configuration file to indicate that the `TSP.Island` objects should be activated remotely (ie. on the G2:P2P network):

```
<configuration>
  <system.runtime.remoting>
    <channels>
      <channel id="G2P2P"
        type="G2P2P.Channel,G2P2P" />
    </channels>
    <application>
      <channels>
        <channel ref="G2P2P" />
      </channels>
      <client url="G2P2P://plas.qut.edu/">
        <activated type="TSP.Island,TSP" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

To make use of this file it must be loaded by the Remoting infrastructure through a call to `System.Runtime.Remoting.RemotingConfiguration.Configure`. Objects are then simply created through a standard call to the `new` operator:

```
Island[] islands = new Island[NumIslands];
for (int i = 0; i < NumIslands; i++)
  islands[i] = new Island();
```

Generally method calls on remote objects should be executed asynchronously to allow multiple methods to be called in parallel. .NET includes a standard method of executing methods asynchronously:

```
foreach (Island i in islands}
{
  StartDelegate d =
        new StartDelegate(i.Start);
  d.BeginInvoke(
        new AsyncCallback(GetResult), d);
}
```

This executes the `Start` method on each island. As the results of this operation are returned the `GetResult` method will be called by the framework. The actual implementation of the method requires no changes to support asynchronous invocation. This asynchronous invocation is not specific to Remoting and is often used even in standard .NET applications.

To allow an island to communicate it needs a reference to the other islands. References may be passed just as any other object.

```
foreach (Island i in islands}
  islands.Peers = islands;
```

Actual communication is then performed through method calls just as the initial client communicates.

```
foreach (Island peer in peers)
  peer.TransferBestMatches(matches);
```

The above examples are simplified for clarity. The actual application uses a generic GA framework which is also being used for research into Internet based parallel genetic algorithms.

6 Related Work

Initial attempts at sharing computing cycles, such as SETI@home (SETI@home 1999-2004) and GIMPS (GIMPS Project 2004), targeted specific applications. After their success, research projects quickly formed to develop generic frameworks (Cappello et al. 1997, Baratloo, Karaul, Kedem & Wyckoff 1996), however, like the single application systems they used a client/server based architecture. Scalability issues soon prompted more complex architectures such as the hierarchical model developed in Javelin++ (Neary, Brydon, Kmiec, Rollins & Cappello 2000). The DREAM project (Arenas, Collet, A. E. Eiben, Merelo, Paechter, Preuβ & Schoenauer 2002) has moved to a fully decentralised model, however, it is restricted to the specific application domain of evolutionary computing and relies on certain assumptions from that domain.

Programming for these systems is generally performed by creating tasks which run in parallel on volunteer machines. Javelin and Charlotte took advantage Java's cross-platform properties to ease development and deployment of applications. Tasks on these systems are developed by implementing specific interfaces. G2 (Kelly et al. June 2002) used a web services abstraction to provide a more familiar abstraction for implementing and calling tasks. G2:Remoting (Kelly & Frische 2003) also adapted the G2 framework to provide a distributed object model.

Significant research has been performed into structured P2P networks. There are four main DHT based projects Pastry (Rowstron & Druschel 2001a), Tapestry (Zhao et al. January 2004), Chord (Stoica et al. 2001) and CAN (Ratnasamy et al. 2000). All four can offer object lookup in $O(\log N)$ hops, however, the major difference that sets Pastry and Tapestry apart is their use of locality in the routing algorithm. A wide variety of projects use these networks as a basis for applications including file storage (Rowstron & Druschel 2001b, Rhea, Eaton, Geels, Weathersppon, Zhao & Kubiatowicz March 2003), publish/subscribe (Rowstron, Kermarrec, Castro & Druschel 2001) and Spam filtering (Zhou, Zhuang, Zhao, Huang, Joseph & Kubiatowicz 2003). Further efforts have also been made at improving the performance of DHT networks. Brocade (Zhao, Duan, Joseph & Kubiatowicz 2002) recognises the different network characteristics of individual nodes and improves bandwidth usage by elevating nodes with higher bandwidth to "supernode" status.

7 Future Work

Currently the object distribution system of G2:P2P does not attempt to minimise communication costs by allocating communicating objects physically near each other. We are looking at methods of adjusting object distribution to improve communication between objects – either placing them on the same volunteer or on volunteers in the same subnet. Providing this mechanism requires adjustments to the object distribution scheme, programming model and volunteer identification. This is a considerable challenge, though it has the potential of impressive benefits.

The current programming model is somewhat restrictive in requiring remote method calls to be short lived. Future work will aim to relax this restriction and provide more control to the programmer during migration and checkpointing.

8 Conclusion

G2:P2P uses a fully decentralised network to provide a scalable, robust cycle-stealing framework. It improves on existing frameworks by providing a distributed object programming model and allowing direct communication between these objects. A robust fault tolerance scheme has been developed to ensure application correctness. The scheme is adaptable depending on the requirements of the application developer. It is entirely automated requiring no explicit checkpointing from the application. The framework is provided as an extension to the .NET Remoting infrastructure allowing an easy migration path for existing distributed programmers.

References

Arenas, M. G., Collet, P., A. E. Eiben, M. J., Merelo, J. J., Paechter, B., Preuβ, M. & Schoenauer, M. (2002), 'A framework for distributed evolutionary algorithms', *Lecture Notes in Computer Science* **2439**, 665–675.

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

Cappello, P., Christiansen, B. O., Ionescu, M. F., Neary, M. O., Schauser, K. E. & Wu, D. (1997), Javelin: Internet-Based Parallel Computing Using Java, *in* G. C. Fox & W. Li, eds, 'ACM Workshop on Java for Science and Engineering Computation'.

Cooney, D. & Roe, P. (2003), Experiences with a mobile process orientated middleware, *in* 'AusWeb '03, Gold Coast, Australia'.
URL: *http://ausweb.scu.edu.au/aw04/papers/-refereed/cooney/paper.html*

GIMPS Project (2004), 'Mersenne prime search'.
URL: *http://www.mersenne.org/prime.htm*

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

Kelly, W. & Frische, L. (2003), G2 remoting: A cycle stealing framework based on .net remoting, *in* '2003 APAC Conference on Advanced Computing, Grid Applications and eResearch, Gold Coast'.

Kelly, W., Roe, P. & Sumitomo, J. (June 2002), G2: A grid middleware for cycle donation using .net, *in* '2002 International Conference on Parallel and Distributed Processing Techniques and Applications'.

Luby, M. G., Mitzenmacher, M., Shokrollahi, M. A., Spielman, D. A. & Stemann, V. (1997), Practical loss-resilient codes, *in* 'Proceedings of the twenty-ninth annual ACM symposium on Theory of computing', pp. 150–159.

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

Obermeyer, P. & Hawkins, J. (2001), 'Microsoft .net remoting: A technical overview'.
URL: *http://msdn.microsoft.com/library/-default.asp?url=/% -library/en-us/dndotnet/-html/hawkremoting.asp*

Philippsen, M. & Zenger, M. (1997), 'JavaParty — transparent remote objects in Java', *Concurrency: Practice and Experience* **9**(11), 1225–1242.

Plank, J. S. (1997), 'A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems', *Software, Practice and Experience* **27**(9), 995–1012.

Ratnasamy, S., Francis, P., Handley, M., Karp, R. & Shenker, S. (2000), A scalable content addressable network, Technical Report TR-00-010, Berkeley, CA.

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

Roe, P. & Szyperski, C. (February 1998), Gardens: An integrated programming language and system for parallel programming across networks of workstations, *in* '21st Australasian Computer Science Conference (ACSC'98), Perth, Australia', Springer.

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

Rowstron, A. & Druschel, P. (2001*b*), Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, *in* '18th ACM Symposium on Operating Systems Principles (SOSP'01)', pp. 188–201.

Rowstron, A. I. T., Kermarrec, A.-M., Castro, M. & Druschel, P. (2001), SCRIBE: The design of a large-scale event notification infrastructure, *in* 'Networked Group Communication', pp. 30–43.

Ryu, K. D. & Hollingsworth, J. K. (2000), 'Exploiting fine-grained idle periods in networks of workstations', *IEEE Transactions on Parallel and Distributed Systems* **11**(7), 683–??

SETI@home (1999-2004), 'The search for extraterrestrial intelligence at home'.
URL: *http://setiathome.berkeley.edu*

Sharman Networks (2004), 'Kazaa'.
URL: *http://www.kazaa.com*

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. & Balakrishnan, H. (2001), 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', ACM Press, pp. 149–160.

Sun Microsystems (2004), 'Java remote method invocation - distributed computing for java'.
URL: *http://java.sun.com/products/jdk/rmi/-reference/whitepapers/javarmi.html*

Zhao, B. Y., Duan, Y., Joseph, L. H. A. D. & Kubiatowicz, J. D. (2002), Brocade: Landmark routing on overlay networks, *in* '1st International Workshop on Peer-to-Peer Systems (IPTPS), March 2002'.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D. & Kubiatowicz, J. D. (January 2004), 'Tapestry: A resilient global-scale overlay for service deployment', **22**(1).

Zhou, F., Zhuang, L., Zhao, B. Y., Huang, L., Joseph, A. D. & Kubiatowicz, J. (2003), Approximate object location and spam filtering on peer-to-peer systems, *in* 'Middleware, June 2003'.