Enhancing Communication and File I/O in AgentTeamwork

University of Washington, Bothell

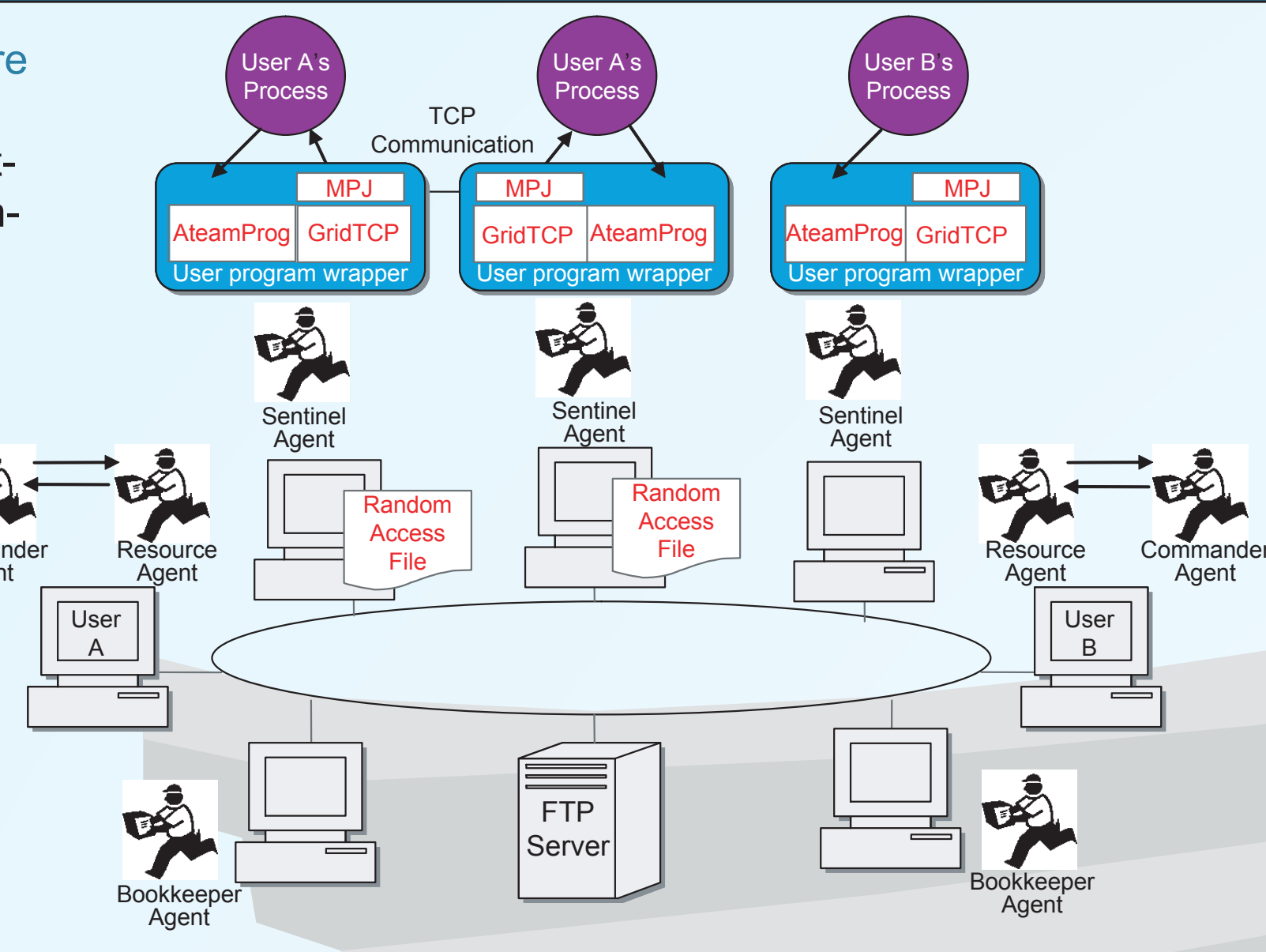
Computing and Software Systems 497 Colloquium 2007

Introducing AgentTeamwork

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.

The purpose of this project was to make AgentTeamwork easier to use and increase the simplicity, capability, and efficiency of its I/O components.

Java user applications	
mpiJava API	
mpiJava-S	mpiJava-A
Java socket	GridTcp
User program wrapper	
Commander, resource, sentinel, and bookkeeper agents	
UWAgents mobile agent execution platform	
Operating systems	



AgentTeamwork's execution layers (left) and system overview (above). AgentTeamwork components relative to this project are labled in RED.

Simplifying a User Program

Originally, creating an AgentTeamwork user program was somewhat clumsy. The user was required to define specific member variables for use by AgentTeamwork's user program wrapper. Also, Java does not allow access to a process's program counter, so for a desiginated mobile agent to truly capture a user program's execution state, it was necessary to partition the program into discreet functions (e.g. - func_0, func_1, etc.). The user program wrapper would then call each function consecutively after first generating a snapshot.

To simplify AgentTeamwork's user programs, *AteamProg was created*: an abstract class that, if extended, supports a user program by:

- Eliminating the need to explicitly define required AgentTeamwork members.
- Eliminating the need to partition a user program for snapshots.
- Eliminating the need to use specialized I/O classes. GridFile streams and GridTcp connections have been wrapped with standard I/O names and methods.
- Providing user-initiated snapshots through the *takeSnapshot* method.
- Providing non-member variable serialization/deserialization through the *registerLocalVar* and *retrieveLocalVar* methods.

```java
import java.io.*; import AgentTeamwork.Ateam.GridFile.*;
import AgentTeamwork.Ateam.GridTcp.*;
public class MyApplication implements Serializable {
    public GridIpEntry ipEntry[];                       // system required
    public int funcId;                                  // system required
    public GridTcp tcp;                                 // system required
    public GridFile gridfile;                           // system required
    public int nprocess;                                // system required
    public int rank;                                    // system required
    private GridFileInputStream gfis;                   // a user input stream
    private GridSocket gsock;                           // a user socket

    public int func_0( String args[] ) {                // constructor
        gfis = new GridFileInputStream(gridfile);       // create input stream
        gsock = new GridSocket(3,22418, tcp);           // create socket
        ...;
        return 1;                                       // calls func_1( )
    }

    public int func_1( ) {                              // called from func_0
        int data = gfis.read( );                        // read a byte of data
        InputStream is = gsock.getInputStream( );       // create a socket stream
        ...;
        return 2;                                       // calls func_2( )
    }

    public int func_2( ) {                              // called from func_2,
        gfis.close( );                                  // close file stream
        gsock.close( );                                 // close socket
        ...;
        return -2;                                      // application terminated
    } }
```

```java
import AgentTeamwork.Ateam.*;
public class MyApplication extends AteamProg {
    private int phase;                                  // snapshot management
    private FileInputStream fis;                        // a user input stream
    private Socket sock;                                // a user socket
    public MyApplication(Object o){}                    // system reserved

    public MyApplication( ) {                           // user constructor
        phase = 0;
        fis = new FileInputStream( );                   // create input stream
        sock = new Socket(3,22418);                     // create socket
    }

    private void compute( ) {                           // user computation
        int data = fis.read( );                         // read a byte of data
        InputStream is = sock.getInputStream( );        // create a socket stream
        ateam.takeSnapshot(phase);                      // check-pointing
        ...;
        fis.close( );                                   // close file stream
        gsock.close( );                                 // close socket
    }

    private boolean userRecovery( ) {
        phase = ateam.getSnapshotId( );                 // version check
    }

    public static void main( String[] args ) {
        MyApplication program = null;
        if ( ateam.isResumed( ) ) {                     // program resumption
            program = (MyApplication)
            ateam.retrieveLocalVar( "program" );
            program.userRecovery( );
        } else {                                        // program initialization
            MPI.Init( args );                           // javaMPI invoked
            program = new MyApplication( );
            ateam.registerLocalVar( "program", program );
        }
        program.compute( );                             //now go to computation
    }
}
```

An AgentTeamwork user program before *AteamProg* (top) and after (bottom). Note the partitioning and specialized member variables of the old method.

Fixing MPJ

- Debugged errors that caused JGF (Java Grande Forum) Benchmark tests to fail.
- Reformatted code to improve maintainability.
- Generated JavaDoc.

Enhancing GridTcp

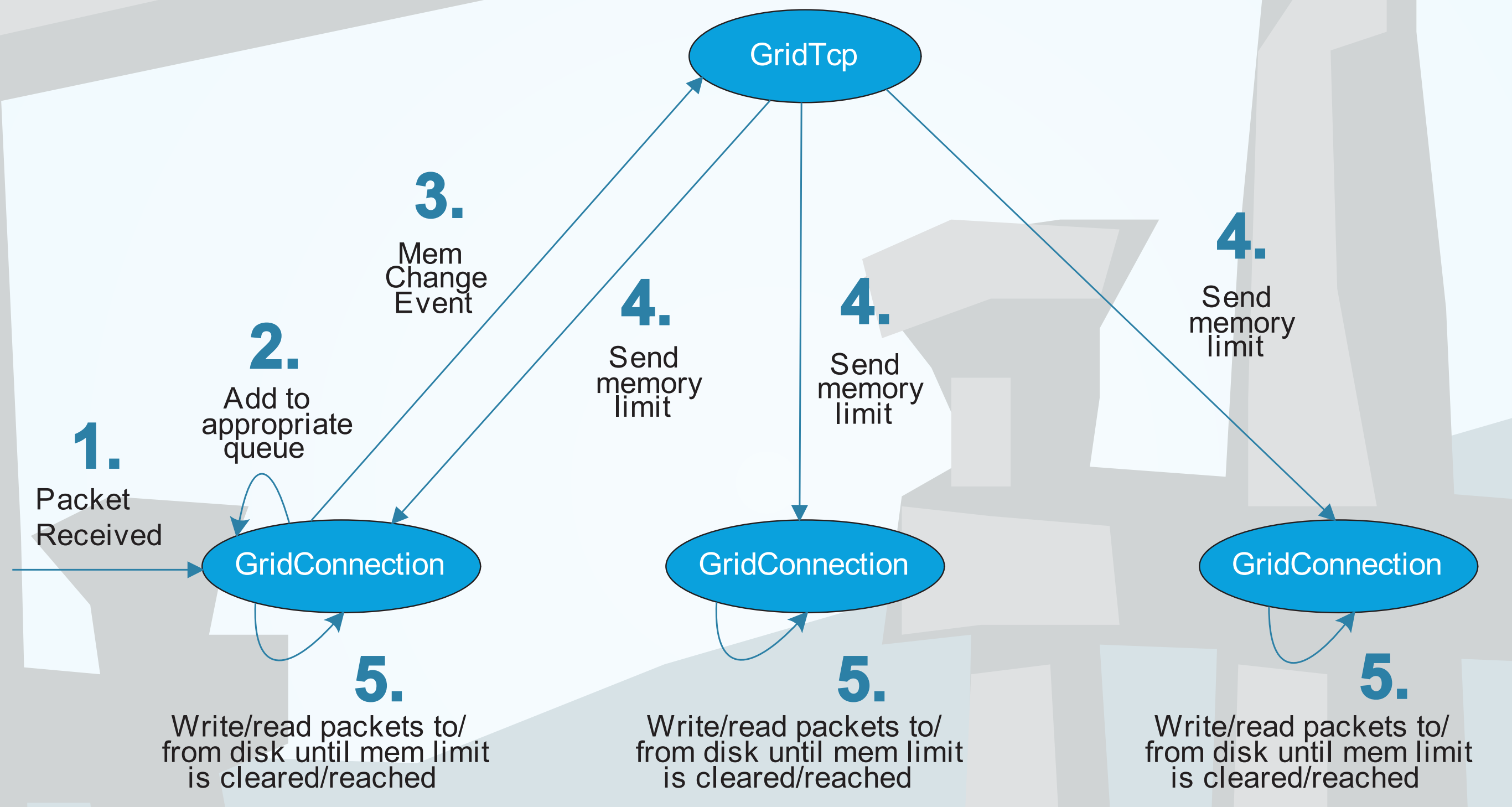
GridTcp is an error-recoverable transmission protocol that continuously saves old and in-transit messages for recovery. The frequency of taking execution snapshots depends on the duration of each func_X in a partitioned user program or the call frequency of *takeSnapshot* in an *AteamProg* user program. Therefore, *GridTcp* often maintained too many old messages in memory which resulted in a quick *OutOfMemory* crash.

To alleviate this problem *GridTcp* was enhanced to:

- Manage its own memory usage by writing any older messages above a specified threshold to disk and loading those messages into memory automatically when the threshold is cleared
- Use a simple flow control that "pauses" a client sending messages when the server becomes overloaded and "resumes" the client when not

Memory Management

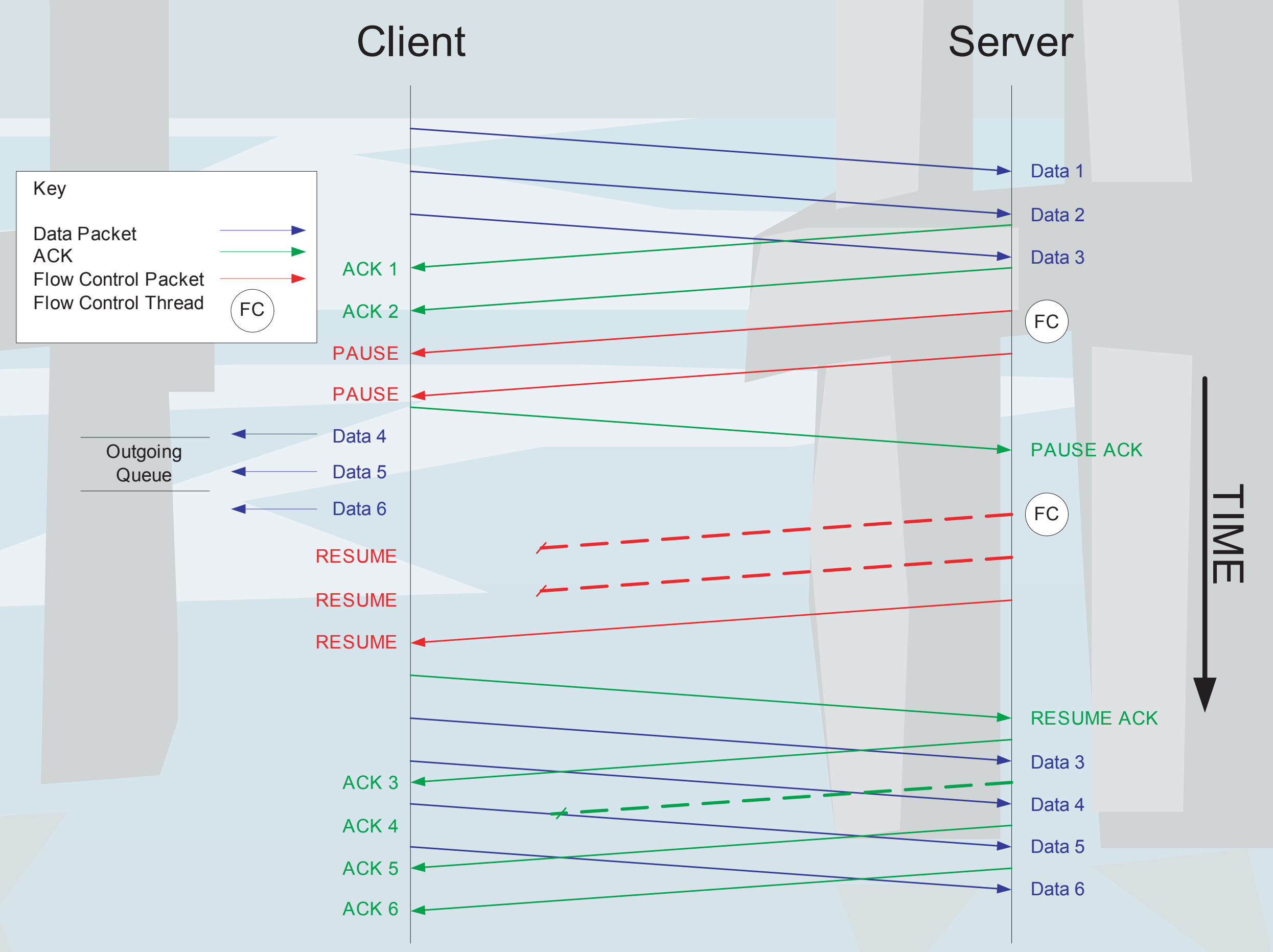
All AgentTeamwork programs have a single *GridTcp* object that manages multiple *GridConnection*s (the core of a *GridSocket*). Each *GridConnection* maintians multiple queues to organize it's messages (e.g. - outgoing, incoming, forwarding and backup). To allow a *GridTcp* object to fairly manage memory between all of its *GridConnection*s without continuously polling them and therefore wasting CPU cycles, memory management was designed with Java *events*. To ensure that on-disk messages are included in snapshots, the *DiskVector* class was created. *DiskVector* behaves like Java's *Vector* class with the exception that its elements are stored in files. *DiskVector*'s serialization/deserialization methods have been overridden to include these files in memory when a snapshot is created.



GridTcp
1. Packet Received
2. Add to appropriate queue
3. Mem Change Event
4. Send memory limit
4. Send memory limit
4. Send memory limit
GridConnection
GridConnection
GridConnection
5. Write/read packets to/ from disk until mem limit is cleared/reached
5. Write/read packets to/ from disk until mem limit is cleared/reached
5. Write/read packets to/ from disk until mem limit is cleared/reached

Flow Control

GridTcp's flow control simply pauses a client's message sending when its memory is overloaded and resumes it when it is not.



Test Results

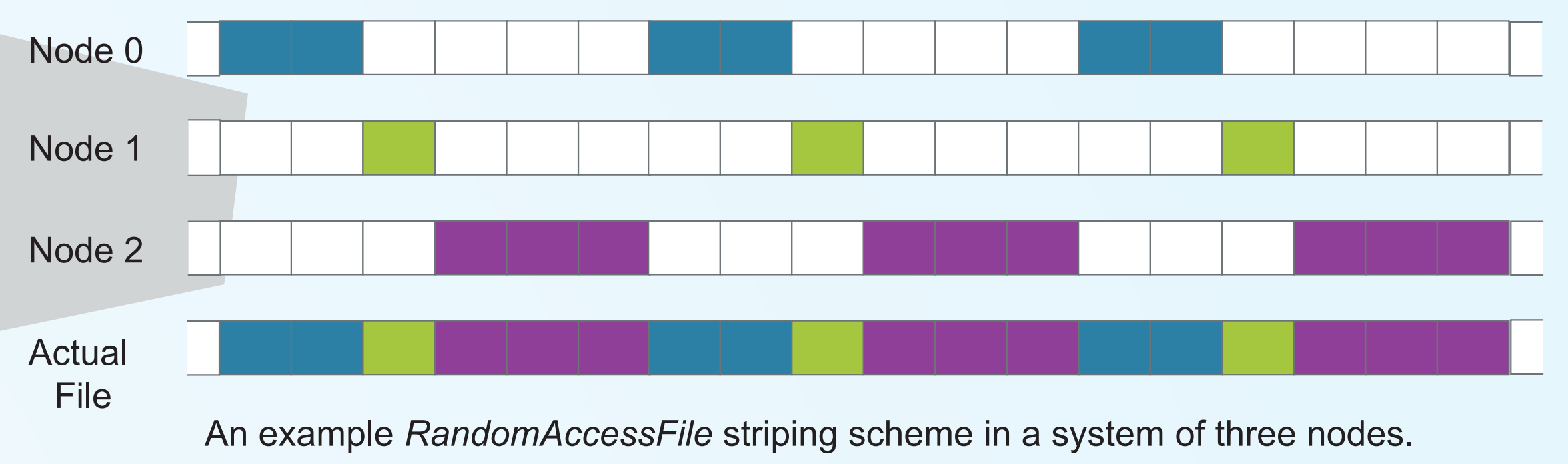
Testing *GridTcp* after the modifications were made proved that the enchancements successfully limited the amount of queued messages to the threshold specified by a user. However, a "memory leak" unrelated to these modifications was discovered that, over time, ocasionally causes a crash. Due to the time contraints of the project and the complexity of locating the leak in AgentTeamwork's many layers, this problem has been left for a future project.

Enhancing File I/O

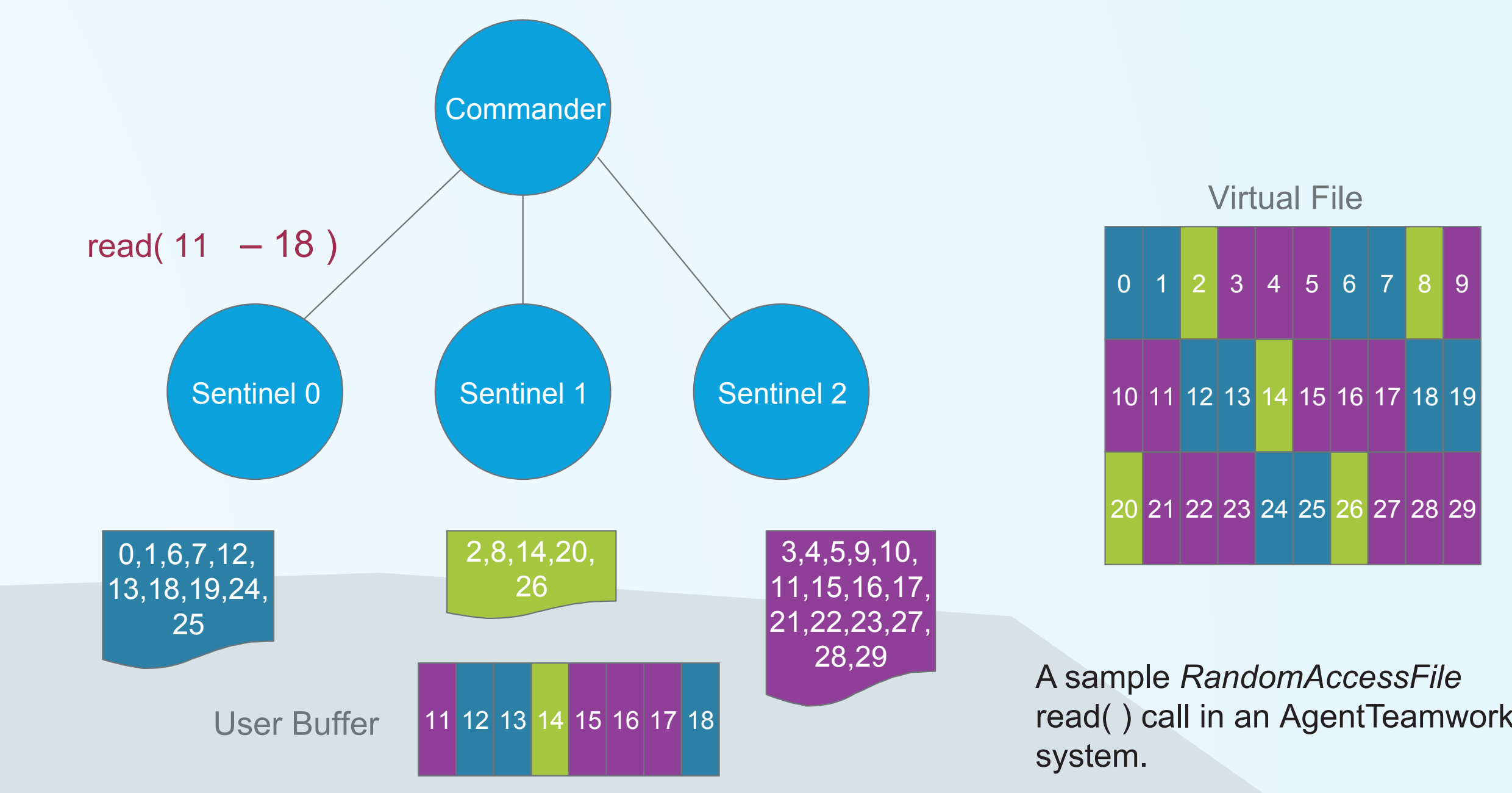
Originally, the only recoverable file I/O that AgentTeamwork provided to a user program was *GridFileInputStream* and *GridFileOutputStream.* These components were limited in that they could only present a user with a *stream* of bytes; that is, random access to the bytes was not possible. AgentTeamwork's *RandomAccessFile* was created to remove this restriction.

RandomAccessFile is a recoverable, distributed file where each node in the system receives only a "stripe" of the whole file yet maintains a virtual view of that 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 API *RandomAccessFile*. If a node requests data that it owns locally, it simply reads from that partition. If it requests or submits data remotely, that data is transferred transparently between nodes. These read/write operations are atomic and to help a user maintain strict ordering, *RandomAccessFile* offers a blocking, collective method called *barrier*.

The striping model (based on the MPI-IO standard's *file view*) allows a node to specify the data they will most likely access the most. This provides quick access to the most heavily used data and significantly reduces the network bandwidth required to send whole, replicated files to every node in the system.



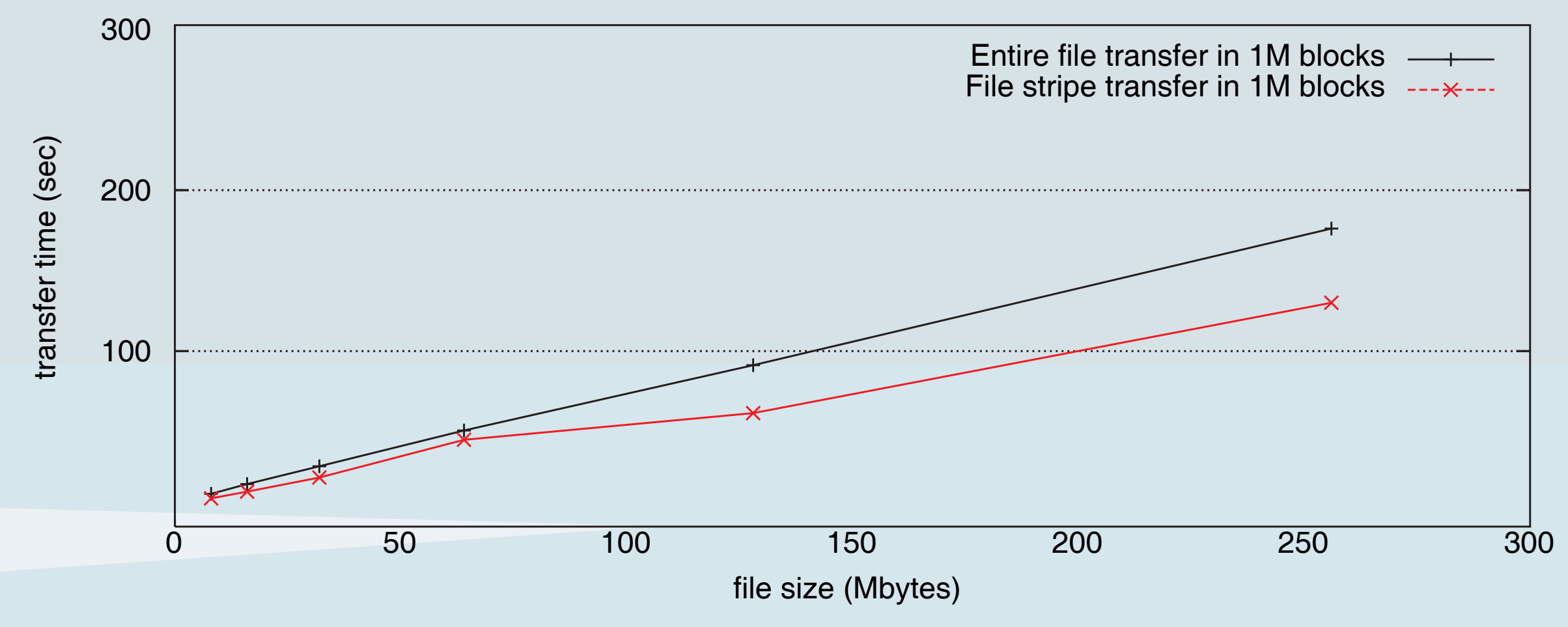
An example *RandomAccessFile* striping scheme in a system of three nodes.



A sample *RandomAccessFile* read( ) call in an AgentTeamwork system.

Test Results

The figure below 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 transfer and Sun NFS respectively when sending a 256 MB random access file. Performance increases with larger file sizes and a larger amount of nodes in the system.



Using Education

Major Technical Skills Applied:

- Parallel programming
- The MPI API
- Multithreaded modeling, programming, and debugging
- Understanding of network stacks and TCP
- OO Programming: interfaces, abstract classes, method overriding, etc.
- Understanding of Java, specificially: Serialization, I/O, streams, the Virtual Machine, packages, compilation, JavaDoc, reflection
- Linux shell scripting
- Technical writing
- Good commenting practices
- Code reading
- Modifying pre-existing, large, complex software systems
- Locating performance bottlenecks / algorithm optimization

Major Core Competencies used:

- Information Gathering
- Systematic Thinking
- Thoroughness
- Creativity
- Learning by Doing
- Collaboration & Team Building
- Leadership
- Writing
- Speaking
- Listening
- Managing Change & Uncertainty
- Decision-Making

Joshua Phillips (jawsh@u.washington.edu)   Dr. Munehiro Fukuda (mfukuda@u.washington.edu)