18-842: Distributed Systems **Spring 2015**
Lab 0: Communications Infrastructure

Objective: Students will develop an abstraction of the interprocess communication mechanisms that will be used for further labs. This communications "shim" will allow for fine grained testing and control of message passing between well-defined processes in a distributed system. Further, a sample configuration harness will be developed.

Administrative details: You will work in teams of two students. Your teammate will be randomly selected. The lab is due at 9:30 am on 30 January. At that time, an archive of your source code needs to be deposited on ALE. Each team will give a quick demo (maximum of 30 minutes), using the identical code, before 9:30 am on 4 February.

Specification: You will develop a "MessagePasser" object (in Java) through which communication will pass. In later labs, your code will use this object to send messages to other processes on (potentially) other hosts. The destination process will also use a **`MessagePasser`** object to receive the messages. The **`MessagePasser`** interface is:

```
public class MessagePasser {

  public MessagePasser(String configuration_filename, String local_name);

  void send(Message message);

  Message receive( );  // may block.  Doesn't have to.

}

public class Message implements Serializable {

  public Message(String dest, String kind, Object data);

  // These settors are used by MessagePasser.send, not your app
  public void set_source(String source);
  public void set_seqNum(int sequenceNumber);
  public void set_duplicate(Boolean dupe);

 // other accessors, toString, etc as needed

}
```

Discussion: You will write an *interactive* application program that instantiates the MessagePasser just once. It will pass in two parameters that have been obtained via the command line (or via GUI if you prefer). The first parameter specifies a configuration file, format details below. The second parameter distinguishes this process from any others -- it is the **`name`** string of this particular process. Your **`MessagePasser`** object will parse the configuration file and set up sockets for communicating with all processes listed in the configuration section of the file. Port numbers listed are for initial communication, perhaps subsequent messages need to go to a different port. The constructor will initialize buffers to hold incoming and outgoing messages to the rest of the nodes in the system. Additional state (threads?) may be required as well. Use TCP as your transport. Details of exactly how the sockets get managed is up to you. What is important is that all processes will be able to communicate (send and receive). Additionally, you are supposed to be well-versed in network mechanics, so use them efficiently. In particular, you should ensure that a single TCP connection remains live between each pair of nodes. You don't want to go through the setup/teardown of the connection for each message.

Your application program can create messages (of class **`Message`**) and call the MessagePasser's **`send`** method to get them sent. **`MessagePasser`** will set the sequence number of the message before sending it. Sequence Numbers should be non-reused, strictly incrementing integer values, starting at zero. Each **`Message`** in your entire system will have a unique combination of source/sequence number.

Your application program can also call the receive method to get anything currently waiting in the MessagePasser's receive buffer. The easy implementation is to make **`receive`** block. Enterprising groups may wish to make non-blocking versions of the method as well.

The messages themselves are pretty easy to build, as they are static. They have a header and a payload. In the header is a destination node (by name), a sequence number (which the **`MessagePasser`** should generate and should be unique among all messages sent by the local node), a duplicate flag (used only by the **`MessagePasser`**, not your application) and a message kind (which need not be unique). The payload is any object you care to send. We won't worry too much about segmentation, MTU or any of that stuff, as I assume you learned it in your network class.

When the **`send`** method is called, the **`MessagePasser`** will check the message against any SendRules (specified in the configuration file and further explained below) before delivering the message to the socket.

On the receive side, the **`MessagePasser`** will be getting messages from the socket as they are delivered. The **`MessagePasser`** will be checking each received message against ReceiveRules (again, explained below) and storing them in an input queue. Whenever the application calls **`MessagePasser.receive( )`**, the **`MessagePasser`** will deliver a single message from the front of this input queue.

The Configuration File: The configuration file will allow you to determine at *runtime* how the object will handle each message. To facilitate testing on long-running processes, the file should be checked to see if it was modified and, if so, re-read before processing each send and receive method. It can be very useful to have the configuration file live on a distributed filesystem (like AFS or dropbox) so each node can read copies without needing to FTP them around.

The configuration file is in YAML format (see yaml.org). I recommend using snakeYAML (code.google.com/p/snakeyaml) library to parse the config file. If you aren't familiar with

YAML, the wikipedia article can get you up to speed (en.wikipedia.org/wiki/Yaml) The first items are used for system configuration and thus should only be referenced during the initial setup in the constructor, even if they change at runtime.

```yaml
# Defines which nodes are used in the system and how to connect to them.
# This element should only be referenced during the constructor's initial
# setup.
# All names must be unique.
# No other nodes are known.
# These names will be used for src and dest addresses for the Message
# The local node’s name is included in this list
configuration :
  - name : alice
    ip   : 192.168.1.52
    port : 12344        # This is the incoming port on which MP will listen
                        # May be changed for further communication as usual
  - name : bob
    ip   : 192.168.1.112
    port : 14255
  - name : charlie
    ip   : 128.2.130.19
    port : 12998
  - name : daphnie
    ip   : 192.168.1.87
    port : 1987

sendRules :
  - action : drop # Ack message number 4 from bob to alice will not be sent
    src    : bob
    dest   : alice
    kind   : Ack
    seqNum : 4
  - action : delay # Every Lookup message in the system will be delayed
    kind   : Lookup

receiveRules :
  - action : duplicate      # 3rd message from Charlie that anyone
    src    : charlie        # receives will be duplicated
    seqNum : 3
```

Rule Processing: The configuration section is pretty self explanatory. Unfortunately, the SendRules and ReceiveRules sections need some explanation.

Each of these sections specify a list of rules, each of which must contain an Action and may contain Src, Dest, Kind, SeqNum and Duplicate fields. Before sending each message, the send method will compare the fields in the message (src, dest, kind, sequenceNumber and duplicate) against the fields in the rule to figure out what to do with it. If **all** of the fields in the rule match variables in the message, then the rule's action will be fulfilled and rule processing will be stopped (i.e. The first rule to match the message is the only one that will be acted upon). Any fields not specified are *wildcard* and match all values of that variable. Note that a rule consisting of just an Action field will match all messages.

There are three possible values for the Action field of a rule: drop, duplicate, and delay.

Drop: Any message which matches a drop rule will be ignored. It will not be sent (if this is a SendRule), nor will it be delivered to the application (if this is a ReceiveRule).

Duplicate: Any message which matches a duplicate rule will be duplicated (i.e. send two copies if this is a SendRule. Deliver two copies to the application if this is a ReceiveRule). Note that it is possible for an application to **`receive()`** 4 copies of a message if it is duplicated at sender and receiver. If a duplicate is created due to a SendRule, the first message will have the duplicate field set to False and the second message will have the duplicate field set to True. The duplicate fields *only*[1] use is to be able to distinguish them for ReceiveRule processing (i.e. you can Drop just the duplicate at the receiver, which would otherwise be impossible). With the exception of the duplicate field, the duplicate messages will be identical — in particular, ensure they have the same sequenceNumber.

Delay: Any message which matches a delay rule will be set aside temporarily and not sent / delivered until after after another non-delayed message is sent / delivered. Note that this means that multiple messages may be set aside, if they all match a delay rule. In such a case, a single non-delayed message sent / received will trigger the sending / receiving of all such delayed messages.

Demonstrate: Develop a testing harness that allows you to send and receive enough messages among 4 processes on at least 2 different computers to prove your infrastructure works well.

Your testing harness needs to allow for interactive specification of messages to be sent. It must have a User Interface, which need not be Graphical. No compilation can be required in order to have a different scenario. If a TA asks that you run a particular scenario (i.e send a message of kind "Request" from Alice to Bob) you must be able to execute this scenario without changing your code or recompiling.

It would be possible for your scenario to be specified in another file that your testing harness reads. However, this is difficult to demonstrate, so is discouraged. Note that such an approach can be really useful for your project work, as you can keep a set of files that detail regression tests and run them automatically.

Finally, be prepared to demonstrate the actual network message content using Wireshark. In particular, be able to show that there is only one TCP connection between each pair of participants, and that the connection remains throughout the entire demo.

---

[1] Only use! Applications can't distinguish duplicated messages, for instance.

18-842 Lab Teams

Team 1: Brandon Wolfe (blwolfe) and Joel Krebs (jkrebs)
Team 2: Advaya Krishna (advayak) and Yvonne Yuan Yuan (yyuan1)
Team 3: Joyce Chung (joycechu) and Rachita Chandra (rchandra)
Team 4: Na Tian (ntian) and Jackie Jiang (xiaotiaj)
Team 5: Harshad Shirwadkar (hshirwad) and Subodh Asthana (sasthana)
Team 6: Suraj Kasi (skasi) and Yang Wu (yangwu)
Team 7: Abhijeet Gautam (abhijeeg) and Suril Dhruv (sjdhruv)
Team 8: Ding Zhao (dingz) and Wallace Wang (xinyuwan)
Team 9: Jing Yu (jingyu) and Honghanh Nguyen (honghanh)
Team 10: Anirudh Nambiar (anambiar) and Norman Wu (luow)
Team 11: Qing Zhou (qzhou) and Debjani Biswas (dbiswas)
Team 12: Srikant Avasarala (savasara) and Jian Wang (jianwan3)
Team 13: Zhengyang Zuo (zzuo) and Nicolas Mellis (nmellis)
Team 14: Omkar Gawde (ohg) and Lawrence Tsang (ltsang)
Team 15: Congshan Lv (congshal) and Brody Anderson (bcanders)
Team 16: Kenny Sung (tsung) and Ruei-Min Lin (rueiminl)
Team 17: Roshani Bhandari (rjbhanda) and Subramanian Natarajan (snatara1)
Team 18: Lixun Mao (lmao) and Chanjuan Zheng (chanjuaz)
Team 19: Saud Almansour (salmanso) and Mohan Yang (mohany)
Team 20: Yuyan Zhao (yuyanz) and Qinyu Tong (qtong)
Team 21: Naman Jain (namanj) and Prabhanjan Batni (pbatni)
Team 22: Xing Wei (xingw1) and Wanchao Liang (wanchaol)
Team 23: Keane Lucas (kjlucas) and Bujar Tagani (btagani)
Team 24: Muyun Chen (muyunc) and Cassie Urmano (curmano)
Team 25: Joe Vessella (jvessell) and Hongyi Zhang (hongyiz)
Team 26: Xuan Zhang (xuanz1) and Xinkai Wang (xinkaiw)
Team 27: Jeff Brandon (jdbrando) and Ryan Cutler (rcutler)
Team 28: Will Snavely (wsnavely) and Nicholas Caron (ncaron)
Team 29: Sheng-Ho Wang (shenghow) and Yujing Zhang (yujingz1)
Team 30: Anish Jain (anishj) and Joseph Carlos (jcarlos)
Team 31: Hailei Yu (haileiy) and Mayur Sharma (mayurs)
Team 32: Debanshu Das (debanshd) and Hailun Zhu (hailunz)
Team 33: Yachen Wang (yachenw) and Yibin Yan (yibiny)
Team 34: Xingchi Jin (xingchij) and Siqi Wang (siqiw)
Team 35: Rohan Sehgal (rohanseh) and Sean Klein (smklein)
Team 36: Vinay Bhat (vbhat) and Samarth Mittal (samarthm)

18-842 Lab Teams

Team 37: Eryue Chen (eryuec) and Kaidi Yan (kaidiy)
Team 38: Xiaokai Sun (xiaokais) and Hsueh-Hung Cheng (hsuehhuc)
Team 39: Shruti Ratnam (sratnam) and Chun-Ning Chang (chunninc)
Team 40: Wenhan Lu (wenhanl) and Rahul Muthoo (rmuthoo)
Team 41: Jonathan Lim (jlim2) and Shan Gao (shang)
Team 42: Shuo Chen (shuoc) and Samantha Allen (slallen)
Team 43: Pranav Bagree (pbagree) and Huacong Cai (hcai)
Team 44: Gaurav Jain (gmjain) and Vikram Nair (vikramn)
Team 45: Rohith Jagannathan (rjaganna) and Huiyuan Wang (huiyuanw)
Team 46: Nick Winski (nwinski) and Ara Macasaet (lmacasae)
Team 47: Yin Lin (yinlin) and Venno Fang (kangf)
Team 48: Qiwen Guo (qiweng) and Alexandros Mavrogiannis (amavrogi)
Team 49: Ke Wang (kewang1) and Haoyu Chen (haoyuche)
Team 50: Jin Xi (jinxi) and Grivan Thapar (gthapar)
Team 51: Tiffany Pan (tpan) and Pushen Gao (pusheng)
Team 52: Minghan Chen (minghan2) and Hao Ni (haon)
Team 53: Rao Li (raol) and Edward Huang (zilongh)
Team 54: Bin Lan (blan) and Jian Jiao (jianj)
Team 55: Wei Li (weili1) and Mrikondu Barkakati (mbarkaka)
Team 56: Priyank Sanghavi (psangha1) and Sparshith Puttaswamy Gowda (sputtasw)