Multi-consistency in Peer-to-Peer computing [*]

Kevin Glynn
glynn@info.ucl.ac.be

Peter Van Roy
pvr@info.ucl.ac.be

28th February, 2005

Abstract

This technical report examines the applicability of Distributed Shared Memories to the problem of building scalable, collaborative, distributed applications on Peer-to-Peer networks. We identify certain relaxed consistency protocols that improve scalability while maintaining an acceptable programming model.

We present a prototypical implementation of a Causally Consistent Distributed Store which we have developed with P2PKit (a service oriented Peer-to-Peer programming environment). We describe how the store is adapted to support nodes joining and leaving the store at any time and identify improvements which will make it more scalable and reliable.

[*]The work in this paper was partially funded by European project PEPITO IST-2001-33234.

Contents

1 Introduction

In this research report[1] we describe part of our ongoing research into suitable programming models for developing scalable, distributed applications on peer-to-peer networks. Our goal is to develop applications that scale smoothly as users (and additional computing resources) are added, or removed, from the system.

Research into structured peer-to-peer networks based on Distributed Hash Tables (DHTs) such as Chord [21], CAN [18], Pastry [20], and Tapestry [13], has made a significant contribution at the level of the underlying communication infrastructure. These networks self-optimise to maintain routing guarantees, so that the number of physical connections a node manages and the maximum path between nodes grow only logarithmically with respect to the total network size.

Within the PEPITO project we have improved upon these systems to implement more flexible peer-to-peer networks with better connectivity, DKS [4], S-Chord [16], Tango [7], and P2PS [17, 8]. Additionally, the PEPITO implementations employ correction-on-use protocols which mean that the network self-organisation happens as a consequence of application messages, with little additional overhead [10].

However, to date, very little research has been carried out into suitable scalable programming models for building general, distributed applications on these architectures.

When considering a scalable method for building distributed applications we have been inspired by the organisational structures prevalent in society. Effective social networks that scale to a large number of members can be categorised into a small number of models. For example, in practice a social organisation in which a large number of members have complex, unpredictable, interactions with a lot of other members at the same time is rare. In practical social networks small sub-groups form *clusters*, i.e., within a sub-group we see complex, unpredictable interactions, but interactions between clusters are relatively rare.

We hypothesise that the following three models are a useful first approximation to the structures of practical distributed applications. We do not pretend to have made an exhaustive analysis of the appropriate structures in distributed systems, but they are in line with our experience of building and reviewing such applications. We use these models to evaluate the effectiveness of lower-level mechanisms and services in supporting these structures. Once we have more experience with building scalable applications we can review these assumptions.

The Chat Room Model Chat Room applications are highly concurrent, but structured as a large number of communicating groups. Each communicating group is sharing state (working in the same chat rooms) and tends to be small. Groups are highly dynamic and unpredictable, members join and leave often. Groups may have a short lifetime.

Interactions between groups is also unpredictable, but limited. As the application scales up the number of communicating groups increases, but the size of the groups themselves grows much more slowly.

The Stadium Model In the Stadium model there is a small number of actively collaborating groups. There are a large number of observers. These observers are interested in the state of the active groups. Observers may occasionally interact with the active groups.

The Structured Model In the structured model the distributed application is broken down into highly structured small tasks, each operating on a small part of the problem. Each component has a small

[1] Also Deliverable 2.1 for the PEPITO project (`http://www.sics.se/pepito`).

number of immediate neighbours and communications between components are static and highly predictable.

Examples of the structured model are parallel matrix computations, climate applications where tasks are responsible for a particular area and only communicate with their immediate neighbours, hierarchically structured applications.

We are particularly interested in applications which follow the Chat Room model. These applications consist of a large number of collaborating participants. Scalability is much more important than absolute performance. As an example of the Chat Room model consider an application to manage on-line business meetings. The application supports many concurrent meetings. Participants in a meeting collaborate to share and develop shared documents, white boards, etc. Often social protocols within the meeting help to reduce inconsistencies due to two participants updating the same item concurrently, and help to resolve these inconsistencies when they do occur.

In order for an application to scale smoothly to hundreds or thousands of nodes its distributed instances must be able to make sensible decisions on the basis of incomplete, indeed inconsistent, knowledge of the global state of the application. In general there will no longer be a single consistent global application state.

We have found that software implemented Distributed Shared Memory (DSM) systems address similar problems and we want to apply the lessons they have learnt to peer-to-peer systems. Previous research in DSMs has focused on their applicability for solving highly parallel scientific problems. Except for some niche scientific applications, DSMs are often considered to have failed as a technique for parallel, high performance computing, [22] (Section 1.4) . But we believe that further investigation will show that they are a better fit for concurrent, collaborative applications. Such applications can be built so that they tolerate a certain amount of inconsistency.

DSM systems provide the illusion of shared memory across processors that do not physically share memory. DSMs can make it straightforward to port multi-threaded applications which communicate through reading / writing to memory to distributed networks of computers. The network is completely transparent to the application, the DSM delays memory reads and writes, and copies data between processors to maintain a consistent view of the shared memory by the threads. Although initial systems provided *sequential consistency*, where all threads in the distributed system saw the same order of reads and writes to the DSM, it was quickly realised that these could never offer sufficient performance and scalability.

Since then a number of interesting *relaxed* consistency models have been proposed. These improve the scalability of DSM by allowing nodes, under certain circumstances, to see reads and writes to the DSM in different orders. By using appropriate programming styles it is still possible for the programmer to consider the shared store as sequentially consistent. These relaxed models are much more scalable.

To investigate these ideas we have built a distributed, shared object store on top of our peer-to-peer application development toolkit, P2PKit [11, 12]. Our object store provides *causal consistency* for reads and writes to objects in the store.

In Section 2 we define Distributed Shared Memory and describe some of the more commonly implemented consistency models. In Section 3 we describe our implementation of Distributed Causal Store (DCS), a causally consistent distributed store on a peer-to-peer network. In Section 4 we discuss some improvements to our implementation and its suitability for scalable distributed applications. Finally, in Section 5 we conclude and discuss future work.

2 Distributed Shared Memory

For a thorough overview of software distributed shared memory see Chapter 6 of Tanenbaum and van Steen [22]. In this section we briefly introduce the main classes of coherence. In the next section we describe an implementation of causal consistency in a peer-to-peer system.

DSMs come in two forms, address-based and object-based.

An address-based DSM, such as TreadMarks [5], looks to the application exactly like local memory. Programs operate via pointers into an array of memory locations. This is necessary to support languages, such as C and C++, which allow pointer arithmetic. Address-based DSMs often maintain consistency amongst copies at the page level, they use virtual memory translation tables to trap reads and writes of locations in the shared memory and take appropriate action to maintain consistency. Page-based DSMs suffer from excessive communications due to *false sharing*, where unrelated data items appear in the same page - although this can be reduced by only distributing changes to pages, rather than the page itself.

An object-based DSM, such as DOSA [14] or Orca [6], views the shared memory as a mapping from identifiers to objects. Consistency is maintained amongst objects. These systems are appropriate for modern, “safe” languages, such as Java, which do not allow pointers directly to memory locations.

In this paper, we are using the term DSM to refer to both address-based and object-based shared stores.

As well as a shared store, DSMs provide the usual synchronisation facilities, e.g., semaphores, locks, monitors, barriers, and so on, so that an application can control its concurrent access to data items.

Sequentially Consistent In a sequentially consistent DSM all processes in the distributed system agree on a common order for all reads and writes. In this way, all processes are guaranteed to be seeing the same contents. Reads and writes to a copy of the store may be delayed by the implementation in order to maintain sequential consistency.

Sequentially consistent DSMs are simple for programmers. They behave like the memory in shared memory multiprocessors. However, they are expensive to implement. Processes updating the memory must synchronise with other processes. There are many schemes for implementing this synchronisation, for example:

- Before updating a data item all other copies of the data item must first be invalidated. When a remote process wishes to use the data item it will have to first fetch a fresh copy.
- Each data item has a single owner. Updates must be performed on the owner. Updates are flushed to copies.
- Each data has a single owner, if a process wishes to update an item it must transfer ownership to itself. Updates are flushed to copies.

Sequential consistency is expensive to implement and scalable applications must be carefully designed.

FIFO Consistency FIFO Consistency (also called PRAM Consistency) is much weaker than Sequential Consistency. It only guarantees that writes performed by a single process are seen in the same order by all processes (i.e., in the order the process made them). There is no guaranteed ordering between writes from different processes, each node may receive a different interleaving of writes from other nodes. For this reason, each node’s copy of the DSM may be different.

FIFO Consistency is simple, and cheap to implement, but not suitable for most applications.

Weak Consistency Weak Consistency takes advantage of the synchronisation primitives (e.g., locks and barriers) which applications must use to guarantee exclusive access to critical regions to reduce the amount of consistency that must be guaranteed. For example, if processes in a distributed application always take a lock before accessing a data item then while the lock is held it is not necessary to synchronise updates across the DSM. It is only necessary to ensure that data is up to date before the lock is granted, or before it is released.

As long as an application accesses data only while it has the required locks the store will behave as if it is sequentially consistent.

Orca is an object-based language that provides a weakly consistent shared store. When an application calls a method Orca first ensures the local copy of the data is up to date and then locks the object’s data (in shared mode for a read only method and exclusive mode for a updating method) for the duration of the method call.

Causally Consistent In a Causally Consistent DSM [2], all processes agree on the relative ordering of updates that are *causally* related. An update is causally related to the data items that were read in order to make that update. Updates by a process to its local store are only propagated to other copies of the store once that copy is at least as current as the store in which the update was originally made.

For example (taken from [22], Section 6.2.3), suppose process P_1 writes to a variable, x, then process P_2 reads the variable x and writes a variable y. The value written to y potentially depends on the value of x. In a causally consistent store no process will see this new value of y until they have also seen the value of x it depended on.

However, writes that are not causally related may be seen in different orders on different stores so stores are not identical.

Causally consistent stores have the nice property that reads and writes are never blocked, they can always take place on the local store immediately. However, writes that are propagated to remote stores must be queued until the remote store is sufficiently up to date to apply the write.

Causally consistency appears to be a good fit for collaborative applications which fit the Chat Room model. It is easy to implement and it mimics the expectations of users. Imagine a collaborative drawing application in which multiple users are drawing a causally consistent shared image. If one user draws a house and then another user, after seeing the house, draws windows on the house then all users should see the windows on top of the image of the house. However, if two users draw in the same part of the image at the same time then there is no guarantee about the order in which the images will be seen by other users.

3 DCS: Causal Consistency for a Peer-to-Peer System

We have made publically available DCS, a Distributed Shared Memory prototype written in Oz [1] and built with the P2PKit toolkit [11]. The latest version is available from

`http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/CausalDSM`

Our prototype DCS provides a causally consistent DSM and is based on previously published algorithms [2]. However, previous work assumes the distributed shared memory is running on a fixed network of computers. We adapt the algorithms to work in a peer-to-peer environment where nodes may enter and leave the system at any time. In the next section we examine the scalability of our implementation and suggest improvements in the light of our targeted application architectures.

Our implementation is provided as a P2PKit service running on a P2PS network [17]. P2PKit is a decentralized service architecture which provides a high-level API for deploying, upgrading, and communicating with services running on structured peer-to-peer networks. A technical report, presentation, code, and examples, are available from the P2PKit web site [12].

First we describe the normal execution of DCS nodes, and then we describe how nodes initiate, join, and leave a DCS.

A Distributed Store Supporting Causal Consistency In this section we assume that we have a P2PS network where each node has an instance of the shared store. Each node has a timestamp, an integer that is incremented each time it writes a value to its copy of the shared store.

To provide causal consistency each node also has a *vector timestamp* [15, 9]. The vector timestamp has timestamp values for other nodes in the system, the values say which writes from that node have already been applied to this DCS instance.

Figure 1 shows the code for reading, writing and atomic exchange for the distributed shared store.

The `Store` is a dictionary, so entries in the DCS are indexed by dictionary keys. Since the local instance is always kept causally consistent, reads and writes to the local instance happen immediately - they are never blocked. A write increments the local node's timestamp in the vector timestamp (a dictionary called `TSVec`), updates the `Store`, and broadcasts an update message to the other nodes. The update message containing this node's id, the key and value for the update, and this node's new vector timestamp. An exchange is the same as a write, except that we perform an atomic exchange operation on the store.

Figure 2 gives the code that processes the incoming updates from other nodes. It is a thread that processes incoming messages as they arrive.

An update is applicable to the local store (function `Applicable`) if it is the next update to be applied from the sending node (`RemTSVec.Id == {Dictionary.condGet TSVec Id 0}+1`) and for all other nodes our store is at least as recent as the store the sending node had at the time of the update (`RemTSVec.Id =< {Dictionary.condGet TSVec Id 0}`).

If the incoming update is not currently applicable then we insert it to our list of pending updates (`ToDo`). The list of pending updates is a priority queue, updates are stored in an applicability order so that when we can update our store we can make a single pass through the pending list applying all the updates that are now ready to be applied.

If the incoming update is applicable then we apply it to our store and make a pass through the pending updates queue, removing and applying all updates that are now applicable.

```
%% Variables:
%% Store is the local instance of the DCS (a dictionary)
%% TSVec is this node’s vector timestamp  (a dictionary)
%% DCSService is the name of this DCS group (an atom)
%% NodeId is this node’s id

%% DCS is a handler returned to users of the DCS. User operations
%% are read / write / (atomic) exchange
proc {DCS Request}
    case Request of
       read(Key Val) then
       %% Return ’undefined’ if Location is empty
       {Dictionary.condGet Store Key undefined Val}
    [] write(Key Val) then OTS NTS in
       %% Increment this node’s timestamp
       {Dictionary.exchange TSVec NodeId OTS NTS}
       {Wait OTS}     %% Block until we have unique access
       {Dictionary.put Store Key Val}
       NTS = OTS+1    %% And free any concurrent write
       {P2PKitPeer.broadcast DCSService
         updLoc(src: NodeId key:Key val:Val
                tsvec:{Dictionary.toRecord upd TSVec})}
    [] exchange(Key OldVal NewVal) then OTS NTS UpdM in
       %% Increment this node’s timestamp
       {Dictionary.exchange TSVec NodeId OTS NTS}
       {Wait OTS}     %% Block until we have unique access
       {Dictionary.condExchange Store Key undefined OldVal NewVal}
       NTS = OTS+1    %% And free any concurrent write
       {P2PKitPeer.broadcast DCSService
         updLoc(src: NodeId key:Key val:NewVal
                tsvec:{Dictionary.toRecord upd TSVec})}
    end
end
```

Figure 1: Causal DCS Operations on Node `NodeId`.

```
%% Variables:
%% Store is the local instance of the DCS (a dictionary)
%% TSVec is this node’s vector timestamp  (a dictionary)
%% UpdStream is the stream of writes from other nodes
%% ToDo is a priority queue of pending updates
%%      received from remote nodes

%% We apply the writes as soon as they are consistent
thread
   %% Is this update applicable?
   fun {Applicable RemId RemTSVec}
      {Record.allInd RemTSVec
       fun {$ Id _}
         if Id == RemId then
           %% Is this the next write from Remote Node
           RemTSVec.Id == {Dictionary.condGet TSVec Id 0}+1
         else
           %% Are we up to date with the view from Remote Node
           RemTSVec.Id =< {Dictionary.condGet TSVec Id 0}
         end
       end}
   end
in
   for Msg in UpdStream do
      if {Applicable Msg.src Msg.tsvec} then
        %% This record can be applied
        {Dictionary.put Store Msg.key Msg.val}
        {Dictionary.put TSVec Msg.src Msg.tsvec.(Msg.src)}
        %% Pass through ToDo priority queue applying those
        %% records that are now applicable
        ToDo := {ApplyPending ToDo}
      else
        %% Insert update at appropriate place in ToDo priority queue
        ToDo := {Insert Msg Todo}
      end
   end
end
```

Figure 2: Applying Updates on Node `NodeId`.

It is easy to see that updates received from remote nodes are not applied to the local store until it is at least as current as the store where the update was originally made, thus maintaining causal consistency.

Starting and Joining a DCS Installing the DCS service on the nodes in a `P2PS` network causes a DCS instance to become available on each node. Multiple, non-sharing, DCSs can be available at the same time in a network if each runs with a different service name. For simplicity we do not discuss this further. P2PKit supports the lazy installation of services. When a node receives a message for a service it does not have it automatically downloads the service from other nodes in the network. Thus for a node to join a DCS it is sufficient to connect it to the peer-to-peer network, when it receives a local request for the DCS or receives update messages from other nodes it will start the DCS service.

When the DCS service starts on a node it does not know whether the shared store already exists, in which case it should take a copy from an existing node, or if it must create a fresh shared store. The following, completely decentralized, protocol handles these two cases.

- All the nodes in a P2PS network form a circle through their successor links. When the DCS service starts up on a node it sets its state to `<init>` and sends a `getDCSStore` message to its successor node asking for a copy of the DCS. This message contains the address of the originating node and a Time-To-Live parameter. The Time-To-Live parameter should be set to a number greater than the number of nodes in the system (for example, the maximum network size).
- While the node is in the `<init>` state it queues all incoming update messages it receives from other nodes. These pending updates will be processed once it has created its local store.
- If the successor receiving the `getDCSStore` message is also in the `<init>` state then it decrements the Time-To-Live counter and passes the message on to its successor.
- If the originating node gets its `getDCSStore` message back then it knows that the message has gone through all nodes in the system and they were all in state `<init>`. Therefore, this is a fresh DCS system and the originating node creates an empty `Store` and a local timestamp `TSVec`, with the entry for this node set to `0`. The node's state is set to `<ok>` and any pending update messages are processed as described in the previous section. The node is then ready for normal operation.
- If the Time-To-Live counter becomes zero then the originating node must have died (otherwise, it would have received the message and created a fresh store). In this case, the receiving node simply drops the message.
- If a node is in state `<ok>` and it receives a `getDCSStore` message it sends the originating node a copy of its store, a copy of its vector timestamp, and a copy of its pending updates. The originating node initializes its store, vector timestamp, and pending updates with these values, changes its state to `<ok>` and processes any pending update messages. The node is then ready for normal operation.

In this way we can start new nodes at any time, and they automatically join or create a DCS as appropriate.

Nodes Leaving the Network No special action is required when nodes leave the network (either by an explicit `leave` operation or implicitly by dying).

However, it is important that all the writes sent by the leaving node are received by all other nodes. Consider two other nodes in the system, one which has received the update information (A) and one that has not (B). Node A will update its vector timestamp to show the update from the dead node. Now node B will never be able to apply updates from node A because it will leave the update pending until its timestamp entry for the dead node matches the entry node A has.

This scenario will happen whenever broadcasts are not received at all nodes. The algorithm assumes that broadcasts are reliable. That is, a broadcast message always arrives at all nodes in the system. This is not currently provided by P2PS, although it is planned to be available in the future. Providing reliable broadcast from a node that dies is problematic because that node is not able to coordinate the broadcast delivery protocol.

In our prototype we currently assume that broadcasts are reliable. However in Section 4 we discuss an extension to our implementation which would allow it to work even if broadcasts were not reliable.

4 Discussion

The implementation of Causal Consistency in the previous section is adapted to a peer-to-peer environment where nodes contributing to the shared memory can join and leave at any time.

Since the DCS maintains causally consistent local stores then applications are never blocked when reading and writing to the store. However, it is important that updates get applied to remote stores in a timely manner. If this does not happen then applications will suffer the problem of concurrent writes. In addition, calls to sunchronisation primitives will be blocked until the participant's stores are all up to date. We must, therefore, consider the scalability of our implementation, how does increasing numbers of nodes affect the speed with which updates are applied remotely.

It is easy to see that our implementation is not scalable. Each write by a node to its local store results in a message broadcast to all other nodes in the network. This message contains the node's vector timestamp, so its length is proportional to the number of nodes sharing the DCS. Clearly as the number of nodes sharing the DCS increases the quantity and size of messages being distributed will swamp the network's communications infrastructure and each node will be spending most of its time processing the incoming write notifications.

To make a scalable implementation we can take advantage of the structure of the applications.

If an application consists of lots of concurrent components each operating on only a small subset of the DCS (the Chat Rooms model) then we can split the store into areas and control each area with a separate peer-to-peer *group*. A P2PS group is a subset of the nodes in a network. Nodes can join and leave groups at any time, and broadcast to all members of the group.

Before accessing a region a node must join the group controlling that area. Once the node is in the group it can refresh its knowledge of the region and read and write to the region as described in the previous section. However, now messages need only be broadcast to other members of the group and the vector timestamp will only contain entries for nodes in the group.

P2PS supports efficient multicasting of messages to an explicit list of nodes, but does not yet support groups directly (this is planned).

In the Stadium model the actively collaborating nodes can maintain a strong degree of consistency by using a traditional DSM. However, it is likely to be beneficial to combine this with other protocols for other participants, such as observers, who do not have such high consistency requirements. A number of papers, e.g., [3, 19], investigate methods for efficiently supporting hierarchies of clients that have differing consistency requirements.

We have experimented with our DCS prototype on nodes running in the PlanetLab environment:

`http://www.planet-lab.org/`

PlanetLab is a highly dynamic worldwide infrastructure that is sponsored by Intel. It consists of over 500 machines, hosted by over 230 sites, and spanning over 25 countries. Because its operation is reliant on the goodwill and responsiveness of local system and network administrators, it has frequent failures and network problems.

In this environment we see many lost messages in P2PS and we are looking at methods to recover from lost messages. We monitor the pending updates at a node and if a pending update remains inapplicable for a 'long' time we send requests to the nodes it is blocked on to get the missing updates. If this fails we can, as a last resort, re-initialize the store by copying it from another node in the DCS.

5 Conclusion and Further Work

In this report we have examined systems that support scalable distributed applications by controlling the consistency of the application's global state. Through well defined consistency protocols we can improve the scalability of applications without sacrificing program correctness or complexity.

We have implemented DCS, a prototypical Causally Consistent Data Store for a peer-to-peer platform, and extended previous consistency algorithms to work in an environment where nodes may join and leave at any time.

We have seen that adding support for groups and reliable broadcast to our peer-to-peer platform will allow us to make DCS more scalable and reliable.

We intend to use DCS as a component to build a variety of collaborative applications and deploy them on various networks, including PlanetLab and Evergrow (`http://www.evergrow.org/`).

Acknowledgements

Many thanks to Valentin Mesaros and Raphaël Collet for their careful reading of this report.

References

[1] Mozart Consortium, June 2004. Mozart Programming System Release 1.3.0. Available at `http://www.mozart-oz.org`.

[2] Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.

[3] Mustaque Ahamad and Rammohan Kordale. Scalable consistency protocols for distributed services. *IEEE Transaction on Parallel and Distributed Systems*, 10(9):888–903, 1999.

[4] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *Proc. of the 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems – CCGRID2003*, Tokyo, Japan, May 2003.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[6] Henri E. Bal and M. Frans Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 162–177. ACM Press, 1993.

[7] Bruno Carton and Valentin Mesaros. Improving the scalability of loarithmic-degree DHT-based peer-to-peer networks. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 1060 – 1067. Springer, September 2004.

[8] Bruno Carton and Valentin Mesaros. P2PS: Peer-to-Peer System Library, June 2004. Universtité catholique de Louvain, and CETIC, Belgium. Available at `http://www.mozart-oz.org/mogul/info/cetic_ucl/p2ps.html`.

[9] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[10] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Low-bandwidth topology maintenance for robustness in structured overlay networks. In *38^{th} International HICSS Conference*. IEEE Computer Society, January 2005.

[11] Kevin Glynn. Extending the Oz language for peer-to-peer computing. PEPITO project deliverable D3.7, IST-2001-33234, December 2004. Available at http://www.sics.se/pepito/deliverables.html.

[12] Kevin Glynn. P2PKit Library, October 2004. Available at `http://renoir.info.ucl.ac.be/twiki/bin/view/INGI/P2PKit`.

[13] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 41–52. ACM Press, 2002.

[14] Y. Charlie Hu, Weimin Yu, Alan Cox, Dan Wallach, and Willy Zwaenepoel. Run-time support for distributed sharing in safe languages. *ACM Trans. Comput. Syst.*, 21(1):1–35, 2003.

[15] Friedemann Mattern. Virtual time and global states of distributed systems. In *proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1988.

[16] Valentin Mesaros, Bruno Carton, and Peter Van Roy. S-Chord: Using symmetry to improve lookup efficiency in Chord. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, Las Vegas, Nevada, USA, June 2003.

[17] Valentin Mesaros, Bruno Carton, and Peter Van Roy. P2PS: Peer-to-peer development platform for Mozart. In Peter Van Roy, editor, *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

[18] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[19] Michel Raynal, Gerard Thia-Kime, and Mustaque Ahamad. An adaptive protocol for implementing causally consistent distributed services. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1998.

[20] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[21] Ion Stoica, Rovert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for Internet applications. In *ACM SIGCOMM*, August 2001.

[22] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, international edition, 2002.