AgentTeamwork

Final Report

Enhancing Communication and File I/O

Joshua Phillips
University of Washington, Bothell &
愛媛大学　(Ehime University)
0222418
12/16/06 – 02/28/07

Table of Contents

Major Accomplishments

My contributions to AgentTeamwork are divided into seven discrete phases, four of which have been completely detailed in my midterm report and the final three which are described in this document. Each of those three phases and the major accomplishments they realized are listed in detail:

Phase 5 - Enhancement/Implementation of File I/O in AgentTeamwork

Phase 5 was split into three small parts. The first task was to merely create simple wrapper classes for *GridInputStream* and *GridOutputStream* in the *Ateam* package that mimic the *Socket* and *ServerSocket* wrapper classes: they simply simplify user programs and porting by allowing a user to instantiate AgentTeamwork's I/O classes while still using the names and methods found in the Java API. This was accomplished by simply extending the AgentTeamwork versions and creating a constructor that accessed the user program's *GridTcp* and *GridFile* objects via *AteamProg's* static *Ateam* member. With the exception of the user program extending from *AteamProg* this is all transparent to the user.

The second task was modify AgentTeamwork's GUI to allow a user to specify which file should be delivered to each MPI rank. My only contribution was creating the algorithm that partitions the file according to the user's specification; Jumpei implemented the actual transfer algorithm.

Thirdly, I created the first implementation of AgentTeamwork's specialized *RandomAccessFile*. Ateam's *RandomAccessFile* is a distributed file where each node in the system receives only a *partition* of the whole file yet maintains a virtual view of the *whole* file. Ateam's *RandomAccessFile* implements both the Java I/O *DataOutput* and *DataInput* interfaces so it can be treated just as if it were the Java I/O *RandomAccessFile*. If a node requests data that it owns locally, it simply reads from that partition, but if it requests or submits data remotely, that data is transferred transparently between nodes. It is important to note that while these read/write operations are atomic there is no protocol for ordering; the user program must develop its own mechanism for ensuring that network delays do not alter the order of read/writes.

I mentioned that this was only the *first* implementation of *RandomAccessFile*. In fact, two versions were implemented (the second of which is described in Phase 6). The only major difference between these two versions is the method used to partition the data. In this first implementation, the file is distributed in contiguous *chunks* as opposed to the second version's far more complicated *stripes*.

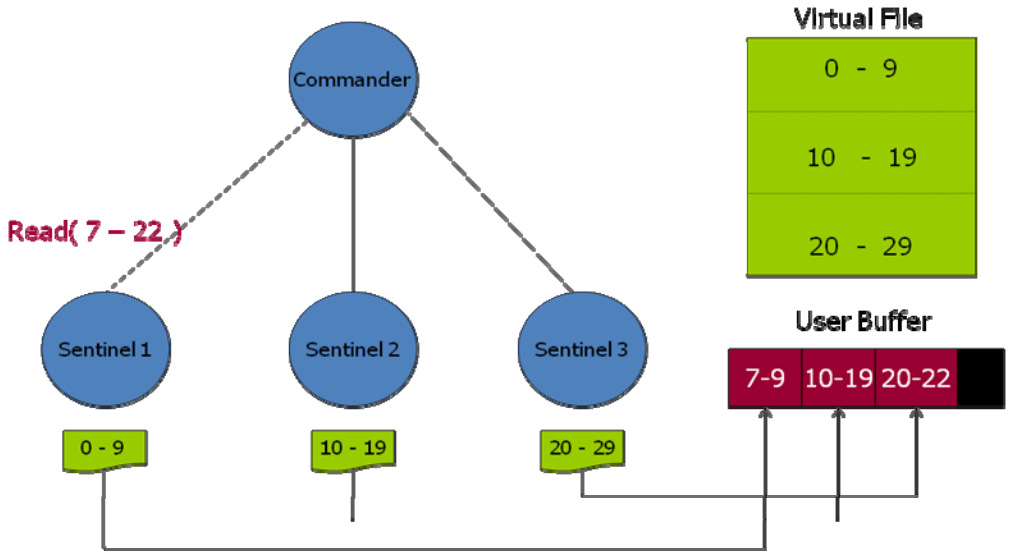

Figure 1. Version 1 of Ateam's RandomAccessFile and an example read.

The most difficult aspect of implementing this distributed *RandomAccessFile* was synchronization. To be able to perform local requests as well as accept incoming requests, *RandomAccessFile* spawns a thread upon instantiation that continually processes read/write requests from other nodes in an AgentTeamwork system. To create atomic read/write requests and prevent deadlock, meticulous planning and use of Java monitors was required.

This version of *RandomAccessFile* provides better performance (especially when comparing sending a whole files to all files in a system and just sending the partitions) but severely limits the flexibility a user has in sending only the data that a specific node needs. To overcome this problem, we looked towards the MPI-IO standards view on parallel file sharing and specifically implemented and modified version of its *file* view model described in Phase 6.

Phase 6 – Enhancement of RandomAccessFile.java

The second version of *RandomAccessFile* included a few minor enhancements and one major upgrade that was significantly difficult to implement correctly *and* efficiently.

The minor additions included a *barrier* method: simply a blocking call that waits for all nodes in a system to call the exact same method before returning. This allows a user, for example, to allow one node to wait on a read until another node finishes a write. I also fully fleshed out all of the *DataInput* and *DataOutput* operations so that *RandomAccessFile* is fully capable of reading and writing Java data types.

The real meat of Phase 6 was the change in *RandomAccessFile*'s partitioning scheme, based off of the MPI-IO standard's *file view*, but slightly different. Our version of *RandomAccessFile* is still 100% *byte-based* while the MPI-IO file read and write operations are etype based. In better terms, MPI-IO will only read and write fully defined data types at a time (such as an integer) while AgentTeamwork's *RandomAccessFile* can do that, via the *DataInput* and *DataOutput* interfaces, as well as behave normally and access only bytes. In fact, the real change (to the user) is just how the data is partitioned to the user nodes. The internals of *RandomAccessFile* are complicated but the external interactions are intentionally simple.

I have created three separate classes that in cooperation make up a *RandomAccessFile*: *RAFPartitionInfo, RandomAccessFile,* and *RandomAccessFileCommThread*.

At the most basic level is *RAFPartitionInfo*: merely a "card" that provides the information about a partition. This card is used in all of *RandomAccessFile*'s algorithms to turn a virtual file pointer into a rank, and local file pointer.

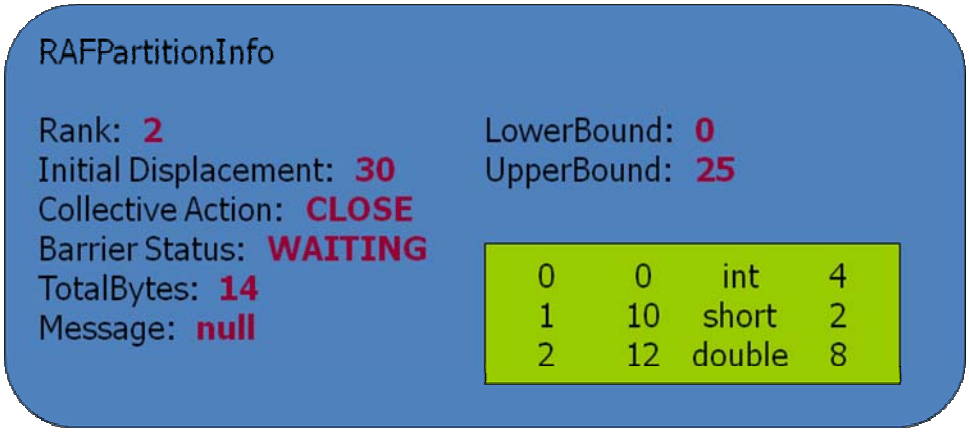

Figure 2. A sample RAFPartitionInfo.

Next, we have *RandomAccessFile* itself. This class provides the interface for the user that allows byte-reads, byte-writes, data-type-reads, data-type-writes, closing, resizing, creating partitions and synchronizing.

Finally, there is *RandomAccessFileCommThread*. This class continually polls an *InputStream* for each of the other nodes in the system and waits for incoming requests. To eliminate synchronization problems (note: ordering is still left to the user), *RandomAccessFile* contains a dummy *Object* member that is merely used as a monitor. This monitor is requested anytime the virtual file pointer is modified or a file's local contents are viewed or updated.

There are three separate algorithms that make *RandomAccessFile* work as well and efficiently as it does: the partition-creation algorithm, the rank-lookup algorithm, and the read/write algorithms.

Partition Creation

RandomAccessFile includes a static method called *createPartitionFiles* that accepts a file called *wholeFile*, a vector of *RAFPartitionInfo*'s and returns a *Vector* of files. As seen above, a *RAFPartitionInfo* contains the data the shows *RandomAccessFile* how the data is partitioned. The partitioning scheme can be better described visually.

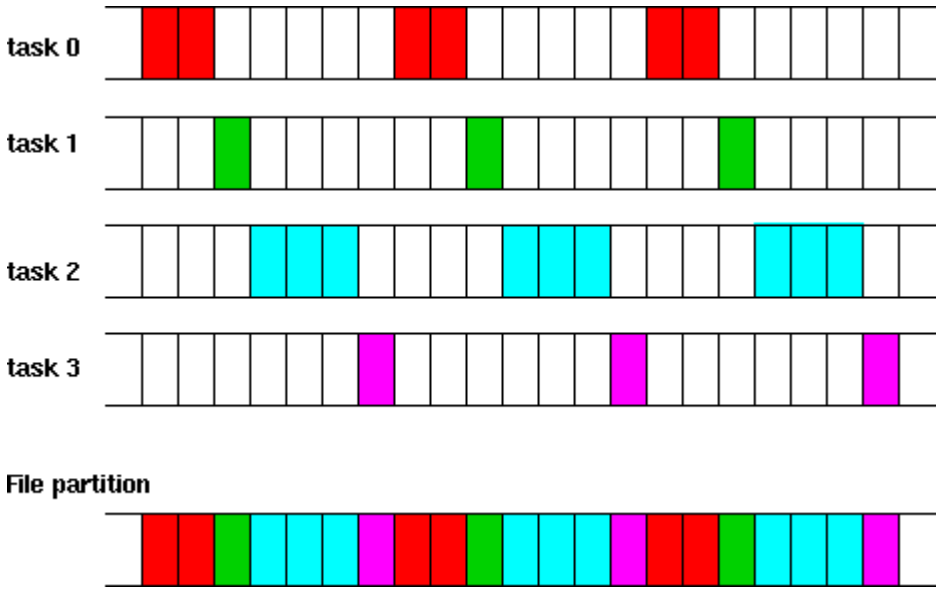

Figure 3. An example of RandomAccessFile's partitioning scheme with a 7 byte filetype and 4 nodes.

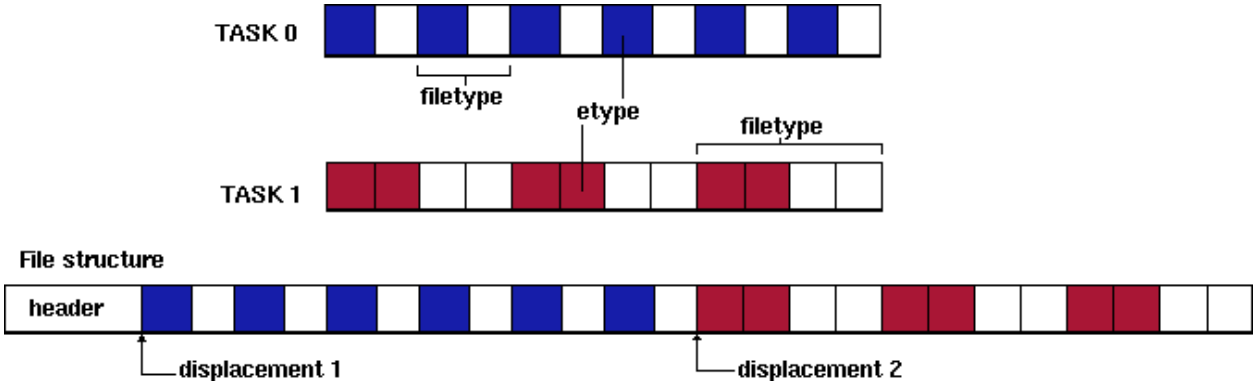

Figure 4. An example of RandomAccessFile's partitioning scheme utilizing an initial displacement.

For a more detailed description of the file partitioning scheme, please see "MPI-IO Standard" in the references section.

Once the partitioning scheme is understood, it is possible to explain the algorithm that accomplishes this task, and why it is different from the other algorithms within *RandomAccessFile*. Originally, the partition-creation algorithm was implemented to create one partition file at a time in the same manner that read and write operations occur. However, upon testing, even an 8MB file that was partitioned between 24 nodes with each node receiving just one byte in a filetype, proved to be tremendously slow. In fact, a 256 MB at first took around 52 minutes just to create the partition files, without sending or reading the files. The source of this problem was the number of disk seeks being made on the whole file. To reduce the use of such a heavy operation, I designed a three-dimensional matrix that is only used within *createPartition*'s scope and is pictured below.

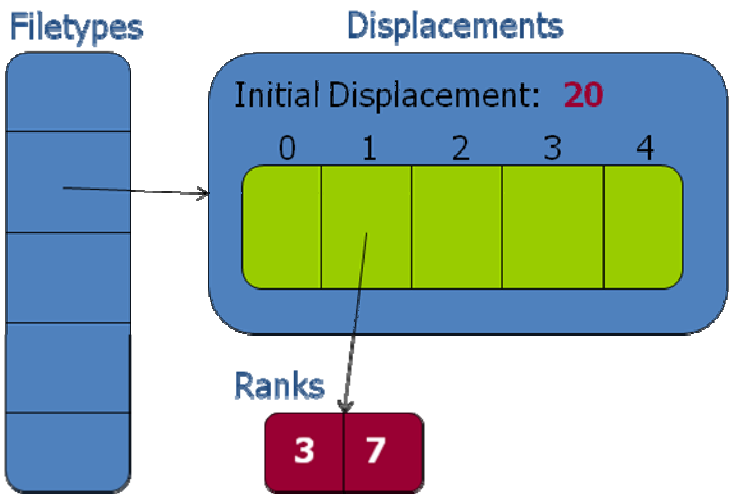

Figure 5. Temporary data structure for fast file-partitioning.

The reason this algorithm (and therefore the data structure) differs from those found in the read and write algorithms stem from the limitations of read and write. Parsing and partitioning a large file must be done sequentially to eliminate gross amounts of expensive disk seeks. However, if the read/write algorithms work sequentially the number of network requests is huge. As explained below, the read and write algorithms actually only send one network request but at the cost of potentially many disk seeks. Due to the fact that an average read/write involves a relatively small range of bytes the seek times are not nearly as noticeable as and much faster than sending and waiting for responses over the network. The data structure illustrated in Figure 5 allows *createPartitionFiles* to find the owning ranks of any byte with only a few light calculations and two direct array accesses.

Rank Lookup

The rank lookup algorithm is used by the read and write algorithms to locate the node(s) that any given byte of the virtual file resides on. Each byte in the given range is traversed. Starting with the local partition (which, due to user-defined partitioning, is the most likely to contain the data in question, each rank's *RAFPartitionInfo* is processed and it's file type table is cross-referenced with the virtual byte position in question. A binary search is then used to find the closest displacement within *RAFPartitionInfo*'s file type displacement table, if it exists. To increase efficiency, if a match is found, the corresponding etype is completely skipped over. (i.e. – if a long is found where the byte exists, all 8 bytes are skipped so that the remaining 7 bytes are not checked needlessly.) Once all bytes have been processed (or skipped) a set of the owning ranks is returned.

Read/Write

The core of *RandomAccessFile* is the read and write algorithms that present to the user a distributed file as completely locally owned. As mentioned before, the read/write algorithms are designed to reduce the number of network requests. They work by sending each node that shares ownership the range of bytes in question. Each node then constructs an array of its bytes (without holes) by looking at each displacement in a rank's filetype and determining if it is within the specified range. If it is, each repetition of that filetype's displacement is copied into the buffer and then the next displacement is processed (see Figure 3 to see how a filetype is repeated).

The alternate method is to search each byte sequentially and separately for the owning rank which can only be accomplished by performing a binary search through each rank's filetype table. This is an extremely slow solution. To illustrate how the more efficient version works, I will describe the read local algorithm in more detail.

1. Using *getOwningRanks*, retrieve the ranks that share ownership in the range specified by the current virtual file pointer and the length parameter passed to read
2. For every rank retrieved
   - 2.1. For every displacement listed in that rank's filetype table
   - 2.2. Convert the virtual file pointer into an offset within the filetype
   - 2.3. Convert the current filetype displacement into a virtual index
   - 2.4. If the virtual index is less than the virtual file pointer, adjust the index to the virtual file pointer and save the number of bytes skipped in the adjustment
   - 2.5. While the adjusted index is less than the end byte of the read range
     - 2.5.1. Calculate the bytes to read as the etype length associated with the current displacement with the previous adjustment subtracted
     - 2.5.2. If the bytes to read is 0 or less
       - 2.5.2.1. update the index to next repetition of the filetype
       - 2.5.2.2. return
     - 2.5.3. If the bytes to read plus the current index is greater than the end byte of the range
       - 2.5.3.1. Reduce the bytes to read to the end byte minus the index
     - 2.5.4. Convert the current virtual index into a local partition index
     - 2.5.5. Seek to local partition to the local partition index
     - 2.5.6. Read (bytes to read) number of bytes into the user buffer

Figure 6. The read local partition algorithm pseudo-code.

Note that although the read local, write local, read remote, and write remote algorithms are essentially the same, what they do in the end is different. This made it too difficult to generalize the algorithm into a single method that could be shared by all operations.

Phase 7 – File I/O Performance Evaluation and Conference Paper Submission

The final phase of my research was to compare the performance of the new, partitioned *RandomAccessFile* to that of the old, un-partitioned *RandomAccessFile*. Due to tight time constraints around the deadline of our conference paper that discusses these comparisons, Jumpei Miyauchi of Ehime University carried out the bulk of this performance evaluation while I offered only modest assistance. The most important discovery of this phase was the severe performance time of partitioning the whole file into stripped files. This, in turn, led to the development and use of the data structure illustrated in Phase 6, Figure 3.

While Jumpei completed the testing, I optimized the algorithms to increase performance. Jumpei has measured the time elapsed for the following sequence of random-access file transfer: (1) a commander agent reads 24 stripes of a given file, each to be delivered to a different sentinel; (2) the commander starts sending them in 1M-byte partitions; (3) agents at each tree level relays file partitions as regrouping them or further dividing them; and (4) all agents send an acknowledgment to the commander when accepting their allocated file stripe. These times are show below in Figure 7.

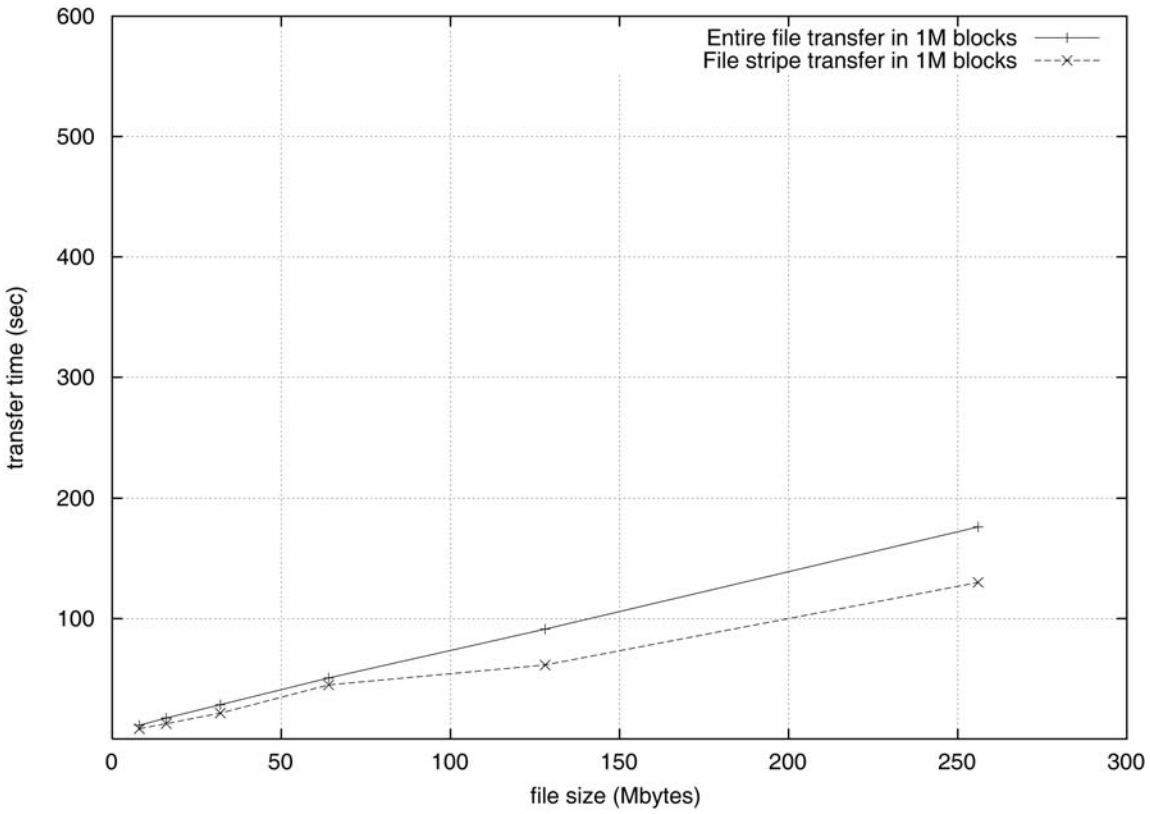

Figure 7. The comparison of transferring a whole file and transferring a partitioned file in Agent Teamwork.

Figure 7 compares this file-stripe transfer with an entire file transfer, both fragmented in 1M-byte partitions. The file-stripe transfer has yielded 1.35 and 4.5 times better performance than the entire file trnasfer and Sun NFS respectively when sending a 256M-byte random access file. Although a commander agent still needs to send 256 messages, (each with 1M-byte file partition) as in sending an entire file, each agent at the bottom of a hierarchy receives only 11 messages. Obviously, the more user processes the better this transfer performs.

Also, functionality tests of *RandomAccessFile* have been included in appendices A and B.

Skills Used and Developed

An important aspect of this project is to develop new skills and use old skills as I prepare for my career. The following is a cumulative list of some of the most important skills I have used and developed so far (in not particular order):

- Parallel programming
- The MPI API
- Multithreaded programming
- Multithreaded modeling
- Multithreaded debugging
- Knowledge of network stacks and TCP
- Serialization
- Inheritance: interfaces, abstract classes, method overriding, etc.

- Understanding of the Java language
- Understanding of the Java Virtual Machine
- Java packaging
- Java compilation
- Javadoc generation
- Linux shell scripting
- Linux security policies
- Technical writing
- Good commenting practices
- Code reading
- Modifying preexisting, large, complex software systems
- Java reflection
- Input/Output and Streams
- Understanding of performance bottlenecks
- Algorithm optimization

Next Steps

The final steps of my research are to work with Professor Fukuda to wrap up any unfinished work and to present my research at the CSS 2007 Spring Colloquium.

Files Created and Modified

The following is a cumulative list of files that have been either created or modified throughout the first half of the project:

File	Status	Changes/Use	Location
All AgentTeamwork source files	Old	Added package statements to almost every source file to restructure AgentTeamwork.	Medusa: /home/uwagent/agentteamwork-dev/
Ateam.java	New	For user-initiated snapshots. Added registerLocalVar() and retrieveLocalVar(). These methods allow local variables that are instantiated in main() to be serialized in a snapshot.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
ATeamException.java	New	For reporting and describing errors that occur within AgentTeamwork	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/
AteamProg.java	New	Allows for the serialization of a static Ateam member that can be accessed in a user program's static main method	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
backupToMedusa.sh	Dep.	Quickly backs up all files from local system to medusa	Koblab: /home/jawsh/tempAgentTeamworkBackup/
ByteFileMaker.java	New	Creates a 256 byte file with data 0 through 255 for testing RandomAccessFile	Medusa: /home/uwagent/agentteamwork-dev/tests/
ByteFileReads.java	New	Displays the contents of a byte file in integer form for testing RandomAccessFile	Medusa: /home/uwagent/agentteamwork-dev/tests/
cleanMyNodeProcesses.sh	New	Kills orphan java process on Medusa's nodes. This	Medusa: /home/uwagent/agentteamwork-

		orphan process sometimes prevent java sockets from binding.	dev/scripts/
Communicator.java	Old	Fixed simple bugs and added some documentation.	Medusa: /home/uwagent/agentteamwork-dev/MPJ/
CommunicatorTest.java	New	Tests the communication methods of MPJ as defined in phase 1 of my statement of work	Medusa: /home/uwagent/agentteamwork-dev/tests/
DiskVector.java	New	A class that extends java's AbstractList<E> and provides a list with the disk as a backing store. This class uses generics so it can be used for many general purposes.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp
FileCopyTest.java	New	Measures the time it takes to copy a whole file in Java one byte at a time. Used for RandomAccessFile's performance comparison.	Medusa: /home/uwagent/agentteamwork-dev/tests/
FileInputStream.java	New	Wraps GridFileInputStream.java	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam
FileOutputStream.java	New	Wraps GridFileOutputStream.java	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam
FileSplitterTest.java	New	Splits a file into partition files meant for four nodes. Used for testing RandomAccessFile	Medusa: /home/uwagent/agentteamwork-dev/tests/
genJavaDoc.sh	New	Creates consistent javadoc with complex command-line options	Medusa: /home/uwagent/agentteamwork-dev/scripts/
GridConnBackup.java	Dep.	A very simple class that includes a backup vector and connection ID's for serialization to disk.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridConnection.java	Old	Added a backup mechanism that writes old backup messages to disk when a specified threshold is reached. Also loads these persistent backups into memory when appropriate. Modified constructors and init() to allow for re-instantiation of a GridConnection with all of the memory management members included. Modified all methods to use DiskVector instead of GridConnBackup. Fixed some bugs.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridConnMemChangeEvent.java	New	An event that is used by GridConnection to notify GridTcp (or other subscribers) of a change in memory	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridConnMemChangeListener.java	New	An interface that any subscriber to GridConnMemChangeEvent must implement.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridFlowControlThread.java	New	Simply continues to send PAUSE or RESUME packets at a specified interval until it is killed	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
All Grid Threads	Old	Each class that extends from the Thread class in GridTcp	Medusa: /home/uwagent/agentteamwork-

		now sets its “thread name” in its constructor. This allows any GridTcp developer or user to easily determine which threads are running at any given time for debugging.	dev/AgentTeamwork/Ateam/GridTcp/
GridPacket.java	Old	Added new packet types: data_ack, pause, resume, pause_ack, resume_ack	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridReceiveThread.java	Old	Added a temporary try/catch block to catch out of memory exceptions so that I can debug GridTcp’s memory issues.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridTcp.java	Old	Added a backup memory space threshold that defines how many bytes a GridTcp connection may store in memory before backing up old messages to persistent storage. Modified to use new functions modified in GridConnection. Modified the receive function to make incoming packet dequeing and sleeping an atomic operation if the packet returned is null. This is necessary because if an enqueue operation is occurring at the same time as a dequeue operation, there may be a readers-writers problem.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
GridTcpClientTest.java	New	Tests changes to GridTcp.	Medusa: /home/uwagent/agentteamwork-dev/tests/
GridTcpServerTest.java	New	Tests changes to GridTcp.	Medusa: /home/uwagent/agentteamwork-dev/tests/
GridUtil.java	Old	Added a simple method that retrieves the logon name of the current user. This is used when storing backup messages to disk. (SINCE REMOVED) Added a new function that prints all active threads currently running within the JVM for debugging purposes.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/GridTcp/
IRecvThread.java	Old	Reformatting, comments, and javadoc	Medusa: /home/uwagent/agentteamwork-dev/MPJ/
ISendThread.java	Old	Reformatting, comments, and javadoc	Medusa: /home/uwagent/agentteamwork-dev/MPJ/
javadoc	New	Javadoc for all of AgentTeamwork	Medusa: /home/uwagent/agentteamwork-dev/doc/
JGF tests	Old	Eliminated the use of the jgfutil package so that it would run correctly. Also ported PingPongBench and AllgatherBench to Ateam programs.	Medusa: /home/uwagent/agentteamwork-dev/MPJ/JGF/
JGFMaster.sh, JGFSlave.sh	New	Runs JGF tests.	Medusa: /home/uwagent/agentteamwork-dev/JGF/
Misc. Script Files	New	For backup, javadoc generation, and compilation of AgentTeamwork	Medusa: /home/uwagent/agentteamwork-dev/scripts
Misc. Test Files	New	For testing serialization and package compilation issues.	Medusa: /home/uwagent/agentteamwork-dev/tests/
Mpjrun.java	Old	Reformatting, comments, and javadoc. Also changed	Medusa: /home/uwagent/agentteamwork-dev/MPJ/

		parameters parsing to look for new versions of parameters. (i.e. –slave instead of –amslave)	
RAFPartitionInfo.java	New	Contains information about a file partition used in Ateam's RandomAcessFile	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
RandomAccessFile.java	New	A partitioned, distributed RandomAccessFile that uses Ateam for communication.	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
RandomAccessFileCommThread.java	New	Aids in communication between nodes that share a RandomAccessFile	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
Request.java	Old	Reformatting, comments, and javadoc. Fixed Illegal Monitor State bug.	Medusa: /home/uwagent/agentteamwork-dev/MPJ/
runCommunicatorTest.sh	New	Launches CommunicatorTest	Medusa: /home/uwagent/agentteamwork-dev/tests/
ServerSocket.java	New	Wraps GridServerSocket.java	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
Socket.java	New	Wraps GridSocket.java	Medusa: /home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/
UPWTest.java	New	Now tests Ateam by extending the AteamProg class.	Medusa: /home/uwagent/agentteamwork-dev/tests/
UserProgWrapper.java	Old	Added support for new and old versions of AgentTeamwork (i.e.-partitioned and non-partitioned). Added support for AteamProg class as well as instantiation of GridTcp for Ateam programs. Added support for AteamProg class as well as instantiation of GridTcp for Ateam programs. Also added a new parameter for main that accepts a port number to use when instantiating GridTcp.	Medusa: //home/uwagent/agentteamwork-dev/AgentTeamwork/Ateam/

Table 1. Files created or modified in Phases 1 - 7

Future Project Recommendations

While working on AgentTeamwork I have compiled a short cumulative list of some project recommendations that future contributers might implement. They are:

- Reformatting and commenting of Communicator.java
- Creation of an MPJException class
- Argument checking and informative exception details for MPJ communication methods. (e.g. – If the user calls Reduce() and receiveBuffer.length < recvCount + recvOffset, an MPJException is thrown in which this problem is explained).
- I recommend that at some point, large packets be fragmented. This will alleviate many of the memory issues that occur within GridTcp. Then, the memory threshold for GridTcp should automatically adjust to be the closest multiple of this packet size. I don't think it would be too difficult to implement.

- It may be possible (further research would be necessary) to provide an additional memory safeguard in GridTcp that would automatically kick in packet backup to disk when the available memory nearly reaches 0.
- Much more advanced and deeper testing of GridTcp's new flow control feature
- Implementation of more advanced synchronization features for Ateam's RandomAccessFile

Appendix A – RandomAccessFile Functionality Test 1

This is test was run between four nodes and runs almost all RandomAccessFile operations on a 256 byte file. Note that the exceptions depicted are caused by a bug in GridTcp. The filetype used in this test is simply four bytes where bytes 0, 4, 8, … 252 are given to node 0, bytes 1, 5, 9, … 253 are given to node 1, bytes 2, 6, 10, … 254 are given to node 2 and bytes 3, 7, 11, … 255 are given to node 3.

Medusa

```
UPW: rank=0 dest=medusa gtwy=null
UPW: rank=1 dest=mnode14 gtwy=null
UPW: rank=2 dest=mnode15 gtwy=null
UPW: rank=3 dest=mnode16 gtwy=null
UPW: starts user program main
Completed constructor
Test 1:  File length: 256
Test 2:  Read local partition data
0:4:8:12:16:20:24:28:32:36:40:44:48:52:56:60:64:68:72:76:80:84:88:92:96:100:104:108:112:1
16:120:124:128:132:136:140:144:148:152:156:160:164:168:172:176:180:184:188:192:196:200:20
4:208:212:216:220:224:228:232:236:240:244:248:252:
Test 3:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:
Test 4:  Write local partition data
Test 5:  Read local partition data
0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0
:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
Test 6:  Read the whole file
0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0
:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:
1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1
:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:
2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2
:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:0:1:2:3:
Test 7:  Write whole file
Test 8:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
```

```
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:
Test 9: Truncate the fileLength = 20
Test 10:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:
Test 11: Extend the file size
Length = 256
Test 12:  Write whole file
Test 13:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:
Test 14: Append an extra twenty bytes
Test 15:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:
UPW.launchUserProgMain : java.lang.reflect.InvocationTargetException
Cause : java.lang.NullPointerException
If this is InvocationTargetException a user program itself caused an exception
UPW.main : java.lang.NullPointerException
Cause : null
UPW.main: end
```

Mnode 14, Mnode 15, Mnode 16

```
UPW: rank=1 dest=mnode14 gtwy=null
UPW: rank=0 dest=medusa gtwy=null
UPW: rank=2 dest=mnode15 gtwy=null
UPW: rank=3 dest=mnode16 gtwy=null
UPW: starts user program main
Completed constructor
Test 1:  Read local partition data
1:5:9:13:17:21:25:29:33:37:41:45:49:53:57:61:65:69:73:77:81:85:89:93:97:101:105:109:113:1
17:121:125:129:133:137:141:145:149:153:157:161:165:169:173:177:181:185:189:193:197:201:20
5:209:213:217:221:225:229:233:237:241:245:249:253:
Test 4:  Write local partition data
Test 5:  Read local partition data
1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1
:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:1:
Test 8:  Read the whole file
0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:
Test 9: Truncate the fileLength = 20
Test 13: Extend the file size
Length = 256
Test 15:  Read the whole file

0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30:31:32:
33:34:35:36:37:38:39:40:41:42:43:44:45:46:47:48:49:50:51:52:53:54:55:56:57:58:59:60:61:62
:63:64:65:66:67:68:69:70:71:72:73:74:75:76:77:78:79:80:81:82:83:84:85:86:87:88:89:90:91:9
2:93:94:95:96:97:98:99:100:101:102:103:104:105:106:107:108:109:110:111:112:113:114:115:11
6:117:118:119:120:121:122:123:124:125:126:127:128:129:130:131:132:133:134:135:136:137:138
```

```
:139:140:141:142:143:144:145:146:147:148:149:150:151:152:153:154:155:156:157:158:159:160:
161:162:163:164:165:166:167:168:169:170:171:172:173:174:175:176:177:178:179:180:181:182:1
83:184:185:186:187:188:189:190:191:192:193:194:195:196:197:198:199:200:201:202:203:204:20
5:206:207:208:209:210:211:212:213:214:215:216:217:218:219:220:221:222:223:224:225:226:227
:228:229:230:231:232:233:234:235:236:237:238:239:240:241:242:243:244:245:246:247:248:249:
250:251:252:253:254:255:0:1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:
UPW.launchUserProgMain : java.lang.reflect.InvocationTargetException
Cause : java.lang.NullPointerException
If this is InvocationTargetException a user program itself caused an exception
UPW.main : java.lang.NullPointerException
Cause : null
UPW.main: end
AgentTeamwork.Ateam.RandomAccessFile: Error occured in communication thread.
```

Appendix B – RandomAccessFile Functionality Test 2

This test was run between four nodes. A master node calls all of *RandomAccessFile*'s *DataOutput* methods (i.e. – *writeLong*) while the three slave nodes read the data in the same order written (via *readLong*). To ensure that a slave node does not read before the master node has completed writing, *RandomAccessFile*'s *barrier* method is used extensively. The filetype used in this test is simply four bytes where bytes 0, 4, 8, … 252 are given to node 0, bytes 1, 5, 9, … 253 are given to node 1, bytes 2, 6, 10, … 254 are given to node 2 and bytes 3, 7, 11, … 255 are given to node 3.

Medusa

```
UPW: rank=0 dest=medusa gtwy=null
UPW: rank=1 dest=mnode14 gtwy=null
UPW: rank=2 dest=mnode15 gtwy=null
UPW: rank=3 dest=mnode16 gtwy=null
UPW: starts user program main
Completed constructor
Test 1: writing boolean: true
Test 2: writing byte: -101
Test 3: writing char: Q
Test 4: writing double: 0.00939
Test 5: writing float: 0.134
Test 6: writing int: 345
Test 7: writing long: 3838308
Test 8: writing short: 500
Test 9: writing unsigned byte: 230
Test 10: writing unsigned short: 709
Test 11: writing UTF: test
Test 12: writing string as bytes: woohoo

Test 13: writing 10 bytes of: 12
UPW.launchUserProgMain : java.lang.reflect.InvocationTargetException
Cause : java.lang.NullPointerException
If this is InvocationTargetException a user program itself caused an exception
UPW.main : java.lang.NullPointerException
Cause : null
UPW.main: end
```

Mnode 14, Mnode 15, Mnode 16

```
UPW: rank=1 dest=mnode14 gtwy=null
UPW: rank=0 dest=medusa gtwy=null
UPW: rank=2 dest=mnode15 gtwy=null
UPW: rank=3 dest=mnode16 gtwy=null
UPW: starts user program main
Completed constructor
Test 1: reading boolean: true
Test 2: reading byte: -101
Test 3: reading char: Q
Test 4: reading double: 0.00939
```

```
Test 5: reading float: 0.134
Test 6: reading int: 345
Test 7: reading long: 3838308
Test 8: reading short: 500
Test 9: reading unsigned byte: 230
Test 10: reading unsigned short: 709
Test 11: reading UTF: test
Test 12: reading line: woohoo
Test 13: reading 10 bytes fully: 12121212121212121212
```

References

1. MPI-IO Standard
   [http://www.mhpcc.edu/training/workshop2/mpi_io/MAIN.html]
2. Java I/O RandomAccessFile
   [http://java.sun.com/j2se/1.4.2/docs/api/java/io/RandomAccessFile.html]