Distributed Systems Project Introduction

The messaging layer we will be using in this class is designed to provide a framework on top of which students can develop distributed systems. The layer is primarily designed to provide debugging support to the student that is difficult if not impossible to achieve without such a framework. It is provided simply as a place to get started, so students do not need to build both a project and a test harness.

The framework is built so that students can create a distributed system and debug it in a single process, then take the same code and run it on multiple machines. In order to do this, the message layer needs to simulate the same failures and asynchronous behavior that occurs in real distributed systems. This allows failures to be controlled and specified by the user or randomly generated. Furthermore, we provide support for deterministic replay of executions.

1 Interface

The messaging layer provides an interface between student-implemented distributed systems and the underlying hardware. Both the layer and student code that utilizes it are written in Java, although the core messaging layer logic and utilities contained within the edu.washington.cs.cse490h.lib package should not be modified by the student. Instead, the student should create a new, single-threaded Node implementation in the proj/ folder that follows the examples provided. All network communication, command input, timer interrupts, and disc reads/writes should then be handled through the Node interface.

The classes that are of particular importance to the student are the Node, Callback, Utility, and PersistentStorageReader/Writer. Familiarity with these classes is essential, although taking a look at the inner workings of the Simulator may also be helpful.

1.1 Network Communication

Nodes communicate with each other by sending byte arrays to each other with the provided methods in Node. For simplicity, it is fine to hard-code the number of nodes, as long as the way to change the value is clearly documented.

send Sends a byte array across the message layer with a protocol byte for multiplexing. When this message is sent, it is not immediately delivered to the destination. The message actually stays in the message layer, where it is potentially dropped or delayed for an arbitrary amount of time.

broadcast Similar to the send method, except the message is sent to all nodes in the network. The messages can arrive at different times, a subset of the messages can be dropped, and the messages can even be sent to failed nodes. Although many networks do not include broadcast support (e.g. the Internet), Ethernet does, so some distributed systems take advantage of

that fact. Note that the broadcast mechanism does not require or give any information on the part of the node about the other participants on the network.

onReceive This method is called by the manager whenever a packet arrives for this node. Its parameters mirror those of the send methods.

1.2 Commands

The student can enter commands to nodes through either the keyboard or a command file. There are several predefined system commands to handle starts, failures, etc., but the user can also define custom commands. In simulator mode, these have the syntax "[*node virtual address*] *command*" and the student's implementation will need to handle parsing of the commands.

onCommand This method is called by the manager whenever a command arrives for this node. The command is either from user input or a command file, depending on the arguments provided to the manager on startup.

1.3 Timer Interrupts

Timer interrupts are provided to implement timeouts, and the timer uses the notion of rounds that are described below.

addTimer This method actually schedules a timer interrupt. It takes a Callback object and the number of time steps after which the interrupt should fire. The student should familiarize themselves with the Callback class.

1.4 Disc I/O

All disc accesses should be done with the provided classes, which model synchronous and relatively instantaneous reads and writes. In general, discs do not operate in such a way, but this model is much simpler and still valid (e.g. NVRAM and file caching for reads). They extend the standard Java classes BufferedReader and BufferedWriter, so it is worth taking a look at the Javadocs for those two classes.

PersistentStorageWriter This class extends BufferedWriter and has almost the exact same semantics. The main difference is that the class ensures that different nodes cannot access each other's data, even when running on the same machine. In addition, write operations with this class are tied to node failures in order to model all possible failure locations (see below discussion of the failure model).

PersistentStorageReader Similar to the PersistentStorageWriter, this class extends BufferedReader and provides separation of storage locations on nodes. Note that the rules for the superclasses apply to these two classes as well.

2 Failures

Distributed systems introduce a multitude of failures that are not present/very rare in local systems. This framework is designed to introduce failures in the form of arbitrary message drops, delays, and reordering, as well as node halting failures. It is true that some of these failures duplicate functionality (a halt failure is indistinguishable from a string of message drops which is indistinguishable

from a string of greatly delayed packets), but these are convenient building blocks. Furthermore, these failures can be composed to represent everything short of byzantine behavior (which will not be covered in this course).

The probability of each event happening can be specified through static functions of the child classes of Node. This is simply for debugging purposes, and students should experiment with different probabilities. Student code is expected to work in all cases, so relative likelihoods of particular cases are not important in the end, but the ability to set probabilities may prove useful for debugging. Failures can also be specified by the user, by setting the failure level on the command line. The possible values are between 0 and 4, with higher values giving more user control over failures (in increasing order: no user control, crashes/restarts, message drops, delays, and event ordering).

One caveat of the failure model is that failures can happen immediately before any write to the disc. If you imagine a function that contains the following:

```
send m1
A
write to disk
B
send m2
C
```

A failure at C is easy since we can fail after the method returns. A failure at B is also fine since it is equivalent to failing after return, then dropping m2 later. A failure at A, however, is impossible without some way to model intra-method crashes since we want the send of m1, without the write. We take a simplified approach by only checking for crash events at specific locations (between method invocations and before writes) though we do not check for them before other externally visible events that do not affect semantics (e.g. console output). This, however, has the side effect that

```
write('a');
write('b');
newLine();
```

is more likely to cause a fault than

```
write("ab\n");
```

3 Time

The notion of time in the simulator is very weak and in the form of rounds. If more commands are available (which is always the case with user commands), rounds are advanced manually through the time command, otherwise, they are advanced automatically after all outstanding events are processed or delayed. Within a round, events can be arbitrarily ordered (which means two file commands that are not separated by a time command are not guaranteed to execute in the same order they appear in the file).

The reason for these rounds is that, if you ignore commands and timeouts, message deliveries and failures can be ordered in an arbitrary fashion. In fact, with only messages and failures/starts, a notion of time is not required since you can choose any in-flight message, failure or restart to execute. But with timeouts and commands, all of a sudden there are constraints on what can execute (a

timer interrupt at t=1 should execute before a timer at t=100 for a given node). Timeouts and commands must execute in a particular round, while messages can be delayed an arbitrary number of rounds, crashes/restarts can occur in any round, and things in the same round can be arbitrarily reordered.

4 Debugging

4.1 Deterministic Replay

In real distributed systems, asynchronous behavior, nondeterminism, and failures can make debugging very difficult as different executions of the same code can produce different results. With this framework, executions can be replayed exactly, as long as all random numbers are obtained using the provided random number generator in the Utility class. In simulator mode, this can be done manually by specifying the same random seed and giving the same input, but using the built-in replay mechanism simplifies this task. It records user input and replays it as well to assist with user-defined failures. Note that this mechanism is designed so that the student can go back and add debugging output to diagnose the problem. If the behavior of the node class changes in terms of its disk writes/reads, message sends/receives, timer actions, or even usages of the provided random number generator, the execution may not be valid and may not reproduce.

5 Getting Started

Leave the jar files in the jars/ directory, and place all student code in the proj/ directory. An example reliable, in-order messaging layer is provided. Take a look at the code in the proj/ directory, the command file in the scripts/ directory and the scripts in the base directory to see how to implement and run a project. Run the Messagelayer main method with the -h argument to see all the possible command-line options.