AgentTeamwork Programming Manual

Munehiro Fukuda[*]

Miriam Wallace[*]

[*] Computing and Software Systems, University of Washington, Bothell

AgentTeamwork Programming Manual

Table of Contents

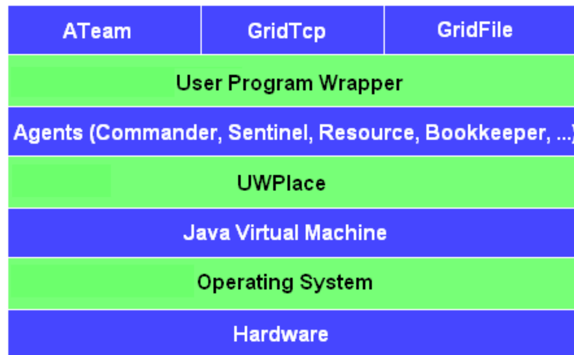
Table of Contents 2
Table of Figures 2
2. Programming Model 3
3. Snapshots 5
  3.1 Using the Snapshot Function 5
  3.2 Recommendations 6
4. Sockets 6
5. MPI-Java 7
6. Files 8
7. C/C++ Programming 8
  7.1 C/C++ Program Flow 8
  7.2 MPI 9
8. Final Comments 9
Appendix A: Code 10
Appendix B: C/C++ MPI Reference 12

Table of Figures

Figure 1. AgentTeamwork implementation layers 3
Figure 2. A sample Agent deployment tree 4

1. Introduction

The AgentTeamwork grid-computing software allows you to program a multi-processor, multi-system application for execution on a cluster (or clusters) within the safety of a fault-tolerant and resource-aware environment. All potential computing nodes which as registered with the XML resource Database can be accessed by AgentTeamwork in the execution of your application.



The ATeam class provides a portal into the power and flexibility of the AgentTeamwork system by providing a class inside which users can develop any number of applications to perform myriad different computational intensive jobs.

The implementation layers provided by the AgentTeamwork system, pictured below in *Figure 1*, illustrate the framework of this relationship in a generic computing node.

Figure 1. AgentTeamwork implementation layers

The UWPlace daemon running on a computing node allows the mobile agents to migrate to that location and execute tasks. Agents perform all the various function of the system from agent and resource management to user program execution via the User Program Wrapper. The GridTcp and GridFile layers allow for user programs to make use programming tools such as Java Sockets and MPI in their program.

2. Programming Model

When a user creates ATeam program to run using the AgentTeamwork framework and deploys it to run on the AgentTeamwork grid, AgentTeamwork first deploys a commander agent which over sees the creation and deployment of bookkeeping and system agents. These agents allow AgentTeamwork to respond to network load changes and crashes dynamically, without the need for user code to account for these anomalies.

Once the AgentTeamwork system agents have been initiated, the user program is initialized and distributed to child nodes for execution. As shown in *Figure 2*, where system agents are colored in green and user program executing nodes are in blue, Resource and Bookkeeper agents may both spawn children in order to assist with their system functions.

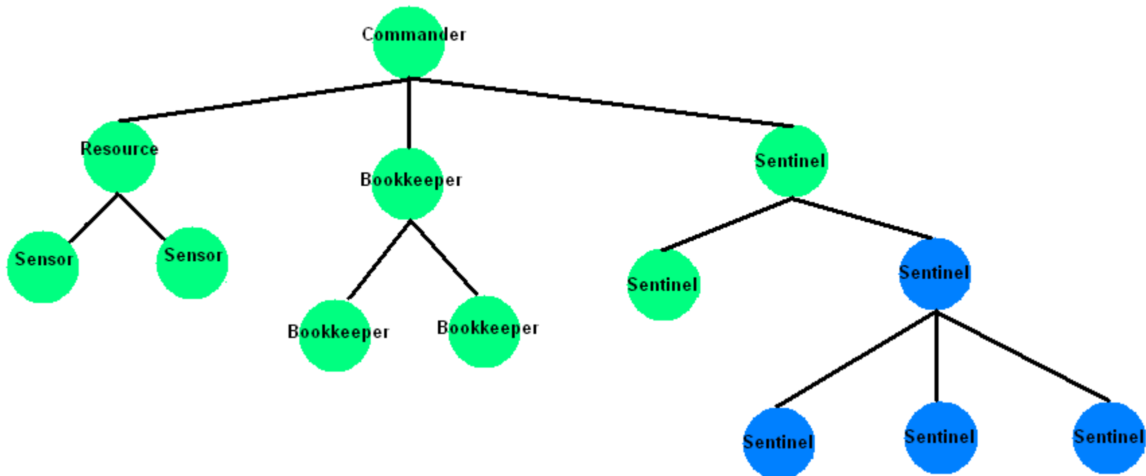


Figure 2. A sample Agent deployment tree

AgentTeamwork provides a User Program Wrapper which abstracts GridTcp and grid file needs apart from the user program, allowing the user freedom to use Socket-based or MPI-based programs and have the flexibility of both Java standard FileInputStream/OutputStream and RandomAccessFile features.

In creating a user program using the ATeam framework, import and extend the ATeam class, as normal. The main method should instantiate and initiate the execution of your class, also as normal, but the main body of the program should be placed in a class function outside of the main method (see the *ATeam sample code* table below).

ATeam sample code – See appendix for complete code

```
import AgentTeamwork.Ateam.*;
import MPI.*;

public class UserProgAteam extends AteamProg {
    ...
    // real const
    public UserProgAteam( String[] args ) {
      phase = 0;
    }
    ...
    // application body
    private void compute( ) {
         System.out.println( "Hello" );
    }
    // start from here
    public static void main( String[] args ) {
      MPJ.Init( args, ateam);
       UserProgAteam program = null;
       ...
      program = new UserProgAteam( args );
      ateam.registerLocalVar( "program", program );
       ...
      program.compute( );
      MPJ.Finalize( );
    }
```

```
}
```

In Socket-based user programs, user may communicate directly with other programs without worrying about what port they are using, as the actual ports are automatically assigned and managed by the AgentTeamwork execution structure.

Similarly, MPI-based programs may take advantage of most MPI features as normal and AgentTeamwork automatically establishes communication connection between all user processes, creating a perfect Java MPI network.

File I/O for the user perspective is also very similar to normal Java programming with a few small changes. Files are distributed quickly and efficiently to agents via a copying and sending throughout the tree structure. By allowing the user to user either FileInput/OutputStreams or RandomAccessFiles AgentTeamwork assures usability to the programmer.

3. Snapshots

In order to recover from crashes, the ATeam class can take snapshots of the current state of the program. These records are then transmitted to another node which is responsible for storing snapshots should they be needed in the case of a program crash. This allows an ATeam program to resume from the most recent snapshot whenever a crash occurs.

3.1 Using the Snapshot Function

The snapshot function takes advantage of Java Serialization and therefore, data and functions to be saved should either be a data member of the class, or be registered using the register method. Names used to register functions and data for cataloguing must be unique, as they are stored in a hash table.

While snapshot functionality may be used anywhere in an ATeam program, taking a snapshot does require a significant amount of overhead and therefore it should be used only as needed for the specific ATeam program. Please see the following code sample for a reference on how to make a snapshot in your program.

Snapshot sample code – See appendix for complete code

```
...
public class UserProgAteam extends AteamProg {
     ...
     // application body
     private void compute( ) {
       for ( phase = 0; phase < 10; phase++ ) {
            try {
              Thread.currentThread( ).sleep( 1000 );
            } catch ( InterruptedException e ) {
            }

            ateam.takeSnapshot( phase );
            System.out.println( "UserProgAteam at rank " +
                         MPJ.COMM_WORLD.Rank( ) +
                         " : took a snapshot " +
                         phase );
```

```
    }
    ...
}
```

3.2 Recommendations

How often to snapshot a given program varies widely based on the program itself, however, a couple of general recommendations can be made.

Programs which have growing data sets (frequently calling new) can benefit from more frequent snapshots in order to prevent loss of data. Also benefiting from taking more frequent snapshots are programs which send and receive messages frequently, as messages are saved in memory between snapshots.

4. Sockets

ATeam `GridSockets` are very similar to the common Java Socket API except that their constructors require some different arguments for instantiation of the class. Instead of passing the IP to a Socket constructor pass the rank of the process (indicated by `myRank`) which specifies which process is establishing the socket. The port required by the constructor can be set to any value as is it a virtual port only.

Communications using these ports will be funneled through the GridTcp which has a wrapper around the actual port chosen by the GridTcp layer. Because they are virutual, port numbers should be used to idenfity different connections within your program, without a connection to the actual ports used by ATeam program communications. ATeams GridTcp Socket-based communications are an error recoverable implementation.

The code below illustrates how the usage of `GridSockets` is nearly identical to that of standard Java Sockets.

Socket sample code – within some function body

```
import AgentTeamwork.Ateam.GridTcp.*;
...
private final int port = 2000;
private GridSocket socket;
private GridServerSocket server;
private InputStream input;
private OutputStream output;
...
// client
for ( int i = start; i < start + trans; i++ ) {
    try {
        output.write( i % 128 );
    } catch ( IOException e ) {
    }
    System.out.println( "Sockets with " + myRank + ": "
                        + " output[" + i + "]=" + i % 128 );
}
...
// server
```

```
for ( int i = start; i < start + trans; i++ ) {
    try {
        System.out.println( "Sockets with " + myRank + ": "
                                + " input[" + i + "]=" +
input.read( ) );
    } catch ( IOException e ) {
    }
}
...
```

5. MPI-Java

User programs run with AgentTeamwork may take advantage of Java's MPI API. AgentTeamwork's MPI specification differs somewhat from the common MPI-Java specification in that it has an additional feature called check pointing and the arguments passed to the constructor are slightly different.

Despite these differences, AgentTeamwork's MPI has the important `myRank` and `nprocess` members which store the current process's rank and the total number of MPI (user-based) processes, respectively.

Additionally, when MPI-Java is used AgentTeamwork automatically establishes a perfect MPI network for the purpose of inter-process communication with every rank having a direct line of communication to every other rank. These communication links between all ranks are directly established by `MPI.Init` which relies on GridTcp for the realization of the connections. This additional functionality is made possible by an additional argument passed to the `MPI.Init(args, ateam)` function (see the below code sample for usage within a program). The "`ateam`" argument allows the MPI implementation to be aware of processes, ram consumed, and other necessary program parameters used by the user's ATeam program.

MPI sample code – See appendix for complete code

```
import AgentTeamwork.Ateam.*;
import MPI.*;

public class UserProgAteam extends AteamProg {
    ...
    // application body
    private void compute( ) {
        ...
    }
    // start from here
    public static void main( String[] args ) {
        MPJ.Init( args, ateam );
        ...
        program.compute( );
        MPJ.Finalize( );
    }
}
```

As with Socket-based communication, the implementation of MPI-Java via the GridTcp layer allows MPI-based communication to be error-recoverable.

6. Files

Java's FileInputStream/OutputStream can be used according to the original API in an ATeam program. These files handled by the grid File layer, which uses AgentTeamwork's tree structure to propagate copies of the files to the agent processes that need them, until each agent has a copy of the file.

Java's RandomAccessFile (RAF) can also be used according to the original API in an ATeam program with one minor exception. RAFs have an additional method `barrier` which allows processes to share file stripes with other processes but only once the local/current process has completed any modifications to its file stripes.

File stripes passed to each process can be highly specified by through submitGUI, allowing for greater control over the distribution of data and tasks within a given ATeam program.

7. C/C++ Programming

User may write ATeam programs using C or C++, however, program flow and MPI in this are slightly different from the Java counter part.

7.1 C/C++ Program Flow

An ATeam program executed in the AgentTeamwork platform requires that the user provide an additional file "libJavaToCpp.so" which is the compiled C/C++ program in a library format usable by the AgentTeamwork system. This file is read into the systems local disk and stored in the temp directory because it cannot be executed dynamically as a Java program would be.

JavaToCpp.cpp is then loaded into memory using dynamic linking. This program then reads the C/C++ program to memory. The JavaToCpp.cpp program sets up the complex JNI specific notations and set up allowing the user program to remain uncluttered. Essentially this process adds another wrapper to the ATeam software layer allowing AgentTeamwork to operate on C and C++ programs as well as Java.

To compile a C/C++ program for execution with AgentTeam work, use a version of the following script, which was written to compile the Helloworld program in the following section.

C/C++ compile.sh – Helloworld.cpp

```
#!/bin/sh

rm -f *.class
javac -classpath MPJ.jar:Ateam.jar:. *.java
# jar cvf GridJNI.jar *.class
jar -cvf GridJNI.jar *.class
javah -jni JavaToCpp
g++ -rdynamic JavaToCpp.cpp -o _libJavaToCpp.so_ -shared -ldl
g++ -shared -o _libHelloWorld.so_ GridJNI_library.cpp
HelloWorld.cpp
```

7.2 MPI

Although C and C++ programs are supported, MPI syntax for both languages is not supported. Due to the increased simplicity of MPI implementation for C (MPIsend() and MPIreceive() functions) as opposed to C++, AgentTeamwork only supports the use of the C MPI API. Below is a sample C/C++ program using MPI function, see Appendix B for list of C/C++ MPI extern functions.

C/C++ MPI sample code – Helloworld.cpp

```
#include <iostream>
using namespace std;
typedef int MPI_Request, MPI_Status, MPI_Comm;

extern void takeSnapshot(int argc);
extern int MPI_Init(int* argc, char*** argv);
extern void MPI_Finalize();
extern int MPI_Comm_rank(MPI_Comm comm, int *rank);
extern int MPI_Comm_size(MPI_Comm comm, int *size);

int main(int argc, char** argv) {
  cerr << "main" << endl;
  cerr << "argc = " << argc << endl;
  cerr << "argv[0] = " << argv[0] << endl;
  cerr << "argv[1] = " << argv[1] << endl;
  MPI_Init(&argc, &argv);
  cout << "MPI Init Successful!" << endl;
  cout << "[HelloWorld.cpp]Calling Rank() and Size()" << endl;
  int rank, size;
  MPI_Comm_rank(0,&rank);
  MPI_Comm_size(0,&size);
  cout << "[HelloWorld.cpp]Rank = " << rank << endl;
  cout << "[HelloWorld.cpp]Size = " << size << endl;
  cerr << "Calling MPI_Finalize()" << endl;
  MPI_Finalize();
  cerr << "finished" << endl;
}
```

8. Final Comments

AgentTeawork, ATeam and associated classes are copy write University of Washington Bothell. No advanced notice of changes and revisions to AgentTeamwork and ATeam or relate classes are required. Users may use classes and associated methods at their own risk.

Appendix A: Code

ATeam sample code – Complete

```
import AgentTeamwork.Ateam.*;
import MPJ.*;

public class UserProgAteam extends AteamProg {

    private int phase;

    // blank const for Ateam
    public UserProgAteam( Ateam o ) { }

    public UserProgAteam( ) {
    }

    // real const
    public UserProgAteam( String[] args ) {
     phase = 0;
    }

    // phase recovery
    private void userRecovery( ) {
     phase = ateam.getSnapshotId( );
    }

    // application body
    private void compute( ) {
     for ( phase = 0; phase < 10; phase++ ) {
         try {
          Thread.currentThread( ).sleep( 1000 );
         } catch ( InterruptedException e ) {
         }

         ateam.takeSnapshot( phase );
         System.out.println( "UserProgAteam at rank " +
                     MPJ.COMM_WORLD.Rank( ) +
                     " : took a snapshot " +
                     phase );
     }
    }

    // start from here
    public static void main( String[] args ) {
     System.out.println( "UserProgAteam: got started" );

     MPJ.Init( args, ateam);

        UserProgAteam program = null;
     // Timer timer = new Timer( );
     if ( ateam.isResumed( ) ) {
```

```
          program = ( UserProgAteam )
ateam.retrieveLocalVar( "program" );
          program.userRecovery( );
      } else {
          program = new UserProgAteam( args );
          ateam.registerLocalVar( "program", program );
      }
      program.compute( );

      MPJ.Finalize( );
    }
}
```

Appendix B: C/C++ MPI Reference

All functions below can be used in a user's C/C++ ATeam MPI-based program and all of the must be prefaced with the keyword "`extern`".

```
void takeSnapshot(int argc)

bool isResumed()

int getSnapshotId()

int MPI_Init(int* argc, char*** argv)

void MPI_Finalize()

int MPI_Send( void* buf, int count, MPI_Datatype datatype, int
              dest, int tag, MPI_Comm comm. )

int MPI_Recv( void* buf, int count, MPI_Datatype datatype, int
              dest, int tag, MPI_Comm comm, MPI_Status *status )

int MPI_Comm_rank(MPI_Comm comm, int *rank)

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Bcast( void* buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm )

int MPI_Pack( void* inbuf, int incount, MPI_Datatype datatype,
              void *outbuf, int outsize, int *position, MPI_Comm
              comm )

int MPI_Unpack( void* inbuf, int insize, int *position, void
                *outbuf, int outcount, MPI_Datatype datatype,
                MPI_Comm comm )

int MPI_Reduce( void* sendbuf, void* recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root,
                MPI_Comm comm )

int MPI_Barrier(MPI_Comm comm)

int MPI_Gather( void* sendbuf, int sendcount, MPI_Datatype
                sendtype, void* recvbuf, int recvcount,
                MPI_Datatype recvtype, int root, MPI_Comm comm )

int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype
                 sendtype, void* recvbuf, int recvcount,
                 MPI_Datatype recvtype, int root, MPI_Comm comm )

int MPI_Allgather( void* sendbuf, int sendcount, MPI_Datatype
```

```
                   sendtype, void* recvbuf, int recvcount,
                   MPI_Datatype recvtype, MPI_Comm comm )

int MPI_Alltoall( void* sendbuf, int sendcount, MPI_Datatype
                  sendtype, void* recvbuf, int recvcount,
                  MPI_Datatype recvtype, MPI_Comm comm )

int MPI_Allreduce( void* sendbuf, void* recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm
                   comm )

int MPI_Reduce_scatter( void* sendbuf, void* recvbuf, int
                        *recvcounts, MPI_Datatype datatype,
                        MPI_Op op, MPI_Comm comm )

int MPI_Isend( void* buf, int count, MPI_Datatype datatype, int
               dest, int tag, MPI_Comm comm, MPI_Request
               *request )

int MPI_Irecv( void* buf, int count, MPI_Datatype datatype, int
               source, int tag, MPI_Comm comm, MPI_Request
               *request )

int MPI_Scatterv( void* sendbuf, int *sendcounts, int *displs,
                  MPI_Datatype sendtype, void* recvbuf, int
                  recvcount, MPI_Datatype recvtype, int root,
                  MPI_Comm comm )

int MPI_Gatherv( void* sendbuf, int sendcount, MPI_Datatype
                 sendtype, void* recvbuf, int *recvcounts, int
                 *displs, MPI_Datatype recvtype, int root,
                 MPI_Comm comm )

int MPI_Allgatherv( void* sendbuf, int sendcount, MPI_Datatype
                    sendtype, void* recvbuf, int *recvcounts, int
                    *displs, MPI_Datatype recvtype, MPI_Comm
                    Comm )

int MPI_Alltoallv( void* sendbuf, int *sendcounts, int *sdispls,
                   MPI_Datatype sendtype, void* recvbuf, int
                   *recvcounts, int *rdispls, MPI_Datatype
                   recvtype, MPI_Comm comm )

int MPI_Gatherv( void* sendbuf, int sendcount, MPI_Datatype
                 sendtype, void* recvbuf, int *recvcounts, int
                 *displs, MPI_Datatype recvtype, int root,
                 MPI_Comm comm )
```