Electrical Engineering and Computer Science   College of Engineering and Computer Science

10-1-2010

p2pstm: A Peer-to-Peer Software Transactional Memory

Phil Pratt-Szeliga
Syracuse University, pcpratts@syr.edu

Jim Fawcett
Syracuse University, jfawcett@twcny.rr.com

Department of Electrical Engineering and Computer Science

Technical Report

SYR-EECS-2010-03   Oct. 1, 2010

p2pstm: A Peer-to-Peer Software Transactional Memory

Phil Pratt-Szeliga   pcpratts@syr.edu
Dr. Jim Fawcett   jfawcett@twcny.rr.com

ABSTRACT: Peer-to-Peer network topologies are such that there is no centralized server in a distributed system but rather each peer is part server and part client. This idea was originally popularized with peer-to-peer file sharing. If one were to distribute computation on a peer-to-peer network topology, consistency of shared data among peers is a problem. Software Transactional Memory (STM) is a lock free mechanism of assuring consistency of memory among threads of execution that has favorable performance over locked methods when the number of threads is large. This paper presents a method to use STM methods with a peer-to-peer network architecture that doesn't use locks. This is accomplished by using peer-to-peer search using two keywords: one for the variable name and the other for the most recent version of the variable. The technique of using Hilbert Space Filling Curves to map from the keyword space to the identifier space without flooding from Squid [2] is used. Deadlocks are prevented by executing reads and writes with a defined partial order that is transparent to the developer. As an example of using the methods presented, a blocking queue is built from the primitives provided.

KEYWORDS: Networks, Peer-to-Peer, Concurrency, Software Transactional Memory

p2pstm: A Peer-to-Peer Software Transactional Memory

Phil Pratt-Szeliga

Department of Computer Science and Electrical
Engineering
Syracuse University
Syracuse, NY, USA
pcpratts@syr.edu

Dr. Jim Fawcett

Department of Computer Science and Electrical
Engineering
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

Abstract— Peer-to-Peer network topologies are such that there is no centralized server in a distributed system but rather each peer is part server and part client. This idea was originally popularized with peer-to-peer file sharing. If one were to distribute computation on a peer-to-peer network topology, consistency of shared data among peers is a problem. Software Transactional Memory (STM) is a lock free mechanism of assuring consistency of memory among threads of execution that has favorable performance over locked methods when the number of threads is large. This paper presents a method to use STM methods with a peer-to-peer network architecture that doesn't use locks. This is accomplished by using peer-to-peer search using two keywords: one for the variable name and the other for the most recent version of the variable. The technique of using Hilbert Space Filling Curves to map from the keyword space to the identifier space without flooding from Squid [2] is used. Deadlocks are prevented by executing reads and writes with a defined partial order that is transparent to the developer. Liveness is ensured by putting fairness into a truncated exponential backoff algorithm. As an example of using the methods presented, a blocking queue is built from the primitives provided. Lastly, a performance measurement of the system created is provided.

Keywords - Networks, Peer-to-Peer, Concurrency, Software Transactional Memory

I. INTRODUCTION

Software transactional memory (STM) is a new approach to maintaining consistency of shared data without locks. In STM, updates to shared data are assumed to be executed mutually exclusive of all other updates to the same shared data. Then, when all of the updates have been completed, the mutual exclusion property is checked by the writer and the updates are restarted if another thread wrote at the same time. This paper is about an algorithm for a distributed STM that does not rely on a centralized server. We focus on not relying on a centralized server because performance problems arise when scaling to Internet sized distributed applications.

We build our system on top of the peer-to-peer search techniques from the Squid [2] paper. Squid uses Hilbert Space Filling Curves to hierarchically search peer-to-peer network overlays such as Pastry [3] without the use of flooding. An important feature of Squid is that numerical ranges can be searched (again without flooding). We create a Versioned Variable primitive that is built on top of the search capabilities of Squid and then build an STM from there.

The rest of this paper is organized as follows: Section 2 provides background information used to help understand the contributions, Section 3 shows the architecture of our system, Section 4 gives an example of using our architecture. The performance of our system is discussed in Section 5. Section 6 summarizes the related work and Section 7 provides the references.

II. BACKGROUND INFORMATION

This section describes 1) FreePastry, a peer-to-peer network overlay that allows network communication without a centralized server, 2) Squid, a peer-to-peer search methodology that does not use flooding and 3) The concept of Software Transactional Memory, a lock free method of maintaining safety properties in shared memory systems.

A. FreePastry

FreePastry is an open source Java implementation of the Pastry [3] peer-to-peer network overlay. Peer-to-peer network overlays allow peers to communicate in, for instance, a video conferencing application such as Skype, without the use of a centralized server. Connection to the network can be achieved by knowing the IP address of any of the peers in the network. Once connected, messages can be routed to any peer in the network using information in a peer-local routing table that is maintained as the network evolves (i.e. as peers enter and leave the network).

In Pastry, each peer has a 160 bit unique identifier (hereafter simply identifier) that, according to its creators, should be created randomly. Communication between peers is done by routing messages to the closest identifier that is found. At each hop along the route a path is chosen such that the message will get numerically closer (in terms of the identifier) to its destination. The identifier space wraps

around at the lower and upper limits of the 160 bit identifier to form a ring. Figure 1 below demonstrates this.
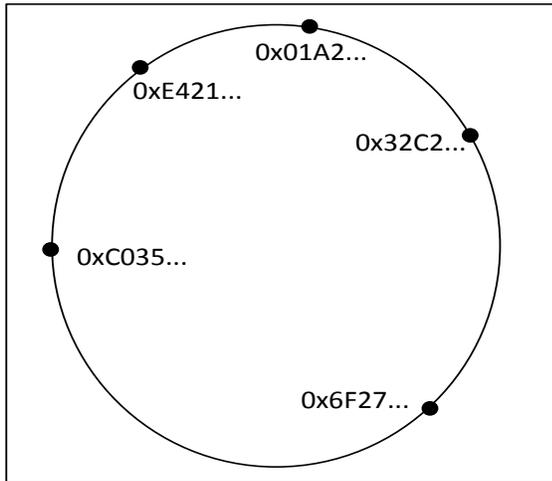

Figure 1 – FreePastry Ring

A host joins a ring of peers by sending a message to one peer that it knows the IP address of. Then the joining host builds a routing table by sending a message to several randomly generated identifiers and tracking the identifiers of each hop on the return route.

B. Squid

Squid [2] is a method of searching a peer-to-peer network overlay using multiple keywords without flooding (searching every possible peer). To accomplish this, Squid uses a Hilbert Space Filling Curve (SFC) to map from the multiple keyword space to the one-dimensional index space. Figure 2 demonstrates a Hilbert Space Filling curve. The numbers inside the squares indicate the distance traveled along the curve. With an alphabetic keyword of "c" and a numeric keyword of "2" the distance along the curve is 8. With keywords that have multiple characters and span the whole alphabet, the curve is refined multiple times.
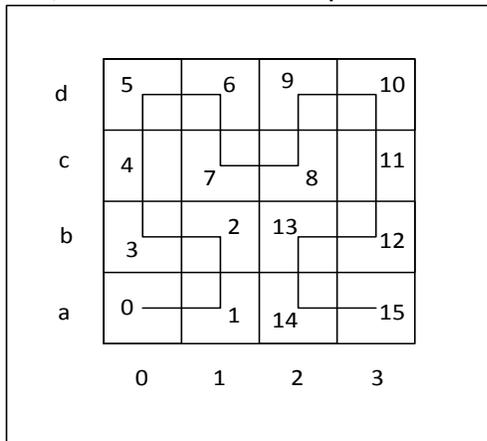

Figure 2 – Hilbert Space Filling Curve

The one-dimensional index space that is the distance along the SFC is mapped to the one-dimensional identifier

space of the existing network overlay. A hierarchical search is done by first using the least possible refined space filling curve and sending a search message there. If the host at that destination does not have the information, the curve is successively refined until either the result is found or the result is not found. This method uses the same underlying routing technique of Pastry in that it is guaranteed that at every hop the algorithm proceeds towards a result. Note the authors of the Squid paper did not specify any efficient computational methods of refining very large Hilbert Space Filling Curves. In our implementation we use the methods from Wang and Shi [6] in two-dimensions.

C. Software Transactional Memory

Software Transactional Memory (STM) [1,4] is a method of maintaining safety conditions without locks in shared memory systems with multiple concurrent processes. There are many variations on the theme. The software transactional memory variation used in this paper is as follows: A set of operations is marked atomic (the view from others is that the entire set is written to memory at once). One method of doing this is to keep a global version of each variable used in the commit process. During a commit the version is incremented and saved for each variable. At the end of a commit all of the versions are read again, if they are the same, the commit succeeds, otherwise the commit is restarted.

An example of an operation that updates a bank account is below.

```
1   atomic {
2     int balance = readBalance( );
3     boolean result;
4     if ( balance – 100 > 0 ) {
5       subtractFromBalance( 100 );
6       result = true;
7     } else {
8       result = false;
9     }
10  }
```

Figure 3 – Example Atomic Operation

III.   P2PSTM ARCHITECTURE

The Peer-to-Peer Software Transactional Memory (p2pstm) System is built in two layers. First the Peer-to-peer Versioned Variable layer is described and then the p2pstm layer is described.

A. Peer-to-peer Versioned Variable Layer

The public interface to the client-side Peer-to-peer Versioned Variable Layer is as follows:

- **publish**(String keyword, int version, String data) Publish data to the peer that handles the given keyword and version. If the keyword:version pair already exists on a peer (including a not validated

pair), then false is returned. Otherwise true is returned.

- **searchLatestVersion**(String keyword) Hierarchically search the ring to find the latest version of a keyword. If it is not found, zero is returned, otherwise the version as an integer is returned.
- **getValueQuery**(String keyword, int version) Get the data that was published using a keyword:version pair. If the keyword:version pair does not exist, the return value indicates so. If the pair has not been validated, the return value also indicates so. Otherwise, the data is returned.
- **sendValidation**(String keyword, int version, boolean commit_succeeded) Send a validate message to the peer corresponding to the keyword:version pair. A boolean is returned indicating the success or failure of this operation. If a validation does not come within an adaptive time window the keyword:version pair is destroyed on that host.
- **clone**(Set<NodeHandle> nodes) Clone the keyword, versions and data of all keyword:version pairs that the current node is now closest to. A peer can become closest to a keyword:value pair after it has been published in some cases by a peer recently joining the ring.

With this public interface the client side of the p2pstm layer can be built.

B. Peer-to-peer Software Transactional Memory Layer (p2pstm)

The algorithm for a commit in p2pstm is as follows:

```
1   boolean commit_succeeded = false
2   while ( commit_succeeded == false ) {
3       searchLatestVersion for all reads in the transaction
4       getValueQuery for all the latest versions
5       execute the transaction that the user has specified
6       publish all of the writes in the transaction
7       if ( a publish returned false ) {
8           sleep with exponential backoff and restart the loop
9       }
10      searchLatestVersion for all reads in the transaction
11      if ( a read version has changed ) {
12          sleep with exponential backoff and restart the loop
13      }
14      sendValidation for each write in the transaction
15      if ( a sendValidation returned false ) {
16          sleep with exponential backoff and restart the loop
17      }
18      commit_succeeded = true;
19  }
```
Figure 3 - The p2pstm commit algorithm

To avoid deadlocks in the algorithm, a partial order of the reads and writes must be maintained. This is done by sorting read and write requests first by keyword and then by version.

When a peer enters the ring, the host that some keyword:version pairs map to can change. This can cause problems because a publish for keyword: x, version: 0 may return true if the new peer handles that identifier, but an old peer has the history that x:0 exists. This is solved by requiring a peer to clone its closest left hand side and right hand neighbors keywords, versions and data that the keyword:version pair is now closest to the joining peer.

To handle nodes leaving the system, periodically a clone happens as well. The algorithm is to clone the nearest neighbor variables that would be numerically closest to the sender peer if the nearest neighbor peer were not present.

To optimize the commit algorithm, at line 10, if a version was read and written to and the write succeeded, there is no need to search for the latest version again.

To make the commit algorithm have a degree of fairness, the truncated exponential backoff algorithm is modified. A short version of the commit that highlights the fairness addition is in Figure 4 below.

```
1   int current_retries = 0;
2   boolean commit_succeeded = false;
3   int sleep_retries = getPreviousRetries( );
4   while ( commit_succeeded == false ) {
5       if ( a part of the transaction failed ) {
6           sleep_retries++;
7           current_retries++;
8           num = random number between 0 & sleep_retries
9           limit num to MAX_RETRIES
10          sleep(num * base_sleep_value)
11          continue;
12      }
13  }
14  setPreviousRetries(MAX_RETRIES – current_retries);
```
Figure 4 – The fairness addition to truncated exponential backoff

C. A Developer's Perspective

The programming interface is such that the developer implements a derived AtomicOperation class that fills in the details of three methods: 1) transaction (what to do during the commit), 2) reads (what variables are read from in the transaction) and 3) writes (what variables are written to in the transaction. This is demonstrated in lines 8 through 20 of Figure 5.

```
1   public class Example {
2
3       public void run(Squid squid){
4           int increment = 1;
```

```
5        AtomicOperation op =
6          new AtomicOperation(increment){
7
8          public String[] reads() {
9            return new String[] {"x"};    }
10
11         public String[] writes() {
12           return new String[] {"x"};    }
13
14         public void transaction() {
15           StmInteger x = getValue("x");
16           x.add((Integer) getArg(1));
17           setValue("x", x);
18           setResult(0, (Integer) x.get());
19           }
20       };
21       op.commit();
22       int result = (Integer) op.getResult(0);
23       System.out.println(result);
24     }
25   }
```
Figure 5 - A User's Perspective of Peer-to-Peer Software Transactional Memory

There is an StmInteger (line 15) class that represents an Integer that has peer-to-peer software transactions backing its value. A write to the integer is kept in a log, that is applied when the AtomicOperation is committing.

The AtomicOperation also provides the following methods to manage data within a transaction:

- **getValue**(String keyword) get an StmInteger corresponding to a string key
- **getArg**(Integer argument_number) get a local argument passed in to the constructor of AtomicOperation indexed by an integer
- **setValue**(String keyword, StmInteger value) send the value of an StmInteger to the peer-to-peer network
- **setResult**(Integer index, Object value) save a local that can be accessed after commit has successfully completed

Lastly, at line 20 the AtomicOperation.commit call is shown, this executes the commit algorithm, using the concrete transaction method.

IV.   P2PSTM BLOCKING QUEUE

As a test of the capability of the primitives provided by p2pstm, a peer-to-peer blocking queue was built. With a peer-to-peer blocking queue, any peer in the ring can enqueue tasks and also any peer in the ring can dequeue tasks. This provides a convenient mechanism for distributing workload in a peer-to-peer computational network. Two algorithms are needed, one for enqueue and one for dequeue.

A.   Enqueue

The algorithm for enqueue is listed below

```
1    void enqueue( String value ) {
2      atomically {
3        read "tail" as an integer
4        write to "item"+tail_value
5        increment the value of "tail"
6      }
7    }
```
Figure 6 – Enqueue Algorithm

The mechanics of writing this algorithm in Java code with the exact interface provided requires that it be broken into two parts so that the "item"+tail_value can be specified ahead of time. Also, in a real implementation, the string variable identifiers are prepended with a queue name in order to support multiple queues in a ring

B.   Dequeue

The algorithm for dequeue is listed below

```
1    String dequeue( ) {
2      while( true ){
3        atomically {
4          read "tail" as an integer
5          read "head" as an integer
6          if tail_value == head_value
7            restart loop
8          read from "item"+head_value
9          increment the value of "head"
10       }
11       return the value read in line 8
12     }
13   }
```
Figure 7 - Dequeue Algorithm

As with the enqueue algorithm, in implementation, it is broken into two parts and a queue name is prepended to identifiers.

V.   PERFORMANCE RESULTS

The commit algorithm described in this paper was implemented in the Java Programming Language and the elapsed time to complete a commit was measured in a computer lab with 2 through 24 peers active. The commit operation executed was to atomically increment four integers. Originally, failures in the algorithm were detected if at any point in time outside a commit all four integers did not have the same value. There are no outstanding failures known in the algorithm at this time. Figure 8 shows the average commit time versus the number of peers when fairness is included in the algorithm. Figure 9 compares the system with and without fairness. The dashed line is without fairness. The solid line (bottom) in Figure 9 is the same data as Figure 8, but scaled differently.
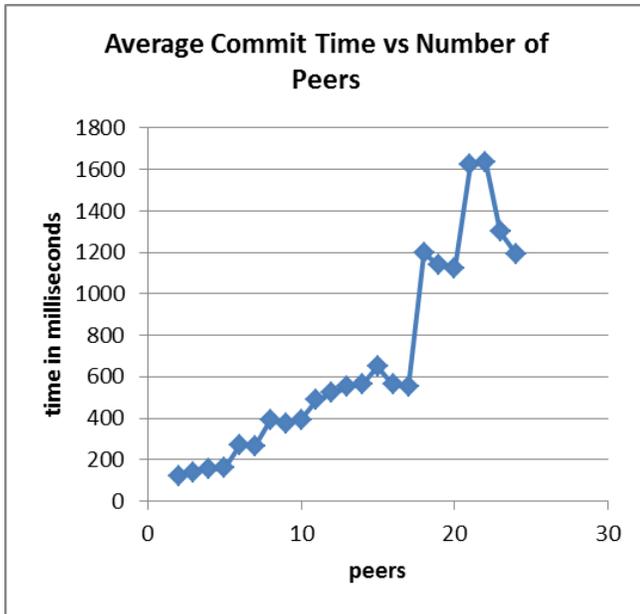
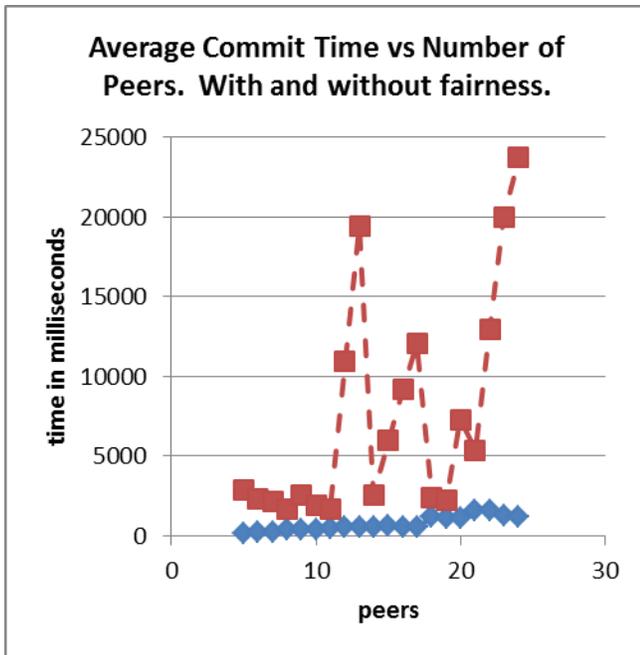Figure 8 – Performance Results with Fairness Included

Figure 9 – Comparison between a system with and without fairness

It can be seen from Figures 8 and 9 that fairness is extremely important to lower the average commit time of each peer. Figure 8 shows that the average commit time for up to 17 concurrent peers is less than a second. After that it is between one and two seconds.

For all performance measurements the peers waited randomly between 3 and 23 seconds between commits (in client code outside of the AtomicOperation class). The

AtomicOperation class uses 10 milliseconds as the base exponential backoff time and the retries where limited to 8 (the MAX_RETRIES value).

VI.   RELATED WORK

In the paper titled "A Transactional System for Structured Overlay Networks" [8] a peer-to-peer software transactional memory is presented. The system uses two-phase locking where the first phase is lock acquisition and the second phase is lock release. Our system uses a more optimistic approach, a write is made to a version and if that version exists, a failure is returned. In their work deadlocks are broken in the lock mechanism by requiring a priority for each transaction. They state that a simple way to assign priorities is by using timestamps and give an algorithm for synchronizing timestamps. Our method requires no synchronization of time stamps, instead the order of each keyword:version pair specifies the order of reading and writing shared state during a commit. In their work they specify that each object has a unique identifier used to locate it in the distributed hash table. They do not specify how this unique identifier is created in such a way that will cause the group of peers to have a consensus on where that object is located. Our method uses Hilbert Space Filling Curves to map from a two dimensional keyword:version pair to a one dimensional node identifier space to have such a consensus.

VII.   ACKNOWLEDGMENTS

REFERENCES

[1] Wikipedia: Software Transactional Memory: http://en.wikipedia.org/wiki/Software_transactional_memory

[2] C. Schmidt and M. Parashar, "Squid: Enabling search in DHT-based systems," Journal of Parallel and Distributed Computing, vol. 68, Jul. 2008, pp. 962-975.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

[4] N. Shavit and D. Touitou, "Software transactional memory," - Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, Ottowa, Ontario, Canada: ACM, 1995, pp. 204-213.

[5] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan, Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service, Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001.

[6] N. Chen, N. Wang, and B. Shi, "A new algorithm for encoding and decoding the Hilbert order," Softw. Pract. Exper., vol. 37, 2007, pp. 897-908.

[7] Kim-Thomas Möller, Marc-Florian Müller, Michael Sonnenfroh and Michael Schöttner, " A Software Transactional Memory Service for Grids" Algorithms and Architectures for Parallel Processing . Lecture Notes in Computer Science, 2009, Volume 5574/2009, 67-78, DOI: 10.1007/978-3-642-03095-6_7

[8] Valentin Mesaros and Raphaël Collet and Kevin Glynn and Peter Van Roy, "A Transactional System for Structured Overlay Networks". 2005