AgentTeamwork

Enhancing Communication and File I/O

Joshua Phillips (University of Washington, Bothell)
Advisor: Professor Munehiro Fukuda

2007

•Name
•Sponsor
•Title and project
•Japan for 5 months
•Disclaimer about random photos

Outline

1. AgentTeamwork Background
2. Fixing MPJ
3. Simplifying a User Program
4. Enhancing GridTcp
5. GridTcp Performance
6. Enhancing File I/O
7. File I/O Performance
8. Conclusions

•7 things to cover today

•Roughly 6 phases

AgentTeamwork Background

- AgentTeamwork is a *grid-computing* middleware system that dispatches a user application with *mobile agents* to a collection of remote computers. User processes running on a different computer are monitored, moved, and resumed by those mobile agents.

•Grid-computing middleware

•Uses mobile agents to run, manage, and recover jobs

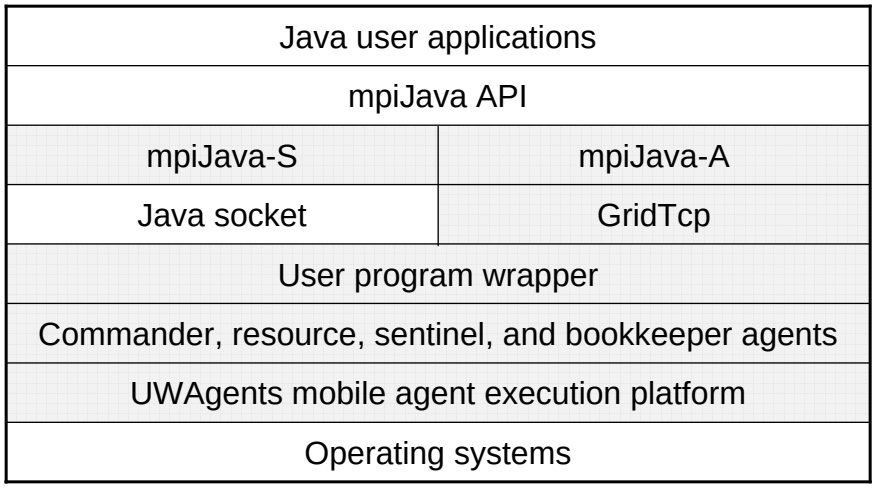
•100% java

•NSF funded

•UW Sponsored

•Developed in collaboration with Ehime University

AgentTeamwork Background (cont)

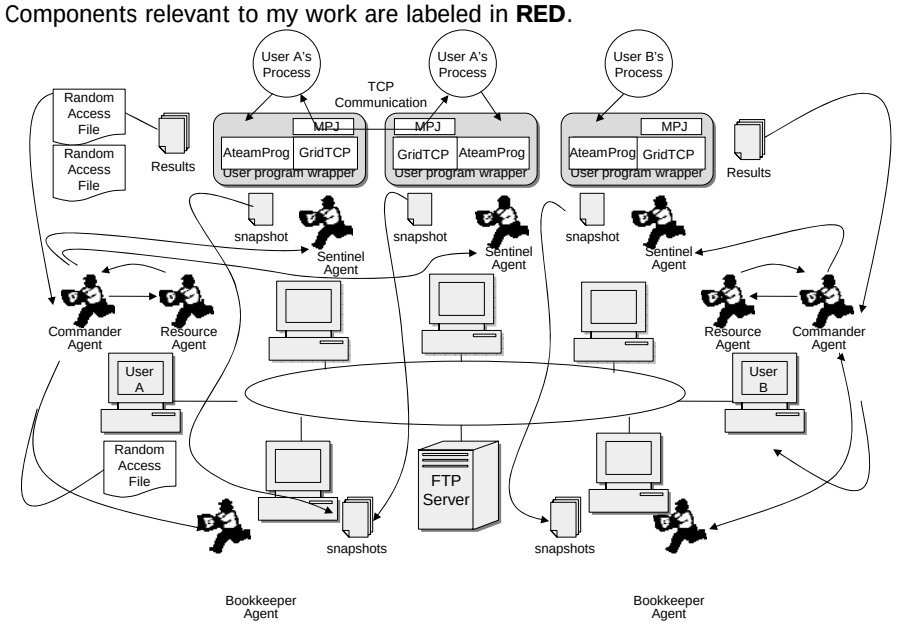


AgentTeamwork Execution Layers

- Execution layers
- See the parts I worked on in red

AgentTeamwork Background (cont)

Components relevant to my work are labeled in **RED**.



- An ftp server and
- 7 hosts make up our system
- A commander agent is launched from the user machine
- Which then spawns a resource agent to locate the best host
- The commander agent then spawns sentinel agents on those hosts
- And bookkeeper agents to manage snapshots on other hosts
- Any files are then split appropriately and sent to the correct nodes
- The user processes are launched
- A snapshot is periodically generated
- And sent to the bookkeeper agents
- Any results are generated and returned to the user

Fixing MPJ

- Debugged errors that caused JGF Benchmark tests to fail
- Reformatted code to improve maintainability
- Generated JavaDoc

Fixing MPJ (cont)

```
44  public void run()
45  {
46  //=============================================SEARCH QUEUE
47
48      for(int i = 0; i < MessageQueue.size(); i++)
49      {
50        if(tag == MPJ.ANY_TAG && source == MPJ.ANY_SOURCE)//any source, any tag
51        {
52          if(((MPJMessage)MessageQueue.elementAt(i)).getType() !=
53                                                  type.GetType() );
54                                                    //match the type
55          else
56          {
57            System.arraycopy(((MPJMessage)MessageQueue.
58                            elementAt(i)).getMessage(),
59                            0, recvbuf, offset, ((MPJMessage)MessageQueue.
60                            elementAt(i)).getCount());
67    /**
68     * Begins execuction of the thread and attempts to receive a message
69     * This is the entry point for the thread.  When the thread is started
70     * it starts here.
71     */
72    public void run( ) {
73        // traverse the message queue
74        for( int i = 0; i < MessageQueue.size( ); i++ ) {
75            // if tag is any tag and source is any source
76            if( tag == MPJ.ANY_TAG && source == MPJ.ANY_SOURCE ) {
77                // if the type in the message queue does not equal the specified
78                // type, do nothing
79                if( ( ( MPJMessage ) MessageQueue.elementAt( i ) ).getType ( )
80                    != type.GetType( ) )
81                        ;
82                // otherwise copy the arry into the receive buffer, remove it from the message queue
83                // and return
84                } else {
```

Top: Code before reformatting

Bottom: Code after reformatting

Fixing MPJ (cont)

Constructor Detail

ISendThread

```
public ISendThread(java.lang.Object sendbuf,
                   int offset,
                   int count,
                   Datatype type,
                   int rank,
                   int tag,
                   java.io.OutputStream OStream)
```

JavaDoc

Instantiates an ISendThread object.

Parameters:

`sendbuf` - the send buffer Object
`offset` - the int offset for the send buffer
`count` - the int number of elements in the send buffer
`type` - the Datatype of the message in the buffer
`rank` - the int rank to send the message to
`tag` - the int tag for the message
`OStream` - an OutputStream for serialization //TODO: is this correct?

Method Detail

run

```
public void run()
```

Begins thread execution and attemps to send the contents of the buffer. When the thread is started it starts here. Error checking must be done before run is called.

Specified by:

Simplifying a User Program

- A user program now must only extend AteamProg
- No need to explicitly define required AgentTeamwork members
- No need to partition a user program into functions. Just call *takeSnapshot()*
- No need to use specialized I/O classes. GridFile streams and GridTcp connections have been wrapped with standard I/O names and methods.

•By extending AteamProg, a user program:

- •Does not have to explicitly define required AgentTeamwork members
- •Does not need to partition a user program for check-pointing. They can just call takeSnapshot
- •Does not need to use specialized I/O classes, “Standard” classes can be used

Simplifying a User Program (cont)

A user program before AteamProg

```
 1 import java.io.*;
 2 import AgentTeamwork.Ateam.GridFile.*;
 3 import AgentTeamwork.Ateam.GridTcp.*;
 4 public class MyApplication {
 5
 6     public GridIpEntry ipEntry[];          // system required
 7     public int funcId;                     // system required
 8     public GridTcp tcp;                    // system required
 9     public GridFile gridfile;              // system required
10     public int nprocess;                   // system required
11     public int rank;                       // system required
12     private GridFileInputStream  gfis;     // a user input stream
13     private GridSocket gsock;              // a user socket
14
15     public int func_0( String args[] ) {   // constructor
16         gfis = new GridFileInputStream(    // create input stream
17             gridfile);                     // gridfile object
18         gsock = new GridSocket(0,22410,    // create socket
19             tcp);                          // tcp object
20         ...;
21         return 1;                          // calls func_1( )
22     }
23
24     public int func_1( ) {                 // called from func_0
25         int data = gfis.read( );           // read a byte of data
26         InputStream is =                   // create a socket stream
27             gsock.getInputStream( );
28         ...;
29         return 2;                          // calls func_2( )
30     }
31
32     public int func_2( ) {                 // called from func_2,
33         gfis.close( );                     // close file stream
34         gsock.close( );                    // close socket
35         ...;
36         return -2;                         // application terminated
37     }
38 }
```

Before ateam prog:

Note:

•the heavy amount of “system required” members

•And the partitioning

Simplifying a User Program (cont)

A user program after AteamProg

```
1  import AgentTeamwork.Ateam.*;
2  public class MyApplication extends AteamProg {
3      private int phase;                          // snapshot management
4      private FileInputStream fis;                // a user input stream
5      private Socket sock;                        // a user socket
6      public MyApplication(Object o){}            // system reserved
7
8      public MyApplication( ) {                   // user-own constructor
9          phase = 0;
10         fis = new FileInputStream( );           // create input stream
11         sock = new Socket(3,22418);             // create socket
12     }
13
14     private void compute( ) {                   // user computation
15         int data = fis.read( );                 // read a byte of data
16         InputStream is =                        // create a socket stream
17             sock.getInputStream( );
18         ateam.takeSnapshot(phase);              // check-pointing
19         ...;
20         fis.close( );                           // close file stream
21         gsock.close( );                         // close socket
22     }
23
24     private boolean userRecovery( ) {
25         phase = ateam.getSnapshotId( );         // version check
26     }
27
28     public static void main( String[] args ) {
29         MyApplication program = null;
30         if ( ateam.isResumed( ) ) {             // program resumption
31             program = (MyApplication)
32             ateam.retrieveLocalVar( "program" );
33             program.userRecovery( );
34         } else {                                // program initialization
35             MPI.Init( args );                   // JavaMPI initialize
36             program = new MyApplication( );
37             ateam.registerLocalVar( "program", program );
38         }
```

No partitioning, no member variables,

However, There must be a system reserved constructor: this allows the User Program Wrapper to instantiate the necessary members before a user attempts to use them in his/her constructor

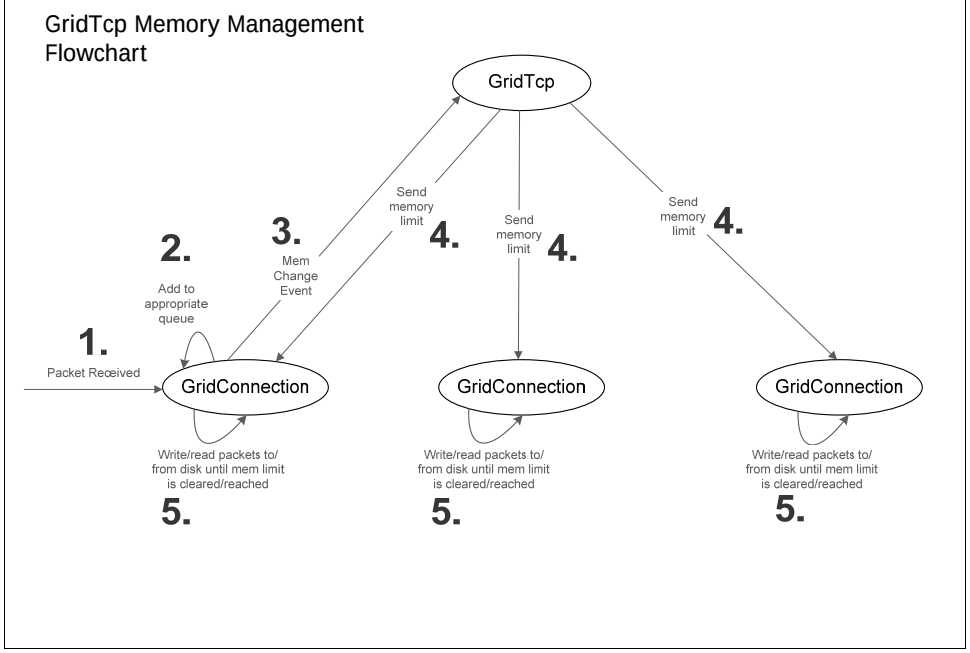
Enhancing GridTcp

- *Memory Management:*
  Because GridTcp is error recoverable it often contains a large amount of backup data. This data can easily cause OutOfMemoryErrors if left unchecked. The new version of GridTcp automatically limits the amount of in memory messages.
- *Flow Control:*
  To prevent a GridTcp object from becoming too overloaded, I have implemented simple flow control.

•Two major aspects of grid tcp enhancement:

- •Memory mangement:
  - •if a user takes a snapshot frequently, a large amount of saved messages can quickly fill up memory, so now we write them to disk
- •Flow control:
  - •Prevent a gridtcp object from becoming too overloaded client requests

Enhancing GridTcp (cont)



•Every user program has a single GridTcp object that manages many gridconnections

•to fairly manage memory between all grid connections and eliminate wasteful polling, java events were used

•When a packet is received :

•It is added to the appropriate queue (incoming, forwarding, backup)

•A memory change event is then generated

•GridTcp calculates the individual thresholds for each connection

•Passes them this limit

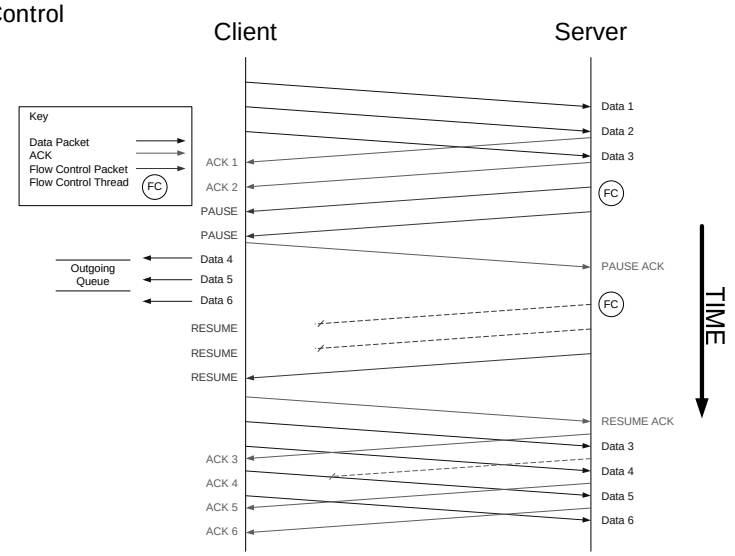
•The grid connections then write the oldest packets to disk until the threshold is cleared

•To simplify this procedure and ensure the on-disk messages are recoverable, I created DiskVector:

- •A List class that uses a disk for a backing store and overwrites serialization/deserialization methods to include on disk files

Enhancing GridTcp (cont)

GridTcp Flow Control Timechart



•Very simple flow control that pauses a client when the server is overloaded, and resumes it when it's not

GridTcp Performance

- In testing the changes to GridTcp I found that memory usage shrank significantly. However, through the use of JProfiler, I have determined that there is a memory leak unrelated to my changes.
- Even though Java is a managed language, if a reference to an object does not get removed, it's memory is not freed.

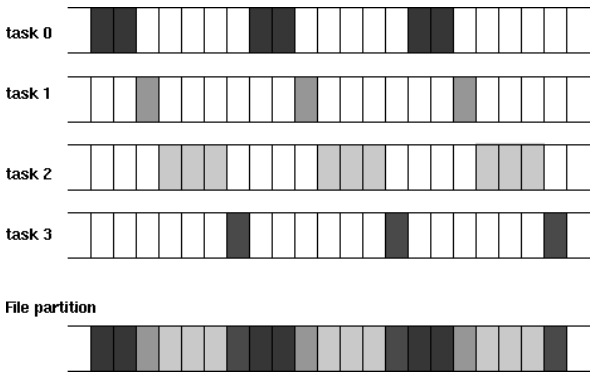
•Message management was successful, but there is still an unrelated memory leak in GridTcp which can cause it to crash

•Java is a managed language, but if references to an object are not removed, the Garbage collector cannot free that memory

•I used JProfiler (profiling tool) to discover the memory leak but the layers are too complicated in AgentTeamwork and I did not have time to locate the leak

Enhancing File I/O

- Implemented a serializable, distributed RandomAccessFile
- Behaves exactly like the Java API's RandomAccessFile
- Each node in a system owns a user-defined *stripe* of the file but has a transparent virtual view of the *whole file*



•Previously, Ateam's recoverable I/O was limited to a stream

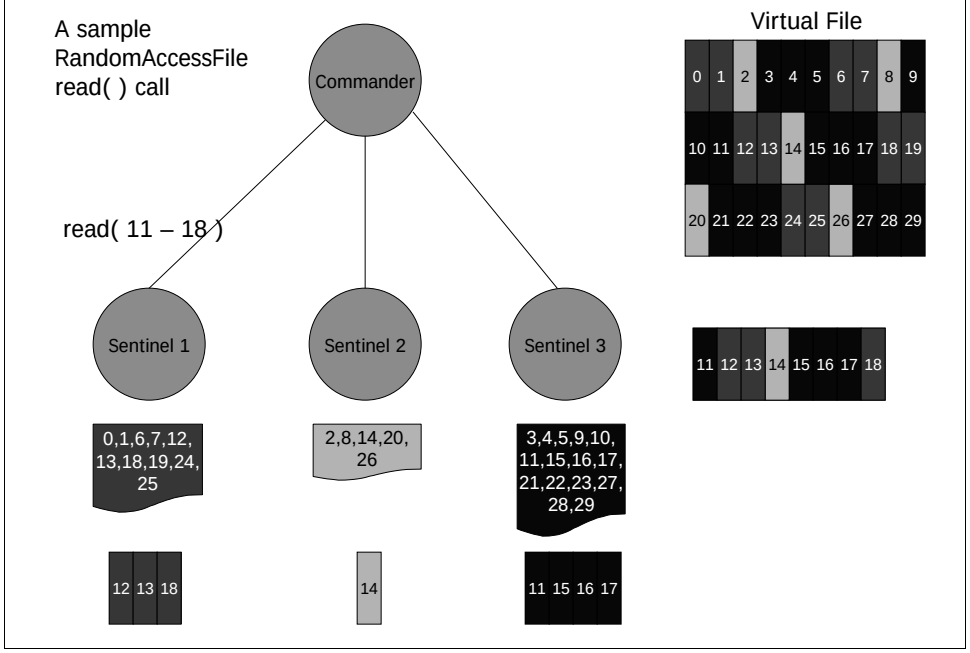
•Now we have a RAF that behaves just like the Java API version.

•Each node owns a user-defined "stripe"

•But maintains a virtual view of the whole file

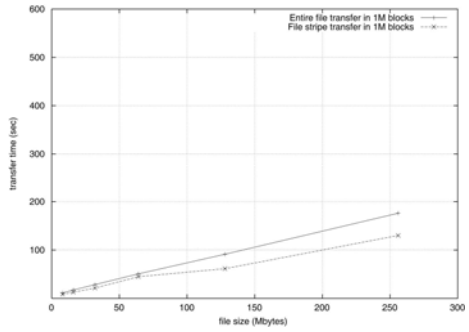
•If data is requested that does not reside locally, it is transparently transferred

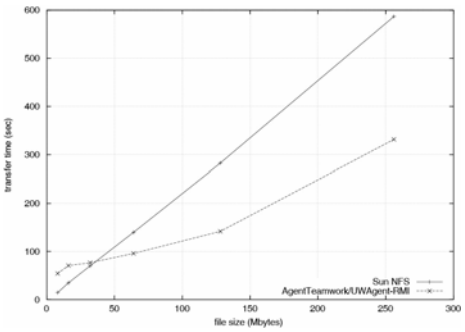
Enhancing File I/O (cont)



File I/O Performance

- Comparison of a file-stripe transfer with an entire file transfer
- The file-stripe transfer has yielded 1.35 and 4.5 times better performance than the entire file transfer when sending a
  256MB file





•left graph is the performance difference in an entire file transfer and file-stripe transfer within AgentTeamwork

•Right graph is the performance difference between SUNFS file transfer and AgentTeamwork file transfer

Conclusions

- AgentTeamwork is now easier to use and I/O is significantly simpler, more capable and more efficient
- Major skills developed/used:
  - Multithreaded programming/debugging
  - Serialization
  - Object-oriented fundamentals (polymorphism/inheritance)
  - Knowledge of network stacks and TCP
  - Java reflection
  - Algorithm optimization

Questions

- Thank you!
- Any questions?