Building Distributed Systems

- distributed components
- have to deal with failures
- Messaging Layer
  - interface with hardware
  - faulty environment
  - debugging
- written in Java
- add/change files in proj/
Reliable In-order Message Layer

- ReliableInOrderMsgLayer.java
- Reliable, in-order delivery in the absence of failures
In-order Message Delivery

- Sequence Numbers, ACKS
- Time outs, retransmissions (like TCP)
- Packet Receipt: `public void RIODataReceive()`
Packet Sending

- public void RIOSend(dst, protocol, payload)
- implementing timeout: register timeout (for each unACK) function as a callback at a certain time

Manager.java/Callback.java
Running the Distributed System

- Environment: simulation/emulation
- Configure Topology/Events
  - configure nodes: `start [n]`
  - event command: `[n] command`
  - time: advance by 1 timestep
  - example: scripts/RIOTest
Implementing the Node Interface

- Example: RIOTester (implements RIONode, which derives from Node)
- Node class identified at command line at the start to the manager (sim/emu)
- commands defined in `onCommand()`
  - example: 'begin' in RIOTester
  - send 20 packets to the first 3 nodes
- Packet types: `Protocol.java`
Failure Modes

• Specified by prob in node class
  ○ getFailureRate, getDropRate, getDelayRate (RIOTester.java)

• ... or by user control (command line)
  ○ 0: all events controlled by probs
  ○ 1: crashes controlled explicitly by user
  ○ 2: drops, 3: delay, controlled by user
Simulator (brief overview)

- Every timestep:
  - process in-flight packets
    - drop, delay, deliver
    - remove dropped pkt from in-flight queue
    - keep delayed pkt in-flight
    - schedule rest as delivery event
      - checkInTransit(currentRoundEvents)
  - schedule timeout events
    - checkTimeouts(currentRoundEvents)
  - schedule node crash events
    - checkCrash(currentRoundEvents)
Project 1: Client Server Filesystem

- 2 nodes in the system: server, client
- Simple RPC protocol
- Set of procedures for file operations (called by client)
- Handle node failures
- commands parsed and executed by onCommand() function in node class
  - specified in command file
  - 0 create 1 foo.txt
Simple Filesystem Routines

- flat hierarchy (no directories)
- small files (fit in one pkt, minus header)
- create server filename
- read server filename
- append server filename contents
- checkVersion server filename
- Handle incorrect operations:
  - e.g., creating an existing file
  - no file changes, error msg sent back
Handling Failure Events

- Detect crash?
- Client failures
  - crash: server still serves request
  - ignore outstanding responses
- Server failures
  - crash/drop after service execution
  - crash/drop before service execution
  - client can't know which one
Server Failure Scenarios

1) Lost Request Message
   ● failure before service execution

2) Lost Response
   ● failure after service execution

How does the client know this?

● timeouts
● resend request
Server Failure Scenarios

- Side-effects of duplicate requests
  - idempotent (can be repeated harmlessly)
    - reads
  - nonidempotent (side-effects)
    - bank transfers (writes)

- How to deal with nonidempotent duplicate requests?
Server Failure Scenarios

3) Crash

- failure before/after service execution
- semantics for recovery:
  - at least once
    - keep trying until success
    - deal with duplicates (client)
    - idempotent operations
  - at most once
    - only one execution, or give up
    - smart server
**Guiding Principles**

- **Correctness**
  - correct action should be performed in the absence of failures
  - if the command executes, result should be correct
- **Simplicity**
  - corner cases (always)
  - e.g., no need for a 3-way handshake, teardown
- **Termination**
  - OK to give up after a reasonable # timeouts